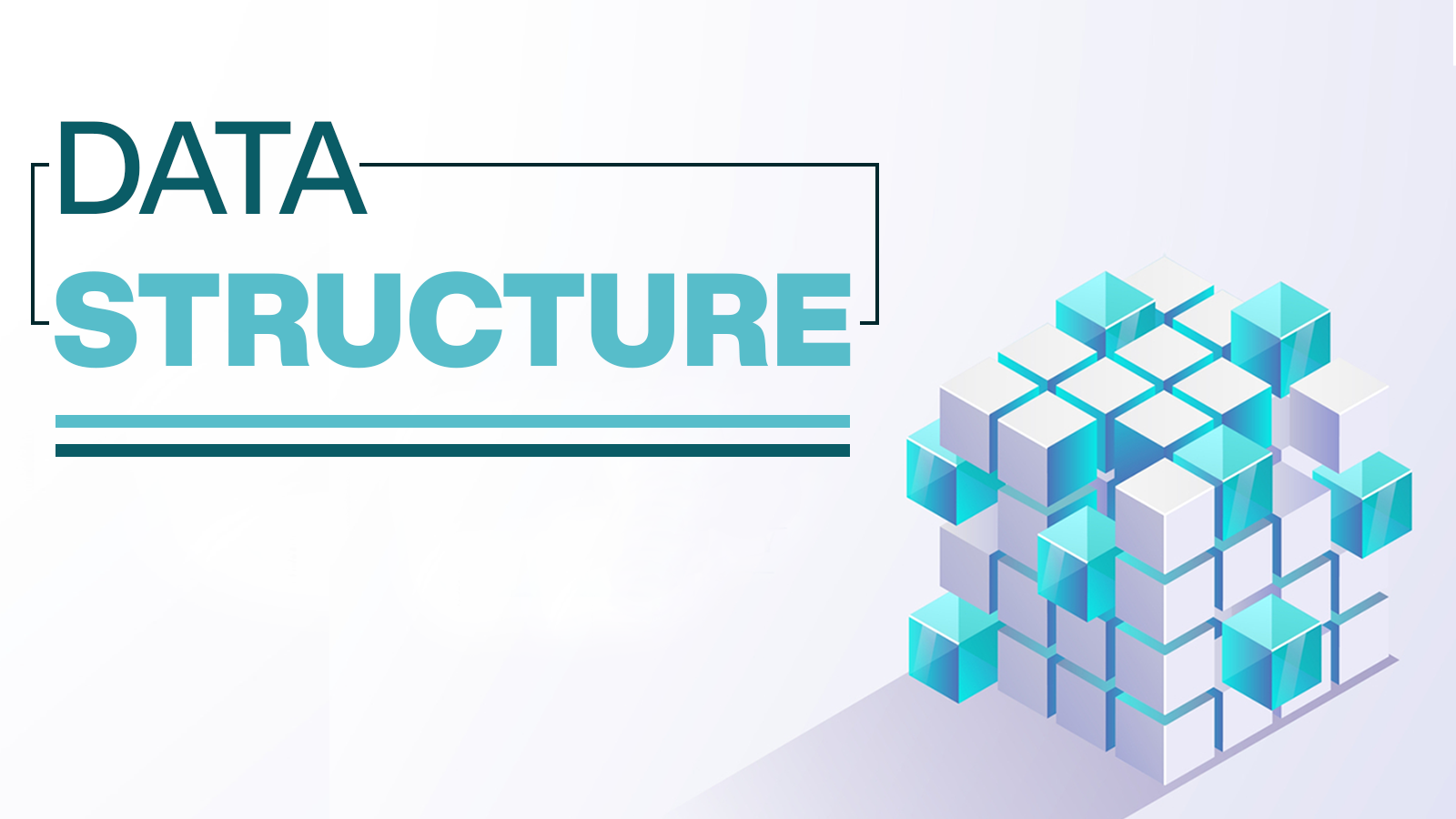
Data structure



Student: issam awamleh

Student ID: 21110241

Course: data structure

Doctor: Abdullah amaren

Part 1:

Specification:

The implement of a min\_priority queue (PQ queue) using a data structure such as linked list is a pointer point to the head of the queue, also with number of the nodes that represent the elements of that queue serially.

Definition:

The PQ queue is a is dynamical linked list that allow:

1\_inserting

2\_removing

3\_also can restoring elements

it's made to be sorting based on the priority of every node of them from the lowest to the highest priority, putting in mind the operation made based on the order of the nodes.

Application:

1\_ Dijkstra: shortest path that use the PQ queue when stored the node in the graph to restored the minimum efficiency way(mathematical).

2\_prim's algorithm: the PQ queue to find the minimum spanning tree of a weightiness of the undirected graph(logical).

3\_load balancing: the PQ using in load balancing of the traffic and directed to the lowest server dealing with the traffic(security).

4\_robotics: the PQ queue implement in the execution of the tasks based on specific manner(robotics).

5\_medical system: the PQ queue could be applying in the system of emergency department in the hospital which prioritize the cases based on the statues of that case.

Operation:

1\_ inserting function:

This function made to insert any node with its elements include the priority of it. So it will be inserting and the place of the node will the depends on the value of the priority part of the node. The Time complexity of the function is big O(N) because if we go throw all the nodes then insert node with high propriety the add will be in the front of the queue all that happened on the worst case.

