Lists

- A List ADT
- Types of Lists
- Using ordered lists – Tournament Maker
- Using indexed lists – Josephus Problem
- Implementing lists with arrays and links
- Lists in Java Collections API
- Analysis of library list implementations
- Reading L&C 6.1-6.6

A List ADT

- A *list* a collection of items in which the items have a position
- It is an inherently familiar concept
- We keep “to-do” lists, shop with a grocery list, and invite a list of friends to a party
- Types of Lists
  - Ordered Lists
  - Unordered Lists
  - Indexed Lists

Ordered Lists

- An ordered list is kept in order based on characteristics of the elements in the list, e.g. alphabetical order for names
- The class for objects kept in an ordered list must implement the Comparable interface
- When an element (object) is added to an ordered list, its position relative to the other elements stored in the list is determined by the `compareTo` method for their class

Unordered Lists

- An unordered list is not based on any characteristics of the elements themselves
- The class for the elements does not need to implement the Comparable interface for them to be stored in an unordered list
- They are stored in an order that is externally controlled by how and when the elements are added to the list

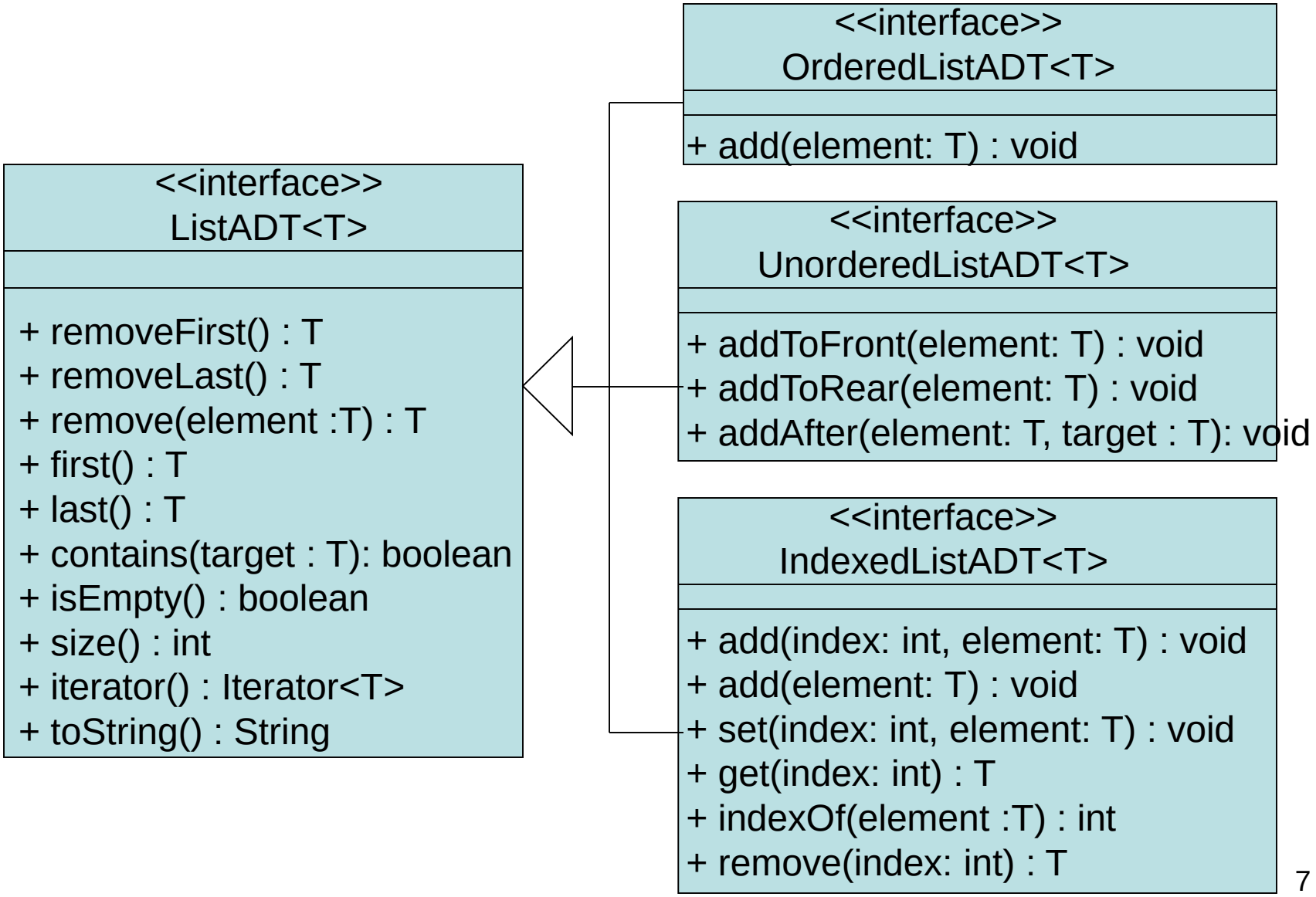
Indexed Lists

- An indexed list is an unordered list that also allows elements to be added, accessed, or removed based on a index value
- Again the order is externally controlled by how and when the elements are added and maybe by the index value used when adding
- You should be familiar with the `ArrayList` class which is a class for an Indexed list

Ordered versus Indexed Lists

- Is it meaningful for a list to be both ordered and indexed?
- No - The two concepts are not compatible
- In an ordered list, elements should be kept in the order determined by the compareTo method of the class of the elements
- In an indexed list, elements should be kept in the order indicated by the indices used when adding the elements

Text's List Interface Hierarchy



Using Ordered Lists

- In a tournament, teams play pairwise with each other in a series of games until the last two teams play in championship game
- To “seed” the tournament in the first round, the best team plays the worst team, the next best team plays the next worst team, and all teams are paired this way
- This is “fairest” because it is impossible for the best team to play the second best team in the first round – eliminating one of them

