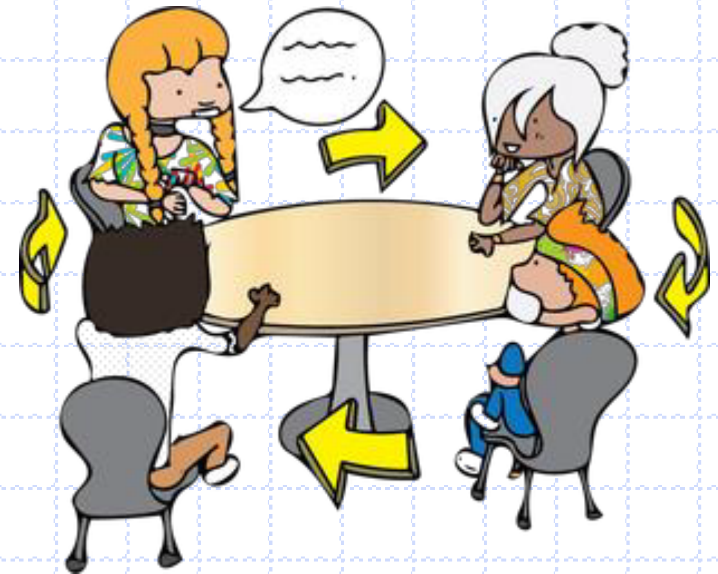
Circular linked Lists

→ مالها بداية ولا
نهاية + بس (Pointer) واحد الي هو (Tali

- Many applications process elements in cyclic fashion:
  - Multi-game players
  - City buses and subways
  - Round-robin scheduling

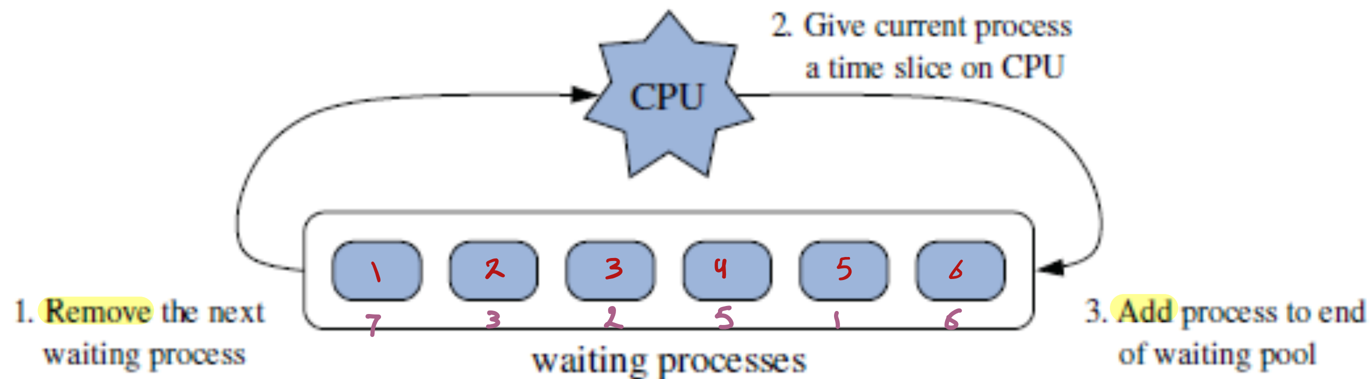


© 2014 Goodrich, Tamassia, Goldwasser

Round-Robin Scheduling

- It is the task of the operating system to schedule the many active processes on one or more processors.
- One method is called round-robin scheduling:
  - A process is given a time-slice (short time to execute),
  - When time-slice ends, process is interrupted, even if job not complete,
  - Each process takes its turn in taking a time slice in a circular fashion,
  - New processes can be added to the system,
  - Processes that complete their work are removed from the system.

