SAMUEL GINN
COLLEGE OF ENGINEERING

Node-based Bag

A Bag collection

Revisit the Bag collection with a look at an alternate implementation that uses dynamic memory for the physical storage instead of an array.

```
public interface Bag<T> {
   boolean     add(T element);
   boolean     remove(T element);
   boolean     contains(T element);
   int         size();
   boolean     isEmpty();
   Iterator<T> iterator();
}
```

```
public class ArrayBag<T> implements Bag<T> {

   private T[] elements;
   . . .
}
```

```
public class LinkedBag<T> implements Bag<T> {

   private Node front;
   . . .
}
```

A Bag collection

A **bag** or multiset is a collection of elements where there is no particular order and duplicates are allowed. This is essentially what `java.util.Collection` describes.

We will **specify the behavior** of this collection with an **interface**:

```
import java.util.Iterator;
public interface Bag<T> {
   boolean     add(T element);
   boolean     remove(T element);
   boolean     contains(T element);
   int         size();
   boolean     isEmpty();
   Iterator<T> iterator();
}
```

A subset of the JCF Collection interface

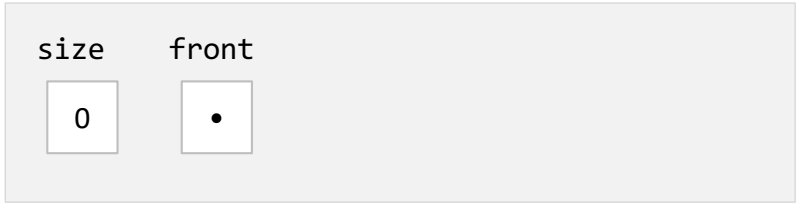
Constructor, size, isEmpty, and add

LinkedBag – constructor, size, isEmpty

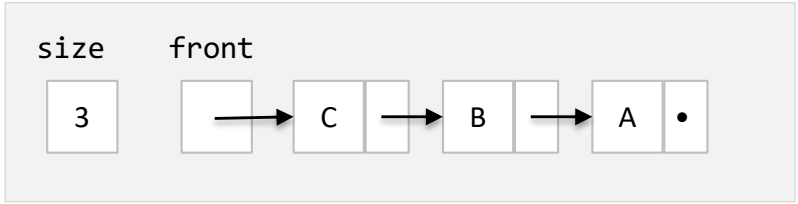
```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public LinkedBag() {
      front = null;
      size = 0;
   }

   public int size() {
      return size;
   }

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

No memory allocation!

```
Bag bag = new LinkedBag();
```



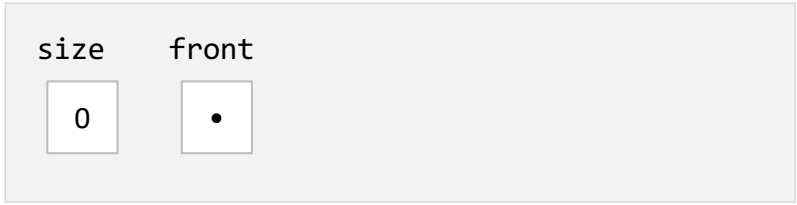

```
bag.add("A");
bag.add("B");
bag.add("C");
```



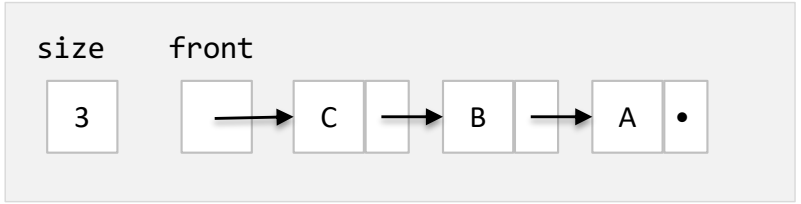
LinkedBag – add

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean add(T element) {




   }




 }
```

```
Bag bag = new LinkedBag();
```



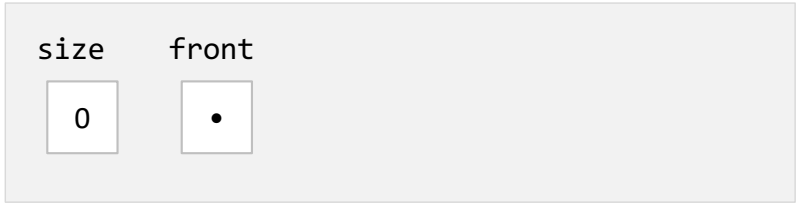

```
bag.add("A");
bag.add("B");
bag.add("C");
```



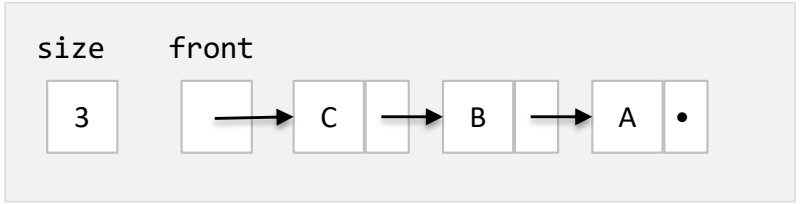
LinkedBag – add

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean add(T element) {
      Node n = new Node(element);
      n.next = front;
      front = n;
      size++;
      return true;
   }

