Design Patterns & Collections

OOP Lecture 6

(2025)

Design patterns

The point of design patterns

- Proven solutions to common design problems
- Why?
  - Crafted by experienced object-oriented practitioners, design patterns make your designs more flexible, more resilient to change, and easier to maintain.
- Related questions:
  - What are common features in good designs that are not in poor designs?
  - What are common issues in poor designs that are not in good designs?
- In the upcoming 6 lectures, we will consistently link one or more design patterns to the topic of each week.
- See also ItJPaDS 11 ed. Chapter 13.10: Class-Design Guidelines

Pattern description format

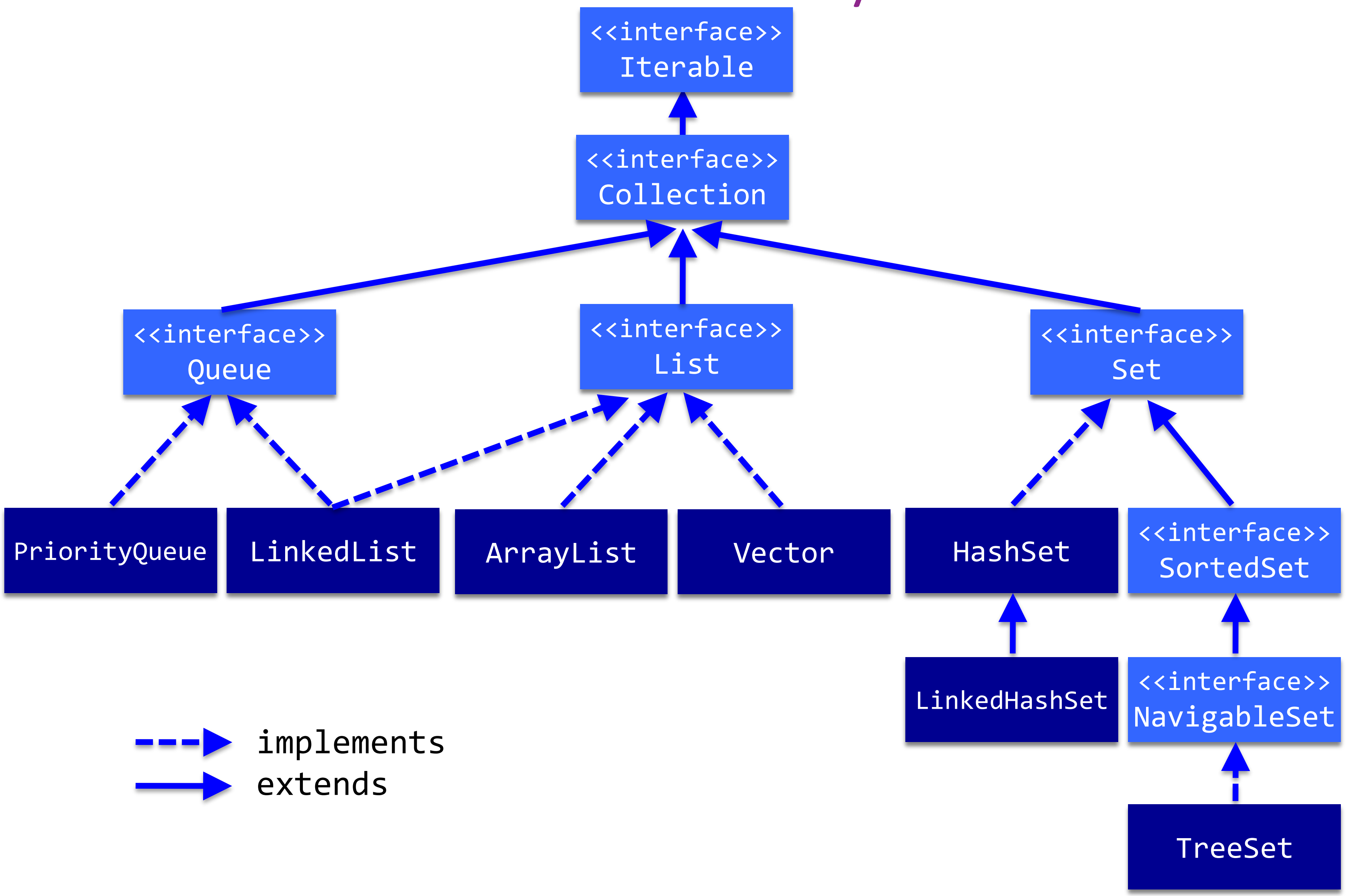
- Each pattern has:
  1. short name
  2. description of the context/problem
  3. prescription for the solution
     - often not programming language specific,
       same pattern works for Java, C#, C++, JavaScript, Python, ..

