**Data structures and algorithms:**

**By: Abdallah Salhab**

**21110181**

**Part1 task 1:**

ArrayList:

In programming, an Array List is a data structure that represents a dynamic array, which size can change on a continual basis.

The array list functions are:

Insert, insert at, remove, remove at, count, clear.

|  |  |
| --- | --- |
| (1) | (2)  Insert(“Ali”) |
| (3)  Insert(“Samer”)  A yellow arrow next to blue squares  Description automatically generated | (4)  IsEmpty()  End is not equal to -1 so it is (False) |
| (5)  Insert(“Maher”) | (6)  ReadAt(1)  It will return what’s inside the list in the index 1 which is (Samer) |
| (7)  Insert(“Tamer”) | (8)  Insert(360) |
| (9)  InsertAt(3,”Hamzah”) | (10)  Insert(“Rami”)  But because it is full we are going to duplicate the max and make a new array with the new max and copy the old data to the new one and make the old name point at the new array |
| (11)  Remove() | (12)  Modify(4,”Waleed”) |
| (13)  Count()  It will return 5 | (14)  InsertAt(3,salem) |
| (15)  RemoveAt(4) |  |

Queue:

A queue is a data structure that represents a collection of elements with two main operations: enqueue and dequeue, it follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue will be the first one to be removed.

Queue functions:

Enqueue, dequeue, isEmpty

|  |  |
| --- | --- |
| (1) | (2)  Enqueue(Ali) |
| (3)  Enqueue(70) | (4)  Enqueue(Samer) |
| (5)  Dequeue() | (6)  Enqueue(Imad) |
| (7)  Enqueue(Rami) | (8)  Dequeue() |
| (9)  isEmpty()  the front is not -1 and the rear is not -1 and they are not the same so it is not Empty so it will return (False) | (10)  Size()  There is a function that adds to a counter when using Enqueue and subtracts from is when using Dequeue  So it will return(3) |
| (11)  Dequeue() | (12)  Dequeue() |

And in the queue there is a problem because if you reach the end of the queue you cant enqueue any more even though the there is usable space because of dequeue of the first spaces so for that the circular queue is made; so when using enqueue the code checks if the queue is empty then the front = 0 and the enqueue is to 0 but if it is not empty the code uses the rear as an index after preforming rear = (rear+1)mod capacity of the queue so if the index is 8 and the capacity is 9 and we want to enqueue the 8+1 mod 9 = o so enqueue at 0 if there is space,

And the same for dequeue but we dequeue from the front after doing front = (front+1) mod capacity.

Stack:

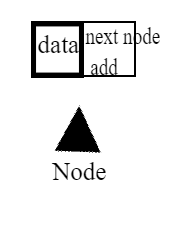
A stack is a data structure that follows the Last In, First Out (LIFO) principle. In a stack, elements are added and removed from the same end, which is referred to as the "top" of the stack. The basic operations associated with a stack are:

Push to add an element, pop to remove and element, top to return what is in the top of the stack and is Empty to check if it is empty.

|  |  |
| --- | --- |
| (1) | (2)  Push(Hashem) |
| (3)  Push(Ali) | (3)  Push(Salem) |
| (4)  isEmpty()  because top() 🡪 2  it returns (False) | (5)  Top()  It will return (Salem) |
| (6)  Push(Qasem) | (7)  Pop() |
| (7)  Pop() | (8)  Push(450) |

Linked list:

linked list is a data structure that consists of a sequence of elements, each of which points to the following element in the sequence, linked list fixes a problem in array list because linked lists do not have a fixed size in memory, and elements can be dynamically added or removed without the need to shift other elements , linked list is made up of nodes, nodes are made out of the data or value that the node holds and pointer to the next node in the sequence, and there is a doubly linked list that every node in it contains references to nodes that come before and after It, the linked list can perform the same operations as the array list.



|  |  |
| --- | --- |
| (1)  insert (5) | (2)  insert (17) |
| (3)  insert (10) | (4)  Insert At (1,12) |
| (5)  remove () | (6)  remove At (1) |

**Part1 task2:**

In my text editor I used the array list functions such as insert at; that enables the user to add data to the array at a specific location (index) , remove at; that enables the user to delete data form a specific location (index) … to be a place to store the data in

|  |  |
| --- | --- |
| (1)if the input is 1 that indicates for the insert then the index then the data (1 0 ali) | (2)input (1 1 samer) |
| (3)input(1 2 emad) | (4)input(2 0)  The number 2 is for the delete function and the 0 is for the index |
| (5) input (1 2 sarah) | (6)input(1 1 fadi) |
| (7)input(2 0) | Lets say that the user wants to input something and that the array isFull() = true    (8)the max is going to double and a new array will be mad and every thing is moved then it is named the same as the first one with double the length |

