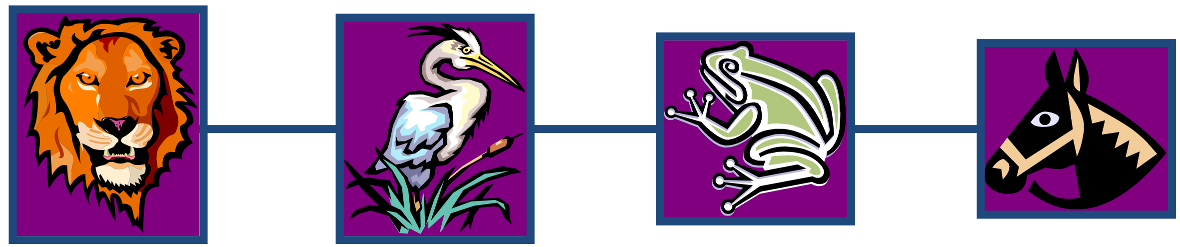
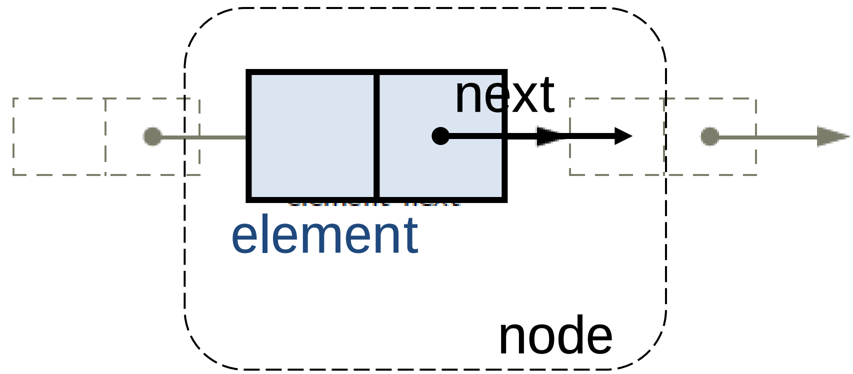


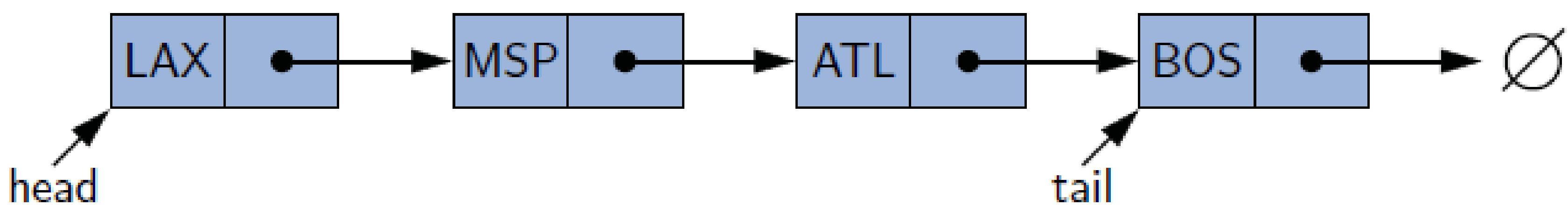
LINKED LISTS

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

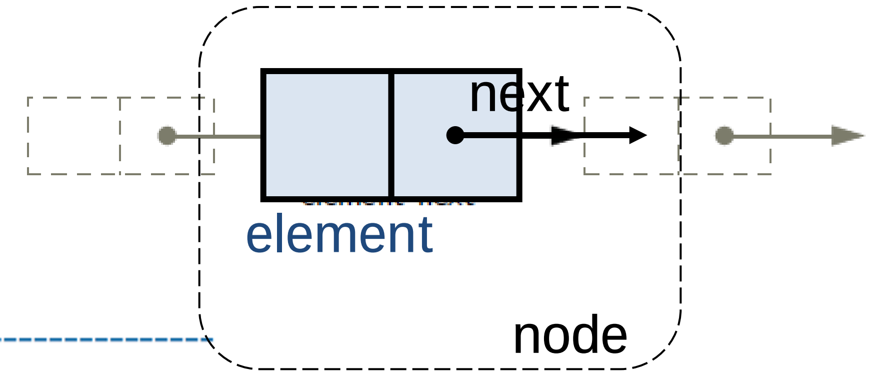
SINGLY LINKED LIST

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





A NESTED NODE CLASS




```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;            // reference to the element stored at this node
    private Node<E> next;         // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
  ... rest of SinglyLinkedList class will follow ...
```

IN package net.datastructures;

ACCESSOR METHODS

```
public class SinglyLinkedList<E> {
  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;           // head node of the list (or null if empty)
  private Node<E> tail = null;           // last node of the list (or null if empty)
  private int size = 0;                  // number of nodes in the list
  public SinglyLinkedList() { }          // constructs an initially empty list
  // access methods
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public E first() {                // returns (but does not remove) the first element
    if (isEmpty()) return null;
    return head.getElement();
  }
  public E last() {                 // returns (but does not remove) the last element
    if (isEmpty()) return null;
    return tail.getElement();
  }
```

INSERTING AT THE HEAD

Algorithm addFirst(e):

