Generic & Recursive Data Structures

Object-Oriented Programming Lecture 6
Introduction to Java Programming (Liang): chapter 15.{4,6} & **24**

March 2, 2021

Radboud University

IN·DEI·NOMINE·FELICITER

today’s lecture

why there are 2 different implementation of the List interface in Java?

today's lecture

why there are 2 different implementation of the List interface in Java?

- complexity of operations

today’s lecture

why there are 2 different implementation of the List interface in Java?

- complexity of operations

how to construct recursive data-types

today’s lecture

why there are 2 different implementation of the List interface in Java?

- complexity of operations

how to construct recursive data-types

Java tooling

- anonymous classes
- lambda-expressions

INNER CLASSES & LAMBDA FUNCTIONS

data versus functions

Java is based classes

- functions/methods exist only as part of a class
- Besides methods these classes contain data (attributes)

data versus functions

Java is based classes

- functions/methods exist only as part of a class
- Besides methods these classes contain data (attributes)

sometimes we need tailor made ‘functions’ without data

- just a simple task or action, no data involved
- e.g. comparing objects, an implementation of a strategy interface, handler for an I/O action, ..

data versus functions

Java is based classes

- functions/methods exist only as part of a class
- Besides methods these classes contain data (attributes)

sometimes we need tailor made ‘functions’ without data

- just a simple task or action, no data involved
- e.g. comparing objects, an implementation of a strategy interface, handler for an I/O action, ..

Java provides several solutions

1. a locally defined class,
   in contrast to classes having their own file

data versus functions

Java is based classes

- functions/methods exist only as part of a class
- Besides methods these classes contain data (attributes)

sometimes we need tailor made ‘functions’ without data

- just a simple task or action, no data involved
- e.g. comparing objects, an implementation of a strategy interface, handler for an I/O action, ..

Java provides several solutions

1. a locally defined class,
   in contrast to classes having their own file
2. an anonymous class

data versus functions

Java is based classes

- functions/methods exist only as part of a class
- Besides methods these classes contain data (attributes)

sometimes we need tailor made ‘functions’ without data

- just a simple task or action, no data involved
- e.g. comparing objects, an implementation of a strategy interface, handler for an I/O action, ..

Java provides several solutions

1. a locally defined class,
   in contrast to classes having their own file
2. an anonymous class
3. a lambda expression

a Person class

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

a Person class

```java
public class Person implements Comparable<Person> {
  private final String name;
  private final int id;

  public Person(String name, int id) {
    this.name = name;
    this.id = id;
  }
  @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
```

a Person class

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

  public Person(String name, int id) {
    this.name = name;
    this.id = id;
  }
  @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
```

compare persons by their name

```
< 0: this < p
= 0: equals p
> 0: this > p
```

a Person class

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

  public Person(String name, int id) {
    this.name = name;
    this.id = id;
  }
  @Override
  public int compareTo(Person p) {
    return name.compareTo(p.name);
  }
  @Override
  public String toString() {
    return name + " (" + id + ")";
  }
  public String getName() { return name; }
  public int    getId()   { return id; }
}
```

compare persons by their name

```
< 0: this < p
= 0: equals p
> 0: this > p
```

a Group of Persons

```
public class Group {
  private List<Person> list;
```

a Group of Persons

```
public class Group {
  private List<Person> list;

  public Group(Person ... array) {
    list = new ArrayList(Arrays.asList(array));
  }
```

a Group of Persons

```
public class Group {
  private List<Person> list;

  public Group(Person ... array) {
    list = new ArrayList(Arrays.asList(array));
  }
```

Java syntax for a sequence of arguments of the same type (passed as an array)

a Group of Persons

```
public class Group {
  private List<Person> list;

  public Group(Person ... array) {
    list = new ArrayList(Arrays.asList(array));
  }

  public Group() {
    this(new Person("Alice",7), new Person("Dave",9),
         new Person("Bob",2),   new Person("Carol",6));
  }
```

Java syntax for a sequence of arguments of the same type (passed as an array)

a Group of Persons

```
public class Group {
  private List<Person> list;

  public Group(Person ... array) {
    list = new ArrayList(Arrays.asList(array));
  }

  public Group() {
    this(new Person("Alice",7), new Person("Dave",9),
         new Person("Bob",2),   new Person("Carol",6));
  }

  public List<Person> getList() { return list; }
  public boolean add(Person p)  { return list.add(p); }

  @Override
  public String toString()      { return list.toString(); }
}
```

Java syntax for a sequence of arguments of the same type (passed as an array)

Sorting a Group of Persons

```
public class GroupOfPersons {
  public static void main(String[] args) {
    run1();
  }
  private static void run1() {
    Group g = new Group();
    Collections.sort(g.getList());
    System.out.println("run1: " + g);
  }
```

Sorting a Group of Persons

```
public class GroupOfPersons {
  public static void main(String[] args) {
    run1();
  }
  private static void run1() {
    Group g = new Group();
    Collections.sort(g.getList());
    System.out.println("run1: " + g);
  }
```

- this yields:

```
run1: [Alice (7), Bob (2), Carol (6), Dave (9)]
```

ad-hoc sorting with **nested class**

```
public class GroupOfPersons {
  private void run2() {
    Group g = new Group();
    Collections.sort(g.getList(), new CompareId());
    System.out.println("run2: " + g);
  }
}
```

to sort persons on Id we need a Comparator class

ad-hoc sorting with **nested class**

```java
public class GroupOfPersons {
  private void run2() {
    Group g = new Group();
    Collections.sort(g.getList(), new CompareId());
    System.out.println("run2: " + g);
  }

  private static class CompareId implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
      return p2.getId() - p1.getId();
    }
  }
}
```

to sort persons on Id we need a Comparator class

the nested class

ad-hoc sorting with **nested class**

```
public class GroupOfPersons {
  private void run2() {
    Group g = new Group();
    Collections.sort(g.getList(), new CompareId());
    System.out.println("run2: " + g);
  }

  private static class CompareId implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
      return p2.getId() - p1.getId();
    }
  }
}
```

to sort persons on Id we need a Comparator class

the nested class

- this yields:

```
run2: [Dave (9), Alice (7), Carol (6), Bob (2)]
```

ad-hoc sorting with **nested class**

```
public class GroupOfPersons {
  private void run2() {
    Group g = new Group();
    Collections.sort(g.getList(), new CompareId());
    System.out.println("run2: " + g);
  }

  private static class CompareId implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
      return p2.getId() - p1.getId();
    }
  }
}
```

to sort persons on Id we need a Comparator class

the nested class

nested class can be `public` so that others can use it

- this yields:

```
run2: [Dave (9), Alice (7), Carol (6), Bob (2)]
```

alternative: multiple classes in one file (only one is public)

```
class ComparePerson implements Comparator<Person> {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.toString().compareTo(p2.toString());
  }
}
public class GroupOfPersons {
  public static void main(String[] args) {
    GroupOfPersons g = new GroupOfPersons();
    g.run3();
  }
  private void run3() {
    Group g = new Group();
    g.add(new Person("Alice",2));
    Collections.sort(g.getList(), new ComparePerson());
    System.out.println("run3: " + g);
  }
}
```

alternative: multiple classes in one file (only one is public)

```
class ComparePerson implements Comparator<Person> {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.toString().compareTo(p2.toString());
  }
}
public class GroupOfPersons {
  public static void main(String[] args) {
    GroupOfPersons g = new GroupOfPersons();
    g.run3();
  }
  private void run3() {
    Group g = new Group();
    g.add(new Person("Alice",2));
    Collections.sort(g.getList(), new ComparePerson());
    System.out.println("run3: " + g);
  }
}
```

in file
GroupOfPersons.java

alternative: multiple classes in one file (only one is public)

```
class ComparePerson implements Comparator<Person> {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.toString().compareTo(p2.toString());
  }
}
public class GroupOfPersons {
  public static void main(String[] args) {
    GroupOfPersons g = new GroupOfPersons();
    g.run3();
  }
  private void run3() {
    Group g = new Group();
    g.add(new Person("Alice",2));
    Collections.sort(g.getList(), new ComparePerson());
    System.out.println("run3: " + g);
  }
}
```

package visible

in file
GroupOfPersons.java

alternative: multiple classes in one file (only one is public)

```
class ComparePerson implements Comparator<Person> {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.toString().compareTo(p2.toString());
  }
}
public class GroupOfPersons {
  public static void main(String[] args) {
    GroupOfPersons g = new GroupOfPersons();
    g.run3();
  }
  private void run3() {
    Group g = new Group();
    g.add(new Person("Alice",2));
    Collections.sort(g.getList(), new ComparePerson());
    System.out.println("run3: " + g);
  }
}
```

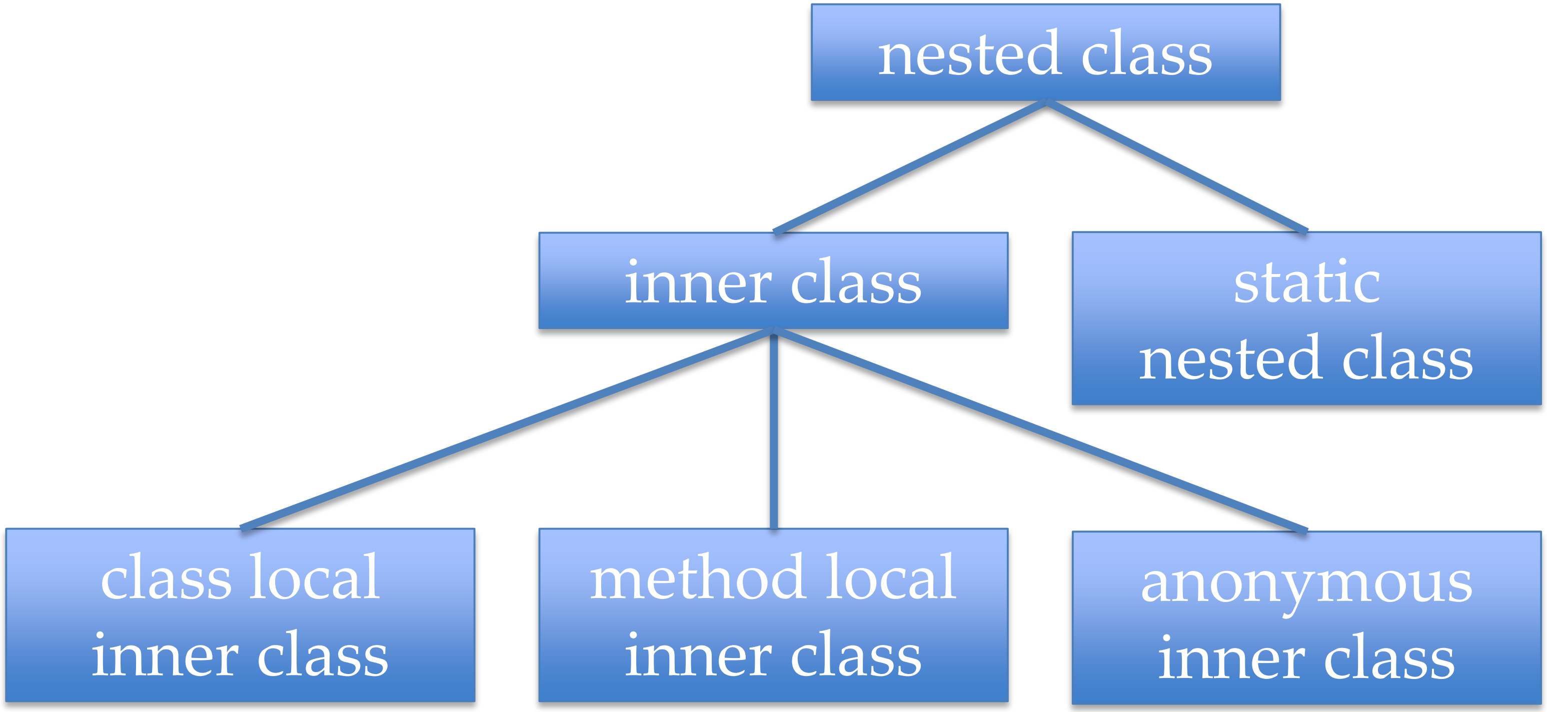
package visible

in file GroupOfPersons.java

- this yields:

```
run3: [Alice (2), Alice (7), Bob (2), Carol (6), Dave (9)]
```

kinds of nested classes in Java



nested class

inner class

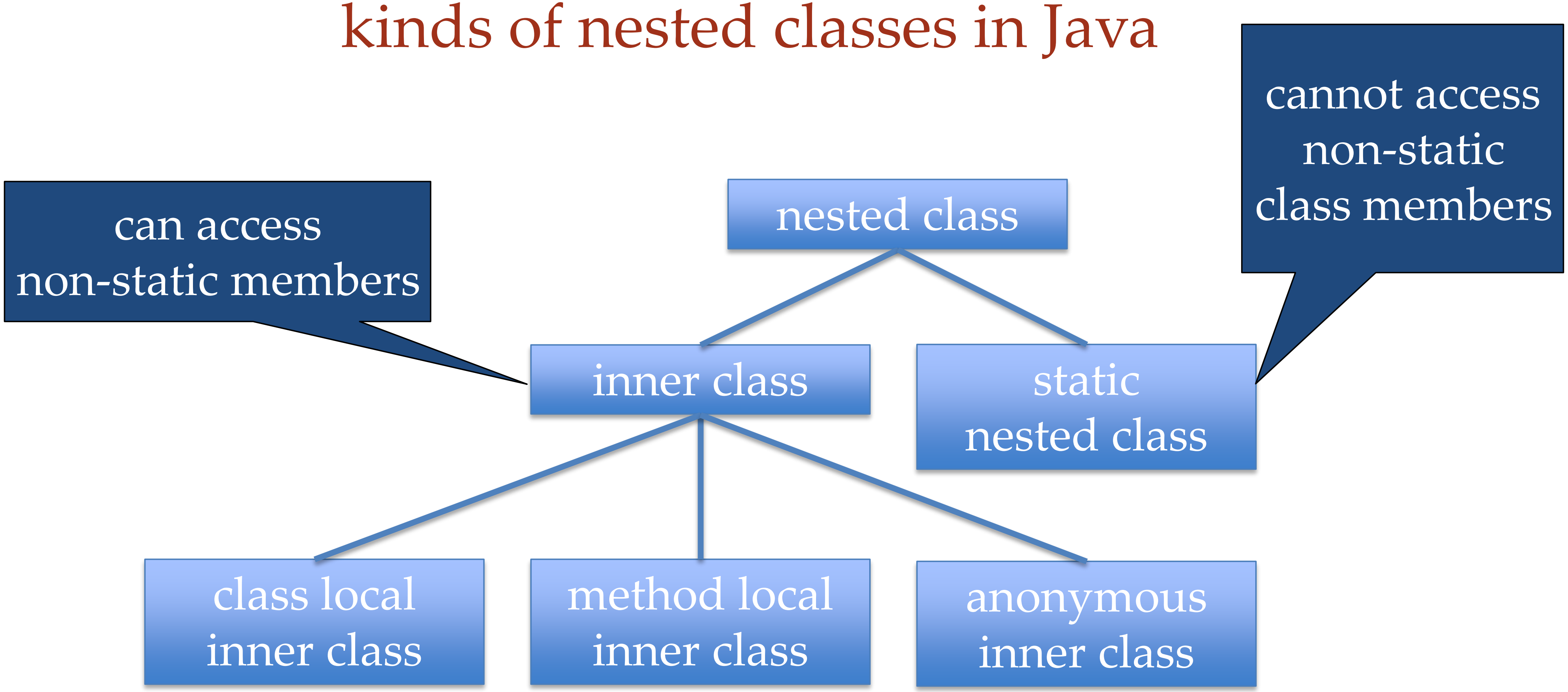
static nested class

class local inner class

method local inner class

anonymous inner class

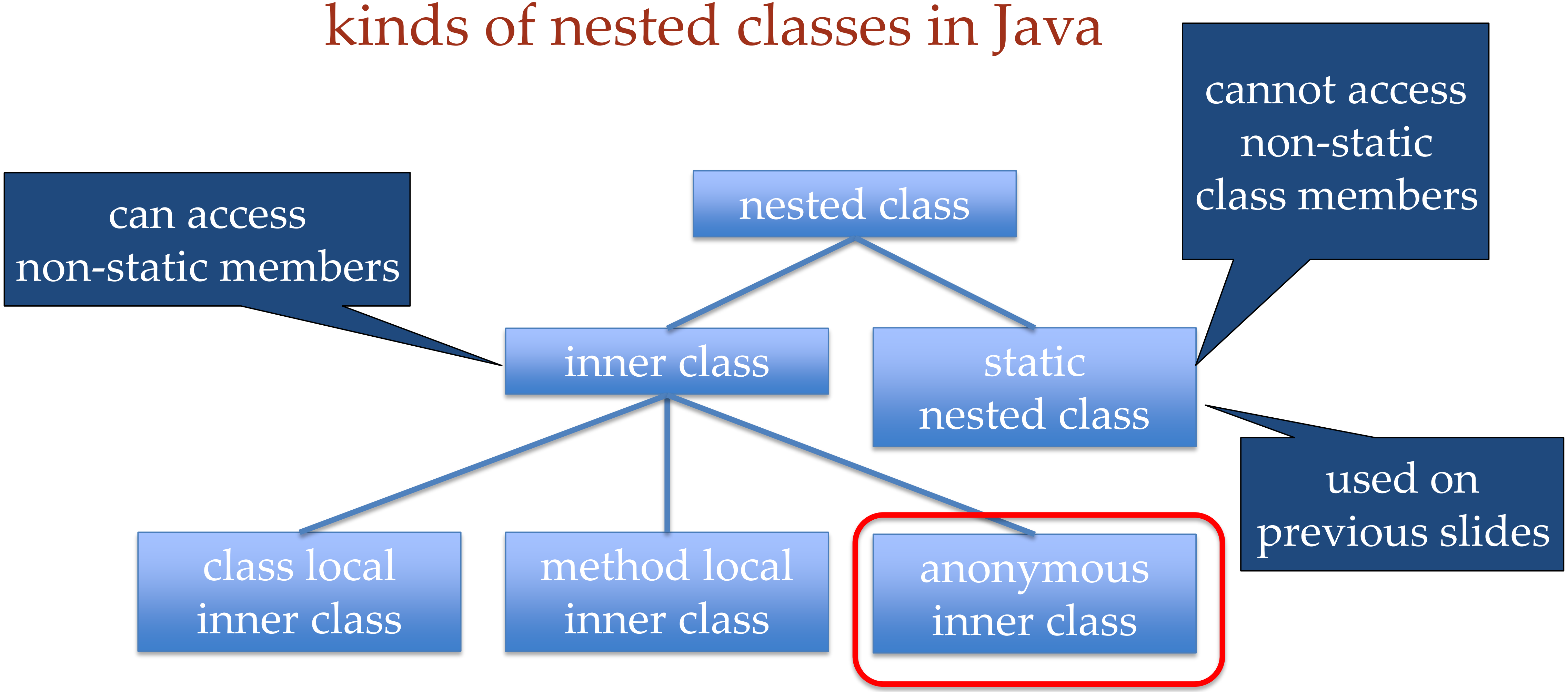
kinds of nested classes in Java



can access non-static members

cannot access non-static class members

kinds of nested classes in Java



can access non-static members

cannot access non-static class members

used on previous slides

ad-hoc sorting with **anonymous** inner class

class is used at one spot and it is not worthwhile to assign a name

ad-hoc sorting with **anonymous** inner class

class is used at one spot and it is not worthwhile to assign a name

```
private void run4() {
  Group g = new Group();
  Collections.sort(g.getList(), new Comparator<Person> () {
    @Override
    public int compare(Person p1, Person p2) {
      return p1.getId() - p2.getId();
  }});
  System.out.println("run4: " + g.getList());
}
```

ad-hoc sorting with **anonymous** inner class

class is used at one spot and it is not worthwhile to assign a name

```
private void run4() {
  Group g = new Group();
  Collections.sort(g.getList(), new Comparator<Person> () {
    @Override
    public int compare(Person p1, Person p2) {
      return p1.getId() - p2.getId();
  }});
  System.out.println("run4: " + g.getList());
}
```

ad-hoc sorting with **anonymous** inner class

class is used at one spot and it is not worthwhile to assign a name

interface or class

```
private void run4() {
  Group g = new Group();
  Collections.sort(g.getList(), new Comparator<Person> () {
    @Override
    public int compare(Person p1, Person p2) {
      return p1.getId() - p2.getId();
  }});
  System.out.println("run4: " + g.getList());
}
```

ad-hoc sorting with **anonymous** inner class

class is used at one spot and it is not worthwhile to assign a name

```
private void run4() {
  Group g = new Group();
  Collections.sort(g.getList(), new Comparator<Person> () {
    @Override
    public int compare(Person p1, Person p2) {
      return p1.getId() - p2.getId();
  }});
  System.out.println("run4: " + g.getList());
}
```

interface or class

all methods of class or interface.
Can have attributes

ad-hoc sorting with **anonymous** inner class

class is used at one spot and it is not worthwhile to assign a name

```
private void run4() {
  Group g = new Group();
  Collections.sort(g.getList(), new Comparator<Person> () {
    @Override
    public int compare(Person p1, Person p2) {
      return p1.getId() - p2.getId();
  }});
  System.out.println("run4: " + g.getList());
}
```

interface or class

all methods of class or interface.
Can have attributes

- this yields:

```
run4: [Bob (2), Carol (6), Alice (7), Dave (9)]
```

anonymous class definition

like a constructor followed by a class body

syntax of this ***expression***:

- new operator
- name of interface or class to extend
- arguments to the constructor,
  an interface has no constructor: use ()
- class declaration body: method definitions

anonymous class definition

like a constructor followed by a class body

syntax of this ***expression***:

- new operator
- name of interface or class to extend
- arguments to the constructor,
  an interface has no constructor: use ()
- class declaration body: method definitions

useful for classes that are only needed at one place

- you make exactly one instance of this class,
  each time the expression is evaluated

anonymous classes can *capture variables*:

- access to all attributes of enclosing class

anonymous class definition

like a constructor followed by a class body

syntax of this ***expression***:

- new operator
- name of interface or class to extend
- arguments to the constructor,
  an interface has no constructor: use ()
- class declaration body: method definitions

useful for classes that are only needed at one place

- you make exactly one instance of this class,
  each time the expression is evaluated

anonymous classes can *capture variables*:

- access to all attributes of enclosing class

ad-hoc sorting with **lambda-expression**

Alternatively, if there is a single method in an anonymous class it is sufficient if we define only that method

```
private void run5() {
  Group g = new Group();
  Collections.sort(g.getList(), (p1, p2) -> p1.getId() - p2.getId(););
  System.out.println("run5: " + g);
}
```

ad-hoc sorting with **lambda-expression**

Alternatively, if there is a single method in an anonymous class it is sufficient if we define only that method

```
private void run5() {
  Group g = new Group();
  Collections.sort(g.getList(), (p1, p2) -> p1.getId() - p2.getId(););
  System.out.println("run5: " + g);
}
```

ad-hoc sorting with **lambda-expression**

Alternatively, if there is a single method in an anonymous class it is sufficient if we define only that method

2nd arg of `sort`: this must be a `Comparator` instance

```
private void run5() {
  Group g = new Group();
  Collections.sort(g.getList(), (p1, p2) -> p1.getId() - p2.getId(););
  System.out.println("run5: " + g);
}
```

ad-hoc sorting with **lambda-expression**

Alternatively, if there is a single method in an anonymous class it is sufficient if we define only that method

2nd arg of `sort`: this must be a `Comparator` instance

```
private void run5() {
  Group g = new Group();
  Collections.sort(g.getList(), (p1, p2) -> p1.getId() - p2.getId(););
  System.out.println("run5: " + g);
}
```

- this yields:

```
run5: [Dave (9), Carol (6), Bob (2), Alice (7)]
```

ad-hoc sorting with **lambda-expression**

Alternatively, if there is a single method in an anonymous class it is sufficient if we define only that method

2nd arg of `sort`: this must be a `Comparator` instance

```
private void run5() {
  Group g = new Group();
  Collections.sort(g.getList(), (p1, p2) -> p1.getId() - p2.getId(););
  System.out.println("run5: " + g);
}
```

reverses order

- this yields:

```
run5: [Dave (9), Carol (6), Bob (2), Alice (7)]
```

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

list of parameters

- if there is only 1 parameter you can omit parentheses
- you can omit the types of the parameters

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

list of parameters

- if there is only 1 parameter you can omit parentheses
- you can omit the types of the parameters

the arrow token `->`

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

list of parameters

- if there is only 1 parameter you can omit parentheses
- you can omit the types of the parameters

the arrow token `->`

body

- single expression
  - does not need statement braces { and }
  - does not need the `return` keyword

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

list of parameters

- if there is only 1 parameter you can omit parentheses
- you can omit the types of the parameters

the arrow token `->`

```
(x, y) -> x.compareTo(y)
```

body

- single expression
  - does not need statement braces { and }
  - does not need the `return` keyword

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

list of parameters

- if there is only 1 parameter you can omit parentheses
- you can omit the types of the parameters

the arrow token `->`

```
(x, y) -> x.compareTo(y)
```

body

- single expression
  - does not need statement braces { and }
  - does not need the `return` keyword
- statement block
  - needs statement braces { and }
  - as many statements as you need, separated by ;

syntax of lambda expressions

works only if we need exactly 1 method: functional interface

- context should identify which class/interface is needed

list of parameters

- if there is only 1 parameter you can omit parentheses
- you can omit the types of the parameters

the arrow token `->`

```
(x, y) -> x.compareTo(y)
```

body

- single expression
  - ➢ does not need statement braces { and }
  - ➢ does not need the `return` keyword
- statement block
  - ➢ needs statement braces { and }
  - ➢ as many statements as you need, separated by ;

```
Person p -> {
  int id = p.getId();
  return id % 3 == 0;
}
```

more lambda expressions 1/3

```
public static <T> List<T> filter (List<T> list, Predicate<T> p) {
  List<T> res = new LinkedList<> ();
  for (T t: list) {
    if (p.test(t)) {
      res.add(t);
    }
  }
  return res;
}
```

- using `Predicate` from the Java platform

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

more lambda expressions 2/3

```
private void run6 () {
  Group g = new Group();
  List<Person> l = filter(g.getList(), (Person p) -> {
    int id = p.getId();
    return id % 3 == 0;
  });
  System.out.println("run6: " + l);
}
```

more lambda expressions 2/3

```
private void run6 () {
  Group g = new Group();
  List<Person> l = filter(g.getList(), (Person p) -> {
    int id = p.getId();
    return id % 3 == 0;
  });
  System.out.println("run6: " + l);
}
```

`test` of the `Predicate` interface.
Yes this can in one expression

more lambda expressions 2/3

```java
private void run6 () {
  Group g = new Group();
  List<Person> l = filter(g.getList(), (Person p) -> {
    int id = p.getId();
    return id % 3 == 0;
  });
  System.out.println("run6: " + l);
}
```

`test` of the `Predicate` interface.
Yes this can in one expression

- this yields:

```
run6: [Dave (9), Carol (6)]
```

more lambda expressions 2/3

```
private void run7 () {
  Group g = new Group();
  List<Person> l = filter(g.getList(), p -> p.getId() > 4);
  System.out.println("run7: " + l);
}
```

more lambda expressions 2/3

```
private void run7 () {
  Group g = new Group();
  List<Person> l = filter(g.getList(), p -> p.getId() > 4);
  System.out.println("run7: " + l);
}
```

- this yields:

```
run7: [Alice (7), Dave (9), Carol (6)]
```

USER DEFINED GENERIC DATA TYPE: ARRAY-BASED LIST

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

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

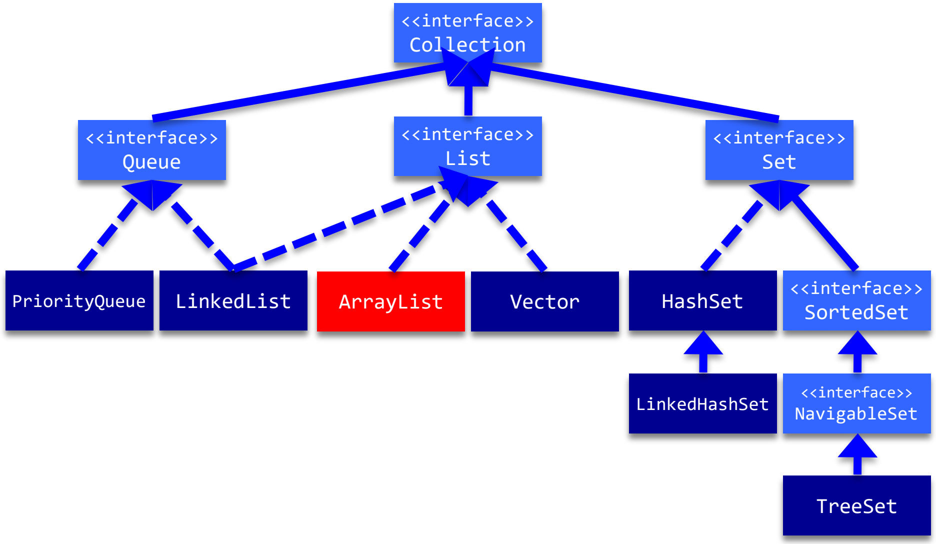
unless you have a very good reason for it