In my text editor I declared the number of operation that can be done and used it to determine the size of the stack and to make the undo function in my text editor I used the stack so when ever I used any function I pushed the number of that function into the stack and when ever I wanted to undo I would look at the top of the stack and if it is insert I would delete and if it is delete I would insert through variables that is saved as the last insert value and index, but the stack that I used is a Linke List Stack

|  |  |
| --- | --- |
| First the user would input the number of operations that he wants to use  Input(5)  So the Max the linked list stack would go is 5 | (2) if the user inputs (1 0 ali)  The stack push(1) |
| (3) if the user deletes input(2 0)  push(2) | (4) the user print (3 1)  push(3) |
| (5) the user insert input(1 1 samer)  push(1) | (6) if the user wants to undo input(4)  The top is 1 so insert, the code deletes at the index where the last insert was  And then push(4)  And the system print that is operations that can be done is finished and ends. |

I used the Queue in my code to make the last operation possible that is history; the history operation when called it returns and prints the operation that was used and if it was the first, second third…

And I managed to do that by using enqueue() for every operation used and dequeue() after returning and printing the operation with a counter for the number of each operation

|  |  |
| --- | --- |
| I made the queue the same length as the stack | (1) Input (1 0 rami)  Enqueue(1) |
| (2) input (2 0)  Enqueue(2) | (3) input (4)  Enqueue(4) |
| (4) input (5) the operation that shows the history  Enqueue(5)  Prints (the operation number 1 was insert)  Dequeue() | (5) inputs(3 1)  Enqueue(3) |

**Part 1, task 3:**

In my code I used the FIFO or the queue in the history operation when called it returns and prints the operation that was used and if it was the first, second third… operation.

And I managed to do that by using enqueue() to the number of every operation used and dequeue() after returning and printing the operation that is pointed at by the (rear) + a number that indicates the order of that operation.

**Part 1, task 4:**

Stack ADT, the ADT describes the kind of operations that can be done with the data, how those operations should behave.

And in the case of the stack they are:

Push: Adds an element to the top of the stack.

Pop: Removes the element from the top of the stack.

Top: Returns the element at the top of the stack.

Is Empty: Checks if the stack is empty.

Count: Returns the number of elements in the stack.

In my text editor I declared the number of operation that can be done and used it to determine the size of the stack, I also used it to make the undo function in my text editor I used the stack so whenever I used any function I pushed the number of that function into the stack and whenever I wanted to undo I would look at the top of the stack and if it is insert I would delete and if it is delete I would insert through variables that is saved as the last insert value and index.

**Part 1, task 5:**

ADT is very helpful when problem solving because it helps break down a big problem into smaller sub problems that can be handled, and it makes it easier to implement, design and test each component alone, it also promotes code reusability if there are another code that need the same thing; like if there are to codes in need for a calculator for a specific math problem, you only write the code once and use it in both codes.

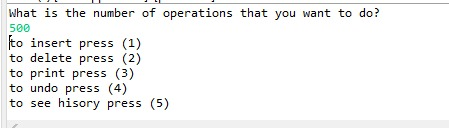
A code written in ADT is easier to maintain because if the coder changed anything to make it better it will be on the background and the code using it wont change.

In my code I used ADT to make the different data structures features and encapsulated the different functions needed to make them work.

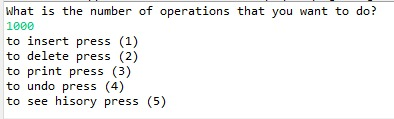
**Part 1, task 7:**

Testing:

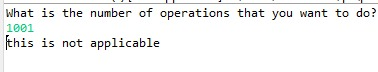
1-first lets test if the limit for the number of operation is working:

so if I enter a viable number it should work

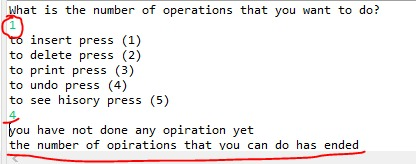
And if I entered a number of operations = 1000 it should work



And if I entered a number of operations bigger than 1000 it should give an error massage.

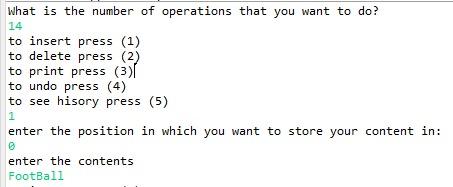


\* what if the number of operations has ended:



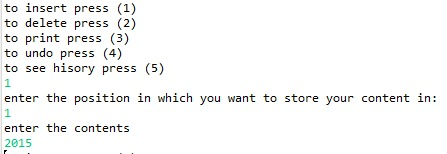
2-secondly, lets test the insert function in the text editor:

Inserting (football ) at position 0



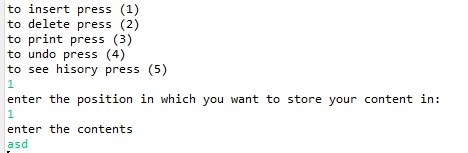


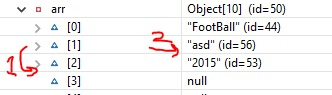
Inserting (2015) into position 1





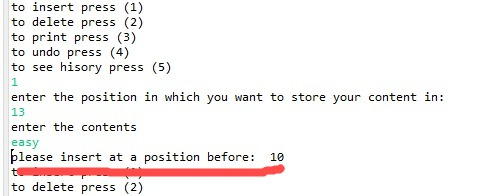
\*now what happens if I insert at a position that is already inserted at