Collections (topic) & iterators (design pattern)

The interface **`Collection`**

- We have several (concrete) containers in Java
  - `StringBuffer`, `ArrayList`, `LinkedList`, `Vector`, `HashSet`
- Many similar operations on these containers
  - `isEmpty`, `contains`, `add`, remove, `size`
- The interface **`Collection`** yields a uniform way to handle these kind of operations
- Warning: there is also a (utility) class **`Collections`**
  - `Collections` is similar to `Arrays`: set of basic operations provided as static methods
  - don't confuse them

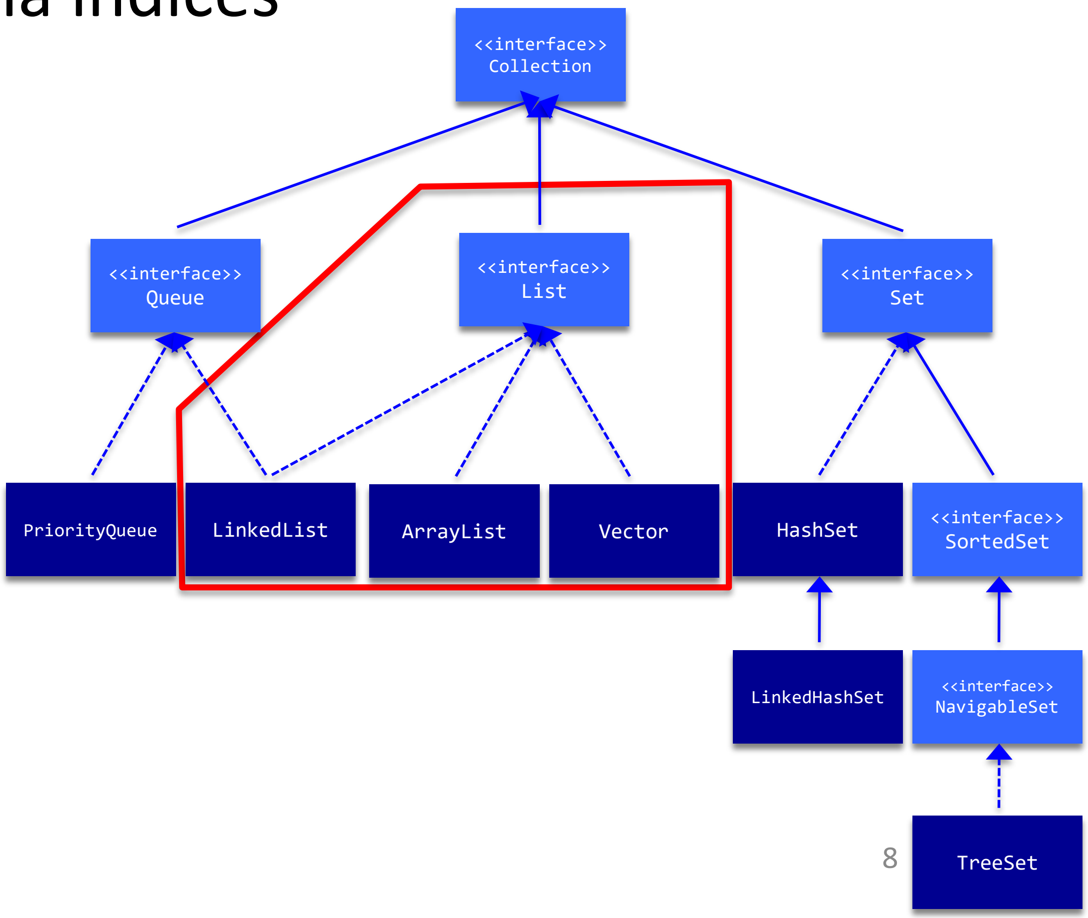
Collection interface hierarchy



Lists

interface **`List`** is an extension of **`Collection`**

- List adds methods to manipulate elements via indices
  - `void` **`add`**`(int index, E element)`
  - `E` **`get`**`(int index)`
  - `E` **`remove`**`(int index)`
  - `E` **`set`**`(int index, E element)`



Collection relationships

- **`List`**
  - elements are ordered (insertion order is maintained)
  - elements can occur more than once
- **`Set`**
  - does not contain duplicates
  - can (sometimes) be sorted !
  - elements are *not* ordered (insertion order is *not* maintained)

The class **`Collections`**

Do not confuse it with the interface **`Collection`**

- contains algorithms for collections
- like **`Arrays`** for arrays

Implemented algorithms:

`sort`, **`binarySearch`**, **`reverse`**, **`shuffle`**,
`fill`, **`copy`**, **`min`**, **`max`**, **`addAll`**,
`frequency`, **`disjoint`**

Different List implementations

- `ArrayList` and `LinkedList` both implement the List interface
- Hence they provide the same operations
- The efficiency of operations differs
- This is the reason to have two implementations

ArrayList

warning:

the **`MyArrayList`** class is only to demonstrate differences between various implementations of the List interface

there is a better reusable solution in Java

never ever implement a `ArrayList` in your own program

unless you have a very good reason for it

MyArrayList

implement the `List` interface

store elements in an array

+ accessing an element is fast $O(1)$
- inserting/deleting elements is expensive $O(N)$

we cannot predict the size of the list

- there is no upper bound
- start with a small array
- allocate a bigger array when the current array is full & copy all elements: $O(N)$
  this is done once every N additions: amortized $O(1)$

`MyArrayList` is quite similar to the standard `ArrayList`

- some simplifications (not all methods are implemented)

MyArrayList: fields & constructor

```
public class MyArrayList<E> extends AbstractList<E> {

  private int size = 0;       // current number of elements in list
  private Object[] data;      // array containing the elements

  public MyArrayList(int capacity) {
    data = new Object[capacity];
  }
  ...
}
```

skeletal implementation of the List interface

MyArrayList: size(), get(index), add(element)

