

AN INTRODUCTION TO LINEAR LINKED LIST

BY JOSLYN A. SMITH

Objective

To learn how to:

- Create a list
- Traverse a list
- Insert items in a list
- Remove items from a list.

Terminologies

Linked List • node • traverse • null

Introduction

Linked list is one of the fundamental data structures used in programming. A list is a finite sequence of elements where insertion of elements occurs at any point and anytime in the list. Also, deletion of elements can take place from any point in the list and at anytime. Linked list does not carry index like array or ArrayList. Hence, whether it is inserting, deleting, or traversing the list, indexing of nodes is not a requirement.

Characteristics of Linked List

A linked list consists of a set of elements called nodes. Each node is connected to the next by address rather than by index. The first node in a list is referenced by a variable described as the header address. The last node connects to no succeeding node, so it is assigned the **null** address, indicating the end of the list. **Figure 1** shows a linked list of four nodes.

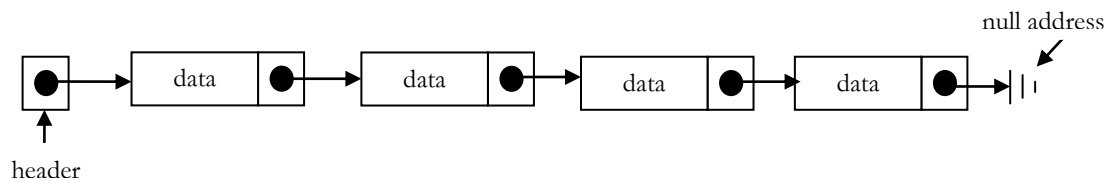


Figure 1 - A Linked List of four nodes.

There are different forms of linked list, such as singly-list (called linear list), doubly-list, multiply linked list and circular linked list. Only linear linked list will be discussed in this article.

In a linear list, a node is comprised of two components. One of the components stores the data, and the other stores the address of its succeeding node. The component that stores the address is called the link field. See **Figure 2**.

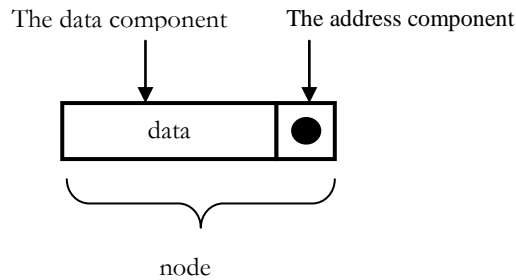


Figure 2 - A node

Figure 3 shows a list consisting of a single node. This node is referenced by the variable called **list**. The part that stores the data we will call **data** for now. Later we will add meaning to it. The address part we will call **next**, any other variable name is just as good. At this point the link field **next** contains the null address, signifying the end of the list. To refer to the data component of the node we say list dot data, written as **list•data**; and to refer to the link field, **next**, we say list dot next, written as **list•next**. The statement **list•next = null** assigns the null address to the link field variable.

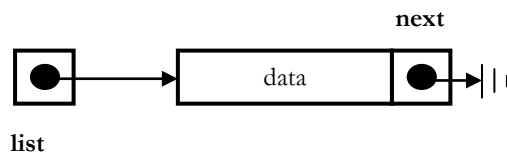


Figure 3 - A Linear Linked List containing one

The statement **list = null** abandons the list, by disconnecting the header reference from the list. See **Figure 4**. In this context there is no way of re-connecting the list with the header reference.



Figure 4 An abandoned list

In a linked list data structure, the header reference and the link reference are of the same type. Hence, linked list are referred to as self referencing data structure. This kind of data structure leads to more complex linked list implementations such as stacks, queues, and binary trees.

Representing Linked In List Java

Java defines a class called **LinkedList**, an implementation of the **List** interface. Its immediate super class is **AbstractSequentialList**, which implements the interface. We will not use this class; instead we will design and construct a linked list, in order to have a better understanding of it.

