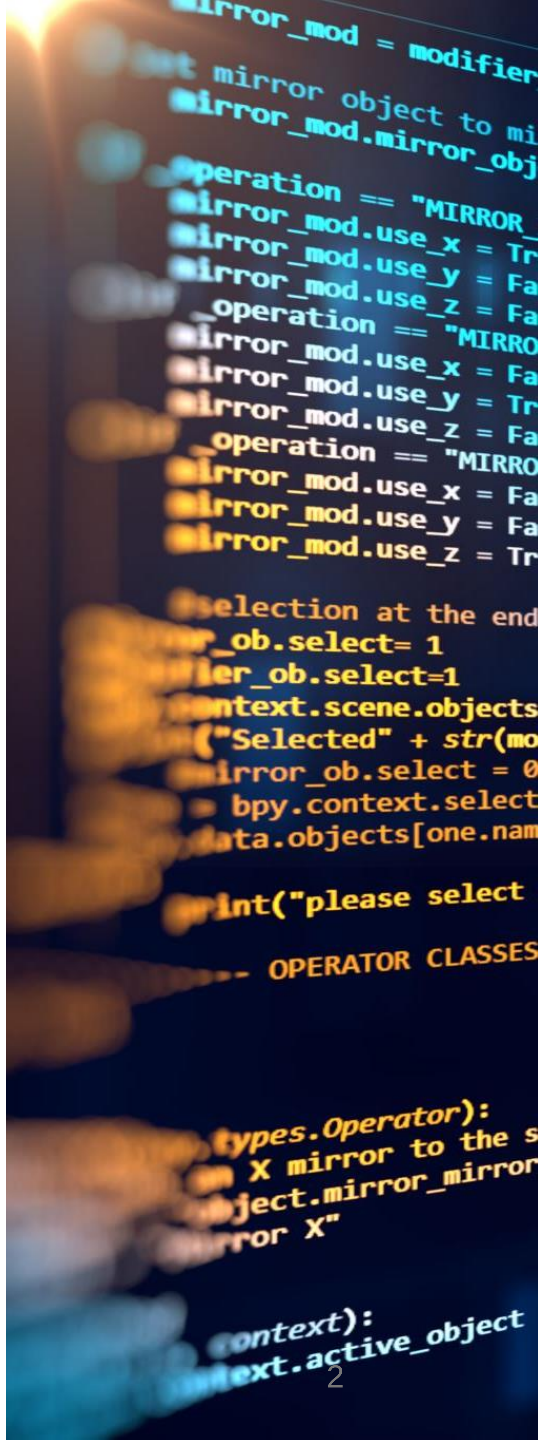
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

