Collections

Lecture 6 (15 March 2022)

Collections

variable length containers

sometimes the size of a container is not known in advance

- **`Strings`** have a fixed length
- **`StringBuffers`** can be changed
  - it is always possible to add a character

ordinary arrays have a fixed length

`ArrayList` and **`LinkedList`** have a *variable length*

- it is always possible to add an element, at any place
- we can remove an element without the need to shift elements explicitly
- both classes implement the (generic) interface **`List<T>`**

list based map (map was introduced in lecture 5)

```
public class MapList<K extends Comparable<K>, V> {
  private final List<Pair<K,V>> map;

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

  public void add(K key, V value) {
    for (var pair: map) {
      if (pair.getKey().equals(key)) {
        pair.setVal( value );
        return;
      }
    }
    map.add(new Pair<>(key, value));
  }

  public V get(K key) {
    for (var pair: map) {
      if (pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }
  ...
}
```

`List` is an interface

no size argument

`ArrayList` is class implementing `List`

always fits

the interface `Collection`

we have several containers in Java

- `ArrayList`, `LinkedList`, `Vector`, `Set`

many similar operations on these containers

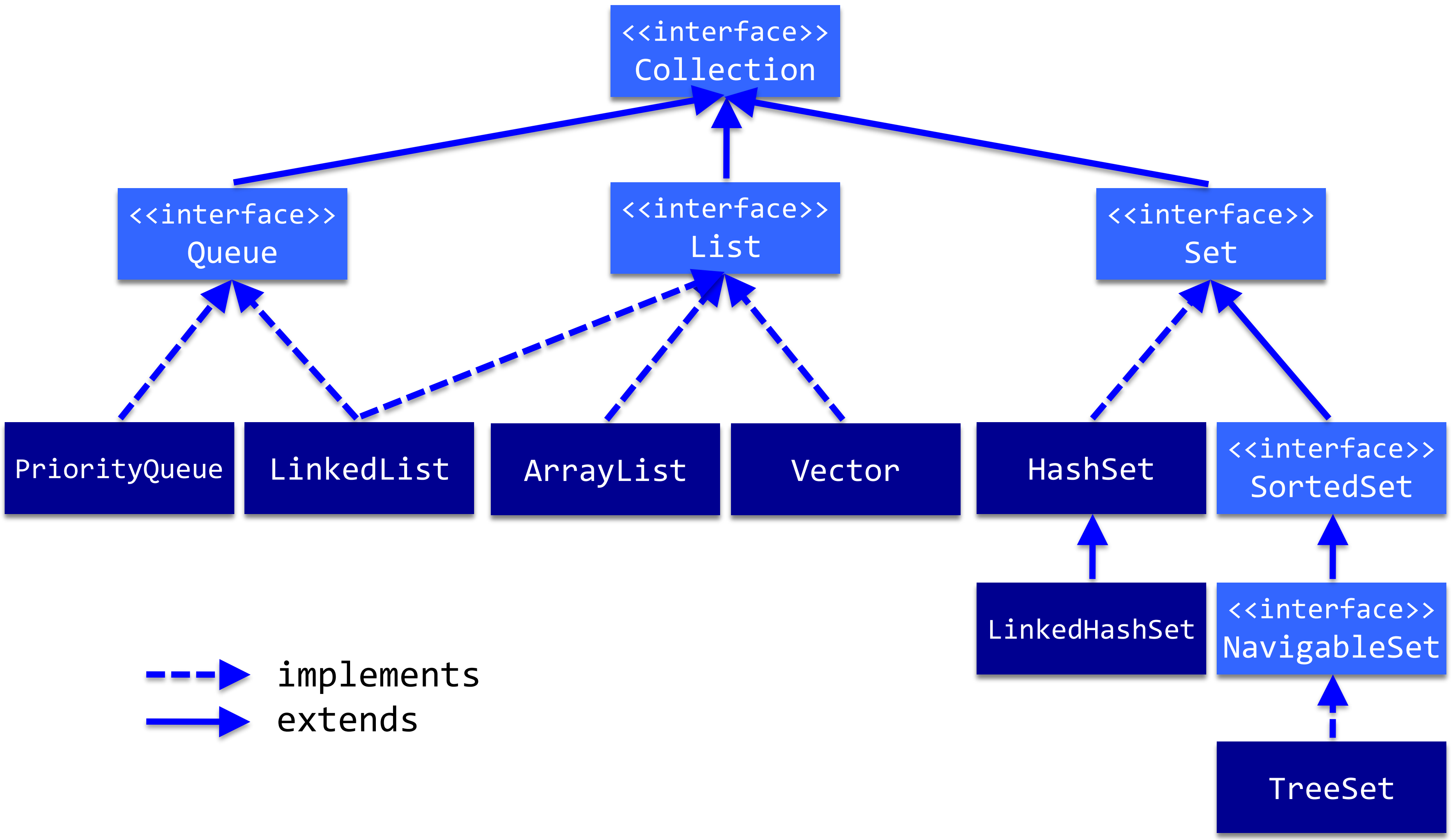
- `isEmpty`, `contains`, `equals`, `size`

the interface **`Collection`** yields a uniform way to handle these kind of operations

