Lab 03: Array Bag vs Linked Bag

CS 0445: Data Structures

TAs: Jon Rutkauskas
Brian Nixon

http://db.cs.pitt.edu/courses/cs0445/current.term/

Sep 22, 2019
University of Pittsburgh, Pittsburgh, PA

Intro

- Differences between array and linked implementations of Bag
- Differences in implementation
- How .equals() works

ArrayBag

- Stores data in a dynamically-resizing array
  - Create array with initial length
  - When full, create a new array with twice the size and copy the data over
  - Allow the old array to be garbage collected
  - This can be complicated and causes some `add()` calls to take additional time, as well as reserving additional memory/capacity that might not be needed
- All data stored in contiguous memory
- Can easily index to any element in the array

LinkedBag

- Store data in a chain of `Node` objects
- Only create new nodes when we need them
- No need to resize.
- Does not allow indexing, to access any node we need to **traverse** to it
- Uses extra memory to keep track of `next` reference
- Easy to remove or add nodes to the chain if needed, no shifting required

```
private class Node {
   private E data; // Entry in bag
   private Node next; // link to next node

   private Node(E dataPortion, Node nextNode) {
      data = dataPortion;
      next = nextNode;
   }
}
```

Differences In Implementation – `contains(E anEntry)`

- Both implementations require examining each entry in the data structure
  - ArrayBag requires indexing to each position
  - LinkedBag requires a traversal through the chain