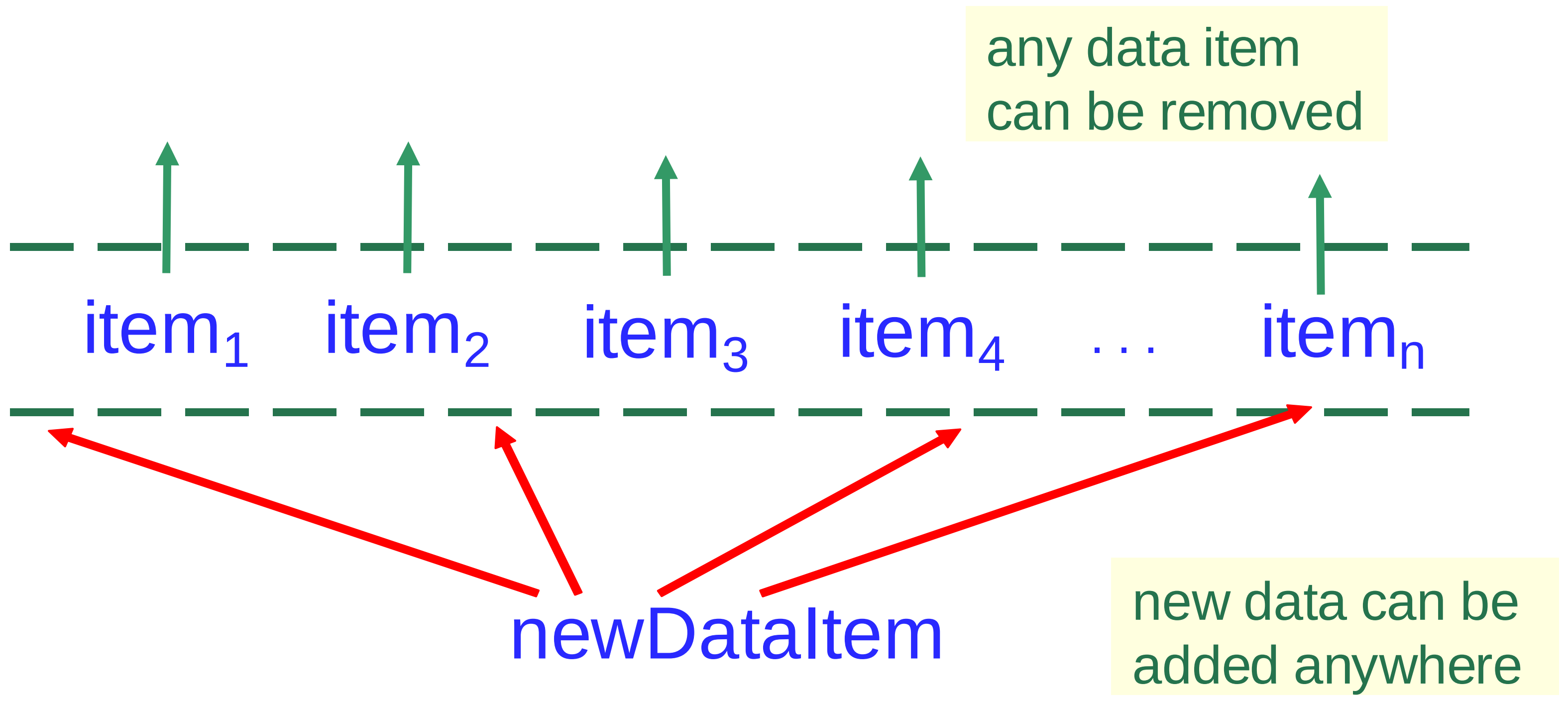
List ADT

Ahmed Ibrahim



List ADT

- 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.



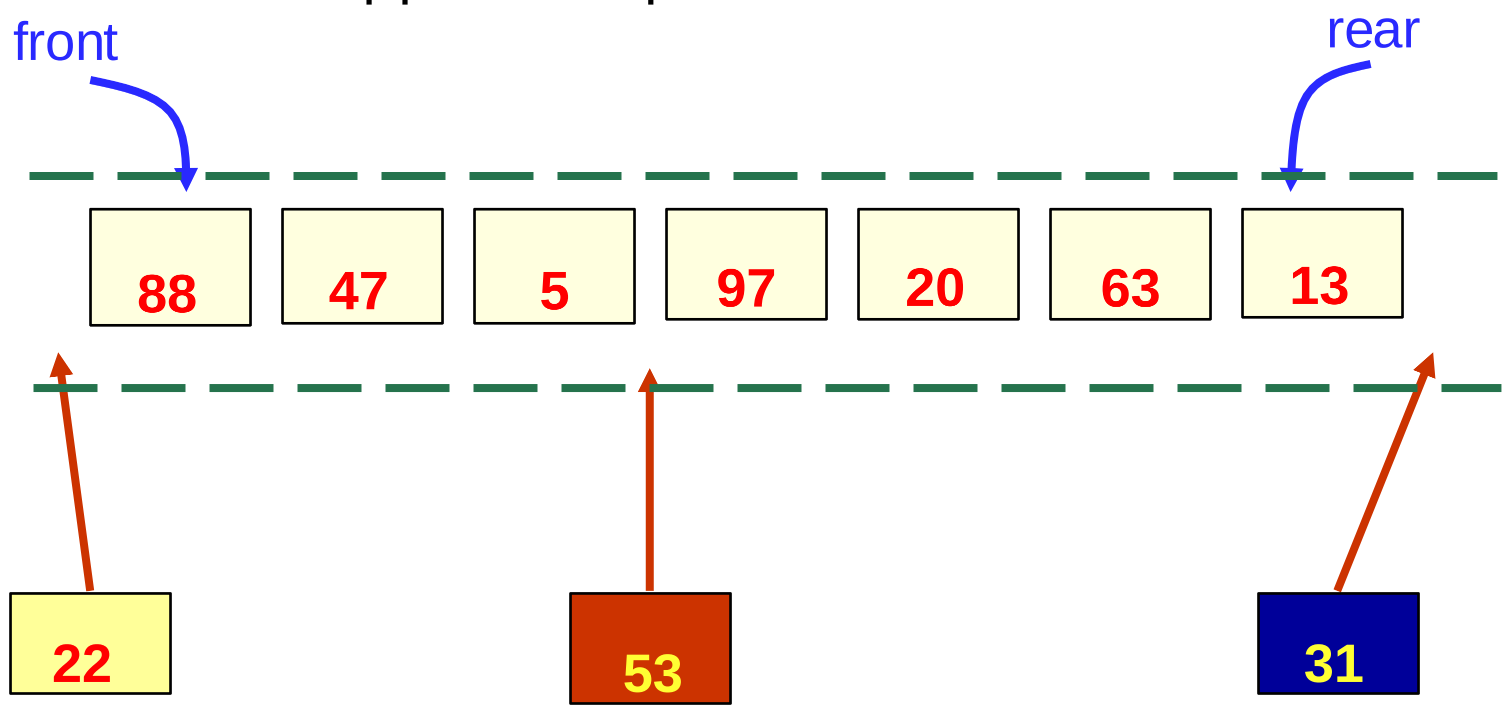
Lists

We will consider three types of lists:

- **Unordered lists** are lists where the items are displayed without specific order.
- **Ordered lists** have a specific sequence, often shown with numbers or letters. The order of the items is meaningful.
- **Indexed lists**—Lists where each item is associated with an **index** or a specific position, making accessing items based on their index easier.

Unordered Lists

- The data items do not appear in a particular order.



