FINAL DS&A

ABSTRACT

The project is system upgrade for civil status department.

Hasan Alhwietat

Data Structures & Algorithms

Contents

Task 1: -

a) You need to create design specification for this data structure supposing that you will use Linked List, also you need to show how is your design will carry out its valid operations. (Report)

Queue: It is a linear data structure in which an element is added to its end and an element is deleted from its beginning. It is also called first in first out FIFO.

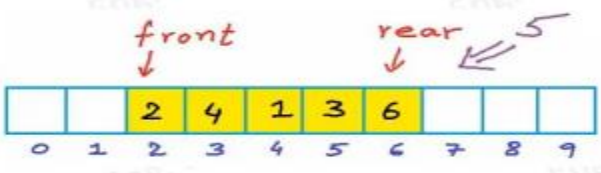
- Insertion occurs from one end called the rear or tail.

- Removal occurs from the end of the unit called the front or head.

Operation Queue linked list: (Dynamic Queue implemented with a linked list size as needed).

1- Insertion occurs from one end called the stern or tail.
2- Removal occurs from one side, called frontal or vertical.
3- To insert an element called EnQueue.
4- To remove an element called DeQueue.
5- To check if the list is empty or not, called IsEmty.
6- The item at the top of the Queue, Front.
7- All operations are performed at a constant time O(1) and space O(1).
8- Access and search-read/write element time complexity O(n).
9- EnQueue time complexity worst case O(n)/ best case O (1)/ average case O (n).
10- DeQueue time complexity worst case O(n)/ best case O (1)/ average case O (n).

In this queue, an element is added from the back and an element from the front is removed. For example, we create a waiting list consisting of 5 elements. We entered the data and filled the queue and wanted to add a number to the list. We will remove an element from the beginning and add it from the back.



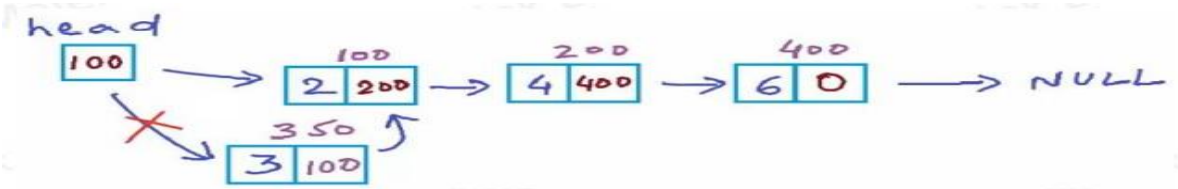
b) In general, interpret what is the trade-off when specifying this Data structure as an ADT. (Report)

Trade-off: It is the case in which one thing increases and something else decreases, and it is a way to solve problems either in less time and using more space or in a small space and more time, so the more efficient the algorithms in terms of time, the less efficient it is in the use of time.

ADT (abstract data type): A mathematical model of a data structure that defines the type of data stored, the operations supported on it, and the types of parameters of the operations. ADT defines each operation but does not specify how it does it.

Type ADT:

1- Stack: It is a linear data structure and a way to store and organize data that is inserted and removed from the same end. This end is called the top of the stack. The stack is called Last-In First-Out (LIFO).

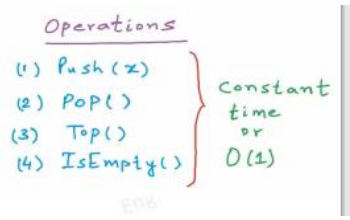


Operations stack Array: (Static stack implemented with an Array fixed size)

1- The process of insertion and deletion occurs from one side called the upper part.
2- To insert an element called push.
3- Remove an item called Pop.
4- Top Returns the element at the top of the stack.
5- IsEmpty to check whether the stack is empty or not.
6- All operations are performed at a constant time O(1) and space O(1).
7- Access and search-read/write element time complexity O(n).
8- Push()- best case O(1) / worst case O(n) / average case O (1).

Operations Stack linked list: (Dynamic stack implemented with a linked list size as needed)

1- The process of insertion and deletion occurs from one side called the upper part.
2- To insert an element called push.
3- Remove an item called Pop.
4- Top Returns the element at the top of the stack.
5- IsEmpty to check whether the stack is empty or not.
6- All operations are performed at a constant time O(1) and space O(1).
7- Access and search-read/write element time complexity O(n).
8- Insert/delete end of list (tail) O(n).
9- Insert/delete begin of list O(1).



Advantages:

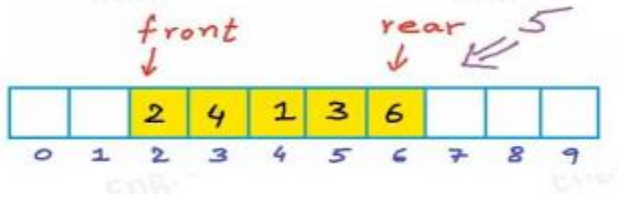
1- Esay to get started.
2- Efficient data management.

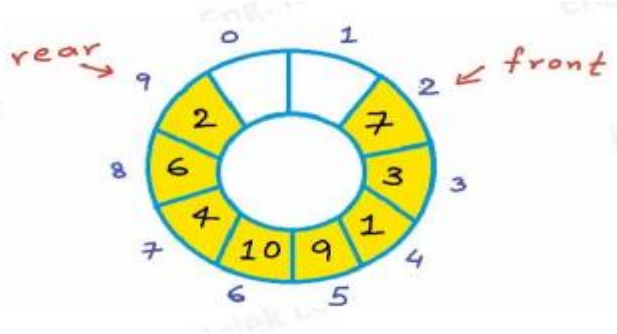
3- Low hardware required.
4- Anyone with access can modify the program.

Disadvanages:

1- Not flexible.
2- lack of scalability.
3- Unable to copy and paste.
4- Limited memory size.

2- Queue: It is a linear data structure in which an element is from its beginning. It is also called first in first out FIFO.





Operation Queue Array: (Static Queue implemented with an array fixed size).

Time and space complexity using array:

OPERATION	BEST	AVERAGE	WORST	BEST	AVERAGE	WORST
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
enqueue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
dequeue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
count()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
peek()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
show()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)

Time

Space

1- Insertion occurs from one end called the stern or tail.
2- Removal occurs from one side, called frontal or vertical.
3- To insert an element called EnQueue.
4- To remove an element called DeQueue.
5- To check if the list is empty or not, called IsEmty.

6- The item at the top of the Queue, Front.
7- All operations are performed at a constant time O(1) and space O(1).
8- Access and search-read/write element time complexity O(n).
9- EnQueue time complexity worst case O(n)/ best case O (1)/ average case O (n).
10- DeQueue time complexity worst case O(n)/ best case O (1)/ average case O (n).

Operation Queue linked list: (Dynamic Queue implemented with a linked list size as needed).

