

# CS 608 Algorithms and Computing Theory

## Week 3: Linked Lists

---

### 1. Abstract Data Types

A set of data values and associated operations that are precisely specified independent of any particular implementation. The abstract data type (ADT) itself refers to this model, not any particular implementation in any particular programming language or paradigm.

We will see several ADTs in this course. This week will focus on an ADT linked list.

---

### 2. Linked lists

We normally use an array to keep a collection of data elements in a program. An example of a collection of data elements is integers. An array takes up one block of memory to hold the elements. The elements of an array are accessed very easily using subscripts.

**An array:**

elements	23	55	17	22	20	45	12	89	34	22	43	21	18	6
subscripts	0	1	2	3	4	5	6	7	8	9	10	11	12	13

**Advantages of using an array:**

- Simple to create and use.
- Accessing elements is fast (constant time).
- Entire collection of data is stores with one name (array name).
- Multidimensional arrays can be defined and manipulated easily.

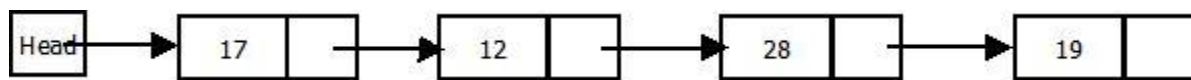
**Disadvantages of using an array:**

- The size of the array is predetermined. The fixed size of an array is a big problem if you don't know how elements are expected. The size of an array cannot change during the execution of the program.
- Certain operations, such as inserting an element, deleting an element, are very complex because these involve movement of data within array.

**Linked list:**

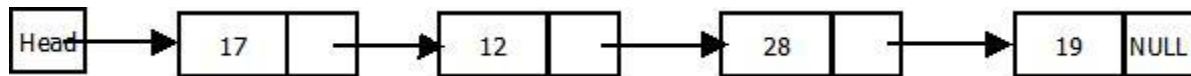
A linked list is a data structure used for storing a collection of data elements. A linked list eliminates the disadvantages in using an array we discussed above.

This is the basic idea of a linked list: Elements in a linked list are stored in boxes with two compartments, one to hold the actual data and the other to hold a pointer to the box holding the next element. These boxes are called nodes.

**Example:**

This linked list has four nodes. Head points to the first node of the linked list. The first node holds element 17 and points to the next node as indicated by the arrow. The next holds element 12 and points to the next node, and so on.

The last node holds element 19 and does not point to the next node because it is the last node. To indicate the last node we place a “null pointer constant” NULL as shown below:



We will see shortly how to implement a linked list in Java. Once we implement a linked list, we will also write methods for certain operations, such as insert an element, delete an element, on linked list. This is the linked list ADT.

---

### 3. Implementation of a linked list

Let us implement a linked list which stores integers. It can be easily generalized to any data type.

Different programmers implement different ways. Ours is a simple implementation, which is easy to follow.

Our linked list is a class and a node is an inner class of the linked list class.

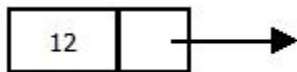
### The Node class:

The Node class has two fields: data and next. Data is the element to be stored in the node and next is a pointer to the next node.

The Node class has one constructor with two arguments (int and Node). The constructor initializes a newly created Node object.

```
private static class Node {  
    private int data;  
    private Node next;  
  
    public Node(int data, Node next)  
    {  
        this.data = data;  
        this.next = next;  
    }  
}
```

In a diagram a Node looks like this:



---

### myLinkedList class:

These are the highlights of myLinkedList class:

One Node field:

```
private Node head;
```

This field keeps the pointer to the first node of the linked list.

➤ A Constructor, which creates an empty linked list:

```
public myLinkedList()  
{  
    head = null;  
}
```



Head

This indicates that the linked list is empty.

---

- A method **isEmpty()**, returns true if the linked list is empty otherwise returns false:

```
public boolean isEmpty()
{
    return (head == null);
}
```

---

- A method **insertAtBeginning(element)**, which creates a new node and inserts the given element at the beginning of the linked list:

```
public void insertAtBeginning(int element)
{
    head = new Node(item, head);
}
```

---

- A method **getFirstElement()**, which returns the first element in the linked list. It does not remove the node from the linked list.

```
public int getFirstElement()
{
    if(head == null) {
        System.out.println("Empty linked list");
        throw new IndexOutOfBoundsException();
    }

    return head.data;
}
```

Notice that the method checks if the linked list is empty before trying to retrieve the element.

---

- A method **removeFirstNode()**, which (1) retrieves the first element and returns it and (2) points head to the next node. So in effect, the former first node is gone.

```
public int removeFirstNode()
{
    int tmp = getFirstElement();
    head = head.next;
    return tmp;
}
```

Let me explain the code using a diagram:

Original linked list:



`int tmp = getFirstElement();` This assigns 17 to tmp variable.