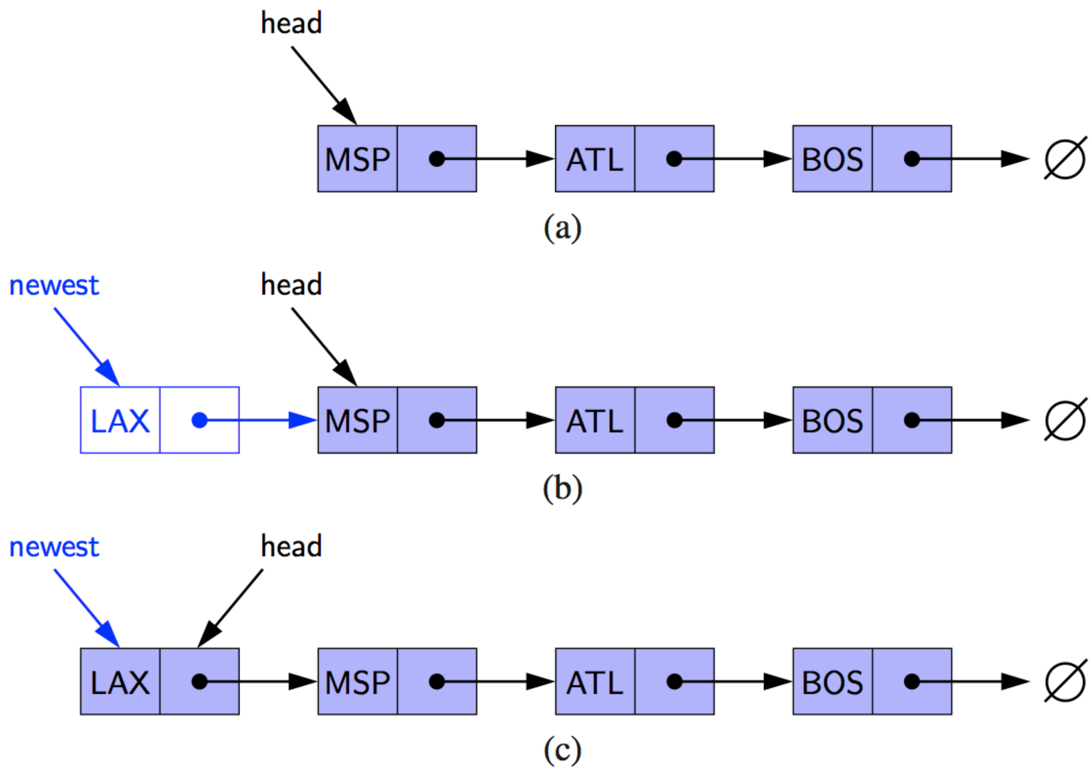
newest = Node(e) {create new node instance storing reference to element e}
newest.next = head {set new node's next to reference the old head node}
head = newest {set variable head to reference the new node}
size = size + 1 {increment the node count}

Allocate new node

Insert new element

Have new node point to old head

Update head to point to new node

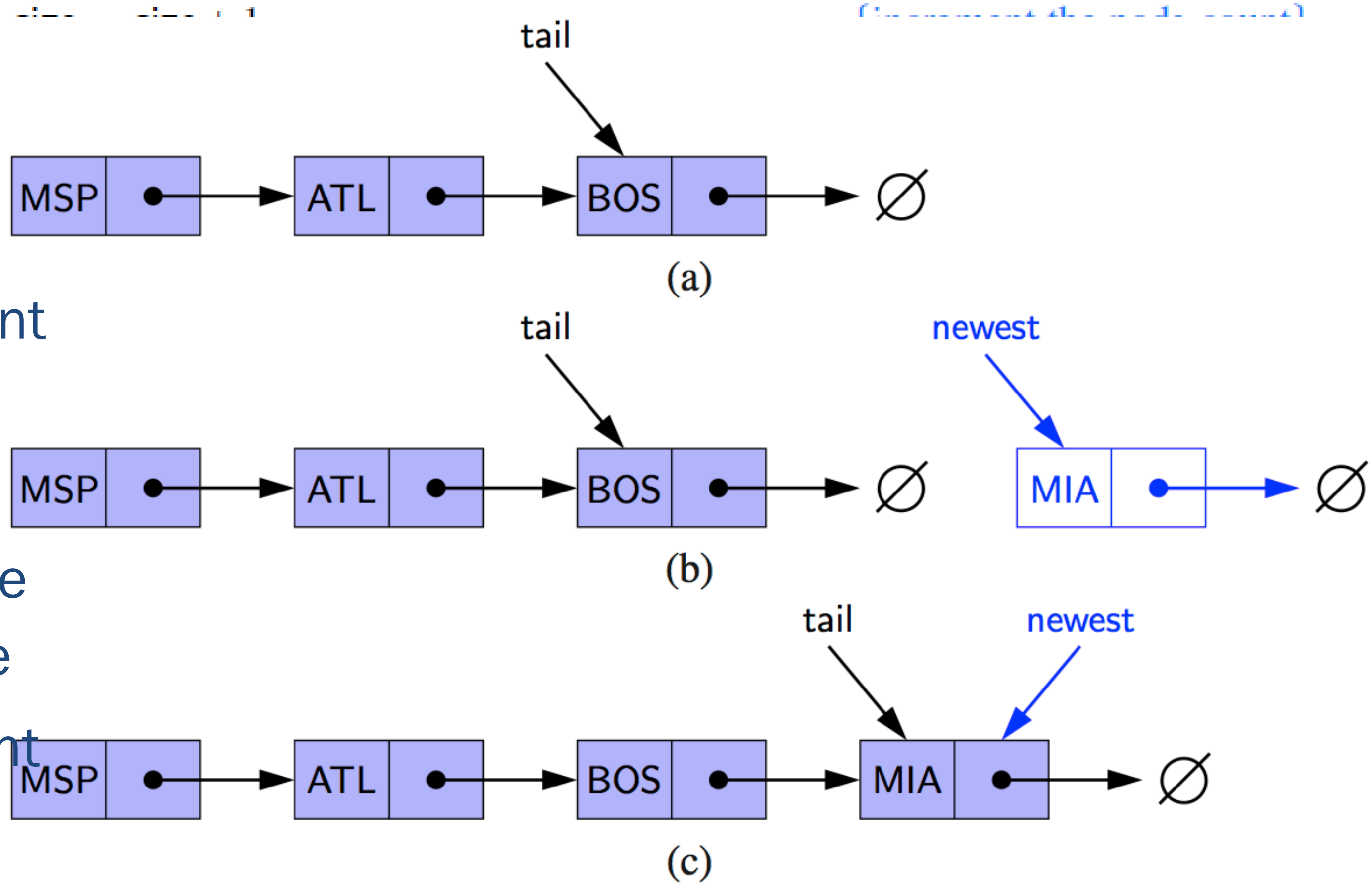


INSERTING AT THE TAIL

Algorithm addLast(e):

