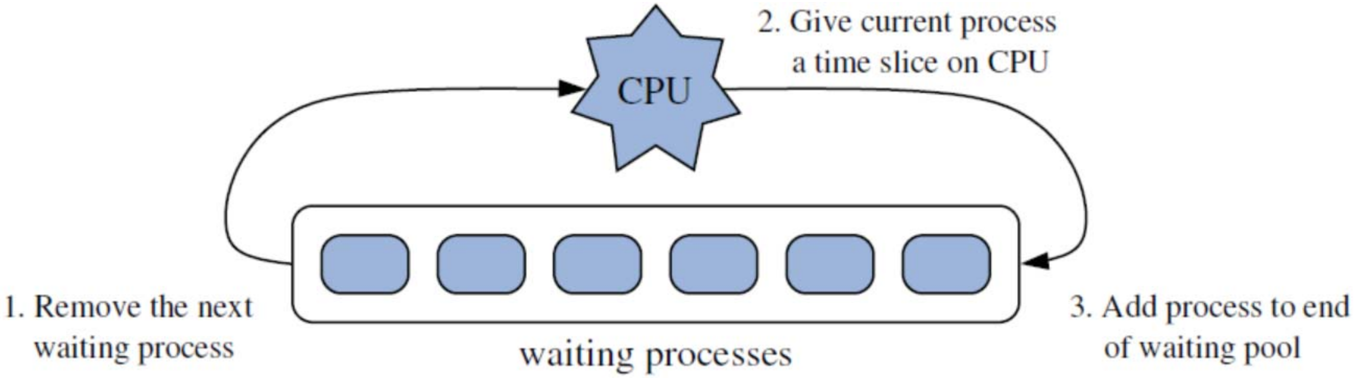
CSE214 Data Structures

Circularly Linked Lists, Doubly Linked Lists

YoungMin Kwon

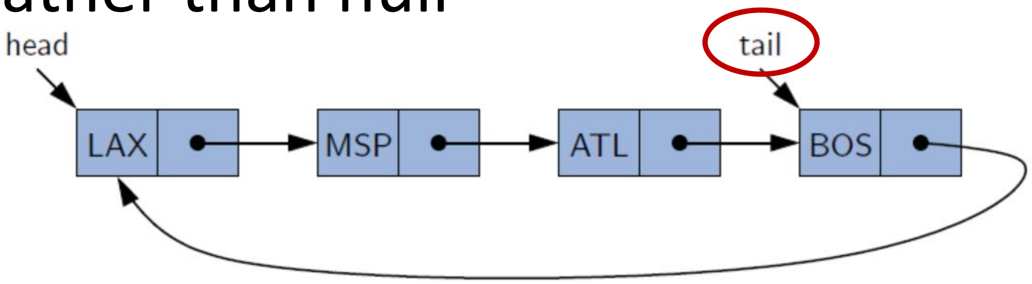
Circularly Linked List

- Cyclic order
  - Well-defined neighboring relationships, but no fixed beginning or end
  - E.g. round-robin scheduling
    - Each active process runs during a short time slice
    - When the time slice is expired, the process is added back to a wait queue


2. Give current process a time slice on CPU
CPU
1. Remove the next waiting process
waiting processes
3. Add process to end of waiting pool

Circularly Linked List

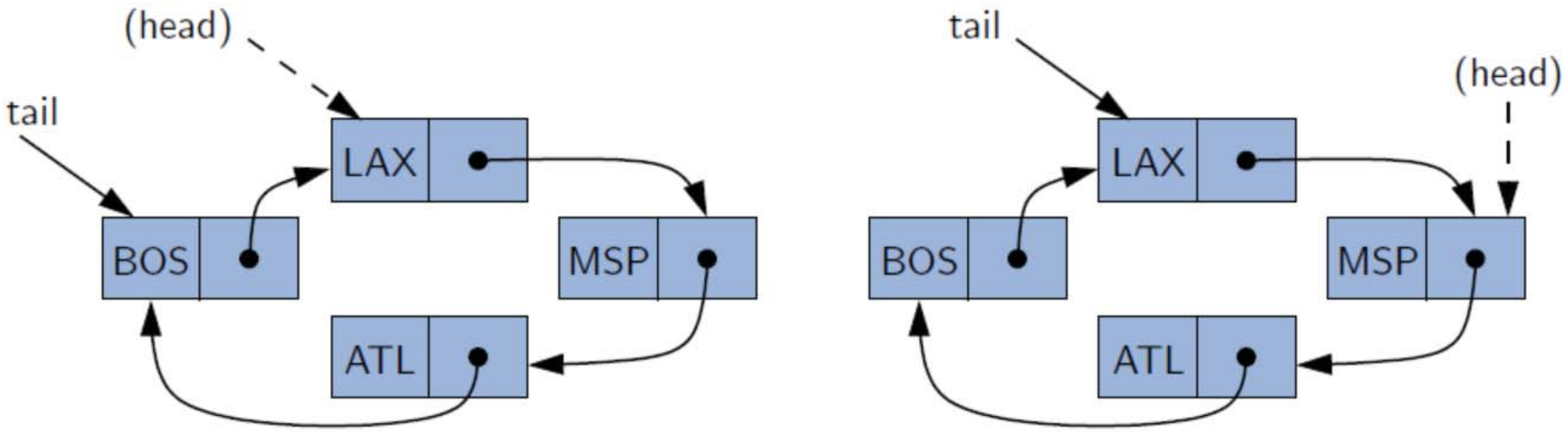
- Circularly linked list
  - A singly linked list whose tail is pointing to the head rather than null