`head = head.next;` This is important to understand. The next value in head is assigned to head. This is what is happening:

Original linked list:



**head** is pointing to the node (this means head holds the address of this node)



The **next** portion of this node holds the address of the node containing 12.

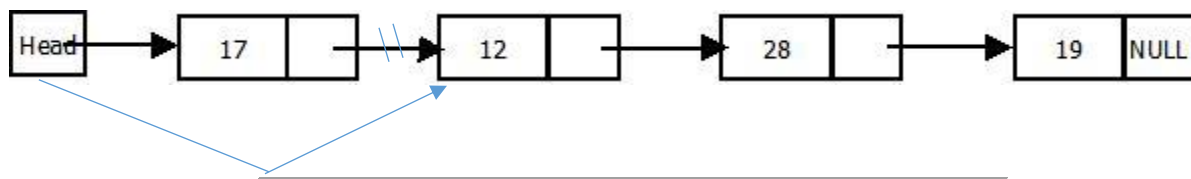
The statement,

`head = head.next;`

Assigns the address of the next node (containing 12) to head.



Thus **head** is now pointing to the node containing 12.



- A method, `LLlength(head)`, which returns the length of the linked list (number of nodes in the linked list):

```
public static int LLlength(Node head){
    int length = 0;
    Node currentNode = head;
    while(currentNode != null){
        length++;
        currentNode = currentNode.next;
    }
    return length;
}
```

- A method, `display(Node head)` to display linked list elements

```
public static void display(Node head){
    if(head == null) {
        System.out.println("Empty linked list");
        throw new IndexOutOfBoundsException();
    }
    Node currentNode = head;
    while(currentNode != null){
        System.out.print(currentNode.data+" ");
        currentNode = currentNode.next;
    }
}
```

- A method, **`displayReverse(head)`** to display linked list elements in reverse order  
The following method uses recursion to print elements in reverse order. This is easy if you are comfortable with recursion. I don't any other easy of doing this. One another way of doing it is creating a new linked list with elements in reverse order and displaying it.

```
public static void displayReverse(Node head){
    if(head == null){}
    else{
        Node currentNode = head;
        displayReverse(currentNode.next);
        System.out.print(currentNode.data+" ");
    }
}
```

---

## Now a complete program:

// A Linked List class with a private static inner Node class.

```
public class myLinkedList
{
    private static class Node
    {
        private int data;
        private Node next;

        public Node(int data, Node next)
        {
            this.data = data;
            this.next = next;
        }
    }

    private Node head;

    // Constructs an empty list

    public myLinkedList()
    {
        head = null;
    }

    // Returns true if the list is empty otherwise returns false

    public boolean isEmpty()
    {
        return (head == null);
    }

    // Inserts a new node at the beginning of this list.

    public void insertAtBeginning(int element)
    {
        head = new Node(element, head);
    }

    // Returns the first element in the list.

    public int getFirstElement()
    {
        if(head == null) {
            System.out.println("Empty linked list");
            throw new IndexOutOfBoundsException();
        }
    }
}
```

```
    return head.data;
}
```

// Removes the first node in the list.

```
public int removeFirstNode() {
    int tmp = getFirstElement();
    head = head.next;
    return tmp;
}
```

// Empties linked list

```
public void clear() {
    head = null;
}
```

// Returns the length of the linked list

```
public static int LLength(Node head){
    int length = 0;
    Node currentNode = head;
    while(currentNode != null){
        length++;
        currentNode = currentNode.next;
    }
    return length;
}
```

// Displays the linked list elements

```
public static void display(Node head){
    if(head == null) {
        System.out.println("Empty linked list");
        throw new IndexOutOfBoundsException();
    }
    Node currentNode = head;
    while(currentNode != null){
        System.out.print(currentNode.data+" ");
        currentNode = currentNode.next;
    }
}
```

// Displays the linked list elements in reverse order

```
public static void displayReverse(Node head){
    if(head == null){}
    else{
        Node currentNode = head;
        displayReverse(currentNode.next);
        System.out.print(currentNode.data+" ");
    }
}
```



```

public static void main(String[] args)
{
    myLinkedList list = new myLinkedList();//We will insert elements to this list
    myLinkedList list2 = new myLinkedList();//This is empty linked list
    list.insertAtBeginning(56);
    list.insertAtBeginning(17);
    list.insertAtBeginning(10);
    list.insertAtBeginning(46);
    list.insertAtBeginning(26);
    list.insertAtBeginning(36);
    display(list.head);
    System.out.println("\nLength of the linked list: "+LLlength(list.head));
    displayReverse(list.head);
}
}

```

---

The main method creates two linked lists, **list** and **list2**. Inserts several elements to list. Displays the elements of **list**, the length of the list and the elements of the list in reverse order.

The purpose of list2 is to test (I did include in main) if the program respond appropriately for empty list. Test it.

