CS 242 - CS252
Data Structures

Linked lists – Part 2

Singly Linked List ADT-Extra operation

- Search List
- Traverse List
- Retrieve Node
- Destroy List

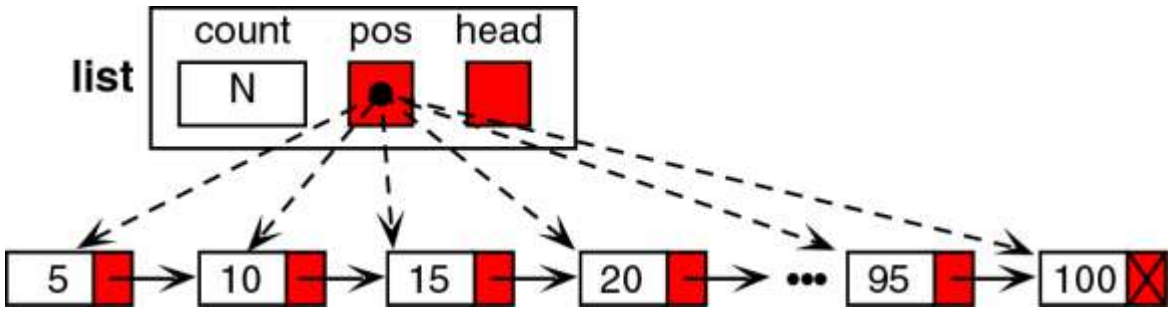
Singly Linked List ADT-Extra operation

Search List

- Locating specific data in a list.
- In linked list we use sequential search.
- We need a key field for search operation.

Traverse List

- Start with the head and access each node until you reach null.
- Do not change the head reference.



Singly Linked List ADT-Extra operation

Retrieve Element

- By using search method.

Destroy List

- Deletes all nodes still in the list.
- It then sets the metadata to a null list condition

Singly Linked List ADT: Extra methods

- **Traverse the list by Curr reference (helper methods)**
  - **next():** change Curr position to the next node.
  - **moveToStart():** change Curr position to the first of list.
  - **moveToEnd():** change Curr position to the end of list.
  - **moveToPos(pos):** move Curr to specific position(pos)
  - **CurrPos():** return the position (index) of Curr.
  - **getValue():** return the element of Curr reference.

Singly linked list implementation: helper methods

```
,,
public void moveToStart()
{
    curr=head;
}
public void moveToEnd()
{
    curr=tail;
}

public void next()
{
if(curr!=tail)
    curr=curr.getNext();
}

public E getValue()
{
    return curr.getElement();
}
```

Singly linked list implementation: helper methods

```
//return the postion of current element
    public int CurrPos()
    {
        Node<E> temp=head;
        int i=0;
        while(temp!=curr)
        {
            temp=temp.getNext();
            i++;
        }
        return i;
    }
```

Singly linked list implementation: helper methods

```
//move curr to postion
    public void moveToPos(int pos)
    {
        if(pos<0 || pos>=size)
            System.out.println("Position out of range!");
        else
        {
        curr=head;
        for(int i=0;i<pos;i++)
        {
            curr=curr.getNext();
        }
        }
    }
```

Comparison of List Implementations

Array Based List	Linked lists
their size must be predetermined before the array can be allocated.	They only need space for the objects actually on the list
Array-based lists cannot grow beyond their predetermined size.	There is no limit to the number of elements on a linked list,
Whenever the list contains only a few elements, a substantial amount of space might be tied up in a largely empty array	as long as there is free-store memory available.
there is no wasted space for an individual element	Linked lists require that an extra pointer be added to every list node.
Array-based lists are faster for random access by position. Positions can easily be adjusted forwards or backwards by the next and prev methods. These operations always take O(1) time.	singly linked lists have no explicit access to the previous element, and access by position requires that we march down the list from the front (or the current position) to the specified position. Both of these operations require O(n) time in the average and worst cases

