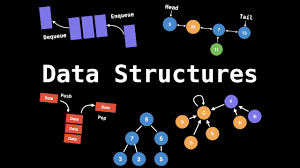
[2023]

LIYAN AQEL

[21110405]



[Final data structure]

[ENG: Dania AL Said]

Table of Contents

[Part One: 2](#_Toc137593635)

[Min Priority Queue 2](#_Toc137593636)

[The Formal Definition 2](#_Toc137593637)

[Abstract Data Type 2](#_Toc137593638)

[Applications of Min Priority Queue 3](#_Toc137593639)

[Three main Operations and Time complexity 3](#_Toc137593640)

[Illustration of min priority queue 7](#_Toc137593641)

[Implementation 11](#_Toc137593642)

[Part 2: 12](#_Toc137593643)

[The Definition of sorting algorithms 12](#_Toc137593644)

[The time complexity between two sorting algorithms 12](#_Toc137593645)

[The performance between two sorting algorithms 15](#_Toc137593646)

[Explain the result with the theoretical time complexity for two sorting algorithms. 16](#_Toc137593647)

[Space complexity 18](#_Toc137593648)

[Dijkstra’s algorithm and Bellman Ford algorithm 20](#_Toc137593649)

[Part 3: 23](#_Toc137593650)

[Stack Data Structure 23](#_Toc137593651)

[Pseudocode 23](#_Toc137593652)

[The Formal Definition 27](#_Toc137593653)

[Abstract data Type 27](#_Toc137593654)

[Call Stack 30](#_Toc137593655)

[Part 4 35](#_Toc137593656)

[Time Complexity 1 35](#_Toc137593657)

[The appropriate data structures and the reasons 36](#_Toc137593658)

[Time complexity 2 37](#_Toc137593659)

[Encapsulation and Information Hiding 38](#_Toc137593660)

[Imperative ADTs are a basis for object-oriented programming. 40](#_Toc137593661)

[Evaluate the benefits of using implementation independent data Structures. 41](#_Toc137593662)

# Part One:

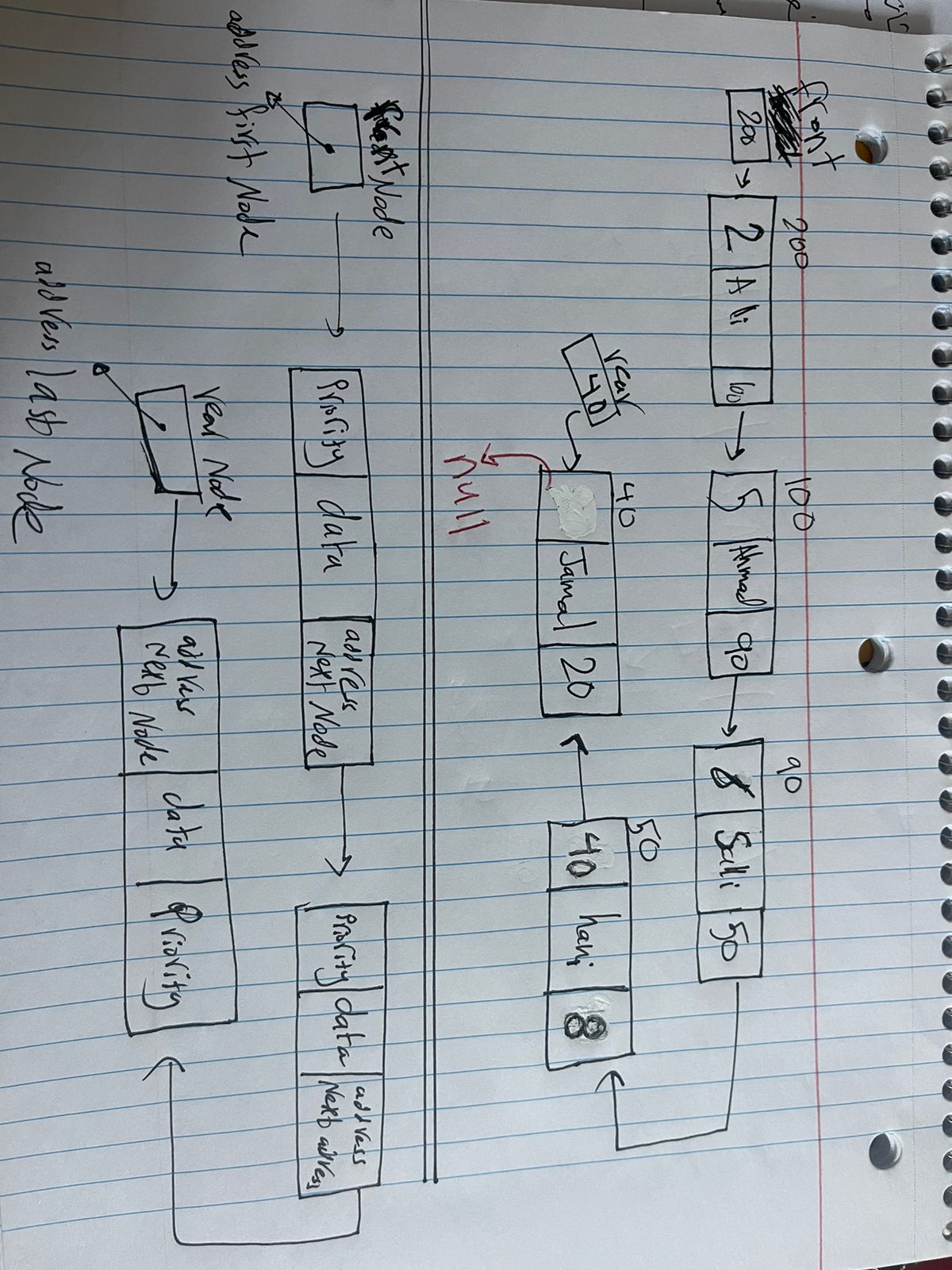
# Min Priority Queue

## The Formal Definition

The min-priority-queue is a data structure which is nearly similar to the queue. Each one of them follows a certain pattern to store data in the memory. The queue follows first in first out ways. It has a front part to remove data and rear part to insert data. In the queue, we can’t insert data in the middle. On the other hand, the min-priority-queue depends on arranging data according to the minimum priority which is in the first and the highest priority in the end. Data can be added in the first, middle or end.

## Abstract Data Type

I drew a specific design using linked List for min-priority-queue. As seen in a picture below, I drew the front node to indicate the first node added to the linked list and help me to know where the linked list starts from. Also, I drew the rear node to indicate the last node and help me to know where the linked list ends. In addition to the front and rear nodes, there is collection of nodes which can be created when I need to insert data to linked list. Each node includes three things which are the data, the priority of data, and the next address to the following node.



## Applications of Min Priority Queue

* **Operating System Process Scheduling:**

This application is an operating system in which many processes are carried out for CPU according to many algorithms which give process different levels or degrees of importance of execution. One of these algorithms follows the min-priority queue that is each process is given a number which shows which process to be carried out first. The process which has the lowest number is the process which has the highest priority to be carried out in the queue. This ensures efficient execution of the most important or urgent processes first before other less important processes. This application is very important in cases which have processes with different degrees of urgency. As a result, it helps effecting execution of processes according to their priorities and so ensures the efficient response of the system and achieving goals on time.

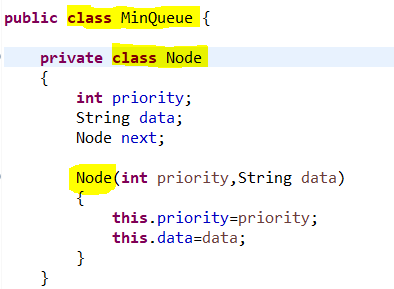
* **Load Balancing:**

This application is a load balancing system which follows the min-priority queue which means that the less loaded severs have the priority to deal with the new requests. In this application, the load is determined by CPU utilization, the processing capacity, or the length of the queue. It distributes the load on different servers equally using the resources available by directing the tasks to the server with the least load. So that no server in the system is overloaded while other servers are not used enough. This application is important because it increases the usage of available resources and prevents overload on some servers, which makes the system’s response better and more efficient.

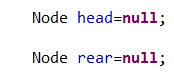
**In short,** it is very important in load balancing by selecting the least loaded server efficiently and distributing the workload evenly among servers and so improve the performance and capacity of the servers in the system.

## Three main Operations and Time complexity

**First,** I created a node class because min priority queue is a collection of nodes so if I want to insert data in this queue, I need to create a node by doing object from node class.



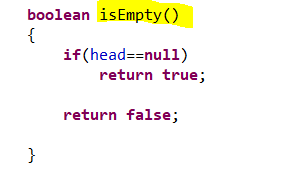
**Then**, I created **the head Node** to indicate the first node in the linked List and **rear node** to indicate the last node in the linked List.



1. **The first operation I wrote is ( isEmpty() )** to check if the queue linked list is empty or not

**Note:** if the head node is null by default, the rear node is null

**Time complexity:** In this operation, the time complexity is **O (1)** because it has one operation which checks if the head node in linked list is null, so this operation takes constant time to be executed and it doesn’t depend on size of linked list.



1. **The second operation I wrote is (void insert(int priority,String data))** to insert value in the queue linked list

**Time Complexity:** In this operation, the time complexity depends on the operation that the user is doing. If the user inserts a value in the empty queue linked list, the **Time Complexity will be O (1)** because we only check if the queue is empty or not.

If the user inserts a value that its priority is less than the priority of the front, the **Time Complexity will be O (1)** because we only need to update the head node.

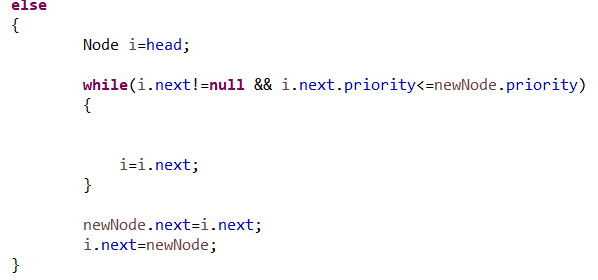
If the user inserts a value that its priority is more than the priority of the rear, the **Time Complexity will be O (1)** because we only need to update the rear node.

