****

**Data Structures & Algorithms**

**40201100**

**T/618/7430**

**Section (7)**

**Submitted to**

Eng Ashraf Smadi.

**Submitted by**

Hamza AL-Risheq.

**spring 2022 – 2023**

Table of Contents

[Part 1: 3](#_Toc137718097)

[Q1) 3](#_Toc137718098)

[Q2) 4](#_Toc137718099)

[Q3) 5](#_Toc137718100)

[Part 2: 7](#_Toc137718101)

[Q4) 7](#_Toc137718102)

[Q5) 7](#_Toc137718103)

[Q6) 8](#_Toc137718104)

[Q7) 8](#_Toc137718105)

[Q8) 9](#_Toc137718106)

[Part 3: 11](#_Toc137718107)

[Q11) 11](#_Toc137718108)

[Q12) 12](#_Toc137718109)

[Q13) 15](#_Toc137718110)

[Part 4: 16](#_Toc137718111)

[Q14) 16](#_Toc137718112)

[Q15) 17](#_Toc137718113)

[Q16) 17](#_Toc137718114)

[Q17) 18](#_Toc137718115)

[Q18) 19](#_Toc137718116)

[References: 20](#_Toc137718117)

# Part 1:

## Q1)

Definition of the minimum priority queue: To fully understand the definition of minimum priority queue, we must first comprehend the queue.

A queue is an abstract data type (ADT) in which the first node in the queue is the first to be removed, resulting in first first out (FIFO).

The minimum queue priority queue works by assigning a priority to each node and sorting all nodes in the queue based on their priority number. The node with the lowest priority number will be the first in the queue, and so on until all nodes are sorted by priority number.

A priority queue can be implemented as an array, linked list, heap, or binary search tree, among other things. Because each method has advantages and downsides, the optimum option will be determined by your application's individual demands.

There are many applications can use the minimum priority queue:

* Finding the shortest paths is one of the most common uses of minimum priority queue, and there are numerous shortest paths algorithms that do so, including Dijkstra's algorithm and A\* search. Dijkstra's algorithm is a graph search technique that discovers the shortest path in a weighted network with non-negative edge weights between a source vertex and all other vertices. We use a minimum priority queue (or min-heap) in the method to efficiently keep track of the least tentative distance from the source vertex to each vertex. The min-heap allows us to always choose the vertex with the least tentative distance from the source vertex, which aids in optimum network exploration. The algorithm operates by setting the distance to the source vertex to 0 and the distance to all other vertices to infinity, then repeatedly extracting the vertex with the smallest tentative distance from the source vertex from the priority queue, updating the distances of adjacent vertices, and adding them to the priority queue. By tracing back, the path from the destination vertex to the source vertex using the parent nodes that were recorded during the method, the shortest path may be found.
* We can also implement a Binary-tree using the minimal priority queue. Therefore, using a minimal priority queue to create a binary tree for data will make it easier to navigate the tree or search for data. Because the data will be stored in the B-tree using a minimum priority queue, making it simple to identify the following child. Additionally, it will make updating the B-tree faster and more effective (updating the tree entails deleting the child, introducing child, and changing child's value).
* In the operating system, the minimum priority queue is a valuable tool for organizing operations and allocating resources. It can be used to schedule tasks according to priority and to allocate memory resources according to the relevance of operations and the amount of memory required. For example, you can use a minimum priority queue in a multitasking operating system to prioritize higher priority tasks and thereby control the order in which tasks are executed. Similarly, queues can be used in a memory management system to prioritize the allocation of memory resources to processes based on process priority or memory needs.
* By altering the timing of traffic signals, traffic signal optimization is a technique used to enhance traffic flow and lessen congestion at crossings. How long the red, green, and yellow lights are on display to regulate traffic flow is determined by the timing of traffic signals. At intersections, sensors are deployed to measure the volume of traffic coming from different directions in order to time traffic signals as efficiently as possible. Using a minimum priority queue and this data, real-time signal timing modifications are made. The traffic signal controller adjusts the signal timings to give the intersection with the most traffic the highest priority in the queue. This ensures that the intersection with the highest traffic volume gets a longer green signal time, while other intersections get shorter or no green signal time. Traffic engineers can quickly modify signal timings based on real-time data by employing a minimum priority queue in traffic signal optimization, which improves traffic flow and lessens congestion at crossings.

