Data Structures and Algorithms

Linked List Fundamentals

Contents

1. List Node

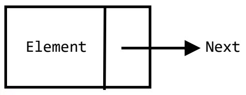
For this example, we would just use a next pointer for ListNode. In future chapters, we will use other pointers to help with other functions for use.

1.1 What's a List Node?

Stores an element, normally a boxed type.

1.2 List Node Construction

```
class ListNode<T> { // make use of Generics!
    // element stored
    public T element;

    // pointer to next node
    public ListNode<T> next;

    public ListNode(T element) {
        this.element = element;
    }

    public ListNode(T element) {
        this.element = element;
    }

    public ListNode(T element, ListNode<T> next) { // remember your overloading?
        this.element = element;
        this.next = next;
    }
```



Exercise 1:

Exercise 1.1: equals()

```
public boolean equals(ListNode<T> other) {
    // compare element and next
}
```

2. Basic Linked List

2.1 Idea behind the Linked List

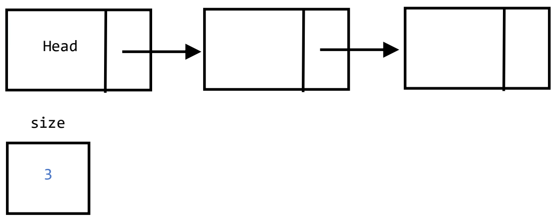
If we were to use an array to store elements: it would be a huge problem.

Array Implementation		List Implementation	
Method	Worst Time Complexity	Method	Worst Time Complextiy
getFirst	O(1)	getFirst	O(1)
addFirst	O(n)	addFirst	O(1)
removeFirst	O(n)	removeFirst	O(1)
contains	O(n)	contains	O(n)

2.2 List Node Construction

1) Just store a head pointer!
2) Store number of elements in your Linked List

```
class BasicLinkedList<T> { // Use Generics!
    // reference to head pointer
    protected ListNode<T> head;

    // reference to number of elements
    protected int size;

    public BasicLinkedList() {
        head = null;
        size = 0;
    }
```



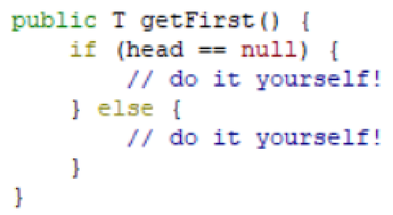
2.3 Linked List Implementation

1) size(): gets the number of elements in the lists
2) isEmpty(): checks whether the list is empty or not

```
public int size() {
    return this.size;
}

public boolean isEmpty() {
    return this.size == 0;
}
```

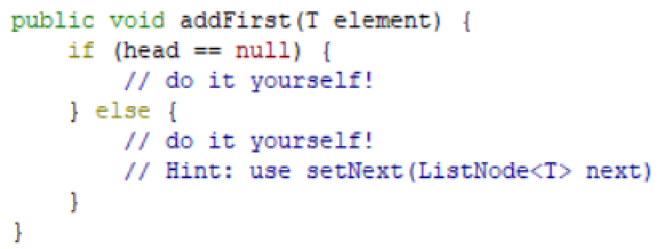
Exercise 2:

Exercise 2.1: getFirst(): returns the element stored in the first node



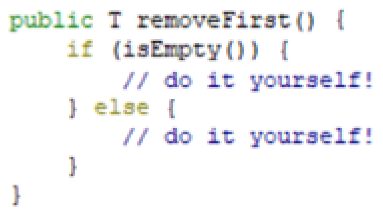
```
public T getFirst() {
    if (head == null) {
        // do it yourself!
    } else {
        // do it yourself!
    }
}
```

Exercise 2.2: addFirst(T element): add the element to the head of the list



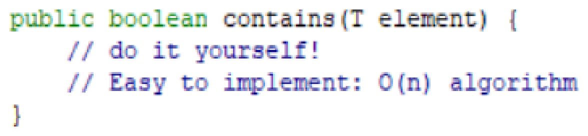
```
public void addFirst(T element) {
    if (head == null) {
        // do it yourself!
    } else {
        // do it yourself!
        // Hint: use setNext(ListNode<T> next)
    }
}
```

