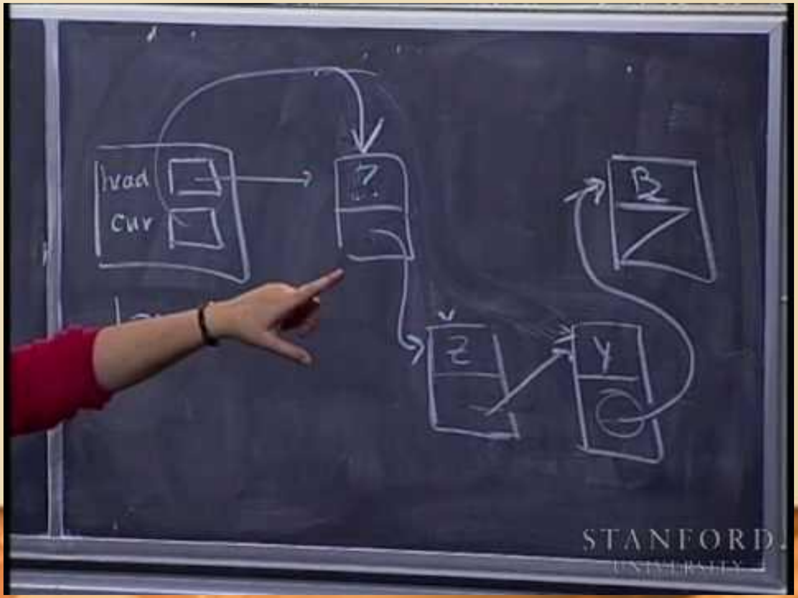
Introduction to Computer Science

Reichman University

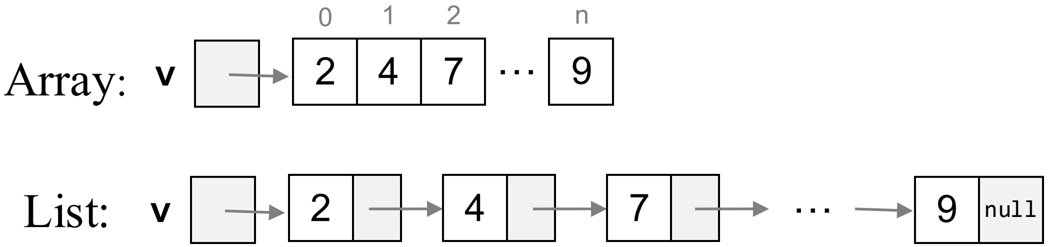
Lecture 10-1

Data Structures I



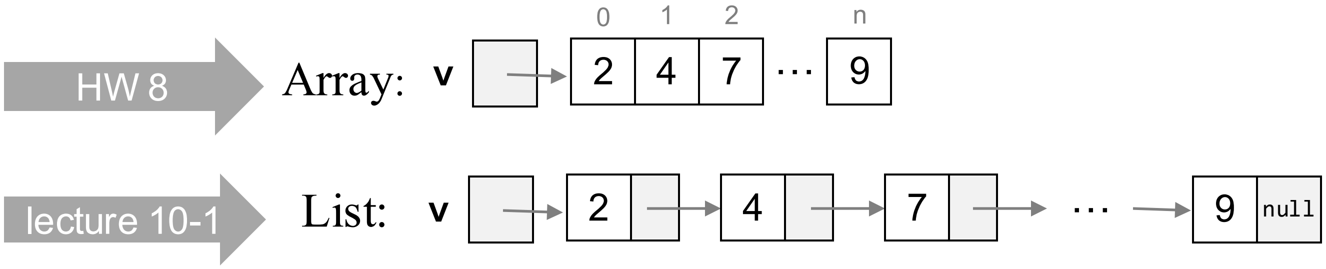
Data structures

Basic data structures



Data structures

Basic data structures



HW 8

Array:

lecture 10-1

List:

Abstract Data Structures (ADTs)

lecture 10-2

Set

Stack

Queue

Map

Tree

...

Typically implemented using arrays or lists

Lists

List: an ordered collection of elements.

Examples

Shopping list

User list

Playlist

Course list

Guest list

File list

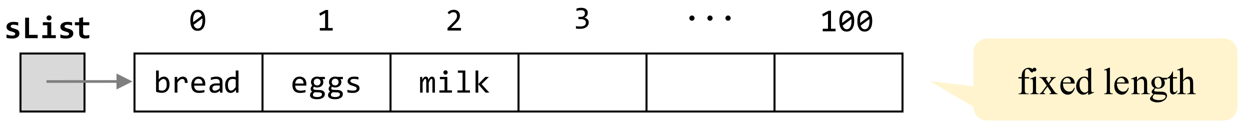
etc.

Typical list-oriented operations

- Create a list
- Add an element
- Find an element
- Remove an element
- Update an element

Arrays vs Lists

Array implementation



sList

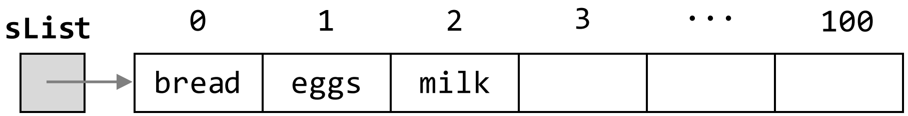
0	1	2	3	…	100
bread	eggs	milk			

fixed length

- The elements are stored in a sufficiently large array, without gaps
- Adding / removing an element in an array of size n: $\boldsymbol{O(n)}$

Arrays vs Lists

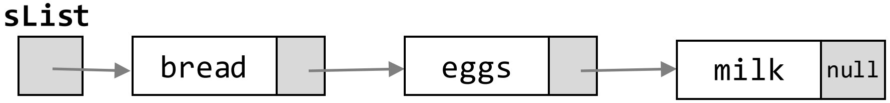
Array implementation



fixed length

- The elements are stored in a sufficiently large array, without gaps
- Adding / removing an element in an array of size n: $\boldsymbol{O(n)}$

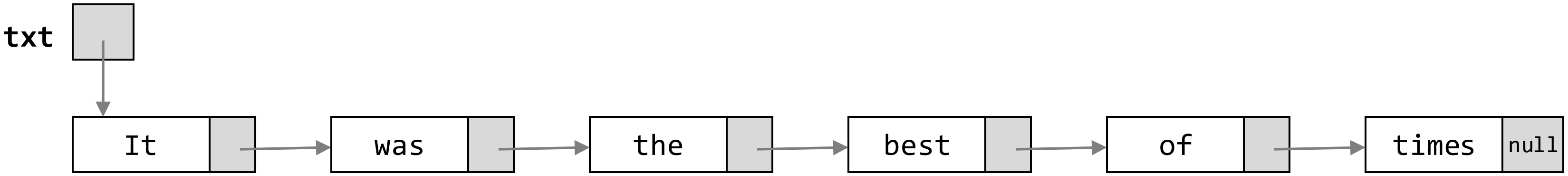
Linked list implementation



dynamic length

- The elements are stored using *nodes*; each node has a value, and a link to the next node
- Adding / removing an element in a list of size n: $\boldsymbol{O(1)}$
- Important data structure that comes to play in numerous applications.

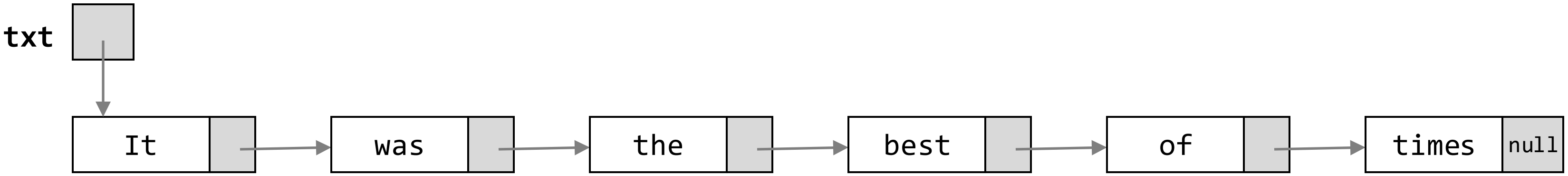
List application example: Word processing



user's view:

```
It was the best of times
```

List application example: Word processing



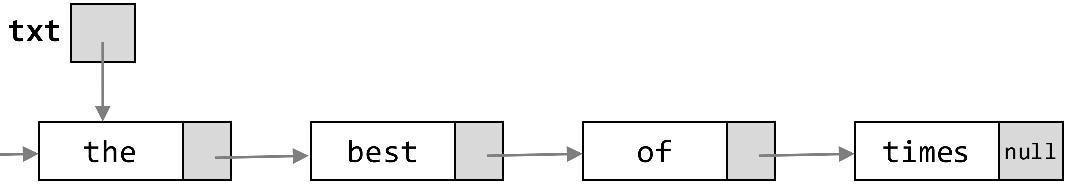
txt

It was the best of times null

user's view:

`It was the best of times`

Remove the first two words:



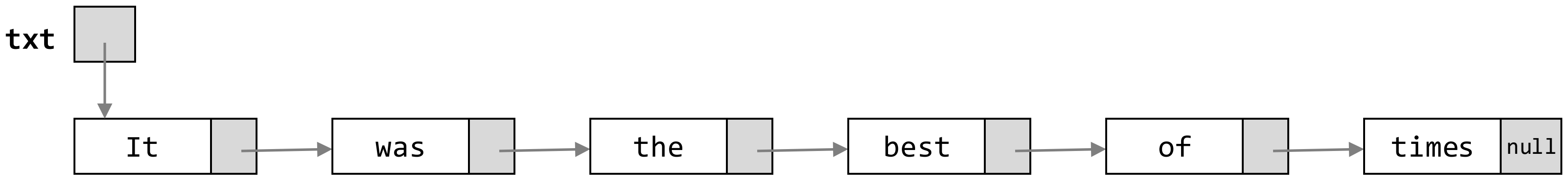
txt

It was the best of times null

user's view:

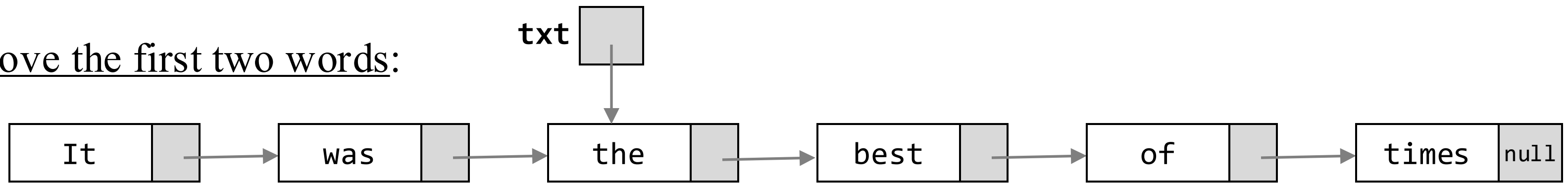
`the best of times`

List application example: Word processing



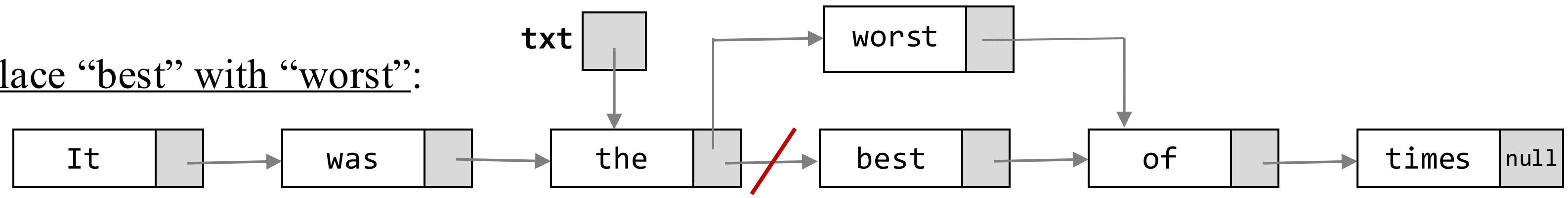
user’s view: `It was the best of times`

Remove the first two words:



user’s view: `the best of times`

Replace “best” with “worst”:



user’s view: `the worst of times`

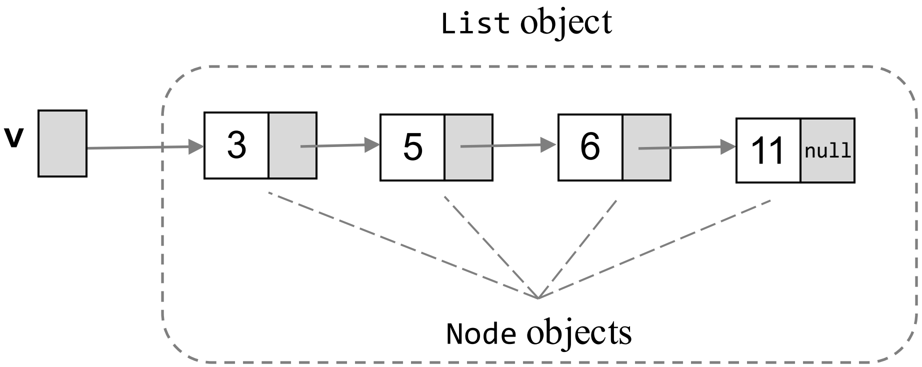
Linked lists: Lecture plan

- Motivation

➡ Architecture

- List operations

Linked lists: Architecture



Based on three classes:

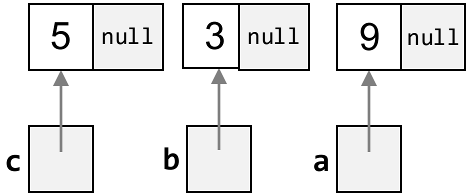
• `Node:`	represents an individual node
• `List:`	represents a list of nodes
• `ListIterator:`	helps process lists (*later*)

Node class: Abstraction

API

```
/** Represents a node in a linked list, containing an integer.
 *  A node has an int value, and a pointer to another node. */
public class Node {

    /** Constructs a node with the given value.
     *  The new node will point to the next node. */
    public Node(int value, Node next)

    /** Constructs a node with the given value.
     *  The new node will point to null. */
    public Node(int value)

    /** Textual representation of this node. */
    public String toString()
}
```

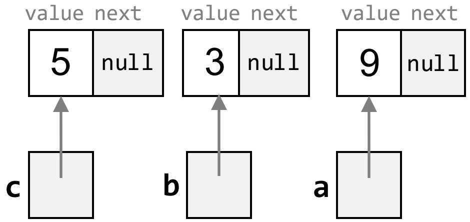
```
// Some class (client code)
class Foo {
  ...
  public static bar() {
           // Builds a list
    Node a = new Node(9);
    Node b = new Node(3);
    Node c = new Node(5);
           ...
```



Node class: Implementation

```
/** Represents a node in a linked list, containing an integer.
 *  A node has an int value, and a pointer to another node. */
public class Node {

   int value;   // data
   Node next;   // pointer

   /** Constructs a node with the given value.
    *  The new node will point to the next node. */
   public Node(int value, Node next) {
      this.value = value;
      this.next = next;
   }

   /** Constructs a node with the given value.
    *  The new node will point to null. */
   public Node(int value) {
      // Calls the other constructor, with next = null
      this(value, null);
   }

   /** Textual representation of this node. */
   public String toString() {
      return "" + value;
   }
}
```

```
// Some class (client code)
class Foo {
  ...
  public static bar() {
           // Builds a list
    Node a = new Node(9);
    Node b = new Node(3);
    Node c = new Node(5);
        ...
```



Node class: Implementation

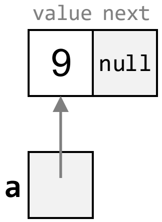
```
/** Represents a node in a linked list, containing an integer.
 *  A node has an int value, and a pointer to another node. */
public class Node {

   int value;   // data
   Node next;   // pointer

   /** Constructs a node with the given value.
    *  The new node will point to the next node. */
   public Node(int value, Node next) {
      this.value = value;
      this.next = next;
   }

   /** Constructs a node with the given value.
    *  The new node will point to null. */
   public Node(int value) {
      // Calls the other constructor, with next = null
      this(value, null);
   }

   /** Textual representation of this node. */
   public String toString() {
      return "" + value;
   }
}
```

```
// Some class (client code)
class Foo {
 ...
 public static bar() {
          // Builds a list
   Node a = new Node(9);
   Node b = new Node(3, a);
   Node c = new Node(5, b);
        ...
```

Node class: Implementation

```
/** Represents a node in a linked list, containing an integer.
 *  A node has an int value, and a pointer to another node. */
public class Node {

   int value;   // data
   Node next;   // pointer

   /** Constructs a node with the given value.
    *  The new node will point to the next node. */
   public Node(int value, Node next) {
      this.value = value;
      this.next = next;
   }

   /** Constructs a node with the given value.
    *  The new node will point to null. */
   public Node(int value) {
      // Calls the other constructor, with next = null
      this(value, null);
   }

   /** Textual representation of this node. */
   public String toString() {
      return "" + value;
   }
}
```

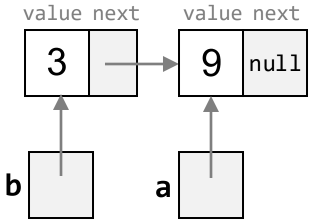
```
// Some class (client code)
class Foo {
  ...
  public static bar() {
          // Builds a list
    Node a = new Node(9);
    Node b = new Node(3, a);
    Node c = new Node(5, b);
         ...
```