Linked List

- Advantages
  - Data easily inserted and deleted
  - No need to shift elements of LL to make room for a new element or to delete an element
- Disadvantages
  - We are limited to a sequential search

Circularly Linked Lists

Many applications in which data can be more naturally viewed as having a cyclic order with well-defined neighboring relationships, but no fixed beginning or end

- Multiplayer games
- City buses
- Operating system

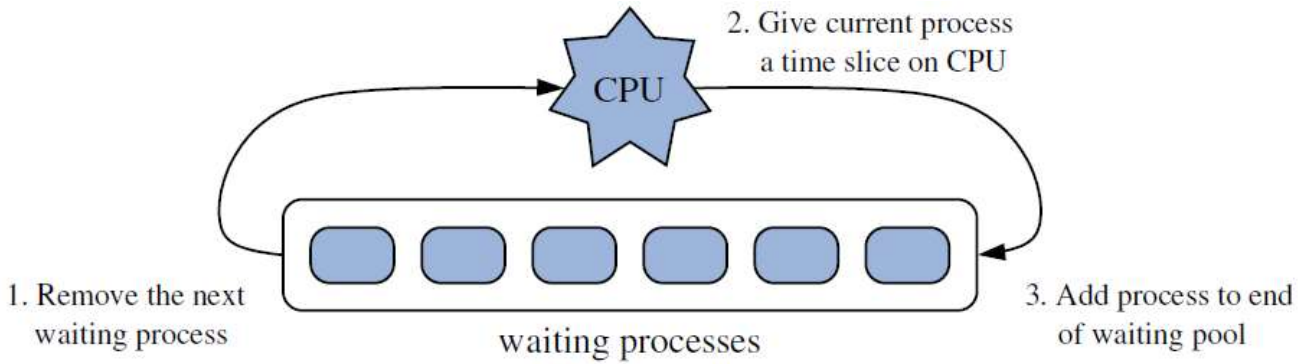
Round-robin scheduling

A process is given a short turn to execute, known as a ***time slice***

- The slice ends and the job is complete.
- The slice ends and the job is not yet complete.

- A traditional linked list, by repeatedly performing the following steps on linked list *L*

1. process *p* = *L*.removeFirst( )
2. Give a time slice to process *p*
3. *L*.addLast(*p*)



Round-robin scheduling

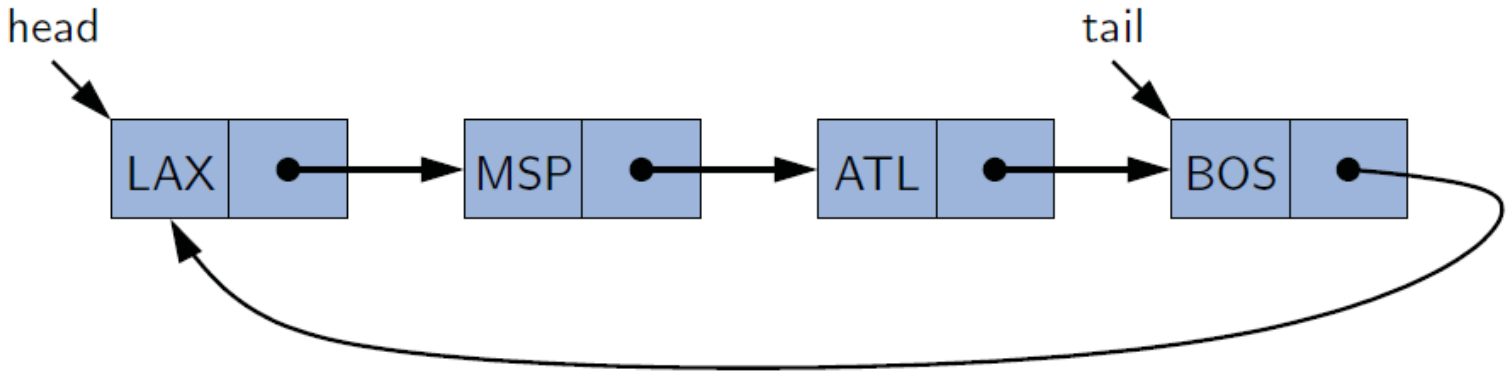
Drawbacks to the use of a traditional linked list