Exercise 2.3: removeFirst(): removes the first node in the list, if any



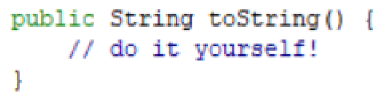
```
public T removeFirst() {
    if (isEmpty()) {
        // do it yourself!
    } else {
        // do it yourself!
    }
}
```

Exercise 2.4: contains(): checks if the List contains the specified element or not.
Hint: transverse through the LinkedList using a pointer.



```
public boolean contains(T element) {
    // do it yourself!
    // Easy to implement: O(n) algorithm
}
```

Exercise 2.5: toString(): returns a String representation of the Linked List.



```
public String toString() {
    // do it yourself!
}
```

3. Extended Linked List

3.1 Can we do better?

We have now understood how to insert and remove at the start of the linked list.
But how do we insert into the middle of the Linked List?
Can we find ways to do better than this?

3.2 Extended Linked List Implementation

1) addAfter(): adds an element after the node in the LinkedList
Note: if node is null, we assume to add to the front of the list
Note: we did not consider if the node exists in the Linked List, try it yourself!

```
public void addAfter(ListNode<T> node, T element) {
    if (node == null) {
        head = new ListNode<T>(element, head);
        size ++;
    } else {
        node.next = new ListNode<T>(element, node.next);
        size ++;
    }
}
```

2) removeAfter(): removes the node after the specified node in the LinkedList
Note: if node is null, we assume to remove from the front of the list
Note: we did not consider if the node exists in the Linked List, try it yourself!

```
public T removeAfter(ListNode<T> node) {
    if (isEmpty()) {
        return null;
    } else if (node == null) {
        if (head == null) {
            return null;
        } else {
            T element = head.element;
            head = head.next;
            size --;
            return element;
        }
    } else {
        if (node.next != null) {
            T element = node.next.element;
            node.next = node.next.next;
            size --;
            return element;
        } else {
            return null;
        }
    }
}
```

Exercise 3:

Exercise 3.1: removes the specified element in the LinkedList

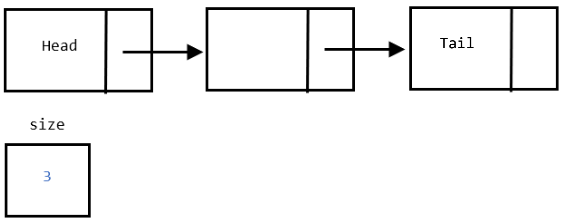
```
public T remove(T element) {
    // do it yourself!
    // Hint: Transverse through the LinekdList and use removeAfter()
    // O(n) algorithm
}
```

4. Tailed Linked List

4.1 Heads and Tails!

Idea: Store a tail pointer.

```
// reference to head pointer
protected ListNode<T> head;

// reference to tail pointer
protected ListNode<T> tail;

// reference to number of elements
protected int size;

public TailedLinkedList() {
    head = null;
    tail = null;
    size = 0;
}
```



4.2 Tailed Linked List Implementation

1) getLast(): get the Last element, if any.

```
public T getLast() {
    if (tail == null) {
        return null;
    } else {
        return tail.element;
    }
}
```

2) addLast(): adds the element to the back of the Linked List

```
public T addLast(T element) {
    size ++;
    if (head == null) {
        tail = new ListNode<T>(element);
        head = tail;
    } else {
        tail.next = new ListNode<T>(element);
        tail = tail.next;
    }
}
```

3) addAfter(): adds the element after the specified Node in the Linked List

Note: if node is null, we assume to add to the front of the list

Note: we did not consider if the node exists in the Linked List, try it yourself!

```
public void addAfter(ListNode<T> node, T element) {
    if (node == null) {
        head = new ListNode<T>(element, node);
        if (tail == null) {
            tail = head;
        }
    } else {
        node.next = new ListNode<T>(element, node.next);
        if (tail.equals(node)) {
            tail = node.next;
        }
    }
}
```