Round-Robin Scheduling



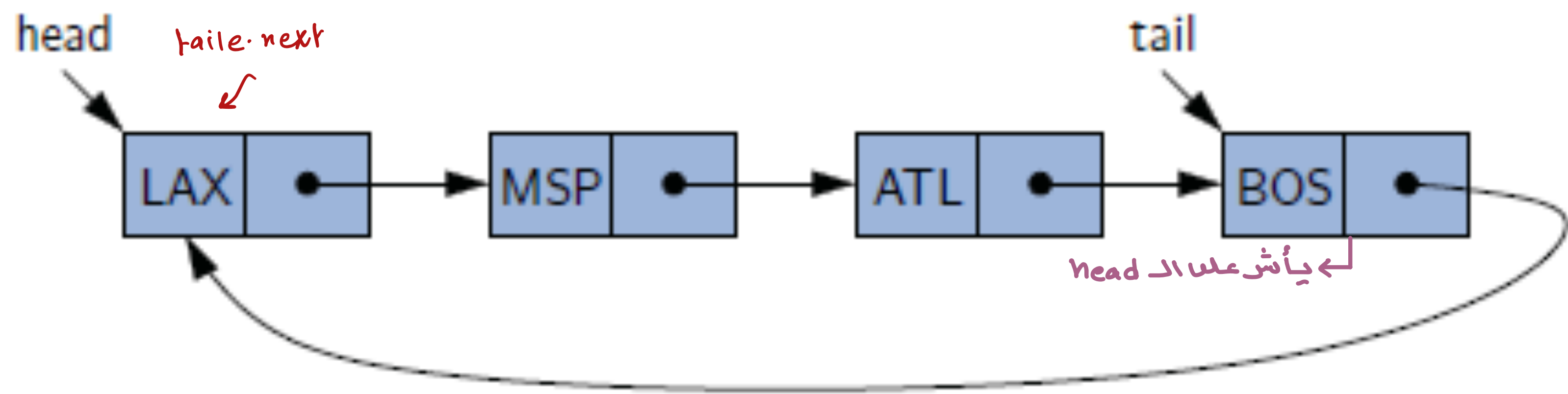
- Possible implementation with traditional SLL:

1. Process p=L.removeFirst()
2. Give a time slice to process p
3. If (not p.terminate()) then L.addLast(p)

Problem: Repeated removal/insertion of nodes.

Circular Linked-Lists

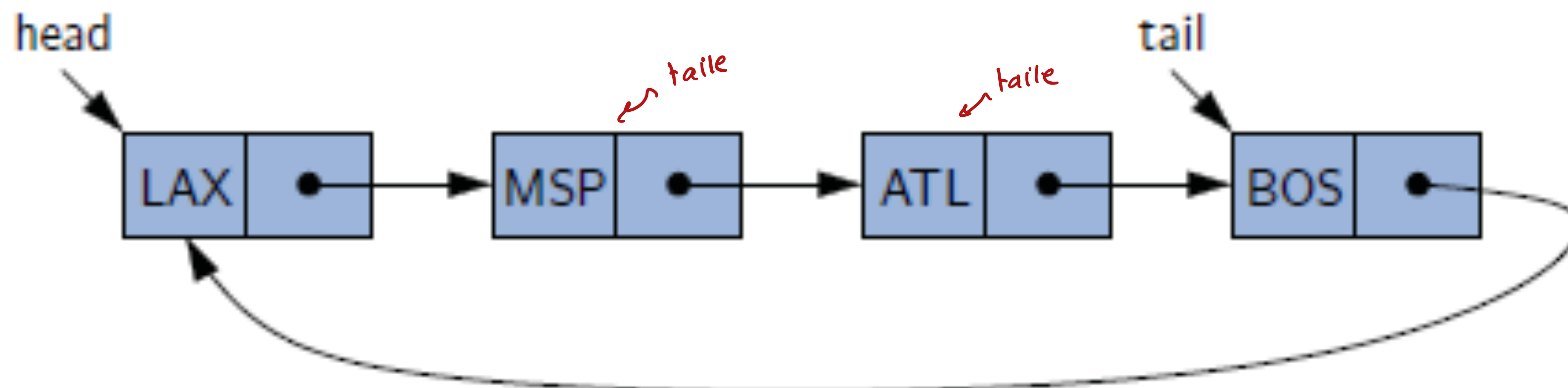
- In a circular linked-list, the tail points back to the head, rather than null value.



