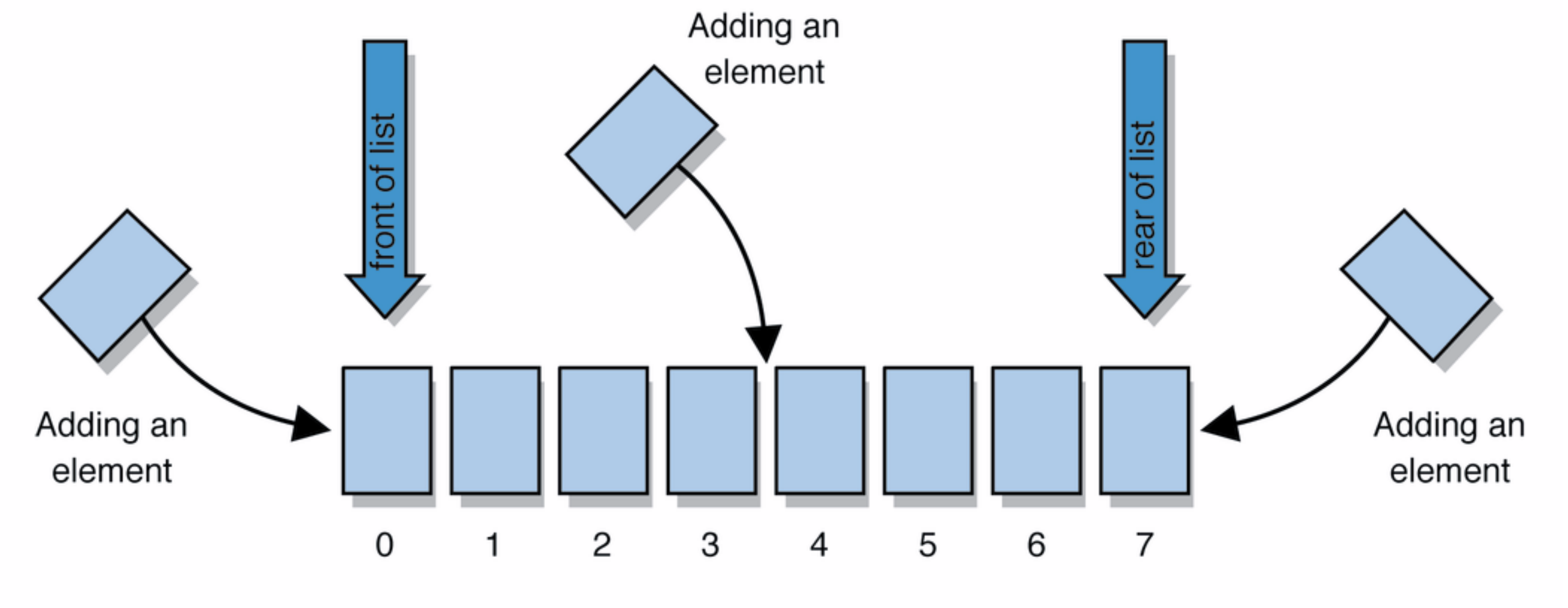
List Implementation

Weiss Ch. 6, pp. 183-205
Weiss Ch. 17, pp. 537-548

An example collection: List

- **list**: an ordered sequence of elements, each accessible by a 0-based index
  - one of the most basic collections



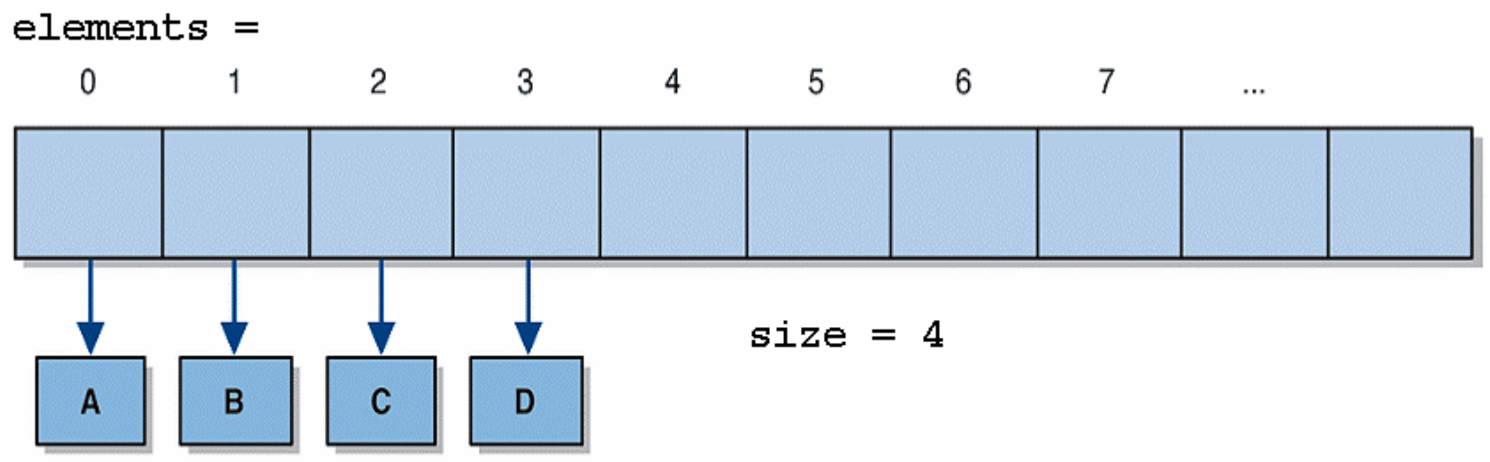
List features

- **ORDERING**: maintains order elements were added (new elements are added to the end by default)
- **DUPLICATES**: yes (allowed)
- **OPERATIONS**: add element to end of list, insert element at given index, clear all elements, search for element, get element at given index, remove element at given index, get size
  - some of these operations are inefficient! (seen later)