 }
```

```
Bag bag = new LinkedBag();
```




```
bag.add("A");
bag.add("B");
bag.add("C");
```

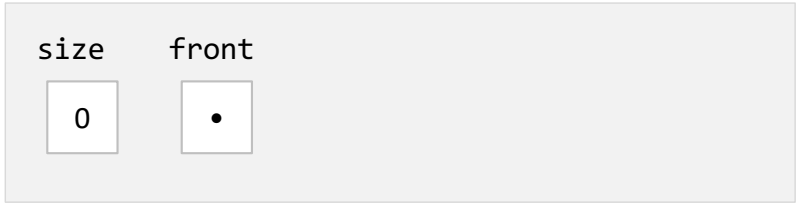


Contains and remove

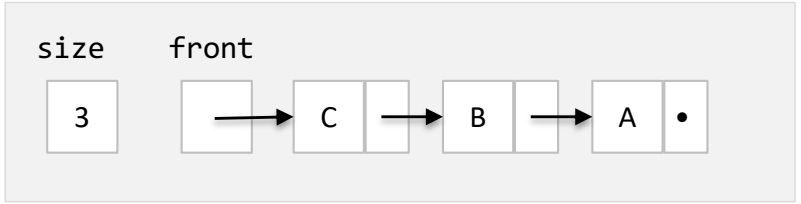
LinkedBag – contains

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean contains(T element) {









   }


 }
```

```
Bag bag = new LinkedBag();
```



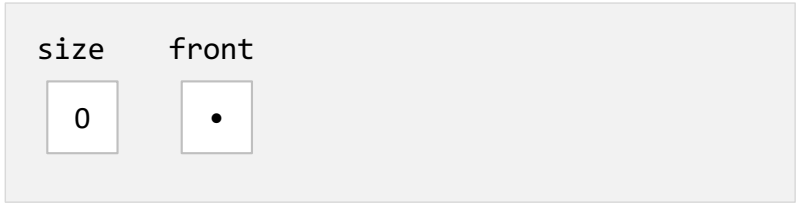

```
bag.add("A");
bag.add("B");
bag.add("C");
```



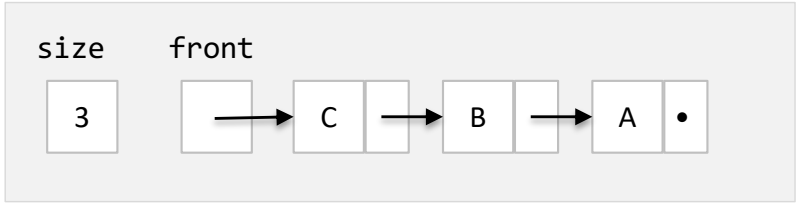
LinkedBag – contains

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean contains(T element) {
      Node p = front;
      while (p != null) {
         if (p.element.equals(element)) {
            return true;
         }
         p = p.next;
      }
      return false;
   }

}
```

```
Bag bag = new LinkedBag();
```

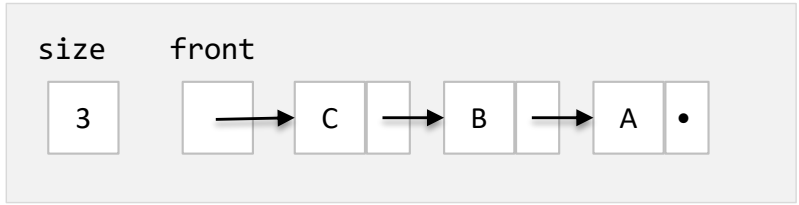



```
bag.add("A");
bag.add("B");
bag.add("C");
```

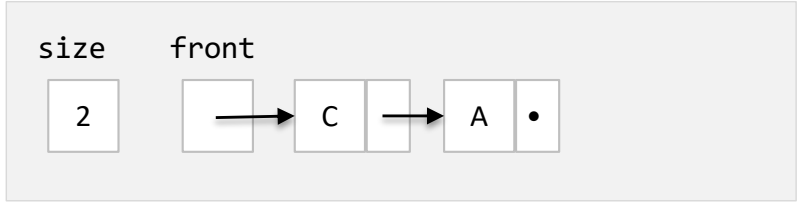


LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {




   }
}
```

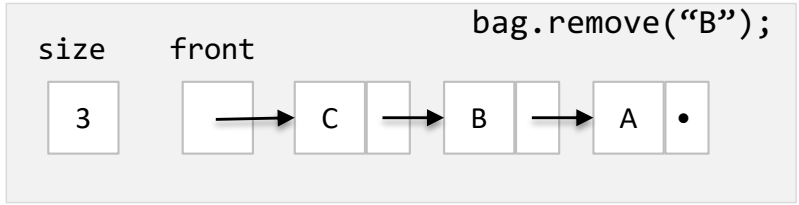



```
bag.remove("B");
```



LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      attempt to locate element

