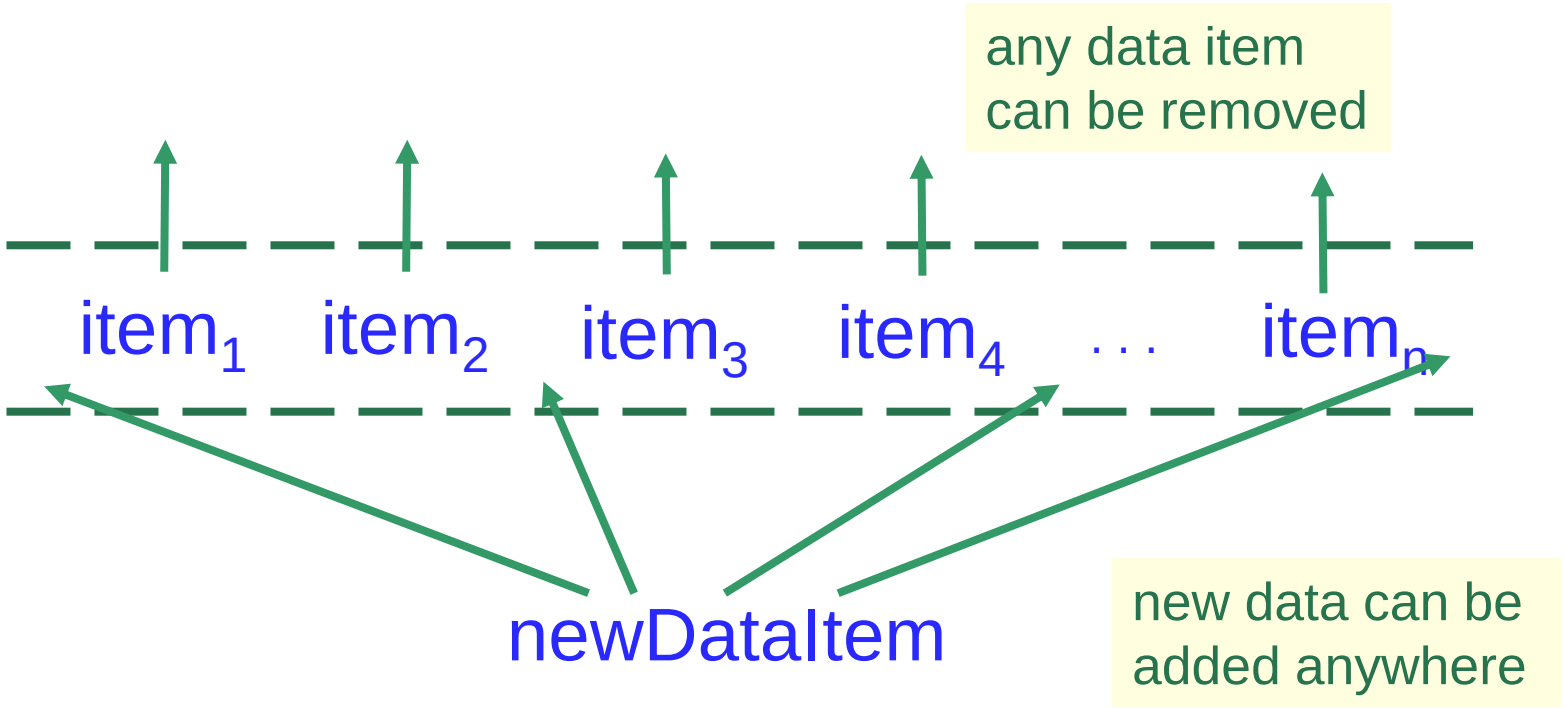
Lists

Objectives

- Define the List abstract data type
- Examine different types of lists
- Compare various list implementations
- Use the Comparable interface to compare objects of a generic type

Lists

A list is a collection of data items with a linear ordering, like stacks and queues, but more flexible: adding and removing data items does *not* have to happen at the ends of the list

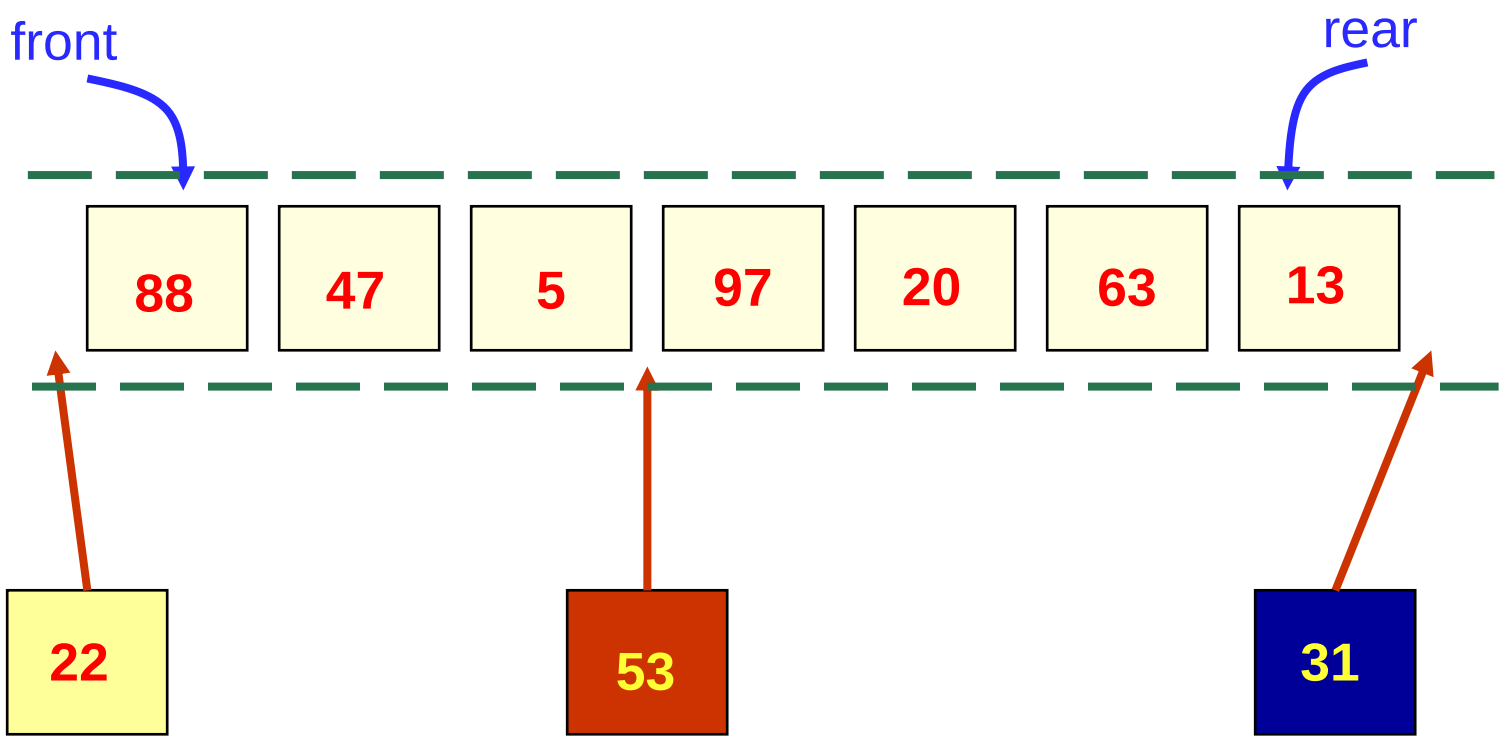


Lists

- We will consider three types of lists:
  - unordered lists
  - ordered lists
  - indexed lists

Unordered Lists

The data items do not appear in a particular order.