  - No need to maintain the head
    - tail.next is the head
  - rotate() operation
    - Move the first element to the end

Circularly Linked List

- Rotate operation
  - No need to move any nodes or elements
  - Simply advance the tail reference



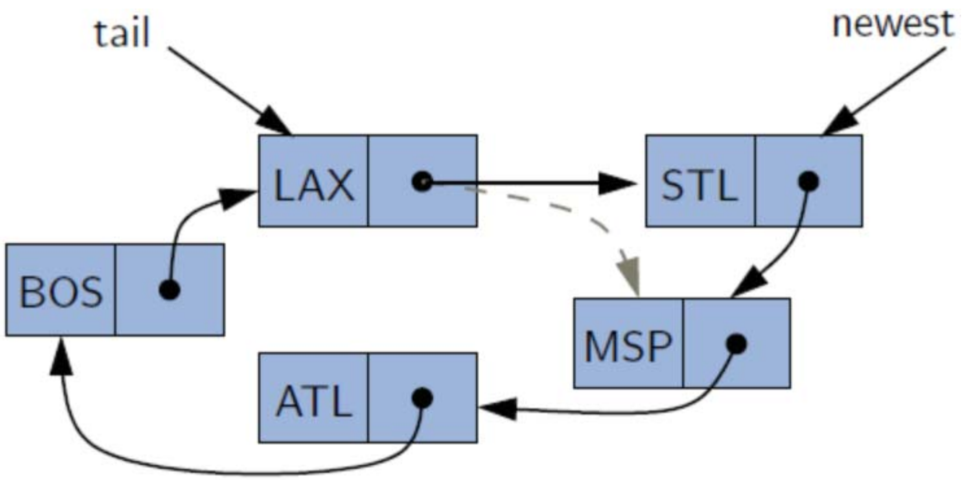
{ LAX, MSP, ATL, BOS }

{ MSP, ATL, BOS, LAX }

Circularly Linked List

- addFirst(e)
  - Create a new node
  - Link it after the tail

- addLast(e)
  - Call addFirst(e)
  - Advance the tail (rotate)



addFirst(STL)

Implementing Circularly Linked List

```
public class CircularlyLinkedList<E> {
    private static class Node<E> {
        private E e;
        private Node<E> next;

        public Node(E e, Node<E> n)    { this.e = e; this.next = n; }
        public E getElement()          { return e; }
        public Node<E> getNext()       { return next; }
        public void setNext(Node<E> n) { next = n; }
    }

    private Node<E> tail;
    private int size;

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

No head:
head is after tail

```java
public E first() { //first is after tail
    return isEmpty() ? null : tail.getNext().getElement();
}

public E last() {  //last is at tail
    return isEmpty() ? null : tail.getElement();
}

public void addFirst(E e) { //add after tail
    if(isEmpty()) {
        tail = new Node<E>(e, null);
        tail.setNext(tail);
    }
    else {
        tail.setNext(new Node<E>(e, tail.getNext()));
    }
    size++;
}

public void addLast(E e) {//add at tail
    addFirst(e);
    tail = tail.getNext();
}
```

```
public E removeFirst() {
    if(isEmpty())
        return null;

    Node<E> head = tail.getNext();
    if(head == tail)
        tail = null;
    else
        tail.setNext(head.getNext());

    size--;
    return head.getElement();
}

public E removeLast() {
    //TODO: implement this method
}
```

```java
    private static void onFalseThrow(boolean b) {
        if(!b)
            throw new RuntimeException("Error: unexpected");
    }

    public static void main(String[] args) {
        CircularlyLinkedList<Integer> list =
                           new CircularlyLinkedList<Integer>();
        list.addLast(2);
        list.addLast(3);
        list.addLast(4);
        list.addFirst(1);

        onFalseThrow(list.removeLast()  == 4);
        onFalseThrow(list.removeLast()  == 3);
        onFalseThrow(list.removeFirst() == 1);
        onFalseThrow(list.removeLast()  == 2);