These three cases are the best-cases scenario of our operation.

A picture containing text, screenshot, font

Description automatically generated

If the user inserts a value that its priority is between the priorities of the two nodes**,** the **Time Complexity will be O (N)** because if the priority is in the middle of the priorities of the two nodes, we need to traversal all the nodes in the queue until we reach the right position according to the priority by using while loop. This is the worst-case scenario of our operation.



**In short,** if the user needs to insert values in the beginning and the end, the time complexity will be O (1). if the user needs to insert values in the middle, the time complexity will be O (N).

At last,

TC=n +1+1+1 🡪 TC= n 🡪 Time complexity of insert method = O (N).

1. **The third operation I wrote is (void removeMin())** to remove value from queue linked list

**Note:** this operation just removes the first node which has minimum priority that is in the front of the queue.

**Time Complexity:** In this operation, the **Time Complexity** is O(1) in all cases.

The first case is when the linked list is empty, the time complexity is O(1) because the operation will only return true or false.

The next case is when the linked list has only one node, the time complexity is O(1) because the operation will only update the value of the head and rear nodes to be null.

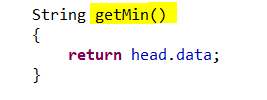
The last case is when the operation is to delete the value of the first node which is the front node, the time complexity is O(1) because the operation will only update the head node and delete the first node.

TC=1+1+1 🡪 TC= 1 🡪 Time complexity of removeMin method = O (1).



1. **The fourth operation I wrote is ( Sting getMin())** to retrieve or return the data from the first node which has minimum priority that is in the front of the queue.

**Time Complexity:** In this operation, the **Time Complexity** is O(1) because it just returns the value of first node and it doesn’t depend on size of linked list.

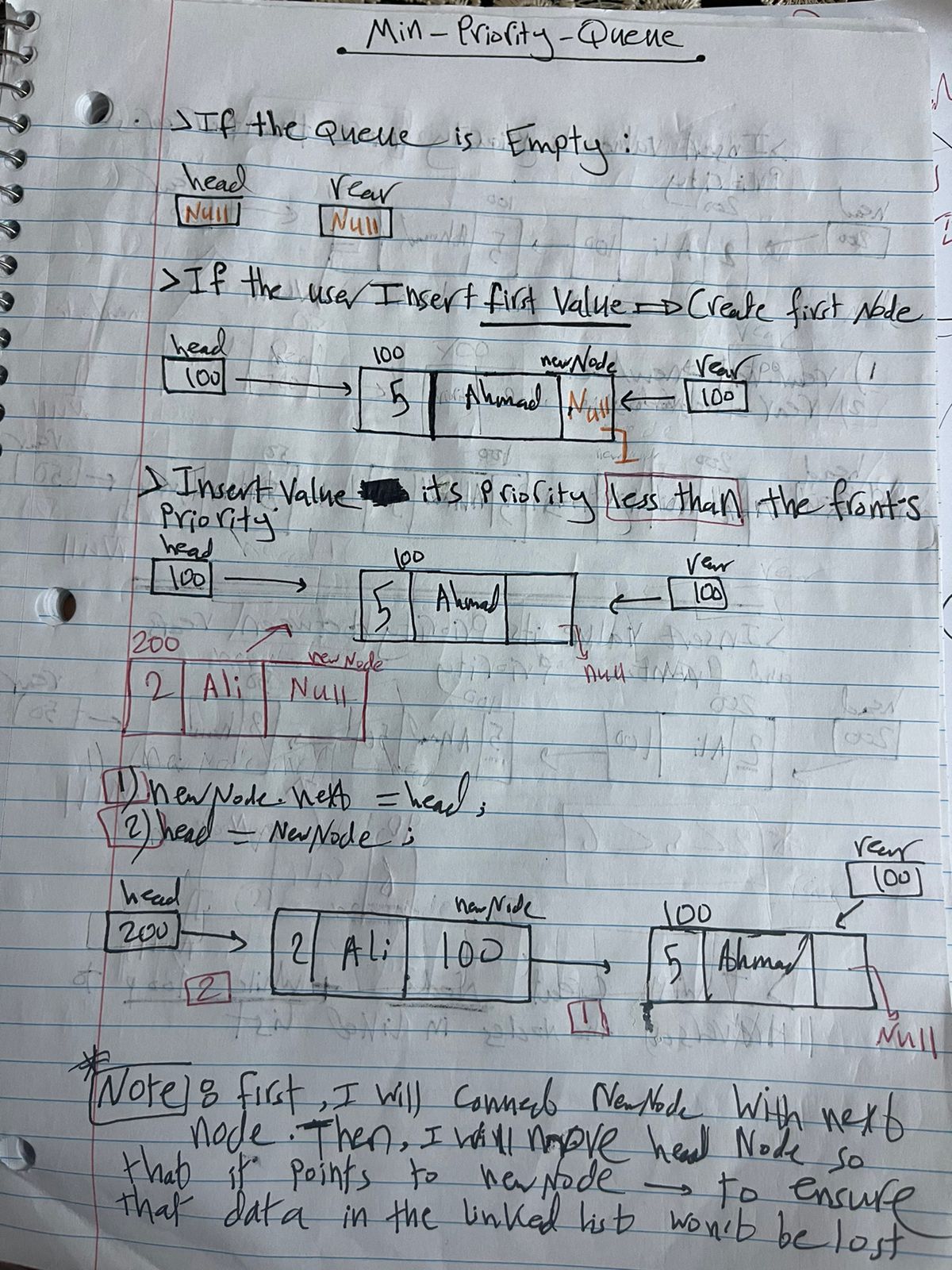


# Illustration of min priority queue

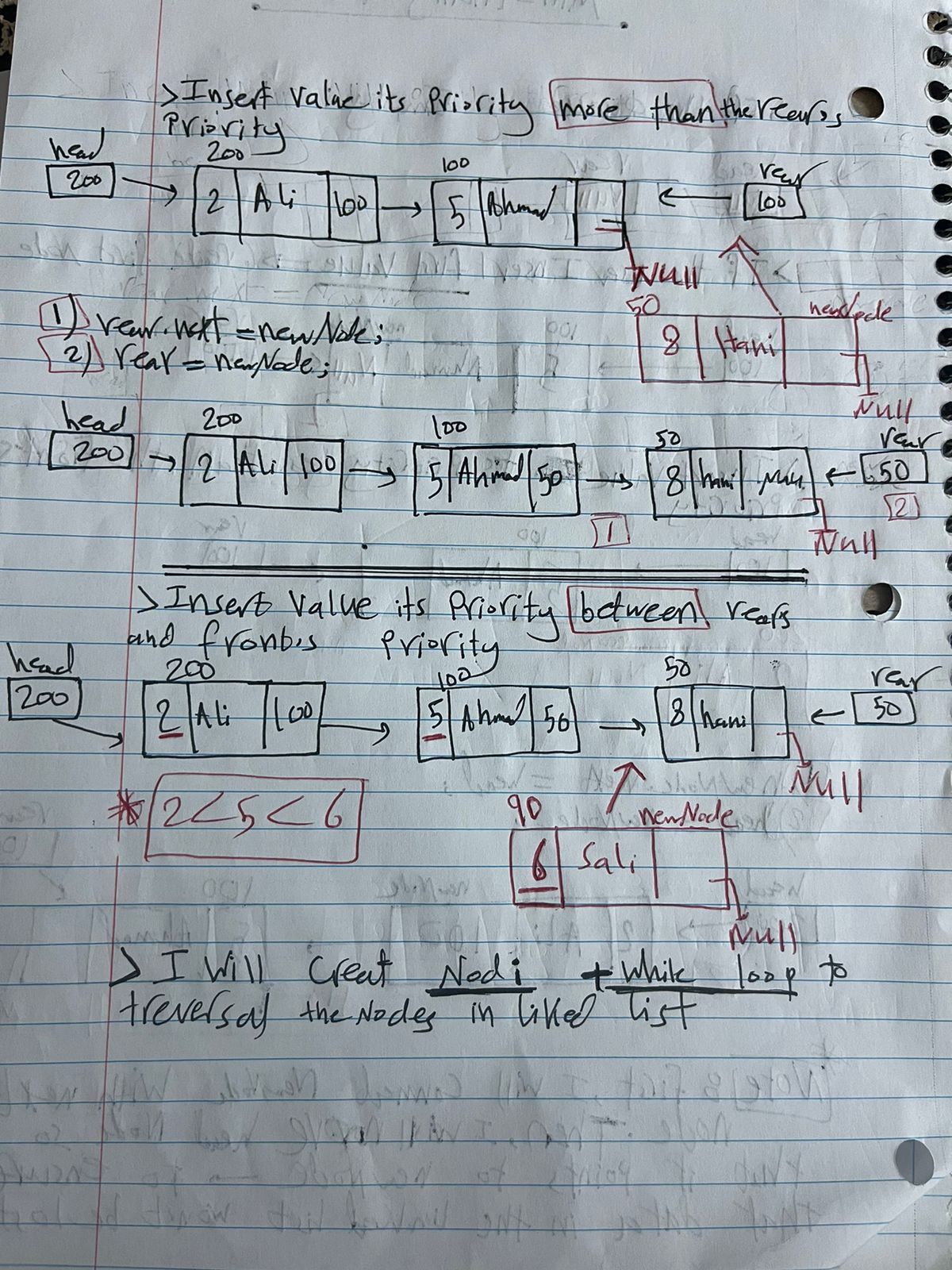
**Now I will draw the illustration of min priority queue:**

**Note: the head node is the same as front node but with different names.**

**Page 1:**



**Page 2:**



**Page 3:**

A piece of paper with writing on it

Description automatically generated with low confidence

**Page 4:**

A piece of paper with writing on it

Description automatically generated with low confidence

**Page 5:**

A paper with writing on it

Description automatically generated with low confidence

# Implementation

The Code is done.