### Some points to remember

- Make sure your methods work for all cases, particularly the first node and the last node.
- Make sure your program captures and displays appropriate messages when head is null.

---

## 4. Additional methods

Some of the other methods you may want to include are:

- Write a method, addAtEnd(element), to add a given element at the end of the list.
- Write a method, getLastElement(), retrieve the last element of the list.
- Write a method, searchFor(x), to search for a given element in the linked list (output true or false)
- Write a method getElementAt(pos), return element at a given position

---

## 5. Java built-in LinkedList

The beauty of Java is it comes with a huge number of API (Application Programming Interface). These APIs are prewritten packages, classes, and interfaces with their respective methods, fields and constructors. There are a large set of APIs for each kind of applications (networking, mobile application, database applications and so on).

Java provides a predefined class called **LinkedList** to create and use a linked list easily without the programmer implementing low level tasks (as we did in our example). LinkedList is an implementation of List interface. For technical details about Java LinkedList class see:

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>.

### Details of Java class LinkedList with examples

#### Constructors:

There are two constructors.

<b>LinkedList()</b>	Constructs an empty list.
<b>LinkedList(Collection c)</b>	Constructs a list containing the elements of the specified collection.

**Note:** We normally use the first form.

Example:

```
LinkedList linkedlist1 = new LinkedList();
```

This creates an empty linked list, linkedlist1.

#### Printing a linked list

You can print the contents of a LinkedList using the usual System.out.println():

```
System.out.println("Linked List Content: " +linkedlist1);
```

### Methods of LinkedList class:

#### ➤ **addFirst(*element*)**

This method inserts the element in the argument at the beginning of this list.

#### **Example:**

```
LinkedList linkedlist1 = new LinkedList();  
linkedlist1.addFirst(30);  
linkedlist1.addFirst(15);  
linkedlist1.addFirst(45);  
linkedlist1.addFirst(90);  
linkedlist1.addFirst(10);  
System.out.println("Linked List Content: " +linkedlist1);
```

Each time the elements are added to the beginning of the linked list.

#### **Output:**

Linked List Contents: [10, 90, 45, 15, 30]

---

#### ➤ **add(*element*):**

This method appends the specified element to the end of this list.

#### **Example:**

```
LinkedList linkedlist1 = new LinkedList();  
linkedlist1.add(30);  
linkedlist1.add(15);  
linkedlist1.add(45);  
linkedlist1.add(90);  
linkedlist1.add(10);  
System.out.println("Linked List Contents: " +linkedlist1);
```

Each time the elements are added at the end of the linked list.

#### **Output:**

Linked List Contents: [30, 15, 45, 90, 10]

➤ **add(*index*, *element*)**

Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices). Positions of elements in the linked list start from 0.

**Example:**

Assume the Linked List Contents: [30, 15, 45, 90, 10]

```
linkedlist1.add(3,6);
```

This statement inserts element 6 in position 3.

**Output:**

The contents now: [30, 15, 45, 6, 90, 10]

➤ **set(*index*, *element*)**

Replaces the element at the specified position in this list with the specified element. Positions of elements in the linked list start from 0.

**Example:**

Assume the linked list contents: [30, 15, 45, 6, 90, 10]

This statement,

```
linkedlist1.set(4, 75);
```

Replaces the 4<sup>th</sup> element (90) by 75.

**Output:**

[30, 15, 45, 6, 75, 10]

➤ **size()**

This method returns the number of elements in this list.

**Example:**

Assume the linked list contents: [30, 15, 45, 6, 75, 10]

This statement,

```
System.out.println(linkedlist1.size());
```

Outputs 6.

### ➤ **getFirst()**

This method returns the first element of the linked list. The element returned is an object. You need to cast it into proper type to use it.

#### **Example:**

Assume, Linked List Contents: [30, 15, 45, 6, 75, 10]

This code,

```
Object sample = linkedlist1.getFirst();  
System.out.println((int)sample);
```

Output:

30

#### **Note:**

Instead of retrieving as an object, we can directly use it in the print statement:

```
System.out.println(linkedlist1.getFirst());  
This will work.
```

### ➤ **getLast()**

This method returns the last element of the linked list. The element returned is an object. You need to cast it into proper type to use it.

#### **Example:**

Assume, Linked List Contents: [30, 15, 45, 6, 75, 10]

This code,

```
Object sample = linkedlist1.getLast();  
System.out.println((int)sample);
```

Output:

10

### ➤ **get(index)**

This method returns the element at the specified index in this list. Index is an integer. The element returned is an object. You need to cast it into proper type to use it.

#### **Example:**

```
Object sample = linkedlist1.get(3);  
System.out.println((int)sample);
```

Assume, Linked List Contents: [30, 15, 45, 6, 75, 10]

#### **Output:**

6

### ➤ **removeFirst()**

This method removes and returns the first element from this list. The element returned is an object. You need to cast it into proper type to use it.

