Java Collection Framework

Intro to Data Structures and programming

Definition of collection

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.

- They typically represent data items that form a natural group, e.g.
  - poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).

The Need for Data Structures

Data structures organize data

$\Rightarrow$ more efficient programs.

Any organization for a collection of records can be searched, processed in any order, or modified.

The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be efficient if it solves the problem within its resource constraints.

- Space
- Time

- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Data Structure Philosophy (cont)

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

What is a Data Structure?

A data structure

- is a collection of data organized in some way.
- Supports the operations for manipulating data in the structure.

For example, an array

- is a data structure that holds a collection of data in sequential order.
- You can find the size of the array, store, retrieve, and modify data in the array.

Array is simple and easy to use, but it has two limitations:

Limitations of arrays

- Once an array is created, its size cannot be altered.

- Array provides inadequate support for inserting, deleting, sorting, and searching operations.

Arrays

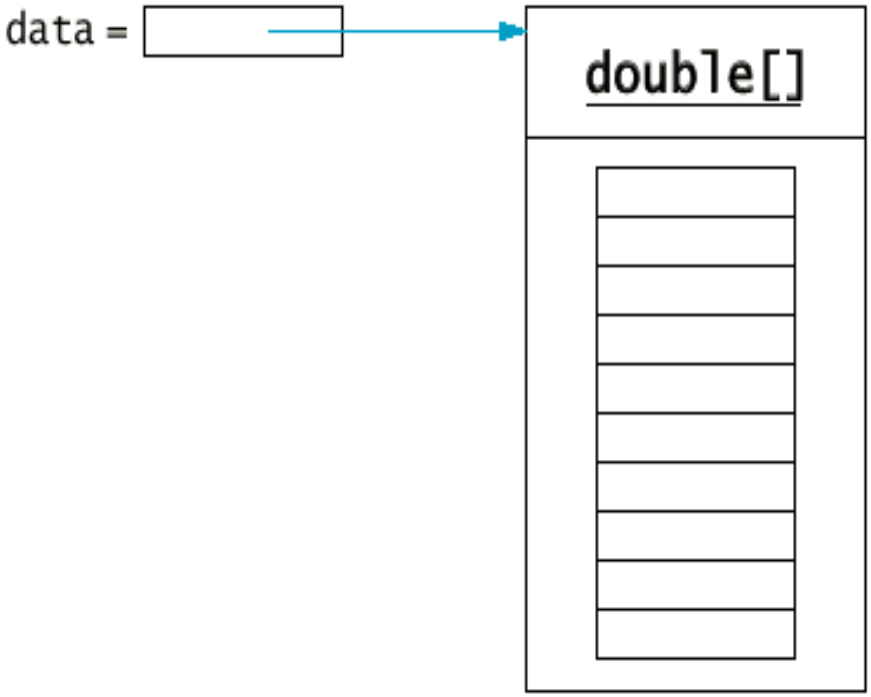
- new is used to construct a new array:

new double[10]

- Store 10 double type variables in an array of doubles

```
double[] data = new double[10];
```

- Use length attribute to get array length.
  - **data.length**
  - (Not a method!)

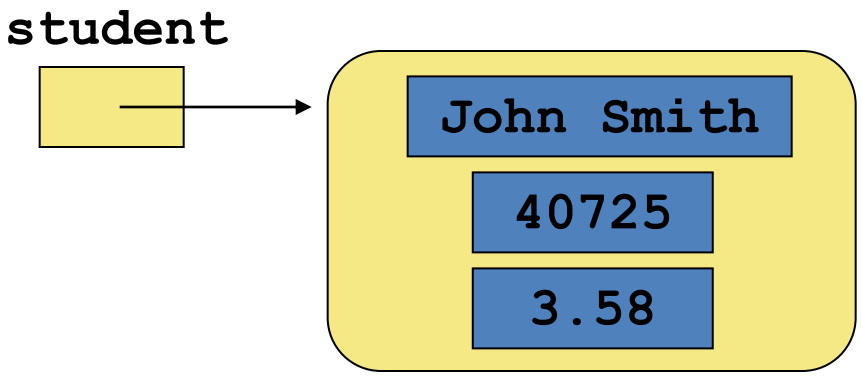


Dynamic Structures

- A *static* data structure has a fixed size
- This meaning is different from the meaning of the `static` modifier
- Arrays are static; once you define the number of elements it can hold, the size doesn't change
- A *dynamic data structure* grows and shrinks at execution time as required by its contents
- A dynamic data structure is implemented using object references as *links*

Object References

- Recall that an *object reference* is a variable that stores the address of an object
- A reference also can be called a *pointer*
- References often are depicted graphically:



References as Links

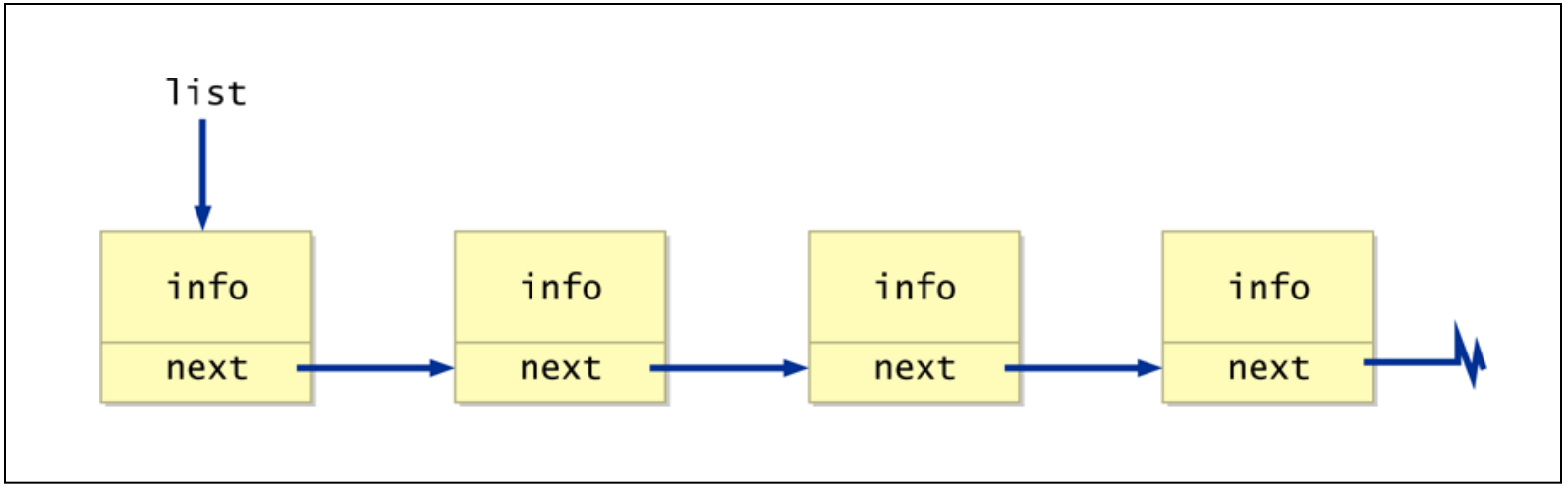
- Object references can be used to create *links* between objects

- Suppose a class contains a reference to another object of the same class:

```
class Node
{
    int info;
    Node next;
}
```

References as Links

- References can be used to create a variety of linked structures, such as a *linked list*:

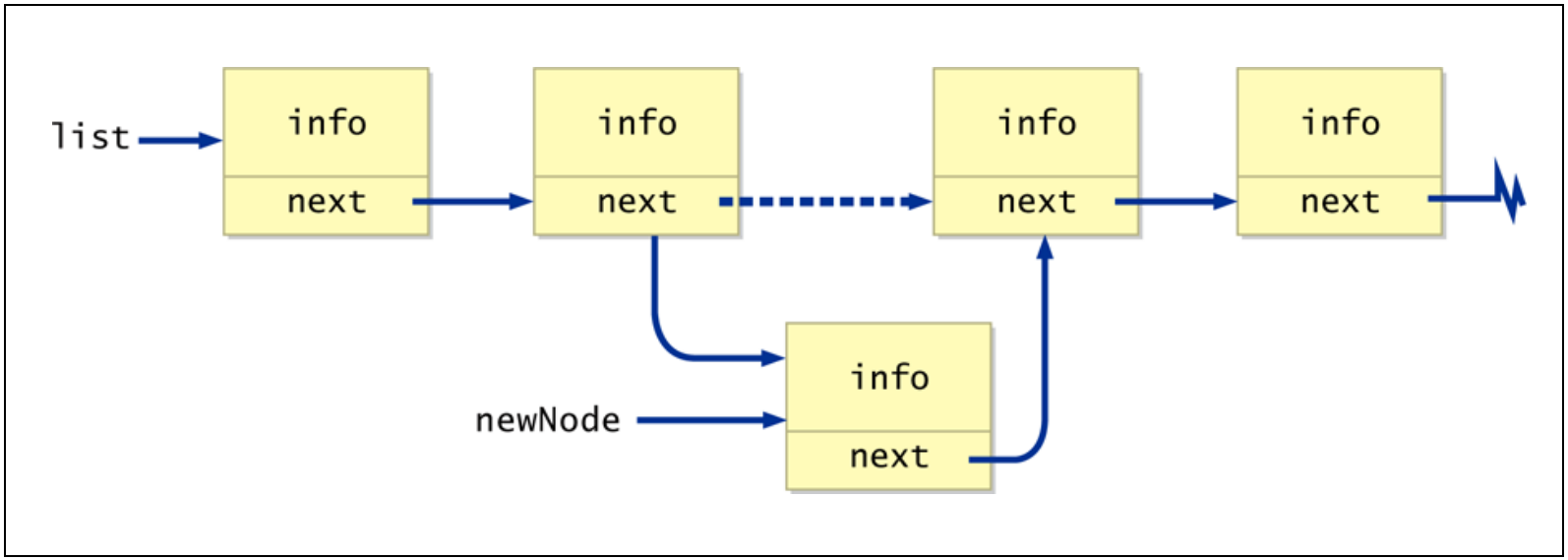


Intermediate Nodes

- The objects being stored should not be concerned with the details of the data structure in which they may be stored
- For example, the `Student` class should not have to store a link to the next `Student` object in the list
- Instead, use a separate node class with two parts:
  - a reference to an independent object
  - a link to the next node in the list
- The internal representation becomes a linked list of nodes

Inserting a Node

- A node can be inserted into a linked list with a few reference changes:



Quick Check

Write code that inserts `newNode` after the node pointed to by `current`.

