CS 91 USACO

Bronze Division

Unit 2: 1-D Data Structures

LECTURE 6: LINKED LISTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

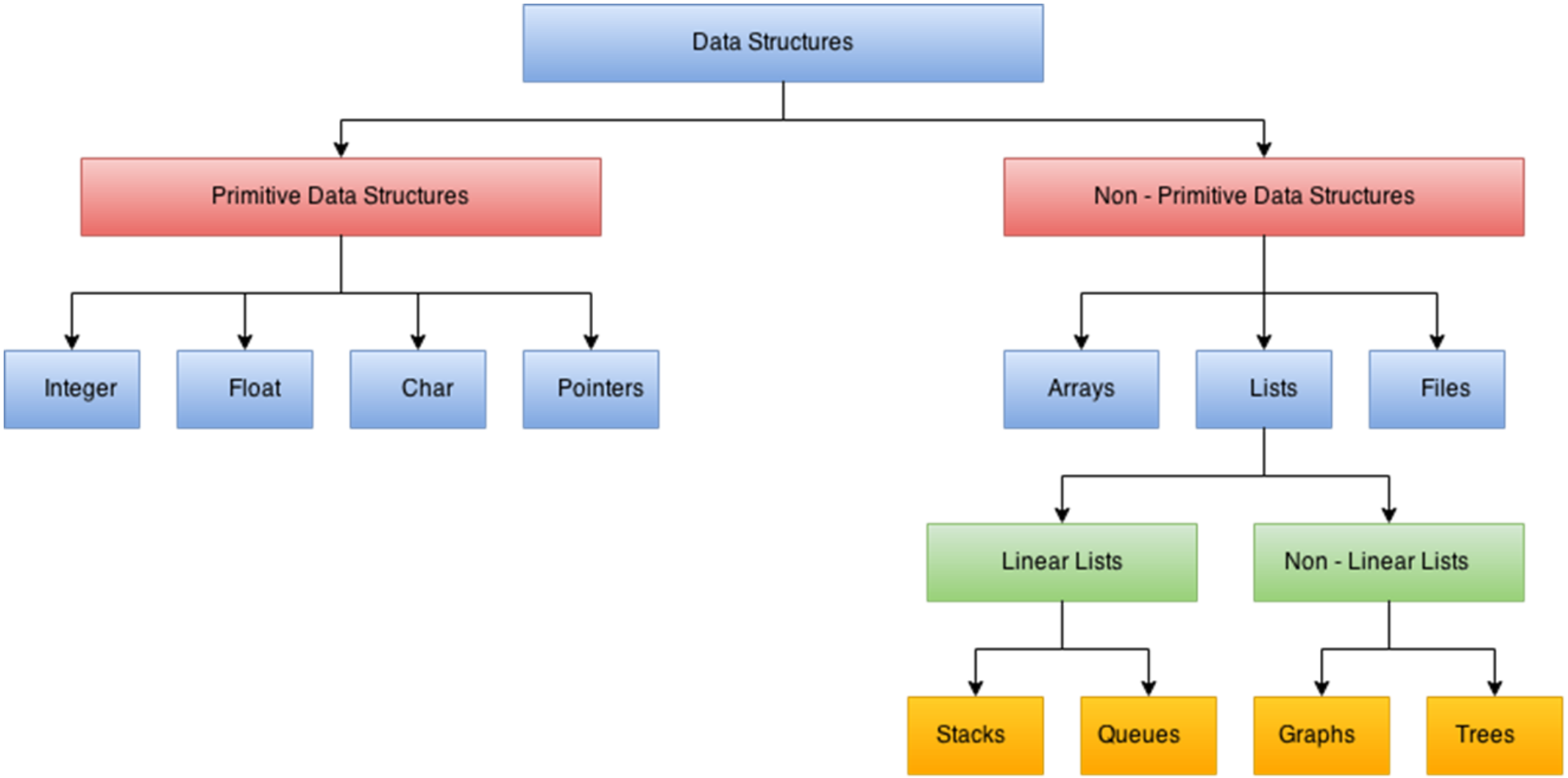
- Overview of Java Data Collection
- Linked lists
  - Abstract data type (ADT)
- Basic operations of linked lists
  - Insert, find, delete, print, etc.

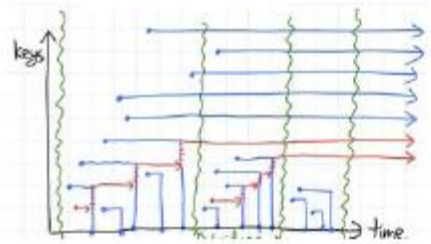
Objectives

- Variations of linked lists
- Applications of linked list
- Equivalence Tests
- Cloning Lists

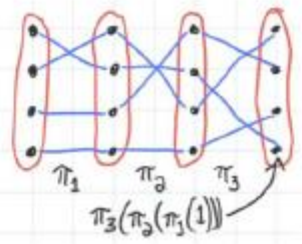
Java Collections

SECTION 1

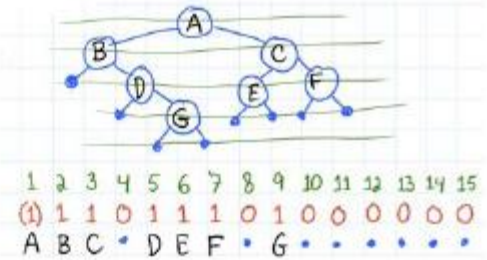

Data Structures
Primitive Data Structures
Non - Primitive Data Structures
Integer
Float
Char
Pointers
Arrays
Lists
Files
Linear Lists
Non - Linear Lists
Stacks
Queues
Graphs
Trees



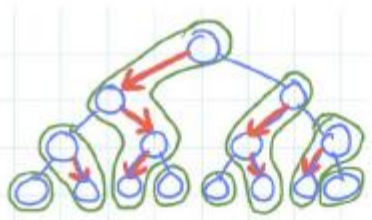
TIME TRAVEL



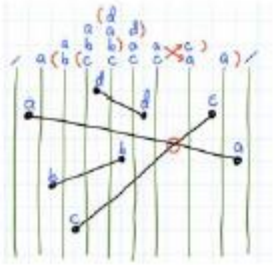
DYNAMIC GRAPHS



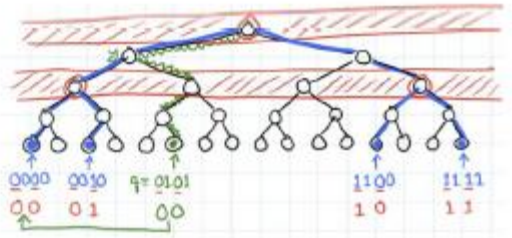
SUCCINCT



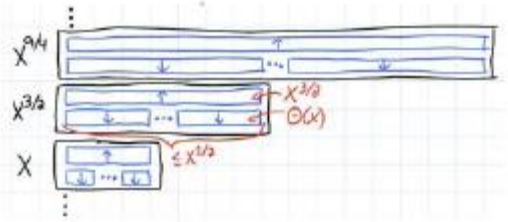
DYNAMIC OPTIMALITY



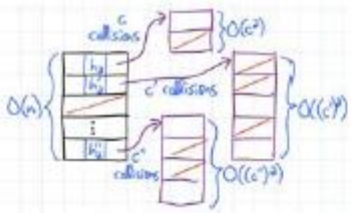
GEOMETRY



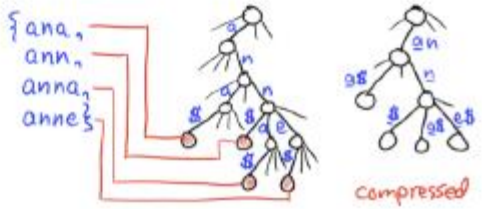
INTEGERS



MEMORY HIERARCHY



HASHING



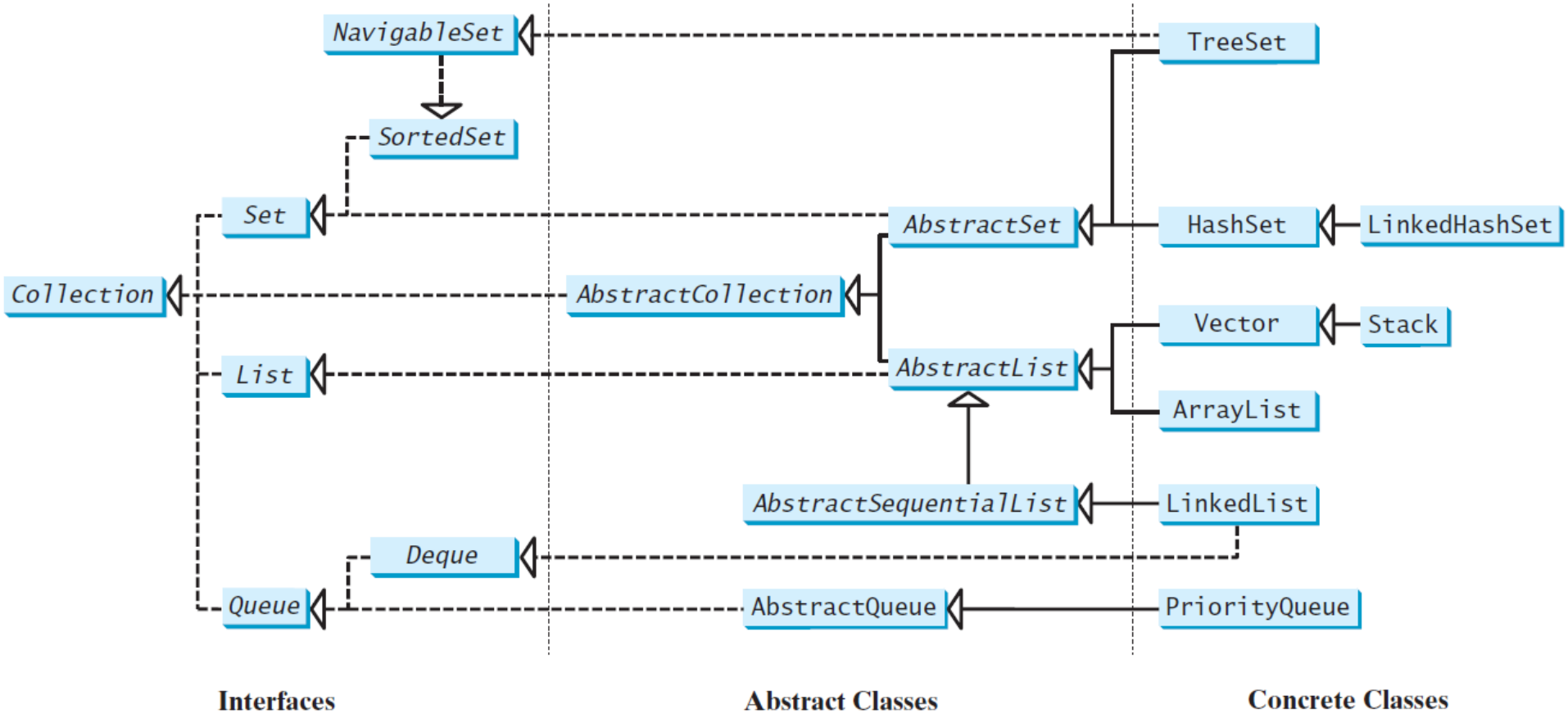
STRINGS

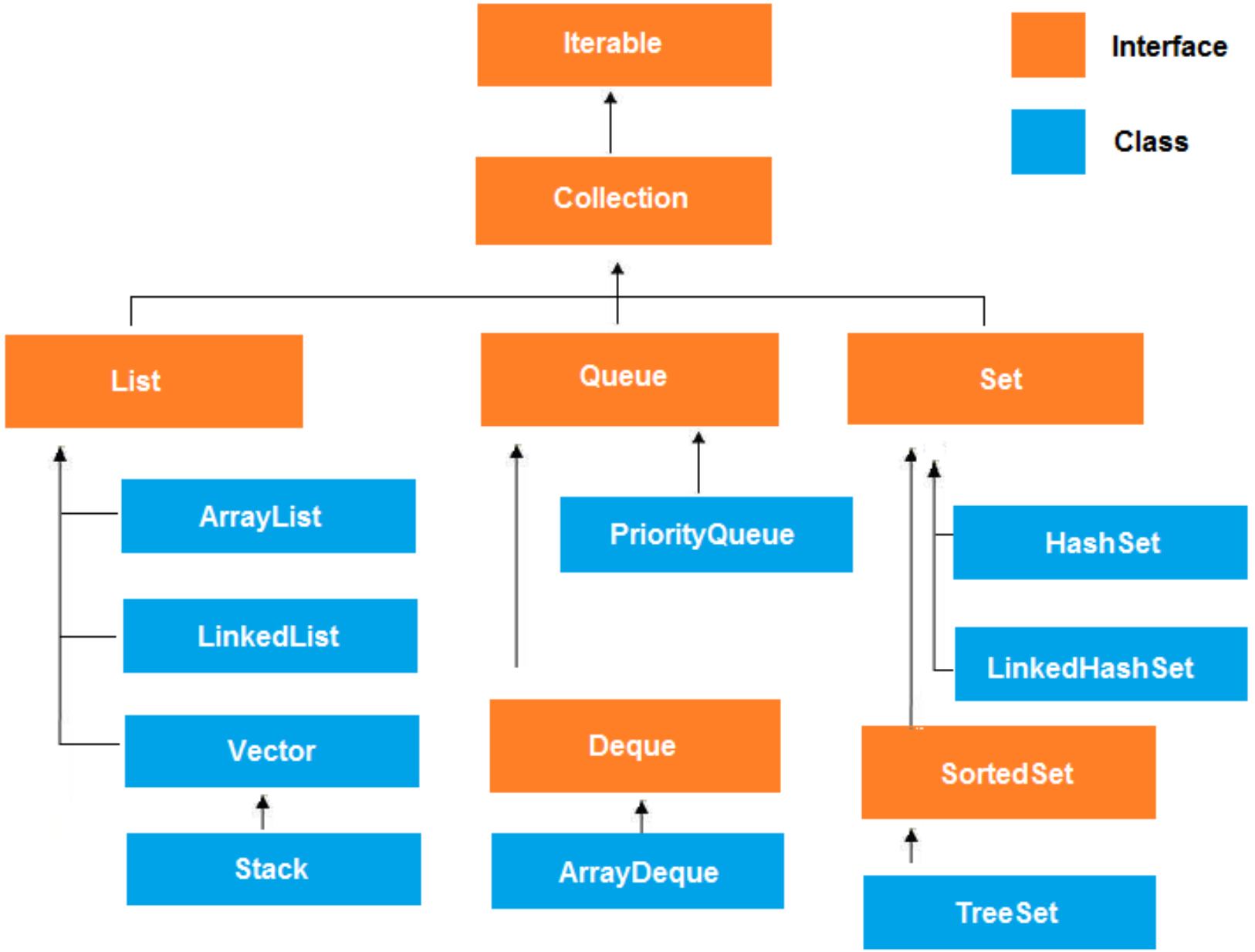
Objectives

- Study Java Collections Framework in 3 Chapters:
  - List and Iterables
  - Stack and Queues
  - Sets and Maps
- In this Chapter, we will focus on Arrays, ArrayList, LinkedList and Vector classes and their applications

