Lists

COMP 2210 – Dr. Hendrix

AUBURN
UNIVERSITY

Samuel Ginn
College of Engineering

Google define:list

http://en.wikipedia.org/wiki/List_(computing))

“In computer science, a **list** or *sequence* is an abstract data structure that implements an **ordered collection of values**”

Ordered …

By element value	Self-ordered lists (sorted)
By absolute position (index number)	Indexed lists (sequence)
By relative position (front, rear, after)	Non-indexed lists (“bullet” list)
By time of insertion	Temporal lists (stacks, queues)
By priority	Priority queues

Define:List according to the text and me

Chapter 4 – Unordered List

Aren't all lists ordered? Isn't that the point?

"An unordered list is a linear collection of entries whose relative positions with respect to one another is irrelevant."

Order is independent of element value and is decided by the client.

```
1. Auburn
2. TCU
3. Oregon
4. Stanford
5. Ohio State
```

```
•   Milk
•   Eggs
•   Bread
•   Cheetos
•   Pizza
```

Chapter 5 – Ordered List

What's a key?

"An ordered list is a linear collection of entries in which the entries are arranged in either ascending or descending order of keys."

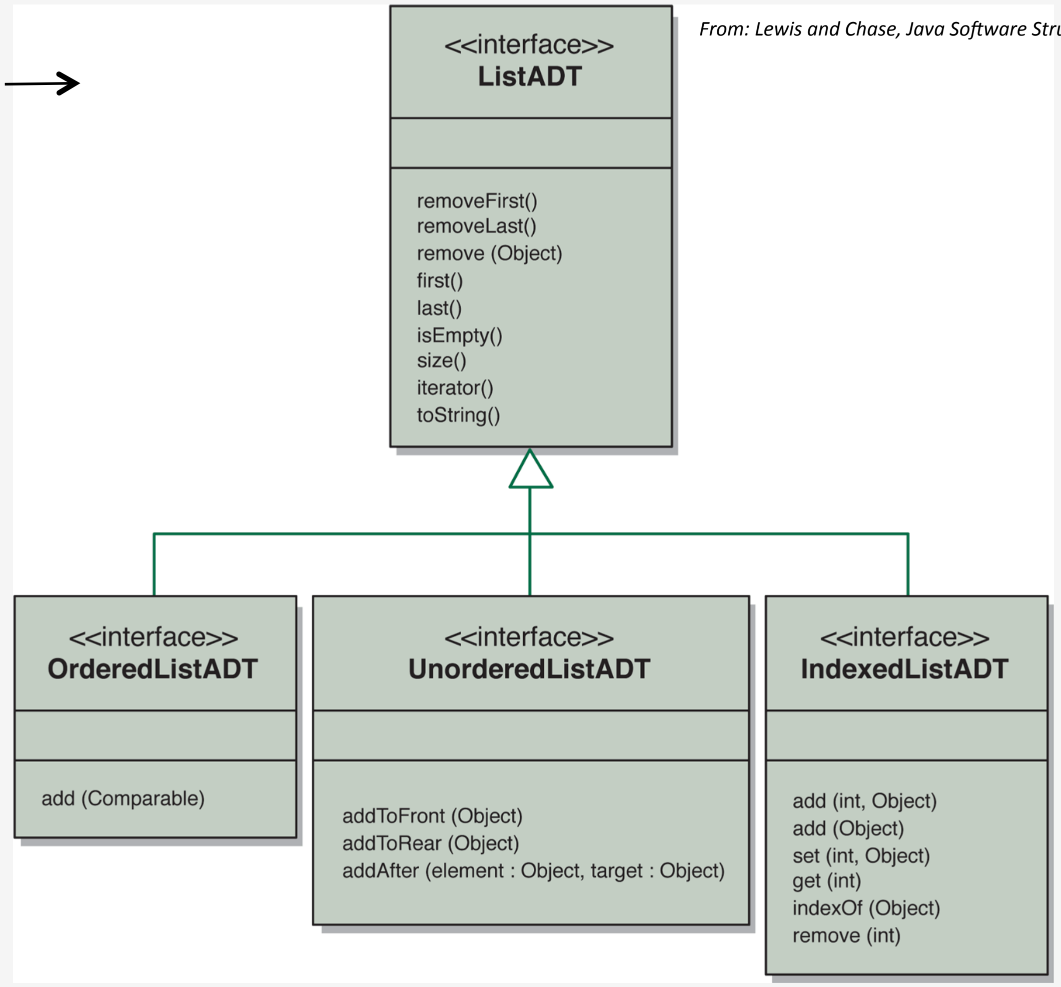
Order is completely determined by element value and is not (arbitrarily) decided by the client.

```
Adams, John, 2, 1797-1801
Adams, John Q., 6, 1825-1829
Arthur, Chester, 21, 1881-1885
Buchanan, James, 15, 1857-1861
Bush, George HW, 41, 1989-1993
```

Designing a list collection

No add method at this level →

From: Lewis and Chase, Java Software Structures



Different kinds of lists would have to support different kinds of add methods.

