



Linked Structures

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();  
}
```

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

```
    private T[] elements;
```

```
    . . .
```

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

```
    private ???;
```

```
    . . .
```

ArrayBag

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

size	elements
3	A B C • • • • • • •
	0 1 2 3 4 5 6 7 8 9

A storage scheme using dynamic memory will address these disadvantages at the cost of losing random access.

Advantages of using an array:

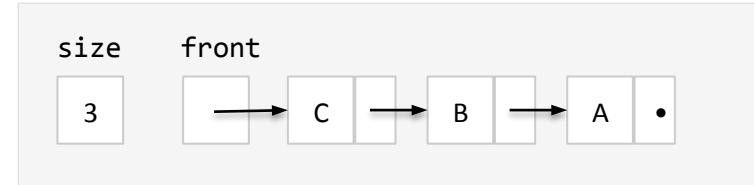
- fast random access to any element
- efficient use of memory
- built into the language; a “common currency” for any data storage scheme

Disadvantages of using an array:

- inefficient to insert or delete anywhere but the end; must shift left/right
- need to “resize” when full/sparse

LinkedBag

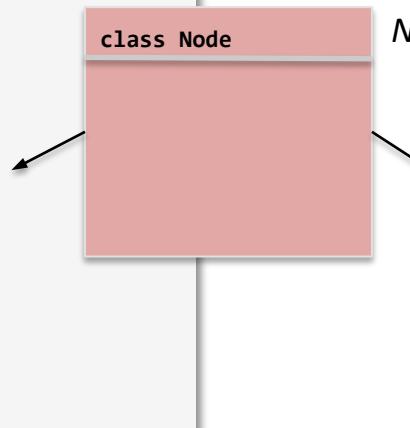
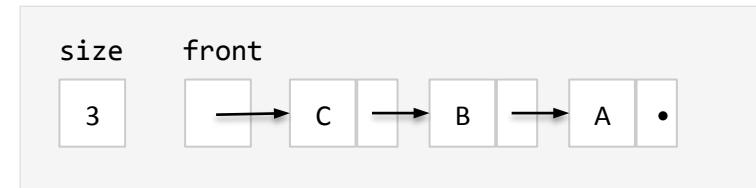
```
public class LinkedBag<T> implements Bag<T> {  
  
    private ??? front;  
    private int size;  
    . . .  
}
```



Individual containers are explicitly linked together. Each container holds one element and a reference to another container.

LinkedBag

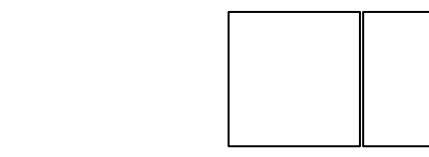
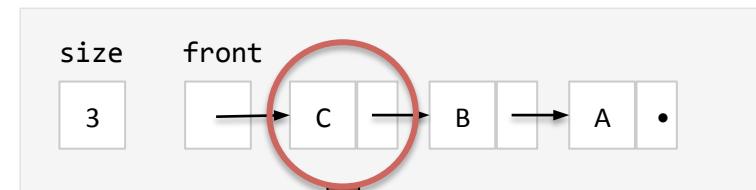
```
public class LinkedBag<T> implements Bag<T> {  
  
    private Node front;  
    private int size;  
    . . .  
  
    private class Node {  
        . . .  
    }  
}
```



Nested or top-level?

LinkedBag

```
public class LinkedBag<T> implements Bag<T> {  
  
    private Node front;  
    private int size;  
    . . .  
  
    private class Node {  
        private T element;  
        private Node next;  
        . . .  
    }  
  
    A recursive structure [more to come...]  
}
```



A reference to
the element/
value this node
stores.

A reference to
the next node in
the chain.

The Node class

```
private class Node {  
    private Object element;  
    private Node next;  
  
    public Node(Object e) {  
        element = e;  
    }  
  
    public Node(Object e, Node n) {  
        element = e;  
        next = n;  
    }  
}
```

Constructors, garbage

