## Topics

1. Implement Node Class
2. Generics
3. Implement SinglyLinkedList Class
4. Implement Basic Methods of SinglyLinkedList

* isEmpty()
* size()
* first()
* last()
* addFirst()
* addLast()
* removeFirst()

بالطبع! سأقوم بتوضيح الخطوات لحل هذه المشكلة بطريقة واضحة ومفهومة باستخدام لغة الجافا. دعونا نبدأ:

1. **تنفيذ فئة العقدة (Node Class):**
   * يجب أن تحتوي العقدة على متغير لتخزين القيمة ومتغير آخر لتخزين العقدة التالية.
   * يمكن أن يكون لديك شيء مشابه للتالي:
2. public class Node<T> {
3. T value;
4. Node<T> next;
5. public Node(T value) {
6. this.value = value;
7. this.next = null;
8. }
9. }
10. **التعميم (Generics):**
    * يمكنك استخدام التعميم لجعل القائمة متعددة الأنواع.
    * يمكنك تحديد نوع القيمة عند إنشاء قائمة.
11. **تنفيذ فئة قائمة متسلسلة (SinglyLinkedList Class):**
    * يجب أن تحتوي القائمة على متغير لتخزين العقدة الأولى.
    * يمكن أن يكون لديك شيء مشابه للتالي:
12. public class SinglyLinkedList<T> {
13. private Node<T> head;
14. public SinglyLinkedList() {
15. this.head = null;
16. }
17. }
18. **تنفيذ الطرق الأساسية للقائمة المتسلسلة:**
    * هذه الطرق تساعد في إدارة القائمة.
    * إليك بعض الأمثلة:
19. public class SinglyLinkedList<T> {
20. // ... (التعريفات الأخرى)
21. public boolean isEmpty() {
22. return head == null;
23. }
24. public int size() {
25. int count = 0;
26. Node<T> current = head;
27. while (current != null) {
28. count++;
29. current = current.next;
30. }
31. return count;
32. }
33. public T first() {
34. if (head != null) {
35. return head.value;
36. }
37. return null;
38. }
39. public T last() {
40. if (head == null) {
41. return null;
42. }
43. Node<T> current = head;
44. while (current.next != null) {
45. current = current.next;
46. }
47. return current.value;
48. }
49. public void addFirst(T value) {
50. Node<T> new\_node = new Node<>(value);
51. new\_node.next = head;
52. head = new\_node;
53. }
54. public void addLast(T value) {
55. Node<T> new\_node = new Node<>(value);
56. if (head == null) {
57. head = new\_node;
58. } else {
59. Node<T> current = head;
60. while (current.next != null) {
61. current = current.next;
62. }
63. current.next = new\_node;
64. }
65. }
66. public void removeFirst() {
67. if (head != null) {
68. head = head.next;
69. }
70. }
71. }

هذه هي الخطوات الأساسية لتنفيذ قائمة متسلسلة بطريقة واضحة ومفهومة باستخدام لغة الجافا. أتمنى أن يكون هذا مفيدًا!

## Homework

1. develop an implementation of the equals method in the context of the SinglyLinkedList class.

public class SinglyLinkedList<T> {

private Node<T> head;

// ... other methods ...

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

SinglyLinkedList<?> otherList = (SinglyLinkedList<?>) obj;

Node<T> currentNode = head;

Node<?> otherNode = otherList.head;

while (currentNode != null && otherNode != null) {

if (!currentNode.data.equals(otherNode.data)) {

return false;

}

currentNode = currentNode.next;

otherNode = otherNode.next;

}

return currentNode == null && otherNode == null;

}

private static class Node<T> {

private T data;

private Node<T> next;

public Node(T data) {

this.data = data;

}

}

}

In this implementation, we override the equals method inherited from the Object class. The method first checks if the compared objects are the same reference, which would mean they are equal. Then it checks if the compared object is a SinglyLinkedList and proceeds to compare the elements of the lists.

We iterate through both lists simultaneously, comparing each element. If any pair of elements is not equal, we return false. If we reach the end of both lists, we return true, indicating that the lists are equal.

Note that this implementation assumes that the equals method is correctly implemented for the Node class or the element type T. If T is a custom class, you should ensure that it overrides the equals method appropriately.

1. Give an algorithm for finding the second-to-last node in a singly linked list in which the last node is indicated by a null next reference.

To find the second-to-last node in a singly linked list, you can use the following algorithm:

1. Initialize two pointers, current and previous, pointing to the head of the linked list.
2. Traverse the linked list until current.next becomes null, indicating the last node has been reached.
3. At each step, update the previous pointer to point to the current node, and move the current pointer to the next node.
4. Once the loop ends, the previous pointer will be pointing to the second-to-last node in the linked list.

Here's the algorithm implemented in Java:

java

Copy

public Node<T> findSecondToLastNode(Node<T> head) {

if (head == null || head.next == null) {

return null; // Empty list or only one node

}

Node<T> current = head;

Node<T> previous = null;

while (current.next != null) {

previous = current;

current = current.next;

}

return previous;

}

In this algorithm, Node<T> represents the node structure of the singly linked list, where T is the element type stored in each node.