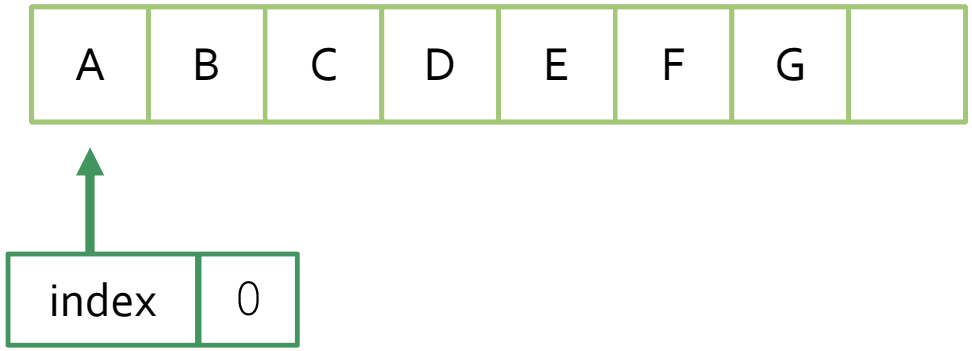
```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
  }
```

```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

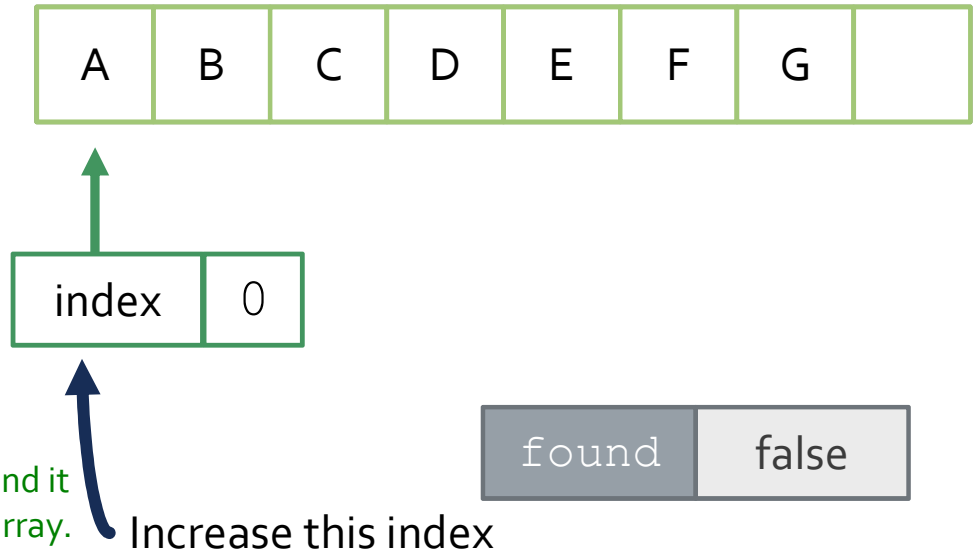
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

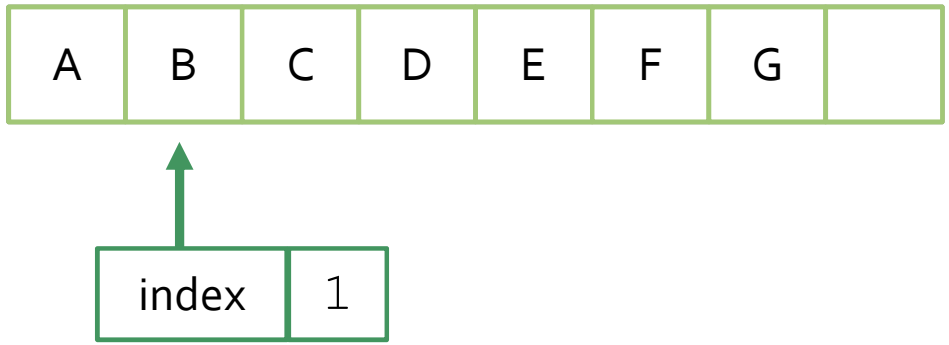
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

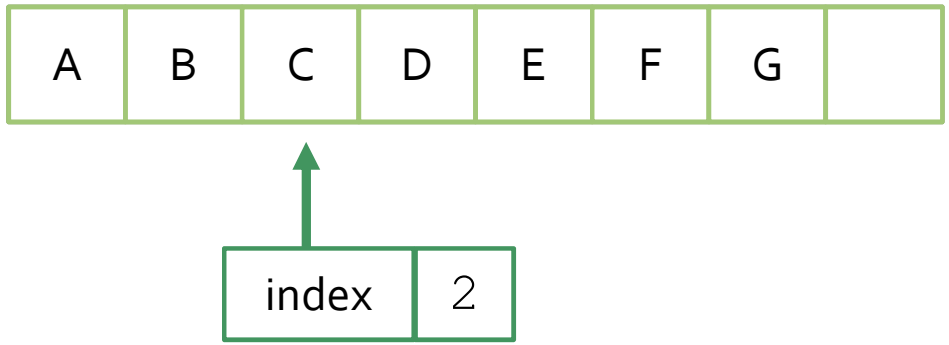
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

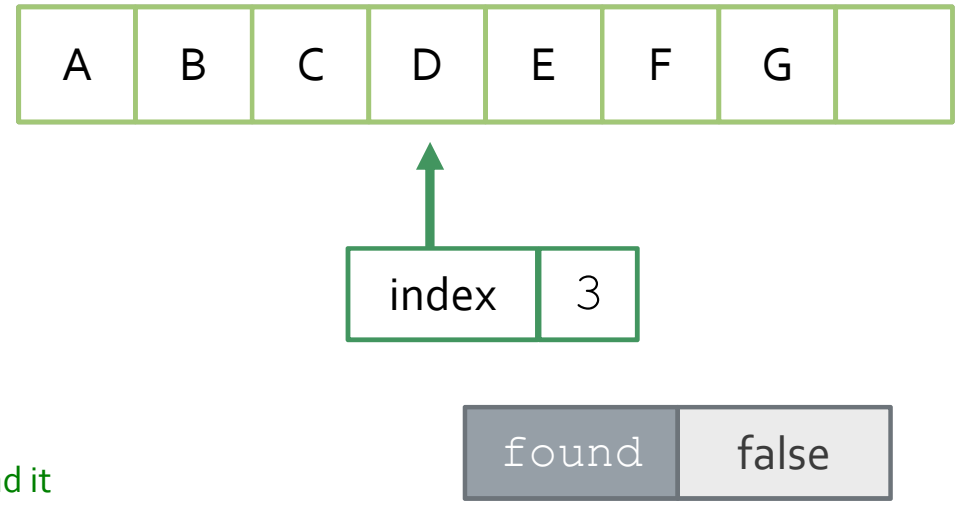
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

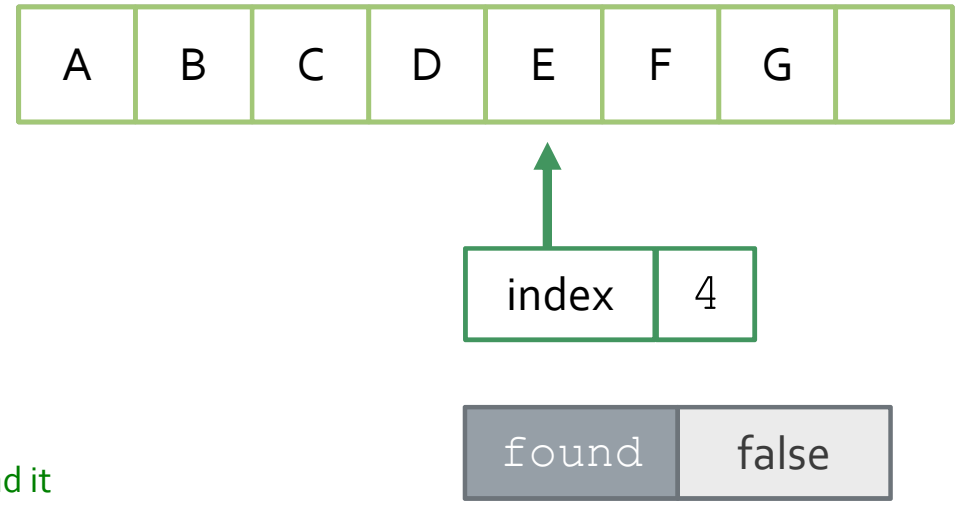
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

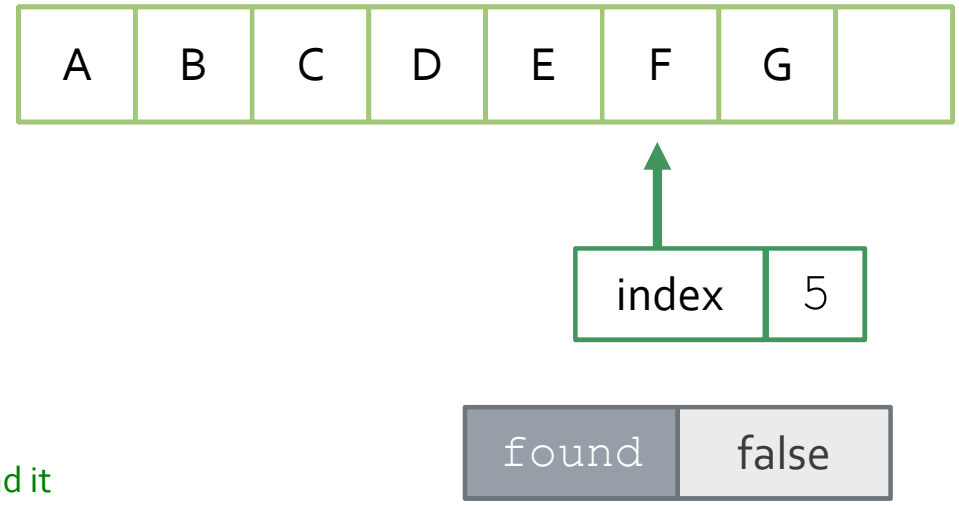
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

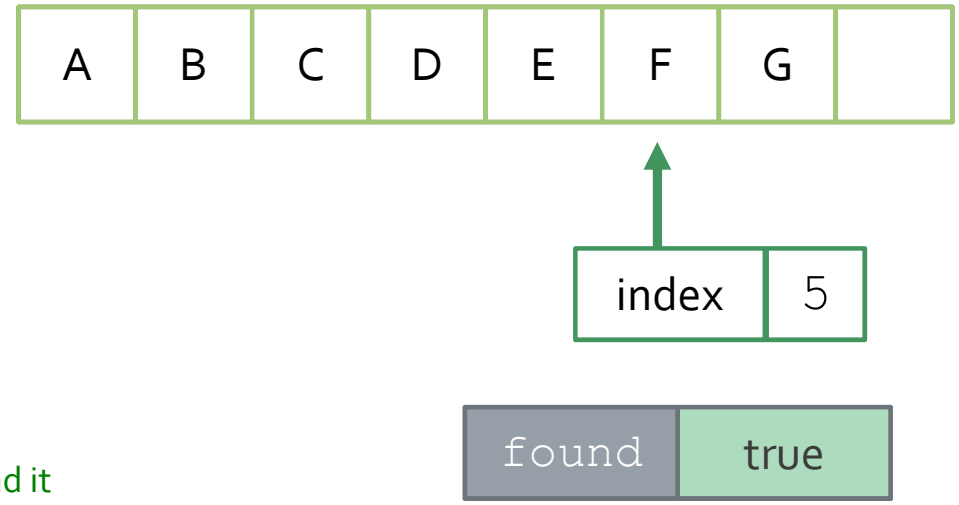
contains(F)?