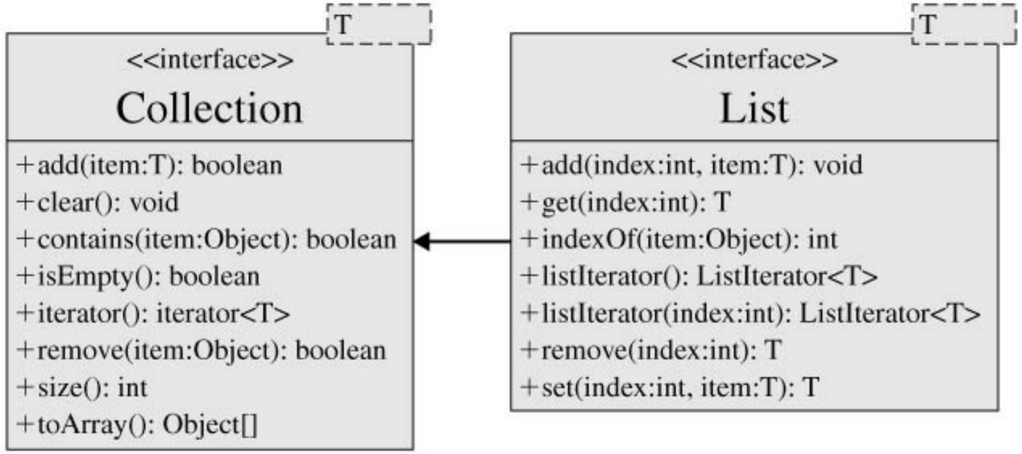
- list manages its own size; user of the list does not need to worry about overfilling it

Array list

- **array list**: a list implemented using an array to store the elements
  - encapsulates array and # of elements (size)
  - in Java: `java.util.ArrayList`
  - when you want to use `ArrayList`, remember to `import java.util.*;`



ArrayList implementation



- recall: ArrayList implements the List interface
  - which is itself an extension of the Collection interface
  - underlying list structure is an array
    - `get(index), add(item), set(index, item)` → O(1)
    - `add(index, item), indexOf(item), contains(item),`
    - `remove(index), remove(item)` → O(N)

ArrayList class structure

- the ArrayList class has as fields
  - the underlying array
  - number of items stored

- the default initial capacity is defined by a constant
  - capacity != size

```
public class SimpleArrayList<E> implements Iterable<E>{
    private static final int INIT_SIZE = 10;
    private E[] items;
    private int numStored;

    public SimpleArrayList() {
        this.clear();
    }

    public void clear() {
        this.numStored = 0;
        this.ensureCapacity(INIT_SIZE);
    }

    public void ensureCapacity(int newCapacity) {
        if (newCapacity > this.size()) {
            E[] old = this.items;
            this.items = (E[]) new Object[newCapacity];
            for (int i = 0; i < this.size(); i++) {
                this.items[i] = old[i];
            }
        }
    }

    .
    .
    .
```

<u>interestingly:</u> you can't create a generic array

```
this.items = new E[capacity];    // ILLEGAL
```

can work around this by creating an array of Objects, then casting to the generic array type

ArrayList: add

- the add method
  - throws an exception if the index is out of bounds
  - calls ensureCapacity to resize the array if full
  - shifts elements to the right of the desired index
  - finally, inserts the new value and increments the count

- the add-at-end method calls this one

```
public void add(int index, E newItem) {
    this.rangeCheck(index, "ArrayList add()", this.size());
    if (this.items.length == this.size()) {
        this.ensureCapacity(2*this.size() + 1);
    }

    for (int i = this.size(); i > index; i--) {
        this.items[i] = this.items[i-1];
    }
    this.items[index] = newItem;
    this.numStored++;
}

private void rangeCheck(int index, String msg, int upper) {
    if (index < 0 || index > upper)
        throw new IndexOutOfBoundsException("\n" + msg +
                ": index " + index + " out of bounds. " +
                "Should be in the range 0 to " + upper);
}


public boolean add(E newItem) {
    this.add(this.size(), newItem);
    return true;
}
```

ArrayList: size, get, set, indexOf, contains

- size method
  - returns the item count
- get method
  - checks the index bounds, then simply accesses the array
- set method
  - checks the index bounds, then assigns the value
- indexOf method
  - performs a sequential search
- contains method
  - uses indexOf

```
public int size() {
    return this.numStored;
}

public E get(int index) {
    this.rangeCheck(index, "ArrayList get()", this.size()-1);
    return items[index];
}

public E set(int index, E newItem) {
    this.rangeCheck(index, "ArrayList set()", this.size()-1);
    E oldItem = this.items[index];
    this.items[index] = newItem;
    return oldItem;
}

public int indexOf(E oldItem) {
    for (int i = 0; i < this.size(); i++) {
        if (oldItem.equals(this.items[i])) {
            return i;
        }
    }
    return -1;
}

public boolean contains(E oldItem) {
    return (this.indexOf(oldItem) >= 0);
}
```

ArrayList: remove

- the remove method
  - checks the index bounds
  - then shifts items to the left and decrements the count
  - note: could shrink size if becomes ½ empty
- the other remove
  - calls indexOf to find the item, then calls remove(index)

```
public void remove(int index) {
    this.rangeCheck(index, "ArrayList remove()", this.size()-1);

    for (int i = index; i < this.size()-1; i++) {
        this.items[i] = this.items[i+1];
    }
    this.numStored--;
}


public boolean remove(E oldItem) {
    int index = this.indexOf(oldItem);
    if (index >= 0) {
        this.remove(index);
        return true;
    }
    return false;
}
```

could we do this more efficiently?

do we care?

ArrayList iterator

- an ArrayList does not really need an iterator
  – get() and set() are already O(1) operations, so typical indexing loop suffices
  – provided for uniformity (`java.util.Collections` methods require *iterable* classes)
  – also required for enhanced for loop to work

- to implement an iterator, need to define a new class that can
  – access the underlying array (→ must be inner class to have access to private fields)
  – keep track of which location in the array is "next"

"foo"	"bar"	"biz"	"baz"	"boo"	"zoo"
0	1	2	3	4	5

nextIndex	0

SimpleArrayList iterator

- `java.lang.Iterable` interface declares that the class has an iterator

- inner class defines an Iterator class for this particular collection (accessing the appropriate fields & methods)