      unable to locate

      located, so remove it

   }
}
```



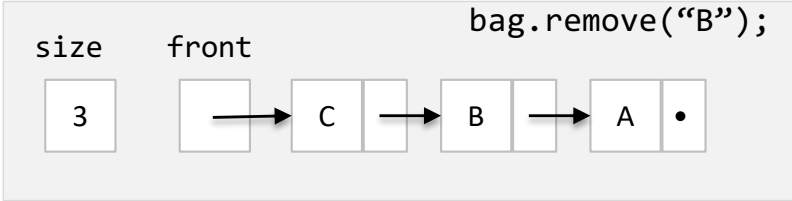
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```

attempt to locate element

unable to locate

located, so remove it



LinkedBag – remove

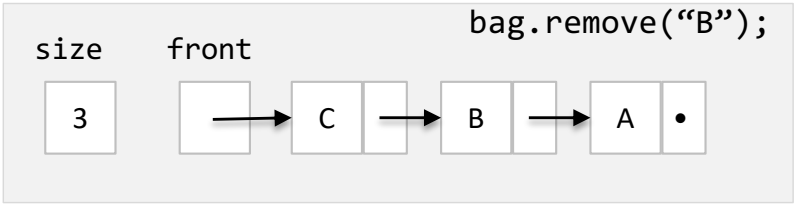
```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```

attempt to locate element

- *n will point to the node containing element or be null.*
- *prev will drag behind n*

unable to locate

located, so remove it



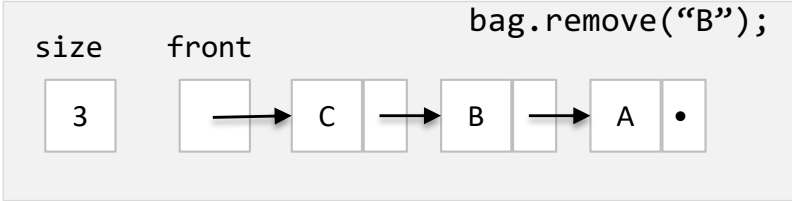
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```

attempt to locate element

unable to locate

located, so remove it



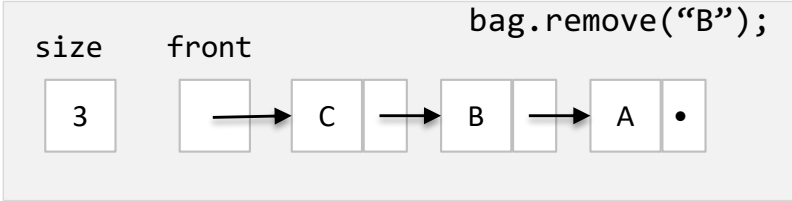
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```

attempt to locate element

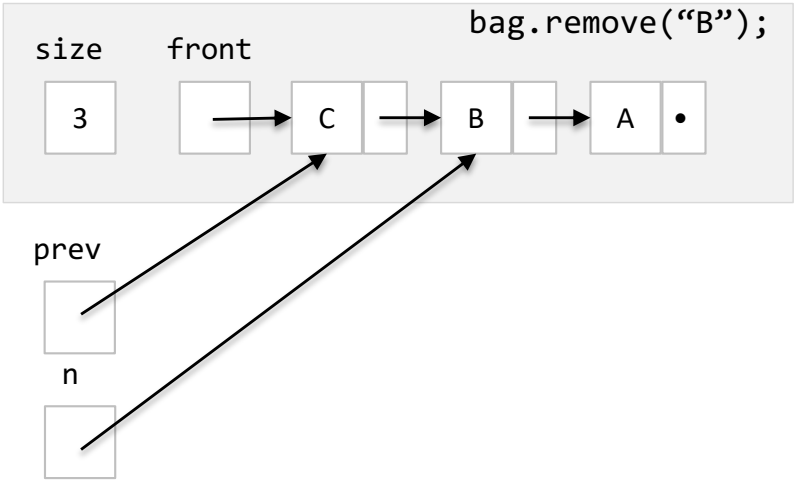
unable to locate

located, so remove it

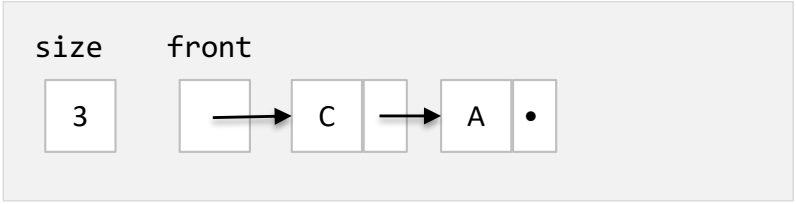


LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```



To delete the node that contains "B" we need a reference to its predecessor.



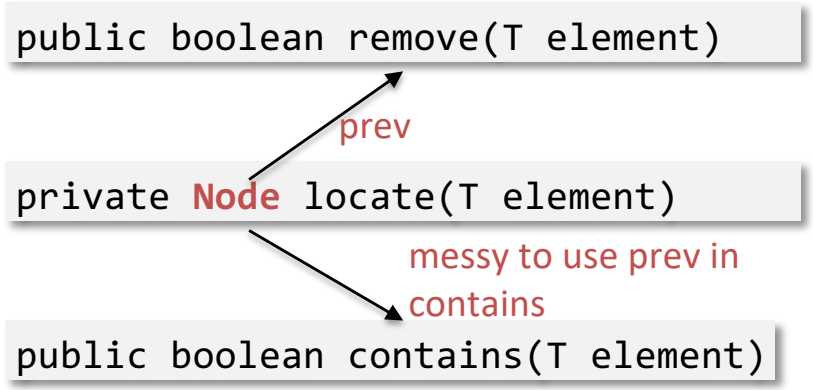
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```

Refactoring

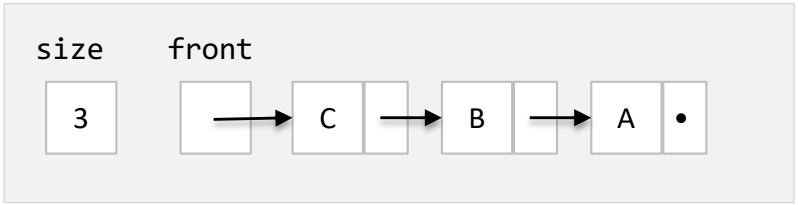
Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

But, the "drag behind" traversal using prev makes this more difficult and messy.