Collection Framework

Data Structure in Java API




Iterable
Collection
List
Queue
Set
ArrayList
LinkedList
Vector
Stack
PriorityQueue
Deque
ArrayDeque
HashSet
LinkedHashSet
SortedSet
TreeSet
Interface
Class

Demonstration Program

TESTCOLLECTIONS.JAVA

Use of Collection Utility Method Methods

Collections.**nCopies**(3, "red");
Collections.**fill**(arrayList, "yellow");
Collections.**shuffle**(arrayList);
Collections.**min**(arrayList);
Collections.**max**(arrayList);
Collections.**sort**(arrayList);
Collections.**binarySearch**(arrayList, "gray");
List syncList = Collections.**synchronizedList**(arrayList);
List unmodifiableList = Collections.**unmodifiableList**(syncList);

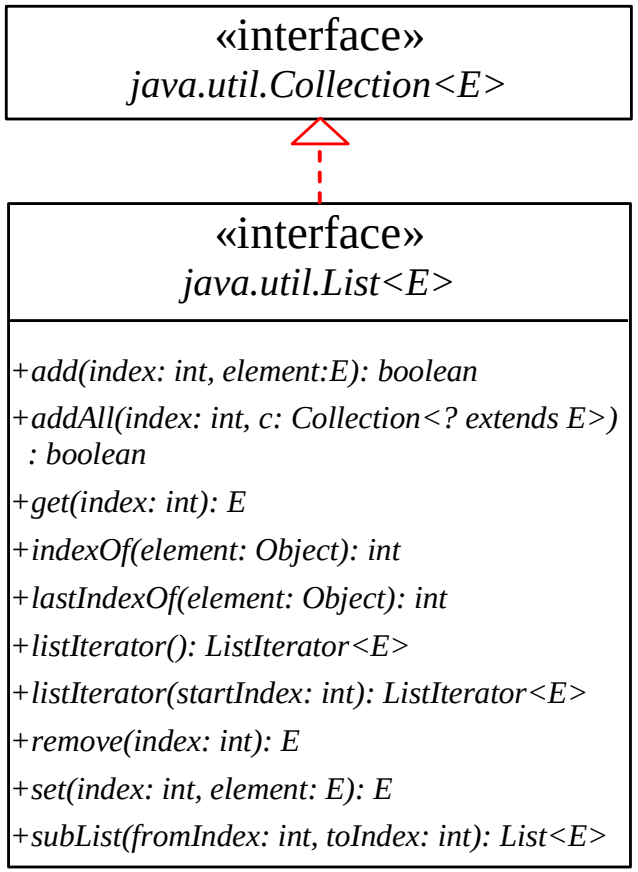
List Interface

SECTION 2

List Interface

List
+add(element : Object) : boolean +add(index : int, element : Object) : void +addAll(collection : Collection) : boolean +addAll(index : int, collection : Collection) : boolean +clear() : void +contains(element : Object) : boolean +containsAll(collection : Collection) : boolean +equals(object : Object) : boolean +get(index : int) : Object +hashCode() : int +indexOf(element : Object) : int +iterator() : Iterator +lastIndexOf(element : Object) : int +listIterator() : ListIterator +listIterator(startIndex : int) : ListIterator +remove(element : Object) : boolean +remove(index : int) : Object +removeAll(collection : Collection) : boolean +retainAll(collection : Collection) : boolean +set(index : int, element : Object) : Object +size() : int +subList(fromIndex : int, toIndex : int) : List +toArray() : Object[ ] +toArray(array : Object[ ]) : Object[ ]

UML for List and Iterator



Adds a new element at the specified index.

Adds all the elements in c to this list at the specified index.

Returns the element in this list at the specified index.

Returns the index of the first matching element.

Returns the index of the last matching element.

Returns the list iterator for the elements in this list.

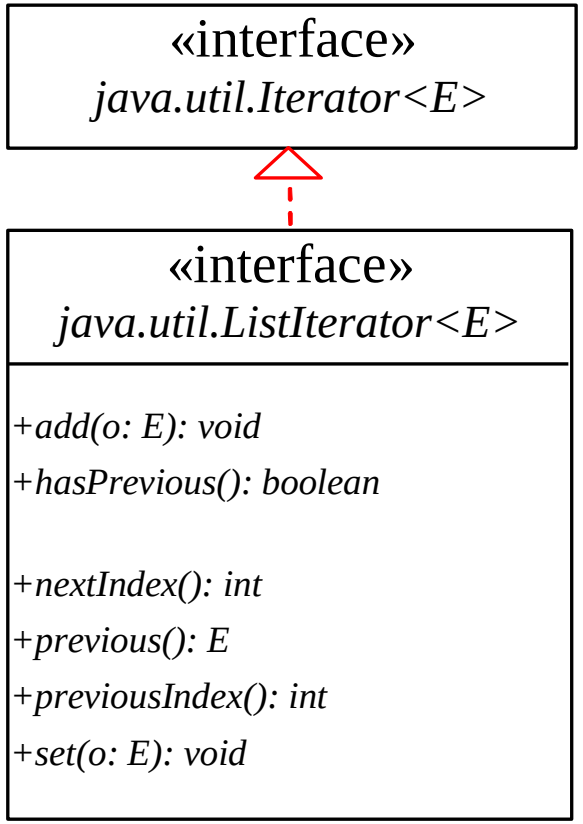
Returns the iterator for the elements from startIndex.

Removes the element at the specified index.

Sets the element at the specified index.

Returns a sublist from fromIndex to toIndex.

UML for List and Iterator



Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

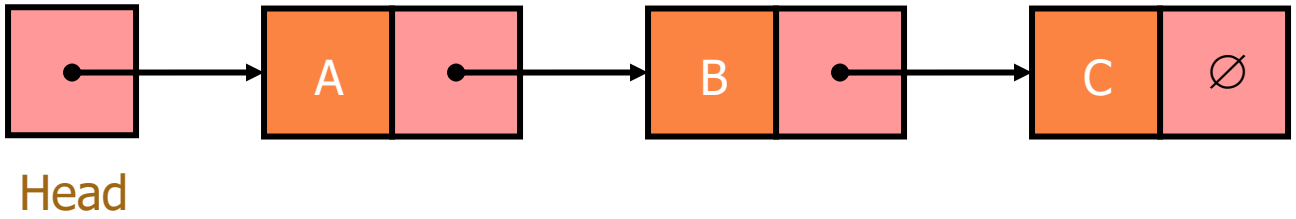
LinkedList Overview

SECTION 3

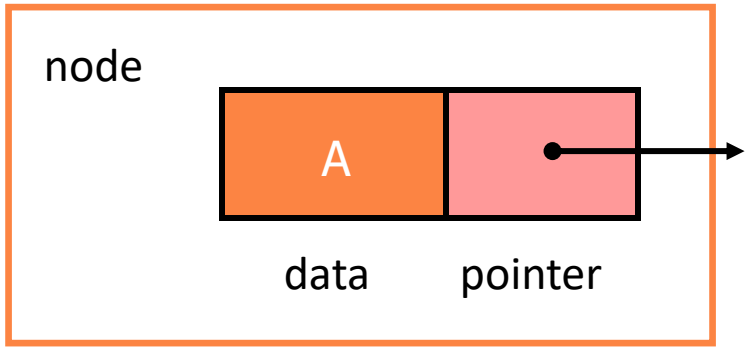
Linked List

- In computer science, a linked list is one of the fundamental dynamic data structures used in computer programming.
- It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes.

Linked Lists



- A **linked list** is a series of connected **nodes**
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- **Head**: pointer to the first node
- The last node points to NULL



Strength and Weakness

- Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.
- A linked list is a self-referential data type because it contains a link to another data of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.
- If adding and deletion is rare and you will be doing random access a lot, an array is much better. But when you have to update list frequently than a linked list would be preferable.

Advantages of linked lists

- Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
- Linked lists have **efficient** memory utilization. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
- Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists

- It consumes more space because every node requires a additional pointer to store address of the next node.
- Searching a particular element in list is difficult and also time consuming.
- Reverse Traversing is difficult in linked list

Types of Linked Lists

Basically we can put linked lists into the following...

1. Single Linked List.
2. Circular Single Linked List.
3. Double Linked List.
4. Circular Double Linked List.
5. Header Linked list
6. Multi Linked List.

Applications of linked list

1. Linked lists are used to represent and manipulate polynomial.
2. Hash tables use linked lists for collision resolution.
3. Any "File Requester" dialog uses a linked list.
4. Linked lists use to implement stack, queue, trees and graphs.
5. To implement the symbol table in compiler construction.

Some Common Variables Used in linked list

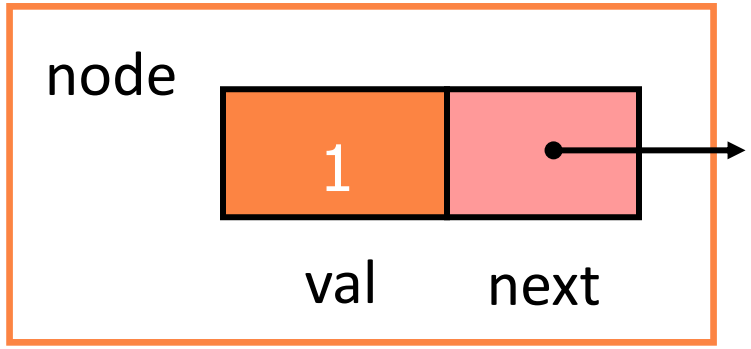
- HEAD : Store the address of first node.
- AVAIL : Store the address of first free space.
- PTR : Points to node that is currently being accessed.
- NULL : End of list.
- NEW_NODE :To store address of newly created node.

Node Class

SECTION 4

Node<T> Class

- Declare `Node<T>` Class for the nodes
  - `val`: `Integer`-type data in this example
  - `next`: a pointer to the next node in the list



Demonstration Program

NODE.JAVA

```java
public class Node<T extends Comparable> implements Comparable<Node<T>>
{
    T val;
    Node<T> next;
    Node(T v, Node<T> n){ val = v; next = n; }
    public T get(){ return val; }
    public void set(T v){ val = v; }
    public Node<T> next(){ return next;}
    public void setNext(Node<T> n){ next = n; }
    public boolean equals(Object other){
       if (!(other instanceof Node)) return false;
       Node<T> that = (Node<T>) other;
       return val.equals(that.val);
     }
    public int compareTo(Node<T> other){
        if (val.compareTo(other.val)>0) return 1;
        else if (val.compareTo(other.val)<0) return -1;
        return 0;
     }
    public String toString(){
       return val.toString();
     }
}
```

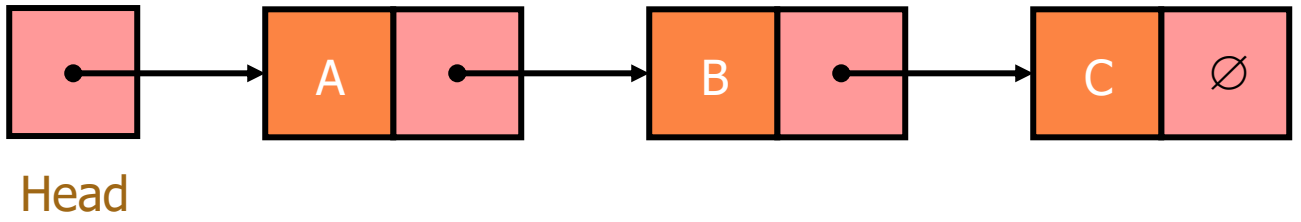
```
public class TestNode
{
    public static void main(String[] args){
      System.out.print("\f");
      Node<Integer> n1 = new Node<Integer>(new Integer(1), null);
      Node<Integer> n2 = new Node<Integer>(new Integer(2), null);
      Node<Integer> n3 = new Node<Integer>(new Integer(3), null);
      n1.setNext(n2); n2.setNext(n3);

      Node<Integer> head = n1;
      Node<Integer> p = head;
      while (p != null){
            System.out.println(p);
            p = p.next();
        }

        System.out.println(n1.compareTo(n2));
      System.out.println(n2.compareTo(n3));
    }
}
```

Generic LinkedList

SECTION 5

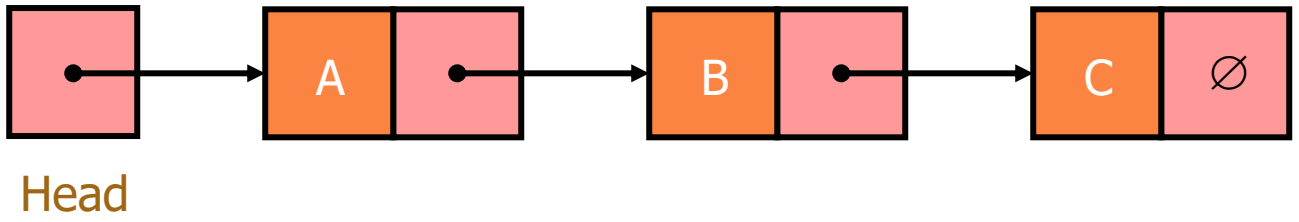
Generic LinkedList

- Operations of `LinkedList`
  - `isEmpty()`: determine whether or not the list is empty
  - `size()`: return the length of this list
  - `toString()`: convert all the nodes in the list to a string.
  - `clear()`: clear the linked list.
  - `get(int idx)`: retrieve the value at index location.
  - `set(int idx, T val)`: set the value at index location with value of val.



Generic LinkedList

- Operations of `LinkedList`
  - `add(Node n)`: add a node to the list.
  - `add(T val)`: add a value.
  - `addFirst(T val)`: add the value to the head
  - `addLast(T val)`: add the value to the tail
  - `add(int idx, T val)`: add the value to the index location
  - `remove()`: remove the value at the tail.
  - `removeFirst()`: remove the value at the head.
  - `removeLast()`: remove the value at the last.
  - `remove(int idx)`: remove the value at the index location.