```
n = new Node(1);  
  
n = new Node(2, n);  
  
n = new Node(3);  
  
n = null;
```

The Node class

```
private class Node {  
    private Object element;  
    private Node next;  
  
    public Node(Object e) {  
        element = e;  
    }  
  
    public Node(Object e, Node n) {  
        element = e;  
        next = n;  
    }  
}
```

Basic linking

```
n = new Node(1);  
n = new Node(2, n);  
n.next = new Node(3, n.next);
```

```
n = new Node(1, new Node(2));  
n.next.next = new Node(3, null);  
n = new Node(4, n.next);
```

Participation



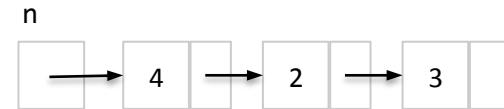
Q: Which chain of nodes is created by the following code?

```
n = new Node(1);  
  
n.next = new Node(2, new Node(3));  
  
n = new Node(4, n.next.next);
```

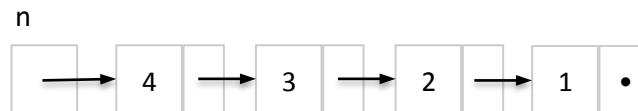
A.



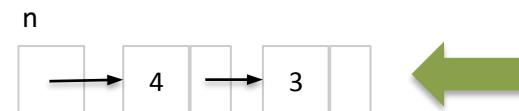
C.



B.

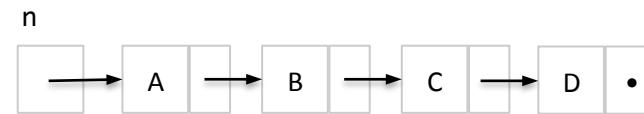


D.



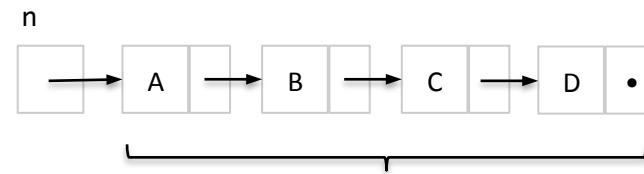
Calculating length

```
public int length(Node n) {  
    }  
}
```



Calculating length

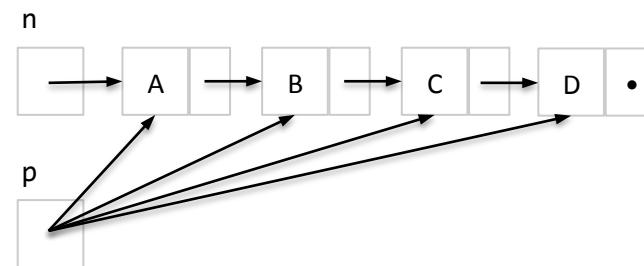
```
public int length(Node n) {  
}  
}
```



There are four nodes
reachable from n, so the
“length” of the chain is 4.

Calculating length

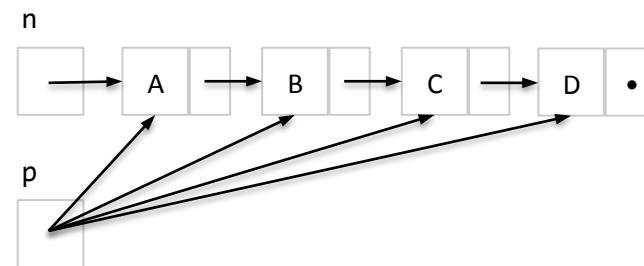
```
public int length(Node n) {  
    Node p = n;  
  