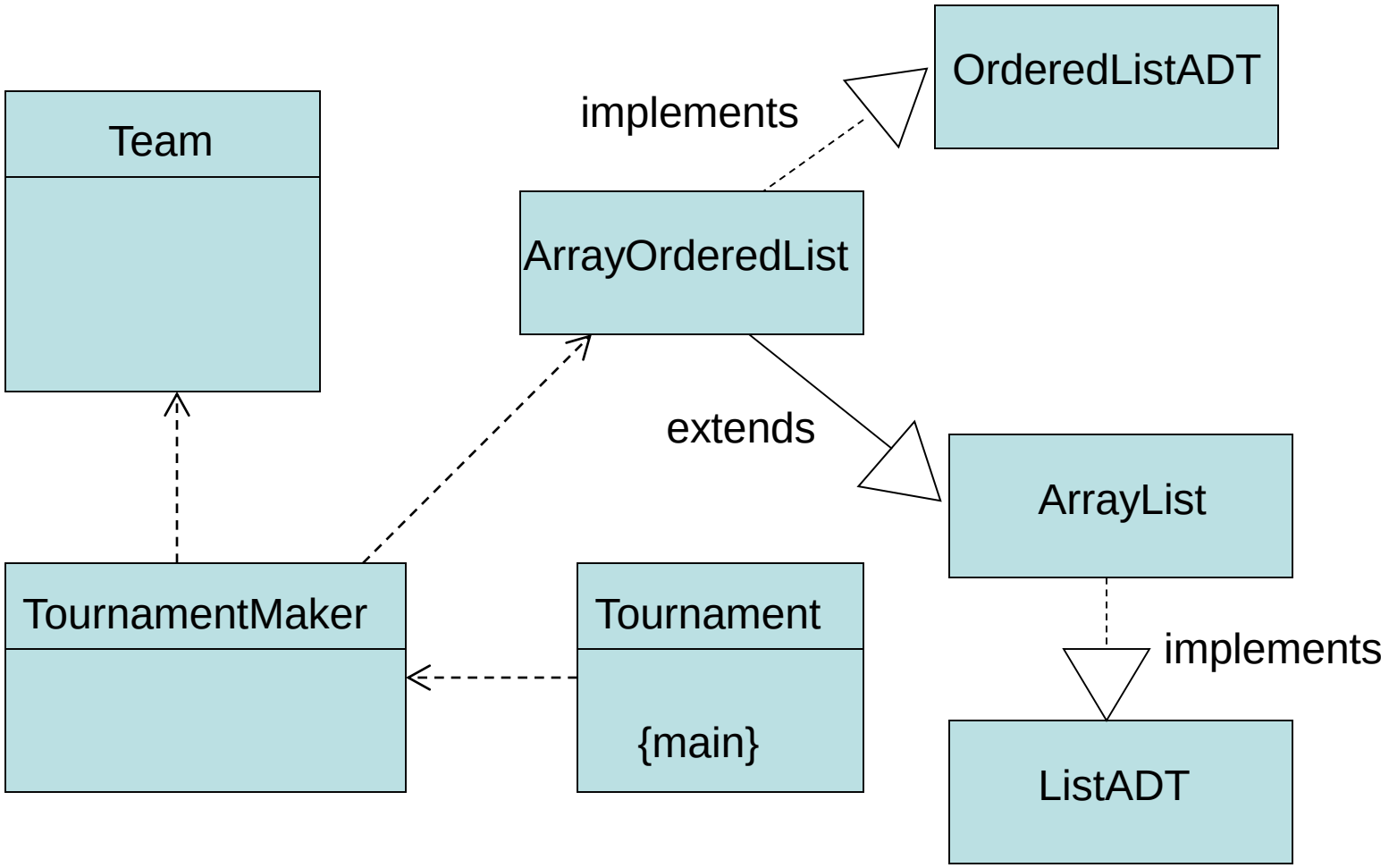
Using Ordered Lists

- Tournament Maker (for a number of teams that is a power of 2)
- Algorithm is based on adding the teams to an ordered list where the team compareTo method uses the number of wins and that determines each team's order in the list
- Algorithm removes first and last team and pairs them for a game until all teams have been paired for a game (i.e, list is empty)

Using Ordered Lists

- For subsequent rounds, the games can be paired based on their game number
- Since the game numbers were determined based on the number of wins of the teams this algorithm will build out the subsequent rounds of play correctly to make it most likely that the best and second best teams make it to the championship game