Let us design a node class called **Node.java**. This class will have two fields: the data field represented by a string variable; and a link field represented by the variable **next**. See **Listing 1**. **Figure 5** shows a pictorial representation of the code.

```

1. class Node
2. {
3.     a. String data;
4.     b. Node next;
5. }

```

Listing 1 - The class Node

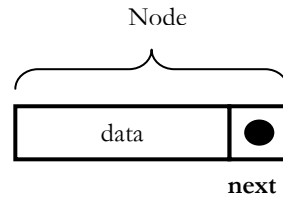


Figure 5 – Representation of the class Node

Listing 2 shows how the constructor initializes each of the fields. Notice that the linked field, **next**, points to no node. It is assigned the null address, as indicated by **Line 9**. **Figure 6** shows a pictorial representation of the class.

```

1. class Node
2. {
3.     String data;
4.     Node next;
5.
6.     Node(String s)
7.     {
8.         data = s;
9.         next = null;
10.    }
11. }

```

Listing 2 - Constructor initializes the

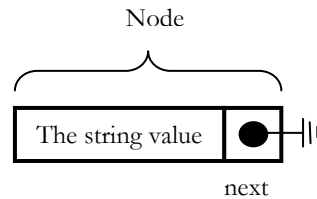


Figure 6 - Representation of the initialized

We will now create an empty list. To create an empty list necessitates implementing a class that will be used to create and maintain the list. Let us call this class **LinkedList**. **Listing 3** shows a partial definition of the class.

```

1. class LinkedList
2. {
3.     Node list;
4.
5.     LinkedList()
6.     {
7.         list = null;
8.     }
9. }

```

Listing 3 - Partial definition of the class LinkedList

Line 3 of the code declares a reference variable of type **Node**, called **list**. This variable will serve to point to the beginning or head of the list. **Figure 7** shows a pictorial representation of the variable.



Figure 7 - Reference variable list of type Node

The code in **Line 7** creates an empty list. See **Figure 8** shows a pictorial representation of the list. Next we will define an accessor method that will serve to determine if the list is empty. See **Listing 4, Line 9**.



Figure 8 - The variable list is assigned the null address

```

1. class LinkNode
2. {
3.     Node list;
4.
5.     LinkNode() { list = null; }
6.
7.     boolean isEmpty()
8.     {
9.         return list == null;
10.    }
11. }

```

Listing 4 boolean method isEmpty

We now define a method called `prepend`, that adds a node at the beginning of the list. See **Listing 5**.

```

1. class LinkNode
2. {
3.     Node list;
4.
5.     LinkNode()
6.     {
7.         list = null;
8.     }
9.
10.    boolean isEmpty()
11.    {
12.        return list == null;
13.    }
14.
15.    void prepend(String s)
16.    {
17.        Node temp = new Node(s);
18.        temp.next = list;
19.        list = temp;
20.    }
21. }

```

Listing 5 - Includes the method prepend

Figures 10 thru 12 show the state of the list when a node is inserted at the front of the list

a) Line 7 creates an empty list, as shown in Figure 9



list

Figure 9 – An empty list

b) Line 17 creates a new node using the string value it gets, and sets the link field to null. See Figure 10

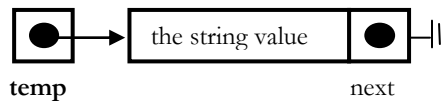


Figure 10 - A new node created

c) Line 18 points the link field of the recent node, to the reference variable list. See Figure 11.

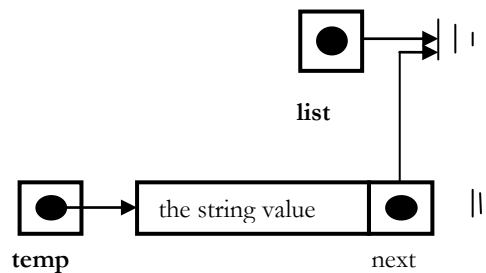


Figure 11 – Link field points to the null address

d) Line 19 – The variable list points to the node being referenced by the variable temp, thereby making the variable list pointing to the head of the list. See Figure 12

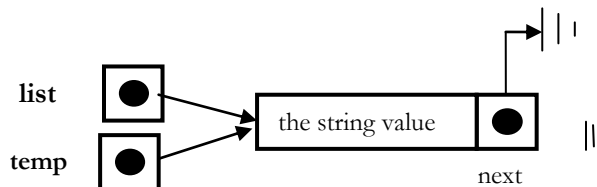


Figure 12 – The variable list is pointing at the beginning of the list

To append a node to a list requires traversing the list, visiting every node, starting from the first to the last, skipping none. That is, loop through the list and update the reference field for each node visited. When traversing a linked list never use the original reference variable that points to the beginning of list; rather, get an auxiliary reference variable to do that task. In this example we declare and initialize an auxiliary reference variable as follows:

```
ListNode current = list;
```

Figure 13 shows the effect of this statement. That is, the variable `current`, points to the first node of the list.