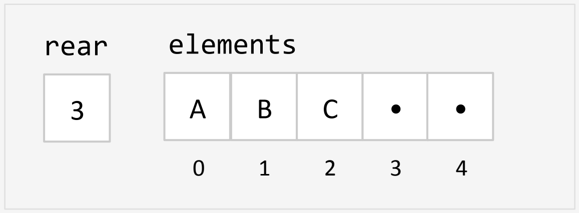
```
Adams, John, 2, 1797-1801
Adams, John Q., 6, 1825-1829
Arthur, Chester, 21, 1881-1885
Buchanan, James, 15, 1857-1861
Bush, George HW, 41, 1989-1993
```

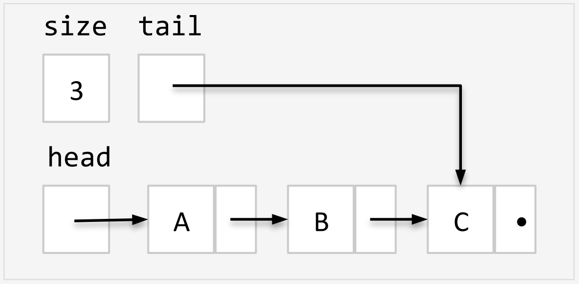
- Milk
- Eggs
- Bread
- Cheetos
- Pizza

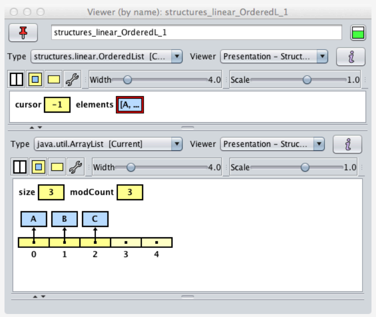
1. Auburn
2. TCU
3. Oregon
4. Stanford
5. Ohio State

Implementation choices

Array-based	Node-based	
• Keep elements left-justified (anchored at 0, no gaps) • Keep a size counter (can serve as a rear marker)	• Singly-linked • Not circular, no dummy • Keep both a front and rear pointer • Keep a size counter	**2210**
• Use an internal java.util.ArrayList • Keep a "cursor" field to implement iteration.	• Singly-linked • Circular, no dummy • Keep only a rear pointer • Keep a size counter	**VText**






Viewer (by name): structures_linear_OrderedL_1

structures_linear_OrderedL_1

Type structures.linear.OrderedList [C... Viewer Presentation – Struct...

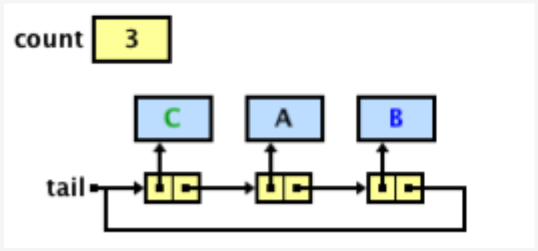
Width 4.0 Scale 1.0

cursor -1 elements [A, ...

Type java.util.ArrayList [Current] Viewer Presentation – Struc...

Width 4.0 Scale 1.0

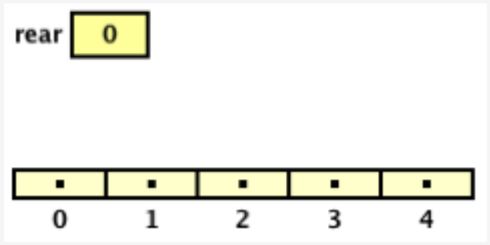
size 3 modCount 3



array-based implementation

Implementing a list collection: Arrays, add method

```
public class ArrayIndexedList<T> implements IndexedList<T>
{
   private T[] elements;
   private int rear;
   ...
```

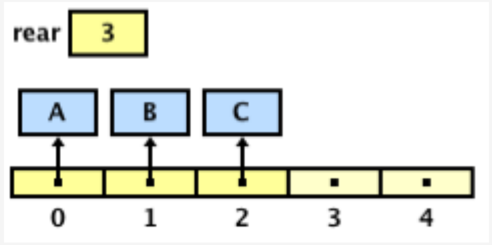


```
ArrayIndexedList<String> = new ArrayIndexedList<String>(5);
```

```
alist.add(“A”);
alist.add(“B”);
alist.add(“C”);
```