Main operation in minimum priority queue:

* Insert: Relying on the node's priority, data is inserted into a minimum priority queue. We have two options to insert a node into a minimum priority queue. Added first to an empty queue. When inserting a second node into a non-empty queue, we must first examine the entire queue to see if any nodes have a higher priority number than the node we want to insert. If so, we must add our node before the node with the higher priority number. However, if there are no other nodes in the queue with a higher priority number than ours, we will insert our node at the end.
* getMin: this operation is for returning the data in the first node in our queue (the node that has the least number of priorities) so the minimum priority node will be in the front in the queue.
* removeMin: this operation is for deleting and first node in the queue which is the front (the node that has the least number of priorities.). And we will shift the front reference to the next node, and it will be the first node in the queue (and it will has the has the least number of priorities).

Big – O for each operation:

* For inserting operation, it will big – O (n).
* For getMin operation, it will big – O (1).
* For removeMin operation, it will big – O (1).

## Q2)

For empty minimum priority queue, the front and rear of the queue will be pointing on null so we will not be able to use the getMin operation because it will give us an exception message that the queue is empty.

If we insert one element: If we insert the following: then we use remove operation one time:

Insert (55,9); Insert (55, 9); removeMin();

Insert (52, -1);

front

front

Insert (54, 5);

front

|  |  |
| --- | --- |
| Priority | data |
| 9 | 55 |

Insert (53, 0);

|  |  |
| --- | --- |
| Priority | data |
| 0 | 53 |

|  |  |
| --- | --- |
| Priority | data |
| -1 | 52 |

|  |  |
| --- | --- |
| Priority | data |
| 5 | 54 |

|  |  |
| --- | --- |
| priority | data |
| 0 | 53 |

rear

|  |  |
| --- | --- |
| Priority | data |
| 9 | 55 |

|  |  |
| --- | --- |
| priority | data |
| 5 | 54 |

rear

|  |  |
| --- | --- |
| priority | data |
| 9 | 55 |

rear

Then using remove operation one more time: In the end we want to print the node in the removeMin(); front of the queue:

|  |  |
| --- | --- |
| Priority | data |
| 5 | 54 |

System.out.Print(getMin());

front

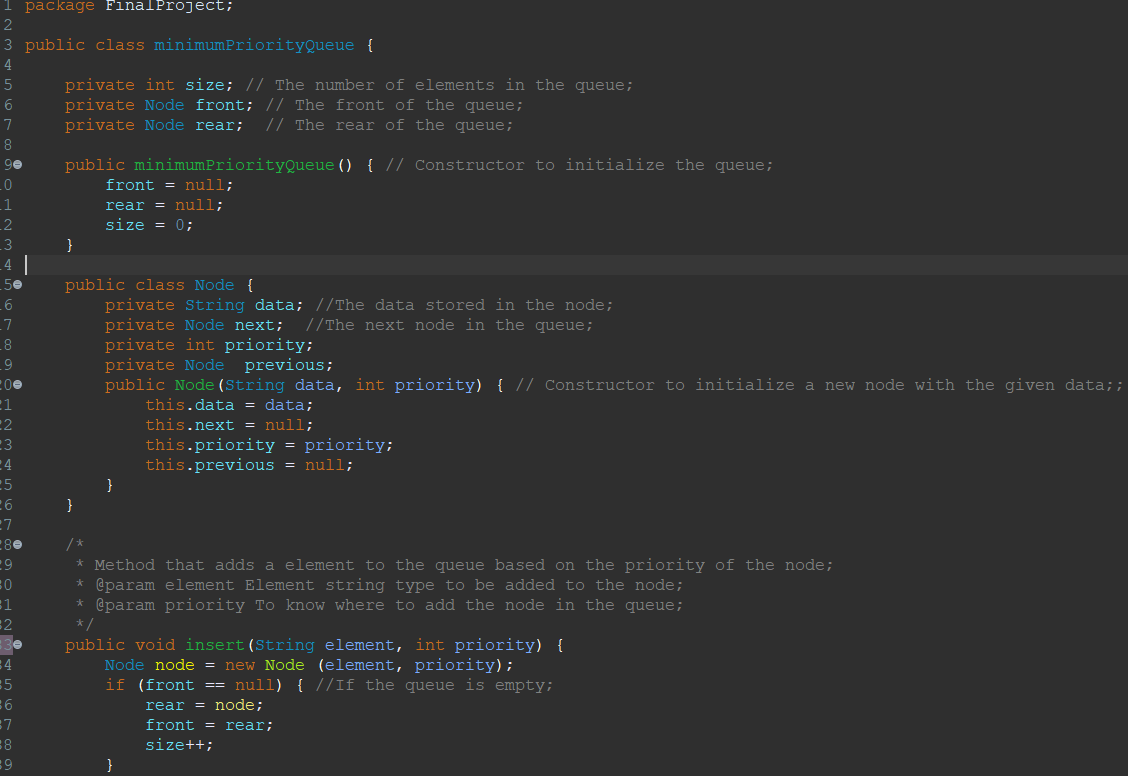
So, we will print the data in the first node:

Data in the first node = 53

|  |  |
| --- | --- |
| Priority | data |
| 9 | 55 |

rear

Q3)

Minimum priority queue implementation:

A screen shot of a computer

Description automatically generated with medium confidenceA screen shot of a computer program

Description automatically generated with low confidence

# Part 2:

The outcomes of running the codes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 5000 | 50000 | 500000 |
| Selection  sort | Sorted | 14 millis | 672 millis | 65121 millis |
| Reversely  sorted | 21 millis | 774 millis | 74539 millis |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 5000 | 50000 | 500000 |
| Merge  Sort | Sorted | 1 millis | 7 millis | 46 millis |
| Reversely  sorted | 0 millis | 8 millis | 28 millis |

## Q4)

|  |  |  |
| --- | --- | --- |
|  | **Selection Sort** | **Merge Sort** |
| **Best case** | O (n2) | O (n log n) |
| **Worst case** | O (n2) | O (n log n) |

## Q5)

The time and number of the data comparison in both reversely sorted data and sorted data is the same big-O(n2) the difference will be in the data moves. According to the results we got, we filled it in table at the start of part 2, the highest performance of the selections sort algorithm when the data is sorted. So, the data are already in the correct sequence, there is no need to do any swap operations between them as result the big-O of the data move operation will be O (1). On the other hand, when use reversely sorted data we will compare each element then we have to swap the element so we will swap all the elements from the lowest to the highest as result of the data moves operation the big O of data moving in reversely sorted will be big-O (n). So, based on the data we got from the table and calculating the big-O of data comparing and moving in both scenarios sorted and reversely sorted we will mention that the sorted data is faster than the reversely sorted and the different in the speed is noticeable.

Now when we talk about merger algorithms here is the opposite. The data moves are the same when we use the reversely sorted data or sorted data the big-O of data moving in both will be big-O (n) on each level. But to determine the fastest between the reverse and sorted data we will check the time of the data comparing time. After calculating the time of data compare in reversely sorted data it will be the minimum of the sub array so it will be (n/2) on each level because reversely sorted data is the best case. And the time for sorted data in data comparing will be the (n/2) on each level because it’s the best case either. So, after calculating the big O of data comparing in both scenarios and based on the data, we took from the table at the first of part 2 we will find out that the reversely sorted data and sorted data have the same time there, so they are very close in the time the difference between them unnoticeable.

## Q6)

Based on the times we got and added them to the table and the theoretical analysis for each algorithm, we will find that the times we got are accepted based on the theoretical analysis because if we look at two algorithms starting with selection, it goes through all the elements in the array to sort it and then swap it, so the time complexity for it in the best and worst case is O (n2), and when we look at the times we got when we used the selection sort for a 5000 array, a 50000 array, and a 500000 array, the times will increase based on the size of the array, and the time for the 500000 arrays is very big, which is normal because of the time complexity for the selection algorithm. On the other hand, when we look at the times we got for the arrays that used the merge algorithm to sort, we will find that the times are very small because the algorithm uses divide and conquer, so the time complexity for it in the best case and worst case is O (n log n), which will make the times we got acceptable.

