## Topics

1. Implement Node Class
2. Implement DoublyLinkedList Class
3. Implement Basic Methods of DoublyLinkedList

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

Sure! Here's an implementation of the Node class for a DoublyLinkedList:

java

Copy

class Node<T> {

T data;

Node<T> prev;

Node<T> next;

public Node(T data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

In this implementation, the Node class represents a node in the DoublyLinkedList. It has three instance variables: data to store the data of the node, prev to reference the previous node, and next to reference the next node.

Now, let's implement the DoublyLinkedList class with the basic methods:

java

Copy

class DoublyLinkedList<T> {

private Node<T> head;

private Node<T> tail;

private int size;

public DoublyLinkedList() {

head = null;

tail = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public T first() {

if (isEmpty()) {

return null;

}

return head.data;

}

public T last() {

if (isEmpty()) {

return null;

}

return tail.data;

}

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

size++;

}

public void addLast(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

head = newNode;

tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

size++;

}

public T removeFirst() {

if (isEmpty()) {

return null;

}

T removedData = head.data;

if (head == tail) {

head = null;

tail = null;

} else {

head = head.next;

head.prev = null;

}

size--;

return removedData;

}

public T removeLast() {

if (isEmpty()) {

return null;

}

T removedData = tail.data;

if (head == tail) {

head = null;

tail = null;

} else {

tail = tail.prev;

tail.next = null;

}

size--;

return removedData;

}

}

In this implementation, the DoublyLinkedList class has instance variables head and tail to reference the first and last nodes of the list, respectively. The size variable keeps track of the number of nodes in the list.

The methods implemented are as follows:

* isEmpty(): Checks if the list is empty.
* size(): Returns the size of the list.
* first(): Returns the data of the first node in the list.
* last(): Returns the data of the last node in the list.
* addFirst(T data): Adds a new node with the given data at the beginning of the list.
* addLast(T data): Adds a new node with the given data at the end of the list.
* removeFirst(): Removes and returns the data of the first node in the list.
* removeLast(): Removes and returns the data of the last node in the list.

Note that in each method, the size of the list is appropriately updated.

## Homework

1. Describe a method for finding the middle node of a doubly linked list with header and trailer sentinels by “link hopping,” and without relying on explicit knowledge of the size of the list. In the case of an even number of nodes, report the node slightly left of center as the “middle.”

To find the middle node of a doubly linked list with header and trailer sentinels without relying on explicit knowledge of the size of the list, you can use the "link hopping" approach. Here's a method that describes this process:

1. Initialize two pointers, slow and fast, to the sentinel node (header.next).
2. Move the fast pointer two steps ahead and the slow pointer one step ahead.
3. Repeat step 2 until the fast pointer reaches the trailer sentinel node (trailer), or the next node of the fast pointer becomes the trailer sentinel node.
4. At this point, the slow pointer will be pointing to the middle node of the list.
5. If the number of nodes in the list is even, the node slightly left of the center is considered the middle node. To find it, move the slow pointer one step back.
6. Return the node pointed to by the slow pointer as the middle node.

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

stylus

Copy

findMiddle():

slow = header.next

fast = header.next

while fast != trailer and fast.next != trailer:

fast = fast.next.next

slow = slow.next

// If the number of nodes is even, move the slow pointer one step back

if fast == trailer:

slow = slow.prev

return slow

In this algorithm, the slow pointer moves one step at a time, while the fast pointer moves two steps at a time. By the time the fast pointer reaches the end of the list, the slow pointer will be at the middle node.

If the number of nodes is even, the fast pointer will reach the trailer sentinel node, and the slow pointer will be slightly ahead of the center. In that case, we move the slow pointer one step back to return the node slightly left of the center as the middle node.

Note that this algorithm only requires two pointers and doesn't rely on the explicit knowledge of the size of the list. The time complexity of this algorithm is O(n), where n is the number of nodes in the list, as it traverses the list once.

Give an implementation of the size( ) method for the DoublyLinkedList class, assuming that we did not maintain size as an instance variable. If the size of the DoublyLinkedList is not maintained as an instance variable, you can implement the size() method by traversing the list and counting the number of nodes. Here's an implementation:

java

Copy

public int size() {

int count = 0;

Node<T> current = header.next; // Start from the first node after the header sentinel

while (current != trailer) {

count++;

current = current.next;

}

return count;

}

In this implementation, we initialize a count variable to 0 and start traversing the list from the first node after the header sentinel. We iterate through the list by moving the current pointer to the next node until we reach the trailer sentinel. For each node encountered, we increment the count by 1.

Finally, we return the count as the size of the DoublyLinkedList.