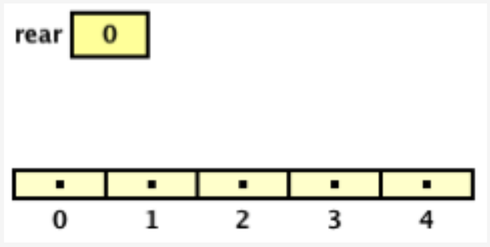
```
alist.add(1, “D”);
```

```
public void add (int index, T element)
{





}
```



Implementing a list collection: Arrays, add method

```
public class ArrayIndexedList<T> implements IndexedList<T>
{
   private T[] elements;
   private int rear;
   ...
```



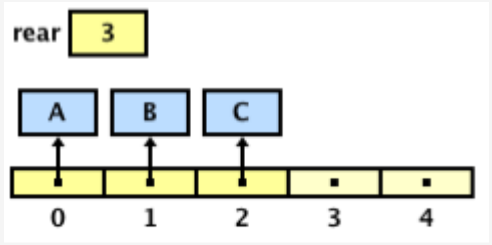
```
ArrayIndexedList<String> = new ArrayIndexedList<String>(5);
```

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

```
alist.add(1, "D");
```

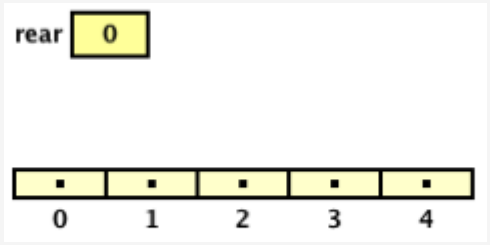
```
public void add (int index, T element)
{
  if ((index < 0) || (index > size()))
  {
     throw new IndexOutOfBoundsException();
  }







}
```

Validate index



Implementing a list collection: Arrays, add method

```
public class ArrayIndexedList<T> implements IndexedList<T>
{
   private T[] elements;
   private int rear;
   ...
```



```
ArrayIndexedList<String> = new ArrayIndexedList<String>(5);
```

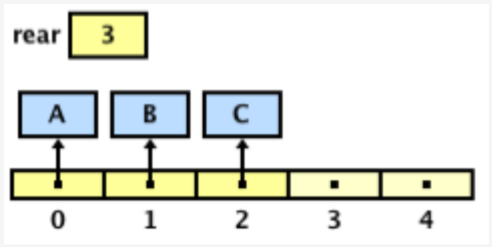
```
alist.add(“A”);
alist.add(“B”);
alist.add(“C”);
```

```
alist.add(1, “D”);
```

```
public void add (int index, T element)
{
  if ((index < 0) || (index > size()))
  {
     throw new IndexOutOfBoundsException();
  }
  if (isFull())
  {
     expandCapacity();
  }



}
```

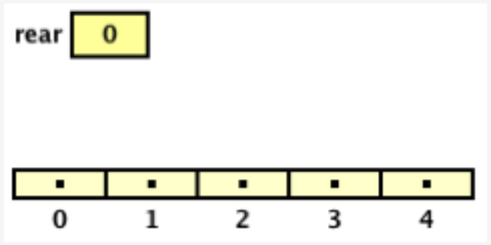
Validate index

Check if full



Implementing a list collection: Arrays, add method

```
public class ArrayIndexedList<T> implements IndexedList<T>
{
   private T[] elements;
   private int rear;
   ...
```



```
ArrayIndexedList<String> = new ArrayIndexedList<String>(5);
```

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

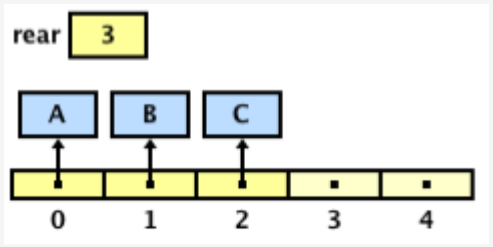
```
alist.add(1, "D");
```

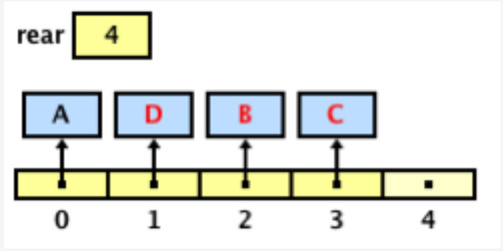
```
public void add (int index, T element)
{
   if ((index < 0) || (index > size()))
   {
      throw new IndexOutOfBoundsException();
   }
   if (isFull())
   {
      expandCapacity();
   }
   shiftRight(index);
   elements[index] = element;
   rear++;
}
```

