# Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

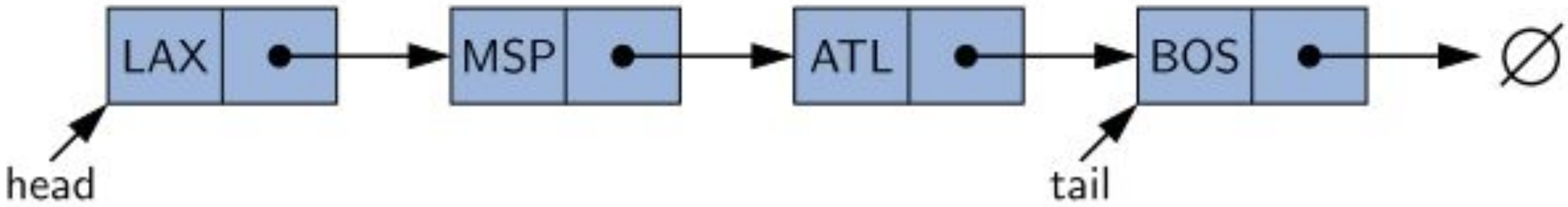
Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

# Singly Linked Lists

- A linked list's representation relies on the collaboration of many objects
- Minimally, the linked list instance must keep a reference to the first node of the list, known as the **head**
- Without an explicit reference to the head, there would be no way to locate that node
- The last node of the list is known as the **tail**
- The tail of a list can be found by *traversing the linked list*— **starting at the head** and *moving from one node to another* by following each **node's next reference**
- We can identify the tail as the node having ***null*** as its next reference



# Circularly Linked Lists

- Linked lists are traditionally viewed as storing a sequence of items in a linear order, from first to last
- Many applications in which data can be more naturally viewed as having a cyclic order
  - For example, many multiplayer games are turn-based, with player A taking a turn, then player B, then player C, and so on
  - City buses often run on a continuous loop, making stops in a scheduled order
  - Round-Robin Scheduling

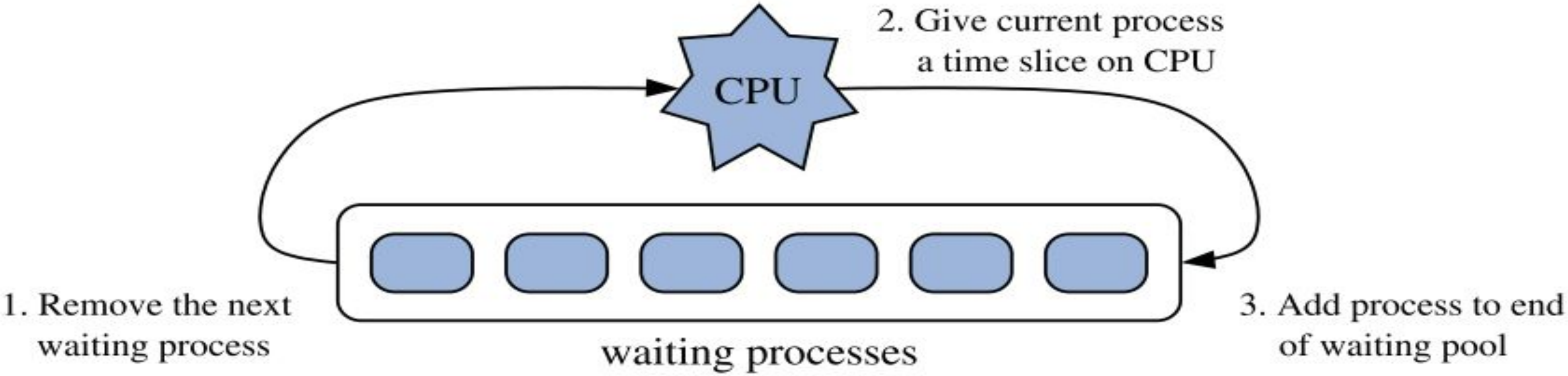
# Round-Robin Scheduling

- An operating system is managing many processes that are currently active on a computer
- Most operating systems allow processes to effectively share use of the CPUs time
- A process is given a short turn to execute, known as a ***time slice***, but it is interrupted when the slice ends, even if its job is not yet complete
- Each active process is given its own time slice, taking turns in a ***cyclic order***
- New processes can be added to the system, and processes that complete their work can be removed