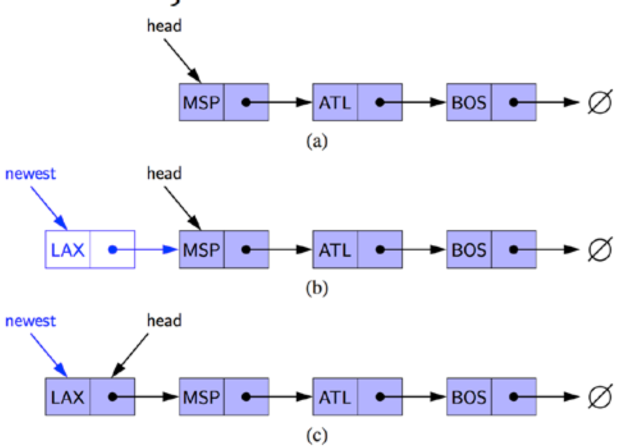
newest = Node(e) {create new node instance storing reference to element e}
newest.next = null {set new node's next to reference the null object}
tail.next = newest {make old tail node point to new node}
tail = newest {set variable tail to reference the new node}

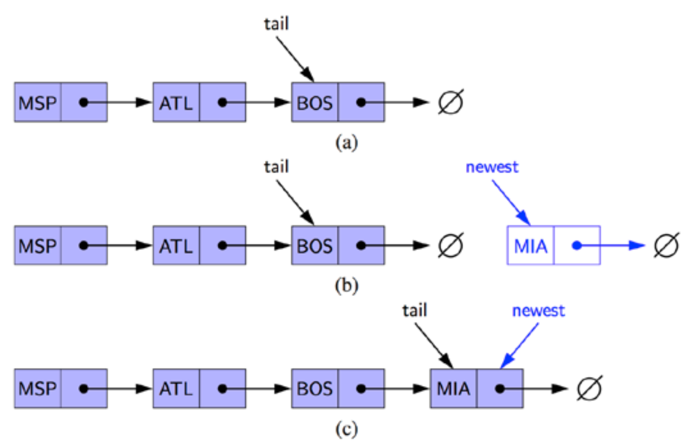
- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



JAVA METHODS

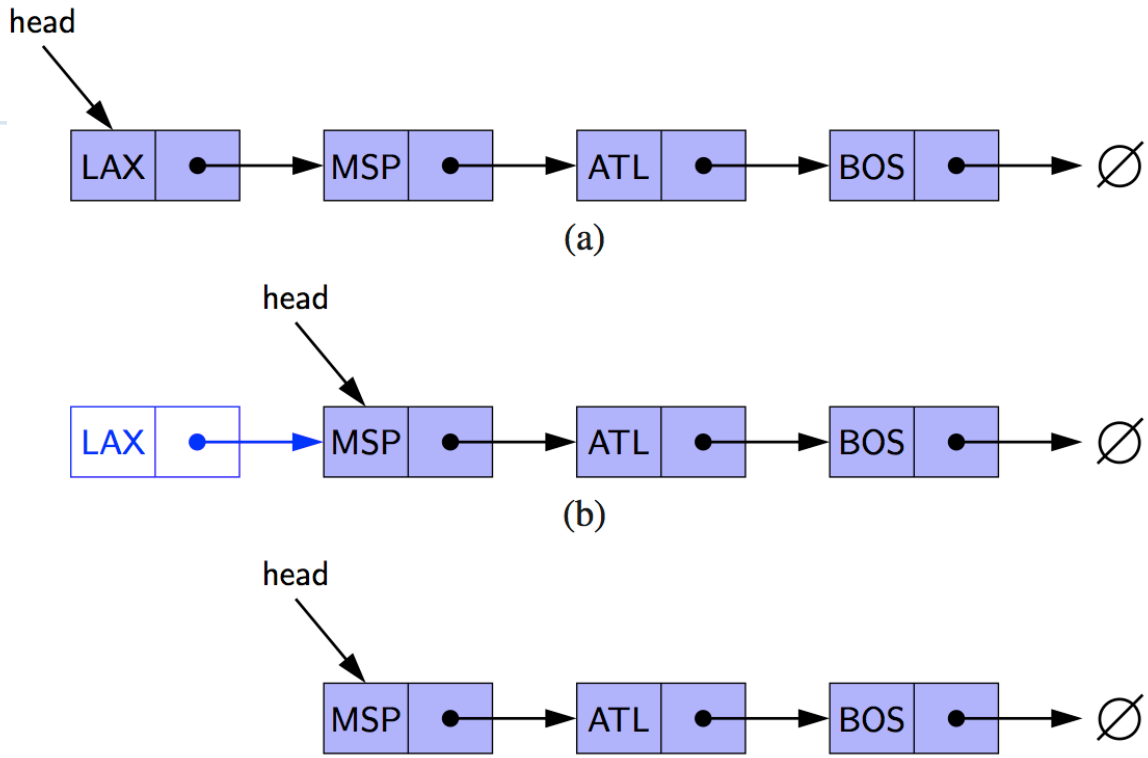
```
public void addFirst(E e) {            // adds element e to the front of the list
  head = new Node<>(e, head);          // create and link a new node
  if (size == 0)
    tail = head;                       // special case: new node becomes tail also
  size++;
}
public void addLast(E e) {             // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null);   // node will eventually be the tail
  if (isEmpty( ))
    head = newest;                     // special case: previously empty list
  else
    tail.setNext(newest);              // new node after existing tail
  tail = newest;                       // new node becomes the tail
  size++;
}
```