Operations

SECTION 6

```
public class LinkedList<T extends Comparable> implements Iterable<T>, Iterator<T>, Cloneable
{
    Node head;
    int size;
    private int count=0;
    LinkedList(){
        head = null;
        size = 0;
     }
    LinkedList(T[] a){
        this();
        for (T x: a){
            add(x);
          }
     }
    LinkedList(ArrayList<T> a){
        this();
        for (T x: a){
            add(x);
          }
     }
```

Basic Functions

clear()

- Reset the linked list to the initial condition.

```
public void clear(){ head=null; size=0; }
```

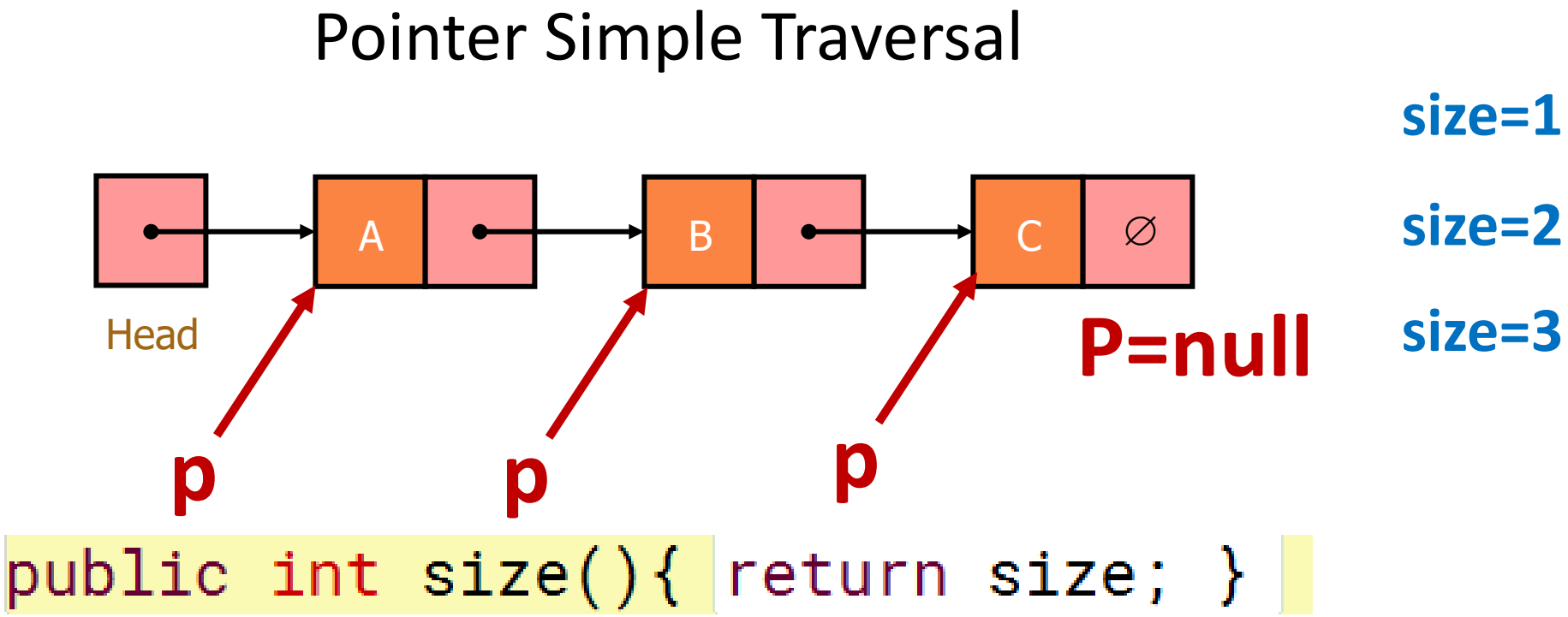
isEmpty()

- To check if a list is empty. Just check the list head pointer. If it is NULL, then the list is empty.

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

size()

- Using running pointer to find the length of the linked list.



```
public int indexOf(T v){
    Node p = head;
    int c =0;
    while (p!=null){
        if (v.equals(p.get())) return c;
        p = p.next();
        c++;
     }
    return -1;
 }
```

indexOf()

- Find the index location with the target value.

contains(T v)

- Check if a value is contained in a linked list.

```
public boolean contains(T v){
    return indexOf(v)>=0;
 }
```

```
public boolean equals(Object otherlist){
    if (!(otherlist instanceof LinkedList)) return false;
    LinkedList that = (LinkedList) otherlist;
    if (this.size() != that.size()) return false;
    Node pthis = head;
    Node pthat = that.head;
    while (pthis!=null){
        if (!pthis.equals(pthat)) return false;
        pthis = pthis.next();
        pthat = pthat.next();
     }
    return true;
 }
```

equals()

- Check if two linked lists are equal.

```
public String toString(){
    String r = "[";
    Node p = head;
    int c=0;
    while (p!=null){
        if (c==0) r += p.toString();
        else r += ", "+p.toString();
        c++;
        p = p.next();
      }
    r += "]";
    return r;
 }
```

toString()

- Convert the values of all nodes into a string.

```
public T get(int idx){
    if (idx<0 || idx>size()-1) return null;
    Node p = head;
    Node q = p;
    int c=0;
    while (c<=idx && p!=null){
          q = p;
          p = p.next();
          c++;
      }
    return (T) q.val;
 }
```

T get(int idx)

- Retrieve the value at index location idx

```
public T set(int idx, T data){
    if (idx<0 || idx>size()-1) return null;
    Node p = head;
    Node q = p;
    int c=0;
    while (c<=idx && p!=null){
        q = p;
        p = p.next();
        c++;
    }
    T old = (T) q.val;
    q.set(data);
    return old;
}
```

T set(int idx, T val)

- Set a node with a new value

Insertion and Deletion Functions

```
public boolean addFirst(T val){
    size++;
    Node n = new Node(val, null);
    if (head == null){
        head = n;  return true;
    }
    Node p = head;
    head = n;
    n.next = p;
    return true;
}
```

boolean addFirst(T val)

- Set a value to the first node of a list.

```
public boolean addLast(T val){ return add(val, null); }
public boolean add(T val){ return add(val, null); }
public boolean add(T val, Node next){
    size++;
    Node n = new Node(val, next);
    if (head == null){
        head = n;  return true;
     }
    Node p = head;
    Node q = p;
    while (p != null){
        q = p;
        p = p.next();
     }
    q.setNext(n);
    return true;
}
```

boolean addLast(T val)

- Add a value to the last node of the list

```
public boolean add(int idx, T val){
    if (idx==0)      { addFirst(val); return true; }
    if (idx==size()) { addLast(val);  return true; }
    if (idx<0 || idx>size()) return false;
    size++;
    int c=0;
    Node p = head;
    Node q = head;
    while (c!=idx && p!=null){
        q = p;
        p = p.next();
        c++;
    }
    Node n = new Node(val, p);
    q.setNext(n);
    return true;
}
```

boolean add(int idx, T val)

- Add a value to a specific index location

```
public T removeFirst(){
    if (head==null) return null;
    size--;

    Node q = head;
    head = head.next();
    q.setNext(null);
    return (T) q.get();
}
```

T removeFirst()

- Remove first value

```
public T removeLast(){
    if (head==null) return null;
    size--;
    Node p = head;
    Node q = p;
    Node r = q;
    while (p!=null){
        r = q;
        q = p;
        p = p.next();
      }
    q.setNext(null);
    if (size==0) head = null;
    else r.setNext(null);
    return (T) q.get();
  }
```

T removeLast()

- Set a node with a new value

```
public T remove(int idx){
    if (idx<0 || idx>size()-1) return null;
    if (idx==0)          { return removeFirst(); }
    if (idx==size()-1) { return remove(); }
    size--;
    Node p = head;
    Node q = p;
    Node r = q;
    int c=0;
    while (c<=idx && p!=null){
        r = q;
        q = p;
        p = p.next();
        c++;
    }
    r.setNext(q.next());
    q.setNext(null);
    return (T) q.get();
}
```

T remove(int idx)

- Remove a value a certain index location

Iterator/Iterable Methods

```
public Iterator<T> iterator(){ return this; }
  public boolean hasNext(){
      if (count < size) return true;
      return false;
  }
  public T next(){
      if (count >= size) throw new NoSuchElementException();
      count++;
      return get(count-1);
  }
public void remove(){
     if (count>0) count--;
     remove(count);
  }
```

Unsupported Functions

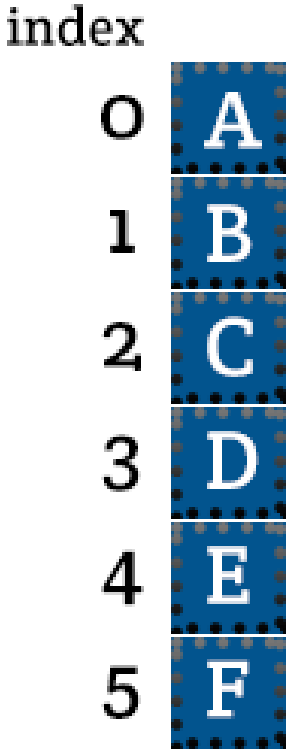
Unsupported Functions

- **subList**: return a subList of a certain range
- **append**: append two lists
- **replace/replaceAll**: replace certain elements with certain values.
- **removeAll**: remove all elements in a list.
- **retain/retainAll**: keep all elements in a list (remove all others).
- **lastIndexOf**: get the index of the last occurrence of an object.

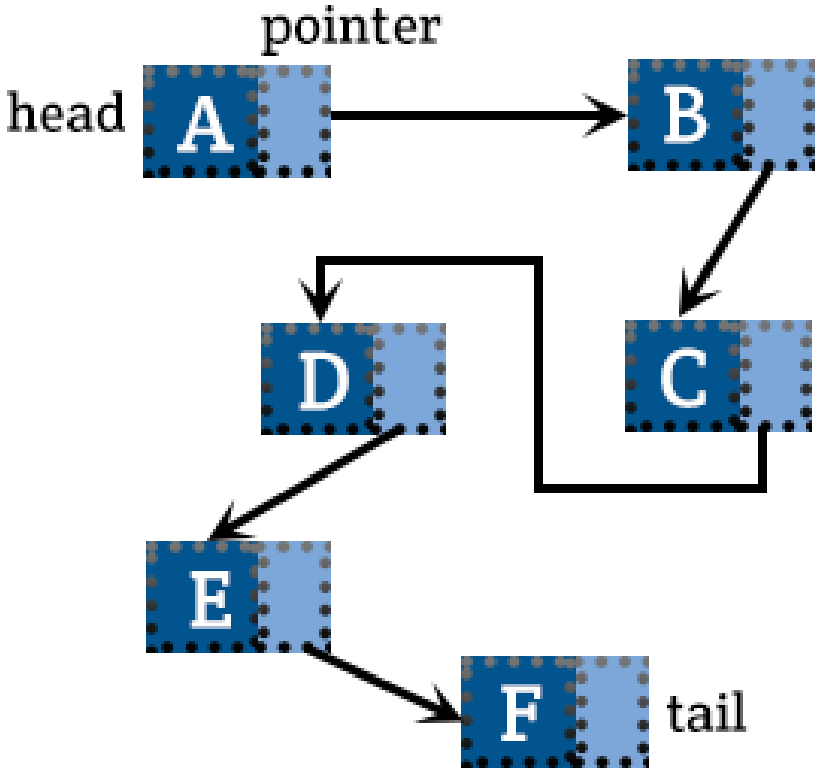
Java Collections Framework

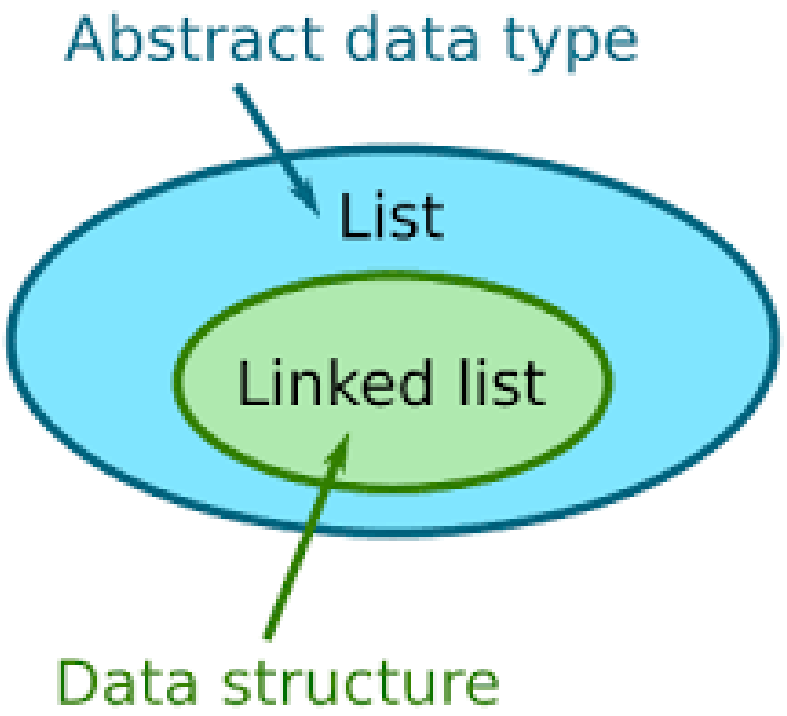
Data Structures

Array



Linked List





`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]"`

General Purpose Implementations

	Hash Table	Resizable array	balanced tree	linked list
Set	HashSet		TreeSet (sortedSet)	
List		ArrayList Vector		LinkedList
Map	HashMap Hashtable		TreeMap (sortedMap)	

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

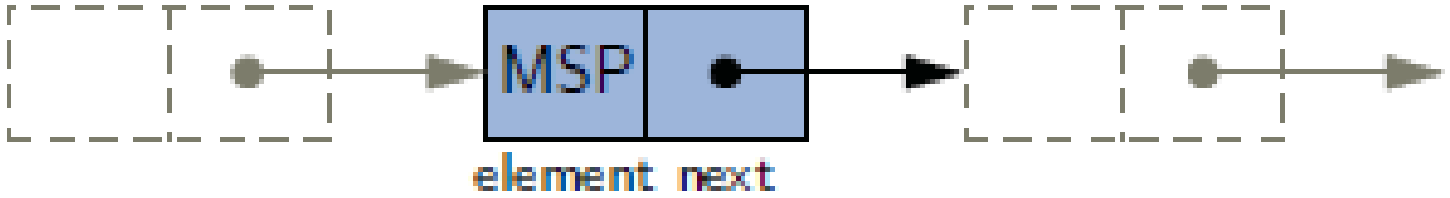
	Get	ContainsKey	Next	Data Structure
HashMap	O(1)	O(1)	O(h / n)	Hash Table
LinkedHashMap	O(1)	O(1)	O(1)	Hash Table + Linked List
IdentityHashMap	O(1)	O(1)	O(h / n)	Array
WeakHashMap	O(1)	O(1)	O(h / n)	Hash Table
EnumMap	O(1)	O(1)	O(1)	Array
TreeMap	O(log n)	O(log n)	O(log n)	Red-black tree
ConcurrentHashMap	O(1)	O(1)	O(h / n)	Hash Tables
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	Skip List

Singly Linked Lists

SECTION 7

Singly Linked Lists

- A linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. In a singly linked list, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.