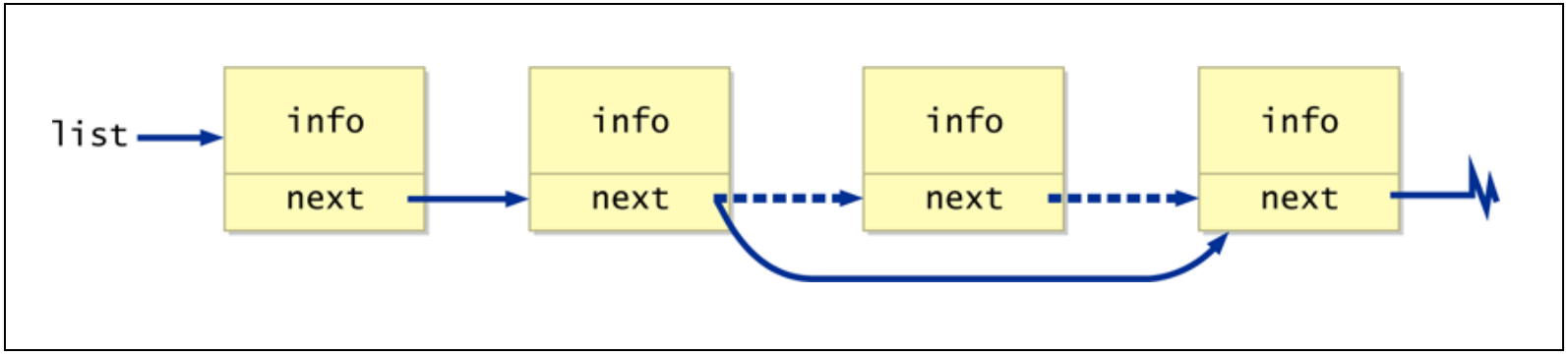
Quick Check

Write code that inserts `newNode` after the node pointed to by `current`.

```
newNode.next = current.next;

current.next = newNode;
```

Deleting a Node

- Likewise, a node can be removed from a linked list by changing the `next` pointer of the preceding node:



What's wrong with Array and Why lists?

- Disadvantages of arrays as storage data structures:
  - slow searching in unordered array
  - slow insertion in ordered array
  - Fixed size
- Linked lists solve some of these problems
- Linked lists are general purpose storage data structures and are versatile.

Linked List Efficiency

- Insertion and deletion at the beginning of the list are very fast, O(1).
- Finding, deleting or inserting in the list requires searching through half the items in the list on an average, requiring O(n) comparisons.
- Arrays require same number of comparisons, the advantage- No items need to be moved after insertion or deletion.
- Fixed size of arrays-- linked lists use exactly as much memory as is needed and can expand.

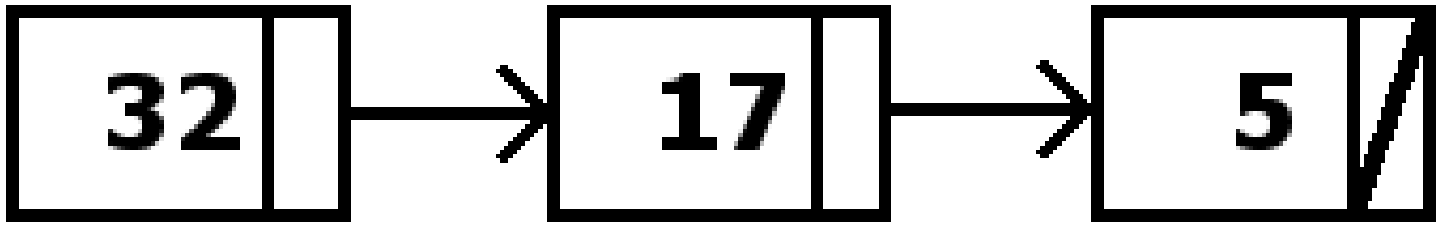
Create a Linked List in Java

Classes required

- Class LinkedList: Methods to support operations such as insertion and deletion
- Class for a Node: data fields to store data (Object reference), Node reference to next Node. (ideally a nested class of LinkedList)
- (Sample solution after this. Try to write code and then look)

Nodes: objects to store elements

- let's make a special "node" type of object that represents a storage slot to hold one element of a list
- each node will keep a reference to the node after it (the "next" node)
- the last node will have `next == null` (drawn as /), signifying the end of the list

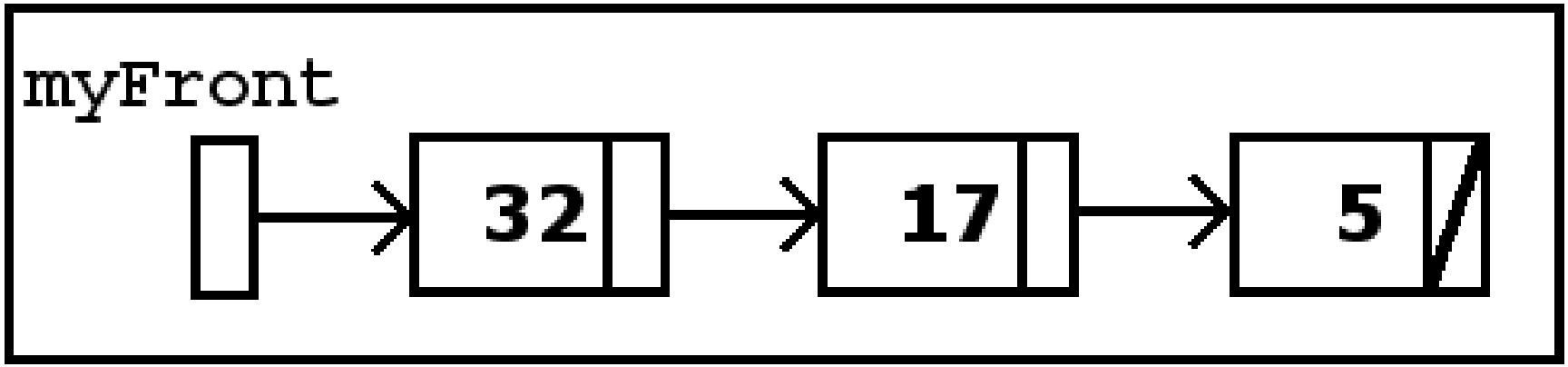


Node implementation

```
/* Stores one element of a linked list.
   */
public class Node {
   public Object element;
   public Node next;

   public Node(Object element) {
      this(element, null);
   }

   public Node(Object element, Node next)
   {
      this.element = element;
      this.next = next;
   }
}
```

Linked list

- **linked list**: a list implemented using a linked sequence of nodes
  - the list only needs to keep a reference to the first node (we might name it `myFront`)
  - in Java: `java.util.LinkedList`
    (but we'll write our own)

