CSE 143

Lecture 7: Linked List Basics

reading: 16.2
References vs. objects

\[ \text{variable} = \text{value}; \]

A \textit{variable} (left side of = ) is an arrow (the base of an arrow)

A \textit{value} (right side of = ) is an object (a box; what an arrow points at)

- For the list at right:
  - \texttt{a.next = value;}
    - means to adjust where \texttt{a} points
  - \texttt{variable = a.next;}
    - means to make \texttt{variable} point at

\begin{tabular}{|c|c|}
\hline
\text{data} & \text{next} \\
\hline
10 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\text{data} & \text{next} \\
\hline
20 & \text{data} \\
\hline
\end{tabular}
Reassigning references

- when you say:
  - `a.next = b.next;`

- you are saying:
  - "Make variable `a.next` refer to the same value as `b.next`."
  - Or, "Make `a.next` point to the same place that `b.next` points."
Linked node question

- Suppose we have a long chain of list nodes:

```
list → [data: 10, next] → [data: 20, next] → ... → [data: 990, next]
```

- We don't know exactly how long the chain is.

- How would we print the data values in all the nodes?
Algorithm pseudocode

- Start at the **front** of the list.
- While (there are more nodes to print):
  - Print the current node's **data**.
  - Go to the **next** node.

- How do we walk through the nodes of the list?

```java
list = list.next; // is this a good idea?
```
Traversing a list?

• One (bad) way to print every value in the list:

```java
while (list != null) {
    System.out.println(list.data);
    list = list.next; // move to next node
}
```

• What's wrong with this approach?
  • (It loses the linked list as it prints it!)
A current reference

- Don't change `list`. Make another variable, and change it.
  - A `ListNode` variable is NOT a `ListNode` object

```java
ListNode current = list;
```

- What happens to the picture above when we write:

```java
current = current.next;
```
Traversing a list correctly

- The correct way to print every value in the list:

  ```java
  ListNode current = list;
  while (current != null) {
      System.out.println(current.data);
      current = current.next;  // move to next node
  }
  ```

- Changing current does not damage the list.
Linked List vs. Array

- Print list values:

```java
ListNode list = ...;

ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```

- Similar to array code:

```java
int[] a = ...;

int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Array Code</th>
<th>Linked List Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go to front of list</td>
<td>int i = 0;</td>
<td>ListNode current = list;</td>
</tr>
<tr>
<td>Test for more elements</td>
<td>i &lt; size</td>
<td>current != null</td>
</tr>
<tr>
<td>Current value</td>
<td>elementData[i]</td>
<td>current.data</td>
</tr>
<tr>
<td>Go to next element</td>
<td>i++;</td>
<td>current = current.next;</td>
</tr>
</tbody>
</table>
Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it

- Java's collection framework describes several ADTs:
  - Queue, List, Collection, Deque, List, Map, Set

- An ADT can be implemented in multiple ways:
  - ArrayList and LinkedList implement List
  - HashSet and TreeSet implement Set
  - LinkedList, ArrayDeque, etc. implement Queue

- The *same* external behavior can be implemented in many different ways, each with pros and cons.
Let's write a collection class named `LinkedIntList`.

- Has the same methods as `ArrayIntList`:
  - `add`, `add`, `get`, `indexOf`, `remove`, `size`, `toString`

- The list is internally implemented as a chain of linked nodes:
  - The `LinkedIntList` keeps a reference to its `front` as a field.
  - `null` is the end of the list; a `null` front signifies an empty list.
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }

    methods go here
}
Implementing `add`

```java
// Adds the given value to the end of the list.
public void add(int value) {
    ...
}
```

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?
Adding to an empty list

- Before adding 20:
  - `front = `[![Node Image](image1)](image1)

- After:
  - `front = `[![Node Image](image2)](image2)
  
    | data | next |
    |------|------|
    | 20   |      |

- We must create a new node and attach it to the list.
The add method, 1st try

// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ...
    }
}


Adding to non-empty list

• Before adding value 20 to end of list:

  - front = [element 0, element 1]
  - element 0: data = 42, next = element 1
  - element 1: data = -3, next = null

• After:

  - front = [element 0, element 1, element 2]
  - element 0: data = 42, next = element 1
  - element 1: data = -3, next = element 2
  - element 2: data = 20, next = null
Don't fall off the edge!

- To add/remove from a list, you must modify the *next* reference of the node *before* the place you want to change.

- Where should `current` be pointing, to add 20 at the end?
- What loop test will stop us at this place in the list?
The `add` method

```java
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
}
```
Implementing `get`

// Returns value in list at given index.
public int get(int index) {
    ...
}

- Exercise: Implement the `get` method.
The `get` method

// Returns value in list at given index.
// Precondition: 0 <= index < size()
public int get(int index) {
    ListNode current = front;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
Implementing `add (2)`

// Inserts the given value at the given index.
public void add(int index, int value) {
    ...
}

- Exercise: Implement the two-parameter `add` method.
The add method (2)

// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
public void add(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    } else {
        // inserting into an existing list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = new ListNode(value, current.next);
    }
}