Time and space complexity using LinkedList:

OPERATION	BEST	AVERAGE	WORST	BEST	AVERAGE	WORST
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
enqueue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
dequeue()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
count()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)
peek()	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
show()	O(1)	O(N)	O(N)	O(1)	O(1)	O(1)

Time

Space

1- Insertion occurs from one end called the stern or tail.
2- Removal occurs from one side, called frontal or vertical.
3- To insert an element called EnQueue.
4- To remove an element called DeQueue.
5- To check if the list is empty or not, called IsEmty.
6- The item at the top of the Queue, Front.
7- All operations are performed at a constant time O(1) and space O(1).
8- Access and search-read/write element time complexity O(n).
9- EnQueue time complexity worst case O(n)/ best case O (1)/ average case O (n).
10- DeQueue time complexity worst case O(n)/ best case O (1)/ average case O (n).

Advantages:

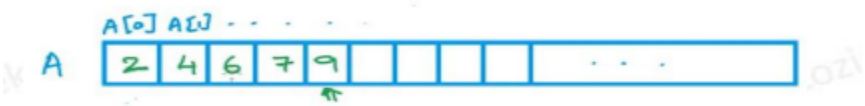
1- Manage big data efficiently and easily.
2- Fast in data communication between operations.
3- Make deletions and insertion easily because it follows the rule (FIFO).

Disadvantages:

1- Insertion and deletion take a long time.
2- Limited space.
3- No item can be added unless the item in the queue is deleted.

4- The size of the queue must be determined in advance.

3- List: It is a set of elements that have a linear relationship with each other. A certain number of elements of a given data type are stored in a sequential and homogeneous manner, for example, a list of names and numbers.



Operations list: (Dynamic list can add or delete items)

1- Insertion and removal is expensive. (All operation time complexity O(1) and space O(1))
2- get() returns an item from the list at any specified position.
3- Insert() Insert an item into any topic in the list.
4- Remove() remove the first item from the list or removeAt() remove an item from any specified location.
5- Size() returns the number of items in the list.
6- isEmpty() returns true if the list is full and false if it is empty.
7- Access- read/write element time O(n).

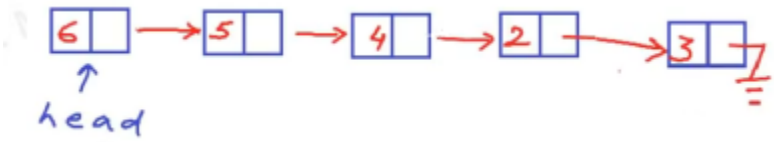
Advantages:

1- It is a generic collection.
2- It uses less memor.

Disadvanages:

1- Lack of time efficiency.

4- Linked list: It is a dynamic arrangement that contains a link to the subsequent elements and is a set of structures arranged due to logical links that are stored as part of the information.



Operation linked list: (Dynamic linked list)

1- InLinked() can insert elements in the linked list and updeate some links.
2- DeLinked() can delete elements in the linked list and updeate some links.
3- All operation time complixtiy O(1) and space complixity O(n).
4- Size() returns the number of items in the list.
5- isEmpty() returns true if the list is full and false if it is empty.
6- Access- read/write element time O(n).

Advantages Linked List:

1- It can grow and shrink at runtime by allocating and deallocating memory due to the dynamic data structure.

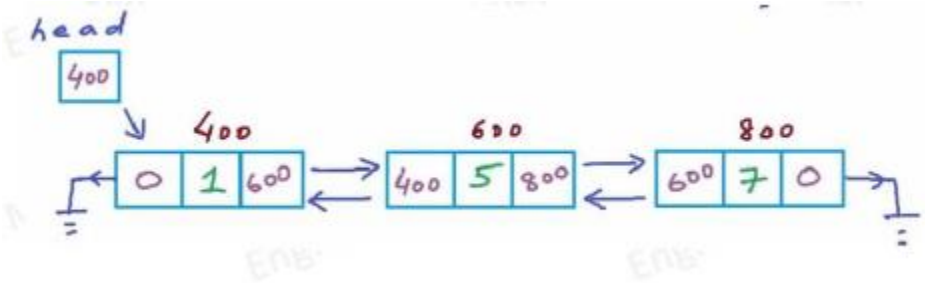
2- No memory loss.
3- Insertion and deletion operations are done easily, because you do not need to remove items after deletion or insertion.

Disadvantages Linked list:

1- It needs more memory.
2- It takes a long time to pass.
3- It is not possible to pass in reverse.
4- Not possible random access due to dynamic memory allocation.

Type linked list:

1- Doubly linked list: Each node has three data segments, the next node and the previous node.



Operation Doubly linked list: (Dynamic doubly linked list)

1- InDoublyLinked() can insert elements in the linked list and updeate some links.
2- DeDoublyLinked() can delete elements in the linked list and updeate some links.
3- All operation time complixtiy O(1) and space complixity O(n).
4- Size() returns the number of items in the list.
5- isEmpty() returns true if the list is full and false if it is empty.
6- Access- read/write element time O(n).
7- Remove at head O(1) and remove in middel O(n) and remove tail O(1).

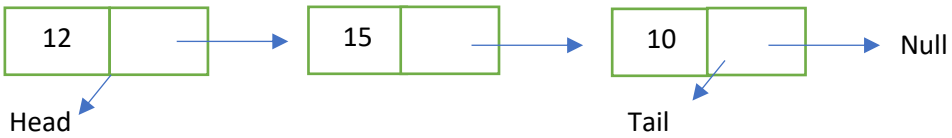
Advantages Doubly Linked list:

1- Crossing is permitted on both front and back sides.
2- You can easily delete and insert.
3- It is easy to pass backwards.
4- A new node can be inserted before a specific node easily.

Disadvantages Doubly Linked list:

1- It consumes extra memory.
2- It is impossible to access the items randomly.
3- You need more time, because you need to deal with indicators during deletion and insertion

2- Singly Linked List: It is a type of unidirectional linked list, meaning that there is only one direction from the head to the last node (the tail).



Operations on singly linked list: (Daynaimic)

1- InSinglyLinked() can insert elements in the linked list.
2- DeSinglyLinked() can delete elements in the linked list.
3- All operation time complixtiy O(1) and space complixity O(n).
4- Size() returns the number of items in the list.
5- isEmpty() returns true if the list is full and false if it is empty.
6- Access- read/write element time O(n).
7- Remove at head O(1) and remove in middel O(n) and remove tail O(n).

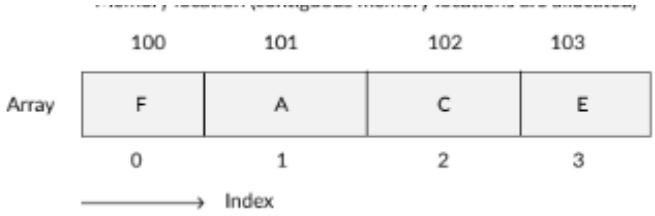
Advantages Singly linked list:

1- Easy insertion and deletion process.
2- It does not need to move the elements to insert and delete them.
3- Its size is not fixed.
4- A little time to reach the node

Disadvantages Singly Linked list:

1- It requires additional memory because memory is wasted.
2- Items cannot be accessed randomly.
3- It is not possible to perform reverse traversal.

