



AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Node-based Bag

COMP 2210 – Dr. Hendrix

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();  
}
```

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();  
}
```

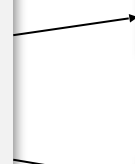


```
public class ArrayBag<T> implements Bag<T> {  
    private T[] elements;  
    . . .  
}
```

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*

Linked Bag

```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public LinkedBag() {  
  
        size = 0;  
    }  
  
}
```

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

Linked Bag

```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public LinkedBag() {  
        front = null;  
        size = 0;  
    }  
  
}
```

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

Linked Bag

```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public LinkedBag() {  
        front = null;  
        size = 0;  
    }  
  
}
```

***No memory
allocation!***

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


Linked Bag

```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public LinkedBag() {  
        front = null;  
        size = 0;  
    }  
  
}
```

***No memory
allocation!***

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

size front

0

•

Linked Bag

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

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

***No memory
allocation!***

```
}
```

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

size	front
0	•

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

Linked Bag

```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public LinkedBag() {  
        front = null;  
        size = 0;  
    }  
  
}
```

***No memory
allocation!***

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

size front

0

•

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

size front

3

→ C → B → A •

Linked Bag

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

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

```
    public int size() {  
        return size;  
    }
```

```
}
```

***No memory
allocation!***

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

size front

0

•

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

size front

3

→ C → B → A •

Linked Bag

```
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();
```

size front

0

•

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

size front

3

→ C → B → A •

Linked Bag

```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public boolean add(T element) {  
  
    }  
  
}
```

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

size front

0

•

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

size front

3

→

C

→

B

→

A

•

Linked Bag

```
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();
```

size front

0

•

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

size front

3

→

C

→

B

→

A

•

Linked Bag

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

    public boolean contains(T element) {

    }

}
```

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

size	front
0	•

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

size front

3 → C → B → A •

Linked Bag

```
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();
```

size front

0

•

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

size front

3

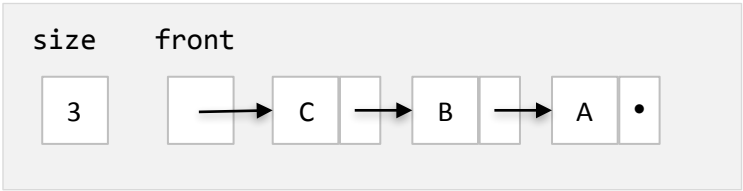
→ C → B → A •

Linked Bag

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

    public boolean remove(T element) {

    }
}
```

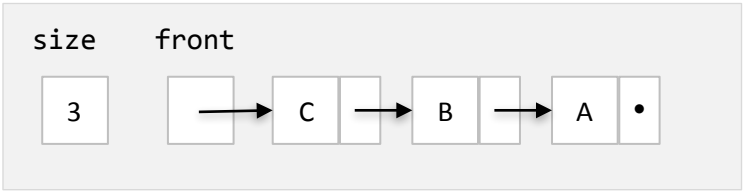


Linked Bag

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

    public boolean remove(T element) {

    }
}
```



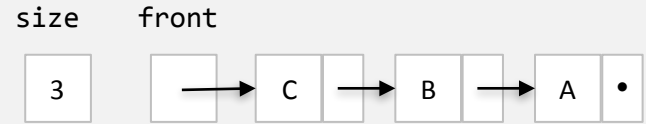
```
bag.remove("B");
```

Linked Bag

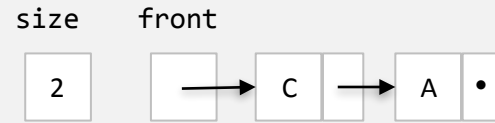
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    public boolean remove(T element) {

    }
}
```