warning: there is also a (utility) class **`Collections`**

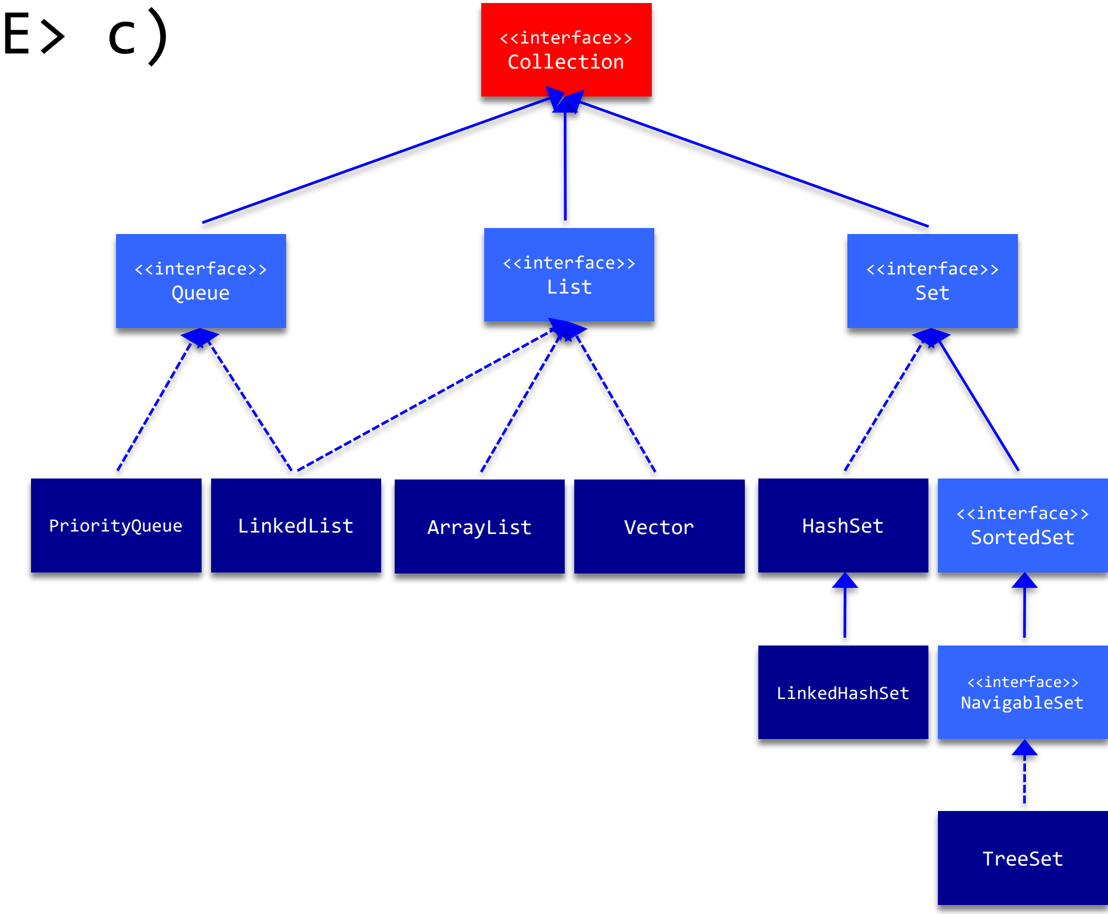
- `Collections` is similar to `Arrays`: set of basic operations provided as static methods
- don't confuse them

Collection interface hierarchy



main methods in interface `Collection<E>`

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
void    clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean equals(Object o)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int     size()
```



iterators

- an `Iterator` offers a standard way to scan and handle all elements of a collection
  - this is a generic interface; many implementations possible
  - 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 modify the collection

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

is there a next object?

yield next object; advance iterator one position

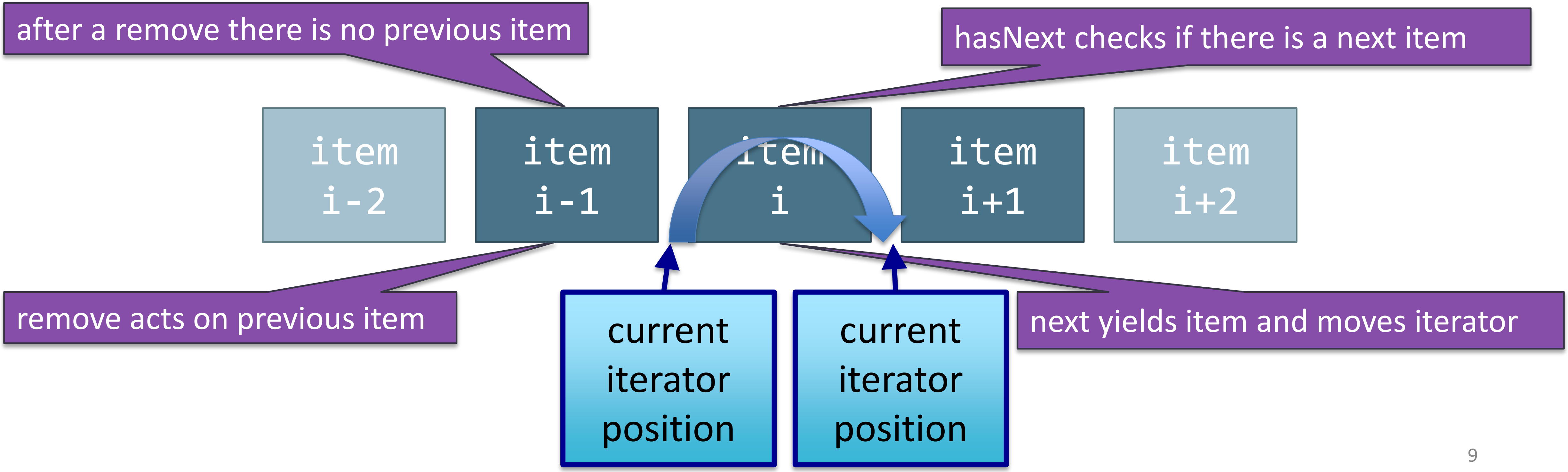
remove last returned object

optional operation, can throw a `NotImplementedException`

`Iterator` interface

an iterator is conceptually between elements;

- it does not refer to a particular object



iterator application in `ListMap`: getting element

get with an enhanced for loop:

```
public class MapList<K extends Comparable, V> {
  private final List<Pair<K,V>> map;
  ...
  public V get(K key) {
    for (var pair: map) {
      if (pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }
}
```

get with an iterator:

```
public class MapList<K extends Comparable, V> {
  private final List<Pair<K,V>> map;
  ...
  public V get(K key) {
    Iterator<Pair<K,V>> mapIt = map.iterator();
    while ( mapIt.hasNext() ) {
      var pair = mapIt.next();
      if (pair.getKey().equals(key)) {
        return pair.getVal();
      }
    }
    return null;
  }
}
```

note: next yields the element and advances the iterator!

using next() twice gives two successive elements: *The* most frequent error with iterators

iterator application in `ListMap`: removing element

remove with an iterator:

```
public class MapList<K extends Comparable, V> {
  private final List<Pair<K,V>> map;
  ...
  public boolean remove(K key) {
    Iterator<Pair<K, V>> mapIt = map.iterator();
    while (mapIt.hasNext()) {
      if (mapIt.next().getKey().equals(key)) {
        mapIt.remove();
        return true;
      }
    }
    return false;
  }
}
```

next used only once: no need to keep it around in a variable

remove with a for loop:

```
public class MapList<K extends Comparable, V> {
  private final List<Pair<K,V>> map;
  ...
  public boolean remove(K key) {
   for (int i = 0; i < map.size(); i++) {
      var pair = map.get(i);
      if (pair.getKey().equals(key)) {
        map.remove(i);
        return true;
      }
    }
    return false;
  }
}
```

get can be inefficient (O(n))

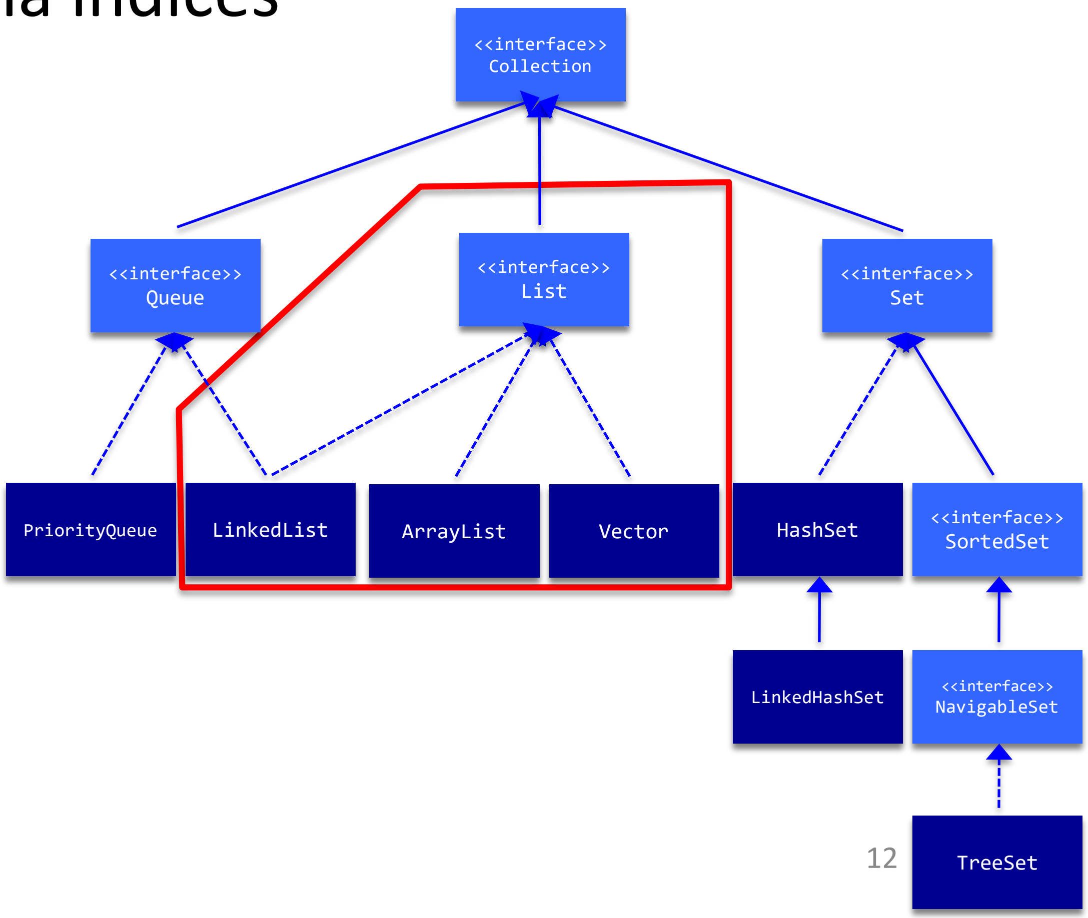
Lists

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)` …

`ArrayList` implements the interface **`List`**

other implementations are
`LinkedList` and **`Vector`**



collection relationships

- **`Set`**
  - does not contain duplicates
  - can (sometimes) be sorted !
- **`List`**
  - elements are ordered (insertion order is maintained)
  - elements can occur more than once
  - **`ArrayList`** and **`LinkedList`** implement the **`List`** interface
  - **`Vector`** is very similar to **`ArrayList`** in API,
    vectors are *thread-safe* and hence somewhat slower (more about threads/thread-safety in lecture 12-14)

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`**

three ways to access all list elements