5- Array: It is a set of data elements that are stored contiguously in memory and is the simplest data structure because the data elements can be accessed directly by the index number.



Operation Array: (Static)

1- InArray() can insert elements in the array.
2- DeArray() can delete elements in the array.
3- All operation time complixtiy O(n) and space complixity O(n).
4- Size() returns the number of items in the array.
5- isEmpty() returns true if the list is full and false if it is empty.
6- Access- read/write element time O(1).

Advantages Array:

1- Multiple data items of the same type using one name.

2- Items can be accessed randomly by index number.
3- It is used to represent two-dimensional matrices.
4- Data is stored in contiguous blocks.

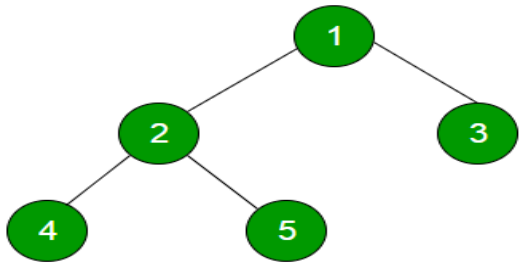
Disadvantages Array:

1- The number of items to be stored must be known in advance.
2- Memory is limited, no additional memory can be allocated.
3- It has a fixed size.
4- Insertion and deletion in an array are very difficult.
5- When you allocate the memory too much, it wastes space, and when you reduce it, it becomes a problem.

c) Using Pseudocode Demonstrate how this data structure will be used to print out the data in level order, write a Pseudocode for a function named printLevelOrder that takes only the reference to the root in a binary tree, and prints all the data in this tree in level order. (Report)

traversal of a tree Breadth-First Approach

level order traversal of a tree: (Pseudocode)



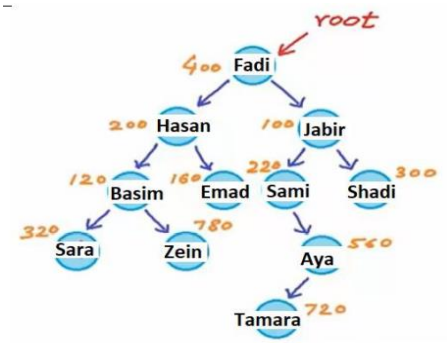

```
printLevelOrder(root)
 { // Start Function
     H ← height(root)
     FOR I IN TO 1, H+1
     {                          // Start Loop
         Print root, i
     }                          // End Loop
 } //End Function
Print root
```

Output:

1 2 3 4 5

d) Suppose all the data is represented in the following binary tree (Figure 1), illustrate how your design will work and its content step by step during printing the data in Level order. (Report)

Level Order Binary Tree Traversal: -



Binary Tree Traversal: An orderly process that visits all nodes in the tree at once.

There are two types of binary tree traversal:

1- Breadth-First.
2- Depth-First

We'll use breadth-first binary tree traversal:

The way it works will pass through the nodes starting from level 0 and then the next level by visiting the nodes from left to right.

So from Figure 1 it will pass from level 0 and the node is Fadi then moves to level 1 and starts from the left and the node is Hasan and then moves to the right and the node is Jabir and then moves to level 2 and starts from the left and the node is Basim and then moves to the right and the node is Emad and then it moves to the right and the node is Sami and then moves to the right and the node is Shady then moves to level 3 and from the left and the node is Sarah and then moves to the right and the node is Zein and then moves to the right and the node is Aya then moves to level 4 and starts from the left and the node is Tamara and then ends the tree.

Level Order traversal of binary tree is:

Fadi, Hasan, Jabir, Basim, Emad, Sami, Shady, Sarah, Zein, Aya, Tamara

e) You are requested to implement this data Structure suing High Level Language, you need to use Linked List in you implementation, and the tail of the Linked List is chosen by your manager to be the removing end (the side that you want to remove the data from it). (Code)

```
# Queue using doubly linked list

class Node:
  def __init__(self, data):
    self.data = data
    self.next = None
    self.prev = None

class Queue:
  def __init__(self):
    self.head = None
    self.last=None

  def enqueue(self, data):
    if self.last is None:
      self.head =Node(data)
      self.last =self.head
    else:
      self.last.next = Node(data)
      self.last.next.prev=self.last
      self.last = self.last.next
```

```
  def dequeue(self):
    if self.last is None:
      return "not value"
    else:
      temp= self.last.data
      self.last = self.last.prev
      self.last.next=None
      return temp

  def top(self):
    return self.head.data

  def size(self):
    temp=self.head
    count=0
    while temp is not None:
      count=count+1
      temp=temp.next
    return count


  def isEmpty(self):
    if self.head is None:
      return True
    else:
      return False
```

```
# Node class
# Function to initialise the node object
# Initialize data as elements
# Initialize next as null
# Initialize prev as null

# Queue class contains a Node object
# Function to initialize head and last == null


# Function to add an element data in the Queue
```

```
  # Function to remove last element


# Function to return top element in the queue

  # Function to return the size of the queue


  # Function to check if the queue is empty or not
```

```
def printqueue(self):
  print("queue elements are:")
  temp=self.head
  while temp is not None:
    print(temp.data,end="->")
    temp=temp.next
```

```
# Function to print the Queue
```

```
[61] s=Queue()
     s.enqueue("Fadi")
     s.enqueue("Hasan")
     s.enqueue("Jabir")
     s.enqueue("Basim")
     s.enqueue("Emad")
     s.enqueue("Sami")
     s.enqueue("Shadi")
     s.enqueue("Sarah")
     s.enqueue("zein")
     s.enqueue("Aya")
     s.enqueue("Tamara")
```

```
s.dequeue()
s.printqueue()
s.top()
s.size()
```

```
queue elements are:
Fadi->Hasan->Jabir->Basim->Emad->Sami->Shadi->Sarah->zein->Aya->10
```

f) You are requested to handle any expected errors that may occur in your code that you wrote in part (e) from this question. (Code)

#1 - The queue was empty and I wanted to delete an item from it. There is a problem and the reason is t hat the item's value remains empty after deleting the item, so it will print that there is no value.

```
[74] # Task 1 f
     #1 - The queue was empty and I wanted to delete an item from it. There is a problem and the reason is that the item's value remains empty after deleting the ite
     s.dequeue()
```

```
'not value'
```

```
[79] #2 - The queue is empty and you want to print the node, then there will be a problem because only the first node will appear and the rest of the node are null
     s.dequeue()
     s.printqueue()
```

```
queue elements are:
Fadi->
```

```
# 3-Writing function and class on the same level
class Node:
def __init__(self):
```

```
  File "<ipython-input-81-48501df47b36>", line 3
    def __init__(self):
    ^
IndentationError: expected an indented block
```

SEARCH STACK OVERFLOW

```
# 4-Forgetting to write (:) after the (if) conditional
  if self.empty()
```