New values can be inserted anywhere in the list

Ordered Lists

The data items of the list are ordered by their value. The value of a data item depends on its type.

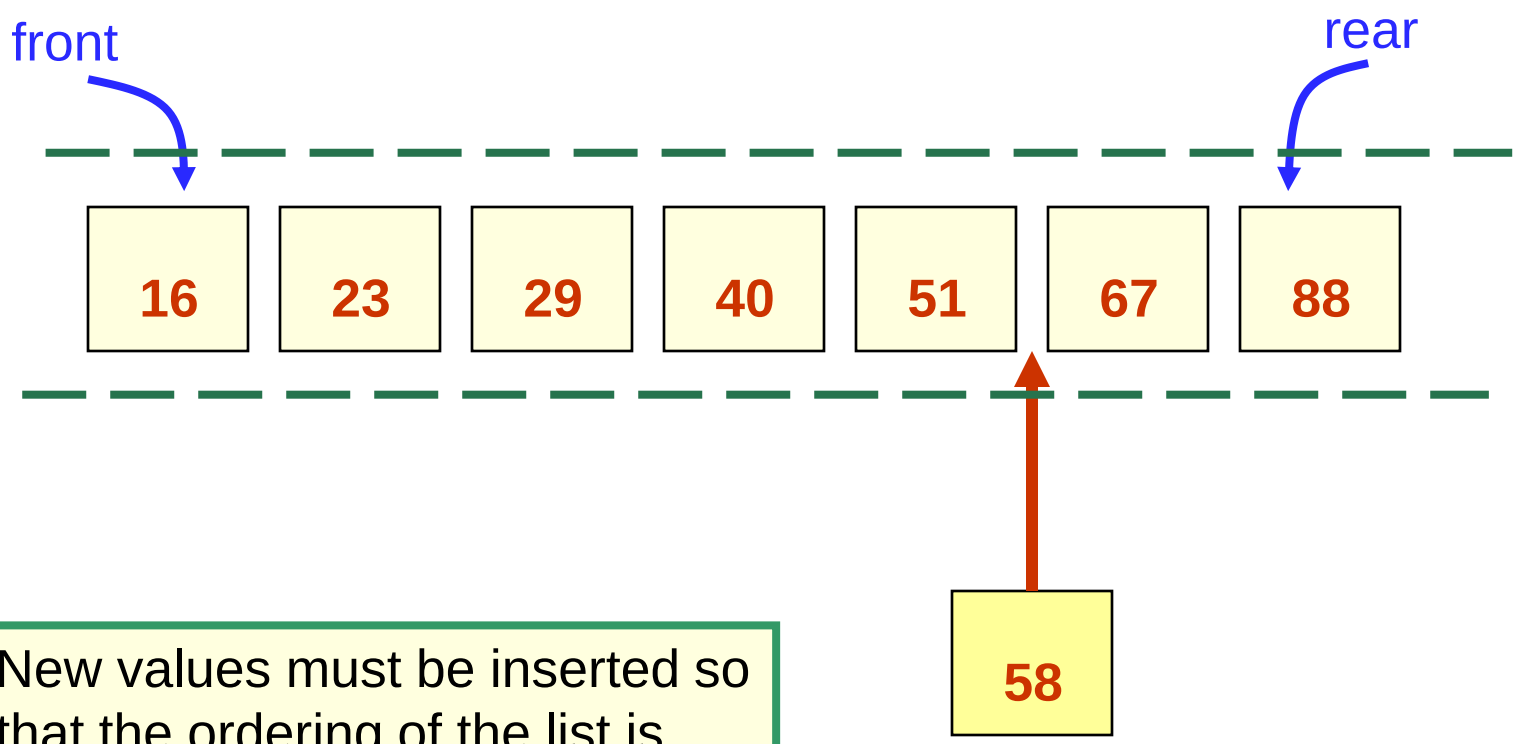
- For example, names can be assigned values alphabetically or lexicographically

Doe Joe	Fair Dean	Hill Hilly	Mo Moe	Pea Pete

- Course grades can be assigned values equal to the numeric grades.