- Inefficient to remove a node from one end of the list and then create a new node for the same element.
- Decrement and increment the list's *size*
- Unlink and relink nodes

- Can we build a more efficient data structure for representing a cyclic order
  - A modification to our singly linked list implementation

Circularly linked list

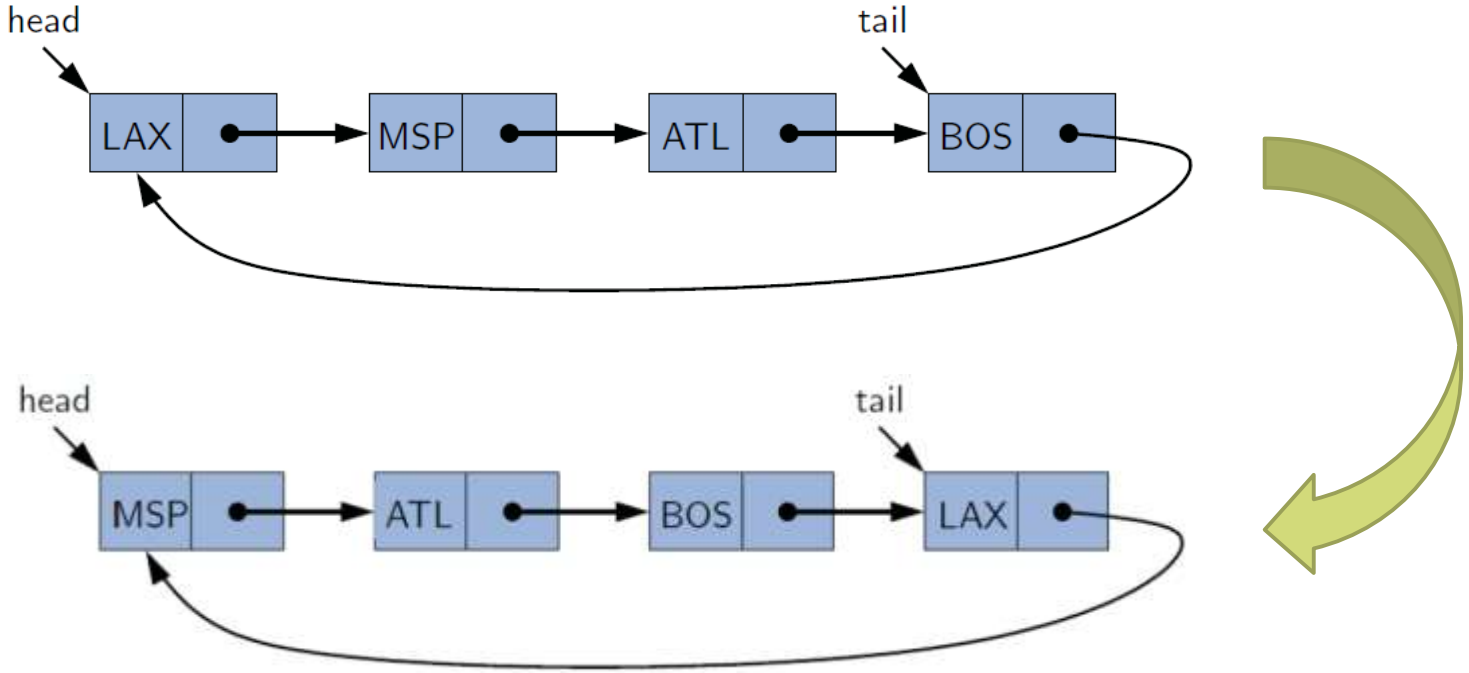
- A singularly linked list in which the next reference of the tail node is set to refer back to the head of the list (rather than null)



Circularly linked list

Supports all the public methods of our SinglyLinkedListclass

- One additional update method ***rotate( )***
- Moves the first element to the end of the list.