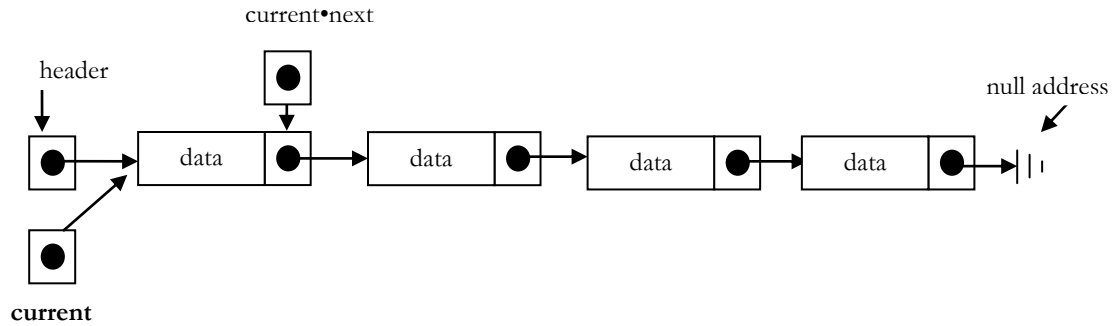


Figure 13 - Reference variable `current` points to the head of the

In **Figure 13**, the address component of the first node (`current.next`) points to the second node in the list. In order for the reference, `current` to visit the second node, it must point to the node to which `current.next` is pointing. The required code for this statement is:

```
current = current.next;
```

Figure 14 shows the effect of the above statement.

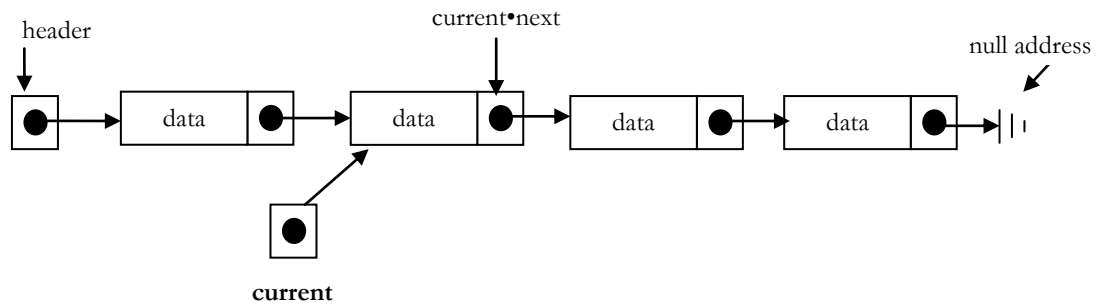


Figure 14 - `current = current.next;`

If we repeat the statement a second time, then the reference variable `current` will now be pointing to the third node. In addition to traversing the list, you must determine when `current.next` is pointing to null. When it is determined, divert `current.next` from pointing to null, and let it point to the new node that `temp` is pointing to. See **Figure 15**.