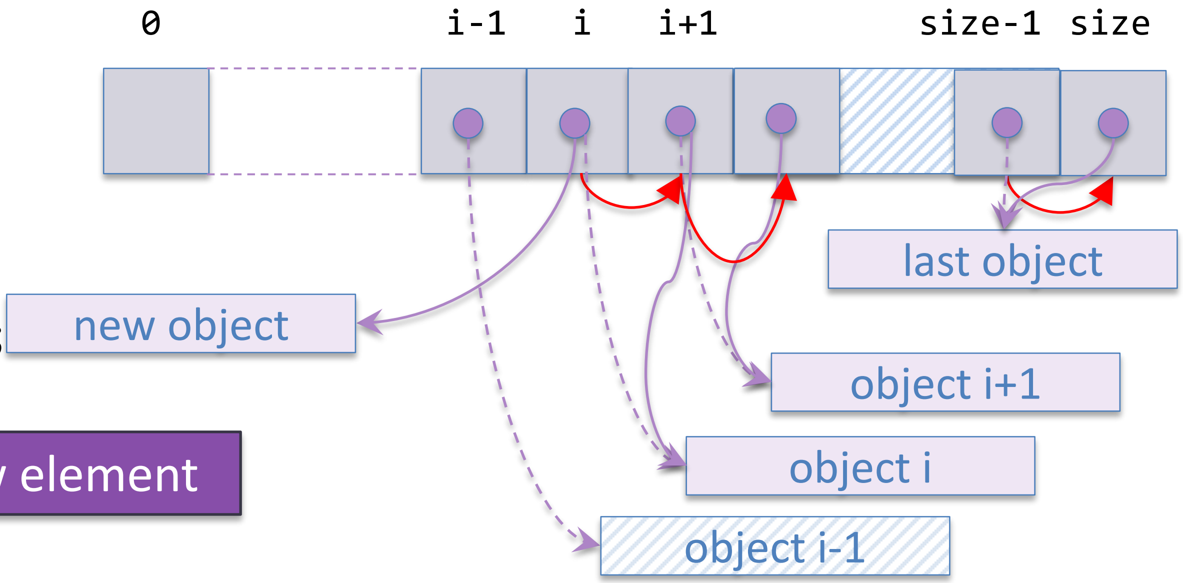
```
@Override
public int size() {
  return size;
}

@Override
public E get(int index) {
  checkBound(index);
  return (E) data[index];
}

@Override
public boolean add(E e) {
  ensureCapacity(size + 1);
  data[size++] = e;
  return true;
}
```

helper method (next slide)

type cast

helper method (next slide)

MyArrayList: add(index, element), ensureCapacity(size)




```java
@Override
public void add(int i, E e) {
  checkBoundInclusive(i);
  ensureCapacity(size + 1);
  System.arraycopy(data, i, data, i + 1, size - i);
  data[i] = e;
  size++
}
```

makes room for the new element

```java
private void ensureCapacity(int cap) {
  if (cap > data.length) {
    var es = new Object[Math.max(data.length * 2, cap)];
    System.arraycopy(data, 0, es, 0, size);
    data = es;
  }
}
```

new array will be twice as big

MyArrayList: remove(index), checkBound(index)

```
@Override
public E remove(int i) {
  checkBound(i);
  E r = (E) data[i];
  size--;
  System.arraycopy(data, i + 1, data, i, size - i);
  data[size] = null;
  return r;
}

private void checkBound(int i) {
  if (i < 0 || i >= size) {
    throw new IndexOutOfBoundsException("Index: " + i + ", size: " + size);
  }
}
```

`MyArrayList`: `Iterator<E> iterator()`

- *Iterator* lets you traverse elements of a collection without exposing its underlying representation
- An iterator offers a standard way to scan and handle all elements of a collection
  - `Iterator` is an interface
  - Every collection provides a *factory method* called `iterator` that creates an `Iterator` object.
  - the class implementing this interface mostly remains hidden
- The Iterator keeps track of the current element in a collection
- There are methods to advance to the next element and to delete the current element from a collection

Iterator interfaces

E: generic type of the elements

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

is there a next object?

yield next object; advance iterator one position

remove last returned object

optional operation, can throw a `NotImplementedException`

```
interface Iterable<E> {
  Iterator<E> iterator()
}
```

(factory) method for creating an iterator over elements of type E

`interface Collection<E> extends Iterable<E>`

Iterator example

```java
public class MyMap<K,V> {
  private Collection<Pair<K,V>> map;

  public MyMapCol() {
    map = new ArrayList<>();
  }

  public void put(K key, V value) {
    map.add(new Pair<>(key,value));
  }

  public boolean replace (K key, V value) {
    boolean contains = false;
    Iterator<Pair<K,V>> mapIt = map.iterator();
    while ( mapIt.hasNext() && ! contains ) {
      Pair<K,V> p = mapIt.next();
      if ( p.key().equals(key) ) {
        mapIt.remove();
        contains = true;
      }
    }
    put(key, value);
    return contains;
  }
}
```

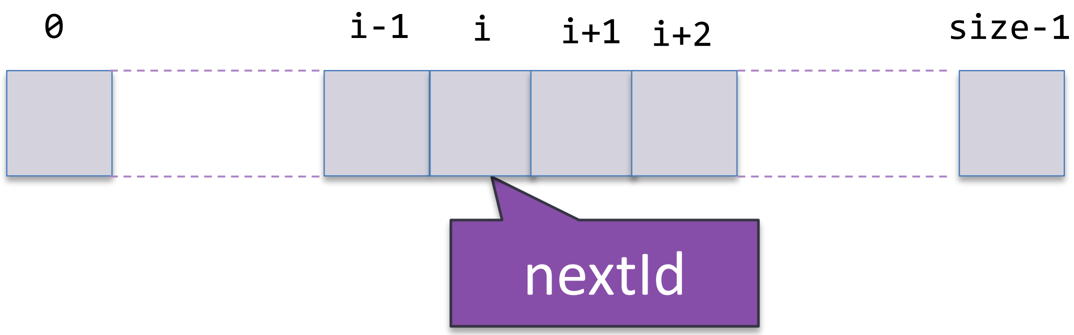
creates an iterator for map

is there another element?