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

1. Implement the equals( ) method for the DoublyLinkedList class.

To implement the equals() method for the DoublyLinkedList class, we need to compare the elements of two lists and check if they have the same values and the same order. Here's an implementation:

java

Copy

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

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

return false;

}

DoublyLinkedList<T> otherList = (DoublyLinkedList<T>) obj;

if (size() != otherList.size()) {

return false;

}

Node<T> currentNode = header.next;

Node<T> otherNode = otherList.header.next;

while (currentNode != trailer) {

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

return false;

}

currentNode = currentNode.next;

otherNode = otherNode.next;

}

return true;

}

In this implementation, we first perform several checks to ensure that the provided object is a DoublyLinkedList and has the same size as the current list. If any of these checks fails, we can immediately determine that the lists are not equal.

Next, we create two pointers, currentNode for the current list and otherNode for the other list, both starting from the first node after the respective header sentinels. We iterate through the lists, comparing the data of each corresponding node. If at any point the data is not equal, we return false.

If we reach the end of the lists without finding any unequal elements, we can conclude that the lists are equal and return true.

Note that for the equals() method, we override the default implementation of Object.equals() to provide our own logic for comparing two DoublyLinkedList objects.

1. Give an algorithm for concatenating two doubly linked lists L and M, with header and trailer sentinel nodes, into a single list L′.

To concatenate two doubly linked lists L and M into a single list L', you can follow the algorithm below:

1. Create a new doubly linked list L' with header and trailer sentinel nodes.
2. Set L'.header.next to L.header.next.
3. Set L.header.next.prev to L'.header.
4. Set L'.trailer.prev to L.trailer.prev.
5. Set L.trailer.prev.next to L'.trailer.
6. Set L.header.next and L.trailer.prev to null to disconnect L from L'.
7. Set L'.trailer.next to M.header.next.
8. Set M.header.next.prev to L'.trailer.
9. Set L'.trailer to M.trailer.
10. Set M.header.next and M.trailer.prev to null to disconnect M from L'.

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

stylus

Copy

concatenateLists(L, M):

L' = new DoublyLinkedList()

L'.header.next = L.header.next

L.header.next.prev = L'.header

L'.trailer.prev = L.trailer.prev

L.trailer.prev.next = L'.trailer

L.header.next = null

L.trailer.prev = null

L'.trailer.next = M.header.next

M.header.next.prev = L'.trailer

L'.trailer = M.trailer

M.header.next = null

M.trailer.prev = null

return L'

This algorithm creates a new list L' and connects it to the nodes of L and M, forming a single concatenated list. It disconnects L and M from L' by setting their header.next and trailer.prev pointers to null.

The time complexity of this algorithm is O(1) as it performs a constant number of operations regardless of the size of the lists L and M.

1. Our implementation of a doubly linked list relies on two sentinel nodes, header and trailer, but a single sentinel node that guards both ends of the list should suffice. Reimplement the DoublyLinkedList class using only one sentinel node.
2. Certainly! Here's an implementation of the DoublyLinkedList class using only one sentinel node that guards both ends of the list:
3. java
4. Copy
5. public class DoublyLinkedList<T> {
6. private Node<T> sentinel;
7. private static class Node<T> {
8. T data;
9. Node<T> prev;
10. Node<T> next;
11. Node(T data) {
12. this.data = data;
13. }
14. }
15. public DoublyLinkedList() {
16. sentinel = new Node<>(null);
17. sentinel.prev = sentinel;
18. sentinel.next = sentinel;
19. }
20. public void addFirst(T data) {
21. Node<T> newNode = new Node<>(data);
22. newNode.next = sentinel.next;
23. newNode.prev = sentinel;
24. sentinel.next.prev = newNode;
25. sentinel.next = newNode;
26. }
27. public void addLast(T data) {
28. Node<T> newNode = new Node<>(data);
29. newNode.next = sentinel;
30. newNode.prev = sentinel.prev;
31. sentinel.prev.next = newNode;
32. sentinel.prev = newNode;
33. }
34. public void removeFirst() {
35. if (isEmpty()) {
36. throw new NoSuchElementException("The list is empty.");
37. }
38. sentinel.next = sentinel.next.next;
39. sentinel.next.prev = sentinel;
40. }
41. public void removeLast() {
42. if (isEmpty()) {
43. throw new NoSuchElementException("The list is empty.");
44. }
45. sentinel.prev = sentinel.prev.prev;
46. sentinel.prev.next = sentinel;
47. }
48. public boolean isEmpty() {
49. return sentinel.next == sentinel;
50. }
51. // Other methods such as getFirst(), getLast(), size(), etc.
52. // ...
53. }
54. In this implementation, the DoublyLinkedList class consists of a private static inner class Node, which represents the individual nodes of the list. The DoublyLinkedList class itself maintains a single sentinel node that guards both ends of the list.
55. The sentinel node is initialized in the constructor of the DoublyLinkedList class. Its prev and next pointers are set to point to itself, indicating an empty list.
56. The addFirst() method adds a new node with the given data at the beginning of the list, while the addLast() method adds a new node with the given data at the end of the list. These methods correctly update the prev and next pointers of the nodes to maintain the doubly linked structure.
57. The removeFirst() and removeLast() methods remove the first and last nodes from the list, respectively. These methods adjust the prev and next pointers of the nodes to properly unlink the removed nodes from the list.
58. The isEmpty() method checks if the list is empty by verifying if the sentinel's next pointer points to the sentinel node itself.
59. Note that this implementation uses a singly linked list structure between the sentinel node and the actual data nodes. The sentinel node acts as a "dummy" node and facilitates the operations on the list.
60. Share
61. Implement a circular version of a doubly linked list, without any sentinels, that supports all the public behaviors of the original as well as two new update methods, rotate( ) and rotateBackward.