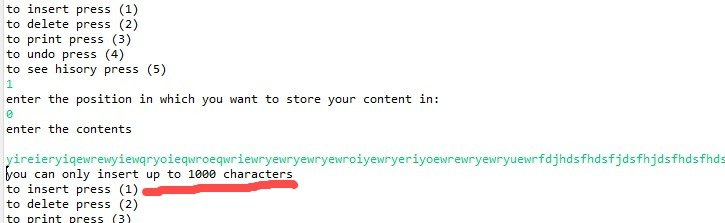


So it is going to shift the old data into the position after it and insert the new data at the position

\* now what if I insert at a position over the limit of the array:



\* now what if I inserted over 1000 characters:



3- now lets test the delete function:

First lets delete at an index that is empty:

A white grid with black text

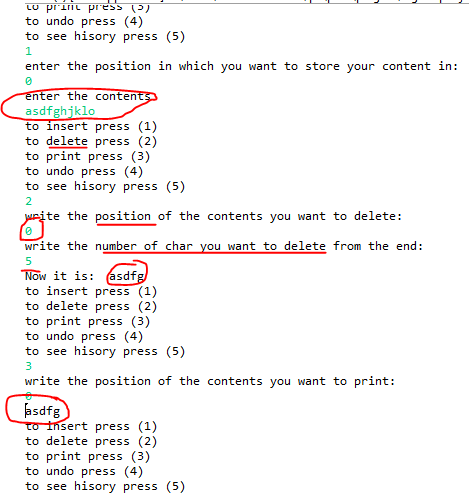
Description automatically generated

A white background with black text

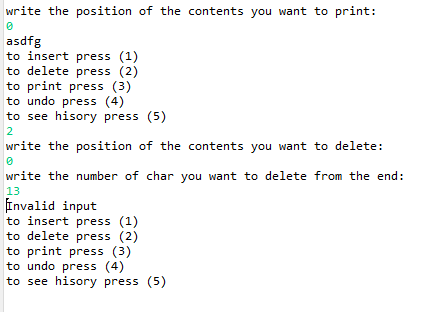
Description automatically generated

It prints (“it is empty”)

\_ now lets delete 5 char from a 10 char string:



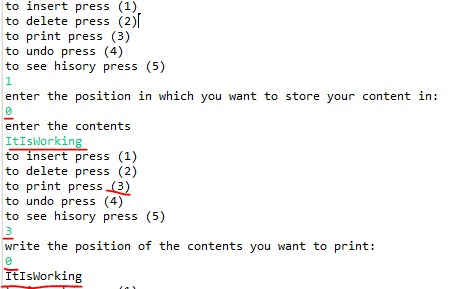
-what if I inserted a number of characters that is bigger than the string it self



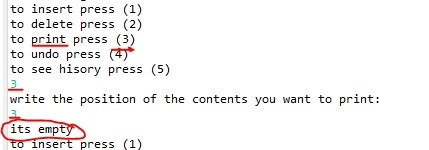
It print (invalid input)

4-now lets test the print function:

Inserted at a position and printed what is in that position:



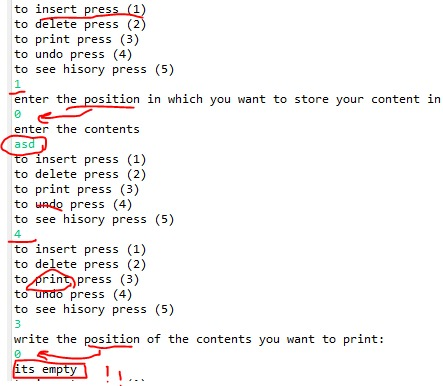
and if there is nothing inserted into that position:



5-now lets test the undo function:

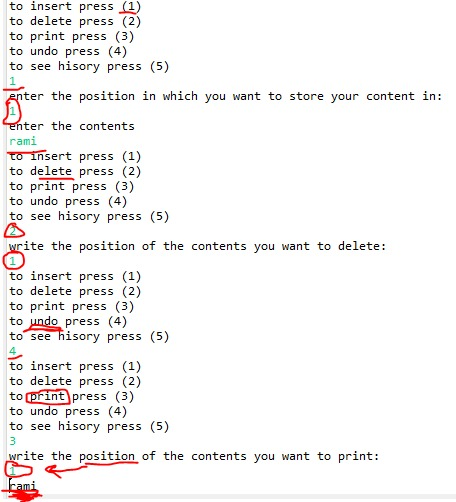
undo for insert:

inserted then undo and to check I printed at the position of the insertion:

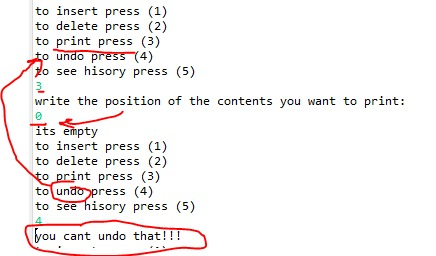


undo for the delete:

inserted data and deleted it and then undo, and to check I printed at the position of deletion:

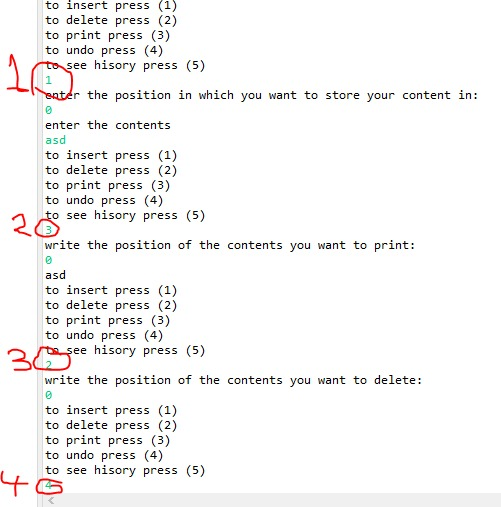


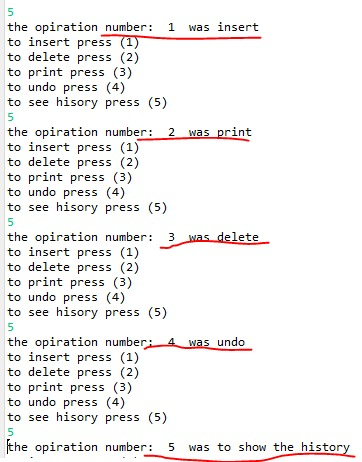
undo for the print or any other function other than insert and delete:



6-now lets test the history function:

first I inserted then I printed, then I deleted, then I used the undo:





It gave me the functions and their order.

**Part 1, task 8:**

In my code I used Encapsulation; because all the classes attributes are private and can be initiated by the constructor or by setters and getters, I also used abstraction to encapsulate the features of the data structures (functions), so the user only needs to interact with the public interface without needing to know about the inner functions of it

(A) Encapsulation advantages:

Data hiding is enabled by encapsulation. To prevent accidental data manipulation so we limit direct access to an object's attributes.

Code reusability, easily maintainable, security, modularity, code organization.

I used abstraction that is focusing on the essential characteristics of the system without specifying the internal functions and working of it, as I used to make the Operations of my data structures and making their internal functions abstract and implied on their operation in a general way, in like insert that is general and the user could easily know what it does without knowing the inner functions.

**Part 1, task 9: (B)**

An algorithm's complexity is a function that indicates how effective the algorithm is given the amount of data it has to process.

Time complexity and space complexity are used to assess an algorithm.

Time complexity: it refers to how long an algorithm takes to complete depending on the size of the input.

Big O notation is frequently used to represent it since it gives a limit on the rate at which the running time grows in relation to the input size.

Space complexity: The amount of memory an algorithm needs in relation to the size of the input.

A key factor when evaluating a system performance is algorithm complexity, achieving the best possible performance requires selecting algorithms with the right complexity characteristics (time, space) for the needs of a given application.

**Part 1, task 10:**

Time complexity represents the amount of time an algorithm takes to complete as a function of the input size.

A white paper with red text

Description automatically generated

For storin data inside a variable, or declaring an array that time complexity is O(1)

The loop time complexity is O(n) because it depends on how many times the loop is going to be done

In a nested loop the time complexity is O(N to the power of 2)

A computer code with black text

Description automatically generated

If there is an if statement the time complexity is the one that is worse out of the cases

A screenshot of a computer code

Description automatically generated

- Space complexity is a measure of the amount of memory space an algorithm uses in relation to the input size.

In this example the space complexity is O(1) because the amount of memory usage remains consistant

A computer screen shot of a math problem

Description automatically generated

A black text on a white background

Description automatically generatedin this example the size of the memory used is determined by the size of the array inputted.

A white background with black and white text

Description automatically generatednow this is a function that has a space complexity of O(N to the power of 2) because the memory needed a space that is the size of the input squared.

**Part 1, task 11: (C)(D)**

Normal data types are specified and defined by the programming language and the operations that can be done on them are previously defined and cant be changed, they are low-level and serve as the foundation for complex data structures, their implementation is fixed and usually provided by the programming language

In abstract data types are like used defined data types that defines the operations and values using functions with out showing how it works using encapsulation, so it is higher-level constructions that define relationships and operations, The user is not exposed to the implementation details of ADT. which makes it possible to use the same ADT in several implantations.

**Part1, task 12:**  
Time complexity:

First to calculate the time complexity of my code I calculated the complexity of the functions