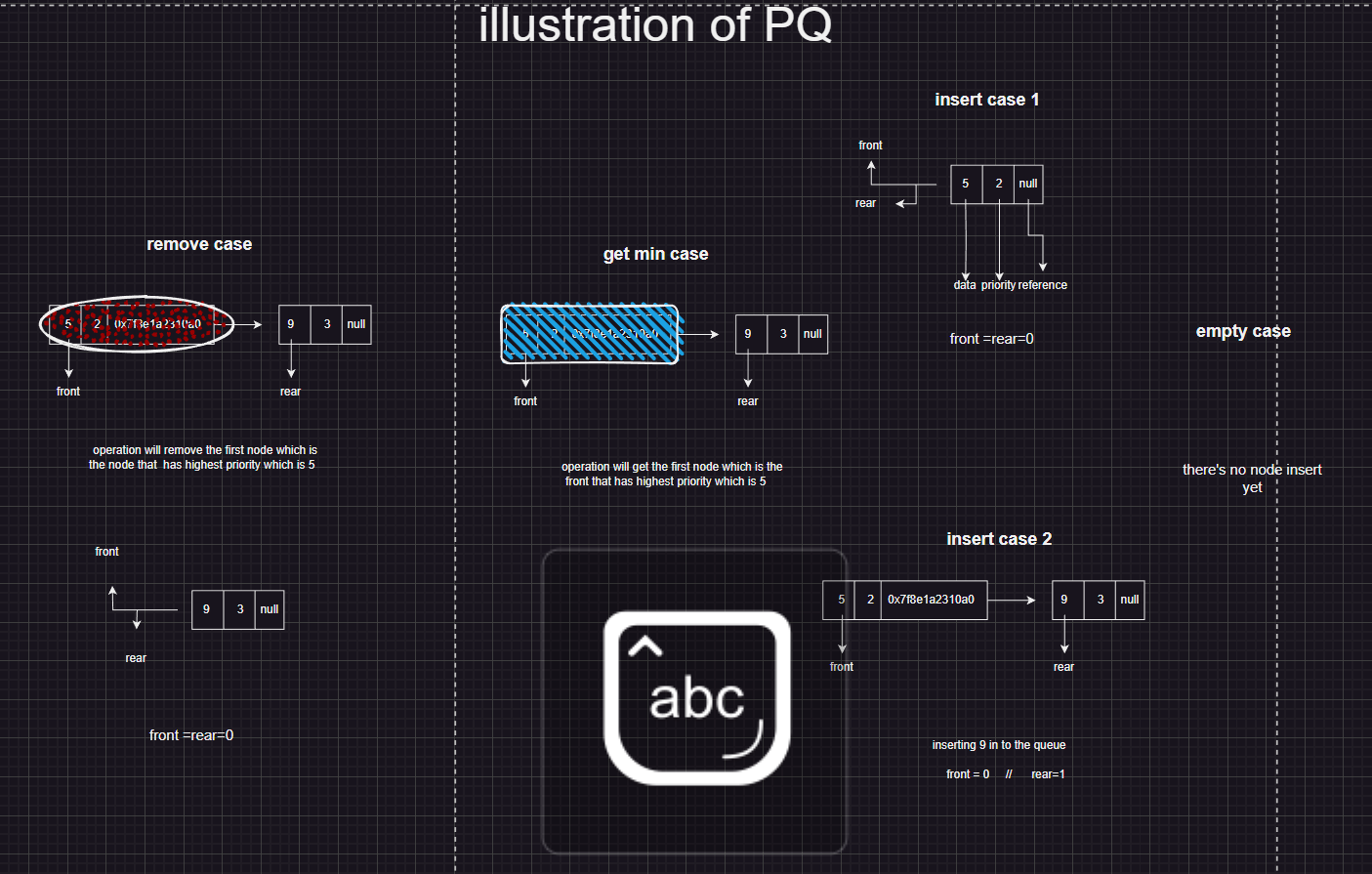
2\_get minimum:

The function basically returns the value of the lowest (value) priority from the priority queue by the comparison operation to return the value of it the time complexity would big O(1) that because the min value would be in the front node.

3\_ remove Min:

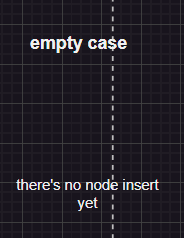
This function to remove node of the queue obviously from the front of the queue that means the time complexity will be big O(1) it's quiet clear because it's in the front.

[The illustration](illustration%20(1).drawio) (use draw.io to open this link to view all the illustration).



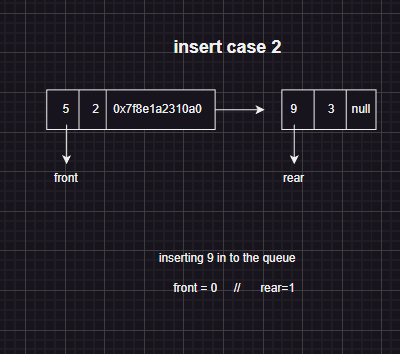
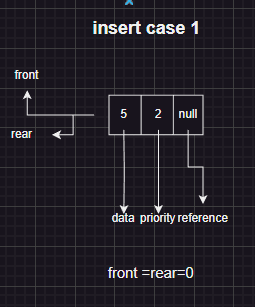
Let's bring it in function after the another:

Empty MP queue:



In this case the first (front, rare) are -1 because we don't have any nodes yet the account of them start with 0 and the

Inserting operation:

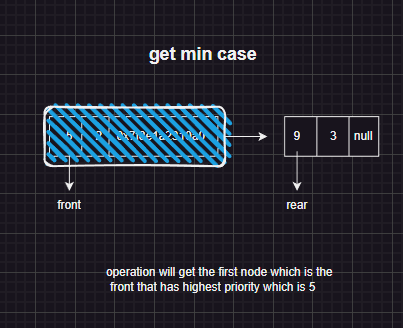
Inserting has 2 case the first is when you add one node and that the only case that the front and rear be equal.

The second when you have nodes already in the MP queue in this case depends on the priority the node will placed putting in consideration tow points:

1\_Rare will be adding one.

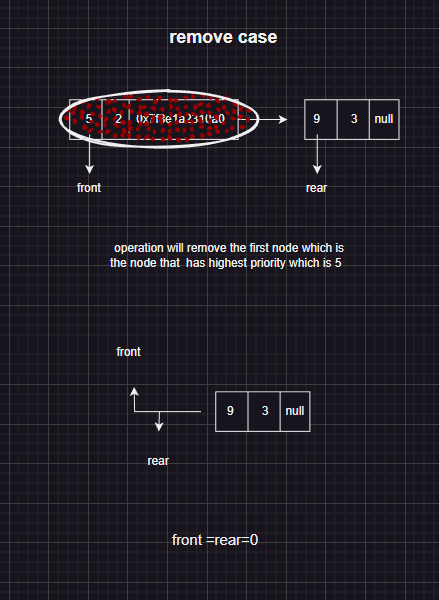
2\_Whatever the place of the new node must give it reference to the previous node to be in the reference and the reference of the next node will be the reference of the new node.

Get operation:



The get function it depends on return the data of the node is the min priority value which is the front node.

Removing operation:

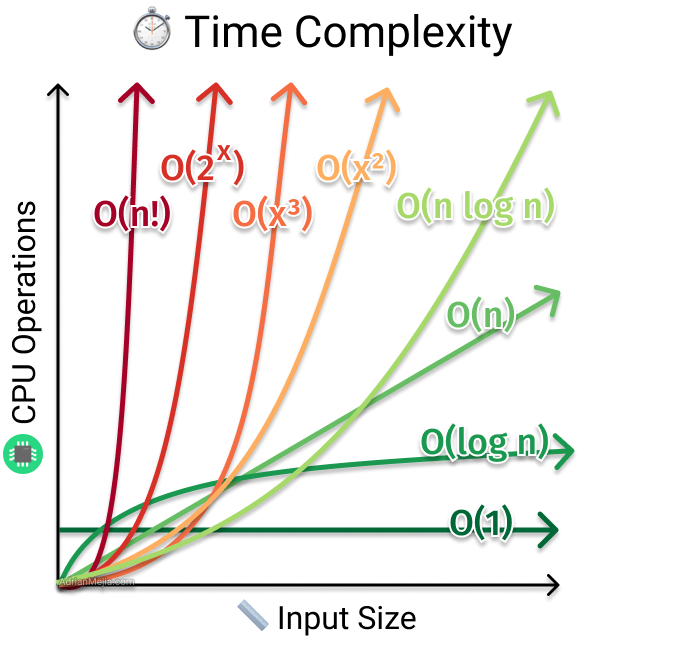


The removing operation is to remove the front (high priority) and just to shift the nodes using reference.