    while (p != null) {  
  
        p = p.next;  
    }  
}
```



This is a common traversal pattern that you will use in many different situations when you have to **traverse** the chain of nodes one by one.

Calculating length

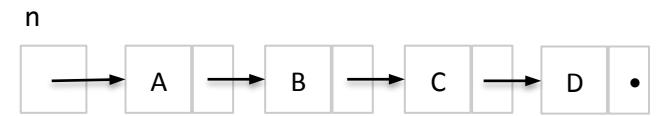
```
public int length(Node n) {  
    Node p = n;  
    int len = 0;  
    while (p != null) {  
        len++;  
        p = p.next;  
    }  
    return len;  
}
```



Simply add the statements that perform the desired computation as each node is accessed.

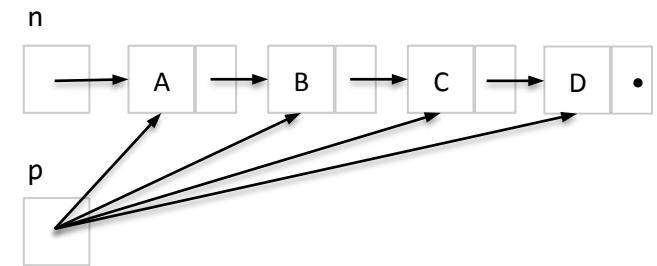
Linear search

```
public boolean contains(Node n, Object target) {  
    }  
}
```



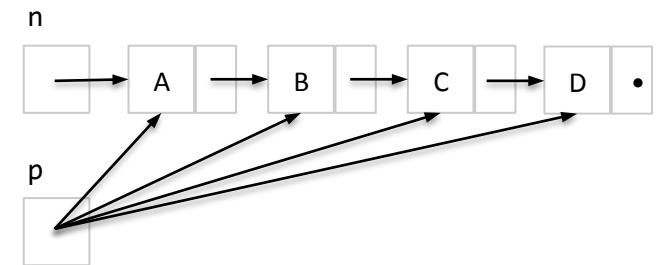
Linear search

```
public boolean contains(Node n, Object target) {  
    Node p = n;  
    while (p != null) {  
  
        p = p.next;  
    }  
}
```



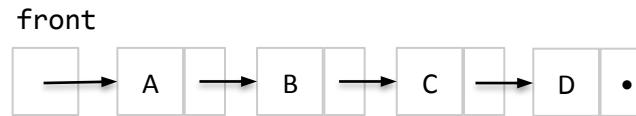
Linear search

```
public boolean contains(Node n, Object target) {  
    Node p = n;  
    while (p != null) {  
        if (p.element.equals(target)) {  
            return true;  
        }  
        p = p.next;  
    }  
    return false;  
}
```

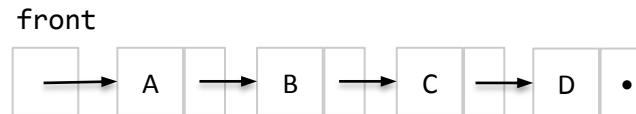


Inserting nodes

Inserting a new first node

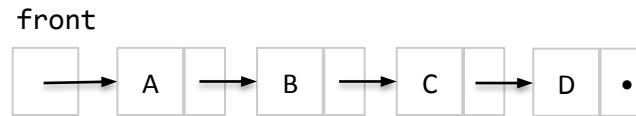


Inserting a new node somewhere else



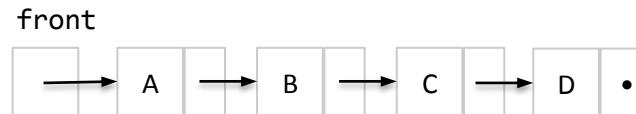
Inserting nodes

Inserting a new first node



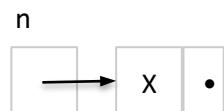
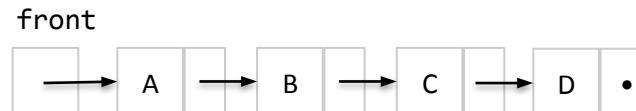
```
Node n = new Node("X");
```

Inserting a new node somewhere else



Inserting nodes

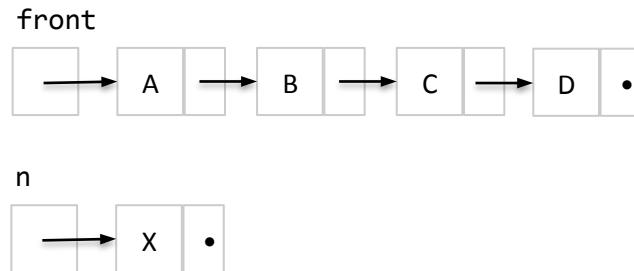
Inserting a new first node