```
public boolean contains(E anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
  }

// Locates a given entry within the array bag.
  // Returns the index of the entry, if located,
  // or -1 otherwise.
  private int getIndexOf(E anEntry) {
    int where = -1;
    boolean found = false;
    int index = 0;

    while (!found && (index < size)) {
      if (anEntry != null && anEntry.equals(bag[index])) {
        found = true;
        where = index;
      }
      index++;
    }

    // Assertion: If where > -1, anEntry is in the array bag, and it
    // equals bag[where]; otherwise, anEntry is not in the array.

    return where;
  }
```

ArrayBag

contains(F)?

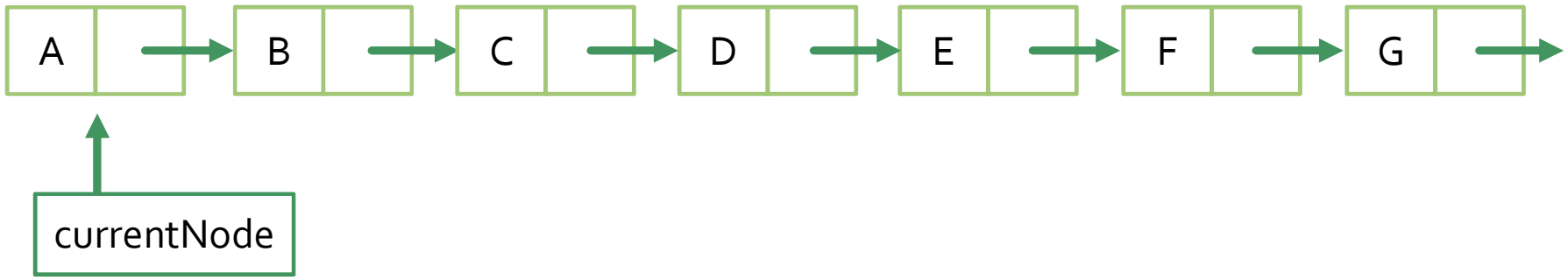


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
}
```

LinkedBag

`contains(F)?`

found	false

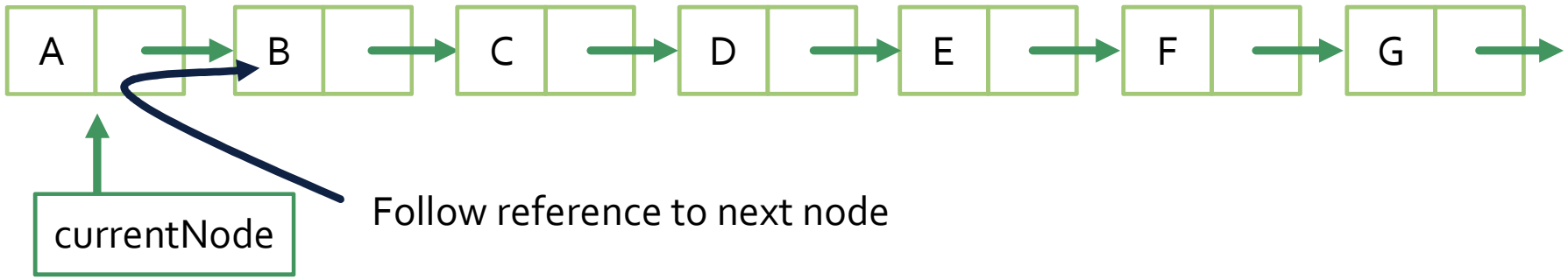


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
}
```

LinkedBag

`contains(F)?`

found	false

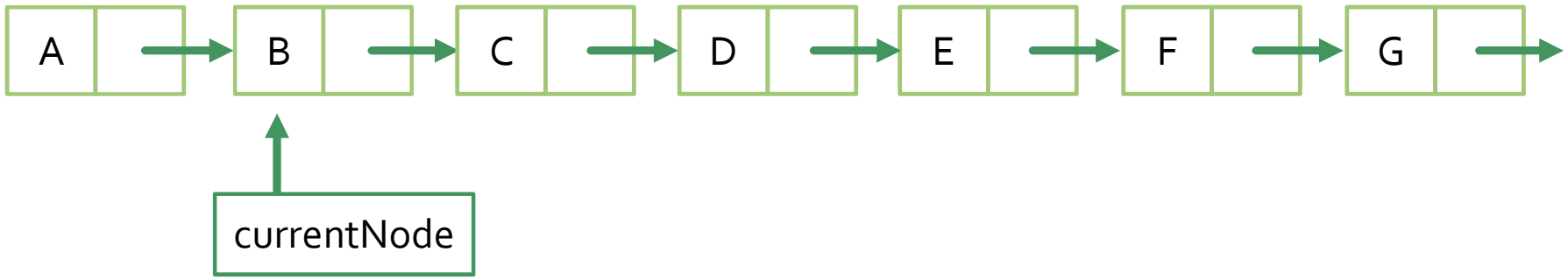


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
  }
```