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

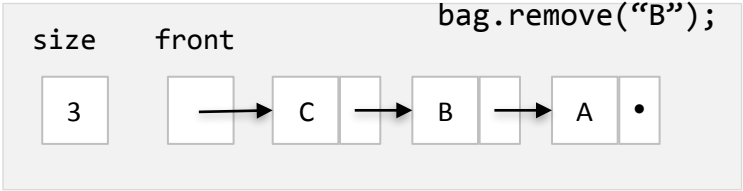


Linked Bag

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

    public boolean remove(T element) {

    }
}
```

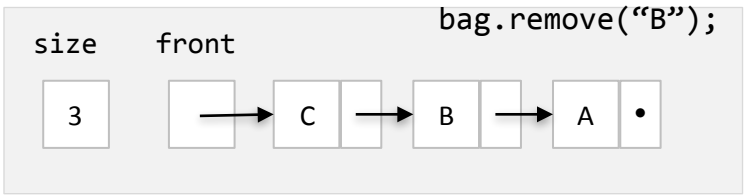


Linked Bag

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

    public boolean remove(T element) {

    }
}
```

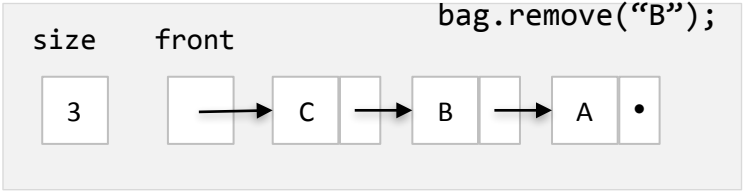


Linked Bag

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

    public boolean remove(T element) {

    }
}
```



attempt to locate element

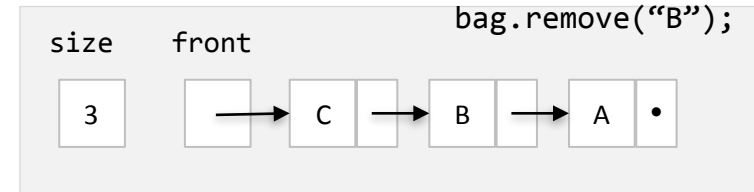
unable to locate

located, so remove it

Linked Bag

```
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;
    }
}
```



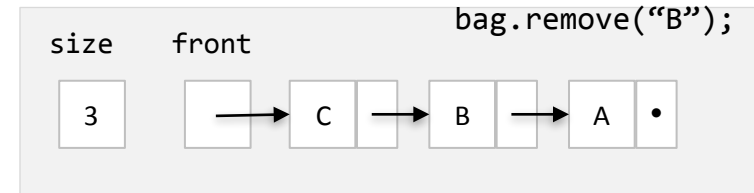
attempt to locate element

unable to locate

located, so remove it

Linked Bag

```
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;  
    }  
}
```



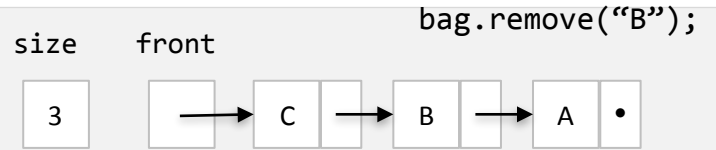
attempt to locate element

unable to locate

located, so remove it

Linked Bag

```
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*

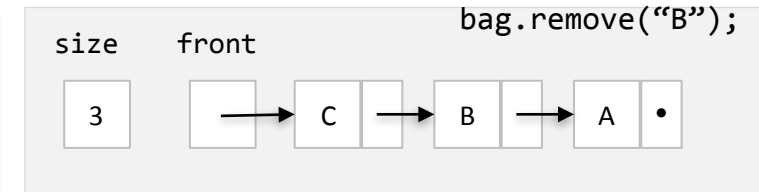
unable to locate

located, so remove it

Linked Bag

```
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

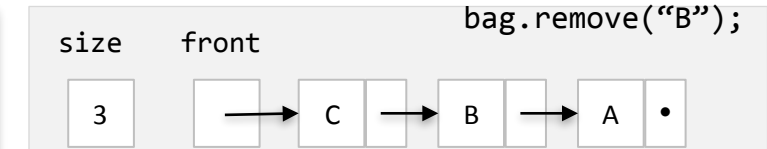
unable to locate

located, so remove it

Linked Bag