# Round-Robin Scheduler Implementation

- We can implement a round robin scheduler using a queue ***Q*** by repeatedly performing the following steps:
  - ***e = Q.dequeue()***
  - ***Service element e***
  - ***Q.enqueue(e)***

- A round-robin scheduler could be implemented with a traditional linked list, by repeatedly performing the following steps on linked list ***L***
  - ***process p = L.removeFirst()***
  - Give a time slice to ***process p***
  - ***L.addLast(p)***

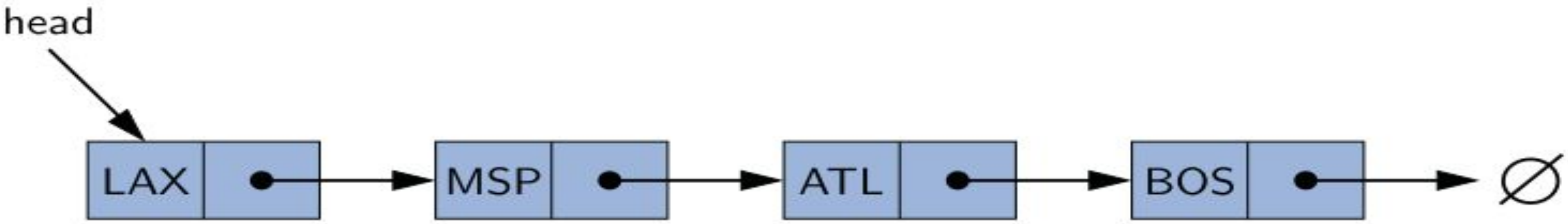


# Round-Robin Scheduler Implementation Challenge

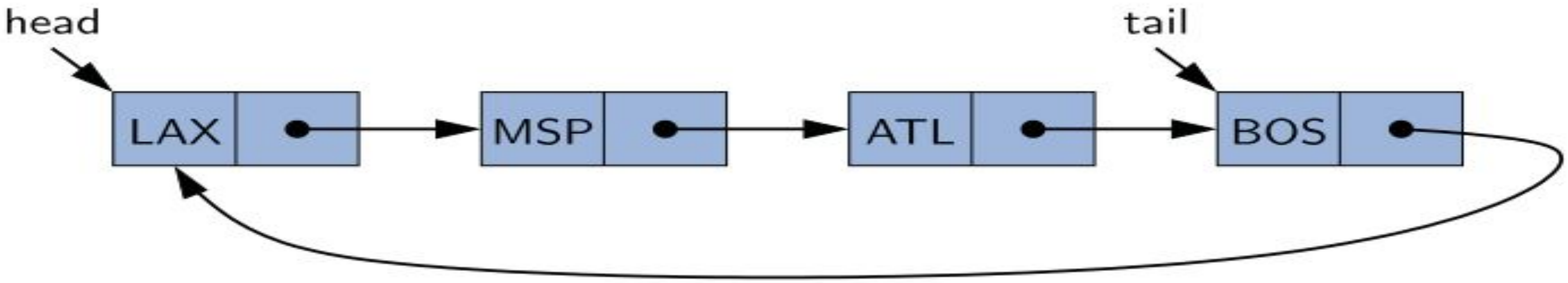
- Unfortunately, there are drawbacks to the both approaches discussed
- It is unnecessarily inefficient to *repeatedly throw away a node from one end and create a new node for the same element at other end*
- *How to provide a more efficient data structure for representing a cyclic order?*

# Designing and Implementing a Circularly Linked List

- Extend a singularly linked list to a Circularly Linked List




- Essentially a singularly linked list in which the *next reference of the tail node is set to refer back to the head of the list* (rather than null)



# Designing and Implementing a Circularly Linked List

- We use new discussed model to design and implement a new ***CircularlyLinkedList*** class, which supports all of the public behaviors of SinglyLinkedList class and one additional method
  - **rotate()**: Moves the first element to the end of the list
- We need to following update methods of SinglyLinkedList class
  - **addFirst()**
  - **addLast()**
  - **removeFirst()**
- Now, round-robin scheduling can be efficiently implemented by repeatedly performing the following steps on a circularly linked list C:
  - Give a time slice to process C.first()
  - C.rotate()
- In this implementation, we remove the head reference, and get the head as ***tail.getNext()***