```
Node n = new Node("X");
```

Inserting nodes

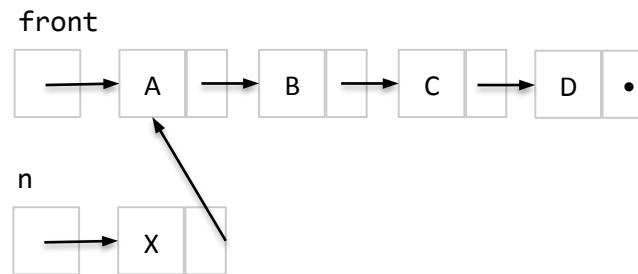
Inserting a new first node



```
Node n = new Node("X");  
  
if (inserting a new first node) {  
  
}  
  
}
```

Inserting nodes

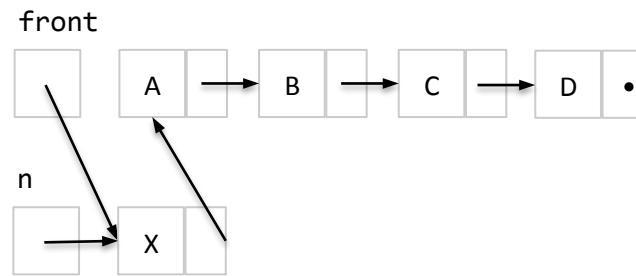
Inserting a new first node



```
Node n = new Node("X");  
  
if (inserting a new first node) {  
    n.next = front;  
}  
}
```

Inserting nodes

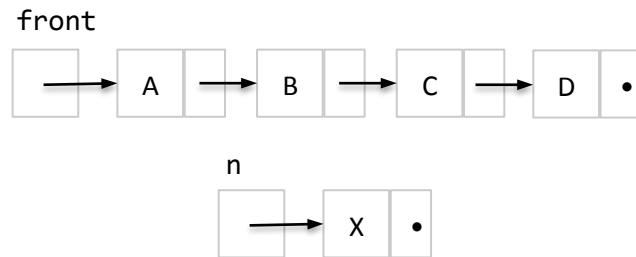
Inserting a new first node



```
Node n = new Node("X");  
  
if (inserting a new first node) {  
    n.next = front;  
    front = n;  
}
```

Inserting nodes

Inserting a new node somewhere else



```
Node n = new Node("X");

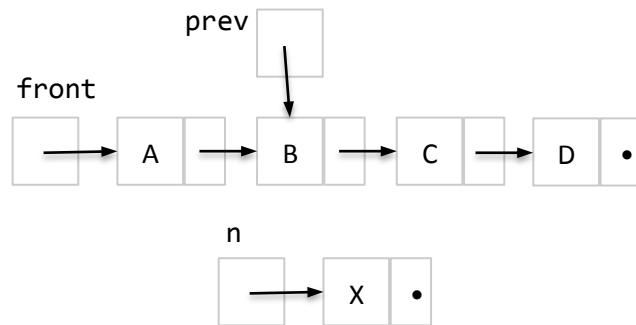
if (inserting a new first node) {
    n.next = front;
    front = n;
}

else {

}
```

Inserting nodes

Inserting a new node somewhere else



```
Node n = new Node("X");

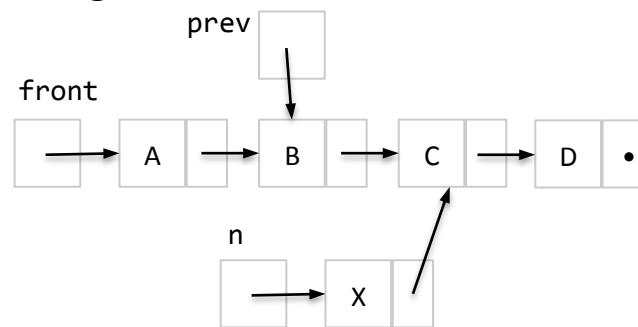
if (inserting a new first node) {
    n.next = front;
    front = n;
}

