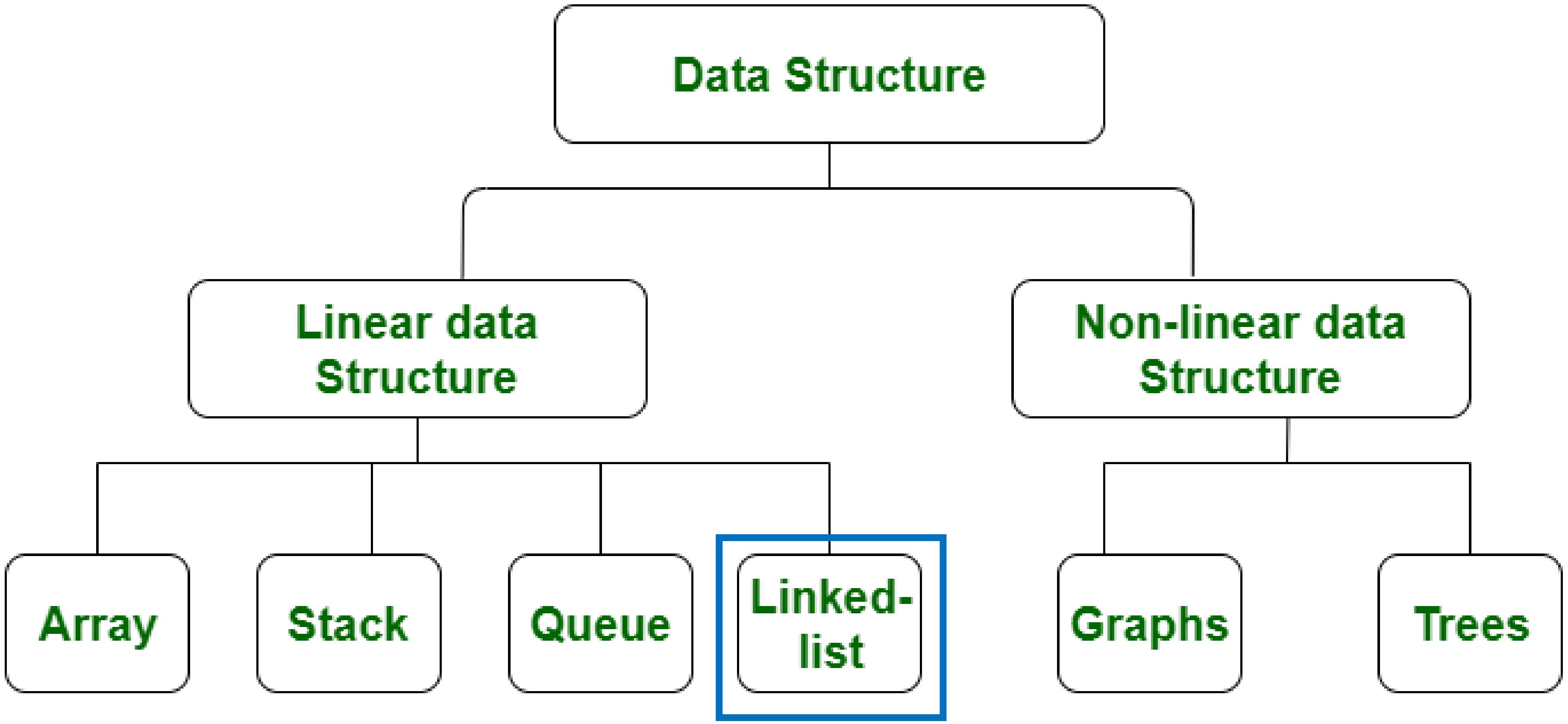
Linked List

Algorithm and Data Structure Teaching Team
2022/2023


Data Structure
Linear data Structure
Non-linear data Structure
Array
Stack
Queue
Linked-list
Graphs
Trees

Learning Outcome

- Students must be able to understand basic concept of linked list
- Students must be able to have a good knowledge of steps for implementing linked list to solve a problem

Introduction

- *linked list* brings a solution for the limitation of *array* that has static in size
- *Array* will create memories on the declaration. The memory allocation will still remains even there is no data inside.

Definition

- Linked list : linear data structure consists of one or more interconnected nodes that occupy memory allocations dynamically
- **Node** : a place to store the data, that consists of 2 attribute/**field**.
- **Field** 1 for **Data**, that will store the value of data.
- **Field** 2 for **Pointer**, that will save a reference address that points to other node .
  - Well known as **link** → it will create a link to connect to other node

Definition

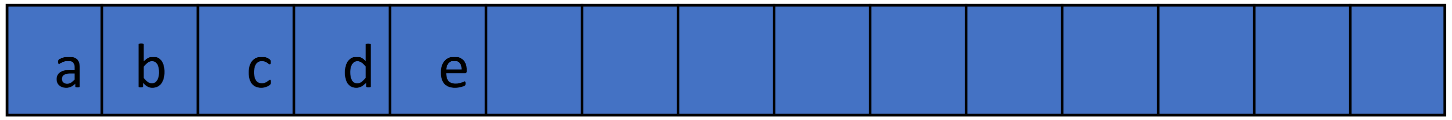
- If there is only one **Node** inside a linked list, then the pointer is NULL
- If there are more than one Node inside a linked list, then the Pointer will save the reference address of the next Node. It means that this Pointer connects a Node to the Next Node.
- The last Node will refer to Null (null represents no value/nothing/unset reference).
- The first Node of a *Linked List* called as ***head***.
- The last Node of a Linked List refered as **tail**
- **Note**: *head* and *tail* is not a different node, but it only a reference name that refer to a node located at the first position and the last position.
- If *Linked List* is empty, then *head* and tail refer to *null*.

Array VS Linked List

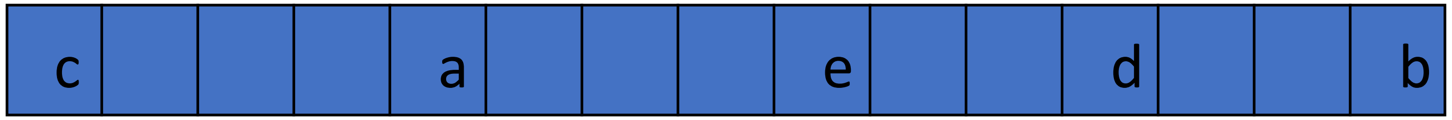
ARRAY	LINKED LIST
Static	Dynamic
Addition/Deletion is limited	Addition/Deletion is not limited
Random Access	Sequential access
Homogeneous element (each element must be in the same type)	Nodes are connected via a pointer to the next node. So it doesn't have to have a homogeneous data type

Array VS Linked List

- Linked list managed the elements non-contiguously.
    - Elements can be located in remote memory locations. Compare this with an array where each element will be located in a sequential memory location.



Array representation



Linked list representation

Array VS Linked List

- Allows adding or removing elements in the middle of a collection requiring only a constant number of element moves.
  - Compare with arrays. How many elements must be moved when inserting an element in the middle of the array?

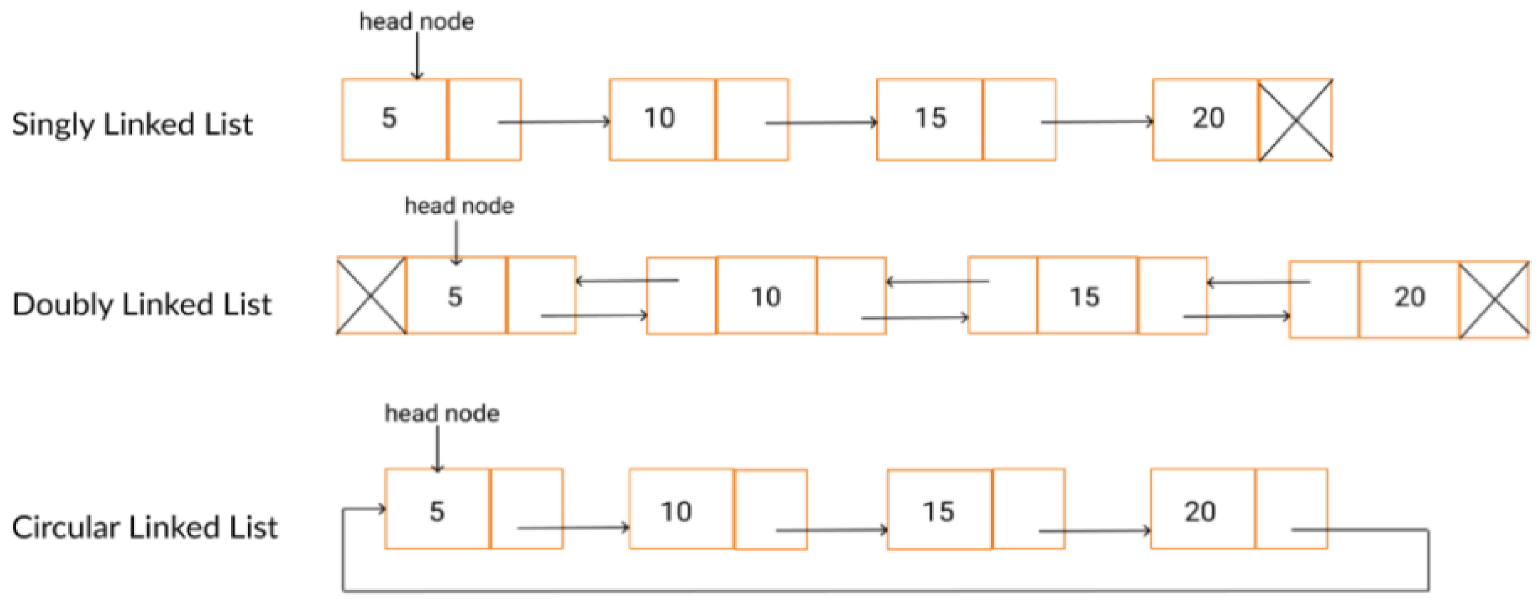
Pros & Cons of Linked Lists

- **Pros:** Dynamic data structure, the number of nodes can increase according to data needs.

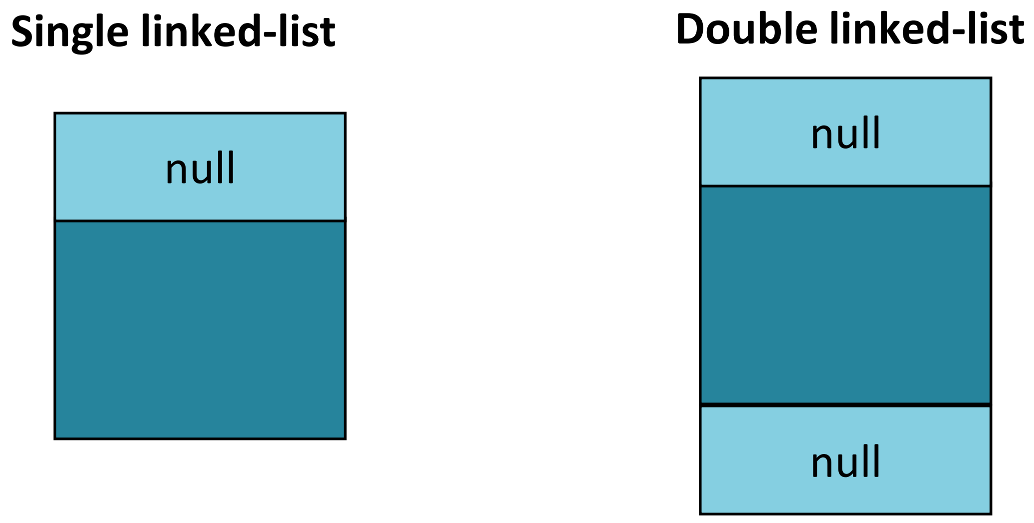
- **Cons:** This data structure cannot access data by index. If this approach is needed, it is necessary to process the head and follow the next pointer until the desired data / index is obtained.