# Part 2:

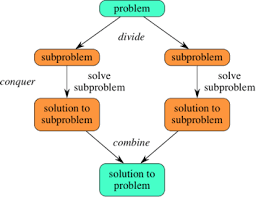
## The Definition of sorting algorithms

**The selection sort:**

It is a type of algorithms sorting which works by comparing the minimum or maximum value with the elements in the array. The comparison is different from one code to another according to goals achieved from the code. It is suitable and efficient for simple array rather than complex array.

**The Merge sort:**

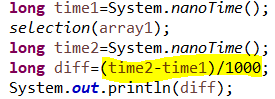
Merge sort algorithm is a type of algorithms sorting which follows divide and conquer strategy which means repeatedly dividing the problem into parts which are called sub parts, and then the solution will be found for each problem, then all solutions will be collected in one solution. Dividing the problem into parts from the same main problem is called recursion, which is a way to solve parts of the problem in which a function calls the same function inside itself. Both recursion and divide and conquer strategy are used to solve big or complex problems at the same time. This algorithm is suitable and efficient for a large array.



**There are two rules to sort data which are sorting data of the same type such as numbers, characters, and symbols which is called homogenous. Another rule is sorting data depending on a different criterion such as ascending or descending.**

## The time complexity between two sorting algorithms

**Note: I calculated the time complexity by microsecond.**



**Selection sort**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 5000 | 50000 | 500000 |
| Sorted | 9616 | 234436 | 21648806 |
| Reversely sorted | 14901 | 619699 | 64733009 |

**Merge sort**

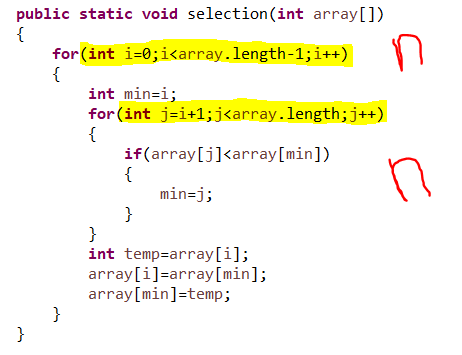
|  |  |  |  |
| --- | --- | --- | --- |
|  | 5000 | 50000 | 500000 |
| Sorted | 1456 | 8150 | 45998 |
| Reversely sorted | 1215 | 9712 | 60364 |

**What is the theoretical time complexity (Big-O) of each of the sorting algorithms?**

|  |  |  |
| --- | --- | --- |
|  | Selection sort | Merge sort |
| Best case | O (n^2) | O (n log n) |
| Worst case | O (n^2) | O (n log n) |

**The selection sort algorithm has O (n^2) time complexity in the best and worst** because this algorithm has two for loop which are outer loop and inner loop. The outer loop iteratives on all elements in array, and in each loop the inner loop search for the minimum element from unsorted elements in the array and then if the inner loop finds the minimum element, it changes the minimum element with the index of the current element then swapping between them.

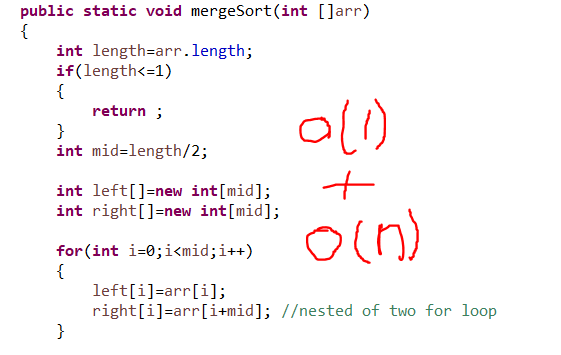
So, if the array is sorted or not in two cases the algorithm will iterate on all the array in outer loop and in each loop, It compare the minimum element from unsorted elements in inner loop. So regardless of the number of comparisons and the swapping the result of time complexity is O (N^2) because it requires N for the external loop (outer loop) and N for the internal loop (inner loop).



**The merge sort algorithm has O (n log n) time complexity in the best and worst case** because this algorithm is done by two methods which are the mergeSort() method and the merge() method.

The mergeSort() method will divide the array into parts recursively until it reaches the base case which is when the length of the array is less than 2 (equals 1) in our case .

When we reach the base case, all the arrays will be stored because each one only has one element. the dividing process has taken O (1) time complexity until the code reaches the base case and the filling process of the array has taken O (n) time complexity.



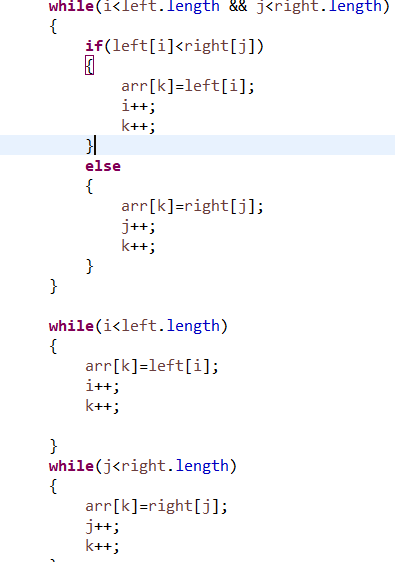
In this method the process of calling the same method recursively inside itself needs O (log n) time complexity.



every time I called the recursion method, the dividing and filling processes of the array repeated inside it, so the time complexity is n log(n)

By doing the previous steps above, I will have finished sorting the array by diving it into parts. Then I will collect all the parts into one part. This means I will collect all the subarrays to find the sorted array. This is done by calling the merge () method. So now I will calculate the time complexity for this method.

The merge () method will merge two subarrays in a sorted way in the main array by comparing and replacing the elements, so it needs a while loop to iterate the elements on the two arrays until we reach a sorted array and this need O (N) time complexity.



So the time complexity of the merge algorithm is 🡪 T (N)= 1+ n+ n log n 🡪 O (n log n)

## The performance between two sorting algorithms

**The selection sorting algorithm 🡪 sorted data.**

if we insert sorted data in this algorithm, it will keep iterating on all the elements in outer loop. In the inner loop, it will search for the minimum element by comparing the element with the other elements in array. This algorithm makes unnecessary processing such as comparison and swapping the elements and wastes time, so this takes O (n^2) time complexity, so this algorithm is not efficient for sorted data.

**The selection sorting algorithm 🡪 reversely sorted data**

if we insert reversely sorted data in this algorithm, it will keep iterating on all the elements in outer loop. In the inner loop, it will search for the minimum element by comparing the element with the other elements in array. In this case, it will iterate on all the elements in each loop, so this takes O (n^2) time complexity, and it doesn’t have any features in reversely sorted data. As a result, this algorithm is not efficient in this case.

**The merge sorting algorithm 🡪 sorted data.**

if we insert sorted data in this algorithm, It works by dividing recursively the array into smaller parts which are called subarrays until every subarray has one element, and then arranging and collecting them individually to find sorted array. These processes are required even though the inserted data is sorted or unsorted. The first steps of this process are essential, which are the dividing and the recursion steps, but other steps can be improved such as, merge process because the data is already sorted and when we divide it the subarrays will be already sorted so in the merge process, we will skip some process and the algorithm will be faster. In short, in this algorithm time complexity which is O (n log n) stays the same because the processes are performed in the same way, but the algorithm will be faster because some steps are skipped.

**The merge sorting algorithm 🡪 reversely sorted data**

If we insert reversely sorted data in this algorithm, it will divide recursively the array into smaller parts which are called subarrays until every subarray has one element, and then arranging and collecting them individually to find sorted array. In this algorithm the number of processing is the same and no steps are skipped; on the contrary, the steps are repeated so we need more effort because we insert reversely sorted data. In other words, the merge process cannot be skipped because I need to compare and merge two elements in two subarrays. it takes the time complexity which Is O (n log n).

**In short,** the merge algorithm and its O (n log n) time complexity has better performance than the selection algorithm with its O (N^2) time complexity either we used sorted array or reversely sorted data. However, the merge sort is more efficient for large data. Moreover, there are other factors determining the choice of the algorithm such as the data, the problem, and the resources available.

## Explain the result with the theoretical time complexity for two sorting algorithms.

**The selection sorting algorithm 🡪 sorted data and reversely sorted data.**

when I calculated the time complexity for the selection sort in the previous table whether I inset sorted array with size 5000/50000/500000 or inset reversely sorted array with size 5000/50000/500000, the time complexity increases quadratically with the size of the array regardless the inserted data is ascending or descending. **Moreover,** the time needed to sort an array sorted is less than sort a reversely array sorted. The selection algorithm has time complexity that is O(n^2) and n refers to the size of the array.

**The merge sorting algorithm 🡪 sorted data and reversely sorted data.**

when I calculated the time complexity for the merge sort in the previous table whether I inset sorted array with size 5000/50000/500000 or inset reversely sorted array with size 5000/50000/500000, the time complexity increases logarithmically with the size of the array regardless the inserted data is ascending or descending. **Moreover,** the time needed to sort an array sorted is less than sort a reversely array sorted. The merge algorithm has time complexity that is O (n log n) and n refers to the size of the array.

The selection algorithm increasing quadratically with the size of the array and the result of time in above tables increasing quadratically with the size of the array regardless the inserted data is ascending which means sorted array or descending which means reversely sorted array.

The result of the theoretical time in the tables above and the graphs below prove that the theoretical time complexity compliance with the time the algorithm needs to sort the arrays.

The merge algorithm increasing logarithmically with the size of the array and the result of time complexity in above tables increasing logarithmically with the size of the array regardless the size of the array and the inserted data is ascending which means sorted array or descending which means reversely sorted array. So, the result of the theoretical time complexity in the tables above and the graphs below prove that the theoretical time complexity compliance with the time the algorithm needs to sort the arrays.