REMOVING AT THE HEAD

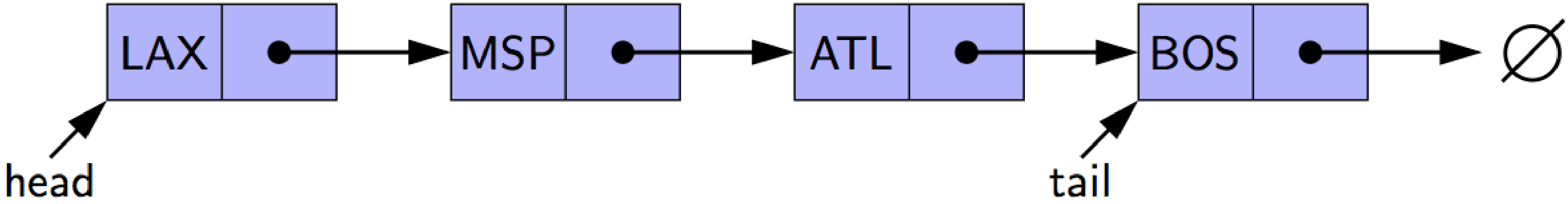
- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node




```
public E removeFirst( ) {                    // removes and returns the first element
  if (isEmpty( )) return null;               // nothing to remove
  E answer = head.getElement( );
  head = head.getNext( );                    // will become null if list had only one node
  size−−;
  if (size == 0)
    tail = null;                             // special case as list is now empty
  return answer;
}
}
```

REMOVING AT THE TAIL

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



CIRCULARLY LINKED LIST

- 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).
- Supports all of the public behaviors of our SinglyLinkedList class and one additional update method

rotate( ): Moves the first element to the end of the list.

- Nodes store:
  - element
  - link to the next node

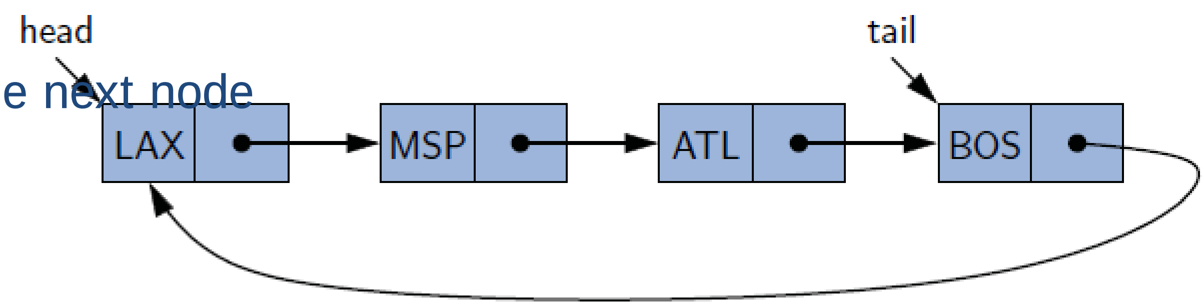


Figure 3.16: Example of a singly linked list with circular structure.