#### **Example:**

```
System.out.println("Linked List Contents before removeFirst(): " +linkedlist1);  
Object firstElement = linkedlist1.removeFirst();  
System.out.println("First element: " +(int)firstElement);  
System.out.println("Linked List Contents after removeFirst(): " +linkedlist1);
```

#### **Output:**

Linked List Contents before removeFirst(): [30, 15, 45, 6, 75, 10]

First element: 30

Linked List Contents after removeFirst(): [15, 45, 6, 75, 10]

### ➤ **removeLast()**

This method removes and returns the last element from this list. The element returned is an object. You need to cast it into proper type to use it.

**Example:**

```
System.out.println("Linked List Contents before removeLast(): " +linkedlist1);  
Object lastElement = linkedlist1.removeLast();  
System.out.println("Last element: " + (int)lastElement);  
System.out.println("Linked List Contents after removeLast(): " +linkedlist1);
```

**Output:**

```
Linked List Contents before removeLast(): [30, 15, 45, 6, 75, 10]  
Last element: 10  
Linked List Contents after removeLast(): [30, 15, 45, 6, 75]
```

**➤ contains(*element*)**

This method searches the linked list for the specified element and returns true if this list contains the specified element, otherwise returns false.

**Example:**

```
System.out.println("Linked List Contents: " +linkedlist1);  
System.out.println("45 is found in the linked list: "+ linkedlist1.contains(45));  
System.out.println("40 is found in the linked list: "+ linkedlist1.contains(40));
```

**Output:**

```
Linked List Contents: [30, 15, 45, 90, 10]  
45 is found in the linked list: true  
40 is found in the linked list: false
```

**➤ remove(*index*)**

This method removes the element at the specified index from this list. The element is returned.

**Example:**

```
System.out.println("Linked List Contents: " +linkedlist1);  
System.out.println("2nd element is removed from the linked list: "+ linkedlist1.remove(2));  
System.out.println("Linked List Contents: " +linkedlist1);
```

**Output:**

Linked List Contents: [30, 15, 45, 15, 10]

2nd element is removed from the linked list: 45

Linked List Contents: [30, 15, 15, 10]

**➤ indexOf(*element*)**

This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

**Example:**

```
System.out.println("Linked List Contents: " +linkedlist1);
```

```
System.out.println("Index of 15: " + linkedlist1.indexOf(15));
```

```
System.out.println("Index of 25: " + linkedlist1.indexOf(25));
```

**Output:**

Linked List Contents: [30, 15, 45, 15, 10]

Index of 15: 1

Index of 25: -1

**➤ lastIndexOf(*element*)**

This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

**Example:**

```
System.out.println("Linked List Contents: " +linkedlist1);
```

```
System.out.println("Last index of 15: " + linkedlist1.lastIndexOf(15));
```

```
System.out.println("Last index of 25: " + linkedlist1.lastIndexOf(25));
```

**Output:**

Linked List Contents: [30, 15, 45, 15, 10]

Last index of 15: 3

Last index of 25: -1

**➤ peek()**

This method retrieves, but does not remove, the head (first element) of this list.



**Example:**

```
System.out.println("Linked List Contents before peek(): " +linkedlist1);
System.out.println("peek() returns: "+ linkedlist1.peek());
System.out.println("Linked List Contents after peek(): " +linkedlist1);
```

**Output:**

```
Linked List Contents before peek(): [30, 15, 45, 15, 10]
peek() returns: 30
Linked List Contents after peek(): [30, 15, 45, 15, 10]
```

**➤ poll()**

This method retrieves and removes the head (first element) of this list.

**Example:**

```
System.out.println("Linked List Contents before poll(): " +linkedlist1);
System.out.println("poll() returns: "+ linkedlist1.poll());
System.out.println("Linked List Contents after poll(): " +linkedlist1);
```

**Output:**

```
System.out.println("Linked List Contents before poll(): " +linkedlist1);
System.out.println("poll() returns: "+ linkedlist1.poll());
System.out.println("Linked List Contents after poll(): " +linkedlist1);
```

**➤ clear()**

This method removes all the elements form the linked list. That it, empties the linked list.

**Example:**

```
System.out.println("Linked List Contents: " +linkedlist1);
linkedlist1.clear();
System.out.println("Size of the linked list after clear(): " +linkedlist1.size());
```

**Output:**

```
Linked List Contents: [30, 15, 45, 15, 10]
Size of the linked list after clear(): 0
```

---

## 6. Using Java ListIterator

ListIterator is an interface in java.util package, which is used to traverse through the elements in a linkedlist in either direction. It is also used in other collections. But we will use to iterate through linked lists. See <https://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html> for more technical details.

### Syntax to obtain a ListIterator of a linked list:

```
ListIterator myList = linkedlist.listIterator();
```

Before we start discussing ListIterator methods, it is important to understand how it traverse a linked list.