**Sorted array 🡪 selection.**



**Reversely Sorted array 🡪 selection**

A graph with a dotted line

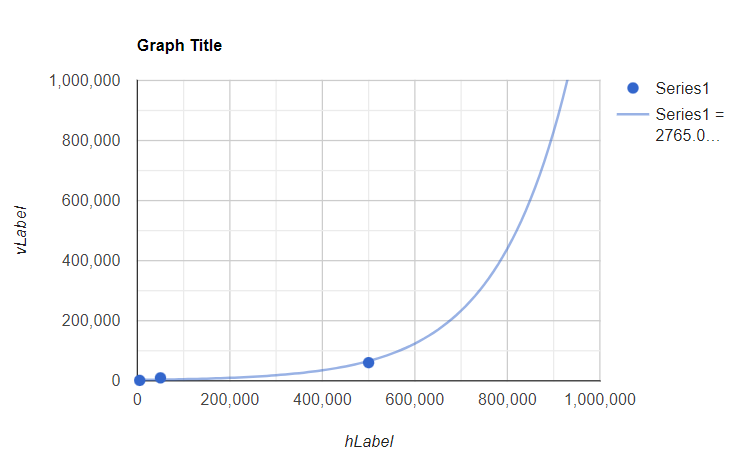
Description automatically generated with low confidence

**Sorted array 🡪 merge.**

A graph with a line

Description automatically generated with low confidence

**Reversely Sorted array 🡪 merge**



## Space complexity

**Big O notation** is a mathematical symbol to describe the worst-case of the algorithms regarding time or space complexity. It can be expressed in many ways such as O(1) which is called constant time/space complexity and it is the best one, O(log n) which is called logarithmic time/space complexity, O(N) which is called linear time/space complexity, O(N^2) which is called quadratic time/space complexity, and O(2^N) which is called exponential time/space complexity and it is the worst one.

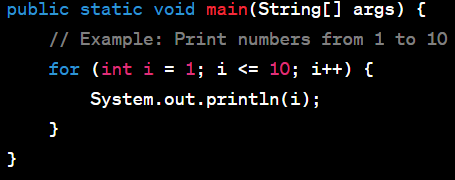
**Time complexity** is a way to describe the relationship between the runtime of the function and the size of the input.

As we learnt before, time complexity is one measure of the efficiency of algorithms. There is another important measure which should not be missed, that is space complexity.

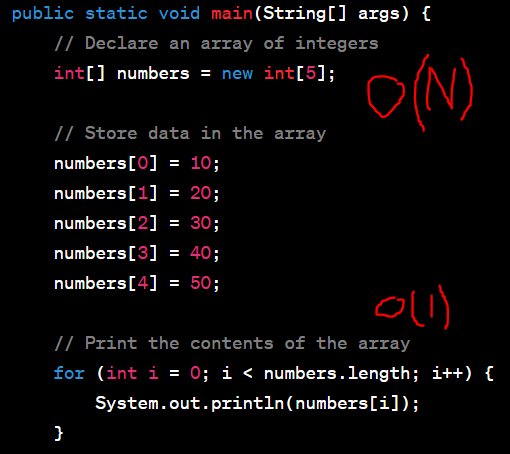
**Space complexity** is the relationship between the input size and the extra memory that the algorithm uses. That means the amount of space which I need to reserve of the total space of the memory depending on what algorithm I used to solve the problem. To find out the space complexity of the algorithm, we take into account the worst-case which means the largest size of the input.

It is important to focus on space complicity because we may have two algorithms with the same time complexity, but they have different space complexity. For example, the array list and the linked list may have the same time complexity which is O(N) but the space complexity is not the same. The first example is the array. When I make declaration of the array in the memory, I need to determine fixed size which cannot be modified later but this has two disadvantages. First, when the array reaches the maximum size, data cannot be added. In this case I need to create a new array with a bigger size which needs more space. Another disadvantage is that I may make array with very big size which I don’t need. So, I reserve extra space in the memory which may not be used. The second example is the linked list. It is a collection of nodes which is impossible to be determined by a fixed size because it is dynamic size which means its size increases when data is inserted and decreases when data is deleted.

In short, the linked list is more flexible because its size is dynamic, but the array is fixed size. However, the array doesn’t require extra memory for the links between data, so it is more efficient in this case.



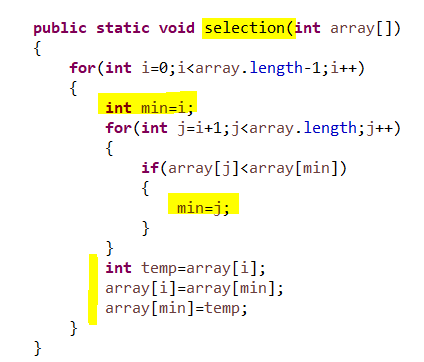
In the above code, space complexity is O(1) that is it doesn’t depend on extra input size because the memory is only used for the loop by using one variable which is (i).



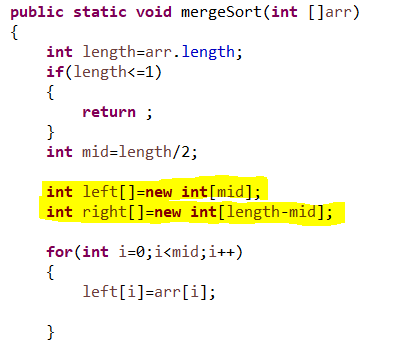
In the above code, space complexity is O(N) n= the size of array which depends on the extra input size. If the size of the array increases, the space will increase linearly while the next part of the code has loop which depends one variable which is (i) and doesn’t depend on the size of the array so its space complexity is O (1).

**Here is a description of the space complexity of the sorting algorithms.**

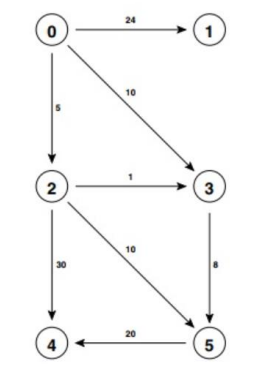
**The selection sorting algorithm has O (1) space complexity** which means that this algorithm uses fixed space, and it doesn’t need to reserve additional space in the memory. As seen in the code below, only the temporary variable is needed, and it will be compared to the data in the memory then swapping between these two elements in the same array so we will not need to use new data structure.



**The merge sorting algorithm has O (n) space complexity** which means that this algorithm will need to use additional space in the memory according to the size of the input data for any data structure. As seen in the code below, during the stage of dividing the array into two parts, we needed to define two new arrays which are called sub arrays taking into account the size of the inserted array. As a result, we needed extra space in the memory.



## Dijkstra’s algorithm and Bellman Ford algorithm



**Dijkstra algorithm**

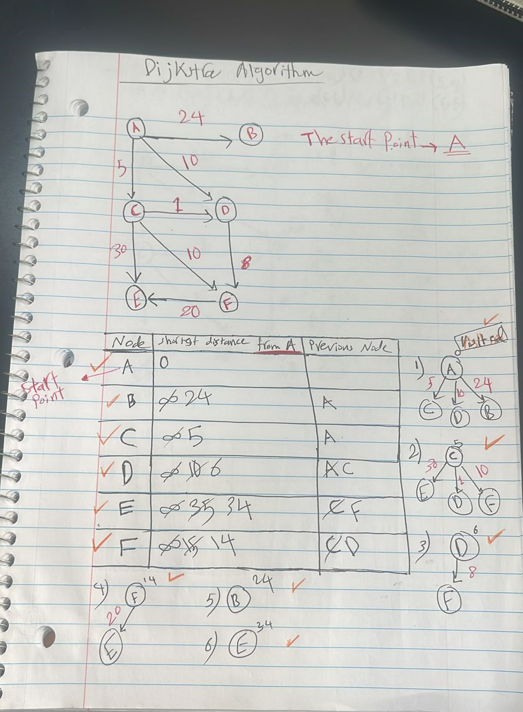
**It is done in three steps. The First step,** find the start vertex which is A, thengive the starting vertex distance 0 and other vertices ∞.

|  |  |  |  |
| --- | --- | --- | --- |
|  | The vertices | The shortest distance from 0 | The previous vertices |
| 0 | A | 0 |  |
| 1 | B | ∞ |  |
| 2 | C | ∞ |  |
| 3 | D | ∞ |  |
| 4 | E | ∞ |  |
| 5 | F | ∞ |  |

**The second step,** we sum the distance of the current vertex to the distance of its neighbors and compare it with the distance of the neighbor, if it the sum is less than the distance of the neighbor then we will replace the distance of the neighbor with the sum of distances ,add or replace the current vertex to the previous vertices’ column and make the current vertex a visited vertex.

|  |  |  |  |
| --- | --- | --- | --- |
| **The name of the vertices** | **The Index of the vertices** | **Shortest distance from 0 (A)** | **Previous vertices** |
| **A** | **0** | **0** |  |
| **B** | **1** | ∞ 🡪 24 | **A** |
| **C** | **2** | ∞ 🡪 5 | **A** |
| **D** | **3** | ∞ 🡪 10🡪6 | **A🡪 C** |
| **E** | **4** | ∞🡪35🡪 34 | **C🡪 F** |
| **F** | **5** | ∞🡪15🡪 14 | **C🡪D** |

**Third step,** visit the vertex that is the shortest distance and not visited, then repeat the second and third steps until all vertices are visited.



**Bellman Ford algorithm**

**It is done in two steps. The First Step, write all lists of edges.**

(0,1), (0,2), (0,3), (2,3), (2,4), (2,5), (3,5), (5,4) 🡪 (A,B), (A,C),(A,D),(C,D), (C,E),(C,F),(D,F), (F,E)

The number of these edges should equal the number of the edges in the graph which are 8.

