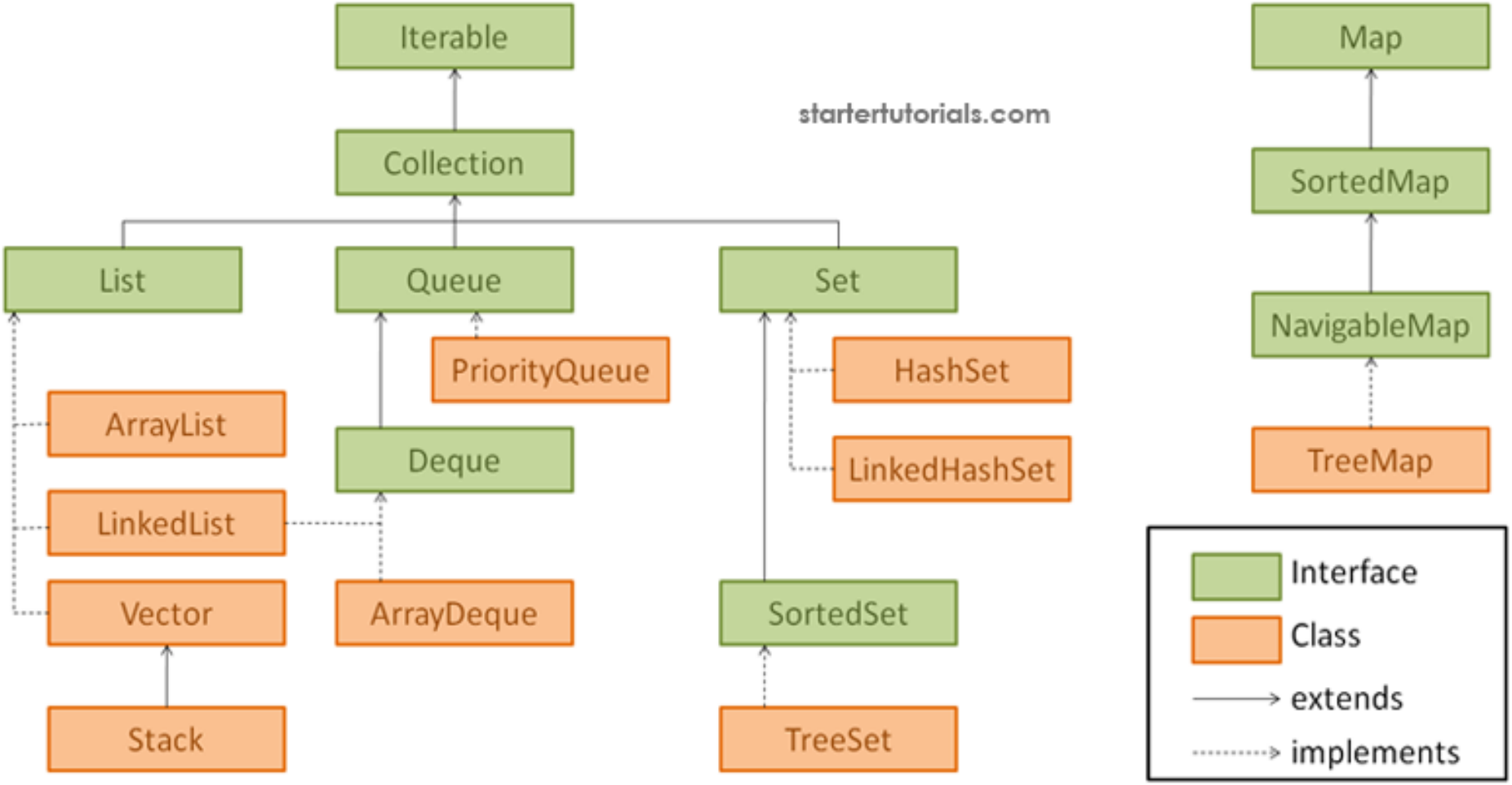
FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES (CTDL)

Semester 1, 2021/2022

Java Collection framework



Looping in Arrays

How to iterate elements in an array?

```
int[] array = {1, 3, 5, 7, 9};
for(int i=0; i<array.length; i++) {
   //...
}
```

```
int[] array = {1, 3, 5, 7, 9};
for(int e : array) {
   //...
}
```

FOR EACH

Similar:

WHILE

DO … WHILE

Another approach is using Iterator!!!

Iterator

Iterator

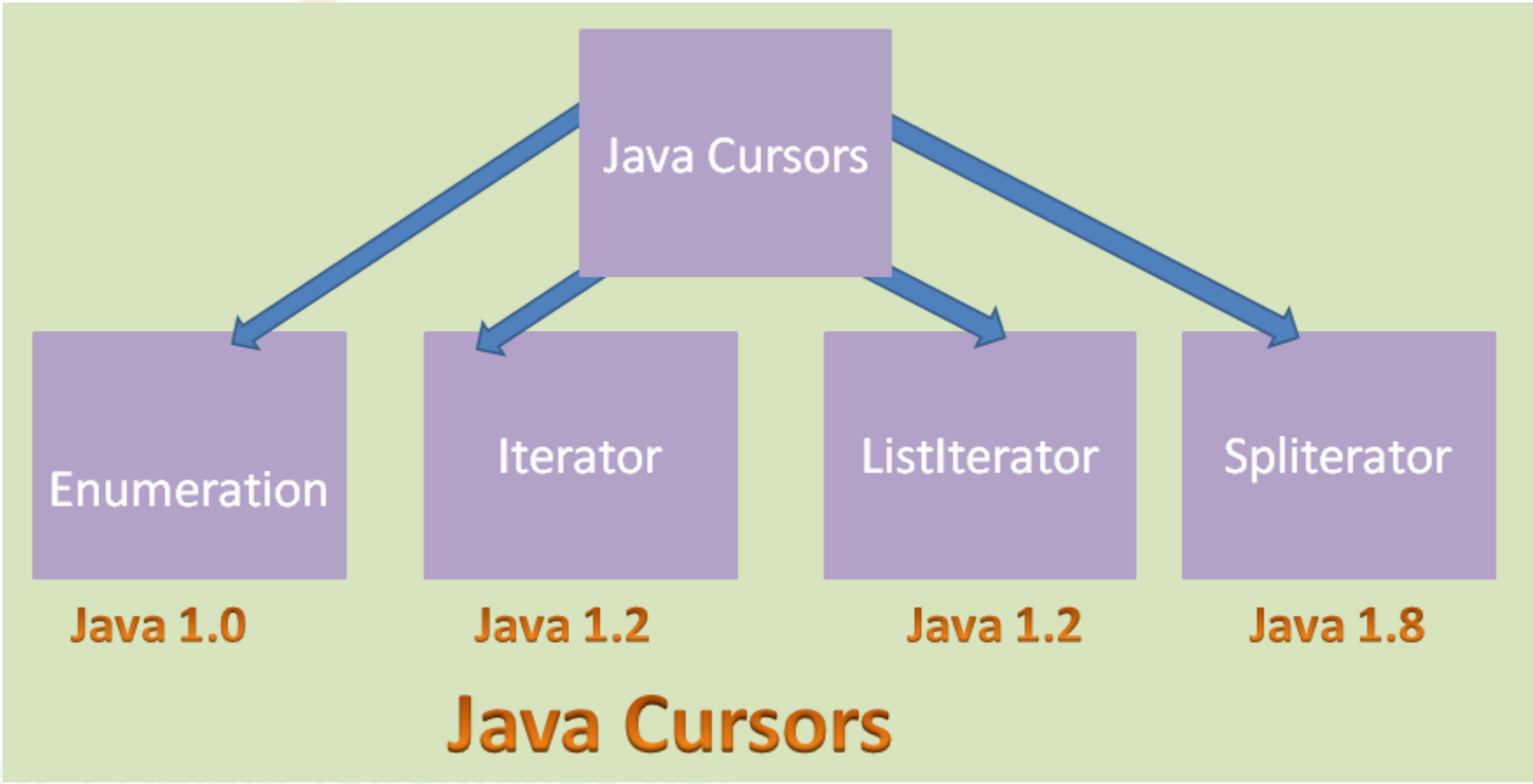
- **Iterator**: an interface which belongs to collection framework.
- Allows us to traverse the collection, access the data element and remove the data elements of the collection.

Every data structure in java can convert to iterator by method iterator()

Types of Java Iterators

- Java has four cursors: **Enumeration**, **Iterator**, **ListIterator**, and **Spliterator** (from JDK 8).
- Two main types:
  - Uni-Directional Iterators:
    - supports only Forward Direction iterations.
    - For instance, **Enumeration**, **Iterator**, etc. are Uni-Directional Iterators.
  - Bi-Directional Iterators:
    - supports Both Forward Direction and Backward Direction iterations.
    - For instance, **ListIterator** is Bi-Directional Iterator.

Types of Java Iterators



Methods

- **boolean hasNext()**:
  - returns true if Iterator has more element to iterate.
- **E next()**:
  - returns the next element in the collection until the hasNext()method return true.
  - This method throws 'NoSuchElementException' if there is no next element.
- **void remove()**:
  - removes the current element in the collection.
  - This method throws 'IllegalStateException' if this function is called before next( ) is invoked.