This algorithm has a time complexity of O(n), where n is the number of nodes in the linked list. It iterates through the entire list until the last node is reached.

1. Give an implementation of the size( ) method for the SingularlyLinkedList class, assuming that we did not maintain size as an instance variable.

If the SinglyLinkedList class does not maintain a size variable as an instance variable, you can implement the size() method by traversing the entire linked list and counting the number of nodes. Here's an example implementation:

java

Copy

public class SinglyLinkedList<T> {

private Node<T> head;

// ... other methods ...

public int size() {

int count = 0;

Node<T> current = head;

while (current != null) {

count++;

current = current.next;

}

return count;

}

private static class Node<T> {

private T data;

private Node<T> next;

public Node(T data) {

this.data = data;

}

}

}

In this implementation, the size() method initializes a counter variable count to 0 and a current variable to the head of the linked list. It then iterates through the linked list, incrementing the counter for each node encountered. Once the end of the list is reached (current becomes null), the method returns the final count.

This implementation has a time complexity of O(n), where n is the number of nodes in the linked list, as it traverses the entire list to count the nodes.

1. Implement a rotate( ) method in the SinglyLinkedList class, which has semantics equal to addLast(removeFirst( )), yet without creating any new node.

To implement a rotate() method in the SinglyLinkedList class that has semantics equal to addLast(removeFirst()) without creating any new nodes, you can follow these steps:

1. Check if the linked list is empty or has only one node. If so, there is no need to rotate, and you can simply return.
2. Create a temporary variable temp and set it to the second node in the linked list (i.e., head.next).
3. Traverse the linked list until you reach the last node.
4. Set the next pointer of the last node to the current head node.
5. Set the next pointer of the node before the last node to null (to make it the new last node).
6. Update the head pointer to point to the node stored in the temp variable.
7. Finally, return the rotated linked list.

Here's an example implementation of the rotate() method:

java

Copy

public class SinglyLinkedList<T> {

private Node<T> head;

// ... other methods ...

public void rotate() {

if (head == null || head.next == null) {

return; // Nothing to rotate

}

Node<T> temp = head.next;

Node<T> current = head;

while (current.next != null) {

current = current.next;

}

current.next = head;

head = temp;

current.next = null;

}

private static class Node<T> {

private T data;

private Node<T> next;

public Node(T data) {

this.data = data;

}

}

}

In this implementation, the rotate() method first checks if the linked list is empty or has only one node. If so, there is no need to rotate, and the method returns early.

Otherwise, it creates a temporary variable temp and sets it to the second node in the linked list (head.next). Then, it iterates through the linked list to find the last node. Once the last node is found, it updates the pointers to perform the rotation: the next pointer of the last node is set to the current head node, the next pointer of the node before the last node is set to null, and finally, the head pointer is updated to point to the node stored in the temp variable.

Note that this implementation does not create any new nodes and performs the rotation in-place.

1. Describe an algorithm for concatenating two singly linked lists L and M, into a single list L′ that contains all the nodes of L followed by all the nodes of M.

To concatenate two singly linked lists L and M into a single list L' that contains all the nodes of L followed by all the nodes of M, you can follow these steps:

1. Check if either L or M is empty. If one of them is empty, simply return the other list as the concatenated list.
2. Traverse the first linked list L until you reach its last node.
3. Set the next pointer of the last node of L to the head of the second linked list M.
4. If M is not empty, update the head of L' to the head of L. Otherwise, L' will be the same as L.
5. Return L' as the concatenated list.

Here's a high-level description of the algorithm:

xquery

Copy

concatenate(L, M):

if L is empty:

return M

if M is empty:

return L

lastNode = L.head

while lastNode.next is not null:

lastNode = lastNode.next

lastNode.next = M.head

if M is not empty:

L' = L

else:

L' = L

return L'

Note that this algorithm assumes that the linked lists are implemented using a head pointer that references the first node of the list.

Implementing this algorithm in a specific programming language would require defining the SinglyLinkedList class and its associated methods.

1. Describe in detail an algorithm for reversing a singly linked list L using only a constant amount of additional space.

To reverse a singly linked list L using only a constant amount of additional space, you can follow these steps:

1. Initialize three pointers: current, previous, and next. Set current to the head of the linked list, previous to null, and next to null.
2. Traverse the linked list iteratively until the current pointer becomes null.
3. At each step, perform the following operations:
   * Set next to the next node of the current node.
   * Set the next pointer of the current node to the previous node (reversing the link).
   * Update previous to the current node.
   * Move current to the next node.
4. After the traversal, the previous pointer will be pointing to the last node of the original linked list, which will become the new head of the reversed list.
5. Update the head pointer of the linked list to the node pointed to by previous.
6. Return the reversed linked list.

Here's a high-level description of the algorithm:

ini

Copy

reverse(L):

current = L.head

previous = null

next = null

while current is not null:

next = current.next

current.next = previous

previous = current

current = next

L.head = previous

return L

Note that this algorithm only uses three pointers (current, previous, and next) and a constant amount of additional space to reverse the linked list in-place. The time complexity of this algorithm is O(n), where n is the number of nodes in the linked list, as it iterates through the entire list once.