16.8	25.7	44.0	56.7	67.7	75.6	78.7	96.5

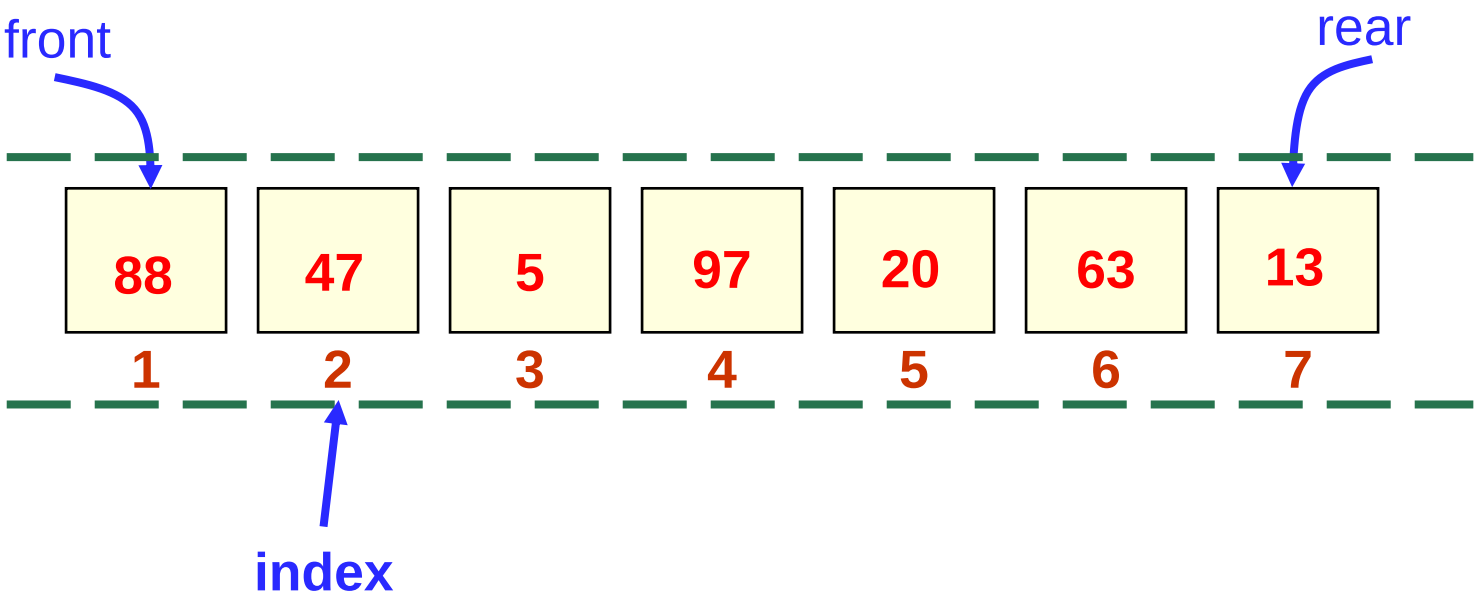
Conceptual View of an Ordered List



New values must be inserted so that the ordering of the list is maintained

Indexed Lists

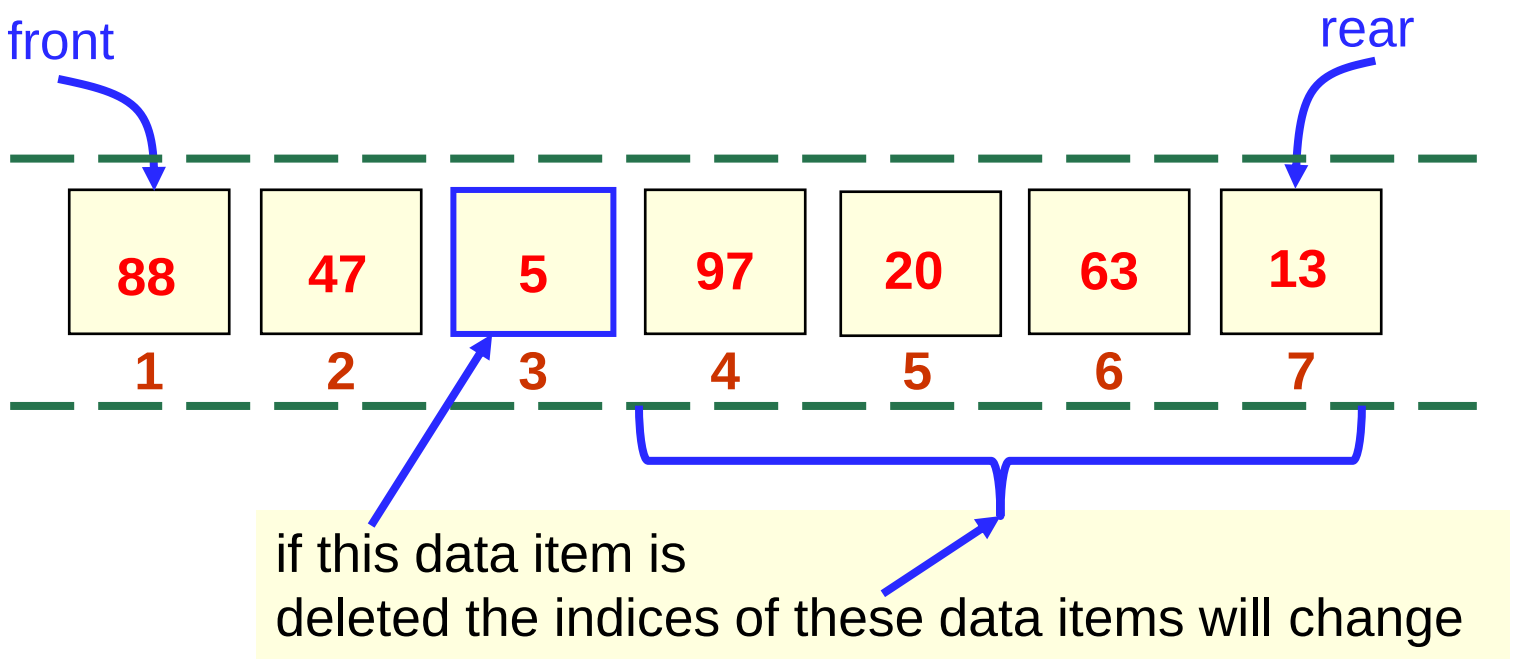
The data items are referenced by their position in the list, called their index



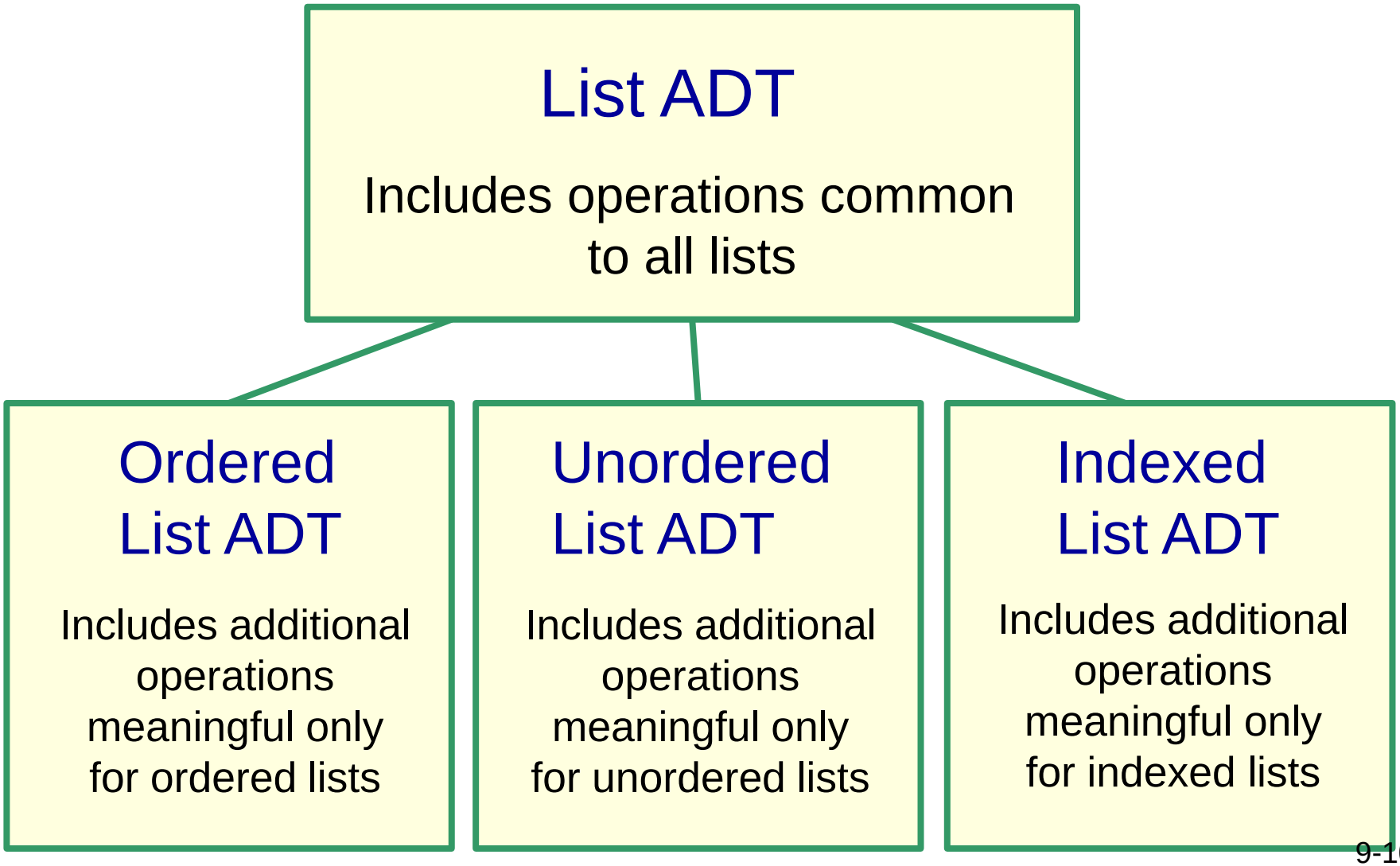
Note that indices do not need to start at 0. A List is and ADT, not an array!