else {
    Node prev;
    // find the right spot with prev

}
```

Inserting nodes

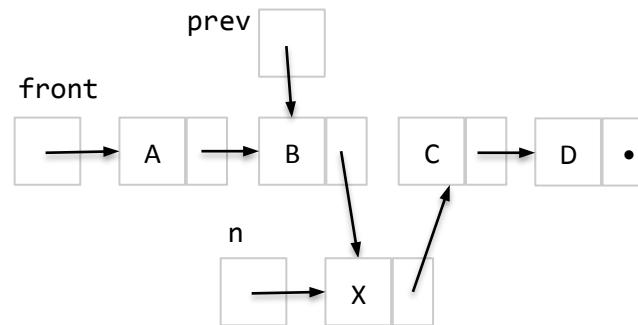
Inserting a new node somewhere else



```
Node n = new Node("X");  
  
if (inserting a new first node) {  
    n.next = front;  
    front = n;  
}  
  
else {  
    Node prev;  
    // find the right spot with prev  
    n.next = prev.next;  
}
```

Inserting nodes

Inserting a new node somewhere else



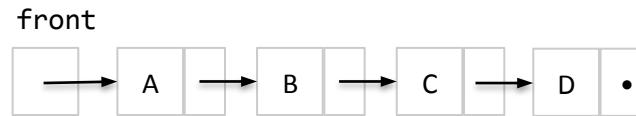
```
Node n = new Node("X");

if (inserting a new first node) {
    n.next = front;
    front = n;
}

else {
    Node prev;
    // find the right spot with prev
    n.next = prev.next;
    prev.next = n;
}
```

Deleting nodes

Deleting the first node



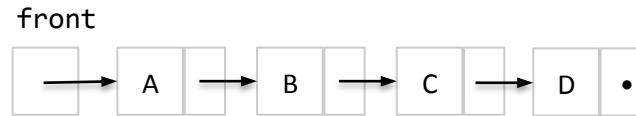
```
if (deleting the first node) {
```

```
}
```

```
else {
```

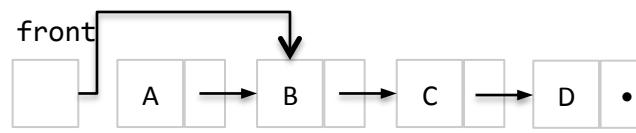
```
}
```

Deleting any other node



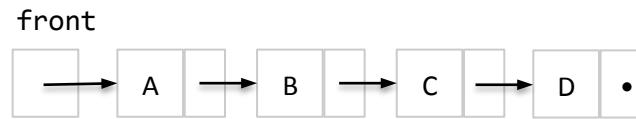
Deleting nodes

Deleting the first node



```
if (deleting the first node) {  
    front = front.next;  
}
```

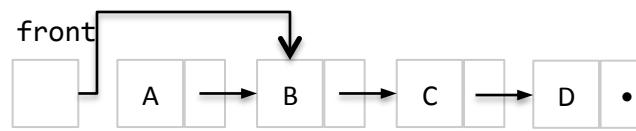
Deleting any other node



```
else {  
}  
}
```

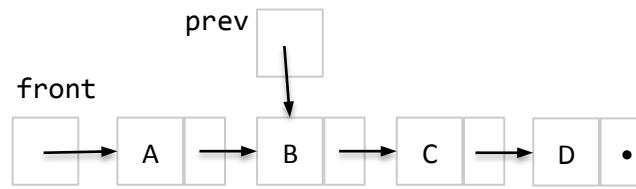
Deleting nodes

Deleting the first node



```
if (deleting the first node) {  
    front = front.next;  
}
```

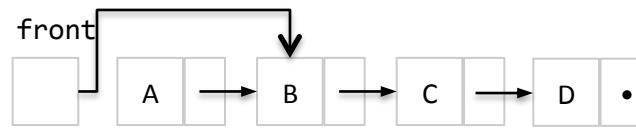
Deleting any other node