LinkedBag

`contains(F)?`

found	false

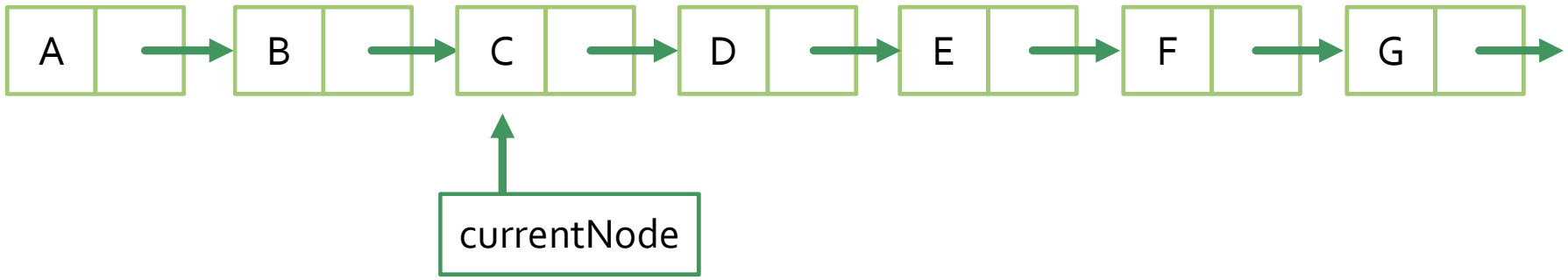


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
}
```

LinkedBag

`contains(F)?`

found	false

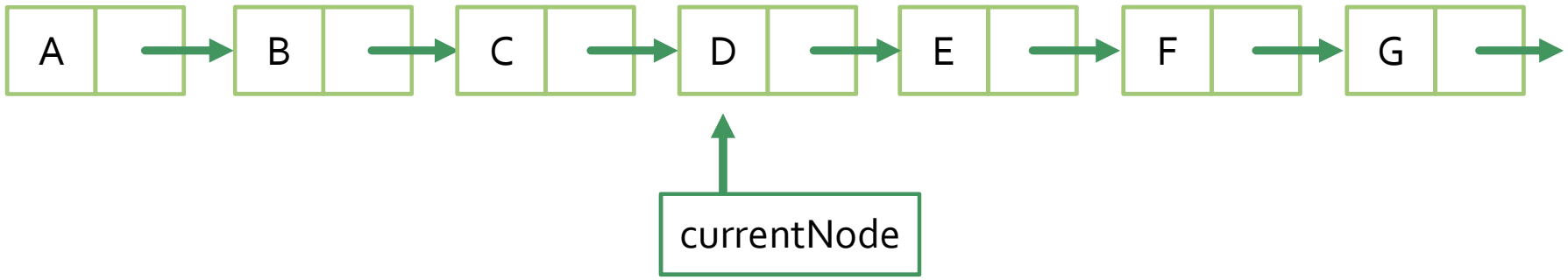


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
  }
```

LinkedBag

`contains(F)?`

found	false

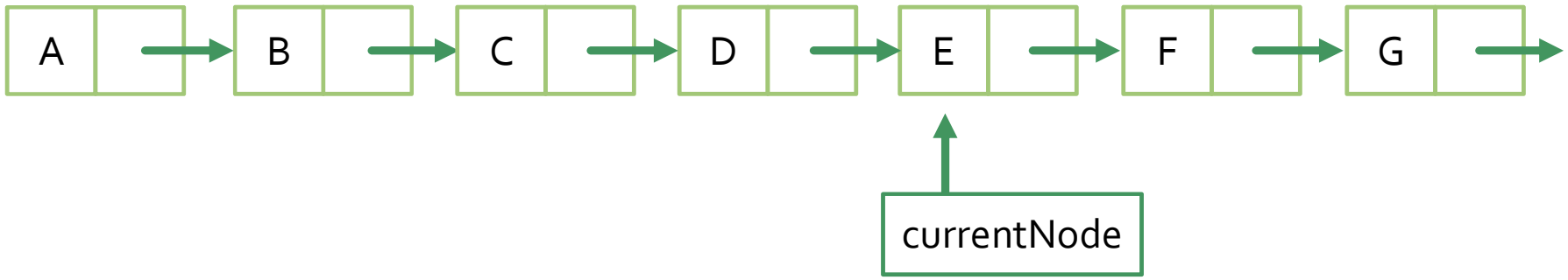


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
  }
```

LinkedBag

`contains(F)?`

found	false

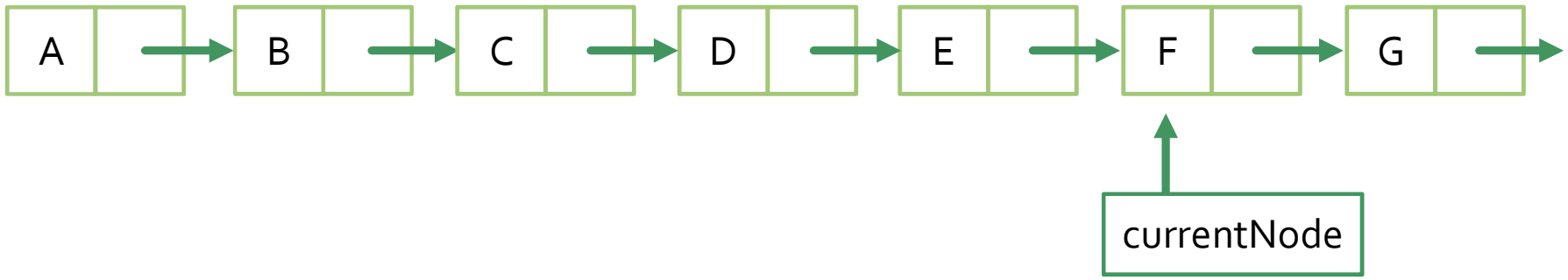


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
  }
```

LinkedBag

`contains(F)?`

found	false

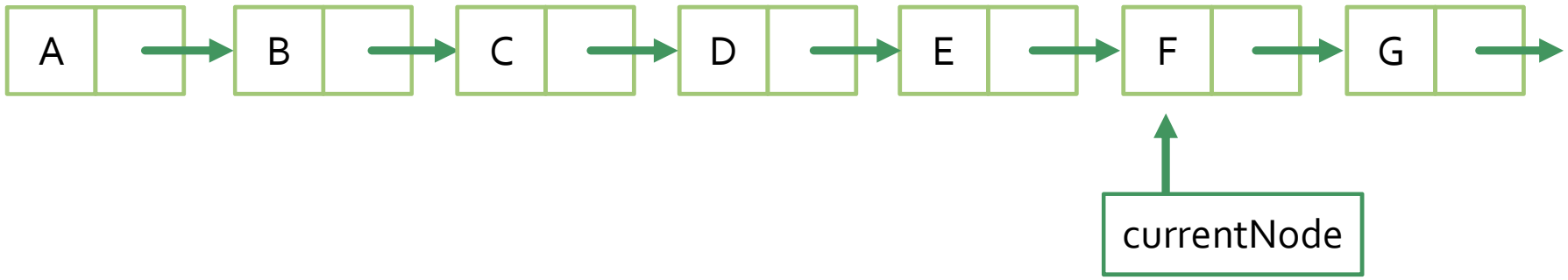


```
public boolean contains(E anEntry) {
    boolean found = false;
    Node currentNode = head;
    while (!found && (currentNode != null)) {
      if (anEntry != null && anEntry.equals(currentNode.data)) {
        found = true;
      } else {
        currentNode = currentNode.next;
      }
    }
    return found;
}
```