Validate index

Check if full

Insert new element





Arrays, add method – time complexity

```
public class ArrayIndexedList<T> implements IndexedList<T>
{
   private T[] elements;
   private int rear;
   ...
```

The add method is O(N).

```
public void add (int index, T element)
{
   if ((index < 0) || (index > size()))
   {
      throw new IndexOutOfBoundsException();
   }
   if (isFull())
   {
      expandCapacity();
   }
   shiftRight(index);
   elements[index] = element;
   rear++;
}
```

Validate index O(1)

Check if full O(N) *We can amortize this*

Insert new element O(N)

Two important points:

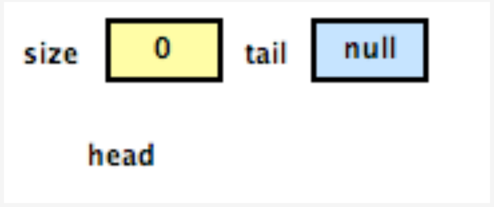
(1) expandCapacity() should not be called often.
Use "repeated doubling."
Consider a reduceCapacity() for remove.

(2) The physical insertion is O(1).
Having to shift elements is O(N).
This is unavoidable with order.

node-based implementation

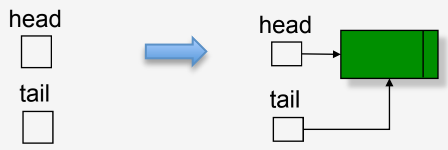
Implementing a list collection: Nodes, add method

```
public class LinkedIndexedList<T> implements IndexedList<T>
{
   private Node<T> head;
   private Node<T> tail;
   private int size;
   ...
```

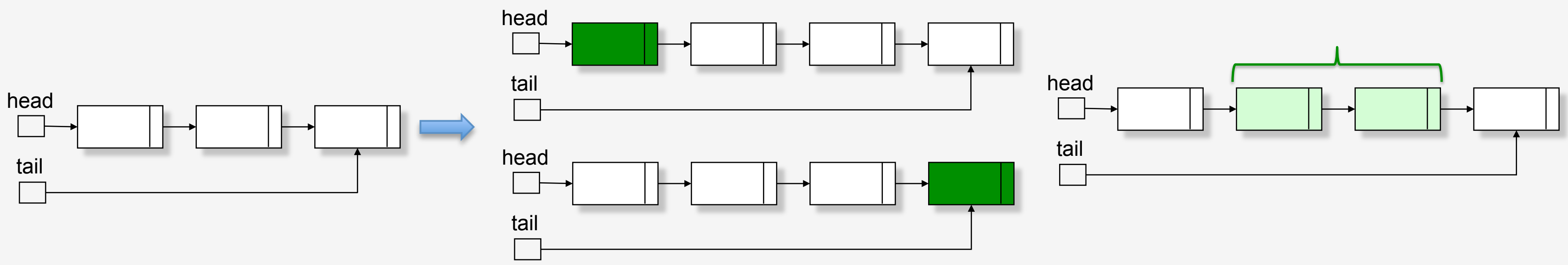


```
 public void add (int index, T element)
```

Empty



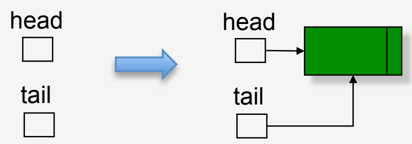
Not empty



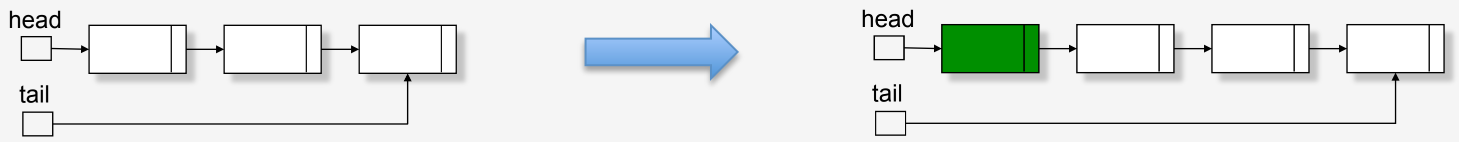
Implementing a list collection: Nodes, add method