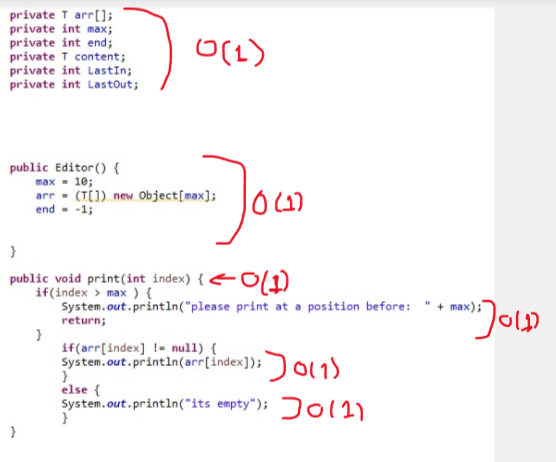
A screenshot of a computer code

Description automatically generated A screen shot of a computer code

Description automatically generatedA screenshot of a computer code

Description automatically generated A screenshot of a computer code

Description automatically generated



A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

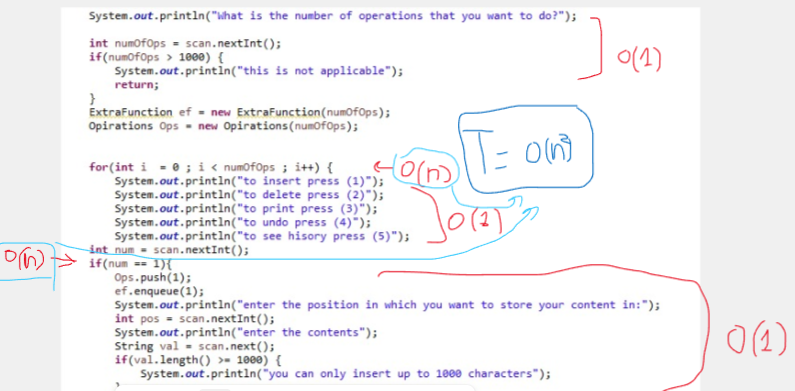
A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

And according to that I could calculate the Main class:



A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

So the time complexity is:

T = O(n^2)

**Space complexity:**

The Node class space complexity that is O(1)

A screenshot of a computer code

Description automatically generated

The My Linked List class space complexity that is O(1)

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

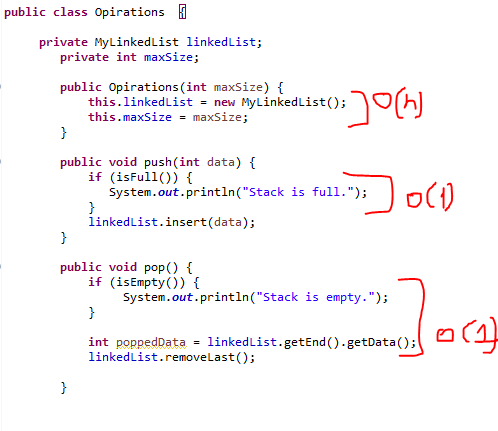
A screenshot of a computer code

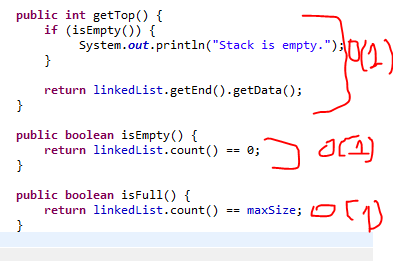
Description automatically generated

A screenshot of a computer code

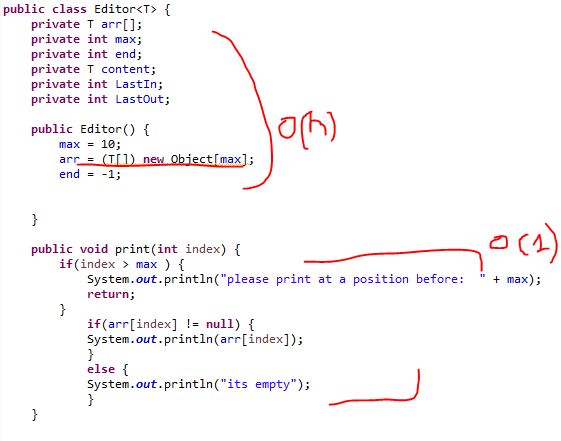
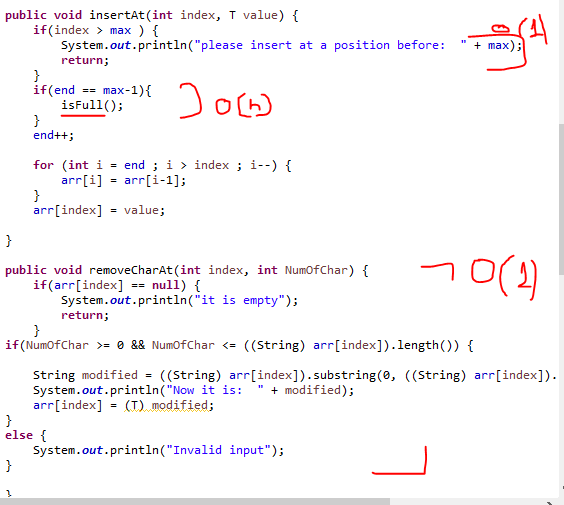
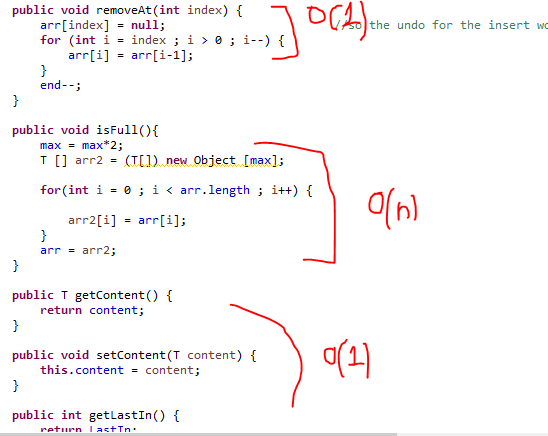
Description automatically generated

