Linked Lists

Based on the notes from David Fernandez-Baca and Steve Kautz
Bryn Mawr College
CS206 Intro to Data Structures

Practice 2: Implementing

Singly-Linked Lists Collection

Goal: To implement Collection using null-terminated, singly-linked lists without dummy nodes.

```java
public class SinglyLinkedCollection<E> extends AbstractCollection<E>{
    private Node head;
    private int size;
    private class Node{
        public E data;
        public Node next;
        public Node(E pData, Node pNext){
            data = pData;
            next = pNext;
        }
    }
    public int size() { return size; }
```
add()

```java
public boolean add(E item) {
    // add an item at the end
    Node temp = new Node(item, null);
    if (head == null) {
        // add to empty list
        head = temp;
    } else {
        // add after tail
        Node current = head;
        while (current.next != null) { current = current.next; }
        current.next = temp;
    }
    ++size;
    return true;
}
```

First attempt: remove()

```java
public boolean remove(Object obj) {
    Node current = head;
    Node previous = null;
    while (current != null) {
        // TODO: if current.data matches obj,
        // unlink current and return true
        // (don't forget special case for head)
        previous = current;
        current = current.next;
    }
    return false;
}
```

To remove an element in a singly linked list, we need a reference to the **predecessor** of the node to be removed. More later…
Iterators

The state of an iterator is represented by three variables:

- **cursor**: Points to the predecessor of the next element. The value of cursor is null if the next element is head, or if the list is empty.

- **canRemove**: A Boolean variable that indicates whether it is legal to invoke remove().

- **previous**: Points to the predecessor of the node to be removed. More precisely:
  - If canRemove is true and cursor != head, then previous points to the predecessor of cursor.
  - If cursor == head, then previous == null.
  - If canRemove is false, then previous == null.

Iterators (cont.)

```java
public Iterator<E> iterator() { return new LinkedIterator(); }

private class LinkedIterator implements Iterator<E> {
    private Node cursor = null;
    private Node previous = null;
    private boolean canRemove = false;
	next() {
        if (!hasNext()) throw new NoSuchElementException();
        // we know size > 0 and either cursor is null
        // or cursor.next is non-null
        if (cursor == null) { // next element to return is head
            previous = null; cursor = head;
        } else { previous = cursor; cursor = cursor.next; }
        canRemove = true;
        return cursor.data;
    }
```
Iterators (cont.)

```java
public void remove() {
    if (!canRemove) throw new IllegalStateException();
    if (previous == null) {
        // removing first element
        head = head.next;
        cursor = null;
    } else {  // removing element at cursor
        previous.next = cursor.next;
        cursor = previous;
    }
    --size;
    canRemove = false;
    previous = null;
}
```

Time Complexity

- Getting an element at a given index
  - Linked list: \(O(n)\)
    - Reason: We have to traverse the list to find the position
  - Array list: \(O(1)\)
- Checking contains(object)
  - Linked list: \(O(n)\)
  - Array list: \(O(n)\)
- Checking size()
  - Linked list: \(O(1)\)
  - Array list: \(O(1)\)
Time Complexity (cont.)

- Adding a new element at the end
  - Linked list: O(1)
  - Array list: O(1): during a sequence of such operations, we may have to resize the array several times. In practice we don’t worry about this, because we can claim that adding n elements to an array list requires time O(n). That is, the amortized cost per add is O(1).

  **Explanation (Optional):** Suppose we start out at size 1 and that n is a multiple of 2, say n is 2^p. Each time we run out of space, we double the capacity. Then, there are resize operations for sizes 1, 2, 4, 8, 16, ..., 2^{p-1}.
  A resize operation on an array of size k takes time O(k). So the total cost of all the resize operations is
  \[ O(1 + 2 + 4 + ... + 2^{p-1}) = O(2^p) = O(n). \]

Time Complexity (cont.)

- Adding or removing at a given position
  - Linked list: O(n) – we have to traverse the list to find the position
  - Array list: O(n) – we have to shift elements to add/remove in the middle of the array

- Removing a given element
  - Linked list: O(n) – we have to find the element
  - Array list: O(n) – we have to find the element, then shift elements to remove, but O(n) + O(n) = O(n)
Time Complexity (cont.)

- Adding or removing an element during iteration
  - Linked list: $O(1)$ – Already have the node, so linking or unlinking is $O(1)$
  - Array list: $O(n)$ – We have to shift elements to add/remove

- Given a list of length $n$, iterate over the list and perform $k$ adds or removes at arbitrary locations:
  - Linked list: $O(n) + O(k) = O(n + k)$ or $O(\max(n, k))$
  - Array list: $O(nk)$ – each of the $k$ operations is potentially $O(n)$

Space Complexity

- Array lists potentially waste some space because of the empty cells. (But remember, each empty cell is just an object reference – it takes up 4 bytes, not the space of the object itself!)
- For many short lists, linked lists may be more space-efficient. On the other hand, linked lists potentially waste space, because each element has to be wrapped in its own node object.