Pointers

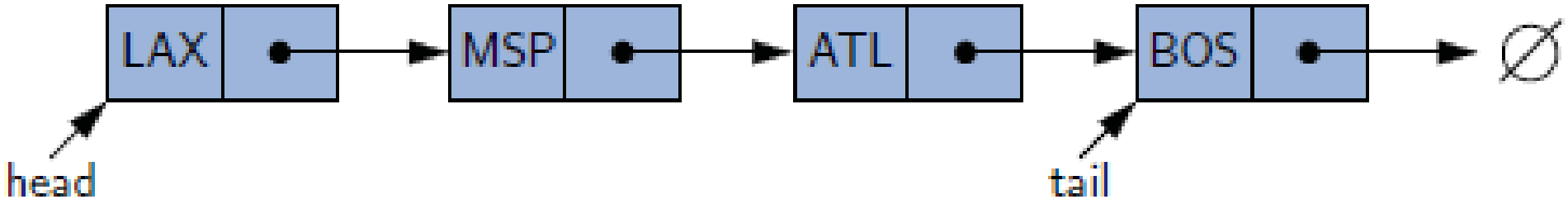
- Head and Tail:

  Minimally, the linked list instance must keep a reference to the first node of the list, known as the head. Without an explicit reference to the head, there would be no way to locate that node (or indirectly, any others). The last node of the list is known as the tail.

- The tail of a list can be found by traversing the linked list—starting at the head and moving from one node to another by following each node's next reference. We can identify the tail as the node having null as its next reference.

- This process is also known as **link hopping** or **pointer hopping**.

Pointers



Implementing a Singly 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.

Demonstration Program

SINGLYLINKEDLIST.JAVA AND ITS TESTER

Equivalence Testing with Linked Lists

SECTION 8

```
1   public boolean equals(Object o) {
2     if (o == null) return false;
3     if (getClass( ) != o.getClass( )) return false;
4     SinglyLinkedList other = (SinglyLinkedList) o;        // use nonparameterized type
5     if (size != other.size) return false;
6     Node walkA = head;                                    // traverse the primary list
7     Node walkB = other.head;                              // traverse the secondary list
8     while (walkA != null) {
9       if (!walkA.getElement( ).equals(walkB.getElement( ))) return false; //mismatch
10      walkA = walkA.getNext( );
11      walkB = walkB.getNext( );
12    }
13    return true;      // if we reach this, everything matched successfully
14  }
```

equals

```
public boolean equals(Object otherlist){
    if (!(otherlist instanceof LinkedList)) return false;
    LinkedList that = (LinkedList) otherlist;
    if (this.size() != that.size()) return false;
    Node pthis = head;
    Node pthat = that.head;
    while (pthis!=null){
        if (!pthis.equals(pthat)) return false;
        pthis = pthis.next();
        pthat = pthat.next();
     }
    return true;
 }
```

Cloning Arrays

SECTION 9

Cloning Arrays

- Although arrays support some special syntaxes such as a[k] and a.length, it is important to remember that they are objects, and that array variables are reference variables. This has important consequences. As a first example, consider the following code:

```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data; // warning; not a copy
```

- The assignment of variable backup to data does not create any new array; it simply creates a new alias for the same array, as portrayed in Figure 3.23.

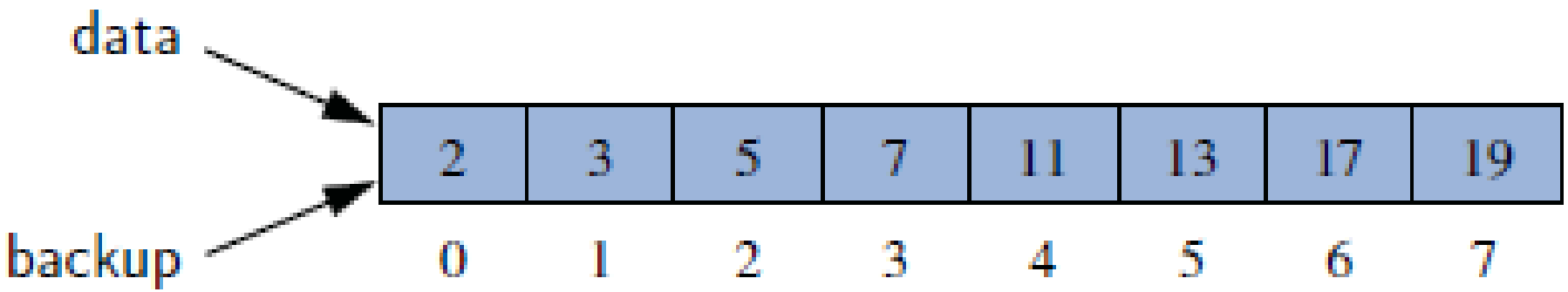


Figure 3.23: The result of the command backup = data for int arrays.

Cloning Arrays

- Instead, if we want to make a copy of the array, data, and assign a reference to the new array to variable, backup, we should write:

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

- The clone method, when executed on an array, initializes each cell of the new array to the value that is stored in the corresponding cell of the original array. This results in an independent array, as shown in Figure 3.24.

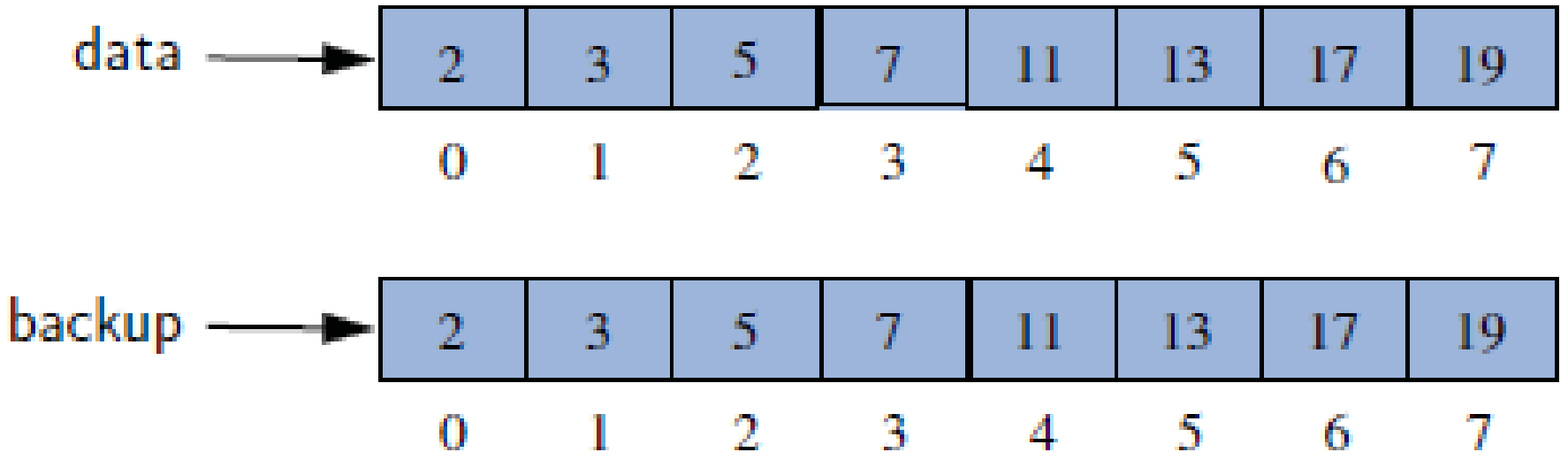


Figure 3.24: The result of the command backup = data.clone( ) for int arrays.

Cloning Arrays

- If we subsequently make an assignment such as data[4] = 23 in this configuration, the backup array is unaffected.
- There are more considerations when copying an array that stores reference types rather than primitive types. The clone( ) method produces a shallow copy of the array, producing a new array whose cells refer to the same objects referenced by the first array.

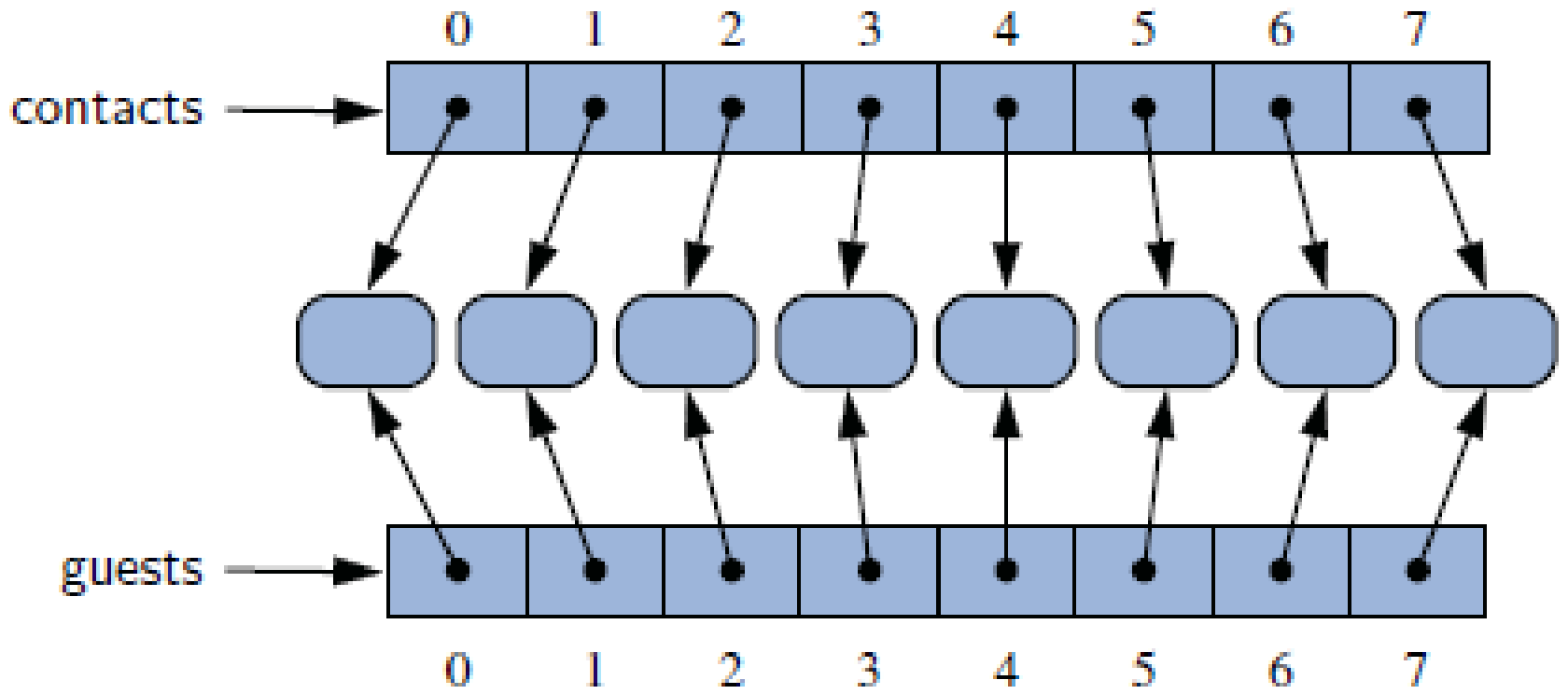


Figure 3.25: A shallow copy of an array of objects, resulting from the command guests = contacts.clone( ).

Shallow Copy

- If you have shallow copy, one of the array changes, the other will also be affected.
- If the data is read only, this will be fine.

Deep Copy

- A ***deep copy*** of the contact list can be created by iteratively cloning the individual elements, as follows, but only if the Person class is declared as Cloneable.

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

- Because a two-dimensional array is really a one-dimensional array storing other one-dimensional arrays, the same distinction between a shallow and deep copy exists. Unfortunately, the java.util.Arrays class does not provide any “deepClone” method. However, we can implement our own method by cloning the individual rows of an array, as shown in Code Fragment 3.20, for a two-dimensional array of integers.

Demonstration Program

CIRCLE.JAVA AND TESTCIRCLE.JAVA

```java
static int[][] m1 = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
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(); // copy row k
    return backup;
}
public static void doubleMatrix(int[][] m){
  for (int i=0; i<m.length; i++){
     for (int j=0; j<m[i].length; j++){
        m[i][j] *=2;
      }
   }
}
```

```
Before doubling:
[
  1,   2,   3
  4,   5,   6
  7,   8,   9
]
[
  1,   2,   3
  4,   5,   6
  7,   8,   9
]
After doubling:
[
  1,   2,   3
  4,   5,   6
  7,   8,   9
]
[
  2,   4,   6
  8,  10,  12
 14,  16,  18
]
```

Cloning Linked Lists

- In this section, we add support for cloning instances of the SinglyLinkedList class from Section 3.2.1. The first step to making a class cloneable in Java is declaring that it implements the Cloneable interface. Therefore, we adjust the first line of the class definition to appear as follows:

```
public class SinglyLinkedList<E>
implements Cloneable {
```

- The remaining task is implementing a public version of the clone( ) method of the class, which we present in Code Fragment 3.21.

Cloning Linked Lists

```
1   public SinglyLinkedList<E> clone( ) throws CloneNotSupportedException {
2     // always use inherited Object.clone() to create the initial copy
3     SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone( ); // safe cast
4     if (size > 0) {                                  // we need independent chain of nodes
5       other.head = new Node<>(head.getElement( ), null);
6       Node<E> walk = head.getNext( );    // walk through remainder of original list
7       Node<E> otherTail = other.head;    // remember most recently created node
8       while (walk != null) {                         // make a new node storing same element
9         Node<E> newest = new Node<>(walk.getElement( ), null);
10        otherTail.setNext(newest);                   // link previous node to this one
11        otherTail = newest;
12        walk = walk.getNext( );
13      }
14    }
15    return other;
16  }
```

Code Fragment 3.21: Implementation of the SinglyLinkedList.clone method.