LinkedBag – doubly linked nodes

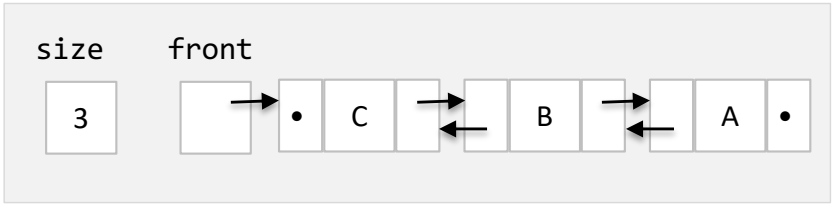
```
public class LinkedBag<T> implements Bag<T> {

   private Node front;
   private int size;
   . . .

   private class Node {
      private T element;
      private Node next;


   }

}
```



Singly linked

LinkedBag – doubly linked nodes

```
public class LinkedBag<T> implements Bag<T> {

   private Node front;
   private int size;
   . . .

   private class Node {
      private T element;
      private Node next;
      private Node prev;

   }

}
```



Doubly linked

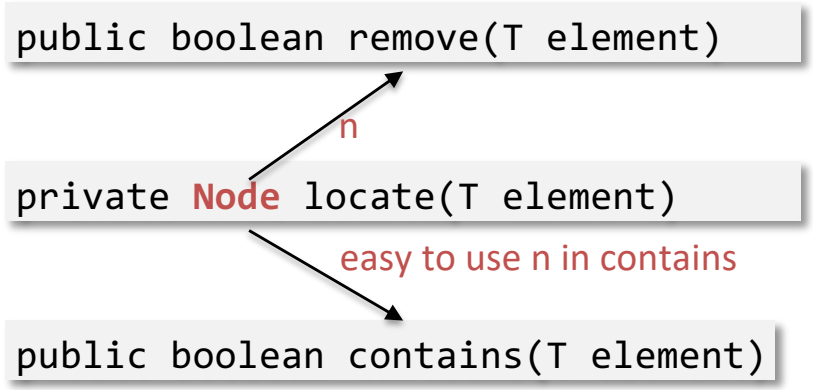
LinkedBag – refactoring

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = front;
      Node prev = null;
      while ((n != null) &&
             (!n.element.equals(element))) {
         prev = n;
         n = n.next;
      }
      if (n == null)   return false;
      if (n == front)  front = front.next;
      else             prev.next = n.next;
      size--;
      return true;
   }
}
```

Refactoring

Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

Having prev built into the node allows us to refactor and only return n.



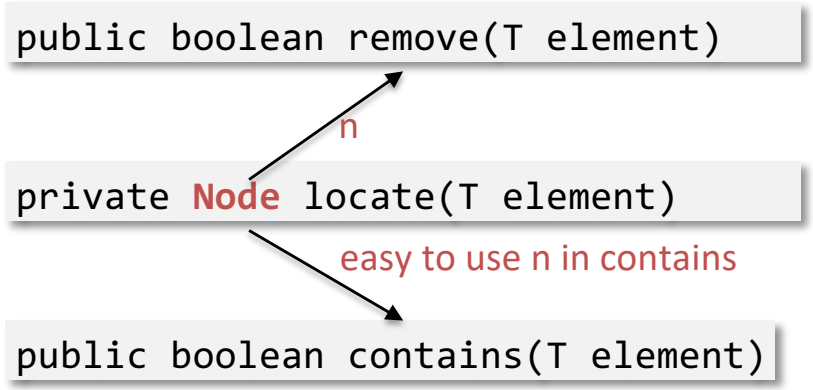
LinkedBag – refactoring

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   private Node locate(T element) {
      Node n = front;
      while (n != null) {
         if (n.element.equals(element))
            return n;
         n = n.next;
      }
      return null;
   }

 }
```

Refactoring

Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

Having prev built into the node allows us to refactor and only return n.



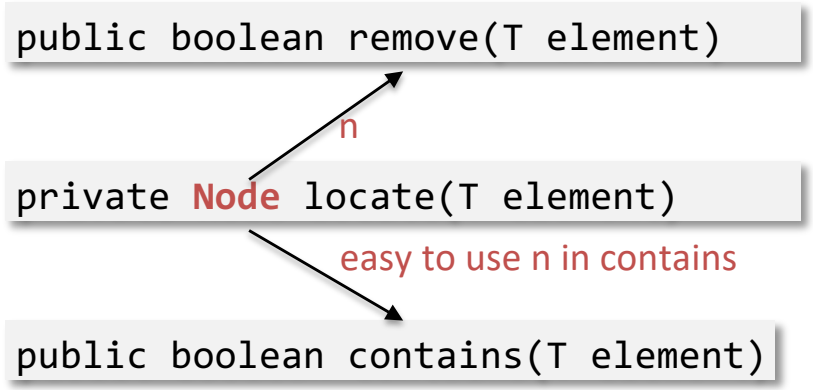
LinkedBag – contains

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean contains(T element) {
      return locate(element) != null;
   }