Node class: Implementation

```
/** Represents a node in a linked list, containing an integer.
 *  A node has an int value, and a pointer to another node. */
public class Node {

   int value;   // data
   Node next;   // pointer

   /** Constructs a node with the given value.
    *  The new node will point to the next node. */
   public Node(int value, Node next) {
      this.value = value;
      this.next = next;
   }

   /** Constructs a node with the given value.
    *  The new node will point to null. */
   public Node(int value) {
      // Calls the other constructor, with next = null
      this(value, null);
   }

   /** Textual representation of this node. */
   public String toString() {
      return "" + value;
   }
}
```

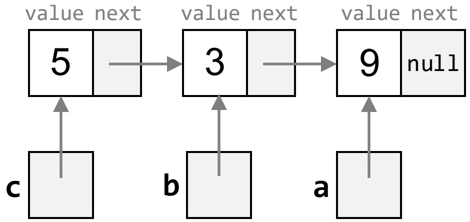
```
// Some class (client code)
class Foo {
  ...
  public static bar() {
           // Builds a list
    Node a = new Node(9);
    Node b = new Node(3, a);
    Node c = new Node(5, b);
         ...
```



Node class: Implementation

```
/** Represents a node in a linked list, containing an integer.
 *  A node has an int value, and a pointer to another node. */
public class Node {

   int value;   // data
   Node next;   // pointer

   /** Constructs a node with the given value.
    *  The new node will point to the next node. */
   public Node(int value, Node next) {
      this.value = value;
      this.next = next;
   }

   /** Constructs a node with the given value.
    *  The new node will point to null. */
   public Node(int value) {
      // Calls the other constructor, with next = null
      this(value, null);
   }

   /** Textual representation of this node. */
   public String toString() {
      return "" + value;
   }
}
```

```
// Some class (client code)
class Foo {
  ...
  public static bar() {
          // Builds a list
    Node a = new Node(9);
    Node b = new Node(3, a);
    Node c = new Node(5, b);
          ...
```



- We created a list, starting at pointer c.
- But, the list construction was messy and unsafe, and so will be its processing
- Solution: Create and use a separate `List` class.

List class: Demo

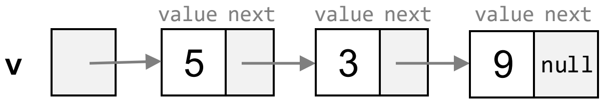
API

```
/** A linked list of integer values. */
public class List {

    /** Constructs an empty list. */
    public List()

    /** Returns a string representation of this list,
     *  in the form (e1, e2, e3, ...) */
    public String toString()

    /** Adds the given value to the end of this list. */
    public void add(int val)

    /** Adds the given value at location i of this list. */
    public void add(int i, int val)

    ...
    /** Returns the location of the given value in this list. */
    public int indexOf(int val)

    /** Returns the value at location i of this list. */
    public int valueAt(int i)

    /** Removes the element at location i from this list. */
    public boolean remove(int i)
    ...
    /** Returns an iterator over the elements in this list,
     *  starting at the first element of the list. */
    public ListIterator listIterator()
}
```

```
// Some class (client code)
class Foo {
  ...
  public static bar() {
    ...
          // Builds a list
    List v = new List();

    v.add(5); v.add(3); v.add(9);

    System.out.println(v);
          ...
    }
    ...
}
```

```
// Output
(5 3 9)
```

Client view:



Linked lists: Lecture plan

Motivation

Architecture

List operations:

➡ Constructing

- Iterating
- Adding elements
- Removing elements
- List iterator

Constructing a list

API

```
/** A linked list of integer values. */
public class List {

    /** Constructs an empty list. */
    public List()

    /** Returns a string representation of this list,
     *  in the form (e1, e2, e3, ...) */
    public String toString()

    /** Adds the given value to the end of this list. */
    public void add(int val)

    /** Adds the given value at location i of this list. */
    public void add(int i, int val)
    ...
    /** Returns the location of the given value in this list. */
    public int indexOf(int val)

    /** Returns the value at location i of this list. */
    public int valueAt(int i)

    /** Removes the element at location i from this list. */
    public boolean remove(int i)
    ...
    /** Returns an iterator over the elements in this list,
     *  starting at the first element of the list. */
    public ListIterator listIterator()
}
```

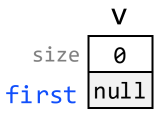
```
// Some class (client code)
class Foo {
  ...
  public static bar() {
    ...
          // Builds a list
          v = new List();
     v.add(5); v.add(3); v.add(9);
     System.out.println(v);
           ...
    }
    ...
}
```

Constructing a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    /** Constructs an empty list */
    public List() {
        first = null;
        size = 0;
    }

    ...

    /** Adds the given value to the end of this list. */
  public void add(int val) {
    // To be discussed later
  }


    ...
}
```

```
// Some class (client code)
class Foo {
  ...
  public static bar() {
    ...
        // Builds a list
        v = new List();
    v.add(5); v.add(3); v.add(9);
    System.out.println(v);
        ...
  }
  ...
}
```

Constructing an empty list

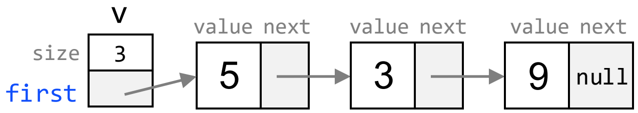


Constructing a list

```
/** A linked list of integer values. */

    private Node first;
    private int size;

    /** Constructs an empty list */
    public List() {
        first = null;
        size = 0;
    }

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
    // To be discussed later
  }


    ...
}
```

```
// Some class (client code)
class Foo {
  ...
  public static bar() {
    ...
          // Builds a list
          v = new List();
    v.add(5); v.add(3); v.add(9);
    System.out.println(v);
          ...
   }
   ...
}
```

Adding elements



(Implementation: Later)

Linked lists: Lecture plan

Motivation

Architecture

List operations:

- Constructing

➡ Iterating

- Adding elements
- Removing elements
- List iterator

Iterating over a list

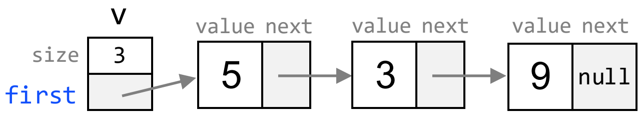
API

```
/** A linked list of integer values. */
public class List {

    /** Constructs an empty list. */
    public List()

    /** Returns a string representation of this list,
     *  in the form (e1, e2, e3, ...) */
    public String toString()

    /** Adds the given value to the end of this list. */
    public void add(int val)

    /** Adds the given value at location i of this list. */
    public void add(int i, int val)
    ...
    /** Returns the location of the given value in this list. */
    public int indexOf(int val)

    /** Returns the value at location i of this list. */
    public int valueAt(int i)

    /** Removes the element at location i from this list. */
    public boolean remove(int i)
    ...
    /** Returns an iterator over the elements in this list,
     *  starting at the first element of the list. */
    public ListIterator listIterator()
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```

Example: toString)

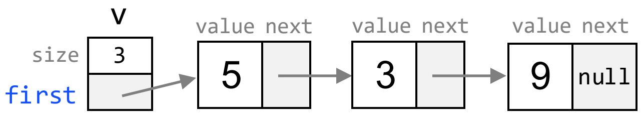


Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0, str.length() - 1) + ")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```

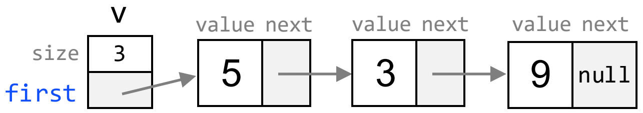


Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0, str.length() - 1) + ")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```

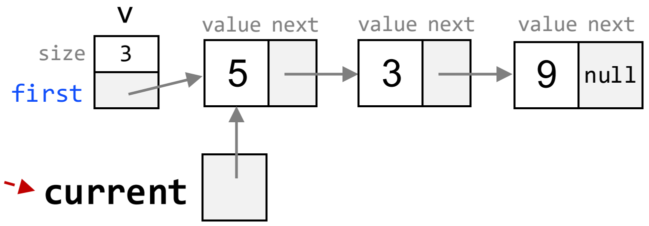


Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0,str.length()-1)+")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



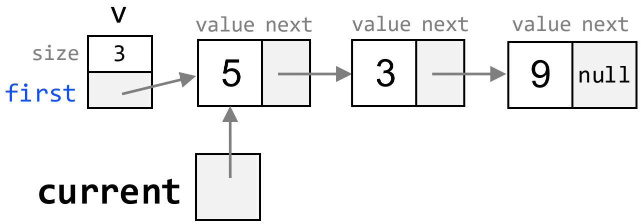
str = (

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ‘)’
        return str.substring(0, str.length() - 1) + ")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