        System.out.println("Success!");
    }
}
```

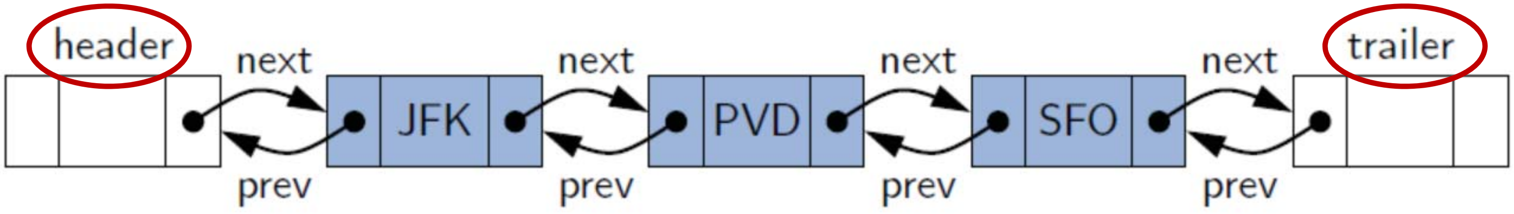
Doubly Linked List

- A drawback of a singly linked list
  - Hard to remove a node from a tail
  - In general, hard to remove an interior node
  - Because of the lack of a backward link

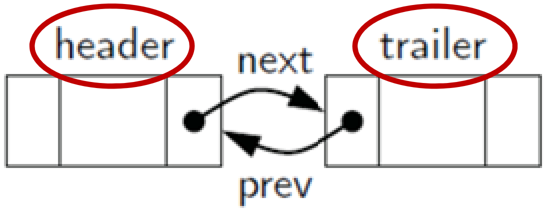
- Doubly linked list
  - Each node keeps a forward link (next) and a backward link (prev)