Iterator usage

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

```
Integer[] array = { 1, 3, 5, 7, 9 };
Iterator<Integer> iter = Arrays.asList(array).iterator();
while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

Spliterator

- JDK 8 introduced the new type of iterator called Spliterator.

- Spliterator supports parallel programming. But, you can use Spliterator even if you don't need parallel execution.

- A Spliterator may traverse elements individually (tryAdvance()) or sequentially in bulk (forEachRemaining()).

Example

- List of countries:

```
List<String> countries = new ArrayList<String>();

countries.add("Australia");
countries.add("Canada");
countries.add("India");
countries.add("USA");
```

Example: Iterator

- Using iterator to access elements:

```
System.out.println("Using Iterator to display contents of countries list");
// Using Iterator to display contents of countries list
Iterator<String> iter1 = countries.iterator();
while (iter1.hasNext()) {
    String element = iter1.next();
    System.out.print(element + " ");
}
```

Example: ListIterator

- Using ListIterator to access elements:

```
// Using ListIterator to display contents of countries in reverse order
ListIterator<String> iter2 = countries.listIterator();
while (iter2.hasNext()) {
    String element = iter2.next();
    iter2.set(element + "1");
}

// Using ListIterator to display contents of countries in reverse order
while (iter2.hasPrevious()) {
    String element = iter2.previous();
    System.out.print(element + " ");
}
```

Example: ListIterator

- Using Spliterator to access elements:

```
System.out.println("\n\nUsing Spliteror tryAdvance() to loop through countries list");
// Using Spliterator tryAdvance() to loop through countries list
Spliterator<String> iter3 = countries.spliterator();
while (iter3.tryAdvance(country -> System.out.print(country + " ")));

System.out.println("\n\nUsing Spliteror forEachRemaining() to loop through countries list");
// Using Spliterator forEachRemaining() to loop through countries list
Spliterator<String> iter4 = countries.spliterator();
iter4.forEachRemaining(country -> System.out.print(country + " "));
```

Inserting in Arrays

How to Insert New Element into an Array?

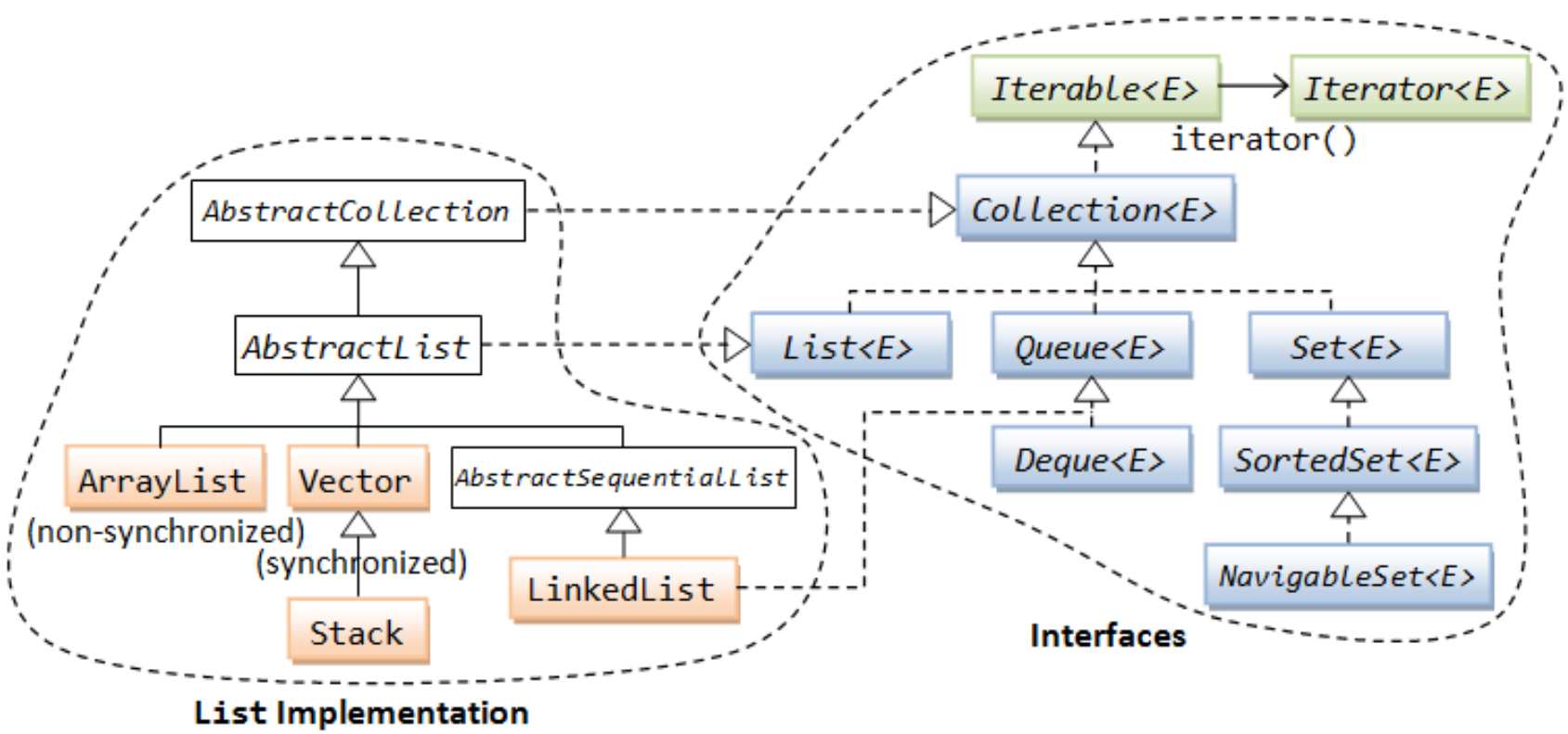
```
// Insert element into array at index i;
public static int[] insert(int[] array, int i, int element) {
    // TODO
    return array;
}
```

- Insert 10 at index 2:

Notes on Arrays

- The most fundamental collection data type
- **Fixed length** data structure
- **read** or **write** a list item by referring to its index in **constant time.**
- However, some array operations – such as **add**, **remove**, and **search** – can be quite **expensive**, with worst-case **linear time complexity (O(n)).**

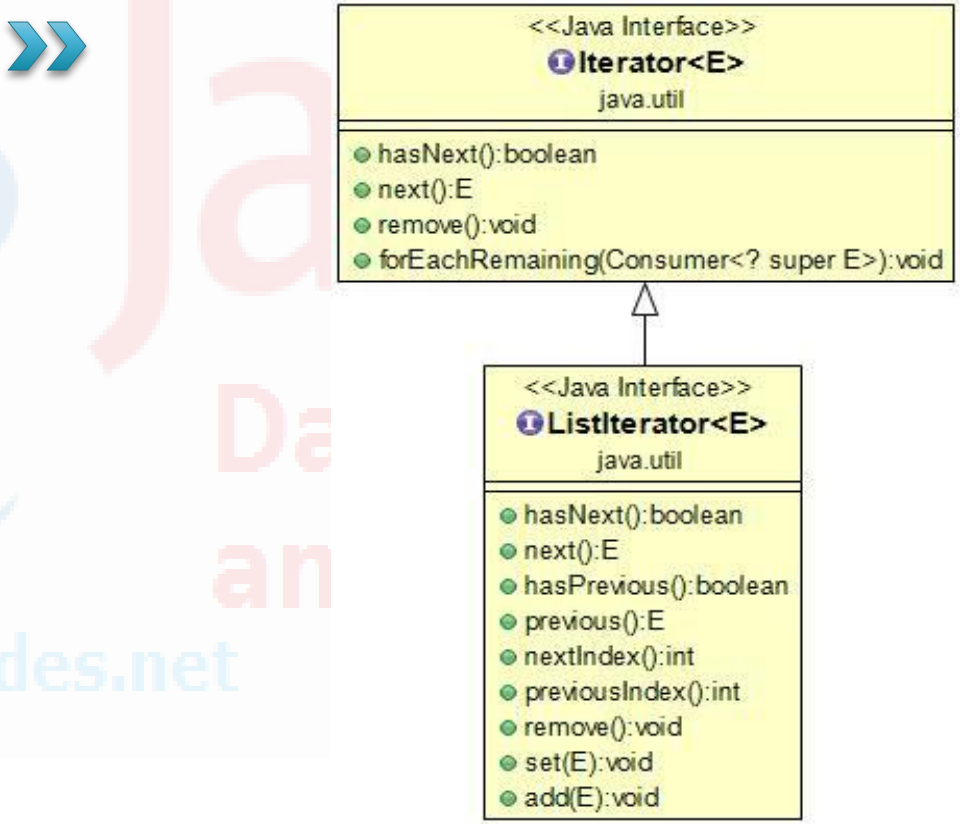
List


Iterable<E>
Iterator<E>
iterator()
AbstractCollection
Collection<E>
AbstractList
List<E>
Queue<E>
Set<E>
Deque<E>
SortedSet<E>
NavigableSet<E>
ArrayList
Vector
AbstractSequentialList
(non-synchronized)
(synchronized)
LinkedList
Stack
Interfaces
List Implementation

What is List?

- List is sequential data structure
- Lists typically allow duplicate elements
- It easily add or remove element by position
- Common types of List in Java collection framework:
  - ListIterator – interface
  - ArrayList – class
  - LinkedList – class
  - Stack – class
  - Vector– class

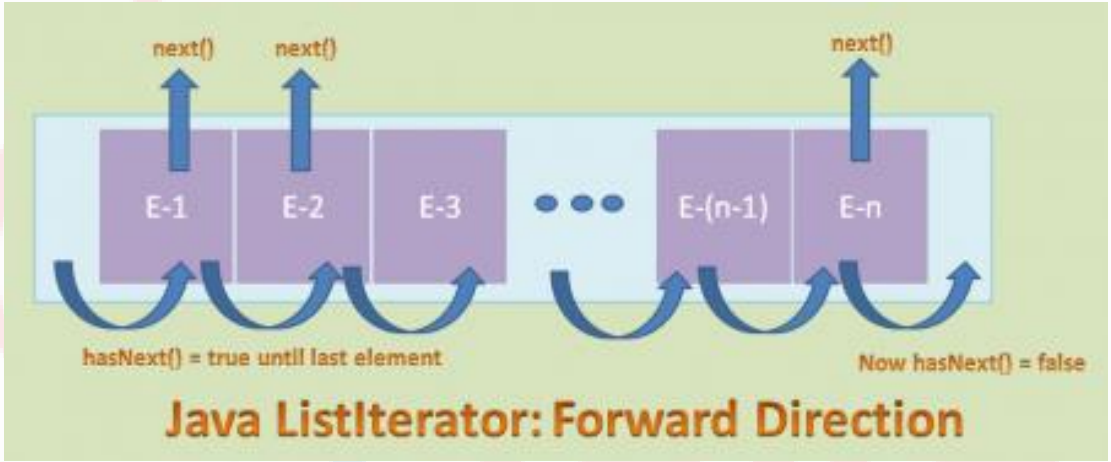
ListIterator

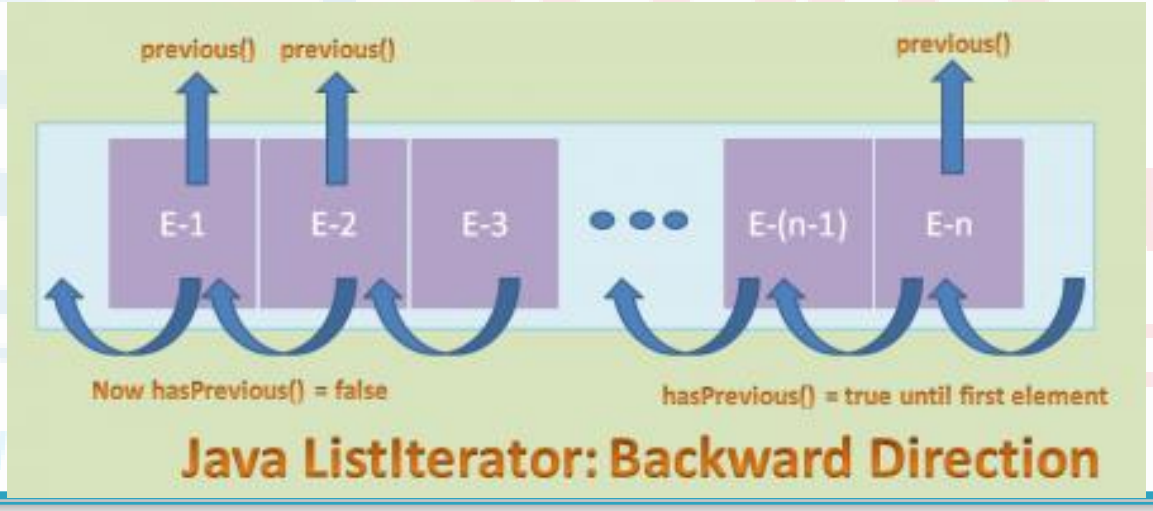


ListIterator

- **ListIterator** works in both directions (an Interface in Java)
- Forward Direction Iteration methods:
  - hasNext()
  - next()
  - nextIndex()
- Backward Direction Iteration methods
  - hasPrevious()
  - previous()
  - previousIndex()

ListIterator (cont.)





Methods

- void **add**(**E** e): Inserts the specified element into the list (optional operation).
- boolean **hasNext**(): Returns true if this list iterator has more elements when traversing the list in the forward direction.
- boolean **hasPrevious**(): Returns true if this list iterator has more elements when traversing the list in the reverse direction.
- **E next**(): Returns the next element in the list and advances the cursor position.
- int **nextIndex**(): Returns the index of the element that would be returned by a subsequent call to **next()**.

Methods (cont.)

- **E previous**(): Returns the previous element in the list and moves the cursor position backwards.
- int **previousIndex**(): Returns the index of the element that would be returned by a subsequent call to **previous()**.
- void **remove**(): Removes from the list the last element that was returned by **next()** or **previous()** (optional operation).
- void **set**(**E** e): Replaces the last element returned by **next()** or **previous()** with the specified element (optional operation).

ListIterator usage