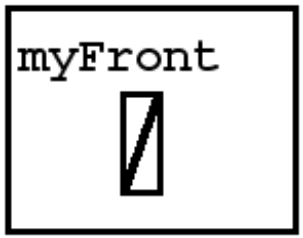


Linked list implementation

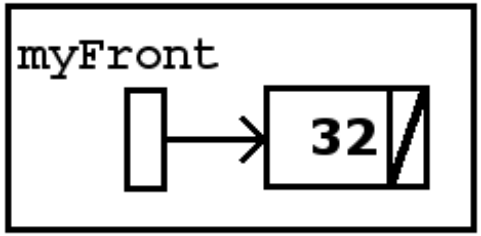
```
/* Models an entire linked list. */
public class MyLinkedList {
  private Node myFront;

  public MyLinkedList() {
    myFront = null;
  }

  /* Methods go here */
}
```

Some list states of interest

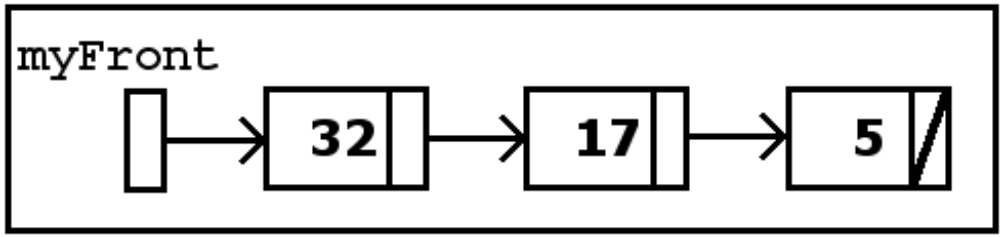
- empty list
  (`myFront == null`)



- list with one element



- list with many elements



Few operations for the LinkedList class

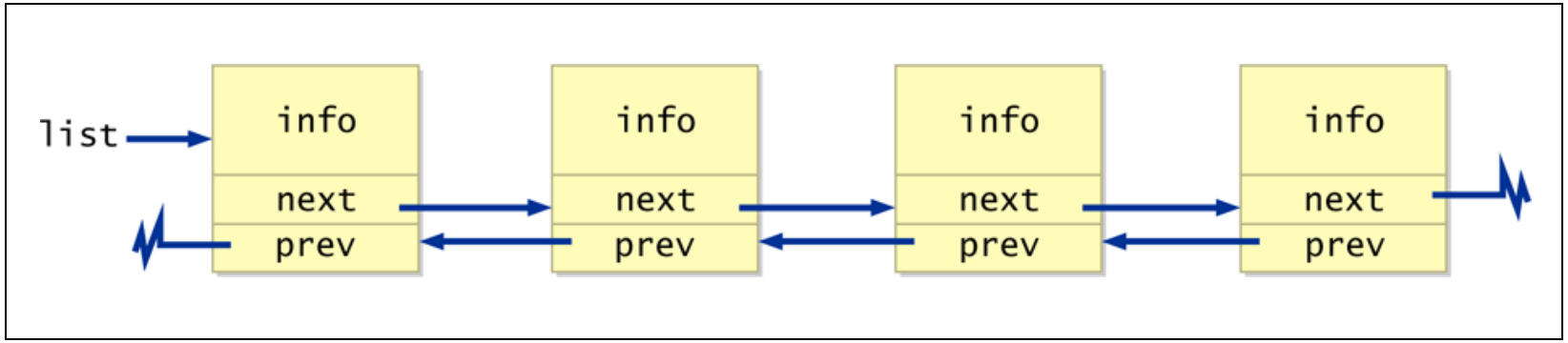
- an add operation
  - at the front, back, and middle
- a remove operation
- a get operation
- a set operation
- an index of (searching) operation

Analysis of `LinkedList` runtime

OPERATION	RUNTIME (Big-Oh)
add to start of list	O(1)
add to end of list	**O(*n*)**
add at given index	**O(*n*)**
clear	O(1)
get	**O(*n*)**
find index of an object	**O(*n*)**
remove first element	O(1)
remove last element	**O(*n*)**
remove at given index	**O(*n*)**
set	**O(*n*)**
size	**O(*n*)**
toString	**O(*n*)**