Four cases to consider for add

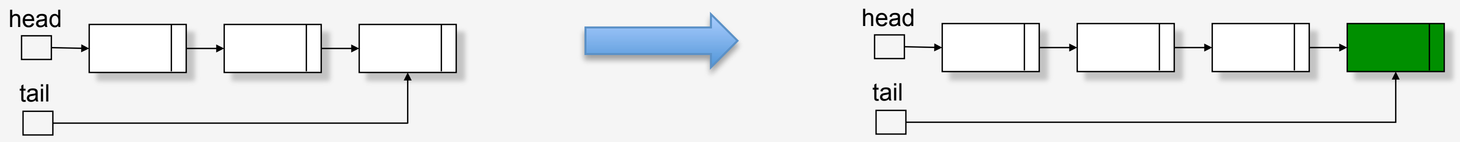
Empty



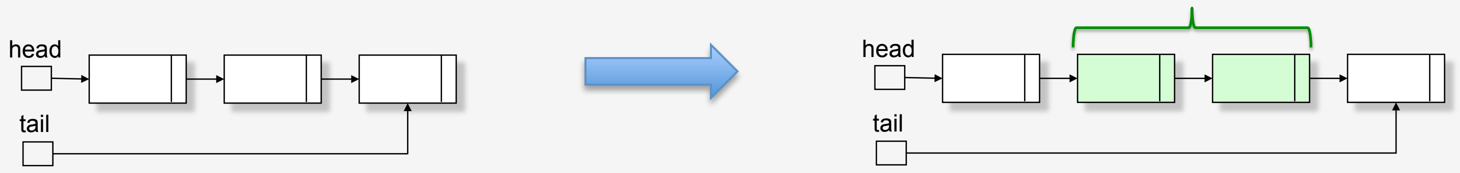
Non-Empty, index == 0



Non-Empty, index == size()



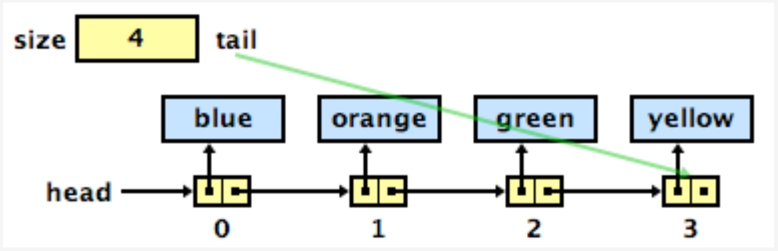
Non-Empty, index in the middle



Implementing a list collection: Nodes, add method

Validate index, allocate memory

```java
public void add (int index, T element)
{
   if ((index < 0) || (index > size))
   {
      throw new IndexOutOfBoundsException();
   }
   LinearNode<T> temp = new LinearNode<T>(element);
   if (isEmpty())
   {
      head = temp;
      tail = temp;
   }
   else if (index == 0)
   {
      temp.setNext(head);
      head = temp;
   }
   else if (index == size)
   {
      tail.setNext(temp);
      tail = temp;
    }
   else
   {
      LinearNode<T> p = head;
      for (int i = 0; i < index-1; i++)
      {
         p = p.getNext();
      }
      temp.setNext(p.getNext());
      p.setNext(temp);
   }
   size++;
}
```



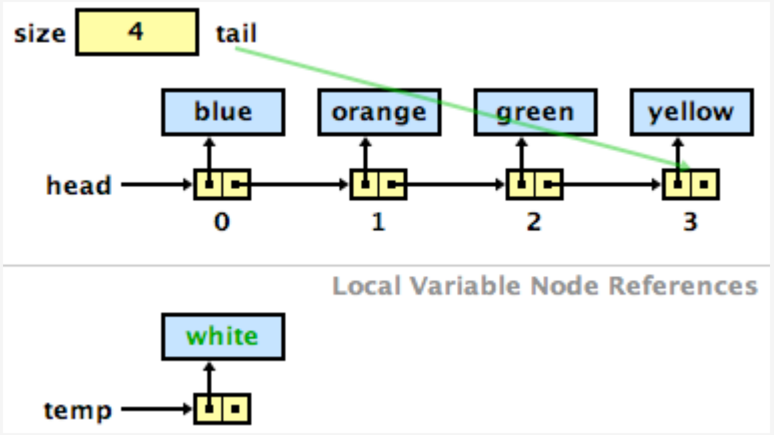
index < 0 || index > 4

```
list.add(-1, "red");
-----
Exception in evaluation thread
java.lang.IndexOutOfBoundsException
```

```
list.add(10, "red");
-----
Exception in evaluation thread
java.lang.IndexOutOfBoundsException
```