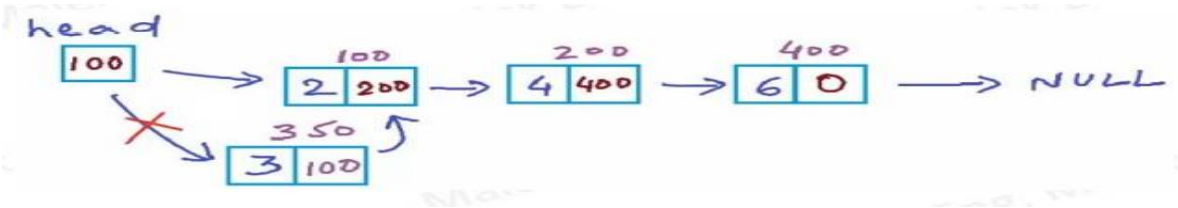
```
  File "<ipython-input-82-aee37f92d596>", line 2
    if self.empty()
    ^
IndentationError: unexpected indent
```

SEARCH STACK OVERFLOW

Task 2: -

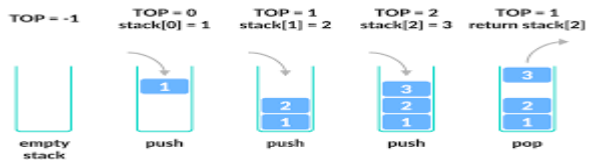
a) The computer uses specific Data structure to handle the function calls, specify this Data Structure as Abstract Data Type (ADT) showing the formal definition, Logical view, and the valid operations with it. (Report)

Stack: It is a linear data structure and a way to store and organize data that is inserted and removed from the same end. This end is called the top of the stack. The stack is called Last-In First-Out (LIFO).

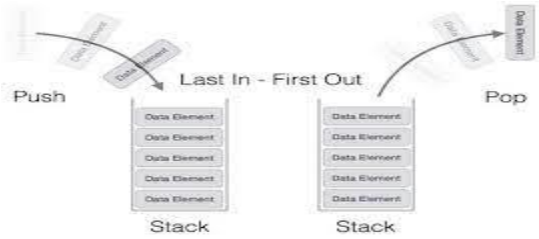


Operations Stack linked list: (Dynamic stack implemented with a linked list size as needed)

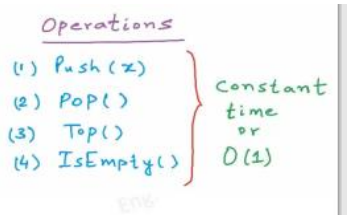
1- The process of insertion and deletion occurs from one side called the upper part.
2- To insert an element called push().



3- Remove an item called Pop().



4- Top Returns the element at the top of the stack.
5- IsEmpty to check whether the stack is empty or not.
6- All operations are performed at a constant time O(1) and space O(1).
7- Access and search-read/write element time complexity O(n).
8- Insert/delete end of list (tail) O(n).
9- Insert/delete begin of list (head) O(1).



Advantages Stack:

1- Esay to get started.
2- Efficient data management.
3- Low hardware required.
4- Anyone with access can modify the program.

Disadvanages Stack:

1- Not flexible.
2- lack of scalability.

3- Unable to copy and paste.
4- Limited memory size.

Logical view Stack:

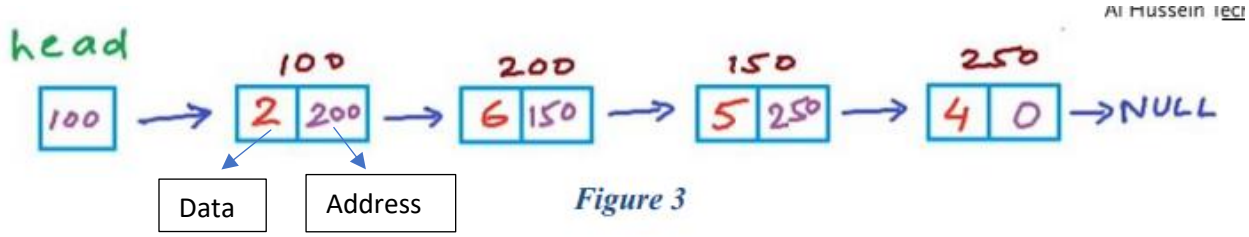


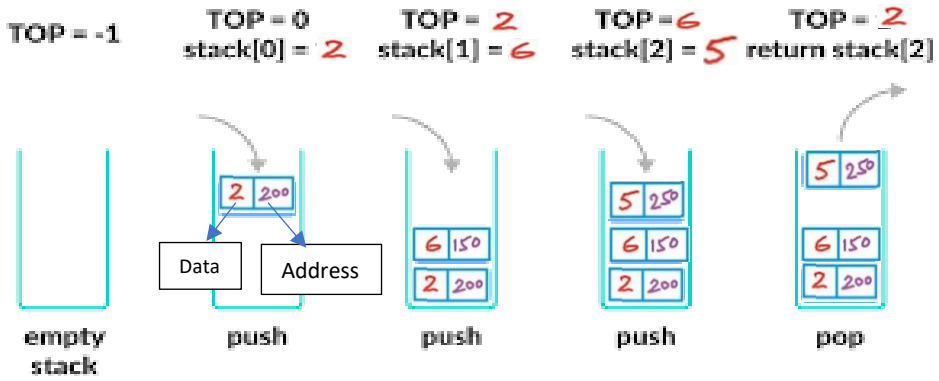
Figure 3

The logical representation of the stack is a container open from one side. For example, in the example, we will enter via push () and the stack will contain 4 integers. The elements can also be removed by pop () and we can check if it is and check whether the stack is empty from the same end.

When you add a node at the beginning, you need to create a node and you need to build two links, the first link from the new node to the first node, the second link from the head to the new node. As for deleting a node from the beginning, you need to cut the link from the head to the first node, then we will create a new link from the head to the second node.

b) Show how this Data structure that you mentioned in part (a) from this question is used to handle function calls by showing its content step by step through code execution in this task. (Report)

At first and according to the figure, the first function which is print(head) will read the header, then it will go to the second function which is print (node), the code will know whether the node is empty or not, and if it is empty, it will print a space, then it will go to the third function which is print (node.next) will read the next node and then return to the second function which is print (node), then it will print the value of the node followed by a space. If it is not empty, it will go to the third function print(node.next) it will read the next node and then return to the second function print(node), then it will print the value of the node followed by a space.



Task 3: -

a) Determine two available measurement tools that you can use to measure the performance of each Algorithm, and briefly explain them, illustrate your answer by an example. (Report)

Tools measurement performance algorithm:

1- Time complexity: It is the number of times a set of code is executed i.e. when we give any algorithm commands to execute it we measure the time it takes to execute these commands so time greatly affects the effectiveness of the algorithm so will give a large number of inputs to measure its efficiency the algorithm. When the input is small the execution time is fast.