UML for Tournament Maker



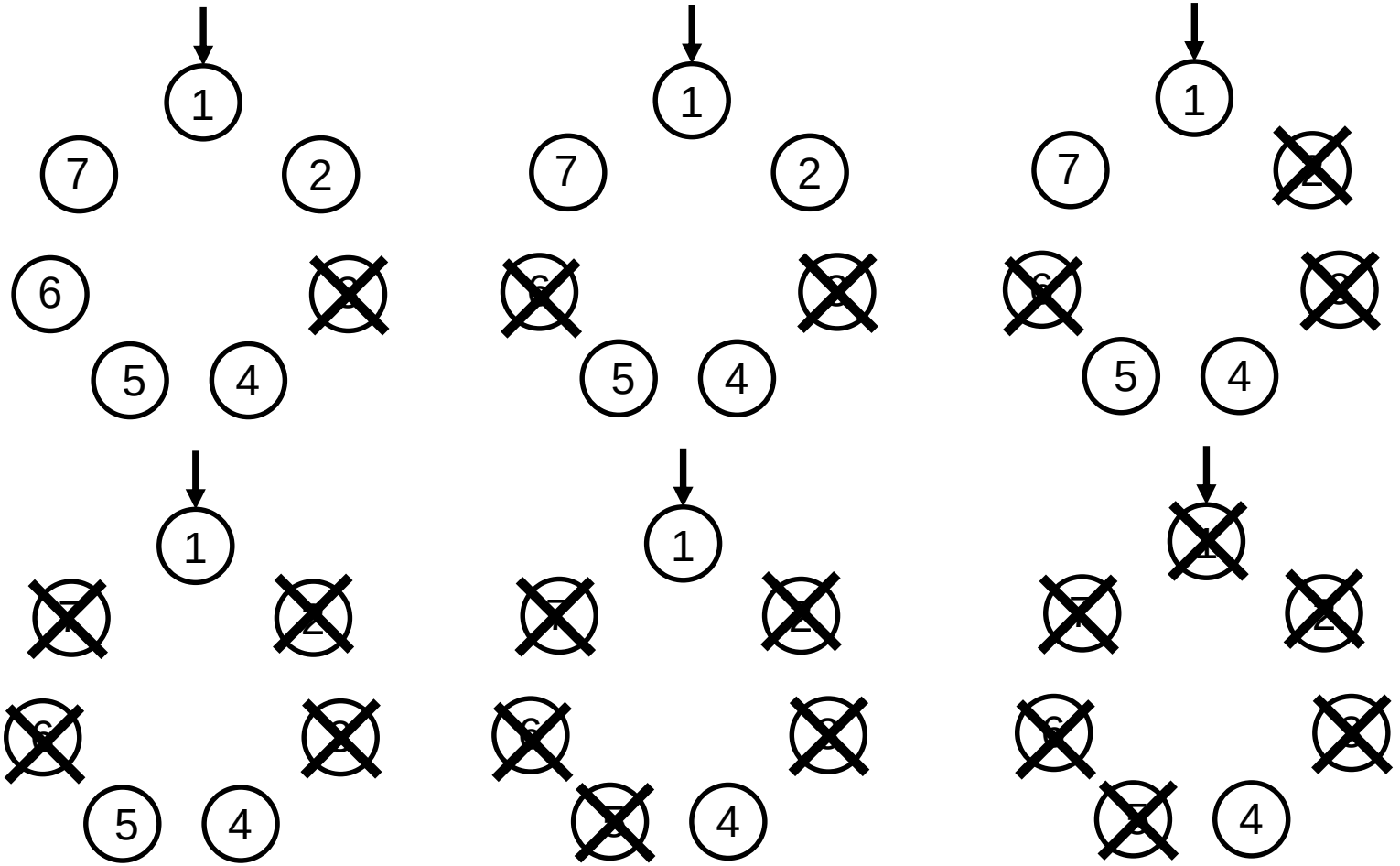
Note: Corrections to L&C Figure 6.10

Using Indexed Lists

- The Josephus Problem described in the text is a classic computer science problem
- Going around in a circle of n players, we successively eliminate every mth player until we reach the last player who wins
- A solution that is O(n) can be based on an indexed list

Using Indexed Lists

- The Josephus Problem: n = 7, m = 3



Using Indexed Lists

- Josephus Solution based on an ArrayList

```
int numPeople = 7, gap = 3;
int counter = gap - 1;
ArrayList<Integer> list=new
  ArrayList<Integer>();  // init to 1, 2, ...
...
while (!list.isEmpty())
{
   System.out.println(list.remove(counter));
   numPeople = numPeople - 1;
   if (numPeople > 0)
     counter = (counter + gap - 1) % numPeople;
}
```

Josephus Problem

- A recursive solution to the Josephus problem that provides the survivor's number for gap = 2 only

```