LinkedBag

`contains(F)?`

found	true



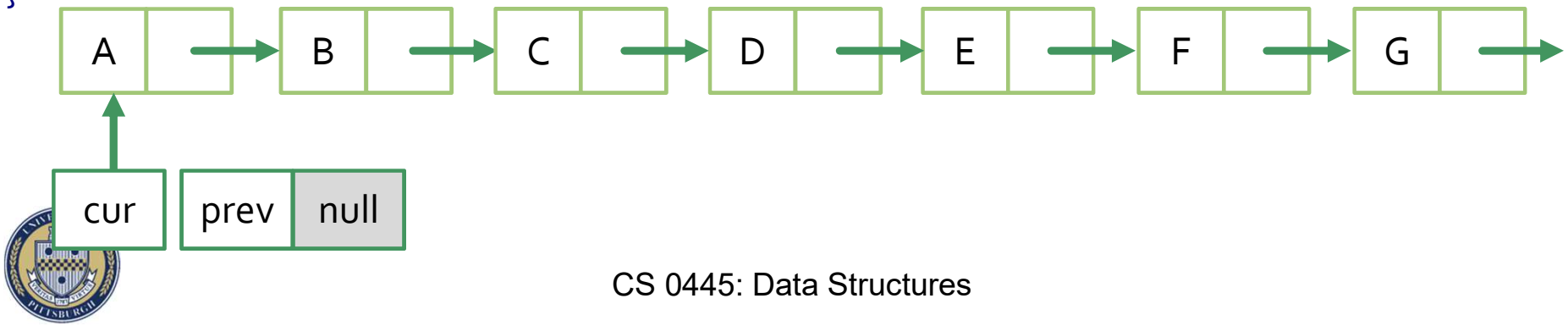
Differences In Implementation – `remove(E anEntry)`

- Both implementations require examining each entry in the data structure
  - ArrayBag requires indexing to each position
  - LinkedBag requires a traversal through the chain
- Removing
  - ArrayBag swaps removed element with its last element
  - LinkedBag: Two different approaches
    - Swap data with head; remove head
    - **Remove entry from chain**

```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

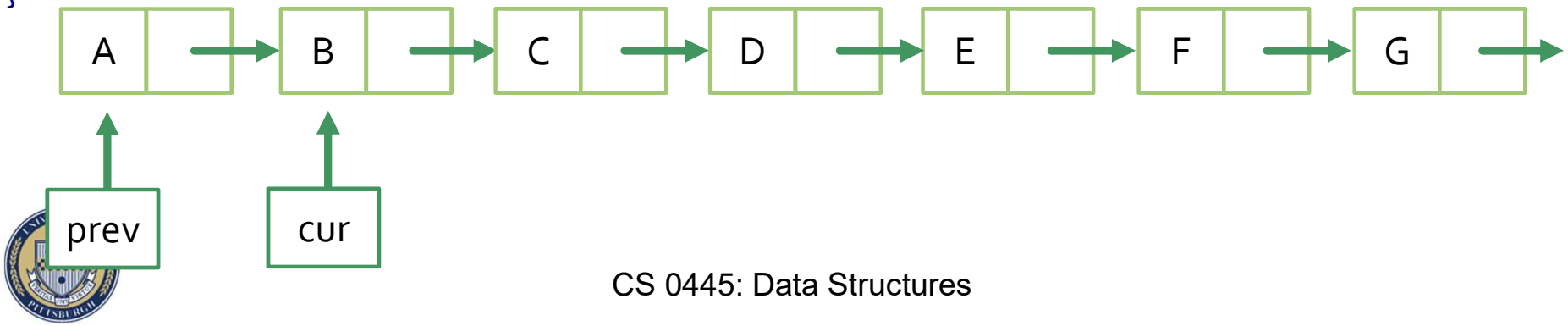
`remove(E)`



```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

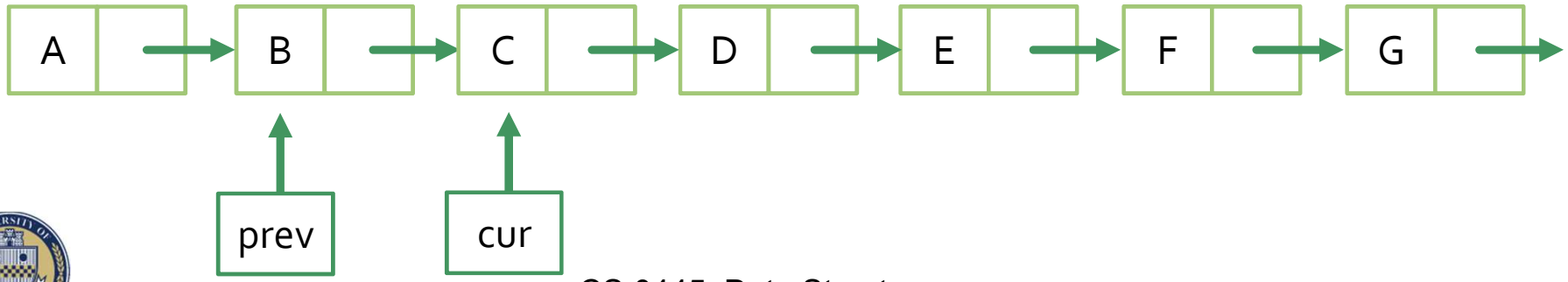
`remove(E)`



```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

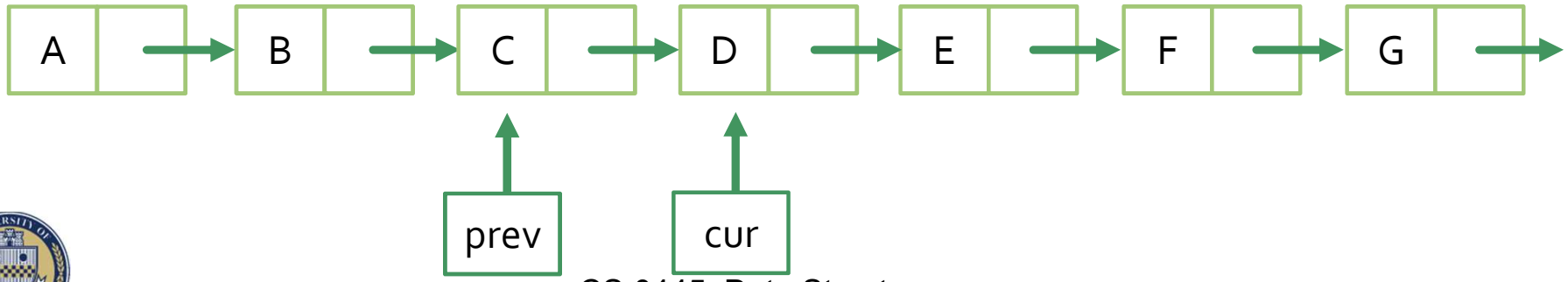
`remove(E)`