Types of Linked List

- Single Linked List: Only has one pointer (**next**) that points to the next node
- Double Linked List: has 2 pointers, **next** will point to the next node and **prev** will point to the previous node
- Circular linked list: the last node's pointer will point to the first node



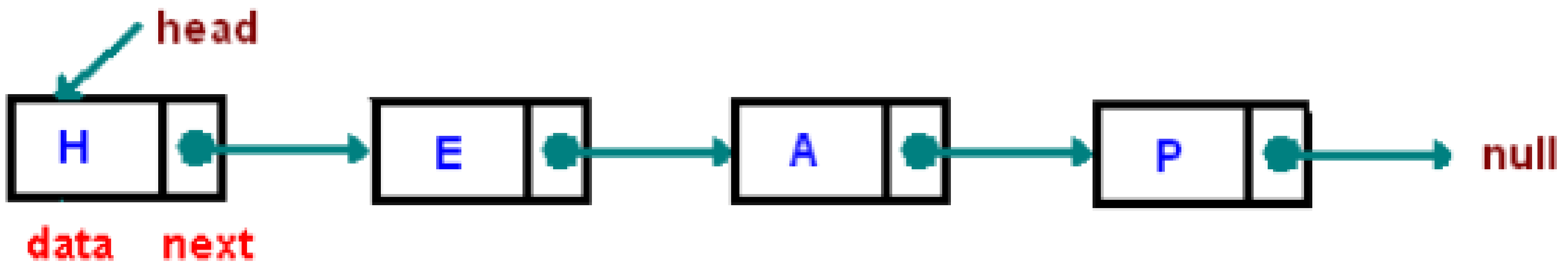
Node



🟦	**Link / pointer**
■	**data**

Single Linked List Basic Concept

- *Linked List* dynamic data structure.
- Number of *node* could be added as well as deleted per the needs.
- Program that will manage the data with unknown size/number of data, it will be beter to implement *Linked List*.



Single Linked List Basic Concept

- Single: there is only 1 pointer in a node. it points to the next node.
- The last node will point to NULL which will be used as a stop condition when reading the contents of the linked list.
- Linked List does not use memory cells in a row (row). However, it makes use of random memory.
- Then how does the computer know that the nodes are the same linked lists?
- The key is the data that is stored in the node, each node also stores the memory address for the next node in a linked list.

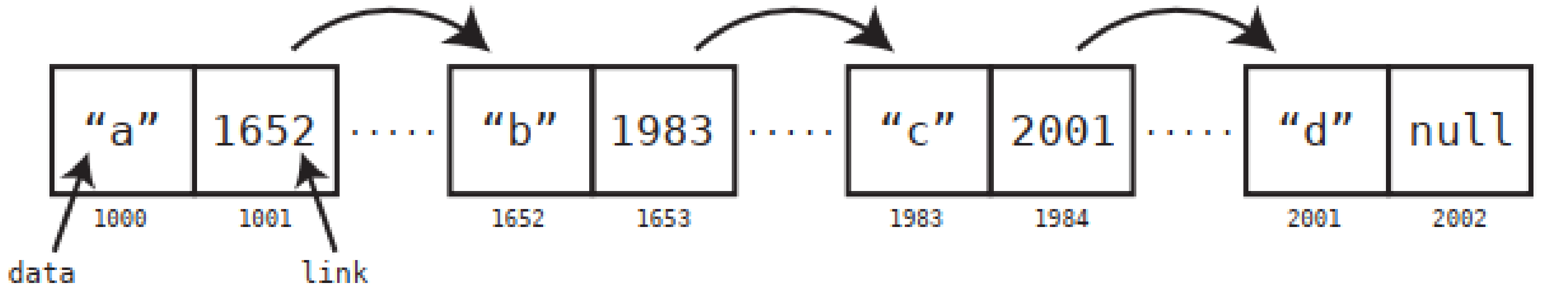
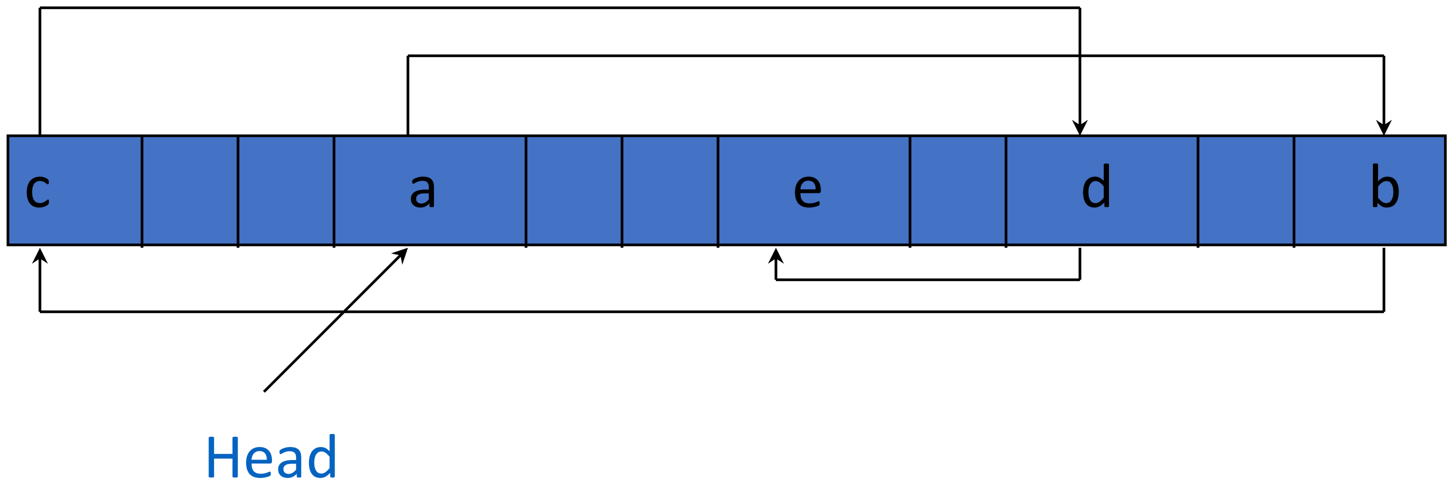


Illustration of Single Linked List

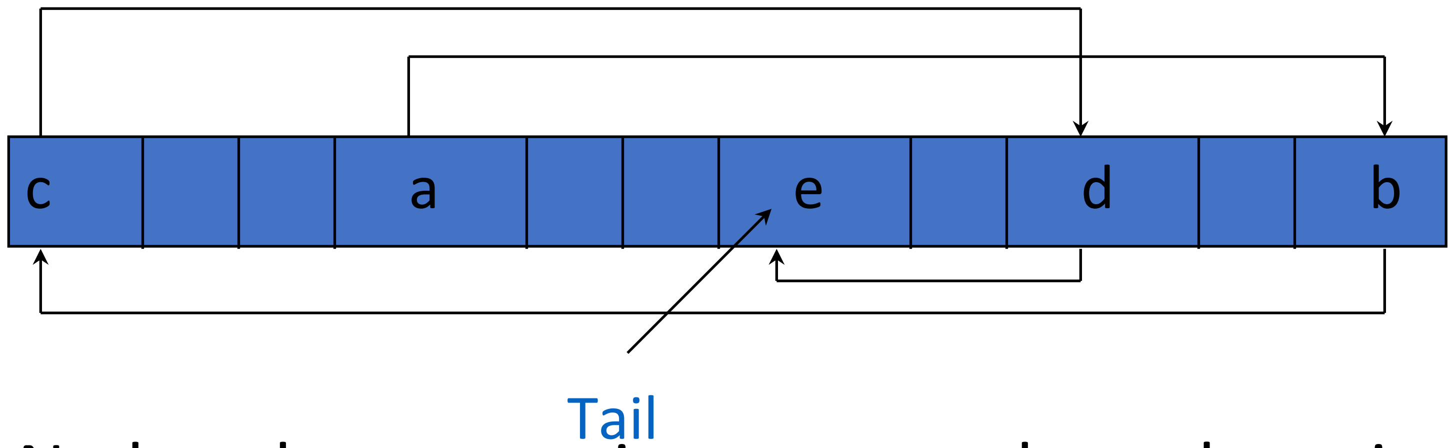
- Ilustration of single linked list in memory :



- Since the node is not pointed to by any node, it is the most front node (head node).

Illustration of Single Linked List

- Illustration of Single Linked List:

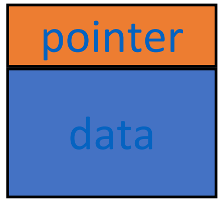


- Node e does not point to any node so the pointer of node e is NULL. It can be concluded that this node is the backmost node (tail node).

“Single Node” Representation

```
class Node
{
   String data;
   Node pointer;
}
```

Ilustration :

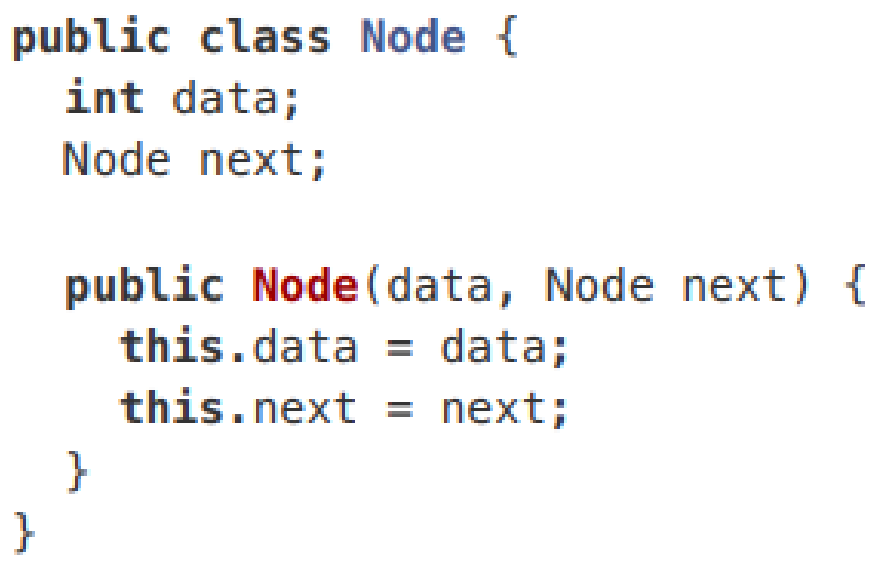


Note:

- Creating a class called Node which contains 2 fields / attributes, (1) **data** →String data type and (2) **pointer** → Node class data type
- **Data** field: used to store data / values in the linked list. **pointer**: used to store the address of the next node.

Implementation of Linked Lists (Node)

- To represent data elements, a node is required. The implementation in Java is as follows :

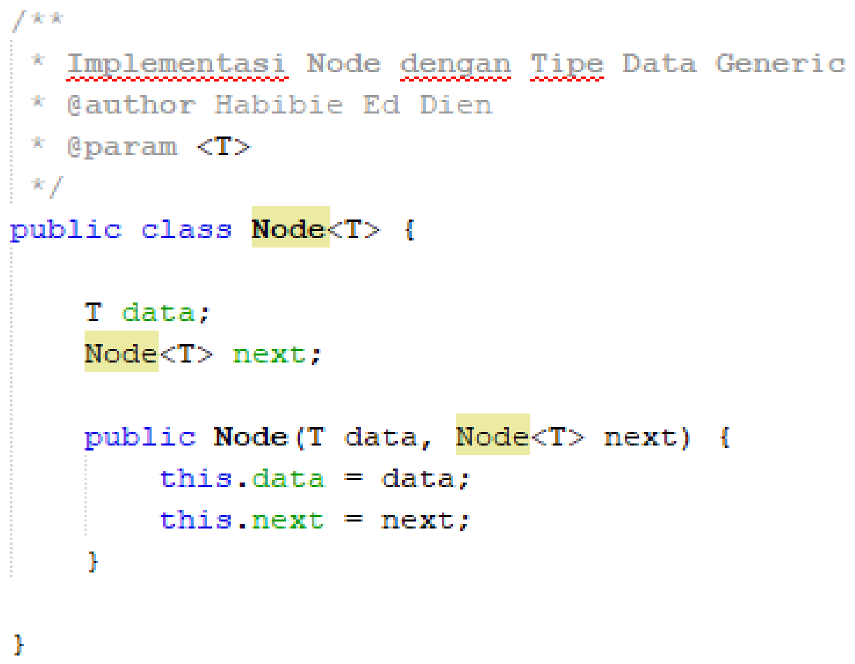


```
public class Node {
  int data;
  Node next;

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

- There are two main attributes on the node, namely “**data**” and the “**next**” pointer that connects with the next data.

Implementation of Linked Lists (Node) using Generic Data Type

- For more flexible storage of data types, **generic** concepts can be considered. Find the example at the following code.

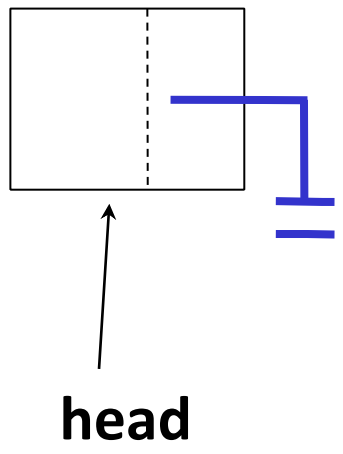


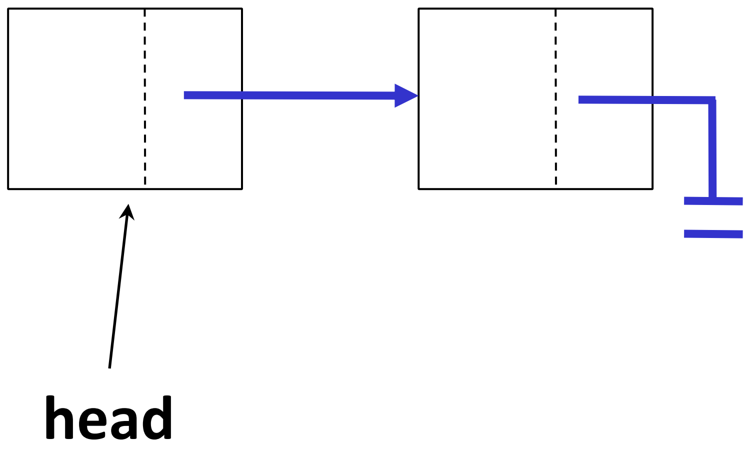
```
/**
 * Implementasi Node dengan Tipe Data Generic
 * @author Habibie Ed Dien
 * @param <T>
 */
public class Node<T> {