4) removeAfter(): removes the element after the specified Node in the Linked List
Note: if node is null, we assume to remove from the front of the list
Note: we did not consider if the node exists in the Linked List, try it yourself!

```
public T removeAfter(ListNode<T> node) {
    if (isEmpty()) {
        return null;
    } else if (node == null) {
        if (head == null) {
            return null;
        } else {
            T elem = head.element;
            head = head.next;
            size --;
            if (isEmpty()) {
                tail == null;
            }
        }
    } else {
        if (node.next == null) {
            return null;
        } else {
            T elem = node.next.element;
            node.next = node.next.next;
            size --;
            if (node.next == null) {
                tail = node;
            }
        }
    }
}
```

Exercise 4:

Exercise 4.1: addFirst(): adds the element to the front of the Linked List

```
public void addFirst(T element) {
    // do it yourself!
}
```

Exercise 4.2: removeFirst(): removes the first element of the Linked List

```
public T removeFirst(T element) {
    // do it yourself!
}
```

Exercise 4.3: removeLast(): removes the last element of the Linked List

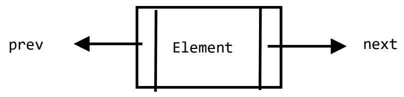
```
public T removeLast() {
    // do it yourself!
}
```

5. Doubly Linked List

5.1 Revisiting List Node

Idea: Your List Node now has a pointer to the previous node.
Reasoning: Makes addition and deletion in the middle of a list easier.

```
// element stored
public T element;

// pointer to previous node
public ListNode<T> prev;

// pointer to next node
public ListNode<T> next;

public ListNode(T element) {
    this.element = element;
}

public ListNode(T element, ListNode<T> prev, ListNode<T> next) { // remember your overloading?
    this.element = element;
    this.prev = prev
    this.next = next;
}
```



Note: You make changes to the equals() method itself too! Do it yourself!

5.2 Doubly Linked List Implementation

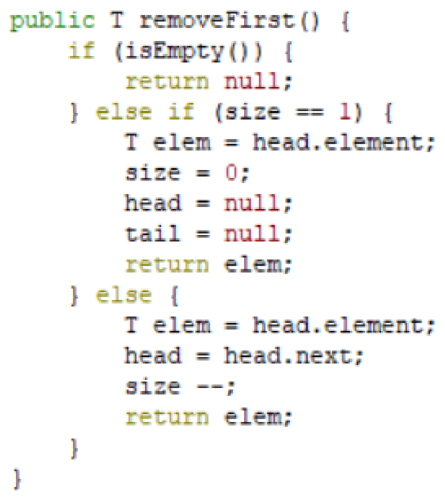
1) addFirst(): adds the element to the front of the Linked List
Note: Consider the case when number of elements in Linked List = 0

```
public void addFirst(T element) {
    if (isEmpty()) {
        head = new ListNode<T>(element);
        tail = head;
        size ++;
    } else {
        ListNode<T> temp = new ListNode<T>(element);
        head.prev = temp; // pointers
        temp.next = head; // pointers
        head = temp;
        size ++;
    }
}
```

2) addLast(): adds the element to the end of the Linked List
Note: Consider the case when number of elements in Linked List = 0

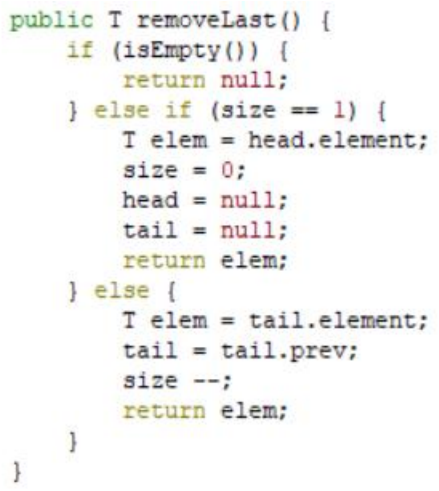
```
public void addLast(T element) {
    if (isEmpty()) {
        head = new ListNode<T>(element);
        tail = head;
        size ++;
    } else {
        ListNode<T> temp = new ListNode<T>(element);
        tail.next = temp; // pointers
        temp.prev = tail; // pointers
        tail = temp;
        size ++;
    }
}
```

3) removeFirst(): removes the first element in the Linked List



```
public T removeFirst() {
    if (isEmpty()) {
        return null;
    } else if (size == 1) {
        T elem = head.element;
        size = 0;
        head = null;
        tail = null;
        return elem;
    } else {
        T elem = head.element;
        head = head.next;
        size --;
        return elem;
    }
}
```