There are many different types time complexity:

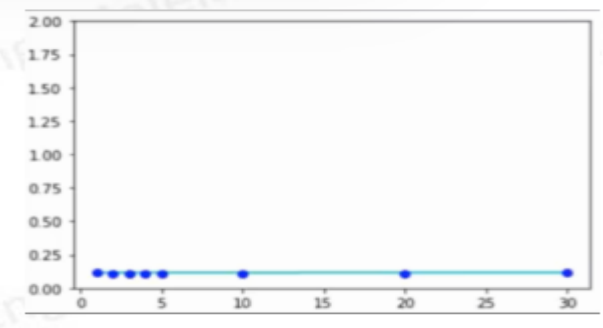
1- Linear Time O(n).
2- Constant Time O (1).
3- Quadratic Time O(n^2).
4- Logarithmic Time O (log n).

Example: In this example, we defined a variable and gave it a value equal to 0 and we returned this value. We will measure the execution time for different n and we give it different values from 1 to 30 and we found that the time is fixed for different operations and here we say that the algorithm is excellent because it does not consume a lot of time to perform the operation so time complicity is O (1).

```
static int testFunction(int []n) {

    int total = 0;

    return total;
}
```



2- Space complexity: It is the memory space that each program requires to execute. To store what is needed for any algorithm must is proportional to the amount of input that the algorithm takes. Here we note that when the space used to implement the algorithm is small, the algorithm is fast, and the larger the space, the slower the implementation, and then the algorithm becomes inappropriate.

Example: In the example, we will compare selection sort with quick sort. Firstly, the selection sort is an algorithm that sorts the elements in ascending order from smallest to largest. As for quick sorting, which is an algorithm that selects an axis and then separates the array into two arrays, and thus we discover that the selection sort takes O(n) space, while quick sort consumes O (log n) space, and here we note that the selection sorting algorithm is better because it consumes less space.

The fewer resources the algorithm uses, the more efficient it is. We also note that there is no relationship between space and time complexity. They are not related to each other.

b) Evaluate the complexity of each algorithm of the two algorithms. (Report)

Compare	Selection Sort	Quick sort
	It is a simple sorting algorithm and is an algorithm that sorts the elements by taking the smallest element in the array and bringing it to the front so that the elements are arranged in ascending order from smallest to largest.	It is a divide-and-conquer algorithm. This algorithm works by selecting an element of the array (pivot) and dividing the other elements into two sub-arrays, according to whether the element is less than the axis or greater than the axis. It is also called partition-exchange sort. All items in the first sub list are arranged to be smaller than the axis, while all items in the second sub list are arranged to be greater than the axis.
Time complexity	Since the algorithm has two nested loops, the time complexity is O(n^2). There are two loops in the algorithm, the first loop is to select the element in the array one by one so the complex time is O(n) And the second loop is to compare the smaller element with the other elements in the array if the complex time is O(n) When calculating the total complex time, which is O(n) * O(n) = O(n^2) There are three cases Time complexity: - The worst case: O (n^2). The best case: O (n^2). The average case: O (n^2).	This algorithm uses two recursive calls and the time depends on the array being entered and the partitioning strategy is different. There are three cases: - Worst case: When selecting the pivot element as the last element as it will be O(n^2). Best case: When selecting the middle element as the axis, it will be O (n log (n)) Intermediate case: lies between the best case and the worst case. All possible permutations of the matrix must be considered. The items are placed in one array and the items in another array. If the time complexity will be O (n log (n)).
Space complexity	The extra space used in memory is for the element while swapping the two elements in the array so it only makes one swap so it's O (n).	This algorithm is a positional sort because it uses extra space to store recursive function calls but not to process the elements so it is O (log n)

c) Compare the performance of these two algorithms and you should show which one is better and why. (Report)

Compare	Selection Sort	Quick sort
Advantages	1. It is used for small menus because it works very well. 2. It does not require much space. 3. Positional algorithm. 4. Works well with sorted items.	1. Good performance capable of handling huge list. 2. No additional storage is required. 3. The most effective among the sorting algorithms. 4. Quickly solve problems. 5. Positional algorithm.
Disadvantage	1. Poor performance when dealing with huge menus. 2. The number of iterations during sorting is O(n^2) (n) is the number of items in the list. 3. The small space O (n).	1. Its worst-case performance is equal to that of a bubble or insertion. 2. The algorithm is unstable because it does not preserve the original order of the key-value pair.

Performance	It performs well when you work with small lists, but its performance becomes weak when you work with large lists.	It performs well, is fast, is more efficient, and works well with huge lists.

Which is the better?

Selection sort works well in a small list because it is a positional algorithm, but its performance decreases when the list is large, it is only suitable for small lists.

The better Quick sort This is because its performance is better than selection sorting and it is also considered the best sorting algorithms. And it deals with huge lists and is considered the most effective and most used of any size, but its performance in the worst cases is similar to the average performance of bubble or selection types.

A simple example when the number of elements is n=1000, the quick sort needs about 10,000 for the list to be ready, while the specific sort needs to be 1,000,000, meaning that the quick sort is 100 times faster.

d) What is asymptotic analysis that you used and how it is used to assess the effectiveness of any algorithm. (Report)

Asymptomatic analysis: It is an arithmetic operation to calculate the time the running the algorithm in arithmetic units to find the limitations of a programs or runtime. Asymptotic analysis is the big idea that covers all the problems in analyzing algorithms. Also, asymptotic analysis evaluates the performance of any algorithm to find the efficiency of the algorithm. It is evaluated in terms of the size of the input and we calculate how the time or space taken by any algorithm increases with input size.

While checking the runtime of any algorithm, the following must be done:

1- The algorithm must be independent of the machine.
2- The algorithm must work on all inputs.
3- It must be general and not specific to the programming language.

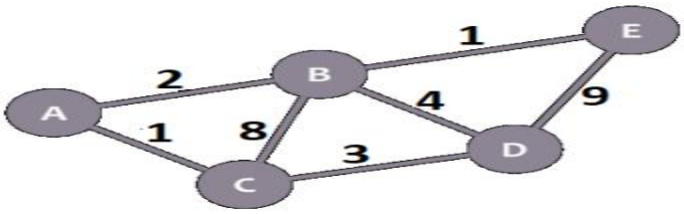
The growth of any function depends on how much the runtime increases as the size of the input increases. And analytics is hardware-independent and software-independent.

For example: Let's explain in this example how the asymptotic analysis is to take linear search and binary search with respect to time linear search and binary search Now we will analyze we will run a linear search on a fast device and set fixed values and run a binary search in a slow device We will notice that the linear search took 1s and search Binary is the 50s, which means that linear search is 50 times faster than binary search, but with the change of the matrix values, we will notice that binary search becomes faster than linear search despite running a binary search on a slow device. We note that the size of the binary search input is logarithmic while a linear search is linear.

Elements (n)	Linear search	Binary search
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

Task 4: -

a) Apply Dijkstra's Algorithm and Bellman-Ford Algorithm to find the solution using each algorithm, show your steps clearly in each Algorithm. (Report)



well-known location as a vertex named C and the department as a vertex named E

There is no difference between the two algorithms, they both calculate the shortest distance from one vertex to all the other vertices, the only difference is that Bellman-Ford works on graphs with negative edge weights, while Dijkstra's does not.