```
else {  
    Node prev;  
    // find the right spot with prev  
}  
}
```

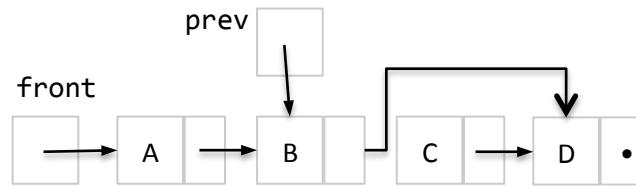
Deleting nodes

Deleting the first node



```
if (deleting the first node) {  
    front = front.next;  
}
```

Deleting any other node



```
else {  
    Node prev;  
    // find the right spot with prev  
    prev.next = prev.next.next;  
}
```

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

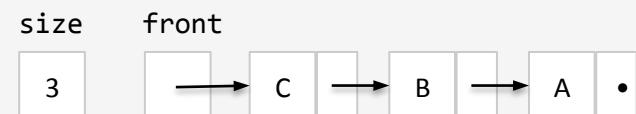
```
public class LinkedBag<T> implements Bag<T> {  
    private Node front;  
    private int size;  
  
    public LinkedBag() {  
        front = null;  
        size = 0;  
    }  
  
    public int size() {  
        return size;  
    }  
  
    public boolean isEmpty() {  
        return size ==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

No memory allocation!

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



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



Linked Bag

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

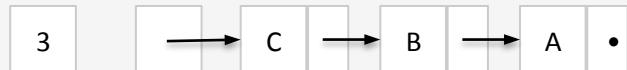
```
Bag bag = new LinkedBag();
```

size front



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

size front



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(target)) {  
                return true;  
            }  
            p = p.next;  
        }  
        return false;  
    }  
}
```

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

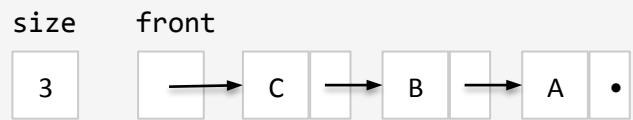
size	front
0	•

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

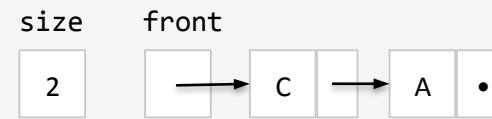
size	front
3	→ C → B → A •

Linked Bag

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

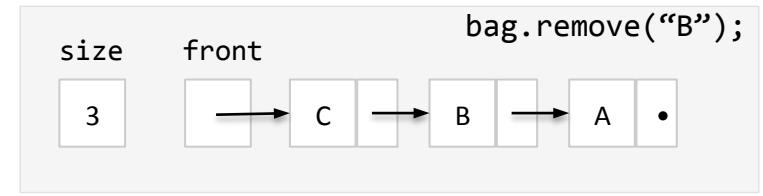


bag.remove("B");



Linked Bag

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



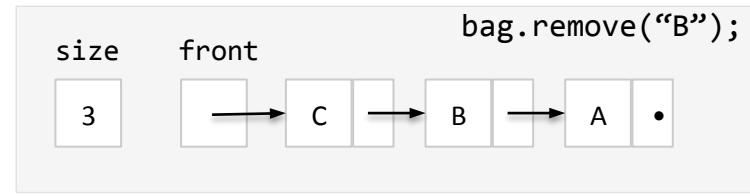
attempt to locate element

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;  
    } }
```