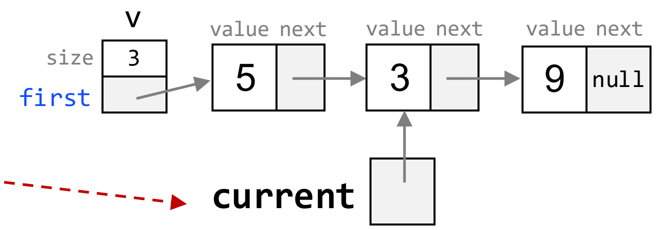
str = (5

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0, str.length() - 1) + ")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



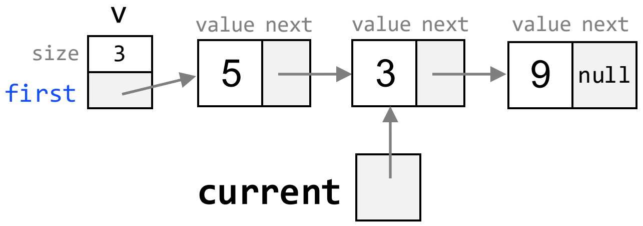
```
str = (5
```

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0, str.length() - 1) + ")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



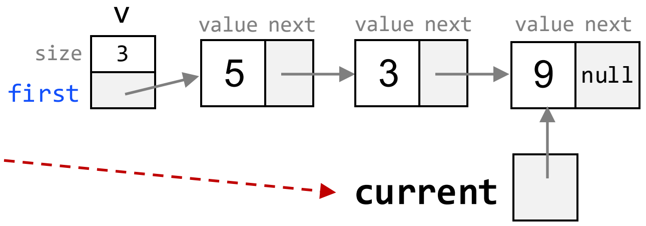
str = (5 3

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0,str.length()-1)+")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



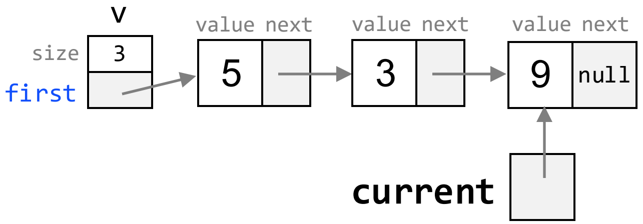
str = (5 3

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ')'
        return str.substring(0, str.length() - 1) + ")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



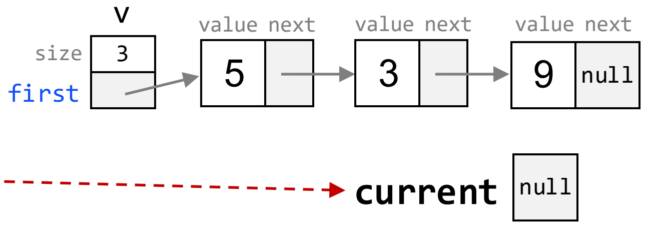
str = (5 3 9

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ‘)’
        return str.substring(0,str.length()-1)+")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



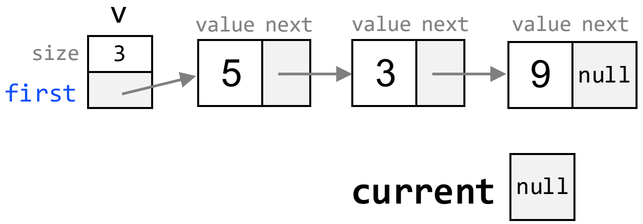
str = (5 3 9

Iterating over a list

```
/** A linked list of integer values. */

    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ‘)’
        return str.substring(0,str.length()-1)+")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



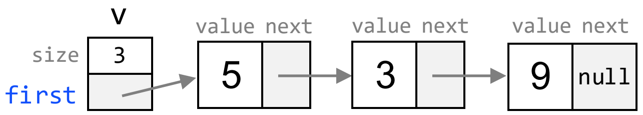
str = (5 3 9)

Iterating over a list

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns a string representation of this list. */
    public String toString() {
        if (size == 0) return "()";
        // Starting from the first node, iterates through this list
        // and builds the string incrementally
        String str = "(";
        Node current = first;
        while (current != null) {
            str += current.value + " ";
            current = current.next;
        }
        // Removes the trailing space and adds the ‘)’
        return str.substring(0,str.length()-1)+")";
    }
    ...
}
```

```
// Client code: Builds a small list and prints it
List v = new List();
v.add(5); v.add(3); v.add(9);
System.out.println(v.toString());
...
```

```
// Output
(5 3 9)
```



str = (5 3 9)

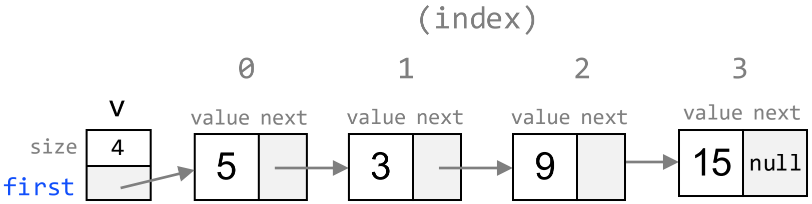
(current disappeared because it is a local variable)

Iterating over a list: indexOf / valueAt

API

```
/** A linked list of integer values. */
public class List {

    /** Constructs an empty list. */
    public List()

    /** Returns a string representation of this list,
     *  in the form (e1, e2, e3, ...) */
    public String toString()

    /** Adds the given value to the end of this list. */
    public void add(int val)

    ...

    /** Returns the location of the given value in this list. */
    public int indexOf(int val)

    /** Returns the value at location i of this list. */
    public int valueAt(int i)

    ...
}
```

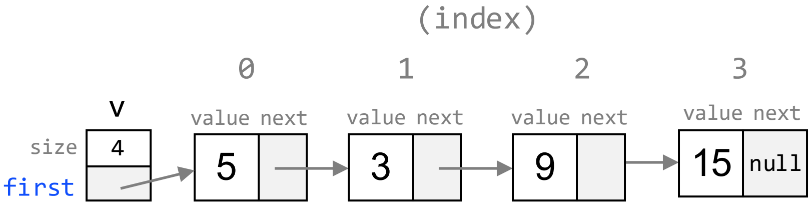
```
// Client code: Built a list (code omitted)...
...
v.indexOf(9); // 2
...
v.valueAt(1); // 3
...
```



Iterating over a list: indexOf / valueAt

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns the location of the given value in this list,
     *  or -1 if not found. */
    public int indexOf(int val) {
        Node current = first;
        int index = 0;
        while (current != null) {
            if (current.value == val) {
                return index;
            }
            current = current.next;
            index++;
        }
        return -1;  // Value not found
    }

    /** Returns the value at the given location in this list.
     *  If the index is invalid, throws an exception. */
    public int valueAt(int i) {
        // Similar
    }
    ...
}
```

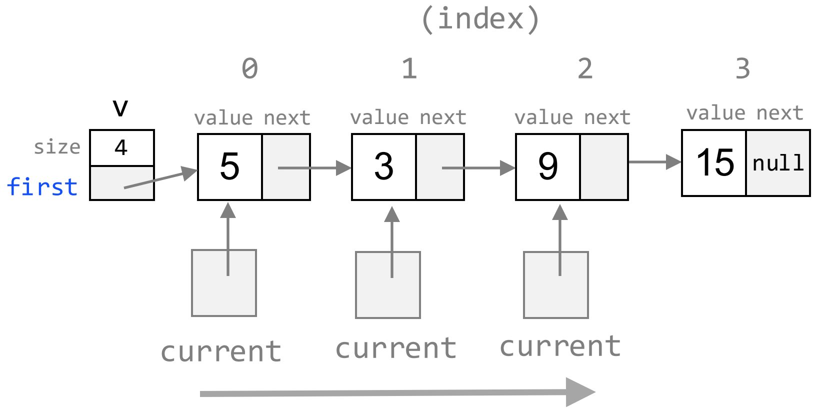
```
// Client code: Built a list (code omitted)...
...
v.indexOf(9); // 2
...
v.valueAt(1); // 3
...
```



Iterating over a list: indexOf / valueAt

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns the location of the given value in this list,
     *  or -1 if not found. */
    public int indexOf(int val) {
        Node current = first;
        int index = 0;
        while (current != null) {
            if (current.value == val) {
                return index;
            }
            current = current.next;
            index++;
        }
        return -1;  // Value not found
    }

    /** Returns the value at the given location in this list.
     *  If the index is invalid, throws an exception. */
    public int valueAt(int i) {
        // Similar
    }
    ...
}
```

```
// Client code: Built a list (code omitted)...
...
v.indexOf(9); // 2
...
v.valueAt(1); // 3
...
```



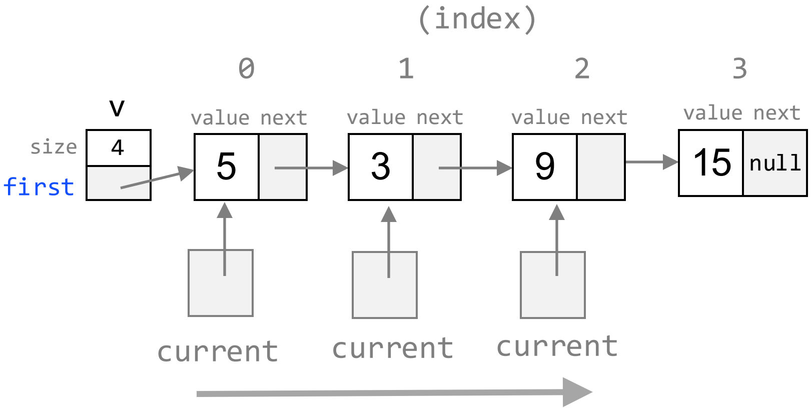
Processing logic:

Iteration, similar to `toString()`.