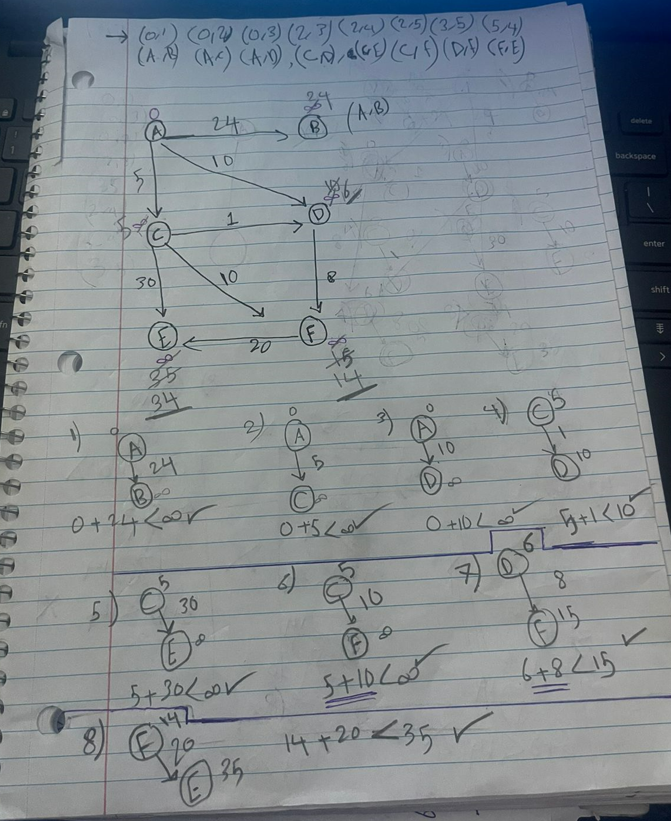
**Then,** give the starting vertices distance 0 and other vertices ∞ same as Dijkstra’s algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
|  | The vertices | The shortest distance from 0 | The previous vertices |
| 0 | A | 0 |  |
| 1 | B | ∞ |  |
| 2 | C | ∞ |  |
| 3 | D | ∞ |  |
| 4 | E | ∞ |  |
| 5 | F | ∞ |  |

**The second step, Do relaxation for all edges 🡪 |V|-1.**

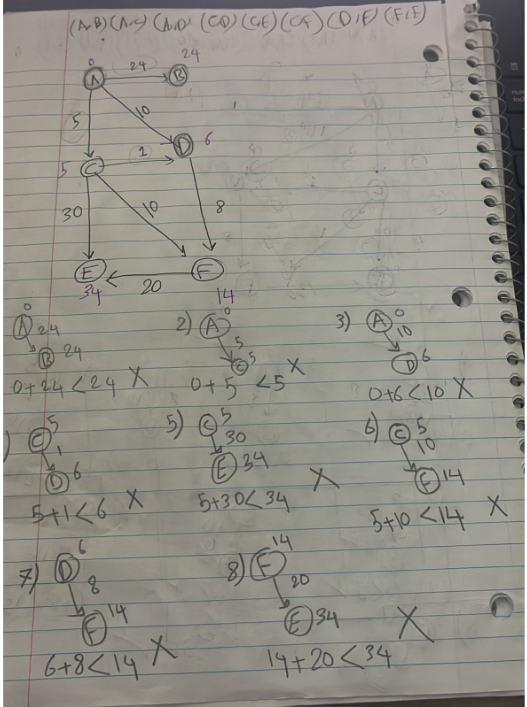
**The relaxation means that** we sum the distance of the current vertex to the distance of its neighbors and compare it with the distance of the neighbor, if it the sum is less than the distance of the neighbor then we will replace the distance of the neighbor with the sum of distances and the previous vertices.

**The First relaxation**



|  |  |  |  |
| --- | --- | --- | --- |
|  | The vertices | The shortest distance from 0 | The previous vertices |
| 0 | A | 0 |  |
| 1 | B | ∞ -> 24 | A |
| 2 | C | ∞ ->5 | A |
| 3 | D | ∞ ->10->6 | A -> C |
| 4 | E | ∞ -> 35-> 34 | C -> f |
| 5 | F | ∞ -> 15 -> 14 | C-> D |

**The second relaxation**



|  |  |  |  |
| --- | --- | --- | --- |
|  | The vertices | The shortest distance from 0 | The previous vertices |
| 0 | A | 0 |  |
| 1 | B | ∞ -> 24 | A |
| 2 | C | ∞ ->5 | A |
| 3 | D | ∞ ->10->6 | A -> C |
| 4 | E | ∞ -> 35-> 34 | C -> f |
| 5 | F | ∞ -> 15 -> 14 | C-> D |

**Here in the second iteration, the distances haven’t changed so here we can stop the relaxation loop.**

# Part 3:

# Stack Data Structure

## Pseudocode

Main()

{

Scanner scanner = new Scanner (System.in)

String text=scanner.nextLine(); //get the word from the user

If (is\_palindrome(text))

Print (is palindrome)

Else

Print (is not palindrome)

}

lower(c)

{

int asci = (int) c //

if (asci >= 65 && asci <= 90) //if the letter is capital

return (char) (asci +32); //convert the capital letter to the small letter

return c;

}

is\_equal(char [] s1,char [] s2)

{

if(s1.length() != s2.length())

return false;

for(i=0 to i<s1.length()) {

if(lower(s1[i]) != lower(s2[i])

return false;

}

Return true

}

is\_palindrome( String text)

{

Char [] s= text.toCharArray(); //convert the string to array of characters

LinkedStack stack=new LinkedStack();

For(i=0 to s.length())

{

Stack.push(s[i])

}

Char [] reversed = new char[s.length()];

For(i=0 to s.length())

{

Reversed[i]=stack.top();

Stack.pop()

}

Return is\_equal(s,reversed)

}

//Start class linked Stack

Class LinkedStack()

{

Class Node

{

char data

Node next

Node(data)

{

this.data=data

}

} //end class Node

top=null

isEmpty()

{

Return top==null;

}

Push(char data)

{

Node newNode= new Node(data)

If(top== null)

{

top=newNode

}

Else

{

newNode.next=top

top=newNode;

}

}

Char Pop ()

{

Char element;

If(isEmpty())

{

throw new IllegalArgumentException()

}

Else

{

Element =top.data

Delete=top

top=top.next

delete.next=null

}

Return element

}

Char top()

{

Return top.data;

}

} //end class linked stack.

## The Formal Definition

The stack is a type of data structure that is liner, and it works according to Last in First out (LIFO) which means the last element which is added will be deleted first from the same place which is called the top of the stack. It has four valid operations which will be explained later. It can be applied in two ways which are the array stack or the Linked stack. Either way, in any operation the time complexity is always O (1).

## Abstract data Type

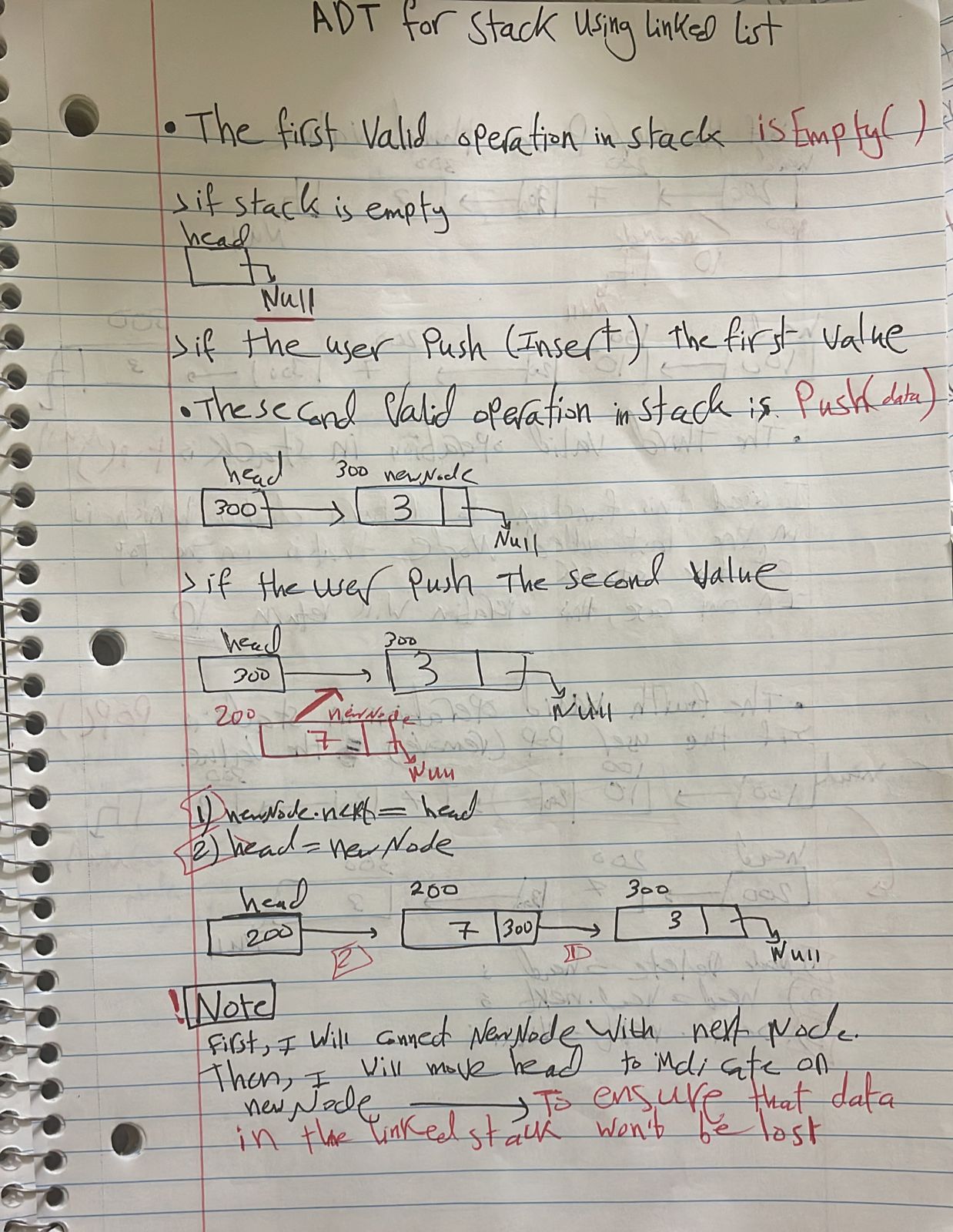
Push (data) 🡪 this function takes the data as a parameter in any data type, and it doesn’t return data. We used this function to insert data into the top of the stack.

