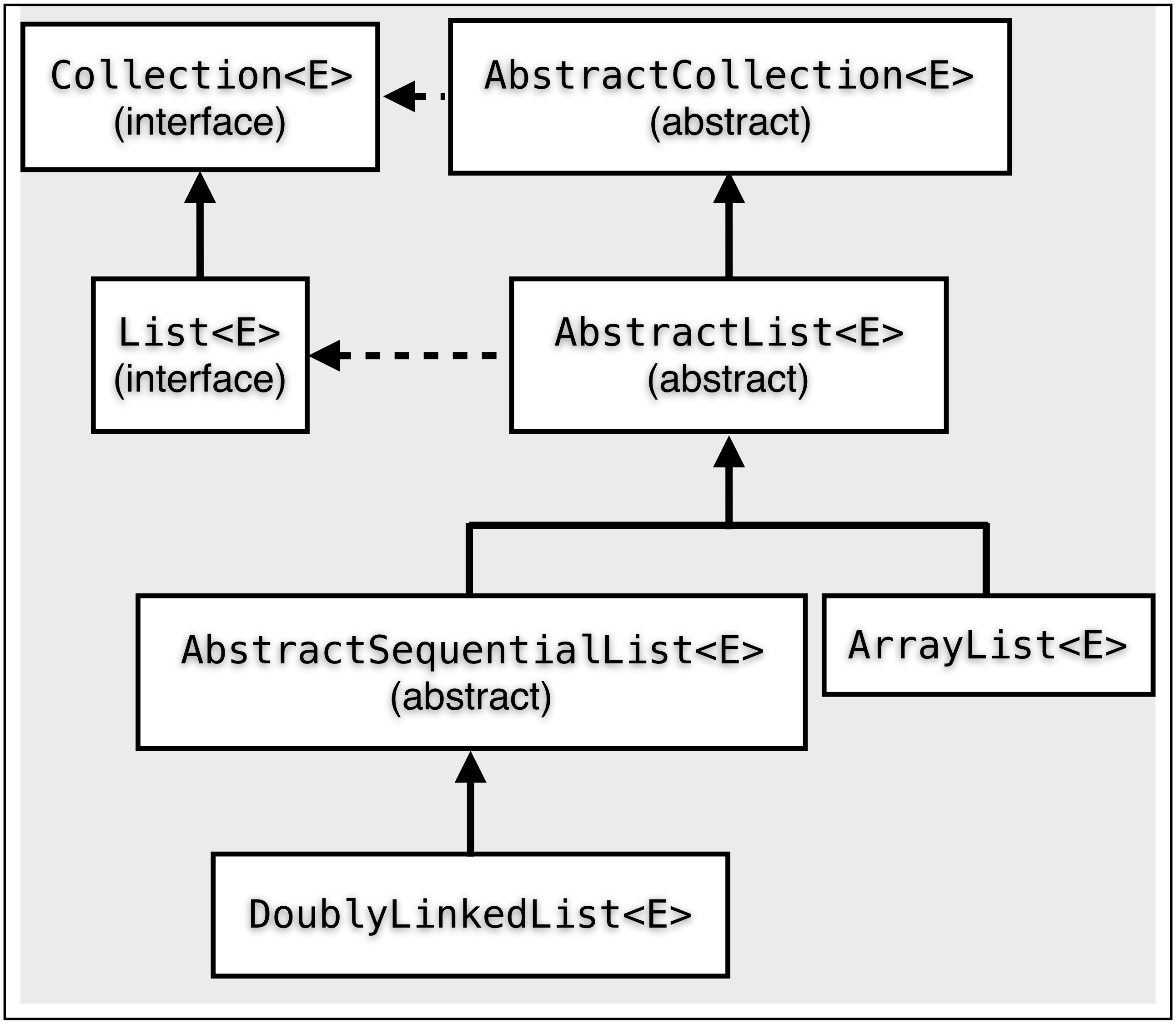
CS 228: Introduction to Data Structures
Lecture 21

The `AbstractSequentialList` Class

Implementing the full `List` interface would be overwhelming. Instead, we build upon the existing abstract class `AbstractSequentialList`, where all methods other than `size()` and `listIterator(pos)` are optional. The class hierarchy is on the next page.