    T data;
    Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

}
```

- Nodes can accept various data types: Integer, Float, String, Boolean and etc

Head Pointer

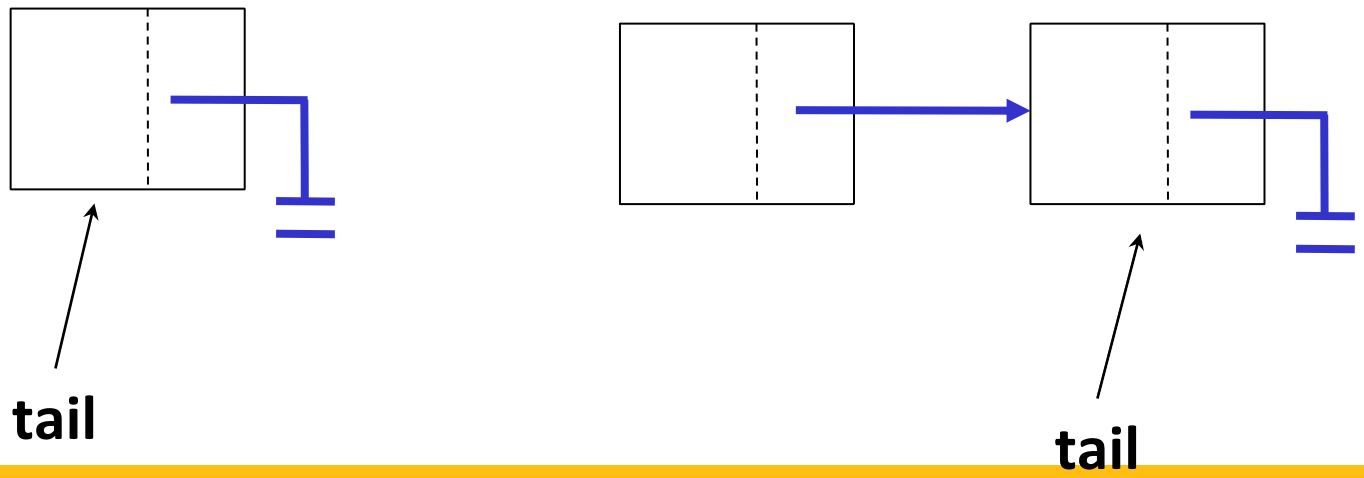
- To refer to the first node (**head** node) a pointer is used which stores the address of the first node.
- This pointer is usually given the name **head**.





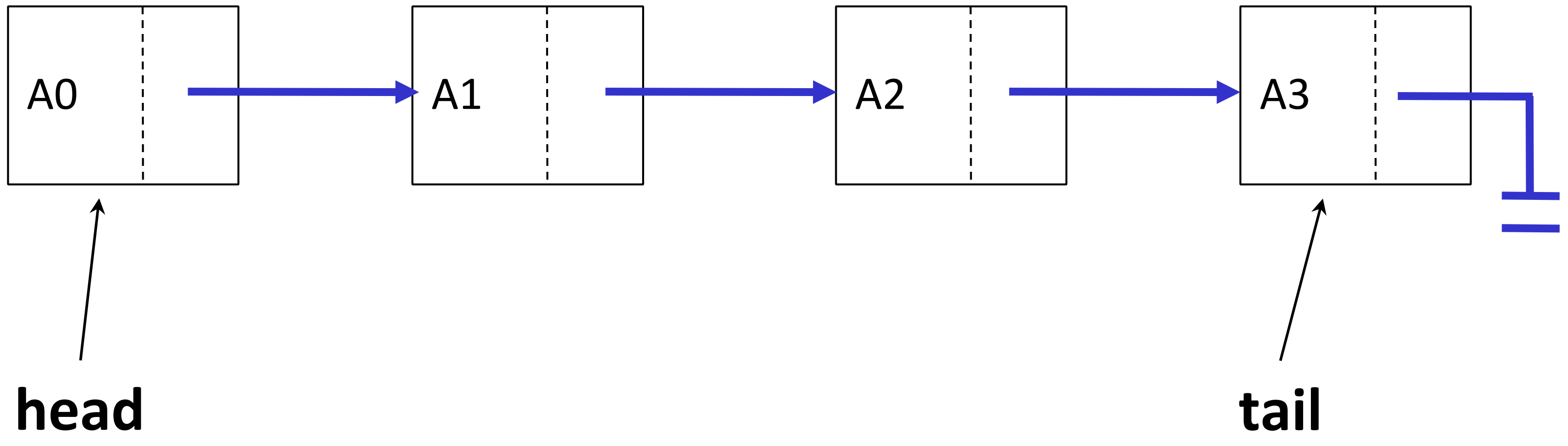
Tail Pointer

- To refer to the last node (**tail** node) a pointer is used which stores the address of the last node.
- This pointer is usually given the name **tail**.



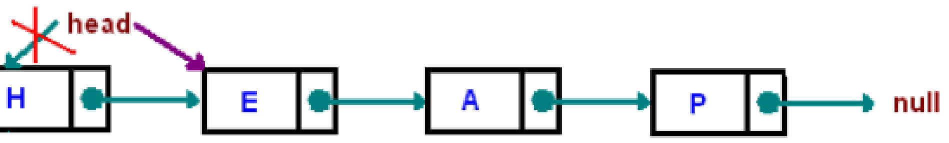
Example

- Linked list has 4 nodes :

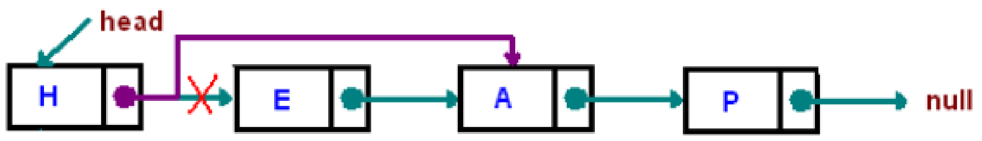


How Linked List Works

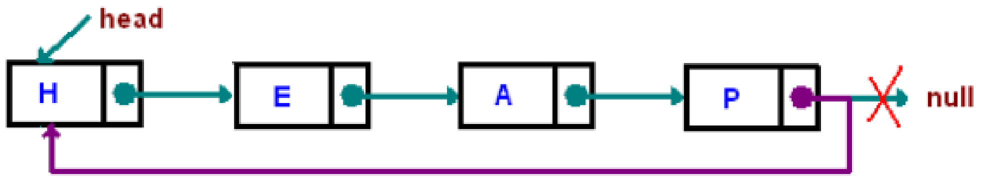
```
head = head.next;
```



```
head.next = head.next.next;
```



```
head.next.next.next.next = head;
```



Operation of Linked Lists

- **isEmpty**: to check whether **head == null** (kosong).
- **Print**: print all elements of Linked Lists.
- **Add**:
  - AddFirst
  - AddLast
  - InsertAfter
- **Remove**:
  - RemoveFirst
  - RemoveLast
  - Remove
- Operation of Linked List on a certain position/index
  - Accessing data
  - Accessing index
  - Add data
  - Remove data

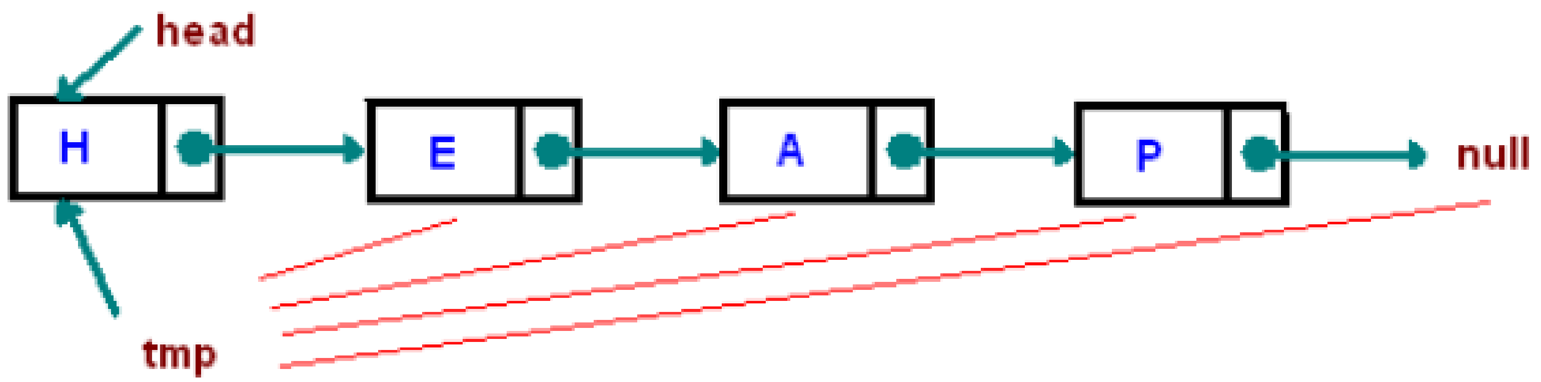
isEmpty()

- To check whether the linked list is empty or not.
- Empty → **head=tail=null** or **size == 0**

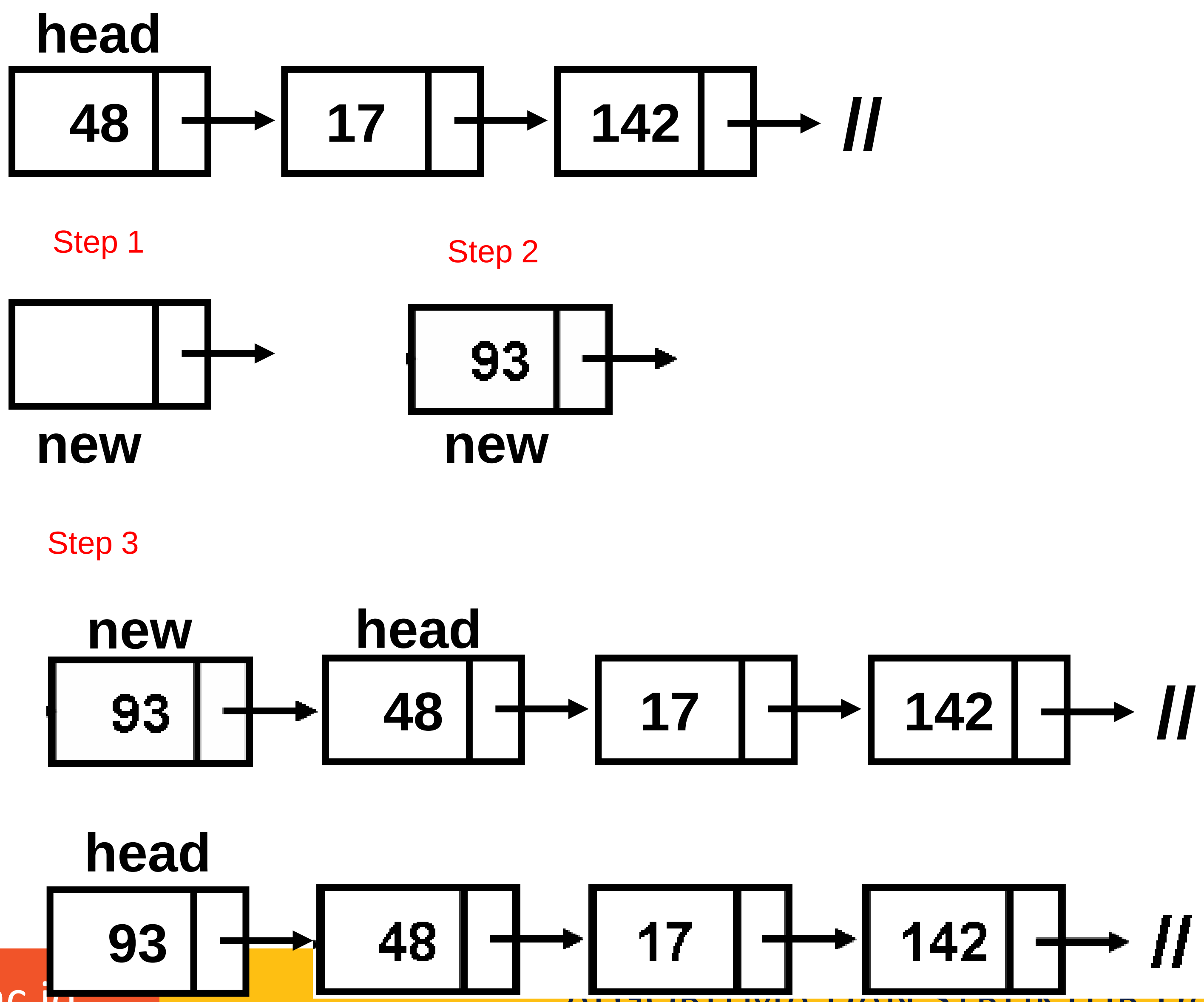
```
 boolean isEmpty()
    {
          return size==0;
    }
```

Treverse on a Linked List

- The process of visiting each node exactly once. By making a complete visit, you will get a linear sequence of information stored in the Linked List.
- This process is carried out in data printing operations, adding data at the end of Linked Lists and accessing Linked Lists using indexes
- This process starts from the beginning of the data (head) until it reach the last node that points to null. This process will not change the reference of the head.



AddFirst

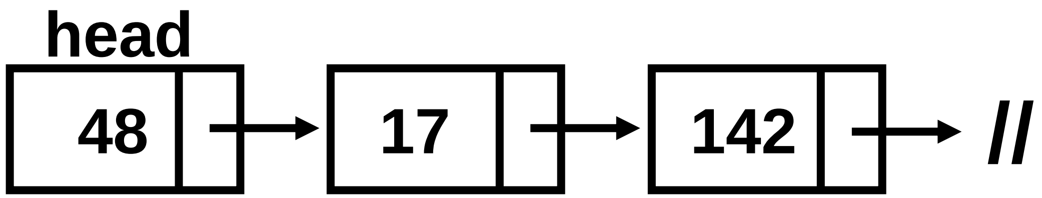

head
48
17
142
//
Step 1
Step 2
new
93
new
Step 3
new
head
93
48
17
142
//
head
93
48
17
142
//

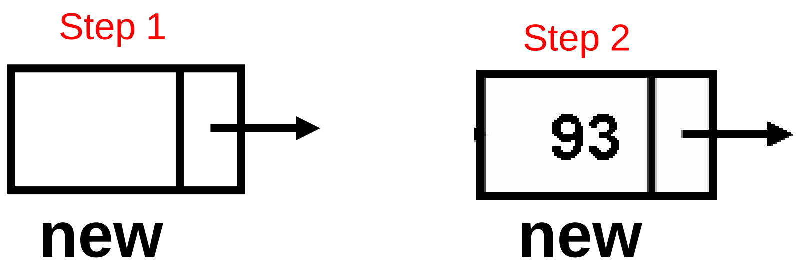
AddFirst

- Step 1 : create a **new** node
- Step 2 : input the data into **new** node
- Step 3 :
  - Point the **next** pointer of **new** node to the **head** node
  - Set the **new** node as the new **head** node