New values can be inserted anywhere in the list

Ordered Lists

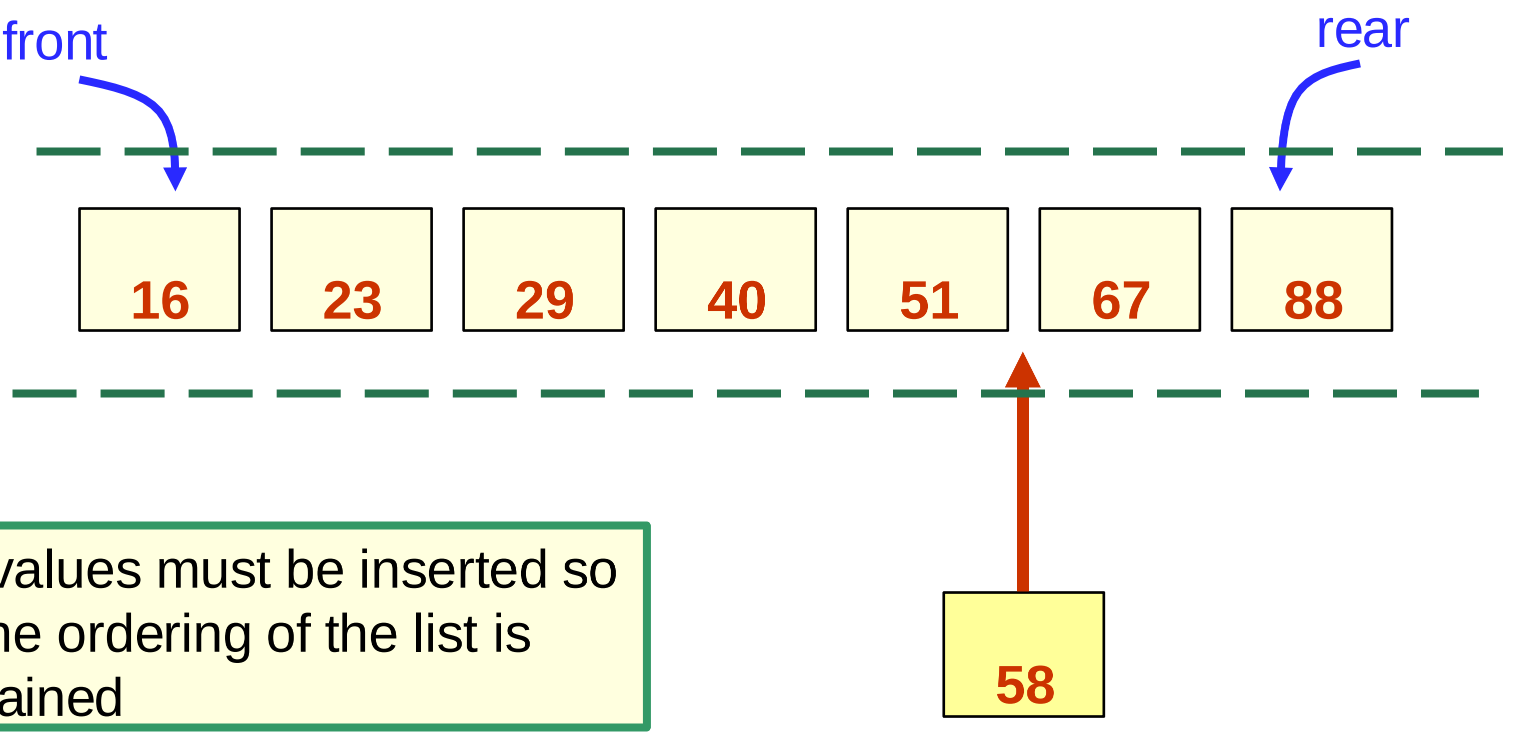
- The data items of the list are ordered by their value. The value of a data item depends on its type.
- For example, names can be assigned values alphabetically or lexicographically.
  - Lexicographical ordering is like the way words are sorted in a dictionary.