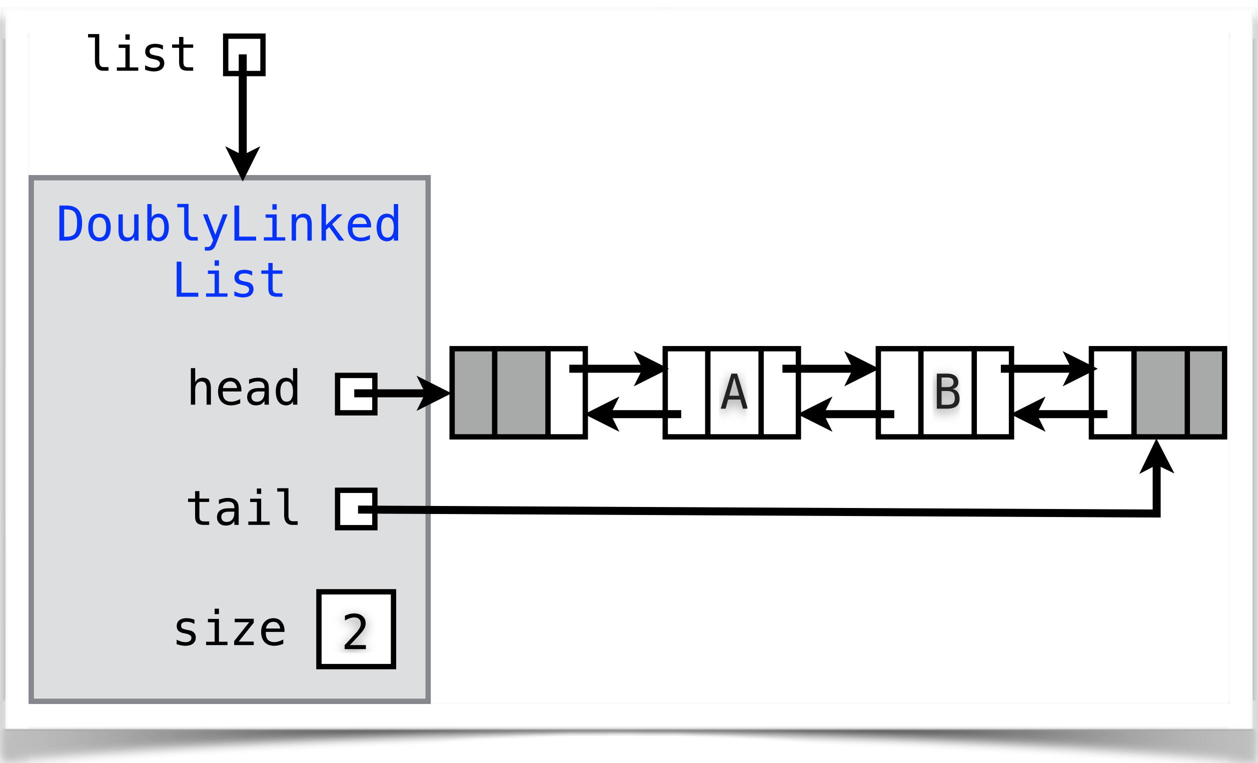
Note that `AbstractSequentialList` assumes that the the list is represented via a sequential access data structure. Contrast this with `ArrayList`, which assumes an array as the backing store.


Collection<E>
(interface)
AbstractCollection<E>
(abstract)
List<E>
(interface)
AbstractList<E>
(abstract)
AbstractSequentialList<E>
(abstract)
ArrayList<E>
DoublyLinkedList<E>

A Doubly-Linked `List` Implementation

Singly-linked lists are a poor choice for implementing the `List` interface, because it isn't easy to iterate backwards on them. Therefore, we focus instead on a doubly-linked list implementation of the `List` interface; it is called `DoublyLinkedList`.

Our lists will have ***dummy nodes*** at head and `tail`. Here is a `DoublyLinkedList` object, called `list`, that stores two strings.