```
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

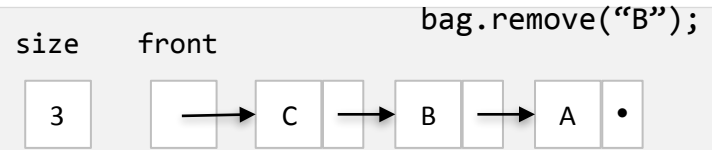
unable to locate

located, so remove it

Linked Bag

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

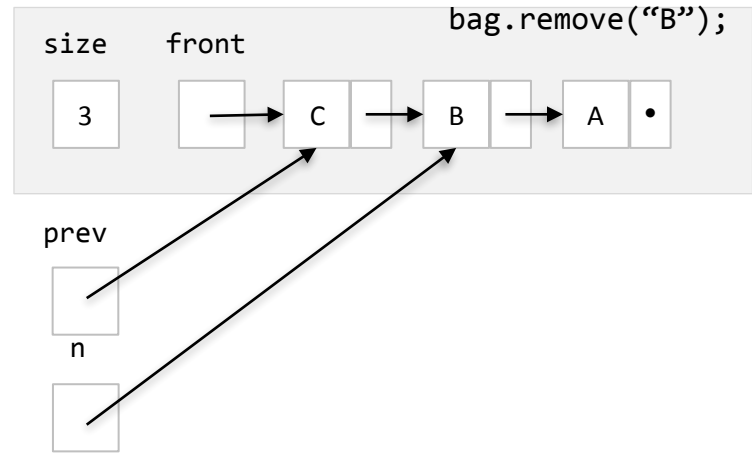
    public boolean remove(T element) {
        Node n = front;
        Node prev = null;
        while ((n != null) &&
            (!n.element.equals(element))) {
            prev = n;
            n = n.next;
        }
        if (n == null) return false;
        if (n == front) front = front.next;
    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



Linked Bag

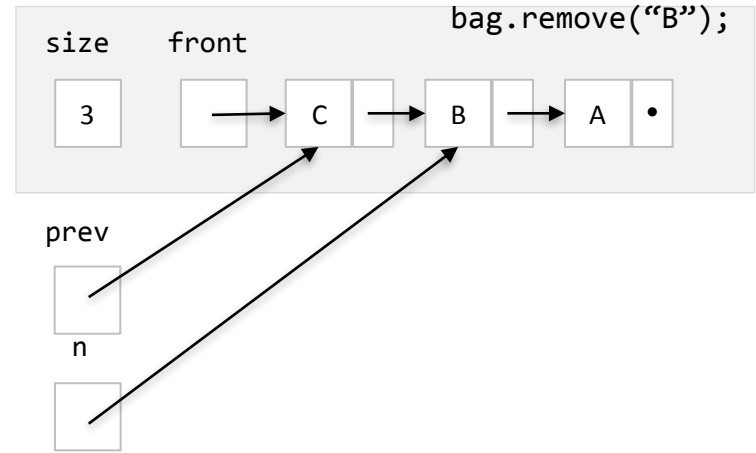
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    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;
    }
}
```



Linked Bag

```
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;  
    }  
}
```

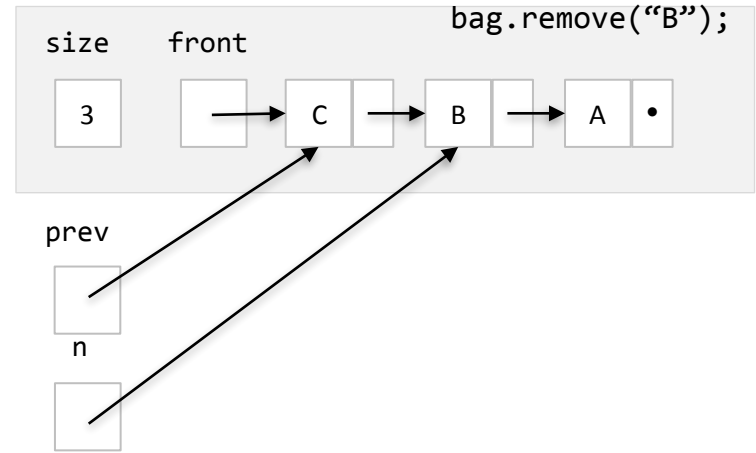


To delete the node that contains "B" we need a reference to its predecessor.

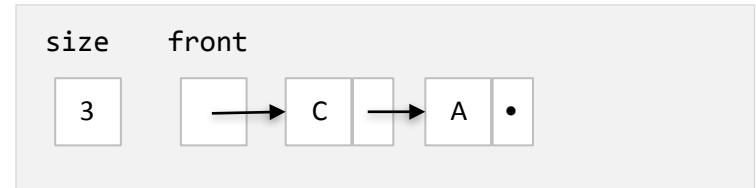
Linked Bag

```
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.




```
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.