```
for (int i = 0; i < list.size(); i++) {
    Card card = list.get(i);
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen1: " + card);
    }
}
```

+ any order possible
– get(i) can be inefficient

```
for (Card card : list) {
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen2: " + card);
    }
}
```

+ compact
+ efficient
– list cannot be changed
- order is fixed

```
Iterator<Card> iter = list.iterator();
while (iter.hasNext()) {
    Card card = iter.next();
    if (card.face == Card.Face.Queen) {
        System.out.println("Queen3: " + card);
    }
}
```

+ efficient
+ flexible
- order is fixed
+/- remove only last item

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 E[] data;            // array containing the elements

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

type cast: we have no constructor for E[ ]

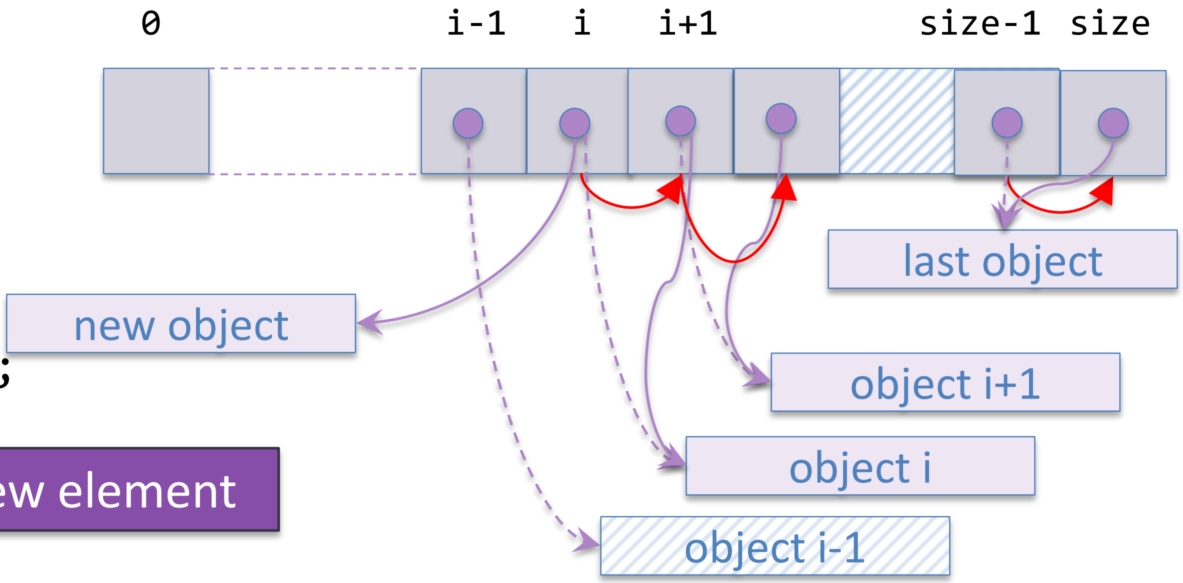
MyArrayList: size(), get(index), add(element)

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

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

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

helper method (next slide)

helper method (next slide)

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




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

makes room for the new element

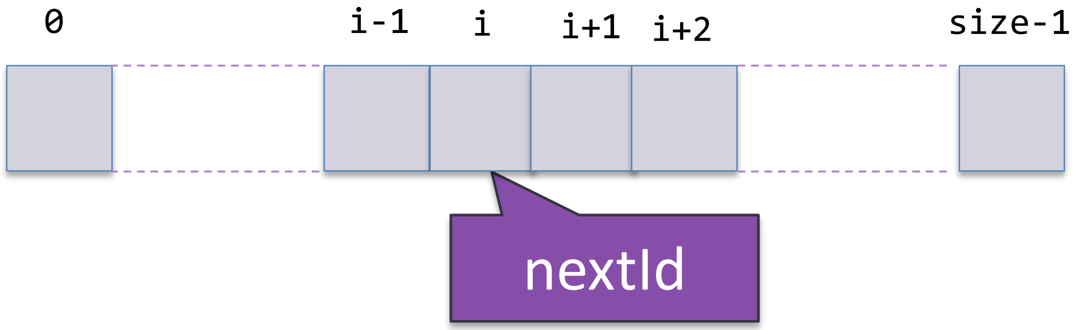
```java
private void ensureCapacity(int cap) {
  if (cap > data.length) {
    E[] es = (E[]) new Object[Math.max(data.length * 2, cap)];
    System.arraycopy(data, 0, es, 0, size);
    data = es;
  }
}
```

new array will be twice as big

MyArrayList: remove(index), checkBound(index)

```
@Override
public E remove(int i) {
  checkBound(i);
  E r = 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()



- nextId: index of next element

MyArrayList: Iterator<E> iterator()

```
@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 data[nextId++];
    } else {
      throw new NoSuchElementException();
    }
  }
}
```

local/nested/inner class (more next week)

`MyArrayList`: evaluation

Java **`ArrayList`** is very (not really) similar to **`MyArrayList`**

simple and works fine in many situations

unless:

- use `add(i,e)` a lot (with `i < size`)
- remove a lot of elements
- these are all $O(N)$ work

