**A picture containing graphics, font, symbol, red

Description automatically generated**

**Data Structures & Algorithms**

**Made by: Nasrullah Rami Hamad**

**Supervised By: Hana’ AL Rasheed.**

**Part 1:**

**Q1:**

The Min-Priority Queue is a data structure that works the same way as the queue. But the Min-Priority queue works a bit differently, with each element having an integer number called a priority, the priority becomes higher if its value was lower, and vice versa.

When we want to insert an element, the element will have a priority input as an integer, and that integer will compare itself to the other integers inside the queue, if its priority is lower than the others then it will be placed at the rear, if it has the highest priority (lower than all the values) then it will be placed at the front, if its priority was in between elements, then it will be placed in between the elements.

Using the linked list data structure with this type of priority queue will allow us to insert an element with its priority inside the same node while having an address that targets the next node.

The Min priority queue has many real-life applications since it can be used in Dijkstra’s algorithm which can be used in GPS, by giving priority to the road with the lowest distance to the targeted destination and ignoring the roads with the higher distance since their priority is lower.

The Queue has three main operations that can be used inside the design. Such as, insert, getMin, and removeMin. The insert function works by inserting an element with a certain data type, and the priority as an integer number. The getMin will grant us the element that’s placed at the front. While the removeMin will remove the element at the front and shift all the elements to the left, the same way a normal queue works.

Each of the three Min Priority Queue operations has a time complexity (Big-O) as follows.

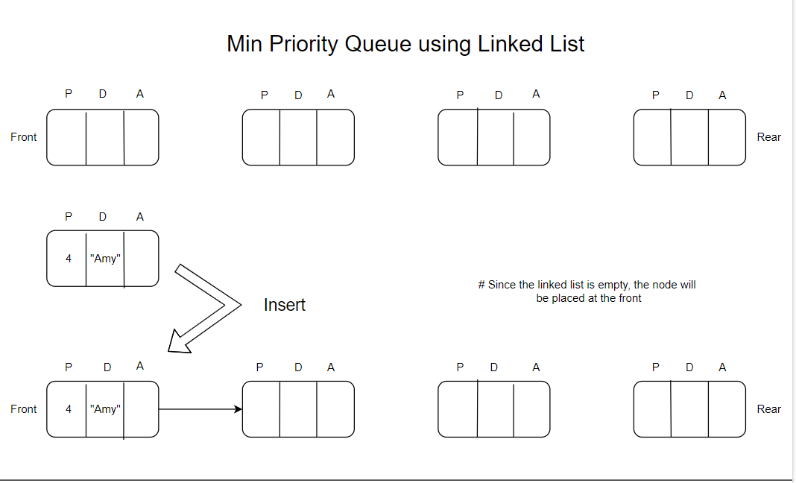
The (Insert) time complexity is O(n) because it has one loop inside its implementation.

The (getMin) time complexity is O (1) because it has no loops inside its implementation.

The (removeMin) time complexity is O (1) because it has no loops inside its implementation.

**Q2:**

**The design of the min priority queue consists of several operations that were used inside the design.**



A picture containing diagram, text, plan, technical drawing

Description automatically generated

A picture containing diagram, text, line, screenshot

Description automatically generated

A picture containing diagram, text, plan, technical drawing

Description automatically generated

A picture containing diagram, technical drawing, plan, line

Description automatically generated

**Q3:**

**The code is sent via github & E-learning.**

**Part 2:**

The time needed to do every sorting algorithm according to the input size and the data being sorted or no:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 5000 | 50000 | 500000 |
| Selection Sort | Sorted | 70 | 3240 | 279703 |
| Reversely Sorted | 59 | 2088 | 161866 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 5000 | 50000 | 500000 |
| Merge Sort | Sorted | 2 | 15 | 79 |
| Reversely Sorted | 1 | 11 | 63 |

**Q4:** The theoretical time complexity for each of the sorting algorithms, covering the best and the worst case if found.

|  |  |  |
| --- | --- | --- |
| ###################### | Selection Sort | Merge Sort |
| Best case | O(n^2) | O(n \* log n) |
| Worst case | O(n^2) | O(n \* log n) |

**Q5: Comparing both sorting algorithms on sorted data and reversely sorted data and explain which one is better and why:**

The Selection sort algorithm gave a lower performance than the Merge sort algorithm due to their time complexity difference, since the results given while they had the same input size were different.