```
Integer[] array = {1,3,5,7,9};
ListIterator< Integer > li =
   Arrays.asList(array).listIterator();
```

- li.hasNext() ; // return true
- li.next() ; // return 1
- li.next(); // return 3
- li.previous(); //return 3
- li.previous(); //return 1

Notes

```
// compute the sum of elements in list iterator
public static int sum(ListIterator<Integer> iter) {
    int result = 0;
    while (iter.hasNext()) {
        result += (Integer) iter.next();
    }
    return result;
}

public static void main(String[] args) {
    Integer[] array = { 1, 3, 5, 7, 9 };
    ListIterator<Integer> li = Arrays.asList(array).listIterator();
    int sum = sum(li);//???
    System.out.println(sum);
    int sum2 = sum(li);//???
    System.out.println(sum2);
}
```

25

0

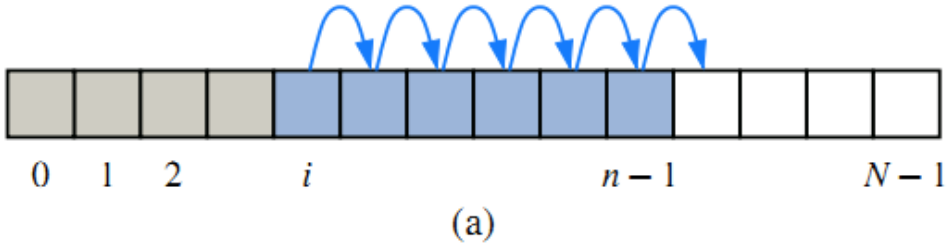
ListIterator is mutable!

ArrayList

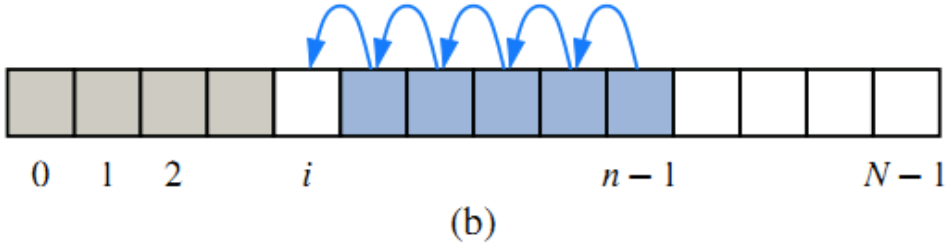
What is ArrayList?

- ArrayList: an array of elements
- Unlike the plain arrays the length of the array list can grow dynamically
- add and remove elements without knowing in advance how many elements we will have at the end

Insertion at index i



Deletion at index i



ArrayList implementation

- ArrayList: implemented with a plain array

```
1  public class ArrayList<E> implements List<E> {
2    // instance variables
3    public static final int CAPACITY=16;      // default array capacity
4    private E[ ] data;                        // generic array used for storage
5    private int size = 0;                     // current number of elements
6    // constructors
7    public ArrayList( ) { this(CAPACITY); }   // constructs list with default capacity
8    public ArrayList(int capacity) {          // constructs list with given capacity
9      data = (E[ ]) new Object[capacity];     // safe cast; compiler may give warning
10   }
```

Methods

- size( ):
  - Returns the number of elements in the list.
- isEmpty( ):
  - Returns a boolean indicating whether the list is empty.
- get(i):
  - Returns the element of the list having index i; an error condition occurs if i is not in range [0, size( )−1].
- set(i, e):
  - Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range [0, size( )−1].

Methods (cont.)

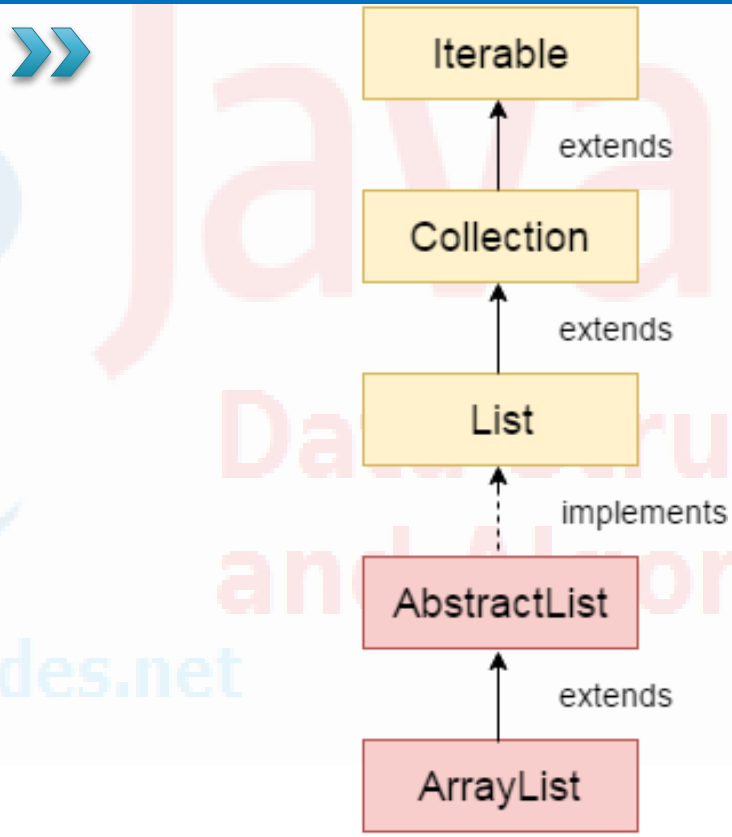
- add(i, e):
  - Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range [0, size( )].

- remove(i):
  - Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range [0, size( )−1].

Performance

Method	Running time
size()	O(1)
isEmpty()	O(1)
get(i)	O(1)
set(i, e)	O(1)
add(i,e)	O(n)
remove(i)	O(n)

ArrayList in Java Collective Framework



ArrayList

- Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.
- The important points about Java ArrayList class are:
  - Can contain duplicate elements.
  - Maintains insertion order.
  - Is non synchronized.
  - Allows random access because array works at the index basis.
  - Manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

ArrayList (cont.)

- In general, an **`ArrayList`** serves the same purpose as an array, except that an **`ArrayList`** can change length while the program is running
- Constructors:
  - ArrayList(): is used to build an empty array list.
  - ArrayList(Collection<? extends E> c): is used to build an array list that is initialized with the elements of the collection c.
  - ArrayList(int capacity): is used to build an array list that has the specified initial capacity.

ArrayList methods 1

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

ArrayList methods 2

`addAll(`**list**`)` `addAll(`**index**`, `**list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list (seen later)
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from**`, `**to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the
  type of elements it will contain between < and >.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

ArrayList of primitives?

- The type you specify when creating an `ArrayList` must be an object type; it cannot be a primitive type.

```
// illegal -- int cannot be a type parameter
ArrayList<int> list = new ArrayList<int>();
```

- But we can still use `ArrayList` with primitive types by using special classes called *wrapper* classes in their place.

```
// creates a list of ints
ArrayList<Integer> list = new ArrayList<Integer>();
```

Wrapper classes

Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- A wrapper is an object whose sole purpose is to hold a primitive value.

- Once you construct the list, use it with primitives as normal:

```
ArrayList<Double> grades = new ArrayList<Double>();
grades.add(3.2);
grades.add(2.7);
...
double myGrade = grades.get(0);
```

Normal for loop

```
public class NormalFor {
  public static void main(String[ ] args)   {
       ArrayList<Integer> list = new ArrayList<Integer>;
       list.add( 42 );
       list.add( 57 );
       list.add( 86 );

       // "for each Integer, i, in list"
       for( int i = 0; i <list.size(); i++)
          System.out.println( list.get(i));
  }
}
//-- Output ---
42
57
86
```

The "For Each" Loop

- The **`ArrayList`** class is an example of a *collection* class

- Starting with version 5.0, Java has added a new kind of for loop called a *for-each* or *enhanced for* loop
  - This kind of loop has been designed to cycle through all the elements in a collection (like an **`ArrayList`**)

“for-each” example

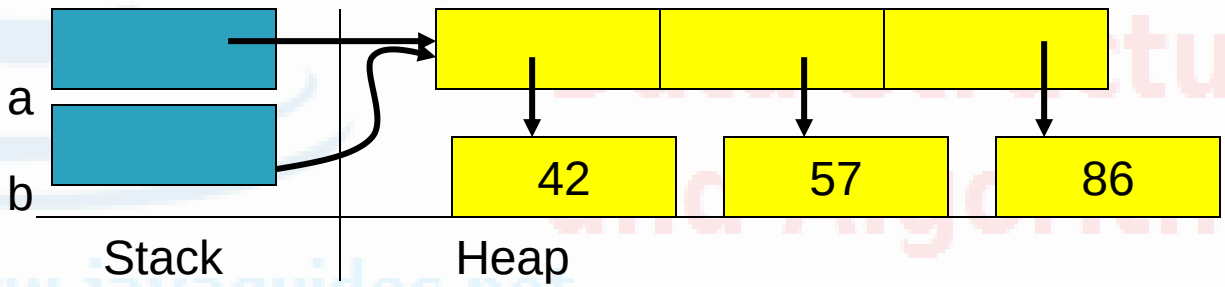
```
public class ForEach {
  public static void main(String[ ] args)   {
       ArrayList<Integer> list = new ArrayList<Integer>;
       list.add( 42 );
       list.add( 57 );
       list.add( 86 );