CircularlyLinkedList class

- Round-robin scheduling can be efficiently implemented by repeatedly performing the following steps on a circularly linked list *C*
  1. Give a time slice to process C.first( )
  2. C.rotate( )

- Additional optimization
  - We no longer explicitly maintain the head reference
  - We can locate the head as tail.getNext( )

CircularlyLinkedList

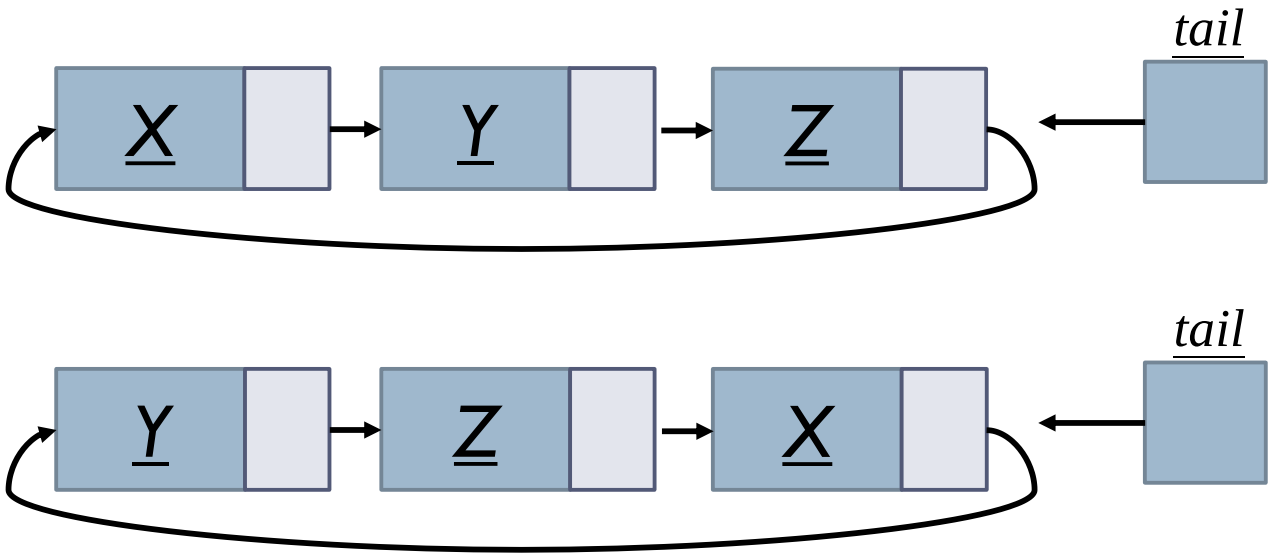
```
1   public class CircularlyLinkedList<E> {
...     (nested node class identical to that of the SinglyLinkedList class)
14      // instance variables of the CircularlyLinkedList
15      private Node<E> tail = null;                // we store tail (but not head)
16      private int size = 0;                       // number of nodes in the list
17      public CircularlyLinkedList( ) { }          // constructs an initially empty list
```

CircularlyLinkedList

```
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    }
```

CircularlyLinkedList

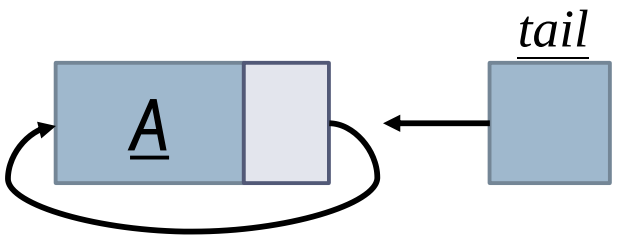
```
29    // update methods
30    public void rotate( ) {          // rotate the first element to the back of the list
31      if (tail != null)                      // if empty, do nothing
32        tail = tail.getNext( );              // the old head becomes the new tail
33    }
```