```
new.next = head
head = new;
```

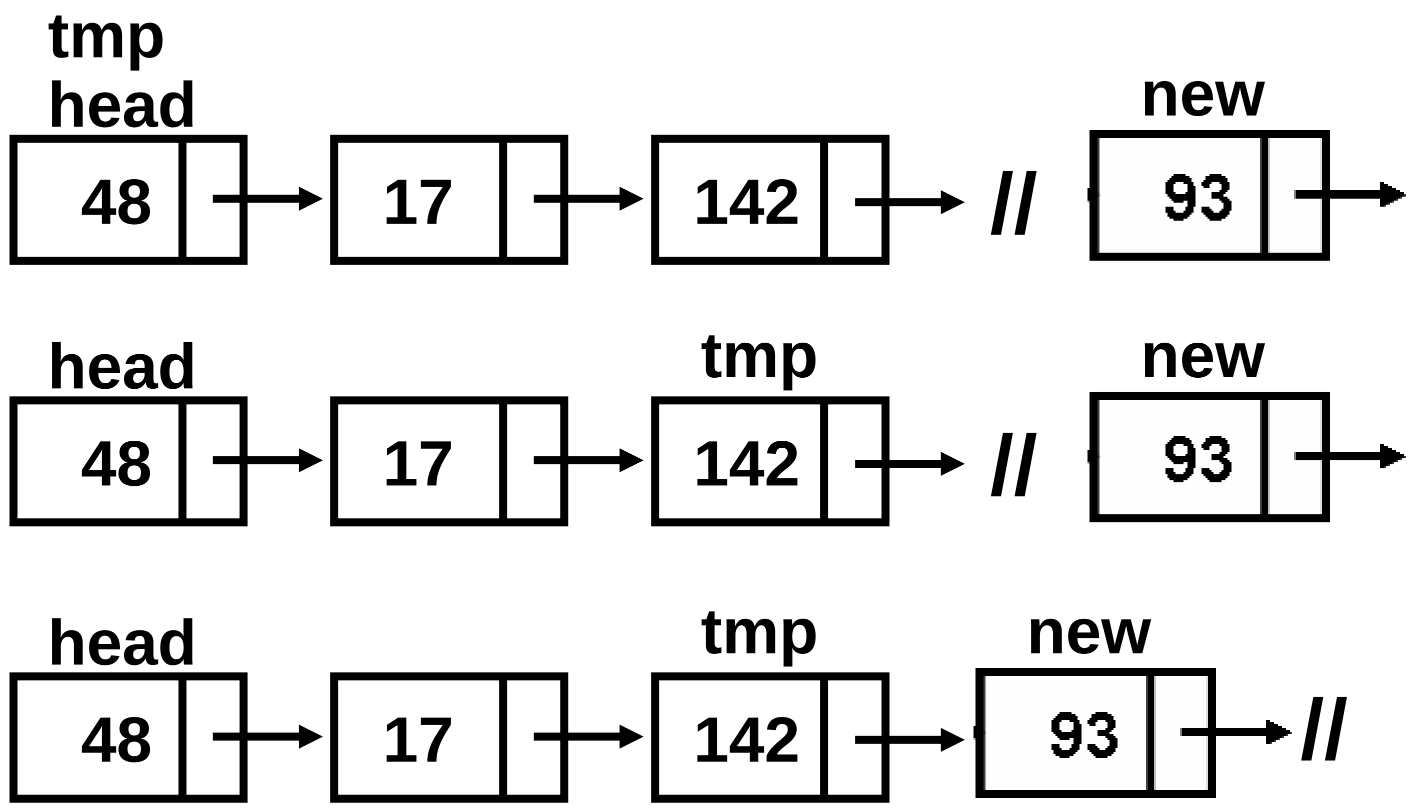
AddLast





AddLast

Step 3

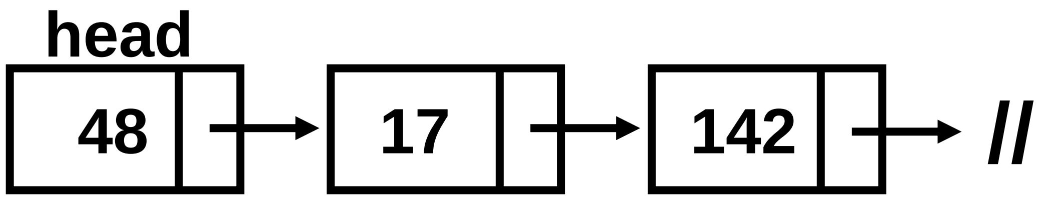


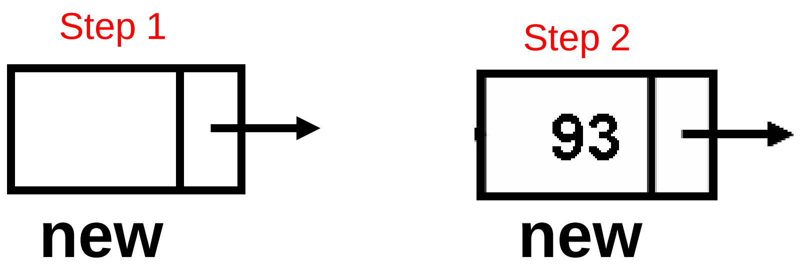
AddLast

- Step 1 : create a **new** node
- Step 2 : input the data into **new** node
- Step 3 :
  - Declare a **tmp** node and initialize it to **head**
  - By using loop, shift **tmp** until it is located at the last node
  - Point the **next** pointer of **tmp** node to the **new** node
  - Point the **next** pointer of **new** node to **null**