To implement a circular version of a doubly linked list without any sentinels, which supports all the public behaviors of the original doubly linked list class, as well as two new update methods rotate() and rotateBackward(), you can modify the original implementation of the DoublyLinkedList class as follows:

java

Copy

public class CircularDoublyLinkedList<T> {

private Node<T> head;

private int size;

private static class Node<T> {

T data;

Node<T> prev;

Node<T> next;

Node(T data) {

this.data = data;

}

}

public CircularDoublyLinkedList() {

head = null;

size = 0;

}

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

newNode.next = newNode;

newNode.prev = newNode;

head = newNode;

} else {

newNode.next = head;

newNode.prev = head.prev;

head.prev.next = newNode;

head.prev = newNode;

head = newNode;

}

size++;

}

public void addLast(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

newNode.next = newNode;

newNode.prev = newNode;

head = newNode;

} else {

newNode.next = head;

newNode.prev = head.prev;

head.prev.next = newNode;

head.prev = newNode;

}

size++;

}

public void removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("The list is empty.");

}

if (size == 1) {

head = null;

} else {

head.next.prev = head.prev;

head.prev.next = head.next;

head = head.next;

}

size--;

}

public void removeLast() {

if (isEmpty()) {

throw new NoSuchElementException("The list is empty.");

}

if (size == 1) {

head = null;

} else {

head.prev.prev.next = head;

head.prev = head.prev.prev;

}

size--;

}

public void rotate() {

if (!isEmpty()) {

head = head.next;

}

}

public void rotateBackward() {

if (!isEmpty()) {

head = head.prev;

}

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

// Other methods such as getFirst(), getLast(), etc.

// ...

}

In this implementation, the CircularDoublyLinkedList class maintains a head pointer that points to the first node in the list. The size variable keeps track of the number of nodes in the list.

The addFirst() method adds a new node with the given data at the beginning of the list. If the list is empty, the head pointer is set to the new node, and its prev and next pointers are set to itself. Otherwise, the new node is inserted before the head node, and the necessary link adjustments are made.

The addLast() method adds a new node with the given data at the end of the list. If the list is empty, the head pointer is set to the new node, and its prev and next pointers are set to itself. Otherwise, the new node is inserted after the last node (i.e., the node before the head node), and the necessary link adjustments are made.

The removeFirst() and removeLast() methods remove the first and last nodes from the list, respectively. If the list is empty, an exception is thrown. Otherwise, the necessary link adjustments are made to remove the node, and the head pointer is updated if necessary.

The rotate() method rotates the list by moving the head pointer to the next node. If the list is empty, nothing happens.

The rotateBackward() method rotates the list in the opposite direction by moving the head pointer to the previous node. If the list is empty, nothing happens.

The isEmpty() method checks if the list is empty based on the size variable.

The size() method returns the current size of the list.

Note that this circular doubly linked list implementation does not require any sentinel nodes, and the head pointer maintains the circular nature of the list.

1. Implement the clone( ) method for the DoublyLinkedList class.
2. To implement the clone() method for the DoublyLinkedList class, you can create a new instance of the DoublyLinkedList class and iterate through the original list, adding each element to the new list. Here's an implementation:
3. java
4. Copy
5. public DoublyLinkedList<T> clone() {
6. DoublyLinkedList<T> newList = new DoublyLinkedList<>();
7. Node<T> currentNode = header.next;
8. while (currentNode != trailer) {
9. newList.addLast(currentNode.data);
10. currentNode = currentNode.next;
11. }
12. return newList;
13. }
14. In this implementation, the clone() method creates a new instance of the DoublyLinkedList class called newList.
15. It then iterates through the original list, starting from the first node after the header sentinel, and adds each element to the newList using the addLast() method.
16. Finally, the newList is returned, which is a clone of the original list.
17. Note that the clone() method assumes that the elements stored in the list are cloneable or immutable. If the elements are mutable objects that need to be deep-copied, you might need to implement custom cloning logic for the elements themselves.