 }
```

Refactoring

Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

Having prev built into the node allows us to refactor and only return n.



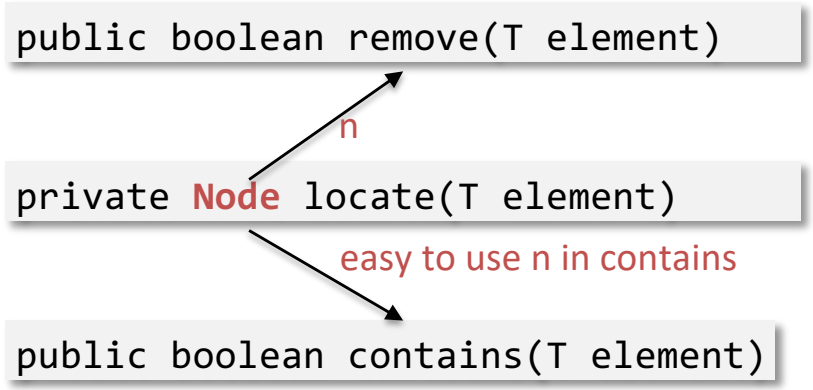
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      Node n = locate(element);

      if (n == null)   return false;

      if (n == front)  front = front.next;
      else             prev.next = n.next;

      size--;
      return true;
   }


 }
```

Refactoring

Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

Having prev built into the node allows us to refactor and only return n.



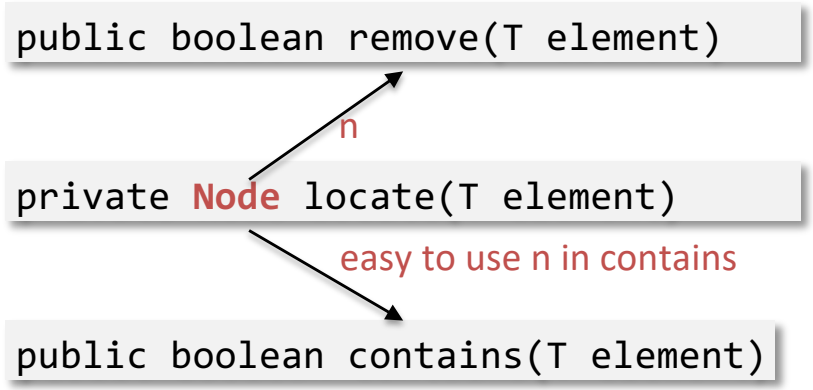
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```

Refactoring

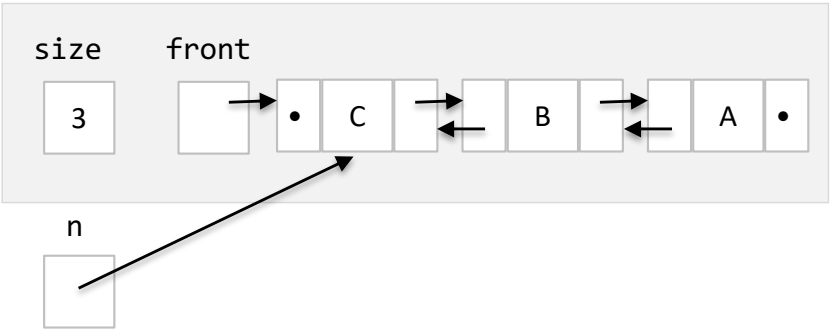
Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

Having prev built into the node allows us to refactor and only return n.



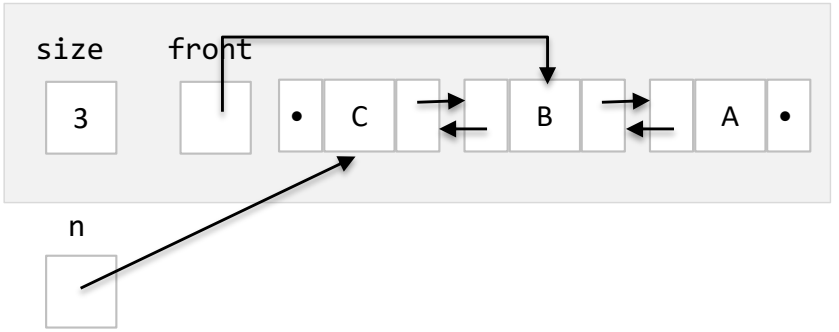
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



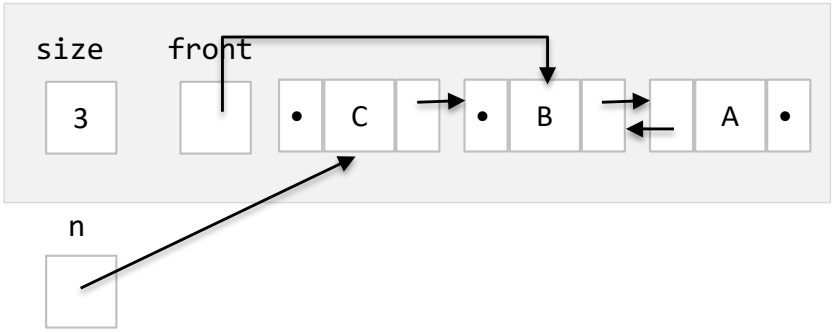
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



