Data Structures

Lists II

CS284

Structure of this week's classes

- Change of office hour?
- Double-Linked List
- Excercises on SingleLinkedList
- LeetCode question: Finding the intersection node of two linked lists

List

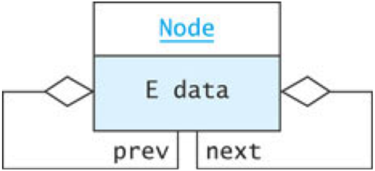
- Last class we introduced lists
- We studied an array based implementation
- We also studied a linked-list based implementation (Single Linked Lists)
- Next we present a double-linked list implementation (Double Linked Lists)
- Also, we present Iterators

Limitations of a single-linked list

- Insertion at the front is O(1); insertion at other positions is O(n)
- Insertion is convenient only after a referenced node
- Removing a node requires a reference to the previous node
- We can traverse the list only in the forward direction
- We can overcome these limitations by:
  - Add a reference in each node to the previous node, creating a double-linked list

Node Class

```
private static class Node<E> {
  private E data;
  private Node<E> next = null;
  private Node<E> prev = null;
  private Node(E dataItem) {
    data = dataItem;
  }
  private Node(E dataItem, Node<E> p, Node<E> n ) {
    data = dataItem;
    prev = p;
    next = n;
  }
}
```



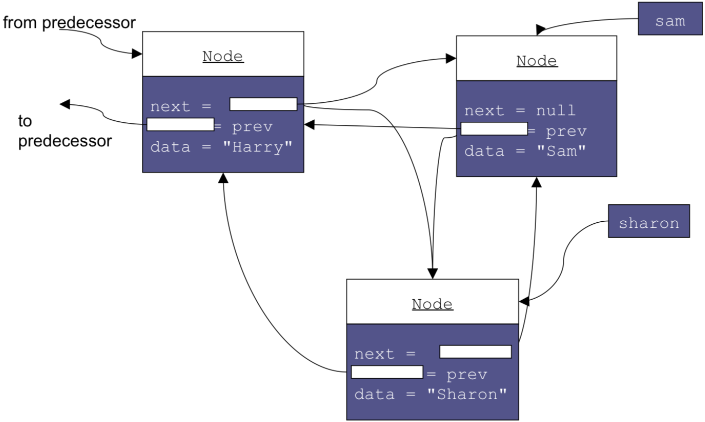
Inserting into a Double-Linked List

```
Node<String> sam = new Node<String>("Sam");
Node<String> harry = new Node<String>("Harry");
harry.next = sam;
sam.prev = harry;
```

- Let's draw a diagram

Inserting into a Double-Linked List

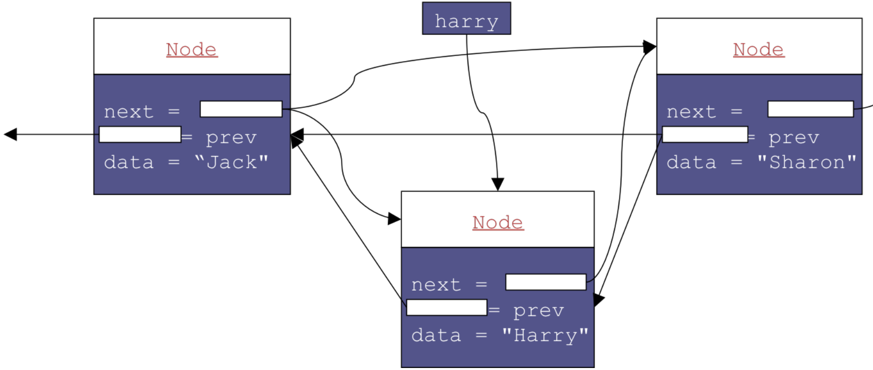
```
Node<String> sharon = new Node<String>("Sharon");
sharon.next = sam;
sharon.prev = sam.prev;
sam.prev.next = sharon;
sam.prev = sharon
```



How do we remove a node?

Consider the execution of the following additional lines

```
harry.prev.next = harry.next
harry.next.prev = harry.prev
```



The class DLList<E>

```
public class DLList<E> {

    private class Node<E> {
          /* As defined above */
          ...
    }
    /** The first element in the list */
    private Node<E> head;
    /** The last element in the list */
    private Node<E> tail;
    /** The size of the list */
    private int size = 0;

  // Operations should follow
}
```

Implement `public void add(E item)`

- This operation should add the item in a new node at the beginning of the list

Double-Linked List

- So far we have worked only with internal nodes
- As with the single-linked class, it is best to access the internal nodes with a double-linked list object
- A double-linked list object has data fields:
  - `head` (a reference to the first list Node)
  - `tail` (a reference to the last list Node)
  - `size`
- Insertion at either end is $\mathcal{O}(1)$; insertion elsewhere is still $\mathcal{O}(n)$