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

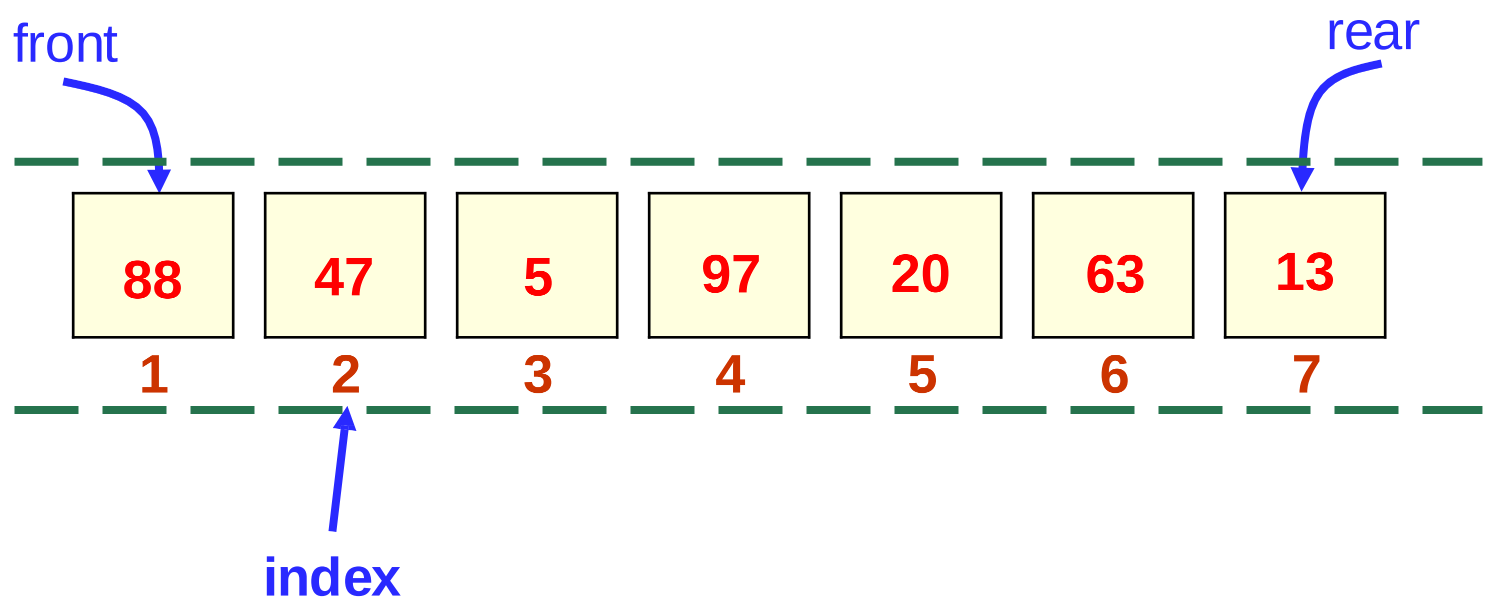
Conceptual View of an Ordered List



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

Indexed Lists

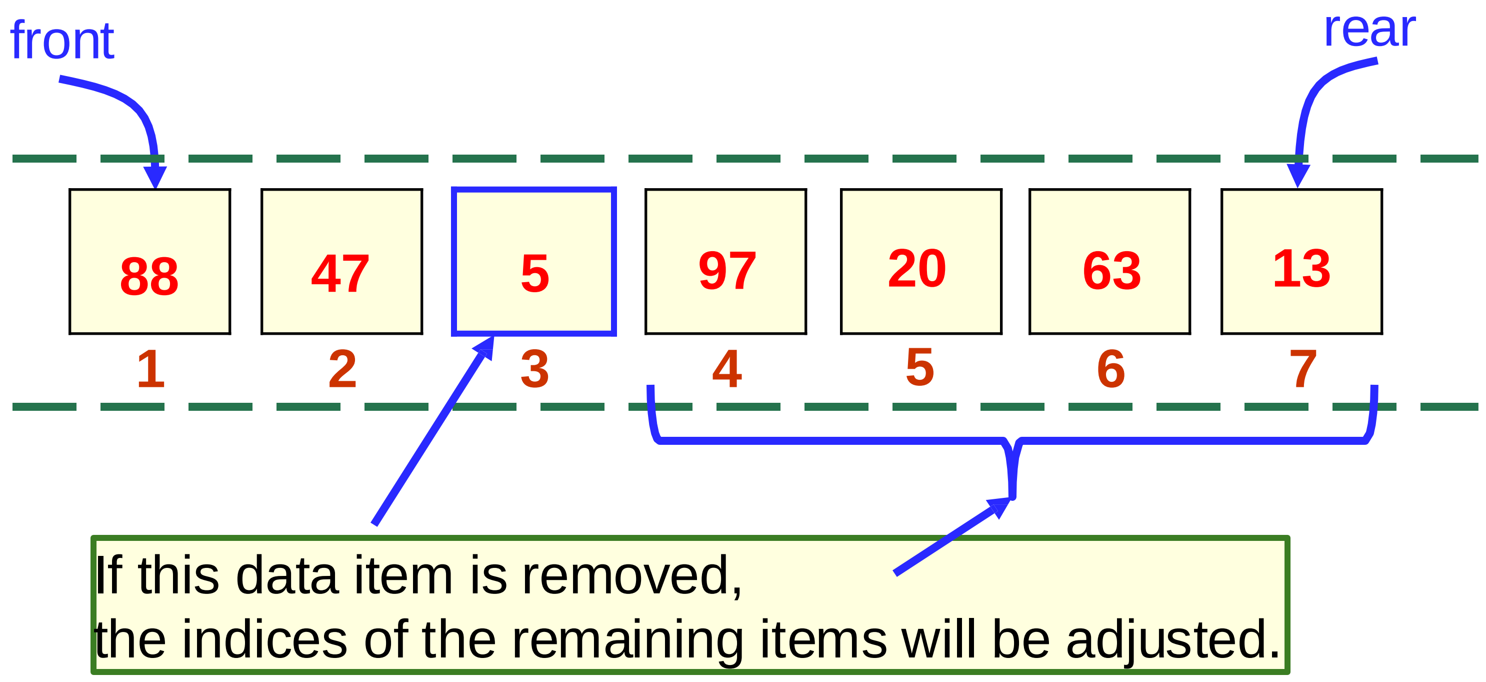
- The data items are referenced by their position in the list, called their index.



Keep in mind that indices don't have to begin at 0. A list is an Abstract Data Type (ADT), not an array!