## Q7)

There are numerous measures that can be used to compare algorithms. Some of these measures are space complexity, simplicity, and the efficiency of handling huge amounts of data. Space complexity refers to how much memory space the algorithm will consume to solve the problem. For example, some algorithms utilize a constant amount of memory space all the time, while others use changing amounts of memory in each step of the algorithm.

As an example of a constant space memory selection sort. The space of used memory in the selection method is continually unchanging since the utilized space is the same from the beginning to the last step of the algorithm, which is one of the advantages of the selection sort. Merge sort, on the other hand, will take different memory areas dependent on each stage of the algorithm, therefore it is changeable. If we calculate the space complexity of selection algorithm, we will find out that the selection algorithm takes O (1) constant because it did not use more than the array length space but on the other hand merge algorithm takes O (n) because it uses divide and conqueror way to sort the data.

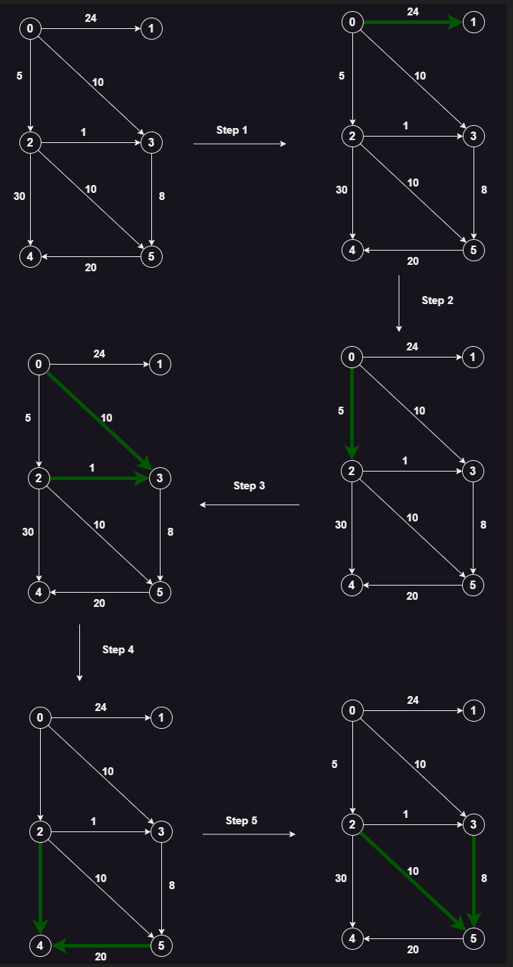
## Q8)

**Dijkestra algorithm**

## 

|  |  |  |
| --- | --- | --- |
| Vertices | Shortest path from 0 to 5 | Previous vertices |
| 0 | 0 |  |
| 1 | ∞ 24 | 0 |
| 2 | ∞ 5 | 0 |
| 3 | ∞ 10 6 | 0 2 |
| 4 | ∞ 35 | 2 |
| 5 | ∞ 15 14 | 2 3 |

**Bellman ford algorithm**



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | ∞ | 24 | 24 | 24 | 24 | 24 |
| 2 | ∞ | 5 | 5 | 5 | 5 | 5 |
| 3 | ∞ | 10 | 6 | 6 | 6 | 6 |
| 4 | ∞ | ∞ | 35 | 35 | 35 | 35 |
| 5 | ∞ | ∞ | 15 | 14 | 14 | 14 |

|  |  |
| --- | --- |
| Vertices | Parent |
| 0 |  |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 2 |
| 4 | 2 |
| 5 | 2 3 |

# Part 3:

## Q11)

Pseudocode for the palindrome code:

1. Create a LinkedListQueue and LinkedStack

2. Read input from user and store in a string variable called "word".

3. Iterate over each character in "word", enqueue into "queue" and push into "stack".

4. While "queue" is not empty, check if the first element in "queue" is equal to the top delement in "stack".

5. If they are equal, dequeue from "queue" and pop from "stack".

6. If they are not equal, return false.

7. If the while loop completes without returning false, return true.

Translating pseudocode to another pseudocode language:

Palindrome:

queue ← LinkedListQueue()

stack ← LinkedStack()

word ← Input from user

for each character c in word:

queue.enqueue(c)

stack.push(c)

while not queue.isEmpty():

if queue.peek() = stack.top():

queue.dequeue()

stack.pop()

else:

return false

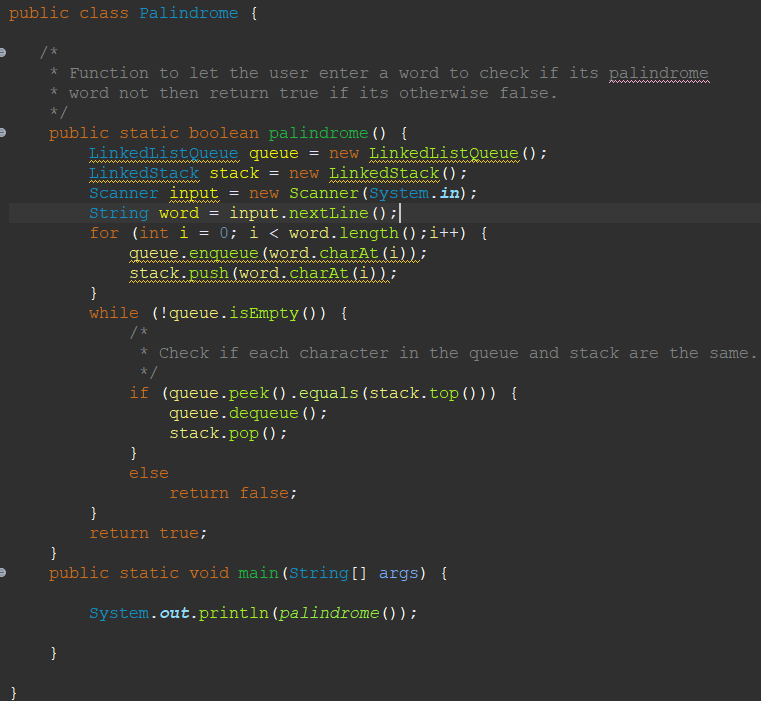
end if

return true

Main:

Print palindrome()

Implementation of the pseudocode:



## Q12)

For implementing the palindrome system, I used two ADT which they are queue and stack:

Queue definition: A queue is an abstract data type with a front and a rear endpoint that stores elements progressively. The first piece added to the queue will be the first to be removed from it. As a result, it employs the FIFO (first in, first out) approach.

Valid operations that can be used in queue:

* + enqueue: use rear to insert an element at the end of the queue. Big-O (1).
  + dequeue: to remove the initial item from the queue. Big-O (1).
  + isEmpty: determines whether or not the queue is empty if its empty it will return true. Big-O (1).
  + front: returns the element at the beginning of the queue. Big-O (1).

Scenario on queue ADT:

Assume we have a word connecter system that takes several characters from the user and connects them to form a word (we will use a linked list queue):

We have the following characters in order: D, a, t, a

So, we will insert all these characters to our empty linked queue:

|  |  |  |
| --- | --- | --- |
| Node Id | Data | Next |
| 67 | D | null |

enqueue (D);

Rear

Front

enqueue (a);

|  |  |  |
| --- | --- | --- |
| Node Id | Data | Next |
| 67 | D | 8942 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 8942 | a | null |

Rear

Rear

Front

|  |  |  |
| --- | --- | --- |
| Node Id | Data | Next |
| 67 | D | 8942 |

enqueue (t);

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 8942 | a | 473 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 473 | t | null |

Front

enqueue (a);

|  |  |  |
| --- | --- | --- |
| Node Id | Data | Next |
| 67 | D | 8942 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 8942 | a | 473 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 473 | t | 673 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 673 | a | null |

Front

Rear

Then we print the front then dequeue it directly:

System.out.print (front ());

dequeue ();