CircularlyLinkedList (addFirst)

```
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    }
```

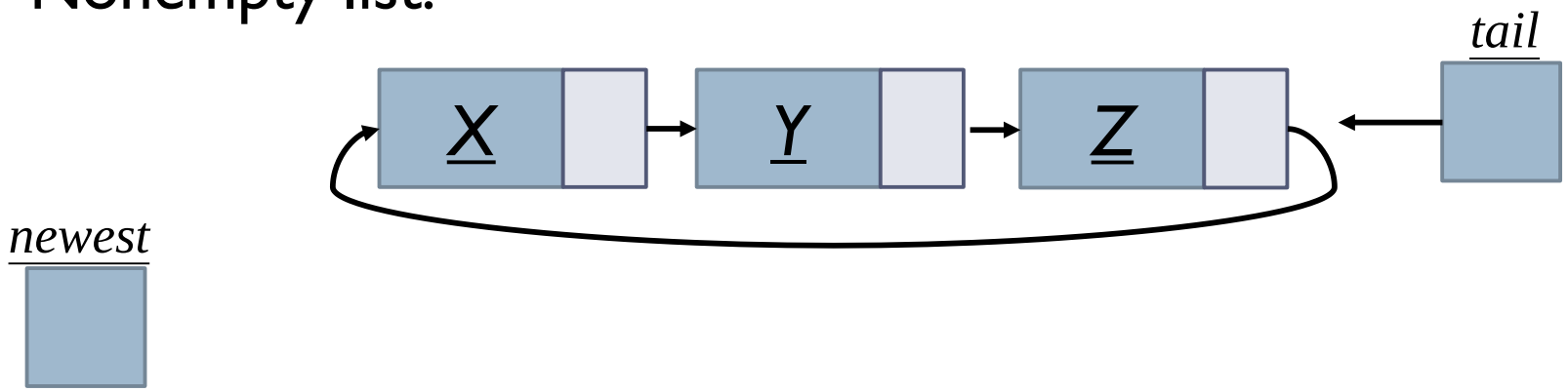
- Empty list:



CircularlyLinkedList (addFirst)

```
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    }
```

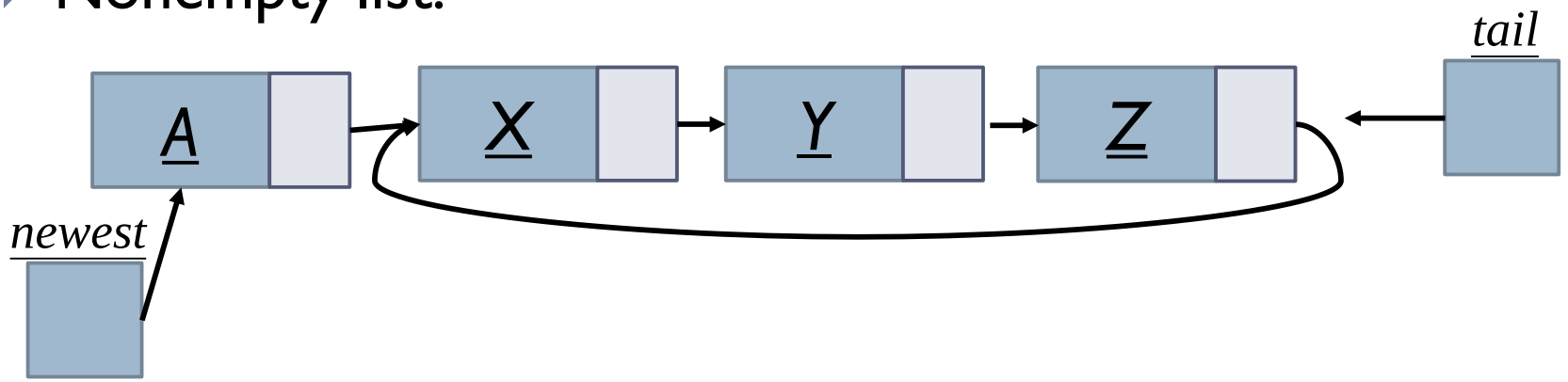
- Nonempty list:



CircularlyLinkedList (addFirst)

```
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    }
```

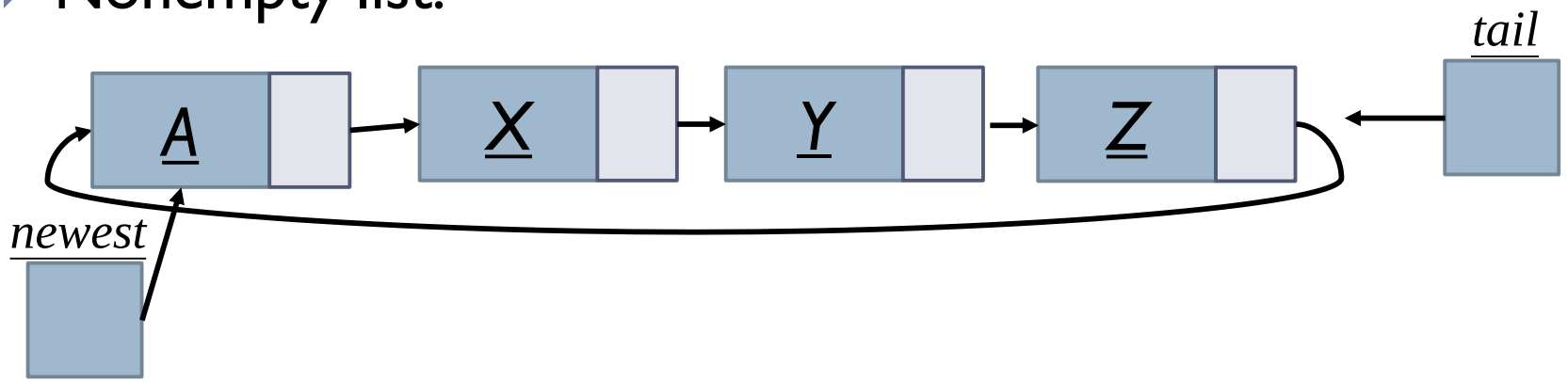
- Nonempty list:



CircularlyLinkedList (addFirst)