index >= 0 && index <= 4

```
list.add(2, “white");
```

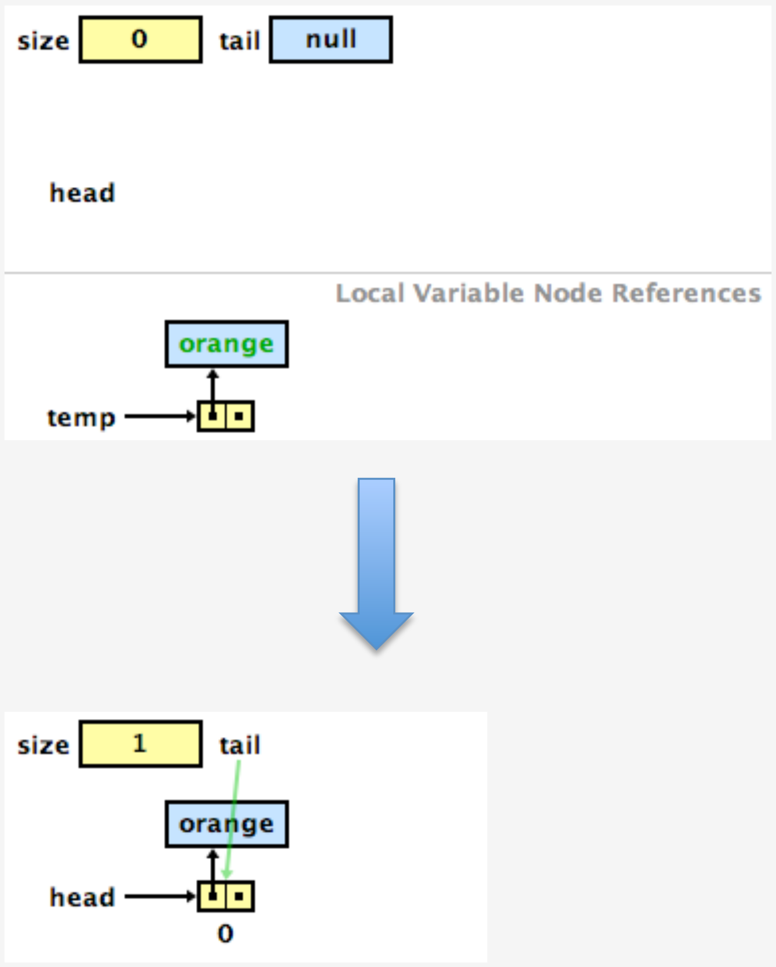


Implementing a list collection: Nodes, add method

Add to an empty list

```
public void add (int index, T element)
{
   if ((index < 0) || (index > size))
   {
      throw new IndexOutOfBoundsException();
   }
   Node<T> temp = new Node<T>(element);
   if (isEmpty())
   {
      head = temp;
      tail = temp;
   }
   else if (index == 0)
   {
      temp.setNext(head);
      head = temp;
   }
   else if (index == size)
   {
      tail.setNext(temp);
      tail = temp;
   }
   else
   {
      Node<T> p = head;
      for (int i = 0; i < index-1; i++)
      {
         p = p.getNext();
      }
      temp.setNext(p.getNext());
      p.setNext(temp);
   }
   size++;
}
```

```
list.add(0, “orange");
```