public static int J(int n)     // n is number of soldiers
  {
    if (n == 1)
      return 1;                // base case
    else if (n % 2 == 0)
      return 2 * J(n/2) - 1;   // n is even
    else
      return 2 * J(n/2) + 1;   // n is odd
  }
```

- The explanation of how it works is in the link:

  http://www.cut-the-knot.org/recurrence/r_solution.shtml

Josephus Problem

- Sample run for 1 to 16 soldiers with the gap = 2

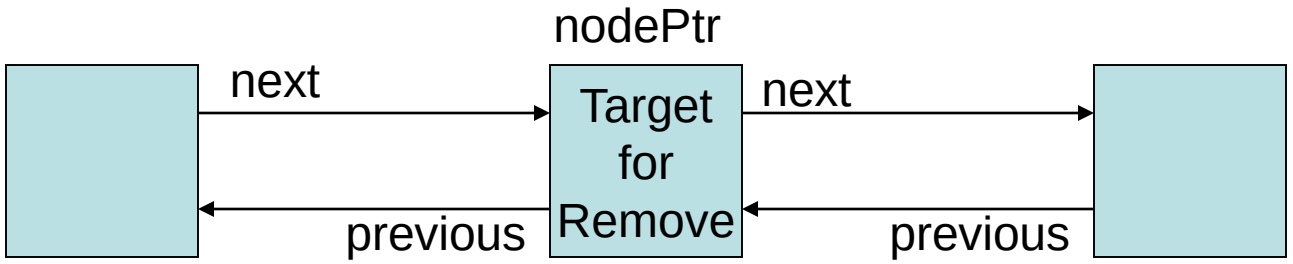
```
> run Josephus
 1.  1
 2.  1
 3.  3
 4.  1
 5.  3
 6.  5
 7.  7
 8.  1
 9.  3
10.  5
11.  7
12.  9
13. 11
14. 13
15. 15
16.  1
>
```

Note the interesting pattern in the number of the survivor as the number of soldiers goes up. The weblink explains the reason.

Implementing Lists

- Please study L&C sections 6.4 and 6.5
- Much of the method code is similar to code used for previously studied data structures
- Note the suggested use of a doubly linked list to simplify code for the remove method