MyArrayList

store elements in an array

implement the `List` interface

- + find elements fast O(1)
- - inserting/deleting elements is expensive O(N)

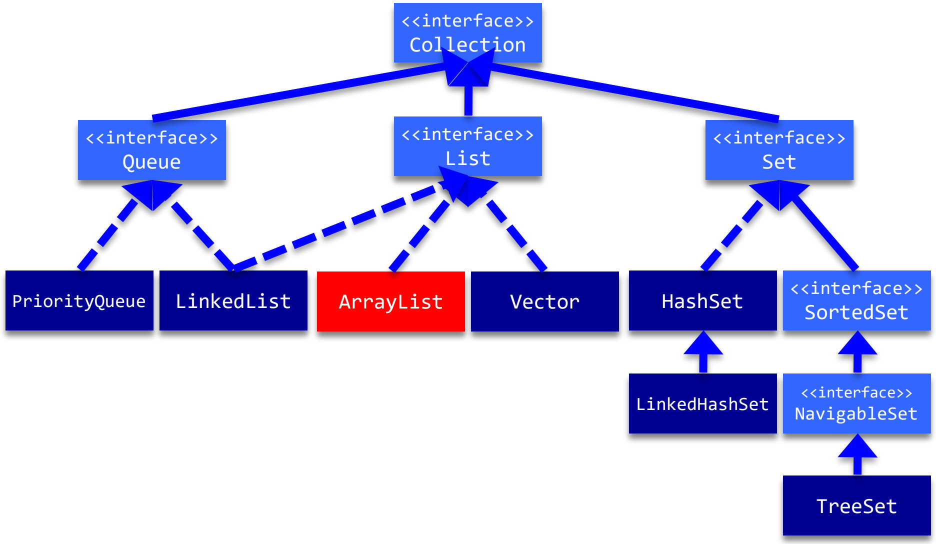


MyArrayList

store elements in an array

implement the `List` interface

+ find elements fast O(1)
- inserting/deleting elements is expensive O(N)



we cannot predict the size of the list

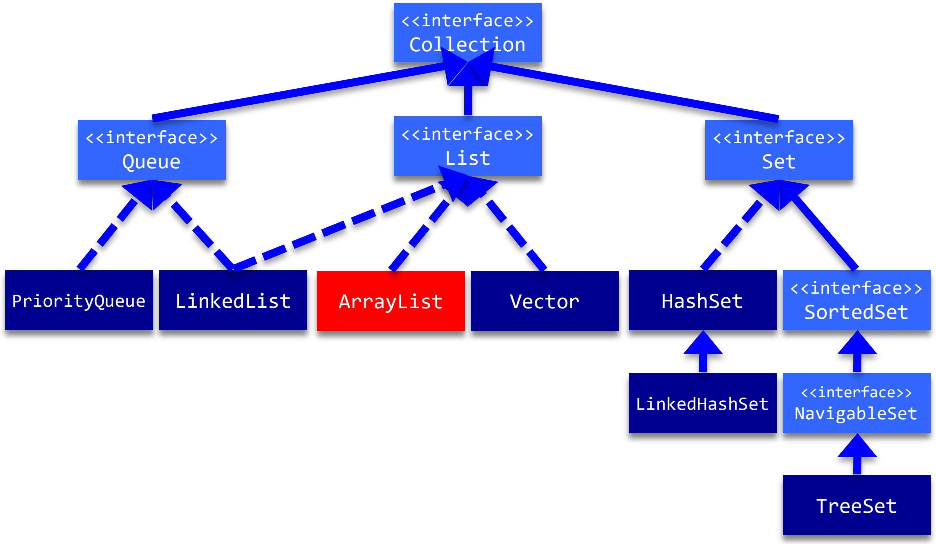
- there is no upper bound
- start with a small array
- allocate a bigger array when the current array is full & copy all elements: O(N)
  this is done once every N additions: amortized O(1)

MyArrayList

store elements in an array

implement the `List` interface

- + find elements 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: attributes & constructor

```
public class MyArrayList<E> extends List<E> {
   private int size = 0;       // current number of elements in list
   private E[] data;           // array containing the elements
   private int modCount = 0;   // number of changes of this list
```

MyArrayList: attributes & constructor

```
public class MyArrayList<E> extends List<E> {
   private int size = 0;      // current number of elements in list
   private E[] data;          // array containing the elements
   private int modCount = 0;  // number of changes of this list

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

MyArrayList: attributes & constructor

```
public class MyArrayList<E> extends List<E> {
    private int size = 0;        //  current number of elements in list
    private E[] data;            //  array containing the elements
    private int modCount = 0;    //  number of changes of this list

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

type cast: we have no constructor for E[ ]

MyArrayList: attributes & constructor

```
public class MyArrayList<E> extends List<E> {
   private int size = 0;       // current number of elements in list
   private E[] data;           // array containing the elements
   private int modCount = 0;   // number of changes of this list

   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;
}
```

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

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

    checkBoundExclusive(index);

    return data[index];
}
```

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

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

O(1)

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

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

O(1)

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

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

O(1)

O(1)

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

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

O(1)

worst case: O(N),
amortized: O(1)

O(1)

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

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

O(1)

worst case: O(N),
amortized: O(1)

O(1)

interface requires this

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

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

O(1)

count modifications

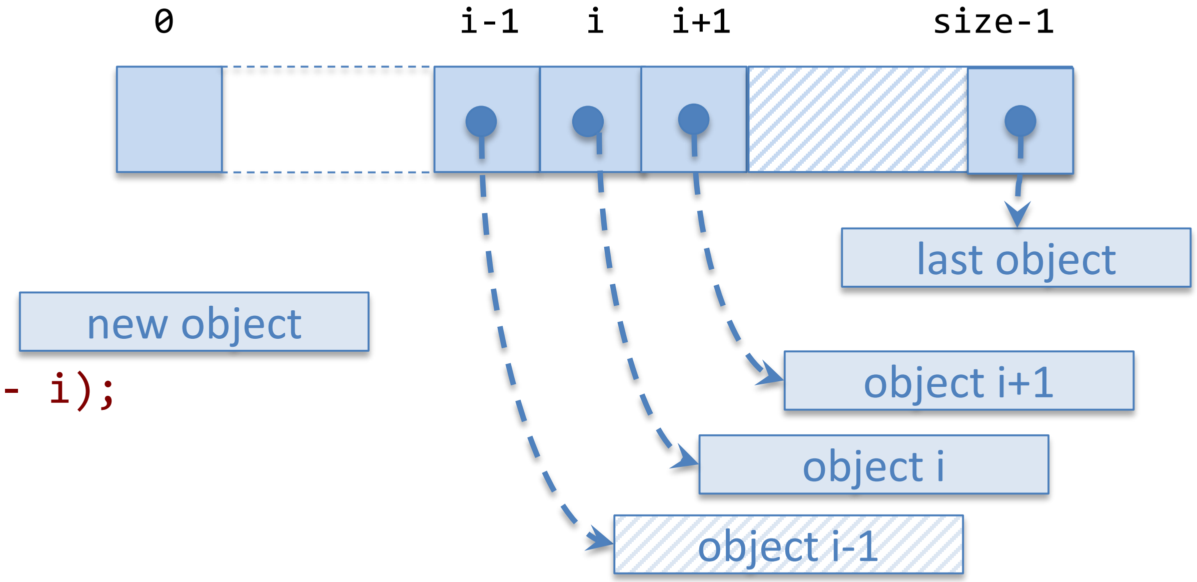
worst case: O(N), amortized: O(1)

O(1)

interface requires this

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

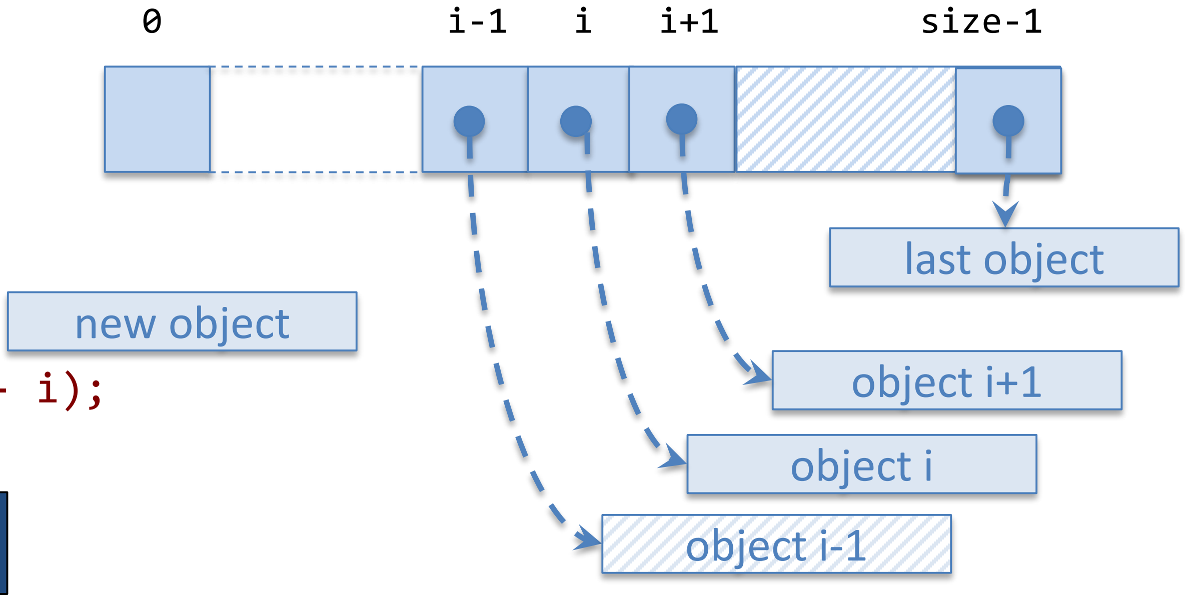
```
@Override
public void add(int i, E e) {
    checkBoundInclusive(i);
    modCount += 1;
    ensureCapacity(size + 1);
    System.arraycopy(data, i, data, i + 1, size - i);
    data[i] = e;
    size += 1;
}
```



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

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

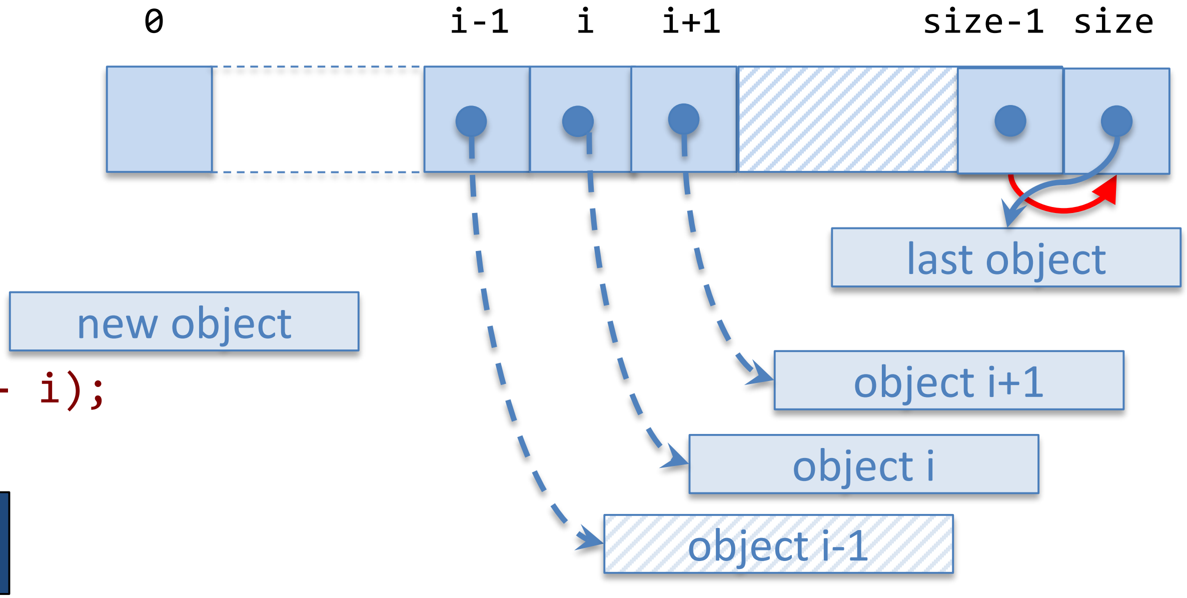
make space: O(N)



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

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

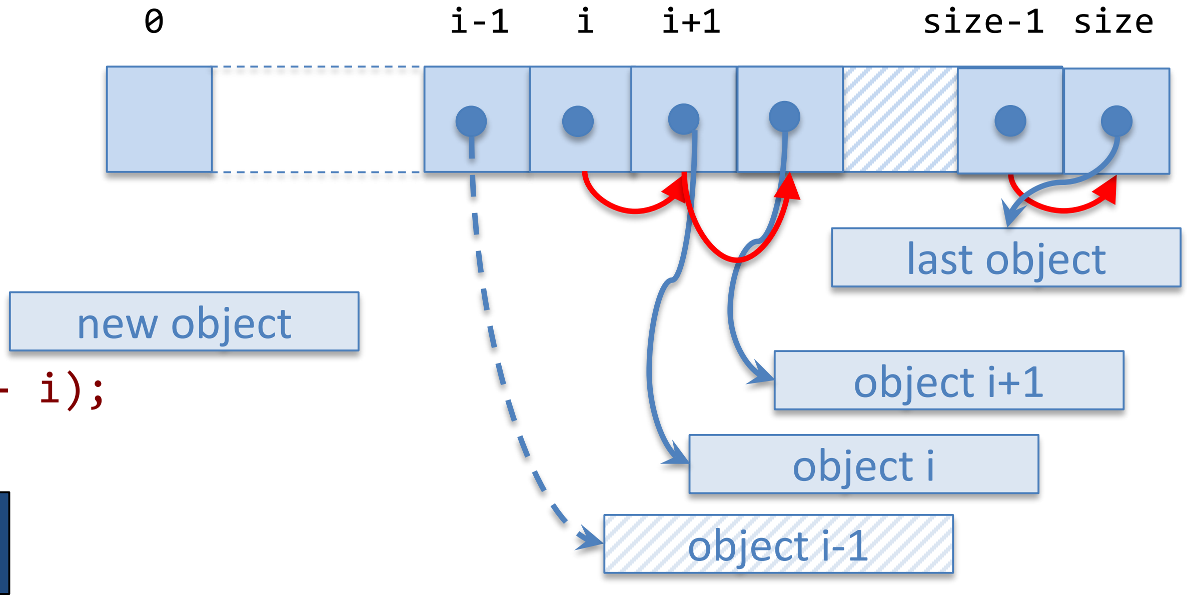
make space: O(N)



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

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

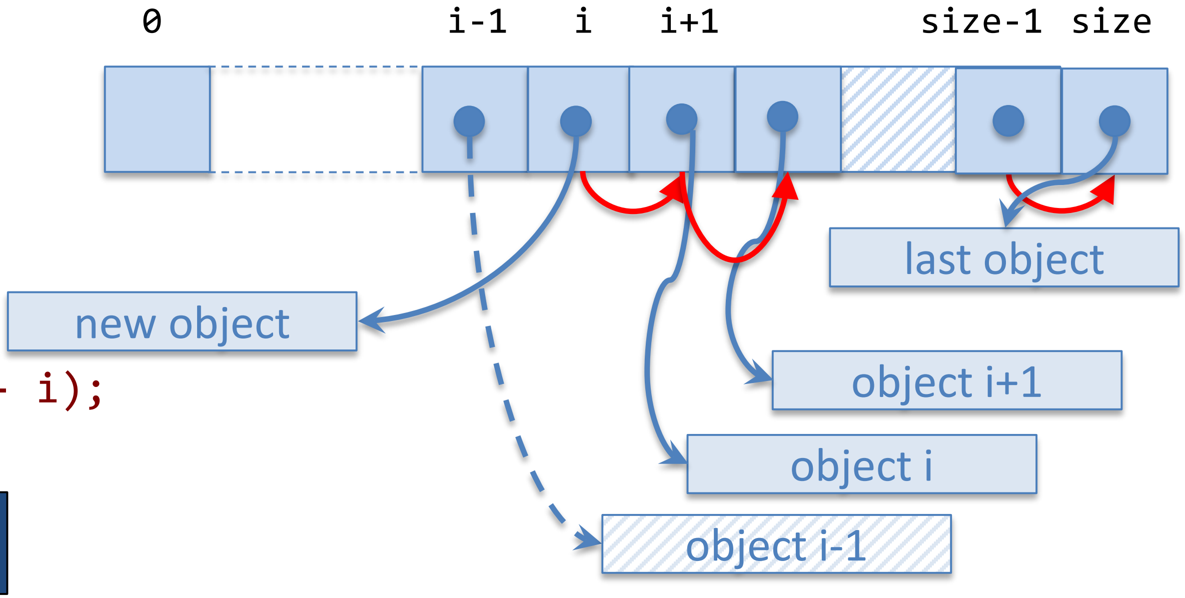
make space: O(N)



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

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

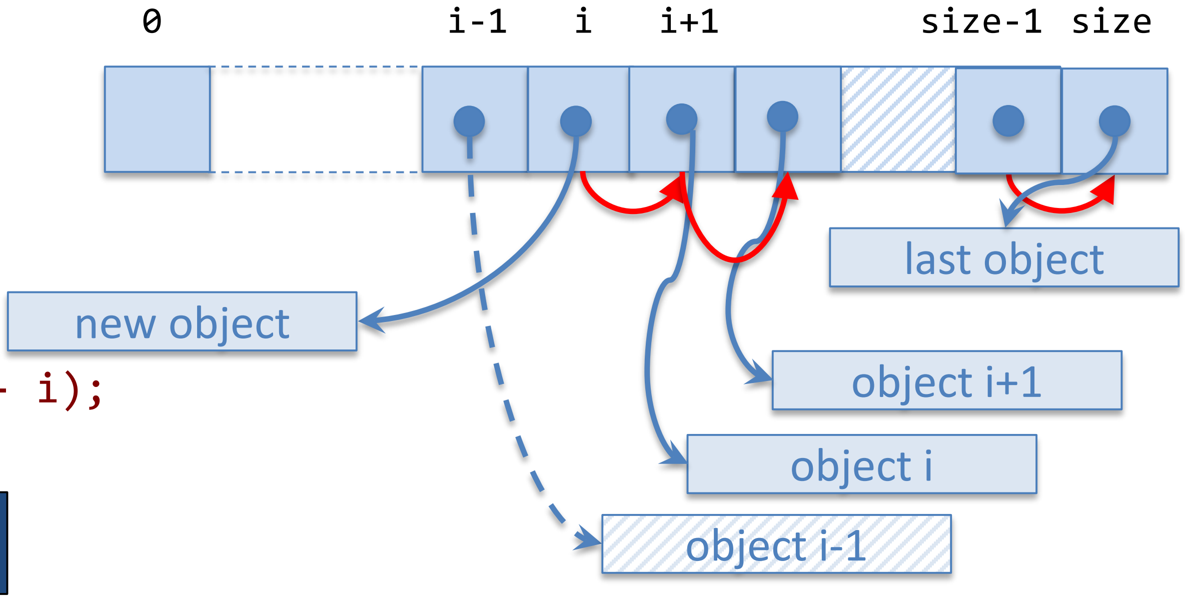
make space: O(N)



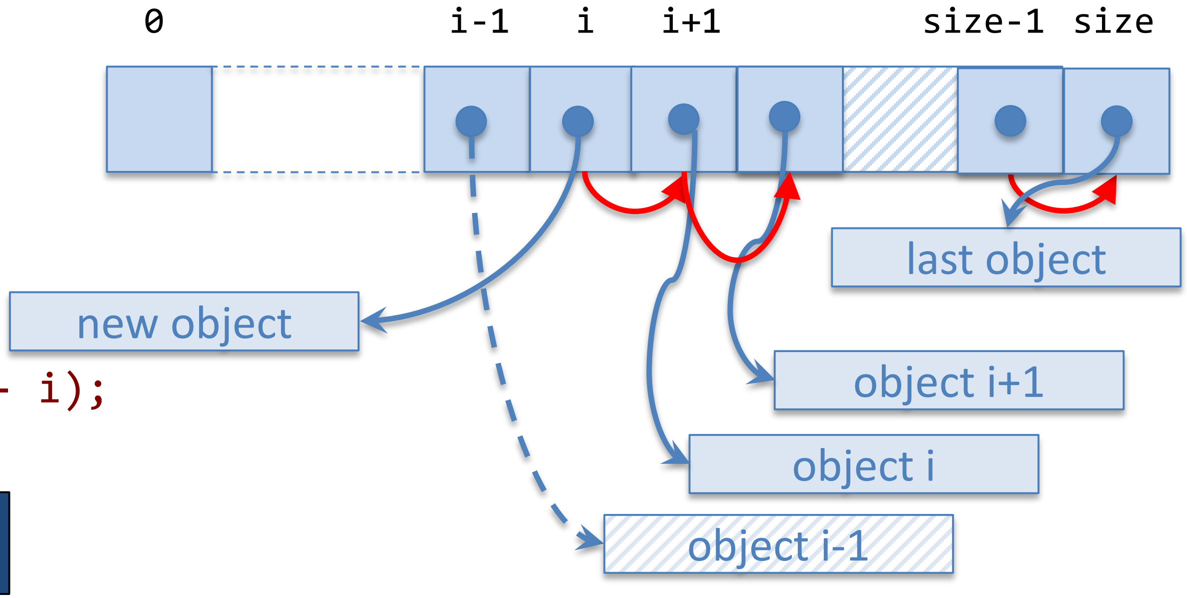
MyArrayList: `add(index, element)`, `ensureCapacity`

```
@Override
public void add(int i, E e) {
    checkBoundInclusive(i);
    modCount += 1;
    ensureCapacity(size + 1);
    System.arraycopy(data, i, data, i + 1, size - i);
    data[i] = e;
    size += 1;
}
private void ensureCapacity(int c) {
  if (c > data.length) {
    E[] es = (E[]) new Object[Math.max(data.length * 2, c)];
    System.arraycopy(data, 0, es, 0, size);
    data = es;
  }
}
```

make space: O(N)



MyArrayList: add(index, element), ensureCapacity



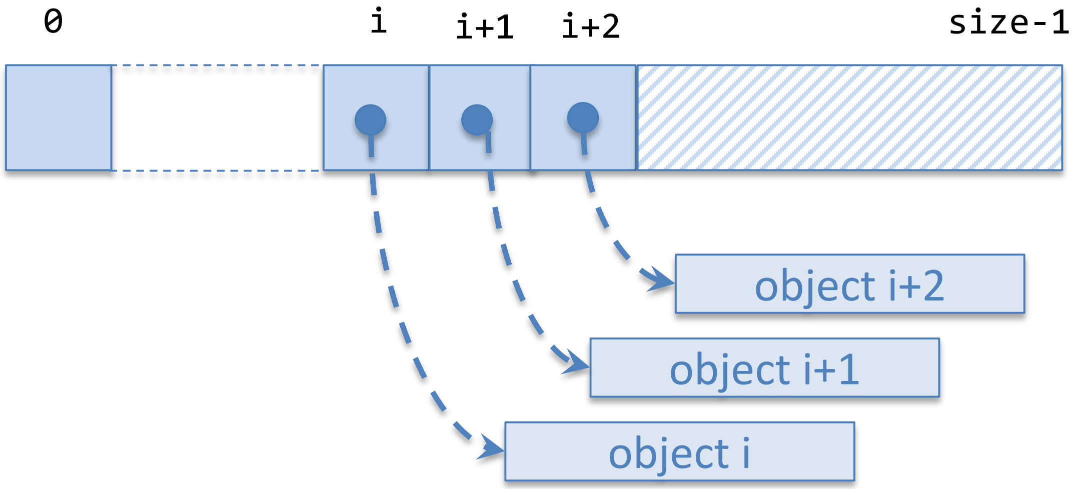
```
@Override
public void add(int i, E e) {
    checkBoundInclusive(i);
    modCount += 1;
    ensureCapacity(size + 1);
    System.arraycopy(data, i, data, i + 1, size - i);
    data[i] = e;
    size += 1;
}
private void ensureCapacity(int c) {
  if (c > data.length) {
    E[] es = (E[]) new Object[Math.max(data.length * 2, c)];
    System.arraycopy(data, 0, es, 0, size);
    data = es;
  }
}
```

make space: O(N)

once every N additions, amortized O(1)

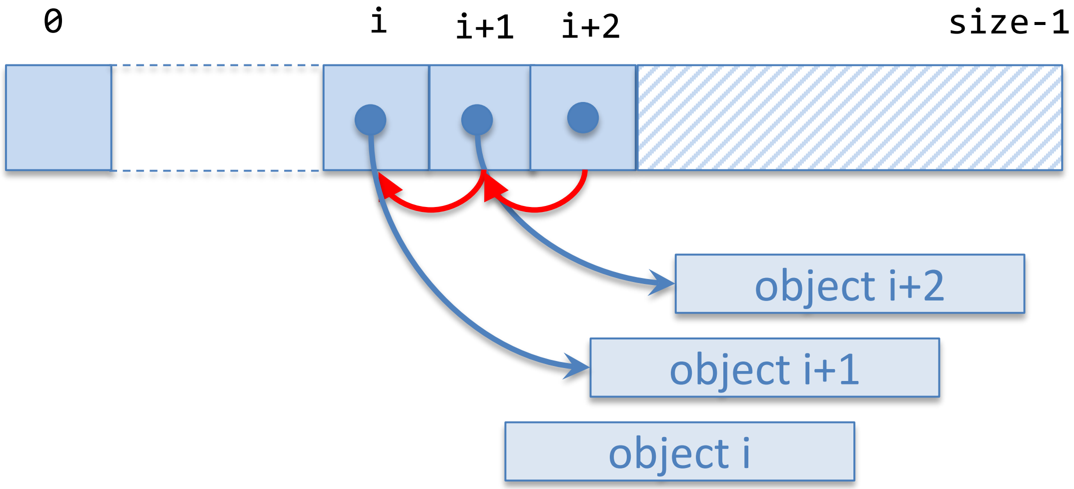
MyArrayList: remove(index), checkBounds

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



MyArrayList: remove(index), checkBounds

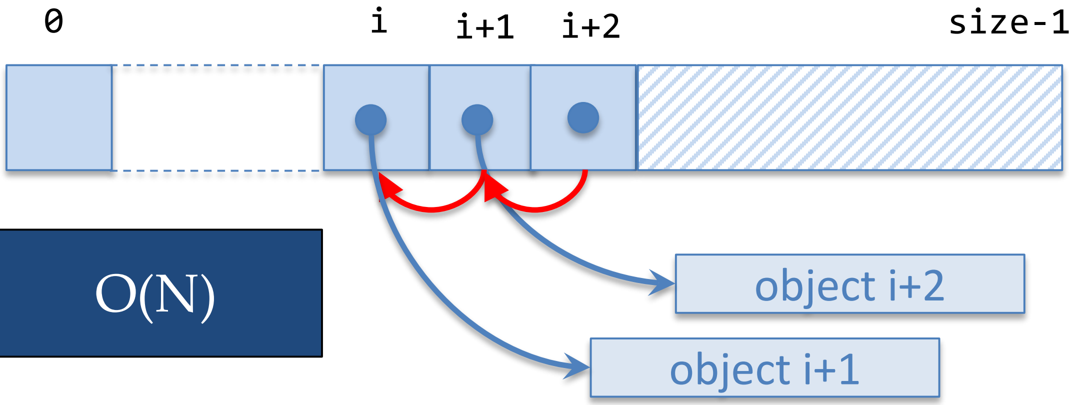
```
@Override
public E remove(int i) {
    checkBoundExclusive(i);
    modCount += 1;
    E r = data[i];
    size -= 1;
    System.arraycopy(data, i + 1, data, i, size - i);
    data[size] = null;
    return r;
}
```



MyArrayList: remove(index), checkBounds

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

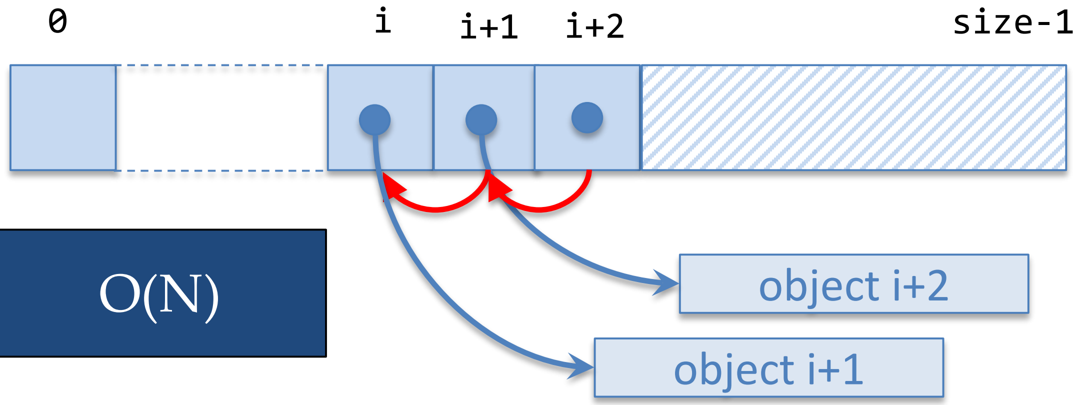
O(N)



MyArrayList: remove(index), checkBounds

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

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

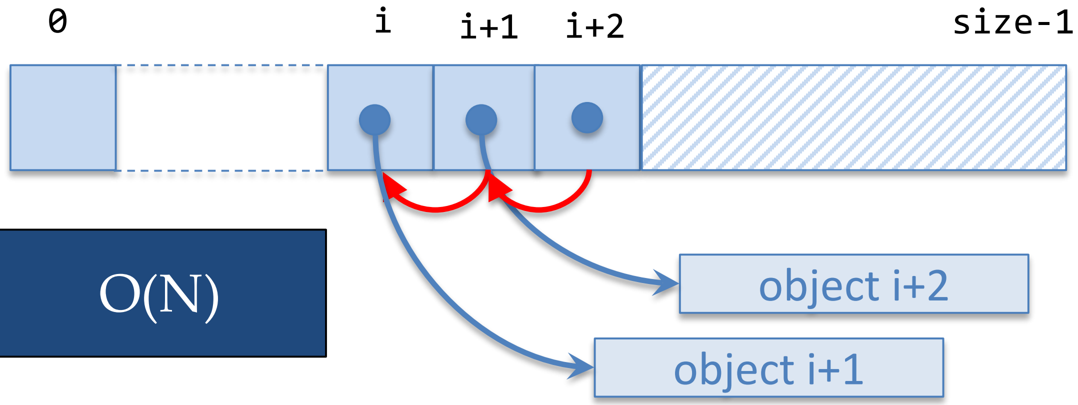
O(N)



MyArrayList: remove(index), checkBounds

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

O(N)



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

MyArrayList: toString