4) removeLast(): removes the last element in the Linked List



```
public T removeLast() {
    if (isEmpty()) {
        return null;
    } else if (size == 1) {
        T elem = head.element;
        size = 0;
        head = null;
        tail = null;
        return elem;
    } else {
        T elem = tail.element;
        tail = tail.prev;
        size --;
        return elem;
    }
}
```

5) removeBefore(): removes the element before the node in the Linked List

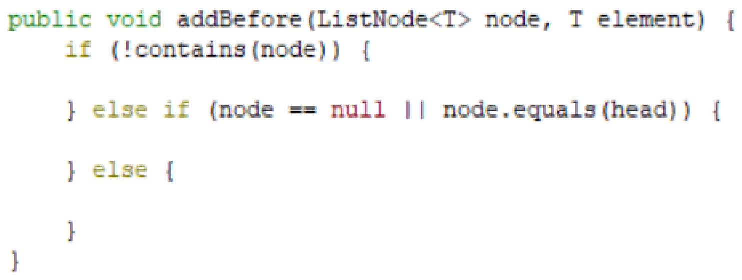
```
public T removeBefore(ListNode<T> node) {
    if (node == null || !contains(node) || node.equals(head)) {
        return null; // do nothing in this case
    } else if (node.equals(head.next)) {
        return removeFirst(); // removing the element before head.next is just removeFirst()
    } else {
        ListNode<T> temp = node.prev;
        ListNode<T> previous = node.prev.prev;
        node.prev = previous;
        previous.next = node;
        size --;
        return temp.element;
    }
}
```

6) removeAfter(): removes the element after the node in the Linked List

```
public T removeAfter(ListNode<T> node) {
    if (node == null || !contains(node) || node.equals(tail)) {
        return null; // do nothing in this case
    } else if (node.equals(tail.prev)) {
        return removeLast(); // removing the element after tail.prev is just removeLast()
    } else {
        ListNode<T> temp = node.next;
        ListNode<T> following = node.next.next;
        node.next = following;
        following.prev = node;
        size --;
        return temp.element;
    }
}
```

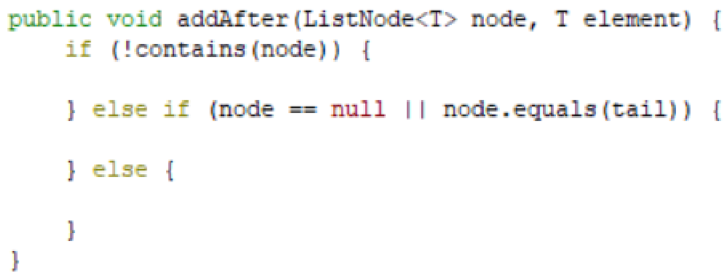
Exercise 5:

Exercise 5.1: addBefore(): adds the element before the specified node in the Linked List



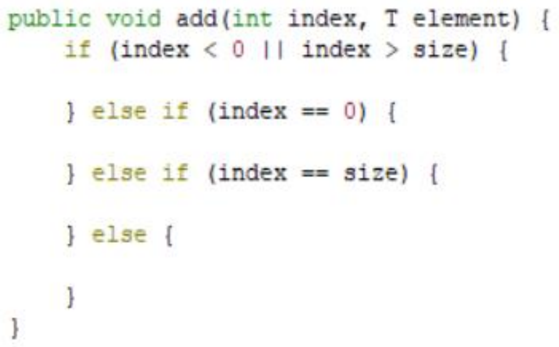
```java
public void addBefore(ListNode<T> node, T element) {
    if (!contains(node)) {

    } else if (node == null || node.equals(head)) {

    } else {

    }
}
```

Exercise 5.2: addAfter(): adds the element after the specified node in the Linked List



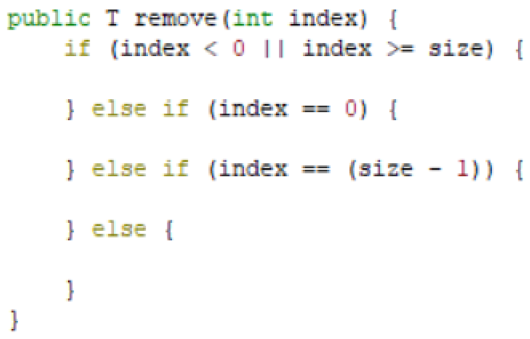
```java
public void addAfter(ListNode<T> node, T element) {
    if (!contains(node)) {

    } else if (node == null || node.equals(tail)) {

    } else {

    }
}
```