© 2014 Goodrich, Tamassia, Goldwasser

Circular Linked-Lists

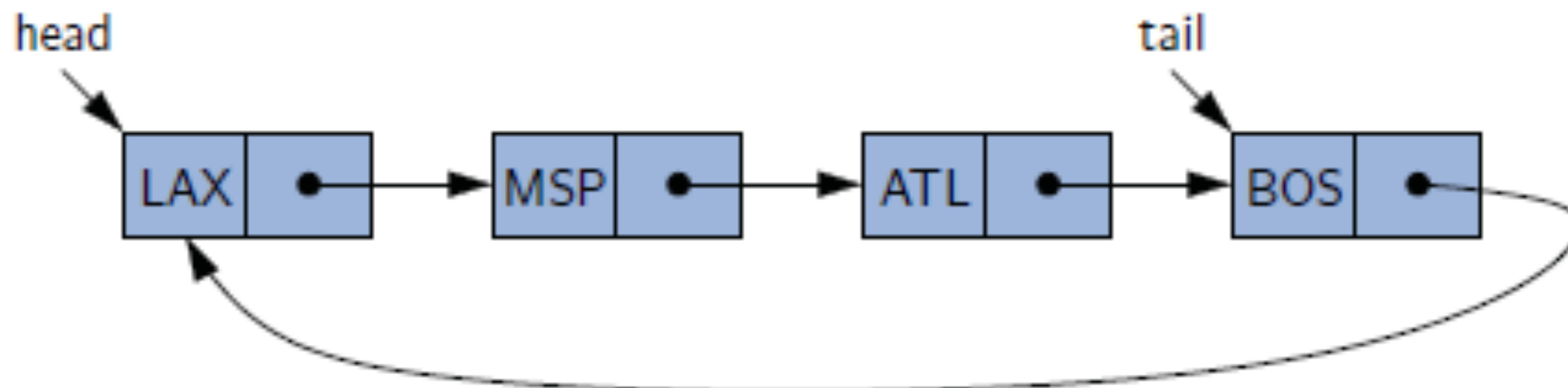
- The CircularLinkedList class supports all methods in SinglyLinkedList class, in addition to rotate() method.
- rotate(): moves the first element to the end of the list.



Circular Linked-Lists

- Round-robin can be efficiently implemented by these steps:
  1. Give time slice to process C.first()
  2. C.rotate()

tail = tail.next



Circular Linked-Lists: rotate()

- Further, there is no need for head, since it is tail.getNext().