Adding Clone function to LinkedList

```
public LinkedList<T> clone() throws CloneNotSupportedException {
    LinkedList<T> r = (LinkedList<T>) super.clone();
    if (size()<1) return r;
    Node p = head;
    int c=0;
    Node rTail=null;
    while (p!=null){
         Node q = new Node(p.get(), null);
         if (c==0) { rTail=q; r.head = q; r.size=1; }
         else { rTail.setNext(q);
                rTail = q;
                r.size++;
              }
         p=p.next();
         c++;
      }
    return r;
 }
```

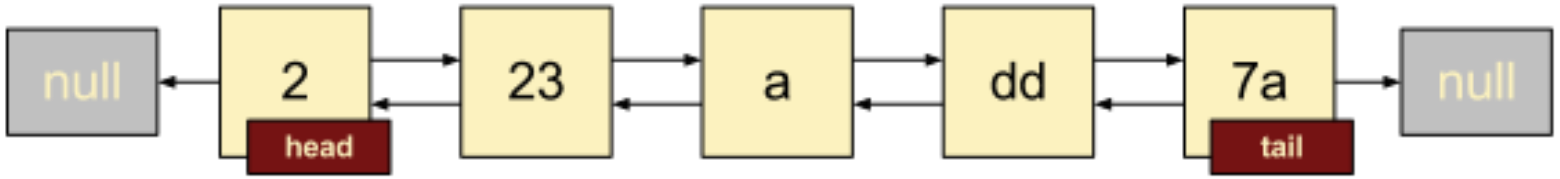
Java LinkedList

SECTION 10

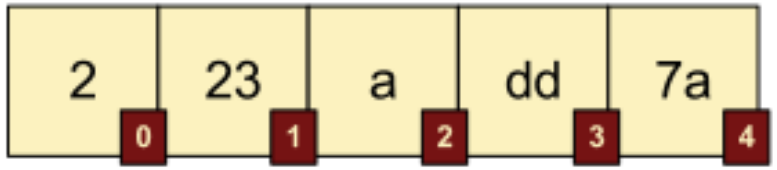
Linked List

Array vs. Linked List

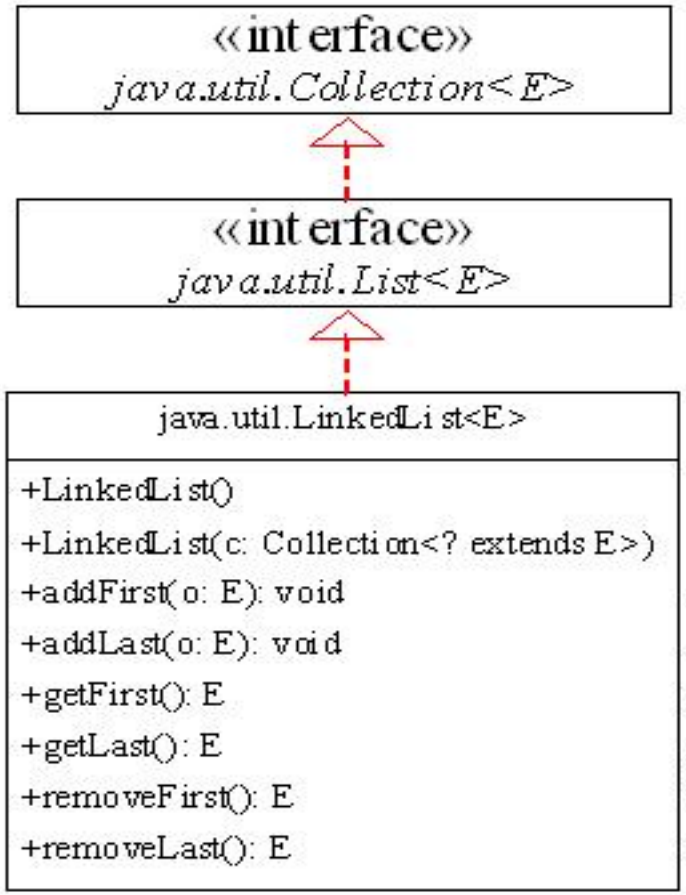
Linked List



Array



LinkedList



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

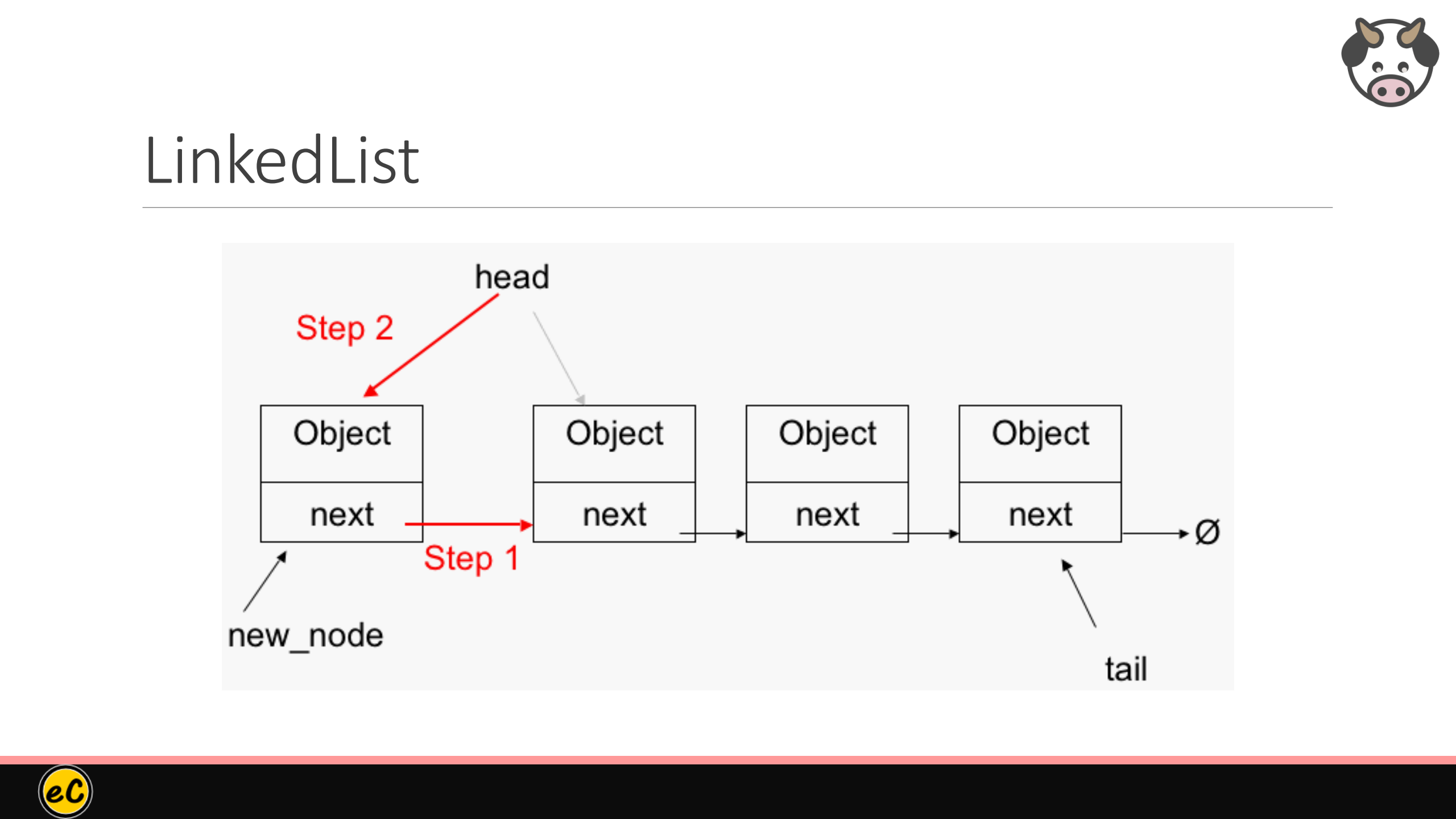
Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

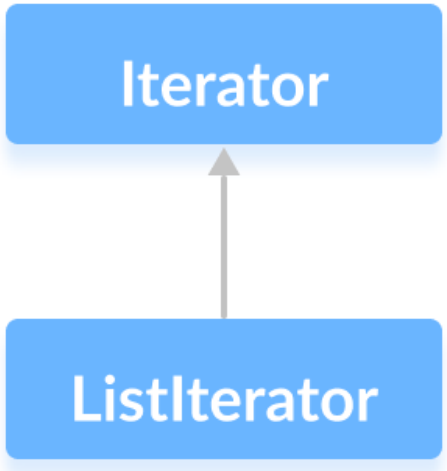
Returns and removes the first element from this list.

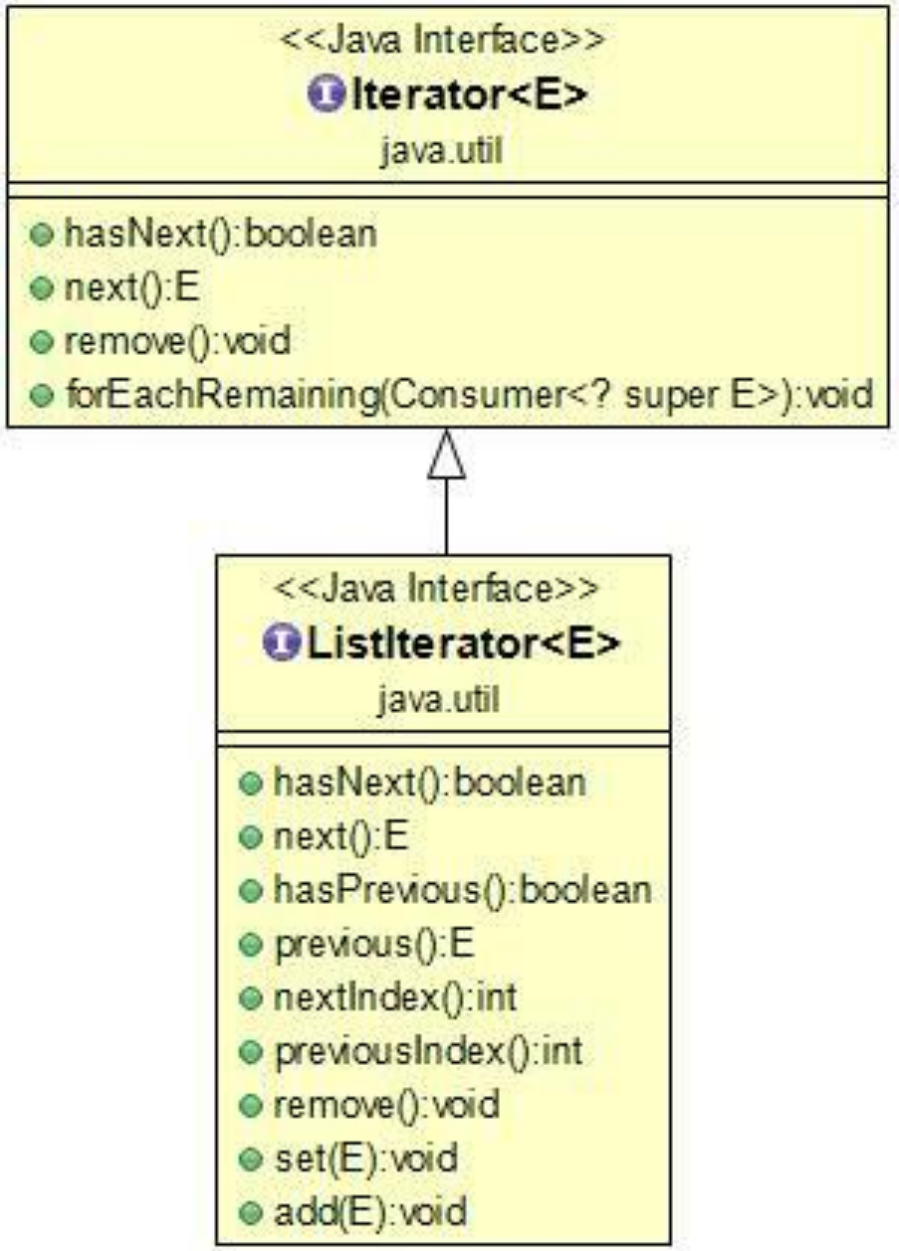
Returns and removes the last element from this list.



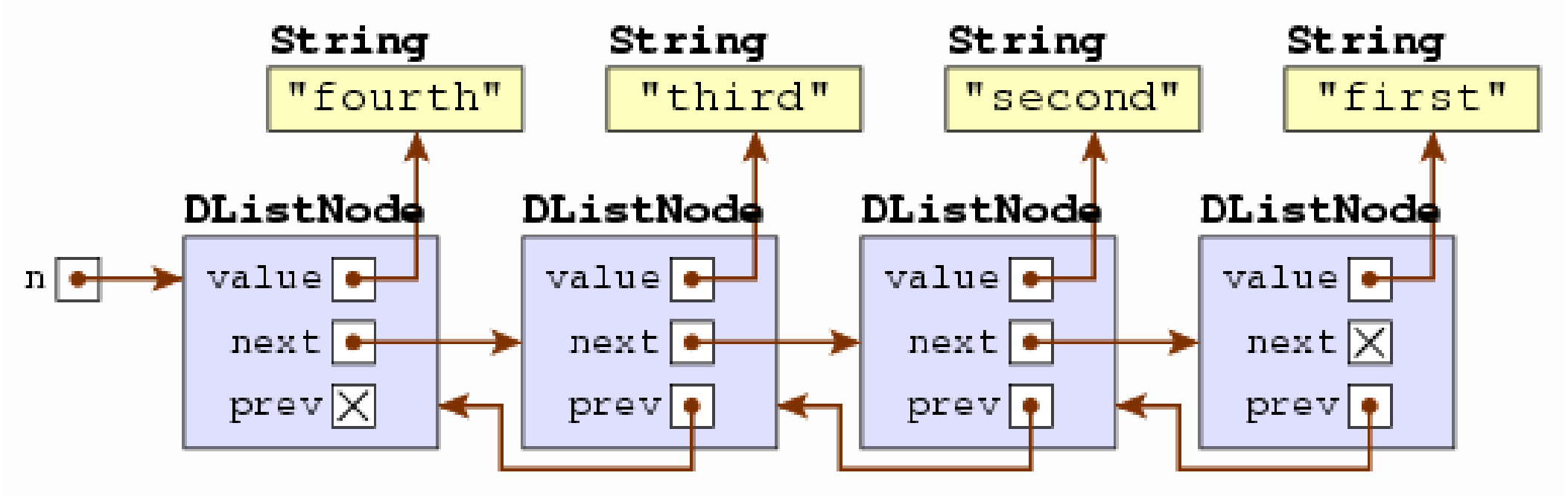
LinkedList

head
Step 2
Object
next
Object
next
Object
next
Object
next
Ø
Step 1
new_node
tail


Iterator
ListIterator


<<Java Interface>>
Iterator<E>
java.util
hasNext():boolean
next():E
remove():void
forEachRemaining(Consumer<? super E>):void
<<Java Interface>>
ListIterator<E>
java.util
hasNext():boolean
next():E
hasPrevious():boolean
previous():E
nextIndex():int
previousIndex():int
remove():void
set(E):void
add(E):void

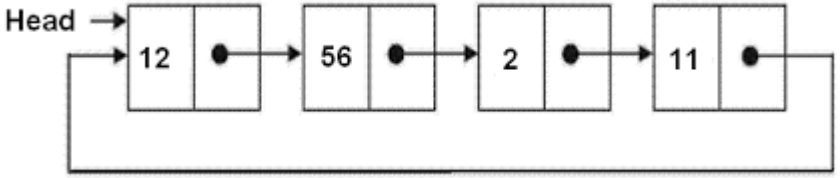
LinkedList


String
"fourth"
String
"third"
String
"second"
String
"first"
DListNode
DListNode
DListNode
DListNode
n
value
next
prev
value
next
prev
value
next
prev
value
next
prev