Now the space complexity of the operations class that is O(n):

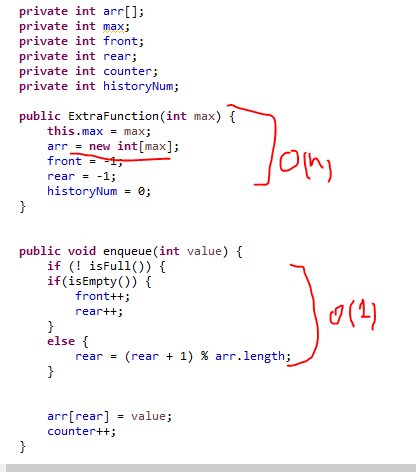
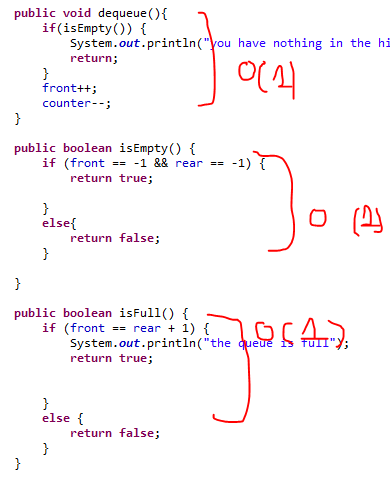
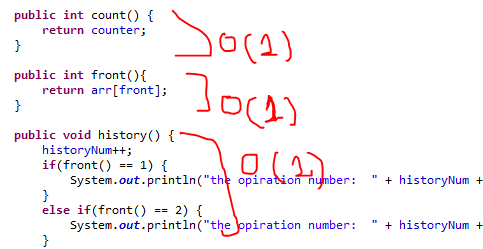
O(n) depends on the initial size of the linked list,



Now the editor class that is O(n):

now the extra function class that is O(n):

The space complexity of the main depends on the editor and opirations and extrafunction classes that has a space comlexity of O(n) so the main is O(n).

**Part 2, task 1:**

The memory stack during a sequence of recursive function calls (Fibonacci):

In this example the program is calculating Fib(6);

\*\*I made a mistake, I had to Push Fib(6) in the Beginning.

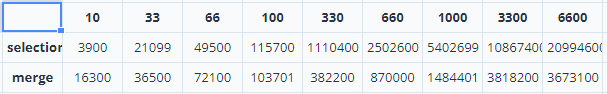
|  |  |
| --- | --- |
| (1)Push Fib(5) | (2)Push Fib(4) |
| (3)Push Fib(3) | (4)Push Fib(2) |
| (5)Push Fib(1)    Base Case: | (6) Return 1;  Pop() |
| (7)Push Fib(0)    Base Case: | (8) Return 0;  Pop() |
| (9) Return 1;  Pop() | (10)Push Fib(1)    Base Case: |
| (11) return 1;  Pop() | (12)return 2;  Pop() |
| (13)Push Fib(2) | (14)Push Fib(1)    Base Case: |
| (15)return 1;  Pop() | (16)push Fib(0)    Base Case: |
| (17)return 0;  Pop() | (18)return 1;  Pop() |
| (19)return 3;  Pop() | (20)Push Fib(3) |
| (21)Push Fib(2) | (22)Push Fib(1)    Base Case: |
| (23)return 1;  Pop() | (24)Push Fib(0)    Base Case: |
| (25)return 0;  Pop() | (26)return 1;  Pop() |
| (27)Push Fib(1)    Base Case: | (28)return 1;  Pop() |
| (29)return 2;  Pop() | (30)return 5;  Pop() |
| (31)Push Fib(4) | (32)Push Fib(3) |
| (33)Push Fib(2) | (34)Push Fib(1)    Base Case: |
| (35)return 1;  Pop() | (36)Push Fib(0)    Base Case: |
| (37)return 0;  Pop() | (38)return 1;  Pop() |
| (39)Push Fib(1)    Base Case: | (40)return 1;  Pop() |
| (41)return 2;  Pop() | (42)Push Fib(2) |
| (43) Push Fib(1)    Base Case: | (44) return 1;  Pop() |
| (45)Push Fib(0)    Base Case: | (46) return 0;  Pop() |
| (47)return 1;  Pop() | (48)return 3;  Pop() |
| (49) return 8;  Pop() |  |

The Final answer for Fib(6) is 8.

**Part 2, task 2:**

To analyze the performance of two sorting algorithms in this case(merge sorting, selection sorting) I used different lengths arrays that needs sorting, I used arrays with the length of 10, 33, 66, 100, 330, 660, 1000, 3300, 6600, and I entered those array into both the algorithms, and I recorded the execution times for them using two long s and using the difference between their nano Times I recorded the times.