Doubly Linked List

- Header and trailer sentinels
  - To avoid any special operations at the boundaries
  - Header node at the beginning of the list
  - Trailer at the end of the list



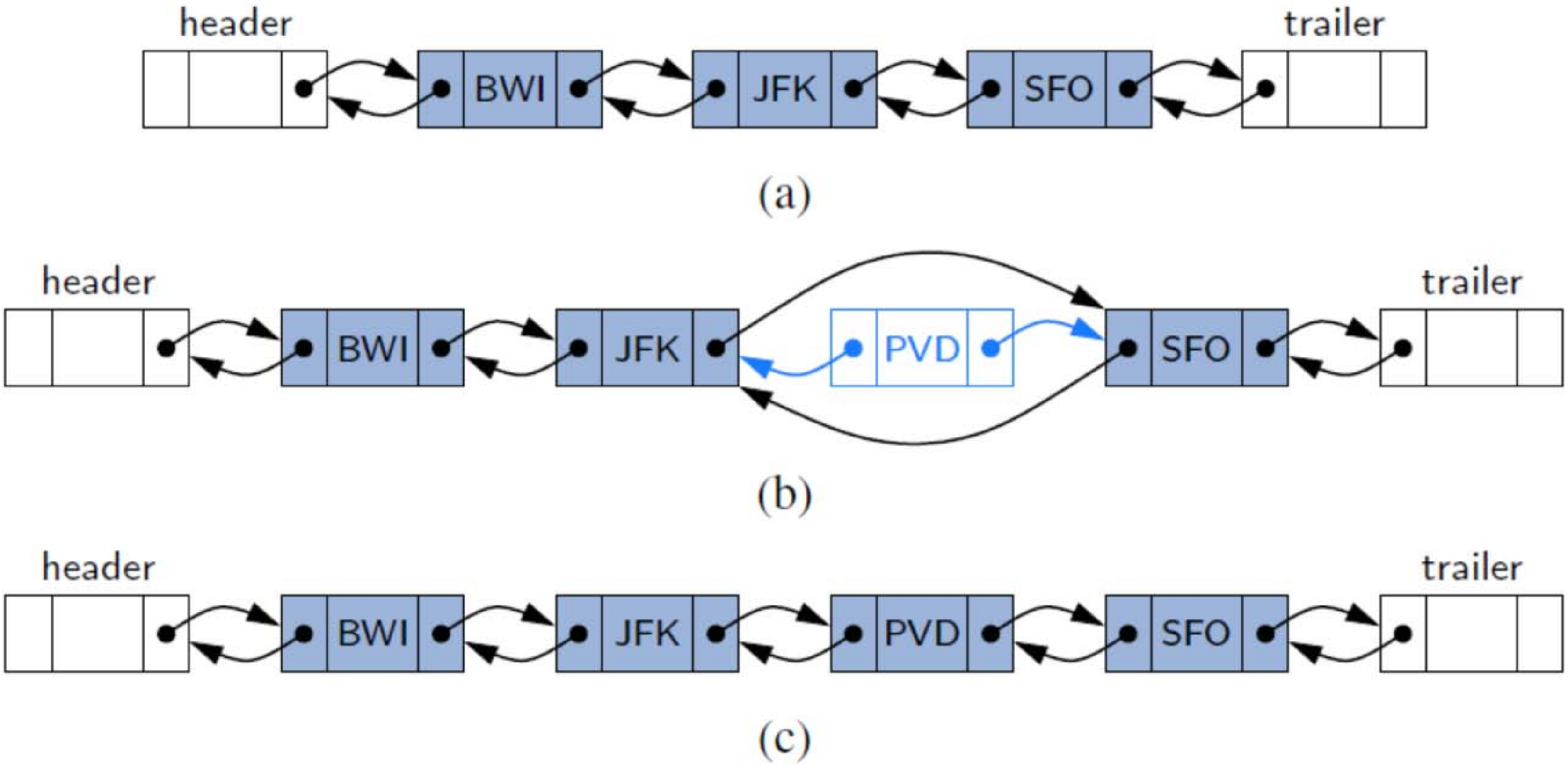
  - List is initialized such that next of header points to trailer and prev of trailer points to header



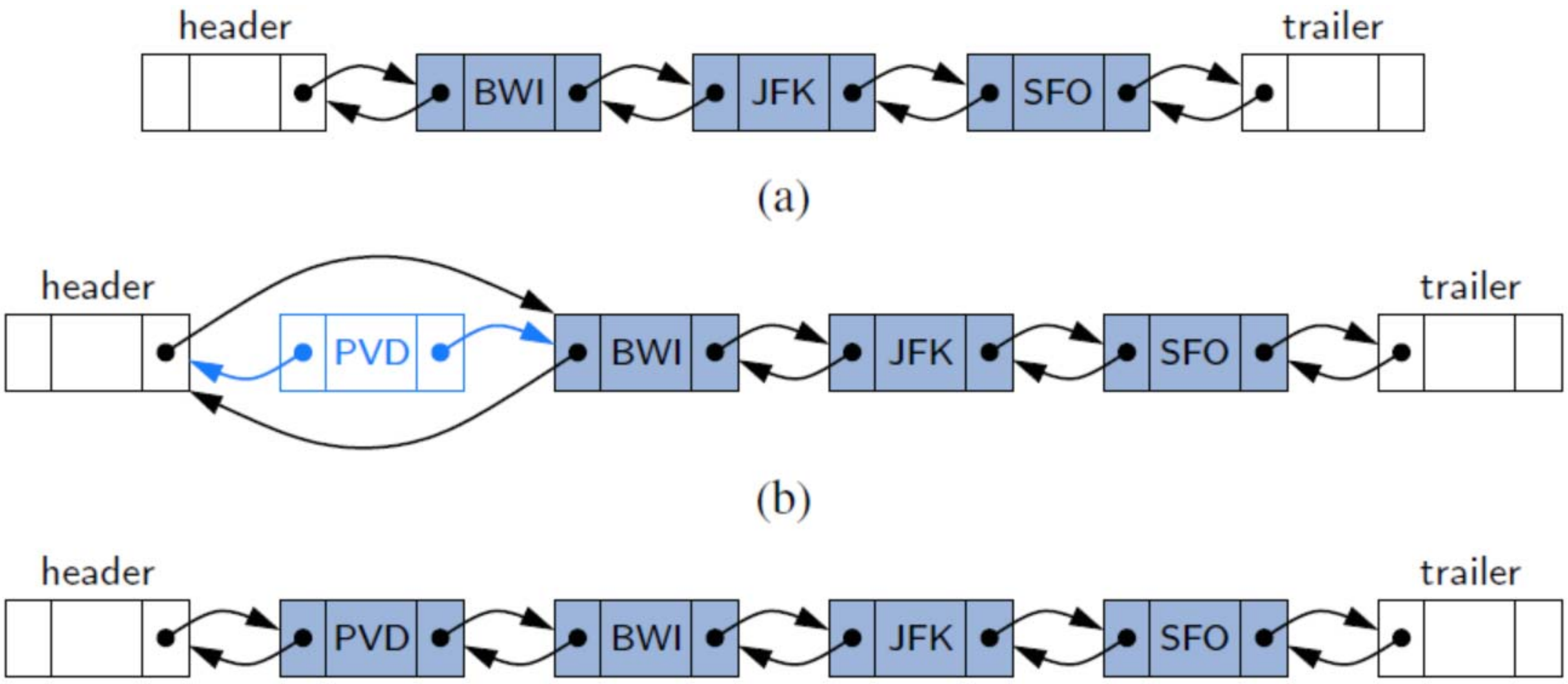
Doubly Linked List