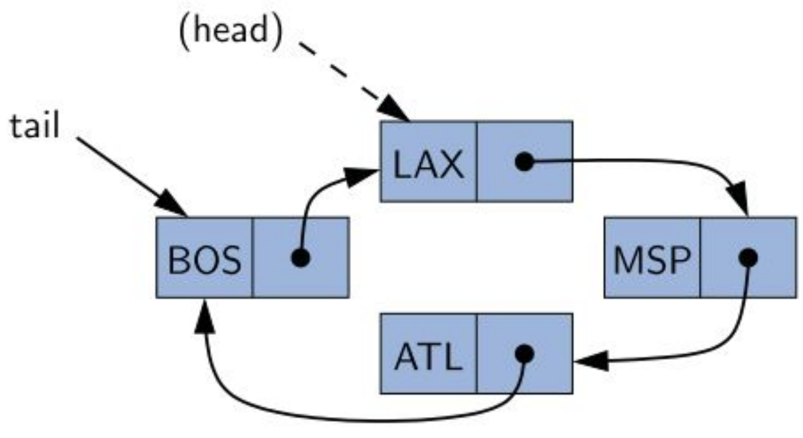
# Implementing a Circularly Linked List

```
public class CircularlyLinkedList<E> {  /* node class definition*/
    private Node<E> tail = null;            // we store tail (but not head)
    private int size = 0;                   // number of nodes in the list
    public CircularlyLinkedList() { }       // constructs an initially empty list
    public int size() { return size; }
    public boolean isEmpty() { return size == 0; }
    public E first() {
        if (isEmpty()) return null;
        return tail.getNext().getElement();    // the head is *after* the tail
    }
    public E last() {                       // returns (but does not remove) the last element
        if (isEmpty()) return null;
        return tail.getElement();
    }
```

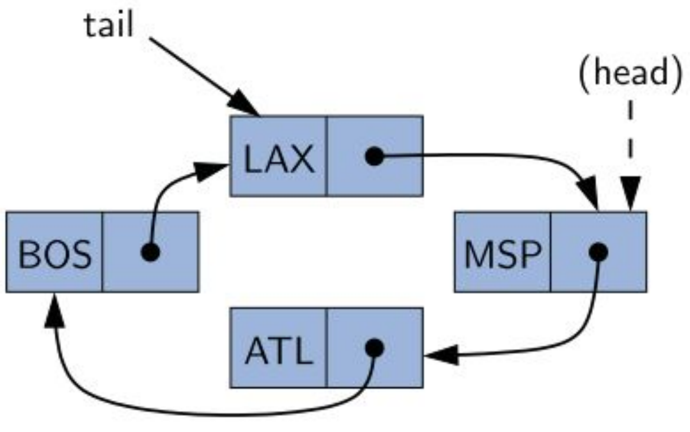
# Implementing a Circularly Linked List

```
public void rotate() {    // rotate the first element to the back of the list
    if (tail != null)     // if empty, do nothing
    tail = tail.getNext();     // the old head becomes the new tail
}
```

- We **do not move any nodes or elements**, we simply *advance the tail reference to point to the node that follows it* (the implicit head of the list)



(a)



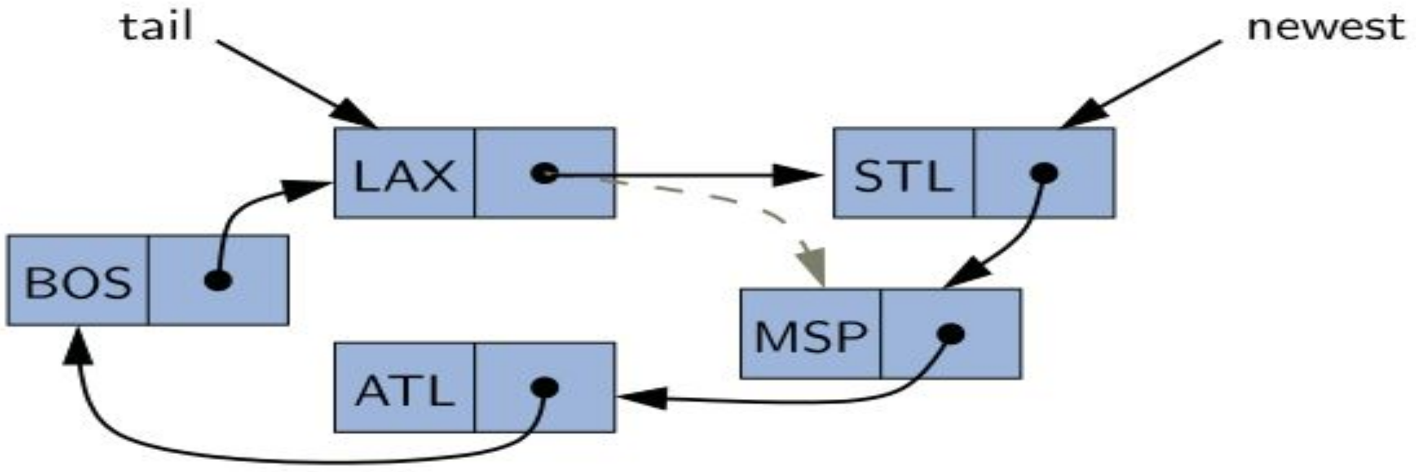
(b)