- the iterator() method creates and returns an object of that class

```
public class SimpleArrayList<E> implements Iterable<E> {

    . . .

    public Iterator<E> iterator() {
        return new ArrayListIterator();
    }

    private class ArrayListIterator implements Iterator<E> {
      private int nextIndex;
      public ArrayListIterator() {
          this.nextIndex = 0;
      }

      public boolean hasNext() {
          return this.nextIndex < SimpleArrayList.this.size();
      }

      public E next() {
          if (!this.hasNext()) {
              throw new java.util.NoSuchElementException();
          }
          this.nextIndex++;
          return SimpleArrayList.this.get(nextIndex-1);
      }

    public void remove() {
        if (this.nextIndex <= 0) {
            throw new RuntimeException("Iterator call to " +
                      "next() required before calling remove()");
        }
        SimpleArrayList.this.remove(this.nextIndex-1);
        this.nextIndex--;
     }
  }
}
```

Iterators & the enhanced for loop

- given an iterator, collection traversal is easy and uniform

```
SimpleArrayList<String> words;
. . .
Iterator<String> iter = words.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

- as long as the class implements Iterable<E> and provides an iterator() method, the enhanced for loop can also be applied

```
SimpleArrayList<String> words;
. . .
for (String str : words) {
    System.out.println(str);
}
```

Analysis of `ArrayList` runtime

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

Open questions

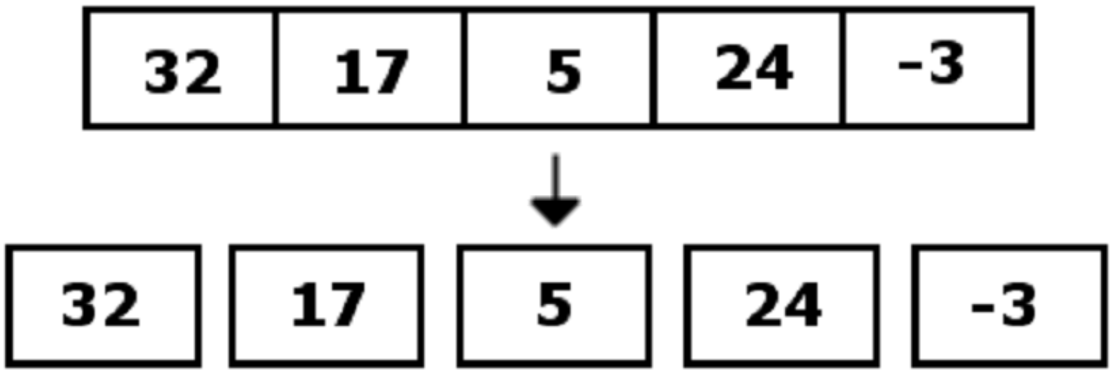
- Based on the preceding analysis, when is an `ArrayList` a good collection to use? When is it a poor performer?

- Is there a way that we could fix some of the problems with the `ArrayList`?

- Should we represent our list in a different way?

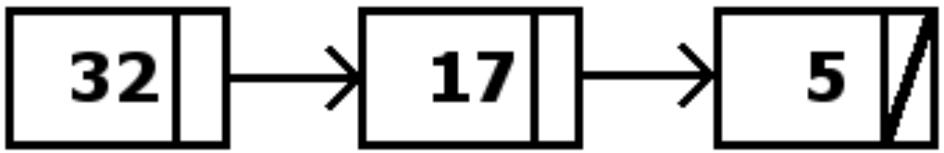
The underlying issue

- the elements of an `ArrayList` are too tightly attached; can't easily rearrange them
- can we break the element storage apart into a more dynamic and flexible structure?



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



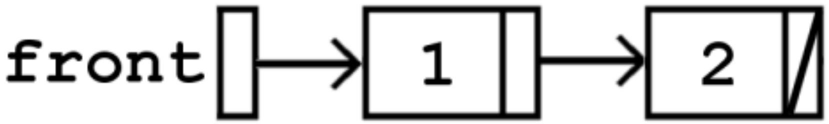
Node implementation

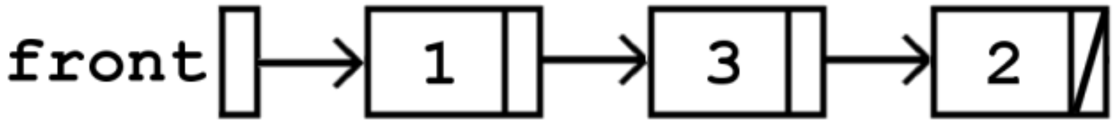
```
/* Stores one element of a linked list. */
public class Node {
  public Object element;
  public Node next;

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

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

Linked node problems (a)

- Let's examine sample chains of nodes together, and try to write the correct code for each
  - each Node stores an `Integer` object

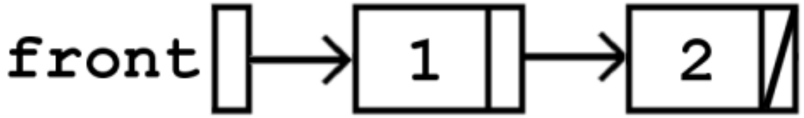
1.

- before: 

- after: 

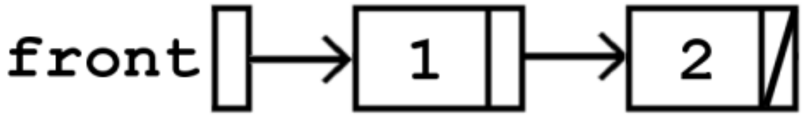
Linked node problems (b)

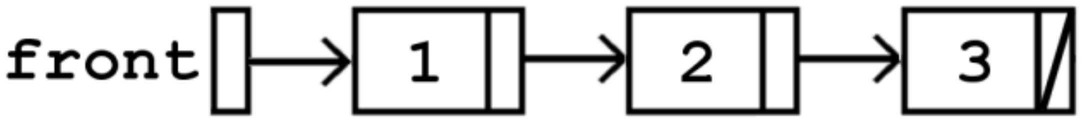
2.

- before: 

- after: 

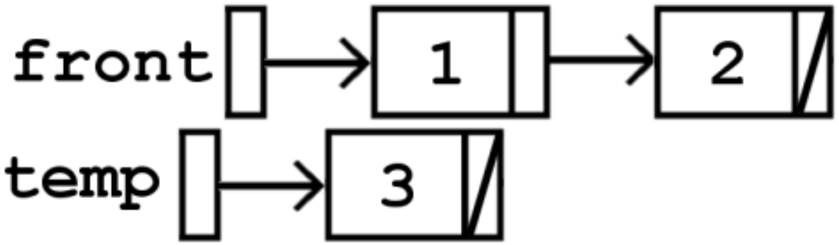
3.