get next element

remove element returned by next from map

MyArrayList: Iterator<E> iterator()



- nextId: index of next element

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

MyArrayList: Iterator<E> iterator()

```java
@Override
public Iterator<E> iterator(){
  return new MyArrayListIterator<>();
}

private class MyArrayListIterator implements Iterator<E>{
  private int nextId = 0;

  @Override
  public boolean hasNext() {
    return nextId < size;
  }

  @Override
  public E next() {
    if (nextId < size) {
      return (E) data[nextId++];
    } else {
      throw new NoSuchElementException();
    }
  }
}
```

local/nested/inner class (more next week)

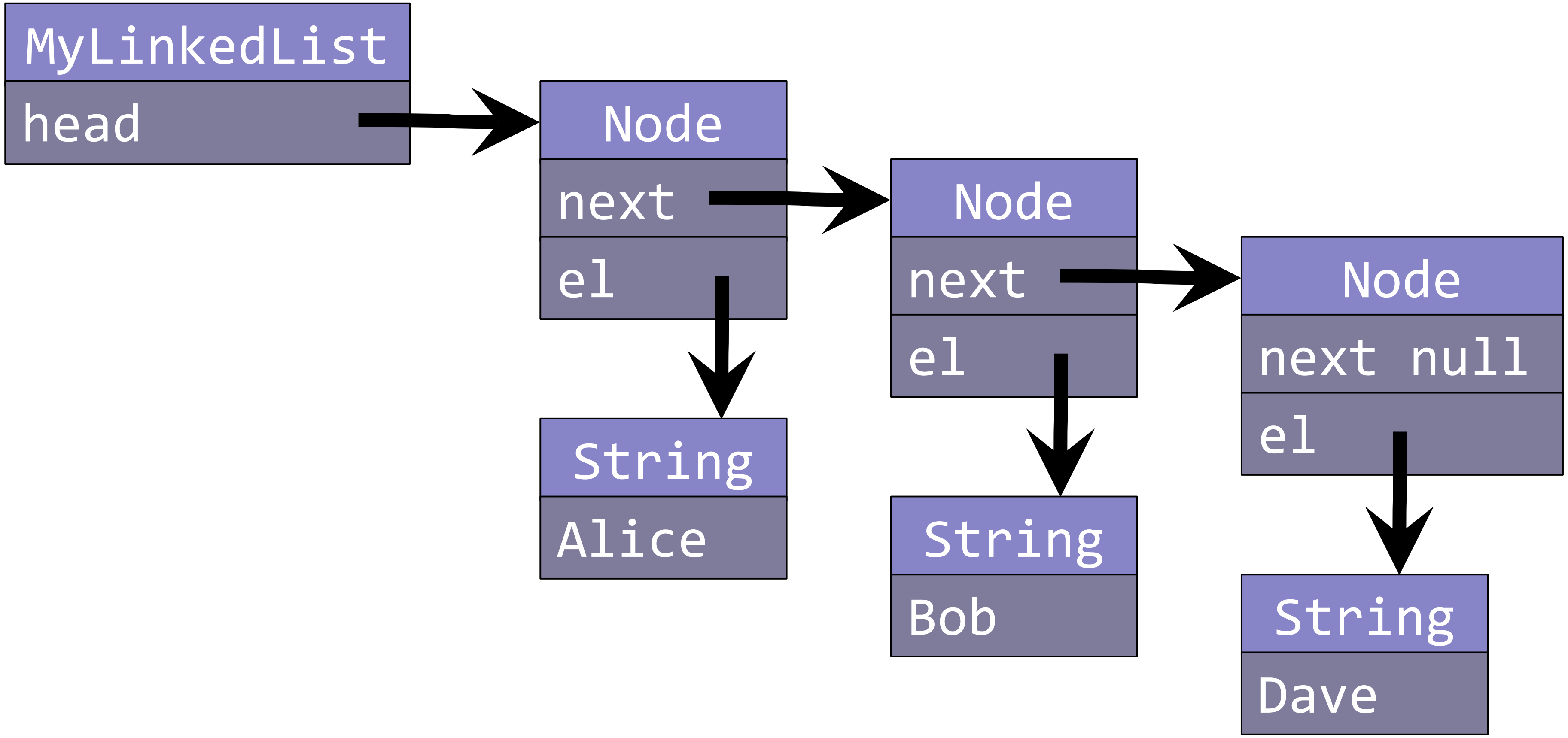
`MyArrayList`: evaluation

- Java **`ArrayList`** is quite similar to **`MyArrayList`**
  - Simple and works properly in many situations
- Unless:
  - use `add(i,e)` a lot (with `i < size`)
  - remove a lot of elements
  - these are all $O(N)$ work
- How to improve the $O(N)$ operations?
- Use a *linked data structure* (recursive data structure)