Pop () 🡪 this function doesn’t take a parameter and it may return data which is the element that is removed. It may not return the data. We used this function to remove data in the stack from the same part which we insert data in that is called the top of the stack.

isEmpty() 🡪 this function doesn’t take a parameter and it returns Boolean value . We use it to check if the stack is empty or not.

TOP() 🡪 this function doesn’t take a parameter and it returns data . We use it to return the data which is in the top of the stack without removing or modifying it.

**Page 1:**



**Page 2:**

A piece of paper with writing on it

Description automatically generated with low confidence

**Page 3:**

A piece of paper with writing on it

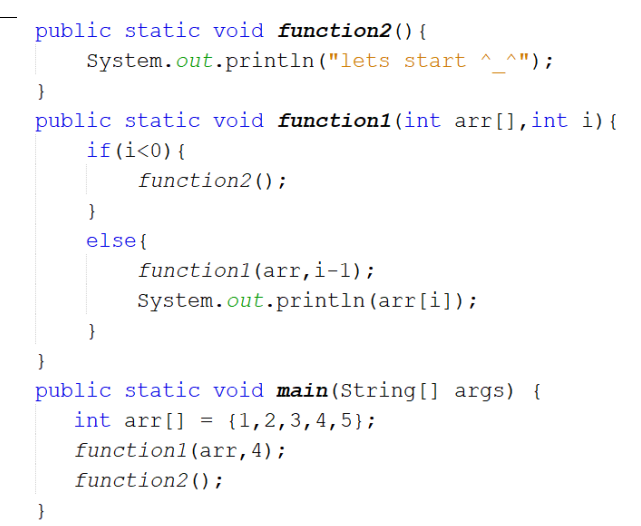
Description automatically generated with low confidence

# Call Stack

Now, I will use call stack to know how the code will execute the called functions.

As we know, any program will be stored in the main memory and the operating system will reserve the blook of segments for the program from the main memory. Each segment will store different data such as the segment stack which will store the called function and its local variable. Another example is the heap segment which will store the data of nonprimitive data type regardless of where the nonprimitive data type is declared in the class as global data type or in the method as a local data type.

**The segment stack** in the memory is a collection of frames. Each frame contains the called function and its local variables. First, the stack will be empty before running the program and call the functions, then the first frame for any program will contain the main method and its local variables.



**In our program**, the first frame will contain the main method with args and arr references only but not their data.

Next frame will contain function1(arr,4) with arr references and variable (i) with its value which is 4. Now we will check if the i<0 🡪 No because i=4 so block else will be executed and call function1(arr,3). Thus, the stack will pause function1(arr,4).

Next frame will contain function1(arr,3) with arr references and variable (i) with its value which is 3. Now we will check if the i<0 🡪 No because i=3 so block else will be executed and call function1(arr,2). Thus, the stack will pause function1(arr,3).

Next frame will contain function1(arr,2) with arr references and variable (i) with its value which is 2. Now we will check if the i<0 🡪 No because i=2 so block else will be executed and call function1(arr,1). Thus, the stack will pause function1(arr,2).

Next frame will contain function1(arr,1) with arr references and variable (i) with its value which is 1. Now we will check if the i<0 🡪 No because i=1 so block else will be executed and call function1(arr,0). Thus, the stack will pause function1(arr,1).

Next frame will contain function1(arr,0) with arr references and variable (i) with its value which is 0. Now we will check if the i<0 🡪 No because i=0 so block else will be executed and call function1(arr, -1). Thus, the stack will pause function1(arr,0).

Next frame will contain function1(arr,-1) with arr references and variable (i) with its value which is -1. Now we will check if the i<0 🡪 YES because i=-1 so block if will be executed and call function2() inside function1(arr,-1).

**Note: function1(arr,-1) is the base case in the recursion code so if it doesn’t exist, the code will be infinite and the stack will be overflow.**

Now, stack will execute function2() so print Lets start ^\_^ to the console, and then the stack will pop (remove) the function2() and the function1(arr,-1), then the stack will return to execute the previous call method which is fucntion1(arr,0) and inside it, print statement will be executed 🡪 arr[0]= 1 which is printed on the console, and then the stack pop(remove) this function .

After printing (1), the stack will return to execute the previous call method which is fuction1(arr,1) and inside it, print statement will be executed 🡪 arr[1]=2 which is printed on the console, and then the stack pop(remove) this function .

After printing (2), the stack will return to execute the previous call method which is fucntion1(arr,2) and inside it, print statement will be executed 🡪 arr[2]=3 which is printed on the console, and then the stack pop(remove) this function .

After printing (3), the stack will return to execute the previous call method which is fucntion1(arr,3) and inside it, print statement will be executed 🡪 arr[3]=4 which is printed on the console, and then the stack pop(remove) this function .

After printing (4), the stack will return to execute the previous call method which is fucntion1(arr,4) and inside it, print statement will be executed 🡪 arr[4]=5 which is printed on the console, and then the stack pop(remove) this function .

After printing (5), the stack will return to execute the previous call method which is main method and inside it, function2() will be executed and it prints Lets start ^\_^ to the console, and then the stack pop(remove) this function and the main function. Finally, the stack is empty, and the call of stack operation is ended.