Iterators

- An iterator can be viewed as a moving place marker that keeps track of the current position in a particular linked list
- An Iterator object for a list starts at the list head
- The programmer can move the Iterator by calling its next method
- The Iterator stays on its current list item until it is needed

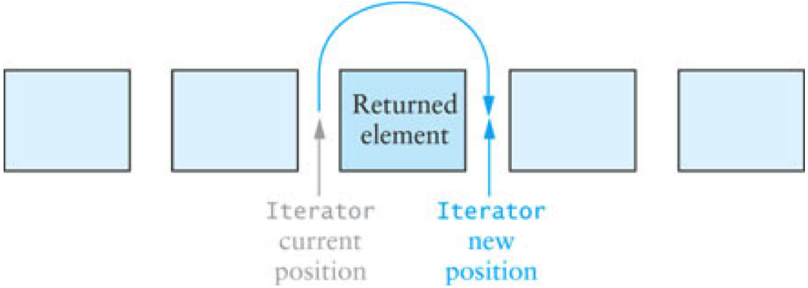
Iterator interface

- The Iterator interface is defined in java.util
- The List interface declares the method `iterator()` which returns an Iterator object that iterates over the elements of that list

Method	Behavior
`boolean hasNext()`	Returns true if the next method returns a value
`E next()`	Returns the next value. If there are now more elements, throws the `NoSuchElementException`
`void remove()`	Removes the last element returned by the next method

Iterator interface

An Iterator is conceptually between elements; it does not refer to a particular object at any given time



Iterator interface

In the following loop, we process all items in `List<Integer>` through an Iterator

```
Iterator<Integer> iter = aList.iterator();
while (iter.hasNext()) {
  int value = iter.next();
  // Do something with value
  ...
}
```

Demo: SingleLinkedList

```java
/**
 * @return a boolean indicating whether all the
 elements in the list are non-null
 */
public SingleLinkedList<E> clone() {
    if (this.head != null) {
        Node<E> previous = new Node<E>(head.data);
        Node<E> head_2 = previous;
        Node<E> node = this.head;
        while (node.next != null) {
            node = node.next;
            Node<E> node_2 = new Node<E>(node.data);
            previous.next = node_2;
            previous = node_2;
        }
        return new SingleLinkedList<E>(head_2);
    }
    else {
        return null;
    }
}
```

Demo: SingleLinkedList

```java
/**
 *
 * @return appends the two lists
 */
public void append(SingleLinkedList<E> l2){
    if (this.head != null) {
        Node<E> node = this.head;
        while (node.next != null) {
            node = node.next;
        }
        node.next = l2.head;
    }
    else {
        this.head = l2.head;
    }
}
```

Demo: SingleLinkedList

```
/**
 * returns a new list in which n copies of the
 original list have been juxtaposed
 * @param n
 */
public void repeatLN(int n){
    SingleLinkedList<E> sll_clone = this.clone();
    for (int i = 0; i < n - 1; i ++)
        this.append(sll_clone);
}
```

Demo: SingleLinkedList

```
/**
 * repeats each element in the list n times
 * @param n
 */
public void stutterNL(int n){
    if (this.head != null) {
        Node<E> node = this.head;
        Node<E> node_next = node.next;
        ArrayList<Node<E>> sub_list = this.sub_copy(node,
        Node<E> all_head = sub_list.get(0);
        Node<E> new_tail = sub_list.get(1);
        while (node_next != null) {
            node = node_next;
            node_next = node.next;
            sub_list = this.sub_copy(node, n);
            Node<E> new_head = sub_list.get(0);
            new_tail.next = new_head;
            new_tail = sub_list.get(1);
        }
        this.head = all_head;
    }
}
```

Demo: SingleLinkedList

```java
    public ArrayList<Node<E>> sub_copy(Node<E> node, int n) {
        Node<E> head = node;
        for (int j = 0; j < n - 1; j ++) {
            Node<E> copy_node = new Node<E>(node.data);
            node.next = copy_node;
            node = copy_node;
        }
        ArrayList<Node<E>> ret_array = new ArrayList<Node<E>>(
        ret_array.add(head);
        ret_array.add(node);
        return ret_array;
    }
```