       // “for each Integer, i, in list”
       for( Integer i : list )
          System.out.println( i );
  }
}
//-- Output ---
42
57
86
```

Copying an ArrayList

```
// create an ArrayList of Integers
ArrayList<Integer> a = new ArrayList<Integer>( );
a.add(42); a.add(57); a.add(86);
```

- Assignment doesn't work
  - As we've seen with any object, using assignment just makes two variables refer to the same ArrayList.

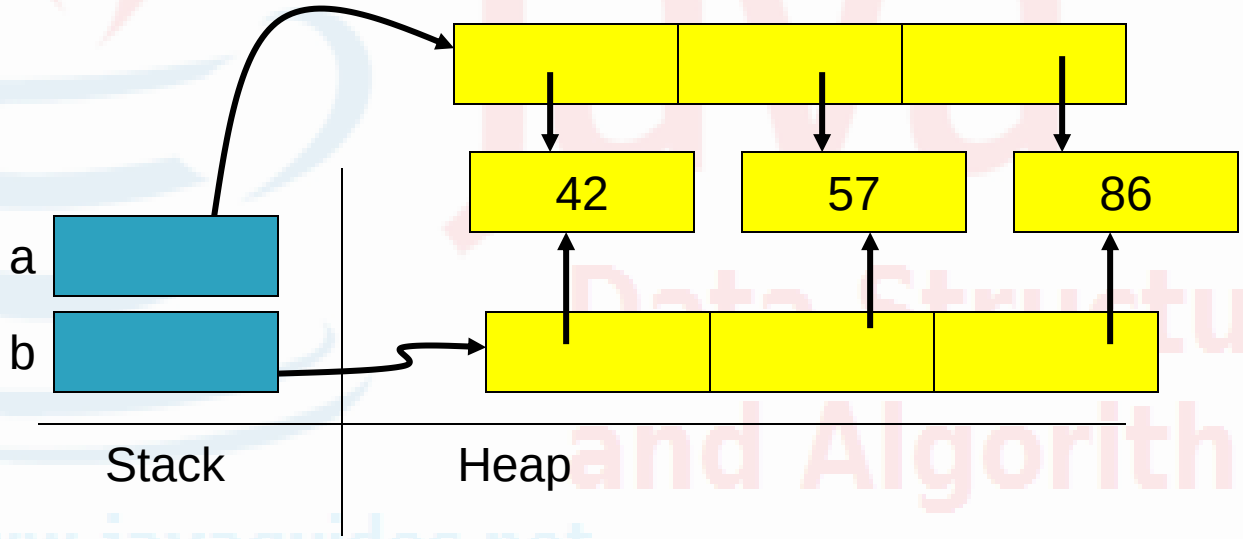
```
ArrayList<Integer> b = a;
```



Copying an ArrayList

ArrayList's clone( ) method makes a shallow copy

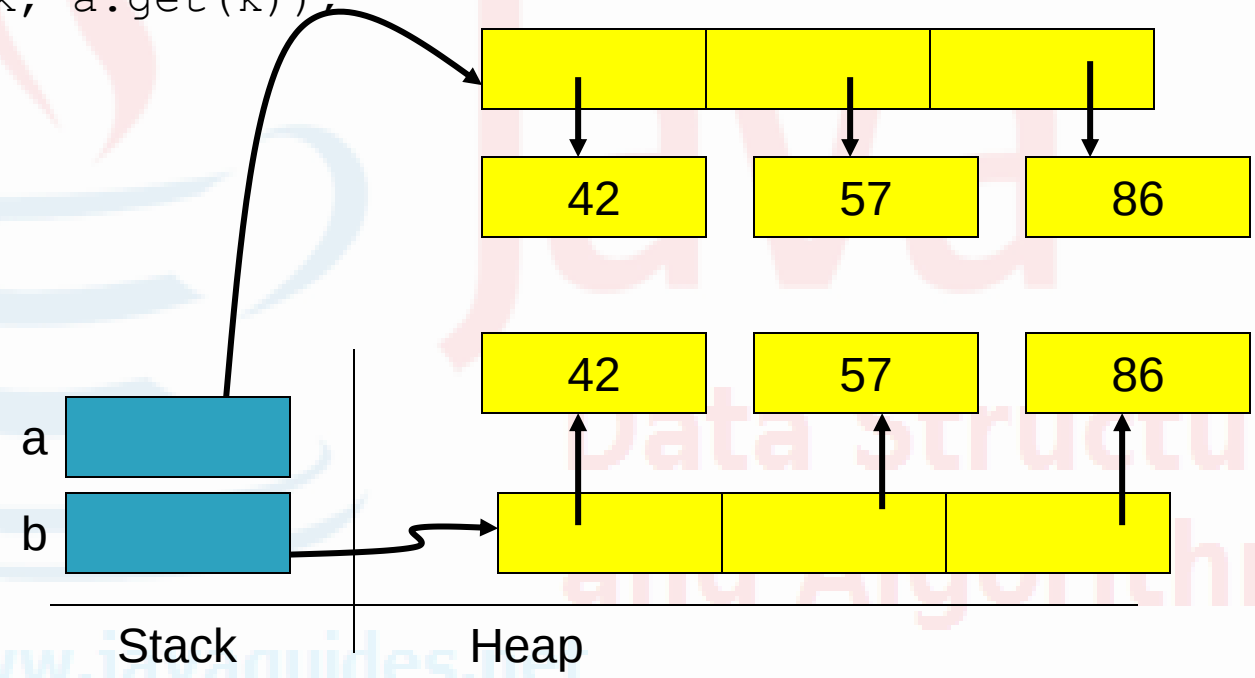
```
ArrayList<Integer> b = a.clone( );
```



Copying an ArrayList

We need to manually make a deep copy

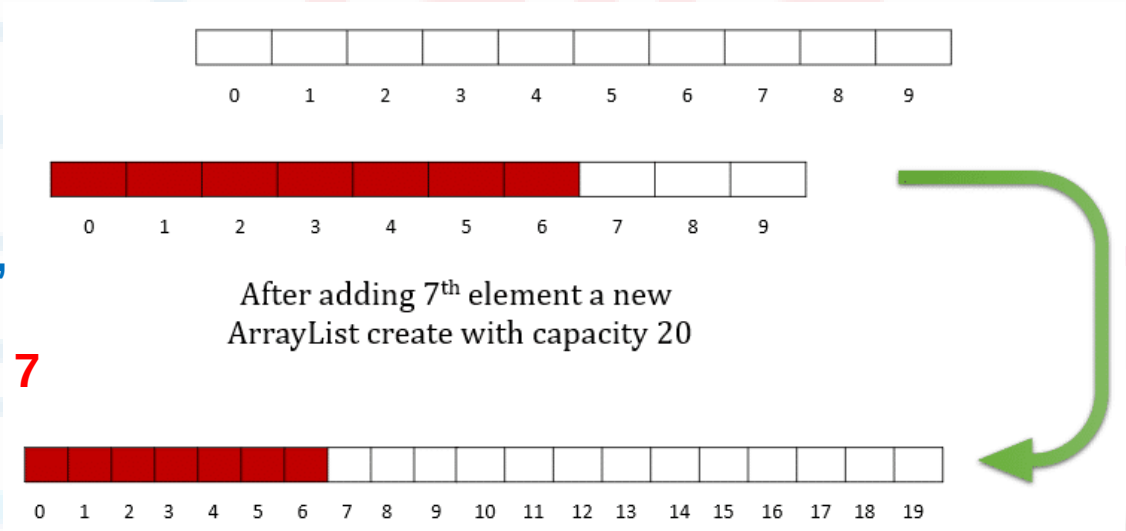
```
ArrayList<Integer> b = a.clone( );

for( int k = 0; k < b.size( ); k++)
  b.set(k, a.get(k));
```



Load factor?

- The size of ArrayList grows automatically which is fully based on **load factor** and **current capacity**.
- The load factor is the measure that decides when to increase the capacity of the ArrayList.
- The default load factor of an ArrayList is **0.75f**.

ArrayList: 10 elements,
LoadFactor=0.75f
Threshold = 10 * 0.75 = 7



When using ArrrayList?

- For most programming tasks, array lists are easier to use than arrays
  - Array lists can grow and shrink.
  - Arrays have a nicer syntax.
- Recommendations
  - If the size of a collection never changes, use an array.
  - If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
  - Otherwise, use an array list.

Arrays or ArrayList?

Table 3 Comparing Array and Array List Operations

Operation	Arrays	Array Lists
Get an element.	`x = values[4];`	`x = values.get(4);`
Replace an element.	`values[4] = 35;`	`values.set(4, 35);`
Number of elements.	`values.length`	`values.size()`
Number of filled elements.	`currentSize` (companion variable, see Section 7.1.4)	`values.size()`
Remove an element.	See Section 7.3.6.	`values.remove(4);`
Add an element, growing the collection.	See Section 7.3.7.	`values.add(35);`
Initializing a collection.	`int[] values = { 1, 4, 9 };`	No initializer list syntax; call `add` three times.

Vector

Vector is a synchronized dynamically growable array with efficient access by index

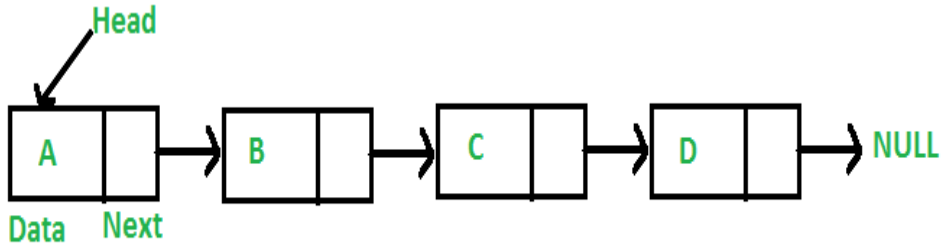
Example:

initialCapacity is optional