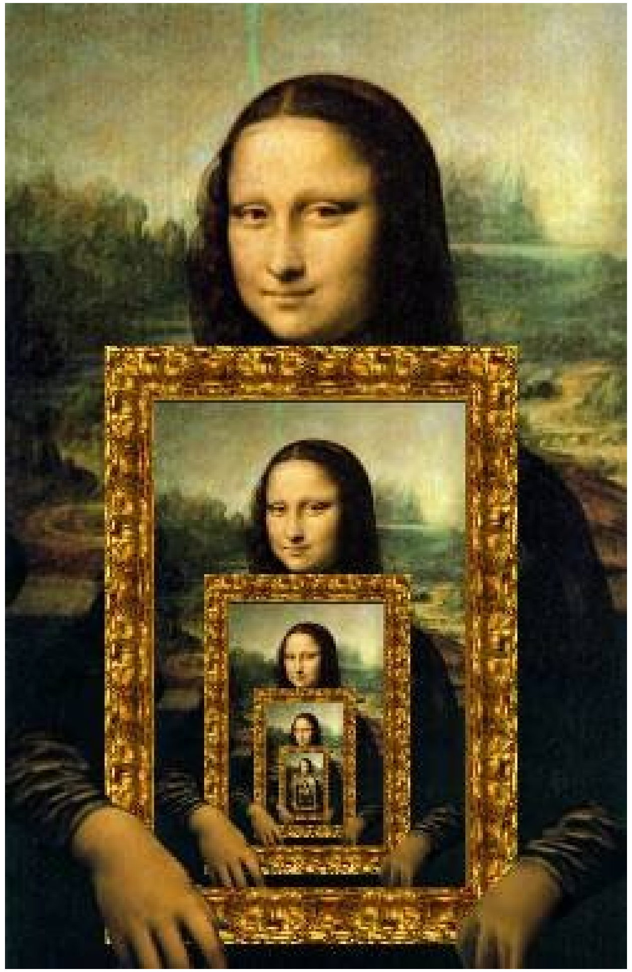
LinkedList

Linked List

basic idea:

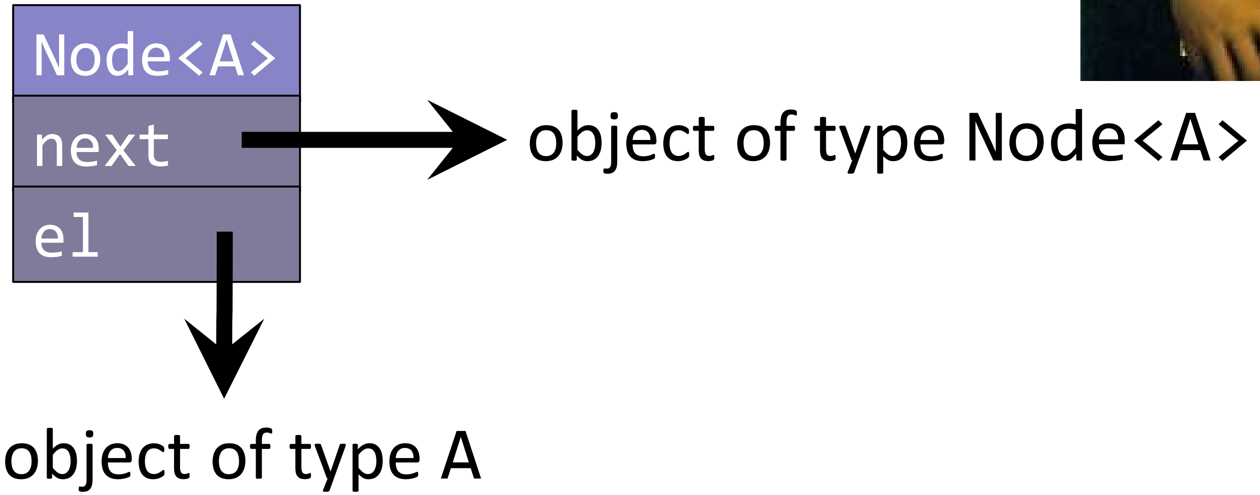


MyLinkedList<E>: Node class



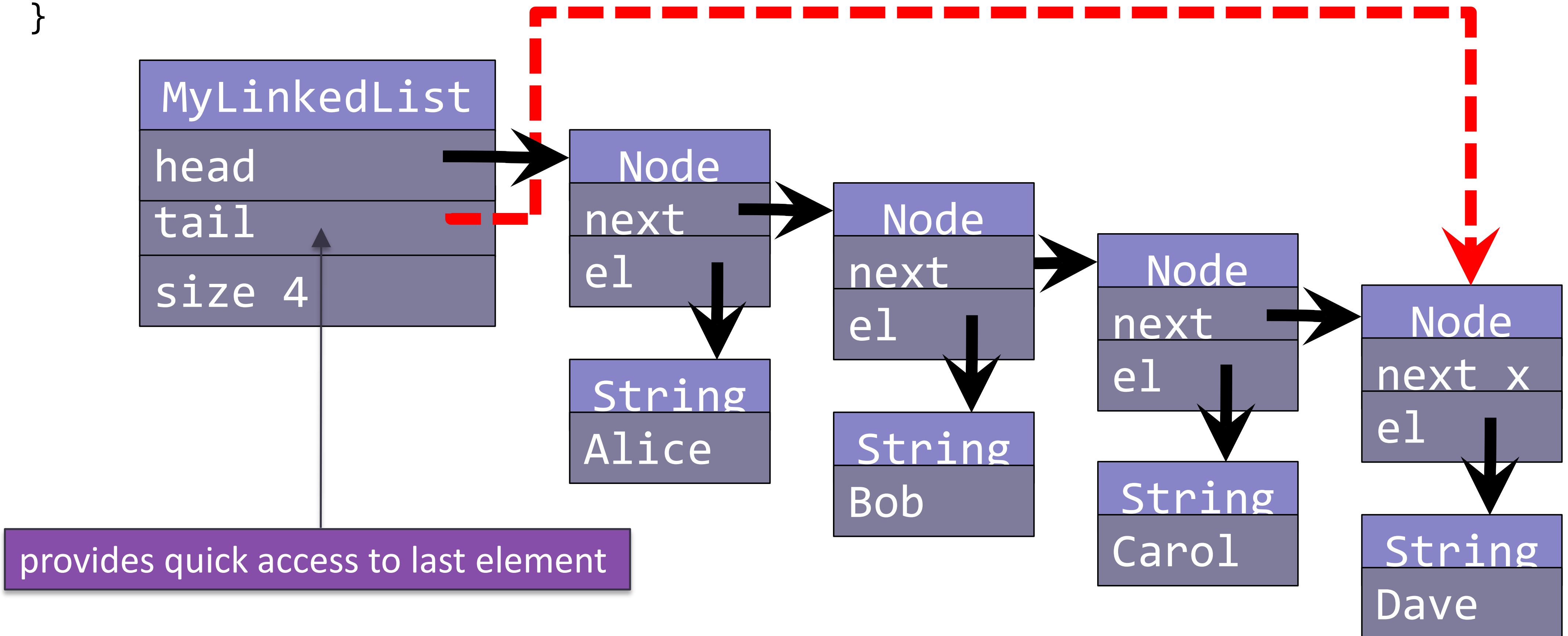
```java
public class MyLinkedList<E> extends AbstractList<E> {
  ...
  private static class Node<A> {
    private A el;
    private Node<A> next;

    public Node(A e, Node<A> n) {
      el = e;
      next = n;
    }

    public Node(A e) {
      this(e, null);
    }
  }
  ...
}
```

recursive datatype/class



`MyLinkedList<E>`: fields (no constructor)

```
public class MyLinkedList<E> extends AbstractList<E> {
  private Node<E> head = null, tail = null;
  private int size;
  ...
}
```



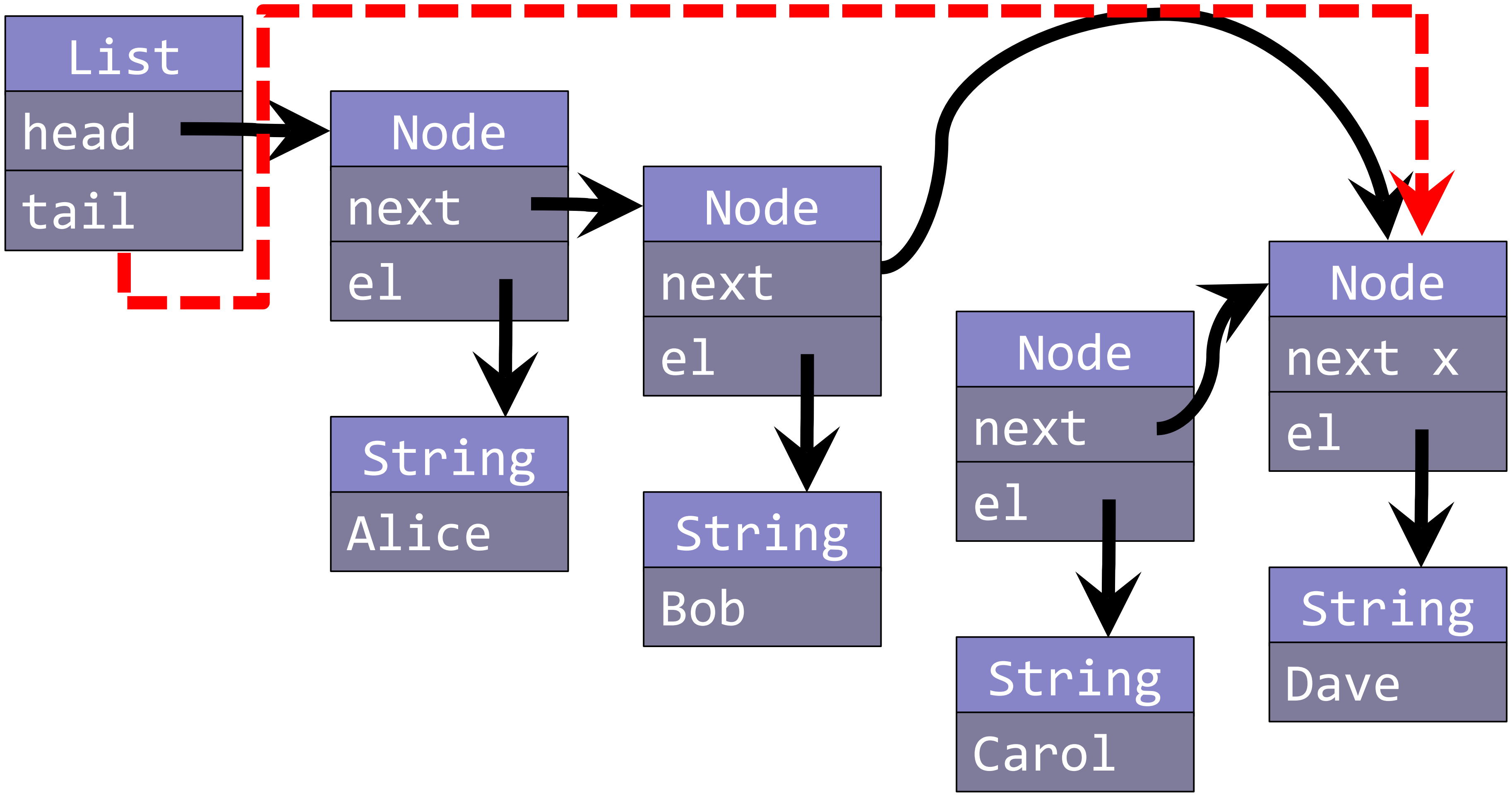
MyLinkedList<E>: get(index)

```java
@Override
public E get(int index) {
  return getNode(index).el;
}

private Node<E> getNode(int index) {
  checkBound(index);
  Node<E> n = head;
  for (int i = 0; i < index; i++) {
    n = n.next;
  }
  return n;
}
```