Demo: SingleLinkedList

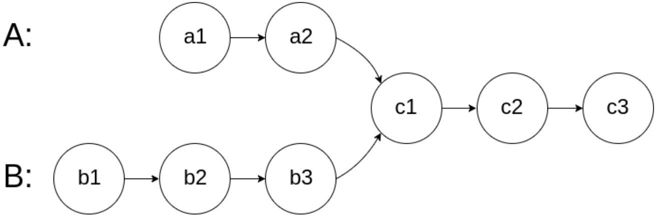
```java
/**
 * removing the adjacent duplicate items
 */
public void removeAdjacentDuplicates() {
    if (this.head != null) {
        Node<E> node = this.head;
        Node<E> new_head = node;
        Node<E> current_node = node;
        E prev_Data = node.data;
        while (node.next != null) {
            node = node.next;
            if (node.data.equals(prev_Data) == false) {
                current_node.next = node;
                current_node = node;
            }
            prev_Data = node.data;
        }
        this.head = new_head;
    }
}
```

Demo: SingleLinkedList

```java
    /**
     * Provide a solution in which a new list is constructed
     * @param l2
     */
    public void zipL(SingleLinkedList<E> l2){
        if (this.head != null) {
            Node<E> node = this.head;
            Node<E> new_head = node;
            Node<E> node_2 = l2.head;
            Node<E> node_next = node.next;
            Node<E> node_next_2 = node_2.next;

            while (node_next != null) {
                node.next = node_2;
                node_2.next = node_next;

                node = node_next;
                node_2 = node_next_2;

                node_next = node.next;
                node_next_2 = node_2.next;
            }
```

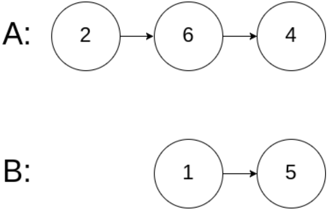
Finding Intersection of Two Lists

Write a program to find the node at which the intersection of two singly linked lists begins.
For example, the following two linked lists:



Finding Intersection of Two Lists (Non-Overlap)

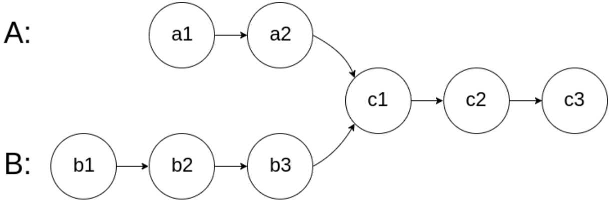
Write a program to find the node at which the intersection of two singly linked lists begins.
For example, the following two linked lists:



Ideas and Thoughts

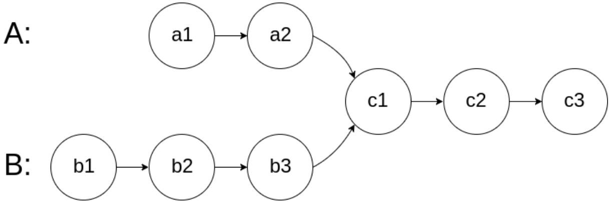
Linear time complexity is necessary:

- The list is only singly linked;
- In the worst case, have to traverse the entire lists to reach the intersection;
- We can traverse the linked list for constant number of times;



Ideas and Thoughts

- The problem with a single linked list is we don't know how far the pointer is to the intersection node;
- Sychronized two pointers: make the pointer the same distance to the intersection, and they will meet;
- Idea: move the pointer for list B for n positions, where $n = B.length - A.length$;



Algorithms

Steps:

- For each list, traverse and get $A.length$ and $B.length$;
- If $A.length > B.length$, move pointer of list A for n positions, where $n = A.length - B.length$;
- Otherwise, move pointer of list B for n positions, where $n = B.length - A.length$;
- Make synchronized move for list A and B;
- If they meet somewhere, return that node; otherwise, return null;

Implementation

Traverse and get the lengths:

```
    /**
     * get the intersection node between list starting
     with head1 and head2
     * @param head1
     * @param head2
     * @return
     */
    public Node<E> getIntersectionNode(Node<E> head1,
    Node<E> head2) {
        if (head1 == null || head2 == null)
            return null;
        Node<E> node1 = head1;
        Node<E> node2 = head2;
        int len1 = len2 = 0;
        while (node1 != null) {
            node1 = node1.next;
            len1 ++;}
        while (node2 != null) {
            node2 = node2.next;
            len2 ++;}
```

Implementation

Shift the pointers:

```
node1 = head1;
node2 = head2;

if (len1 > len2) {
    // shift node1 for len1 - len2 times;
    int steps = len1 - len2;

    for (int i = 0; i < steps; i ++)
        node1 = node1.next;
}
else if (len2 > len1){
    int steps = len2 - len1;

    for (int i = 0; i < steps; i ++)
        node2 = node2.next;
}
```

Implementation

Make synchronized moves:

```
        // now make sync move for node 1 and node2

        while (node1 != null) {
            if (node1.equals(node2) == true)
                return node1;
            node1 = node1.next;
            node2 = node2.next;
        }

        return null;
    }
```