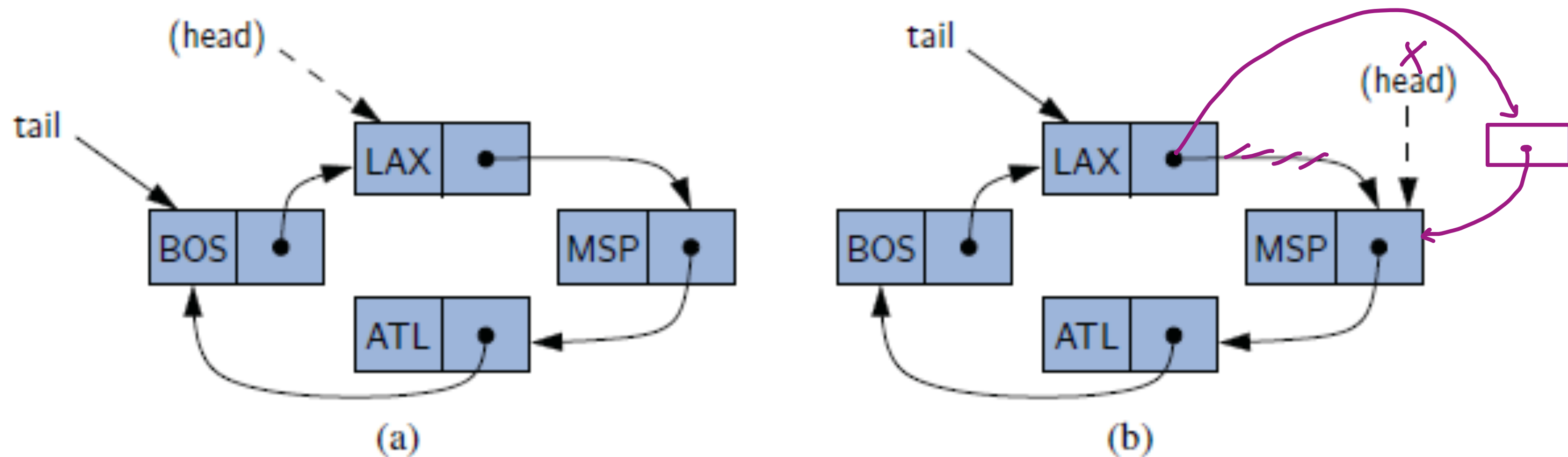


Figure 3.17: The rotation operation on a circularly linked list: (a) before the rotation, representing sequence { LAX, MSP, ATL, BOS }; (b) after the rotation, representing sequence { MSP, ATL, BOS, LAX }. We display the implicit head reference, which is identified only as tail.getNext() within the implementation.

© 2014 Goodrich, Tamassia, Goldwasser

Circular Linked-lists: addFirst()

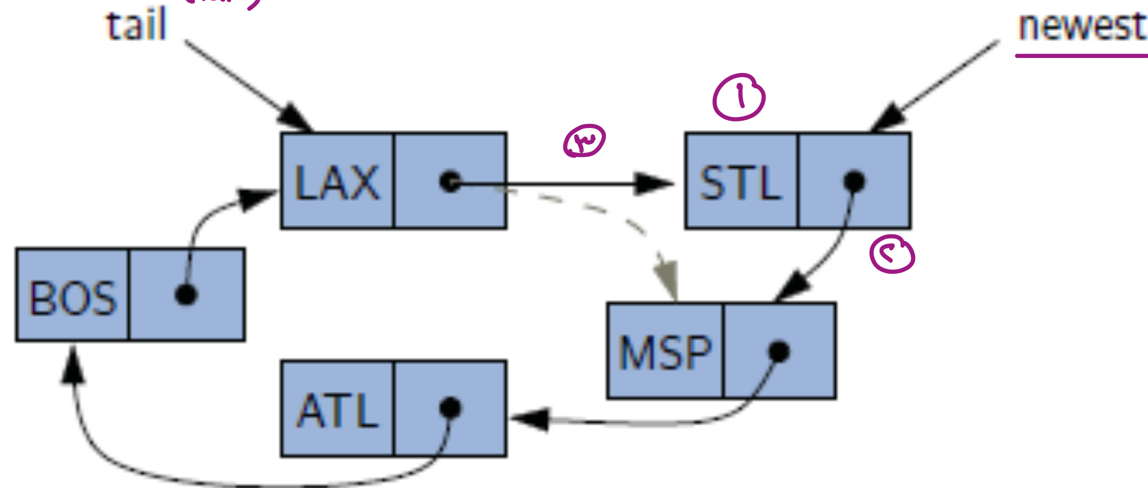
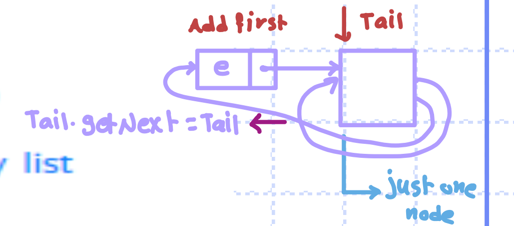
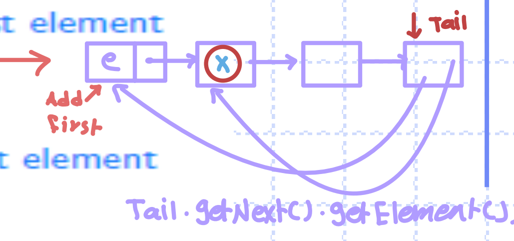
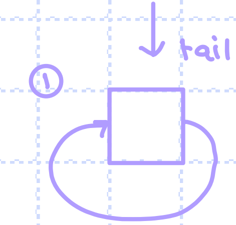


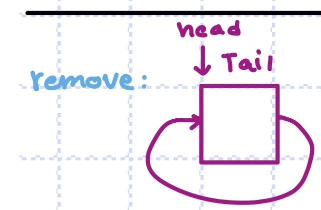
Figure 3.18: Effect of a call to addFirst(STL) on the circularly linked list of Figure 3.17(b). The variable newest has local scope during the execution of the method. Notice that when the operation is complete, STL is the first element of the list, as it is stored within the implicit head, tail.getNext().