The results:



When I inserted the results into a graph;

A graph with a line graph

Description automatically generated

\*\*after looking at the graph it seems that the selection sorting is actually faster under the point of 66 but after that the merge sorting becomes faster and is less affected by the size of the data unlike the selection sorting that gets affected a lot after that point.

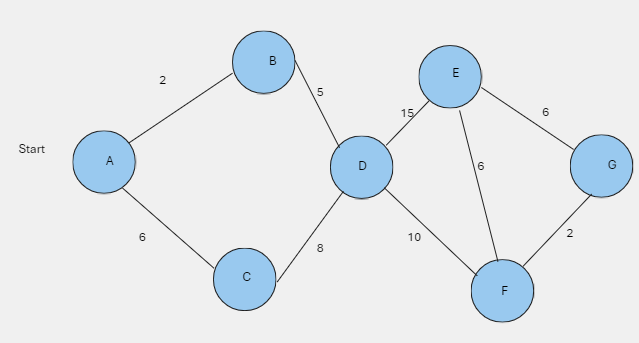
\*\*the graph can give us the time complexity of each sorting algorithm the merge sorting algorithm has a complexity of n log n and the selection has a complexity of n^2 (n to the power of 2).

But in terms of the space complexity the merge sorting algorithm needs a temporary space for the sub lists so the space complexity of it is O(n), on the other hand the selection sorting algorithm does not need an additional memory space so it s complexity is O(1)

The conclusion; the merge sorting is faster time wise but is worse space wise, on the other hand the selection algorithm is slower time wise but is more efficient space wise, and they are used according to the resources available in a specific application like time or space.

Part 2, task 3:

First lets start with the Dijkstra algorithm that has 7 vertexes and the start is A and the end is G



|  |  |  |
| --- | --- | --- |
| vertex | Distance | Previous |
| A |  |  |
| B |  |  |
| C |  |  |
| D |  |  |
| E |  |  |
| F |  |  |
| G |  |  |

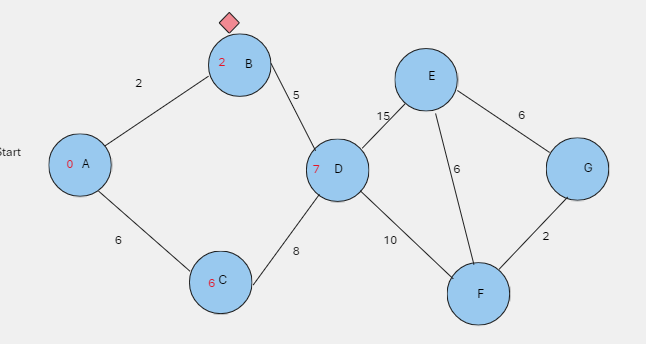
So the algorithm will start at A and will update the weights of the vertexes that are directly connected to it, and any other vertex will have the weight infinity.

A diagram of a network

Description automatically generated

|  |  |  |
| --- | --- | --- |
| vertex | Distance | Previous |
| A | 0 |  |
| B | 2 | A |
| C | 6 | A |
| D |  |  |
| E |  |  |
| F |  |  |
| G |  |  |

Then the A will become visited and the algorithm will chose the next vertex that has the lest weight, and reupdate the weights.



|  |  |  |
| --- | --- | --- |
| vertex | Distance | Previous |
| A | 0 |  |
| B | 2 | A |
| C | 6 | A |
| D | 7 | B |
| E |  |  |
| F |  |  |
| G |  |  |

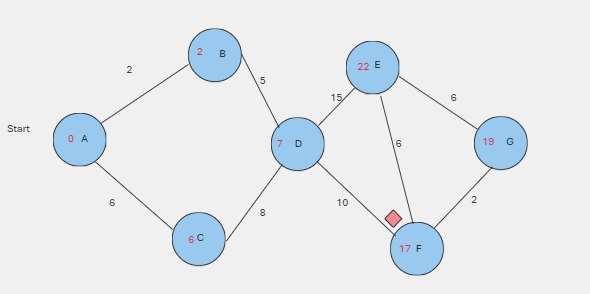
Then the B will become visited and the algorithm will chose the next vertex that has the lest weight and is not visited, and reupdate the weights.

A diagram of a molecule

Description automatically generated

|  |  |  |
| --- | --- | --- |
| vertex | Distance | Previous |
| A | 0 |  |
| B | 2 | A |
| C | 6 | A |
| D | 7 | B |
| E | 22 | D |
| F | 17 | D |
| G |  |  |

Then the D will become visited and the algorithm will chose the next vertex that has the lest weight and is not visited, and reupdate the weights.



|  |  |  |
| --- | --- | --- |
| vertex | Distance | Previous |
| A | 0 |  |
| B | 2 | A |
| C | 6 | A |
| D | 7 | B |
| E | 22 | D |
| F | 17 | D |
| G | 19 | F |

……………………………………………………………………………………..