Part 2:

|  |  |  |
| --- | --- | --- |
| 500K | 50K | 5K |
| 259107 | 604 | 12 | SORTED |
| 260114 | 2606 | 16 | RESERVELY  SORTED | SELECTION  SORT |
|  |
| 51 | 9 | 1 | SORTED |
| 70 | 11 | 2 | RESERVELY  SORTED | MERGE  SORT |

Q4: the time complexity of the algorithms:



1\_ selection sort:

Best case: the best case here is to have the array sorted but still have to go throw all the elements and compare the element by element with the whole array to swapping then the time complexity O(n^2)

Worst case: the worst case is to have the array in reverse order and still go throw all the elements and comparing element by element with the all array and the swapping so it's O(n^2).

2\_ merge sort:

Best case: the array is sorted but still the merge process which is divide array and regrouping it again so the time complexity is O (n log n).

Worst case: the array is reversed order and go throw the process of the merging so it's O (n log n).

Q5:

1\_sorting

\_ Selection Sort: Even with perfectly sorted data, selection sort still has an O(n2) time complexity. This is due to the fact that each pass must compare and swap items even while swaps are not necessary.

\_merge: On sorted data, the merge sort has an optimal time complexity of O (n log n). The method can skip the merging phase and merely conduct the merge operation once because the array is already sorted. Compared to selection sort, this gives a performance that is more effective.

2\_ reverse sorting:

\_selection reverse: In the worst-case scenario for data that has been reverse-sorted, selection sort has an O(n^2) time complexity. This is due to the algorithm's requirement for several iterations, which involves switching items at each stage in order to advance the larger components closer to the array's end.

\_merge reverse: On reverse-sorted data, merge sort still has an O (n log n) time complexity. The algorithm effectively overcomes the reverse order and effectively sorts the elements by recursively splitting the array and then merging the subarrays.

Q6:

1\_sorted data:

\_selection sort: the time complexity is O(n^2) that because we have nested for loop and in the best case is sorted array but still go throw the whole process.

Additions points:

\_In the worst situation, selection sort's theoretical time complexity is O(n^2) (As we mentioned).

\_The input we add which is the array with different sizes like 5k, 50k and 500k, the number of comparisons and swaps carried out by the method also grows in a quadratic as the size of the input array does.

\_As a result, as the size of the array rises, the time required to sort the arrays will also climb dramatically. The length of the selection sort will increase with the size of the array.

\_merge sort: let's get in the points directly:

the time complexity is O (n log n) that's applied in all scenario.

Merge can handle sorting of the array with the different size we provide.

Using a divide-and-conquer strategy, merge sort repeatedly divides the array into smaller subarrays until they are all of size 1, then merges them back together in sorted order. So the temporal complexity is unaffected by the input array's order. Because of this, the temporal complexity is unaffected by the input array's order.

The size of the input array will affect how long it takes to sort the arrays in a logarithmic way.

Conclusion: in small array the two could do the same with extremely unnoticeable difference between them. In the big array the efficiency will be for the merge.

The efficiency disparity is more pronounced as the array size grows. Due to the quadratic time complexity of the selection sort, there are significantly more comparisons and swaps, which significantly increases sorting times for big arrays. On the other hand, merge sort is more effective for larger inputs since its time complexity is more consistent regardless of the size of the array.

Q7:

Space complexity: \_

One of the faces of the comparison is the space complexity the merge sort can give O(n) but in this point the selection for the space is better because it's O (1).

It's crucial to keep in mind, though, that in some cases, space complexity may not be as important as time complexity. The effect of space complexity on algorithm efficiency may differ depending on the amount of memory that is available and the quantity of the input data. Consequently, it's critical to take into account both time complexity and space complexity, as well as other aspects, to choose the best method for a given issue or system.

\_input size: it's a one of the factor that can effect the performance of the algorithm.

\_stability: During the sorting process, stability refers to the maintenance of the relative order of elements with equal keys. Maintaining stability may not be a problem in some situations, but it may be in others.

Q8:

Dijkstra's Algorithm:

step 1:

Initialize the algorithm by setting the distance of the source vertex to 0 and the distance of all other vertices to infinity.

Step 2:

Select the vertex with the smallest distance (original source vertex) and mark it as visited.

Step 3:

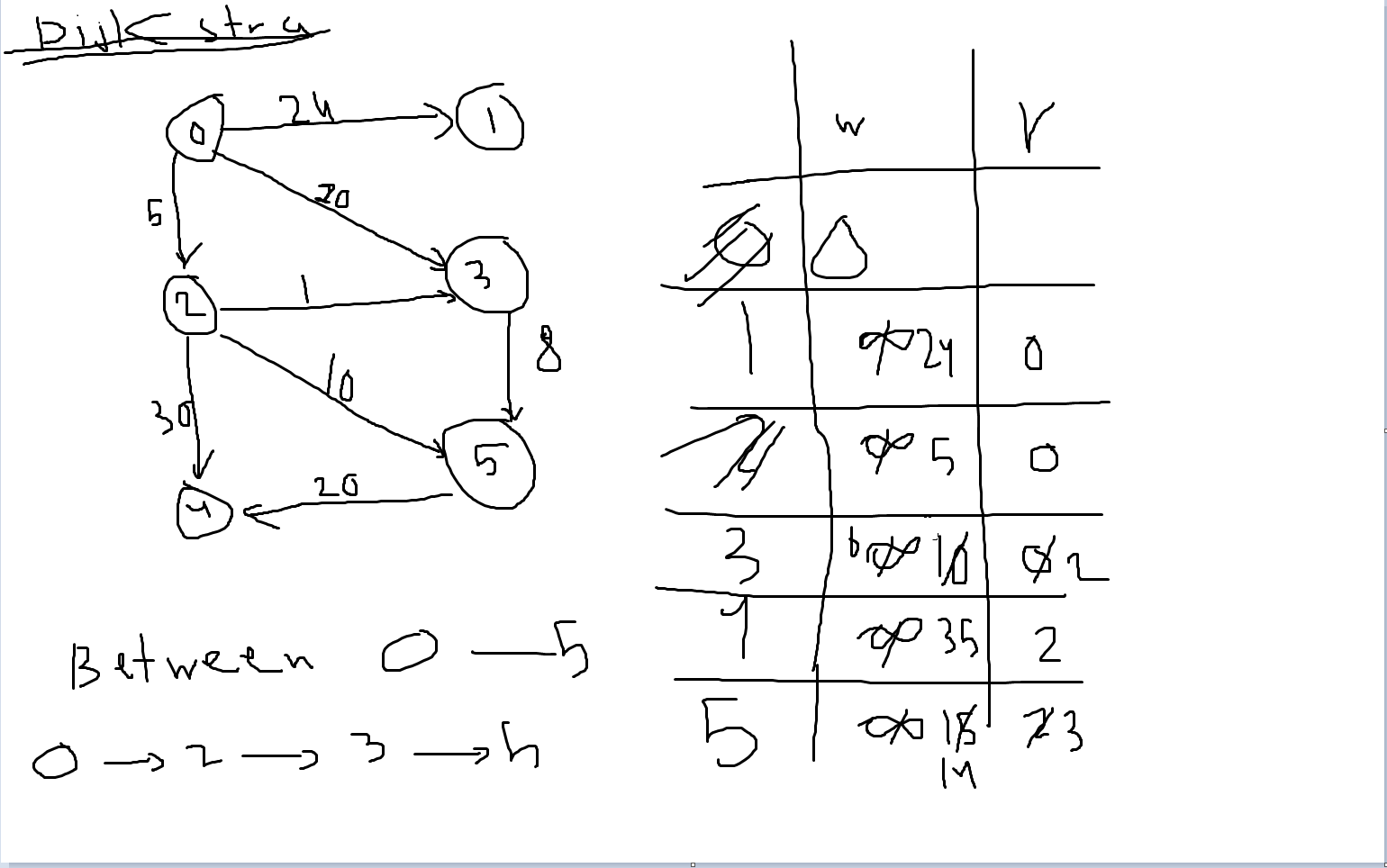
For selected vertices, update the distances of adjacent vertices if a shorter path is found. Compute the new distance as the sum of the distances of the current vertices and the weights of the edges connecting them.

Step 4:

Repeat steps 2 and 3 until all vertices have been visited or all reachable vertices have reached the final shortest distance.

Step 5:

The algorithm ends and determines the shortest path from the source vertex to each vertex.



Bellman-Ford Algorithm:

step 1:

Initialize the algorithm by setting the distance of the source vertex to 0 and the distance of all other vertices to infinity.

Step 2:

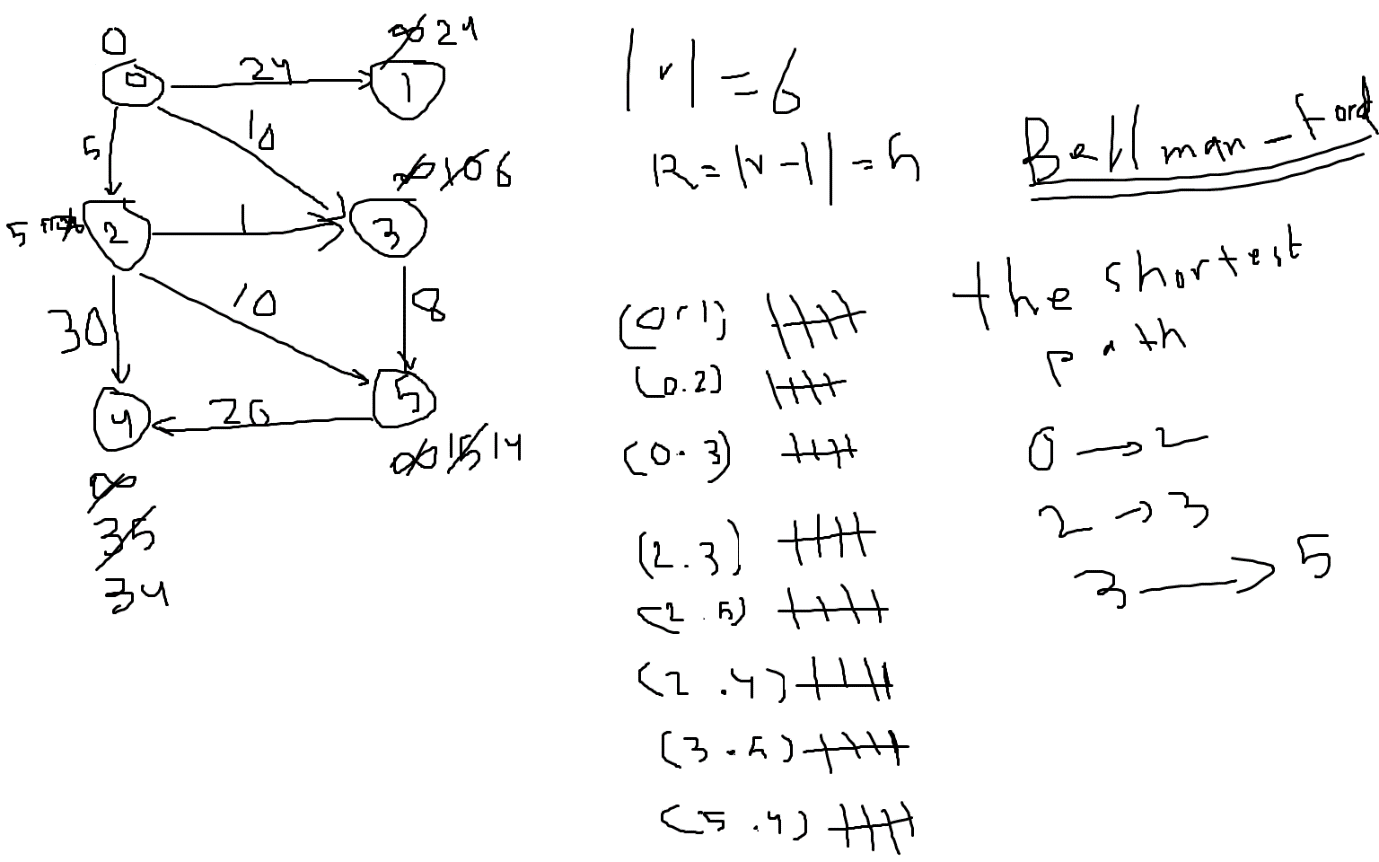
Once a short path is found, iteratively relaxes all edges and updates the vertex spacing. Repeat this step |V|-1 times. where |V| is the number of vertices in the graph.