Indexed Lists

When the list changes, the index of some data items may change



List Hierarchy



The List ADT

Operation	Description
removeFirst	Removes the first data item from the list
removeLast	Removes the last data item from the list
remove(dataItem)	Removes the given dataItem from the list
first	Gets the data item at the front of the list
last	Gets the data item at the rear of the list
contains(dataItem)	Determines if a particular data item is in the list
isEmpty	Determines whether the list is empty
size	Determines the number of data items in the list
toString	Returns a string representation of the list

Operation Particular to an Ordered List

Operation	Description
add (dataItem)	Adds *dataItem* to the list in the correct place so the resulting list is ordered

Operations Particular to an Unordered List

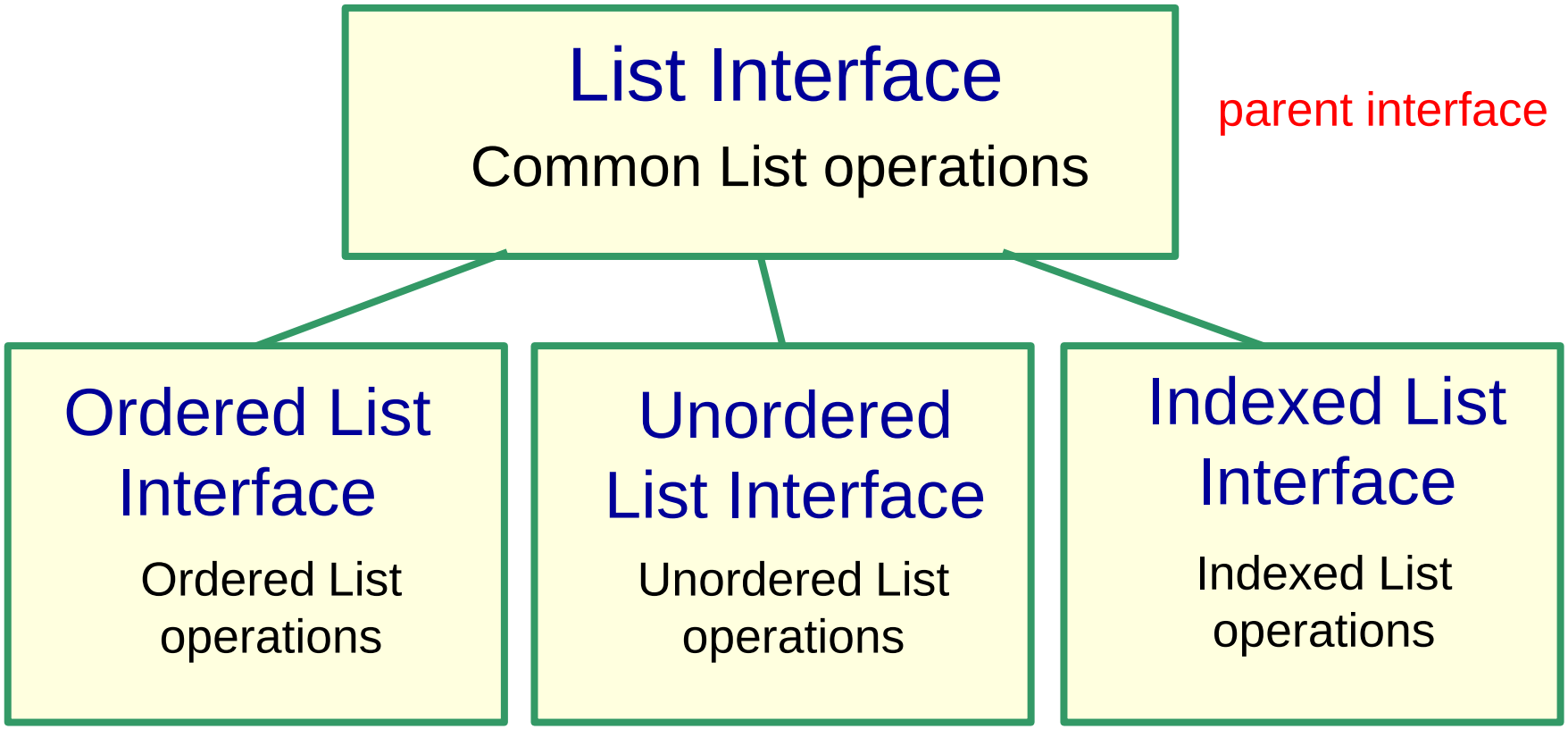
Operation	Description
addToFront	Adds a data item to the front of the list
addToRear	Adds a data item to the rear of the list
addAfter(dataItem)	Adds a data item after a particular dataItem already in the list

Operations Particular to an Indexed List

Operation	Description
add (index,dataItem)	Adds a dataItem at the specified index
set (index,dataItem)	Sets dataItem at the specified index overwriting any data that was there
get (index)	Returns the data item at the specified index
indexOf (dataItem)	Returns the index of dataItem
remove (index)	Removes and returns the data item at the specified index

Java Interface Hierarchy

There is a Java interface hierarchy, similar to the Java class hierarchy.