```
@Override
public String toString () {
    StringBuilder sb = new StringBuilder();
    sb.append("{");
    if (size > 0) {
       sb.append(data[0]);
       for (int i = 1; i < size - 1; i += 1) {
          sb.append(", ").append(data[i]);
       }
    }
    sb.append("}");
    return sb.toString();
}
```

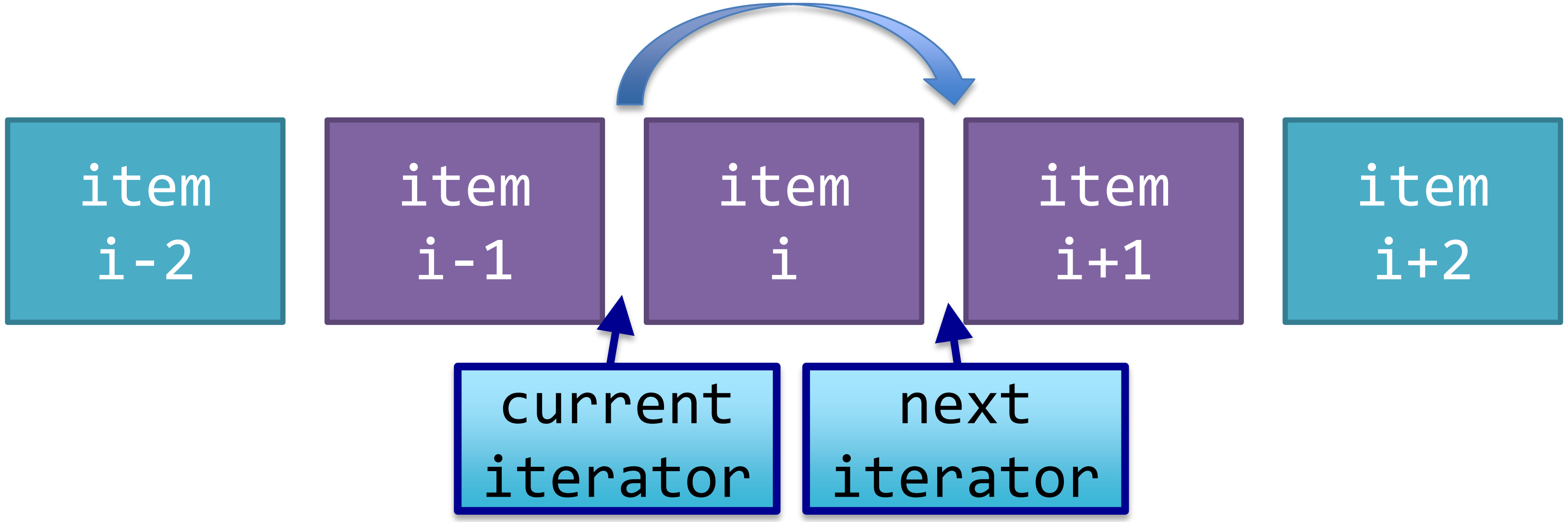
MyArrayList: Iterator<E> iterator()

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

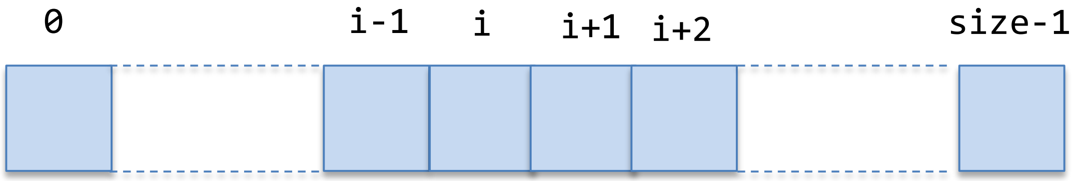
checks if there is a next item

returns next item

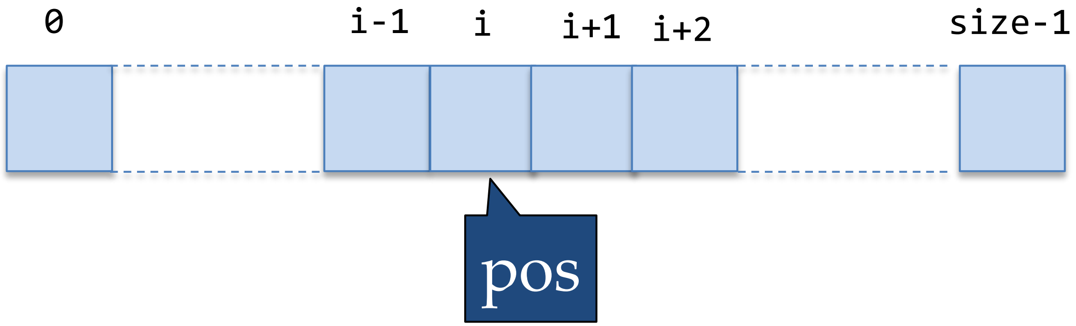
removes last item returned



MyArrayList: iterator

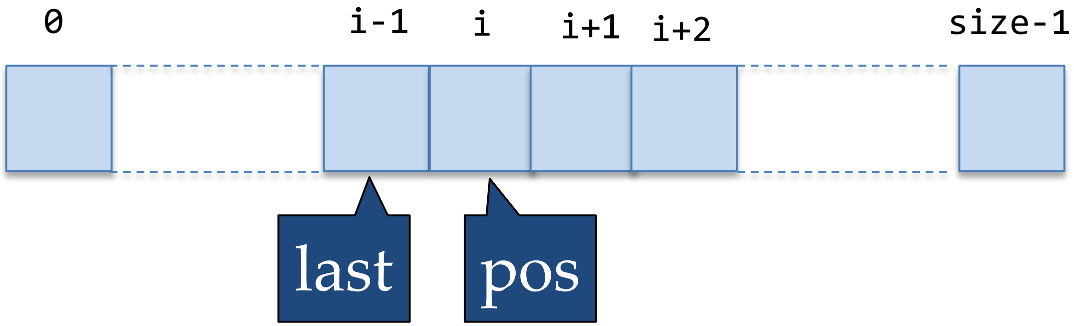


MyArrayList: iterator



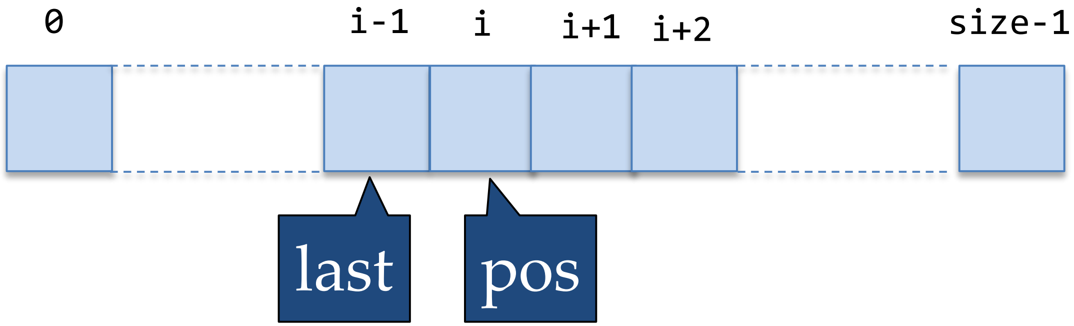
- pos: index of next object

MyArrayList: iterator



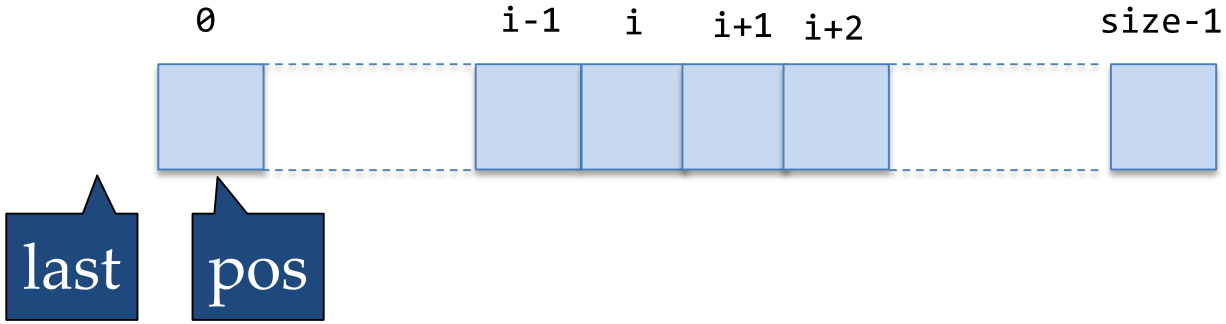
- `pos`: index of next object
- `last`: index of previous object; -1 if there is no previous

MyArrayList: iterator



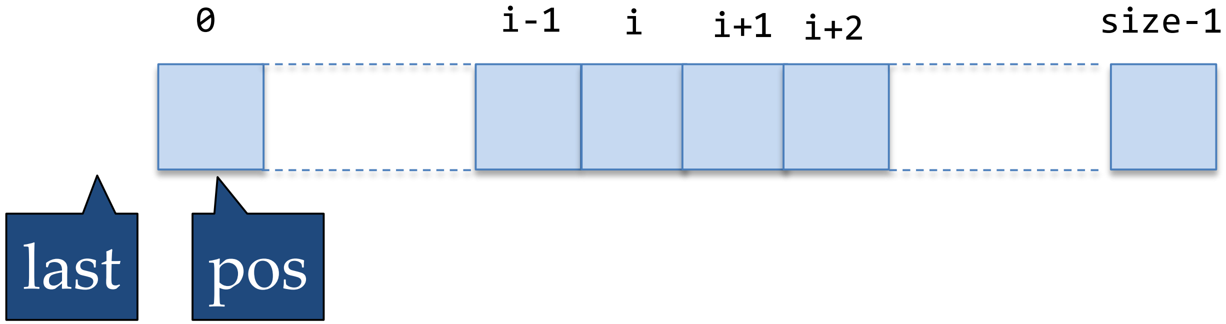
- `pos`: index of next object
- `last`: index of previous object; -1 if there is no previous
- `knownMod`: known value of modification counter of `MyArrayList`

MyArrayList: iterator



```
@Override
public Iterator<E> iterator() {
  return new Iterator<E>() {
    private int pos = 0, last = -1, knownMod = modCount;
```

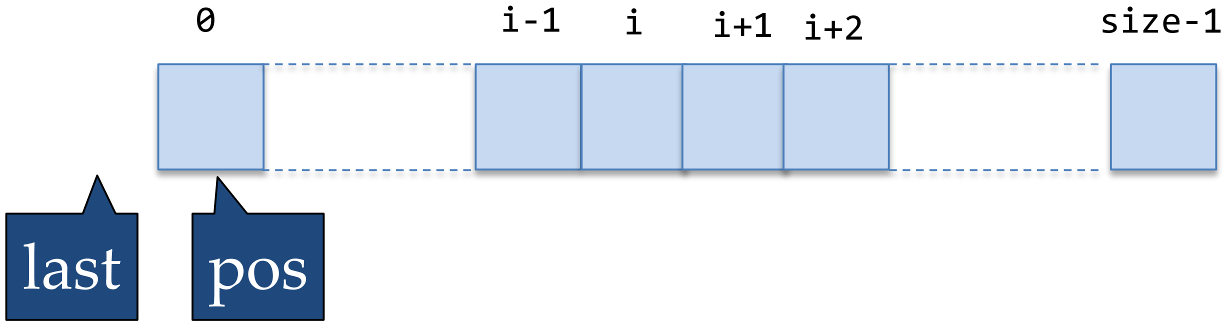
MyArrayList: iterator



```
@Override
public Iterator<E> iterator() {
  return new Iterator<E>() {
    private int pos = 0, last = -1, knownMod = modCount;
```

method-local inner class

MyArrayList: iterator

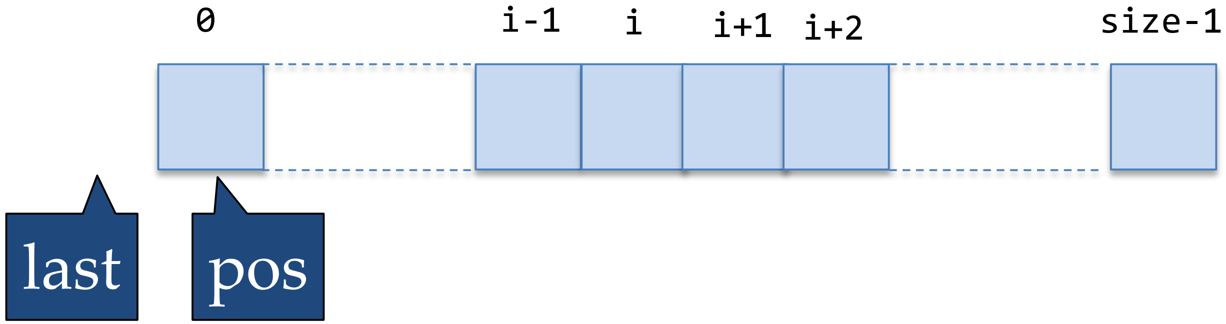


```
@Override
public Iterator<E> iterator() {
  return new Iterator<E>() {
    private int pos = 0, last = -1, knownMod = modCount;
```

method-local inner class

last element given by next()

MyArrayList: iterator

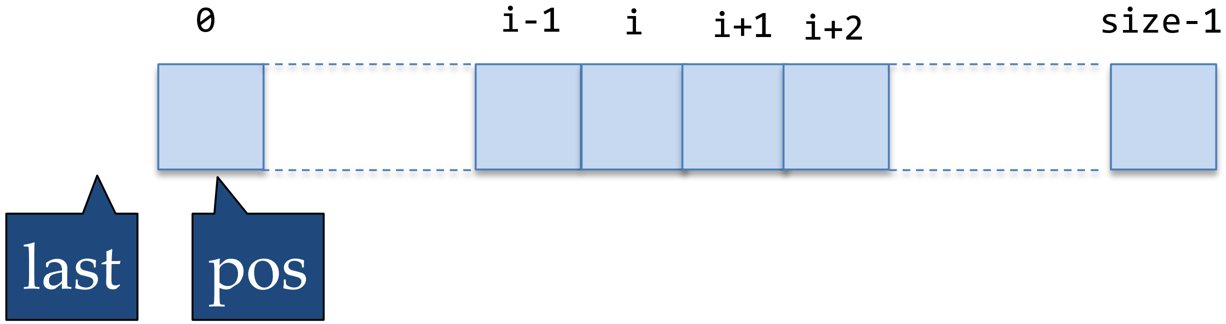


```
@Override
public Iterator<E> iterator() {
  return new Iterator<E>() {
    private int pos = 0, last = -1, knownMod = modCount;

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

method-local inner class

last element given by next()

MyArrayList: iterator



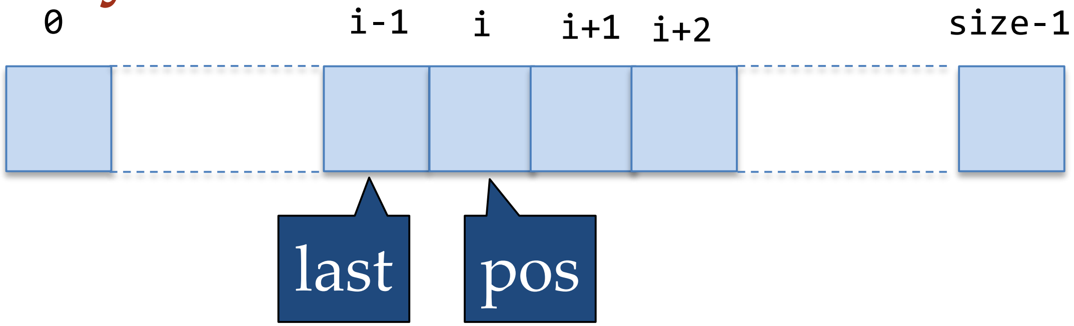
```
@Override
public Iterator<E> iterator() {
  return new Iterator<E>() {
    private int pos = 0, last = -1, knownMod = modCount;

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

method-local inner class

last element given by `next()`

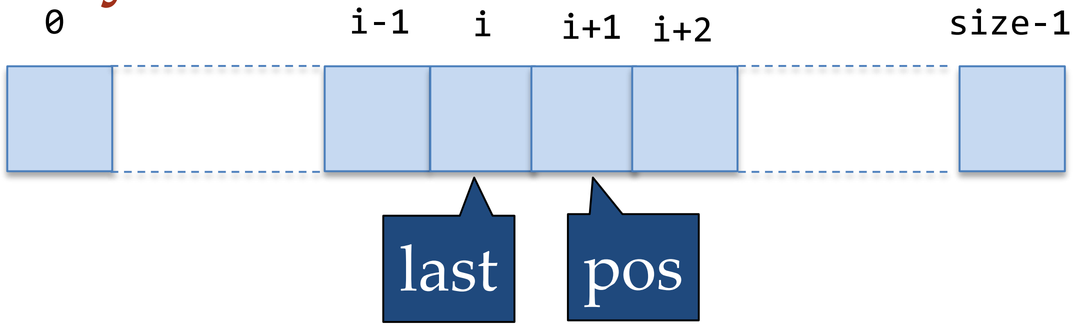
attribute of `MyArrayList`

MyArrayList: iterator



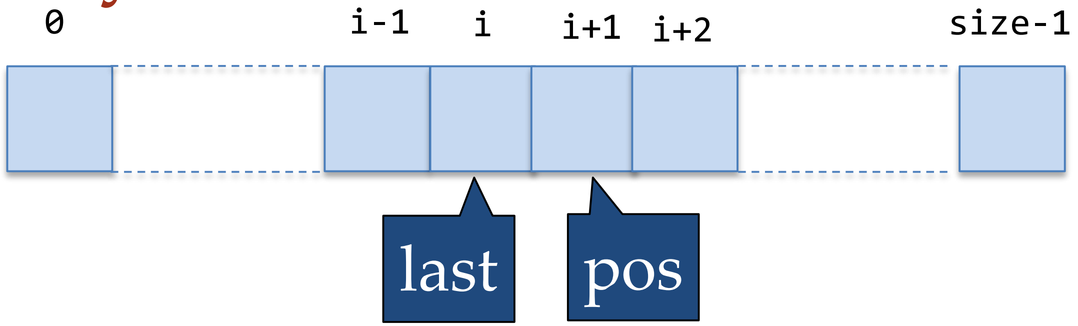
```
@Override
public E next() {
    checkVersion();
    if (pos >= size)
        throw new NoSuchElementException();
    else {
        last = pos;
        pos += 1;
        return data[ last ] ;
    }
}
```

MyArrayList: iterator



```
@Override
public E next() {
    checkVersion();
    if (pos >= size)
        throw new NoSuchElementException();
    else {
        last = pos;
        pos += 1;
        return data[ last ] ;
    }
}
```

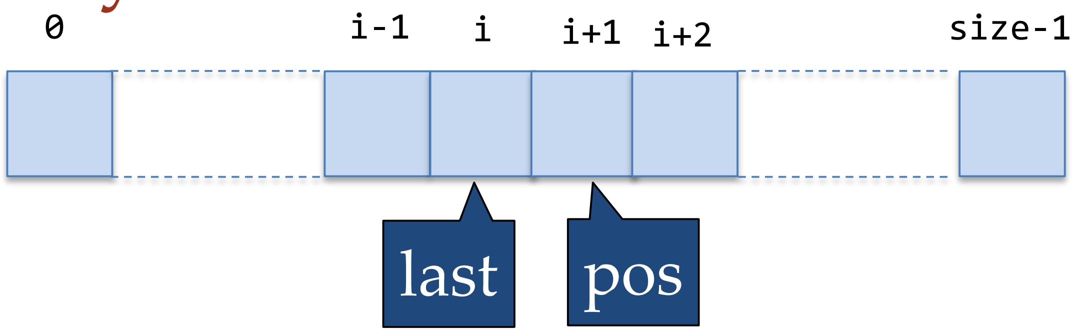
MyArrayList: iterator



```
@Override
public E next() {
    checkVersion();
    if (pos >= size)
        throw new NoSuchElementException();
    else {
        last = pos;
        pos += 1;
        return data[ last ] ;
    }
}
```

O(1)

MyArrayList: iterator



```
@Override
public E next() {
    checkVersion();
    if (pos >= size)
        throw new NoSuchElementException();
    else {
        last = pos;
        pos += 1;
        return data[ last ] ;
    }
}
private void checkVersion() {
    if (knownMod != modCount)
        throw new ConcurrentModificationException();
}
```

O(1)

MyArrayList: iterator

```
  @Override
  public void remove () {
      checkVersion();
      if (last < 0)
          throw new IllegalStateException();
      else {
          MyArrayList.this.remove( last );
          last = -1;
          pos  -= 1;
          knownMod = modCount;
      }
  }

};
```

MyArrayList: iterator

```
    @Override
    public void remove () {
        checkVersion();
        if (last < 0)
            throw new IllegalStateException();
        else {
            MyArrayList.this.remove( last );
            last = -1;
            pos  -= 1;
            knownMod = modCount;
        }
    }

};
```

remove from MyArrayList O(N)

I code better at night
11:00 PM
11:05 PM

JOKE OF THE WEEK

MyArrayList: evaluation

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

simple and works fine in many situations

MyArrayList: evaluation

`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

MyArrayList: evaluation

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

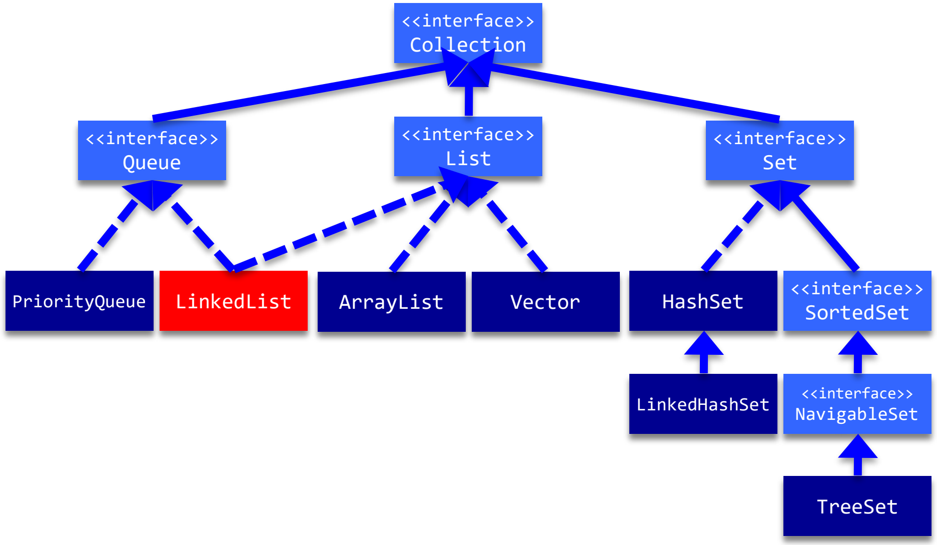
simple and works fine in many situations

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?

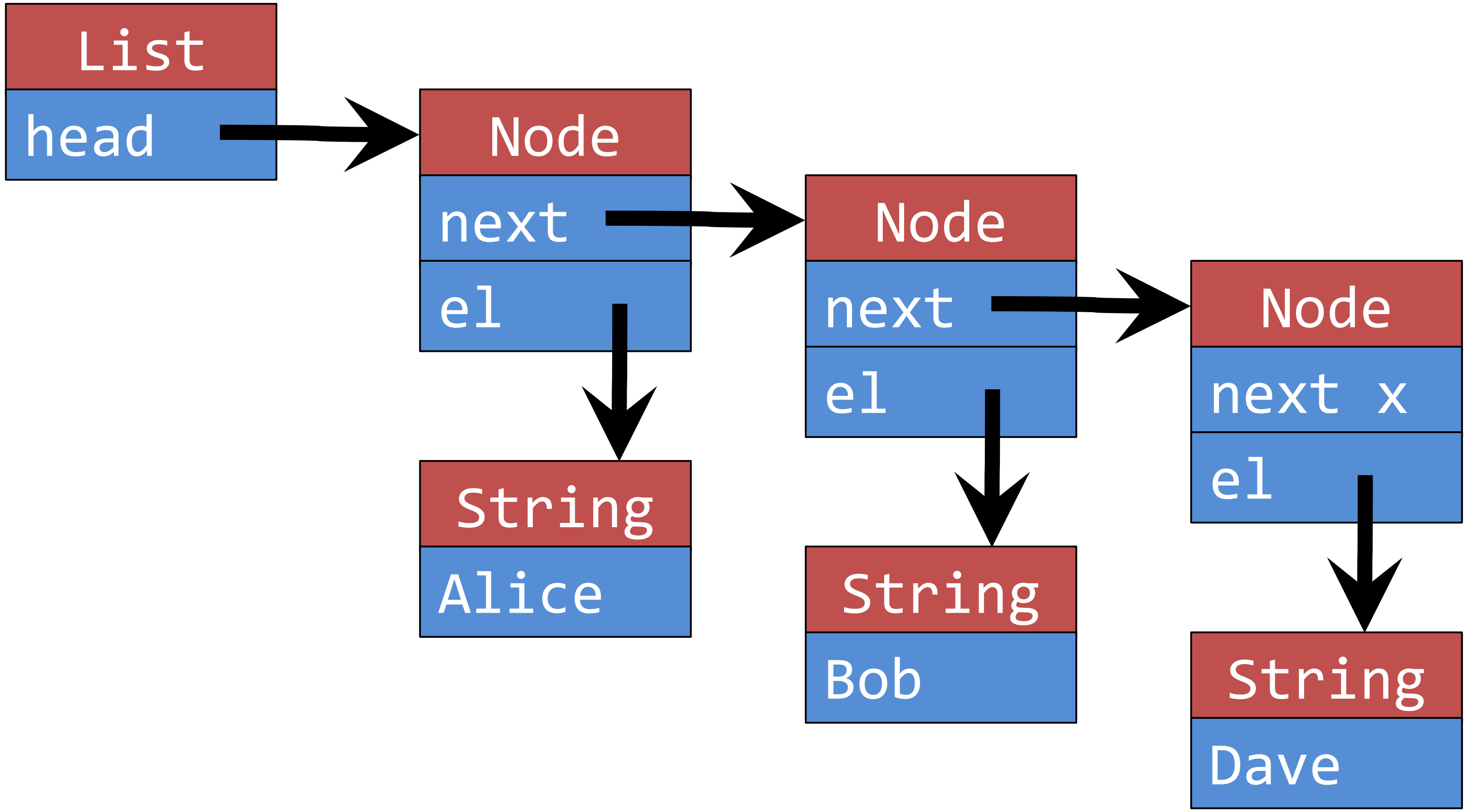
- use a linked data structure (recursive data structure)



GENERIC *RECURSIVE* DATA TYPE: LINKED LIST

Linked List

basic idea:

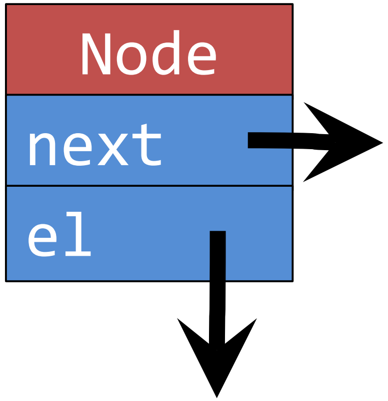


MyLinkedList: constructor