Step 3:

Look for negative cycles. If a shorter path is found during the (|V|-1) the iteration, the figure has a negative cycle and the algorithm cannot provide the shortest path.

Step 4:

The algorithm ends and determines the shortest path from the source vertex to each vertex.



Part 3:

Q11: pseudo code

1\_Create the function kind Boolean to return the true or false.

2\_Create the variable.

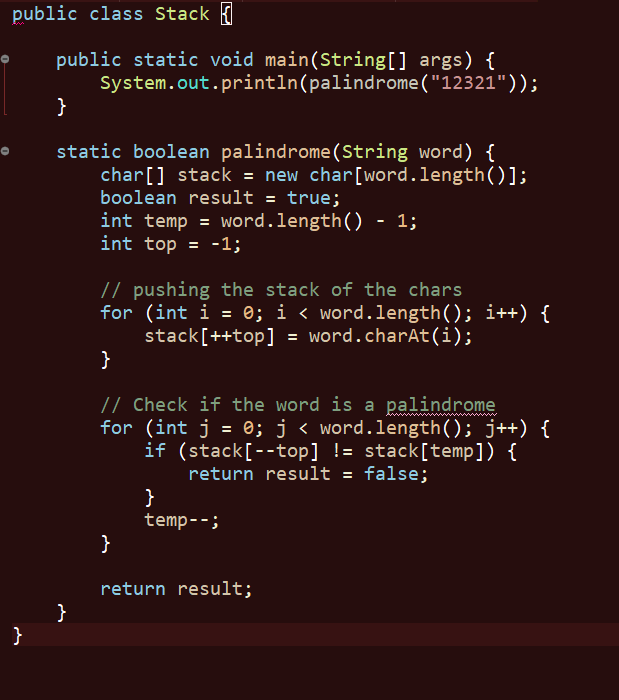
3\_pushing the stack from the strings using for loop the bounders is the word length.

4\_ start with loop to the word length.

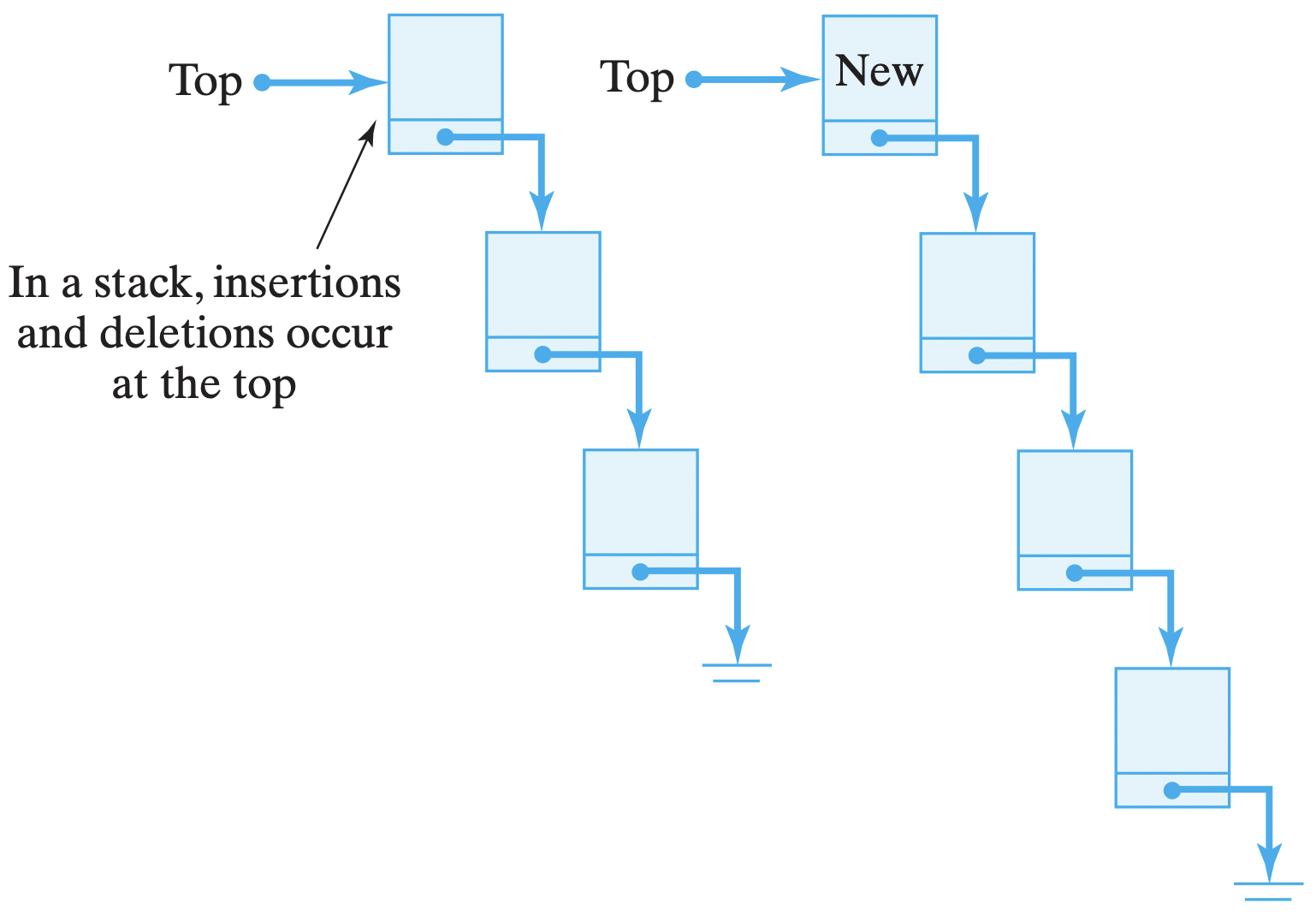
5\_Check if the first pointer which in the first array and the last index that second pointer point to is not equal if it's not equal (true if statement) then the result will be false.

6\_If are equal the loop continue and the second pointer will be subtracted and the first will added already from the loop.

7\_It keeps the two pointers get closer until they get to the midpoint when they reach then the function and the string will be true.



Q12:



The abstract data structure is the description of data structure determine the behavior and function or operation of that data structure.

Stack: is a data structure depends on a concept of (LIFO) last in first out which means the last element will be in stack the first one will be removed.

There's a something called top which is variable that point to the last element that add.

There's some operation could operate on the stack which:

1\_ push: this operation is for inserting elements in the stack.