Notice On the left side, ListIterator (capital L) and on the right side, listIterator (lowercase l).

For example, let us take a linked list with elements,  
30, 15, 45, 15, 10

We will use a pointer to represent the position in the linked list. We treat the pointer to be in between two nodes (think of that as just before a node or just after a node). In the beginning the pointer will be just before the first node:



### ListIterator Methods

#### ➤ hasNext()

This method returns true if this list iterator has more elements when traversing the list in the forward direction. Otherwise returns false.

In a diagram, this means there is a node after the pointer:



In this diagram there is a node after pointer. So result is true.

If the situation was this:

30, 15, 45, 15, 10

The result would be false (there is no node after the pointer).

We will not show diagram in all our examples. When in doubt, use a pointer to understand which node is being processed. Informally, next element refers to the element on the right (forward direction of the linked list) of the pointer and previous element refers to the element on the left (backward direction of the linked list) of the pointer.

➤ **next()**

This method returns the next element in the list and advances the pointer position. This method may be called repeatedly to iterate through the list.

Consider the diagram,

30, 15, 45, 15, 10



If you apply `next()` once, it returns element 45 and moves the pointer to next position:

30, 15, 45, 15, 10

### Example:

Assume Linked List Content: [30, 15, 45, 15, 10]

```
ListIterator myList = linkedlist1.listIterator();
```

```
while (myList.hasNext()) {
    System.out.println(myList.next());
}
```

**Output:**

30  
15  
45  
15  
10

### ➤ **nextIndex()**

This method returns the index of the element that would be returned by a subsequent call to **next()**. (Returns list size if the list iterator is at the end of the list.)

#### **Example:**

Assume Linked List Content: [30, 15, 45, 15, 10]

```
System.out.println(myList.nextIndex());
```

```
if(myList.hasNext()) myList.next();
```

```
System.out.println(myList.nextIndex());
```

```
if(myList.hasNext()) myList.next();
```

```
System.out.println(myList.nextIndex())
```

#### **Output:**

0

1

2

### ➤ **remove()**

This method removes from the list the last element that was returned by **next()**.

#### **Example:**

Assume Linked List Content: [30, 15, 45, 15, 10]

```
myList.next();
```

```
myList.next();
```

```
if(myList.hasNext()) System.out.println(myList.next());
```

```
myList.remove();
```

The methods (three times) moves the pointer to 45. That node is removed.

Output:

Linked List Content: [30, 15, 45, 15, 10]

45

Linked List Content: [30, 15, 15, 10]

### ➤ **hasPrevious()**

As mentioned earlier, `ListIterator` can be used to traverse through the elements in a linkedlist in either direction. This method returns true if this list iterator has more elements when traversing the list in the reverse direction. (In other words, returns true if `previous()` would return an element rather than throwing an exception.) Otherwise returns false.

### **Example:**

```
System.out.println("Linked List Content: " +linkedlist1);  
ListIterator myList = linkedlist1.listIterator();  
System.out.println(myList.hasPrevious());
```

Notice the pointer is at the beginning. So there is no previous node. So returns false.

### **Output:**

```
Linked List Content: [30, 15, 45, 15, 10]  
false
```

### **Example:**

Let us move the pointer by two nodes:

```
System.out.println("Linked List Content: " +linkedlist1);  
ListIterator myList = linkedlist1.listIterator();  
myList.next();  
myList.next();  
System.out.println(myList.hasPrevious());
```

### **Output:**

```
Linked List Content: [30, 15, 45, 15, 10]  
true
```

### ➤ previous()

This method returns the previous element in the list and moves the cursor position backwards.

#### Example:

Assume Linked List Content: [30, 15, 45, 15, 10]

```
myList.next();  
myList.next();  
if(myList.hasPrevious()) System.out.println(myList.previous());  
if(myList.hasPrevious()) System.out.println(myList.previous());
```

#### Output:

15  
30

### ➤ previousIndex()




This method returns the index of the element that would be returned by a subsequent call to previous(). (Returns -1 if the list iterator is at the beginning of the list.)

#### Example:

Assume Linked List Content: [30, 15, 45, 15, 10]

```
myList.next();  
myList.next();  
if(myList.hasPrevious()) System.out.println(myList.previousIndex());  
if(myList.hasPrevious()) System.out.println(myList.nextIndex());
```

Let us take a diagram:

In the beginning	
After one myList.next();	
After another myList.next();	

previousIndex()	1
nextIndex()	2

Output:

Linked List Content: [30, 15, 45, 15, 10]

1

2

### ➤ **set(element)**

This method replaces the last element returned by next() or previous() with the specified element.