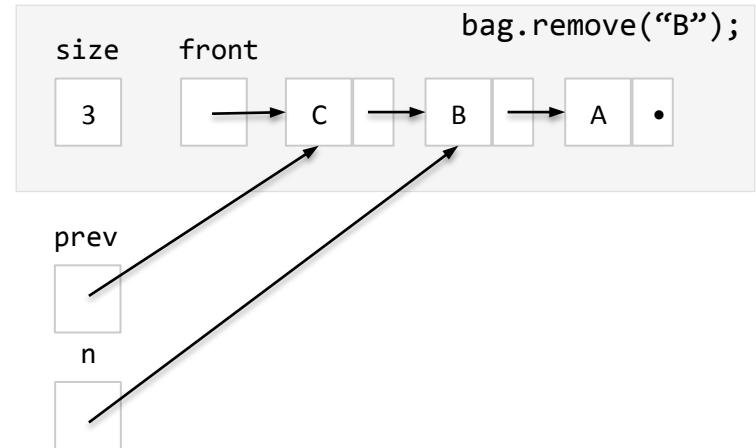
attempt to locate element

unable to locate

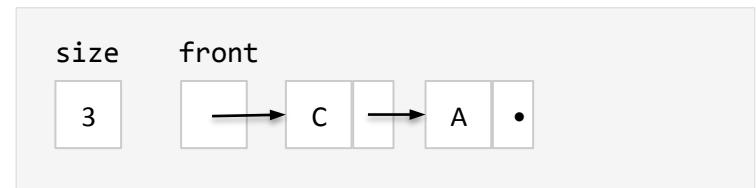
located, so remove it

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;  
    } }
```



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;
    }
}
```

Refactoring

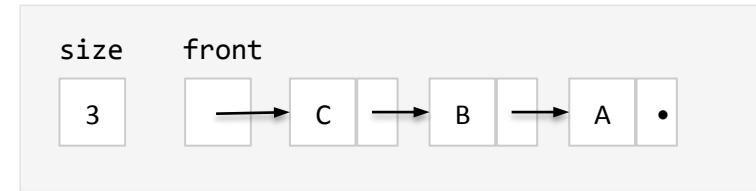
Just as in the array-based implementation, the linear search common to both contains and remove is an obvious candidate for refactoring.

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

```
public boolean remove(T element)
    private Node locate(T element)
        messy to use prev in contains
    public boolean contains(T element)
```

LinkedBag

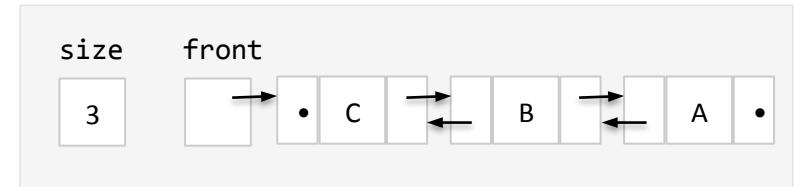
```
public class LinkedBag<T> implements Bag<T> {  
  
    private Node front;  
    private int size;  
    . . .  
  
    private class Node {  
        private T element;  
        private Node next;  
  
    }  
  
}
```



Singly linked

LinkedBag

```
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;  
    }  
  
}
```



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)  
    n  
private Node locate(T element)  
    easy to use n in contains  
public boolean contains(T element)
```

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)  
    easy to use n in contains  
public boolean contains(T element)
```

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)  
    easy to use n in contains  
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 = 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 boolean remove(T element)  
private Node locate(T element)  
easy to use n in contains  
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) {  
        ...  
        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.

```
public boolean remove(T element)  
    n  
private Node locate(T element)  
    easy to use n in contains  
public boolean contains(T element)
```

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() {  
  
    }  
}
```

```
public class LinkedIterator<T>  
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.*

