Practical 3: Applications of Linked Lists

Aim

Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement **addition, subtraction, multiplication, and exponentiation** operations. Limit exponents to be positive integers.

What is the **asymptotic running time** for each of your operations, expressed in terms of the number of digits for the two operands of each function?

Program

1. **Addition**

```
package PRACTICAL3;

public class Addition {
   static Node head1, head2;

   static class Node {
      int data;
      Node next;
      Node(int d) {
         data = d;
         next = null;
      }
   }

   void addTwoLists(Node first, Node second) {
      Node start1 = new Node(0);
      start1.next = first;
      Node start2 = new Node(0);
      start2.next = second;

      addPrecedingZeros(start1, start2);
      Node result = new Node(0);
      if (sumTwoNodes(start1.next, start2.next, result) == 1) {
         Node node = new Node(1);
         node.next = result.next;
         result.next = node;
      }
      printList(result.next);
   }

   // Adds lists and returns the carry
   private int sumTwoNodes(Node first, Node second, Node result) {
      if (first == null) {
         return 0;
      }
      int number = first.data + second.data + sumTwoNodes(first.next, second.next, result);
      Node node = new Node(number % 10);
      node.next = result.next;
```

```
    result.next = node;
    return number / 10;
}

// Appends preceding zeros in case a list is having lesser nodes than the other one
private void addPrecedingZeros(Node start1, Node start2) {
    Node next1 = start1.next;
    Node next2 = start2.next;
    while (next1 != null && next2 != null) {
        next1 = next1.next;
        next2 = next2.next;
    }
    if (next1 == null && next2 != null) {
        while (next2 != null) {
            Node node = new Node(0);
            node.next = start1.next;
            start1.next = node;
            next2 = next2.next;
        }
    } else if (next2 == null && next1 != null) {
        while (next1 != null) {
            Node node = new Node(0);
            node.next = start2.next;
            start2.next = node;
            next1 = next1.next;
        }
    }
}

// Function to print linked list
void printList(Node head) {
    while (head != null) {
        System.out.print(head.data + " ");
        head = head.next;
    }
    System.out.println("");
}

public static void main(String[] args) {
    Addition list = new Addition();

    // creating first list
    head1 = new Node(7);
    head1.next = new Node(4);
    head1.next.next = new Node(1);
    head1.next.next.next = new Node(6);
    head1.next.next.next.next = new Node(7);
    System.out.print("First list :");
    list.printList(head1);

    // creating second list
    head2 = new Node(4);
```

```
        head2.next = new Node(3);
        System.out.print("Second list : ");
        list.printList(head2);

        System.out.print("Added list : ");
        list.addTwoLists(head1, head2);
    }
}
```

Output

```
First list :7 4 1 6 7
Second list : 4 3
Added list : 7 4 2 1 0
```

2. Subtraction

```
package PRACTICAL3;

import java.lang.*;

class Subtraction {
    static Node head;
    boolean borrow;

    static class Node {
        int data;
        Node next;
        Node(int d){
            data = d;
            next = null;
        }
    }

    int getLength(Node node){
        int size = 0;
        while (node != null) {
            node = node.next;
            size++;
        }
        return size;
    }

    Node paddZeros(Node sNode, int diff){
        if (sNode == null) {
            return null;
        }
        Node zHead = new Node(0);
        diff--;
        Node temp = zHead;
        while ((diff--) != 0) {
```

```
            temp.next = new Node(0);
            temp = temp.next;
        }
        temp.next = sNode;
        return zHead;
    }

    Node subtractLinkedListHelper(Node l1, Node l2) {
        if (l1 == null && l2 == null && borrow == false)
            return null;

        Node previous
                = subtractLinkedListHelper(
                (l1 != null) ? l1.next
                        : null,
                (l2 != null) ? l2.next : null);

        int d1 = l1.data;
        int d2 = l2.data;
        int sub = 0;
        if (borrow) {
            d1--;
            borrow = false;
        }
        if (d1 < d2) {
            borrow = true;
            d1 = d1 + 10;
        }
        sub = d1 - d2;
        Node current = new Node(sub);
        current.next = previous;
        return current;
    }
    Node subtractLinkedList(Node l1, Node l2) {
        if (l1 == null && l2 == null){
            return null;
        }
        int len1 = getLength(l1);
        int len2 = getLength(l2);

        Node lNode = null, sNode = null;
        Node temp1 = l1;
        Node temp2 = l2;
        if (len1 != len2) {
            lNode = len1 > len2 ? l1 : l2;
            sNode = len1 > len2 ? l2 : l1;
            sNode = paddZeros(sNode, Math.abs(len1 - len2));
        }
```

```
        else {
            while (l1 != null && l2 != null) {
                if (l1.data != l2.data) {
                    lNode = l1.data > l2.data ? temp1 : temp2;
                    sNode = l1.data > l2.data ? temp2 : temp1;
                    break;
                }
                l1 = l1.next;
                l2 = l2.next;
            }
        }
        borrow = false;
        return subtractLinkedListHelper(lNode, sNode);
    }

    static void printList(Node head){
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }

    public static void main(String[] args) {
        Node head = new Node(3);
        head.next = new Node(8);
        head.next.next = new Node(5);
        System.out.print("First list : ");
        printList(head);

        Node head2 = new Node(4);
        head2.next = new Node(3);
        System.out.print("\nSecond list : ");
        printList(head2);

        Subtraction ob = new Subtraction();
        Node result = ob.subtractLinkedList(head, head2);
        System.out.print("\nSubtracted list : ");
        printList(result);
    }
}
```

Output

```
First list : 3 8 5
Second list : 4 3
Subtracted list : 3 4 2
```

3. Multiplication

```
package PRACTICAL3;

public class Multiplication {
   static class Node {
      int data;
      Node next;
      Node(int data){
         this.data = data;
         next = null;
      }
   }

   static long multiplyTwoLists(Node first, Node second) {
      long N = 1000000007;
      long num1 = 0, num2 = 0;

      while (first != null || second != null){
         if(first != null){
            num1 = ((num1)*10)%N + first.data;
            first = first.next;
         }
         if(second != null) {
            num2 = ((num2)*10)%N + second.data;
            second = second.next;
         }
      }
      return ((num1%N)*(num2%N))%N;
   }

   static void printList(Node node){
      while(node != null) {
         System.out.print(node.data);
         if(node.next != null)
            System.out.print(" ");
         node = node.next;
      }
      System.out.println();
   }

   public static void main(String args[]) {
      Node first = new Node(9);
      first.next = new Node(4);
      first.next.next = new Node(6);
      System.out.print("First List : ");
      printList(first);

      Node second = new Node(8);
```

```
        second.next = new Node(4);
        System.out.print("Second List : ");
        printList(second);

        System.out.print("Multiplied list : ");
        System.out.println(multiplyTwoLists(first, second));
    }
}
```

Output

```
First List : 9 4 6
Second List : 8 4
Multiplied list : 79464
```

Analysis of Algorithm

For the singly linked list implementation of integers of unlimited size, the asymptotic running time for each of the operations would be as follows:

- **Addition**: O(n + m), where n and m are the number of digits for the two operands. This is because each node of the linked list must be traversed in order to add each corresponding digit, and the operation will take longer if the operand with more digits is used.
- **Subtraction**: O(n + m), where n and m are the number of digits for the two operands. This is similar to addition, as each node of the linked list must be traversed in order to subtract each corresponding digit.
- **Multiplication**: O(n * m), where n and m are the number of digits for the two operands. This is because for each digit of the first operand, the entire second operand must be traversed in order to multiply each corresponding digit.