```
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.

```
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.

```
private Node locate(T element)
```

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```

```
public boolean contains(T element)
```

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```



```
public boolean contains(T element)
```

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```



```
public boolean contains(T element)
```

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```

```
public boolean contains(T element)
```

`prev`



Linked Bag

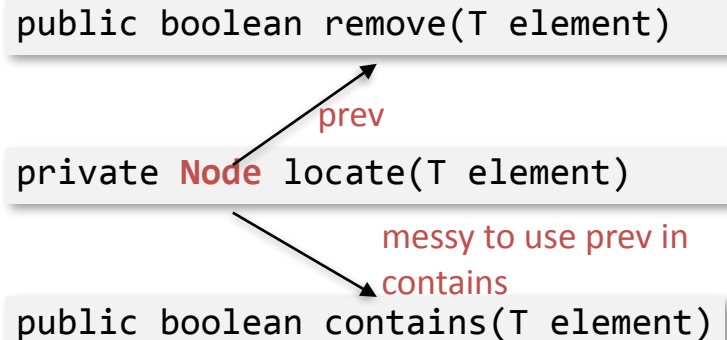
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    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

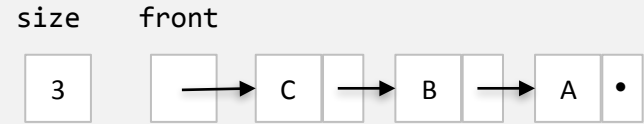
Just as in the array-based implementation, the linear search common to both `contains` and `remove` is an obvious candidate for refactoring.

But, the “drag behind” traversal using `prev` makes this more difficult and messy.



LinkBag

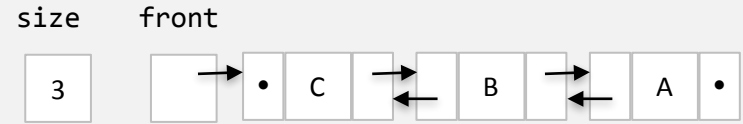
```
public class LinkBag<T> implements Bag<T> {  
  
    private Node front;  
    private int size;  
    . . .  
  
    private class Node {  
        private T element;  
        private Node next;  
    }  
}
```



Singly linked

LinkBag

```
public class LinkBag<T> implements Bag<T> {  
  
    private Node front;  
    private int size;  
    . . .  
  
    private class Node {  
        private T element;  
        private Node next;  
        private Node prev;  
    }  
}
```



Doubly linked

```
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.

```
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.

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```

```
public boolean contains(T element)
```

Linked Bag

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```



```
public boolean contains(T element)
```

Linked Bag

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```



```
public boolean contains(T element)
```

Linked Bag

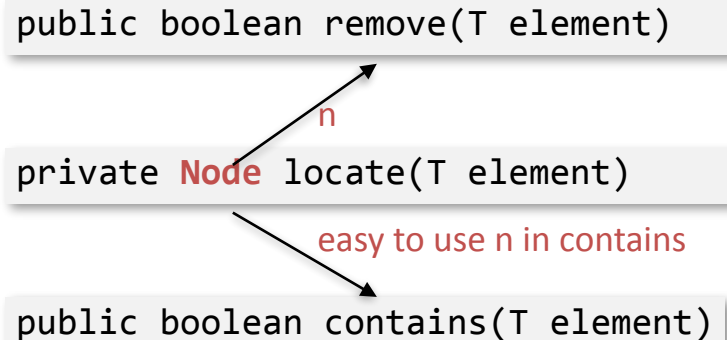
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    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.



Linked Bag

```
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`.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```

```
public boolean contains(T element)
```

`n`

easy to use `n` in `contains`

Linked Bag

```
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.