# Implementing a Circularly Linked List

```
public void addFirst(E e) {       // adds element e to the front of the list
    if (size == 0) { tail = new Node<>(e, null);
        tail.setNext(tail);    // link to itself circularly
    } else { Node<E> newest = new Node<>(e, tail.getNext());
        tail.setNext(newest);
    }
    size++;
}
```

```
public void addLast(E e) {  // adds element e to the end of the list
    addFirst(e); // insert new element at front of list
    tail = tail.getNext(); // now new element becomes the tail
}
```

- We can add a new element at the front of the list by **creating a new node** and ***linking it just after the tail*** of the list
- To implement the **addLast** method, we can rely on the use of a **call to addFirst** and then ***immediately advance the tail reference*** so that the newest node becomes the last



# Implementing a Circularly Linked List

```
public E removeFirst() {   // removes and returns the first element
    if (isEmpty()) return null;          // nothing to remove
    Node<E> head = tail.getNext();
    if (head == tail)
        tail = null;       // must be the only node left
    else
        tail.setNext(head.getNext());     // removes “head” from the list
    size−−;
    return head.getElement();
}
} //end of class definition
```

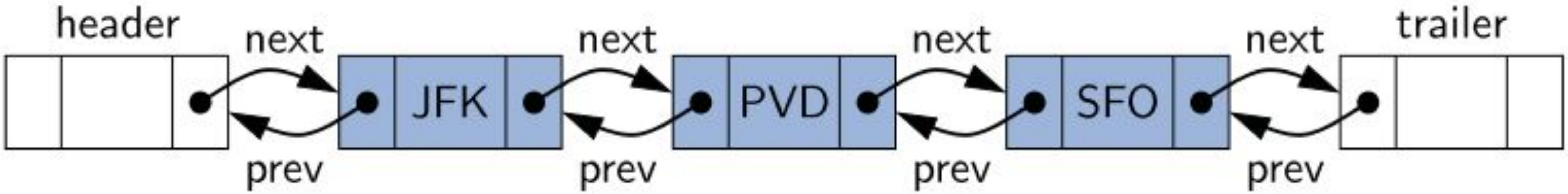
- Removing the first node from a circularly linked list can be accomplished by simply **updating the next field of the tail node to bypass the implicit head**

# Doubly Linked Lists

- In a singly linked list, each node maintains a reference to the node that is immediately after it
- A potential problem with singly linked lists is that it's **difficult to traverse backward** along the list
- There are limitations that stem from the asymmetry of a singly linked list
  - We *cannot efficiently delete an arbitrary node from an interior position of the list* if only given a reference to that node ( because we cannot determine the node that immediately precedes the node to be deleted )
- Symmetry in a linked list is each node keeps an explicit reference to the node before it and a reference to the node after it. Such a structure is known as a **doubly linked list**
- The doubly linked list **allows you to traverse backward as well as forward** through the list
- These lists allow a greater variety of O(1)-time update operations, including **insertions and deletions at arbitrary positions within the list**

# Header and Trailer Sentinels

- In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list:
  - a **header node** at the beginning of the list, and
  - a **trailer node** at the end of the list
- These “dummy” nodes are known as sentinels (or guards), and they do not store elements of the primary sequence