2\_ pop: this operation for removing the last elements add. Subtract one from the top and return the top

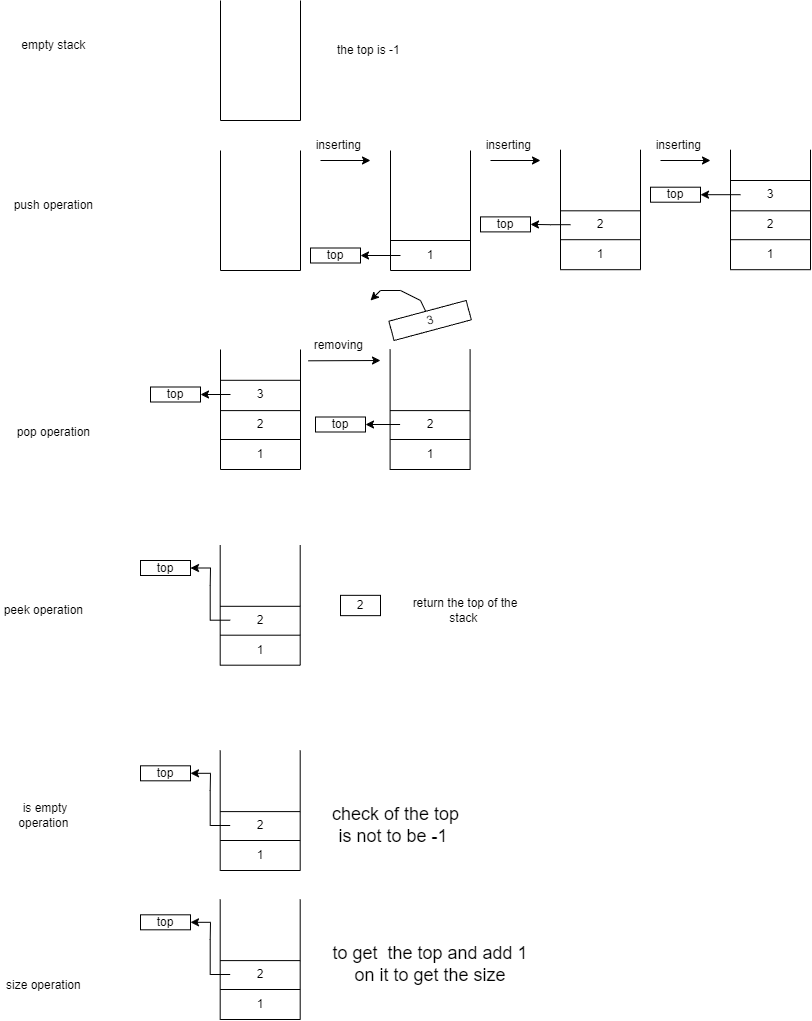
3\_peek: return the top without doing any changes.

4\_is empty: is a function to check if the stack is an empty or not.

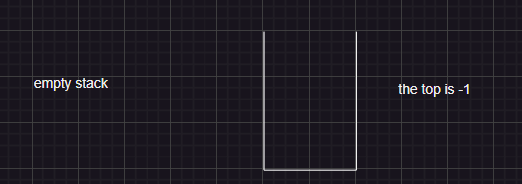
5\_size: to return

Illustration:

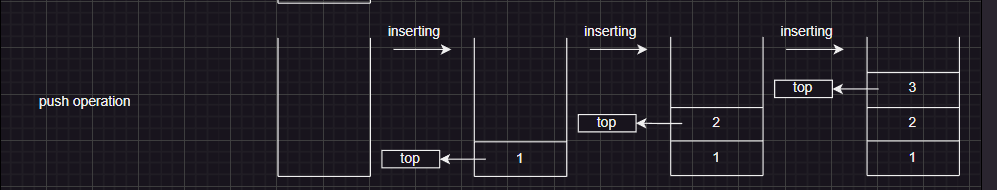
Scenario: CJ and bob are tennis player and they share the stack tennis balls together so let's illustrate what operation could we do in that stack



Empty stack: When the tow uses the all balls and leave the stack empty

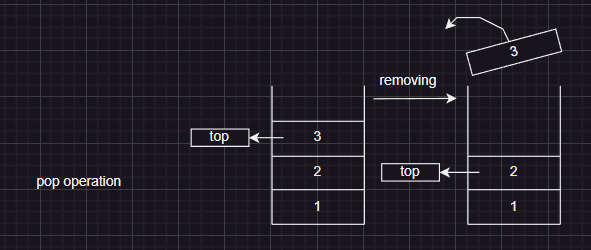


Push operation: The stack when they add some balls in it.

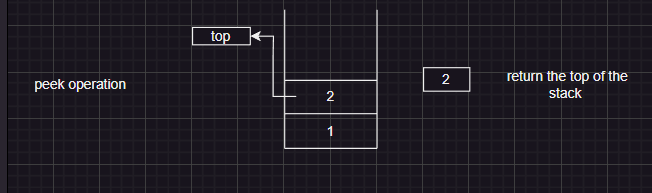


We notice that the top is increasing when we're adding more elements.

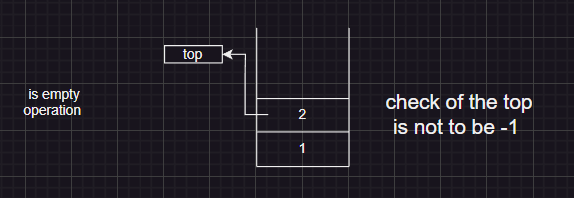
Pop operation: the stack when they get out some balls of the stack to play with.



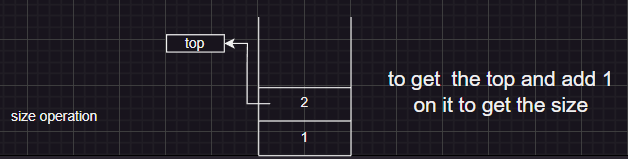
Peak operation:



Is empty operation:



Size counting operation:



Q13:

The call stack determines the mechanism of calling function flow and keep tracking the code's procedure. So it's guarantee calling the functions, how and when can the function to resume executing the functions after return from other functions also to keep the flow in correct track.

For example, it's explanation of a code and that call stack illustration, let's go step by step in it:

1\_ the first function is called for sure will be main function there's declaration and initialization of the array called 'arr' that has numbers from 1\_5.

2\_ then called the function 1 which has perimeter which are 'i' and the array the value of i is 4.

3\_ the function checks if the value of I is less than 0 if is true the function2 will be called that contain print statement.