start at head; follow i next pointers: $O(i)$

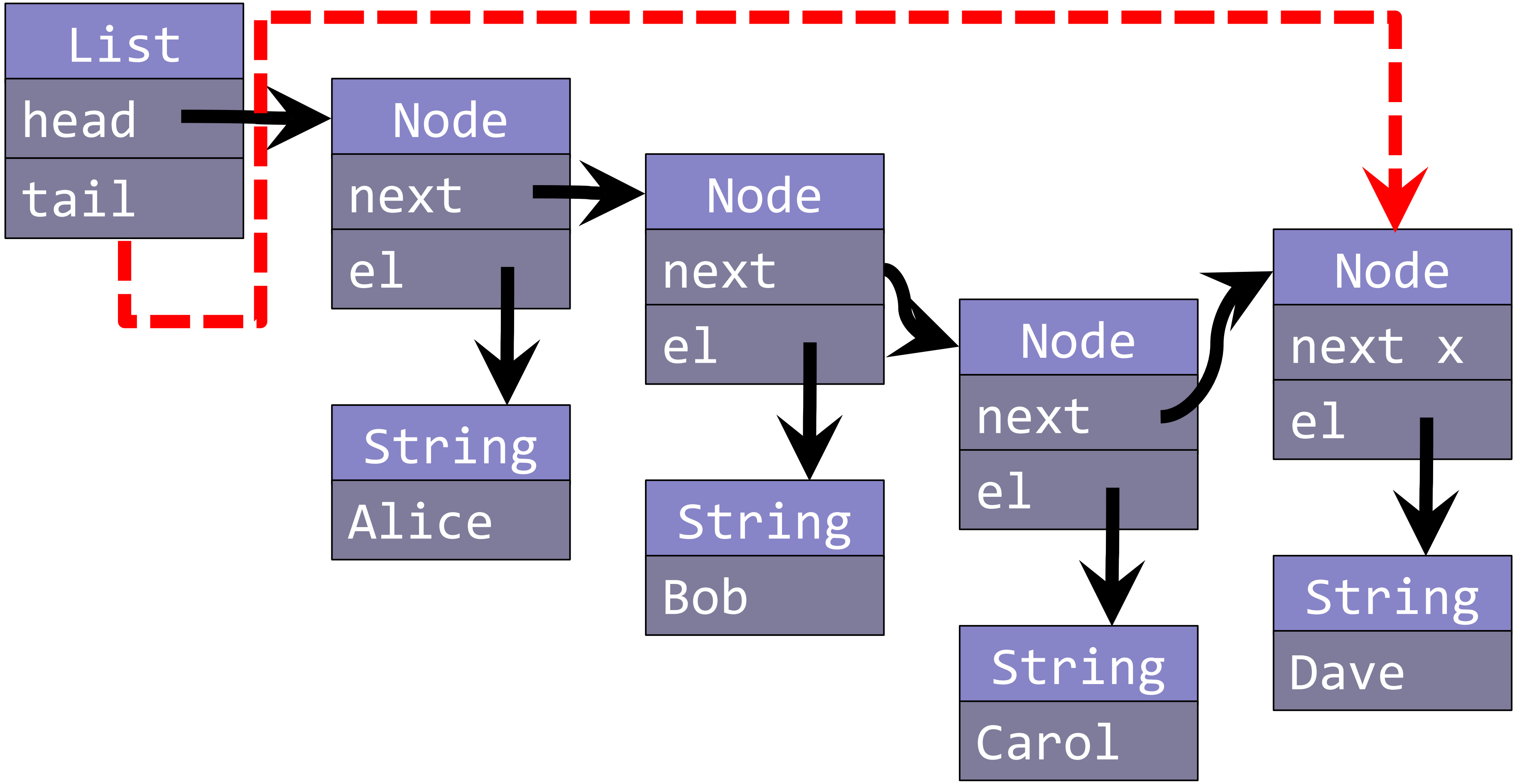
Linked List: add Carol

- If we already have a reference to the insertion point then it can be done in constant time $O(1)$
- However, getting to the right place (via getNode(i)) is $O(i)$!