```
Node tmp = head;
while(tmp.next!=null){
    tmp = tmp.next;
}
tmp.next = new;
new.next = null;
```

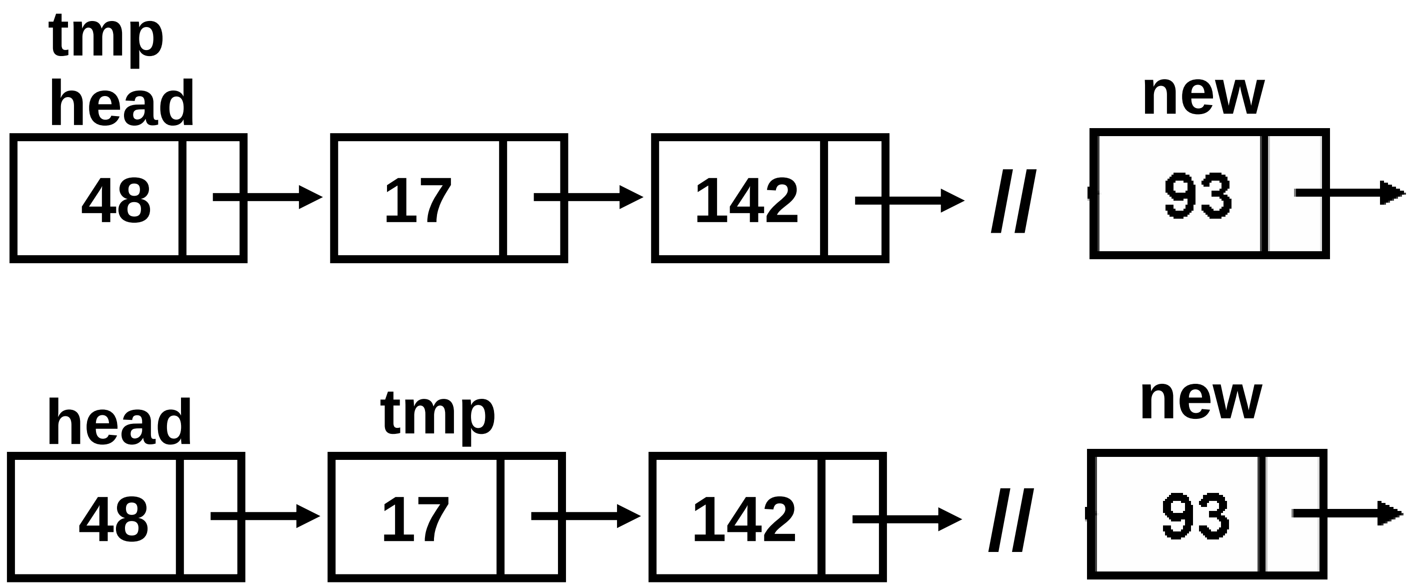
AddMiddle (Index,e)





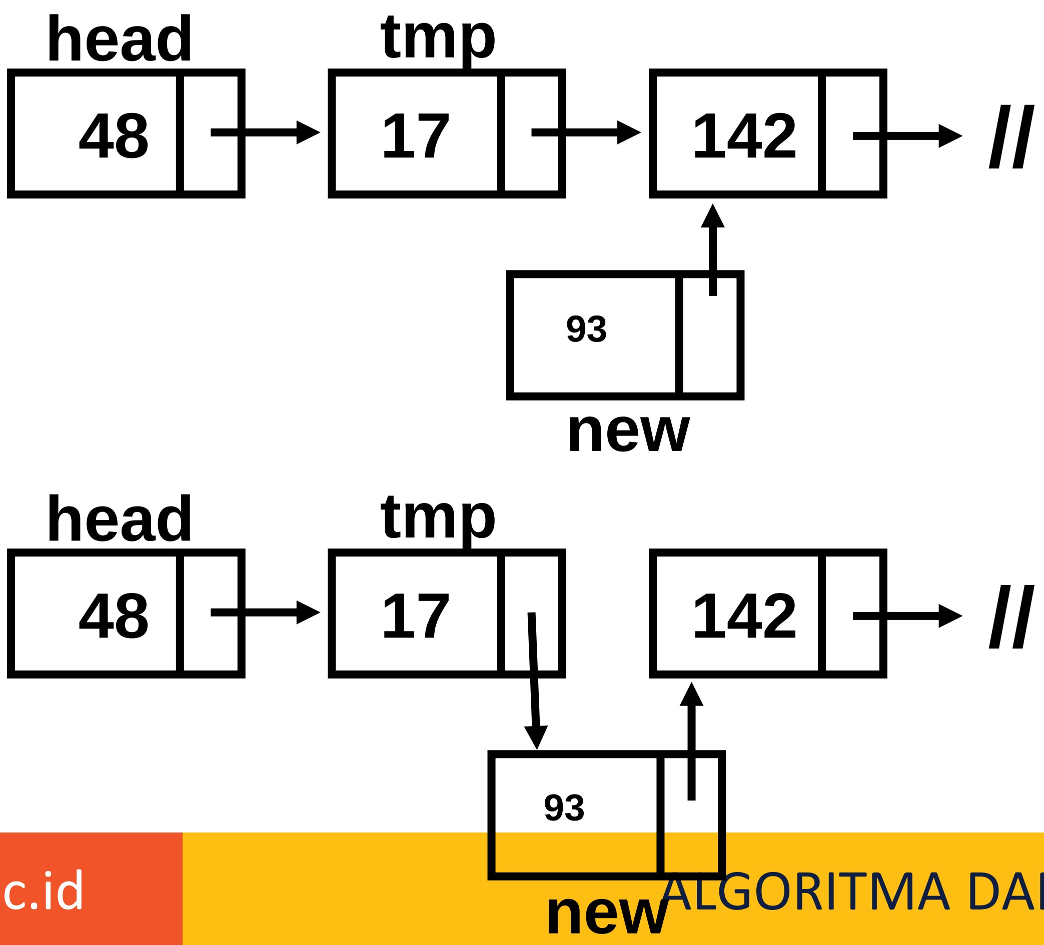
AddMiddle (Index,e)

Step 3



AddMiddle (Index,e)

Step 4

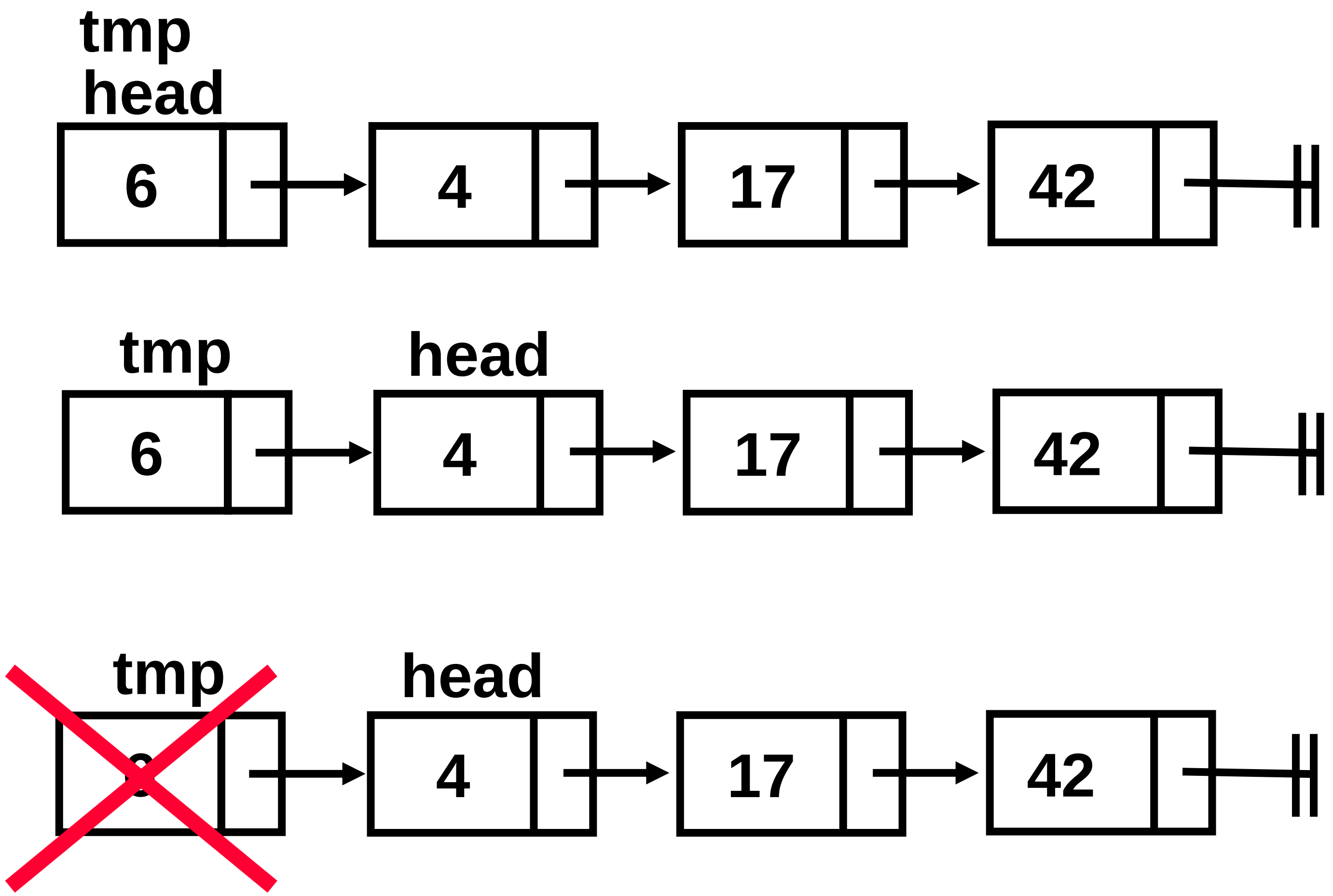


AddMiddle

- Step 1 : create a **new** node
- Step 2 : input the data into **new** node
- Step 3 :
  - Declare a **tmp** node and initialize it to **head**
  - By using loop, shift **tmp** until it is located at the needed location
- Step 4
  - Point the **next** pointer of **new** node to the node pointed by **next** pointer of **tmp**
  - Point the **next** pointer of **tmp** node to **new** node

```
Node tmp = head;
int currentPosition = 0;
while(currentPosition<neededPosition){
     tmp = tmp.next;
}
new.next = tmp.next;
tmp.next = new;
```

RemoveFirst

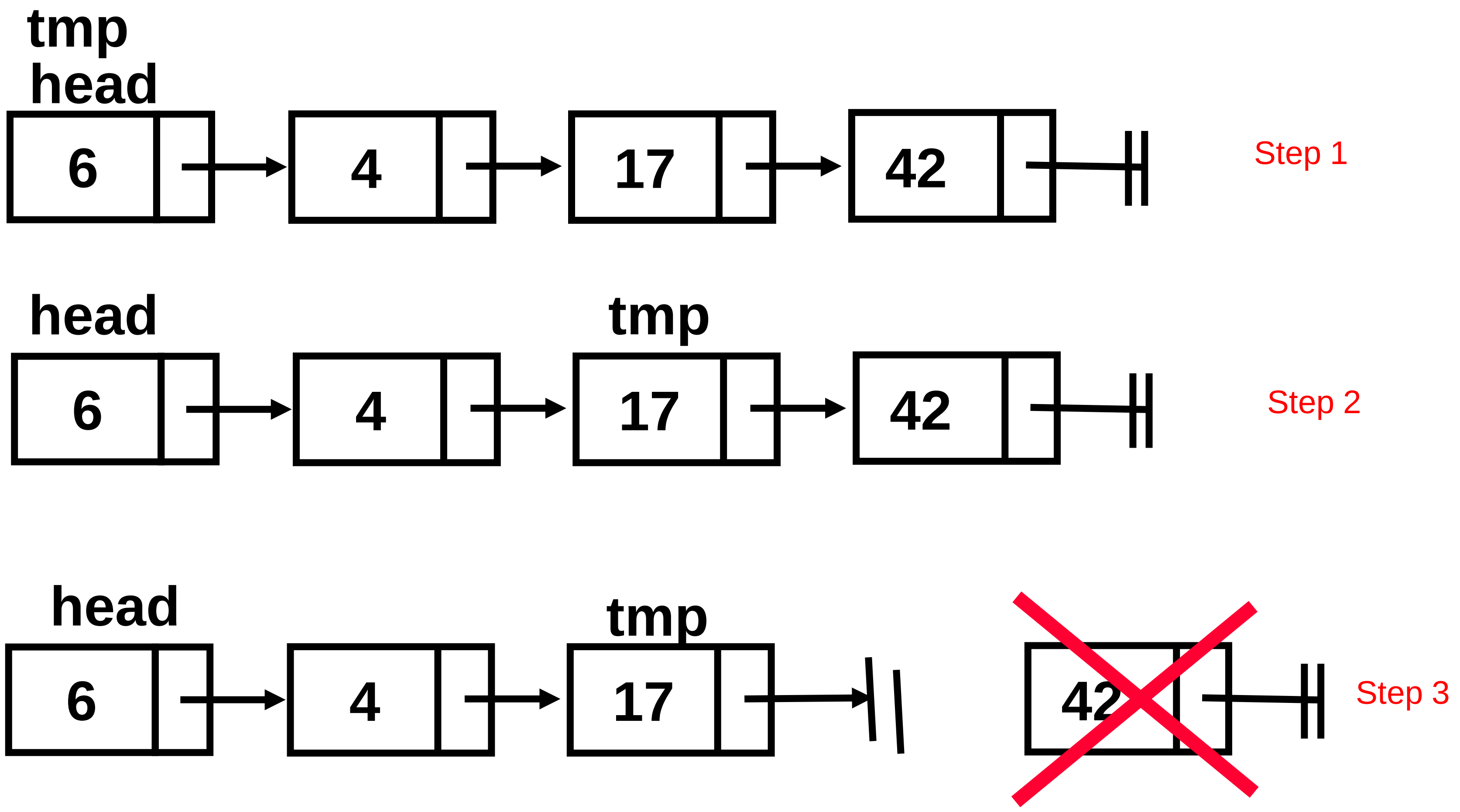


RemoveFirst

- Step 1 : create a **tmp** node and initialize it to **head** node
- Step 2 : move **head** node to the **next** node of **tmp**
- Step 3 : remove **tmp**