Iterating over a list: indexOf / valueAt

```
/** A linked list of integer values. */
    private Node first;
    private int size;

    ...

    /** Returns the location of the given value in this list,
     *  or -1 if not found. */
    public int indexOf(int val) {
        Node current = first;
        int index = 0;
        while (current != null) {
            if (current.value == val) {
                return index;
            }
            current = current.next;
            index++;
        }
        return -1;  // Value not found
    }

    /** Returns the value at the given location in this list.
     *  If the index is invalid, throws an exception. */
    public int valueAt(int i) {
        // Similar
    }
    ...
}
```

```
// Client code: Built a list (code omitted)...
...
v.indexOf(9); // 2
...
v.valueAt(1); // 3
...
```



Processing logic:

Iteration, similar to `toString()`.

Linked lists: Lecture plan

Motivation

Architecture

List operations:

- Constructing
- Iterating

➡ Adding elements

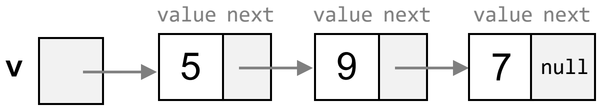
- Removing elements
- List iterator

Adding elements (to the list's end)

API

```
/** A linked list of integer values. */
public class List {

    /** Constructs an empty list. */
    public List()

    /** Returns a string representation of this list,
     *  in the form (e1, e2, e3, ...) */
    public String toString()

    /** Adds the given value to the end of this list. */
    public void add(int val)

    /** Adds the given value at location i of this list. */
    public void add(int i, int val)

    /** Adds the given value to the beginning of this list. */
    public void addFirst(int i, int val)

    ...
}
```

```
// Client code: Builds a list and adds some elements
List v = new List();
v.add(5); v.add(9); v.add(7);
...
```

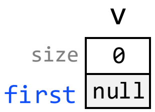


Adding elements (to the list’s end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
  public void add(int val) {
   // Creates a new node with the given value
   Node newNode = new Node(val);
   // If the list is empty, the new node becomes the first node
   if (first == null) {
      first = newNode;
   } else {
      // Iterates to the last node
      Node current = first;
      while (current.next != null) {
         current = current.next;
      }
      // Points the last node to the new node
      current.next = newNode;
   }
   size++;
  }

    ...
}
```

```
// Client code: Builds a list and adds some elements

List v = new List();

v.add(5); v.add(9); v.add(7);

...
```

If the list is empty:

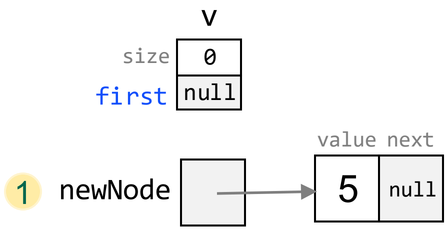


Adding elements (to the list's end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
    // Creates a new node with the given value
    No 1 newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
       first = newNode;
    } else {
       // Iterates to the last node
       Node current = first;
       while (current.next != null) {
          current = current.next;
       }
       // Points the last node to the new node
       current.next = newNode;
    }
    size++;
  }

    ...
}
```

```
// Client code: Builds a list and adds some elements
List v = new List();
v.add(5); v.add(9); v.add(7);
...
```

If the list is empty:

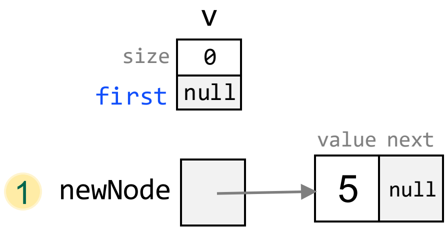


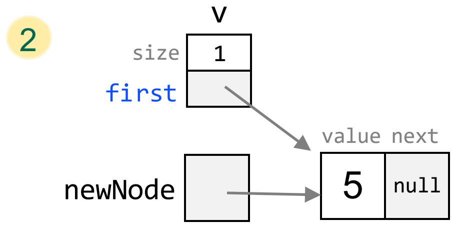
Adding elements (to the list’s end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
    // Creates a new node with the given value
    No 1 newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
       first = n 2 Node;
    } else {
       // Iterates to the last node
       Node current = first;
       while (current.next != null) {
          current = current.next;
       }
       // Points the last node to the new node
       current.next = newNode;
    }
    size++;
  }

    ...
}
```

```
// Client code: Builds a list and adds some elements
List v = new List();
v.add(5); v.add(9); v.add(7);
...
```

If the list is empty:



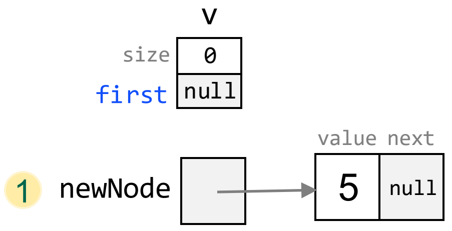


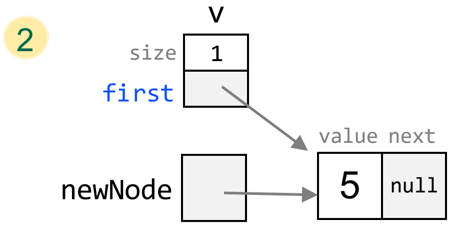
Adding elements (to the list's end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
      // Creates a new node with the given value
      No 1 newNode = new Node(val);
      // If the list is empty, the new node becomes the first node
      if (first == null) {
         first = n 2 Node;
      } else {
         // Iterates to the last node
         Node current = first;
         while (current.next != null) {
            current = current.next;
         }
         // Points the last node to the new node
         current.next = newNode;
      }
      size++;
   }

    ...
}
```

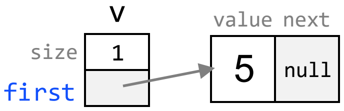
```
// Client code: Builds a list and adds some elements

List v = new List();

v.add(5); v.add(9); v.add(7);

...
```

If the list is empty:





Final result (adter Add terminates):

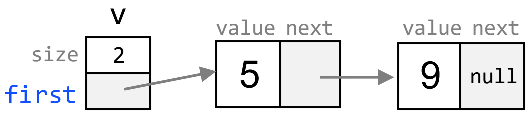


Adding elements (to the list's end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
    // Creates a new node with the given value
    Node newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
      first = newNode;
    } else {
      // Iterates to the last node
      Node current = first;
      while (current.next != null) {
         current = current.next;
      }
      // Points the last node to the new node
      current.next = newNode;
    }
    size++;
  }

    ...
}
```

```
// Client code: Builds a list and adds some elements

List v = new List();

v.add(5); v.add(9); v.add(7);

...
```

If the list is not empty:

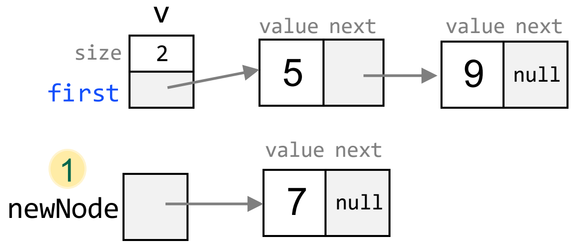


Adding elements (to the list’s end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
    // Creates a new node with the given value
    No 1 newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
       first = newNode;
    } else {
       // Iterates to the last node
       Node current = first;
       while (current.next != null) {
          current = current.next;
       }
       // Points the last node to the new node
       current.next = newNode;
    }
    size++;
  }

    ...
}
```

```
// Client code: Builds a list and adds some elements

List v = new List();

v.add(5); v.add(9); v.add(7);

...
```

If the list is not empty:

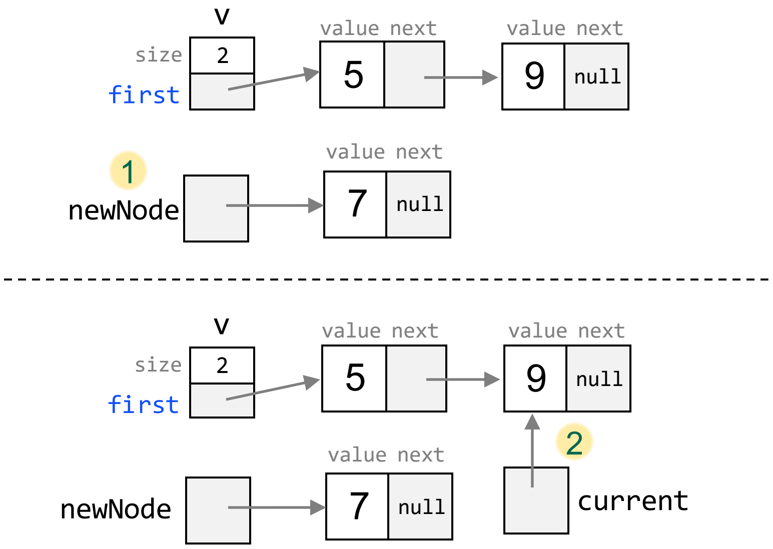


Adding elements (to the list's end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
   ublic void add(int val) {
    // Creates a new node with the given value
    No 1 newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
       first = newNode;
    } else {
       // Iterates to the last node
       Node current = first;
       while (( 2 ent.next != null) {
          current = current.next;
       }
       // Points the last node to the new node
       current.next = newNode;
    }
    size++;
  }

    ...
}
```

```
// Client code: Builds a list and adds some elements

List v = new List();

v.add(5); v.add(9); v.add(7);

...
```

If the list is not empty:

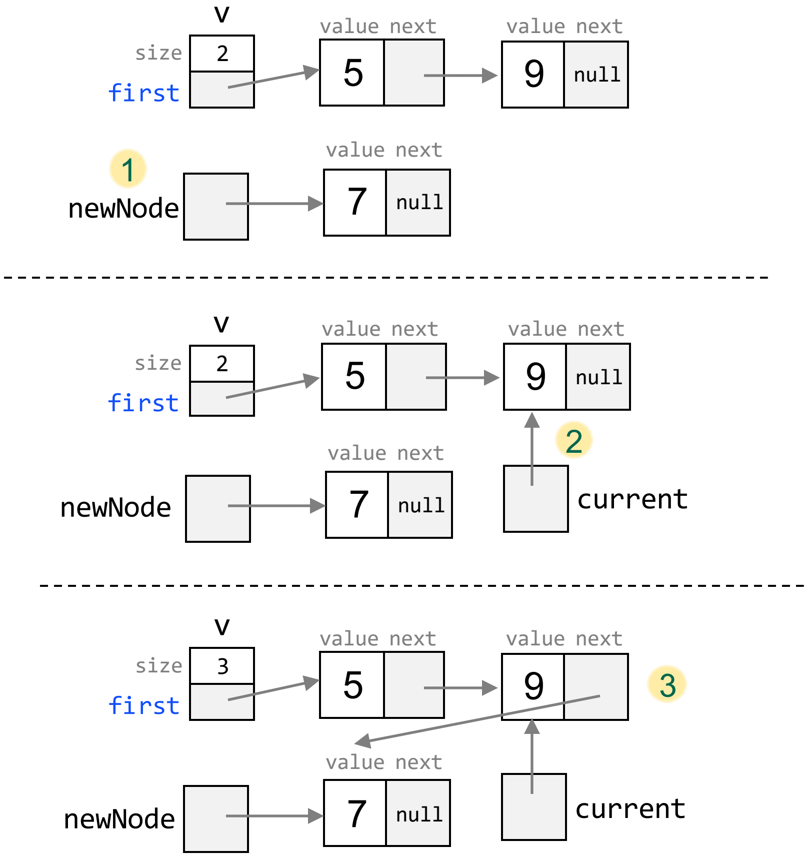


Adding elements (to the list’s end)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Adds the given value to the end of this list. */
  public void add(int val) {
    // Creates a new node with the given value
    Node newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
       first = newNode;
    } else {
       // Iterates to the last node
       Node current = first;
       while (current.next != null) {
          current = current.next;
       }
       // Points the last node to the new node
       current.next = newNode;
    }
    size++;
  }

    ...

}
```

```
// Client code: Builds a list and adds some elements

List v = new List();

v.add(5); v.add(9); v.add(7);

...
```

If the list is not empty:

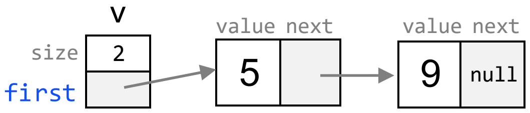


Adding elements (to the list's end)

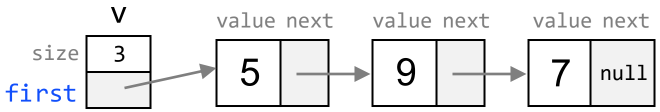
```
/** A linked list of integer values. */
public class List {
    private Node first;
    private int size;
    ...

    /** Adds the given value to the end of this list. */
  public void add(int val) {
    // Creates a new node with the given value
    Node newNode = new Node(val);
    // If the list is empty, the new node becomes the first node
    if (first == null) {
      first = newNode;
    } else {
      // Iterates to the last node
      Node current = first;
      while (current.next != null) {
        current = current.next;
      }
      // Points the last node to the new node
      current.next = newNode;
    }
    size++;
  }
    ...
}
```

```
// Client code: Builds a list and adds some elements
List v = new List();
v.add(5); v.add(9); v.add(7);
...
```

If the list is not empty:



Final result (after `Add` terminates):



Adding elements (to the list’s beginning)

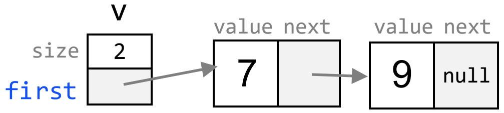
```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    /** Constructs an empty list */
    public List() {
        first = null;
        size = 0;
    }

    ...

    /** Adds the given value to the beginning of this list. */
    public void addFirst(int val) {

        Node newNode = new Node(val); // creates a new node

        newNode.next = first; // new node → first node

        first = newNode;      // first → new node

        size++;

    }

    ...

}
```

```
// Client code: Builds a list and adds some elements
List v = new List();

v.addFirst(9); v.addFirst(7); v.addFirst(5);
```

Adding elements (to the list’s beginning)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    /** Constructs an empty list */
    public List() {
        first = null;
        size = 0;
    }
    ...
    /** Adds the given value to the beginning of this list. */
    public void addFirst(int val) {

        Node newNode = new Node(val); // creates a new node

        newNode.next = first;  // new node → first node

        first = newNode;       // first → new node

        size++;

    }
    ...
}
```

```
// Client code: Builds a list and adds some elements
List v = new List();

v.addFirst(9); v.addFirst(7); v.addFirst(5);
```



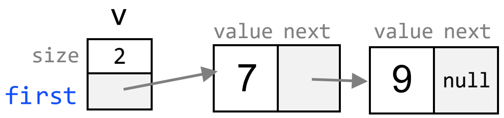
Suppose that we’ve added `9` and `7;`

We’ll track how `5 is added.`

Adding elements (to the list’s beginning)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    /** Constructs an empty list */
    public List() {
        first = null;
        size = 0;
    }

    ...

    /** Adds the given value to the beginning of this list. */
    public void addFirst(int val) {

     1  Node newNode = new Node(val); // creates a new node

     2  newNode.next = first;  // new node → first node

     3  first = newNode;       // first → new node

        size++;

    }

    ...

}
```

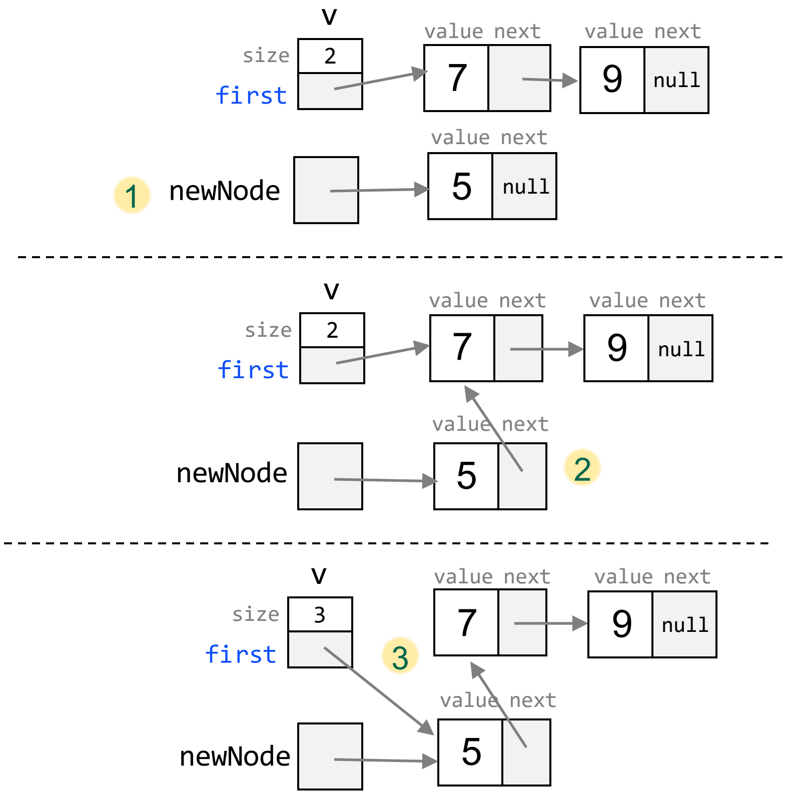
```
// Client code: Builds a list and adds some elements
List v = new List();

v.addFirst(9); v.addFirst(7); v.addFirst(5);
```



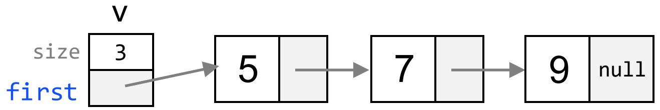
Adding elements (to the list's beginning)

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    /** Constructs an empty list */
    public List() {
        first = null;
        size = 0;
    }

    ...