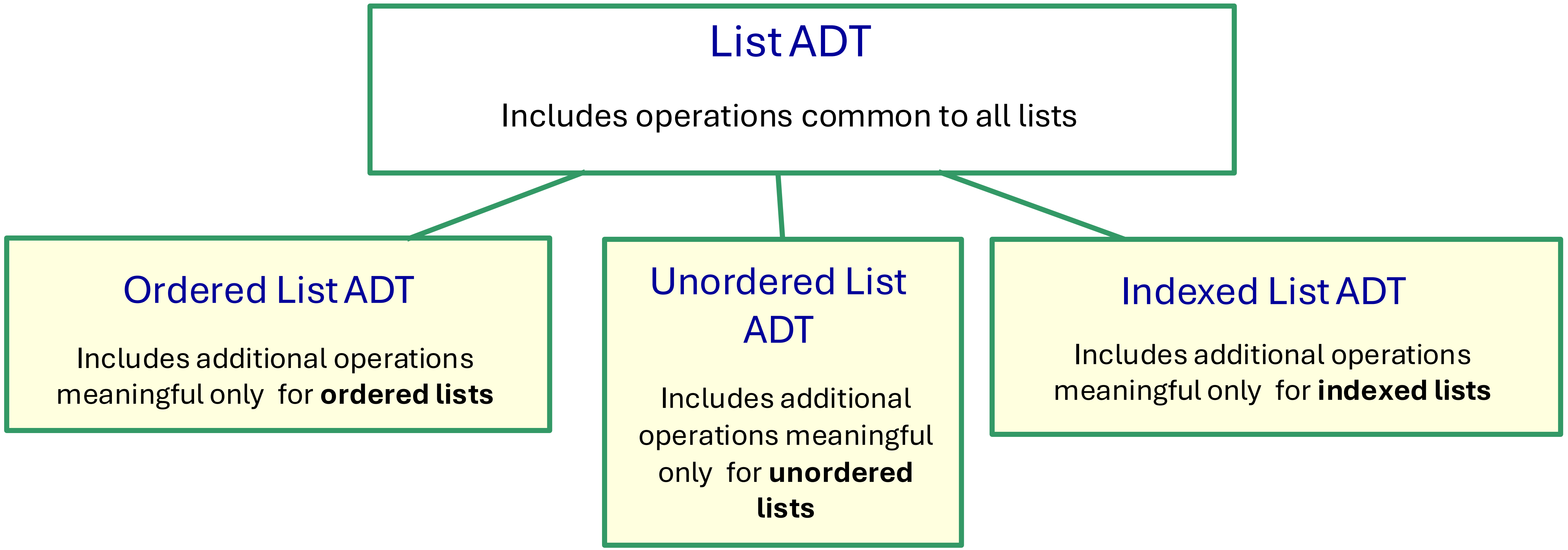
Indexed Lists (cont.)

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



front
rear
88
47
5
97
20
63
13
1
2
3
4
5
6
7
If this data item is removed,
the indices of the remaining items will be adjusted.

List Hierarchy



List ADT

Includes operations common to all lists

Ordered List ADT

Includes additional operations meaningful only for **ordered lists**

Unordered List ADT

Includes additional operations meaningful only for **unordered lists**

Indexed List ADT

Includes additional operations meaningful only for **indexed lists**

The List ADT Operations

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

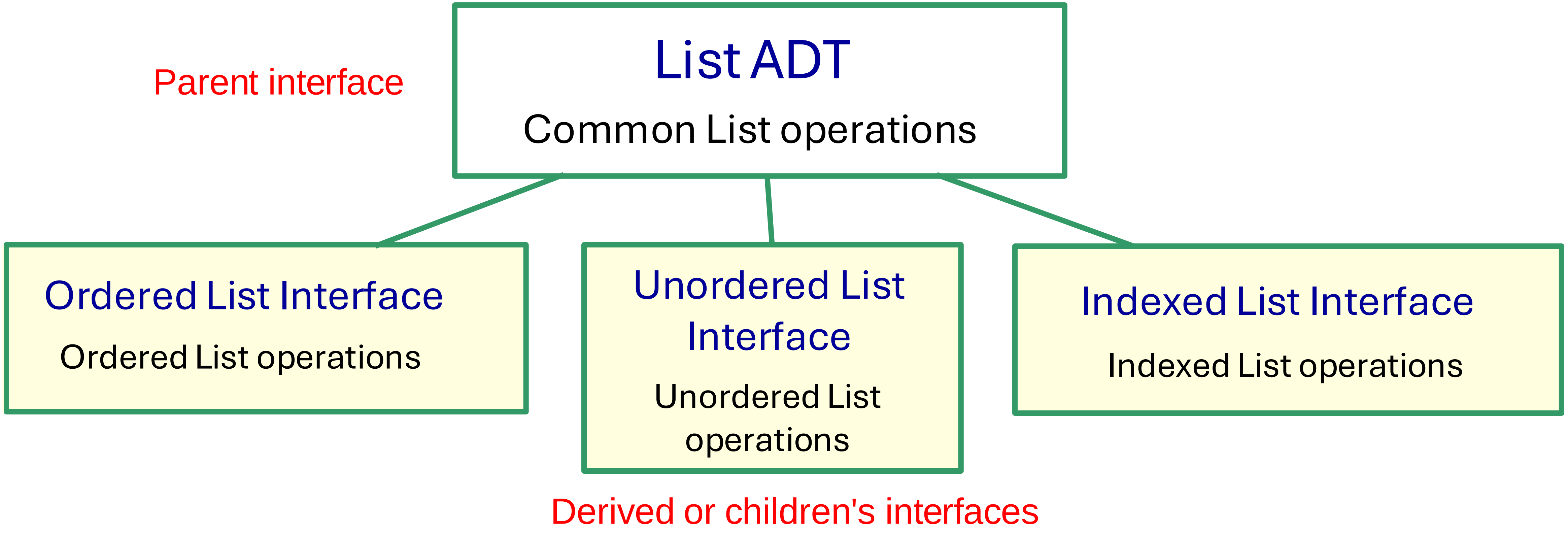
Operation	Description
addToFront(dataItem)	Adds a data item to the front of the list
addToRear(dataItem)	Adds a data item to the rear of the list
addAfter(dataItem)	Adds a data item after a particular dataItem already in the list