For example, a sorted 5000 input sized array gave 70ms using the selection sort, but for the merge sort the result was 2ms.

For an unsorted 50000 input sized array, the merge sort algorithm gave the results in a lower time scale being 11ms, comparing it to the selection sort in the same type of data and array size, the results were 2088ms which is way longer than the merge sort.

Using an unsorted and a sorted 500000 input sized array for the selection sort, the results were the following:

For the unsorted array, the selection sort gave a time scale of 161866ms while it gave a time scale of 279703ms for the sorted data. The results for the merge sort algorithm for the same input size and using both types of data, the time scales given were 79ms for the unsorted type and 63ms for the sorted type.

Judging by the results above, the merge sort made its sorting technique faster and in a shorter time scale, and the selection sort algorithm was slower in performing its sorting technique and was in a longer time scale.

**Q6:**

For the selection sort, the best- and worst-case scenarios had the same time complexity o(n^2). Due to there being a nested loop inside the code, one for holding the sorted elements, and one for searching for a lower value than the sorted one.

For the merge sort, the best- and worst-case scenarios also had the same time complexity o (n \* log n). Since the divide and conquer strategy holds more than one loop with dividing operations. That gives a time complexity of O (log n). And since that loop was done n times, the time complexity of the operation becomes O (n \* log n).

Judging by the results above, the merge sort made its sorting faster in a short period, and the selection sort algorithm’s performance was slower compared to the merge sort algorithm. Which is the reason why the performance is worse for the selection sort and the results were way higher.

**Q7:**

Looking at how the time complexity compared both the algorithms and their results. Both the sorting algorithms (Selection / Merge) can also be compared by the space complexity. Space complexity sums the total space taken by both the algorithms while taking note of the input size.

The space complexity for both the merge sort and the selection sort are different. For the merge sort, since the array keeps dividing into smaller parts then put together, that way the space will change everytime it splits, therefore the complexity O(n). But since the selection sort makes the sorting process on the same array, without splitting or dividing the array to smaller pieces, the space complexity of it is better and constant O (1).

Based on the results above, the selection sort is better than the merge sort in space complexity but is worse than it in time complexity. The merge sort is better than the selection sort in time complexity but is worse than it in space complexity.

**Q8:**

Dijkstra’s algorithm:

A picture containing diagram, circle, line, number

Description automatically generatedBased on the question, we need to find the shortest distance going from vertex 0 to vertex 5. To do that, first we need to set up all the vertices and then put their distance to INFINITY, and the distance from the source to itself as 0.

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous | Visited |
| V0 | 0 | V0 | False |
| V1 | INF |  | False |
| V2 | INF |  | False |
| V3 | INF |  | False |
| V4 | INF |  | False |
| V5 | INF |  | False |

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous | Visited |
| V0 | 0 | V0 | True |
| V1 | 24 | V0 | False |
| V2 | 5 | V0 | False |
| V3 | 10 | V0 | False |
| V4 | INF | // | False |
| V5 | INF | // | False |

After that, we must check the neighbor vertices from each vertex and see whether it is directed or undirected. Then write the distance from each vertex while building up the distance from the source. As follows:

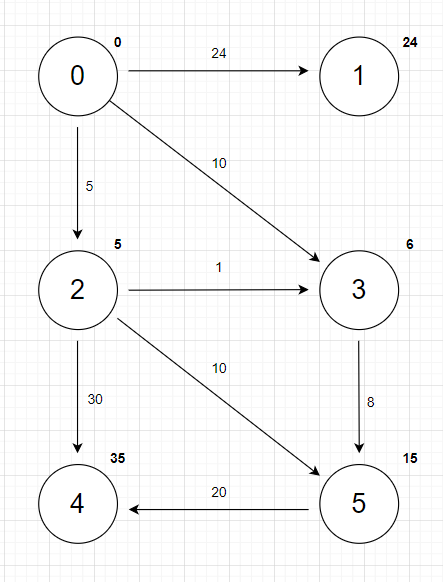
As the table shows, vertex 0 is connected to Vertex 1 with a distance of 24, and to vertex 2 With a distance of 5, and to vertex 3 with a distance of 10.

Now that the first vertex’s neighbors are all covered, the algorithm now sees vertex 0 as a visited vertex therefore it will be removed from the unvisited list of vertices.

After that, the dijkstra’s algorithm goes to find the minimum distance from the unvisited list of vertices, which is going to be vertex 2 with a distance of 5.

We will now check the neighboring vertices from vertex 2, which are vertex 3 with a distance of 1, and vertex 4 with a distance of 30, and vertex 5 with a distance of 10. We will update our distance column in the table, and check if there is a lower distance while building the distance from the source, then we will update it, if it is higher than the previous distance then we will keep it as it is.

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous | Visited |
| V0 | 0 | V0 | True |
| V1 | 24 | V0 | False |
| V2 | 5 | V0 | True |
| V3 | 6 | V2 | False |
| V4 | 35 | V2 | False |
| V5 | 15 | V2 | False |



Notice that the distance from vertex 2 to vertex 3 is 1 but from the source is (5 + 1 = 6) therefore, vertex 3 now has a new minimum distance of 6 with the path 0 🡨 2 🡨 3.

Now that vertex 2 is visited, we will check the new minimum vertex from the unvisited element which is vertex 3 with one neighboring vertex which is vertex 5 with a distance of 8. So, after building the distance from the source, the updated table will now be:

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous | Visited |
| V0 | 0 | V0 | True |
| V1 | 24 | V0 | False |
| V2 | 5 | V0 | True |
| V3 | 6 | V2 | True |
| V4 | 35 | V2 | False |
| V5 | 14 | V3 | False |

After updating the table and vertex 3 becoming visited, we will go to the new minimum vertex which is vertex 5. Vertex 5 has one neighboring vertex which is vertex 4 with a distance of 20. By updating the table, the new V4 distance from the source is 34.

After that, the next vertex is going to be vertex 1, but as the figure shows, vertex 1 doesn’t have any neighboring vertices. Therefore, it can’t be led anywhere, and it will become visited with no changes. Then with vertex 4, which also doesn’t have any neighboring vertices, it will become visited with no changes too. The final table of dijkstra’s algorithm is:

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous | Visited |
| V0 | 0 | V0 | True |
| V1 | 24 | V0 | True |
| V2 | 5 | V0 | True |
| V3 | 6 | V2 | True |
| V4 | 34 | V5 | True |
| V5 | 14 | V3 | True |

**Based on the table above, the shortest path from vertex 0 to vertex 5 is 14, following the path 0 🡨 2 🡨 3 🡨 5.**

**Bellman Ford:**

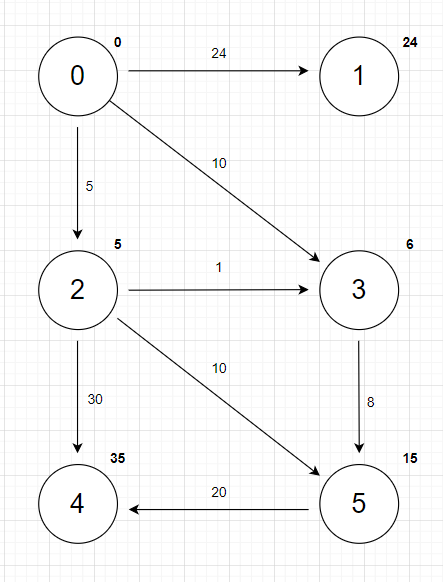
To perform the bellman ford algorithm to find the shortest path for each vertex starting from the source (V0), we will let every node have Infinity distance from the source same as the Dijkstra’s algorithm.