- Advantage of using sentinels
  - Treat all insertions and removals in a unified manner at a slight memory overhead
    - Insertion: all insertions are in between existing nodes
    - Removal: all nodes to be deleted have neighbors on both sides
  - E.g. SinglyLinkedList: special handling for
    - Insertion to an empty list
    - Removal of the last element

Adding to a Doubly Linked List



Adding to a Doubly Linked List
(to the front)

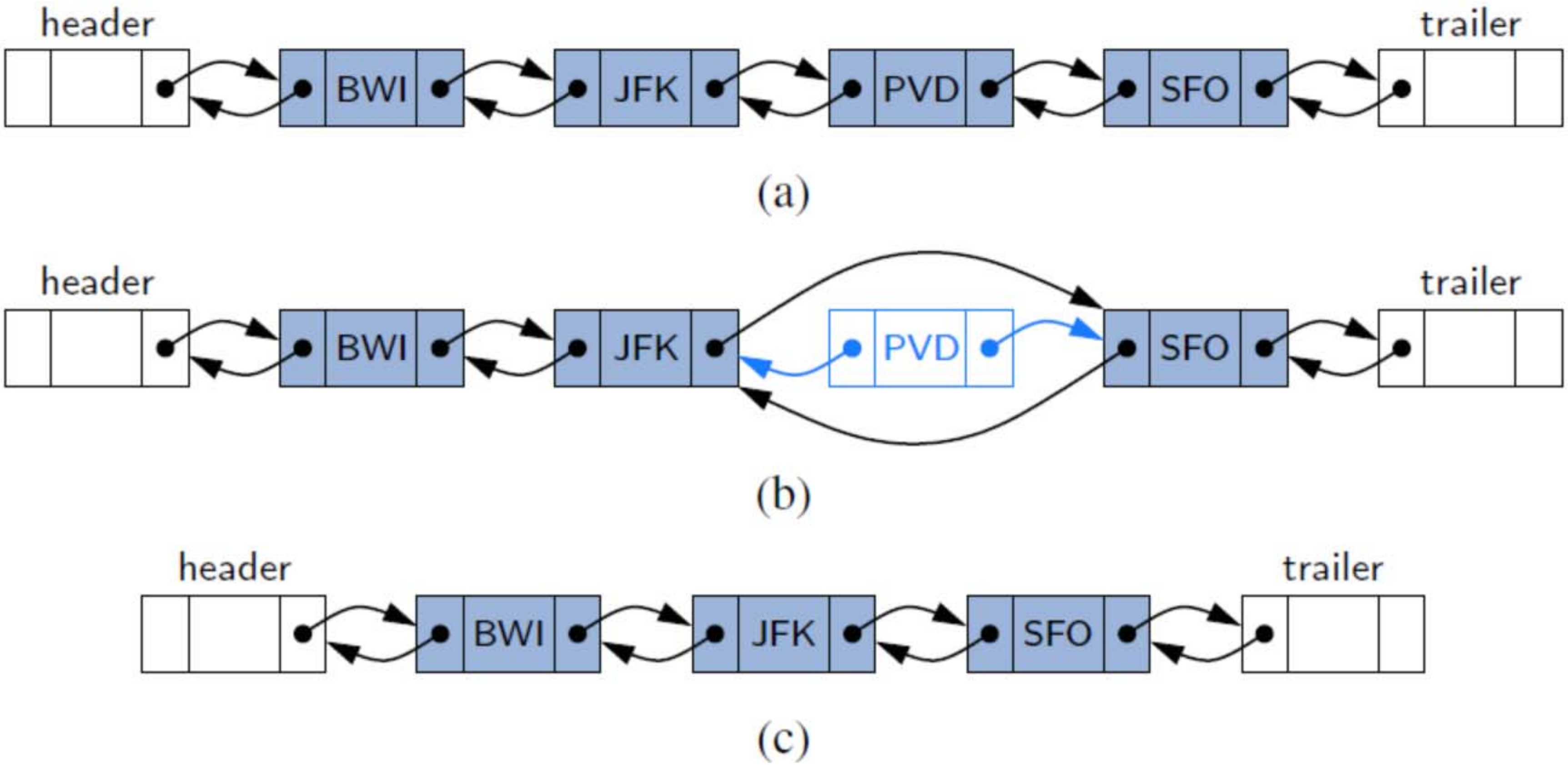


(a)

(b)

(c)

Removing from a Doubly Linked List

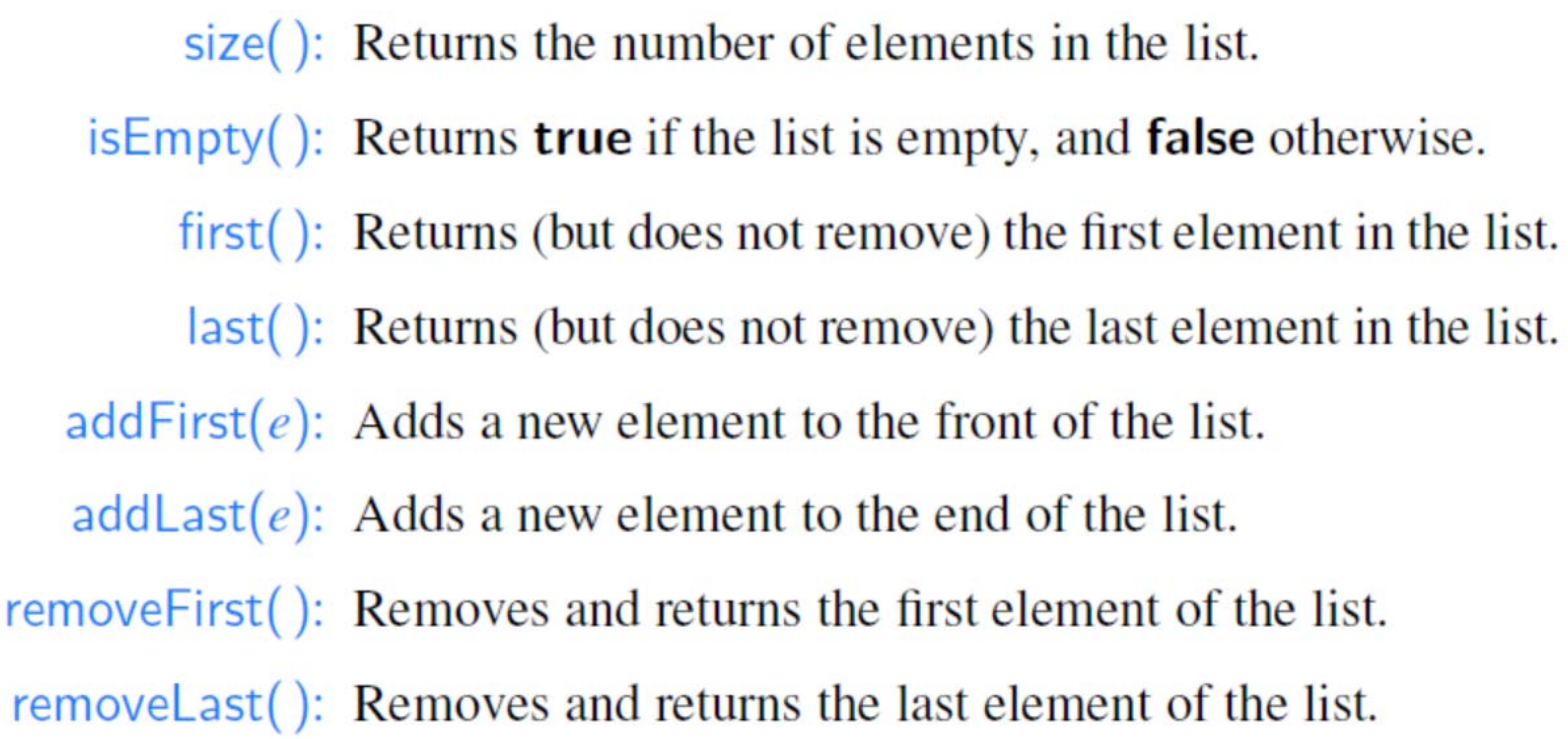


(a)

(b)

(c)

Operations on Doubly Linked List



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.

- We can add to/remove from an internal position, but we will not support them for now

Implementing Doubly Linked List