```
Node tmp = head;
head = tmp.next;
tmp = null;
```

RemoveLast

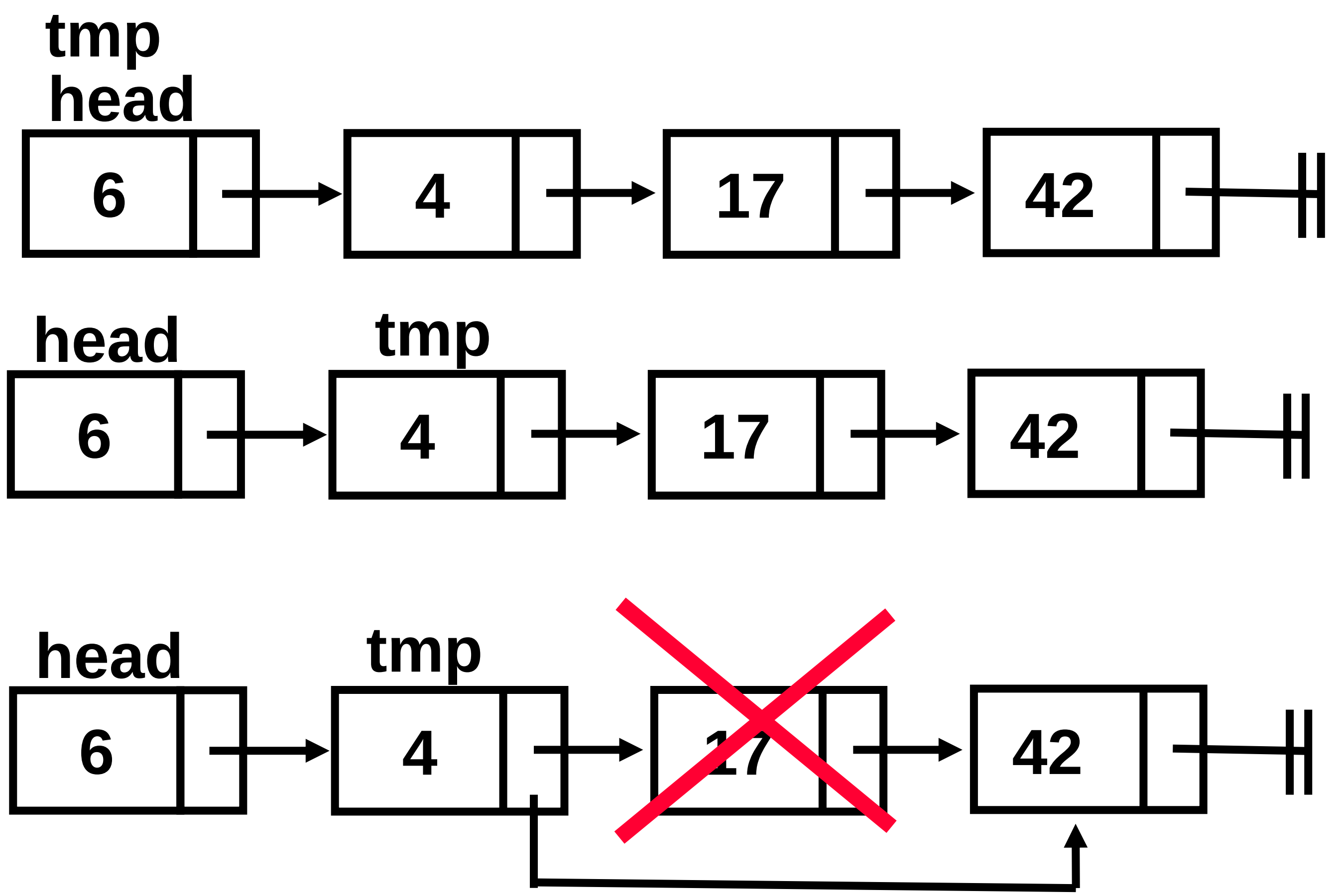

tmp
head
6
4
17
42
Step 1
head
tmp
6
4
17
42
Step 2
head
tmp
6
4
17
42
Step 3

RemoveLast

- Step 1 : create a **tmp** node and initialize it to **head** node
- Step 2 : by using loop, move **tmp** node until it is located at a node before the last node
- Step 3 : point **next** pointer ot **tmp** to the **null**

```
Node tmp = head;
while(tmp.next.next!=null){
    tmp = tmp.next;
}
tmp.next = null;
```

RemoveMiddle (index,e)



Step 1

Step 2

Step 3

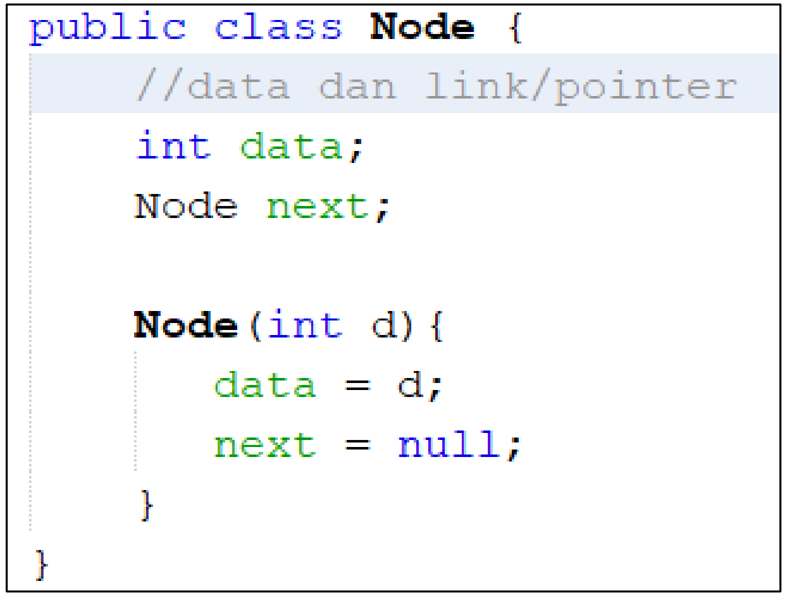
RemoveLast

- Step 1 : create a **tmp** node and initialize it to **head** node
- Step 2 : by using loop, move **tmp** node until it is located at a node **before** the **key**
- Step 3 : point **next** pointer of **tmp** to **2 node after**

```
Node tmp = head;
while(tmp.next.data!=key){
    tmp = tmp.next;
}
tmp.next = tmp.next.next;
```

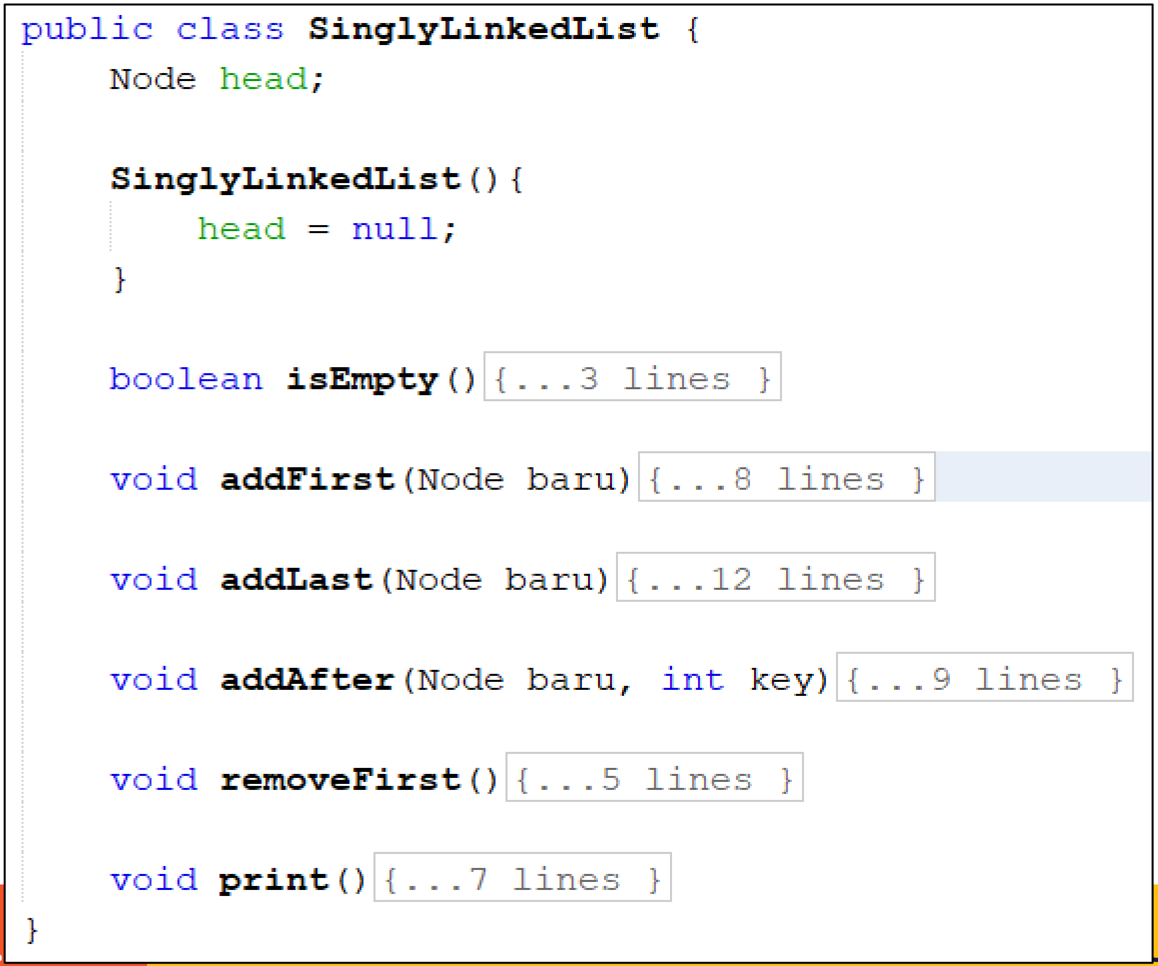
Singly Linked List Implementation

- Class Node


```java
public class Node {
    //data dan link/pointer
    int data;
    Node next;

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

Singly Linked List Implementation

- Class SinglyLinkedList

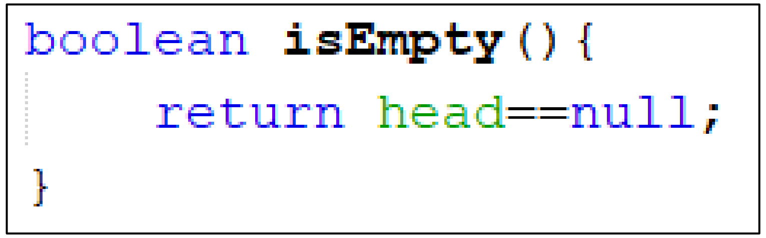


```
public class SinglyLinkedList {
    Node head;

    SinglyLinkedList(){
        head = null;
    }

    boolean isEmpty(){...3 lines }

    void addFirst(Node baru){...8 lines }

    void addLast(Node baru){...12 lines }

    void addAfter(Node baru, int key){...9 lines }

    void removeFirst(){...5 lines }

    void print(){...7 lines }
}
```

Methods

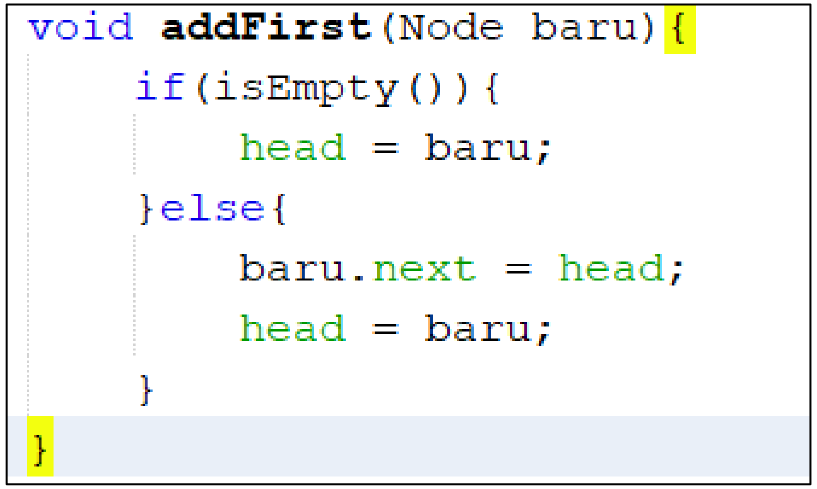
- isEmpty()
- addFirst()
- addLast()
- insertAfter()
- insertBefore()
- removeFirst()
- removeLast()
- remove()
- find()
- printNode()

isEmpty()



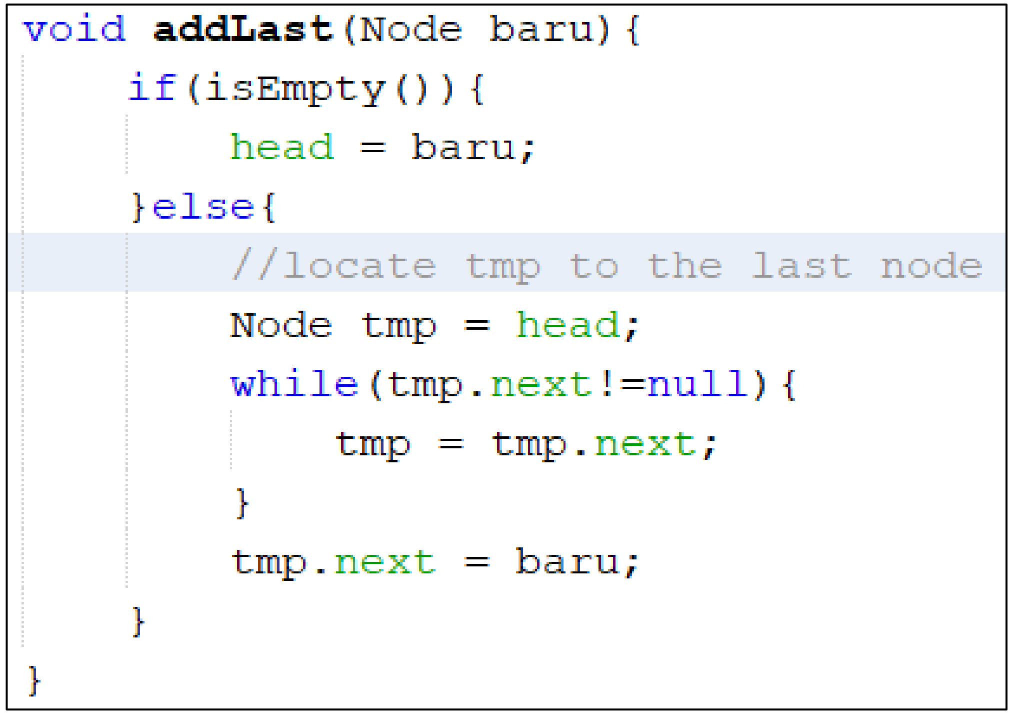
```
boolean isEmpty(){
    return head==null;
}
```

addFirst()