After that, we will get the shortest distance from each vertex while building them from the source.

A picture containing diagram, circle, line, number

Description automatically generatedFirst Iteration:

|  |  |  |
| --- | --- | --- |
| Edge | Distance | Previous |
| V0 -> V1 | 24 | V0 |
| V0 -> V3 | 10 | V0 |
| V0 -> V2 | 5 | V0 |
| V2 -> V3 | 6 | V2 |
| V2 -> V4 | 35 | V2 |
| V2 -> V5 | 15 | V2 |
| V3 -> V5 | 14 | V3 |
| V5 -> V4 | 34 | V5 |

After getting all the edges while building on them starting from the source, we will do a 2nd iteration that will check on each edge and make sure that they are the shortest path possible,

|  |  |  |
| --- | --- | --- |
| Edge | Distance | Previous |
| V0 -> V1 | 24 | V0 |
| V0 -> V3 | 10 | V0 |
| V0 -> V2 | 5 | V0 |
| V2 -> V3 | 6 | V2 |
| V2 -> V4 | 35 | V2 |
| V2 -> V5 | 15 | V2 |
| V3 -> V5 | 14 | V3 |
| V5 -> V4 | 34 | V5 |

It looks like the 2nd iteration didn’t update on the table and the shortest distances from each vertex stayed the same, this means that the first iteration was enough to reach the shortest distances on each vertex.

**Q9:**

**The Dijkstra’s algorithm code is sent via Github**

**Q10:**

**The Dijkstra’s algorithm code with error handling is sent via Github**

**PART 3:**

**Q11:** The palindrome pseudo code’s concept was based on two stacks, one stack getting elements from a string pushed into it, and the elements are copied into the second stack until the original stack is empty. Since the stack is a FILO structure (First in Last out), the second stack becomes a reversed stack from the original one. Then it starts comparing each element for both the stacks. If each element is equal to its equivalent in the reversed stack, then the string is a palindrome. If this is not the case, then the string is not a palindrome. Here’s the pseudo-code for the palindrome:

**START:**

Boolean palindrome(word): *// declaration of the function and creation of the two stacks.*

Originalstack = new stack

Reversedstack = new stack

For each letter in word: *// inserting the word inside the first stack.*

Originalstack.push(letter)

While originalstack is not empty:  *// Copying the elements inside the first stack to the second one.*

i = originalstack.pop ()

reversedstack.push(i)

While originalstack is not empty: *// Comparing the original stack to the new reversed stack.*

Letterfirst = originalstack.pop ()

Lettersecond = reversedstack.pop ()

If letterfirst is not equal to lettersecond: *// compares each element from the first stack to the second stack.*

Return false  *// false if the elements do not match (word is not a palindrome)*

Else

Return true  *// True if the elements match (word is a palindrome)*

**END:**

**Q12:**

For the palindrome pseudocode, the stack data structure was used. It is a collection of elements that are inserted and deleted from one end, usually called the top. The stack is a Last-In-First-Out (LIFO) data structure with two main operations. Push and Pop.

Push is the operation where elements are inserted in the stack, and they are inserted from one way only. The elements get stacked on top of each other.

Pop is the operation where elements are removed from the same end the elements were inserted from. It removes the recently inserted element in the stack.

There are two other minor operations that can be used in stacks, which are IsEmpty, and Top.

IsEmpty which returns True if the stack is empty. And False if the stack is not empty.

Top is an operation that returns the element that is on the top of the stack.

All these operations are performed in O(1) constant time.