```
public class MyLinkedList<E> extends List<E> {

    private int size;
    private Node head, tail;
```

MyLinkedList: constructor

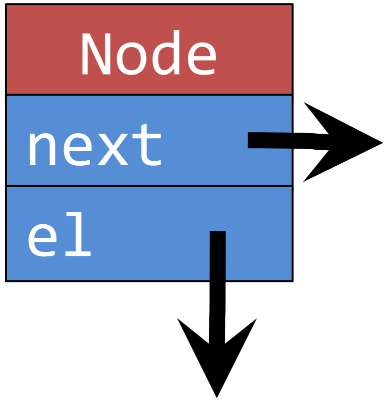
```
public class MyLinkedList<E> extends List<E> {

    private int size;
    private Node head, tail;

    private class Node {
        public E el;
        public Node next;
        public Node (E e, Node n) {
            el = e;
            next = n;
        }
        public Node (E e) {
            this(e, null);
        }
    }
```



MyLinkedList: constructor

```
public class MyLinkedList<E> extends List<E> {

    private int size;
    private Node head, tail;

    private class Node {
        public E el;
        public Node next;
        public Node (E e, Node n) {
            el = e;
            next = n;
        }
        public Node (E e) {
            this(e, null);
        }
    }
```

Node is private for MyLinkedList, attributes can be public



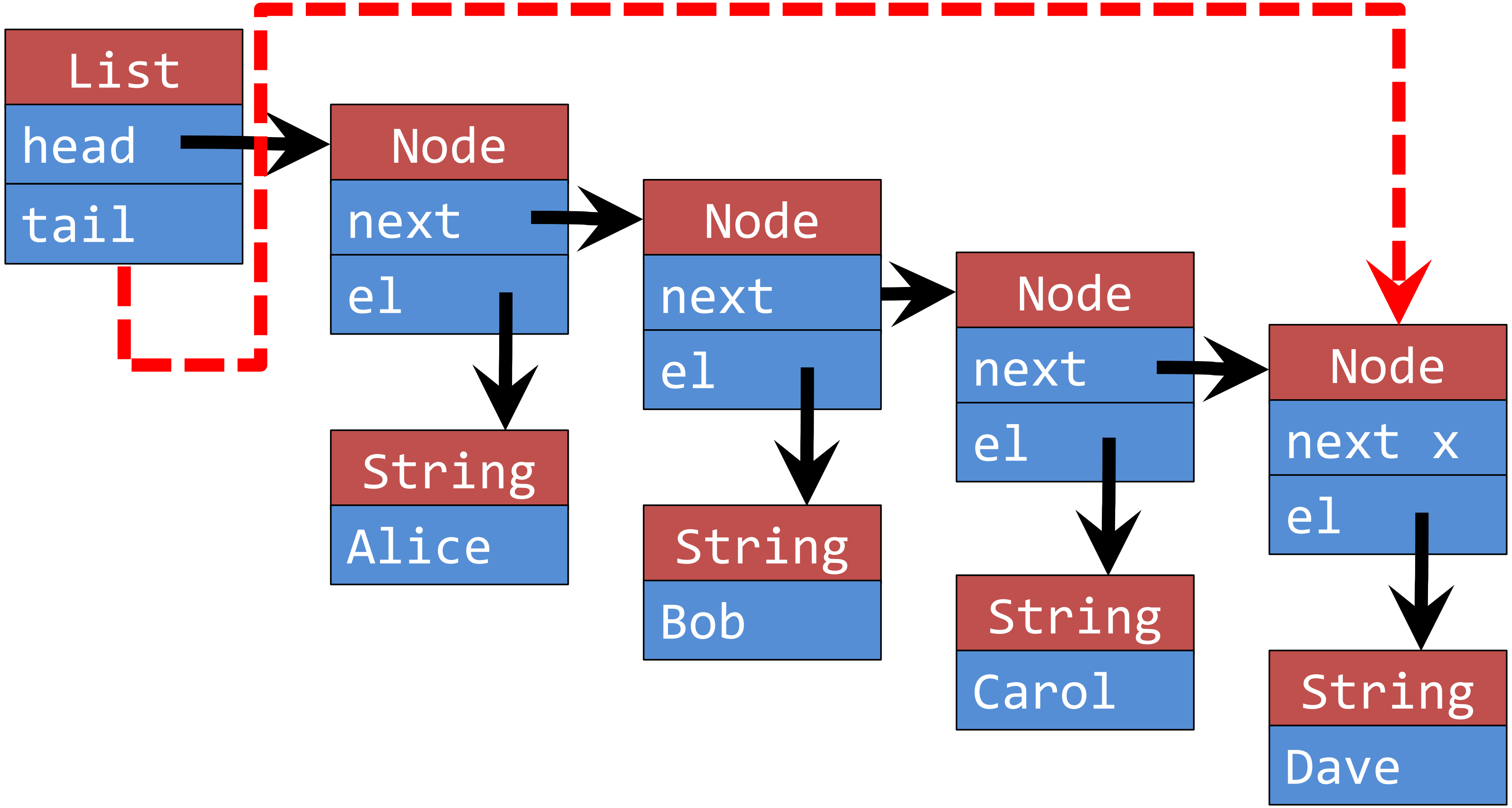
Node

next

el

Linked List: get(i)

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



MyLinkedList: get(index)

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

MyLinkedList: get(index)

```
@Override
public E get(int index) {
    return getNode(index).el;
}
private Node getNode(int index) {
    checkBound(index);
    Node n = head;
    for (int i = 0; i < index; i += 1)
        n = n.next;
    return n;
}
```

MyLinkedList: get(index)

```
@Override
public E get(int index) {
    return getNode(index).el;
}
private Node getNode(int index) {
    checkBound(index);
    Node n = head;
    for (int i = 0; i < index; i += 1)
        n = n.next;
    return n;
}
```

O(N)

MyLinkedList: get(index)

```
@Override
public E get(int index) {
    return getNode(index).el;
}
private Node getNode(int index) {
    checkBound(index);
    Node n = head;
    for (int i = 0; i < index; i += 1)
        n = n.next;
    return n;
}
```

O(N)

O(N)

MyLinkedList: get(index)

```
@Override
public E get(int index) {
    return getNode(index).el;
}
private Node getNode(int index) {
    checkBound(index);
    Node n = head;
    for (int i = 0; i < index; i += 1)
        n = n.next;
    return n;
}
private void checkBound(int i) {
  if (i < 0 || i >= size)
    throw new IndexOutOfBoundsException("Index: " + i + ", size: " + size);
}
```

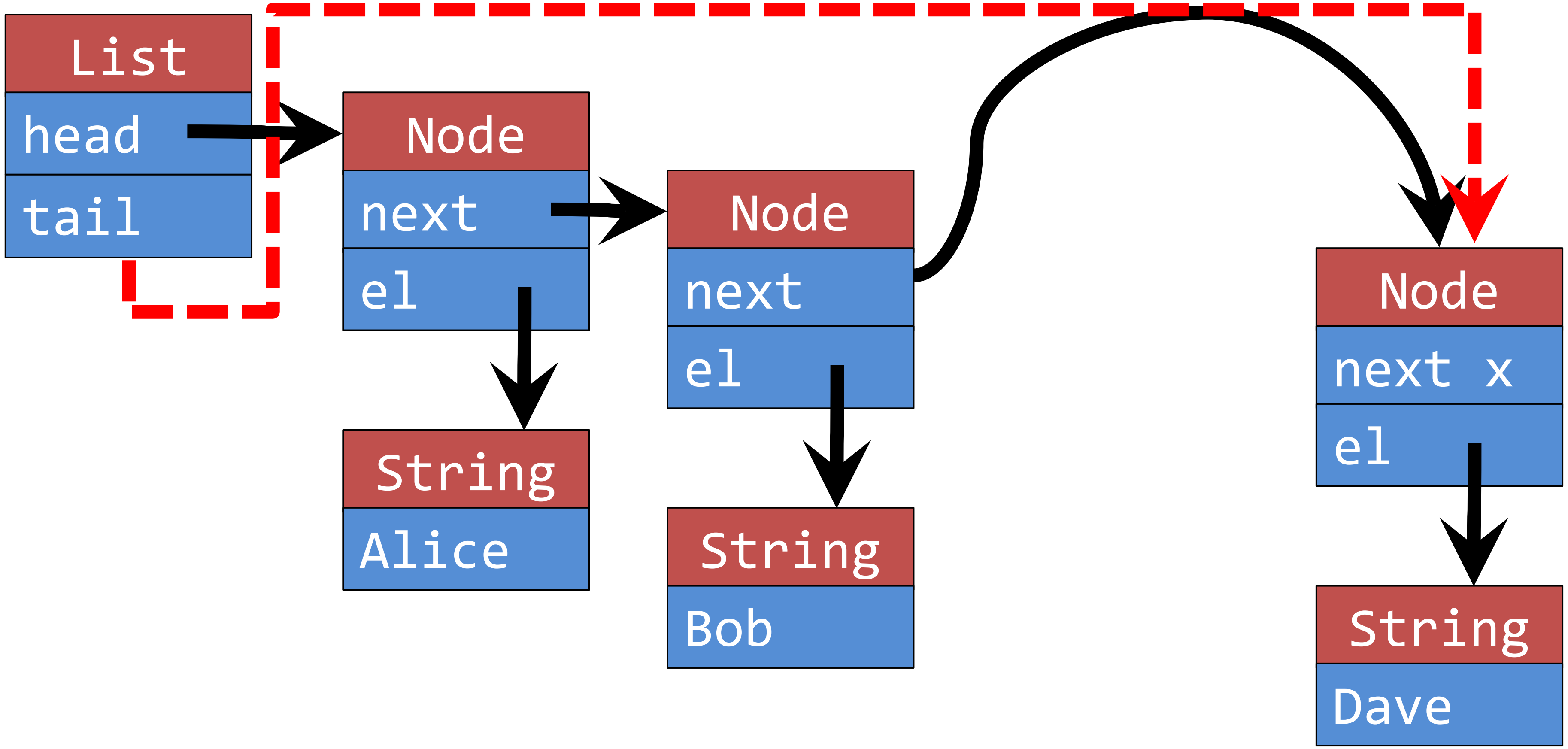
O(N)

O(N)

Linked List: add Carol

this can be done in constant time (O(1)), if we already know the place

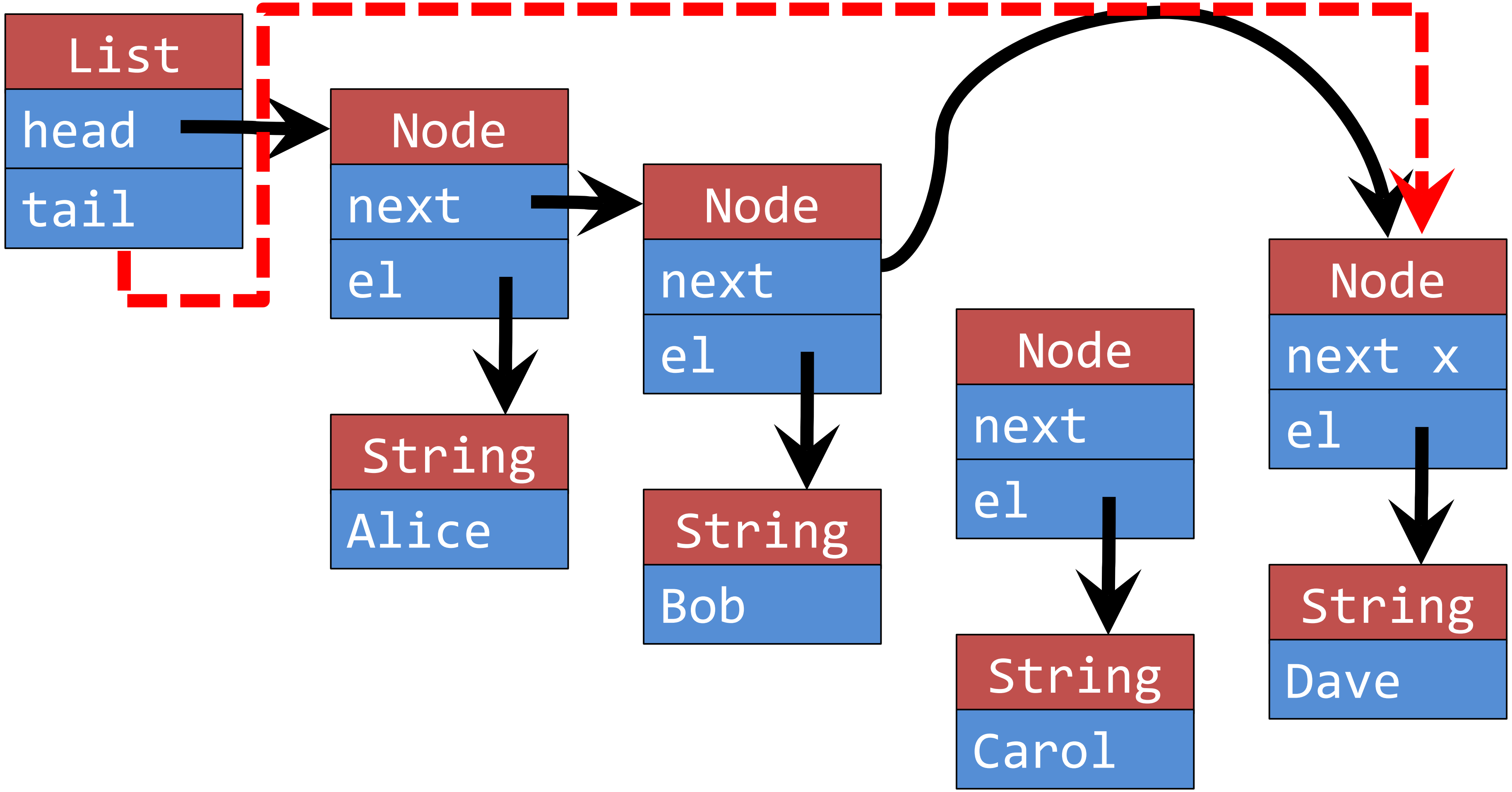
finding the place == `get(i)` is O(N)!



Linked List: add Carol

this can be done in constant time (O(1)), if we already know the place

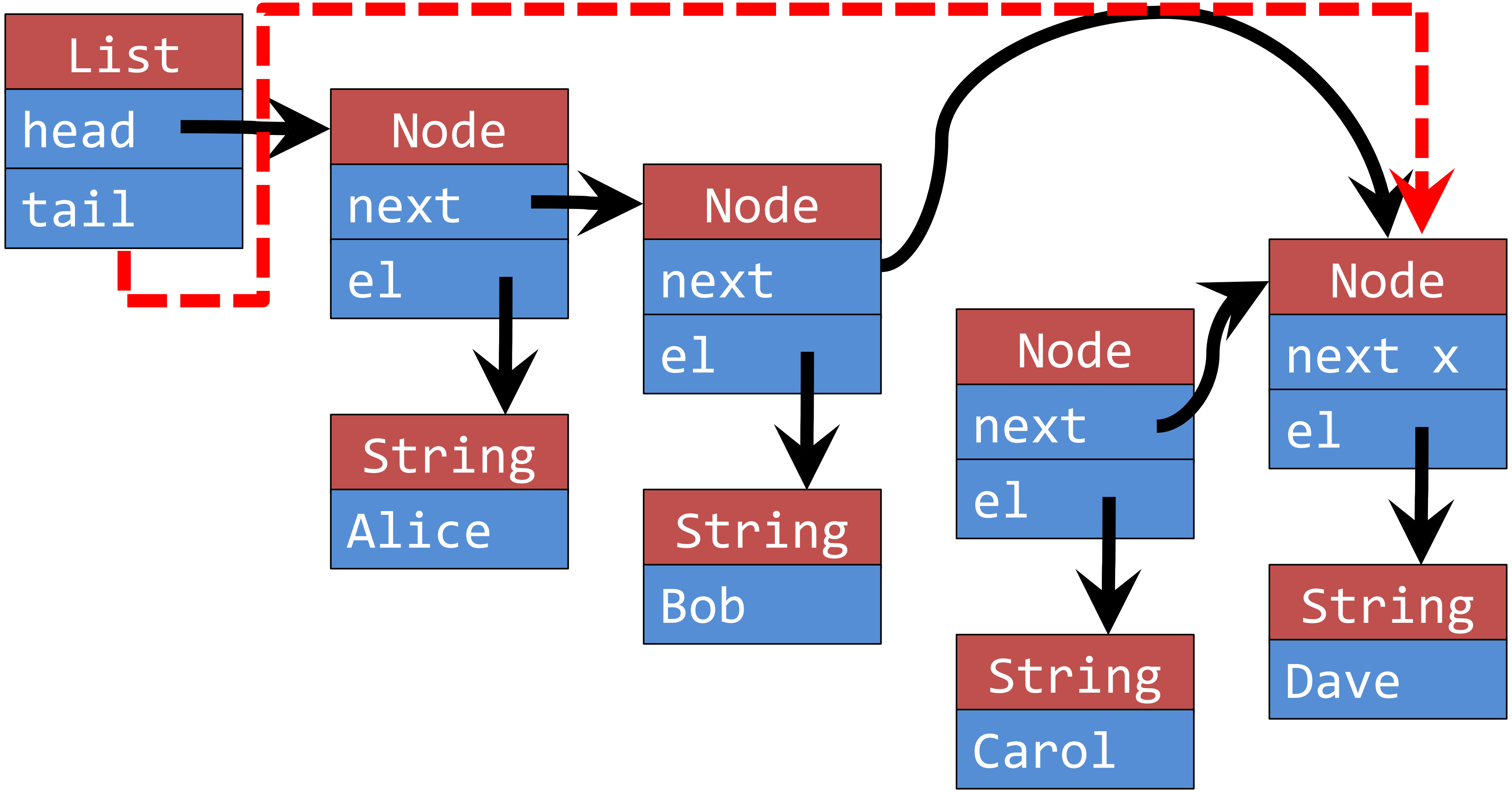
finding the place == `get(i)` is O(N)!



Linked List: add Carol

this can be done in constant time (O(1)), if we already know the place

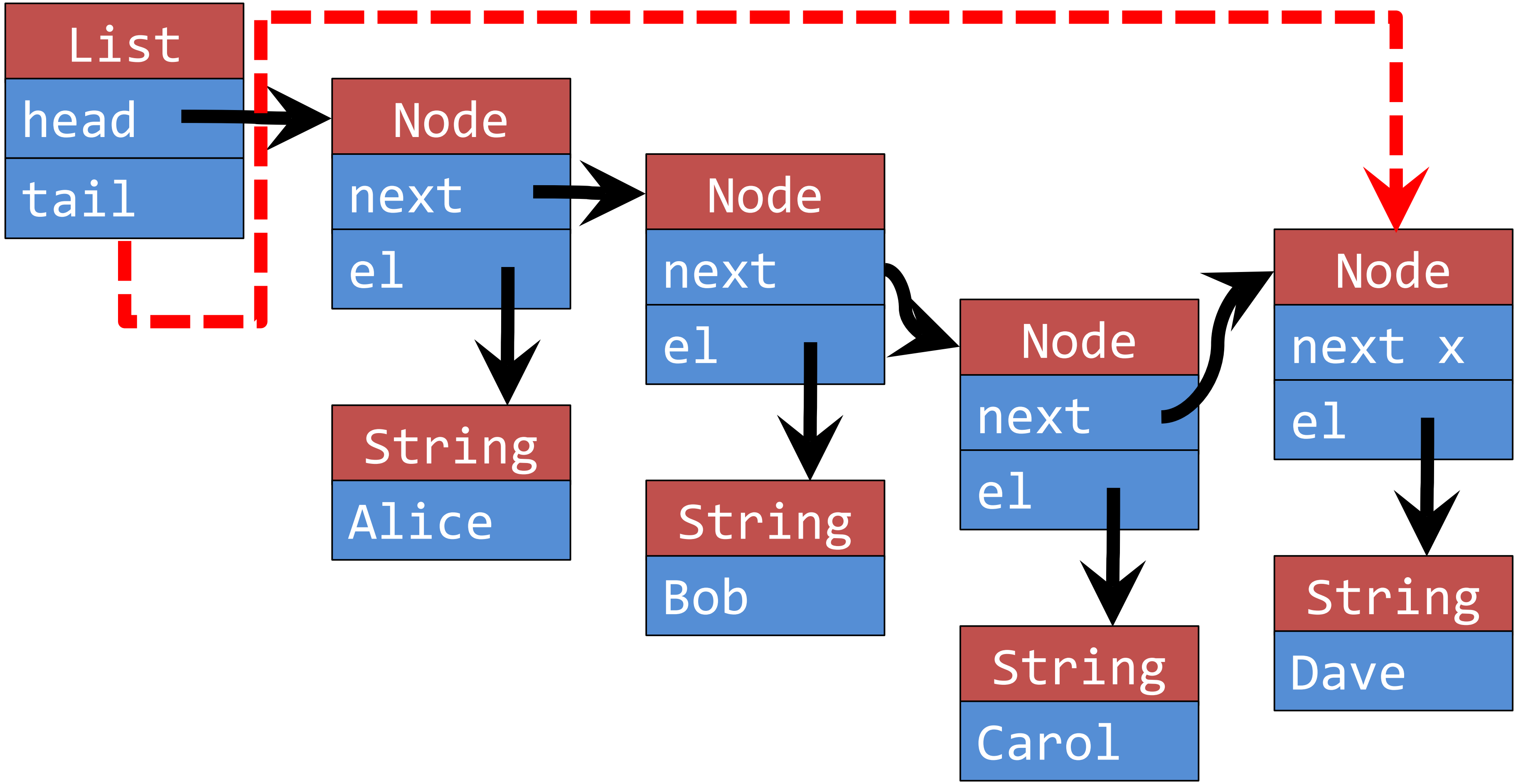
finding the place == `get(i)` is O(N)!



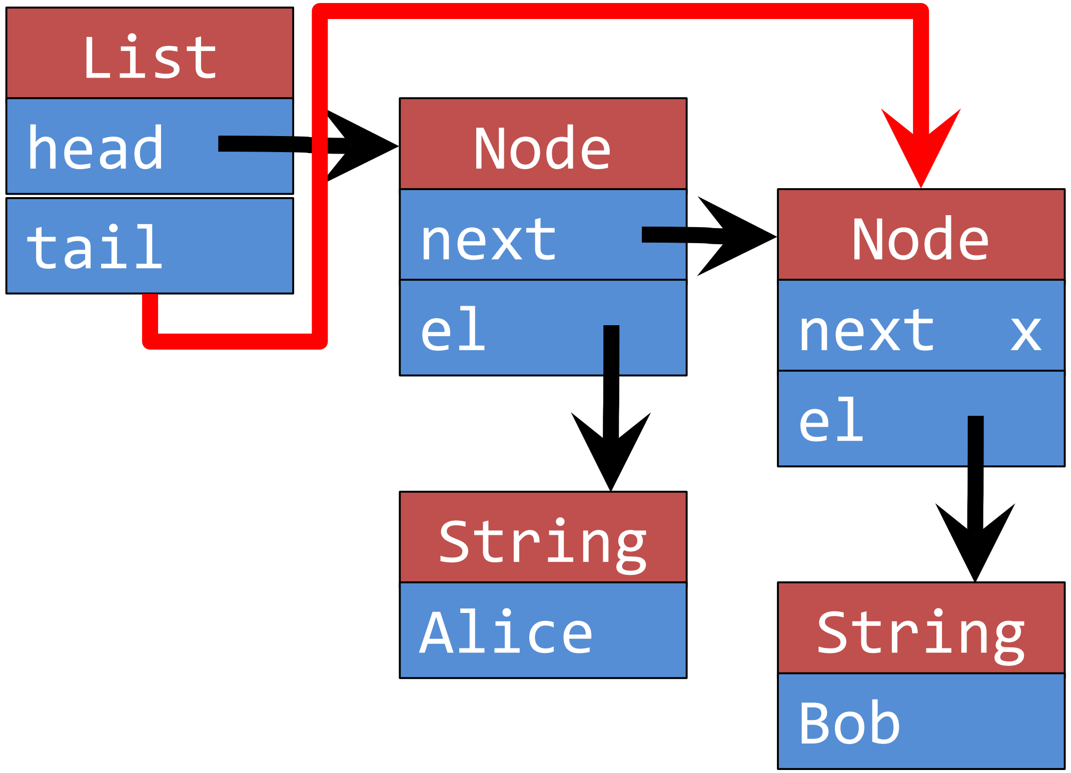
Linked List: add Carol

this can be done in constant time (O(1)), if we already know the place

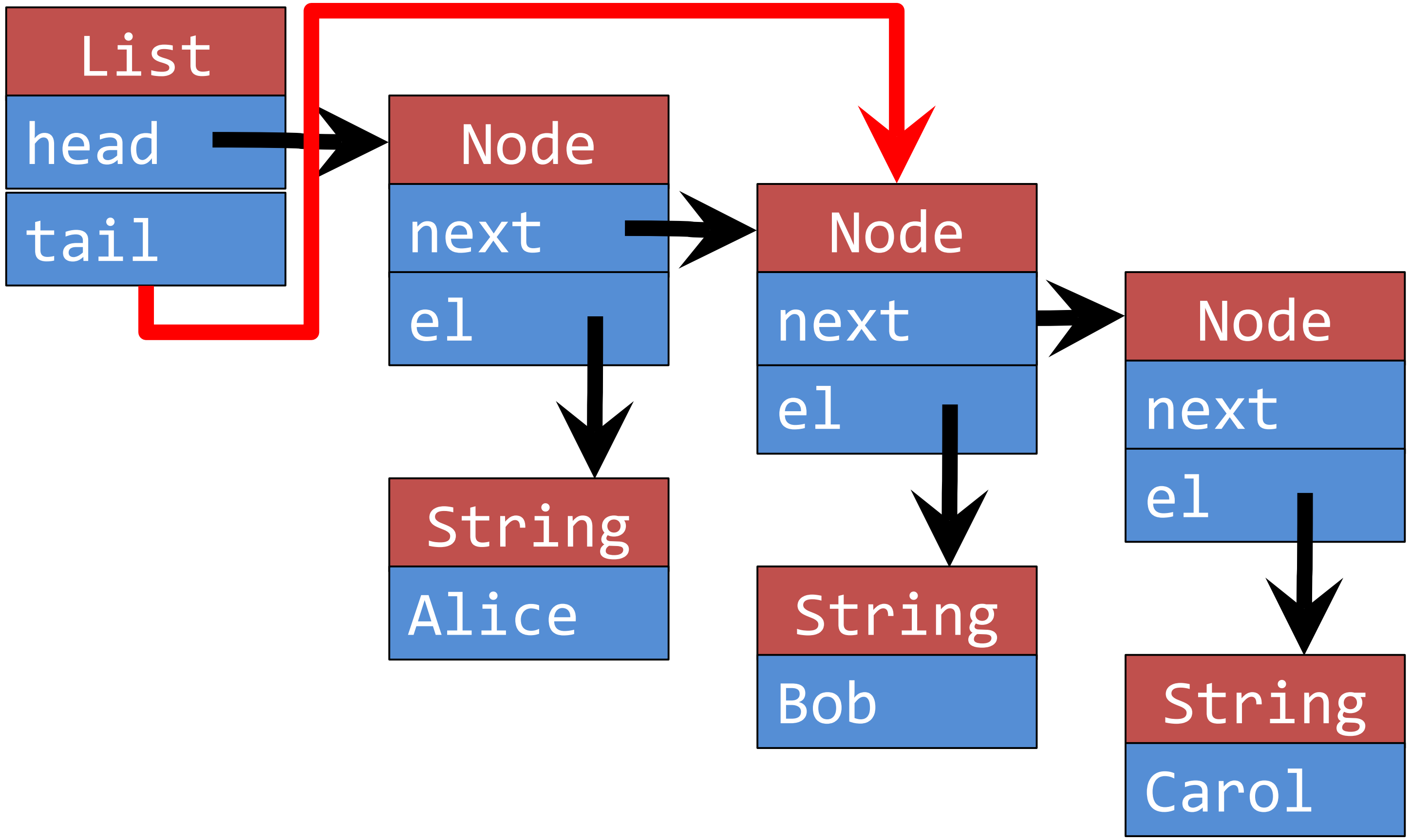
finding the place == `get(i)` is O(N)!



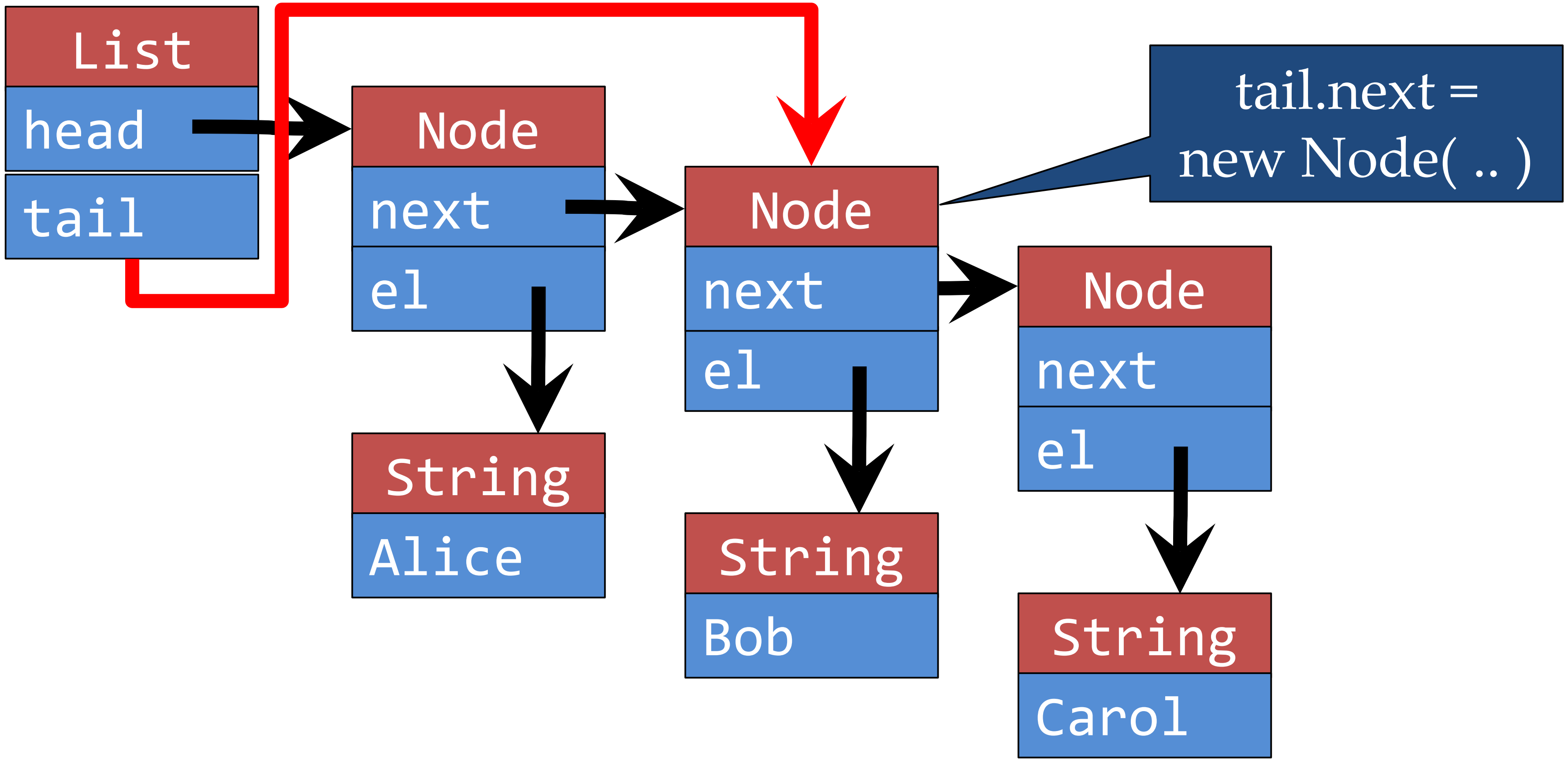
Linked List: efficient add to tail



Linked List: efficient add to tail

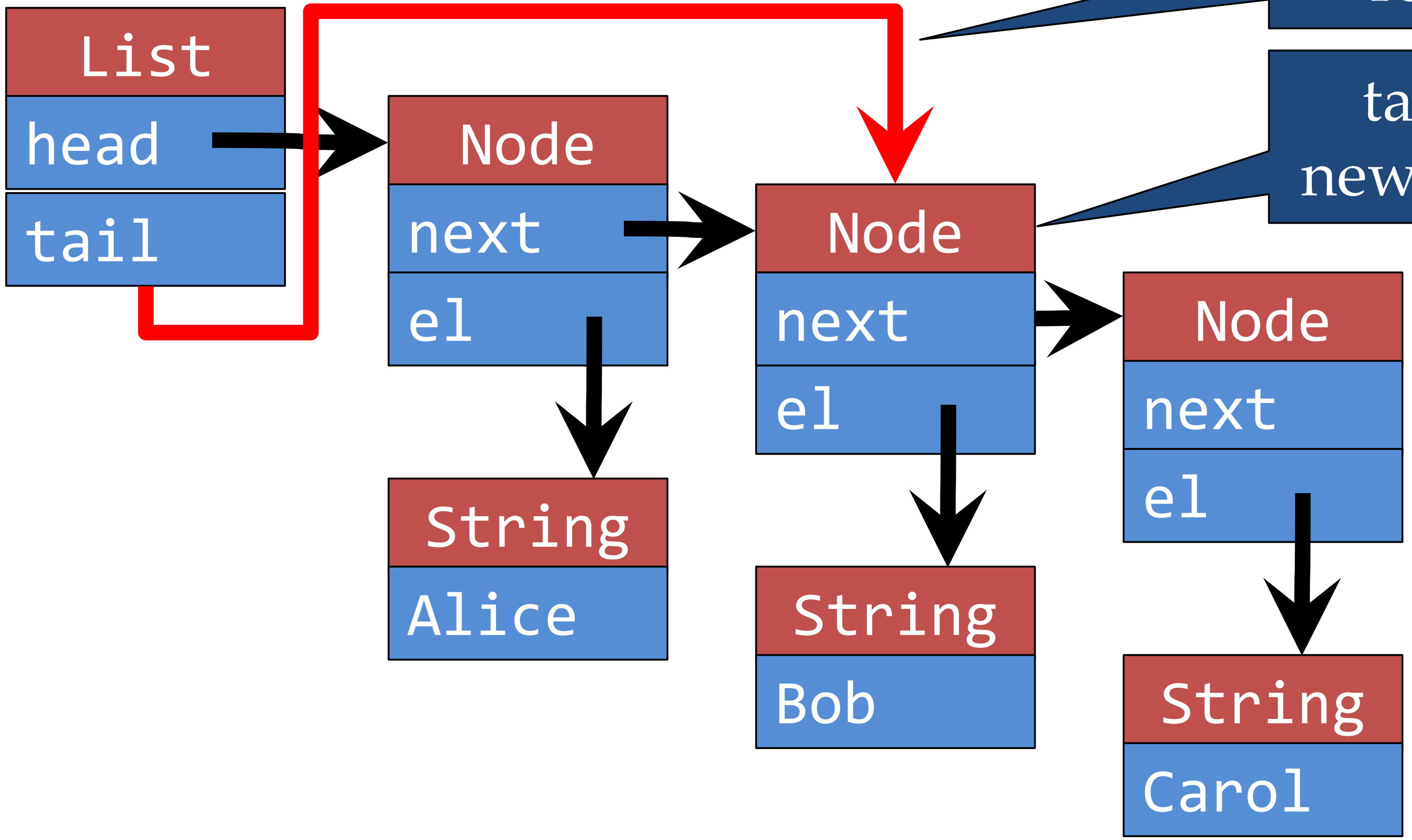


Linked List: efficient add to tail



tail.next =
new Node(..)

Linked List: efficient add to tail

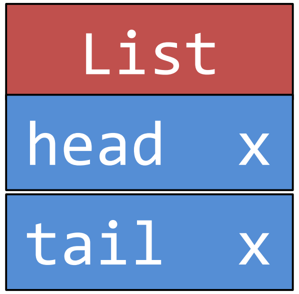


update this reference

tail.next = new Node(..)

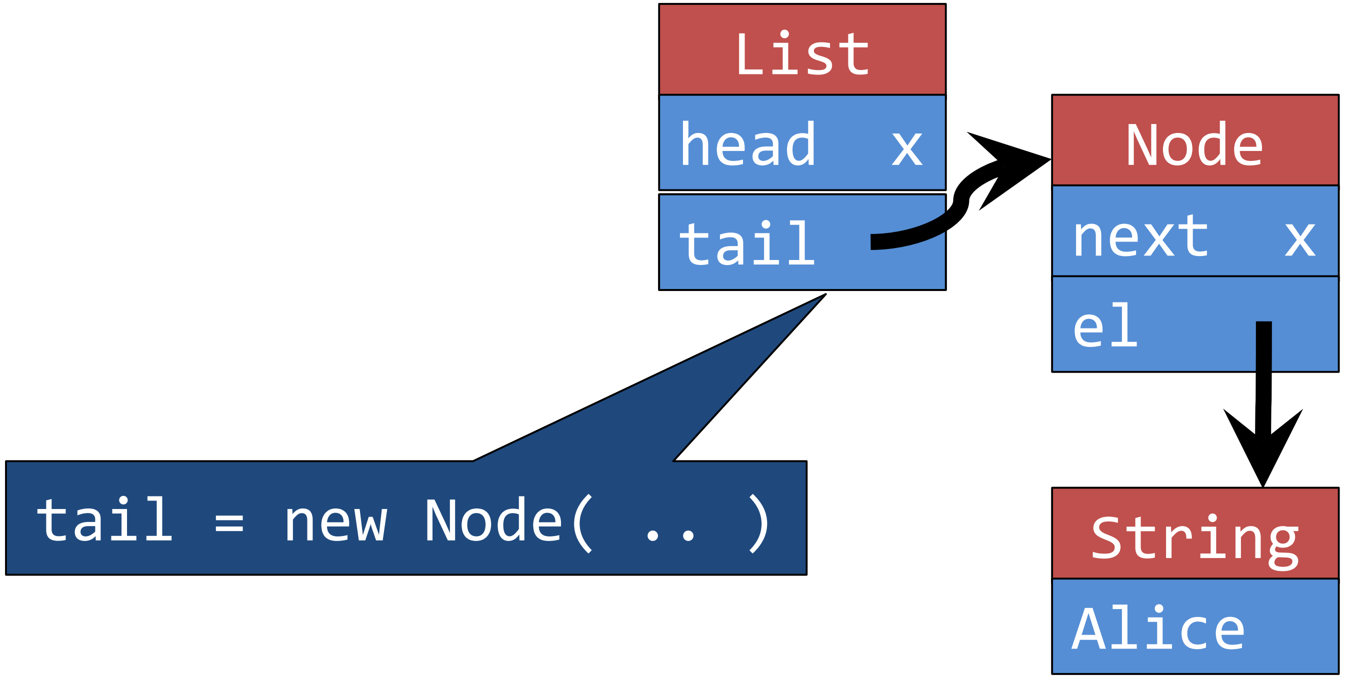
Linked List: add when list is empty

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



Linked List: add when list is empty

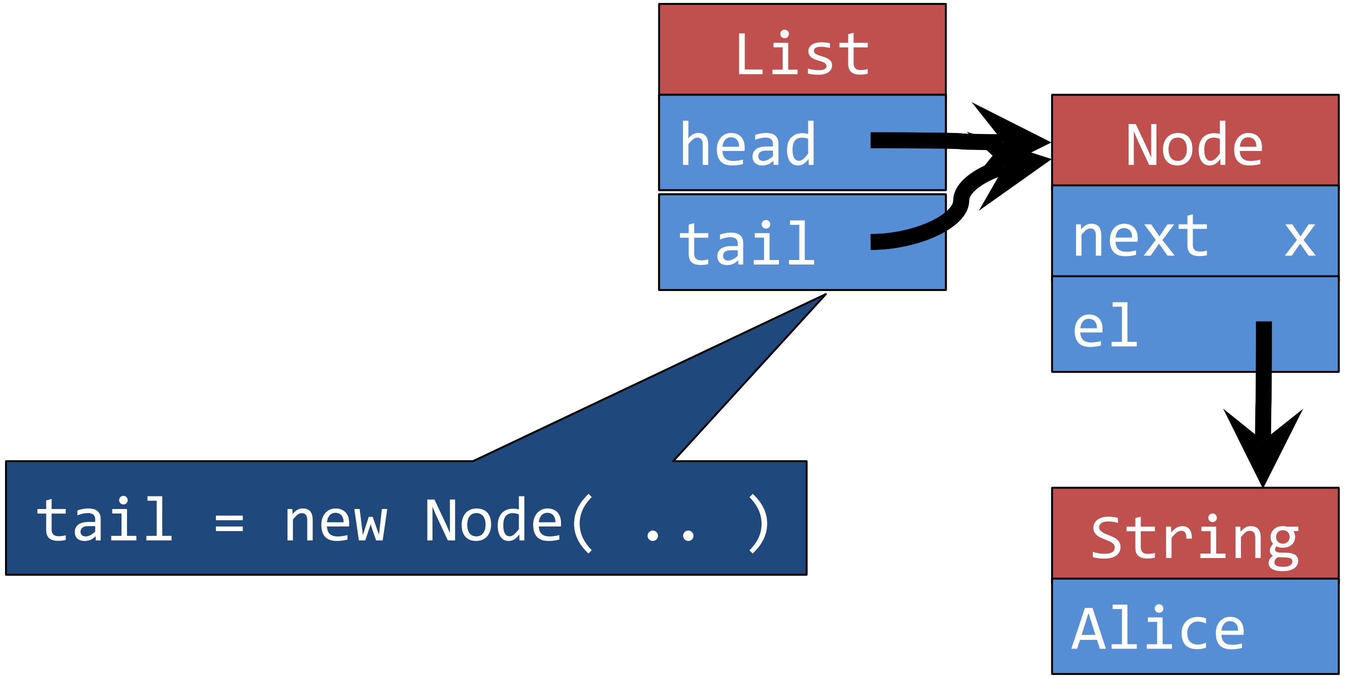
for adding the first node in a list we need a special case