- before: 

- after: 

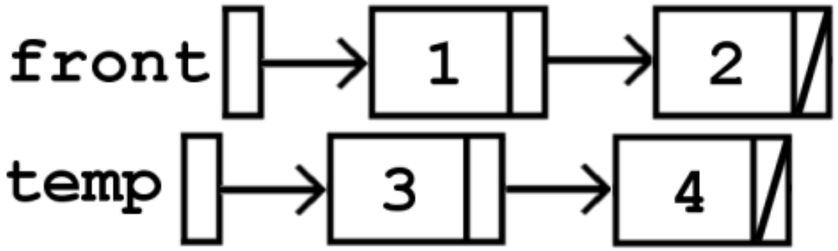
Linked node problems (c)

4.
- before:

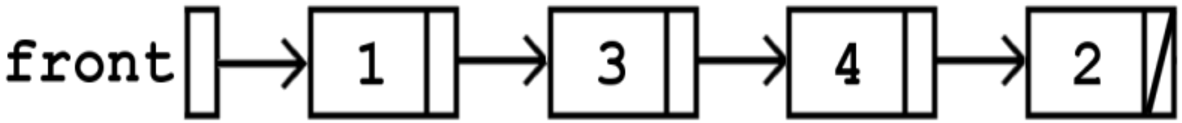


- after:

5.
- before:



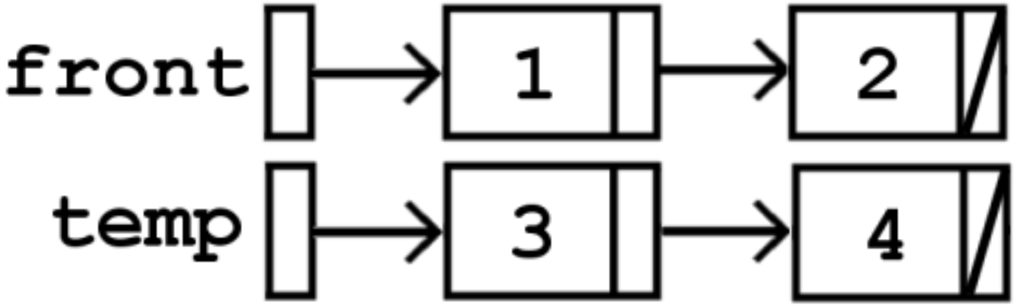
- after:



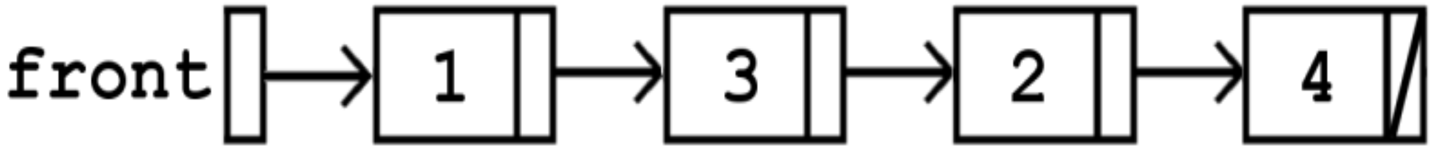
Linked node problems (d)

6.

- before:

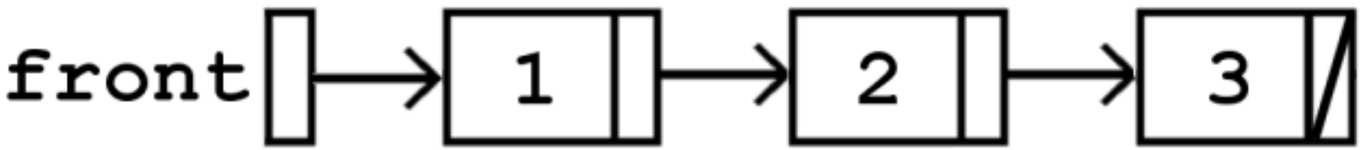


- after:

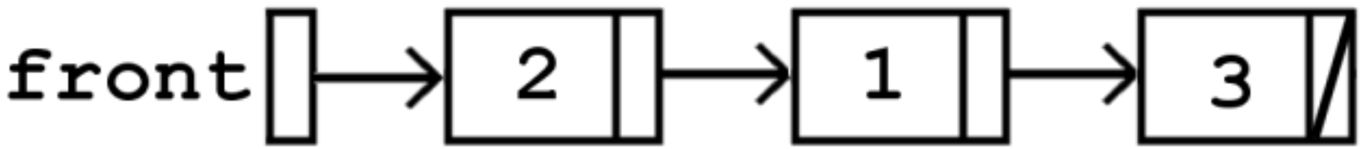


7.

- before:

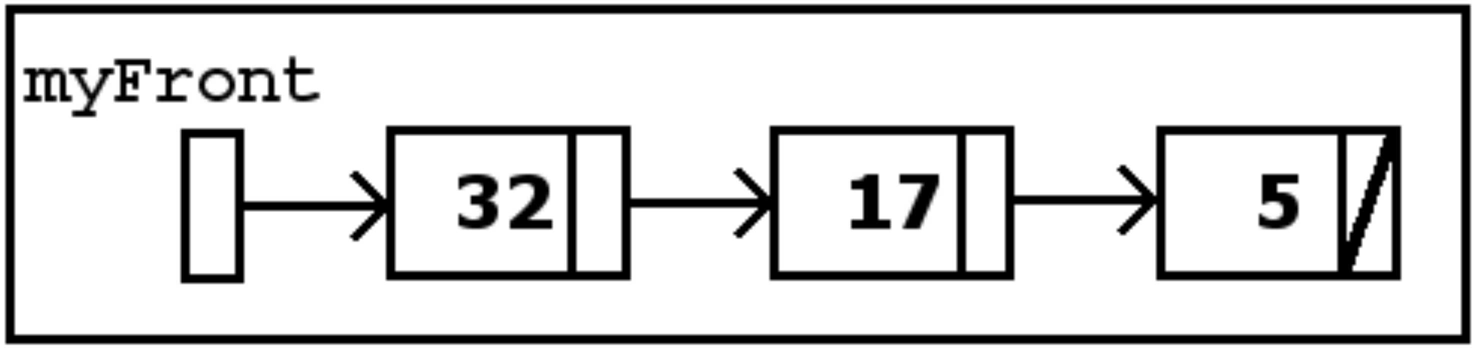


- after:



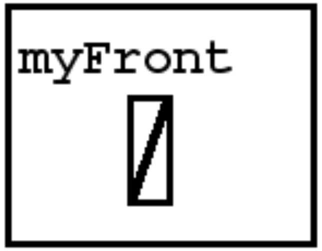
Linked list

- **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)

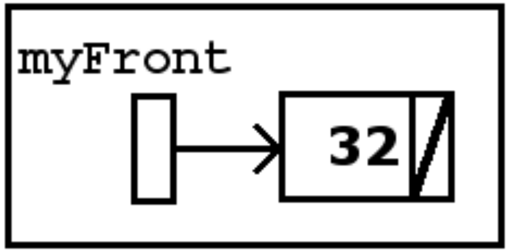


Some list states of interest

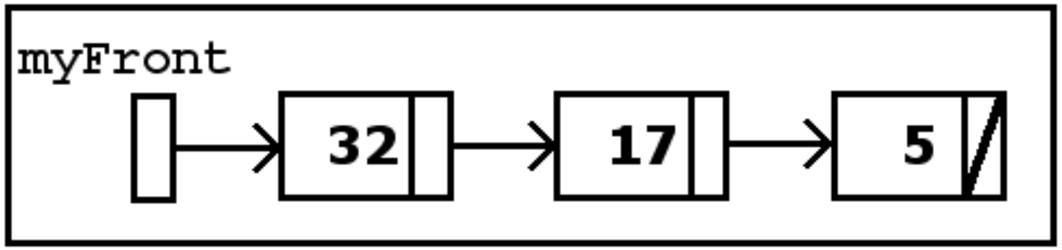
- empty list
  (`myFront == null`)



- list with one element



- list with many elements

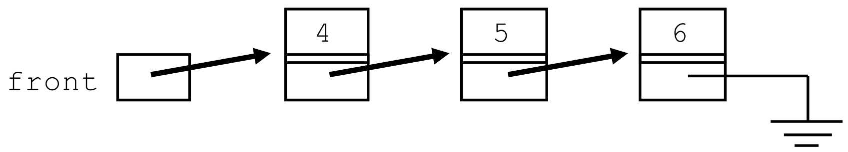


Let's draw them together...

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

Singly-linked lists

- Singly-linked lists
  - the list was made of Nodes, each of which stored data and a link to the next node in the list
  - can provide a constructor and methods for accessing and setting these two fields
  - a reference to the front of the list must be maintained

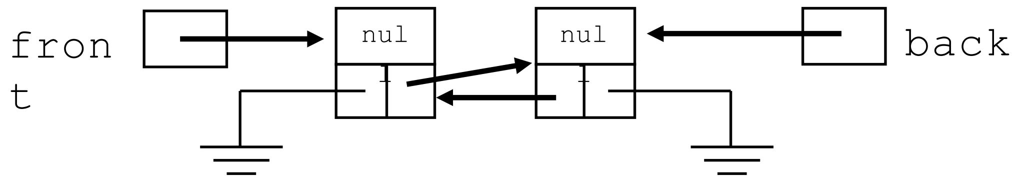



```java
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```

LinkedList class structure

- the LinkedList class has an inner class
  - defines the DNode class
- fields store
  - reference to front and back dummy nodes
  - node count
- the constructor
  - creates the front & back dummy nodes
  - links them together
  - initializes the count

```
public class SimpleLinkedList<E> implements Iterable<E>{
    private class DNode<E> {
        . . .
    }

    private DNode<E> front;
    private DNode<E> back;
    private int numStored;

    public SimpleLinkedList() {
        this.clear();
    }

    public void clear() {
        this.front = new DNode(null, null, null);
        this.back = new DNode(null, front, null);
        this.front.setNext(this.back);
        this.numStored = 0;
    }
```



Exercises

- to create an empty linked list:

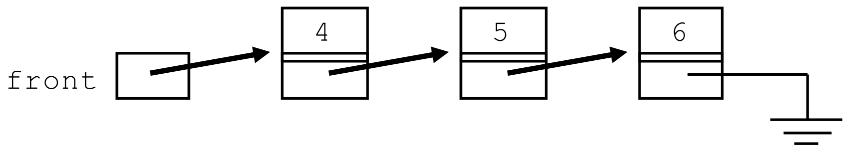
```
front = null;
```

- to add to the front:

```
front = new Node(3, front);
```

- remove from the front:

```
front = front.getNext();
```

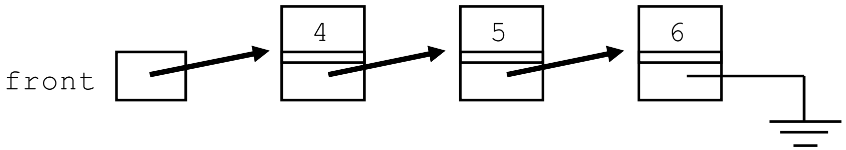



```
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```

Exercises

- get value stored in first node:
- get value in kth node:

- indexOf:
- add at end:
- add at index:
- remove:
- remove at index:



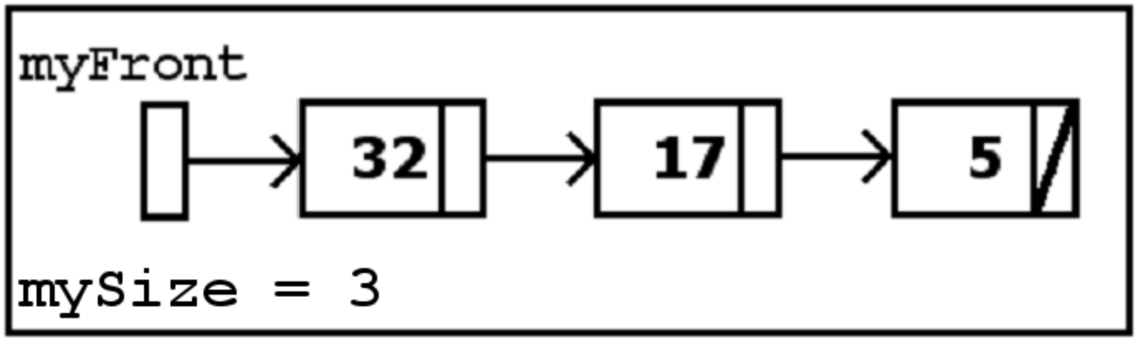

```
public class Node<E> {
    private E data;
    private Node<E> next;

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```

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*)**

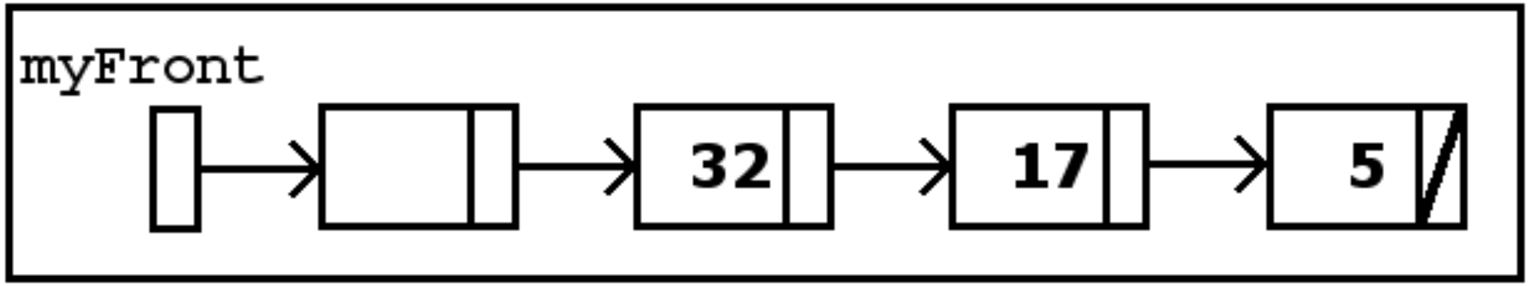
An optimization: `mySize`

- problem: array list has a O(1) `size` method, but the linked list needs O(*n*) time
- solution: add a `mySize` field to our linked list
  - what changes must be made to the implementation of the methods of the linked list?



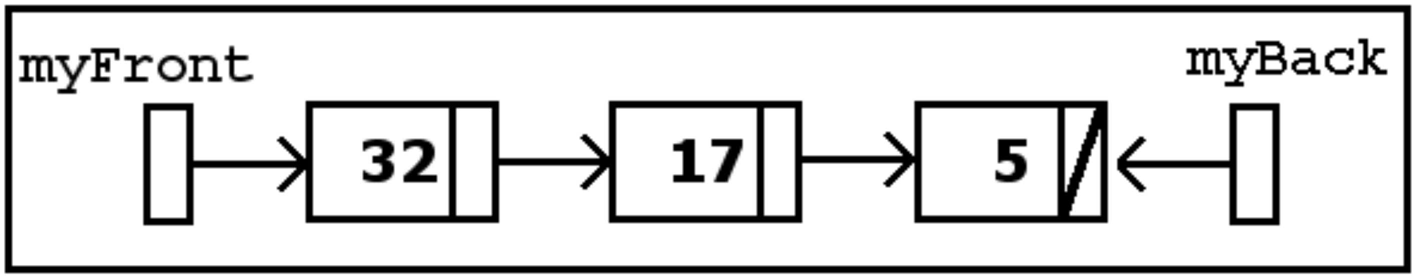
A variation: dummy header

- **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



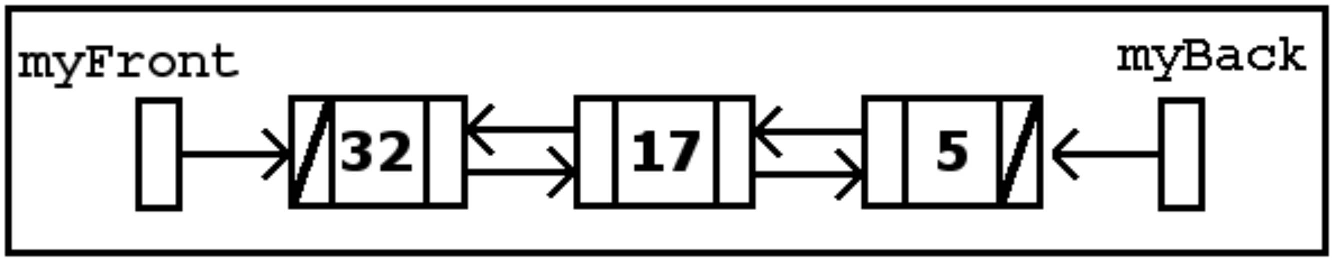
An optimization: `myBack`

- problem: array list has O(1) get/remove of last element, but the linked list needs O(*n*)
- solution: add a `myBack` 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?



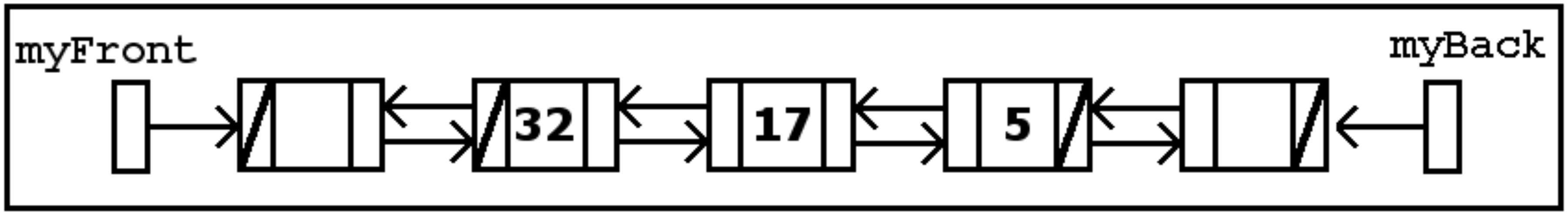
Doubly-linked lists

- add a `prev` pointer to our `Node` class
- allows backward iteration (for `ListIterator`)
- 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



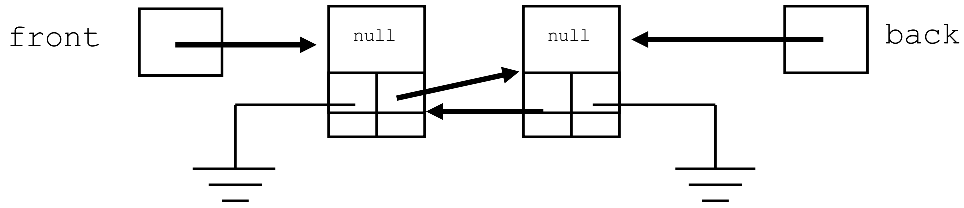
LinkedList class structure

- the LinkedList class has an inner class
  - defines the DNode class

- fields store
  - reference to front and back dummy nodes
  - node count

- the constructor
  - creates the front & back dummy nodes
  - links them together
  - initializes the count

