JOBSHEET IX

LINKED LIST

1.1. Learning Objective

After learning this practicum, students will be able to:

1. Create a linked list data structure
2. Create a program that implements linked list
3. Differentiate the problems that can be solved with linked list

1.2. 1st Lab Activities

In this practicum, we will implement how to create single linked list with **nodes** data representation, accessing the linked list, and adding the data.

1.2.1. Steps

1. Create a new package named **week11**
2. Add these following classes:
   a. `Node.java`
   b. `SingleLinkedList.java`
   c. `SLLMain.java`
3. Create Node class

```
public class Node {
    int data;
    Node next;

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

}
```

4. Add these following attributes in class **SingleLinkedList**

```
public class SingleLinkedList {
    Node head; // Initial position in linked list
    Node tail; // Last position in linked list
}
```

5. For the next step, we will implement methods that are exist in **SingleLinkedList**

```
public class SingleLinkedList {
    Node head; // Initial position in linked list
    Node tail; // Last position in linked list
}
```

6. Add method isEmpty()

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

7. Implement this method to display the data with traverse process

```
public void print(){
    if(!isEmpty()){
        Node tmp = head;
        System.out.print("Linked list content: \t");
        while (tmp != null) {
            System.out.print(tmp.data + "\t");
            tmp = tmp.next;
        }
        System.out.println("");
    }else{
        System.out.println("Linked list is empty");
    }
}
```

8. Implement method **addFirst**()

```
public void addFirst(int input){
    Node ndInput = new Node(input, null);
    if(isEmpty()){ // if linked list is empty
        head = ndInput; //head and tail is equal with node input
        tail = ndInput;
    }else{
        ndInput.next = head;
        head = ndInput;
    }
}
```

9. Implement method **addLast**()

```
public void addLast(int input){
    Node ndInput = new Node(input, null);
    if(isEmpty()){ // if linked list is empty
        head = ndInput; //head and tail is equal with node input
        tail = ndInput;
    }else{
        tail.next = ndInput;
        tail = ndInput;
    }
}
```

10. Implement method **`insertAfter`** (), to insert a node that stores data that were inputted by the user after data ***key***

```
public void insertAfter(int key, int input) {
    Node ndInput = new Node(input, null);
    Node temp = head;
    do {
        if (temp.data == key) {
            ndInput.next = temp.next;
            temp.next = ndInput;
            if(ndInput.next==null) tail=ndInput;
            break;
        }
        temp = temp.next;
    } while (temp != null);

}
```

11. Add these following codes to add a node based on defined index

```
public void insertAt(int index, int input){
    if(index < 0){
        System.out.println("Wrong index");
    }else if(index == 0){
        addFirst(input);
    }else{
        Node temp = head;
        for (int i = 0; i < index - 1 ; i++) {
            temp = temp.next;
        }
        temp.next = new Node(input, temp.next);
        if(temp.next.next == null) tail = temp.next;
    }
}
```

12. In class **SLLMain,** create main function and instantiate a new object from **SingleLinkedList** class

```
public class SLLMain {
    public static void main(String[] args) {
        SingleLinkedList singLL=new SingleLinkedList();
```

13. Add methods for inserting data, as well as displaying the data for each insert process so that we can track the changes

```
SingleLinkedList singLL=new SingleLinkedList();
singLL.print();
singLL.addFirst(890);
singLL.print();
singLL.addLast(760);
singLL.print();
singLL.addFirst(700);
singLL.print();
singLL.insertAfter(700, 999);
singLL.print();
singLL.insertAt(3, 833);
singLL.print();
```

1.2.2. Result

Check if the result match with following image:

```
run:
Linked list is empty
Linked list content:    890
Linked list content:    890     760
Linked list content:    700     890     760
Linked list content:    700     999     890     760
Linked list content:    700     999     890     833     760
BUILD SUCCESSFUL (total time: 2 seconds)
```

1.2.3. Questions

1. Why the output of the program in first line is “Linked list is empty”?
2. Please explain the usage of these following codes in:

```
ndInput.next = temp.next;
temp.next = ndInput;
```

3. In **SingleLinkedList,** what is the usage of this following code in **insertAt**?

```
if(temp.next.next==null) tail=temp.next;
```

1.3. 2nd Lab Activities

In this practicum, we will try to learn and implement how to access node elements, get index, and node removal in a Single Linked List

1.3.1. Steps

1. Implement methods to access data and index in linked list
2. Add methods to get data based on certain index from class **SingleLinkedList**

```
public int getData(int index) {
    Node tmp = head;
    for (int i = 0; i < index; i++) {
        tmp = tmp.next;
    }
    return tmp.data;
}
```

3. Implement method **indexOf**

```
public int indexOf(int key) {
    Node tmp = head;
    int index = 0;
    while (tmp != null && tmp.data != key) {
        tmp = tmp.next;
        index++;
    }

    if (tmp == null) {
        return -1;
    } else {
        return index;
    }
}
```

4. Add method **removeFirst()** in class **SingleLinkedList**

```
public void removeFirst(){
    if(isEmpty()){
        System.out.println("Linked list is empty. cannot remove a data");
    }else if(head == tail){
        head = tail = null;
    }else{
        head = head.next;
    }
}
```