Demonstration Program

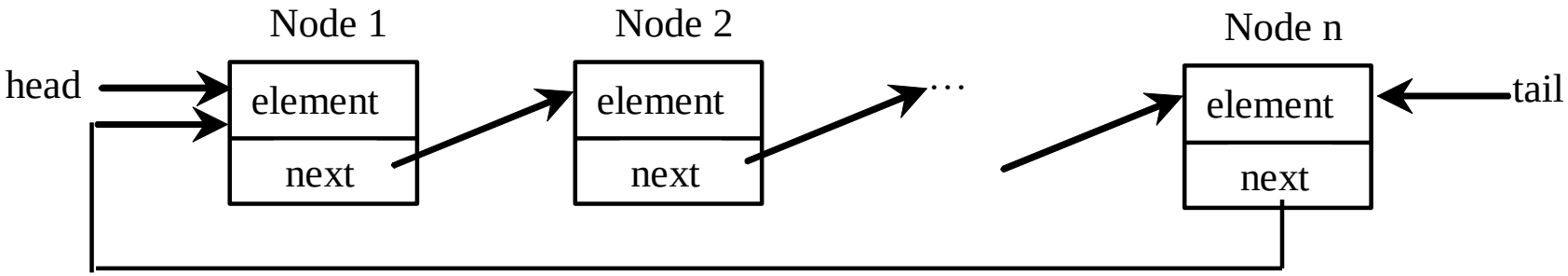
LINKEDLISTDEMO.JAVA

Methods	MyArrayList/ArrayList	MyLinkedList/LinkedList
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$



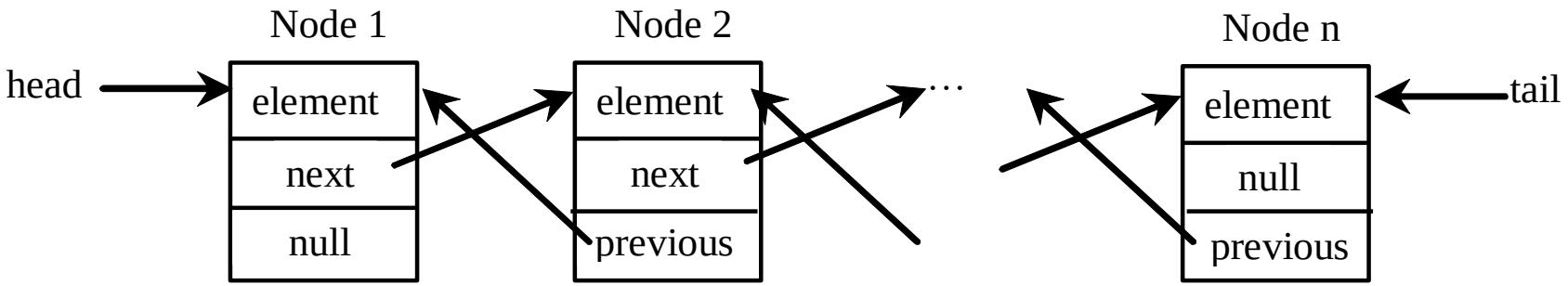
Circular Linked Lists

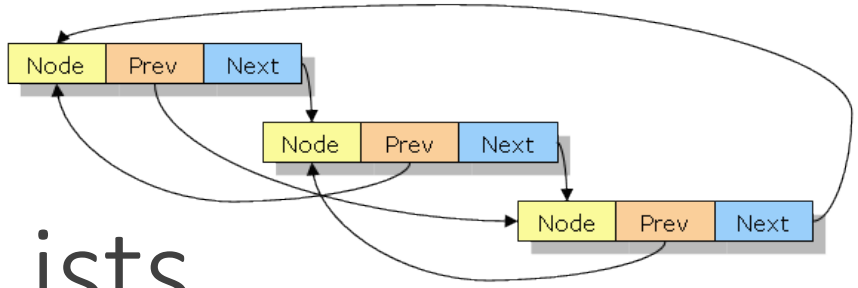
A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



Doubly Linked Lists

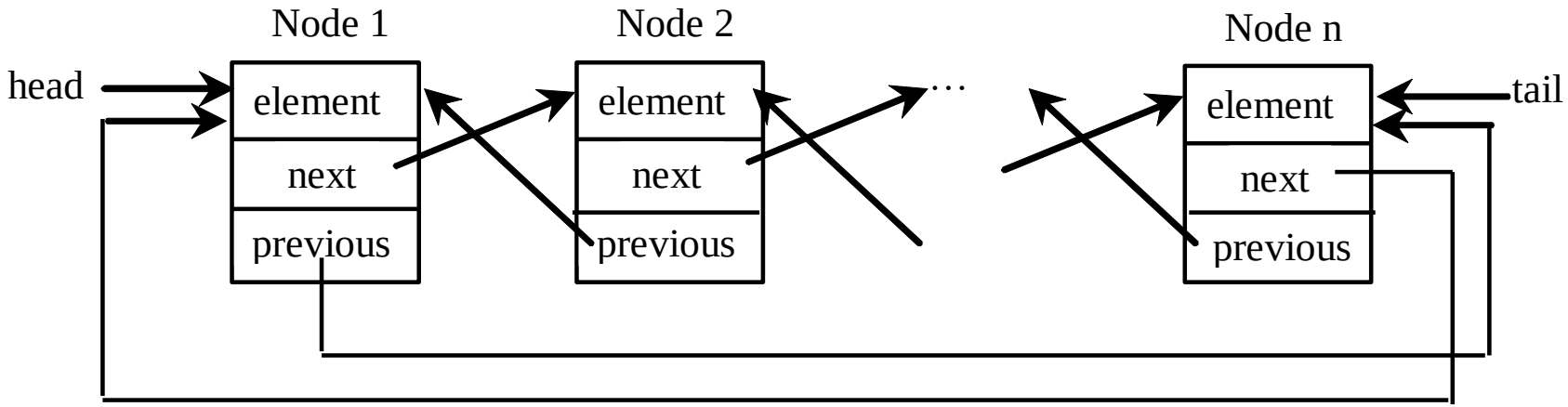
A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.





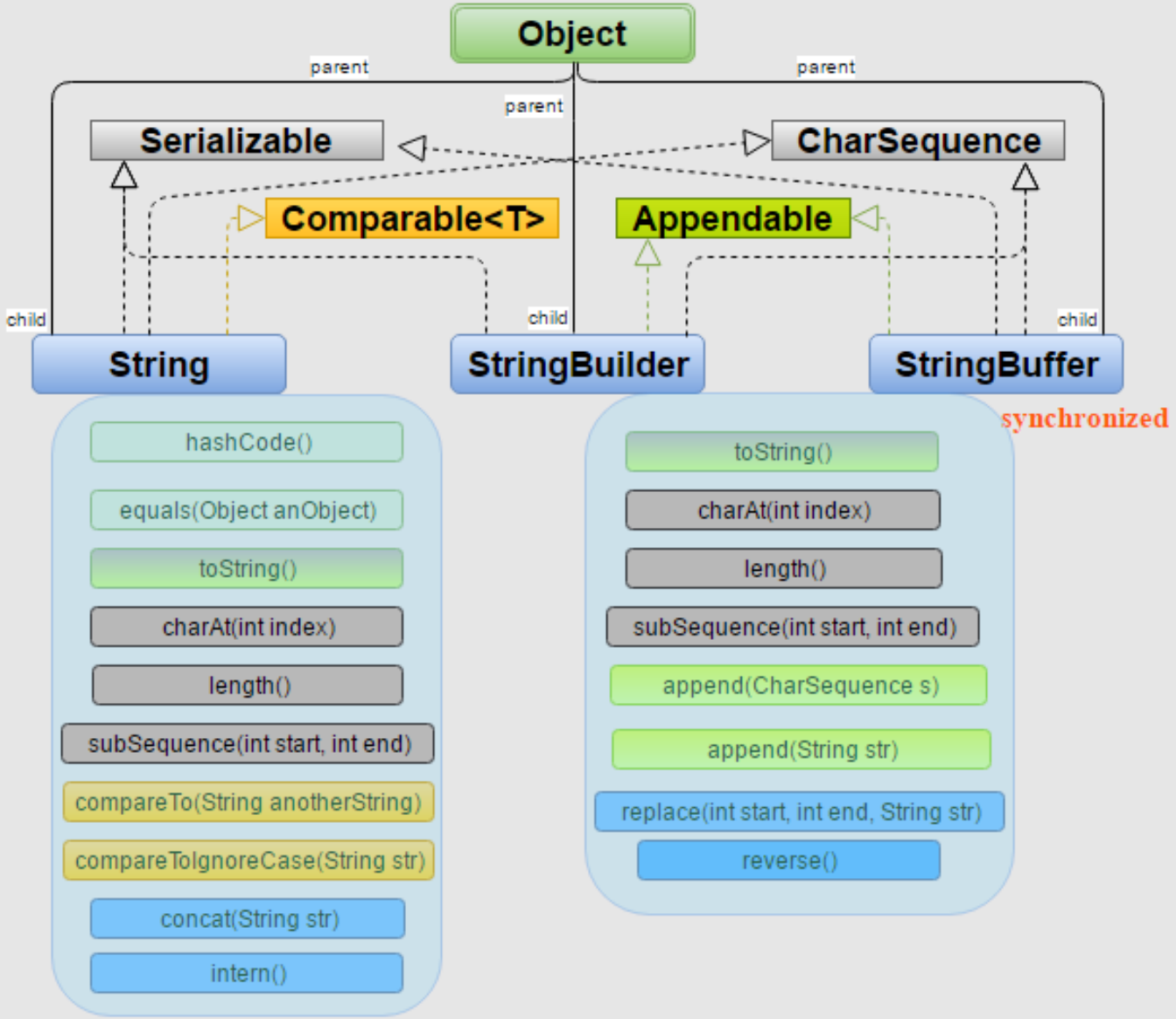
Circular Doubly Linked Lists

A *circular*, *doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



StringBuilder Class

SECTION 11


Object
parent
parent
parent
Serializable
CharSequence
Comparable<T>
Appendable
child
child
child
String
StringBuilder
StringBuffer
synchronized
hashCode()
equals(Object anObject)
toString()
charAt(int index)
length()
subSequence(int start, int end)
compareTo(String anotherString)
compareToIgnoreCase(String str)
concat(String str)
intern()
toString()
charAt(int index)
length()
subSequence(int start, int end)
append(CharSequence s)
append(String str)
replace(int start, int end, String str)
reverse()