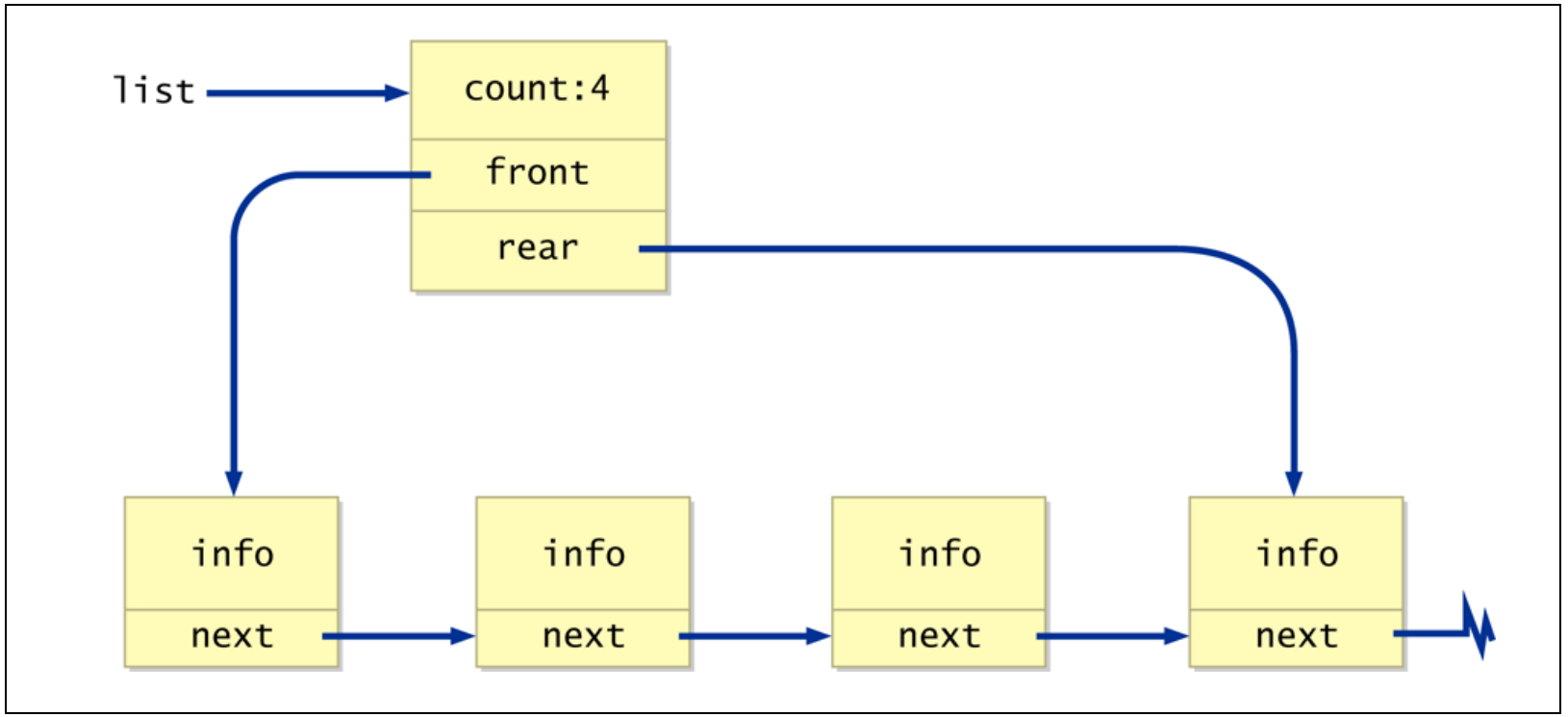
Other Dynamic Representations

- It may be convenient to implement a list as a *doubly linked list*, with `next` and `previous` references:



Other Dynamic Representations

- Another approach is to use a separate *header node*, with a count and references to both the front and rear of the list:



An optimization

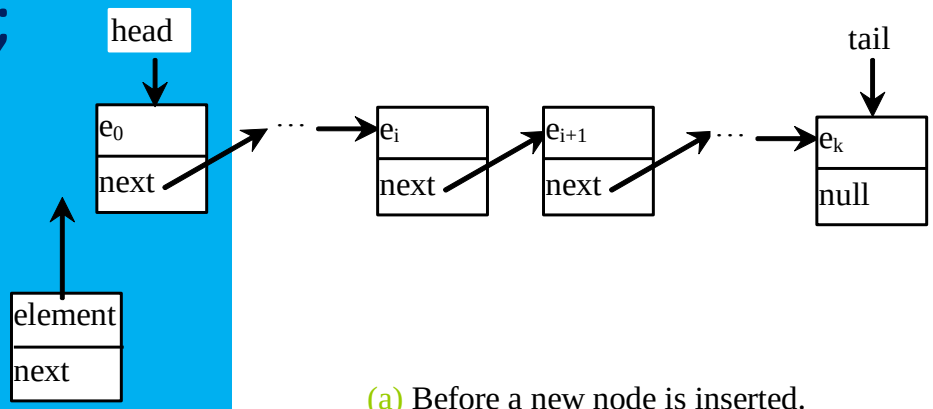
- problem: array list has O(1) get/remove of last element, but the linked list needs O(*n*)
- solution: add a tail pointer to the last node
  - which methods' Big-Oh runtime improve to O(1)?
  - what complications does this add to the implementation of the methods of the list?
- Linked list needs O(n) to get size
- Solution: add a size attribute

Linked list implementation

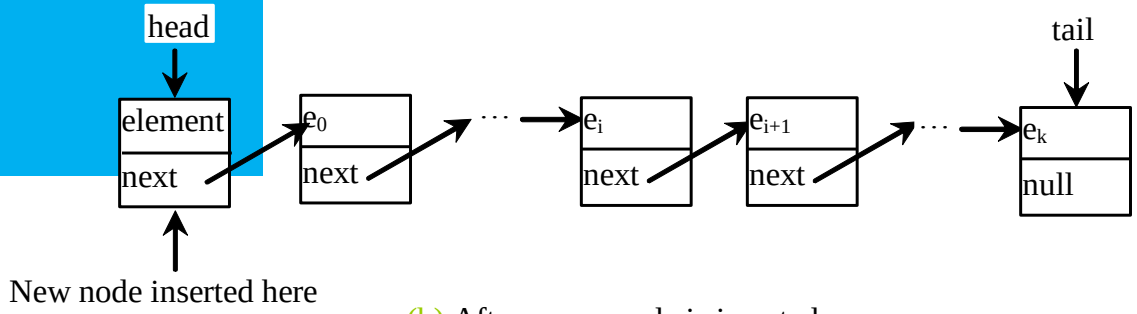
```
/* Models an entire linked list. */
public class MyLinkedList {
  private Node head;
  private Node tail;
  private int size;

  public MyLinkedList() {
    head = null;
  }

  /* Methods go here */
}
```

Implementing addFirst(Object o)

```
public void addFirst(Object o)
{
  Node newNode = new
Node(o);
  newNode.next = head;
  head = newNode;
  size++;
  if (tail == null)
    tail = head;
}
```