The result of the code 🡪

lets start ^\_^

1

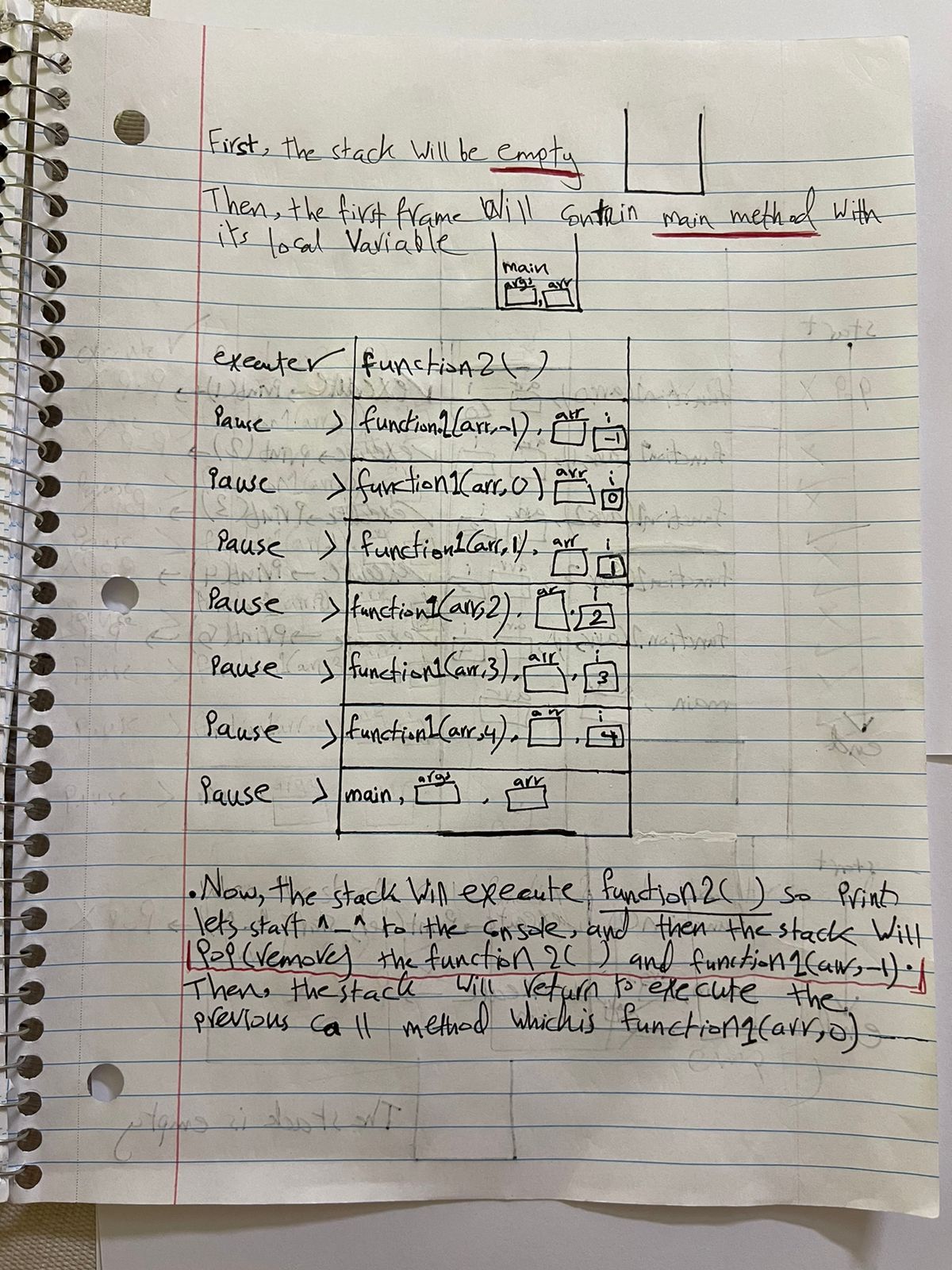
2

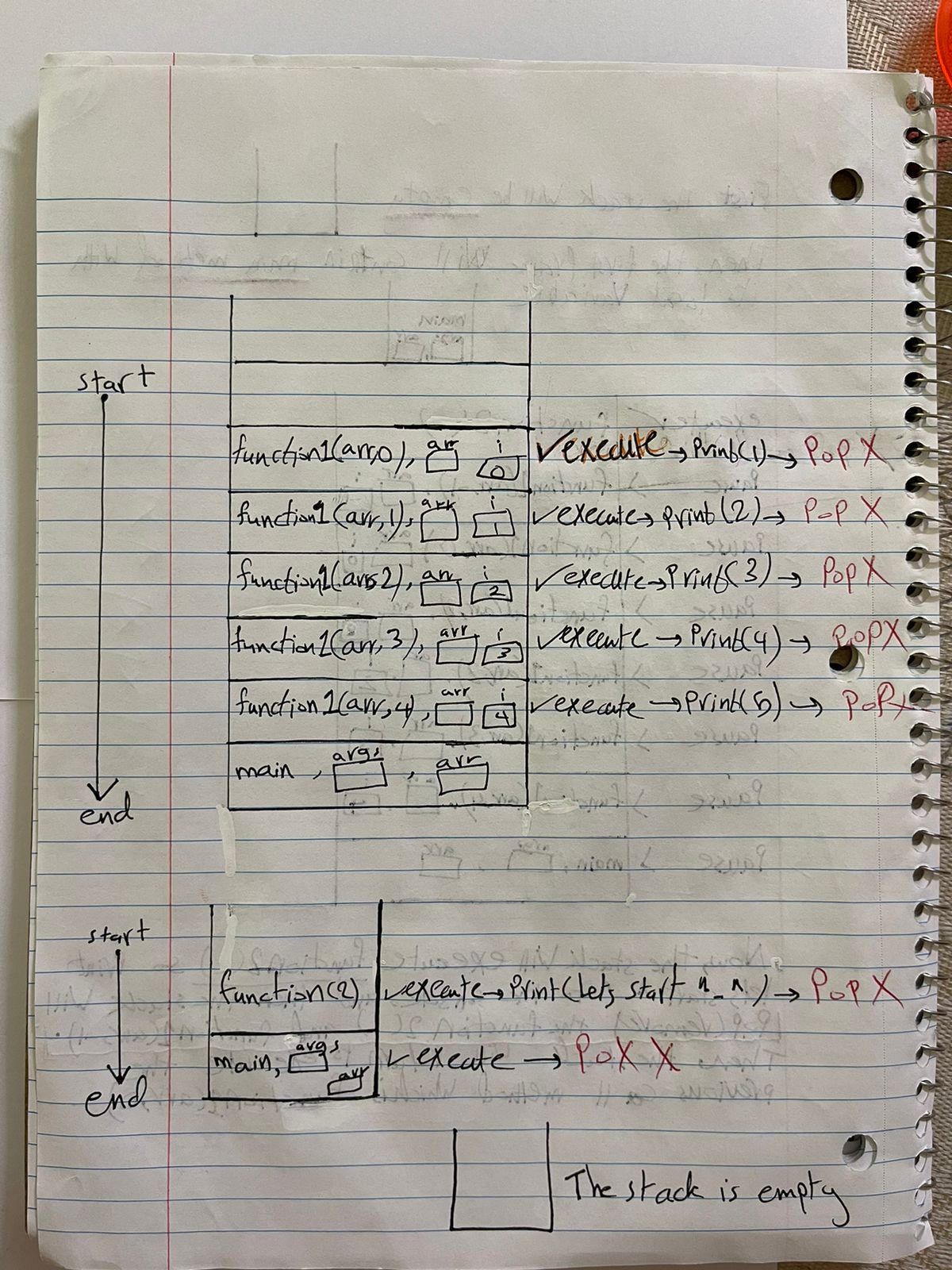
3

4

5

lets start ^\_^



s

# Part 4

## Time Complexity 1

Fill the following table with the time complexity of the operations based on the implemented data structure:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linked list | Array sorted | Array Unsorted |
| search | Linear search 🡪 O (N) | Binary search🡪 (log n) linear search 🡪 O (N) | Linear search 🡪 O (N) |
| Insert in the beginning | O (1) | O(N) | O(N) |
| Insert in the end | O (N) | O (N) | O (1) |
| Remove in the beginning | O (1) | O(N) | O(N) |
| Remove in the end | O (N) | O (1) | O (1) |

**First,** we start using list to solve the problems either by using array or linked. Each data structure has advantages and disadvantages.

**The Array List** is featured by **random access** so if I want to access any element, I need O (1) time complexity because the indexes for every element are Known. But its disadvantage is that it is not efficient for **the memory** because it reserves a block of cells next to each other. Meanwhile I can’t edit the size of the block which means the size of the array.

**The LinkedList** is featured by its flexibility of **memory** so that It is a collection of nodes and it is impossible to be determined by a fixed size because it is dynamic size which means its size increases when data is inserted and decreases when data is deleted. But in the linked list, the random access is not supported, so I need to traversal each element to access the required element, so I need O (N) time complexity.

**In short,** the linked list is more flexible because its size is dynamic, but the array is fixed size. But the array is more efficient than linked list regarding the accessing of elements.

**Now, I will explain the table above.**

In case I do search on the linked list, I will use the linear search not the binary search because this algorithm is restricted by the direct access to the element and as we know the linked list does not support random access. The linked list has a structure like a chain because every element has a reference to indicate the next element so that I need to traverse the chine from the beginning until the end to reach the required element. Also, I used linear search because I can move from one element to the next one until I reach the required element. It has the time complexity O (N).

In case I do search on the Array is Unsorted, I will use the linear search not the binary search because this algorithm is restricted by the fact that data must be sorted either ascending or descending. For example, if I search for element 4 in the unsorted array, the middle element is 6. According to the way the algorithm works, 4 will be on the left because it is less than 6 so the algorithm will search on the left side only and miss the right side of the array. Because the array is unsorted, there is a large possibility that element 4 is not in the left side but it may be in the right side.as a result the algorithm may not find it in the array even though it is there so the result of search will be wrong. Also, I used linear search because it is impossible not to see the required element as I move from one element to the next one. It has the time complexity O (N).

In case I insert element in the end of the unsorted array, and the array is full, I need to declare a new larger array and copy the elements which are already in the old array to the new array, I need O (N) time complexity.

when I want to remove the last value In the sorted/unsorted array, I just need O (1) time complexity to remove the last value because I know the size of the array.

## The appropriate data structures and the reasons

**Before choosing the suitable data structure for any algorithm we should Know some pieces of information.** First, the required operations especially the most operation used, the size of the data inserted, the type of the data used.

**The First Algorithm:**

Spell checker application. The data structure is used to store a dictionary of words. The data structure is constructed so that each node contains a word. When a user types a word, the spell checker searches for that word. If the word is found, it is considered spelled correctly. If the word is not found, the spell checker can suggest a list of words that are similar in spelling to the word that was typed, by searching for words that are similar in their prefix.

This algorithm needs many searching processes so I will choose a data structure which is very efficient in searching that is a sorted array because it is the **only** **one** of the three data structures mentioned before which allows the use of binary search. It is faster and has less time complexity which is O (log n) than other linear algorithm. Maybe we can consider the array sorted in some cases which is more efficient than linked list because we don’t need extra pointers in the array especially if the dictionary is very large.

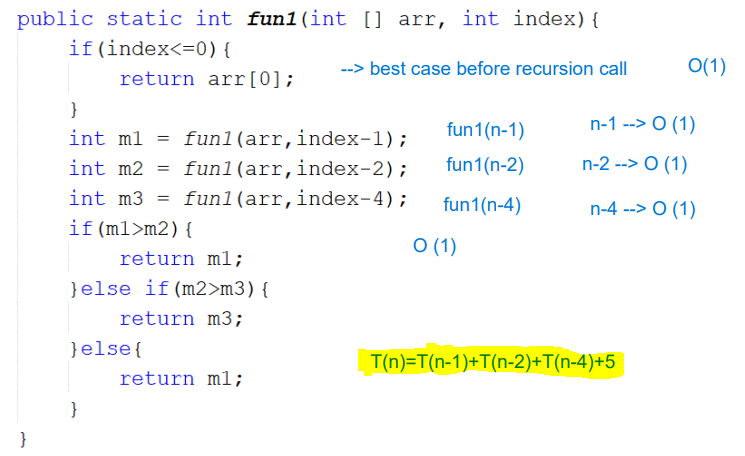
**The Second Algorithm:**

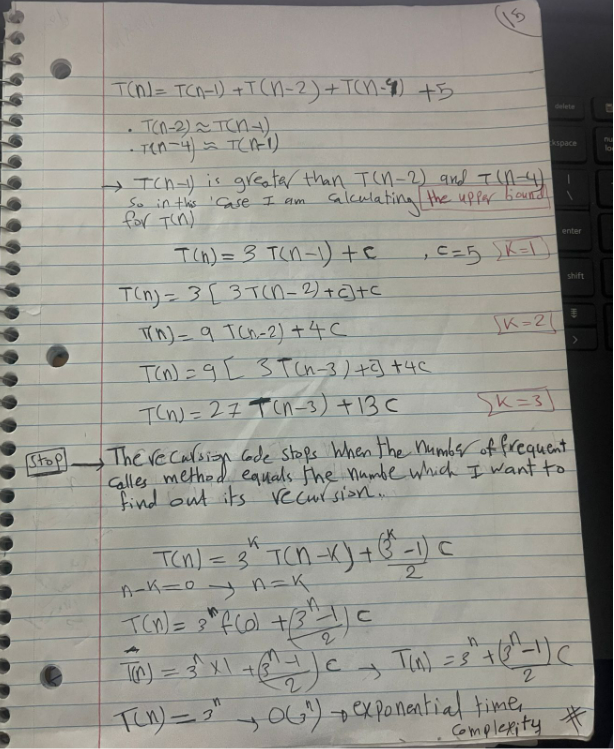
Priority queues. The data structure can be used to implement priority queues efficiently. It should allow for efficient insertion and deletion operations.

This algorithm needs many insertion and deletion processes so I will choose a data structure which is very efficient in deletion and insertion that is a linked list because it has O (1) time complexity in case deletion or insertion is in the beginning. This is because we only need to adjust the pointer.

Also, it has O (n) time complexity in the case of searching but it is not an important operation in the priority queue algorithm. In addition, I didn’t choose the array because it has O (n) time complexity in all the cases of insertion and deletion because it needs to shift the element. **In short,** the most suitable data structure of the choices given is the linked list because it is capable of implementing the priority queue with the processes of insertion and deletion in the most efficient way.