```
Vector<Integer> vec =
    new Vector<Integer>(10/*initialCapacity*/);
vec.add(7);
```

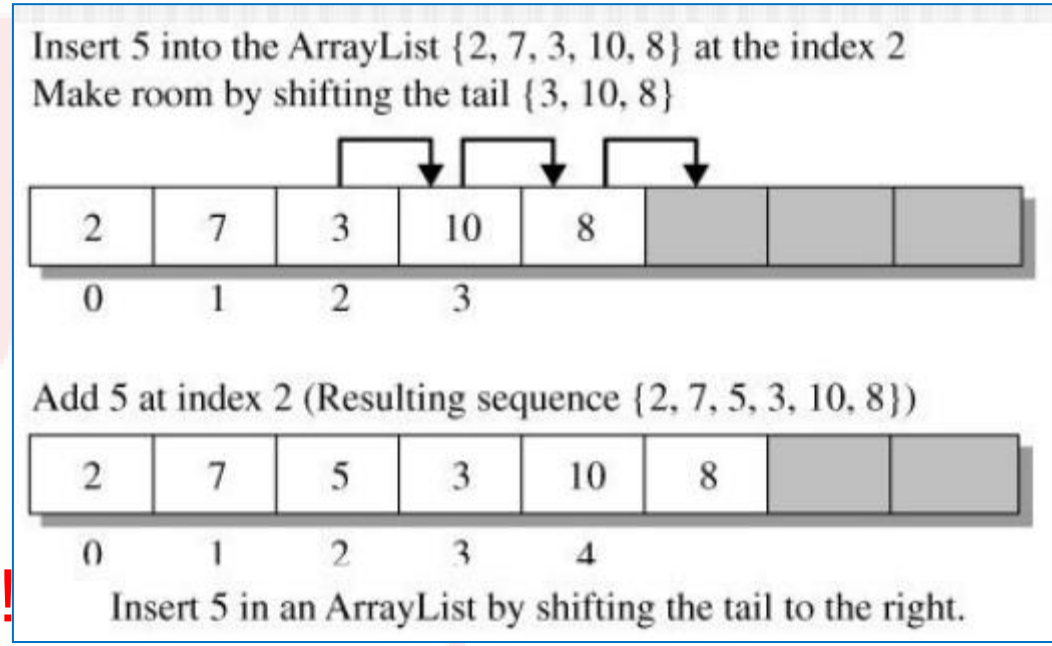
Vector is an old (Java 1.0) container and is less in use today, replaced mainly by ArrayList (Java 1.2) which is not synchronized

Linked List



Insertion/Deletion in ArrayLists

- To insert or delete an element at an interior location in an ArrayList:
  - requires shifting of data
  - is an O(n) operation.



➔Time comsuming!!!

LinkedList

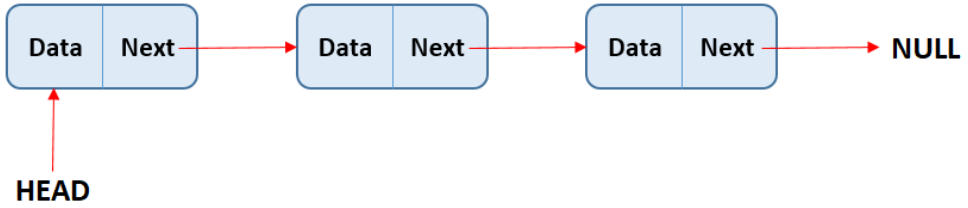
- We need an alternative structure that stores elements in a sequence but allows for more efficient insertion and deletion of elements at random positions in the list.

- In a linked list, elements contain links that reference the **previous** and the **successor elements** in the list.

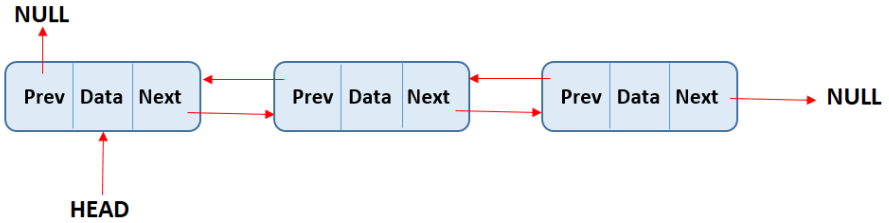
- Inserting and deleting an element is a local operation and requires updating only the links adjacent to the element. The other elements in the list are not affected

Types of LinkedList

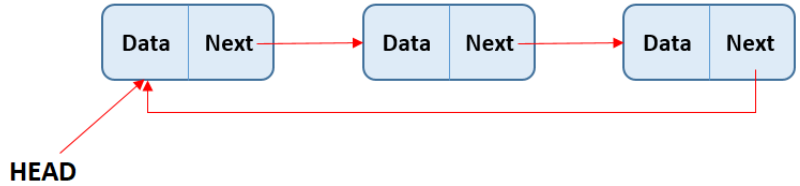
- Singly **Linked List**



- Doubly **Linked List**



- Circularly **Linked List**



Singly Linked List

- Elements in a linked list are nodes. These are Node objects that have two instance variables.
  - The first variable, Data, is of generic type T.
  - The second variable is a Node reference called next that provides a link to the next node.

Singly Linked List Implementation

```
public class Node<E> {
    // data held by the node
    public E element;
    // next node in the list
    public Node<E> next;

    // default constructor with no initial value
    public Node() {
        element = null;
        next = null;
    }

    // initialize element to e and set next to null
    public Node(E e) {
        element = e;
        next = null;
    }
}
```

Singly Linked List Implementation

- Head: the node of the list
- Tail: the last node of the list

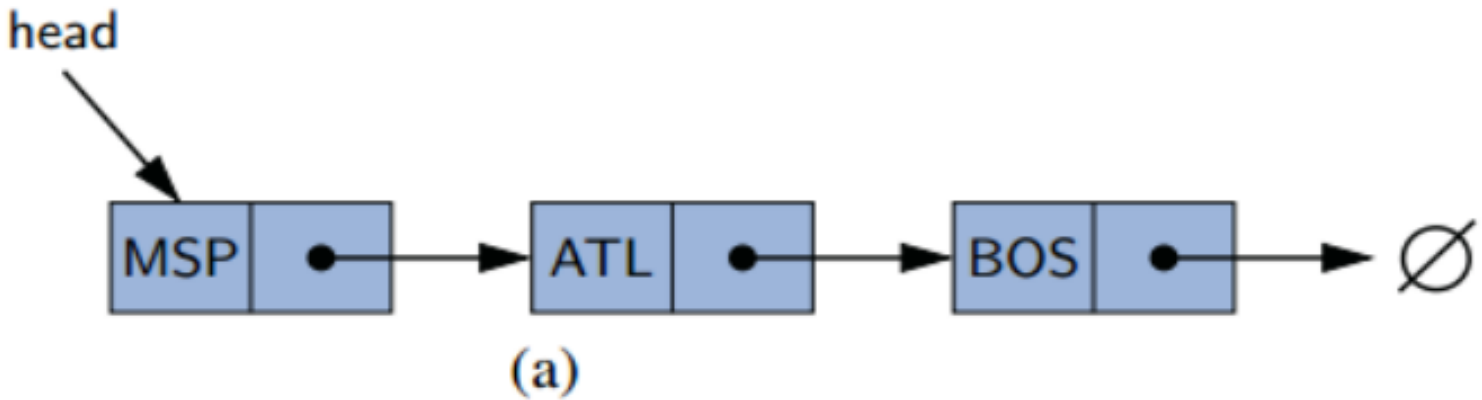
```
public class SinglyLinkedList<E> {
    private Node<E> head = null; // head node of the list (or null if empty)
    private Node<E> tail = null; // last node of the list (or null if empty)
    private int size = 0; // number of nodes in the list