- Dijkstra's Algorithm: C → E

1- First, we made a table of vertexes and find out the shortest distance to the vertex and the previous vertex, and we made two lists to find out the vertices we visited and a list of the vertices that have not been visited yet.

Vertex	Shortest distance from C	Previous vertex
C		
A		
B		
D		
E		

Visited = []

Unvisited = [C, A, B, D, E]

2- Second, we will set the point from which we will start, which is C, and then we will calculate the distance from C to C, which is 0. Then we will put the distances from C to the other vertices, which are unknown, we will put ∞. Then we will put C in the list of visited vertices.

Vertex	Shortest distance from C	Previous vertex
C	0	
A	∞	
B	∞	
D	∞	
E	∞	

Visited = [C]

Unvisited = [A, B, D, E]

3- Third, we will go to the adjacent vertices, which are A, B, D. We will calculate the distance from C to the adjacent vertices, and then we will put the distances in the table.

Vertex	Shortest distance from C	Previous vertex
C	0	
A	1	C
B	8	C
D	3	C
E	∞	

Visited = [C]

Unvisited = [A, B, D, E]

4- Fourth, we will move the adjacent vertex with the shortest distance between A, B, D. show that A is the shortest distance and we will put it in the visited list

Vertex	Shortest distance from C	Previous vertex
C	0	
A	1	C
B	8	C
D	3	C
E	∞	

Visited = [C, A]

Unvisited = [B, D, E]

5- Fifthly, we will do the same as the previous steps and look at the adjacent vertices to A, which is only B, then we calculate the distance, then we will adjust the value of the shortest distance to the vertex B because the distance is shorter than the previous distance, and we modify the last vertex we passed through, which is point A, and then we will put the vertex C to the list of visited summits.

Vertex	Shortest distance from C	Previous vertex
C	0	
A	1	C
B	3	A
D	3	C
E	∞	

Visited = [C, A, B]

Unvisited = [D, E]

6- 6- Sixth, we will calculate the shortest distance to D. We found that the shortest distance is not from B and that the shortest distance is the previous distance, which was from C. Then we will move the summit to the list of visited vertices.

Vertex	Shortest distance from C	Previous vertex
C	0	
A	1	C
B	3	A
D	3	C
E	∞	

Visited = [C, A, B, D]

Unvisited = [E]

7- Seventh, we will calculate the shortest distance to the last vertex, which is E. We will calculate the distance from D, which is 12, and we calculate the distance from D, then B, which is 8, and we calculate the distance from B, which is 4, if the distance from B is the shortest distance to the summit C. Finally we have visited All vertices and calculate the shortest distance for all vertices and this completes the table.

Vertex	Shortest distance from C	Previous vertex
C	0	
A	1	C
B	3	A
D	3	C
E	4	B

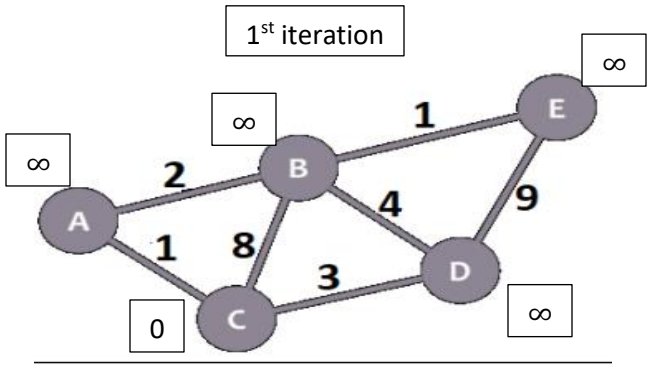
Visited = [C, A, B, D, E]

Unvisited = []

We are required to calculate the shortest distance from summit C to summit E, which is 4.

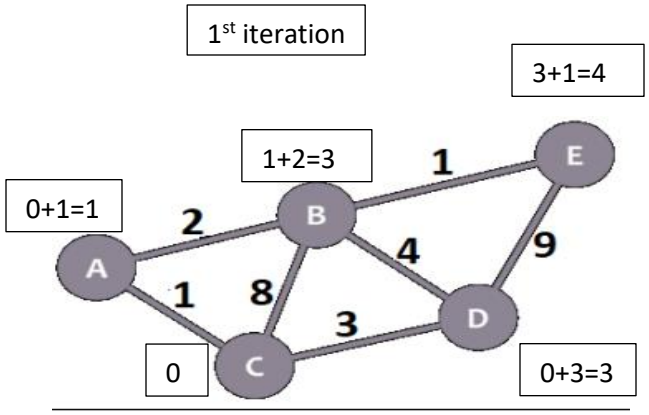
- Bellman-Ford: C → E

1- We do not calculate the number of edges by (|V| - 1) and since the number of vertices in the graph is 5, we must relax all the edges 4 times, then we will list all the edges and put the starting vertex which is C and equal to 0 and the other vertices ∞.



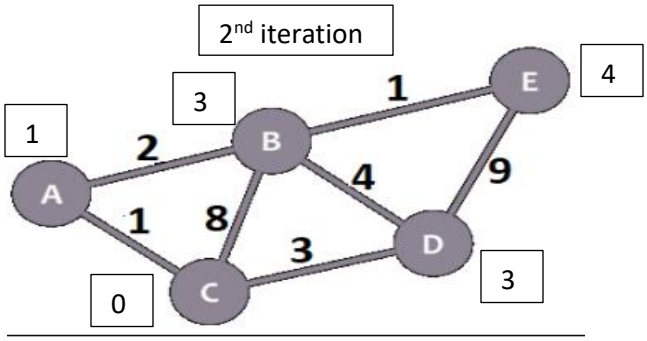
Edges
(C, B)
(C, A)
(C, D)
(A, B)
(B, D)
(D, E)
(B, E)

2- Secondly, we will pass over the list of the edges of the first two edges (C, B) which are equal to = 0 + 8 = 8, then we go to the other two edges, which are (C, A) and they are equal to = 0 + 1 = 1, then we move to the other two edges, which are (C, D) equals = 0 + 3 = 3, then we move to the other two edges, which are (A, B) and equal to = 1 + 2 = 3, so here we notice that it is the shortest distance from the previous point and we adjust the distance and then move to the other two edges, which are (B, D) and equal to = 3 + 4 = 7, and then we move to the other two edges, which are (D, E) and it is equal to =3 + 9 = 12 then we move to the other two edges, which are (B, E) and it is equal to = 3 + 1 = 4. We note here that this distance is shorter than the distance of the two edges (D, E).