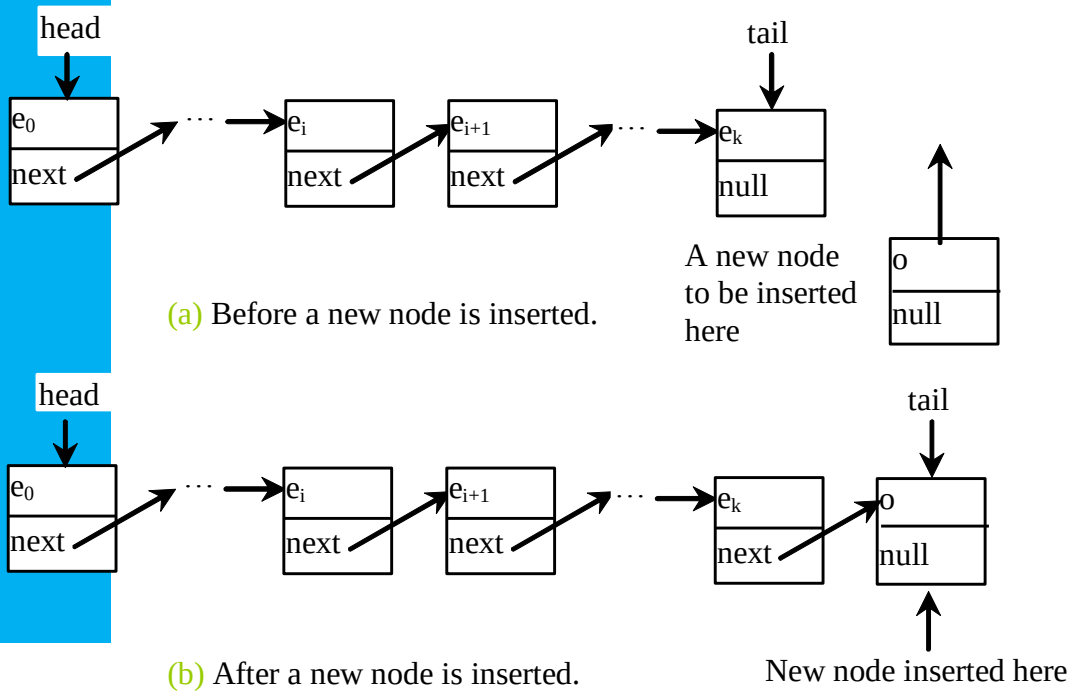
(a) Before a new node is inserted.



(b) After a new node is inserted.

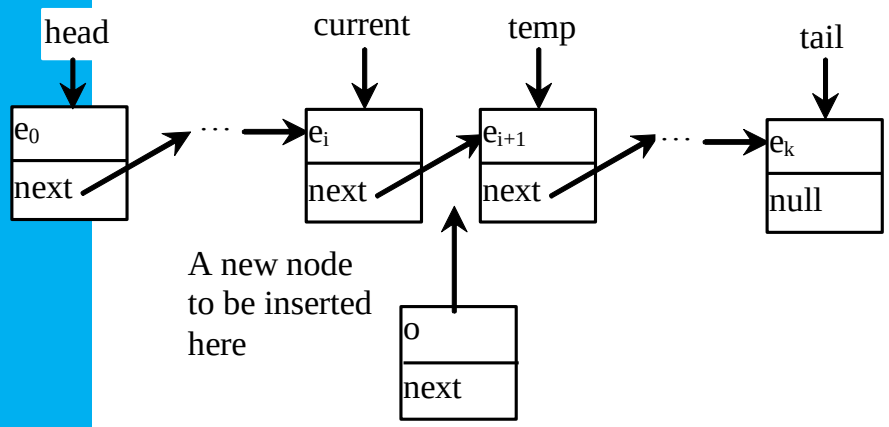
Implementing addLast(Object o)

```
public void
addLast(Object o) {
  if (tail == null) {
    head = tail = new
Node(o);
  }
  else {
    tail.next = new
Node(o);
    tail = tail.next;
  }
  size++;
}
```

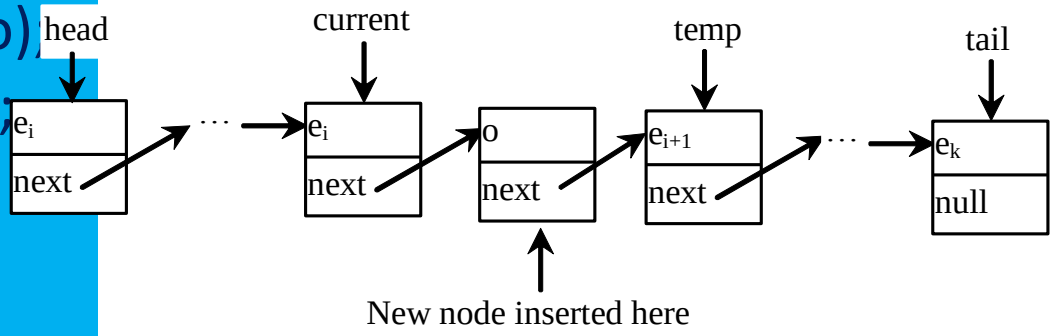


Implementing add(int index, Object o)

```
public void add(int index,
Object o) {
  if (index == 0) addFirst(o);
  else if (index >= size)
addLast(o);
  else {
    Node current = head;
    for (int i = 1; i < index; i++)
      current = current.next;
    Node temp = current.next;
    current.next = new Node(o);
    (current.next).next = temp;
    size++;
  }
}
```



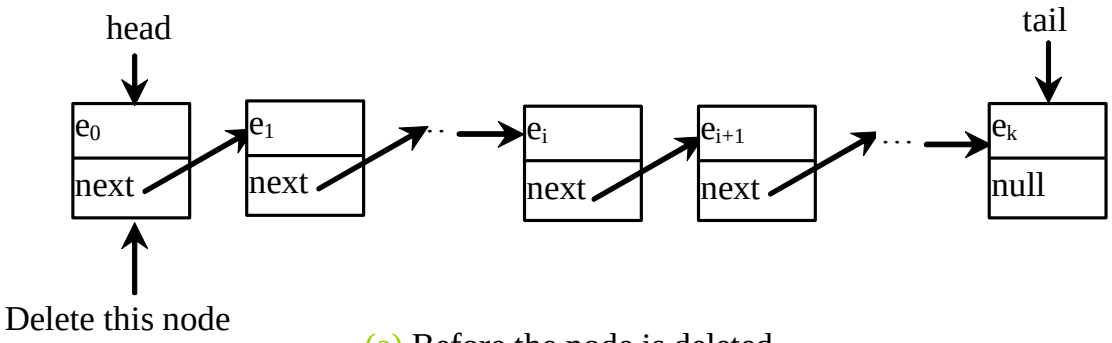
(a) Before a new node is inserted.