Exercise 5.3: add(): adds the element at the index in the Linked List



```java
public void add(int index, T element) {
    if (index < 0 || index > size) {

    } else if (index == 0) {

    } else if (index == size) {

    } else {

    }
}
```

Exercise 5.4: remove(): removes the element at the index in the Linked List

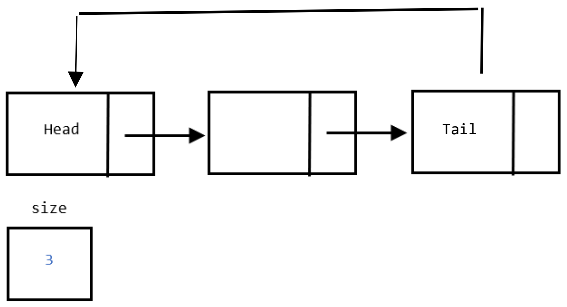


```java
public T remove(int index) {
    if (index < 0 || index >= size) {

    } else if (index == 0) {

    } else if (index == (size - 1)) {

    } else {

    }
}
```

6. Circular Linked List

6.1 Right round round round

```
public void makeCircular() {
    if (tail != null) {
        tail.next = head;
    }
}
```



Note: This method is very useful. By adding this to every method, you will "magically" turn every Linked List to a Circular Linked List.

7. Linked List Java API

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

7.1 How do you make it?

```
import java.util.List;
import java.util.LinkedList;

LinkedList<Integer> myList = new LinkedList<Integer>();
List<Integer> myList2 = new LinkedList<Integer>();
List<Integer> myList3 = new LinkedList< >();
```

7.2 How do you do it?

1) How do you convert a LinkedList to an ArrayList?

```
LinkedList<Integer> linkedList = new LinkedList<Integer>();
List<Integer> arrayList = new ArrayList<Integer>(linkedList);
```

2) How do you convert a LinkedList to an Array?

```
LinkedList<Integer> linkedList = new LinkedList<Integer>();
Integer[] array = linkedList.toArray(new Integer[linkedList.size()]);
```

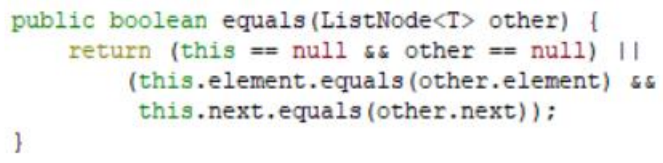
3) How do you iterate through a LinkedList?

Note: You would rarely want to iterate through one, due to O(n) algorithm

```
LinkedList<Integer> linkedList = new LinkedList<Integer>();
Iterator<Integer> iter = linkedList.listIterator();
while (iter.hasNext()) {
    Integer element = iter.next();
}
```

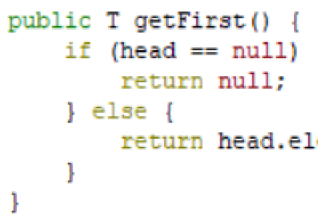
Suggestion Solutions

Exercise 1:

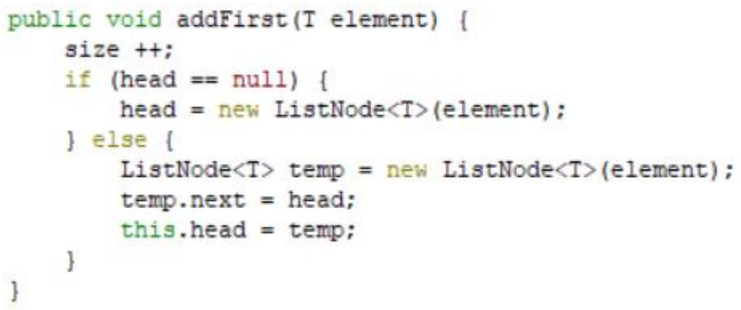


```
public boolean equals(ListNode<T> other) {
    return (this == null && other == null) ||
        (this.element.equals(other.element) &&
         this.next.equals(other.next));
}
```