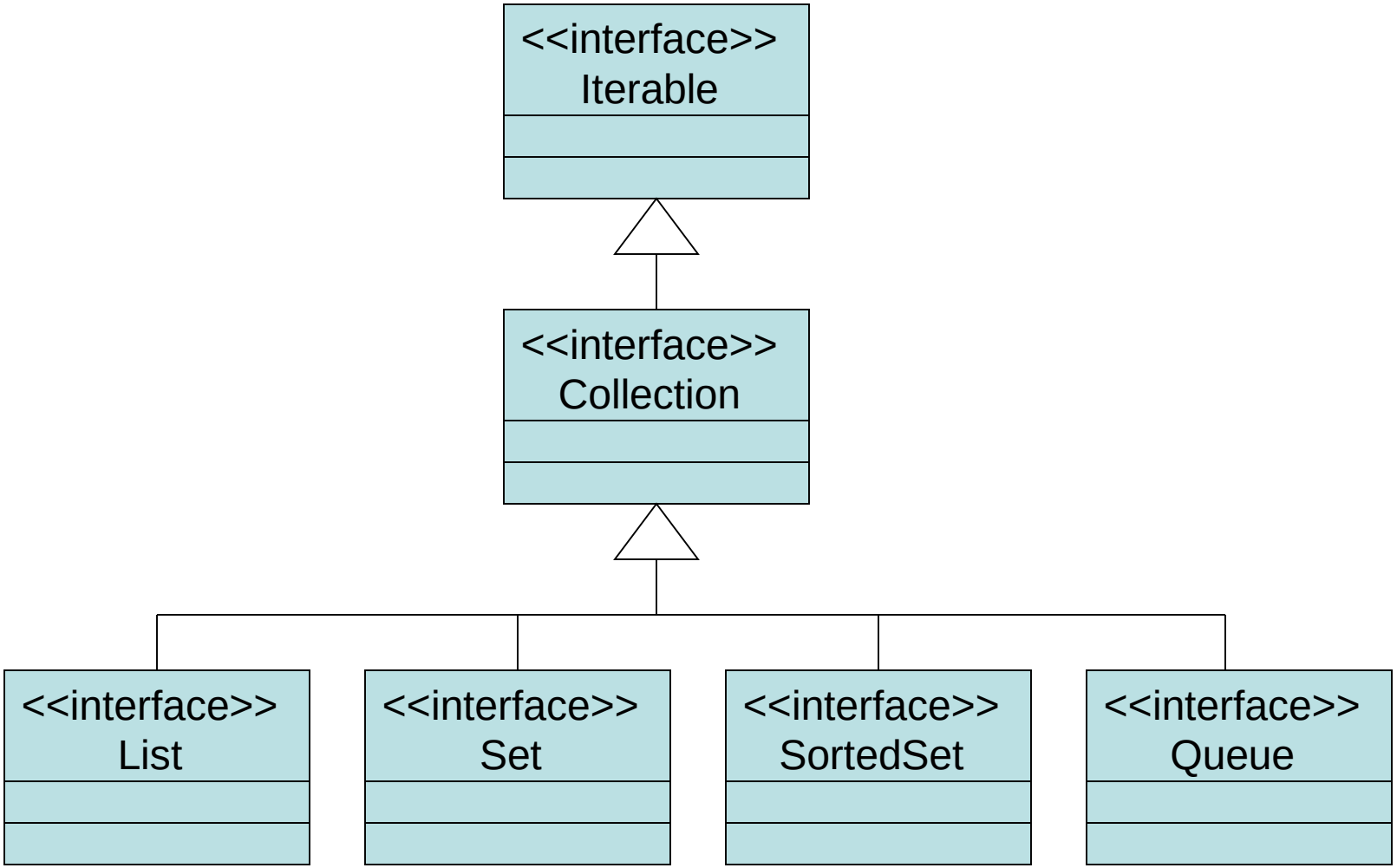
```
nodePtr.getNext().setPrevious(nodePtr.getPrevious());
nodePtr.getPrevious().setNext(nodePtr.getNext());
```



Lists in Java Collections API

- These are the three primary list classes:
  - `java.util.ArrayList` – a list based on an underlying array
  - `java.util.LinkedList` – a list based on an underlying linked structure
  - `java.util.Vector` – a list that is more often used for backward compatibility today and is no longer “in vogue” for new applications (although some of its subclasses such as java.util.Stack are still of interest)

Review: Collection Interface Hierarchy



Analysis of List Implementations

- Much of the time analysis is similar to the analysis of methods for previous structures
- Some methods that are O(1) for the array implementation are O(n) for linked version
  - Set or get element at the specified index
- Some methods that are O(1) for the linked implementation are O(n) for array version
  - Add to front

Analysis: ArrayList vs LinkedList

- The ArrayList is more efficient
  - If items are often added to the end of the list
  - If items are often retrieved by index value
- The ArrayList is less efficient:
  - If items are often added in the middle (all of the following items must be moved out of the way)
- The LinkedList is more efficient:
  - If items are often added to the start of the list
- The LinkedList is less efficient:
  - If items are often retrieved by index value