    /** Adds the given value to the beginning of this list. */
    public void addFirst(int val) {

  1     Node newNode = new Node(val); // creates a new node

  2     newNode.next = first;  // new node → first node

  3     first = newNode;       // first → new node

        size++;

    }

    ...

}
```

```
// Client code: Builds a list and adds some elements
List v = new List();

v.addFirst(9); v.addFirst(7); v.addFirst(5);
```



Final result (after AddFirst terminates):



Linked lists: Lecture plan

Motivation

Architecture

List operations:

- Constructing
- Iterating
- Adding elements

➡ Removing elements

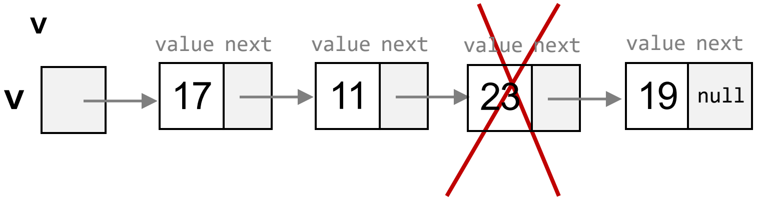
- List iterator

Removing elements

API

```
/** A linked list of integer values. */
public class List {

      /** Constructs an empty list. */
      public List()

      /** Returns a string representation of this list,
       *  in the form (e1, e2, e3, ...) */
      public String toString()

      /** Adds the given value to the end of this list. */
      public void add(int val)

      ...

      /** Removes the first occurrence of the given value from this list,
       *  and returns true; If not found ,returns false.*/
      public boolean removeValue(int val)

      ...

      /** Returns an iterator over the elements in this list,
       * starting at the first element of the list. */
      public ListIterator listIterator()
}
```

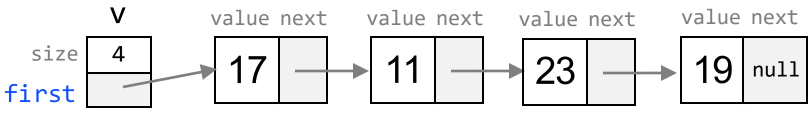
```
// Client code
...
v.removeValue(23);
...
```



Removing elements

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Removes the first occurrence of the given value from this list,
     *  and returns true; If not found ,returns false.*/
    public boolean removeValue(int val) {
        // Finds the node to remove, using two pointers;
        // prev is one step behind current
        Node prev = null;
        Node current = first;
        while (current != null && current.value != val) {
             prev = current;
             current = current.next;
        }
        if (current == null) return false; // not found
        // Remove the elements. If it's the first element, updates first
        if (prev == null) { // it's the first element
             first = first.next;
        }
        else {
             prev.next = current.next;
        }
        size--;
        return true;
    }

    ...
}
```

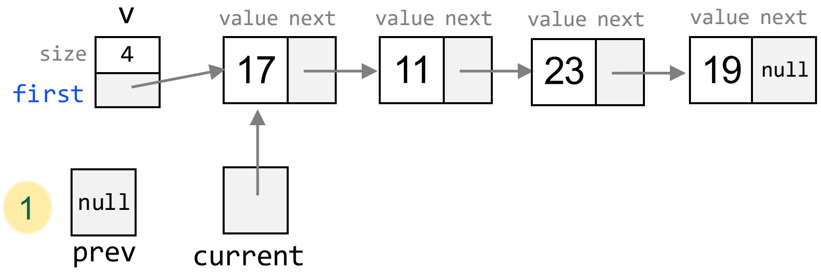
```
// Client code
...
v.removeValue(23);
...
```



Removing elements

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Removes the first occurrence of the given value from this list,
     *  and returns true; If not found ,returns false.*/
    public boolean removeValue(int val) {
       // Finds the node to remove, using two pointers;
       // prev is one step behind current
1      Node prev = null;
       Node current = first;
       while (current != null && current.value != val) {
            prev = current;
            current = current.next;
       }
       if (current == null) return false;  // not found
       // Remove the elements. If it's the first element, updates first
       if (prev == null) {  // it's the first element
            first = first.next;
       }
       else {
            prev.next = current.next;
       }
       size--;
       return true;
    }

    ...
}
```

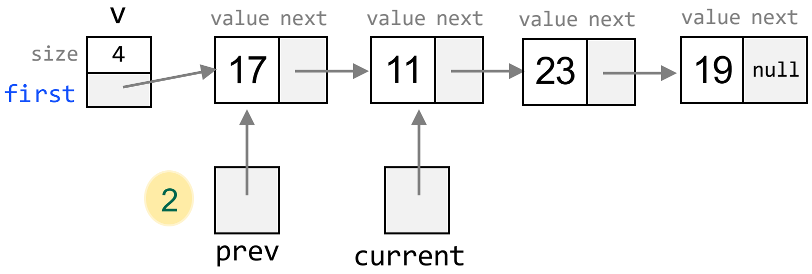
```
// Client code
...
v.removeValue(23);
...
```



Removing elements

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Removes the first occurrence of the given value from this list,
     *  and returns true; If not found ,returns false.*/
    public boolean removeValue(int val) {
       // Finds the node to remove, using two pointers;
       // prev is one step behind current
  1    Node prev = null;
       Node current = first;
       while (current != null && current.value != val) {
            prev = current;
     2      current = current.next;
       }
       if (current == null) return false;  // not found
       // Remove the elements. If it's the first element, updates first
       if (prev == null) {  // it's the first element
            first = first.next;
       }
       else {
            prev.next = current.next;
       }
       size--;
       return true;
    }

    ...
}
```

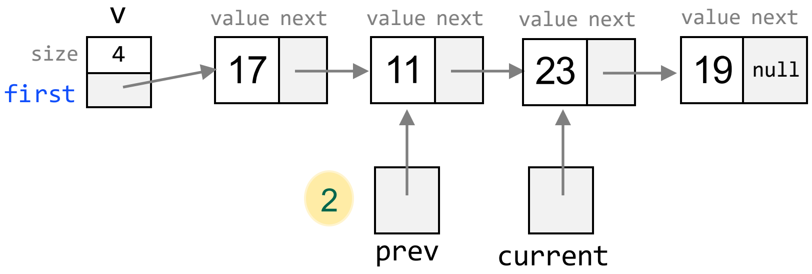
```
// Client code
...
v.removeValue(23);
...
```



Removing elements

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Removes the first occurrence of the given value from this list,
     *  and returns true; If not found ,returns false.*/
    public boolean removeValue(int val) {
        // Finds the node to remove, using two pointers;
        // prev is one step behind current
 1      Node prev = null;
        Node current = first;
        while (current != null && current.value != val) {
            prev = current;
      2     current = current.next;
        }
        if (current == null) return false; // not found
        // Remove the elements. If it's the first element, updates first
        if (prev == null) { // it's the first element
            first = first.next;
        }
        else {
            prev.next = current.next;
        }
        size--;
        return true;
    }

    ...
}
```

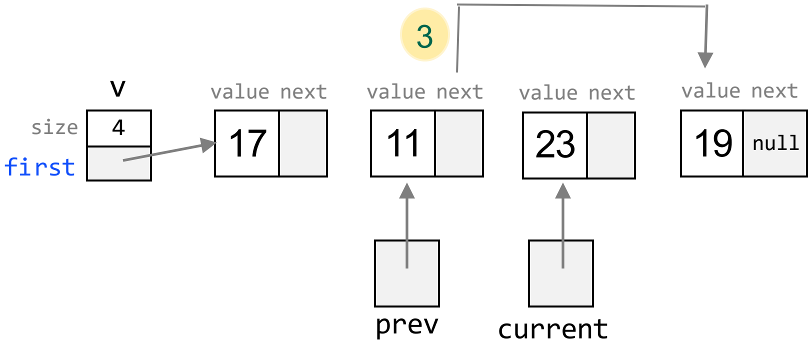
```
// Client code
...
v.removeValue(23);
...
```



Removing elements

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Removes the first occurrence of the given value from this list,
     *  and returns true; If not found ,returns false.*/
    public boolean removeValue(int val) {
        // Finds the node to remove, using two pointers;
        // prev is one step behind current
 1      Node prev = null;
        Node current = first;
        while (current != null && current.value != val) {
             prev = current;
     2       current = current.next;
        }
        if (current == null) return false;  // not found
        // Remove the elements. If it's the first element, updates first
        if (prev == null) {  // it's the first element
             first = first.next;
        }
        else {
             prev.next = current.next;   3
        }
        size--;
        return true;
    }

    ...
}
```