```
public boolean remove(T element)
```

```
private Node locate(T element)
```

```
public boolean contains(T element)
```

n

easy to use n in contains

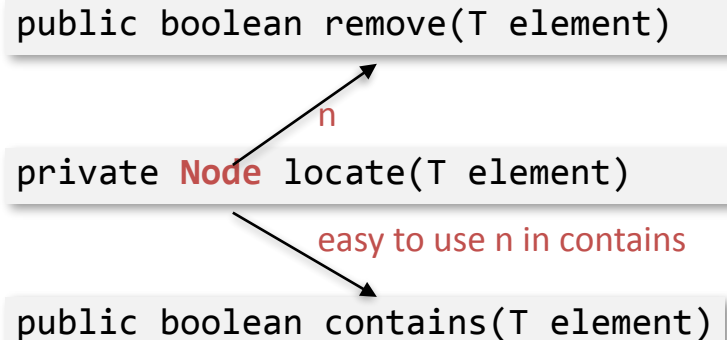
Linked Bag

```
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`.



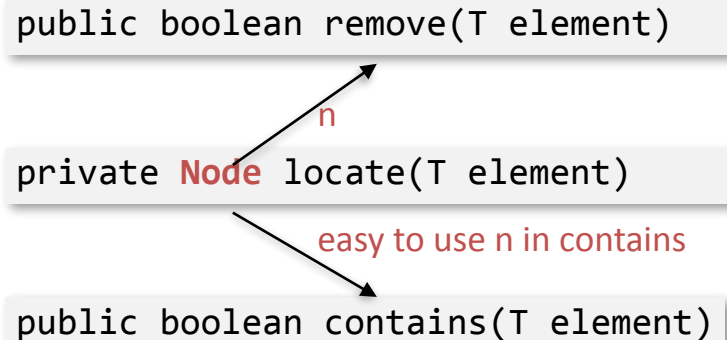
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    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.



Linked Bag

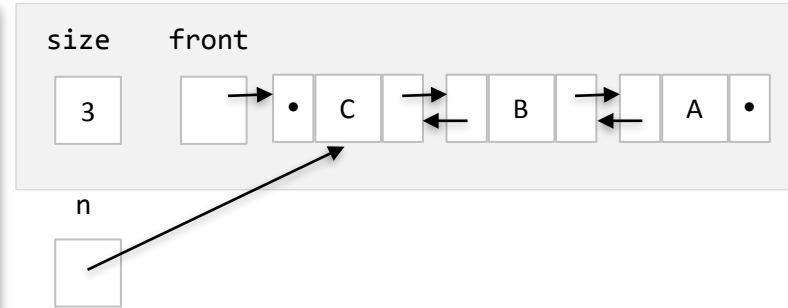
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    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;
            }
        }
        ...
    }
}
```

Linked Bag

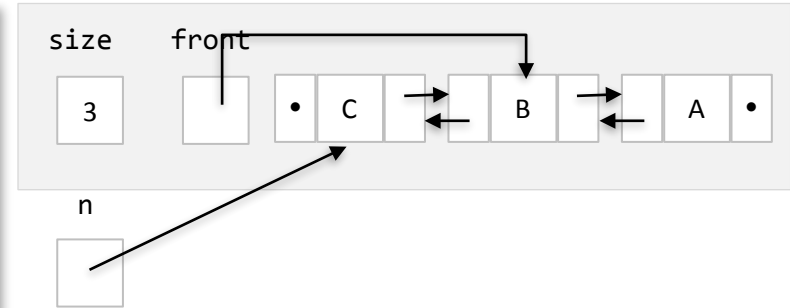
```
public class LinkedBag<T> implements Bag<T> {
    private Node front;
    private int size;