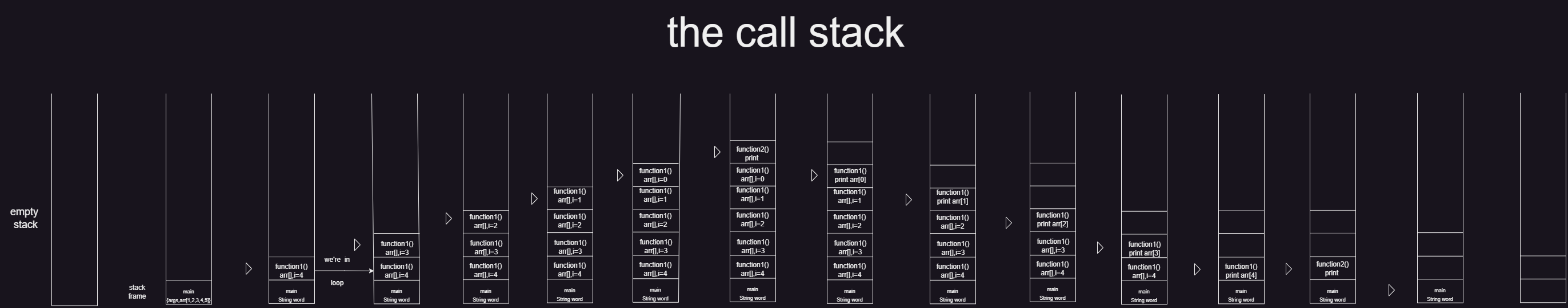
4\_ if the if statement is false we will have recursive function and a print statement to print the element of the index i.

5\_ keep calling the function1 with the same perimeter with subtract (i -1) until we get the if statement true.

6\_ and excite the function2 then back for the recursive and print all the elements of the array.

7\_ and back to the main function to call the fucntion2 to print the statement which is (let's start ^\_^).

Illustration:



Q14:

|  |  |  |  |
| --- | --- | --- | --- |
| Array sorted | Array unsorted | linked List |  |
| O(log n) | O(n) | O(n) | Search |
| O(n) | O(n) | O(n) | Insert |
| O(n) | O(n) | O(n) | Remove |

First scenario:

Explanation: the best choice for the spell checker is the sorted array and that because the data which we deal with must be sorted (in alpha order).

\_ the liked list will take extra space and we could make an array in size of the longest in the dictionary.

Time complicity: in sorted array we could have advantage of the binary search algorithm that has O (log n) and in the worst case O(n) in liner search and that equal to any other data structure which couldn’t be advantage.

Second scenario:

Explanation: The PQ suitable for the linked list which meet the requirements of the case which are:

1\_Dynamics which means free to add and remove with no bounders (not fixed size).

2\_The flexibility of using differ data type and that what we need in priority queue.

The time complexity: the time of the inserting is O(n) because we should go through the all nodes and compare it to choose the suitable place for the inserting and about the removing it's O (1) because always the deleting will be in our PQ is the front.

\*note\*: But in comparing with the array they equal in this point in the inserting and the removing function.

Q15:

To calculate the execution time of the fun1 function, analyze the recursive calls and their frequency of execution. Function fun1 has three recursive calls with indices Index-1, index-2, and Index-4. Each recursive call decrements the index value. This means that the recursion continues until the index reaches the base case of index <= 0. Denote the execution time of fun1 as T(n).

n represents the index value. The runtime can be split into two parts. time spent in recursive calls Time spent on comparison and return statements For recursive calls: The recursive call at index Index-1 is executed T(n-1) times. The recursive call at index Index-2 is executed T(n-2) times.

The recursive call at index Index-4 is executed T(n-4) times.

For comparison and return statements: These operations take constant time and are independent of the value of n. So the execution time can be expressed recursively as:

T(n) = T(n-1) + T(n-2) + T(n-4) + c

Now let's analyze the complexity of the function. From the recursive calls, we can see that the number of recursive calls at each level roughly follows the Fibonacci sequence. 1, 2, 4, 7, 12, 20, 33,...

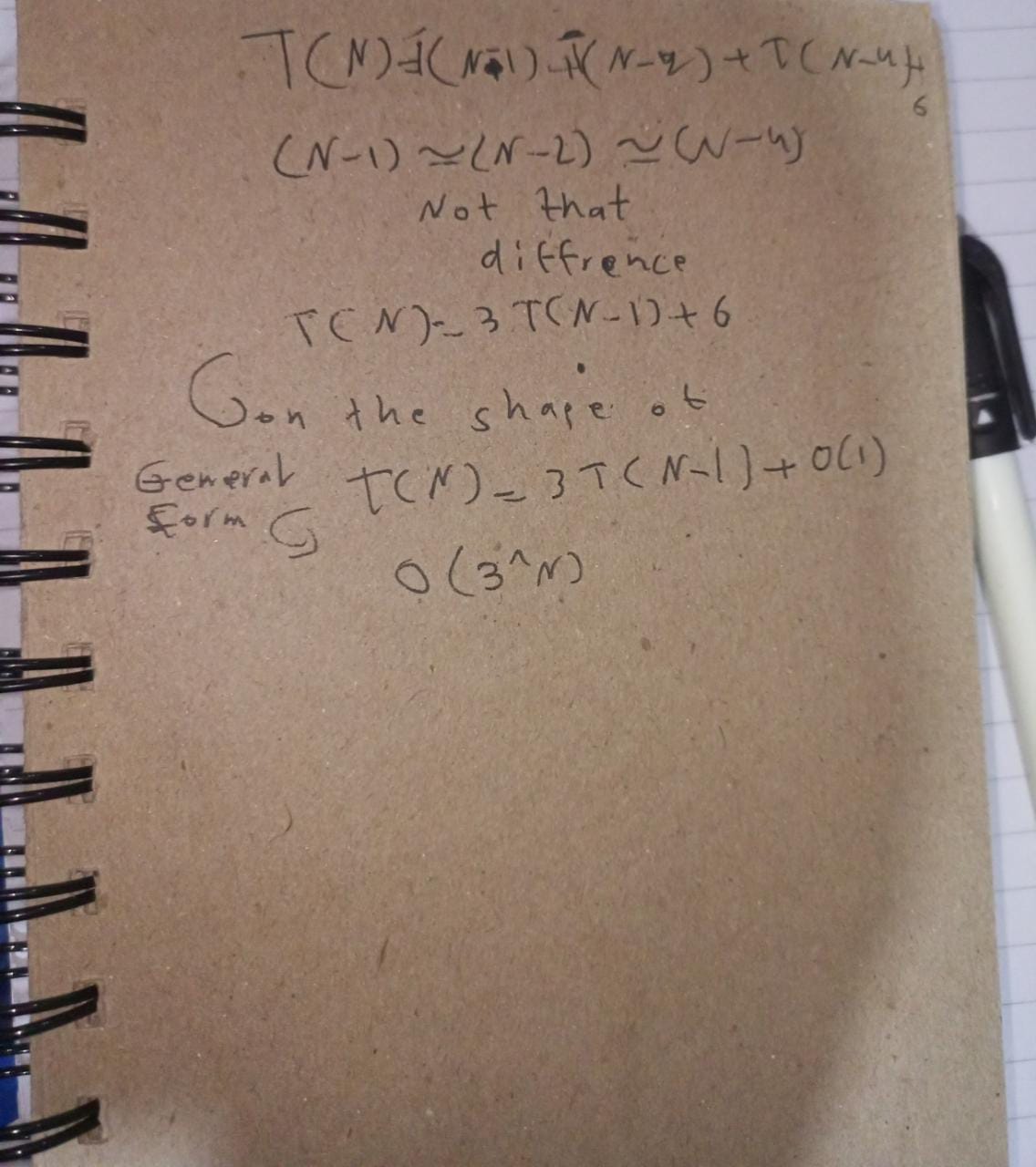
The recursive expression for execution time is: T(n) = T(n-1) + T(n-2) + T(n-4) + c From the recursive call pattern, we can see that T(n) is approximately equal to T(n-1) + T(n-2) + T(n-3). Aside from the constant term, this recurrence formula looks a lot like the Fibonacci sequence.