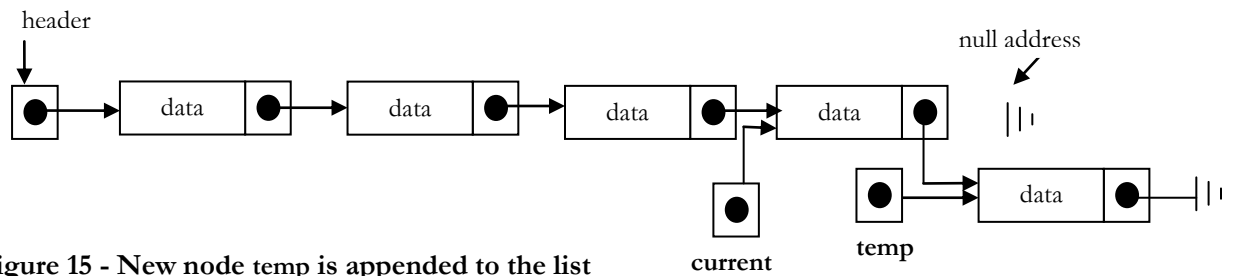


Figure 15 - New node `temp` is appended to the list

In **Listing 6**, the method `append` places the new node, which is being pointing to by `temp`, at the end of the list.

```

1. class LinkNode
2. {
3.     Node list;
4.
5.     LinkNode() { list = null; }
6.
7.     boolean isEmpty() { return list == null; }
8.
9.     void prepend(String s)
10.    {
11.        Node temp = new Node(s);
12.        temp.next = list;
13.        list = temp;
14.    }
15.
16.    void append(String s)
17.    {
18.        Node temp = new Node(s);
19.
20.        if (isEmpty())
21.            list = temp;
22.        else
23.        {
24.            Node current = list;
25.            try
26.            {
27.                while (current.next != null)
28.                    current = current.next;
29.            }
30.            catch(NullPointerException e)
31.            {
32.                e.printStackTrace();
33.            }
34.            current.next = temp;
35.        }
36.    }

```

Listing 6 - shows the code for appending a node to the list.

As shown in **Listing 6**:

- The method accepts a string value as the `string`, and uses the string to create a new temporary node reference by the variable `temp`. See **Line 18**.
- If the list is empty, then this new node will be the first in the list. See **Lines 20 and 21**.
- If the list is not empty, declare an auxiliary reference, and let this auxiliary reference point to the beginning of the list. See **Line 24**.
- Use the auxiliary reference to traverse the list, looking for when `current.next` points to the null address. See **Lines 27 and 28**.
- As soon as we know when `current.next` is pointing to the null address, change its direction, and let it point to the new node instead. See **Line 34**.

Traversing a linked list has the potential of encountering null pointer. Against this background, enclose this portion of the code with the exception handler, **NullPointerException**.

You can insert a node anywhere in the list. Usually we use this method to maintain an ordered list. You can also delete a node just about any point in the list. This property requires traversing the list, looking for a given node. When the node is found, re-arrange the reference fields so as to eliminate that node. We will use an example to illustrate these two situations.

Example

Librarians get books from time to time. Formally, when a librarian gets a book, the title would be recorded manually in alphabetical order. This means frequent re-writing of the list. Write a program such that as a book is received, it is automatically placed in the right sequence. From time to time books become obsolete. When a book becomes obsolete it is removed from the collection. The program must take this into consideration also.

Use the concept of linked list to implement this idea, by designing the following classes:

- Book - this class holds the title of book.
- BookNode – this class creates the node of information and its link field.
- BookList – this class creates and maintains the linked list of book titles in alphabetical order, and.
- Library – this is the test class that co-ordinate the activities of the other classes.

Solution

The solution to this problem runs parallel to the properties of linked list outlined above. The data in this case is referring to books. A book as we know has several attributes: title, ISBN, pages, price, author, etc. To keep things relatively simple we define the class book with a single field, the title. In addition we provide an accessor method to return the title of the book. See **Listing 7**.

```

1. class Book // This class contains the information
2. {
3.     String title;
4.
5.     Book(String t)
6.     {
7.         title = t;
8.     }
9.
10.    public String getTitle()
11.    {
12.        return title;
13.    }
14. }
```

Listing 7 - class Book

The class BookNode is the most important of the four classes. It is used to store the data as well as the reference to the succeeding node. The data field is an object of type Book, and the link field we will call **next**. Notice that the link field is of the type BookNode. The book field is initialized the usual way; however, the link field is assigned the null address. This makes this node potentially the last node in the list. See **Listing 8**


```

1. class BookNode // Creates book node objects
2. {
3.     Book book;
4.     BookNode next;
5.
6.     BookNode(Book b)
7.     {
8.         book = b;
9.         next = null;
10.    }
11.
12.    Book getBook() {return book;}
13.
14.    BookNode getNextNode() {return next;}
15.
16.    void setNode(BookNode b){ next = b;}
17. }

```

Listing 8 - The class BookNode

The problem requires that the titles are to be arranged alphabetically. This means finding the proper place to insert the title. To search for the proper place requires traversing the list and comparing the title of the new book with the title of each book visited in the list.

As before, traversal of the list requires an auxiliary reference. If the data in the new node is alphabetically rank less than the data in the current node of the list, then the position is found. At this point, place the new node in the list. See **Figure 16**. As can be seen, the value of **temp.data** is less than the value at **current.data**. So we have found the right position to place the node. That is, the new node must fall between the node containing the letter **M** and the node containing the letter **P**. This solution has one problem. We cannot reference the node containing the letter **M**, since there is no reference pointing to it.