```
public class DoublyLinkedList<E> {
    private static class Node<E> {
        private E e;
        private Node<E> next, prev;
        public Node(E e, Node<E> p, Node<E> n) {
            this.e = e; prev = p; next = n; }
        public E getElement()          { return e; }
        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; }
    }
    private Node<E> head;
    private Node<E> tail;
    private int size;

    public DoublyLinkedList() {
        head = new Node<E>(null, null, null);
        tail = new Node<E>(null, head, null);
        head.setNext(tail);
    }
```

Sentinels: head and tail are Node instances

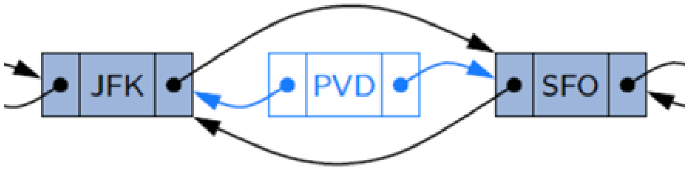
```
public int size()        { return size; }
public boolean isEmpty() { return size == 0; }

public E first() {
    return isEmpty() ? null : head.getNext().getElement();
}
public E last() {
    return isEmpty() ? null : tail.getPrev().getElement();
}

private void addBetween(E e, Node<E> pred, Node<E>succ) {
    Node<E> newest = new Node<E>(e, pred, succ);

    pred.setNext(newest);
    succ.setPrev(newest);

    size++;
}
public void addFirst(E e) {
    addBetween(e, head, head.getNext());
}
public void addLast(E e) {
    addBetween(e, tail.getPrev(), tail);
}
```



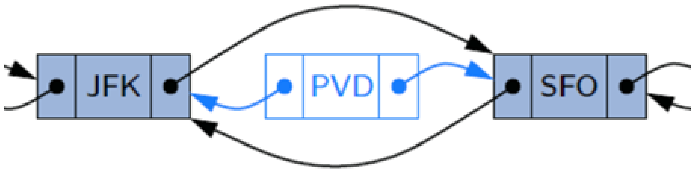
```
private E remove(Node<E> node) {
    Node<E> pred = node.getPrev();
    Node<E> succ = node.getNext();

    pred.setNext(succ);
    succ.setPrev(pred);

    size--;
    return node.getElement();
}

public E removeFirst() {
    if(isEmpty())
        return null;
    return remove(head.getNext());
}

public E removeLast() {
    if(isEmpty())
        return null;
    return remove(tail.getPrev());
}
```



```java
    private static void onFalseThrow(boolean b) {
        if(!b)
            throw new RuntimeException("Error: unexpected");
    }

    public static void main(String[] args) {
        DoublyLinkedList<Integer> list = new DoublyLinkedList<Integer>();
        list.addLast(2);
        list.addLast(3);
        list.addLast(4);
        list.addFirst(1);

        onFalseThrow(list.removeLast()  == 4);
        onFalseThrow(list.removeLast()  == 3);
        onFalseThrow(list.removeFirst() == 1);
        onFalseThrow(list.removeLast()  == 2);