The class definition begins like this:

```java
public class DoublyLinkedList<E>
extends AbstractSequentialList<E> {

   private Node head;

   private Node tail;

   private int size;
```

We use the same `Node` class as in `DoublyLinkedCollection`. The constructor is:

```java
  public DoublyLinkedList()
  {
    head = new Node(null);
    tail = new Node(null);
    head.next = tail;
    tail.previous = head;
    size = 0;
  }
```

We will need a few helper methods. The first of them splices a given node at a specified position in a list.

`void link(Node current, Node newNode)`**:** Inserts newNode into this list after `current` without updating `size`.

Precondition: `current != null`, `newNode != null`

```
private void
link(Node current, Node newNode)
{
  newNode.previous = current;
  newNode.next = current.next;
  current.next.previous = newNode;
  current.next = newNode;
}
```

The next method removes a specified node from the list.

`void unlink(Node current)`: Removes `current` from the list without updating `size`.

Precondition: `current != null`

```
private void unlink(Node current)
{
  current.previous.next = current.next;
  current.next.previous
    = current.previous;
}
```

The last helper methods returns a reference to a node at a specific position in the list.

`findNodeByIndex(int pos)`: Returns the Node whose index is `pos`, which will be head if `pos == −1` and `tail` if `pos == size`.

Precondition: `size >= pos >= −1`.

```java
private Node findNodeByIndex(int pos)
{
  if (pos == -1) return head;
  if (pos == size) return tail;

  Node current = head.next;
  int count = 0;
  while (count < pos)
  {
    current = current.next;
    ++count;
  }
  return current;
}
```

Time complexities of the helper methods. It is not hard to see that `link()` and `unlink()` are O(1)-time operations, while `findNodeByIndex()` takes O(n) time in the worst case, where n is the length of the list.

`add()`. We have two options when adding an element. The first just adds a new item at the end of the list.

```
public boolean add(E item)
{
  Node temp = new Node(item);
  link(tail.previous, temp);
  ++size;
  return true;
}
```

The second adds the item at a specific position.

```
public void add(int pos, E item)
{
  if (pos < 0 || pos > size)
    throw new IndexOutOfBoundsException
              ("" + pos);

  Node temp = new Node(item);
  Node predecessor =
      findNodeByIndex(pos - 1);
  link (predecessor, temp);
  ++size;
}
```

Time complexities of the add() methods. The `add(item)` method takes O(1) time, since we have direct access to the end of the list. On the other hand, `add(pos,item)` takes O(n) time in the worst case — where, as usual n is the length of the list — since we have to traverse the list (using `findNodeByIndex`) to locate the insertion point.

Note. The code posted on Blackboard also has implementations of `get()` and `contains()` — study that code carefully.

List Iterators

As usual, we implement iterators with an inner class, here called `DoublyLinkedIterator`. We give users two options.

```
public ListIterator<E> listIterator()
{
  return new DoublyLinkedIterator();
}

public ListIterator<E>
  listIterator(int pos)
{
  return new DoublyLinkedIterator(pos);
}
```

The class declaration begins like this:

```java
private class DoublyLinkedIterator
implements ListIterator<E>
{
  // direction for remove() and set()
  private static final int BEHIND = -1;
  private static final int AHEAD = 1;
  private static final int NONE = 0;

  private Node cursor;
  private int index;
  private int direction;
```

The following class invariants express the meanings of the instance variables.

Class Invariants

1. The logical cursor position is always between `cursor.previous` and `cursor`.

2. After a call to `next()`, `cursor.previous` refers to the node just returned

3. After a call to `previous()`, `cursor` refers to the node just returned

4. `index` is always the logical index of node pointed to by `cursor`.

5. direction is `BEHIND` if last operation was `next()`, `AHEAD` if last operation was `previous()`, `NONE` otherwise.

We need to provide two constructors.

```
    public DoublyLinkedIterator(int pos)
    {
      if (pos < 0 || pos > size)
        throw new
          IndexOutOfBoundsException
            ("" + pos);

      cursor = findNodeByIndex(pos);
      index = pos;
      direction = NONE;
    }

    public DoublyLinkedIterator()
    {
      this(0);
    }
```

Next time, we will see how to implement the `ListIterator` methods.