```
tail = new Node( .. )
```

Linked List: add when list is empty

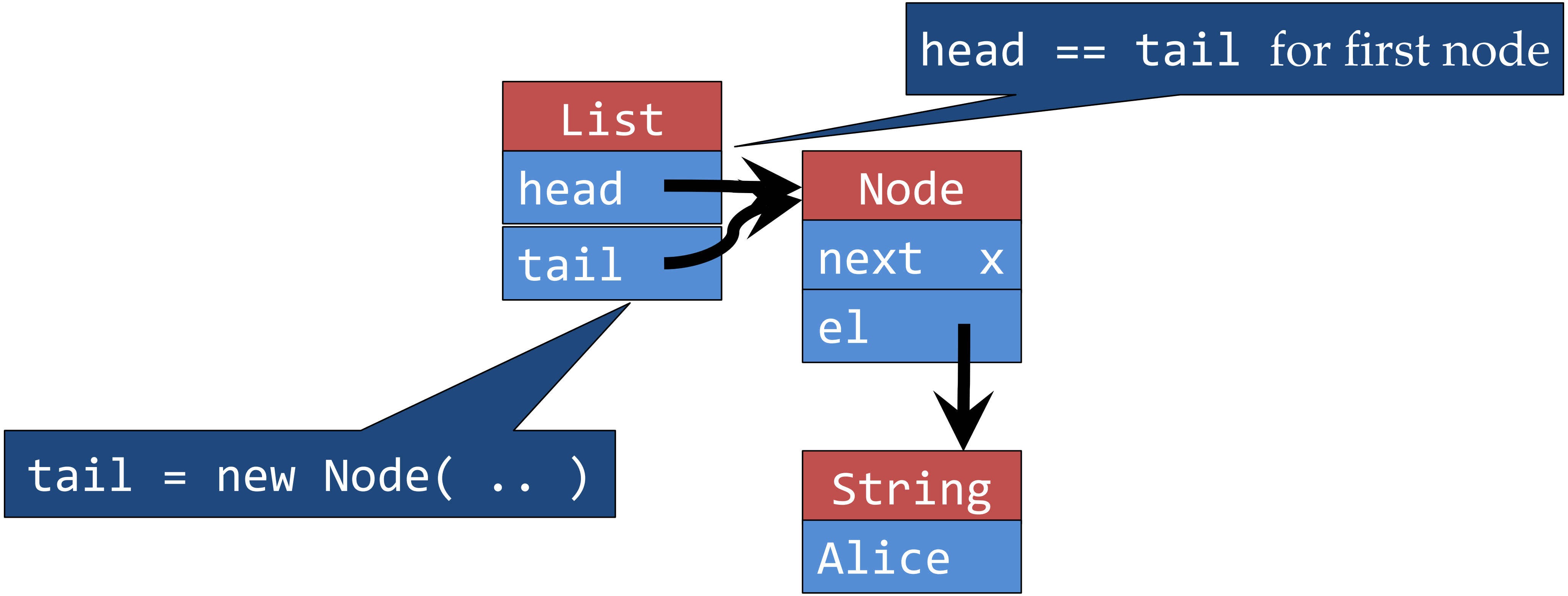
for adding the first node in a list we need a special case



```
tail = new Node( .. )
```

Linked List: add when list is empty

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



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 += 1;
    modCount += 1;
    return true;
}
```

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 += 1;
    modCount += 1;
    return true;
}
```

O(1) ☺

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 += 1;
    modCount += 1;
    return true;
}
```

always be aware of special cases

O(1) ☺

MyLinkedList: `add(index, element)`

```
@Override
public void add(int index, E e) {
    if (index == size) {
        add(e);
        return;
```

MyLinkedList: add(index, element)

```
@Override
public void add(int index, E e) {
    if (index == size) {
        add(e);
        return;
```

at tail: O(1)

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);
```

at tail: O(1)

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);
```

at tail: O(1)

size > 0
at front: O(1)

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 n = getNode(index - 1);
        n.next = new Node(e, n.next);
    }
```

at tail: O(1)

size > 0
at front: O(1)

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 n = getNode(index - 1);
        n.next = new Node(e, n.next);
    }
```

at tail: O(1)

size > 0
at front: O(1)

O(N)