## Time complexity 2





## Encapsulation and Information Hiding

**The Encapsulation:**

It is an essential perception for OOP which makes writing code easier, more secure and more enhanced. It is a way of building and organizing your code in which sensitive data is hidden for users by declaring the attributes in the class as private and provide it with both methods which are getter and setter to access and update the private variable. It is not necessary to give each variable both methods because this depends on the privilege of the user who wants to deal with this data and whether he wants to read or write it.

**information hiding:**

it is a concept which is strongly related with encapsulation, and it is also essential for OOP in which the internal part of the code is hidden from the external code for users. In other words, data is provided with private access modifier in order to put restrictions on access of internal data for users. The restricted data can only be accessed from the class itself to ensure that sensitive data is fixed, and the external code cannot access it.

**In my code below** I provided the class Node and its variables which are priority, data, and next with private access modifier to ensure that these variables can only be accessed by MinQueue class and so preventing any external class from accessing the internal code. Another benefit of private access modifier is keeping the principle of the encapsulation and information hiding by preventing any external manipulation and so it keeps data structure controlled and integrated.

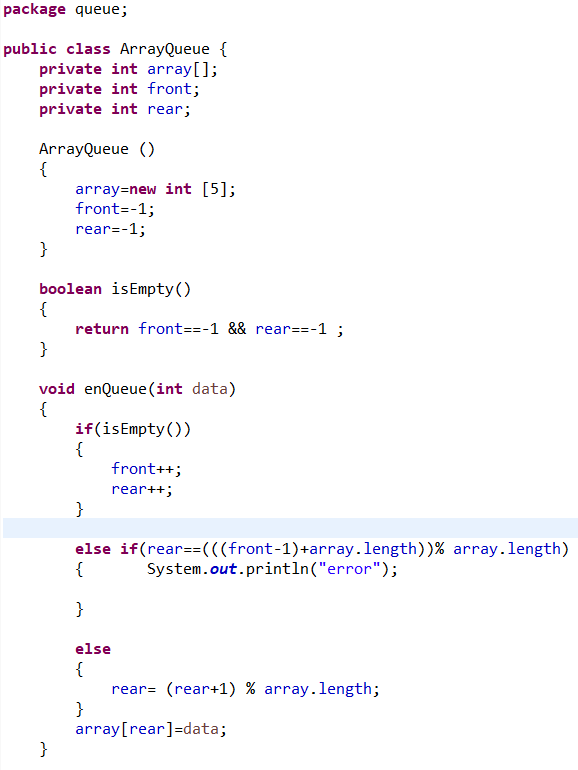


Another example of applying the encapsulation and information hiding on a code is the following.

I implemented the queue data structure by using the array.

**The encapsulation:** in MainQueue class, I restricted the internal data which are array, front, rear as private to prevent manipulation of this data by external code. It can only be accessed by method of the internal code to put necessary restrictions on the access of the sensitive data. These method are isEmpty(), enQueue(), deQueue(), front().

**The information Hiding:** I separated between internal and the external code by hiding the array, front, rear from the external code so that the external code cannot access or modify the rear and front. As a result, the method in the internal class made modification, access, and maintenance on the internal code easier without any interface of the external code.

 A screenshot of a computer program

Description automatically generated with medium confidence

There are many **benefits** of applying these two concepts to any data structure:

**Data integrity and protection:** it helps data integrity and protection and reduces the risk of corruption and inconsistency of data because these two concepts insure hiding the internal data from any external access and modification and using private access modifier.

**Modularity:** there are two ways of organizing data by separating it into classes and giving each class specific functions and operations. This will encourage modularity of the code and make testing, understanding, maintainability, and stability of the code easier because editing and changes on one module may not affect other modules and these modules are separated and don’t have direct access on each other.

**Abstraction:** encapsulation helps to hide the details of implementation of a code and focuses on the main qualities of a system. Abstraction helps to emphasize the necessary functions of any class and neglecting any underlying details. It makes using the code easier and helps reuse the code because it separates and organizes the interaction between the internal and external parts of the code.

**Abstraction:** it is a way of making any complex system or idea simpler by emphasizing the necessary aspects and neglecting any underlying details. It helps to make representations and models that emphasize the main qualities of a system, without focusing on the details of implementing of the code. Also, it helps developers to build a higher-level concept with no need to understand the internal work of the necessary component. **In other words,** it is a scratch that the developer makes before implementing the code.

## Imperative ADTs are a basis for object-oriented programming.

**ADT** stands for abstract data type which is a high-level description of data structure and the performed operations on it with no details of implementation. It is carrying out a set of operations on the data such as inserting, searching, deleting. It can be applied on array list, linked list, stack, queue, tree, graph. It helps to separate the abstract behavior from its implementation. The purpose of ADT is to carry out data structure depending on its expected behavior with no need to think about the detailed implementation of the code. Also, it ensures the maintainability, modularity, and reusability of the code because of using encapsulation of implementation of data within classes and each class has specific operations.

**The view** that imperative ADT is the basis of the OOP is based on the idea that there are many essential principles and concepts they have in common. However, OOP has many other basic principles and concepts as a basis rather than ADT. These concepts are related to the general concept and design of the OOP so we can say that ADT is not the only basis for OOP.

**Here are some reasons which support the view that ADTs are considered a basis of OOP:**

**Encapsulation:** both OOP and ADTs support encapsulation which is an essential perception for OOP and ADTs which makes writing code easier, more secure, reusable, and more enhanced. It is a way of building and organizing your code in which sensitive data is hidden from users. On the one side the ADTs hide the data and the operation inside a certain class or a module and separates the external interface and the internal code. In the same way in OOP data is hidden by objects and so only certain methods can access the internal hidden data.

**Reusing code:** both OOP and ADTs emphasis modularity and reusing code which is the process of using an already existing code without rewriting it from scratch by inviting its components to achieve a certain task or function. because on the one side, define operations in modules or classes so the code can be used again during many stages of the program. In the same way. Objects in the OOP encapsule data so that code can be reused in composition and inheritance of the code. As a result, these two paradigms ensure that the code can be easily reused, maintained, and extensible and make duplication of the code less.

**Foundation for Object concept:** the ADTs is considered an indicator to objects and a starting point to design and organize a code in object-oriented programming because ADTs gather data and operations in classes and use something similar to objects in OOP which adds more features such as polymorphism and inheritance.

**Historical and evolutionary connection:** the basis of OOP is procedural paradigms and programming which depends strongly on initial AOTs. As we know many programming languages emphasize ADTs before using OOP. Thus, capsulating data and operations in classes was the basis and led to gathering data in objects.

**In short,** although ADTs are considered a basis for OOP by supporting capsulating data, code reuse, modularity, and abstraction, the OOP made extensions and add extra features such as polymorphism and inheritance which made code writing more organized, flexible, reusable and extendable than imperative ADTs. For example, inheritance helps to have relationships between classes and so reusing the code and extend its functions. Also, polymorphism helps to treat objects flexibly.

**To conclude,** it is true that ADTs are considered a basis for OOP by emphasizing capsulating data, code reuse, modularity, and abstraction. but it is not the only basis for OOP. There are many others bases such as objects, classes, inheritance, polymorphism, encapsulation, abstraction. Even though OOP extended and added an additional feature to the previous aspects. ADT s are still the precursor and the starting point without which OOP would not have achieved such popularity, flexibility and efficiency.

## Evaluate the benefits of using implementation independent data Structures.

Implementation independent data structure refers to a data structure that depends on other data structure to be implemented, and designing and using data structures regardless of which programming language I am using. In this method, the interface and properties are separate from how this method is carried out which allows developers to use it in a flexible way regardless of using any dependent data structure.

**On the one hand,** there are many advantages of implementation independent data structure. **First of all**, it helps to organize the code in a better way, makes maintenance easier and improves readability of the code because it encourages modular design. This is due to the fact that it separates the data structure’s abstract behavior from its implementation, and this does not affect the code which depends on data structure. Also, it encourages modularity and changes on one module may not affect other modules**. Another advantage** is that it is committed to a common interface which is used by different programming languages so that it is used in many systems and projects. This will result in more productivity and consistency. Also, this will make development faster because developers can use already made interfaces and this reduces errors.

**Moreover,** it allows developers to choose from various implementations according to performance, and memory constraints. For this reason, it is flexible, and it can cope with the continuous change of technological advances and requirements. As a result, the code can develop quickly over time with no need to be modified. In other words, it can cope with new technologies and their requirements. **Moreover,** it makes testing and verification easier because it emphasizes the abstract behaviors of data structure. In other words, developers can make tests which are comprehensive and cover different scenarios of the code. Also, they can use different verification techniques to ensure safety and a high level of performance.

**To conclude,** implementation-independent data structures ensure flexibility, modularity and reusability of data.It makes using the code easier and helps reuse the code because it separates and organizes the interaction between the internal and external parts of the code.

* **References:**
* [**https://www.simplilearn.com/tutorials/data-structure-tutorial/priority-queue-in-data-structure**](https://www.simplilearn.com/tutorials/data-structure-tutorial/priority-queue-in-data-structure)
* [**https://levelup.gitconnected.com/sorting-algorithms-selection-sort-bubble-sort-merge-sort-and-quicksort-75479f8f80b1**](https://levelup.gitconnected.com/sorting-algorithms-selection-sort-bubble-sort-merge-sort-and-quicksort-75479f8f80b1)
* [**https://www.techopedia.com/definition/9523/stack**](https://www.techopedia.com/definition/9523/stack)
* [**http://www.dba-oracle.com/t\_object\_encapsulation\_abstract.htm**](http://www.dba-oracle.com/t_object_encapsulation_abstract.htm)
* [**https://tutorialsinhand.com/tutorials/data-structure-tutorial/data-structure-basics/advantages-and-disadvantages-of-data-structure.aspx**](https://tutorialsinhand.com/tutorials/data-structure-tutorial/data-structure-basics/advantages-and-disadvantages-of-data-structure.aspx)
* [**https://www.linode.com/docs/guides/data-structure/**](https://www.linode.com/docs/guides/data-structure/)