#### **Example:**

```
System.out.println("Linked List Content before set(): " +linkedlist1);
```

```
System.out.println(myList.next());
```

```
System.out.println(myList.next());
```

```
myList.set(100);
```

```
System.out.println("Linked List Content after set(): " +linkedlist1);
```

Assume Linked List Content: [30, 15, 45, 15, 10]

The first **next()** returns 30 and forwards the pointer. The second **next()** returns 15 and forwards the pointer. The **set(100)** applies to the element returned by the last **next()**, which is 15. Thus, **set(100)** replaces 15 by 100.

#### **Output:**

Linked List Content before set(): [30, 15, 45, 15, 10]

30

15

Linked List Content after set(): [30, 100, 45, 15, 10]

#### **Example:**

```
System.out.println("Linked List Content before set(): " +linkedlist1);
```

```
System.out.println(myList.next());
```

```
System.out.println(myList.previous());
```

```
myList.set(100);  
System.out.println("Linked List Content after set(): " +linkedlist1);
```

Assume Linked List Content: [30, 15, 45, 15, 10]

The first **next()** returns 30 and forwards the pointer. The **previous()** returns 30 and moves pointer backwards. The **set(100)** applies to the element returned by the **previous()**, which is 30. Thus, **set(100)** replaces 30 by 100.

### ➤ **add(element)**

This method inserts the specified element into the list. The element is inserted immediately before the element that would be returned by **next()**, if any, and after the element that would be returned by **previous()**, if any. (If the list contains no elements, the new element becomes the sole element on the list.) The new element is inserted before the implicit cursor: a subsequent call to **next** would be unaffected, and a subsequent call to **previous** would return the new element.

```
System.out.println("Linked List Content before add(): " +linkedlist1);  
System.out.println(myList.next());  
System.out.println(myList.next());  
myList.add(100);  
System.out.println("Linked List Content after add(): " +linkedlist1);
```

### **Output:**

```
Linked List Content before add(): [30, 15, 45, 15, 10]  
30  
15  
Linked List Content after add(): [30, 15, 100, 45, 15, 10]
```



---

## 7. Various other tools for processing linked lists

This section introduces various other ways to process linked lists. Everything here is presented briefly without depth. The examples show how to do certain tasks. These are features in Java 8. You need to study Java 8 features to understand these in-depth.

Stream is an interface included in java.util package. We will not discuss streams in details. We introduce briefly just enough to process linked lists.

### ➤ **allMatch(value -> predicate )**

This is a method of java.util.stream. This method returns ( boolean) whether all elements of this stream match the provided predicate.

#### **Example:**

Assume linkedlist1 is a linked list with contents: [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

```
boolean result = linkedlist1.stream().allMatch(value -> (int)value>=10) ;  
System.out.println("Result: "+result);
```

Output: true

#### **Example:**

Assume linkedlist1 is a linked list with contents: [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

```
boolean result = linkedlist1.stream().allMatch(value -> (int)value>=20) ;  
System.out.println("Result: "+result);
```

Output: false

### ➤ **anyMatch(value -> predicate )**

This is a method of java.util.stream. This method returns (boolean) whether any elements of this stream match the provided predicate.

#### **Example:**

Assume linkedlist1 is a linked list with contents: [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

```
boolean result = linkedlist1.stream().anyMatch(value -> (int)value==20) ;  
System.out.println("Result: "+result);
```

Output: true

### **Example:**

Assume linkedlist1 is a linked list with contents: [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

```
boolean result = linkedlist1.stream().anyMatch(value -> (int)value==100) ;  
System.out.println("Result: "+result);
```

Output: false

### ➤ **noneMatch(value -> predicate )**

This is a method of java.util.stream. This method returns (boolean) whether no elements of this stream match the provided predicate.

### **Example:**

Assume linkedlist1 is a linked list with contents: [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

```
boolean result = linkedlist1.stream().noneMatch(value -> (int)value==80) ;  
System.out.println("Result: "+result);
```

Output: false

### ➤ **Count()**

This is a method of java.util.stream. This method returns the count of elements in this stream.

Example:

Assume linkedlist1 is a linked list with contents: [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

```
long result = linkedlist1.stream().count() ;  
System.out.println("Result: "+result);
```

Output: 10

### ➤ **forEach()**

This is a method of `java.util.stream`. You can also call `forEach()` method without obtaining Stream from list e.g. `linkedlist.forEach()`, because the `forEach()` method is also defined in Iterable interface, but obtaining Stream gives you more choices e.g. filtering, mapping or flattening etc.

Basic syntax:

```
forEach(variable -> System.out.println(variable)
```

The variable picks up elements from the linked list and prints it.