    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;
            }
        }
        ...
    }
}
```



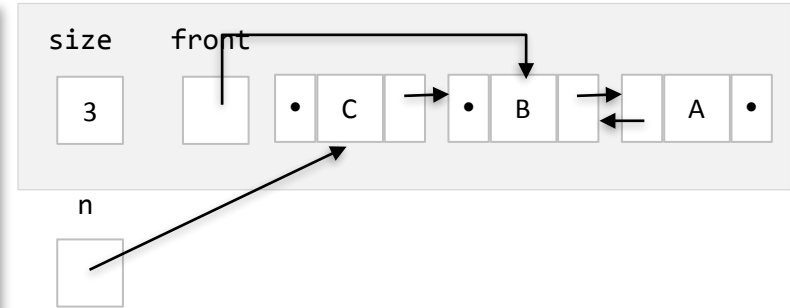
Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



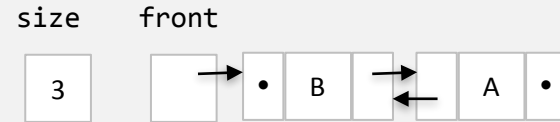
Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



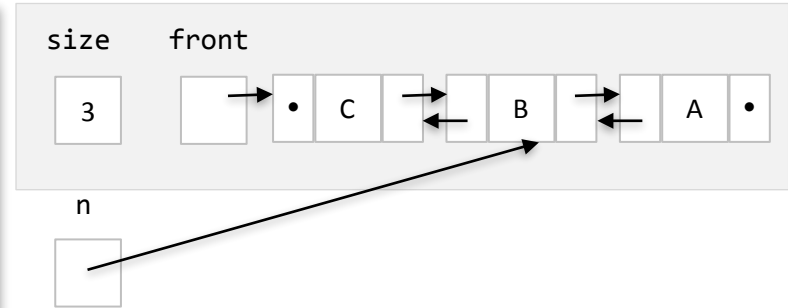
Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



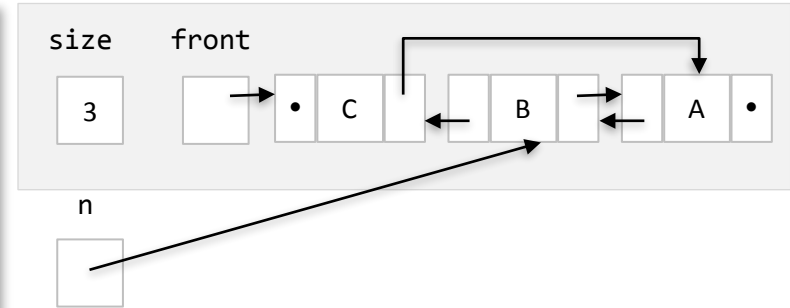
Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



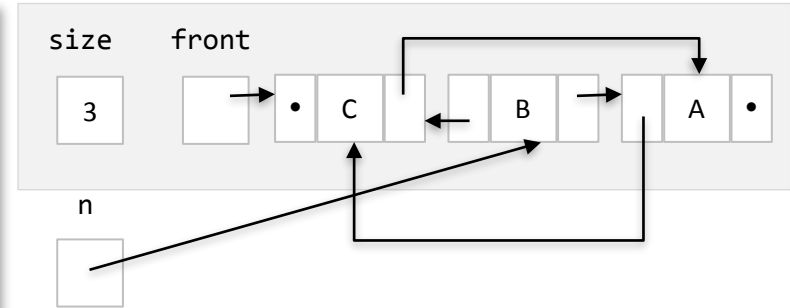
Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



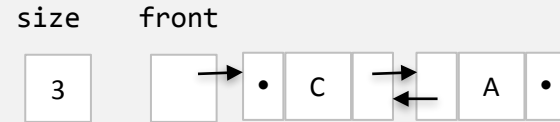
Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



Linked Bag

```
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;
            }
        }
        ...
    }
}
```

Linked Bag

```
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;
            }
        }
        ...
    }
}
```

Linked Bag

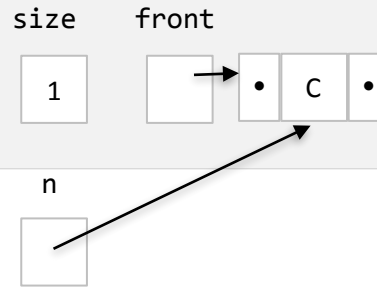
```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public boolean remove(T element) {  
        ...  
        if (n == front) {  
            front = front.next;  
            front.prev = null;  
        }  
        else {  
            n.prev.next = n.next;  
            if (n.next != null) {  
                n.next.prev = n.prev;  
            }  
        }  
        ...  
    }  
}
```



What if size == 1?

Linked Bag

```
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;  
            }  
        }  
        ...  
    }  
}
```



Linked Bag

```
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;
}
```

Linked Bag

```
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.