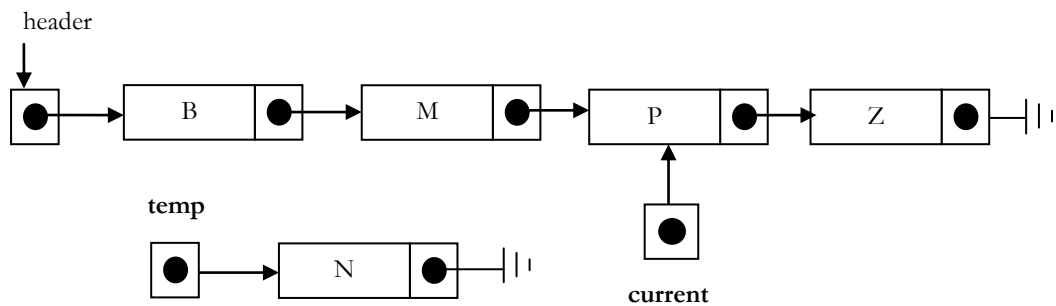


Figure 16 - Traversing the list to find proper place

To fix this problem, a second auxiliary reference is required. This reference will be made to trail the reference, **current**. When the position is found we re-arrange the references to include the new node. We will call this new reference, **back**. This reference must be set to the node being pointed to by the reference **current**, before advancing **current** to its successor node. That is:

1. Set back to point to current. The requisite code is, **back = current**, and

- Advance current to point to its successor. That is, ($\text{current} = \text{current.next}$)

See **Figure 17**.

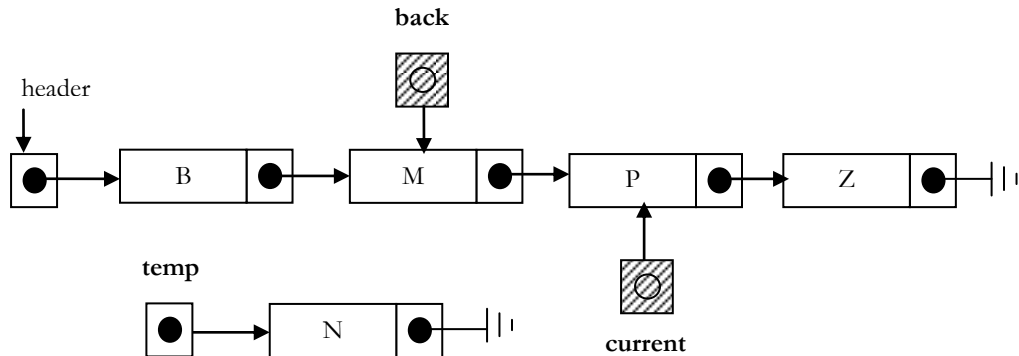


Figure 17 - Introduction of new reference variable back

Once we have these variables in place, it is only a matter of re-arranging them to accommodate the new node. The two required actions are:

- Set temp.next to point to current . That is, $\text{temp.next} = \text{current}$, and
- Set back.next to point to temp . That is, $\text{back.next} = \text{temp}$.

The bold arrows in **Figure 18** show the re-arrangement of the references.

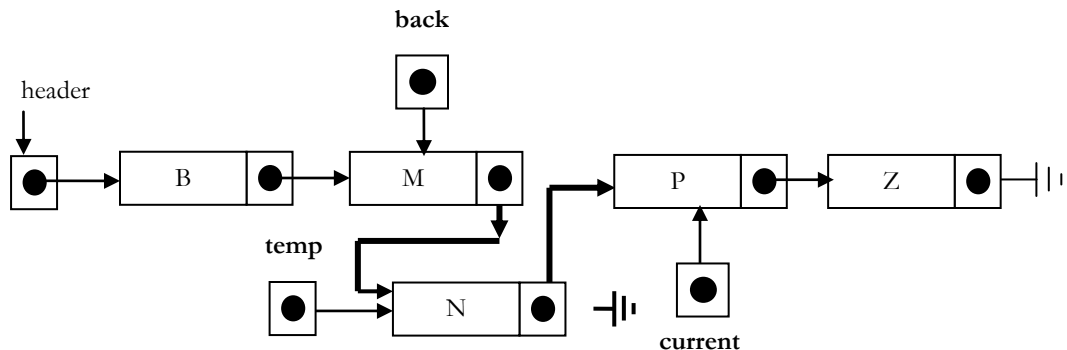


Figure 18 - Placing temp node in list

Initially the reference back must be set to null. If after traversing the list back is still null, then this new node must be placed at the head of the list. On the other hand, if after the list is traversed you did not find a place to insert it, and back is not null, then the new node must be placed at the end of the list.