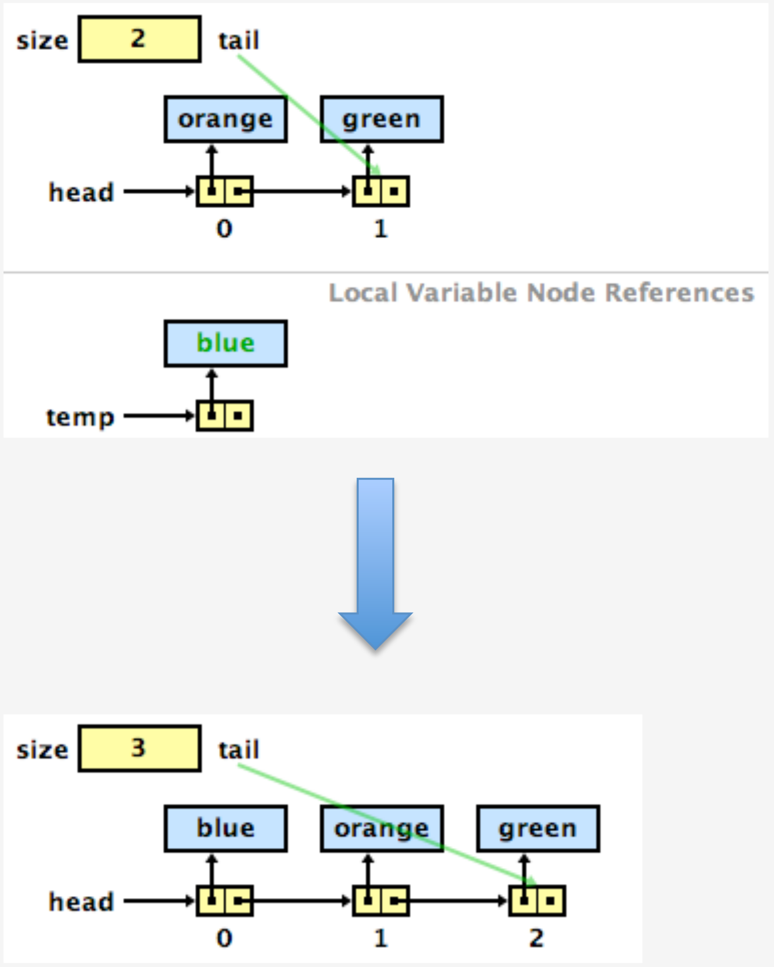


Implementing a list collection: Nodes, add method

Add to a non-empty list at index 0

```
public void add (int index, T element)
{
   if ((index < 0) || (index > size))
   {
      throw new IndexOutOfBoundsException();
   }
   Node<T> temp = new Node<T>(element);
   if (isEmpty())
   {
      head = temp;
      tail = temp;
   }
   else if (index == 0)
   {
      temp.setNext(head);
      head = temp;
   }
   else if (index == size)
   {
      tail.setNext(temp);
      tail = temp;
   }
   else
   {
      Node<T> p = head;
      for (int i = 0; i < index-1; i++)
      {
         p = p.getNext();
      }
      temp.setNext(p.getNext());
      p.setNext(temp);
   }
   size++;
}
```

```
list.add(0, “blue");
```

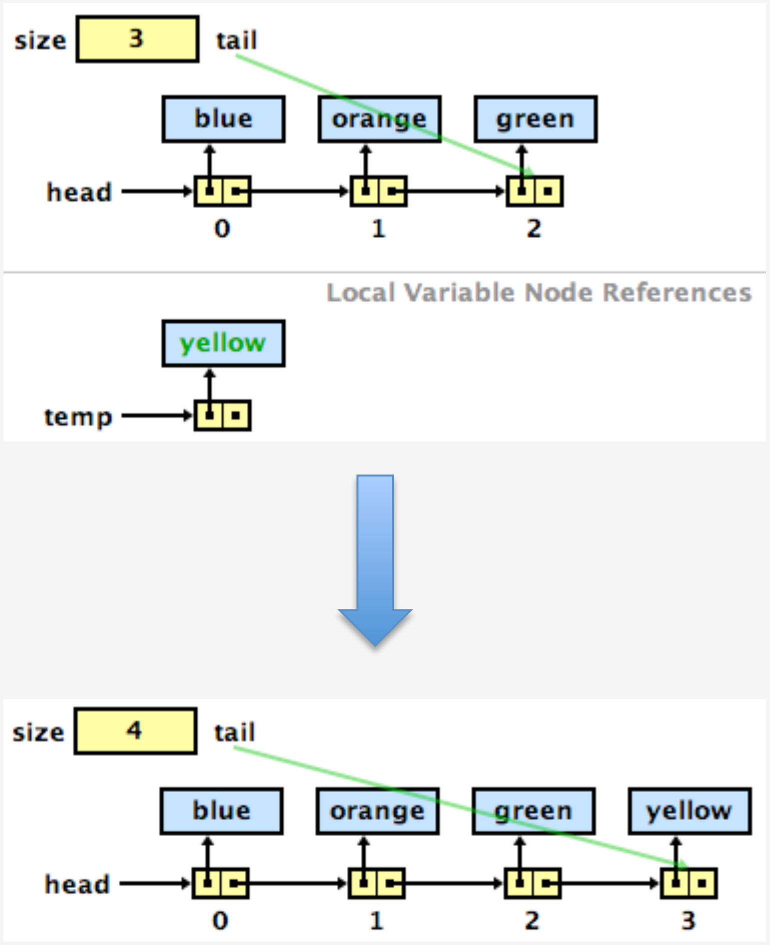


Implementing a list collection: Nodes, add method

Add to a non-empty list at index size()

```
public void add (int index, T element)
{
   if ((index < 0) || (index > size))
   {
      throw new IndexOutOfBoundsException();
   }
   Node<T> temp = new Node<T>(element);
   if (isEmpty())
   {
      head = temp;
      tail = temp;
   }
   else if (index == 0)
   {
      temp.setNext(head);
      head = temp;
   }
   else if (index == size)
   {
      tail.setNext(temp);
      tail = temp;
   }
   else
   {
      Node<T> p = head;
      for (int i = 0; i < index-1; i++)
      {
         p = p.getNext();
      }
      temp.setNext(p.getNext());
      p.setNext(temp);
   }
   size++;
}
```

```
list.add(list.size(), “yellow");
```