```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

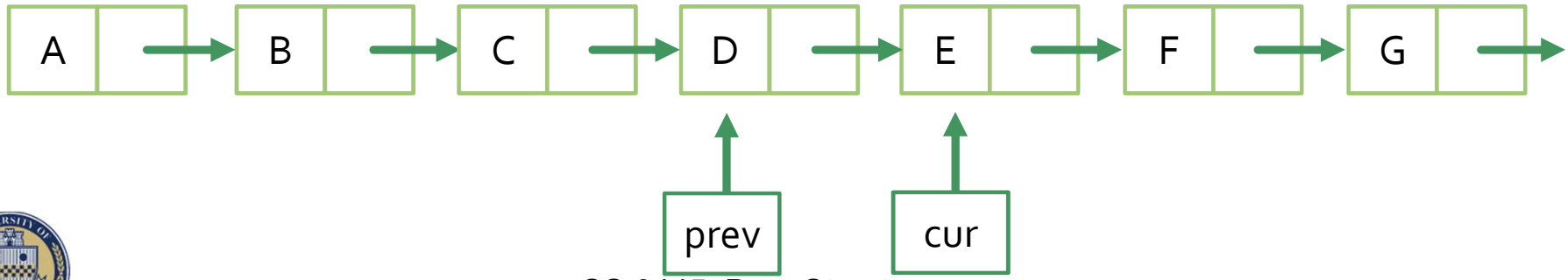
`remove(E)`



```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

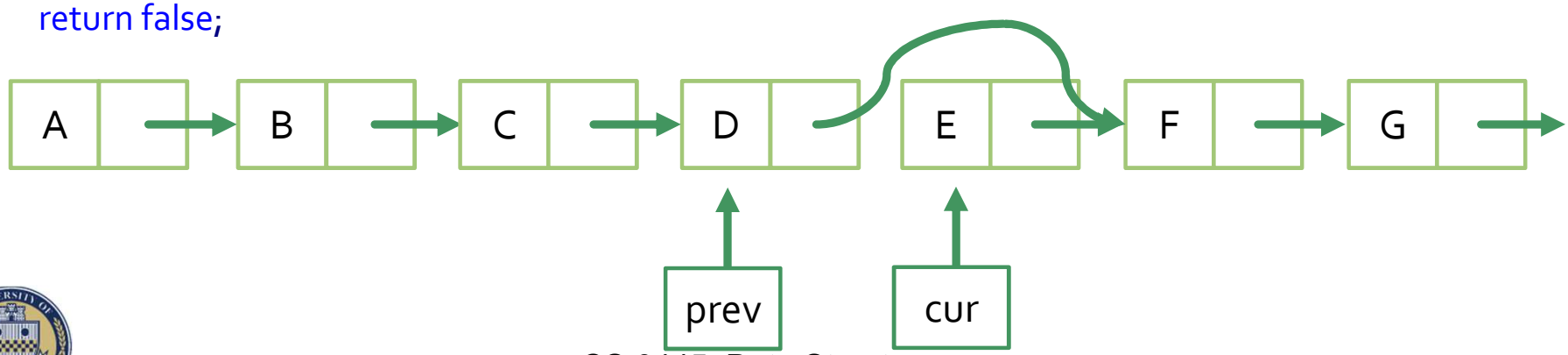
`remove(E)`



```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

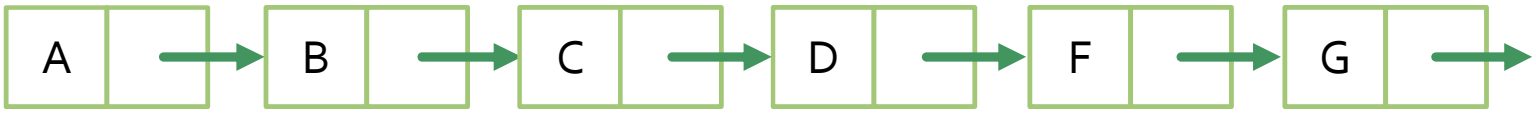
`remove(E)`



```
public boolean remove(E item) {
        Node cur = head;
        Node prev = null;
        boolean found = false;
        while(cur != null) {
                if(item != null && item.equals(cur.data)) {
                        if(prev == null) { //Special case, first item
                                head = head.next;
                        }
                        else {
                                prev.next = cur.next;
                        }
                        size--;
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }
        return false;
}
```

LinkedBag

`remove(E)`



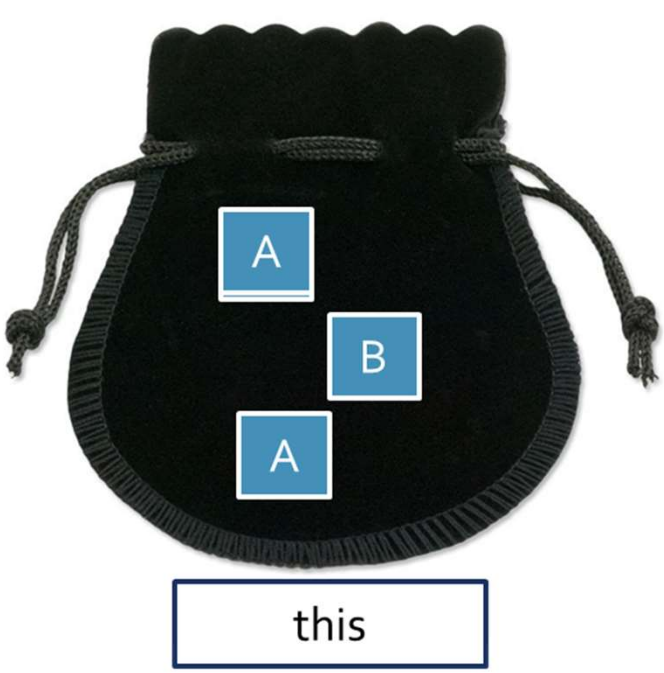
.equals(Object other)

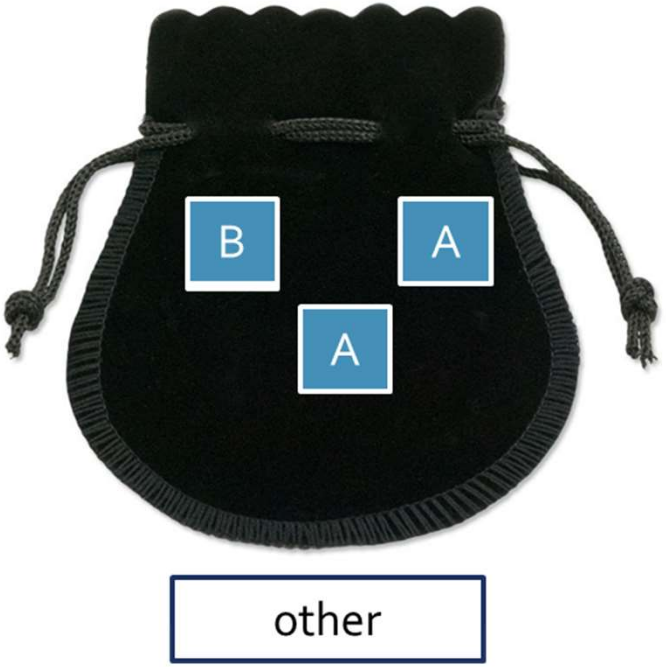
- This method compares two bags of the same type to check if they are "equal"
- Equal means they contain the same amount of the same objects
- E.g., if bagA contains {A, A, B, D} and bagB contains {B, A, D, A}, then bagA.equals(bagB) is true (order doesn't matter)
- If bagC contains {A, A, A, B, D}, bagA.equals(bagC) would be false
- Analogy: If I have a bag with 3 chocolate bars and a cookie, and you have a bag with a cookie and 3 chocolate bars, order doesn't matter, and we can say the bags are equal
  - If Alice comes along with a bag containing 2 chocolate bars, a cookie, and a can of soda, her bag differs in the number of chocolate bars and the fact she has a can of soda, so her bag does not equal ours