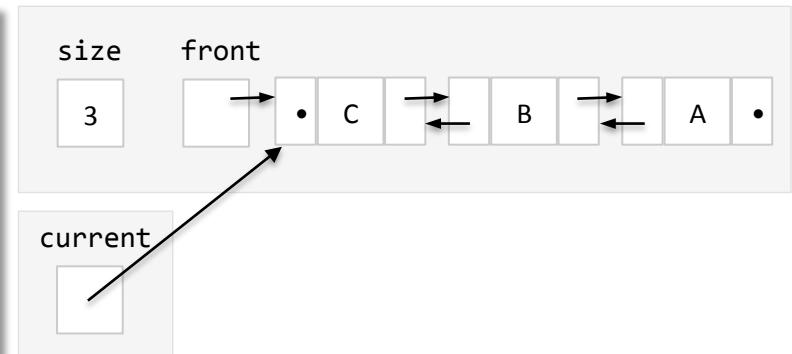
Linked Bag

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



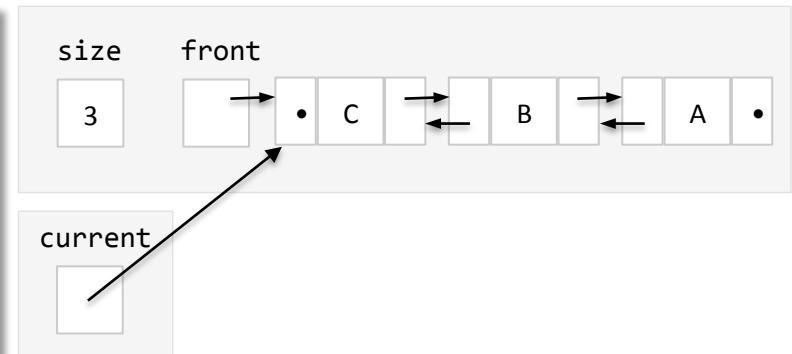
Linked Bag

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

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

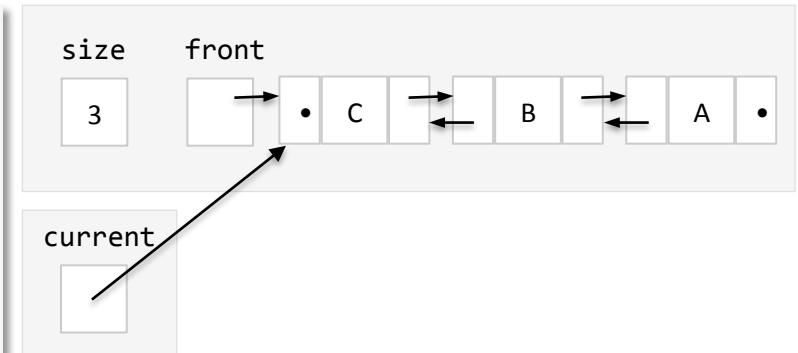
        public boolean hasNext() {
            return current != null;
        }

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



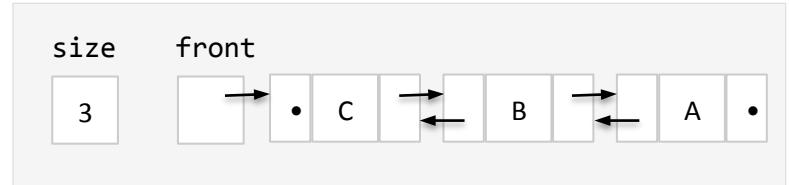
Linked Bag

```
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;  
        }  
    }  
}
```



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:

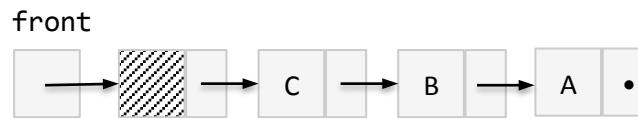
- no random access;
- less efficient use of memory
- not built in; nodes are user-created

Common variations

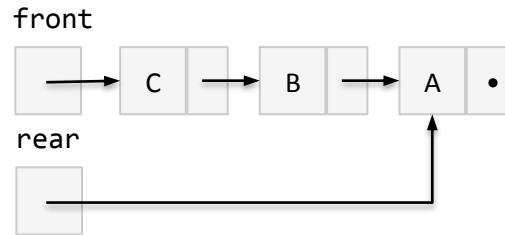
Singly linked v. Doubly linked



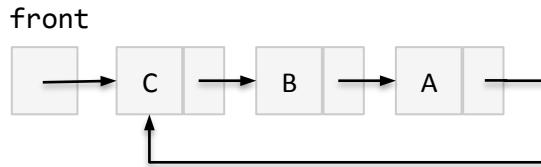
Dummy/Header nodes



Front and rear pointers



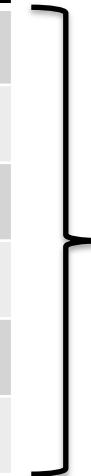
Circular



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 you 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)	$\Theta(N)$
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...]