    public SinglyLinkedList( ) { }
}
```

Implementing a Singly Linked List Class

- Some basic methods:

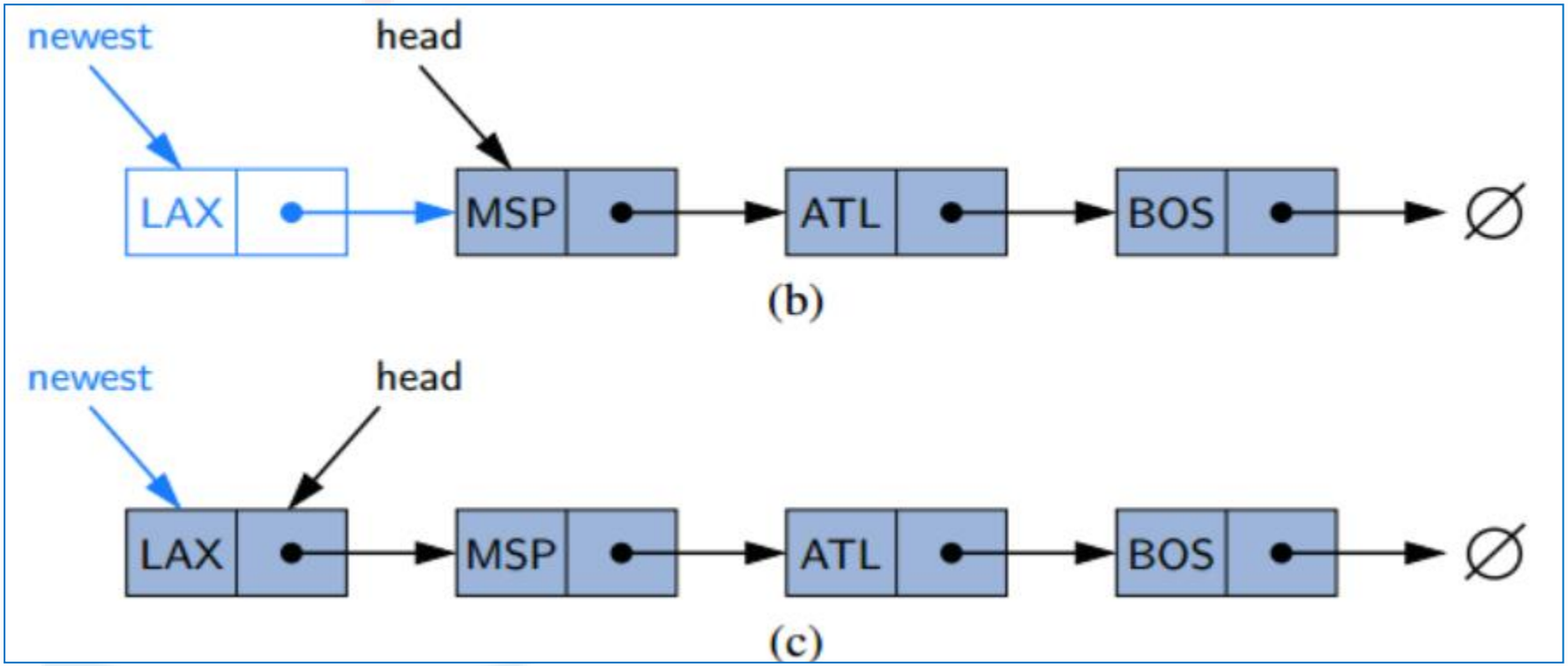
size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns **true** if the list is empty, and **false** otherwise.
first( ):	Returns (but does not remove) the first element in the list.
last( ):	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element 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.

Method: addFirst



(a)

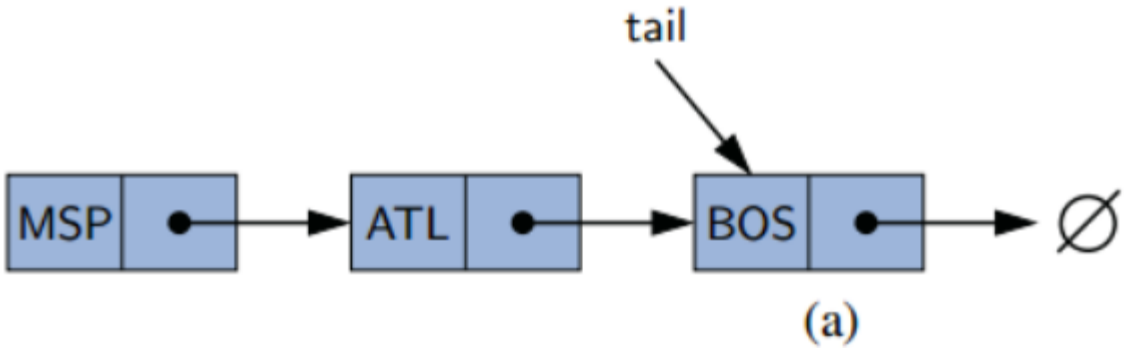
Method: addFirst (cont.)



Method: addFirst (cont.)

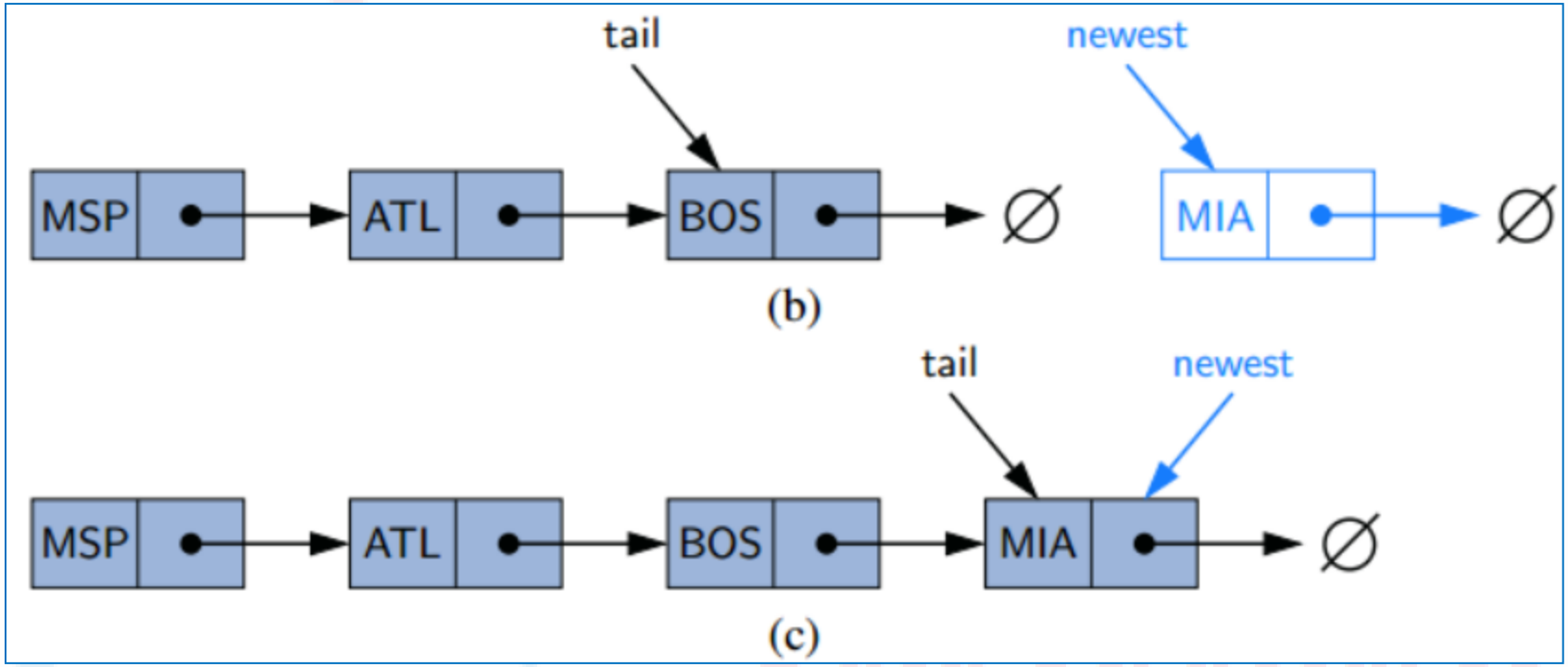
- **Implementation**:
- public void addFirst(E e) {
  - newest=Node(e) {create new node instance storing reference to elemente}
  - newest.next=head {set new node's next to reference the old head node}
  - head=newest {set variableheadto reference the new node}
  - size=size+1 {increment the node count}

Method: addLast



(a)

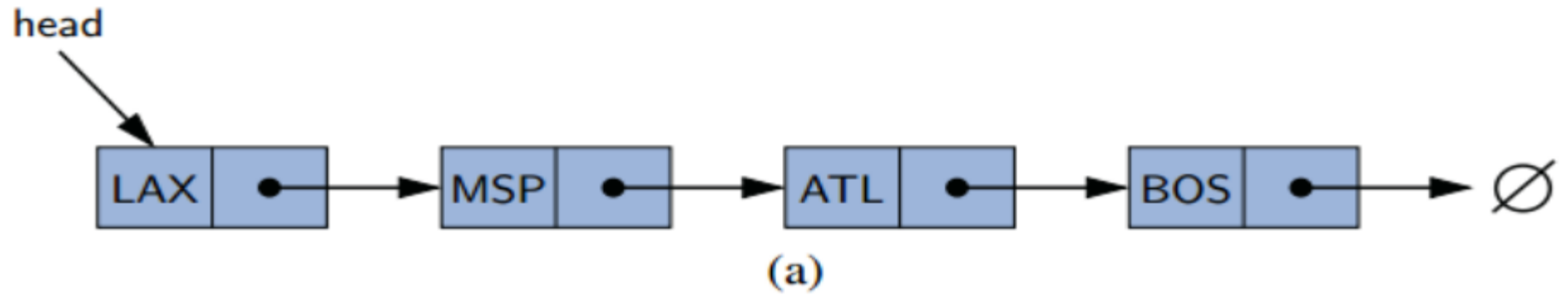
Method: addLast (cont.)



Method: addLast (cont.)

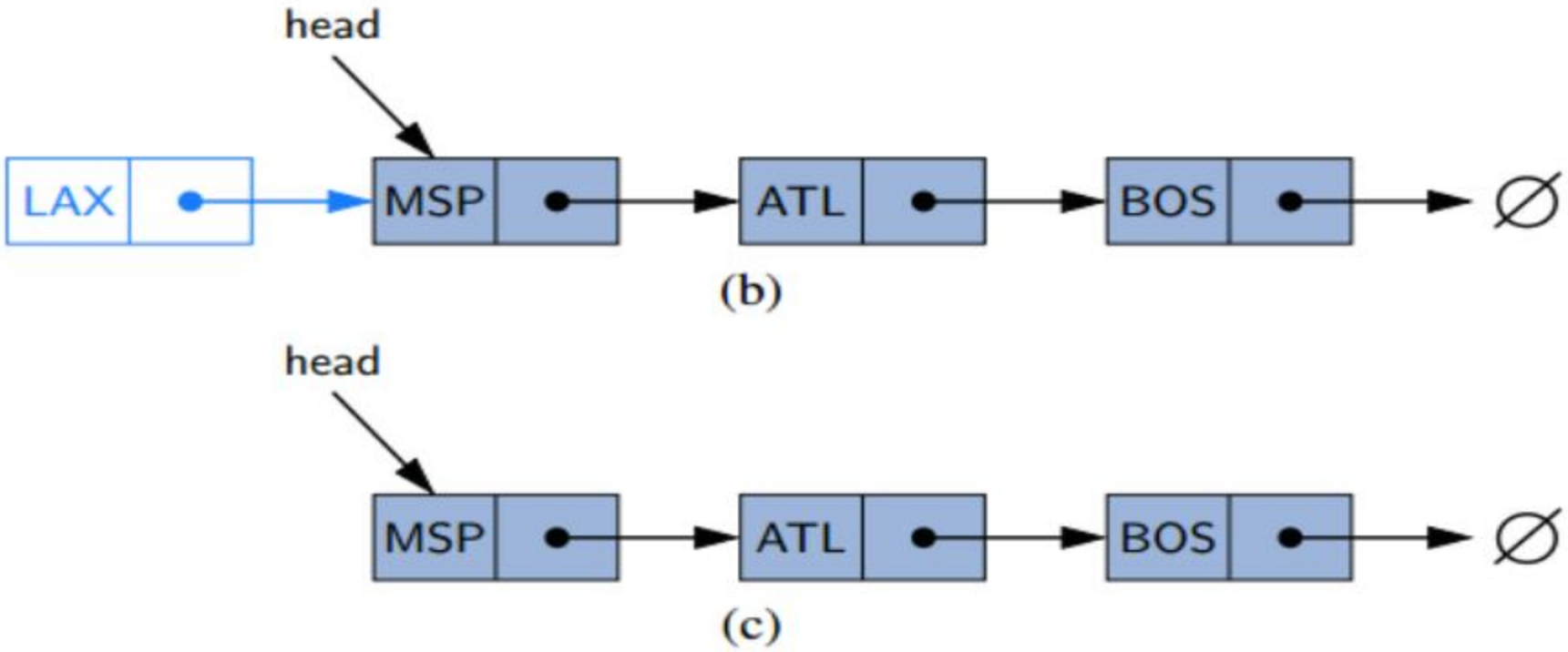
- Implementation:
- public void addLast(E e) {
  - newest=Node(e) {create new node instance storing reference to elemente}
  - newest.next=null {set new node's next to reference thenullobject}
  - tail.next=newest {make old tail node point to new node}
  - tail=newest {set variabletailto reference the new node}