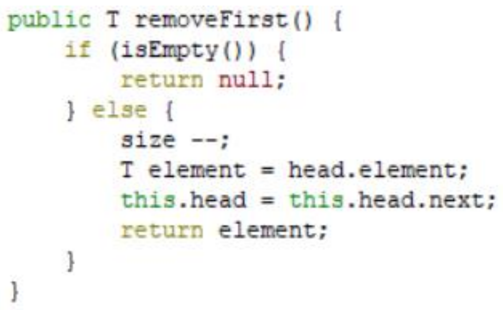
Exercise 2:



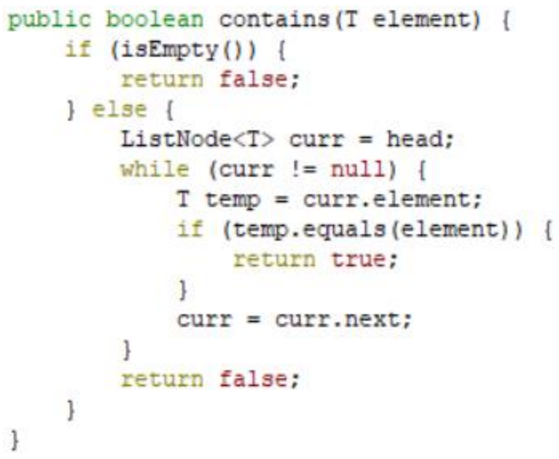
```
public T getFirst() {
    if (head == null) {
        return null;
    } else {
        return head.element;
    }
}
```



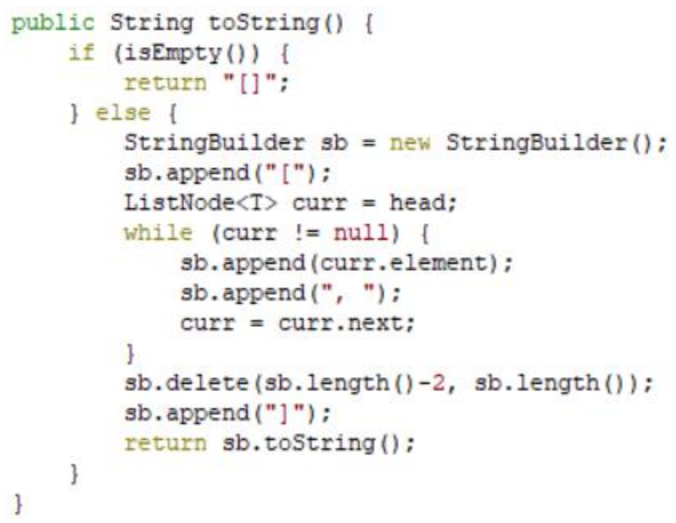
```
public void addFirst(T element) {
    size ++;
    if (head == null) {
        head = new ListNode<T>(element);
    } else {
        ListNode<T> temp = new ListNode<T>(element);
        temp.next = head;
        this.head = temp;
    }
}
```



```
public T removeFirst() {
    if (isEmpty()) {
        return null;
    } else {
        size --;
        T element = head.element;
        this.head = this.head.next;
        return element;
    }
}
```

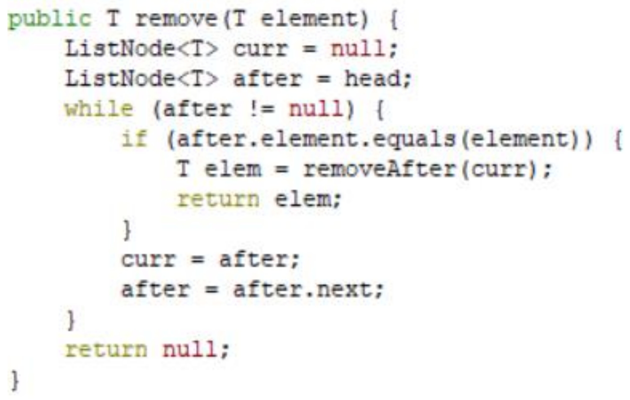


```
public boolean contains(T element) {
    if (isEmpty()) {
        return false;
    } else {
        ListNode<T> curr = head;
        while (curr != null) {
            T temp = curr.element;
            if (temp.equals(element)) {
                return true;
            }
            curr = curr.next;
        }
        return false;
    }
}
```