.equals(Object other) - Algorithm

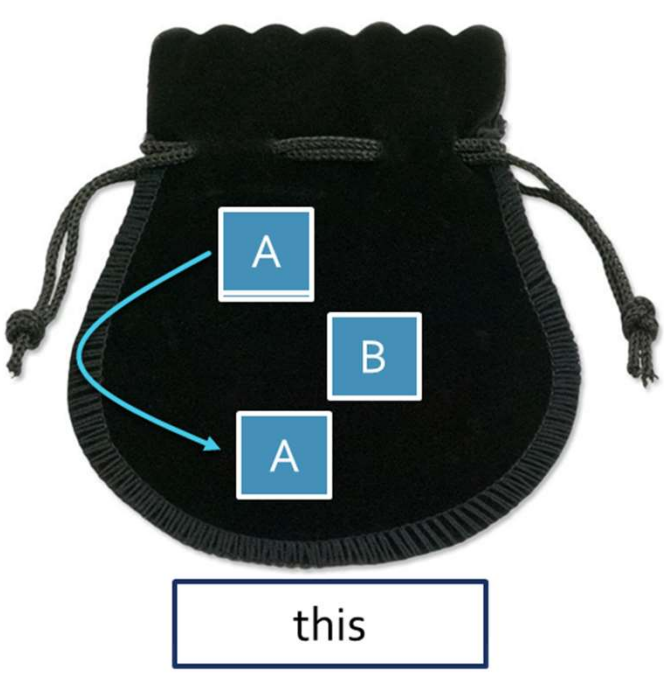
- To check if two bags are equal:
  - Ensure the other object is the same class (E.g., ArrayBag should check that other is also an ArrayBag)
  - Check that the bags are of the same size; if not, they cannot be equal
  - For each item in the bag
    - Check if the number of that item in this bag is the same as the number of that item in the other bag
    - If not, end, the bags are not equal
    - Otherwise, continue with the next item
  - Once all items have been checked, return true

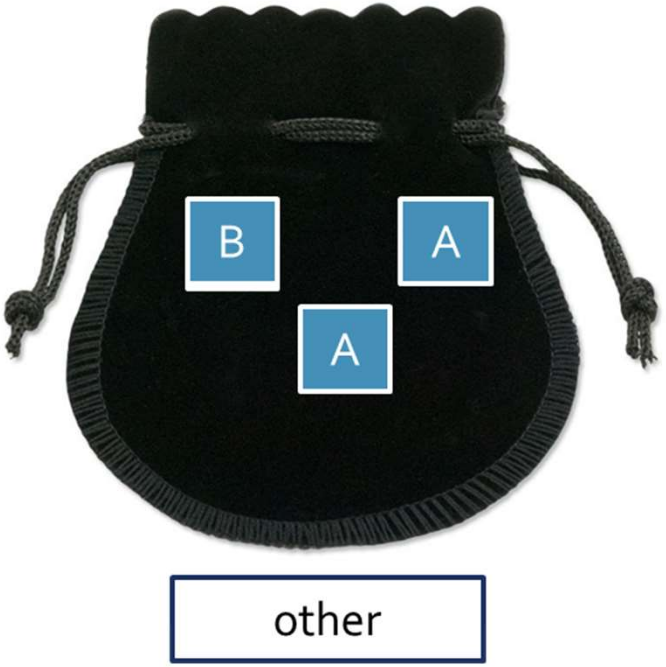
.equals(Object other) – Algorithm Example





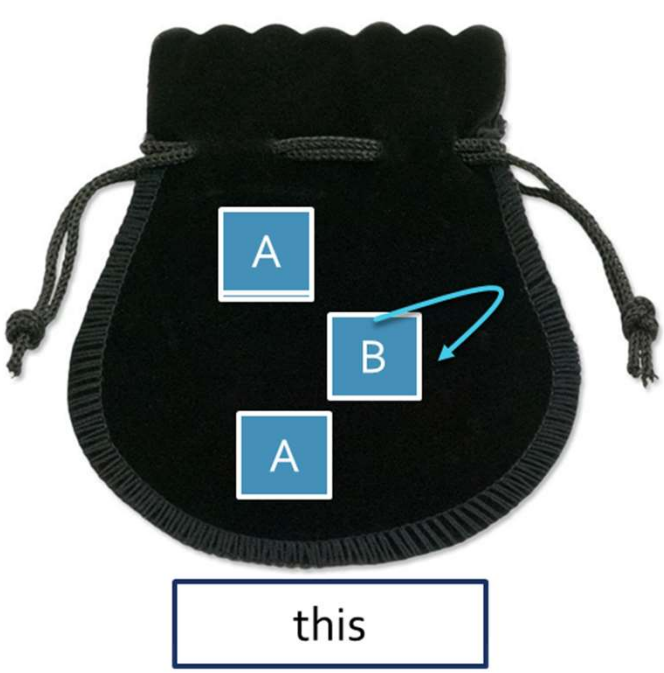
.equals(Object other) – Algorithm Example