The class `BookList` does the following:

- Declares the reference variable `list`, of type `BookNode`. See **Listing 9, Line 3**.
- The constructor creates an empty list, by initializing the reference to null. See **Line 7**.

This variable will be used to point to the beginning of the list at all times. If we need to traverse the list, use an auxiliary reference to do so.

```

1. class BookList
2. {
3.     private BookNode list;
4.
5.     BookList()
6.     {
7.         list = null;
8.     }
9. }

```

Listing 9 - class BookList creates an empty

Listing 10 shows the method `insert` that inserts a node in its proper position in the list. The method receives the book object `b`, and uses it to create a book node object, that is referenced by the variable called `temp`.

```

10. void insert(Book b)
11. {
12.     try
13.     {
14.         BookNode temp = new BookNode(b);
15.
16.         BookNode current = list; // Auxillary node points to the head of the list
17.         BookNode back = null; // This reference will trail current
18.         boolean found = false;
19.
20.         while (current != null && !found)
21.             if (temp.book.getTitle().compareTo(current.book.getTitle()) < 0)
22.                 found = true;
23.             else
24.                 {
25.                     back = current; // Save the current position in the list
26.                     current = current.getNextNode(); // Advance to the next book node in the list
27.                 }
28.
29.         temp.setNode(current);
30.
31.         if (back == null)
32.             list = temp;
33.         else
34.             back.setNode(temp);
35.     }
36.     catch(NullPointerException e){
37.         e.printStackTrace();
38.     }
39. }

```

Listing 10 - The method insert places a node in its proper position in the list

In preparation to traverse the list we introduce the two auxiliary reference variables, one that will traverse the list, and the other that will trail this current reference that is traversing the list. See **Lines 16** and **17**. Both variables are necessary in order to position the new node in its proper place. In addition, we declare and initialize a Boolean variable called `found` to false. This variable will alert us when we have found the position to place the node. See **Line 18**.

The heart of this method is the while loop. The condition to continue looping is that the list should not be exhausted; neither should the position be found. This is represented by the conditional expression in **Line 20**.

```
current != null && !found
```

The **if** statement on **Line 21** compares both titles. As soon as the condition is true, the loop is terminated. If not, set the reference **back** to the reference **current**, then move the reference **current** to the succeeding node pointed to by **current.next**. See **Lines 25** and **26**.

When the while loop is exited re-organize the references. First, let the **temp•next** point to where **current** is pointing. See **Line 29**. Now, depending on the state of **back**, we will know how to position the node. That is, if **back** is still null, then it means that the new node must be at the front of list. If **back** is not null, then the new node must fall between the two nodes – **back** and **current**. Hence we let **back•next** point to **temp**. See **Lines 31** thru **34**.

The problem requires us to be able to remove a node from the list at anytime. To remove a node, requires knowing the title of the book. First we search for the title, using similar principle to the insert method. Once the node is located, re-arrange the references so that the particular node is excluded. See **Listing 11**.

```

47. void remove(String title)
48. {
49.     try
50.     {
51.         BookNode back = null,
52.         current = list;
53.         boolean found = false;
54.
55.         while (current != null && !found)
56.             if (current.book.getTitle().compareToIgnoreCase(title) == 0)
57.                 found = true;
58.             else
59.                 {
60.                     back = current;
61.                     current = current.getNextNode(); //current = current.next;
62.                 }
63.
64.             if (found)
65.                 {
66.                     back.setNode(current.getNextNode()); // back.next = current.next;
67.                     current.setNode(null); //current.next = null;
68.                 }
69.         }
70.     catch(NullPointerException e)
71.     {
72.         e.printStackTrace();
73.     }
74. }
```

Listing 11 - The method removes a node from the list.

As stated earlier, the remove method is similar to the insert method. The first change is the conditional expression of the **if** statement. Instead of testing for the inequality less than ($<$) we test for the equality ($==$). That is:

```
if (current.book.getTitle().compareToIgnoreCase(title) == 0
```

There are two actions that must be taken after exiting the loop:

1. The reference **back.next** must point where the reference of **current.next** is pointing. See **Line 66**.
2. The reference **current.next** must be assigned null, in order to drop the node from the list. See **Line 67**.