Edges
(C, B)
(C, A)
(C, D)
(A, B)
(B, D)
(D, E)
(B, E)

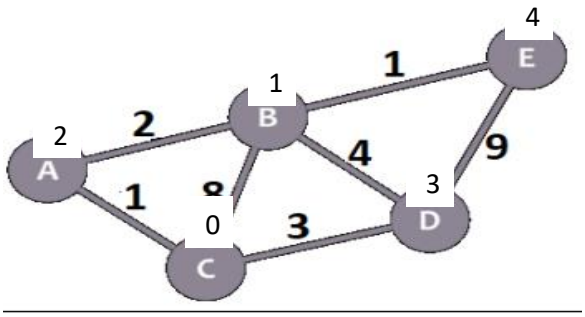
3- Third, we will pass through the sides again with the same steps that we went through. We will start with the first two edges, which are (C, B) = 1+2= 3, then move to the other two edges, which are (C, A) = 1 and then move to the other two edges, which are (C, D) = 3, then we move to the other two edges, which are (A, B) =1+2= 3, then we move to the other two edges, which are (B, D) = 1+2+4= 7, then we move to the other two edges, which are (D, E) = 3+4+1=8, then we move to the other two edges, which are (B, E) =1+2+1= 4, we note that no change occurred in the second iteration and therefore there is no need to move to the third iteration.



Edges
(C, B)
(C, A)
(C, D)
(A, B)
(B, D)
(D, E)
(B, E)

b) Implement the Dijkstra's algorithm as a method called Dijkstra in Java, this method will take Two arguments the Graph and the Source Vertex to start from, suppose the graph is stored in Adjacency Matrix. Your method should print the shortest path from the Source Vertex to every other Vertex in the graph. (Code)

Vertex	Shortest distance from C	Previous vertex
C	0	
A	1	C
B	3	A
D	3	C
E	4	B



```
# Algorithm Dijkstra's shortest

from collections import defaultdict
import sys

class Heap():                                   # Heap class
    def __init__(self):
        self.array = []                         # Initialize array
        self.size = 0                           # Initialize size as 0
        self.pos = []                           # Initialize pos

    def newMinHeapNode(self, v, dist):          # Function v as vertex   , dist as distance
        minHeapNode = [v, dist]
        return minHeapNode

    def swapMinHeapNode(self, a, b):            # Function to swap two nodes
        t = self.array[a]
        self.array[a] = self.array[b]           # of min-heap. Needed for min heapify
        self.array[b] = t
```

```
def minHeapify(self, idx):                # A standard function to heapify at given idx  # This function also updates the position of the nodes # is used to
    smallest = idx
    left = 2*idx + 1
    right = 2*idx + 2

    if (left < self.size and self.array[left][1] < self.array[smallest][1]):
        smallest = left

    if (right < self.size and self.array[right][1] < self.array[smallest][1]):
        smallest = right

    if smallest != idx:                    # The nodes to be swapped in min # heap if idx is not smallest  # Swap positions
        self.pos[self.array[smallest][0]] = idx
        self.pos[self.array[idx][0]] = smallest

        self.swapMinHeapNode(smallest, idx) # Swap nodes
        self.minHeapify(smallest)

def extractMin(self):                       # Function to extract the minimum # node from heap

    if self.isEmpty() == True:              # Return NULL heap is empty
        return
```

```
        root = self.array[0]                    # Store the root node

        lastNode = self.array[self.size - 1]    # Replace root node with last node
        self.array[0] = lastNode

        self.pos[lastNode[0]] = 0               # Update position of last node
        self.pos[root[0]] = self.size - 1

        self.size -= 1                          # Reduce heap size and heapify root
        self.minHeapify(0)

        return root

    def isEmpty(self):                          # Function to check if array is empty or not
        if self.size == 0:
          return True
        else:
          return False

    def decreaseKey(self, v, dist):

        i = self.pos[v]                                 # Get the index of v in  heap array

        self.array[i][1] = dist                         # Get the node and update its dist value
```

```
        while (i > 0 and self.array[i][1] < self.array[(i - 1) // 2][1]):                    # Trave

            self.pos[ self.array[i][0] ] = (i-1)//2    # Swap this node with its parent
            self.pos[ self.array[(i-1)//2][0] ] = i
            self.swapMinHeapNode(i, (i - 1)//2 )

            i = (i - 1) // 2;                          # move to parent index

    def isInMinHeap(self, v):                          # A utility function to check if a given # vertex 'v' is in min heap or not
        if self.pos[v] < self.size:
          return True
        else:
          return False

def printArr(dist, n):                                # function to print vertex and distance from sourse
    print ("Vertex\tDistance from source")
    for i in range(n):
        print ("%d\t\t%d" % (i,dist[i]))
```

```
class Graph():                                        # Graph class
    def __init__(self, V):
        self.V = V
        self.graph = defaultdict(list)

    def addEdge(self, src, dest, weight):             # Adds an edge to an undirected graph

        newNode = [dest, weight]                      # Add an edge from src to dest.  A new node
        self.graph[src].insert(0, newNode)            # is added to the adjacency list of src. The   # node is added at the beginning. The first
                                                      # element of the node has the destination   # and the second elements has the weight
        newNode = [src, weight]                       # Since graph is undirected, add an edge
        self.graph[dest].insert(0, newNode)           # from dest to src also

    def dijkstra(self, src):                          # The main function that calculates distances # of shortest paths from src to all vertices.  #

        V = self.V                                     # Get the number of vertices in graph
        dist = []                                      # dist values used to pick minimum  # weight edge in cut

        minHeap = Heap()                               # minHeap represents set E
```

```
      for v in range(V):                              #  Initialize min heap with all vertices.          # dist value of
          dist.append(1e7)
          minHeap.array.append( minHeap.newMinHeapNode(v, dist[v]))
          minHeap.pos.append(v)

      minHeap.pos[src] = src                          # Make dist value of src vertex as 0 so # that it is extracted first
      dist[src] = 0
      minHeap.decreaseKey(src, dist[src])

      minHeap.size = V;                               # Initially size of min heap is equal to V

      while minHeap.isEmpty() == False:               # In the following loop, # min heap contains all nodes # whose shortest distance is not yet f

          newHeapNode = minHeap.extractMin()          # Extract the vertex # with minimum distance value
          u = newHeapNode[0]

          for pCrawl in self.graph[u]:           # Traverse through all adjacent vertices of  # u (the extracted vertex) and update their  # distance values
              v = pCrawl[0]

              # If shortest distance to v is not finalized
              # yet, and distance to v through u is less
              # than its previously calculated distance
              if (minHeap.isInMinHeap(v) and dist[u] != 1e7 and pCrawl[1] + dist[u] < dist[v]):
                      dist[v] = pCrawl[1] + dist[u]
                      minHeap.decreaseKey(v, dist[v])                    # update distance value # in min heap also
      printArr(dist,V)
```

```
[ ]  # I will test the software to test the above functions
     graph = Graph(5)
     graph.addEdge(0, 1, 1)
     graph.addEdge(0, 2, 8)
     graph.addEdge(0, 3, 3)
     graph.addEdge(1, 2, 2)            #(vertex, which vertex will go to, distance)
     graph.addEdge(2, 3, 4)
     graph.addEdge(2, 4, 1)
     graph.addEdge(3, 4, 9)
     graph.dijkstra(0)
```

```
Vertex  Distance from source
0               0
1               1
2               3
3               3
4               4
```

Task 5: -

a. In reference to the Object-Oriented Programming, what is Encapsulation and what are the advantages of it when you implement a Data structure, you need to examine these advantages. (Report)

Encapsulation: It is the collection of data along with methods that data into a single unit. It is a mechanism to restrict direct access to the components of an object and is defined as linking data and functions that deal with it together.