5. Add this method to remove data that is in the last of the list from class **SingleLinkedList**

```
public void removeLast(){
    if(isEmpty()){
        System.out.println("Linked list is empty. cannot remove a data");
    }else if(head == tail){
        head = tail = null;
    }else{
        Node temp = head;
        while (temp.next != tail) {
            temp = temp.next;
        }
        temp.next = null;
        tail = temp;
    }
}
```

6. Next, we will implement method **remove()**

```
public void remove(int key){
    if(isEmpty()){
        System.out.println("Linked list is empty. cannot remove a data");
    }else{
        Node temp = head;
        while(temp != null){
            if((temp.data == key) && (temp==head)){
                this.removeFirst();
                break;
            }else if(temp.next.data == key){
                temp.next = temp.next.next;
                if(temp.next == null){
                    tail = temp;
                }
                break;
            }
            temp = temp.next;
        }
    }
}
```

7. Create a method to remove a node based on defined index

```
public void removeAt(int index) {
    if (index == 0) {
        removeFirst();
    } else {
        Node temp = head;
        for (int i = 0; i < index - 1; i++) {
            temp = temp.next;
        }
        temp.next = temp.next.next;
        if (temp.next == null) {
            tail = temp;
        }
    }
}
```

8. Next, we will try to access and remove data in main method in class **SLLMain** by adding these codes

```
System.out.println("Data in lst index : " + singLL.getData(1));
System.out.println("Data 3 is in index : " + singLL.indexOf(760));
singLL.remove(999);
singLL.print();
singLL.removeAt(0);
singLL.print();
singLL.removeFirst();
singLL.print();
singLL.removeLast();
singLL.print();
```

9. Method **SLLMain** becomes like this:

```
public class SLLMain {
    public static void main(String[] args) {
        SingleLinkedList singLL = new SingleLinkedList();
        singLL.print();
        singLL.addFirst(890);
        singLL.print();
        singLL.addLast(760);
        singLL.print();
        singLL.addFirst(700);
        singLL.print();
        singLL.insertAfter(700, 999);
        singLL.print();
        singLL.insertAt(3, 833);
        singLL.print();

        System.out.println("Data in lst index : " + singLL.getData(1));
        System.out.println("Data 3 is in index : " + singLL.indexOf(760));
        singLL.remove(999);
        singLL.print();
        singLL.removeAt(0);
        singLL.print();
        singLL.removeFirst();
        singLL.print();
        singLL.removeLast();
        singLL.print();
    }
}
```

10. Execute the class **SLLMain**

1.3.2. Result

Check if the result match with following image:

```
run:
Linked list is empty
Linked list content:    890
Linked list content:    890     760
Linked list content:    700     890     760
Linked list content:    700     999     890     760
Linked list content:    700     999     890     833     760
Data in lst index : 999
Data 3 is in index : 4
Linked list content:    700     890     833     760
Linked list content:    890     833     760
Linked list content:    833     760
Linked list content:    833
BUILD SUCCESSFUL (total time: 2 seconds)
```

1.3.3. Questions

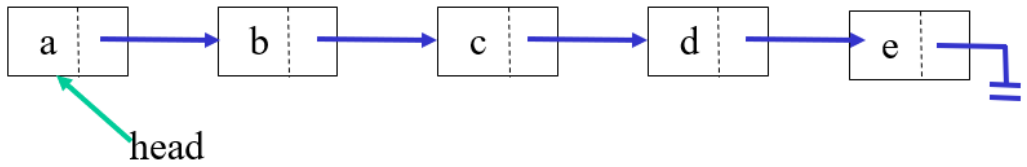
1. Why we use **break** keyword in remove function? Please explain
2. Please explain why we implement these following codes in method remove

```
else if (temp.next.data == key) {
   temp.next = temp.next.next;
```

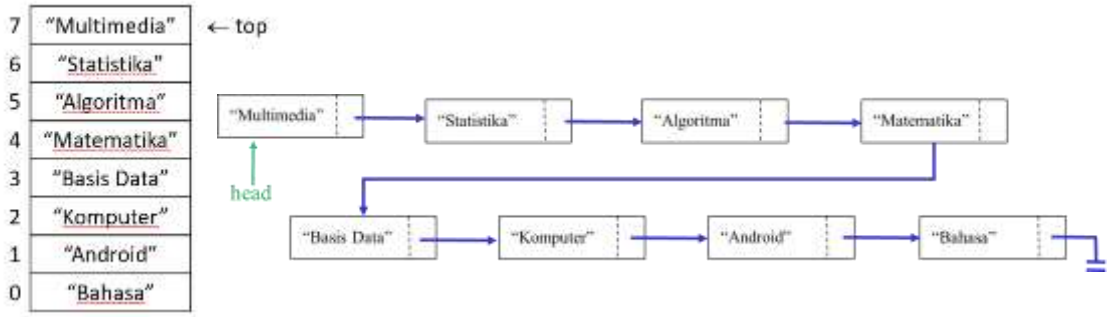
3. What are the outputs of method indexOf? Please explain each of the output!

1.4. Assignments

1. Create a method **insertBefore()** to add node before the desired keyword
2. Implement the linked list from this following image. You may use 4 method of adding data we've learnt



3. Create this following **Stack** implementation using Linked List implementation



4. Create a program that helps bank customer using linked list with data are as follows: Name,address, and customerAccountNumber

5. Implement **Queue** in previous number with **linked list** concept