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 8942 | a | 473 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 473 | t | 673 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 673 | a | null |

Rear

Front

System.out.print (front ());

dequeue ();

Front

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 473 | t | 673 |

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 673 | a | null |

Rear

System.out.print (front ());

|  |  |  |
| --- | --- | --- |
| Node ID | Data | Next |
| 673 | a | null |

dequeue ();

Rear

Front

System.out.print (front ());

dequeue ();

now our queue is empty and we can check that by using isEmpty function if return true so its empty: system.out.println(isEmpty());

so overall our output will be:

Data

True

Now we will talk about stack:

Stack definition: A stack is a linear data structure and an abstract data type that employs the FILO (first in, last out) technique. As a result, the first element added to the stack will be the last one to be removed, and the final element added to the stack will be the first to be removed. And the stack only has one pointer, which is the top. As a result, the adding and deleting will be done from the top of the stack.

Valid operations that can be used in queue:

* Push: inserts an element from the top of the stack.
* Pop: the removal of an element from the top.
* Top: returns the value at the very top of the stack.
* isEmpty: checks whether the stack is empty or not; if it is, the function returns true.

Scenario on stack: assuming that you opened 3 websites in order Htu, elearning, and stackoverflow. So, all the websites that you opened will be added to the stack in order.

Pushing the three websites in order to the stack:

push (“Htu”);

Top

push (“elearning”);

push (“stackoverflow”);

stackoverflow

elearning

Htu

Now let’s assume that we want to undo the last website we opened to open the previous website so we will pop the top of the stack:

pop ();

Top

elearning

Htu

Now to get the top of the stack we will use top function:

top ();

## Q13)

Call stack for the main: continue:

pop() and return from

function2 and processing

system.out.print(“lets start ^\_^”);

function1

int arr []

i = -1

pop(), return from function3 and processing

System.out.print(arr[0]);

function1

int arr []

i = 0

function1

int arr []

i = 1

function1

int arr []

i = 2

function1

int arr []

i = 3

function1

int arr[]

i = 4

pop(), return from function3 and processing

System.out.print(arr[1]);

pop() and return

function2 return

system.out.print(“let’s start ^\_^”);

main

int arr []

args

pop(), return from function3 and processing

System.out.print(arr[2]);

pop(), return from function3 and processing

System.out.print(arr[3]);

pop() and return from function2 and processing

System.out.print(arr[4]);

pop() and return from

function2 and processing

System.out.print(“let’s start ^\_^”);

main

int arr []

args

The output of the call stack will be:

lets start ^\_^

1

2

3

4

5

lets start ^\_^

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linked List | Array unsorted | Array sorted |
| Search | Big-O (n) | Big-O (n) | Big-O (log n => binary search)  Big-O (n => linear search) |
| Insert | Big-O (1) | Big-O (1) | Big-O (n) |
| Remove | Big-O (n) | Big-O (n) | Big-O (n) |

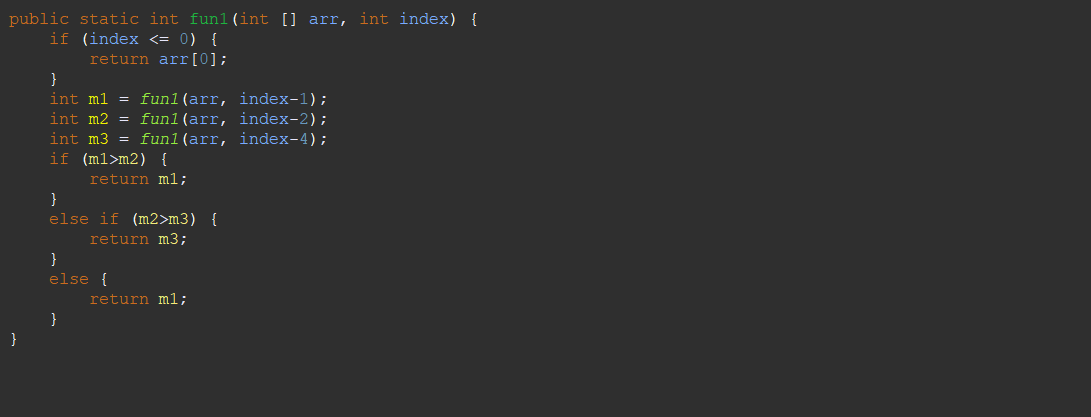
# Part 4:

## Q14)

1. I would use a sorted array because when we want to search for a word in the dictionary, we want something sorted so that we may reduce time by breaking the array into sections. Each section contains words that begin with a certain letter; if we use anything unsorted, we must verify all of the words. Because using sorted array with binary search it will cost me log N on the other hand using unsorted array or linked list it will cost me O (n) and there is big difference between O (log n) and O (n). big-O (n) grows asymptotically faster than big-O (log n). As result the speed for searching inside array sorted faster than searching inside unsorted array or linked list.
2. Because priority queue cares about three main operations inserting, deleting, and returning the front element, we want to choose a data structure that is good in inserting and removing. To implement the priority queue, we will compare the time growth for the inserting and removing operations to choose the best data structures to implement the priority queue. So, the time growth for inserting in linked list and unsorted array is big-O (1) and for removing big-O(n). On the other hand, the time growth for inserting in unsorted array big O (n), and for removing big O (n). As result our options to implement the priority queue are linked list and array sorted.

Since I can insert as many as I like without limitations, linked lists are what I would use for priority queues. I would also use a linked list because for removing a node all is needed to connect the previous node and next node for the node is removed. In the other hand to remove an element from an array you must implement a code that shift all the elements to the left after removing an element.

## Q15)



**T(n) = O(1)**

**T(n) = T(n-2)**

**T(n) = T(n-1)**

**Base case = O(1)**

**T(n) = T(n-4)**

Recurrence equal:

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

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

Now I will use the master theorem to calculate the Big-O of the code:

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

T(n) = 3T(n - 1) + 1

a > 1

so O (aa/b  f (n)) = O (3n)

so the big-O of the code is 3n .

## Q16)

Encapsulation and information hiding are common terms in OOP. Information hiding entails limiting access to certain portions of the encapsulated data, whereas encapsulation refers to the process of combining data and methods that operate on that data into a single unit (integrating data (variables) and code (methods) into a single unit.). Both terms are essential because they are used to implement a lot of codes such as ADTs.

The advantages of implementing ADTs using encapsulation and information hiding:

* Security: We will protect our ADT implementation from alteration when we employ encapsulation and information hiding. Because we'll utilize encapsulation terminology to write our code, all of the variables and methods will be written exactly how we want them to be, and we'll give each method and variable the restrictions we need to prevent unauthorized people from accessing or changing our code. And if someone wants to have access to our ADT, they must create an object from our class type to access what they can from our ADT. So, whatever happens our implementation will not be affected.
* Reusability and flexibility of the code. So, when we implement an ADT using encapsulation it will help to reuse the code in different ways and projects without changing or being able to see the implementation of the ADT. Also, we can create an object from the code and do whatever we like without changing the code.
* Encapsulation and information hiding are a good way to implement the ADT because they provide a clear and consistent user interface for interacting with the ADT. You can hide the complexity and give higher-level, simpler methods by encapsulating the implementation specifics. Also, encapsulation helps to give us a good field to implement our code easily and closer to the real lives because of the objects that we create to have access to the class (class that encapsulation term).
* It helps to debug and correct the bugs because when we implement an ADT using encapsulation terms then we use the ADT by creating object so when we find a bug, or something went wrong we will know the problem in our code not the ADT implementation.

## Q17)

To find out if the ADTs are the basis for object orientation, we have to give the ADTs and object orientation a definition.

ADTs: a concept or representation of a data type the implementation of the data type is irrelevant to the user. The implementation of the functions on a data type is likewise handled by ADT. Here, the user will have ready-to-use preset functions for every data type for any operation. Allowing users to interact with the data structure through a clearly defined interface by offering a mechanism to encapsulate data and processes. Without disclosing how these operations are implemented, the interface lists the various operations, their input parameters, and their return values.