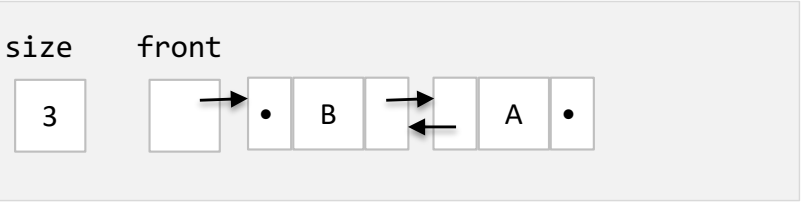
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



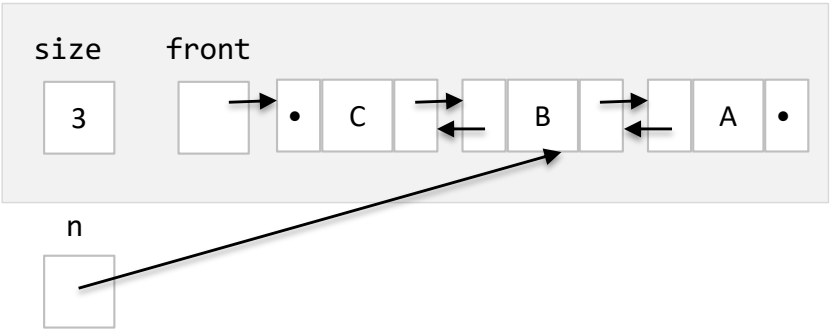
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



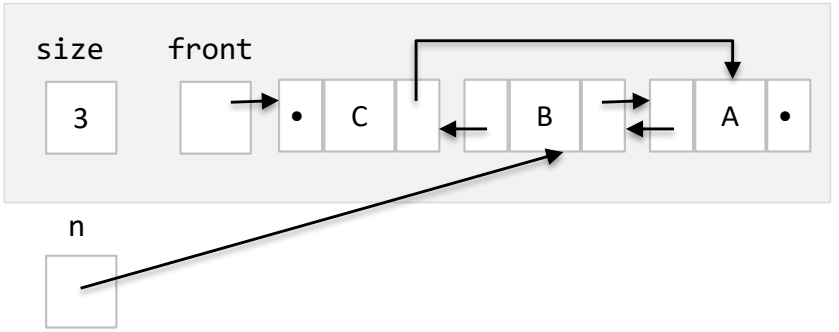
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



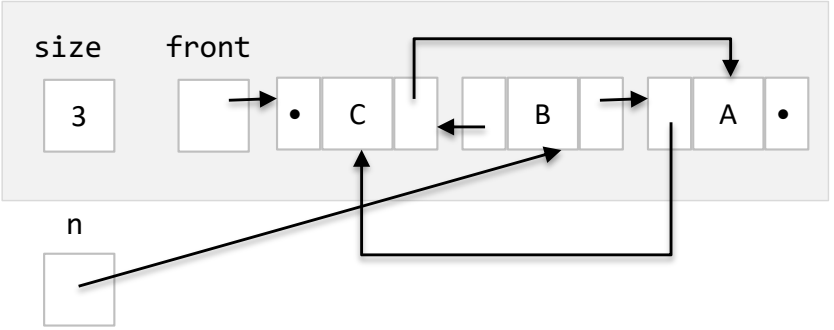
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



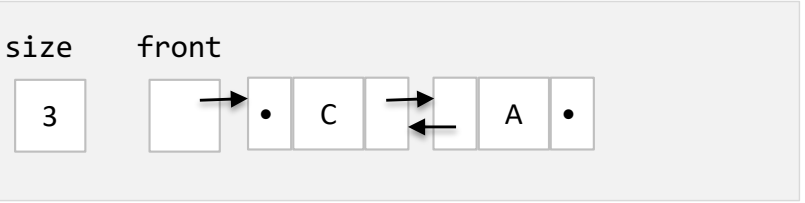
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



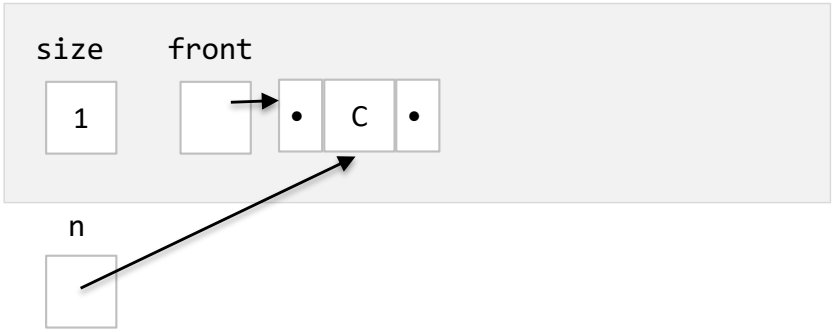
LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```

What if size == 1?