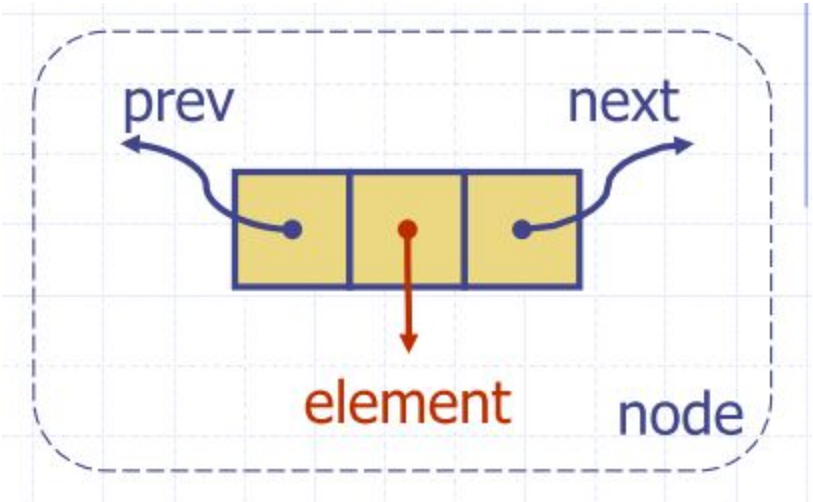
# Doubly Linked List Interface

- A DoublyLinkedList class supports the following public methods:

  size(): Returns the number of elements in the list
  isEmpty(): Returns true if the list is empty, and false otherwise
  first(): Returns (but does not remove) the first element in the list
  last(): Returns (but does not remove) the last element in the list
  addFirst(e): Adds a new element to the front of the list
  addLast(e): Adds a new element to the end of the list
  removeFirst(): Removes and returns the first element of the list
  removeLast(): Removes and returns the last element of the list
  addBetween(e, predecessor, successor): Adds a new element between two nodes predecessor and sucessor

# Implementing a Doubly Linked List Class (Node)

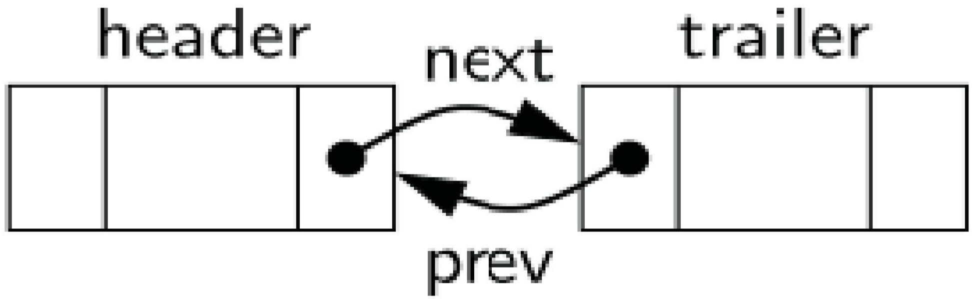
```
private static class Node<E> {
    private E element;      // reference to the element stored at this node
    private Node<E> prev;       // reference to the previous node in the list
    private Node<E> next;       // reference to the subsequent node in the list

    public Node(E e, Node<E> p, Node<E> n) {
        element = e;
        prev = p; next = n;
    }

    public E getElement() { return element; }
    public Node<E> getPrev() { return prev; }
    public Node<E> getNext() { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
}
```




Nodes store:

- element
- link to the previous node
- link to the next node

# Implementing a Doubly Linked List Class

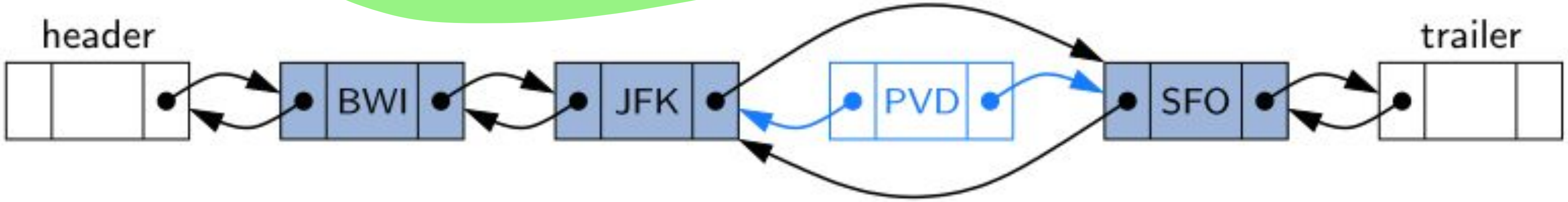



```
public class DoublyLinkedList<E> { /* Node Class*/
    private Node<E> header;
    private Node<E> trailer;
    private int size = 0;

    public DoublyLinkedList() {   \\Constructs a new empty list
        header = new Node<>(null, null, null);  // create header
        trailer = new Node<>(null, header, null);       // trailer is preceded by header
        header.setNext(trailer);              // header is followed by trailer
    }
    public int size() { return size; }              //Returns the number of elements in the linked list
    public boolean isEmpty() { return size == 0; }        //Tests whether the linked list is empty
    public E first() {          //Returns (but does not remove) the first element of the list
        if (isEmpty()) return null;
        return header.getNext().getElement();
    }
    public E last() {          //Returns (but does not remove) the last element of the list
        if (isEmpty()) return null;
        return trailer.getPrev().getElement();
    }
```

# Implementing a Doubly Linked List Class

```
private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
        //Adds element e to the linked list in between the given nodes
        Node<E> newest = new Node<>(e, predecessor, successor);  // create and link a new node
        predecessor.setNext(newest);
        successor.setPrev(newest);
        size++;
}
public void addFirst(E e) { //Adds element e to the front of the list
        addBetween(e, header, header.getNext());
}
public void addLast(E e) {  //Adds element e to the end of the list
        addBetween(e, trailer.getPrev(), trailer);
}
```

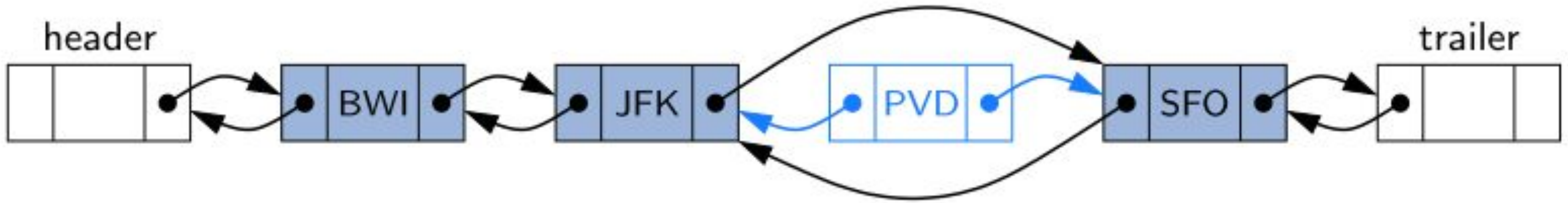
1. **Create a new node,**
2. **Set predecessor NEXT to the new element,**
3. **Set succesor PREV to the new element,**
4. **increase size.**



# Implementing a Doubly Linked List Class

```
private E remove(Node<E> node) { \\Removes the given node from the list and returns its element
        Node<E> predecessor = node.getPrev();
        Node<E> successor = node.getNext();
        predecessor.setNext(successor);
        successor.setPrev(predecessor);
        size−−;
        return node.getElement(); }
public E removeFirst() {  //Removes and returns the first element of the list
        if (isEmpty()) return null;
        return remove(header.getNext());}
public E removeLast() {  //Removes and returns the last element of the list
        if (isEmpty()) return null;
        return remove(trailer.getPrev()); }
} //----------- end of DoublyLinkedList class -----------
```

1. **Find Predecessor and successor of node,**
2. **Set predecessor NEXT to the successor,**
3. **Set succesor PREV to the predecessor,**
4. **decrease size.**



# Exercise

- Implement Stack and Queue ADT using linked list storage data structure
- Stack
  - push(data): theList.addFirst(data)
  - pop(): data = theList.removeFirst()
- Queue
  - enqueue(data):theList.addLast(data)
  - dequeue():theList.removeFirst()

```java
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedStack() { }                  // new stack relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(E element) { list.addFirst(element); }
  public E top() { return list.first(); }
  public E pop() { return list.removeFirst(); }
}
```

```java
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty list
  public LinkedQueue() { }                  // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```