1)

Node.java:

We can see in the given "Node.java" code implementation of Node class looks consistent with all intended functionality. It gives node that contains string value(which is data) and also Linked list(next) all the References to next node.

The data and next instance variables of the Node class additionally have public getter and setter methods, enabling linked list manipulation.

The Node class's constructor accepts a string parameter, which it uses to populate the data instance variable. Although it is not utilized in the provided code, the no-argument constructor is also offered.

The toString() method is also implemented, which returns the string value of the data instance variable. This can be useful for debugging and printing the contents of the linked list.

Moreover, the toString() function is implemented, returning the data instance variable's string value. This can be helpful for printing the linked list's contents and diagnosing problems.

As it offers the required methods to store and handle the data and pointers in a singly linked list, the implementation of the Node class looks to be accurate and comprehensive overall.

SinglyLinkedList.java:

Overall, After verifying the code functionality seems to be implemented effectively. It is challenging to evaluate whether the software satisfies all requirements without more details on the use cases and requirements.

SinglyLinkedList with support of Insertion & Deletion of nodes.

I. **Insert**: Inserting a node in the middle or at the top of the list is handled appropriately.
II. **Remove**: This remove method correctly removes node from the list.
III. **Find**: provides the first node that matches a specified value after conducting a search for that node correctly.
IV. **Size**: Size method of the list written in the code correctly
V. **ToString**: Here ToString method correctly returns to representation of a string in the list.

---

2)

Node.java

The Node.java code is easy to understand, With a string data field and a reference to the next node, the Node class creates a straightforward node structure. For the data and next fields, it offers fundamental getter and setter methods. The toString() function is also used to return the data field's string value.

SinglyLinkedList.java:

Sll.java is good in understanding partially but,we can also make it more better by

But, I feel like by doing some improvement we can do it much better by changes like

- ✓ By inserting comments to describe the purpose of the each class would make explanation in it easy to understand.
- ✓ Variables name apart from shortcut letters we can write complete name which will make everyone to understand easily.
- ✓ For setter method need to add validation checks to make sure input values are correct.
- ✓ Also mentioning proper indentation to the code makes it better in recognizing.

---

3)

Node.java:

For Node.java we can still make documentation even better by describing clear picture on what method does, which type of inputs it takes, and the results that give.

We can fix this by adding detailed information on behavior and need to be required in parameters and return values which I make in code that I submit.

SinglyLinkedList.java:

As we can see for SinglyLinkedList class can be done more better documentation than now because there are no proper comments, information about individual function of its input as well output parameters.

So to make better documentation we can follow few steps which I made are

- ✓ To automatically produce documentation, think about utilizing Javadoc-style comments. This is especially useful for huge codebases.
- ✓ To show how to use each function properly, take into account include examples or sample use code.
- ✓ Each function's input and output parameters, as well as any restrictions or prerequisites, should be documented.
- ✓ Provide comments outlining each function's goal, along with any underlying presumptions or limitations.

So by documenting properly,will help developers to possess the usage of each class correctly to over come mistakes or some syntax errors that commonly takes place.

4)

Node.java:

After verifying the Node.java file I don't find errors in it, so I believe node.java is correctly Implemented.

SinglyLinkedList.java:

SinglyLinkedList.java is partially correct because I noticed couple of minor errors listed below to modify the code for a better outcome.

I. **Insert:** In "Insert method" If the current node along with List is not empty, The written code will traverses the contained list to look for last node and insert the new node. this can be improved by maintaining a reference to the list's last node.
II. **Remove:** In "Remove method" when we run we can notice code will throw an "IllegalArgumentException" why because if the current node is null or consists of no next node.anyway it makes more reasonable to throw a "NullPointerException".In case the code node is "null also to do nothing", so If current node consists of other next node.
III. **Find:** In Find method here can add and check to find out, In this case of list, so we return null in case. If the current node I consists null and to do nothing.

Also attached the Revised code by making all required changes which I mentioned above

---

5)

Node.java

Now I try to calculate the cyclomatic complexity of each of the methods for node.java and I get result **7** because it depend on total number of Independent paths in program's control flow graph.

So formula to calculate decisions points like(switch,if statements and loops) + 1

Here, In given code there are 0(zero) Decisions or loops or conditional statements

Therefore, The Cyclomatic complexity of node.java is " **7** "

Note: For node.java there is no revised code so cyclomatic complexity is "**7**"

WMC(node) = " **7** " (Same for revise code the WMC is 7)

SinglyLinkedList.java:

Now first I are going to find Cyclomatic complexity of sll.java for professor give code.

I. Insert → 2
II. Find → 3
III. Remove → 1
IV. Size → 1

V. toString → 2
VI. Head → 1

weighted methods per class for sll.java: Sum of all cyclomatic complexity for sll.java gives a result which is the weighted methods per class.

Here, 2+3+1+1+2+1 ➔10

So **Weighted methods per class** = 10

Secondly, I'm going to calculate cyclmatic complexity of my revised code sll.java

I. Insert → 5
II. Find → 3
III. Remove → 3
IV. Size → 1
V. toString → 2
VI. Head → 1

weighted methods per class for revised sll.java: Sum of all cyclomatic complexity for revised sll.java gives a result which is the weighted methods per class.

Here, 5+3+3+1+2+1 ➔15

So **Weighted methods per** class = 15

---

6)

computational complexity for Node.java

Here the computational complexity for every method in the "Node" class are:

I. Node Constructor → O(1) [Big O of one]
II. Node(String Data) constructor → O(1)
III. getData() → O(1)
IV. setData(String Data) → O(1)
V. getNext()→ O(1)
VI. setNext(Node n) → O(1)

Node.java has constant time complexity because all methods in Node have same Computation complexity which is O(1).

Note: For Node.java there is no Revise code so it also has same computational complexity which is above.

Computational complexity for SinglyLinkedList.java:

Here the computational complexity for every method in the "SinglyLinkedList" class are:

I. Insert → O(1)
II. Find → O(n)
III. Remove → O(1)
IV. Size → O(1)
V. ToString → O(n)
VI. Head() → O(1)

Computational complexity for Revised code of SinglyLinkedList.java:

Here the computational complexity for every method in the "SinglyLinkedList" class are:

I. Insert → O(n)
II. Find → O(n)
III. Remove → O(1)
IV. Size → O(1)
V. ToString → O(n)
VI. Head() → O(1)

Thankyou Professor,TA's
By: Sampath Kumar Medipudi
Mav_id: 1002032901