        System.out.println("Success!");
    }
}
```

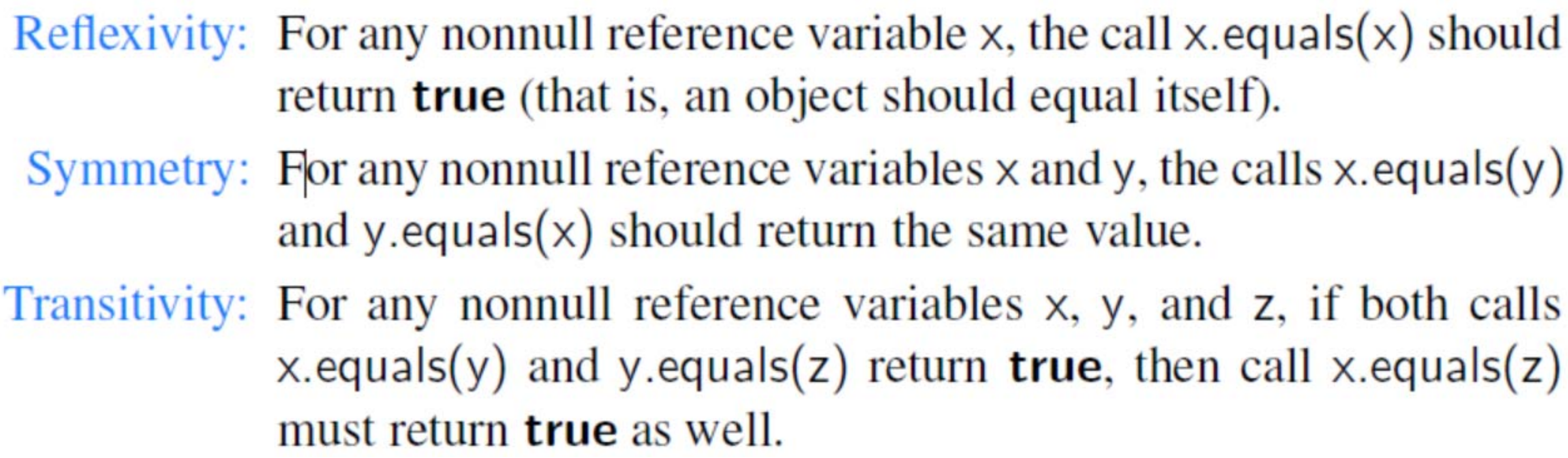
Testing for Equality

- For two reference variables a and b
  - a == b tests whether a and b reference the same object

- equals method of Object class
  - Syntax: a.equals(b)
  - When overriding this method make sure that the equivalence relation is maintained

Testing for Equality

- Equivalence relation

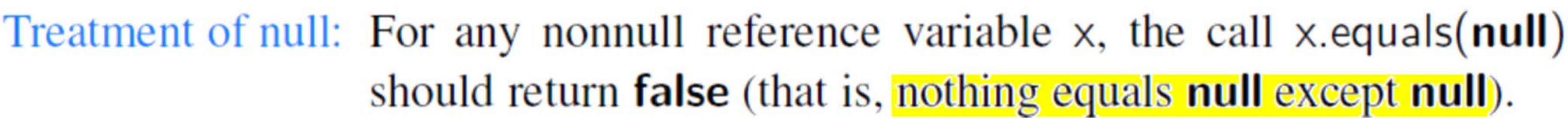


Reflexivity: For any nonnull reference variable x, the call x.equals(x) should return **true** (that is, an object should equal itself).

Symmetry: For any nonnull reference variables x and y, the calls x.equals(y) and y.equals(x) should return the same value.

Transitivity: For any nonnull reference variables x, y, and z, if both calls x.equals(y) and y.equals(z) return **true**, then call x.equals(z) must return **true** as well.

- In addition, Java requires



Treatment of null: For any nonnull reference variable x, the call x.equals(**null**) should return **false** (that is, nothing equals **null** except **null**).

Equivalence Testing with Linked Lists

- Two lists are equivalent
  - If they have the same size
  - The contents are element-by-element equivalent

- Implementing equals for SinglyLinkedList
  - While simultaneously traversing two lists, test x.equals(y) for each pair of corresponding elements x and y

Implementing equals for SinglyLinkedList

```
@SuppressWarnings("unchecked")
public boolean equals(Object o) {
    if(o == null)                   //nothing equals to null
        return false;

    if(getClass() != o.getClass())  //classes should be the same
        return false;

    SinglyLinkedList<E> that = (SinglyLinkedList<E>) o;
    if(size() != that.size())       //size should be the same
        return false;

    //element-wise equivalence
    for(Node<E> a = head, b = that.head; a != null; ) {
        if(!a.getElement().equals(b.getElement()))
            return false;
        a = a.getNext();
        b = b.getNext();
    }
    return true;
}
```

Copying Data Structures

- Shallow copy
  - For the primitive types, copy their values
  - For the reference types, copy the reference
    - The original and the copy point to the same object

- Deep copy
  - For the primitive types, copy their values
  - For the reference types, create an object and copy the contents to the new object
    - The original and the copy point to different objects

Copying Data Structures

- Object class has clone() method

```
protected Object clone() throws
                                CloneNotSupportedException
```

  - If a class did not implement Cloneable interface, the clone() method will throw the exception

- By convention,
  - Implement Cloneable interface
  - Override clone() with public access modifier

Implementing equals for SinglyLinkedList

```
@SuppressWarnings("unchecked")
public SinglyLinkedList<E> clone() throws
                                   CloneNotSupportedException {

    //Always use Object.clone() to create the initial copy
    SinglyLinkedList<E> that = (SinglyLinkedList<E>)super.clone();
    //Object.clone performs the default shallow copy,
    //that.size == this.size
    //that.head == this.head and that.tail == this.tail

    //deep copy now
    that.size = 0;
    that.head = that.tail = null;
    for(Node<E> n = this.head; n != null; n = n.getNext())
        that.addLast(n.getElement());

    return that;
}
```

Exercise

- As an exercise, download LinkedList.java and implement all TODOs
  - This exercise is for your own practice only
  - It will not be graded
  - We will implement it together in a recitation class