Secondly bellmanford algorithm I have a 7 vertexes graph that starts at A

A diagram of a network

Description automatically generated



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| & | & | & | & | & | & | & |

A is the start and the weight to it is 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | & | & | & | & | & | & |

The weight D is the weight of A + the weight of the edge between them

0 + 5 = 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | & | & | 5 | & | & | & |

The weight C is the weight of A + the weight of the edge between them

0 + 5 = 5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | & | 5 | 5 | & | & | & |

The weight B is the weight of A + the weight of the edge between them

0 + 6 = 6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 6 | 5 | 5 | & | & | & |

The weight C is the weight of D + the weight of the edge between them if it is less

5 + 2 = 7 so no

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 6 | 5 | 5 | & | & | & |

The weight E is the weight of C + the weight of the edge between them if it is less

5+1=6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 6 | 5 | 5 | 6 | & | & |

The weight B is the weight of C + the weight of the edge between them if it is less

5-2=3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 6 | & | & |

The weight E is the weight of B + the weight of the edge between them if it is less

3+2=5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | & | & |

The weight F is the weight of D + the weight of the edge between them if it is less

5-1 =4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | & |

The weight G is the weight of E + the weight of the edge between them if it is less

5+3 = 8

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 8 |

The weight G is the weight of F + the weight of the edge between them if it is less

4+3 = 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight D is the weight of A + the weight of the edge between them if it is less

0+5=5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight C is the weight of A + the weight of the edge between them if it is less

0+5=5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight B is the weight of A + the weight of the edge between them if it is less

0+6=6 SO NO

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight C is the weight of D + the weight of the edge between them if it is less

=7 SO NO

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight E is the weight of C + the weight of the edge between them if it is less

=6 SO NO

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight B is the weight of C + the weight of the edge between them if it is less

=3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight E is the weight of B + the weight of the edge between them if it is less

=5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight D is the weight of F + the weight of the edge between them if it is less

=4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight G is the weight of E + the weight of the edge between them if it is less

=8 SO NO

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

The weight G is the weight of F + the weight of the edge between them if it is less

=7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G |
| 0 | 3 | 5 | 5 | 5 | 4 | 7 |

So after one round without any switch we have found the shortest path from the start to each of the vertices.

\*Time and space complexity: (E)

Dijkstra algorithm time complexity is O((V ^2 ) and can be optimized to O((V + E) \* log V). where |v| is the number of vertices and |E| is the number of edges, and the space complexity of it is O(|V|) because each vertex's approximate distances must be stored.

The Bellman-Ford algorithm time complexity is O(|V|+ |E|) and the space complexity is O(|V|)

Dijkstra applications:

Dijkstra's algorithm is used in real-world transportation applications, such as GPS navigation systems on cars and subways, traffic engineering to chose and design traffic flow and in network routing protocols

Bellman-Ford applications:

It is commonly used in network routing protocols if they have negative weights, can be used in path planning for robots, where the graph edges may have negative weights.

Although both algorithms can be used to find the shortest paths, Dijkstra's algorithm is favored in situations where efficiency is critical, particularly in graphs where there are relatively few connections between vertices compared to the total number of vertices, and when dealing with non-negative edge weights. When dealing with situations where the graph may have negative edge weights, Bellman-Ford is the preferred method.

**Part 3, task 1:**

The ADT uses the encapsulate of data and operation. They define a set of operations that can be performed on the data without revealing the internal details of the implementation. and that is an important concept in the OOP,

Abstraction in ADT present an overview of a data structure, pointing out its properties and the operations that it can execute, with out showing their internal, and in OOP the objects represent real entities and abstract away the complexities of their implementation, so they are recognized by the user that knows their implementation in the reals life.

Examples:

In OOP if I want to create a car object:

A computer screen shot of text

Description automatically generated A screenshot of a computer program

Description automatically generated A computer code with blue text

Description automatically generated

In this car object I used encapsulation and abstraction like in the OOP principals

Now in ADT:

A white background with black text

Description automatically generated

A computer code with text

Description automatically generated A computer code with black text

Description automatically generated

ADT uses the principals that then used in OOP.

**Part 3, task 2:**

An independent data structure is a data structure that does not need another data structure to be implemented, and it has the basic operations to store and organize data (ex. Linked list, array list)

A dependent data structure uses the independent data structures operations as a base for their data organization and storing them in specific ways (ex. Stack, queue).

Independent data type:

The linked list does not any other ADT to be implemented .

A screenshot of a computer code

Description automatically generated

While the stack needs another ADT to be implemented:

A screenshot of a computer code

Description automatically generated

(A) <https://www.studysmarter.co.uk/explanations/computer-science/computer-programming/encapsulation-programming/>

(B) <https://www.geeksforgeeks.org/what-is-an-algorithm-definition-types-complexity-examples/>

(C)<https://www.teach.cs.toronto.edu/~csc110y/fall/notes/10-abstraction/04-abstract-data-types.html#:~:text=Abstract%20data%20types%20form%20a,how%20their%20operations%20are%20implemented>.

(D)<https://www.youtube.com/watch?v=ZniDyolzrBw>

(E) <https://iq.opengenus.org/time-and-space-complexity-of-dijkstra-algorithm/#:~:text=This%20will%20bring%20our%20total,edge%20has%20to%20be%20visited>.