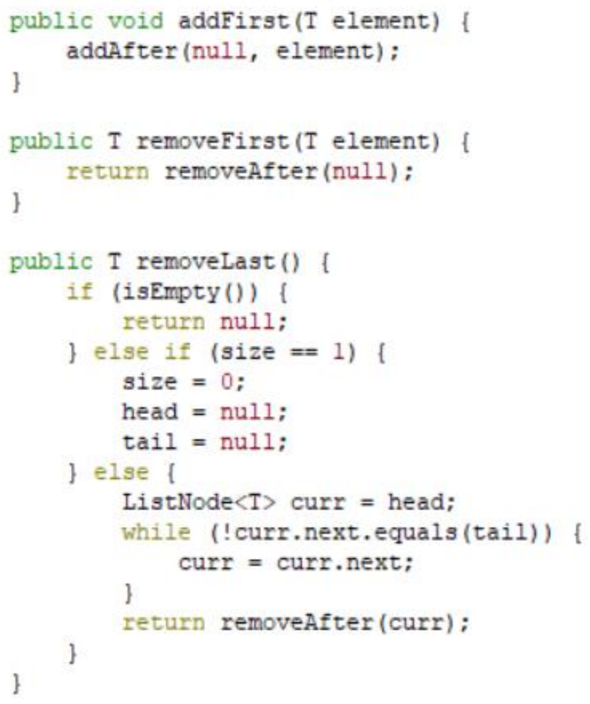
```java
public String toString() {
    if (isEmpty()) {
        return "[]";
    } else {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        ListNode<T> curr = head;
        while (curr != null) {
            sb.append(curr.element);
            sb.append(", ");
            curr = curr.next;
        }
        sb.delete(sb.length()-2, sb.length());
        sb.append("]");
        return sb.toString();
    }
}
```

Exercise 3:



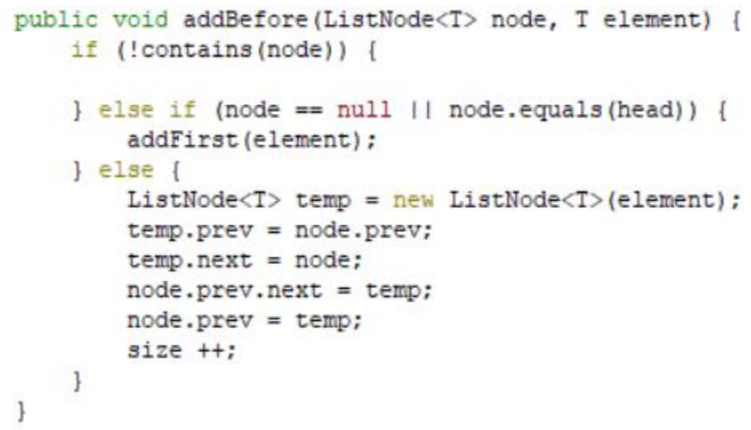
```java
public T remove(T element) {
    ListNode<T> curr = null;
    ListNode<T> after = head;
    while (after != null) {
        if (after.element.equals(element)) {
            T elem = removeAfter(curr);
            return elem;
        }
        curr = after;
        after = after.next;
    }
    return null;
}
```

Exercise 4:

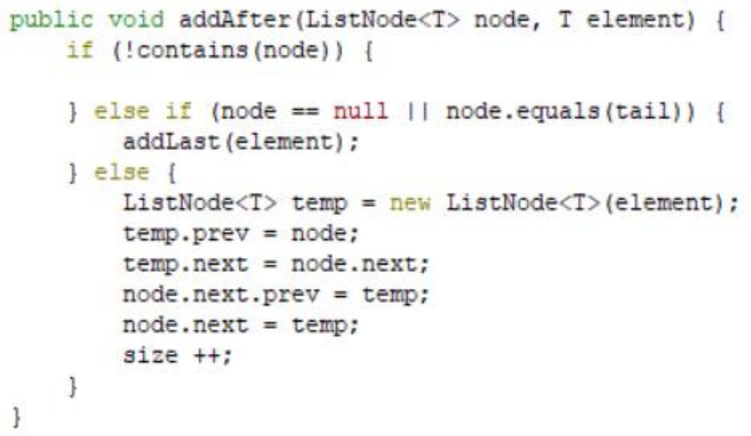


```
public void addFirst(T element) {
    addAfter(null, element);
}

public T removeFirst(T element) {
    return removeAfter(null);
}

public T removeLast() {
    if (isEmpty()) {
        return null;
    } else if (size == 1) {
        size = 0;
        head = null;
        tail = null;
    } else {
        ListNode<T> curr = head;
        while (!curr.next.equals(tail)) {
            curr = curr.next;
        }
        return removeAfter(curr);
    }
}
```

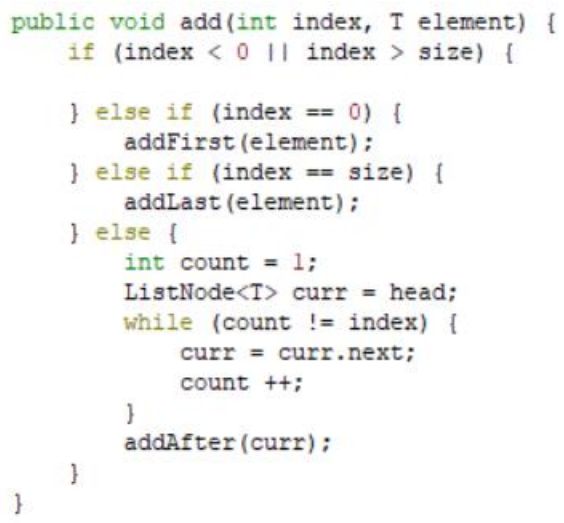
Exercise 5:



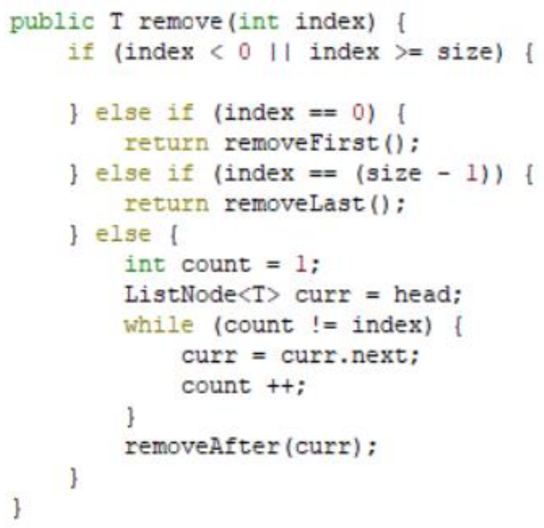
```
public void addBefore(ListNode<T> node, T element) {
    if (!contains(node)) {

    } else if (node == null || node.equals(head)) {
        addFirst(element);
    } else {
        ListNode<T> temp = new ListNode<T>(element);
        temp.prev = node.prev;
        temp.next = node;
        node.prev.next = temp;
        node.prev = temp;
        size ++;
    }
}
```



```
public void addAfter(ListNode<T> node, T element) {
    if (!contains(node)) {

    } else if (node == null || node.equals(tail)) {
        addLast(element);
    } else {
        ListNode<T> temp = new ListNode<T>(element);
        temp.prev = node;
        temp.next = node.next;
        node.next.prev = temp;
        node.next = temp;
        size ++;
    }
}
```



```java
public void add(int index, T element) {
    if (index < 0 || index > size) {

    } else if (index == 0) {
        addFirst(element);
    } else if (index == size) {
        addLast(element);
    } else {
        int count = 1;
        ListNode<T> curr = head;
        while (count != index) {
            curr = curr.next;
            count ++;
        }
        addAfter(curr);
    }
}
```



```java
public T remove(int index) {
    if (index < 0 || index >= size) {

    } else if (index == 0) {
        return removeFirst();
    } else if (index == (size - 1)) {
        return removeLast();
    } else {
        int count = 1;
        ListNode<T> curr = head;
        while (count != index) {
            curr = curr.next;
            count ++;
        }
        removeAfter(curr);
    }
}
```