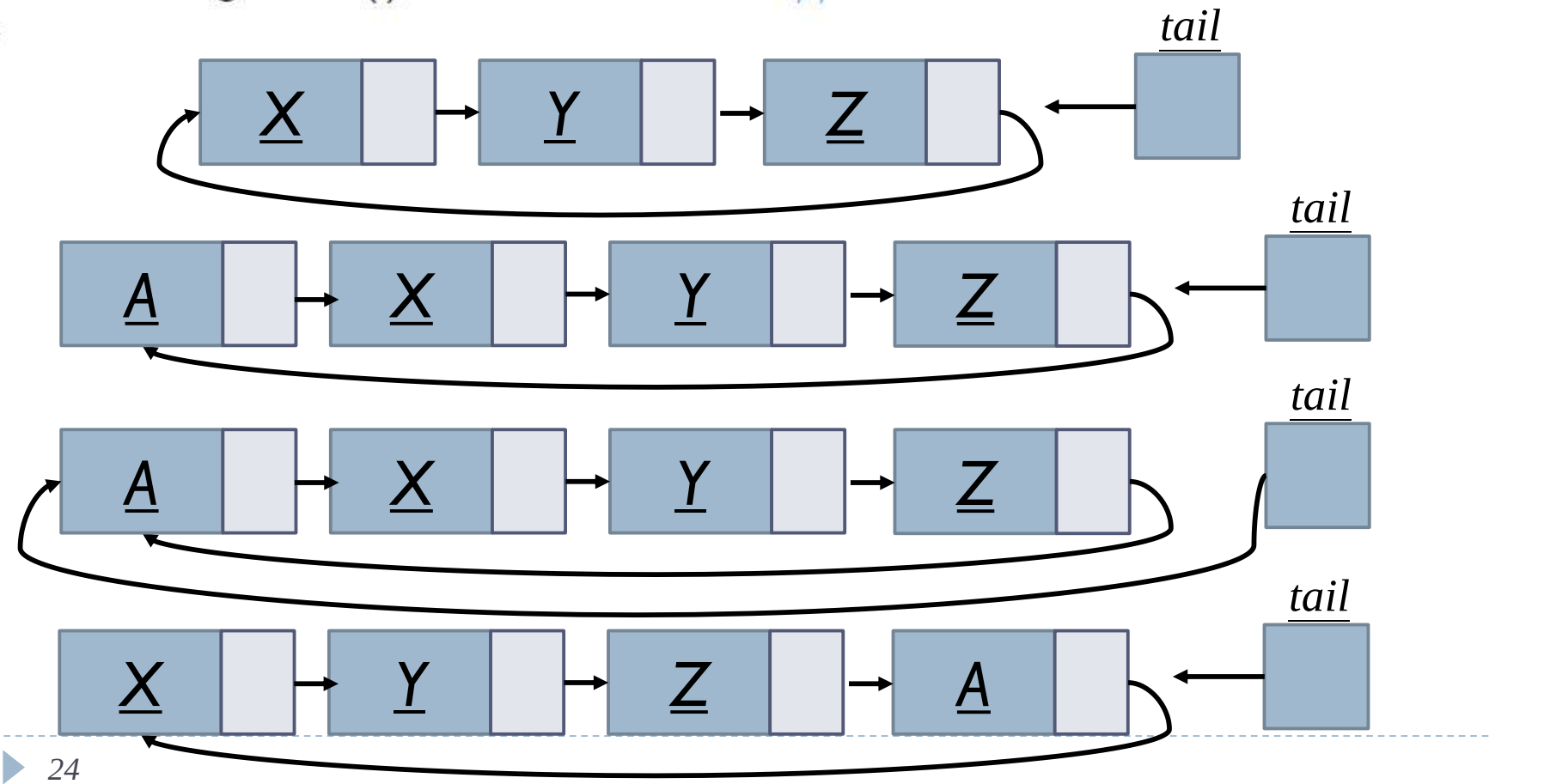
```
34    public void addFirst(E e) {                  // adds element e to the front of the list
35      if (size == 0) {
36        tail = new Node<>(e, null);
37        tail.setNext(tail);                      // link to itself circularly
38      } else {
39        Node<E> newest = new Node<>(e, tail.getNext( ));
40        tail.setNext(newest);
41      }
42      size++;
43    }
```

- Nonempty list:



CircularlyLinkedList (addLast)

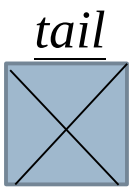
```
public void addLast(E e) {                  // adds element e to the end of the list
  addFirst(e);                              // insert new element at front of list
  tail = tail.getNext( );                   // now new element becomes the tail
}
```



CircularlyLinkedList (removeFirst)

```
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( );
  }
}
```

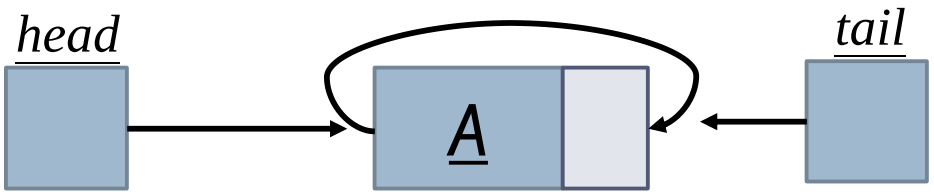
- Empty list:



CircularlyLinkedList (removeFirst)

```
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( );
  }
}
```

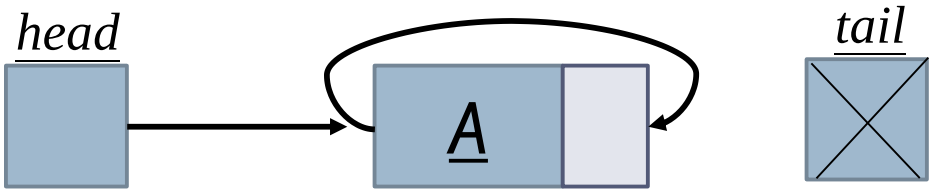
- List with only one node:



CircularlyLinkedList (removeFirst)

```
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( );
  }
}
```

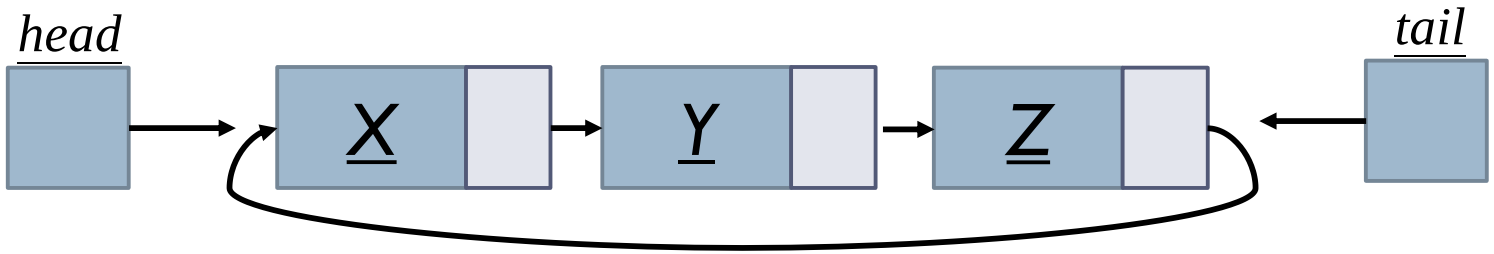
- List with only one node:



CircularlyLinkedList (removeFirst)

```
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( );
  }
}
```

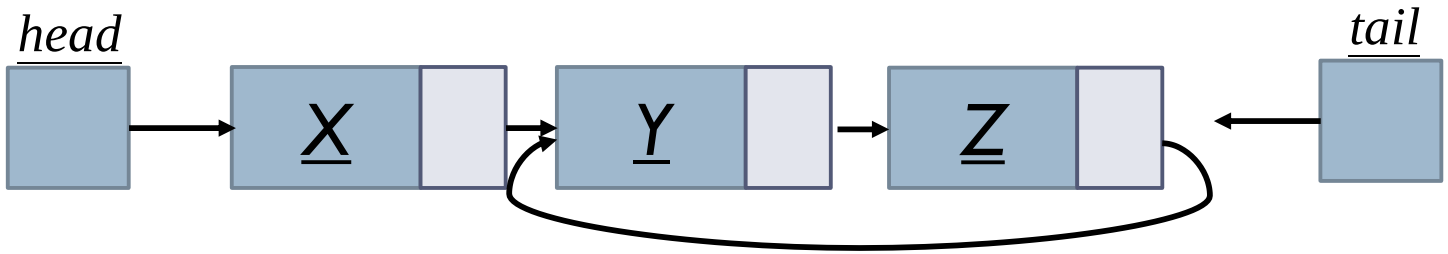
- List with more than one node:



CircularlyLinkedList (removeFirst)

```
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( );
  }
}
```

- List with more than one node:



Exercise

- Describe a method for finding the middle node of a doubly linked list with header and trailer sentinels by "link hopping," and without relying on explicit knowledge of the size of the list. In the case of an even number of nodes, report the node slightly left of center as the "middle." What is the running time of this method?