APPLICATION OF CIRCULARLY LINKED LIST

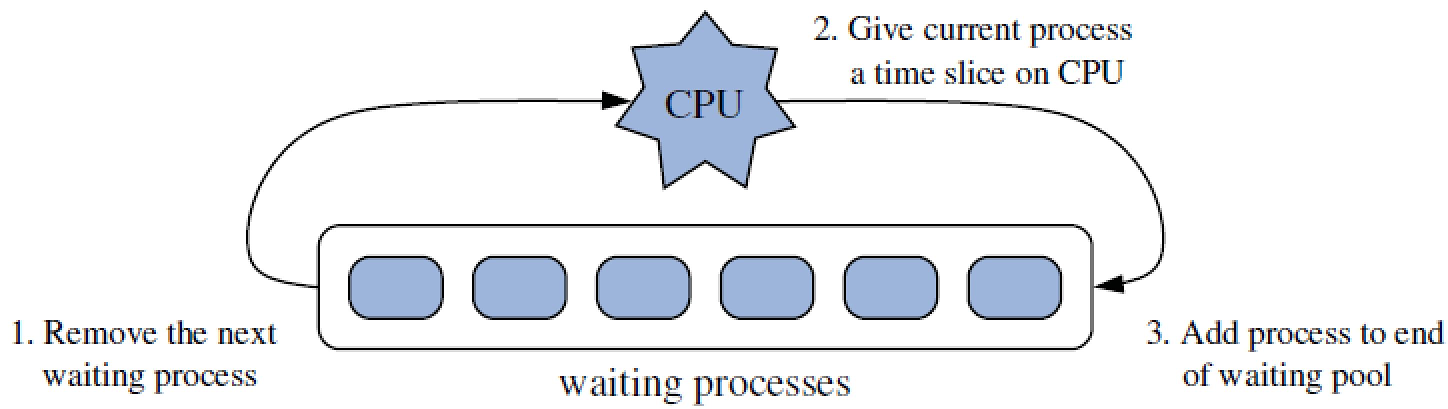
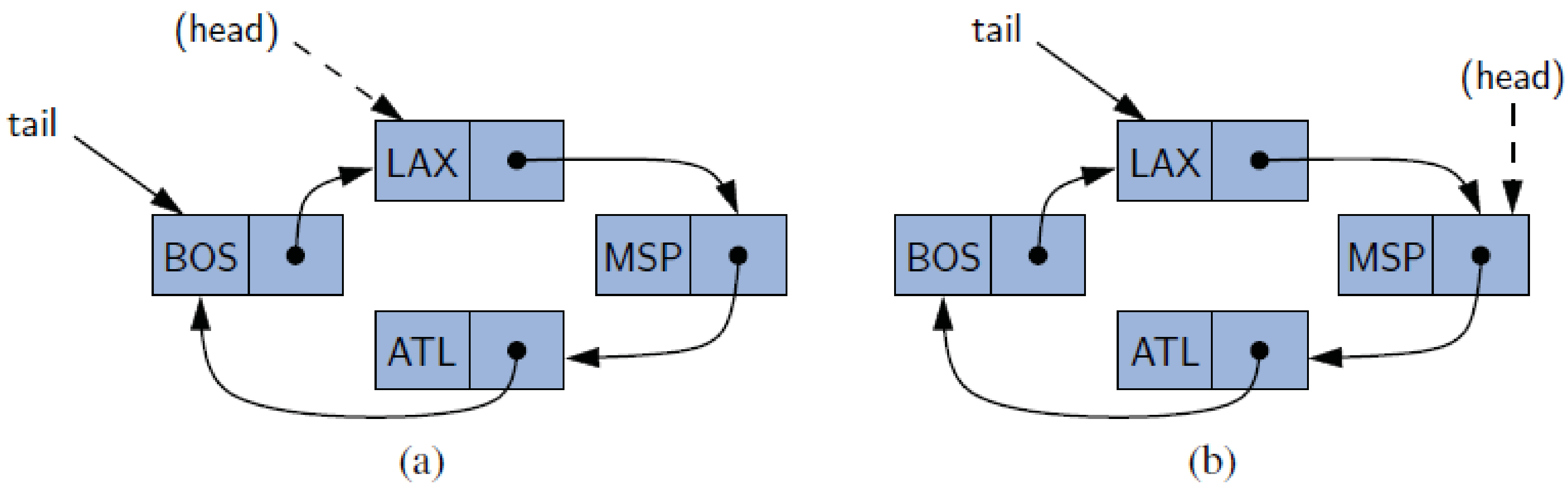


Figure 3.15: The three iterative steps for round-robin scheduling.

ROTATE() ON A CIRCULARLY LINKED LIST

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).

implicit head: tail.getNext( ).

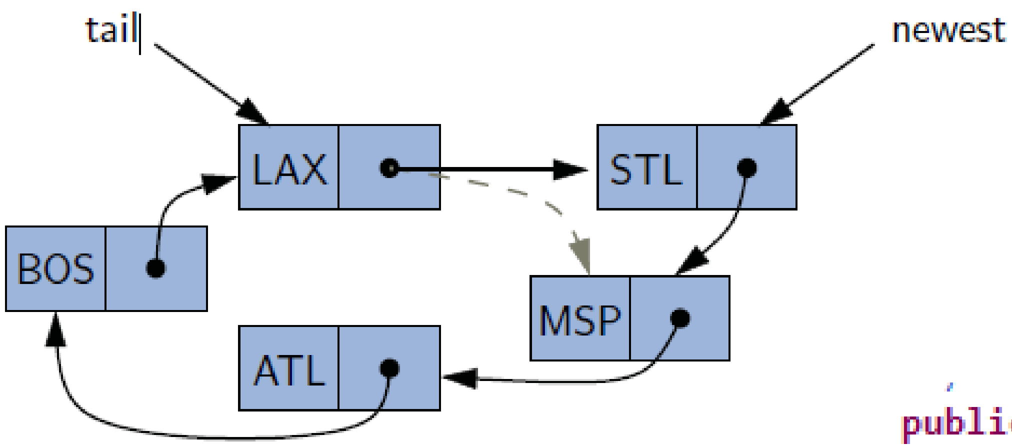



```
public void rotate() {          // rotate the first element to the back of the list
  if (tail != null)             // if empty, do nothing
    tail = tail.getNext();      // the old head becomes the new tail
}
```

Effect of a call to addFirst(STL) on the 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
}
```

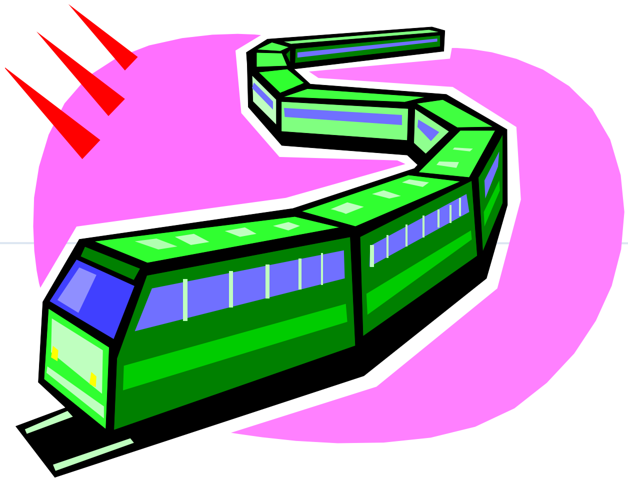


Removing at the tail is still not efficient

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.