Linked Bag

```
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.

Linked Bag

```
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.*

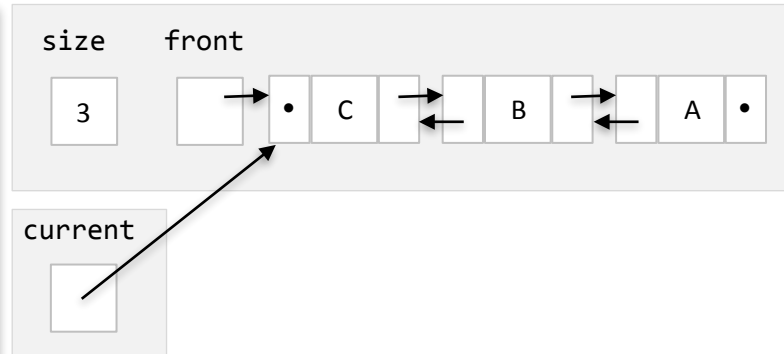
}

Linked Bag

```
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() { ... }  
    }  
}
```



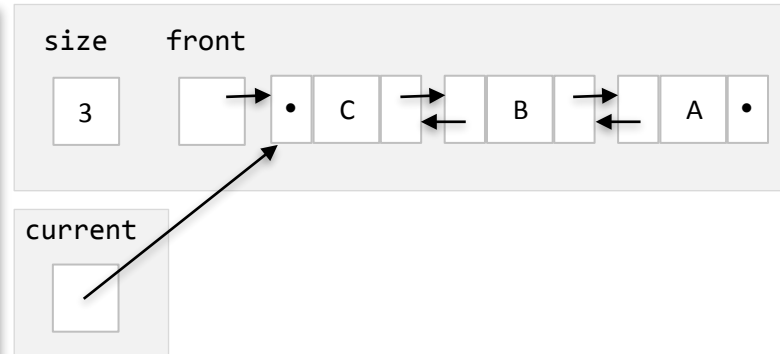
Linked Bag

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

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

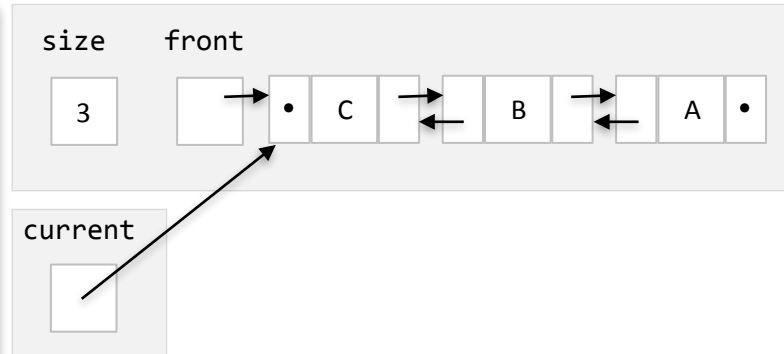
        public boolean hasNext() {
            return current != null;
        }

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



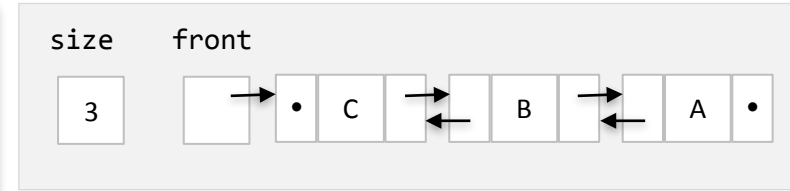
Linked Bag

```
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;  
        }  
    }  
}
```