Expanding the recurring relationship gives:

T(n) = T(n-1) + T(n-2) + T(n-4) = [T(n-2) + T(n-3) + T(n-5)] + [T(n-3) + T(n-4) + T(n-6)] + [T( n-5) + T(n-6) + T(n-8)] = 2 \* [T(n-2) + T(n-3) + T(n-4)] + [T(n-5) + T(n-6) + T(n-8)]

You can see that each step triples the number of recursive calls, giving a factor of 3^n.

Therefore, based on this analysis, we can conclude that the execution time of function fun1 is O(3^n). The function exhibits exponential growth, with run time increasing rapidly as the input size n increases.



Q16:

Encapsulation: is a way to restrict the accessing to the data or methods and that to keep it safe and not available for all.

In the data structure the mechanism of each data structure restrict the accessibility of it and every data in it the only way to access by the objects by calling it in the main function.

The advantages:

\_hiding the data: the user will won't know how's the method work but they just know the value being passed and initialized.

\*note\*: so that exactly what is the data structure ADT which is mathematical model focusing on the behavior of the data structure that has been built not the way of that function work.

\_more flexibility: by using of the getter and setter you can return any value and set any value and easily change it also with the getter is the way of return the value.

\_easy to reuse: the adaption is easily mechanism by using encapsulation to the new requirements of the data structure.

Q17:

Let's break down the concept let's start with the ADT which is the:

Abstract: which means interest of the general functionality of the operation in data structure which consider high-level description of the functionality of that data structure.

Data type: specify the data structure that will deal with the data that has special behavior and certain functions.

Imperative: is the concept of the commands that effect on the sequence and the state of the program so it's consider is determination what's is the priority to be executed according to imperative programing paradigm so controlling the flow of the program

The explanation of the statement: the two concept is so close to each other but can't consider them as a synonym yes, the imperative ADT can serve a basic of the (OOP), the (OOP) depends on the class that encapsulated and can be manipulated by the shape of the object here we use the ADT that control the shape of behavior of that classes.

And every class has behavior and its own operation with own values so the behavior of the objects can be defining by the imperative constructer.

Q18:

The dependent and independent implementation of the array, linked list, queue and stack.

The array is a data structure that made to store the data in certain sequence that has fixed size with the same data type.

The liked list is a data structure is made of nodes in a sequence also linked by the reference of each one of these nodes, it has dynamic size and could contain different data type in the data part of the node.

So the implementation of the both has factors effect on them which are:

1\_Memory

Performance\_2

And the availability of the operation on them \_3

So the characteristics of the two is independent of the implementation.

The queue is a data structure depends on the concept of the first in first out and can be implemented in array and linked list so the queue depends on this two form to be implemented it can have operation such as enqueuer(inserting) and dequeue(removing).

The stack is a data structure follow the concept of the first in last out it's the same tool of implementation of the queue which are array and liked list to be implemented. The operation on it are pop(removing) and push(inserting).

The both queue and stack while implementation could effect on them characteristics and the performance. The concepts that the both follow (FIFO, FILO) is still dependent of their implementation:

The benefits:

1\_ the developer when they use independent implementation that and feel free to choice the proper data structure that meets with the requirements and to optimize the factors which improve the performance such as the memory usage, time complexity and cache efficiency.

2\_ you're free to choice depends on the requirements of the case that you're working on the best data structure to implement liked list or array to do it.

3\_ the independent implementation is more adopting with the future improving and the reason is the its strict linked with specific previsions that give an advantage.

REFRENCES:

1\_*Applications of priority queue* (2023) *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/applications-priority-queue/ (Accessed: 18 June 2023).

2\_Libretexts (2021) *16.3: The stack adt*, *Engineering LibreTexts*. Available at: https://eng.libretexts.org/Bookshelves/Computer\_Science/Programming\_Languages/Java\_Java\_Java\_-\_Object-Oriented\_Programming\_(Morelli\_and\_Walde)/16%3A\_Data\_Structures-\_Lists\_Stacks\_and\_Queues/16.03%3A\_Section\_3- (Accessed: 18 June 2023).

3\_*Encapsulation - definition & overview* (no date) *Sumo Logic*. Available at: https://www.sumologic.com/glossary/encapsulation/ (Accessed: 18 June 2023).

4\_https://www.liquisearch.com/abstract\_data\_type/defining\_an\_abstract\_data\_type\_adt/imperative\_abstract\_data\_type\_definitions

5\_R, V. (2023) *What is data structure: Need, Types & Classification*, *Great Learning Blog: Free Resources what Matters to shape your Career!* Available at: https://www.mygreatlearning.com/blog/data-structure-tutorial-for-beginners/ (Accessed: 18 June 2023).

6\_Sahni, S. (no date) *Data Structures, algorithms, and applications in C++*, *Google Books*. Available at: https://books.google.com/books/about/Data\_Structures\_Algorithms\_and\_Applicati.html?id=LOtqAAAACAAJ (Accessed: 18 June 2023).

7\_(No date) *Introduction to algorithms by Thomas H. Cormen, Charles E. Leiserson ...* Available at: https://archive.org/details/introduction-to-algorithms-by-thomas-h.-cormen-charles-e.-leiserson-ronald.pdf (Accessed: 18 June 2023).

8\_ Sahni, S. (no date) *Data Structures, algorithms, and applications in C++*, *Google Books*. Available at: https://books.google.com/books/about/Data\_Structures\_Algorithms\_and\_Applicati.html?id=LOtqAAAACAAJ (Accessed: 18 June 2023).