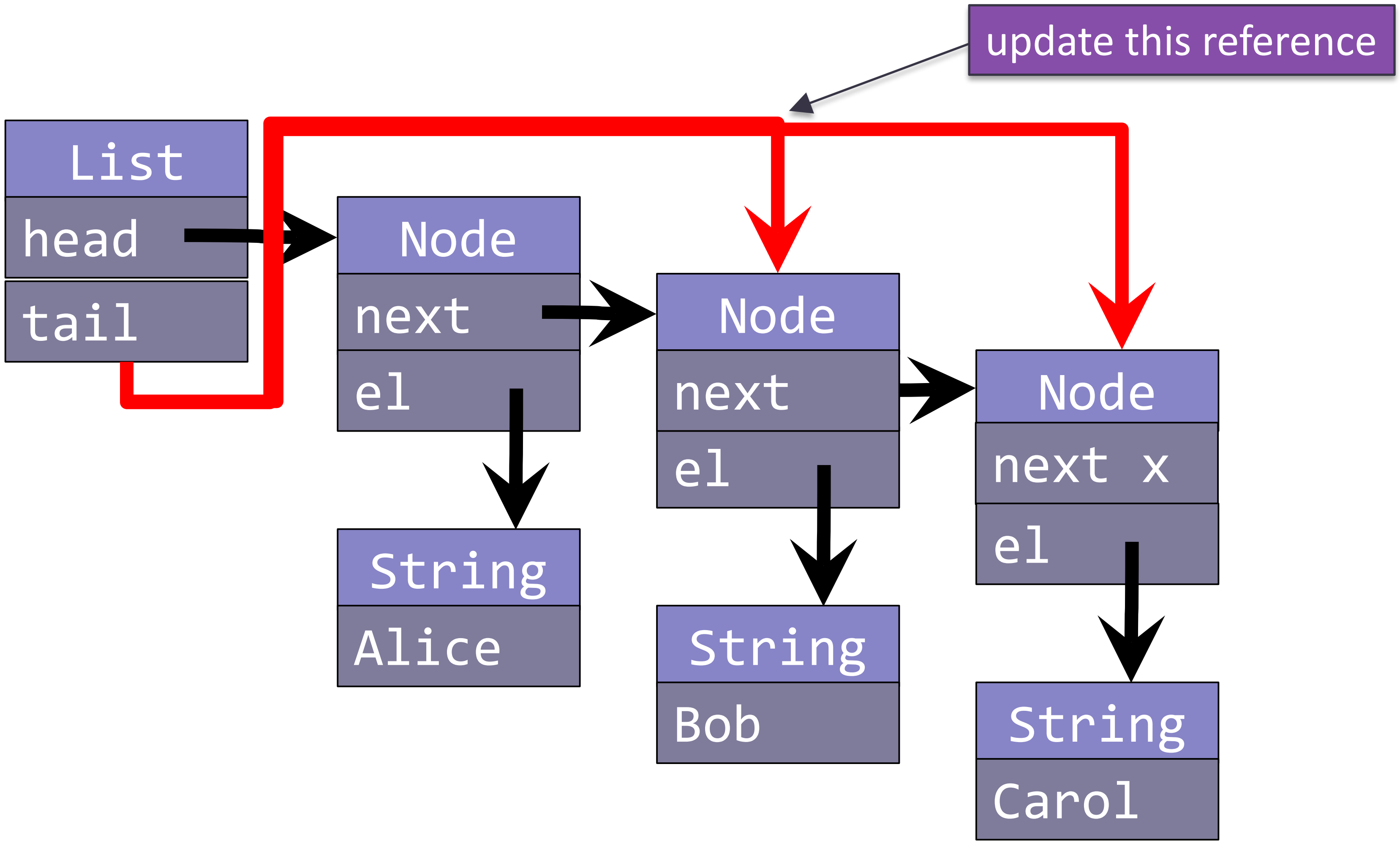


Linked List: add Carol

- If we already have a reference to the insertion point then it can be done in constant time $O(1)$
- However, getting to the right place (via getNode(i)) is $O(i)$!



Linked List: efficient add to the tail



MyLinkedList: add(element) to tail

```
@Override
public boolean add(E e) {
  if (size == 0) {
    head = tail = new Node(e);
  } else {
    tail.next = new Node(e);
    tail = tail.next;
  }
  size++;
  return true;
}
```

for adding the first node in a list we need a special case

MyLinkedList: **add**(index, element)

```java
@Override
public void add(int index, E e) {
 if (index == size) {
   add(e);
   return;
 } else if (index == 0) {
   head = new Node(e, head);
 } else {
   Node<E> n = getNode(index - 1);
   n.next = new Node(e, n.next);
 }
 size++;
}
```

at tail: *O*(1)

at front: *O*(1)

somewhere else: *O*(index)

MyLinkedList: **remove**(index)

explanation: see book ItJPaDS, 24.4

```
@Override
public E remove(int index) {
  checkBound(index);
  E e;
  if (index == 0) {
    e = head.el;
    head = head.next;
    if ( head == null ) {
      tail = null;
    }
  } else {
    Node<E> n = getNode(index - 1);
    e = n.next.el;
    if (index == size - 1) {
      tail = n;
      n.next = null;
    } else {
      n.next = n.next.next;
    }
  }
  size--;
  return e;
}
```

`MyLinkedList` evaluation

- adding elements at the beginning or at the end can be done in $O(1)$ time
- `add(int i, E e)` and `remove(int i)` itself are $O(1)$: we don't have to move the elements like with an arraylist
  - However, finding the right spot is $O(i)$
- idea:
  - extend the iterator:
    `set(E e)`: replace previous element with e
    `add(E e)`: insert e between previous and current element
  - both $O(1)$
  - this is provided by the **`ListIterator`** interface (along with methods for going backwards through the list)
  - only helps if you have to handle all elements anyway

Lecture 7: Lambda expressions & More recursive data structures