* A picture containing text, screenshot, diagram, line

  Description automatically generated**The design illustration for the real-life scenario of the stack:**

A screen shot of a device

Description automatically generated with low confidence

A picture containing text, diagram, screenshot, line

Description automatically generated

A picture containing text, diagram, screenshot, line

Description automatically generated

A screen shot of a diagram

Description automatically generated with low confidence

A screen shot of a computer

Description automatically generated with low confidence

A picture containing text, screenshot, diagram, parallel

Description automatically generated

A picture containing text, diagram, screenshot, line

Description automatically generated

**Q13:** The call stack illustration starts with an empty stack. That fills after every function call, as the following:

1. Main function
2. Six function (1) calls, as it goes down in the index until -1. Each function call includes a print function.
3. Two function (2) calls. The first call happens after getting the index to -1. The other function (2) gets called after finishing all the function (1) calls.
4. After that first function (2) gets called, the function (1) calls start popping out from the stack, and each print their print statement.
5. After every function (1) call gets popped, a second function (2) appears and prints the print statement.
6. The main function then gets popped and the program ends.

A picture containing text, diagram, number, parallel

Description automatically generated

A picture containing text, diagram, plan, screenshot

Description automatically generated

A picture containing line, diagram, plot, rectangle

Description automatically generated

**Part 4:**

|  |  |  |  |
| --- | --- | --- | --- |
| ############### | Linked List | Array Unsorted | Array Sorted |
| Search | O(n) | O(n) | O (Log n) |
| Insert | O(n) / O (1) | O(n) | O(n) |
| Remove | O(n) | O(n) | O(n) |

**Q14:**

**The spell checker** application is an algorithm that searches for an inserted word and checks if it is spelled correctly. Otherwise, it is considered spelled wrong, and it would search for a word from the same prefix and in the same way it was typed in.

We must use Array Sorted with the search operation since it searches for the spelling error if found, and the sorted array was selected due to its time complexity since it is the best out of the other structures. Insert and remove operations cannot be used since it shouldn’t insert words or remove them. It should only alert the user that the word is mistakenly spelled.

**Priority Queue**: this algorithm prioritizes the priority values based on the highest value; it is mainly used to sort arrays based on their priority. The best structure to use for this algorithm is linked lists, since they can store data and its priority in the same node, and make sure it is addressed to the next node by an address given. The node is put in the beginning of the list if the list is empty.

Insert and remove operations can be used with the linked list structure, since the nodes can be inserted inside the priority queue if a new node is present, and the remove operation can be used when the queue serves the front node where it gets removed.

**Q15:**

A picture containing text, screenshot, font, number

Description automatically generated

The following code’s (big-O) notation can be studied by splitting the code into three parts.

* The first IF statement contains a time complexity of O (1).
* The three function declarations, which can be named as O(n-1) and O(n-2) and O(n-4).
* And the last IF statement which also contains constant time complexity.

First: since the three function declarations are in linear time, they can be named as equivalents, by **O(n-1) = O(n-2) = O(n-4).** This means that the time complexity for the three-function declaration is **3 \* O(n-1).** And if we add the constant time from the IF statements, the base statement will be as the following**: 3 \* O(n-1) + 1.** Which is the same as **3 \* O(n-1) + C**.

To evaluate the time complexity of the above code. We should use the code two times. And stack them on each other. As the following:

**First iteration:**

3 \* (3 \* O(n-1) + c) + c

**#by distribution law:**

9 \* O(n-2) + 4c. This is the result of the first iteration.

(Notice that the n-1 became n-2, This is just to keep track of the number of iterations).

**Second Iteration:**

9 \* (3 \* O(n-2) + c) + 4c

We multiplied the 9 with the base statement. To get the result of the second iteration.

**#by distribution law:**

27 \* O(n-3) + 13c. This is the result of the second iteration.

Notice that the number at the beginning of the time complexity keeps increasing in a specific format (3n). Since the number at first was 3 multiplied by the base statement, that was when the n inside the time complexity was 1, which gave a result of 3.