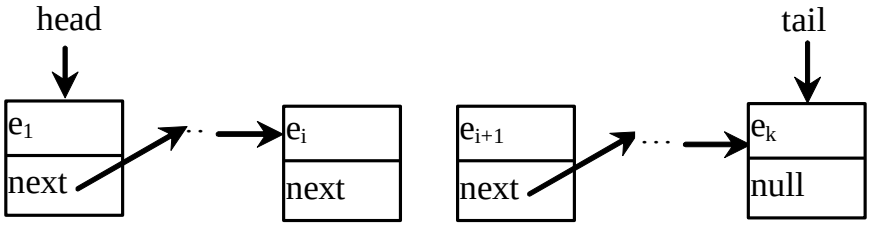
(b) After a new node is inserted.

Implementing removeFirst()

```
public Object removeFirst() {
  if (size == 0) return null;
  else {
    Node temp = head;
    head = head.next;
    size--;
    if (head == null) tail = null;
    return temp.element;
  }
}
```



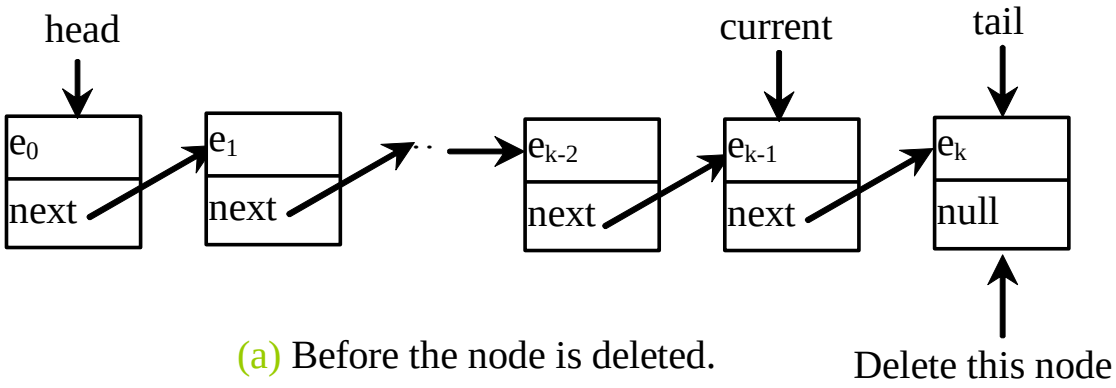
(a) Before the node is deleted.



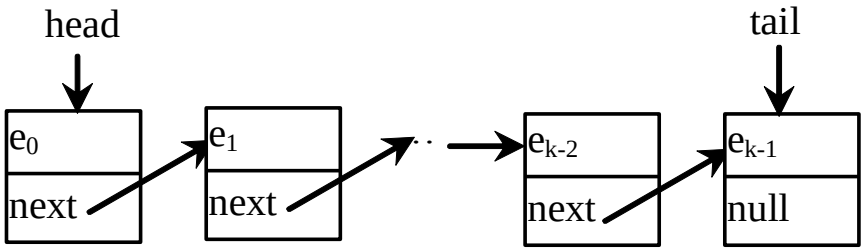
(b) After the first node is deleted

```
public Object removeLast() {
  if (size == 0) return null;
  else if (size == 1)
  {
    Node temp = head;
    head = tail = null;
    size = 0;
    return temp.element;
  }
  else
  {
    Node current = head;
    for (int i = 0; i < size - 2; i++)
      current = current.next;
    Node temp = tail;
    tail = current;
    tail.next = null;
    size--;
    return temp.element;
  }
}
```

Implementing removeLast()



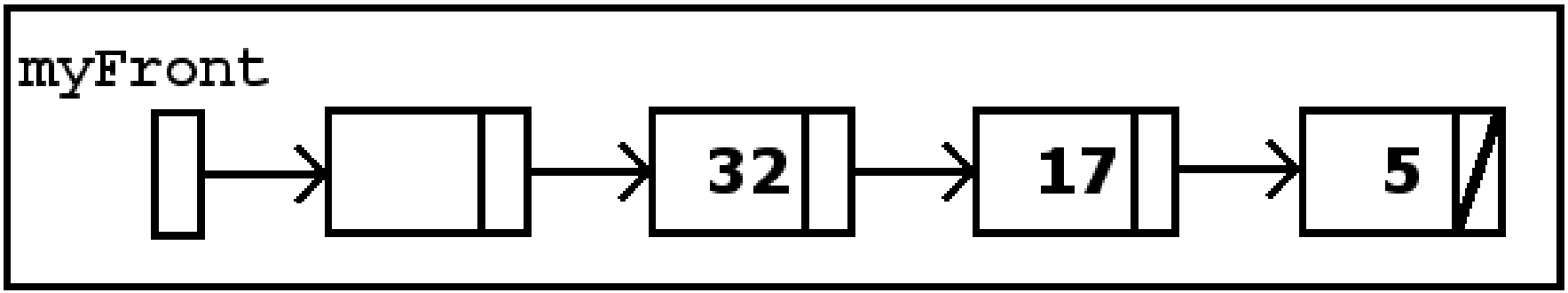
(a) Before the node is deleted.