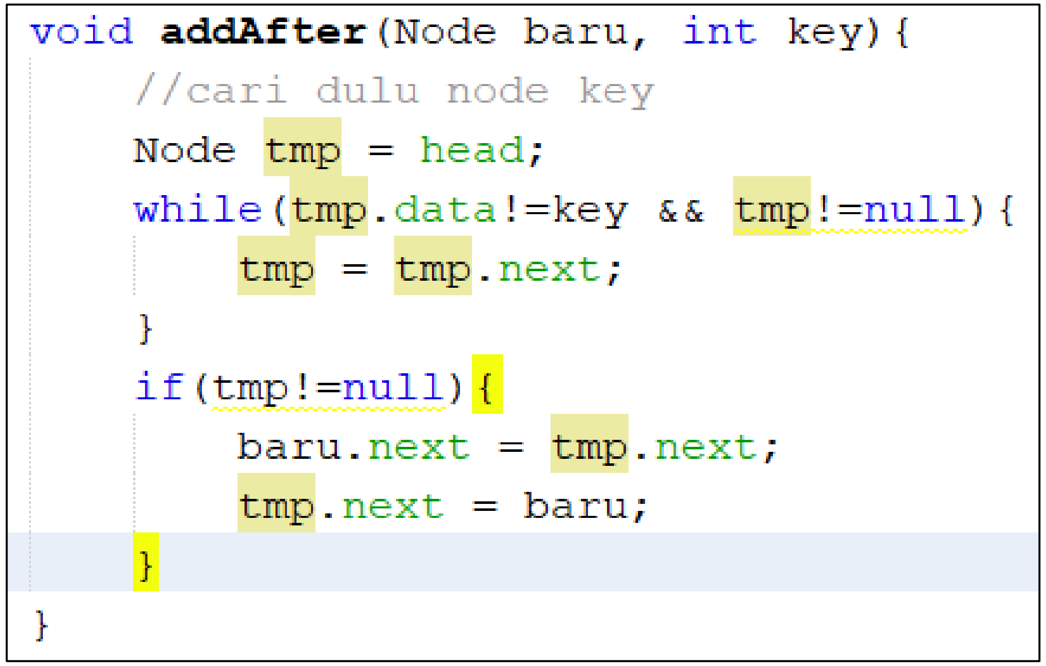
```
void addFirst(Node baru){
    if(isEmpty()){
        head = baru;
    }else{
        baru.next = head;
        head = baru;
    }
}
```

addLast()

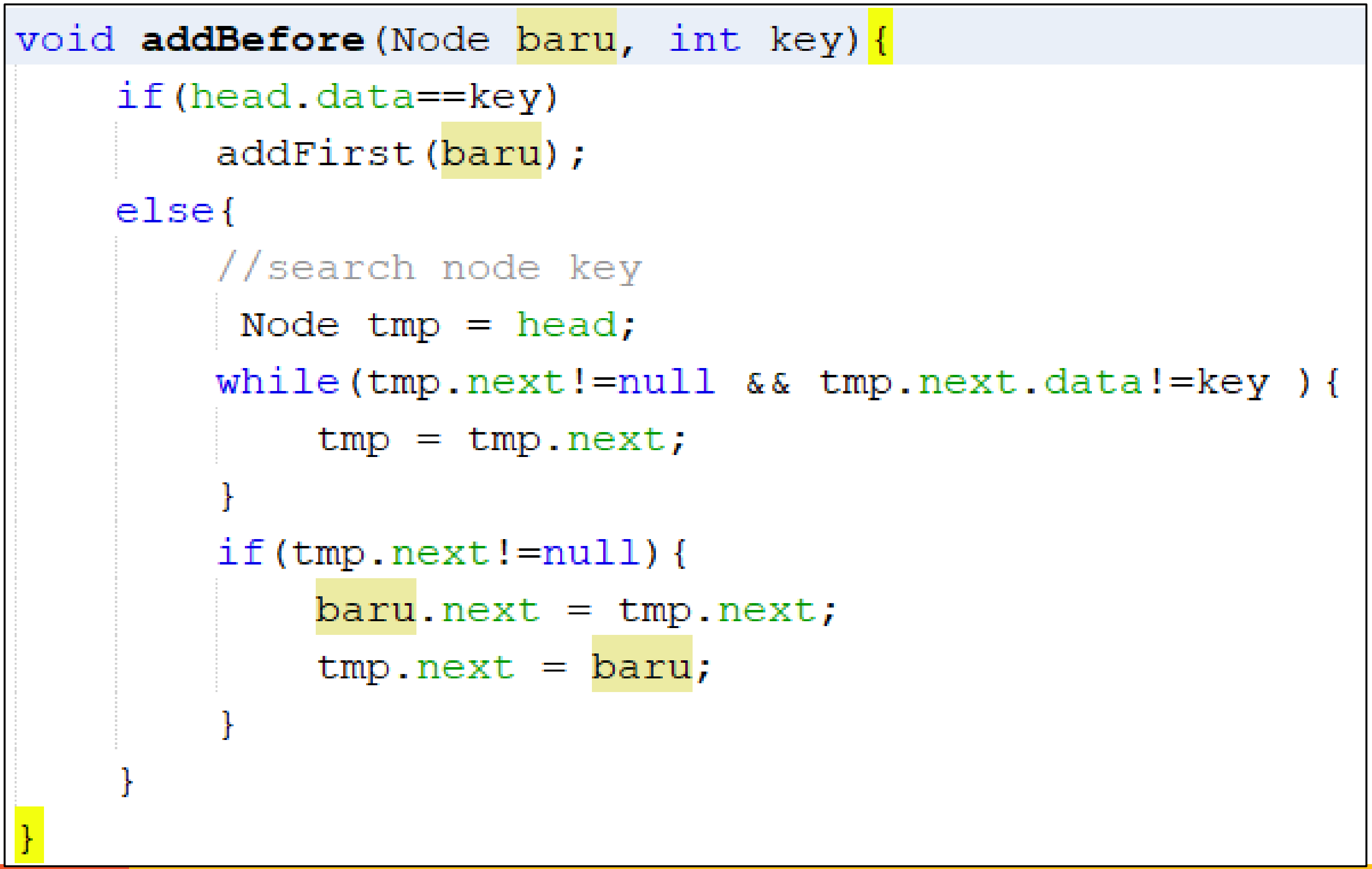


```
void addLast(Node baru){
    if(isEmpty()){
        head = baru;
    }else{
        //locate tmp to the last node
        Node tmp = head;
        while(tmp.next!=null){
            tmp = tmp.next;
        }
        tmp.next = baru;
    }
}
```

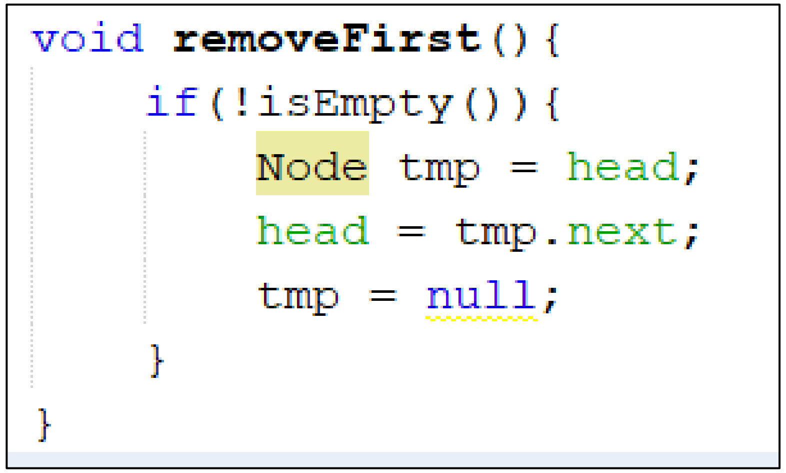
insertAfter()


```
void addAfter(Node baru, int key){
    //cari dulu node key
    Node tmp = head;
    while(tmp.data!=key && tmp!=null){
        tmp = tmp.next;
    }
    if(tmp!=null){
        baru.next = tmp.next;
        tmp.next = baru;
    }
}
```

insertBefore()

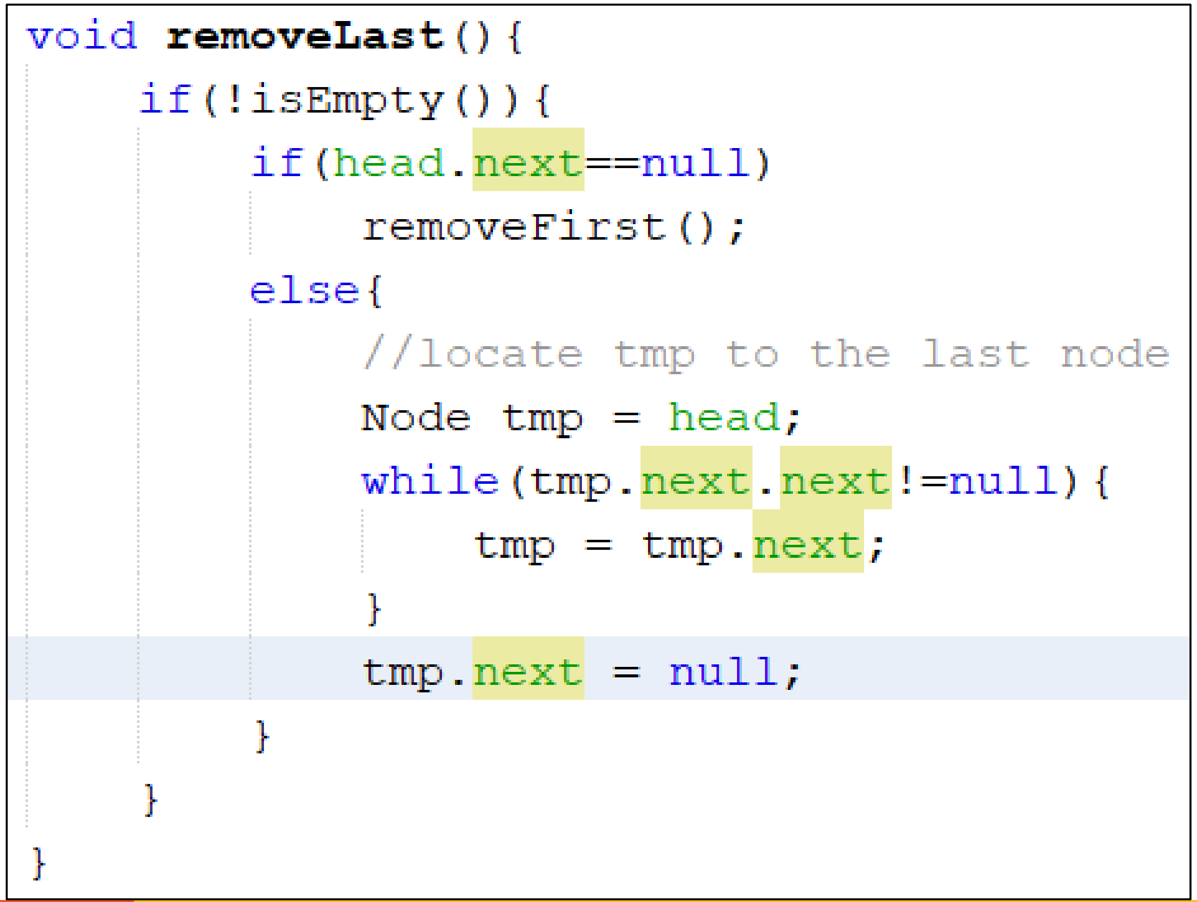

```
void addBefore(Node baru, int key){
    if(head.data==key)
        addFirst(baru);
    else{
        //search node key
         Node tmp = head;
        while(tmp.next!=null && tmp.next.data!=key ){
            tmp = tmp.next;
        }
        if(tmp.next!=null){
            baru.next = tmp.next;
            tmp.next = baru;
        }
    }
}
```

removeFirst()