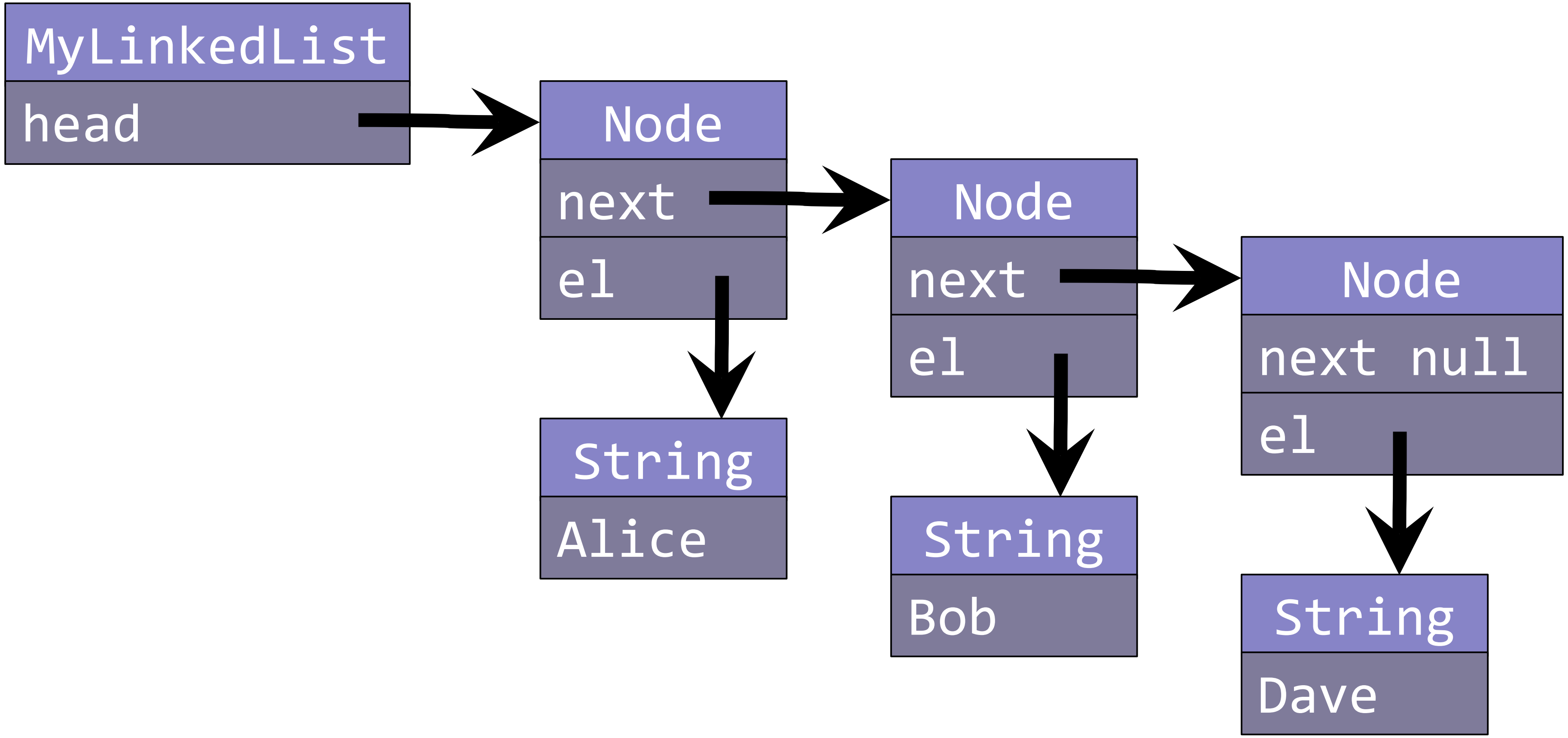
how to improve the $O(N)$ operations?

use a linked data structure (recursive data structure)

LinkedList

Linked List

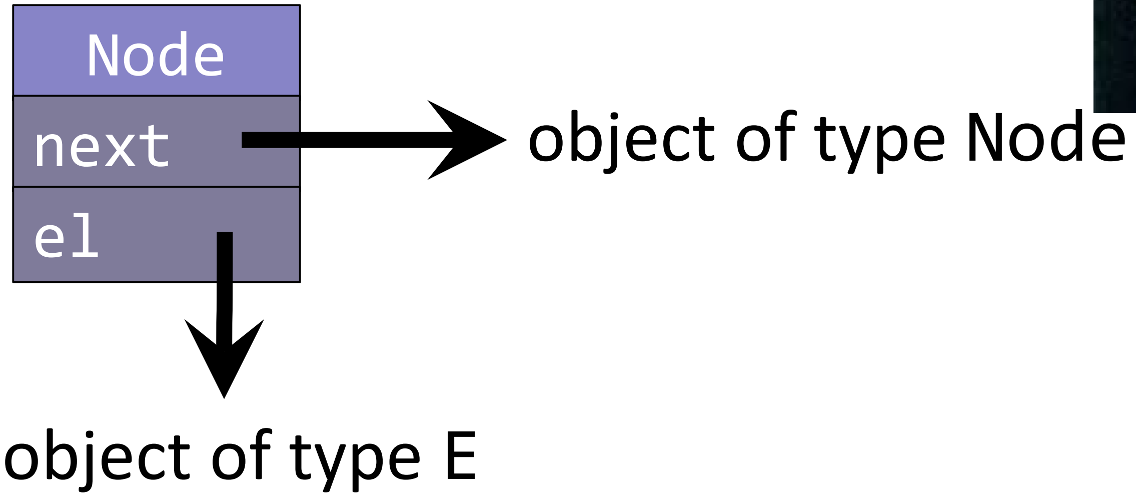
basic idea:

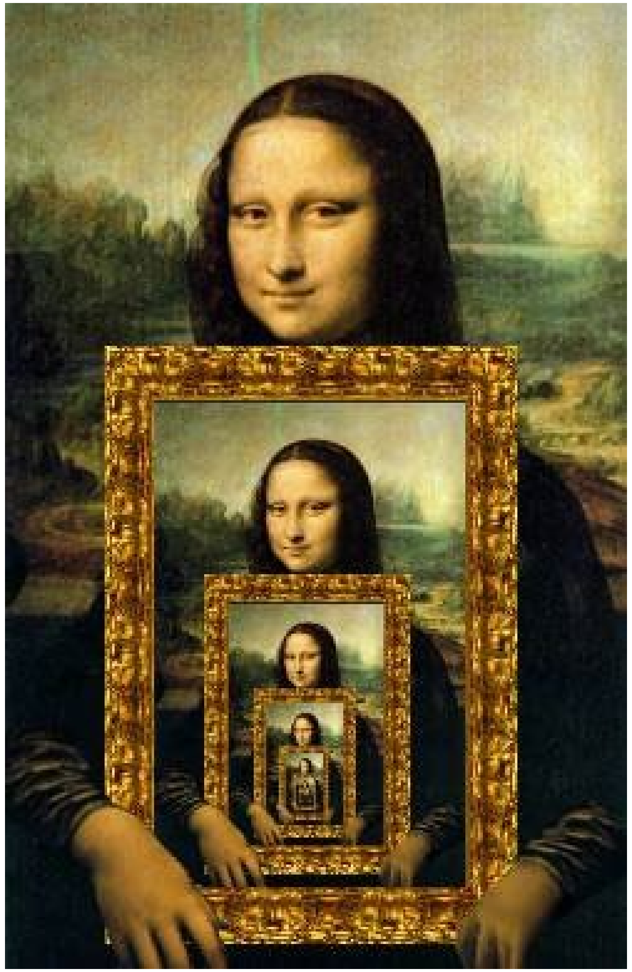


MyLinkedList<E>: Node class

```
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);
    }
  }
  ...
}
```

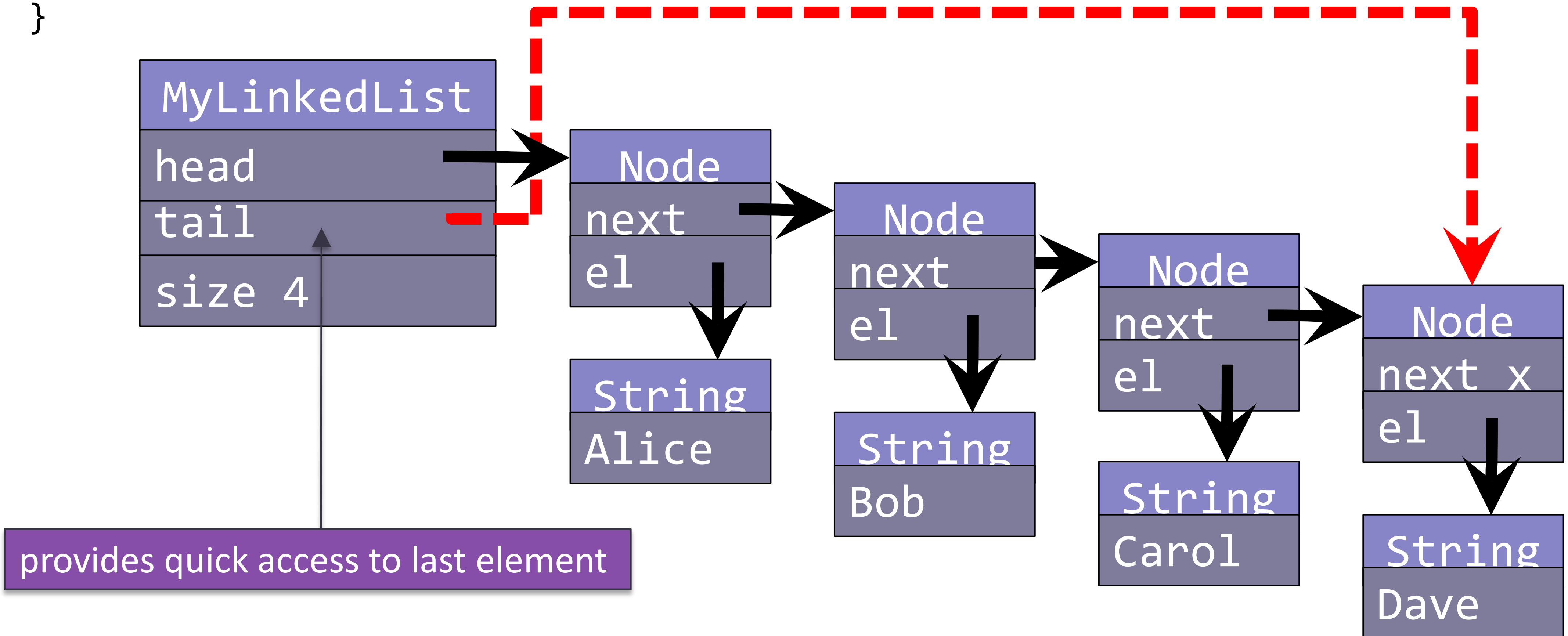
recursive datatype/class





MyLinkedList<E>: fields (no constructor)