(b) After the last node is deleted

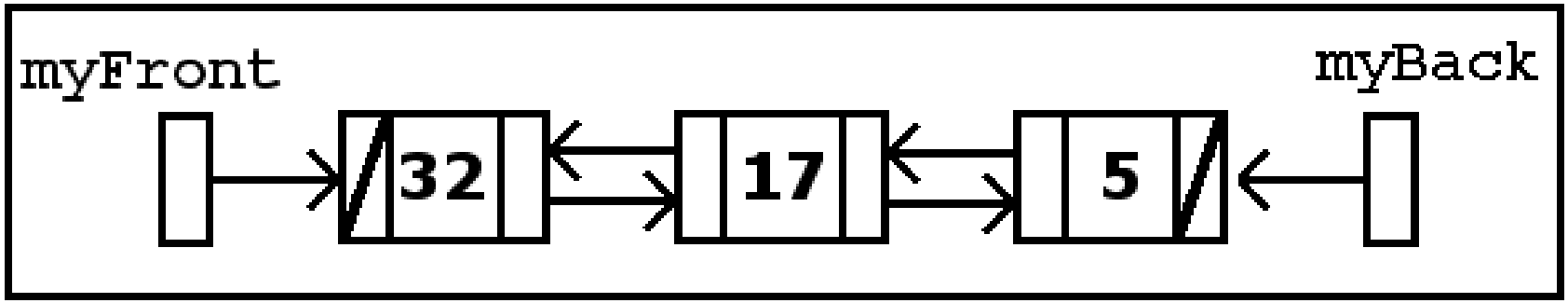
A variation: dummy header

- **dummy header**: a front node intentionally left blank
  - `myFront` always refers to dummy header (`myFront` will never be `null`)
  - requires minor modification to many methods
  - surprisingly, makes implementation much easier



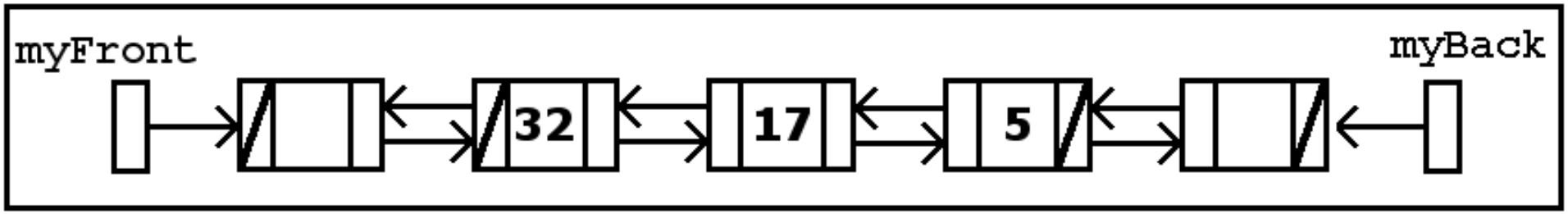
Doubly-linked lists

- add a `prev` pointer to our `Node` class
- allows backward iteration
- some methods need to be modified
  - when adding or removing a node, we must fix the `prev` and `next` pointers to have the correct value!
  - can make it easier to implement some methods such as `remove`



Combining the approaches

- Most actual linked list implementations are doubly-linked and use a dummy header and **dummy tail**
- this actually makes a very clean implementation for all linked list methods and provides good efficiency for as many operations as possible



Improved `LinkedList` runtime

OPERATION	RUNTIME (Big-Oh)
add to start of list	O(1)
add to end of list	O(1)
add at given index	**O(*n*)**
clear	O(1)
get	**O(*n*)**
find index of an object	**O(*n*)**
remove first element	O(1)
remove last element	O(1)
remove at given index	**O(*n*)**
set	**O(*n*)**
size	O(1)
toString	**O(*n*)**

ArrayList

ArrayList

- Array is a fixed-size data structure. Once an array is created, its size cannot be changed.
- Use array to implement dynamic data structures.
  - The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

ArrayList

- Initially, an array, say data of Object[] type, is created with a default size.
- When inserting a new element into the array,
  - first ensure there is enough room in the array.
  - If not, create a new array with the size as twice as the current one.
  - Copy the elements from the current array to the new array.
  - The new array now becomes the current array.

Java ArrayList methods

`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]"`

Java ArrayList methods

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

Type Parameters (Generics)

*ArrayList<**Type**> **names** = new ArrayList<**Type**>();*

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

*ArrayList**<String>** names = new ArrayList**<String>**();*

names.add(“Alice");

names.add(“Bob");

Java's LinkedList class

- Generic class *Specify type of elements in angle brackets:* `LinkedList<Product>`
- Package: `java.util`

Table 1 LinkedList Methods

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as `add`.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. `lst` is now `[Sally, Harry]`.
`lst.getFirst()`	Gets the element stored at the beginning of the list; here `"Sally"`.
`lst.getLast()`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `lst` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements (see Table 2 on page 634).

Implementing a `Queue` Class

- Implement as a `LinkedList` attribute value
  - insertions and deletions from either end are efficient, occur in constant O(1) time
  - good choice
- Implement as an `ArrayList` attribute
  - poor choice
  - adding values at one end, removing at other end require multiple shifts

A particularly slow idiom

```
// print every element of linked list
for (int i = 0; i < list.size(); i++) {
  Object element = list.get(i);
  System.out.println(i + ": " +
  element);
}
```

- This code executes an O(*n*) operation (`get`) every time through a loop that runs *n* times!
  - Its runtime is O(n^2), which is much worse than O(*n*)
  - this code will take prohibitively long to run for large data sizes

The problem of position

- The code on the previous slide is wasteful because it throws away the position each time
  - every call to `get` has to re-traverse the list!
- it would be much better if we could somehow keep the list in place at each index as we looped through it

- Java uses special objects to represent a position of a collection as it's being examined...
  - these objects are called "iterators"