```
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();
}
```

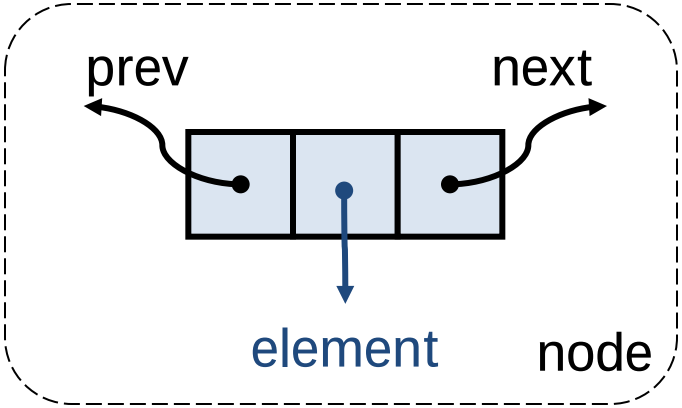
DOUBLY LINKED LISTS

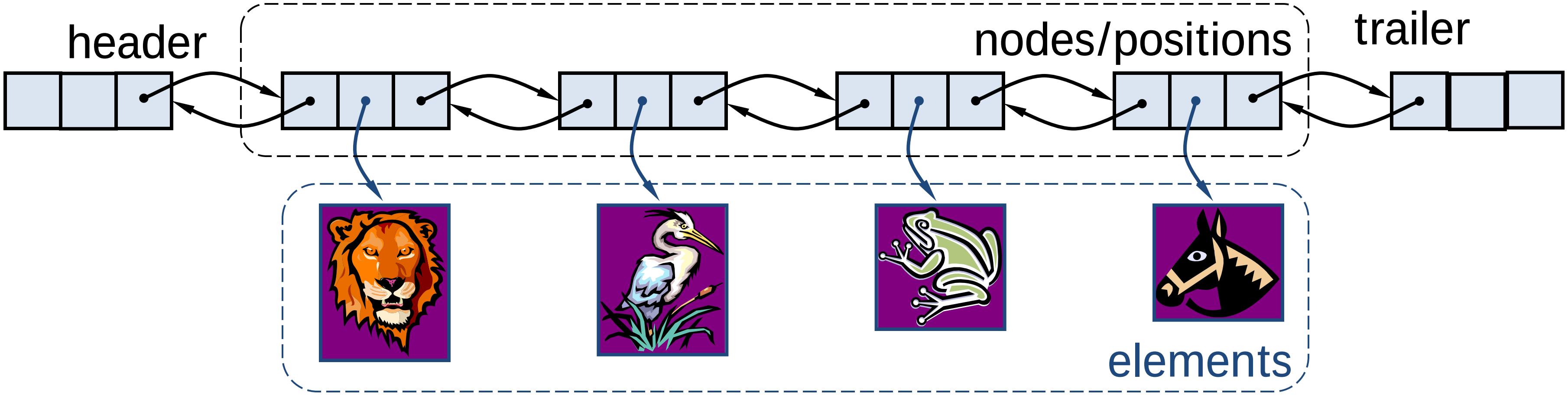


Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

DOUBLY LINKED LIST

- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





DOUBLY-LINKED LIST IN JAVA: NESTED CLASS NODE

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  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; }
  } //----------- end of nested Node class -----------

```

TRAILER AND HEADER NODES

- ***Description:***

  Header node at the beginning of the list, and a ***trailer*** node at the end of the list are known as ***sentinels*** (or guards), and they do not store elements of the primary sequence.

- ***Advantages:***
  - The header and trailer nodes never change
  - We can treat all insertions in a unified manner
    - because a new node will always be placed between a pair of existing nodes
  - Every element that is to be deleted is guaranteed to be stored in a node that has neighbors on each side.

```java
private Node<E> header;                      // header sentinel
private Node<E> trailer;                     // trailer sentinel
private int size = 0;                        // number of elements in the list
/** Constructs a new empty list. */
public DoublyLinkedList() {
  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
}
```