LinkBag

```
public class LinkBag<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

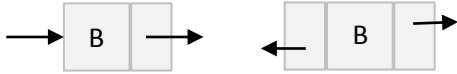
Disadvantages of using linked nodes:

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

Common variations

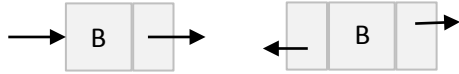
Common variations

Singly linked v. Doubly linked

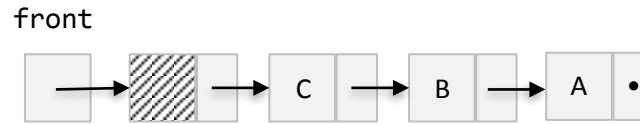


Common variations

Singly linked v. Doubly linked

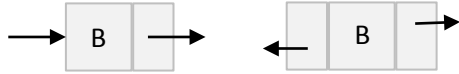


Dummy/Header nodes

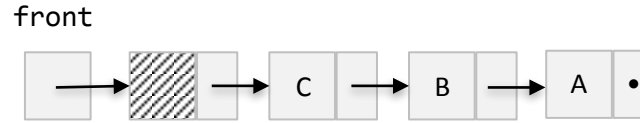


Common variations

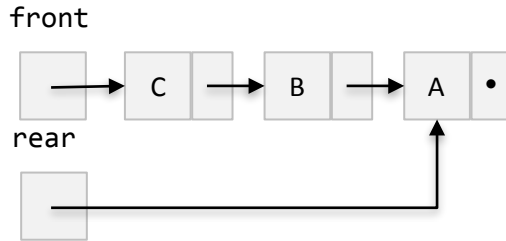
Singly linked v. Doubly linked



Dummy/Header nodes

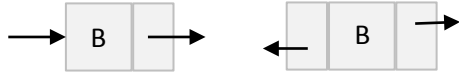


Front and rear pointers

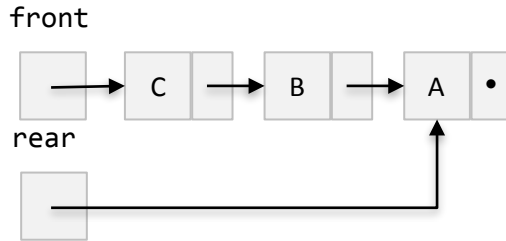


Common variations

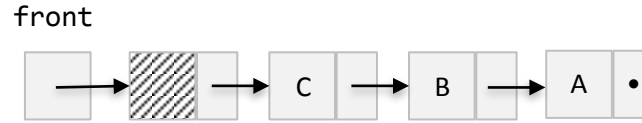
Singly linked v. Doubly linked



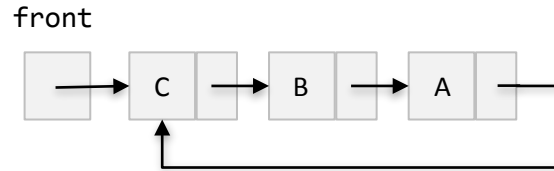
Front and rear pointers



Dummy/Header nodes

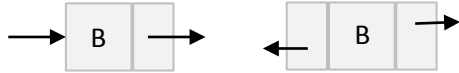


Circular

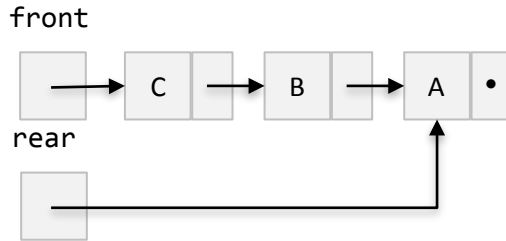


Common variations

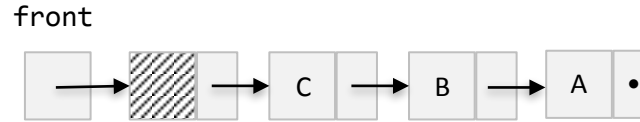
Singly linked v. Doubly linked



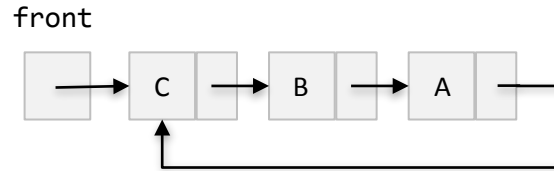
Front and rear pointers



Dummy/Header nodes



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 ()	$O(1)$	$O(1)$

**amortized cost*

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*

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

Interface method	Bag Collection		Set Collection	
	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

Interface method	Bag Collection		Set Collection	
	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	$O(1)$	$O(1)$	$O(N)$	$O(N)$	$O(N)$	$\Theta(N)$
remove	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$\Theta(N)$
contains	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(\log N)$	$\Theta(N)$
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...]*