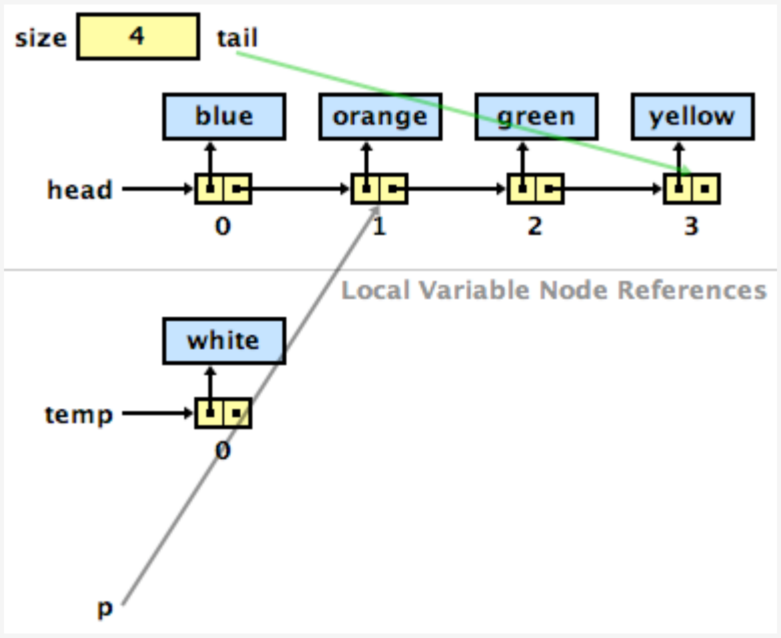
Implementing a list collection: Nodes, add method

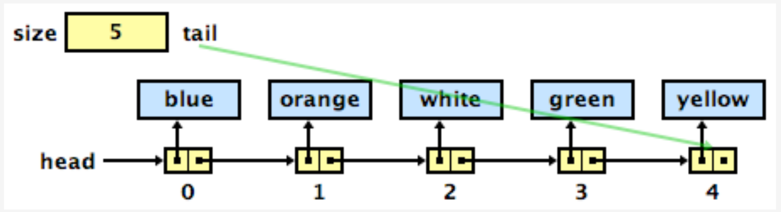
Add to a non-empty list in the middle

```
public void add (int index, T element)
{
   if ((index < 0) || (index > size))
   {
      throw new IndexOutOfBoundsException();
   }
   Node<T> temp = new Node<T>(element);
   if (isEmpty())
   {
      head = temp;
      tail = temp;
   }
   else if (index == 0)
   {
      temp.setNext(head);
      head = temp;
   }
   else if (index == size)
   {
      tail.setNext(temp);
      tail = temp;
   }
   else
   {
      Node<T> p = head;
      for (int i = 0; i < index-1; i++)
      {
         p = p.getNext();
      }
      temp.setNext(p.getNext());
      p.setNext(temp);
   }
   size++;
}
```

Slightly different traversal pattern

```
list.add(2, “white");
```





Nodes, add method – time complexity

```
public void add (int index, T element)
{
   if ((index < 0) || (index > size))
   {
      throw new IndexOutOfBoundsException();     O(1)
   }
   Node<T> temp = new Node<T>(element);
   if (isEmpty())
   {
      head = temp;                O(1)
      tail = temp;
   }
   else if (index == 0)
   {
      temp.setNext(head);            O(1)
      head = temp;
   }
   else if (index == size)
   {
      tail.setNext(temp);            O(1)
      tail = temp;
   }
   else
   {
      Node<T> p = head;
      for (int i = 0; i < index-1; i++)
      {
         p = p.getNext();                   O(N)
      }
      temp.setNext(p.getNext());
      p.setNext(temp);
   }
   size++;
}
```

The add method is O(N).

Important point:

The physical insertion is O(1).

Finding where the new element goes is O(N).

performance comparison

Performance analysis

	Indexed List		Non-indexed List		Self-ordered List	
method	Array	Nodes	Array	Nodes	Array	Nodes
remove(element)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
addAfter(element, target)	•	•	O(N)	O(N)	•	•
add(element)	O(1)	O(1)	•	•	O(N)	O(N)
add(index, element)	O(N)	O(N)	•	•	•	•
get(index)	O(1)	O(N)	•	•	•	•
indexOf(element)	O(N)	O(N)	•	•	•	•

Tell me why … 🟥 Can we do better… 🟦

If we could use binary search on a node-based structure, then add() and remove() would be O(log N) – a huge improvement!

Stay tuned … this is exactly where we're headed.