Example:

Assume `linkedlist1` is a linked list with elements [30, 15, 45, 15, 10]

The statement,

```
linkedlist1.forEach(value -> System.out.println(value));
```

Will output:

30

15

45

15

10

Of course, you can write,

```
inkedlist1.forEach(value -> System.out.print(value+" "));
```

To see the output:

30 15 45 15 10

➤ **forEach() another version**

**forEach(System.out::println);**

**Example:**

linkedlist1.forEach(System.out::println);

This outputs:

30  
15  
45  
15  
10

➤ **forEach() with stream():**

We will briefly introduce stream(), with examples. For details, see

<http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

**Example:**

Assume linkedlist1 is a linked list with elements [30, 15, 45, 15, 10]

The statement,

linkedlist1.stream().forEach(value -> System.**out**.println(value));

Will output:

30  
15  
45  
15  
10

➤ **Using filters**

Follow the syntax carefully.

**Example:**

Linked List Content [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

The following code selects all numbers >30.

```
linkedlist1.stream()
    .filter(value->(int)value>30)
    .forEach(System.out::println);
```

**Output:**

```
45
55
35
51
40
```

**Example:**

Linked List Content [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

The following code counts how many are > 30.

```
long count = linkedlist1.stream()
    .filter(value->(int)value>30)
    .count();
System.out.println(count);
```

Output: 5

➤ **Sorting**

You can easily sort the contents of a linked list.

**Example:**

```
System.out.println("Linked List Content " +linkedlist1);
Collections.sort(linkedlist1);
System.out.println("Linked List Content " +linkedlist1);
```

**Output:**

Linked List Content [30, 15, 45, 15, 10, 20, 55, 35, 51, 40]

Linked List Content [10, 15, 15, 20, 30, 35, 40, 45, 51, 55]

---

## 8. ArrayList and Vector

In addition to LinkedList ADT, Java provides two other ADTs to maintain lists: ArrayList and Vectors. Basically all three are same in functionality. There are a large number of methods in both ArrayList and Vector, which do similar functions as in LinkedList.

For details about ArrayList ADT, see

<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

For details about Vector ADT, see

<http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html>

We will not cover ArrayList and Vector in this course. It is enough if you learn one (LinkedList). Studying ArrayList and Vector is optional.

**Note:** As you saw, LinkedList ADT is a doubly linked list. You can traverse both forward and backward. ArrayList and Vector are singly linked lists (you cannot traverse backwards).

---

---

## 9. Exercises

1. Add a new method `addAtEnd(element)`, to our implementation of linked list, `myLinkedList` class (in my notes).

An incomplete method given below, `addAtEnd(element)`, adds the specified *element* at the end of the linked list. The method is incomplete. Fill the blank (???) to complete the method.

```
public void addAtEnd(int element)
{
    if( head == null)
        insertAtBeginning(element);
    else
    {
        Node tmp = head;
        while(tmp.next != null) tmp = ???;
        ??? = new Node(element, null);
    }
}
```

Use a diagram to understand the logic.

Test your code.

It should work this way:

The original linked list: 36 26 46 10 17 56

`list.addAtEnd(22);`

The new linked list: 36 26 46 10 17 56 22

2. Add a new method `getLastElement()` to our implementation of linked list, `myLinkedList` class (in my notes).

An incomplete method given below, `getLastElement()` retrieves the last element of the linked list. The method is incomplete. Fill the blank (???) to complete the method.

```
public int getLastElement()
{
    if(head == ???) {
        System.out.println("Empty linked list");
        throw new IndexOutOfBoundsException();
    }
    Node tmp = head;
    while(tmp.next != null) tmp = ???;
    return ???;
}
```

Use a diagram to understand the logic.

Test your code.

It should work this way:

The original linked list: 36 26 46 10 17 56

```
System.out.println("\nLast element: "+list.getLastElement());
```

Output:

Last element: 56



3. Add a new boolean method `searchFor(x)` to our implementation of linked list, `myLinkedList` class (in my notes). If the element is found in the linked list, it returns true, otherwise returns false.

An incomplete method given below, `searchFor(x)` searches the linked list for the specified `x`. The method is incomplete. Fill the blank (???) to complete the method.

```
public boolean searchFor(int x){  
    if(head == null) {  
        System.out.println("Empty linked list");  
        throw new ???;  
    }  
    Node tmp = head;  
    while(tmp != null){  
        if(tmp.data == x) return true;  
        tmp = ???;  
    }  
    return ???;  
}
```

Use a diagram to understand the logic.

Test your code.

It should work this way:

The original linked list: 36 26 46 10 17 56

```
System.out.println("element is 26 found: "+ list.searchFor(26));
```

Output:

element is 26 found: true

```
System.out.println("element is 20 found: "+ list.searchFor(20));
```

Output:

element is 26 found: true

4. Add a new method `getElementAt(pos)` to our implementation of linked list, `myLinkedList` class (in my notes).

An incomplete method given below, `getElementAt(pos)`, which returns the element at the specified position. The method is incomplete. Fill the blank (???) to complete the method.

```
public int getElementAt(int pos)
{
    if(head == null) {
        System.out.println("Empty linked list");
        throw new ???;
    }
    Node tmp = head;
    int len = LLlength(head);
    if(pos > ???) {
        System.out.println("Given position is outside linked list");
        throw new IndexOutOfBoundsException();
    }
    for (int k = 0; k <= ???; k++) tmp = tmp.next;
    return tmp.data;
}
```

Test your code.