LinkedBag – remove

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean remove(T element) {
      ...
      if (n == front) {
         front = front.next;
         front.prev = null;
      }
      else {
         n.prev.next = n.next;
         if (n.next != null) {
            n.next.prev = n.prev;
         }
      }
      ...
   }
}
```



LinkedBag – remove

```
public boolean remove(T element) {
   Node n = locate(element);
   if (n == null) {
      return false;
   }

   if (size == 1) {
      front = null;
      size = 0;
      return true;
   }

   if (n == front) {
      front = front.next;
      front.prev = null;
   }
   else {
      n.prev.next = n.next;
      if (n.next != null) {
         n.next.prev = n.prev;
      }
   }
   size--;
   return true;
}
```

Using doubly-linked nodes is a good thing, but it creates more structural considerations and special cases.

LinkedBag – add

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public boolean add(T element) {

      Node n = new Node(element);
      n.next = front;
      if (front != null) {
         front.prev = n;
      }
      front = n;
      size++;
      return true;
   }


 }
```

Refactoring

The add method will have to change to account for the doubly linked node.

Iterator

LinkedBag – iterator

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public Iterator<T> iterator() {

   }

}
```

```
class LinkedIterator
      implements Iterator<T>
```

Nested class

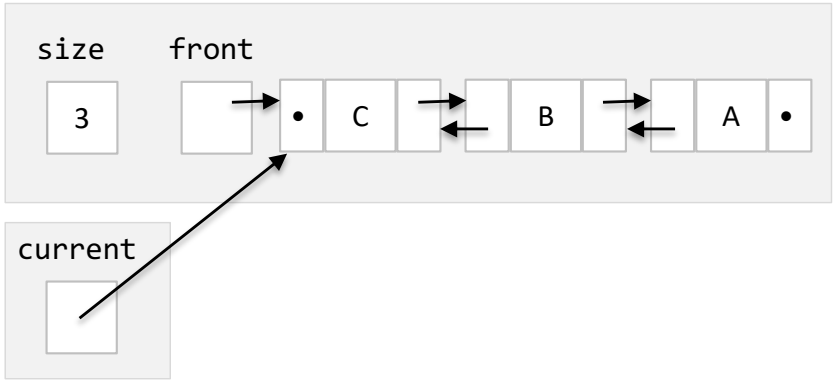
Has access to private fields; don't have to expose them in any way.

Top-level class

Can be used by different collection classes.

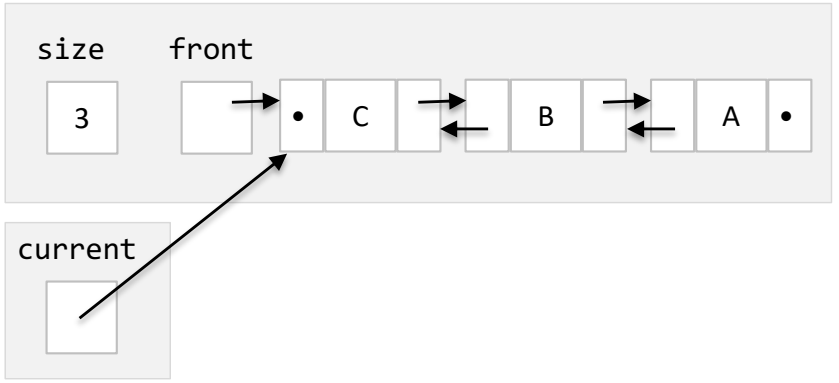
LinkedBag – iterator

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   public Iterator<T> iterator() {
      return new LinkedIterator();
   }

   private class LinkedIterator
                    implements Iterator<T> {
      private Node current = front;

      public boolean hasNext() { ... }
      public T next() { ... }
      public void remove() { ... }
   }

 }
```



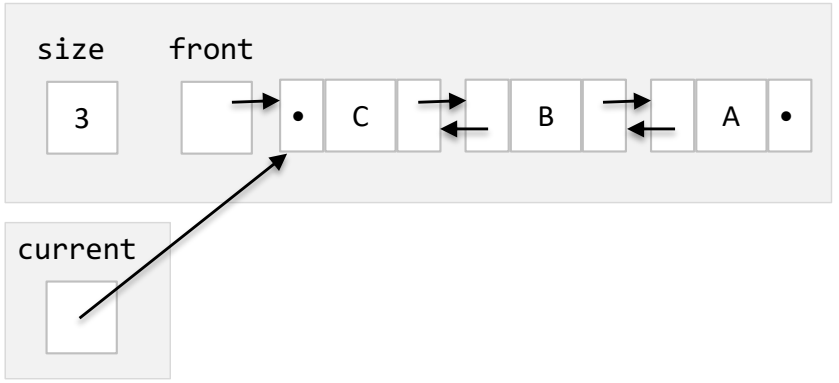
LinkedBag – iterator

```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   private class LinkedIterator
                    implements Iterator<T> {
      private Node current = front;

      public boolean hasNext() {
         return current != null;
      }

      public void remove() {
         throw new
            UnsupportedOperationException();
      }
   }
}
```



LinkedBag – iterator

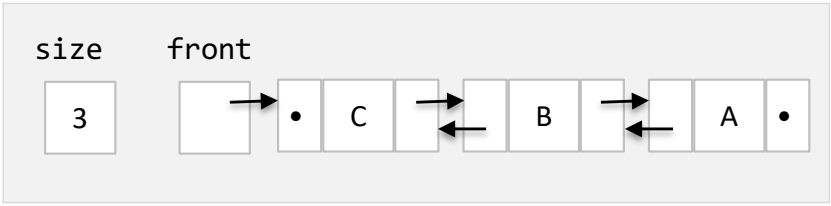
```
public class LinkedBag<T> implements Bag<T> {
   private Node front;
   private int size;

   private class LinkedIterator
                    implements Iterator<T> {
      private Node current = front;

      public T next() {
         if (!hasNext())
            throw new
                   NoSuchElementException();

         T result = current.element;
         current = current.next;
         return result;
      }
   }
}
```