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 n = getNode(index - 1);
        n.next = new Node(e, n.next);
    }
    size += 1;
    modCount += 1;
}
```

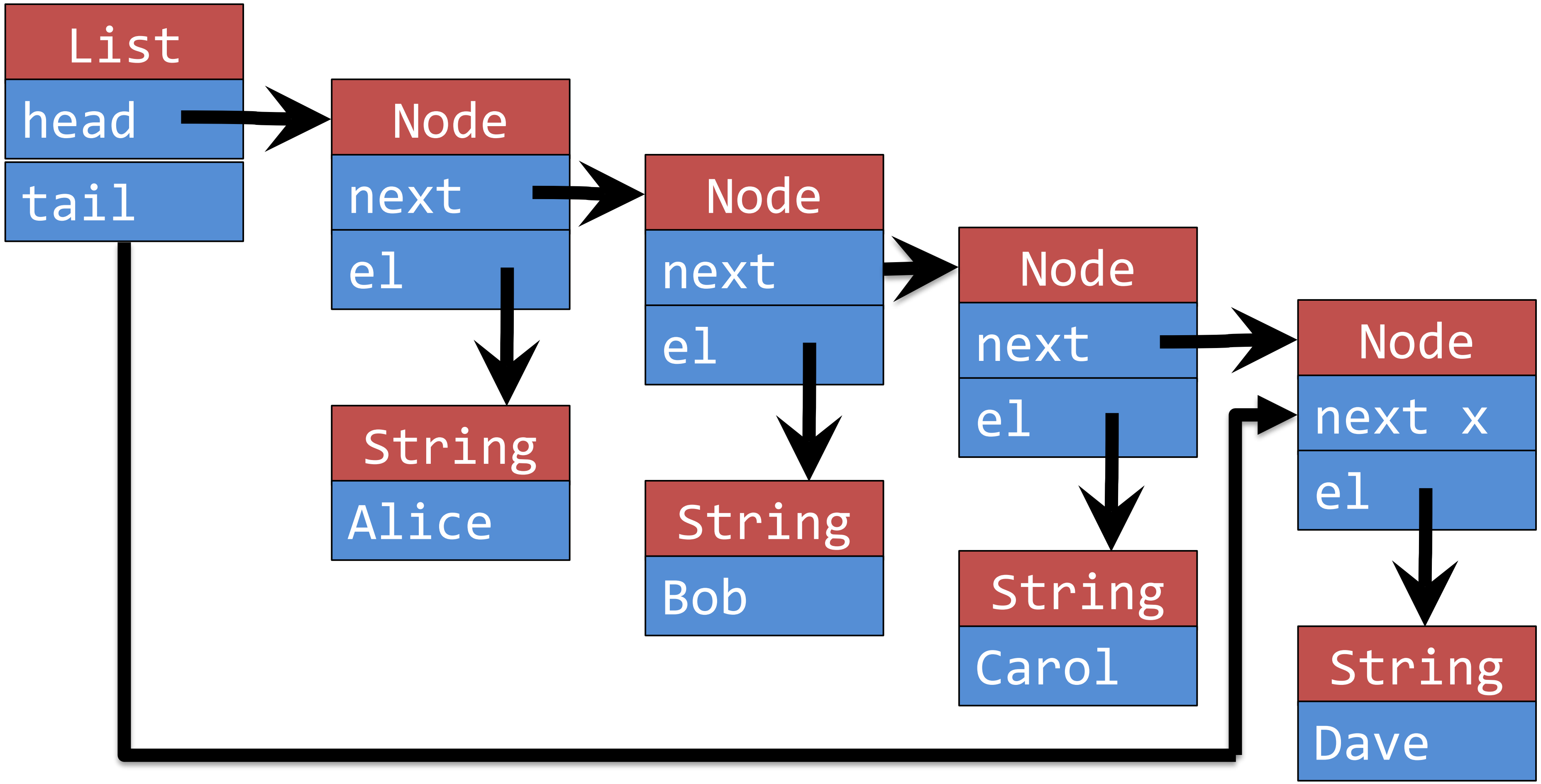
at tail: O(1)

size > 0
at front: O(1)

O(N)

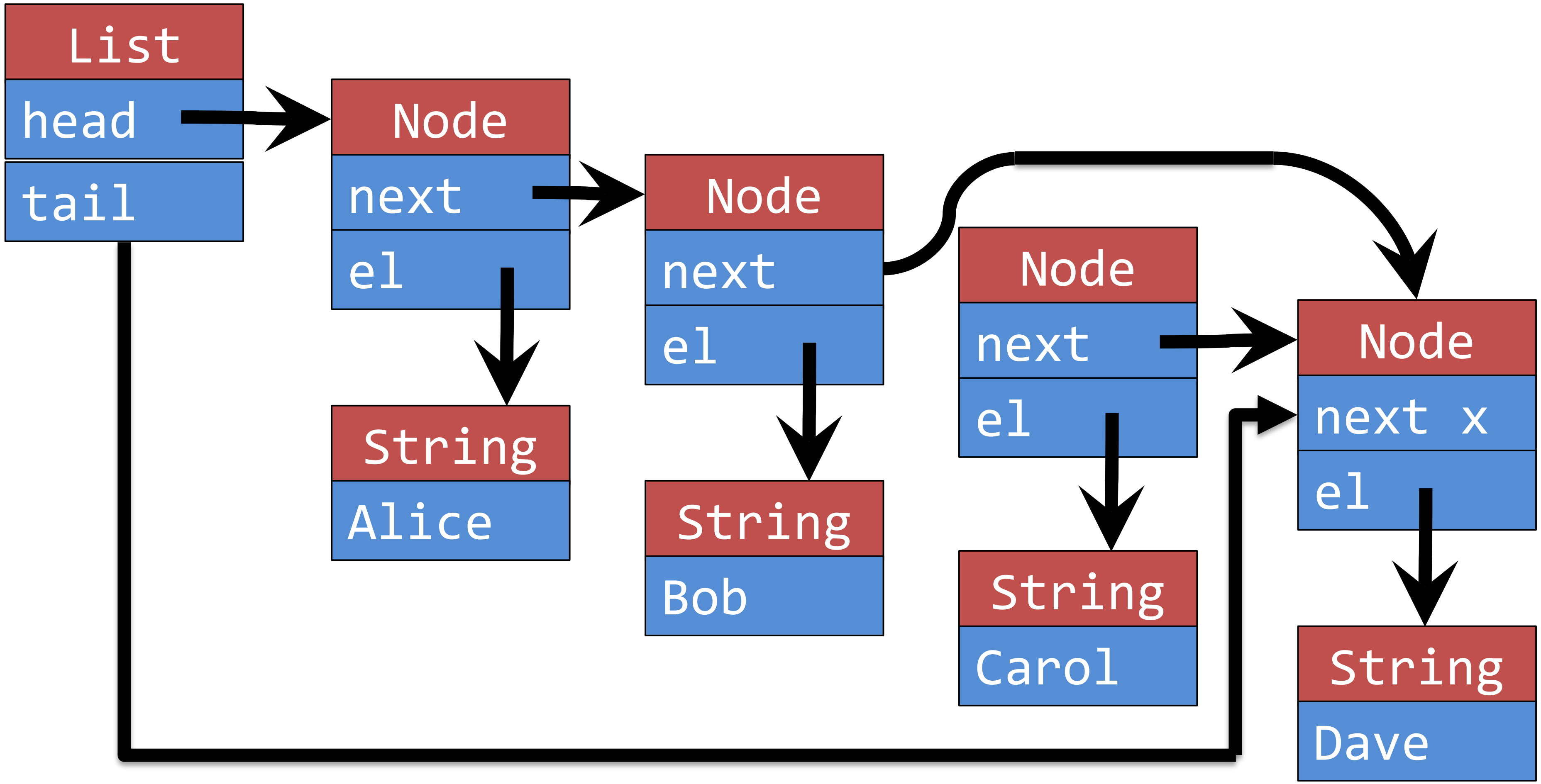
Linked List: remove(i)

start at `head`; follow `i-1` next pointers



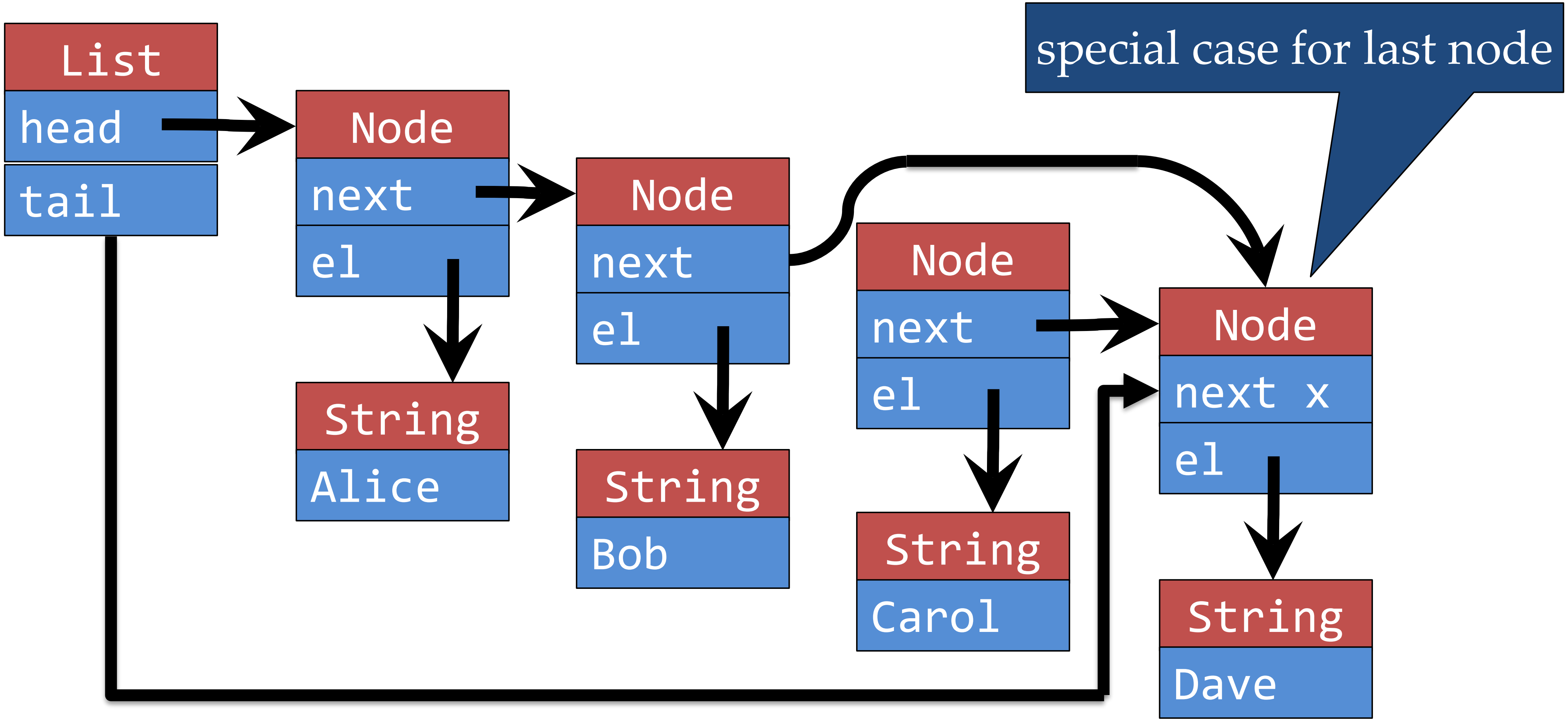
Linked List: remove(i)

start at **`head`**; follow **`i-1`** next pointers



Linked List: remove(i)

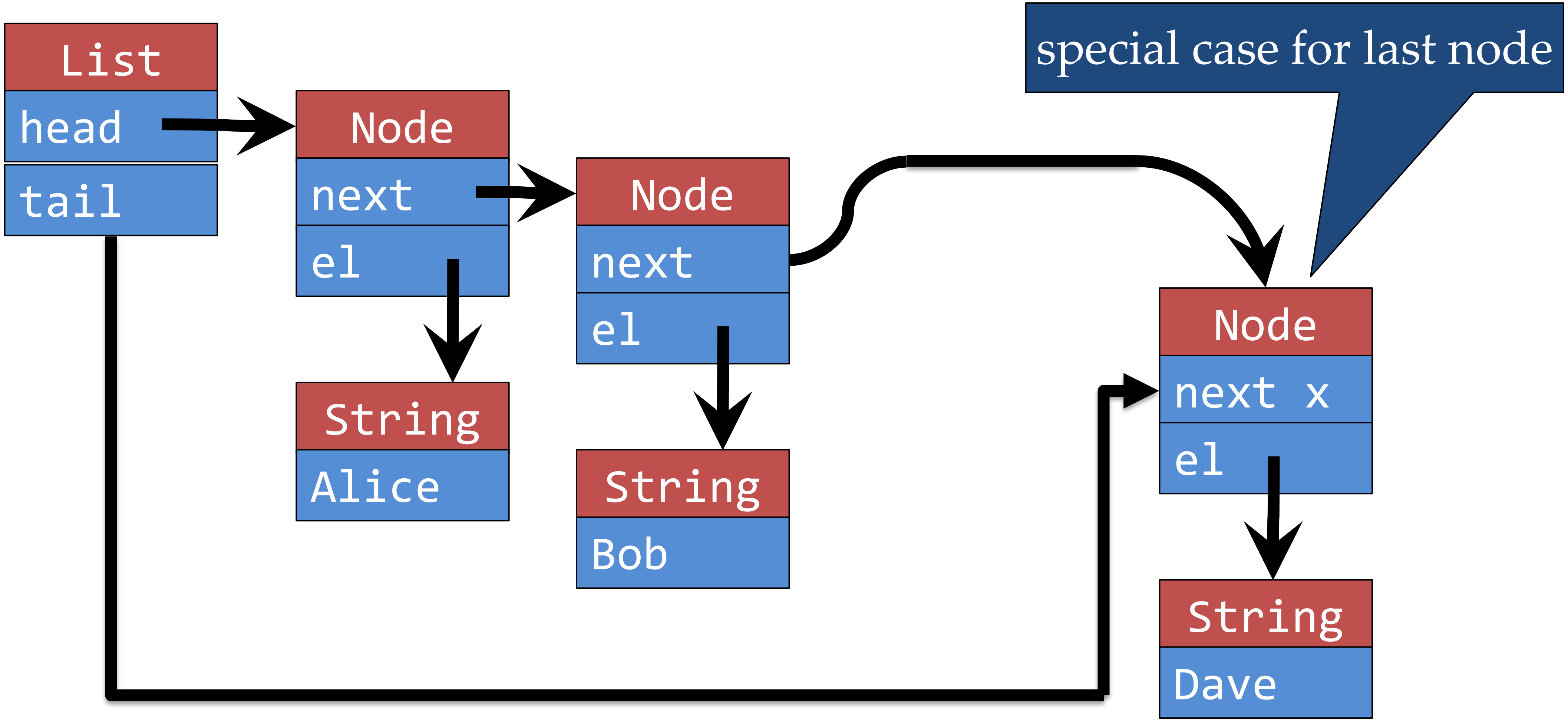
start at `head`; follow `i-1` next pointers



special case for last node

Linked List: remove(i)

start at `head`; follow `i-1` next pointers



special case for last node

MyLinkedList: remove(index)

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

MyLinkedList: remove(index)

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

O(N)

MyLinkedList: `remove(index)`

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

first element special

O(N)

MyLinkedList: remove(index)

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

first element special

O(N)

last element special

MyLinkedList: `remove(index)`

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

first element special

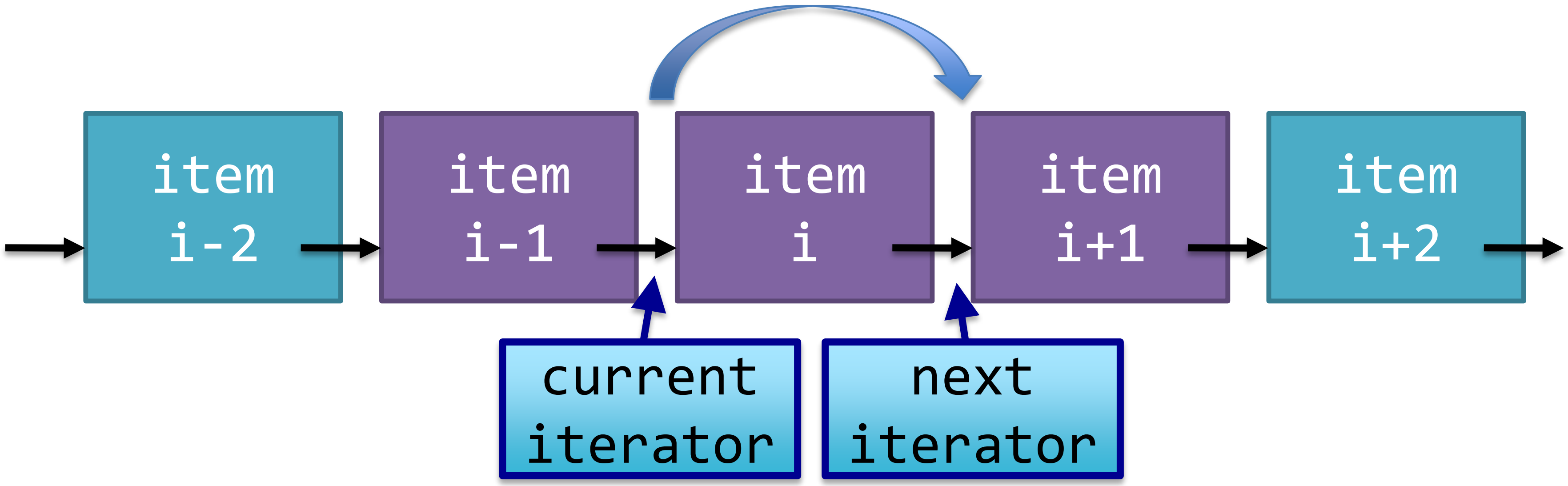
O(N)

last element special

Can't use `tail` for this!

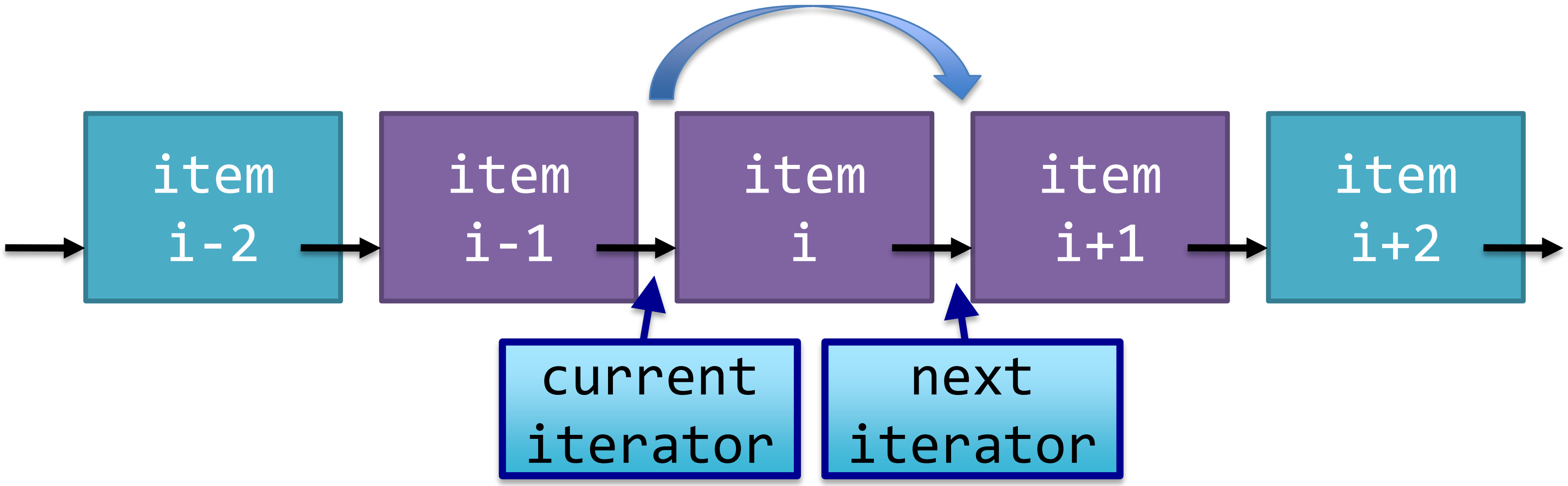
iterator for linked list

`boolean hasNext()`	checks if there is a next item
`E next()`	yields next element
`void remove()`	removes last object produced



iterator for linked list

`boolean hasNext()`	checks if there is a next item
`E next()`	yields next element
`void remove()`	removes last object produced



`Node next`	indicates next object in the list
`Node previous`	indicates last object produced

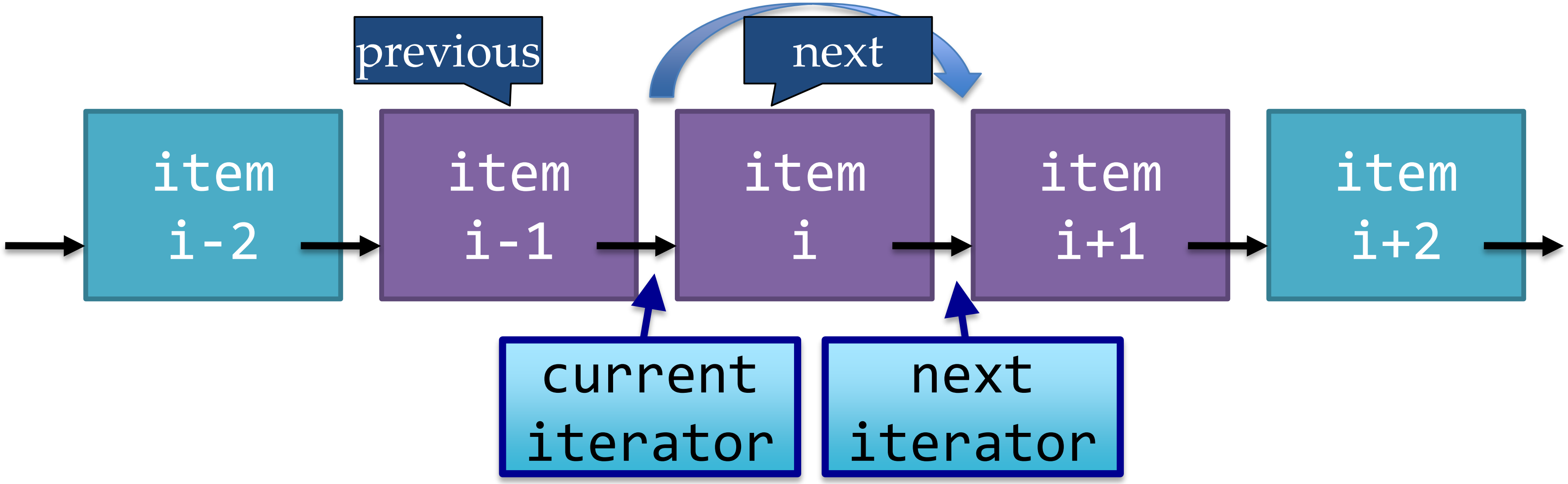
- **`null`** if there is no last object

iterator for linked list

`boolean hasNext()` checks if there is a next item

`E next()` yields next element

`void remove()` removes last object produced



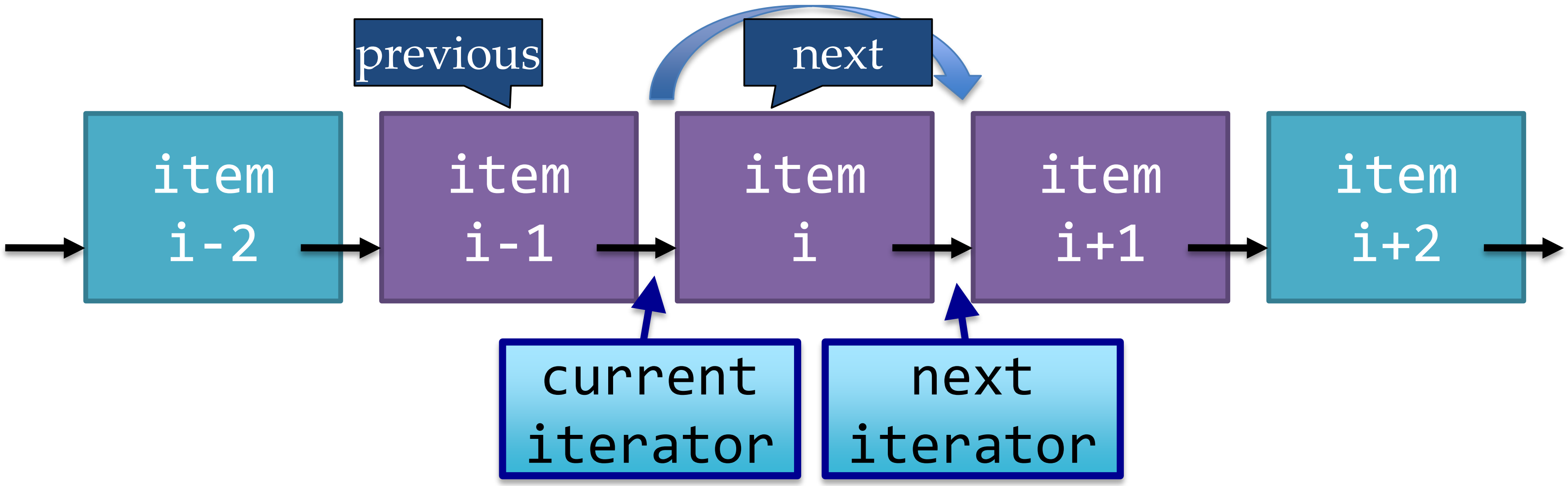
`Node next` indicates next object in the list

`Node previous` indicates last object produced

- **`null`** if there is no last object

iterator for linked list

`boolean hasNext()`	checks if there is a next item
`E next()`	yields next element
`void remove()`	removes last object produced



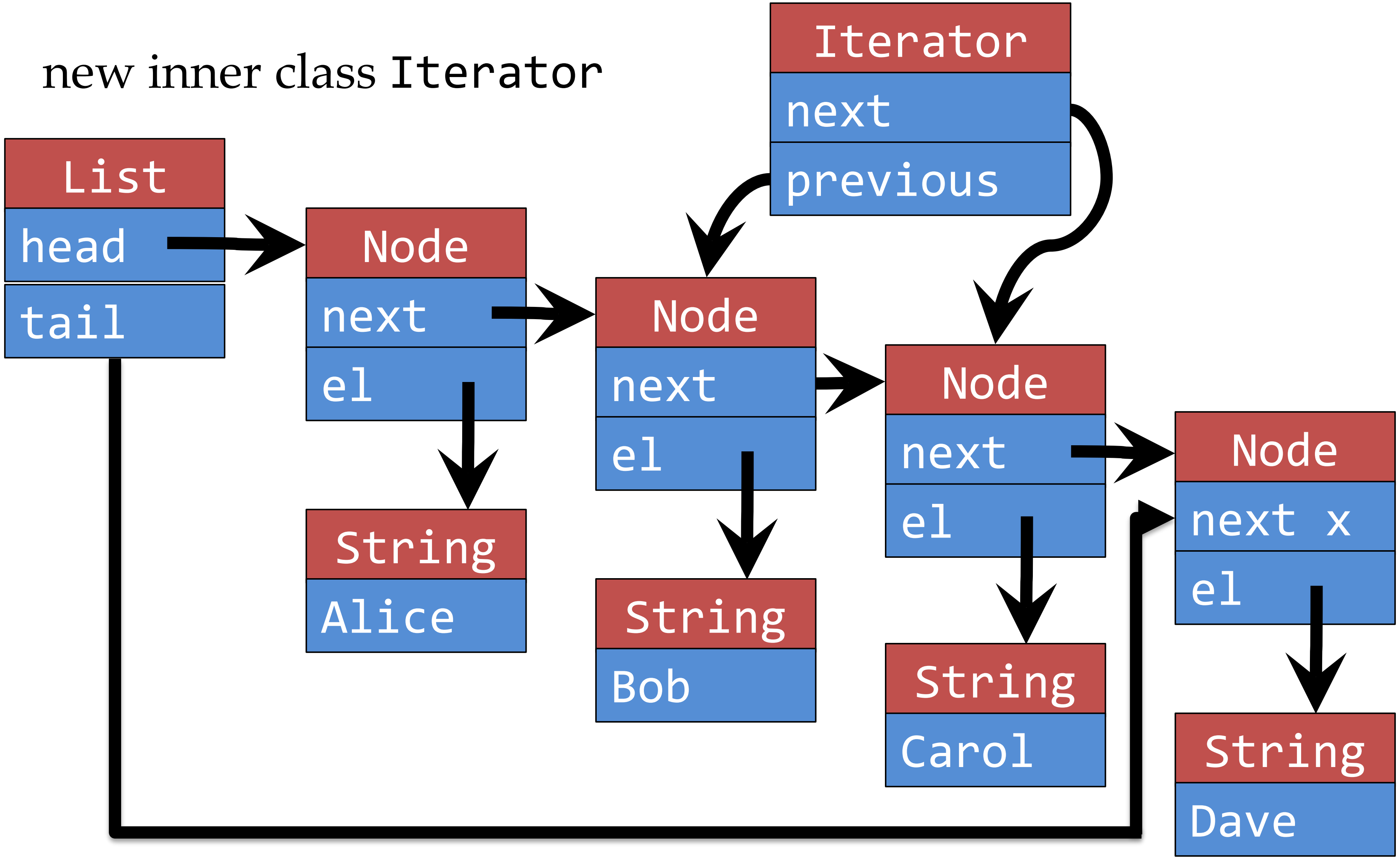
`Node next`	indicates next object in the list
`Node previous`	indicates last object produced

- `null` if there is no last object

`int knownMod`	to identify concurrent modifications

Linked List: Iterator

new inner class `Iterator`



MyLinkedList: Iterator

make a smart iterator implementation, like before

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

MyLinkedList: Iterator

make a smart iterator implementation, like before

```
@Override
public Iterator<E> iterator() {
    return new MyIterator();
}
public class MyIterator implements Iterator<E> {
    protected Node current = head, previous;
```

MyLinkedList: Iterator

make a smart iterator implementation, like before

```
@Override
public Iterator<E> iterator() {
    return new MyIterator();
}
public class MyIterator implements Iterator<E> {
    protected Node current = head, previous;
```

last element given

MyLinkedList: Iterator

make a smart iterator implementation, like before

```
@Override
public Iterator<E> iterator() {
    return new MyIterator();
}
public class MyIterator implements Iterator<E> {
    protected Node current = head, previous;
```

last element given

node of next element

MyLinkedList: Iterator

make a smart iterator implementation, like before

```
@Override
public Iterator<E> iterator() {
    return new MyIterator();
}
public class MyIterator implements Iterator<E> {
    protected Node current = head, previous;

    @Override
    public boolean hasNext () {
        return current != null;
    }
```

last element given

node of next element

MyLinkedList: Iterator

```
@Override
public E next() {
    if (current == null)
        throw new NoSuchElementException();
    else {
        previous = current;
        E e      = current.el;
        current  = current.next;
        return e;
    }
}
```

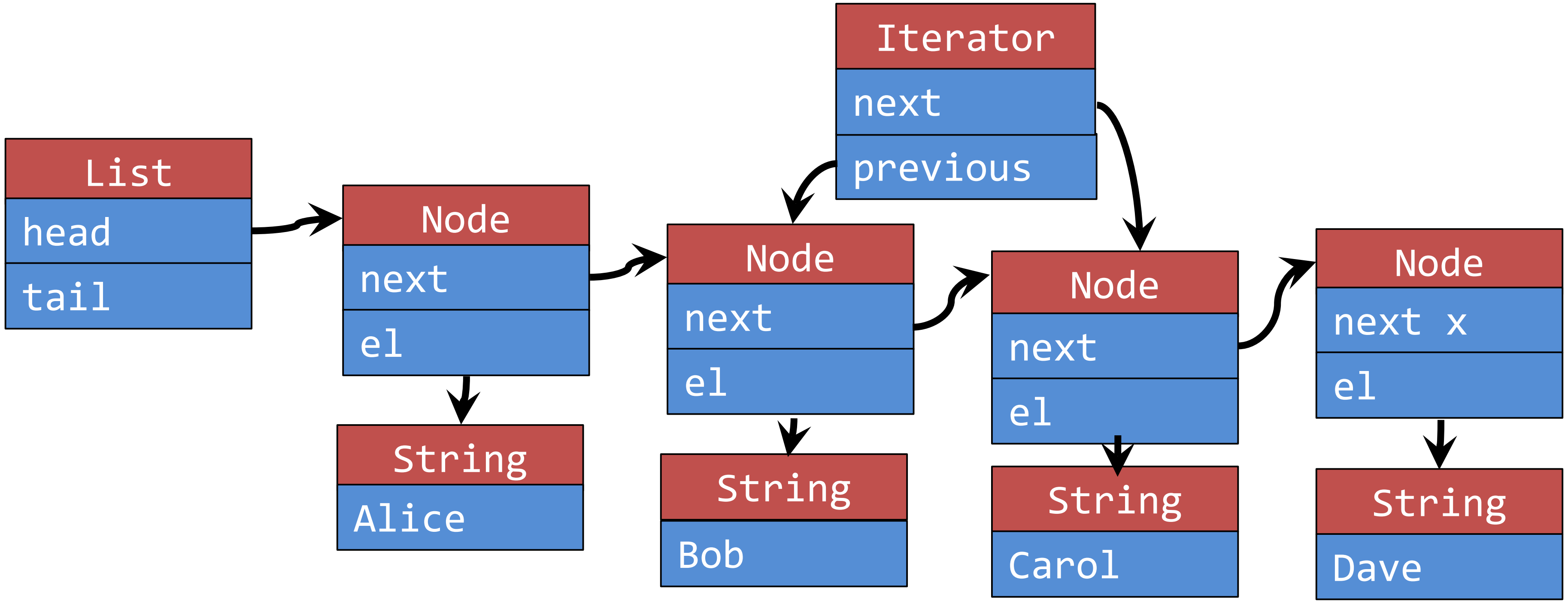
MyLinkedList: Iterator