ListADT Interface

```
public interface ListADT<T> {
   public T removeFirst ( ); //  Removes and returns first data item
   public T removeLast ( ); //  Removes and returns last data item
   public T remove (T dataItem); //  Removes and returns dataItem
   public T first ( );  //  Returns the first data item
   public T last ( );  //  Returns the last data item
   public boolean isEmpty( ); //  Returns true if this list is empty
   //  Returns true if this list contains target
   public boolean contains (T target);
   public int size( ); //  Returns the number of data items in this list
   public String toString( ); //  String representation of this list
}
```

OrderedListADT Interface

```
public interface OrderedListADT<T> extends ListADT<T> {
   //  Adds dataItem to this list at the correct location to keep
   // the list sorted
   public void add (T dataItem);
}
```

UnorderedListADT

```
public interface UnorderedListADT<T> extends ListADT<T> {
   //  Adds the specified dataItem to the front of this list
   public void addToFront (T dataItem);

   //  Adds the specified dataItem to the rear of this list
   public void addToRear (T dataItem);

   //  Adds the specified dataItem after the specified target
   public void addAfter (T dataItem, T target);
}
```

IndexedListADT

```
public interface IndexedListADT<T> extends ListADT<T> {
  //  Inserts the specified dataItem at the specified index
  public void add (int index, T dataItem);

  //  Sets dataItem at the specified index
  public void set (int index, T dataItem);

  //  Returns a reference to the data item at specified index
  public T get (int index);

  //  Returns the index of the specified dataItem
  public int indexOf (T dataItem);

  //  Removes and returns the data item at specified index
  public T remove (int index);
  }
```

Discussion

- Note that the remove method in the IndexedList ADT is overloaded
  - Why? Because there is a remove method in the parent ListADT
    - This is *not* overriding, because the parameters are different

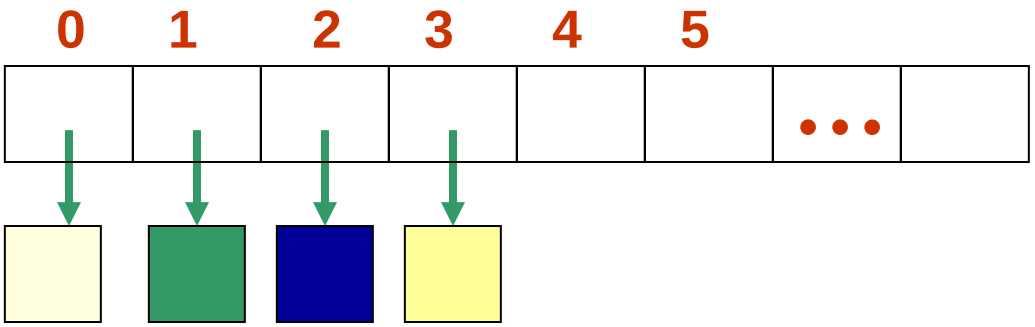
Implementations of the List ADT

We will study several implementations of the Ordered List ADT.

We leave it as an exercise that you implement the List ADT, the Unordered List ADT and the Indexed List ADT.

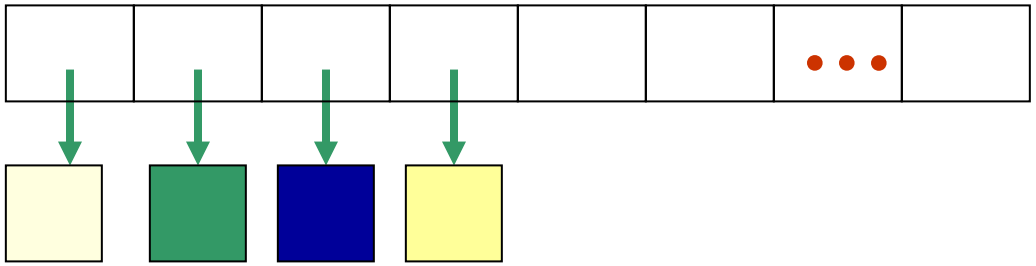
Array Implementation of a List

- We store the data items in an array
- Fix first data item of the list at index 0
- Do we need to shift the data when a new data item is added
  - at the front?
  - somewhere in the middle?
  - at the end?



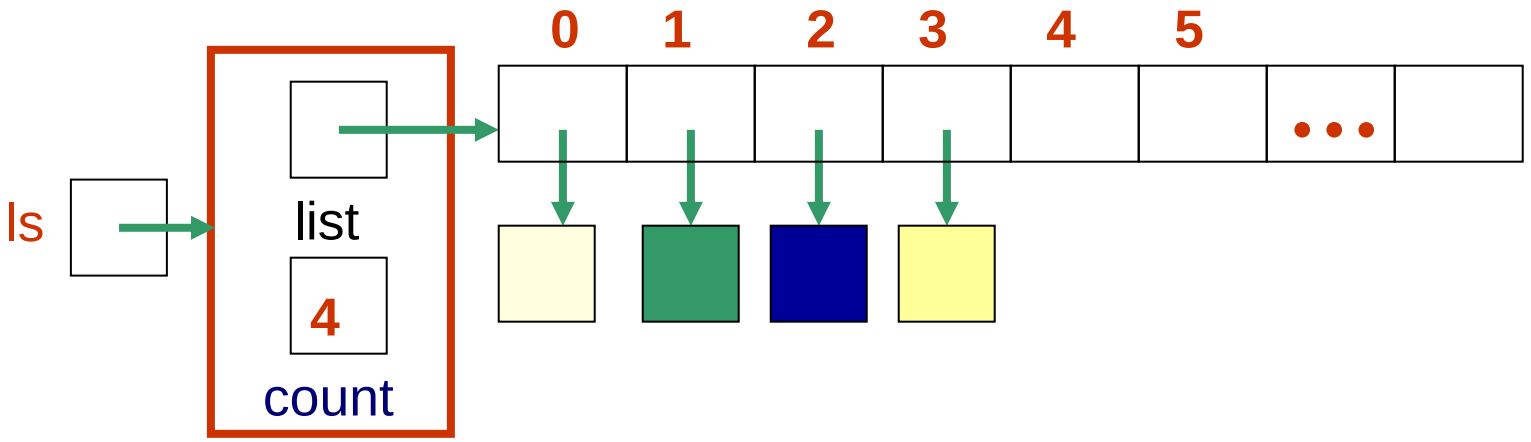
Array Implementation of a List

- Do we need to shift data when a data item is removed
  - from the front?
  - from somewhere in the middle?
  - from the end?