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



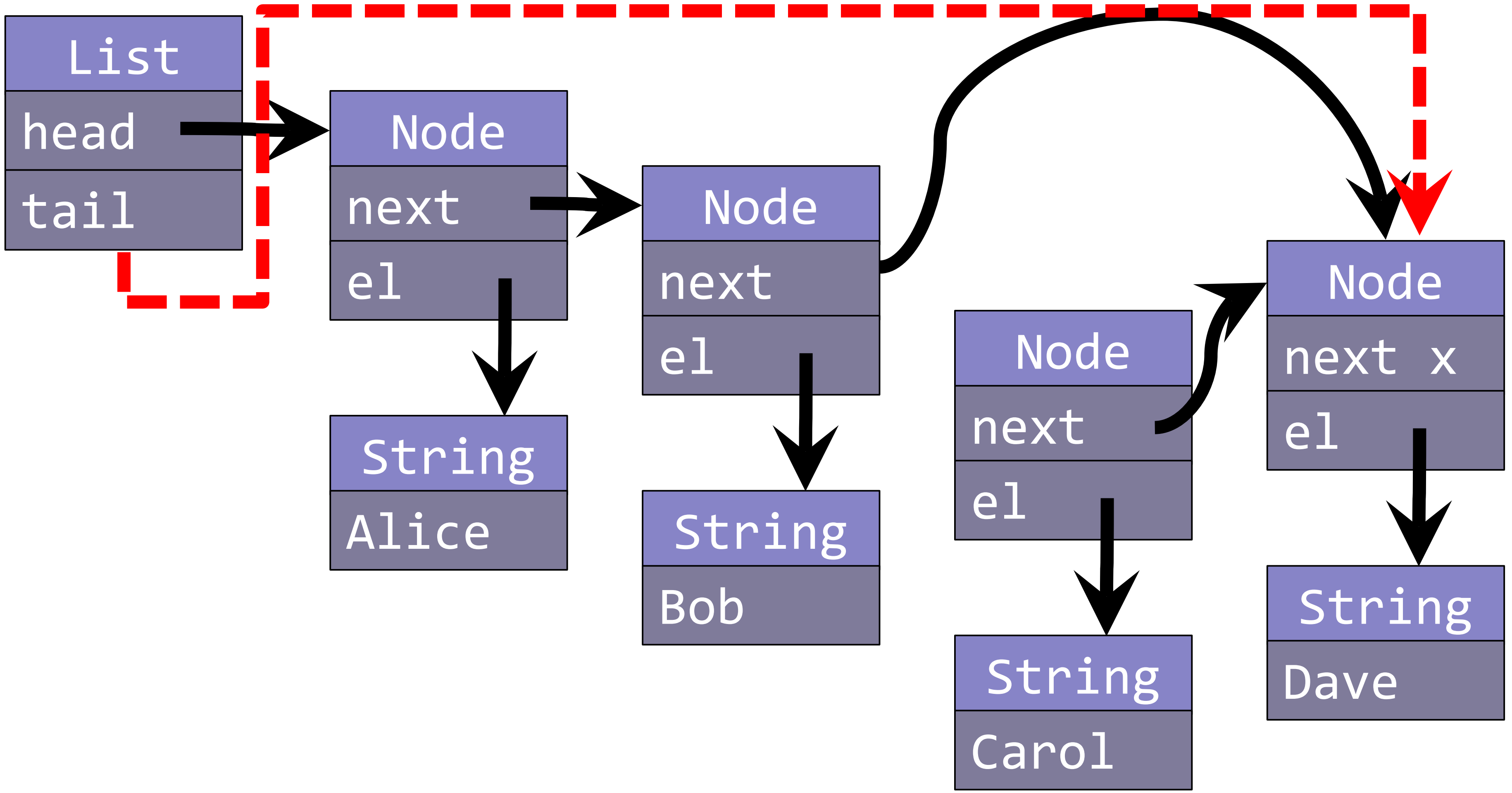
MyLinkedList<E>: get(index)

```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: $O(i)$

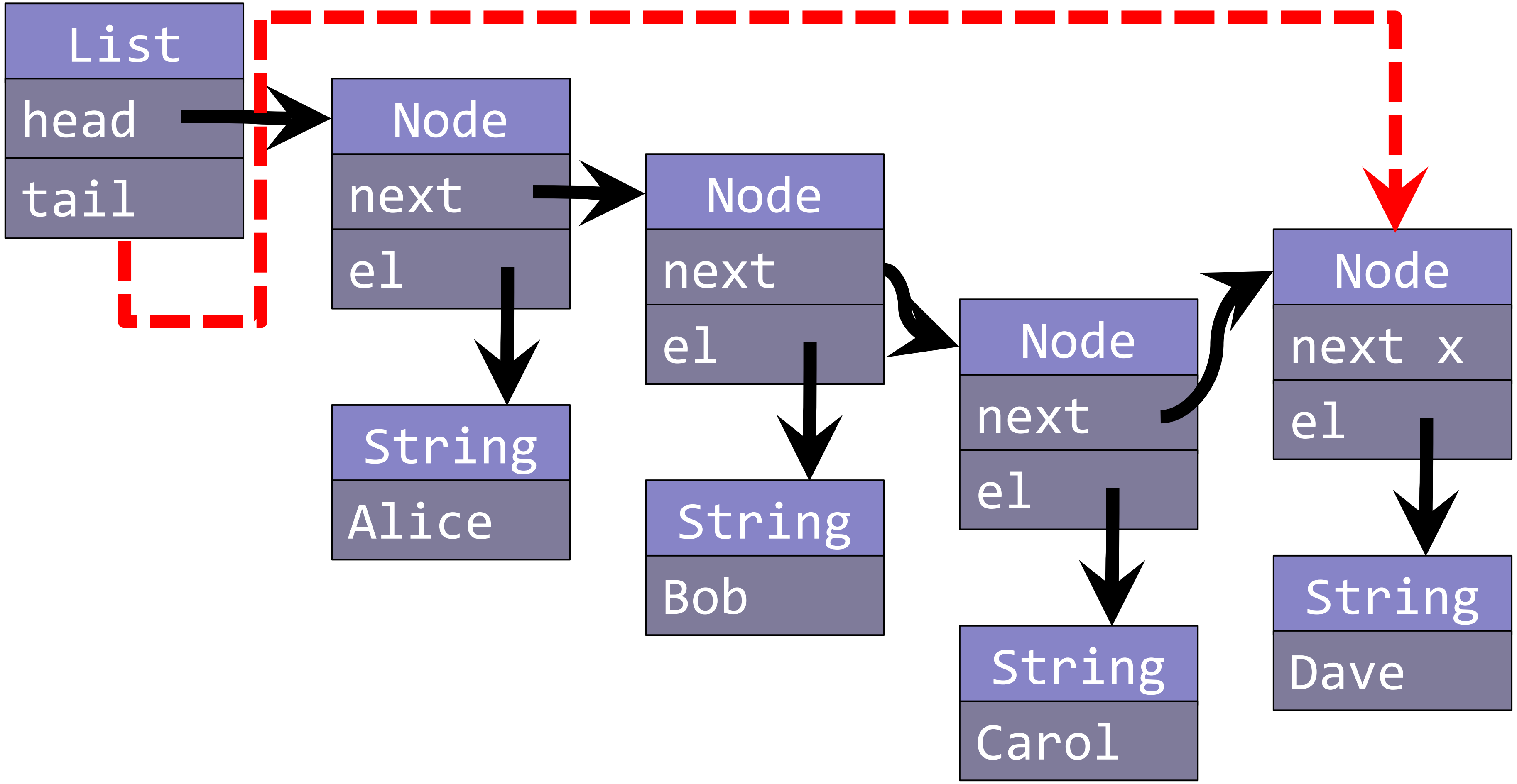
Linked List: add Carol

- this can be done in constant time (O(1)), 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 $O(i)$!