```java
@Override
public E next() {
    if (current == null)
        throw new NoSuchElementException();
    else {
        previous = current;
        E e      = current.el;
        current  = current.next;
        return e;
    }
}
```

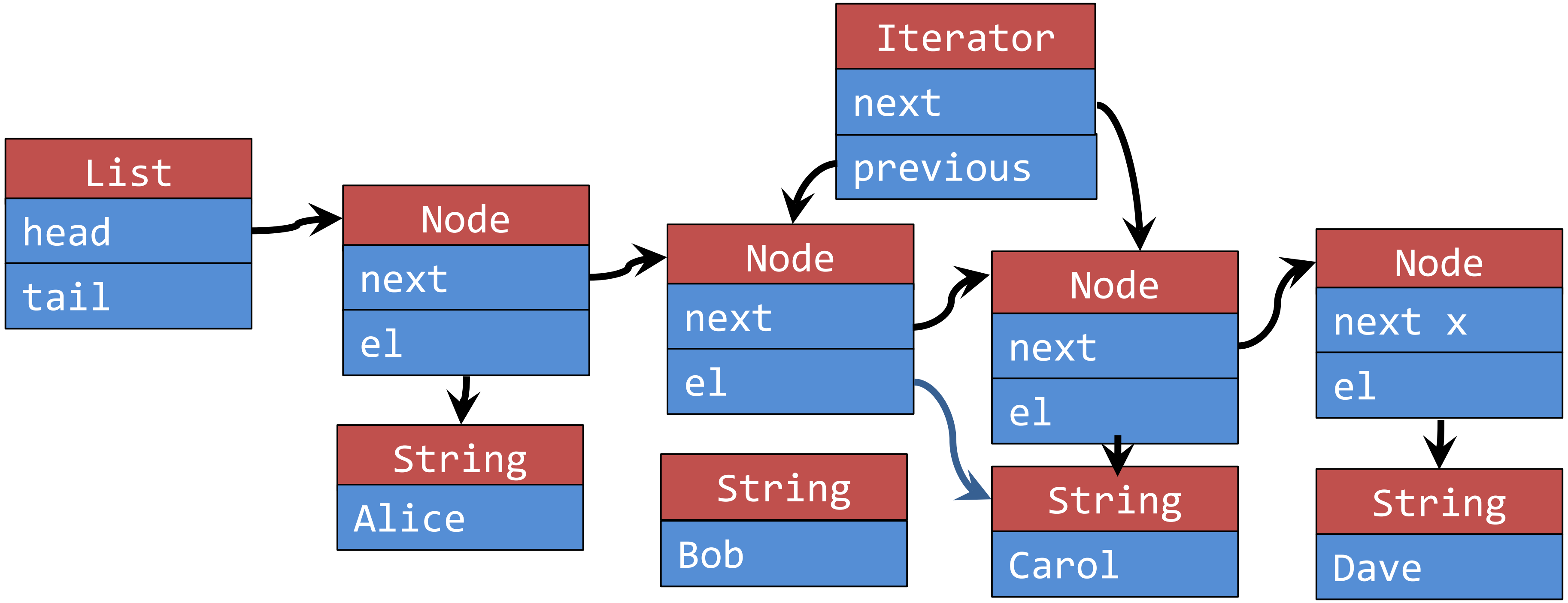
O(1)

Iterator: `remove`

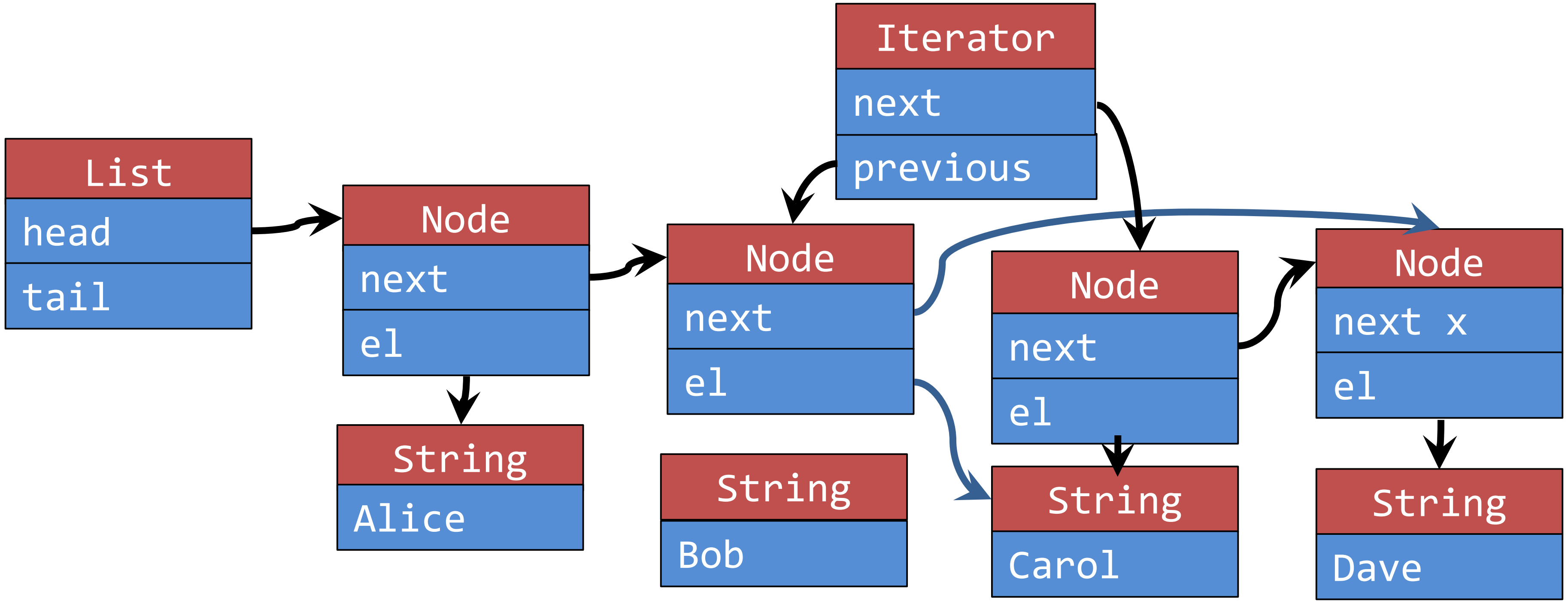
Iterator: `remove`



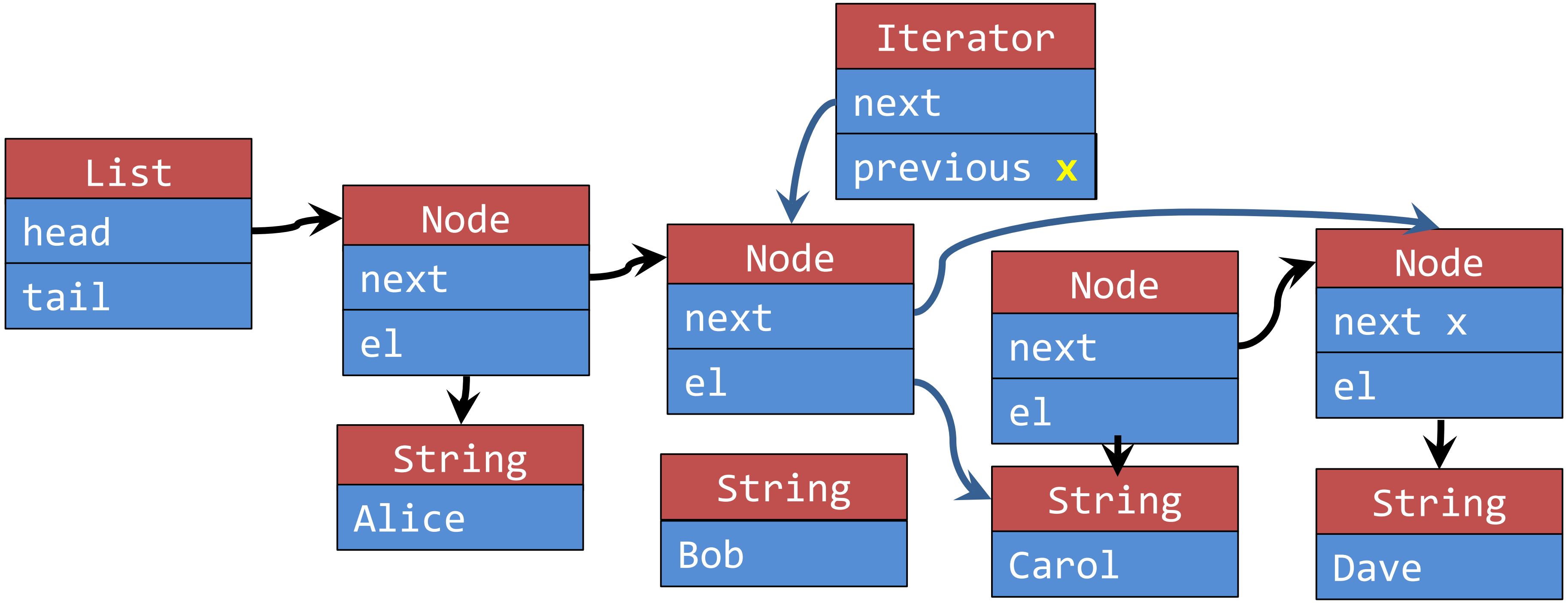
Iterator: remove



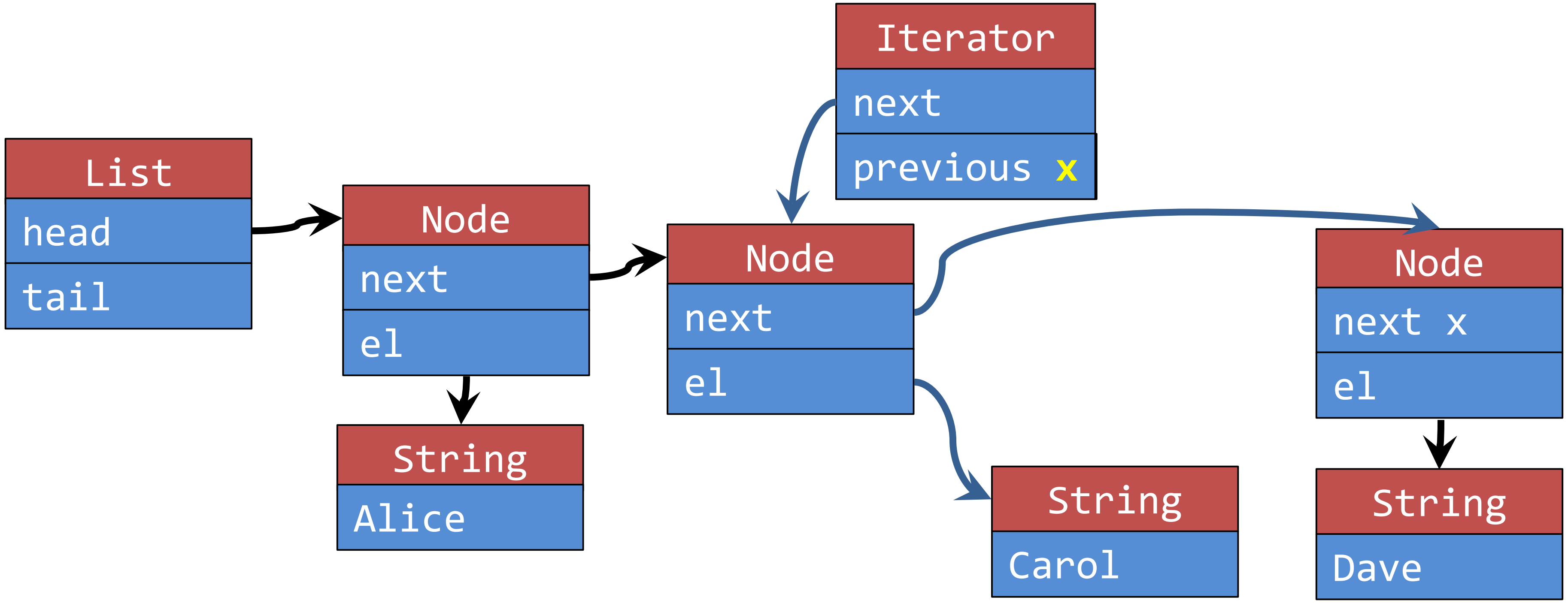
Iterator: `remove`



Iterator: `remove`



Iterator: `remove`



MyLinkedList: `Iterator.remove()`

```
public void remove () {









        previous.el = current.el;
        previous.next = current.next;
        current = previous;







}
```

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();



    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }



}
```

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
```

no repeated removes

```
    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }
}
```

MyLinkedList: Iterator.remove()

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;



    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }



}
```

no repeated removes

MyLinkedList: Iterator.remove()

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;



    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }



}
```

no repeated removes

remove only element

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;



    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }



}
```

no repeated removes

remove only element

remove first element

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;
    } else if (pos == size - 1) {
        tail = previous;
        previous.next = null;
    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }



}
```

no repeated removes

remove only element

remove first element

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;
    } else if (pos == size - 1) {
        tail = previous;
        previous.next = null;
    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }


}
```

no repeated removes

remove only element

remove first element

remove last element

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;
    } else if (pos == size - 1) {
        tail = previous;
        previous.next = null;
    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }
    pos      -= 1;
    size     -= 1;
    previous = null;
    modCount += 1;
    knownMod = modCount;
}
```

no repeated removes

remove only element

remove first element

remove last element

MyLinkedList: `Iterator.remove()`

```
public void remove () {
    checkVersion();
    if (previous == null) {
        throw new IllegalStateException();
    } else if (pos == 0) {
        if (size == 1) {
            current = head = tail = null;
        } else
            current = head = head.next;
    } else if (pos == size - 1) {
        tail = previous;
        previous.next = null;
    } else {
        previous.el = current.el;
        previous.next = current.next;
        current = previous;
    }
    pos      -= 1;
    size     -= 1;
    previous = null;
    modCount += 1;
    knownMod = modCount;
}
```

no repeated removes

remove only element

remove first element

remove last element

only this iterator can continue

MyLinkedList evaluation

we solved the problems with `add(E e)` in `MyArrayList`

- this could be O(N) in `MyArrayList`
- now it is O(1) ☺

MyLinkedList evaluation

we solved the problems with `add(E e)` in `MyArrayList`

- this could be O(N) in `MyArrayList`
- now it is O(1) ☺

`add(int i, E e)` and `remove(int i)` itself are O(1)

- only finding the right spot is O(N) ☹

MyLinkedList evaluation

we solved the problems with `add(E e)` in `MyArrayList`

- this could be O(N) in `MyArrayList`
- now it is O(1) ☺

`add(int i, E e)` and `remove(int i)` itself are O(1)

- only finding the right spot is O(N) ☹

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

MyLinkedList evaluation

we solved the problems with `add(E e)` in `MyArrayList`

- this could be O(N) in `MyArrayList`
- now it is O(1) ☺

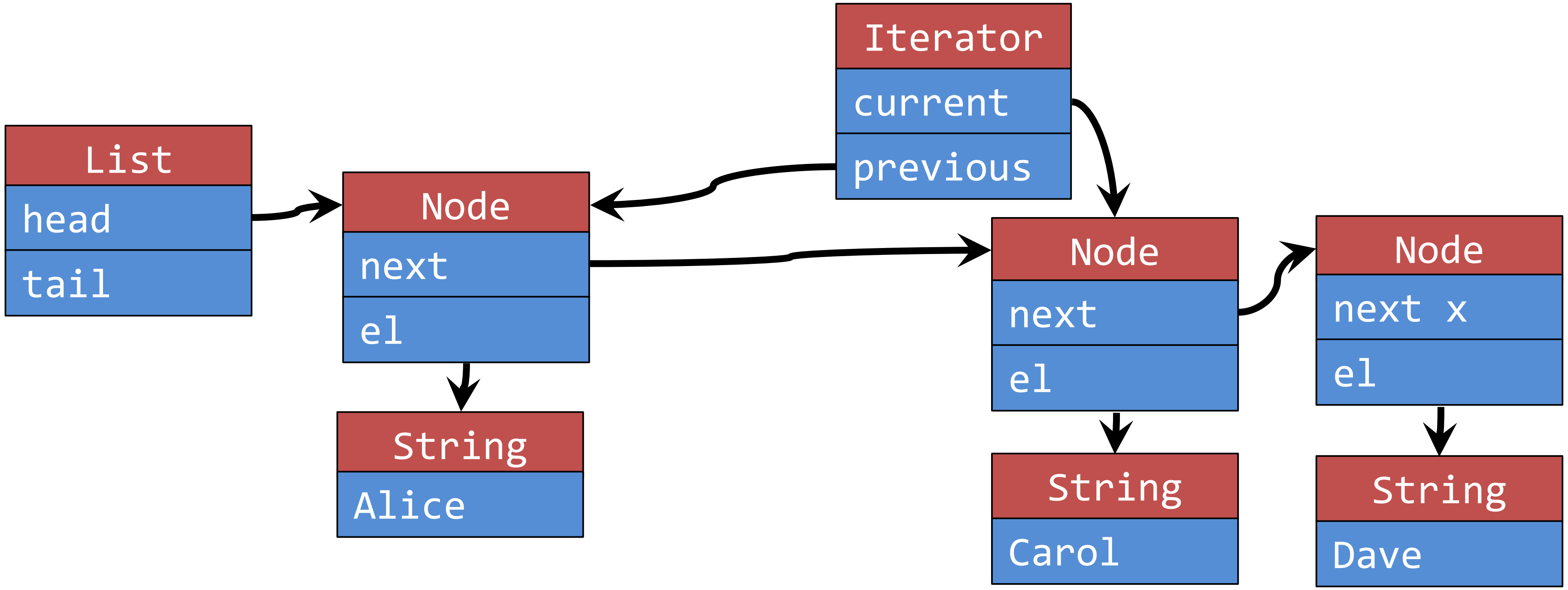
`add(int i, E e)` and `remove(int i)` itself are O(1)

- only finding the right spot is O(N) ☹

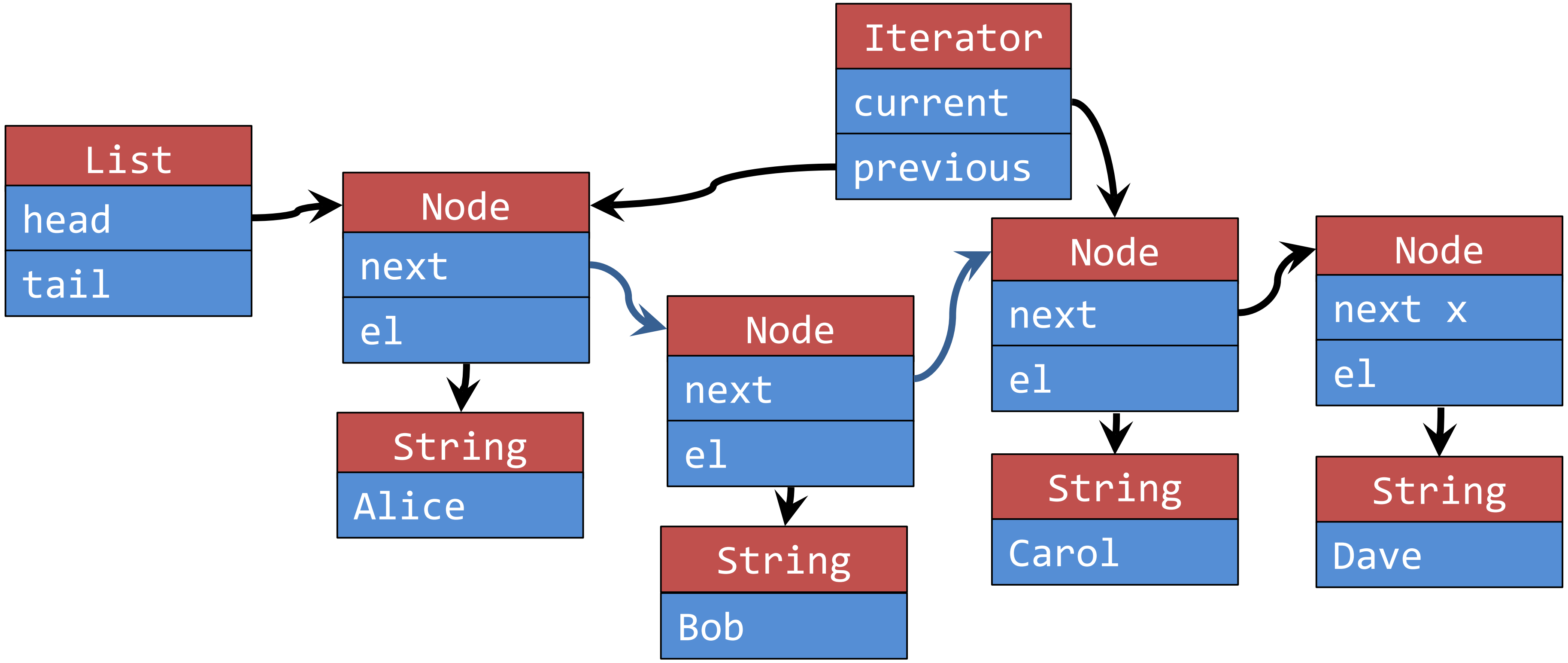
idea:

- 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
- only helps if you have to handle all elements *anyway*

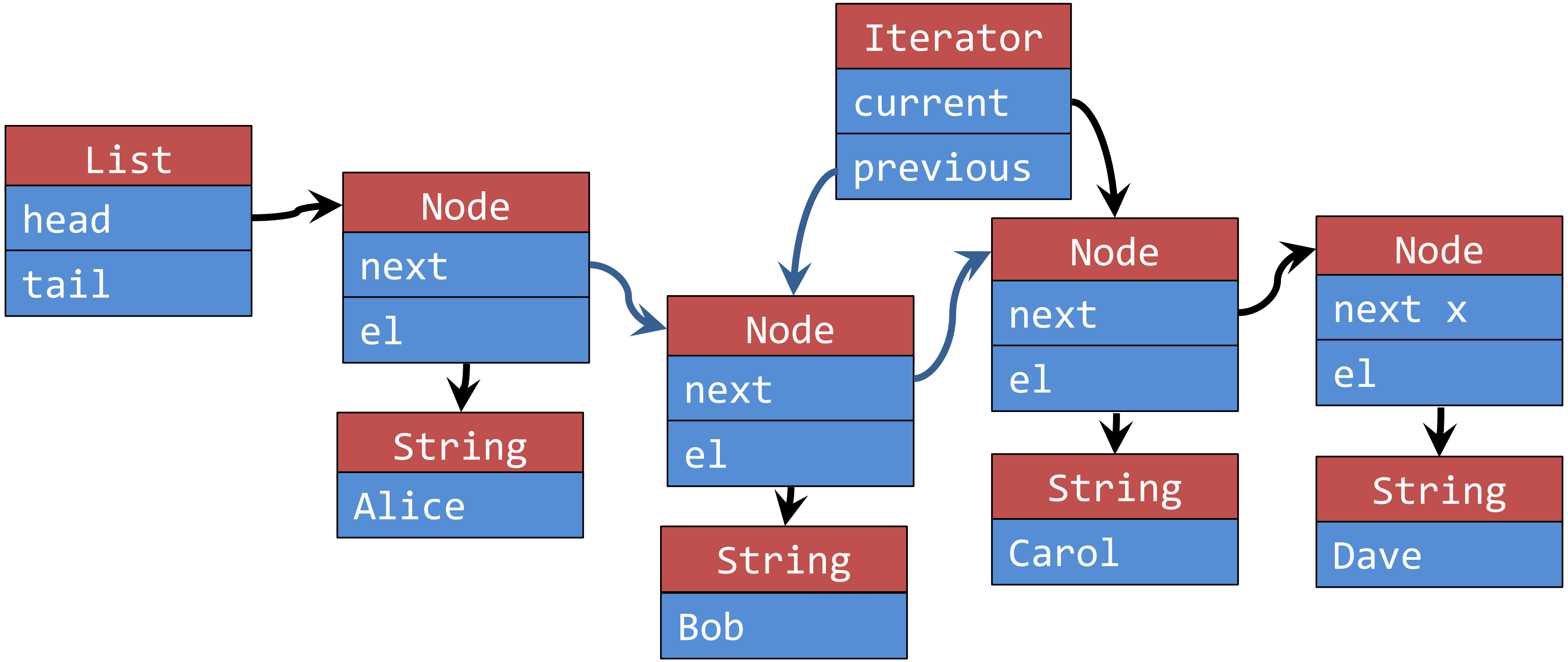
ListIterator add



ListIterator add



ListIterator add



MyLinkedList: ListIterator.add(element)

```
@Override
public void add(E e) {
    if (previous == null) {
        throw new IllegalStateException();
    } else {
        checkVersion();
```

MyLinkedList: ListIterator.add(element)

```
@Override
public void add(E e) {
    if (previous == null) {
        throw new IllegalStateException();
    } else {
        checkVersion();
        previous.next = new Node(e, current);
        previous = previous.next;
```

MyLinkedList: ListIterator.add(element)

```
@Override
public void add(E e) {
    if (previous == null) {
        throw new IllegalStateException();
    } else {
        checkVersion();
        previous.next = new Node(e, current);
        previous = previous.next;
        if (pos == size - 1)
            tail = previous;
        size += 1;
        pos += 1;
        modCount += 1;
        knownMod = modCount;
    }
}
```

ListIterator evaluation

the `ListIterator` solves many problems with O(N) access

- `add(E e)` and `set(E e)` at current position in O(1)

ListIterator evaluation

the `ListIterator` solves many problems with O(N) access

- `add(E e)` and `set(E e)` at current position in O(1)

a limitation is that we can only move from head to tail

- why not add a `previous()` as counterpart of `next()`
- the `ListIterator` interface provides this

MyLinkedList: inefficient `ListIterator.previous()`

```java
@Override
public E previous() {
    if (previous == null)
        throw new IllegalStateException();
    else {
        E e = previous.el;
        current = previous;
        pos -= 1;
        previous = getNode(pos);
        return e;
    }
}
```

MyLinkedList: inefficient ListIterator.previous()

```
@Override
public E previous() {
    if (previous == null)
        throw new IllegalStateException();
    else {
        E e = previous.el;
        current = previous;
        pos -= 1;
        previous = getNode(pos);
        return e;
    }
}
```

O(N)

MyLinkedList: inefficient `ListIterator.previous()`