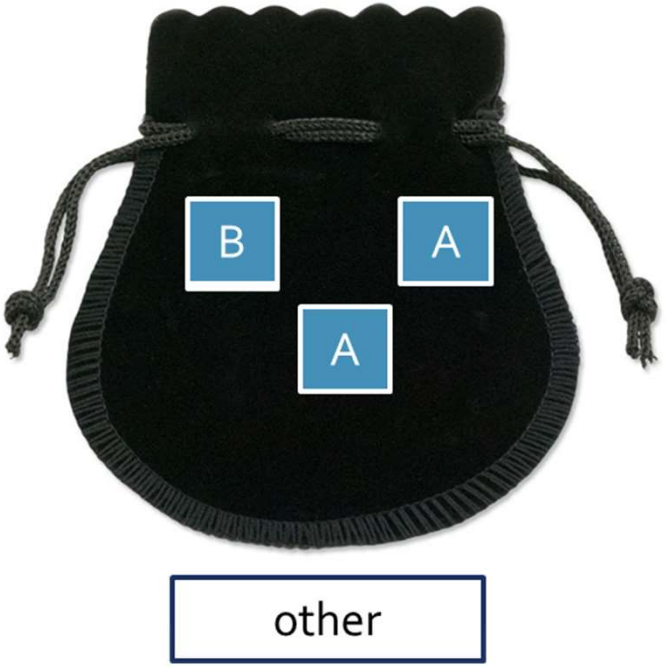




.equals(Object other) – Algorithm Example

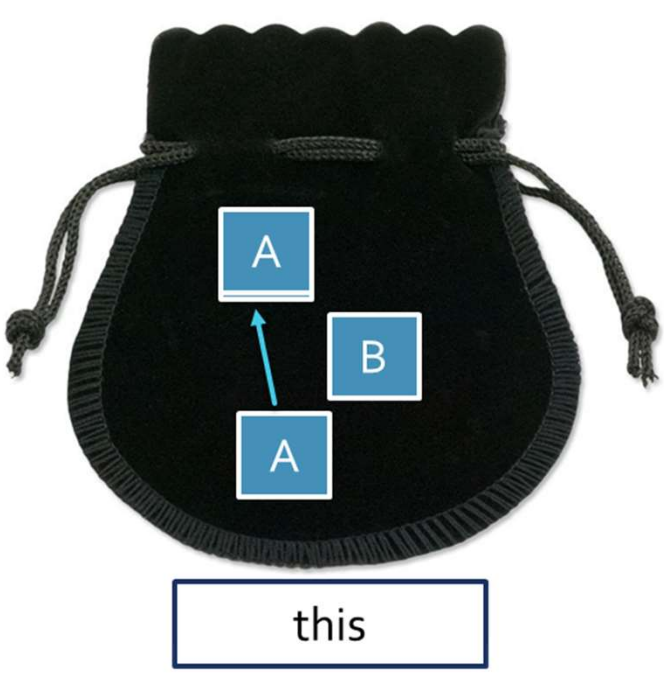
.equals(Object other) – Algorithm Example

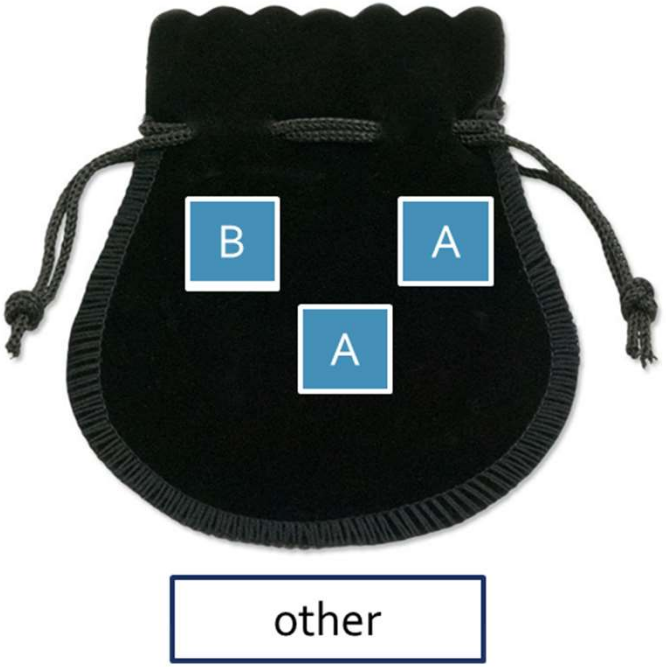




.equals(Object other) – Algorithm Example

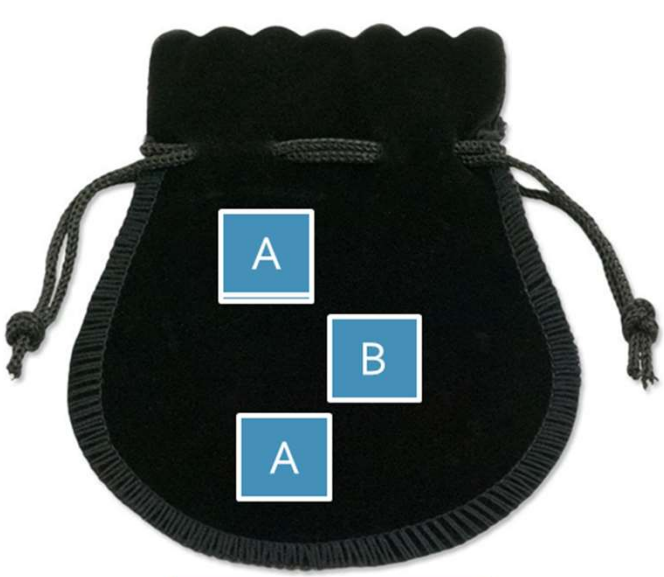
.equals(Object other) – Algorithm Example





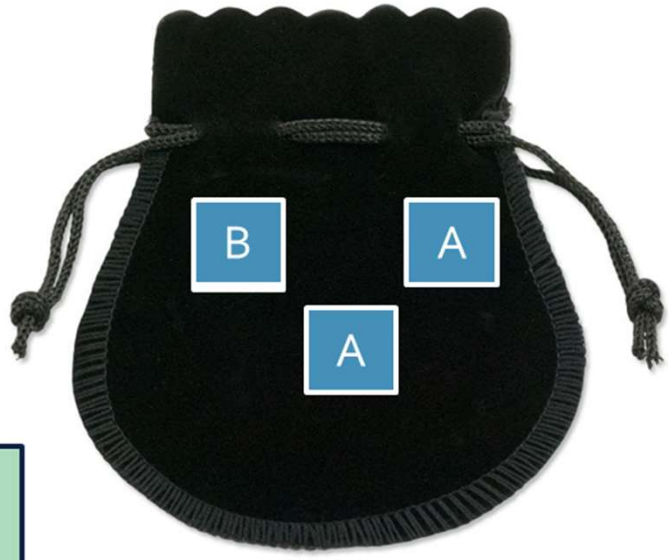
.equals(Object other) – Algorithm Example

.equals(Object other) – Algorithm Example



this

Equal!



other

Your Tasks

- Download the code from the course website
  - http://db.cs.pitt.edu/courses/cs0445/current.term/
- Implement the .equals method for both ArrayBag and LinkedBag
- Test your method with EqualsTest
  - It's also a good idea to add some additional test cases