Figure 19 shows the re-arrangement of the nodes.

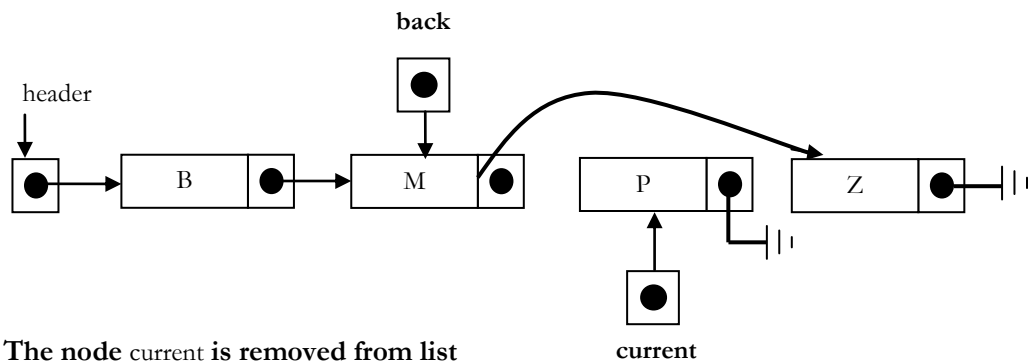


Figure 19 The node **current** is removed from list

Finally, **Listing 12** shows the definition of the **toString** method. This method traverses the list and extracts the title from each node in the list. The titles are appended to the string called **result**. The string is returned for printing.

```

76. /*
77.  * This method traverses the list and extracts the title from the
78.  * linked list of nodes. They are appended to a string.
79.  * The string is returned for printing.
80.  */
81.
82. public String toString()
83. {
84.     String result = "";
85.
86.     BookNode current = list;
87.
88.     while (current != null)
89.     {
90.         result += current.book.getTitle() + "\n";
91.         current = current.next;
92.     }
93.     return result;
94. }
```

Listing 12 - Method that displays the list.

Listing 13 shows the test class called Library. This class creates an empty BookList object, and then adds four book objects to the list. After displaying the list of books, it removes a book from the list and displays the list again. In both cases this procedure maintains the book titles alphabetically.

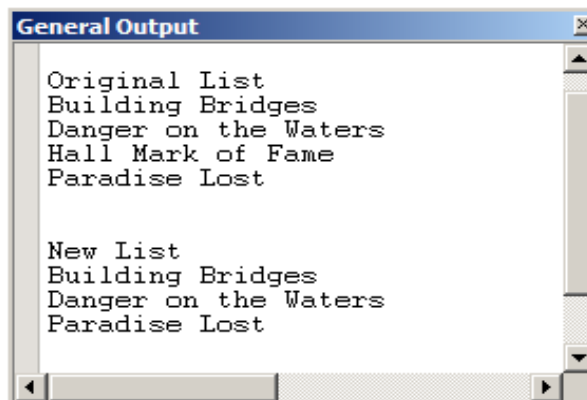
```

1. class Library
2. {
3.     public static void main(String[] arg)
4.     {
5.         // Create a book list object.
6.         BookList books = new BookList();
7.
8.         // Create book nodes and add them to the list.
9.         books.insert(new Book("Danger on the Waters"));
10.        books.insert(new Book("Paradise Lost"));
11.        books.insert(new Book("Building Bridges"));
12.        books.insert(new Book("Hall Mark of Fame"));
13.
14.        // Print the original list
15.        System.out.println("\nOriginal List");
16.        System.out.println(books);
17.
18.        // Remove a book
19.        books.remove("Hall Mark of Fame");
20.
21.        // Print new list
22.        System.out.println("\nNew List");
23.        System.out.println(books);
24.    }
25. }

```

Listing 13 - The test class Library

Figure 20 shows the output generated from the program



```

General Output
Original List
Building Bridges
Danger on the Waters
Hall Mark of Fame
Paradise Lost

New List
Building Bridges
Danger on the Waters
Paradise Lost

```

Figure 20 – List of titles alphabetically

SUMMARY

Linked list is one of the simplest but most commonly used data structures in programming. Linear linked list is fundamental to other complex abstract data structures such as stacks, queues, circular list, and binary trees, and multiply linked lists. Linked lists allow insertion and removal of nodes at any point in the list without disturbing the data from its position. This is a major advantage over arrays and ArrayList.