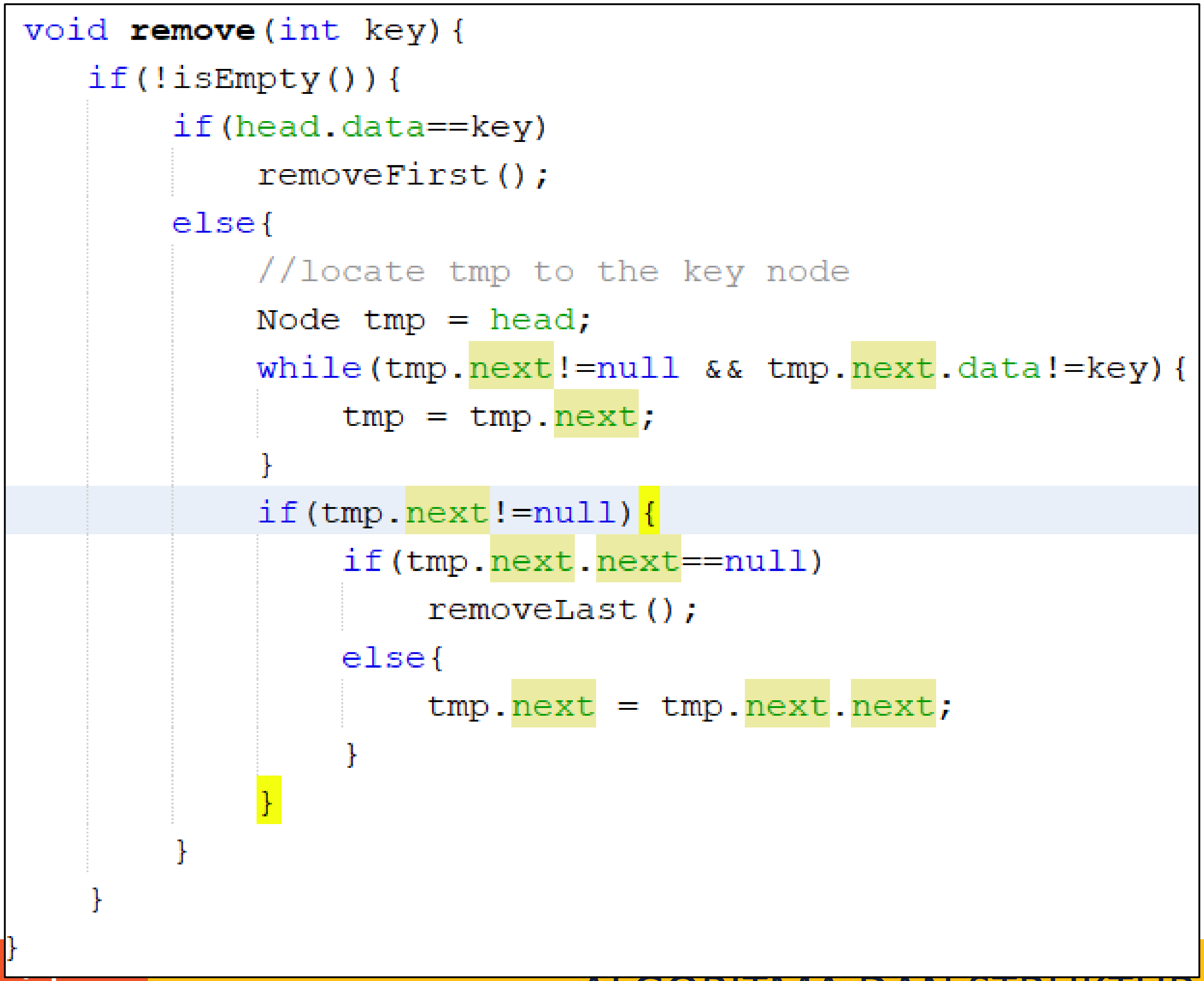
```
void removeFirst(){
    if(!isEmpty()){
        Node tmp = head;
        head = tmp.next;
        tmp = null;
    }
}
```

removeLast()



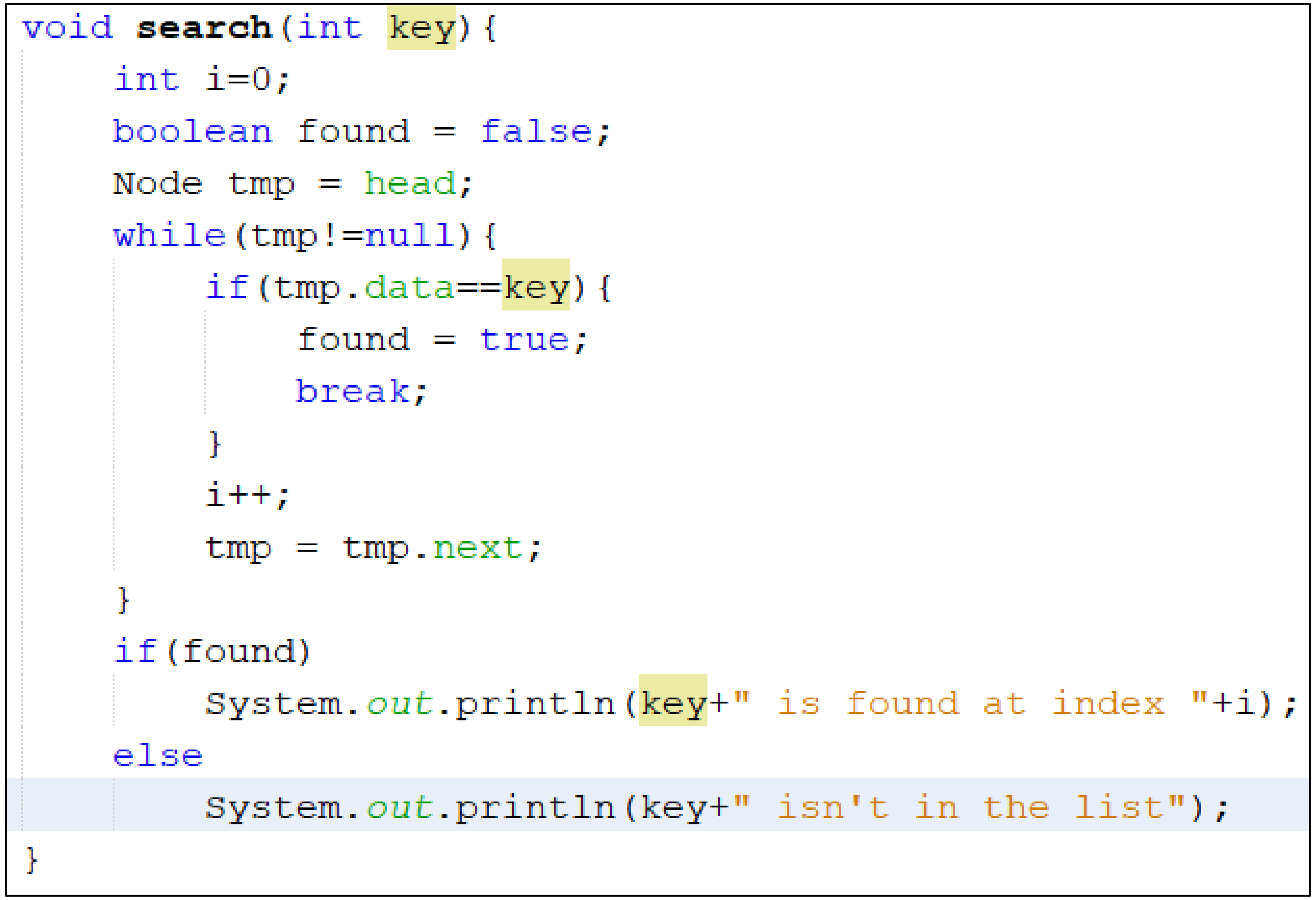
```
void removeLast(){
    if(!isEmpty()){
        if(head.next==null)
            removeFirst();
        else{
            //locate tmp to the last node
            Node tmp = head;
            while(tmp.next.next!=null){
                tmp = tmp.next;
            }
            tmp.next = null;
        }
    }
}
```

remove()



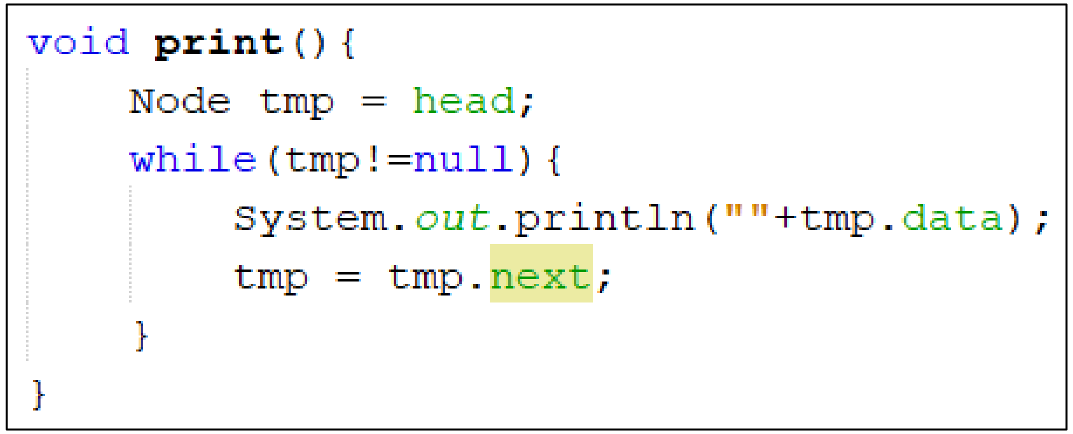
```
void remove(int key){
    if(!isEmpty()){
        if(head.data==key)
            removeFirst();
        else{
            //locate tmp to the key node
            Node tmp = head;
            while(tmp.next!=null && tmp.next.data!=key){
                tmp = tmp.next;
            }
            if(tmp.next!=null){
                if(tmp.next.next==null)
                    removeLast();
                else{
                    tmp.next = tmp.next.next;
                }
            }
        }
    }
}
```

find()



```
void search(int key){
    int i=0;
    boolean found = false;
    Node tmp = head;
    while(tmp!=null){
        if(tmp.data==key){
            found = true;
            break;
        }
        i++;
        tmp = tmp.next;
    }
    if(found)
        System.out.println(key+" is found at index "+i);
    else
        System.out.println(key+" isn't in the list");
}
```

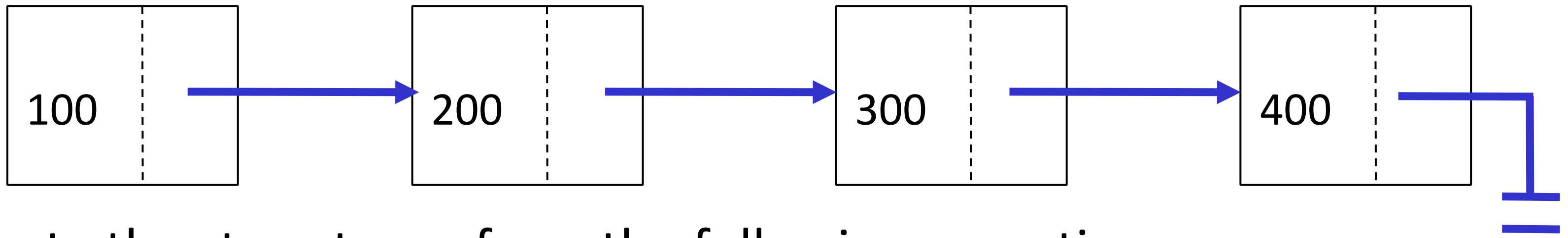
printNode()



```
void print(){
    Node tmp = head;
    while(tmp!=null){
        System.out.println(""+tmp.data);
        tmp = tmp.next;
    }
}
```

Assignments

Based on the following initial linked list:



Create the steps to perform the following operation:

1. Add last 500.
2. Add first 50.
3. Add 250 after 200.
4. Add 150 at index 1 (position 2)
5. Delete first
6. Delete last
7. Delete 300.
8. Delete node at index 3 (position 4)

Thank you ☺