Summary, variations, and performance

LinkedBag

```
public class LinkedBag<T> implements Bag<T> {

   private Node front;
   private int size;
   . . .
}
```



Advantages of using linked nodes:

- Given a reference to a node, efficient to insert or add before or after that node; no shifting required

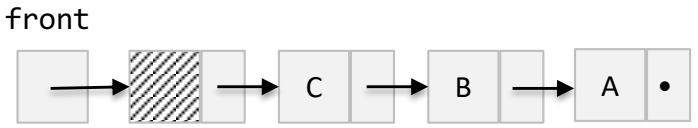
Disadvantages of using linked nodes:

- no random access;
- less efficient use of memory
- not built in; nodes are user-created

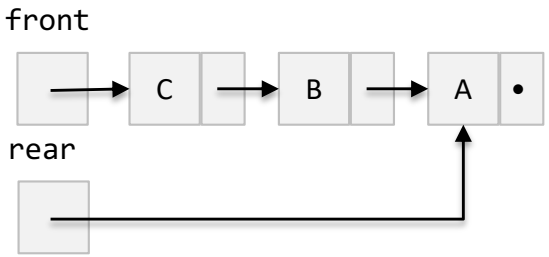
Common variations

Singly linked v. Doubly linked

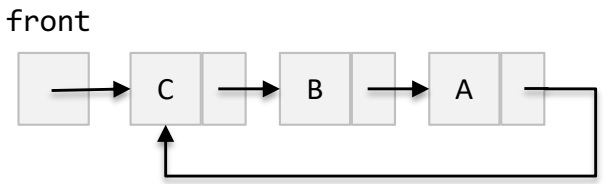
Dummy/Header nodes



Front and rear pointers



Circular



Plus many others, and various combinations of these …

Performance

Bag method	ArrayBag	LinkedBag
boolean **add**(T element)	O(1)*	O(1)
boolean **remove**(T element)	O(N)	O(N)
boolean **contains**(T element)	O(N)	O(N)
int **size**()	O(1)	O(1)
boolean **isEmpty**()	O(1)	O(1)
Iterator<T> **iterator**()	O(1)	O(1)

Make sure your understand why, both at the code level and the conceptual level.

**amortized cost*

No real difference in time performance with either implementation, except for the amortized cost of the array-based add. The linked implementation will have a higher memory overhead, however.

Performance

	Bag Collection		Set Collection	
Interface method	**Array**	**Nodes**	**Array**	**Nodes**
boolean **add**(T element)	O(1)	O(1)		
boolean **remove**(T element)	O(N)	O(N)		
boolean **contains**(T element)	O(N)	O(N)		
int **size**()	O(1)	O(1)		
boolean **isEmpty**()	O(1)	O(1)		
Iterator<T> **iterator**()	O(1)	O(1)		

Performance

	Bag Collection		Set Collection	
Interface method	**Array**	**Nodes**	**Array**	**Nodes**
boolean **add**(T element)	O(1)	O(1)	O(N)	O(N)
boolean **remove**(T element)	O(N)	O(N)	O(N)	O(N)
boolean **contains**(T element)	O(N)	O(N)	O(N)	O(N)
int **size**()	O(1)	O(1)	O(1)	O(1)
boolean **isEmpty**()	O(1)	O(1)	O(1)	O(1)
Iterator<T> **iterator**()	O(1)	O(1)	O(1)	O(1)

Performance

	Bag Collection		Set Collection			
method	**Array**	**Nodes**	**Array**	**Nodes**	**Ordered Array**	**Ordered Nodes**
add	O(1)	O(1)	O(N)	O(N)		
remove	O(N)	O(N)	O(N)	O(N)		
contains	O(N)	O(N)	O(N)	O(N)		
size	O(1)	O(1)	O(1)	O(1)		
isEmpty	O(1)	O(1)	O(1)	O(1)		
iterator	O(1)	O(1)	O(1)	O(1)		

Performance

	Bag Collection		Set Collection			
method	**Array**	**Nodes**	**Array**	**Nodes**	**Ordered Array**	**Ordered Nodes**
add	O(1)	O(1)	O(N)	O(N)	O(N)	O(N)
remove	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
contains	O(N)	O(N)	O(N)	O(N)	**O(log N)**	O(N)
size	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
iterator	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

Performance

	Bag Collection		Set Collection			
method	**Array**	**Nodes**	**Array**	**Nodes**	**Ordered Array**	**Ordered Nodes**
add	O(1)	O(1)	O(N)	O(N)	O(N)	~~O(N)~~
remove	O(N)	O(N)	O(N)	O(N)	O(N)	~~O(N)~~
contains	O(N)	O(N)	O(N)	O(N)	**O(log N)**	~~O(N)~~
size	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
iterator	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

If we could use binary search on a node-based data structure, then add(), remove(), and contains() would all be **O(log N)**. *[more to come…]*