  - size=size+1 {increment the node count}

Method: removeFirst



(a)

Method: removeFirst (cont.)

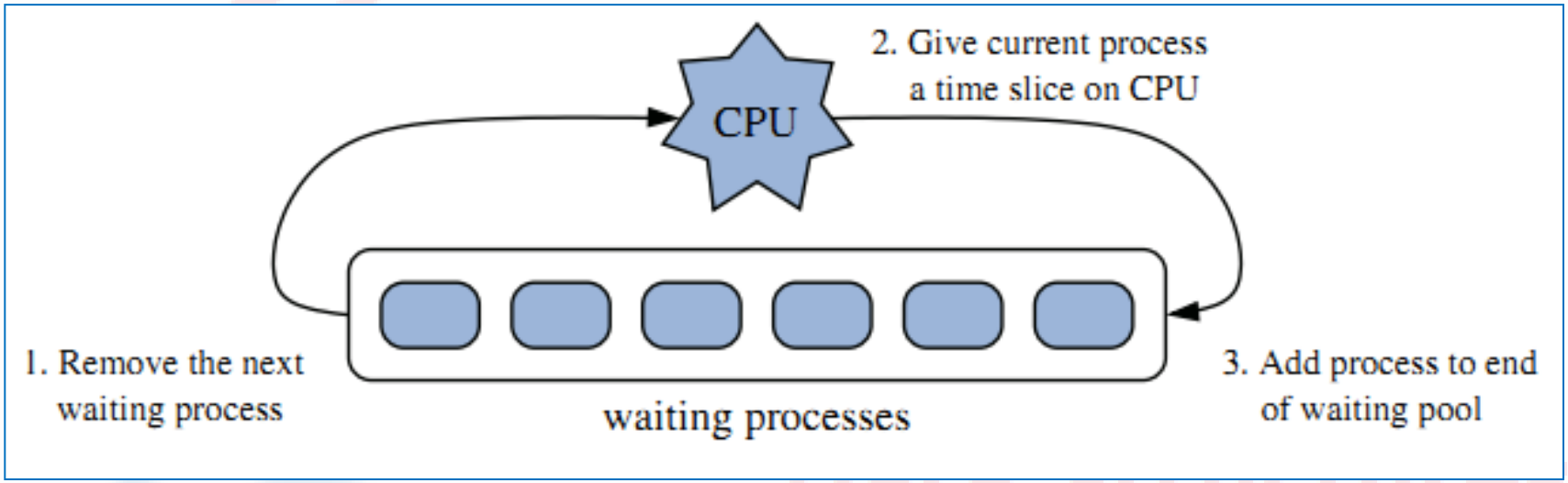


How to implement toString?

Circularly Linked List

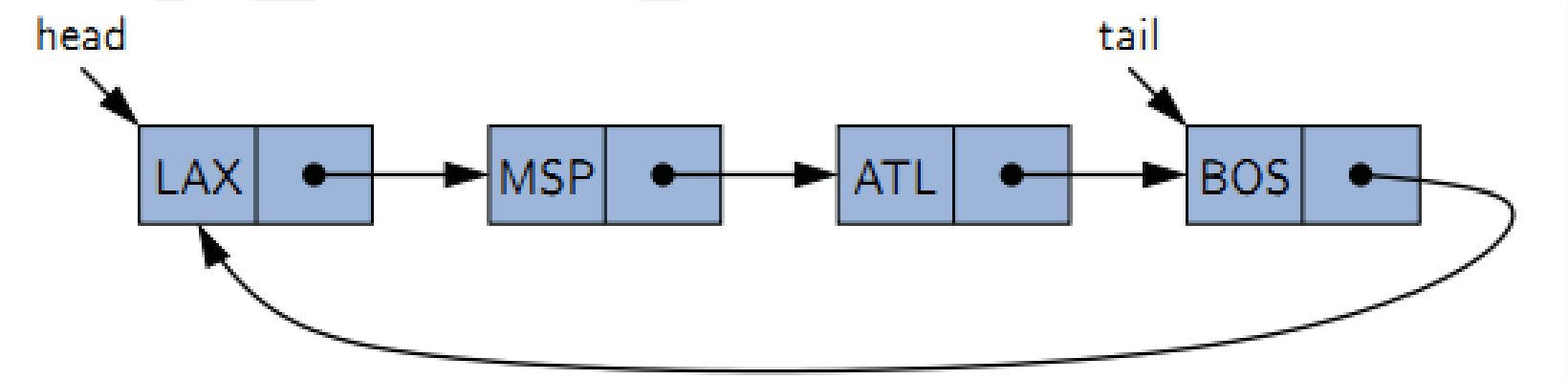
Circularly Linked List

- The three iterative steps for round-robin scheduling:



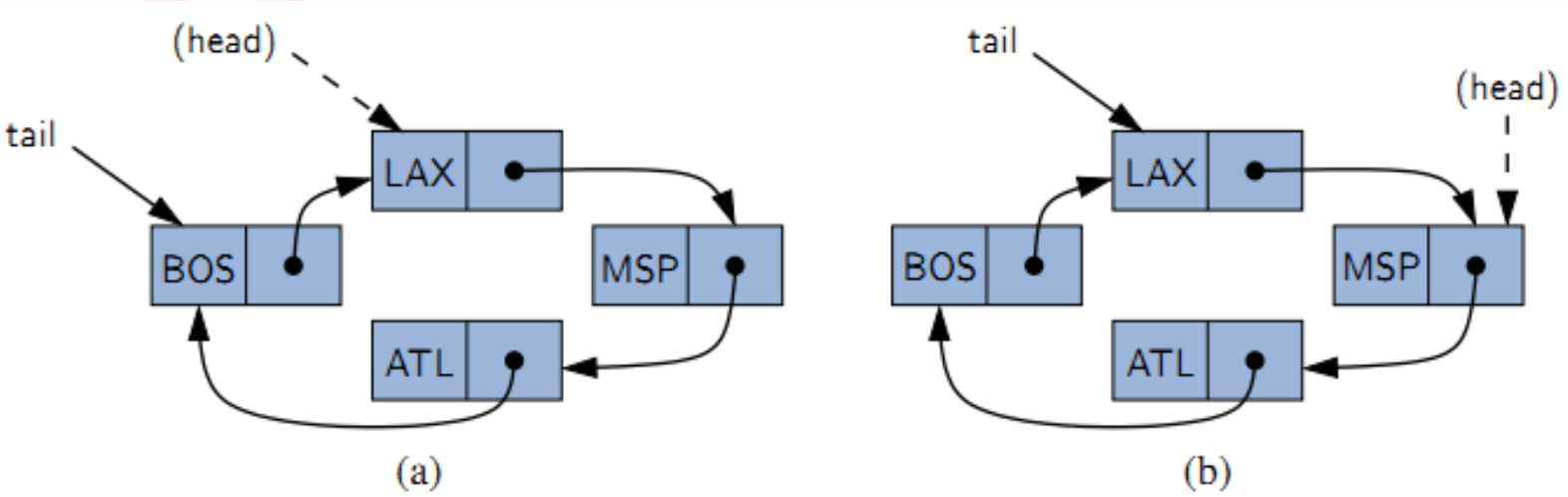
Designing and Implementing a Circularly Linked List

- Example of a singly linked list with circular structure



Circularly Linked List

- The rotation operation on a circularly linked list:



Doubly Linked List

Doubly Linked List

- Doubly-linked list nodes contain two references that point to the next and previous node.
- Each node:
  - element
  - link to the previous node
  - link to the next node

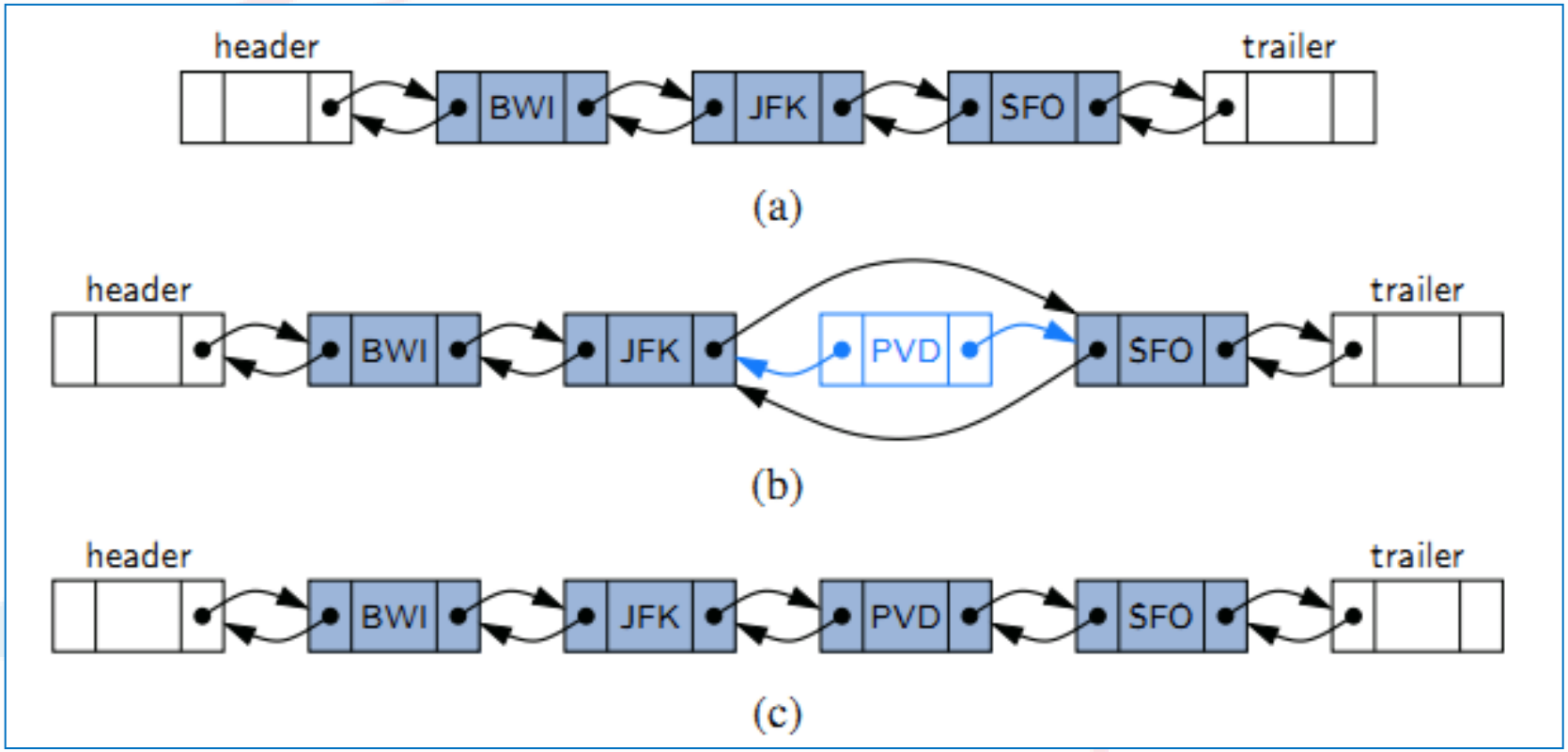
Doubly Linked List (cont.)

- Scanning a doubly-linked list in both directions.
  - The forward scan starts at front and ends when the link is a reference to back.
  - In the backward direction, reverse the process and the references.

Implementing a Doubly Linked List Class

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.

Inserting with a Doubly Linked List



Inserting with a Doubly Linked List

- Algorithm insertAfter(p,e):

  Create a new node v

  v.setElement(e)

  v.setPrev(p) //link v to its predecessor

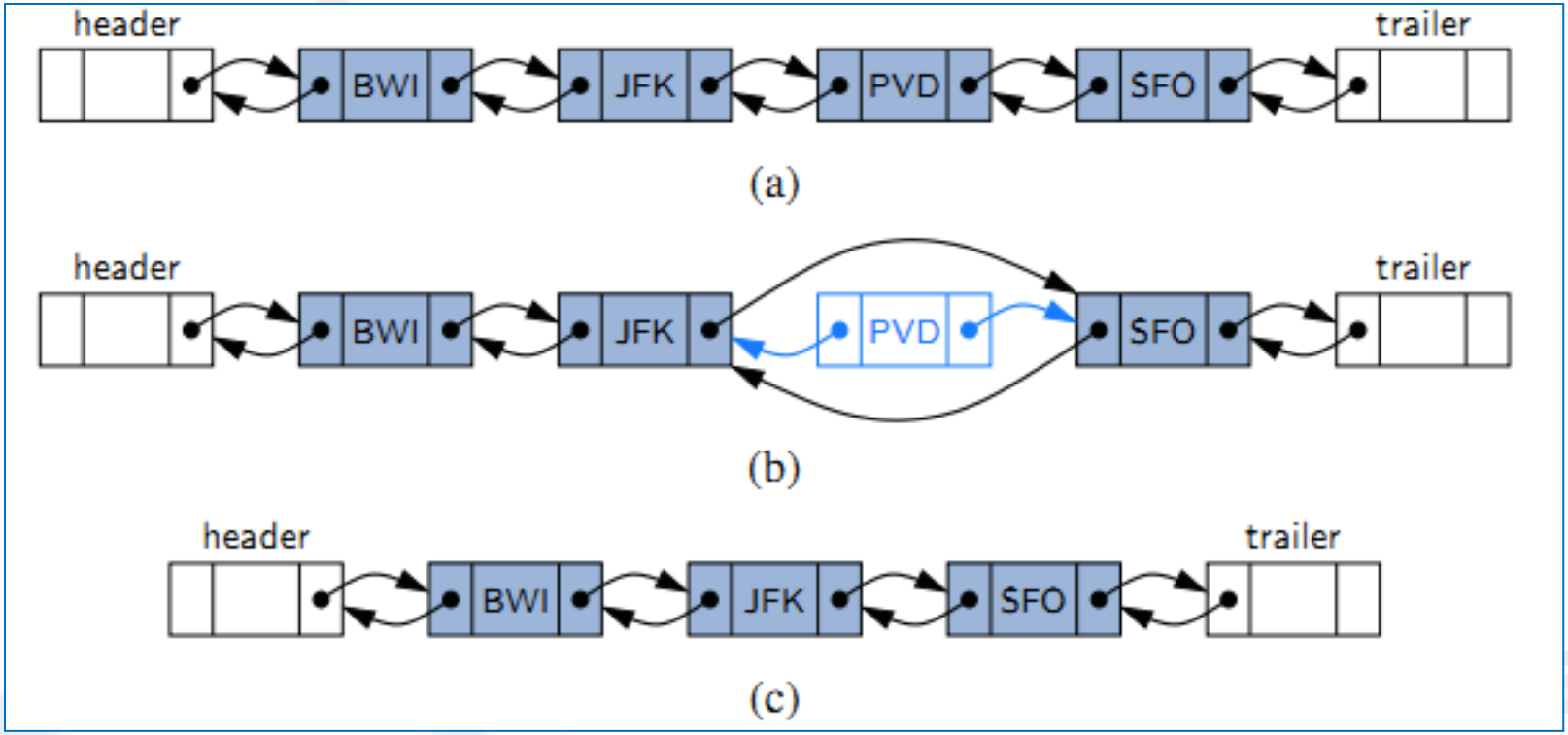