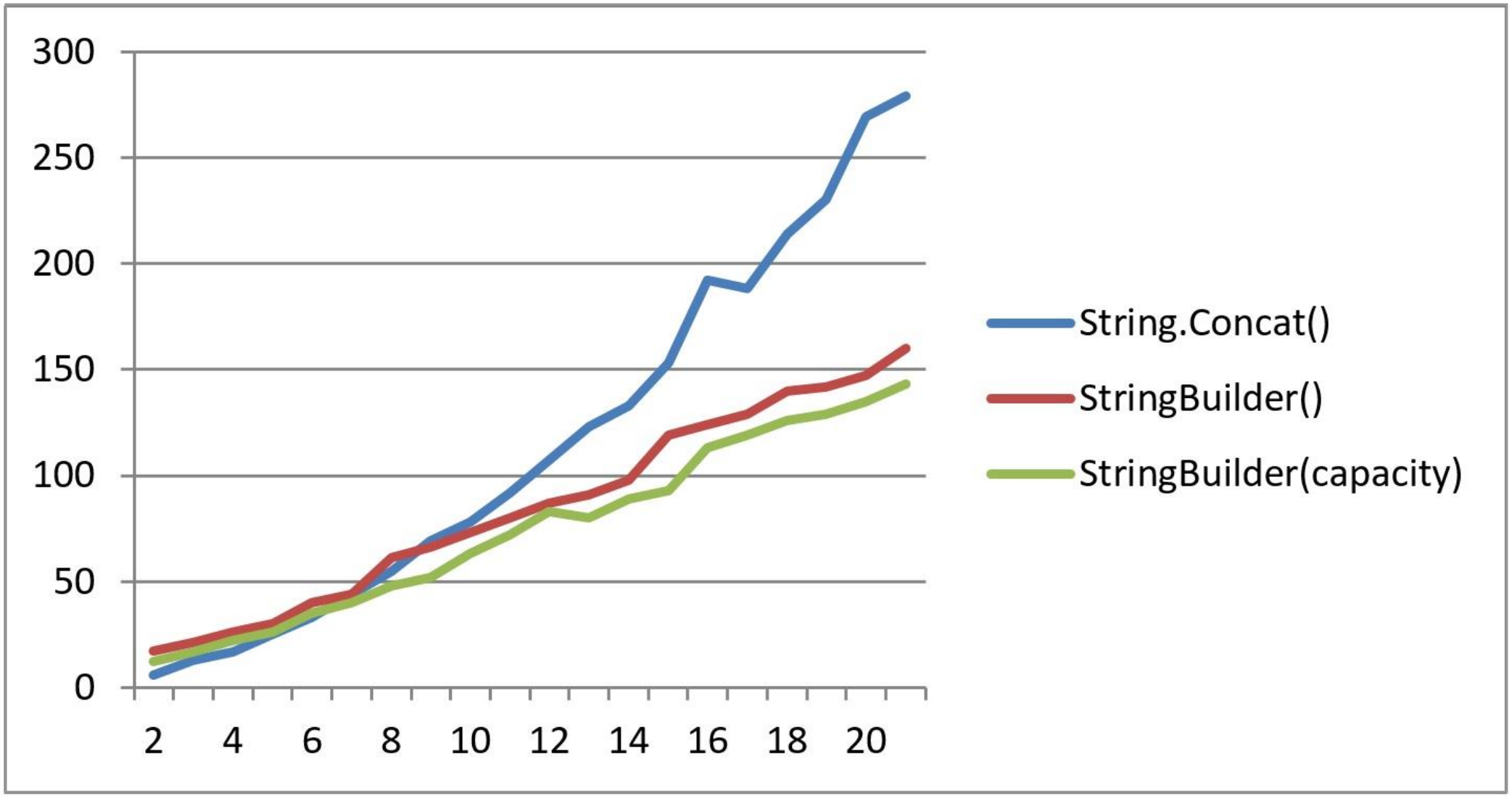
Popular Methods

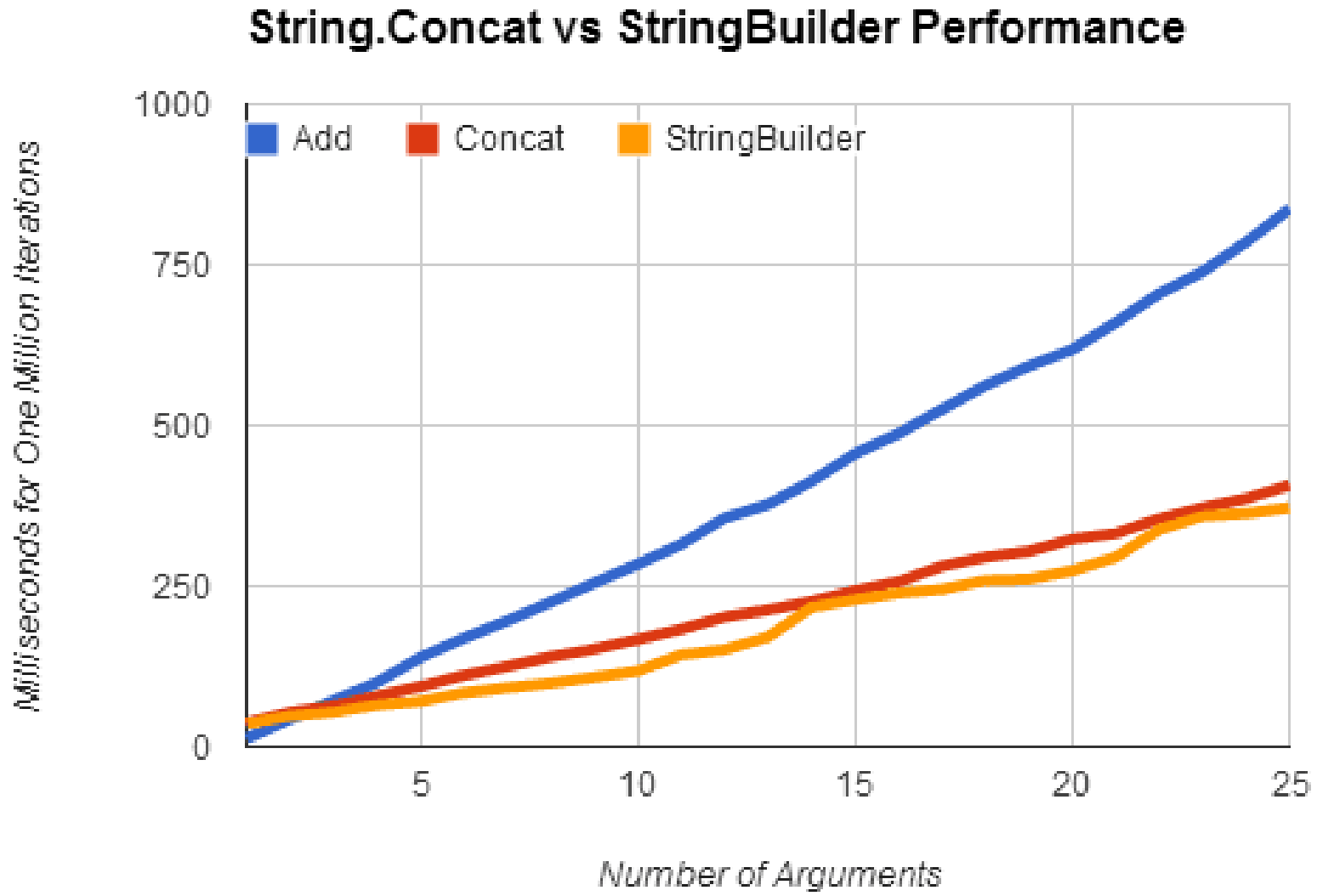
- append(String str)
- replace(int idx1, int idx2, String str)
- capacity()
- ensureCapacity(int size)
- insert(int idx, String str)
- delete(int idx1, int idx2)
- length()
- charAt(int idx)
- deleteCharAt(int idx)
- setCharAt(int idx, char ch)
- indexOf(String str)

	String	StringBuffer	StringBuilder
Storage	String pool	Heap	Heap
Modifiable	No(immutable)	Yes (mutable)	Yes (mutable)
Thread safe	Yes	Yes	No
Synchronized	Yes	Yes	No
Performance	Fast	Slow	Fast

String Builder is Faster

- Two string concatenation cases. The first of those relied on repeated concatenation using the String class, and the second relied on use of Java's StringBuilder class.
- We observed the StringBuilder was significantly faster, with empirical evidence that suggested a quadratic running time for the algorithm with repeated concatenations, and a linear running time for the algorithm with the StringBuilder. We are now able to explain the theoretical underpinning for those observations.


300
250
200
150
100
50
0
2
4
6
8
10
12
14
16
18
20
String.Concat()
StringBuilder()
StringBuilder(capacity)


String.Concat vs StringBuilder Performance
Add
Concat
StringBuilder
1000
750
500
250
0
5
10
15
20
25
Milliseconds for One Million Iterations
Number of Arguments

Demonstration Program

STRINGBUILDERDEMO.JAVA

The Java Collection Framework (Optional)

SECTION 12

Class	Interfaces			Properties			Storage	
	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

List Iterators in Java

add(*e*): Adds the element *e* at the current position of the iterator.

hasNext( ): Returns true if there is an element after the current position of the iterator.

hasPrevious( ): Returns true if there is an element before the current position of the iterator.

previous( ): Returns the element *e* before the current position and sets the current position to be before *e*.

next( ): Returns the element *e* after the current position and sets the current position to be after *e*.

nextIndex( ): Returns the index of the next element.

previousIndex( ): Returns the index of the previous element.

remove( ): Removes the element returned by the most recent next or previous operation.

set(*e*): Replaces the element returned by the most recent call to the next or previous operation with *e*.

Positional List ADT Method	java.util.List Method	ListIterator Method	Notes
size( )	size( )		$O(1)$ time
isEmpty( )	isEmpty( )		$O(1)$ time
	get(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$
first( )	listIterator( )		first element is next
last( )	listIterator(size( ))		last element is previous
before(p)		previous( )	$O(1)$ time
after(p)		next( )	$O(1)$ time
set(p, e)		set(e)	$O(1)$ time
	set(i, e)		A is $O(1)$, L is $O(\min\{i, n-i\})$
	add(i, e)		$O(n)$ time
addFirst(e)	add(0, e)		A is $O(n)$, L is $O(1)$
addFirst(e)	addFirst(e)		only exists in L, $O(1)$
addLast(e)	add(e)		$O(1)$ time
addLast(e)	addLast(e)		only exists in L, $O(1)$
addAfter(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
addBefore(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
remove(p)		remove( )	deletion is at cursor; A is $O(n)$, L is $O(1)$
	remove(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$

Table 7.4: Correspondences between methods in our positional list ADT and the java.util interfaces List and ListIterator. We use A and L as abbreviations for java.util.ArrayList and java.util.LinkedList (or their running times).

List-Based Algorithms in the Java Collections Framework

copy(*Ldest* , *Lsrc*): Copies all elements of the *Lsrc* list into corresponding indice of the *Ldest* list.
disjoint(*C*, *D*): Returns a boolean value indicating whether the collections *C* and *D* are disjoint.
fill(*L*, *e*): Replaces each element of the list *L* with element *e*.
frequency(*C*, *e*): Returns the number of elements in the collection *C* that are equal to *e*.
max(*C*): Returns the maximum element in the collection *C*, based on the natural ordering of its elements.
min(*C*): Returns the minimum element in the collection *C*, based on the natural ordering of its elements.
replaceAll(*L*, *e*, *f*): Replaces each element in *L* that is equal to *e* with element *f* .
reverse(*L*): Reverses the ordering of elements in the list *L*.
rotate(*L*, *d*): Rotates the elements in the list *L* by the distance *d* (which can be negative), in a circular fashion.
shuffle(*L*): Pseudorandomly permutes the ordering of the elements in the list *L*.
sort(*L*): Sorts the list *L*, using the natural ordering of its elements.
swap(*L*, *i*, *j*): Swap the elements at indices *i* and *j* of list *L*.

Converting Lists into Arrays

toArray( ): Returns an array of elements of type Object containing all the elements in this collection.

toArray(*A*): Returns an array of elements of the same element type as *A* containing all the elements in this collection.

Converting Arrays into Lists

asList(*A*): Returns a list representation of the array *A*, with the same element type as the elements of *A*.

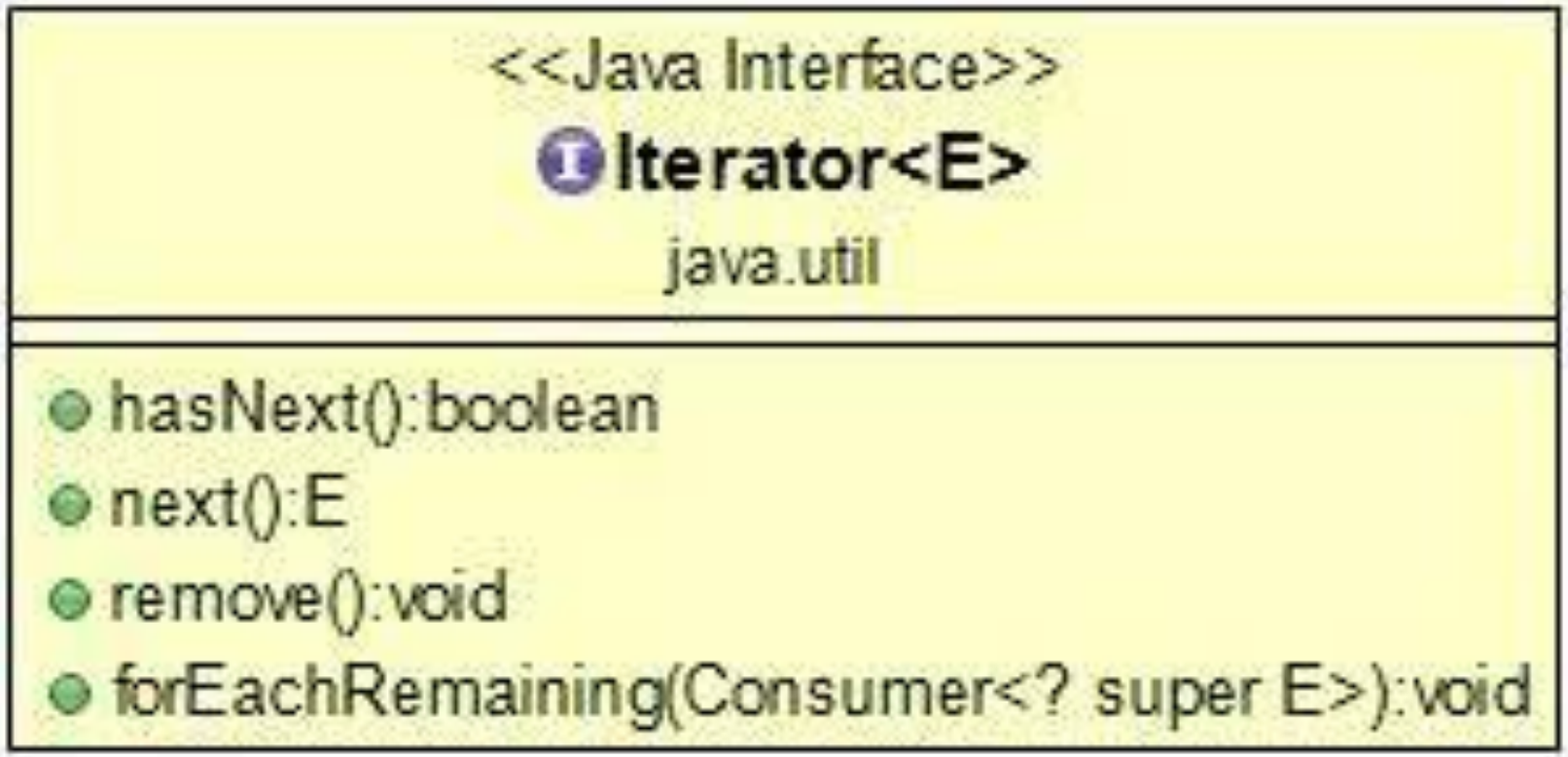
Iterators (Optional)