```
public class SimpleLinkedList<E> implements Iterable<E>{
    private class DNode<E> {
        . . .
    }

    private DNode<E> front;
    private DNode<E> back;
    private int numStored;

    public SimpleLinkedList() {
        this.clear();
    }

    public void clear() {
        this.front = new DNode(null, null, null);
        this.back = new DNode(null, front, null);
        this.front.setNext(this.back);
        this.numStored = 0;
    }
```



LinkedList: add

- the add method
  - similarly, throws an exception if the index is out of bounds
  - calls the helper method getNode to find the insertion spot
  - note: getNode traverses from the closer end
  - finally, inserts a node with the new value and increments the count
- add-at-end similar

```
public void add(int index, E newItem) {
    this.rangeCheck(index, "LinkedList add()", this.size());

    DNode<E> beforeNode = this.getNode(index-1);
    DNode<E> afterNode = beforeNode.getNext();

    DNode<E> newNode = new DNode<E>(newItem,beforeNode,afterNode);
    beforeNode.setNext(newNode);
    afterNode.setPrevious(newNode);

    this.numStored++;
}

private DNode<E> getNode(int index) {
    if (index < this.numStored/2) {
        DNode<E> stepper = this.front;
        for (int i = 0; i <= index; i++) {
            stepper = stepper.getNext();
        }
        return stepper;
    }
    else {
        DNode<E> stepper = this.back;
        for (int i = this.numStored-1; i >= index; i--) {
            stepper = stepper.getPrevious();
        }
        return stepper;
    }
}


public boolean add(E newItem) {
    this.add(this.size(), newItem);
    return true;
}
```

LinkedList: size, get, set, indexOf, contains

- size method
  - returns the item count
- get method
  - checks the index bounds, then calls getNode
- set method
  - checks the index bounds, then assigns
- indexOf method
  - performs a sequential search
- contains method
  - uses indexOf

```
public int size() {
    return this.numStored;
}

public E get(int index) {
    this.rangeCheck(index, "LinkedList get()", this.size()-1);
    return this.getNode(index).getData();
}

public E set(int index, E newItem) {
    this.rangeCheck(index, "LinkedList set()", this.size()-1);
    DNode<E> oldNode = this.getNode(index);
    E oldItem = oldNode.getData();
    oldNode.setData(newItem);
    return oldItem;
}

public int indexOf(E oldItem) {
    DNode<E> stepper = this.front.getNext();
    for (int i = 0; i < this.numStored; i++) {
        if (oldItem.equals(stepper.getData())) {
            return i;
        }
        stepper = stepper.getNext();
    }
    return -1;
}

public boolean contains(E oldItem) {
    return (this.indexOf(oldItem) >= 0);
}
```

LinkedList: remove

- the remove method
  - checks the index bounds
  - calls getNode to get the node
  - then calls private helper method to remove the node

- the other remove
  - calls indexOf to find the item, then calls remove(index)

```
public void remove(int index) {
    this.rangeCheck(index, "LinkedList remove()", this.size()-1);
    this.remove(this.geNode(index));
}

public boolean remove(E oldItem) {
    int index = this.indexOf(oldItem);
    if (index >= 0) {
        this.remove(index);
        return true;
    }
    return false;
}

