**Lecture 8: Linked Lists**

- Review of references
  - How many objects are in this picture?
  - How many references are in this picture? (phone #s)
- In section yesterday, you did these little puzzles rearranging nodes
  - But what if the list was really long?
  - You don't want to say list.next.next.....next.....next
  - Also you may not know exactly how long the list is
  - What can we do?
    - Add a loop!
    - That's what we're going to do for most linked list code
- Let's say we have a list containing a chain of linked nodes, and we want to print out the values in the chain, one per line
  - 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
```

  - Code 1:

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

  - Doesn't work b/c we lose the front of the list!
    - So we need a "temporary reference" that moves through the list
    - (draw the picture of "list" moving)
  - Code 2:

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

  - (draw the picture on a 3-element list)
  - Why can we do this when we can't change list?
    - Because a reference is not an object
  - What would the same code look like for an ArrayIntList?

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

for (int i = 0; i < size; i++) {
    System.out.println(elementData[i]);
```

```
}
```

- Some insights:
  - int i = 0; ----------> ListNode current = list;
  - i < size; ----------> current != null;
  - i++; ----------> current = current.next;
  - elementData[i] ----------> current.data
- We can actually write linked list code with a for loop!

```
for (ListNode cur = list; cur != null; cur = cur.next)
    System.out.println(cur.data);
```

- But usually we use while loops

- I think we now have the basics we need to actually build a linked list
  - Remember that the List abstract type has things like “add”, “remove”, “get”....
  - Same external behavior, but now we’re changing the implementation from arrays to linked nodes
  - Again, we’re only going to consider integers - LinkedIntList
  - And will use ListNode to store the data
- Create our LinkedIntList class
  - What fields do we need?
    - The front of the list - “front”
    - The size of the list - “size”
    - The back of the list - “back”
  - But to start with we’re only going to use the front - all the others are not necessary for correctness
  - This field will be PRIVATE, good encapsulation
  - We’ll end up with 2 classes/files - the node and the list
    - We can use public fields in the node class because we will never let clients get access to our nodes - they will only see ints through add, remove, get....
    - If the clients can never modify our state, then those public fields in the node doesn’t matter
    - Client can create their own nodes and modify them, but they can’t corrupt the list’s state
  - Analogy: painting my house
    - Painter 1: I’ll paint the house but you have to carry the paint around
    - Painter 2: I’ll paint the house and you don’t have to touch the paint
    - Painter 3: I’ll paint the house and use special paint cans that won’t get me dirty
    - Painter 2 is best because I don’t get dirty, and I don’t particularly care if the painter gets dirty
- Simple constructor (no args)
  - What do we initialize our field to? How do we represent an “empty” list?
  - Front is null (note we don’t actually need this constructor)

- Simple add at the end
  - Assume we already have a list with three nodes in it (draw a picture)
  - So first we have to get to the end (in order to add there)
    - Start with the code we had before:

```
ListNode current = front;
while (current != null)
    current = current.next;
```

  - (draw the picture)
  - Then we could execute this line of code

```
current = new ListNode(17);
```

  - But wait a sec - we haven't added it properly (draw picture)
    - It's like threading beads onto a necklace - we've dropped the necklace!
    - Or jumping between train cars - jumping off the caboose
  - There are only TWO ways to change the contents of a list
    - Change "front"
    - Change "<something>.next"
  - So to add properly, we must change "<something>.next" - which .next?
    - The last node currently in the list
    - We must "stop one early" - stop at the last node, and then change that node's .next
    - How do we know we're at the last node? it's .next field is null
  - But is this correct?
    - We immediately test current.next != null
    - Could fail if current is null, because asking for "null.next" throws an exception
    - So we need a special front case (this is very common)

```
if (front == null) {
    front = new ListNode(value);
}
```

    - Note that this is the other way to change a list (modifying front)
- How about "size()"? Counting the number of nodes in the list?

```
int size = 0;
ListNode cur = front;
while (cur != null) {
    cur = cur.next;
    size++;
}
return size;
```

- How about get at an index?
  - Well, we start at the beginning
  - Go for “index” number of times
  - And that’s the node we’re looking for
  - Any preconditions?

```
// Precondition: 0 <= index < size()
ListNode cur = front;
for (int i = 0; i < index; i++) {
    cur = cur.next;
}
return cur.data
```

- And now, remove(index)
  - How would you remove (draw the pictures, 3 elements, remove element 2)

```
ListNode cur = front;
for (int i = 0; i < index - 1; i++) {
    cur = cur.next;
}
cur.next = cur.next.next;
```

  - Remember, stop one early
  - How would you remove the front element?

```
front = front.next;
```

- And finally, add(index, value)

```
public void add(int index, int value) {
     if (index == 0) {
          front = new ListNode(value, front);
     } else {
          ListNode current = front;
          for (int i = 0; i < index - 1 ; i++) {
               current = current.next;
          }
          ListNode temp = new ListNode(value, current.next);
          current.next = temp;
     }
}
```

  - (draw the pictures)
- Other things to talk about:
  - throw exceptions
  - set, toString, indexOf, clear
  - Common special cases
    - middle
    - front
    - back
    - empty
  - Stop one early, vs. go all the way - what cases?