It should work this way:

The original linked list: 36 26 46 10 17 56

```
System.out.println("element at position 3: "+ list.getElementAt(3));
```

Output:

element at position 3: 10

```
System.out.println("element at position 0: "+ list.elementAt(0));
```

Output:

element at position 0: 36

5. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked after the following line of code:

```
linkedlistTest.add(44);
```

6. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked after the following line of code:

```
linkedlistTest.add(2,54);
```

7. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked after the following line of code:

```
linkedlistTest.set(3,4);
```

8. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.size());
```

9. Assume we have created a linked list using Java **LinkedList**:

**LinkedList** *linkedlistTest* = **new LinkedList**();

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

System.**out**.println(linkedlistTest.getFirst());

10. Assume we have created a linked list using Java **LinkedList**:

**LinkedList** *linkedlistTest* = **new LinkedList**();

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

System.**out**.println(linkedlistTest.getLast());

11. Assume we have created a linked list using Java **LinkedList**:

**LinkedList** *linkedlistTest* = **new LinkedList**();

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

System.**out**.println(linkedlistTest.get(4));

Also write the contents of the linked list after this line is executed.

12. Assume we have created a linked list using Java **LinkedList**:

**LinkedList** *linkedlistTest* = **new LinkedList**();

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

System.**out**.println(linkedlistTest.removeFirst());

Also write the contents of the linked list after this line is executed.

13. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.removeLast());
```

Also write the contents of the linked list after this line is executed.

14. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.contains(23));
```

Also write the contents of the linked list after this line is executed.

15. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.remove(10));
```

Also write the contents of the linked list after this line is executed.

16. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.indexOf(10));
```

Also write the contents of the linked list after this line is executed.

17. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.lastIndexOf(20));
```

Also write the contents of the linked list after this line is executed.

18. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.peek());
```

Also write the contents of the linked list after this line is executed.

19. Assume we have created a linked list using Java **LinkedList**:

```
LinkedList linkedlistTest = new LinkedList();
```

Also assume that the current contents of *linkedlistTest*:

[20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
System.out.println(linkedlistTest.poll());
```

Also write the contents of the linked list after this line is executed.

20. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

Recall the concept of a pointer.

Where is the pointer in the beginning?

Answer between: ??? and ???

21. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

Where is the pointer after the following code is executed:

```
myIterator.next();
```

```
myIterator.next();
```

Answer between: ??? and ???

22. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
myIterator.next();  
myIterator.next();  
myIterator.next();  
System.out.println(myIterator.nextIndex());
```

23. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

Show the contents of the linked list after the following code is executed:

```
myIterator.next();  
myIterator.next();  
myIterator.remove();
```



24. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
myIterator.next();  
myIterator.next();  
System.out.println(myIterator.hasPrevious());
```

25. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
myIterator.next();  
myIterator.next();  
System.out.println(myIterator.previous());
```

26. Assume we have created a linked list object, *linkedListTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedListTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedListTrial.listIterator();
```

Assume the current contents of *linkedListTrial*: [20, 35, 40, 25, 10, 20, 15]

What is the output of the following code?

```
myIterator.next();  
myIterator.next();  
myIterator.next();  
System.out.println(myIterator.previousIndex());
```

27. Assume we have created a linked list object, *linkedListTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedListTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedListTrial.listIterator();
```

Assume the current contents of *linkedListTrial*: [20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked list after the following code is executed:

```
myIterator.next();  
myIterator.next();  
myIterator.next();  
myIterator.set(50);
```

28. Assume we have created a linked list object, *linkedListTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedListTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedListTrial.listIterator();
```

Assume the current contents of *linkedListTrial*: [20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked list after the following code is executed:

```
myIterator.next();  
myIterator.next();  
myIterator.previous();  
myIterator.set(50);
```

29. Assume we have created a linked list object, *linkedListTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedListTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedListTrial.listIterator();
```

Assume the current contents of *linkedListTrial*: [20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked list after the following code is executed:

```
myIterator.next();  
myIterator.next();  
myIterator.next();  
myIterator.add(50);
```

30. Assume we have created a linked list object, *linkedlistTrial*, of Java built-in **LinkedList** class.

```
LinkedList linkedlistTrial = new LinkedList();
```

We also have a **ListIterator** object, *myIterator*

```
ListIterator myIterator = linkedlistTrial.listIterator();
```

Assume the current contents of *linkedlistTrial*: [20, 35, 40, 25, 10, 20, 15]

Write the contents of the linked list after the following code is executed:

```
myIterator.next();  
myIterator.next();  
myIterator.next();  
myIterator.previous();  
myIterator.add(50);
```

---