Array Implementation of a List

- We store the data items in an array
- Variable *count* indicates the number of data items in the list



The Ordered List ADT

The ordered List ADT includes all the operations of the List ADT, plus the add operation:

- add (T dataItem): Adds *dataItem* to the ordered list in the correct place so the resulting list is still ordered

The Ordered List ADT

Note that to order a list we need to be able to compare objects of generic type T.

How can we do this if we do not know what class of object T refers to?

The **Comparable** Interface

The Java Comparable interface allows the comparison of objects of a generic type.

The Java Comparable interface has only one operation:

compareTo (T *otherObject*) :

- returns a negative value if **this** object is less than *otherObject*
- returns zero if **this** objects is equal to *otherObject*
- returns a positive value if **this** object is greater than *otherObject*

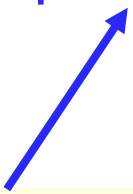
The **Comparable** Interface

```
public class Comparable<T> {
   /* Compares this object with the specified object for
      order. Returns a negative integer, zero, or a positive
      integer as this object is less than, equal to, or greater
      than the specified object.
      Throws NullPointerException if otherObject is null
      Throws ClassCastException if otherObject's type
      prevents it from being compared to this object
    */

   public int compareTo (T otherObject) throws
                  NullPointerException, ClassCastException;

}
```

The **Comparable** Interface

Since for a generic type T the *compiler* does not know whether or not the actual class implements the compareTo method, to compare two variables of type T we cast one of them to be of type Comparable:

```
public class ClassA<T> {
   public void check(T var1, T var2) {
      Comparable<T> tmp = (Comparable<T>) var1;
      if (tmp.compareTo(var2) == 0)
      …
   }
```



This allows us to compare the objects referenced by var1 and var2 as long as the class of the objects implements the interface Comparable

The **Comparable** Interface

If the actual class of the objects referenced by var1 and var2 does not implement the Comparable interface the program will produce a runtime error.

```
public class ClassA<T> {
   public void check(T var1, T var2) {
      Comparable<T> tmp = (Comparable<T>) var1;
      if (tmp.compareTo(var2) == 0)
      …
   }
}
```

For example,

```
ClassA<String> v1 = new ClassA<String>();
v1.check("hello","hi");
```

will not cause a runtime error because class String implements the Comparable interface

The **Comparable** Interface

```
public class ClassA<T> {
   public void check(T var1, T var2) {
      Comparable<T> tmp = (Comparable<T>) var1;
      if (tmp.compareTo(var2) == 0)
      …
   }
}
```

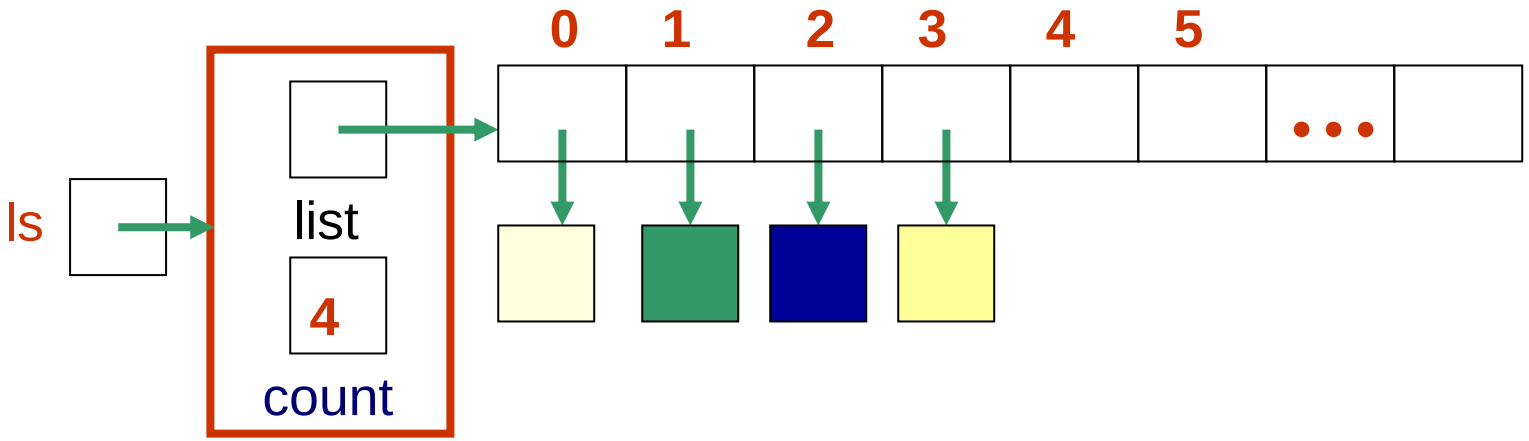
However,

```
ClassA<int[]> v1 = new ClassA<int[]>();
int[] a = {1,2,3};
int[] b = {1,2,3};
v1.check(a,b);
```

will throw a ClassCastException because arrays do not implement the Comparable interface

Array Implementation of the Ordered List ADT

Operation	Description
add (dataItem)	Adds *dataItem* to the list in the correct place so the resulting list is ordered

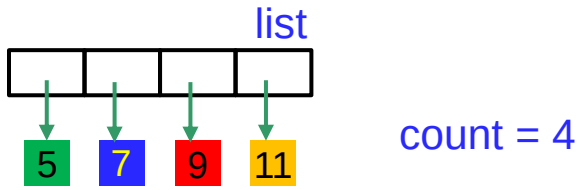


The **Add Operation**

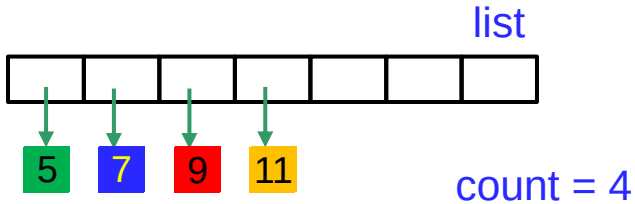
Algorithm add (dataItem)
In: new data item
Out: Nothing, but *dataItem* is added to
list keeping the data sorted

```
if count = list length then expandCapacity()
i = 0
while (i < count) and (dataItem > list[i]) do
    i = i + 1
for j = count downto i + 1 do
   list[j] = list[j-1]
list[i] = dataItem
count = count + 1
```

We need to consider 2 cases



add (8)