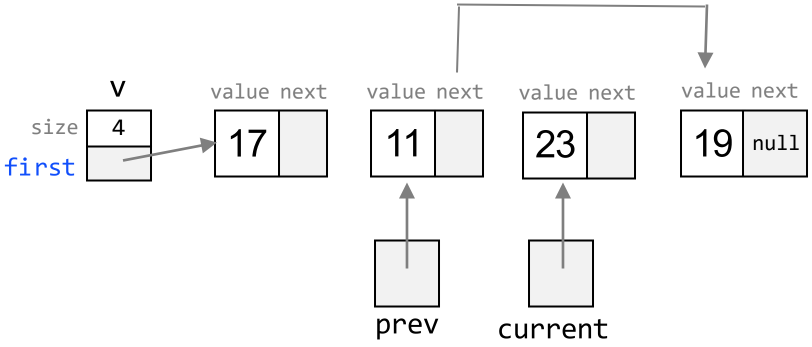
```
// Client code
...
v.removeValue(23);
...
```



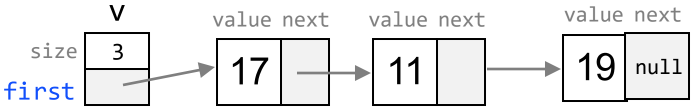
Removing elements

```
/** A linked list of integer values. */
public class List {

    private Node first;
    private int size;

    ...

    /** Removes the first occurrence of the given value from this list,
     *  and returns true; If not found ,returns false.*/
    public boolean removeValue(int val) {
       // Finds the node to remove, using two pointers;
       // prev is one step behind current
       Node prev = null;
       Node current = first;
       while (current != null && current.value != val) {
            prev = current;
            current = current.next;
       }
       if (current == null) return false;  // not found
       // Remove the elements. If it's the first element, updates first
       if (prev == null) {  // it's the first element
            first = first.next;
       }
       else {
            prev.next = current.next;
       }
       size--;
       return true;
    }

    ...
}
```

```
// Client code
...
v.removeValue(23);
...
```



Final result (after removeValue terminates):



Linked lists: Lecture plan

Motivation

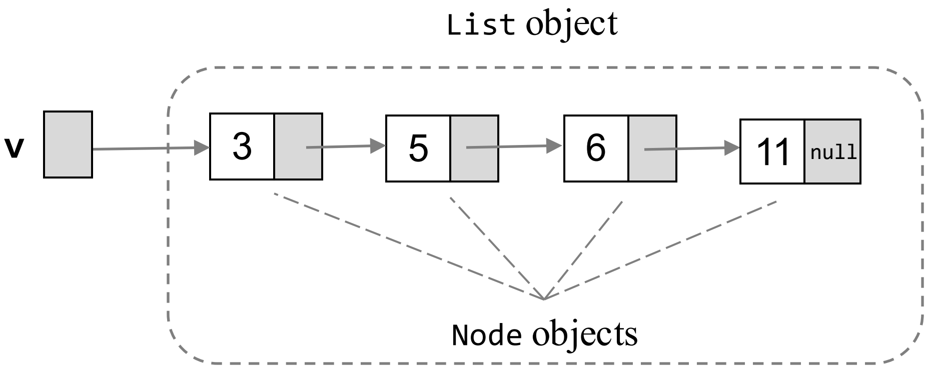
Architecture

List operations:

- Constructing
- Iterating
- Adding elements
- Removing elements

➡ List iterator

List Iterator



Based on three classes:

- `Node:` represents an individual node
- `List:` represents a list of nodes
- `ListIterator:` helps process lists

An *iterator* is an object that provides iteration services through a sequence of elements.

List class: Abstraction

API

```
 /** A linked list of integer values. */
public class List {

    /** Constructs an empty list. */
    public List()

    /** Returns a string representation of this list,
     *  in the form (e1, e2, e3, ...) */
    public String toString()

    /** Adds the given value to the end of this list. */
    public void add(int val)

    /** Adds the given value at location i of this list. */
    public void add(int i, int val)
    ...
    /** Returns the location of the given value in this list. */
    public int indexOf(int val)

    /** Returns the value at location i of this list. */
    public int valueAt(int i)

    /** Removes the element at location i from this list. */
    public boolean remove(int i)
    ...
    /** Returns an iterator over the elements in this list,
     *  starting at the first element of the list. */
    public ListIterator listIterator()
}
```

<u>Design challenge</u>

Clients (programs that do list processing) often want to manipulate lists in ways that the API cannot anticipate

How can we allow clients to iterate lists safely, simply, and generally?

<u>Solution</u>

We'll provide an *iterator* service that does just that.

List iterator

```
/** Implements an iteration over the elements of a List. */
public class ListIterator {                              API

    /** Returns an iterator, starting at the given node. */
    public ListIterator(Node first)

    /** Checks if this iteration has more elements. */
    public boolean hasNext

    /** Returns the value of the current element in this iteration,
     *  and advances the iteration. Should be called only if
     * hasNext() is true.  If hasNext() is not true, throws an
     * exception. */
    public int next()
}
```

The name of the `next()` method can be confusing. It does not return the `next` field of a `Node` object. Rather, it does exactly what its documentation says.

An accurate method name could have been "`getValueAndAdvance`".

We didn't change the method's name since the `next()` name is commonly used in ietarators, and we want to use standard terminology.

An *iterator* is an object that provides iteration services through a list.

```
// Client code:
List v = new List();
v.add(1); v.add(2); v.add(4); v.add(9);
System.out.println(v);

// Uses an iterator to sum up the list values.
ListIterator itr = v.listIterator();
int sum = 0;
while (itr.hasNext()) {
    sum = sum + itr.next();
}
System.out.println("Sum:  " + sum);
```

```
// Output
(1 2 4 9)
Sum: 16
```

Iterator

Provides a standard way to iterate lists without giving access to the list pointers (which makes it a safe technique to process lists).

List iterator

```
import java.util.NoSuchElementException;

/** Implements an iteration over the elements of a List. */
public class ListIterator {

    private Node current;  // current location of the iteration

     /** Returns an iterator, starting at the given node. */
    public ListIterator(Node first) {
        this.current = first;
    }

     /** Checks if this iteration has more elements. */
    public boolean hasNext() {
        return current != null;
    }

    /** Returns the next element in this iteration, and advances
     *   the iteration. Should be called only if hasNext() is true.
     *  If hasNext() is not true, throws an exception. */
    public int next() {
        if (!hasNext()) {
            throw new NoSuchElementException;
        }
        int value = current.value;
        // Advances this iteration to the next element
        current = current.next;
        return value;
    }
}
```

An *iterator* is an object that provides iteration services through a list.

```
// Client code:
List v = new List();
v.add(1); v.add(2); v.add(4); v.add(9);
System.out.println(v);

// Uses an iterator to sum up the list values.
ListIterator itr = v.listIterator();
int sum = 0;
while (itr.hasNext()) {
    sum = sum + itr.next();
}
System.out.println("Sum: " + sum);
```

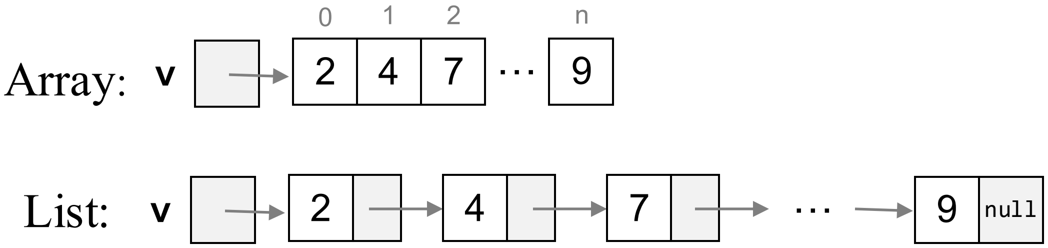
```
// Output
(1 2 4 9)
Sum: 16
```

Observation

Unlike other objects that we saw so far, an iterator is not a data-oriented object, but rather a process-oriented object.

Next lecture

Basic data structures



Abstract Data Structures (ADTs)

Set

Stack

Queue

Map

Tree

...

Typically implemented using arrays or lists