  v.setNext(p.getNext())//link v to its successor

  (p.getNext()).setPrev(v)//link p's old successor to v

  p.setNext(v)//link p to its new successor, v

  return v //the position for the element e

Deleting with a Doubly Linked List

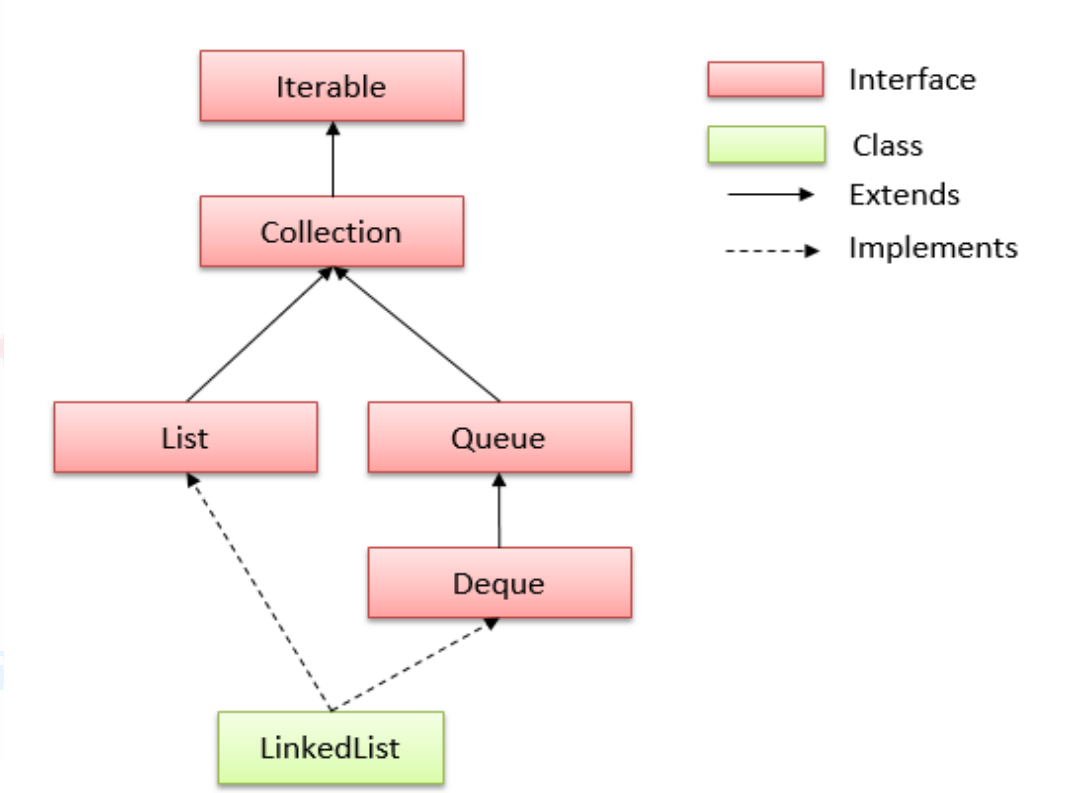


Deleting with a Doubly Linked List

- Algorithm remove(p):

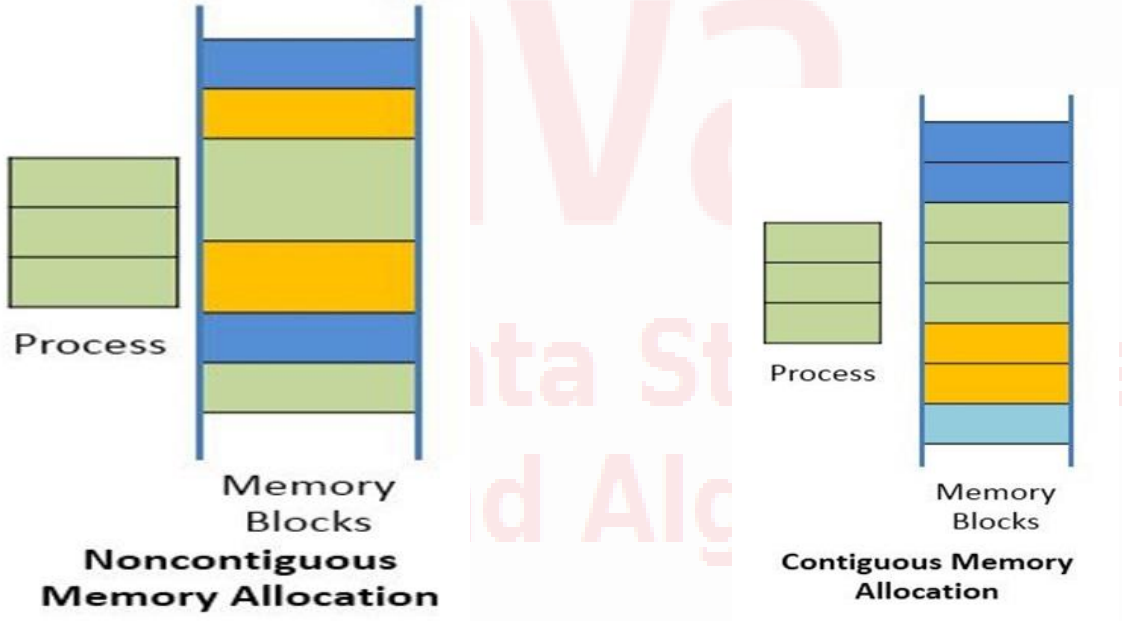
```
t = p.element
//a temporary variable to hold the return value
(p.getPrev()).setNext(p.getNext())
//linking out p
(p.getNext()).setPrev(p.getPrev())
p.setPrev(null)
//invalidating the position p
p.setNext(null)
return t
```

Linked List in Java Collective Framework



LinkedList in Java Collection Framework

- ArrayList uses an array in contiguous memory.
- A LinkedList uses a doubly-linked list whose elements reside in noncontiguous memory locations.



LinkedList: Constructors

- LinkedList():
  - constructs an empty linked list.

- LinkedList(Collection<? extends E> elements):
  - constructs a linked list and adds all elements from a collection.

LinkedList: Selected Methods

- void addFirst(E element):
  - add an element to the beginning of the list.
- void addLast(E element):
  - add an element to the end of the list.
- E getFirst():
  - return the element at the beginning of the list
- E getLast():
  - return the element at the end of the list.
- ...

FACULTY OF INFORMATION TECHNOLOGY

Java
Data Structures