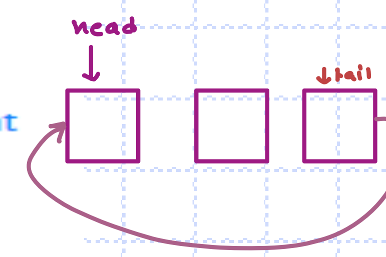
```
1   public class CircularlyLinkedList<E> {
...    (nested node class identical to that of the SinglyLinkedList class)
14     // instance variables of the CircularlyLinkedList
15     private Node<E> tail = null;                  // we store tail (but not head)
16     private int size = 0;                         // number of nodes in the list
17     public CircularlyLinkedList( ) { }            // constructs an initially empty list
18     // access methods
19     public int size( ) { return size; }
20     public boolean isEmpty( ) { return size == 0; }
21     public E first( ) {                  // returns (but does not remove) the first element
22       if (isEmpty( )) return null;
23       return tail.getNext( ).getElement( );       // the head is *after* the tail
24     }
25     public E last( ) {                   // returns (but does not remove) the last element
26       if (isEmpty( )) return null;
27       return tail.getElement( );
28     }
29     // update methods
30     public void rotate( ) {              // rotate the first element to the back of the list
31       if (tail != null)                           // if empty, do nothing
32         tail = tail.getNext( );                   // the old head becomes the new tail
33     }
34     public void addFirst(E e) {                   // adds element e to the front of the list
35       if (size == 0) {
36         tail = new Node<>(e, null);
37         tail.setNext(tail);                       // link to itself circularly
38       } else {
39         Node<E> newest = new Node<>(e, tail.getNext( ));
40         tail.setNext(newest);
41       }
42       size++;
43     }
44     public void addLast(E e) {                    // adds element e to the end of the list
45       addFirst(e);                                // insert new element at front of list
46       tail = tail.getNext( );                     // now new element becomes the tail
47     }
48     public E removeFirst( ) {                     // removes and returns the first element
49       if (isEmpty( )) return null;                // nothing to remove
50       Node<E> head = tail.getNext( );
51       if (head == tail) tail = null;              // must be the only node left
52       else tail.setNext(head.getNext( ));         // removes "head" from the list
53       size--;
54       return head.getElement( );
55     }
56  }
```



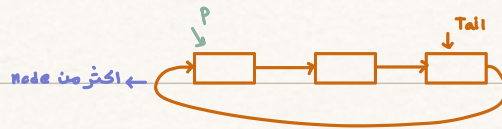


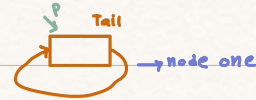






Traverse :





ابداء من اول node واعمل فيها Process

واحد وراء الثاني، لحد ما اوصل لاخر node

ماراح يمشي من الاصل

لان الـ P حتساوي الـ tail .

```
Node < E > P = Tail. getnext ()   → يوقف عند اول وحدة

While ( P ! = tail )

الي ابغاها ← Process  {  P. get element();

        P = P. get next (); → علشان احركه

        }

        P. getelement () ;
```

```
Public int numList Elements () {

 if (is Empty () return 0 ;

 Node < E > P = tail. getNext();

   int count = 1 ;

   While ( P != Tail )

   {

    P = P. getNext();

    count ++ ;
    }

    return count ;
    }
    }
```

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

Lists and Iterators

Iterators

- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

function

hasNext(): Returns true if there is at least one additional element in the sequence, and false otherwise.

next(): Returns the next element in the sequence.

The Iterable Interface

- Java defines a parameterized interface, named Iterable, that includes the following single method:
  - iterator( ): Returns an iterator of the elements in the collection.
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
- Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

The for-each Loop

- Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax:

```
for (ElementType variable : collection) {
  loopBody                          // may refer to "variable"
}
```

for (int x : Array)

is equivalent to:

Type

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                          // may refer to "variable"
}
```