private void remove(DNode<E> remNode) {
    remNode.getPrevious().setNext(remNode.getNext());
    remNode.getNext().setPrevious(remNode.getPrevious());
    this.numStored--;
}
```

could we do this more efficiently?

do we care?

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*)**

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

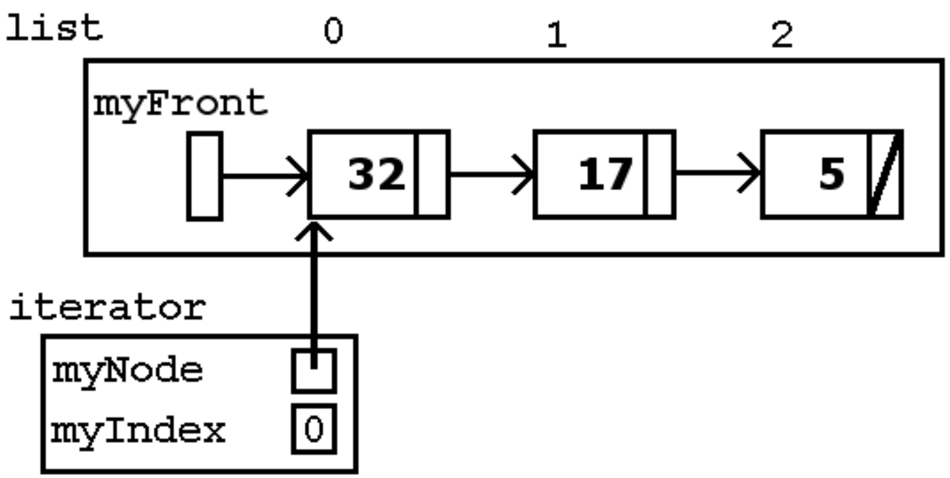
The problem of position

- 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"

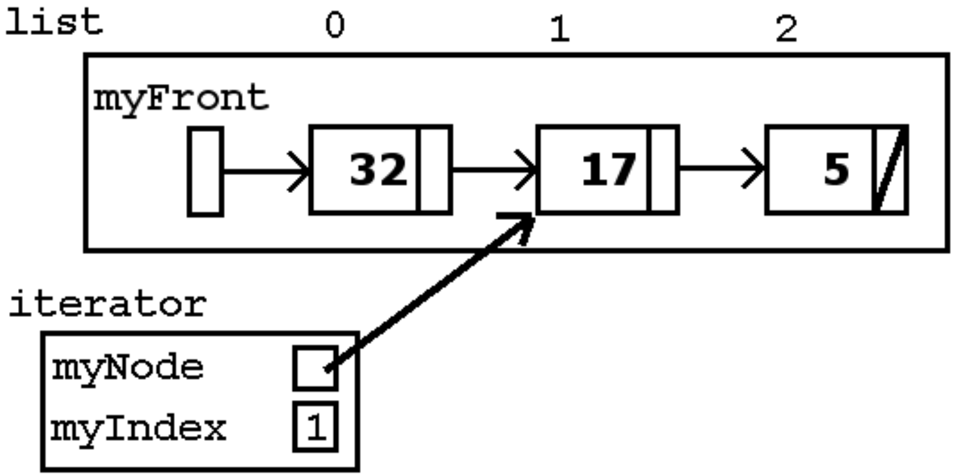
Iterators on linked lists

- an iterator on a linked list maintains (at least) its current index and a reference to that node
- when `iterator()` is called on a linked list, the iterator initially refers to the first node (index 0)



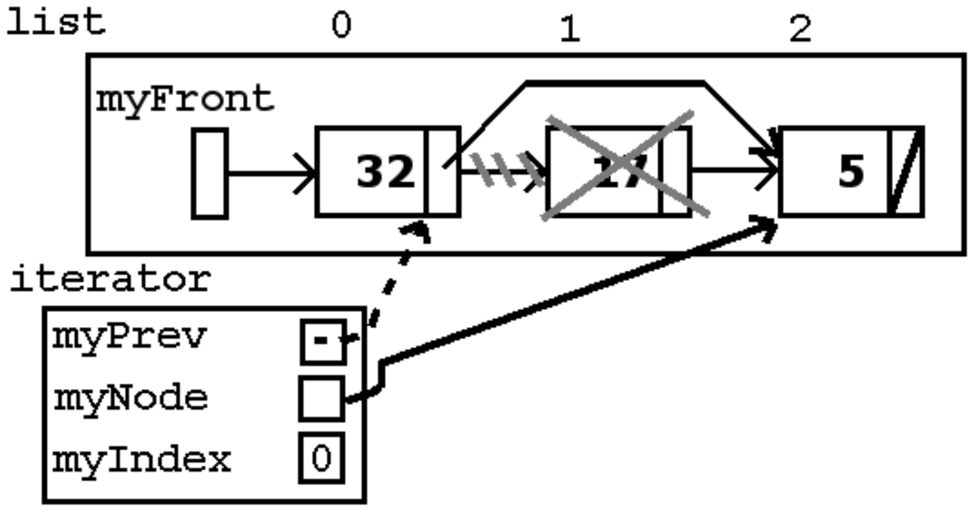
Linked list iterator iteration

- when `next()` is called, the iterator:
  - grabs the current `myNode`'s element value (32)
  - follows the `next` pointer on its node and increments its index
  - returns the element it grabbed (32)
- `hasNext` is determined by whether `myNode` is `null`
  - (Why?)



How does `remove` work?

- remove is supposed to remove the last value that was returned by `next`
- to do this, we need to delete the node *before* `myNode`, which may require modification



Fixing the slow LL idiom

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

Iterator itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
  Object element = itr.next();
  System.out.println(i + ": " + element);
}
```

Iterator usage example

```
MyLinkedList  names = new MyLinkedList();
// ... fill the list with some data ...

// print every name in the list, in upper case
Iterator itr = myList.iterator();
while (itr.hasNext()) {
  String element = (String)itr.next();
  System.out.println(element.toUpperCase());
}

// remove strings from list that start with "m"
itr = myList.iterator();
while (itr.hasNext()) {
  String element = (String)itr.next();
  if (element.startsWith("m"))
    itr.remove();  // remove element we just saw
}
```

Benefits of iterators

- speed up loops over linked lists' elements
  - What is the Big-Oh of each iterator method?
- provide a unified way to examine all elements of a collection
  - every collection in Java has an `iterator` method
    - in fact, that's the *only* guaranteed way to examine the elements of any `Collection` (see Slide 4)
  - don't need to look up different collections' method names to see how to examine their elements
- don't have to use indexes as much on lists

List Iterators Semantics

- Use next()/previous() to move
- next()/previous() returns element "moved over"
- remove() removes element that was returned from last next()/previous()
- Illegal to w/o first calling next/previous
- add(x) puts x before whatever next() would return
- Once you wrap your head around it, not too bad

List Iterators Semantics

- Removing

```
LL l = new LL([A, B, C, D])
itr = l.iterator()
             [ A B C D]
              ^
itr.next()   [ A B C D ]
A               ^
itr.remove() [ B C D ]
              ^
itr.next()   [ B C D ]
B               ^
itr.next()   [ B C D ]
C                 ^
itr.remove() [ B D ]
                ^
itr.remove() [ B D ] //Error
                ^
```

- Next/Previous

```
LL l = new LL([A, B, C, D])
itr = l.iterator()
               [ A B C D ]
                ^
itr.next()     [ A B C D ]
A                 ^
itr.next()     [ A B C D ]
B                   ^
itr.previous() [ A B C D ]
B                 ^
itr.previous() [ A B C D ]
A               ^
itr.next()     [ A B C D ]
A                 ^
itr.remove()   [ B C D ]
                ^
```

Exercise: Draw the Final List

```
LL l = new LL([A, B, C, D])
iter = l.iterator()
iter.next()
iter.next()
iter.add("X")
iter.previous()
iter.add("Y")
iter.next()
iter.next()
iter.remove()
iter.next()
iter.add("W")
iter.previous()
iter.remove()
```

Summary

- lists are ordered, integer-indexed collections that allow duplicates
- lists are good for storing elements in order of insertion and traversing them in that order
- linked lists are faster for add / remove operations at the front and back, but slower than array lists for arbitrary get / set operations
- lists are bad for searching for elements and for lots of arbitrary add / remove operations