The **Add Operation**

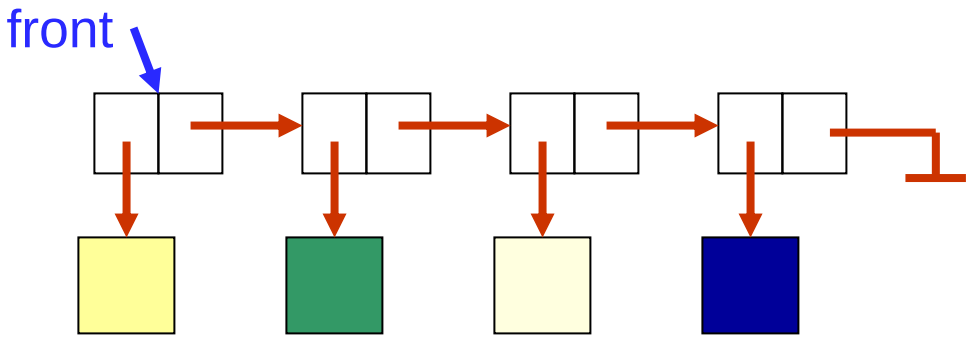
```
//  Adds dataItem to the list keeping it sorted.
public void add (T dataItem) {
   if (count == list.length) expandCapacity( );
   Comparable<T> temp = (Comparable<T>) dataItem;
   int i = 0;
   // Find first value larger than or equal to dataItem
   while (i < count && temp.compareTo(list[i]) > 0) i++;
   if (i < count)
         // Shift values larger than dataItem to the right
         for (int j = count; j > i; j--) list[j] = list[j - 1];
   list[i] = dataItem;
   count++;
}
```

List Implementation Using Circular Arrays

- Recall circular array implementation of queues
- ***Exercise***: implement list and ordered list operations using a circular array implementation

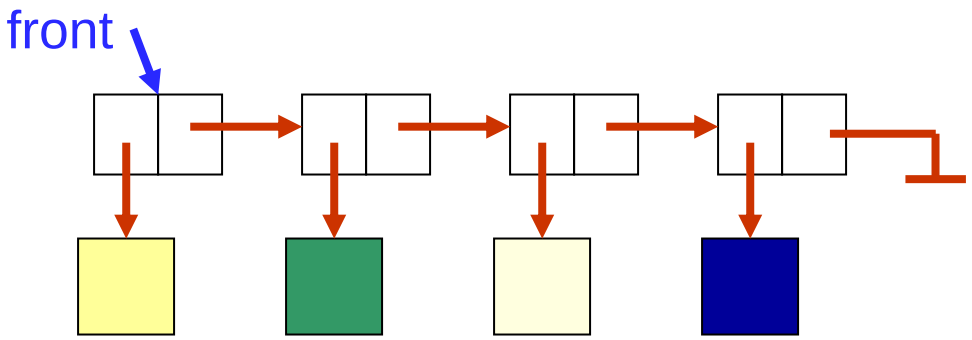
List Implementation Using Links

A list can be represented as a *linked list of nodes*, with each node containing a data item.

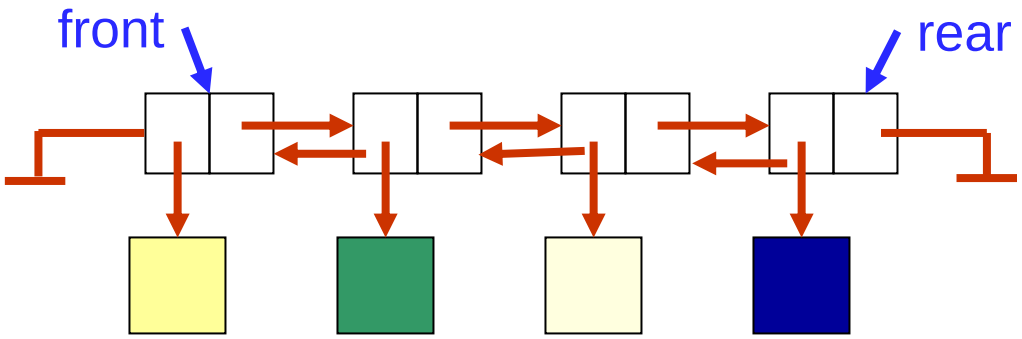


List Implementation Using Links

We will first examine the **add** operation for a singly-linked list implementation.



Then we will look at the remove operation for a doubly-linked list.

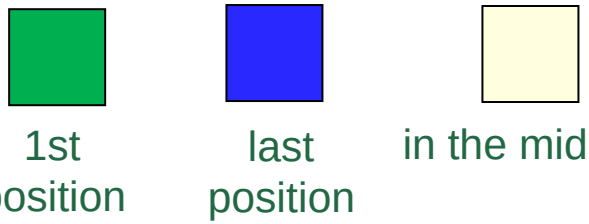


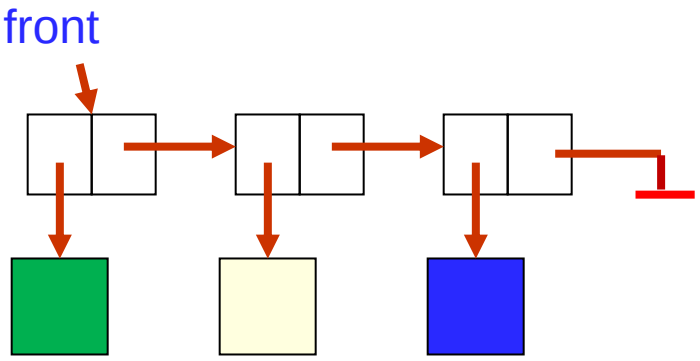
The **Add Operation**

```
Algorithm add (newDataItem)
In: data item to add
Out: Nothing, but add dataItem to the sorted list
node = new node storing newDataItem
prev = null
current = front
while (current ≠ null) and
        (current.getDataItem() < newDataItem) do {
    prev = current
    current = current.getNext()
}
if front = null then front = node
else
    if current = front then {
        node.setNext(front)
        front = node
    }
    else {
        node.setNext(current)
        prev.setNext(node)
count = count + 1
```

Need to consider several cases

newDataItem is added at:



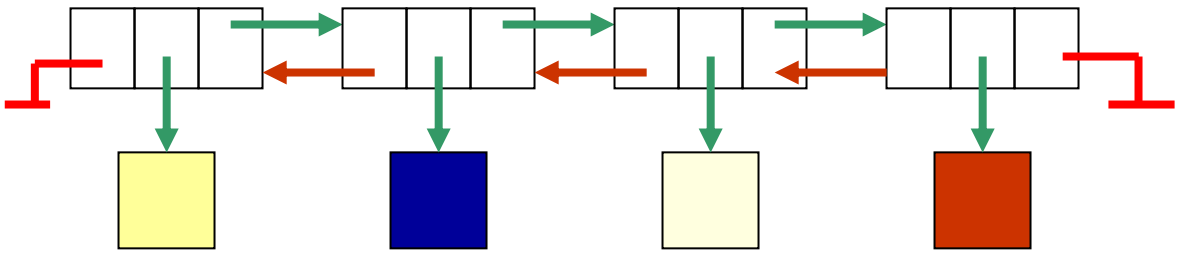


The Add Operation

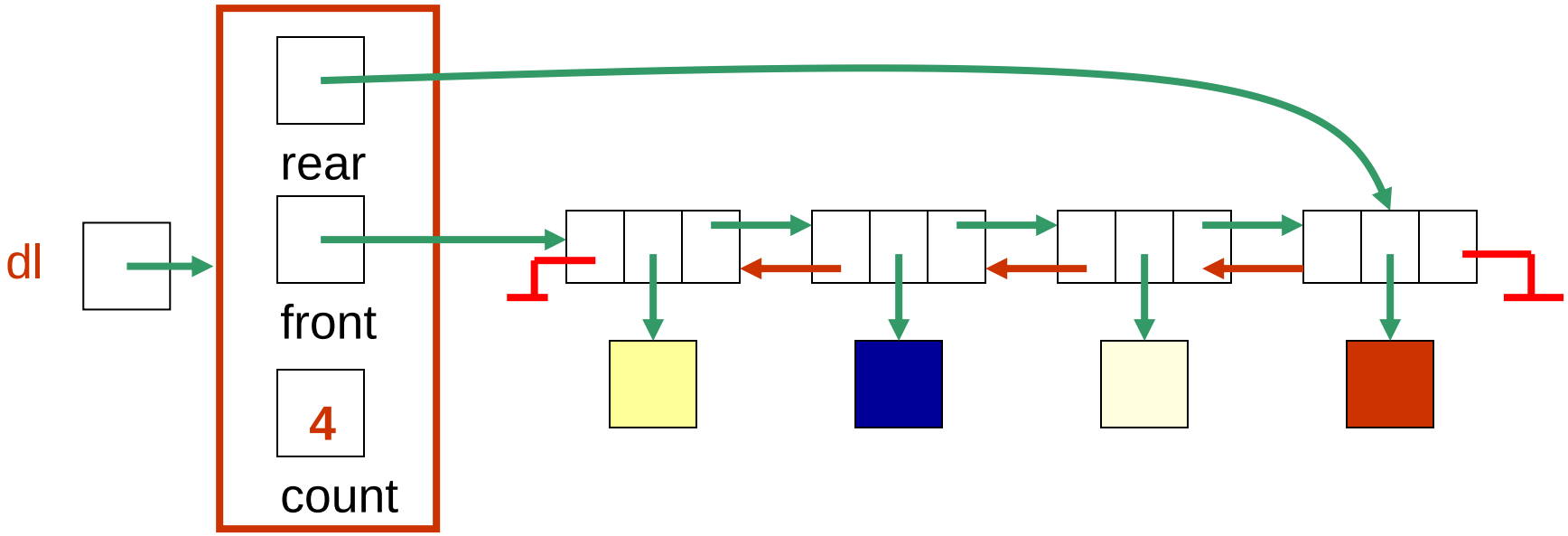
```
public void add (T newDataItem) {
   LinearNode<T> node = new LinearNode<T>(newDataItem);
   Comparable<T> temp = (Comparable<T>) newDataItem;
   LinearNode<T> prev = null, current = front;
   while ((current != null) && (temp.compareTo(current.getDataItem()) > 0))
         prev = current;
         current = current.getNext( );
   }
   if (front == null) front = node;
   else
        if current = front then {
              node.setNext(front)
              front = node
         }
        else {
             node.setNext(current)
             prev.setNext(node)
   count++;
}
```

Doubly Linked Lists

- A doubly linked list has two references in each node:
  - One to the next node in the list
  - One to the previous node in the list
- This makes moving back and forth in a list easier, and eliminates the need for a reference to the previous node in the list



Doubly Linked List Implementation of the Ordered List ADT



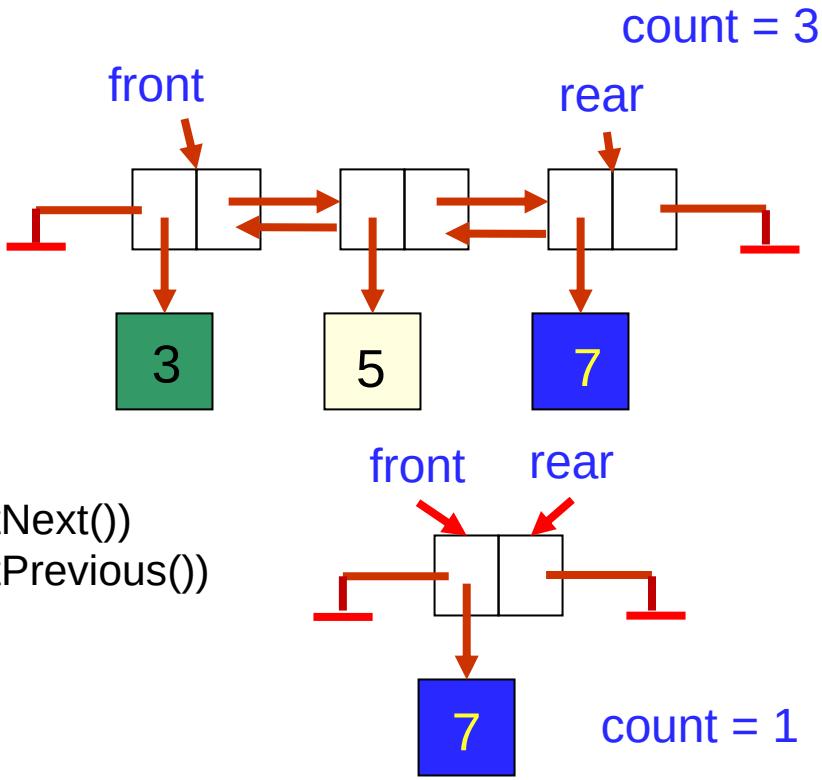
The **Remove Operation**

```
Algorithm remove (target)
In: data item to remove from the list
Out: Nothing, but remove target from the list if it
       is there
current = front
while current ≠  null and current.getDataItem() ≠ target do
    current = current.getNext()
if current = null then return
if count = 1 then front = rear = null
if current = front then {
    front = front.getNext()
    front.setPrevious(null)
}
else if current = rear then {
          rear = rear.getPrevious()
          rear.setNext(null)
       }
       else {
             (current.getPrevious()).setNext(current.getNext())
             (current.getNext()).setPrevious(current.getPrevious())
      }
count = count - 1
```

Need to consider several cases

target is:

- not in the list
- at front node
- at rear node
- at another node



The Remove Operation

```
//  Remove target from the list,  keeping the list sorted.
public void remove (T target) {
```

Exercise

Write java code for this method.