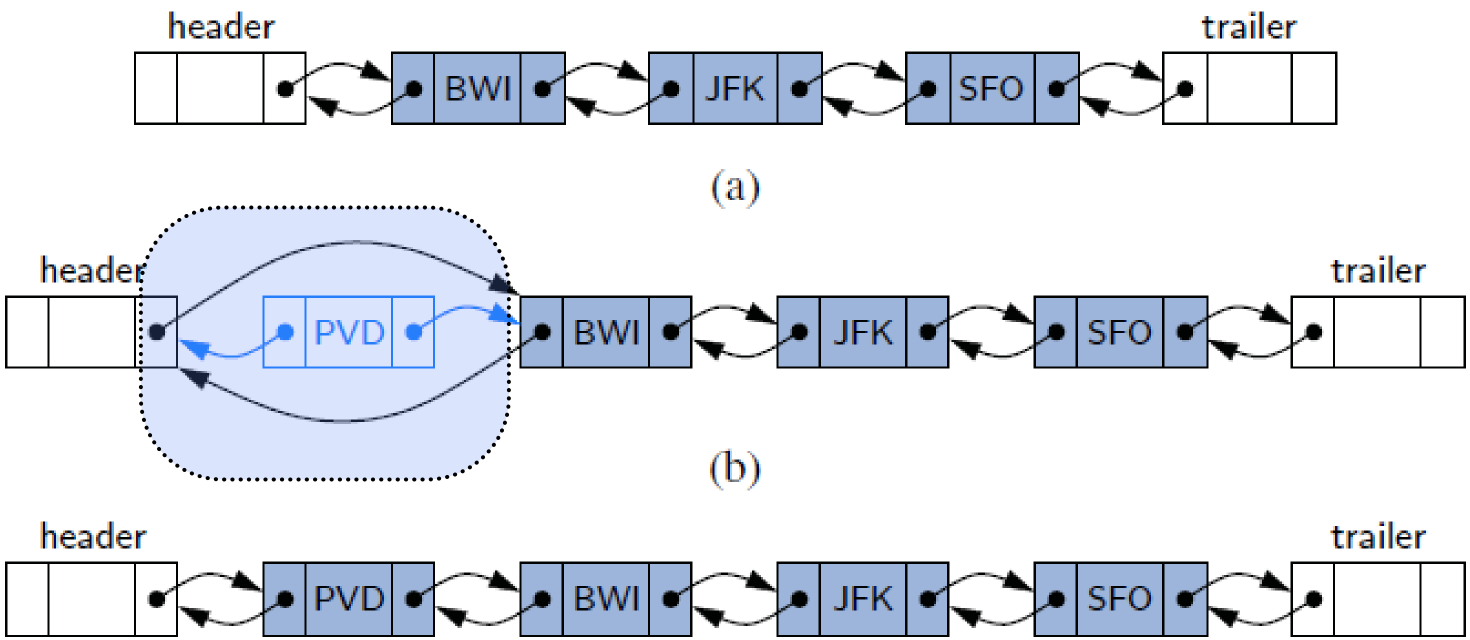
OPERATIONS IN DOUBLY LINKED LIST

- 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.

DOUBLY-LINKED LIST IN JAVA, 2

```
/** Returns the number of elements in the linked list. */
public int size( ) { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the first element of the list. */
public E first( ) {
  if (isEmpty( )) return null;
  return header.getNext( ).getElement( );          // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last( ) {
  if (isEmpty( )) return null;
  return trailer.getPrev( ).getElement( );         // last element is before trailer
}
```

INSERTION


header
trailer
BWI
JFK
SFO
(a)
header
PVD
BWI
JFK
SFO
trailer
(b)
header
PVD
BWI
JFK
SFO
trailer

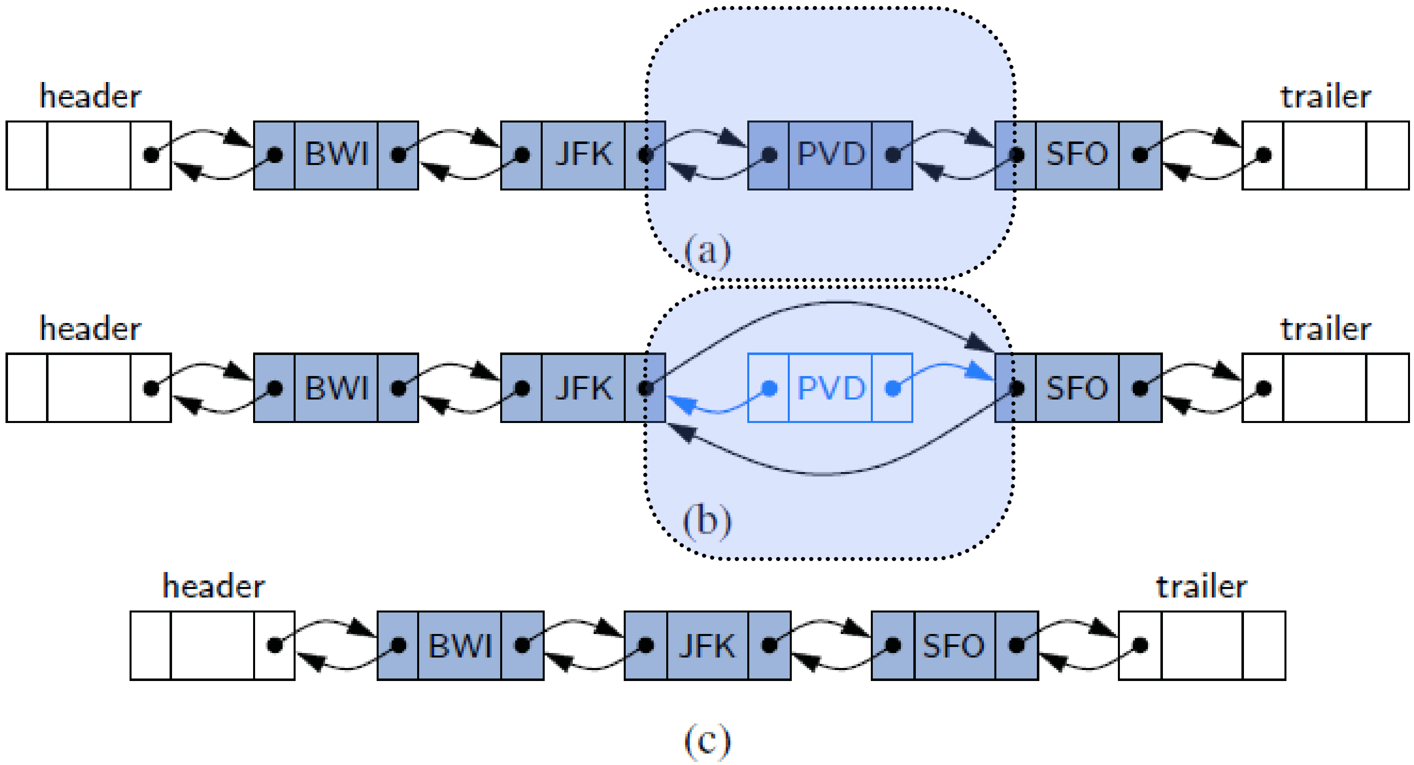
DOUBLY-LINKED LIST IN JAVA: INSERTION

We can treat all insertions in a unified manner

```
// public update methods
/** Adds element e to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext());          // place just after the header
}
/** Adds element e to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer);        // place just before the trailer
}
```

```
// private update methods
/** Adds element e to the linked list in between the given nodes. */
private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
  // create and link a new node
  Node<E> newest = new Node<>(e, predecessor, successor);
  predecessor.setNext(newest);
  successor.setPrev(newest);
  size++;
}
```

DELETION


header
BWI
JFK
PVD
SFO
trailer
(a)
header
BWI
JFK
PVD
SFO
trailer
(b)
header
BWI
JFK
SFO
trailer
(c)