How is the encapsulation? (Encapsulation Protects Data)

It hides the values of the Structured Data Object inside the class making it difficult to access (preventing any unauthorized person with direct access).

Advantages	1- Data protection because no one will be able to see the methods in the code so that he cannot change any data or variables in order to hinder or sabotage the program. 2- The code is more flexible and cleaner and makes it easier to debug the errors inside. The needs can be changed (setter and getter) and the code can be made to be read or write only. 3- Ability to reuse code and change methods. 4- Helps increase security so that external inheritance can't get hold of data and makes it easy to maintain and improve.
Disadvantages	1- The size of the code increases as the methods increase, the length of the code increases. 2- You need more instructions The larger the code, the more instructions you need to provide. 3- Increasing the execution time of the code. The encapsulation affects the execution time of the code, so that when the code is long, you need more instructions, so it will take a longer time to execute.

b. You need to find three benefits of separating the implementation stage of a data structure and define it as ADT, Evaluate these benefits. (Report)

ADT (abstract data type): A mathematical model of a data structure that defines the type of data stored, the operations supported on it, and the types of parameters of the operations. ADT defines each operation but does not specify how it does it.

Advantages of ADT:

1- Strong, easy-to-understand and reusable code within the software system.
2- Reduce your code and clean extraction efforts.
3- Provides a strong data structure and easy access to data at any time and on any device.
4- Independent data representation.
5- Allows data to be stored on hard disks and provides data protection.
6- It helps to design efficient algorithms.
7- The user does not need to know the implementation of a data structure or understand any implementation details of any type (abstraction).
8- Types of data structures provide better images of the real world.

9- It makes it easy to detect errors and the program is robust.
10- It is based on the principles of object-oriented programming.

Disadvantages ADT:

1- It does not support fix and match sentences.
2- Do not reveal derivative operations.
3- Prohibition of arithmetic inference.
4- Only advanced users can make any change to data structures.
5- When there is a problem with the data structure, you need an expert.

what benefits of separating the implementation stage of a data structure:

The separation helps to create a well-organized system of data and storage so that it is able to adapt to change, which gives freedom to design and use programs, provides algorithms with the ability to process in a useful way and helps to become a programmer faster and better.

Three benefits of using implementation-independent data structures:

1- Memory uses can be improved through effective use of data structures such as linked list, array, etc. So that you can use memory in an orderly fashion.
2- When implementing the data structure, you can use it anywhere, which makes you can reuse it.
3- Abstraction serves as the basis for abstract data types that define the structure of the data.

c. When you define data structures as ADTs, there is an opinion said that this is the basis of Object-Oriented programming, with justification state whether you agree or not. (Report)

I agree because object-oriented programming is characterized by the use of procedural data abstraction and centered around data abstraction creators. Abstract data types are based on data type abstraction, that is, they organize data around operations and differ in efficiency, printing and verification in many cases and the strengths of the two parties differ when the strengths of one of the parties are the weak points the other. If we notice that when they are combined with each other, they become better.

This is because the types of data structures depend on the principles of object-oriented programming and are considered the basis of the object-derived direction. Object-oriented programming helps to produce reusable and structured code that is implemented in all programming languages. Also, classes in Object-Oriented Design are essentially ADTs where classes contain objects and a set of functions and methods that perform data operations. Therefore, ADTs are used to analyze and design algorithms and data structures. ADT is the basic pillar of Object Orientation.

There are several applications of object-oriented programming that use ADT, including:

1- Encapsulation as it is one of the basics of object-oriented programming and it maintains the implementation of each object and prevents access to this class and change of procedures, which helps to make the program more secure and avoid data corruption and loss. We note that we used this application in the ADT in the implementation of the data structure.

2- Abstraction, which is a method used to facilitate writing code, making you able to implement what you want without having to know the details, so it aims to relieve the programmer so that he can focus on other things with minimal effort and time. As we have noted, the use of abstraction in ADT makes it easier to implement any algorithm without having to spend a lot of time implementing it.

References: -

1- https://www.scaler.com/topics/data-structures/singly-linked-list/
2- https://www.tutorialscan.com/datastructure/singly-linked-list/
3- https://learningmadesimple360.blogspot.com/2021/08/advantages-and-disadvantages-of-stack.html
4- https://accessdatas.com/qa/what-are-the-advantages-and-disadvantages-of-lists.html
5- https://www.geeksforgeeks.org/advantages-and-disadvantages-of-linked-list/
6- https://www.codingninjas.com/codestudio/library/advantages-disadvantages-and-uses-of-a-doubly-linked-list
7- https://www.geeksforgeeks.org/advantages-and-disadvantages-of-array-in-c/
8- https://www.geeksforgeeks.org/difference-between-a-static-queue-and-a-singly-linked-list/
9- https://www.geeksforgeeks.org/selection-sort/
10- https://www.geeksforgeeks.org/quick-sort/
11- https://www.guru99.com/selection-sort-algorithm.html
12- https://www.techquintal.com/advantages-and-disadvantages-of-quick-sort/
13- https://techvidvan.com/tutorials/data-structures-asymptotic-analysis/
14- https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/
15- https://press.rebus.community/programmingfundamentals/chapter/encapsulation/#:~:text=Encapsulation%20is%20one%20of%20the,parties'%20direct%20access%20to%20them
16- https://www.thejavaprogrammer.com/advantages-and-disadvantages-of-encapsulation-in-java/
17- https://www.enotes.com/homework-help/what-advantages-disadvantages-data-stucture-479073
18- https://link.springer.com/chapter/10.1007/BFb0019443#:~:text=Object%2Doriented%20programming%20and%20abstract%20data%20types%20can%20also%20be,are%20organized%20around%20the%20operations.
19- https://iq.opengenus.org/time-and-space-complexity-of-queue/
20- https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP
21- https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm#:~:text=Asymptotic%20analysis%20is%20input%20bound,in%20mathematical%20units%20of%20computation.

STUDENT ASSESSMENT SUBMISSION AND

DECLARATION

When submitting evidence for assessment, each student must sign a declaration confirming that the work is their own.

Student name: Hasan Alhwietat **Student ID: 20120098**		**Assessor name: Dr. Murad Yaghi**
Issue date: 16/5/2022	**Submission date:** 16/6/2022	**Submitted on:** 16/6/2022
Programme: Computer Science		
HTU Course Name: Data Structures & Algorithms **BTEC UNIT Title**: Data Structures & Algorithms **HTU Course Code:** 30202200 **BTEC UNIT Code:** D/615/1649 **I AM REPEATING THIS UNIT*:** **(YES)** **(NO)**		

Plagiarism:

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalized. It is your responsibility to ensure that you understand **correct referencing practices**. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student signature: Hasan **Date:16/6/2022**