Object orientation is a programming paradigm that arranges software design around objects, which are examples of classes that contain data and behavior in a single unit. The essential building blocks of an OOP program are called objects, which can represent either concrete entities or abstract notions. It gives the users a way to use the variables and the methods without letting them see or modify the implementation of the code.

This quote has come from the understanding of two definitions, so when thinking of a way to implement the ADT, we will find out that we cannot have access to the implementation of the ADT, but we can use the methods that are implemented and public. On the other hand, we will mention that object orientation provides a way to encapsulate data and behavior, and if the user wants to use the methods that are implemented in code, they have to create an object from it. So, based on this understanding, we will see that both of the two terms have something in common, so this quote came from this understanding. We find that both are abstracted, so we can say that ADT is the basis for object orientation.

Now do I agree with the statement and the explanation?

I do not agree because if we go back to the implementation of ADT, such as stack, we will find the following: we used one of the main ideas in object orientation, which is encapsulation. We used to define a set of public and private actions to encapsulate data and behavior. The ADT's behavior can be implemented using secret operations, while only public operations are accessible to the outside world. And to access these behaviors, we must create an object from the ADT class type, which is one of the principles of object orientation.

## Q18)

* Ability to implement independent data in many ways. One of the strongest benefits of independent data is that we can implement it in different ways and with different techniques. Because they aren't restricted to specific implementations or language characteristics and may thus be simply ported and utilized in a variety of environments. For example, we can implement stack and queue by using array or linked list.
* Giving the programmer various ways to solve the problems that they face is important because sometimes we need to use the stack in linked lists, not arrays because the array is fixed size, so if the stack were not independent, we could face a big problem, so it helps to find solutions for the problems we get.
* Increasing the performance. Independent data structures give us many ways and options for fixing the problems so when we choose the right data structure it will help to increase the performance and reduce the bugs, we could face it. Which allow for improving and optimization of performance-critical features such as memory utilization, access time, or concurrency without compromising the data structure's overall functionality.
* Giving the programmers different ways and options to store the data based on the data, the size of the data, the ability for the data to grow, and so on. So, if we did not have independent data structures, it would cause a lot of problems when we wanted to store the data and add it.

# References:

baeldung (2020). *Space Complexity | Baeldung on Computer Science*. [online] www.baeldung.com. Available at: <https://www.baeldung.com/cs/space-complexity>.

Datta, S. (2021). *Time Complexity vs. Space Complexity | Baeldung on Computer Science*. [online] www.baeldung.com. Available at: <https://www.baeldung.com/cs/time-vs-space-complexity>.

www.studytonight.com. (n.d.). *Space Complexity of Algorithms | Studytonight*. [online] Available at: <https://www.studytonight.com/data-structures/space-complexity-of-algorithms>.

Cosentino, S. (2019). *Undestanding Encapsulation and Information Hiding*. [online] Medium. Available at: <https://scottc130.medium.com/undestanding-encapsulation-and-information-hiding-914a11d89b6f>.

www.tutorialspoint.com. (n.d.). *Difference between Data Hiding and Encapsulation*. [online] Available at: <https://www.tutorialspoint.com/difference-between-data-hiding-and-encapsulation>.

GeeksforGeeks (2017). *Abstract Data Types - GeeksforGeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/abstract-data-types/>.

GeeksforGeeks (2015). *Queue Data Structure - GeeksforGeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/queue-data-structure/>.

Techopedia.com. (2020). *What is a Stack? - Definition from Techopedia*. [online] Available at: <https://www.techopedia.com/definition/9523/stack>.

Croy, M.H. (2020). *How To Calculate Time Complexity With Big O Notation*. [online] Medium. Available at: <https://medium.com/dataseries/how-to-calculate-time-complexity-with-big-o-notation-9afe33aa4c46>.

Stack Overflow. (n.d.). *algorithm - Big O, how do you calculate/approximate it?* [online] Available at: <https://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it>.

De Meester, B., Seymoens, T., Dimou, A. and Verborgh, R. (2020). Implementation-independent function reuse. *Future Generation Computer Systems*, 110, pp.946–959. doi:https://doi.org/10.1016/j.future.2019.10.006.