Recitation 2 Answers

3.11

R-3.11 Give an implementation of the size() method for the DoublyLinkedList class, assuming that we did not maintain size as an instance variable.

Implementation Idea

- Start from the node right after the header sentinel.
- Traverse until you hit the trailer sentinel.
- Count how many “real” nodes you pass.
- Return that count.

```java
public int size() {
    int count = 0;
    Node<E> walk = header.getNext();     // first real node (after header)
    while (walk != trailer) {            // stop before hitting trailer sentinel
        count++;
        walk = walk.getNext();           // advance to next node
    }
    return count;
}
```

Answers for discussion

Answer 3.8

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." What is the running time of this method?

Method

1. Initialize two references:
   - left = first real node (header.getNext())
   - right = last real node (trailer.getPrev())
2. Move both inward at the same pace:
   - left = left.getNext()
   - right = right.getPrev()
3. Stop when left == right (odd case, exact middle) or when right.getNext() == left (even case, crossed in the middle).
4. In the even case, return right (the slightly left-of-center node).

```java
public Node<E> findMiddle() {
    Node<E> left = header.getNext();
    Node<E> right = trailer.getPrev();

    while (left != right && left != right.getNext()) {
        left = left.getNext();
        right = right.getPrev();
    }

    return right;  // if odd: left==right; if even: right is left of center
}
```

Answer 3.16

Implement the equals( ) method for the DoublyLinkedList class.

Check if o is the same object (this == o).

Check if o is an instance of DoublyLinkedList.

Compare sizes (if size is tracked, this is O(1); if not, you'd compute by traversal).

Traverse both lists in parallel from their first real node to the end:

- Compare elements with .equals().
- If any mismatch, return false.

If traversal completes with no mismatches, return true.

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;                    // same reference
    if (!(o instanceof DoublyLinkedList<?>)) return false;

    DoublyLinkedList<?> other = (DoublyLinkedList<?>) o;

    // (optional, if we maintain size as a field)
    if (this.size() != other.size()) return false;

    Node<E> walkA = this.header.getNext();
    Node<?> walkB = other.header.getNext();

    while (walkA != this.trailer && walkB != other.trailer) {
        if (!walkA.getElement().equals(walkB.getElement())) {
            return false; // mismatch found
        }
        walkA = walkA.getNext();
        walkB = walkB.getNext();
    }

    return true; // all elements matched
}
```

Answer 3.26

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

We have:
List L
[headerL] <-> (a1) <-> (a2) <-> ... <-> (ak) <-> [trailerL]

List M
[headerM] <-> (b1) <-> (b2) <-> ... <-> (bm) <-> [trailerM]

We want L′ = L + M:
[headerL] <-> (a1) ... (ak) <-> (b1) ... (bm) <-> [trailerM]

Effectively:

- Keep headerL as the new header.
- Keep trailerM as the new trailer.
- Link trailerL.getPrev() (last real node of L) to headerM.getNext() (first real node of M).
- Update pointers accordingly.

```java
public static <E> DoublyLinkedList<E> concatenate(
        DoublyLinkedList<E> L, DoublyLinkedList<E> M) {

    if (L.isEmpty()) return M;  // edge case: L empty → result is M
    if (M.isEmpty()) return L;  // edge case: M empty → result is L

    DoublyLinkedList<E> Lprime = new DoublyLinkedList<>();

    // 1. set header of L' = header of L
    Lprime.header = L.header;
    // 2. set trailer of L' = trailer of M
    Lprime.trailer = M.trailer;

    // 3. connect last node of L with first node of M
    Node<E> lastL = L.trailer.getPrev();
    Node<E> firstM = M.header.getNext();

    lastL.setNext(firstM);
    firstM.setPrev(lastL);

    // (optional) set size if maintained
    Lprime.size = L.size + M.size;

    return Lprime;
}
```

Answer 7.11

Describe an implementation of the positional list methods addLast and addBefore realized by using only methods in the set {isEmpty, first, last, before, after, addAfter, addFirst}.

1. addLast(e)

Normally, addLast inserts at the end of the list. Since we are not allowed to directly call addLast, we can rely on the following logic:

- If the list is empty, then adding last is the same as addFirst(e).

- Otherwise, we can find the last position with last() and then use addAfter(last(), e).

```
public Position<E> addLast(E e) {
    if (isEmpty()) {
        return addFirst(e);        // if list is empty, add at front
    } else {
        return addAfter(last(), e); // otherwise, add after last element
    }
}
```

2. addBefore(p, e)

Normally, addBefore inserts a new element just before position p. Since we don't have a direct addBefore method, we can simulate it:

- If p is the first position, then adding before is the same as addFirst(e).

- Otherwise, find the position before p using before(p), and insert after that position with addAfter.

```java
public Position<E> addBefore(Position<E> p, E e) {
    if (p == first()) {
        return addFirst(e);            // special case: insert before first
    } else {
        Position<E> prev = before(p);  // position just before p
        return addAfter(prev, e);      // insert after prev (i.e., before p)
    }
}
```

Answer 7.12

Suppose we want to extend the PositionalList abstract data type with a method, indexOf(p), that returns the current index of the element stored at position p. Show how to implement this method using only other methods of the PositionalList interface (not details of our LinkedPositionalList implementation).

Task: indexOf(p)

We want a method that returns the 0-based index of the given position p in the list.

Idea

- Start at the first position.
- Traverse forward with after(cur) until we reach p.
- Count how many steps it takes.

```java
public int indexOf(Position<E> p) {
    int index = 0;
    Position<E> walk = first();         // start at the first position
    while (walk != p) {                 // keep moving forward until we find p
        walk = after(walk);             // move to next position
        index++;
    }
    return index;                       // number of steps is the index
}
```

Answer 7.36

Suppose we want to extend the PositionalList interface to include a method, positionAtIndex(i), that returns the position of the element having index i (or throws an IndexOutOfBoundsException, if warranted). Show how to implement this method, using only existing methods of the PositionalList interface, by traversing the appropriate number of steps from the front of the list.

Idea

- Start at the first position.
- Traverse forward with after(cur) exactly i times.
- If i is out of bounds (negative or ≥ size), throw IndexOutOfBoundsException.
- Since we are not allowed to directly use size(), we can detect out-of-bounds while traversing.

```java
public Position<E> positionAtIndex(int i) {
    if (i < 0) {
        throw new IndexOutOfBoundsException("Negative index: " + i);
    }

    Position<E> walk = first();       // start at the beginning
    int index = 0;

    while (walk != null && index < i) {
        walk = after(walk);           // move forward
        index++;
    }

    if (walk == null) {               // ran past the end
        throw new IndexOutOfBoundsException("Index: " + i);
    }

    return walk;                      // found the position at index i
}
```

Question 7.39

Suppose we want to extend the PositionalList abstract data type with a method, moveToFront(p), that moves the element at position p to the front of a list (if not already there), while keeping the relative order of the remaining elements unchanged. Show how to implement this method using only existing methods of the PositionalList interface (not details of our LinkedPositionalList implementation).

Idea

Since we can't directly "unlink and relink" nodes (that would use implementation details), we can simulate the move with the ADT methods:

1. If p == first(), do nothing (it's already at front).

2. Otherwise:

    - Save the element at p.

- Remove it from the list.
- Insert it again at the front with addFirst(e).

```java
public void moveToFront(Position<E> p) {
    if (p != first()) {                 // only if not already at front
        E elem = p.getElement();        // retrieve element at p
        remove(p);                      // remove element from current position
        addFirst(elem);                 // reinsert at front
    }
}
```