On the second iteration, the base statement was multiplied by 9 which is the result when n = 2 in the time complexity. Lastly, on the third iteration, the number multiplied by the base statement was 27 which is the result of 3 to the power of n = 3.

Based on the results above, the time complexity for the code is **(3n).**

**Q16:**

Firstly, the ADT (Abstract Data Type) is how operations are seen after being performed, but not how they were implemented, the ADT doesn’t show how data will be saved inside the memory; it will only show the abstract view of the operations. There are a lot of ADTs such as Lists, Stacks, queues and more.

Stacks are ADTs that are known as LIFO data structures. They have many operations such as pop, push, isEmpty, and top. The stack also consists of nodes, the first node is called the head, and the last node is called the tail. Encapsulation plays a big role in stacks, since the head node is encapsulated, therefore the pointer that points to the stack is the only thing seen when calling the function.

Encapsulation, usually called information hiding, is used to secure the classes and the data inside it, it can also restrict user access to the data inside the structure. As mentioned before, the implementation of each operation is hidden from the users of the ADTs. This is done by the encapsulation process. The advantages of encapsulation are as the following:

* Simplicity for the users: as mentioned, encapsulation prevents the users from seeing how operations are implemented. Just like a black box that contains all the details.
* Information Hiding: If the information and data inside the ADTs are hidden from the public, it can prevent unauthorized altering to the data and the deletion of the information inside the ADTs and structures.
* Flexibility: Since ADTs can vary from structure to another, it can be easy to change from one another. Therefore, the data can be kept safe inside multiple structures depending on the case or how they are used.

**Q17:**

I agree with the fact that ADTs are a basis for object orientation. Since OOP uses objects andclasses and the inheritance algorithm to encapsulate data and provide protection for them while being in full efficiency, they can be known for their connectivity and high flexibility and efficiency in connecting operations to each other. They can also be used in ADTs, since some ADTs such as queues are known for their operations such as ‘enqueue’ and ‘dequeue’ rather than their implementations. The operations of ADTs normally use classes and objects to implement and perform them. The object-oriented paradigm can be used to help implement the ADT operations in high performance.

**Q18:**

Implementation independent data structures are also known as ADTs which don’t focus on the implementation of the structure, rather, ADTs focus on how the operations are performed without getting into details about how they can be performed. The Implementation independent data structures vary in plenty of benefits such as:

* **Modularity**: which means that the structures can be combined and merged with other structures so they can be more flexible in their operations and their performance when operating a specific problem.
* **Maintenance and debugging**: Since the implementation independent data structures don’t focus on the internal details, they can test the structures’ expected result and behavior with simplicity and flexibility, which can increase the performance of the debugging process, since it narrows down the problems to either the implementation or the code itself.
* **Encapsulation:** as mentioned above, encapsulation plays a big role in access control to the data of the implementation independent structures, since it prevents unauthorized access to the structures, therefore, it creates a safer environment for using the data structures and it can be used without worrying about the implementation part and its details for more simplicity and flexibility. Abstraction can also be used in encapsulation since it helps in splitting the two parts, the abstraction, and the implementation.

References:

1. GeeksforGeeks. (2011). *What does ‘Space Complexity’ mean?* [online] Available at: [https://www.geeksforgeeks.org/g-fact-86/.](https://www.geeksforgeeks.org/g-fact-86/)
2. GeeksforGeeks (2017). *Abstract Data Types - GeeksforGeeks*. [online] GeeksforGeeks. Available at: [https://www.geeksforgeeks.org/abstract-data-types/.](https://www.geeksforgeeks.org/abstract-data-types/)
3. Quora. (n.d.). *Is it true that abstract data types (ADT) is one of the basic ideas behind object orientation?* [online] Available at: <https://www.quora.com/Is-it-true-that-abstract-data-types-ADT-is-one-of-the-basic-ideas-behind-object-orientation.>

‌