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

OrderedListADT Interface

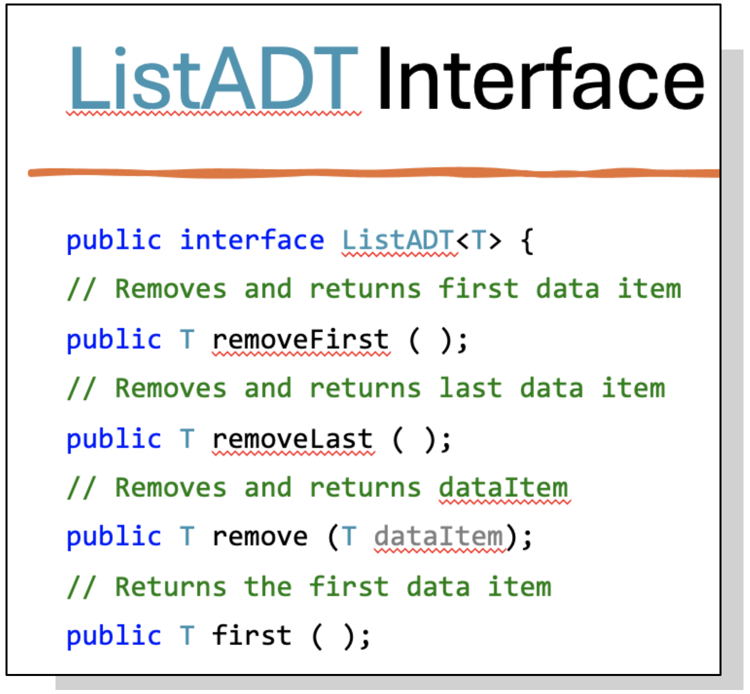
```
public interface OrderedListADT<T> extends
ListADT<T> {
// Adds dataItem to this list at the
//correct location to keep
// the list is sorted
public void add(T dataItem);
}
```

UnorderedListADT Interface

```
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 Interface

```
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);
}
```



ListADT Interface