DOUBLY-LINKED LIST IN JAVA: DELETION

We can treat all deletion in a unified manner

```
/** Removes and returns the first element of the list. */
public E removeFirst( ) {
  if (isEmpty( )) return null;                    // nothing to remove
  return remove(header.getNext( ));               // first element is beyond header
}
/** Removes and returns the last element of the list. */
public E removeLast( ) {
  if (isEmpty( )) return null;                    // nothing to remove
  return remove(trailer.getPrev( ));              // last element is before trailer
}
```

```
  /** Removes the given node from the list and returns its element. */
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev( );
    Node<E> successor = node.getNext( );
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size−−;
    return node.getElement( );
  }
} //----------- end of DoublyLinkedList class -----------
```

Ch3.5~6

JAVA SPECIFIC NOTES ON LINKED LIST

EQUIVALENCE TESTING

- an *equivalence relation* in mathematics, satisfying the following properties:

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

EQUIVALENCE TESTING WITH ARRAYS

Arrays are a reference type in Java, but not technically a class

a == b: Tests if a and b refer to the same underlying array instance.

a.equals(b): Interestingly, this is identical to a == b. Arrays are not a true class type and do not override the Object.equals method.

Arrays.equals(a,b): This provides a more intuitive notion of equivalence, returning **true** if the arrays have the same length and all pairs of corresponding elements are "equal" to each other. More specifically, if the array elements are primitives, then it uses the standard == to compare values. If elements of the arrays are a reference type, then it makes pairwise comparisons a[k].equals(b[k]) in evaluating the equivalence.

Arrays.deepEquals(a,b): Identical to Arrays.equals(a,b) except when the elements of a and b are themselves arrays, in which case it calls Arrays.deepEquals(a[k],b[k]) for corresponding entries, rather than a[k].equals(b[k]).

EQUIVALENCE TESTING WITH LINKED LISTS

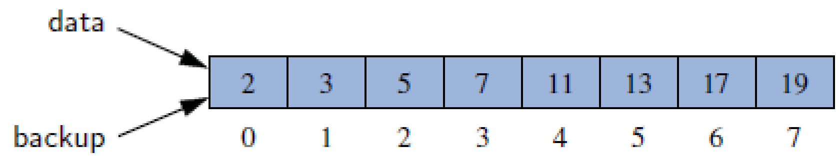
SinglyLinkedList class

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;        // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head;                                    // traverse the primary list
  Node walkB = other.head;                              // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement( ).equals(walkB.getElement( ))) return false; //mismatch
    walkA = walkA.getNext( );
    walkB = walkB.getNext( );
  }
  return true;    // if we reach this, everything matched successfully
}
```

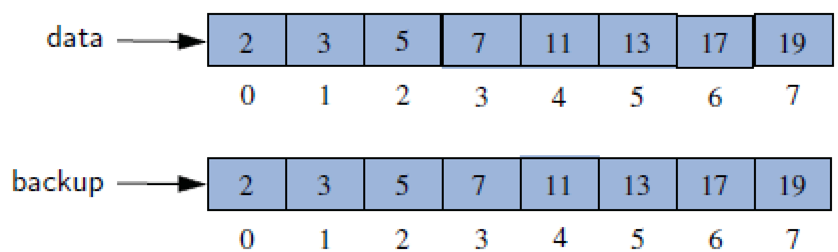
CLONING DATA STRUCTURES

Cloning Arrays

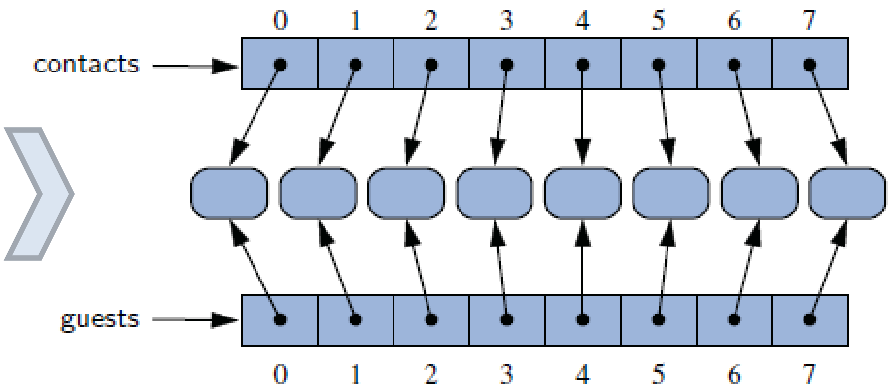
```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data;                              // warning; n
```




```
backup = data.clone( );
```



if the variable contacts refers to an array of hypothetical Person instances, the result of the command guests = contacts.clone( ) produces a **shallow copy**



A ***deep copy*** of the contact list can be created by iteratively cloning the individual elements

```
Person[ ] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
  guests[k] = (Person) contacts[k].clone( );
```

```
public static int[ ][ ] deepClone(int[ ][ ] original) {
  int[ ][ ] backup = new int[original.length][ ];
  for (int k=0; k < original.length; k++)
    backup[k] = original[k].clone( );
  return backup;
}
```

CLONING LINKED LISTS

shallow copy of the original

```
public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
  // always use inherited Object.clone() to create the initial copy
  SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe cast
  if (size > 0) {                          // we need independent chain of nodes
    other.head = new Node<>(head.getElement(), null);
    Node<E> walk = head.getNext();         // walk through remainder of original list
    Node<E> otherTail = other.head;        // remember most recently created node
    while (walk != null) {                 // make a new node storing same element
      Node<E> newest = new Node<>(walk.getElement(), null);
      otherTail.setNext(newest);           // link previous node to this one
      otherTail = newest;
      walk = walk.getNext();
    }
  }
  return other;
}
```