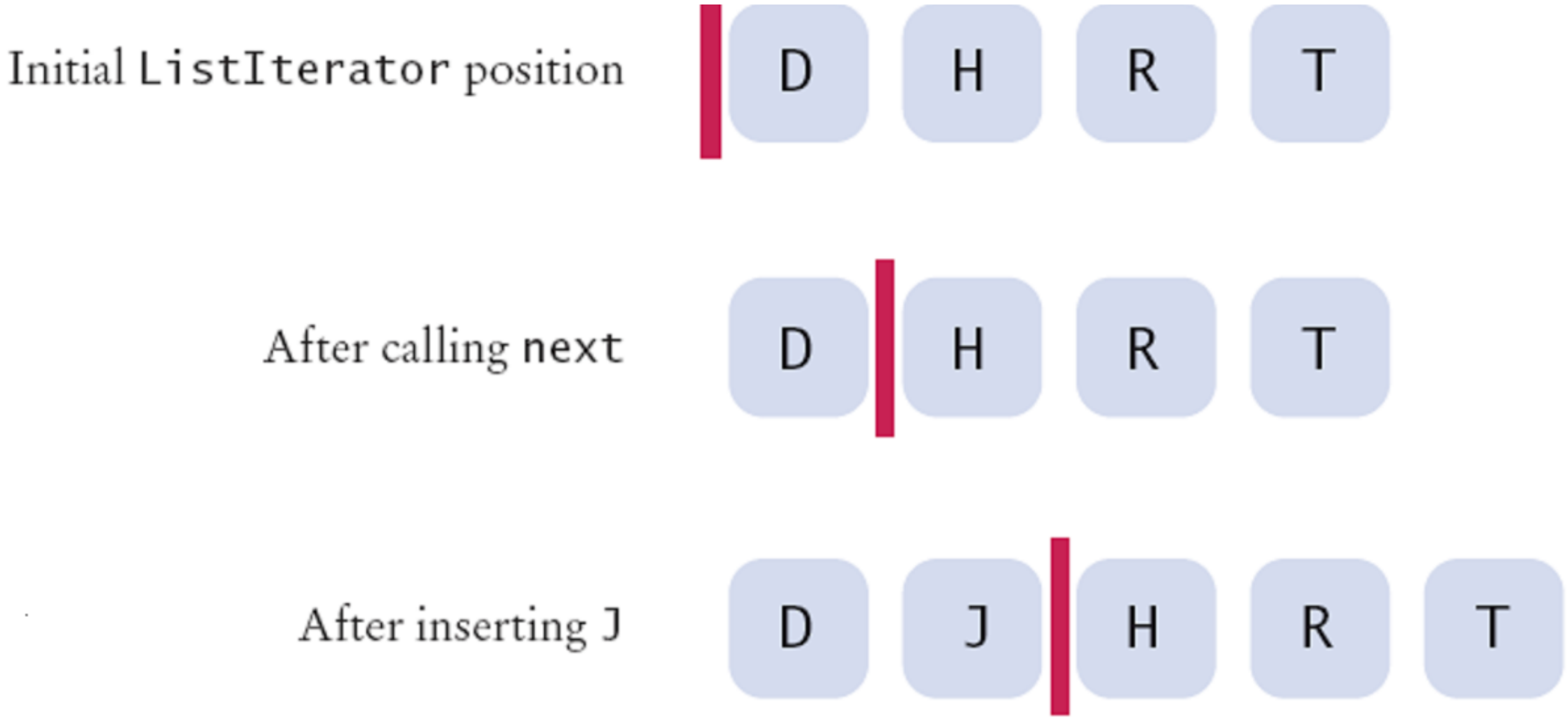
SECTION 13

Enumeration

Java Iterator

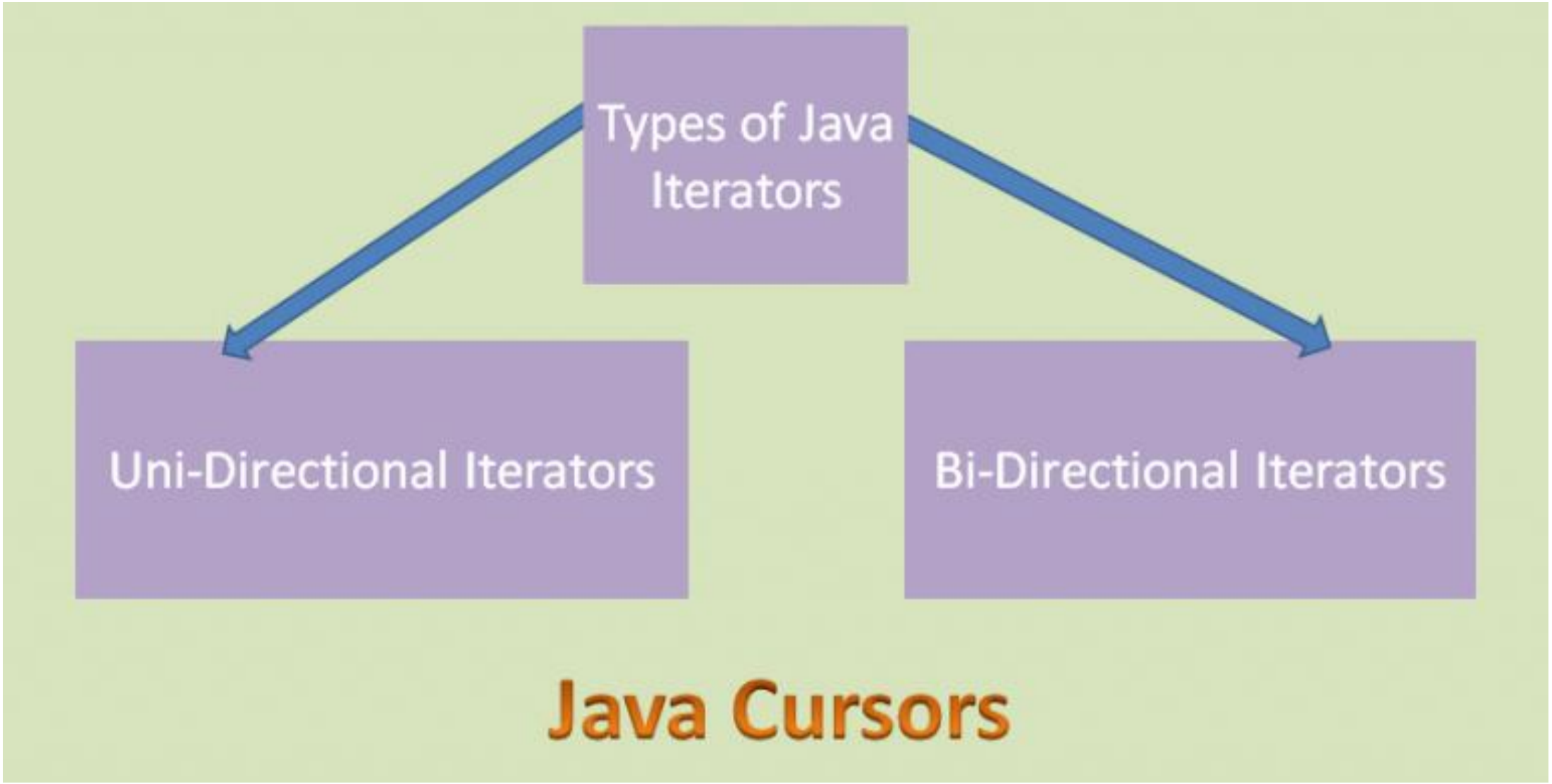
ListIterator

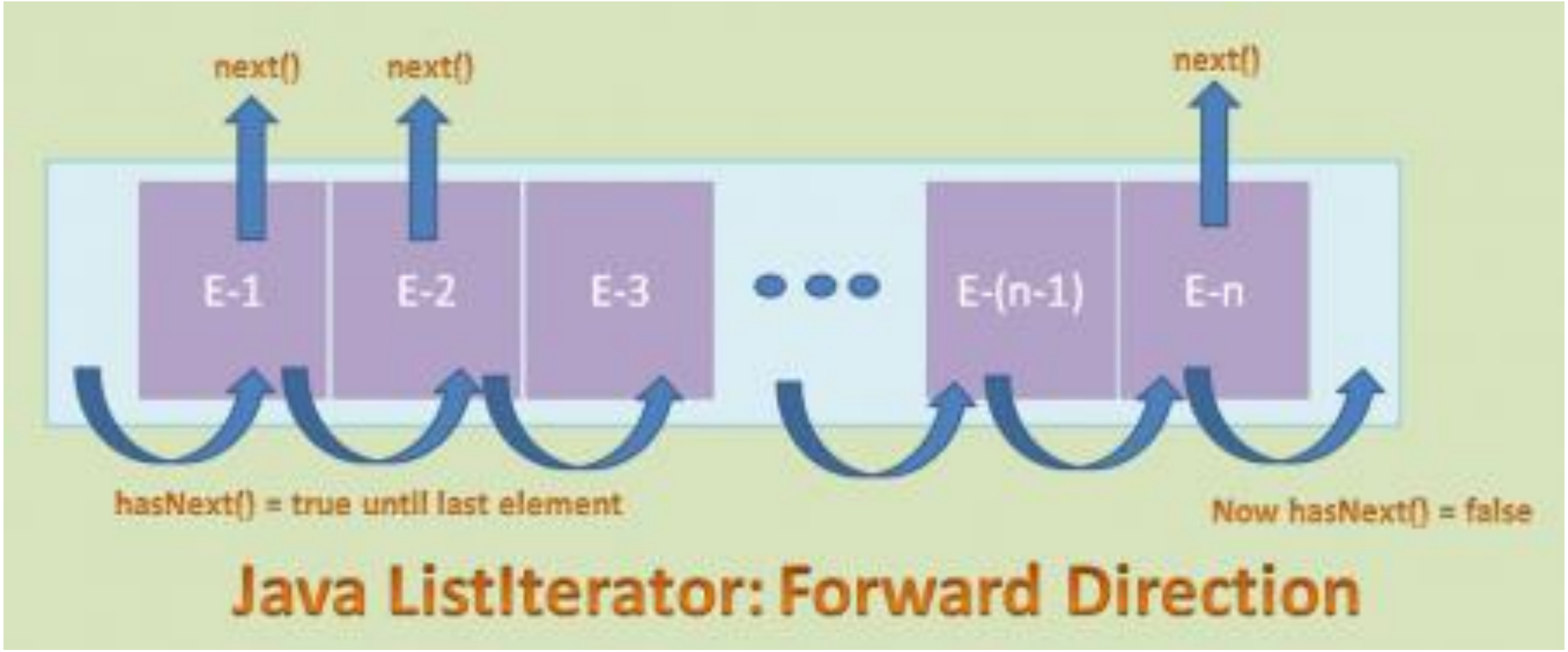

<<Java Interface>>
Iterator<E>
java.util
hasNext():boolean
next():E
remove():void
forEachRemaining(Consumer<? super E>):void

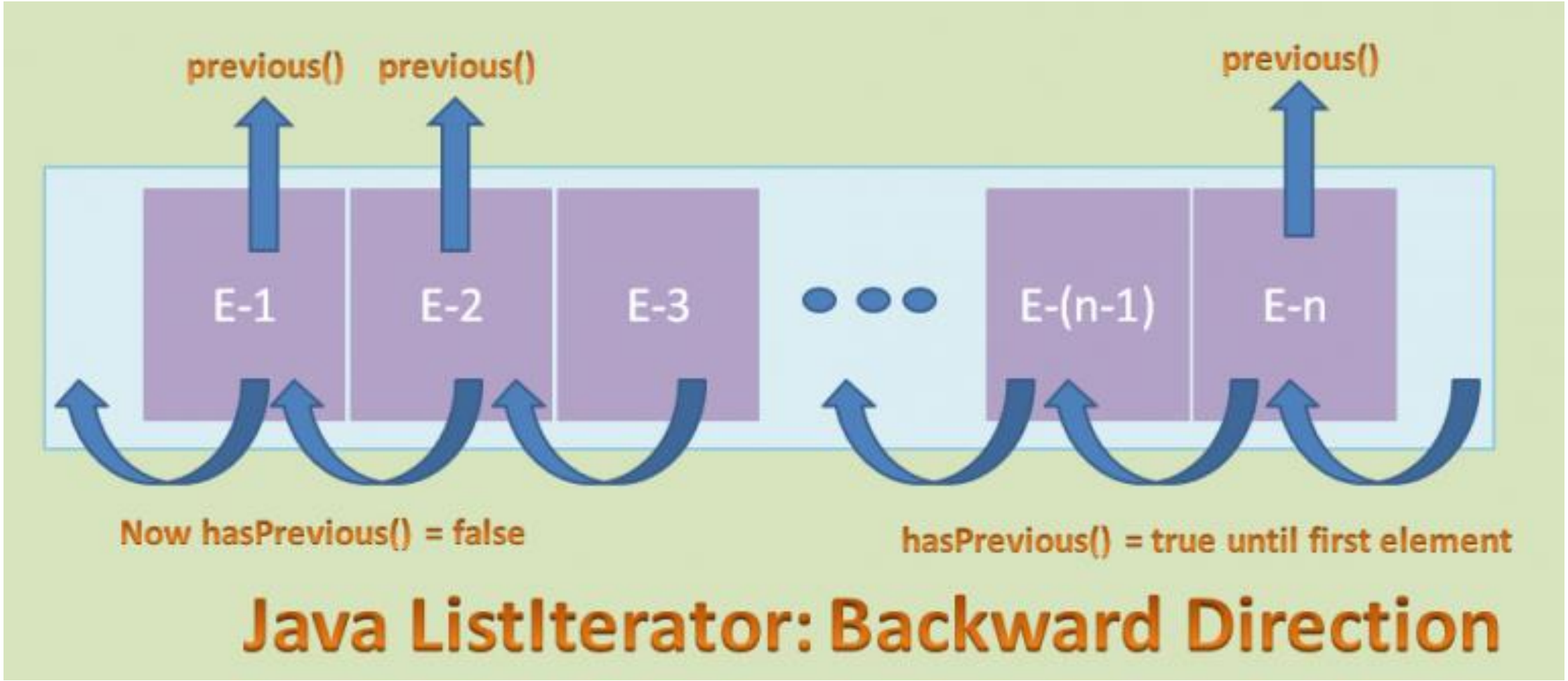


A Conceptual View of the List Iterator


<<Java Interface>>
ListIterator<E>
java.util
hasNext():boolean
next():E
hasPrevious():boolean
previous():E
nextIndex():int
previousIndex():int
remove():void
set(E):void
add(E):void


Types of Java Iterators
Uni-Directional Iterators
Bi-Directional Iterators
Java Cursors


next()
next()
next()
E-1
E-2
E-3
E-(n-1)
E-n
hasNext() = true until last element
Now hasNext() = false
Java ListIterator: Forward Direction


previous()
previous()
previous()
E-1
E-2
E-3
E-(n-1)
E-n
Now hasPrevious() = false
hasPrevious() = true until first element
Java ListIterator: Backward Direction

The Iterable Interface and Java's For-Each Loop

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.

For Each Loop

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

while(iter.hasNext()) loop

Iterator<*ElementType*> iter = *collection*.iterator( );

```
while (iter.hasNext( )) {
    ElementType variable = iter.next( );
    loopBody // may refer to ”variable”
}
```

Filtering Loop

```
ArrayList<Double> data; // populate with random numbers

Iterator<Double> walk = data.iterator( );

while (walk.hasNext( ))
    if (walk.next( ) < 0.0) walk.remove( );
```

Demonstration Program

ARRAY AND TESTARRAY.JAVA

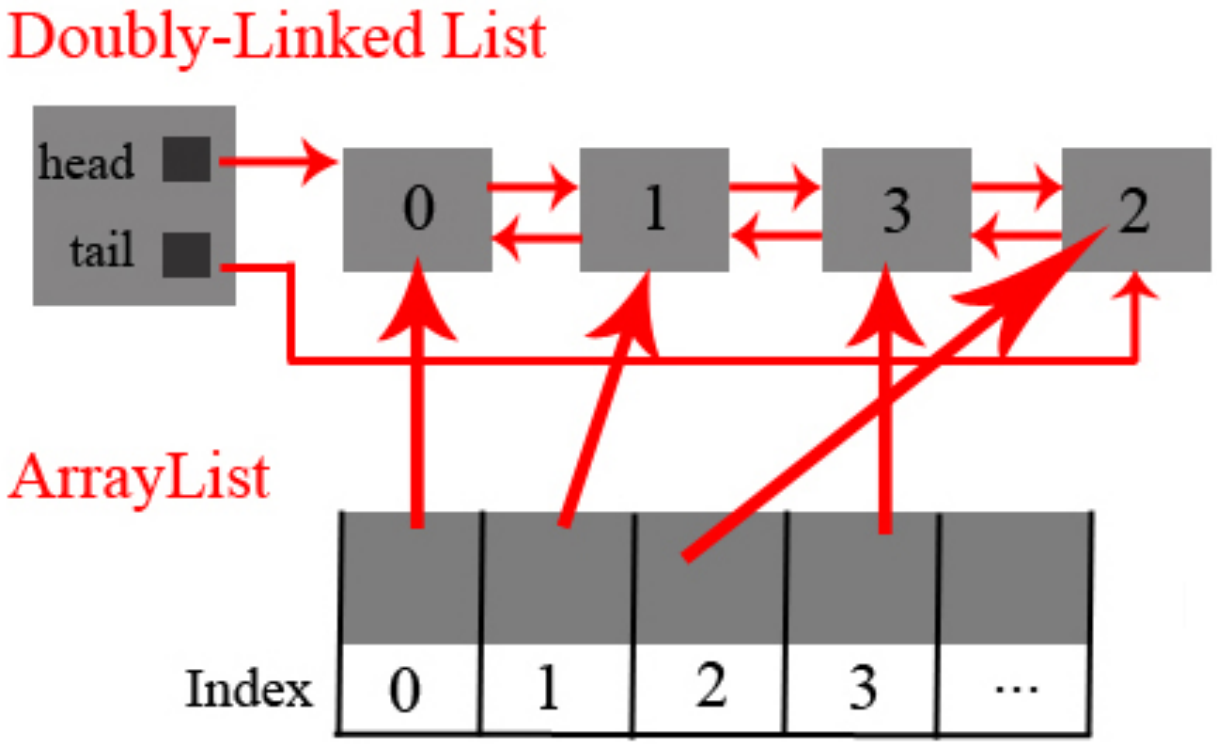
(RE-WRITTEN FROM ARRAYITERATOR CLASS)

ArrayList (Optional)

SECTION 14

ArrayList

List work like an array



ArrayList

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E): void	Appends a new element o at the end of this list.
+add(index: int, o: E): void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first element o from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index. Returns true if an element is removed.
+set(index: int, o: E): E	Sets the element at the specified index.

ArrayList Versus LinkedList

ArrayList	LinkedList
1) ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Demonstration Program

SORTING BY ARRAYLIST
SELECTIONSORT.JAVA

Demonstration Program

SORTING BY ARRAYLIST
SELECTIONSORT.JAVA