```
public interface ListADT<T> {
// Removes and returns first data item
public T removeFirst ( );
// Removes and returns last data item
public T removeLast ( );
// Removes and returns dataItem
public T remove (T dataItem);
// Returns the first data item
public T first ( );
```

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

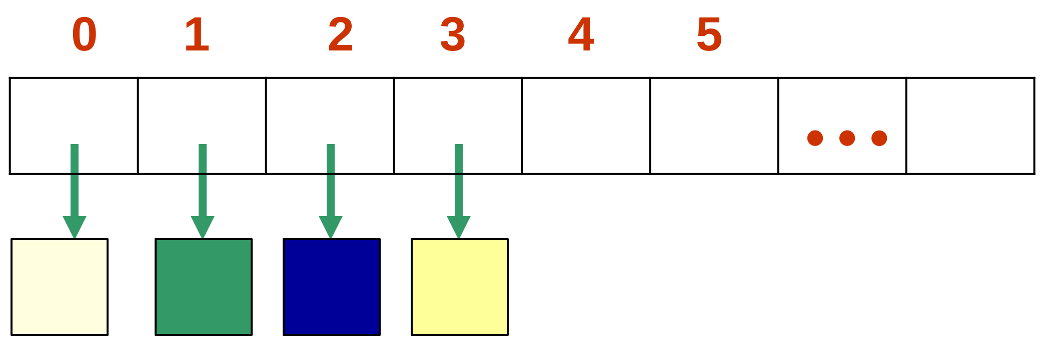
- We will study several implementations of the Ordered List ADT.
- We leave it as an exercise for you to implement the List ADT, the Unordered List ADT, and the Indexed List ADT.

Implementations of the List ADT

Array Implementation of a List

- We store the data items in an array
- Fix the first data item of the list at index 0
- Do we need to shift the data when

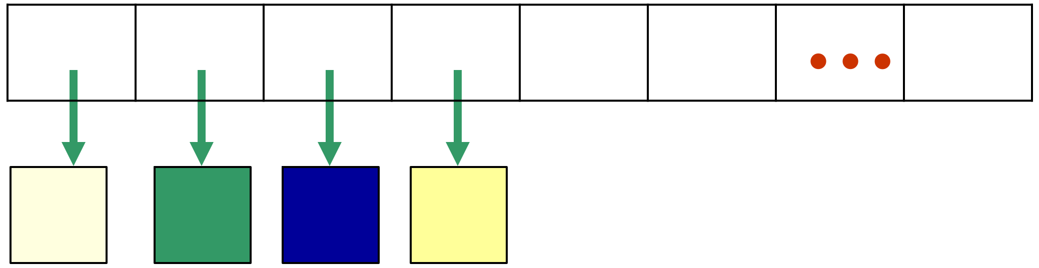
a new data item is added
  - at the front?
  - somewhere in the middle?
  - at the end?



- When adding a new item, it's important to consider if shifting data is necessary:
  - At the front: Adding an item here would require shifting all existing elements one position forward.
  - In the middle: Inserting here would involve moving items from the insertion point onward.
  - At the end: Adding at the end typically does not require shifting unless resizing is needed.

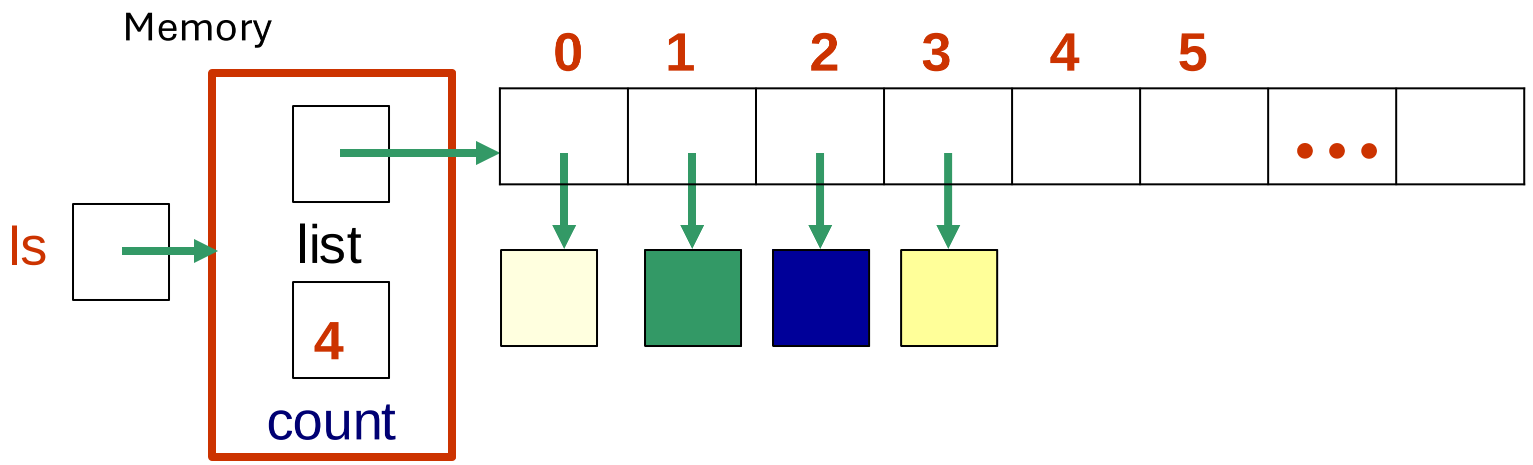
Array Implementation of a List

- 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 correctly so the resulting list is still ordered.
- 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

```java
public interface Comparable<T> {
/**
* Compares the current object with the specified object to determine their order.
* Returns a negative integer if this object is less than the specified object,
* zero if they are equal, or a positive integer if this object is greater than
* the specified object.
*
* @param otherObject the object to be compared
* @return 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 the type of otherObject prevents it from
* being compared to this object
*/
int compareTo(T otherObject) throws NullPointerException, ClassCastException;
}
```

Implement the **Comparable** interface

- In Java, several classes in the standard library implement the Comparable interface, allowing their objects to be naturally ordered.
- Some common classes implement **Comparable**: Byte, Short, Integer, Long, Float, Double, Character, and String (Wrapper Classes for Primitive Types).

Custom Classes Implementing **Comparable**

```
public class Person implements Comparable<Person> {
private String name;
private int age;

@Override
public int compareTo(Person other) {
   // Orders by age
   return Integer.compare(this.age, other.age); }
 }
```

Handling Generics with **Comparable**

- Since the ***compiler*** doesn't know if a generic type T implements the *compareTo* method, we cast one of the variables to **Comparable** to enable comparison.
- In this example, only *var1* is cast to Comparable<T> so we can use its compareTo method to compare *var1* with *var2*.

