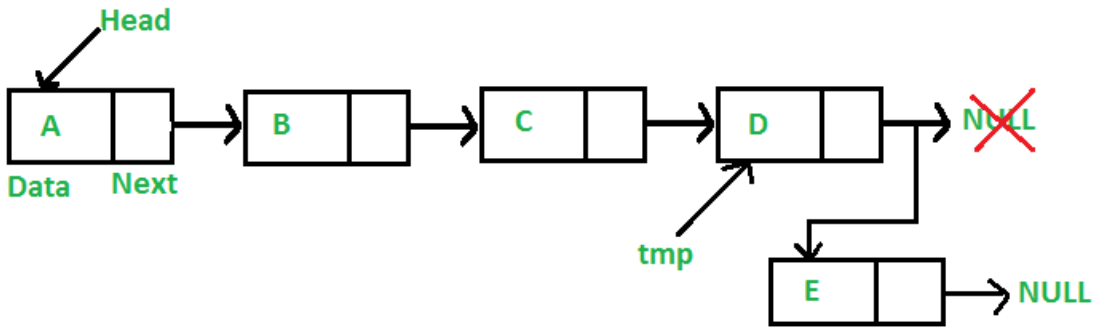
Singly Linked List Tutorial

By: Adriaan Veney

How it works/stores data:

- A Linked List works by having a collection of objects called "nodes" linked together. Each node holds a value and a pointer to the address in memory of its successive node.
- The first node in a Linked List is called the "head"
- The final node in a Linked List is called the "tail"
- The tail's next pointer points to null



How a client interacts with it (interface):

- Clients interact with a Linked List by declaring an instance of its class. Methods are used to insert, remove, and search for data in the list.
- Insertion methods:
  - append(): Updates the list tail
  - prepend(): Updates the list head
  - insertAfter(): Adds a node to a list before a given node

- Remove methods:
  - removeNode(): Deletes a node from the list

- Search methods:
  - search(): Search for a given node. True will be returned if the node is found. False will

be returned if the node is not in the list.

Pros of using a Linked List:

- Dynamic data structure able to add/remove elements at runtime
- Memory is not wasted when nodes are deleted
- The utilize generics and are not limited to a specific data type

Cons of using a Linked List:

- O(N) time complexity for some operations
- Traversal of the list is required to return a specific node

Linked List Time Complexity:

The methods that yield O(N) time complexity require a traversal of the list. Methods with O(1) have a constant runtime.

- Insertion:
  - append(): O(N)
  - prepend(): O(1)
  - insertAfter(): O(N)

- removeNode(): O(N)
- search(): O(N)

Building a Linked List

Step 1: Generate Node Class

```
public class Node<T> {

    // Field to hold the node's data
    T data;
    // Pointer to the next element's address in memory
    Node<T> next;

    // Constructor
    Node(T data) {
        this.data = data;
    }

    // No-argument constructor
    public Node() {

    }
}
```

Step 2: Generate the SinglyLinkedList Class

```
public class SinglyLinkedList<T> {

    // Head of the linked list
    Node<T> head;

    public boolean isEmpty(){}
    public void append(T value) {}
    public void prepend(T value) {}
    public void insertAfter(T currentVal, T newVal) {}
    public void removeNode(T value) {}
    public boolean searchList(T value) {}
    public void printList() {}
}
```

Step 3: Method implementation for SinglyLinkedList

- **isEmpty() method:**

```
    // Determines if the list is empty by checking if the
    // list head has been set to a value
    public boolean isEmpty(){
        return head == null;
    }
```

- **append() method:**

```
    // Updates the end of the list by first traversing to the tail, then
    // setting its next pointer to the address of the new node
    public void append(T value) {

        // Create a new node object with the given value
        Node<T> node = new Node(value);
        node.next = null;

        // If the list is empty, set the head to the new node
        if (head == null) {
            head = node;
        } else {
            // Start traversal from the head of the list
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            // Set the tail element to the new node
            current.next = node;
        }
    }
```

- **prepend() method:**

```
    // Updates the head of the list by first setting the new node's
    // next element to the current head. Then, the new node
    // becomes the new head.
    public void prepend(T value) {

        // Create a new node object with the given value
        Node<T> newNode = new Node(value);

        // Set new node's next value to the head
        newNode.next = head;
        // Set the head to the new node
        head = newNode;
    }
```

- **insertAfter() method:**

```
    // Inserts a node into the list after a given node
    public void insertAfter(T currentVal, T newVal) {

        // Create a new node object with the given value
        Node<T> newNode = new Node(newVal);

        // Start traversing the list from the head
        Node<T> currentNode = head;

        while (currentNode != null) {
            // If the current value is found
            if (currentNode.data == currentVal) {

                // Set new node's next value to the current node's next val
                newNode.next = currentNode.next;

                // Set current node's next value to the new node value
                currentNode.next = newNode;
                break;
            } else {
                // Continue with traversal of the list
                currentNode = currentNode.next;
            }
        }
    }
```

- **removeNode() method:**

```
 // Removes a node from the list by traversing the list while keeping
// track of the previous and current nodes. If the current node is the
// desired element to be deleted, the previous node's next pointer is
// directed towards the current node's next element.

 public void removeNode(T value) {

     // Declare the head and previous nodes
     Node<T> currentNode = head;
     Node<T> prevNode = null;

     // If the node to be deleted is found in the head
     if ((currentNode != null) && (currentNode.data == value)) {
         head = currentNode.next;
     }

     // Search for the node to be deleted
     // Keep updating the previous and temporary nodes until the
     // value is found
     while ((currentNode != null) && (currentNode.data != value)) {
         prevNode = currentNode;
         currentNode = currentNode.next;
     }

     // If the value to be deleted is not found
     if (currentNode == null) {
         return;
     }

     // Unlink the node from the linked list
     prevNode.next = currentNode.next;
 }
```

- **searchList() method:**

```
 // Searches for a value in the list by traversal.
// True is returned if the node's element is found.
// False is returned if the node's element is not found.
public boolean searchList(T value) {

    // Generate a new node object
    Node<T> searchedValue = new Node(value);

    // Start traversal from the head of the list
    Node<T> current = head;

    while (current != null) {
        // If the current node's data is equal to the desired node
        if (current.data == searchedValue.data) {
            // Return true
            return true;
        } else {
            // Else, continue to the next node
            current = current.next;
        }
    }
    // Return false if the node is not found
    return false;
}
```

- **printList() method:**

```
// Print the list by traversing from the head and printing
// each node's data at each iteration
public void printList() {

    // Start traversal from the head
    Node<T> node = head;

    while (node.next != null) {

        // Print the node's data
        System.out.println(node.data);

        // Continue to next node
        node = node.next;
    }
    System.out.println(node.data);
}
```

Examples with Linked Lists

```
public static void main (String[] args) {

        // Create a singly linked list object
        SinglyLinkedList list = new SinglyLinkedList();

        // This section will print the elements in the following order:
        // 10, 4, 7, 5, hello
        list.append(4);
        list.append(5);
        list.insertAfter(4, 7);
        list.append("hello");
        list.prepend(10);
        list.printList();

        // This section will print the elements in the following order:
        // 10, 7, 5, hello, true
        System.out.println();
        list.removeNode(4);
        list.printList();
        System.out.println(list.searchList(10));
    }
```