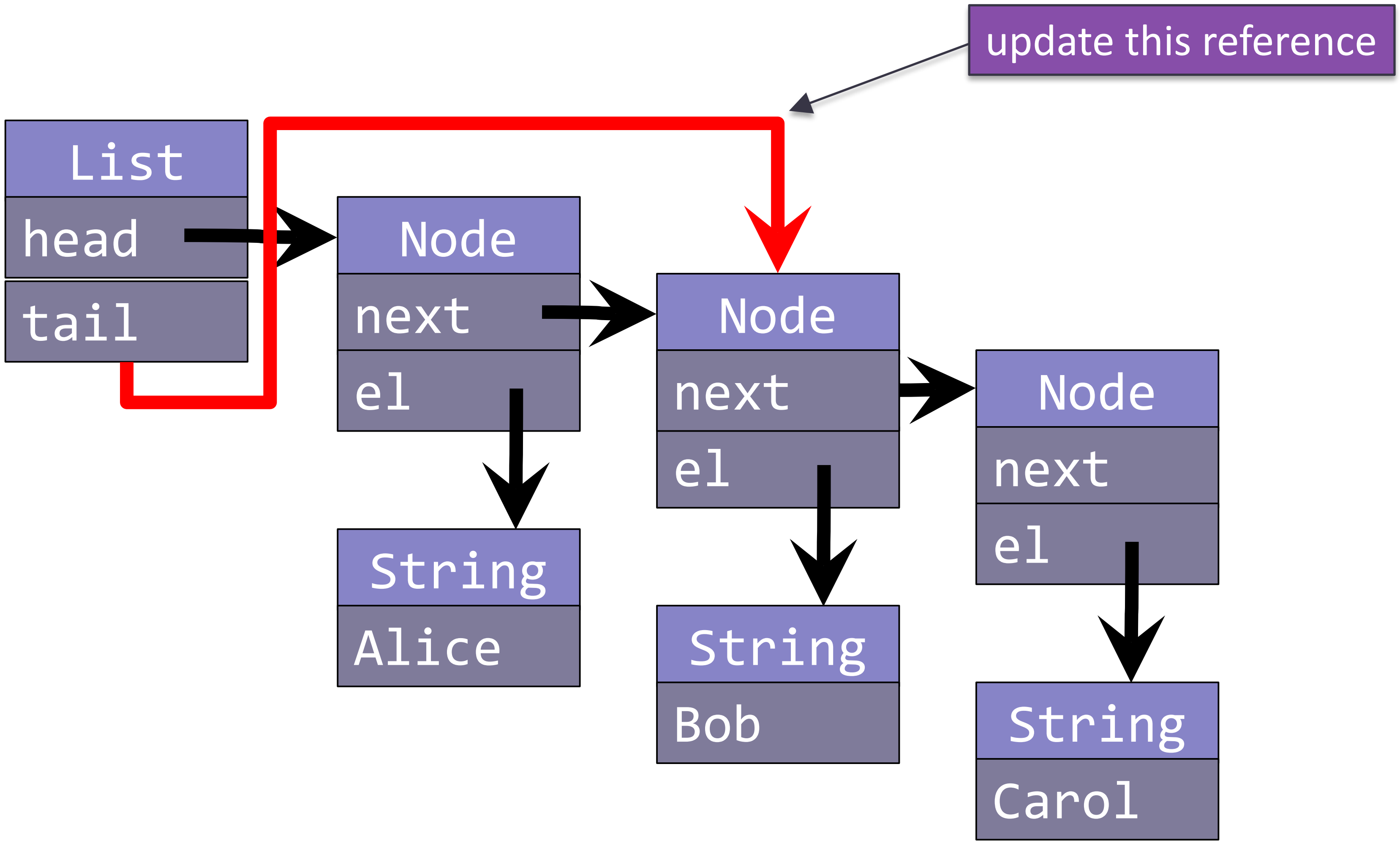


Linked List: add Carol

- this can be done in constant time (O(1)), 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)

```
@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

```java
@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

Finally

Lecture 7: Lambda expressions & More recursive data structures