```
public class ClassA<T> {
/**
* Checks if two objects of type T are equal in terms of ordering.
* Assumes T might implement Comparable and casts one of the objects to
Comparable<T>.
* @param var1 the first object to compare
* @param var2 the second object to compare
*/
public void check(T var1, T var2) {
// Cast var1 to Comparable<T> to use compareTo, assuming T implements
Comparable
Comparable<T> tmp = (Comparable<T>) var1;
// Compare var1 and var2
if (tmp.compareTo(var2) == 0) {
System.out.println("var1 is equal to var2 in terms of order.");
} else {
System.out.println("var1 is not equal to var2 in terms of order.");}}}
```

This lets us compare var1 and var2 if their class implements Comparable.

Runtime Error Risks

- 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**.
- This is because the compiler does not enforce that T implements **Comparable**—we only assume it does by using this cast.
- If T does not implement **Comparable**, the cast will fail, resulting in a **runtime error**.

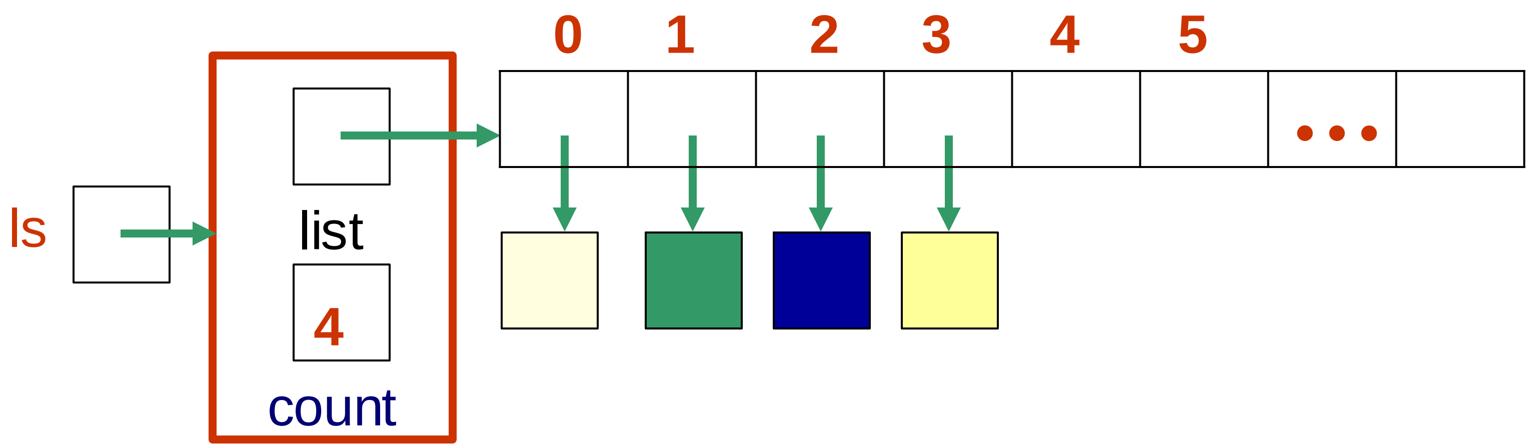
```
ClassA<String> v1 = new ClassA<>();
  v1.check("hello", "hi");
```

This code will run without error because String implements the **Comparable** interface. ✓

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

This code will throw a ClassCastException because arrays do not implement the **Comparable** interface ✗

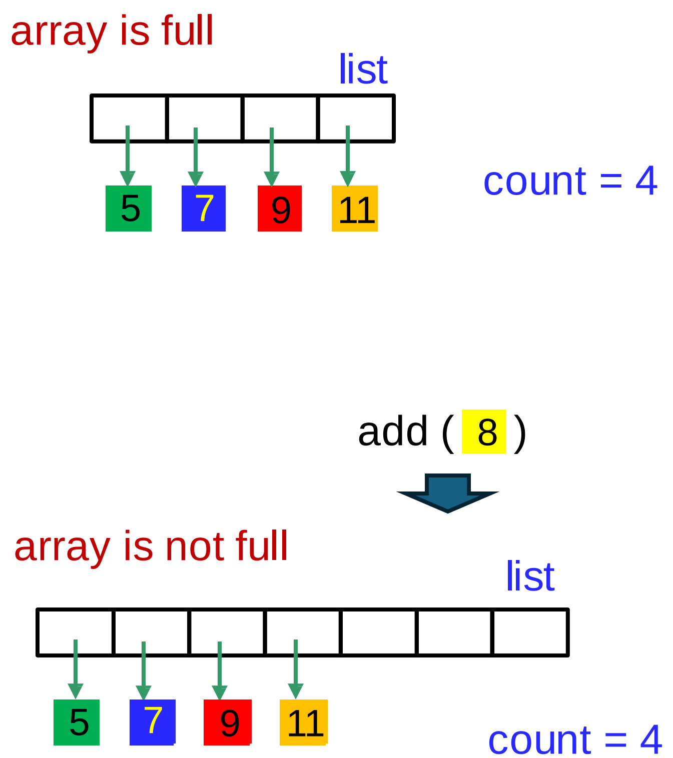
Array Implementation of the Ordered List ADT



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)
Input: A new data item to add
Output: None, but the data item is inserted into the list,
 maintaining sorted order.
1. If count equals the length of list, call expandCapacity() to increase the list size.
2. Initialize i to 0.
3. Find the insertion point:
    1. While i is less than count and dataItem is greater than list[i]:
        1. Increment i by 1.
4. Shift elements to make space:
    1. If i is less than count:
        1. For each index j from count down to i + 1:
            1. Set list[j] to list[j - 1].
5. Insert dataItem at list[i].
6. Increment count by 1 to reflect the addition.
```



The **Add Operation**

```
// Adds dataItem to the list while maintaining sorted
order.
public void add(T dataItem) {
// Expand the list capacity if it's full
if (count == list.length) {expandCapacity();}
// Cast dataItem to Comparable to enable comparison
Comparable<T> temp = (Comparable<T>) dataItem;
int i = 0;
// Find the insertion point: the first element larger
than or equal to dataItem
while (i < count && temp.compareTo(list[i]) > 0) {i++;}
// Shift elements to the right to make space for dataItem
for (int j = count; j > i; j--) {list[j] = list[j - 1];}
// Insert dataItem at the found position
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.

Thank you