```
@Override
public E previous() {
    if (previous == null)
        throw new IllegalStateException();
    else {
        E e = previous.el;
        current = previous;
        pos -= 1;
        previous = getNode(pos);
        return e;
    }
}
```

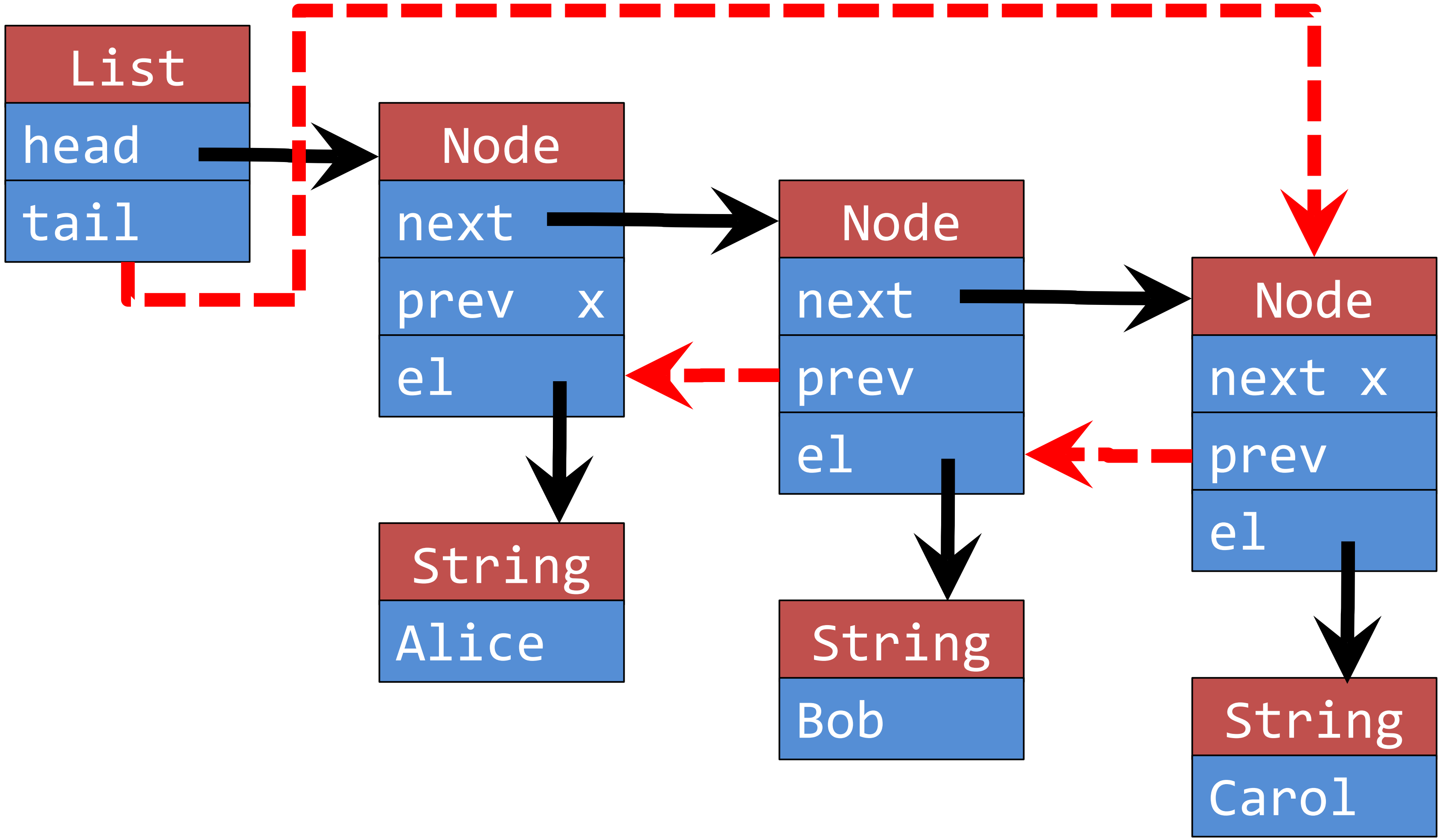
O(N)

a doubly linked list solves this problem:

- add references to the previous node

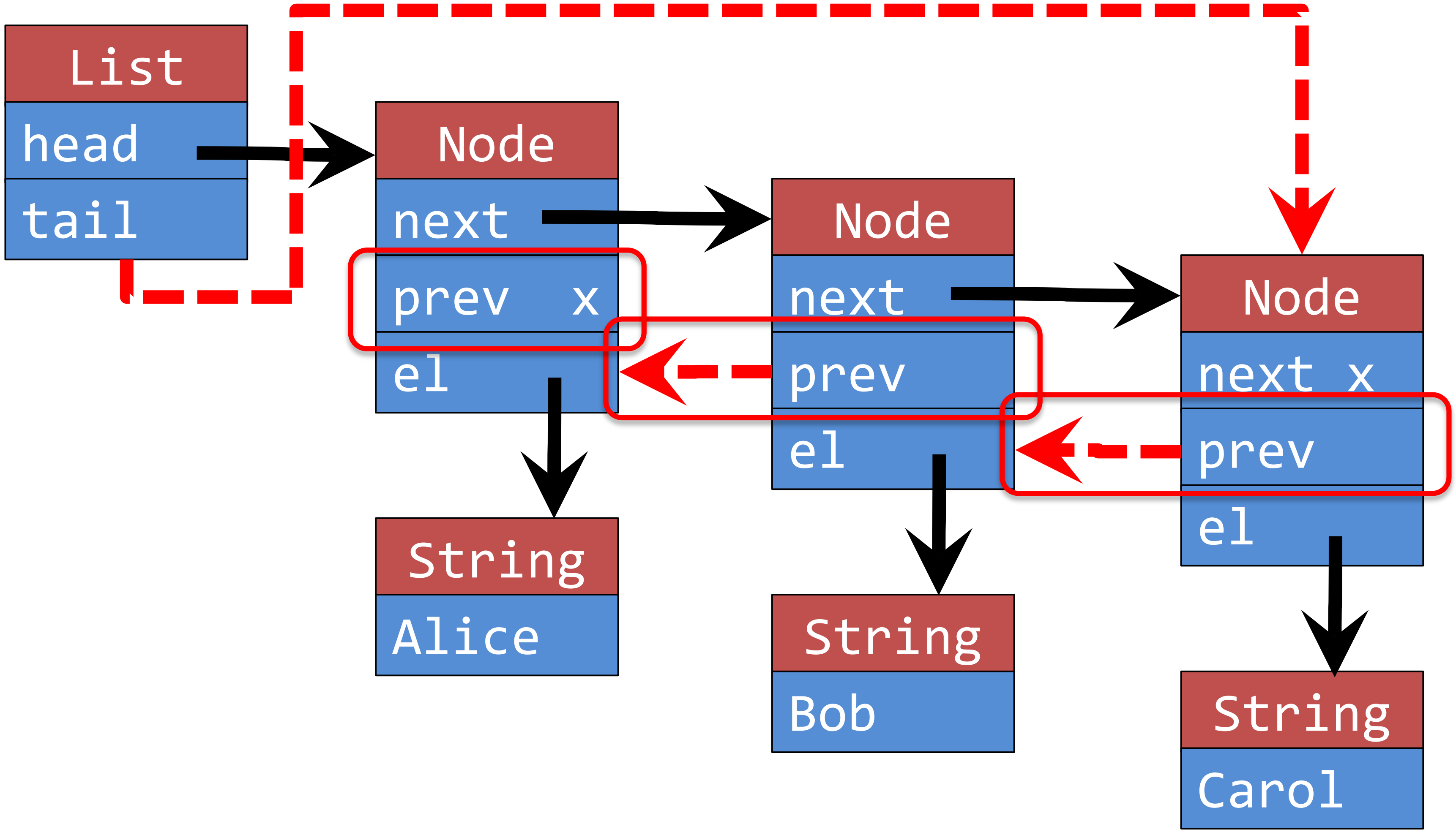
Doubly Linked List

basic idea:



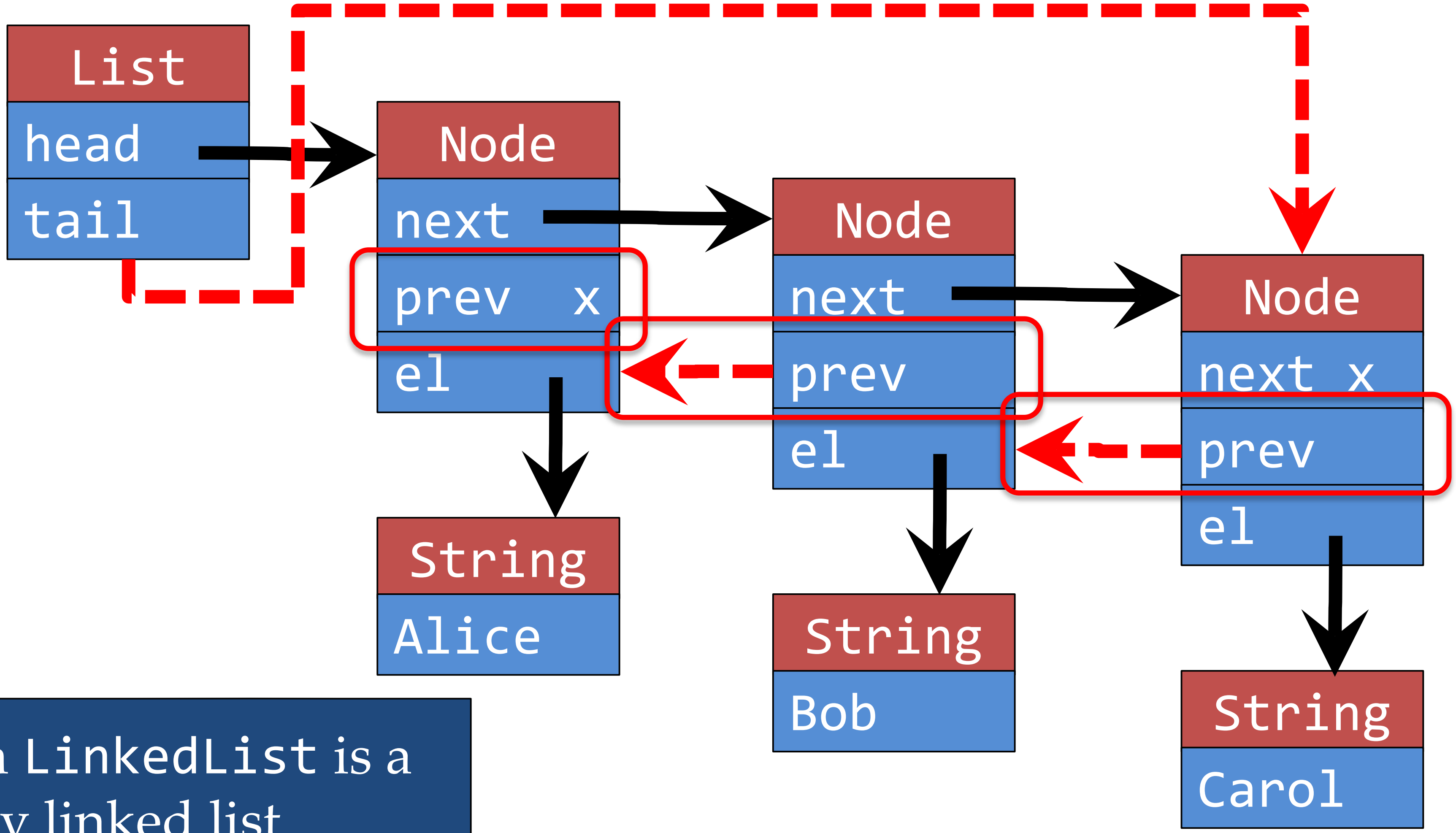
Doubly Linked List

basic idea:



Doubly Linked List

basic idea:



The built-in LinkedList is a doubly linked list

GENERIC RECURSIVE TYPE WITH *MULTIPLE CHILDREN PER NODE*: TREE

trees

in the same spirit we can make nodes with two successors (children)

- or even 3 or *n* children
- binary trees (2 children) is most common

trees

in the same spirit we can make nodes with two successors (children)

- or even 3 or *n* children
- binary trees (2 children) is most common

these data structures are called ***trees***

- sometimes we use different kinds of nodes
  e.g. Leaf (no children) and Fork (with children)

trees

in the same spirit we can make nodes with two successors (children)

- or even 3 or n children
- binary trees (2 children) is most common

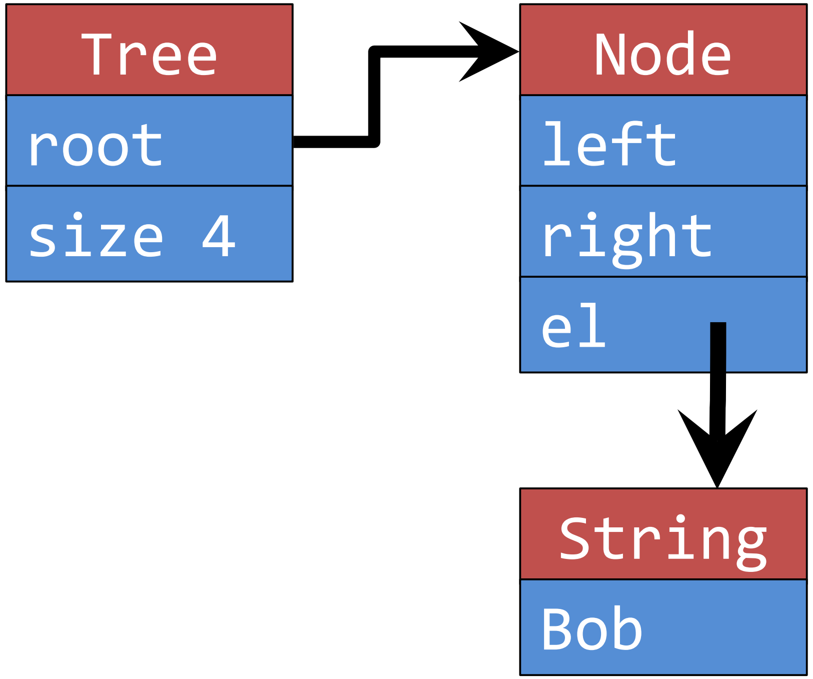
these data structures are called ***trees***

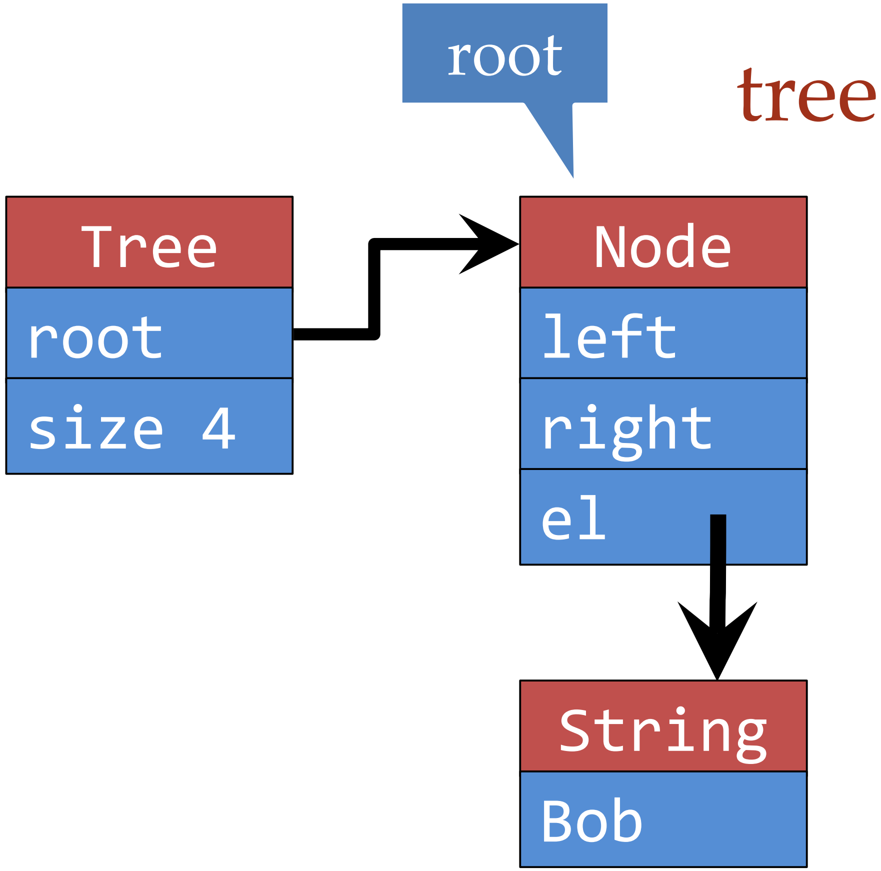
- sometimes we use different kinds of nodes
  e.g. Leaf (no children) and Fork (with children)

a frequently used variant is binary ***search*** tree

- (at most) two children
- all elements in the left subtree are smaller than element in node
- all elements in right subtree are bigger
- hence we have no duplicates

tree





root

tree

Tree

root

size 4

Node

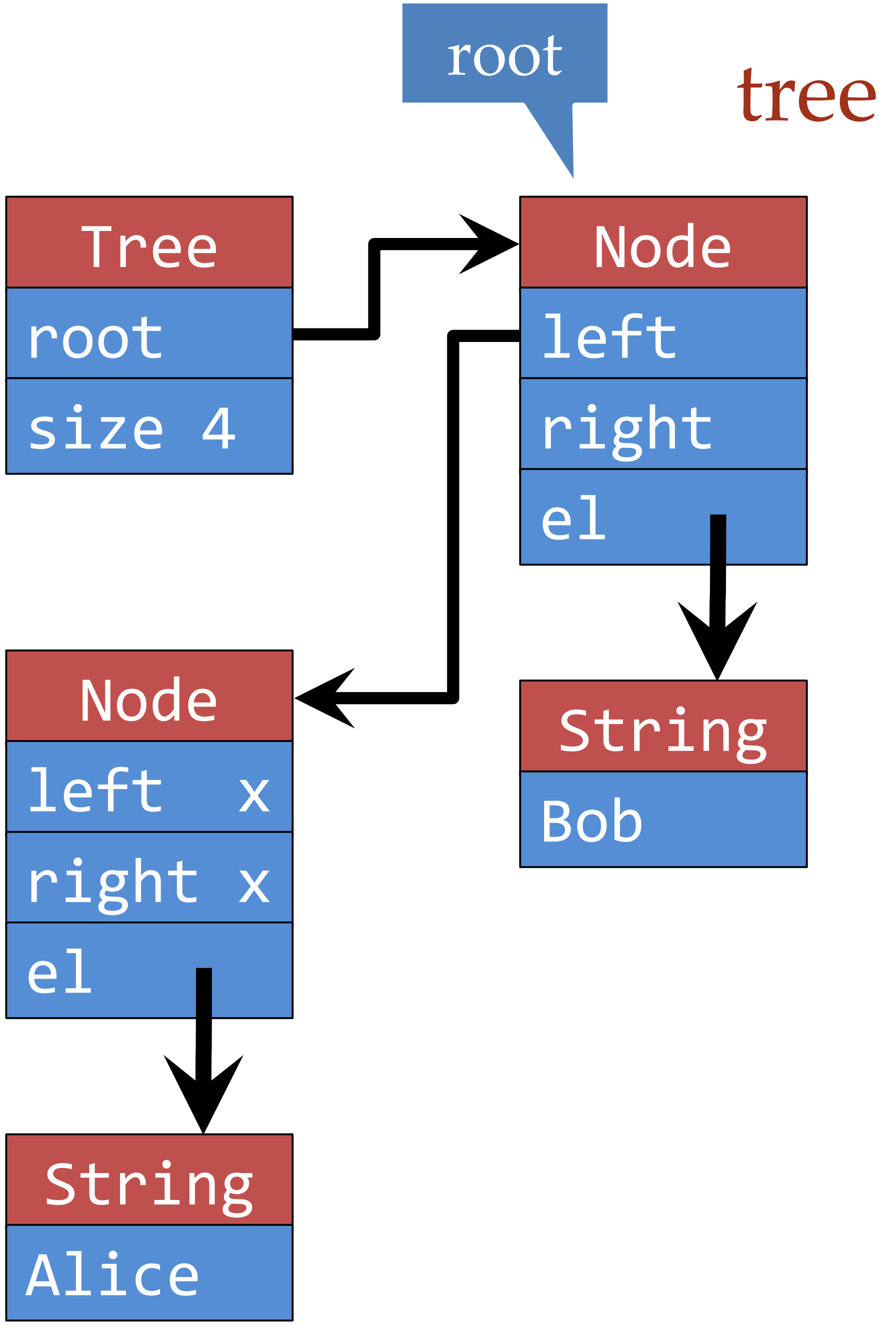
left

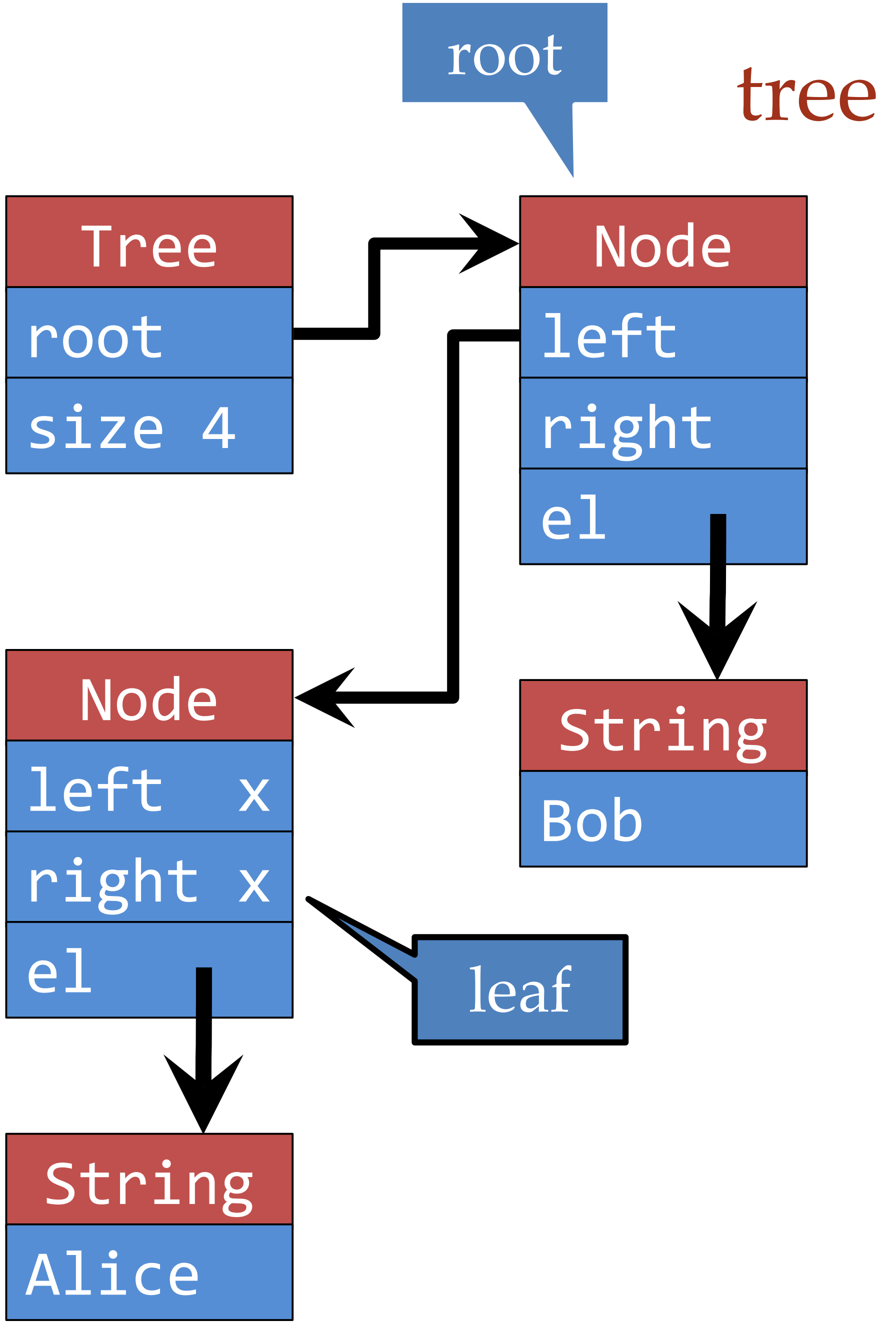
right

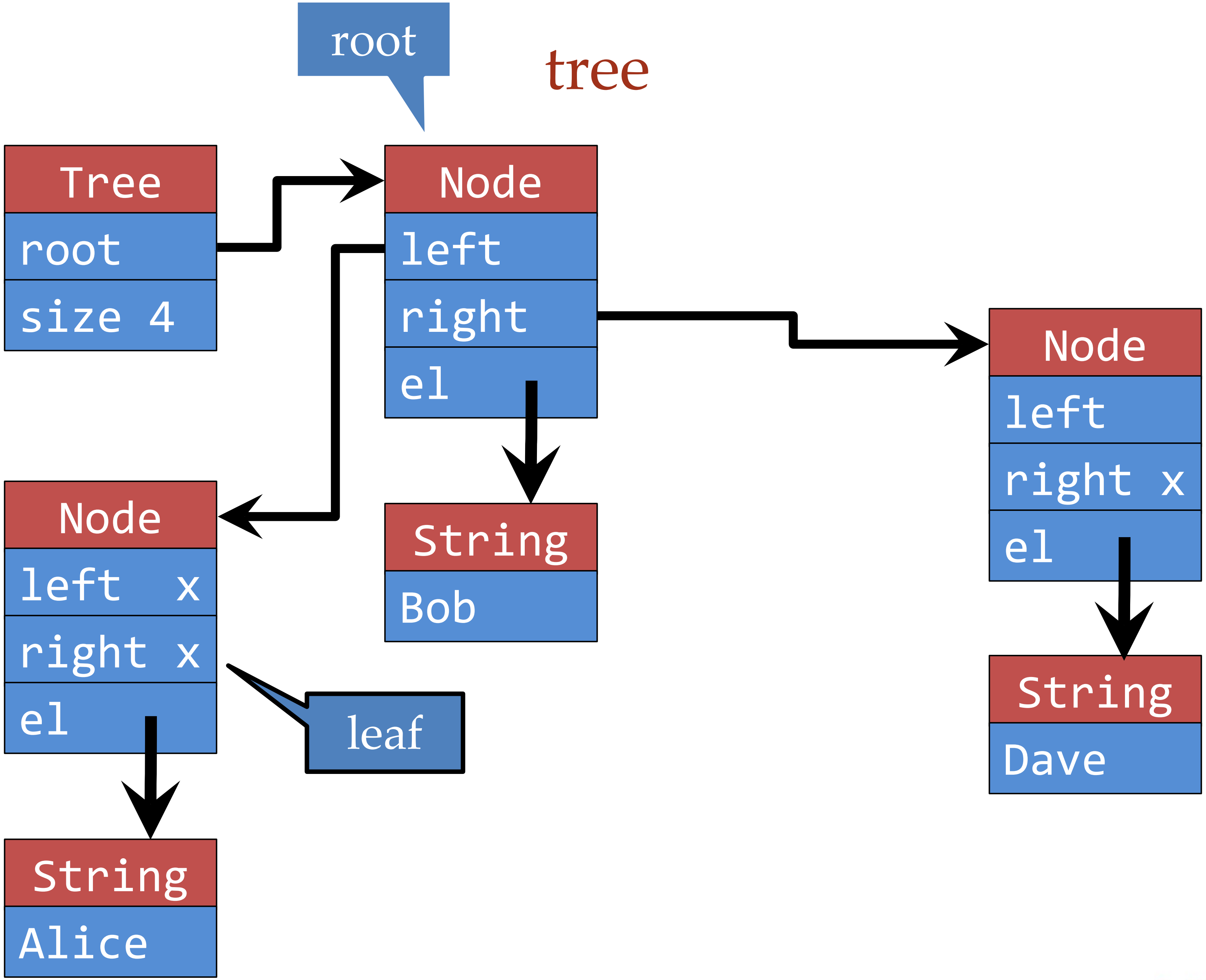
el

String

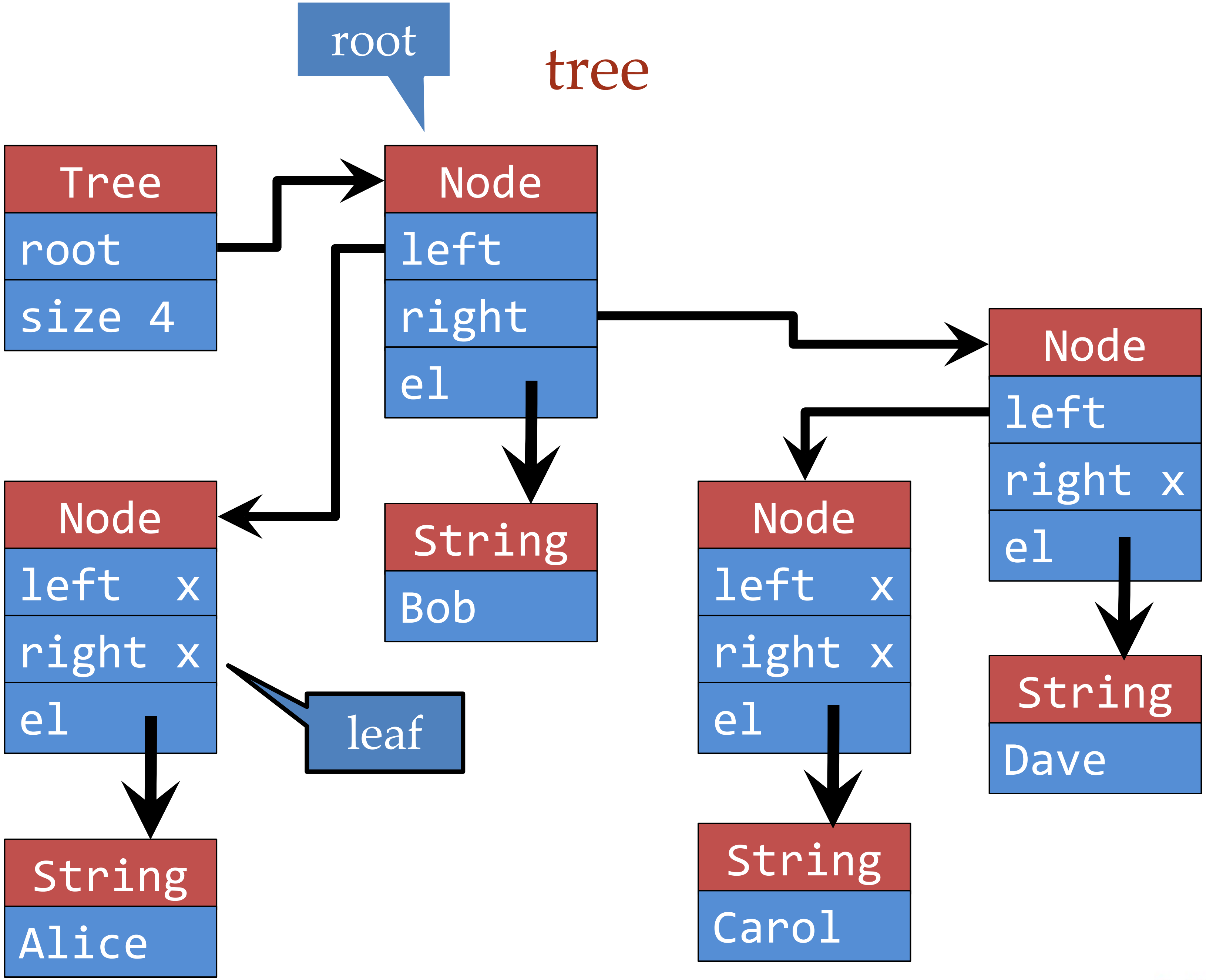
Bob

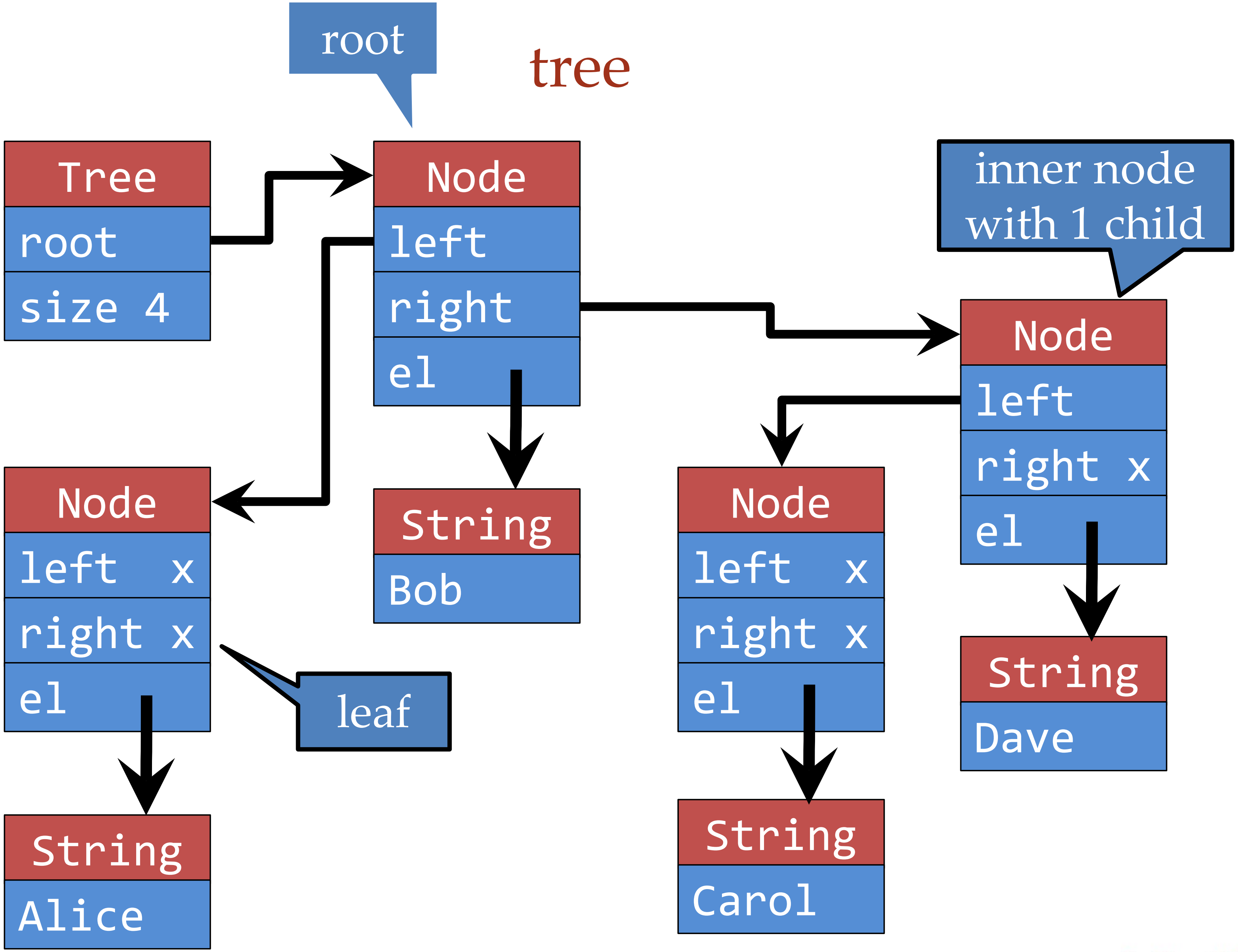

root
tree
Tree
root
size 4
Node
left
right
el
String
Bob
Node
left x
right x
el
String
Alice

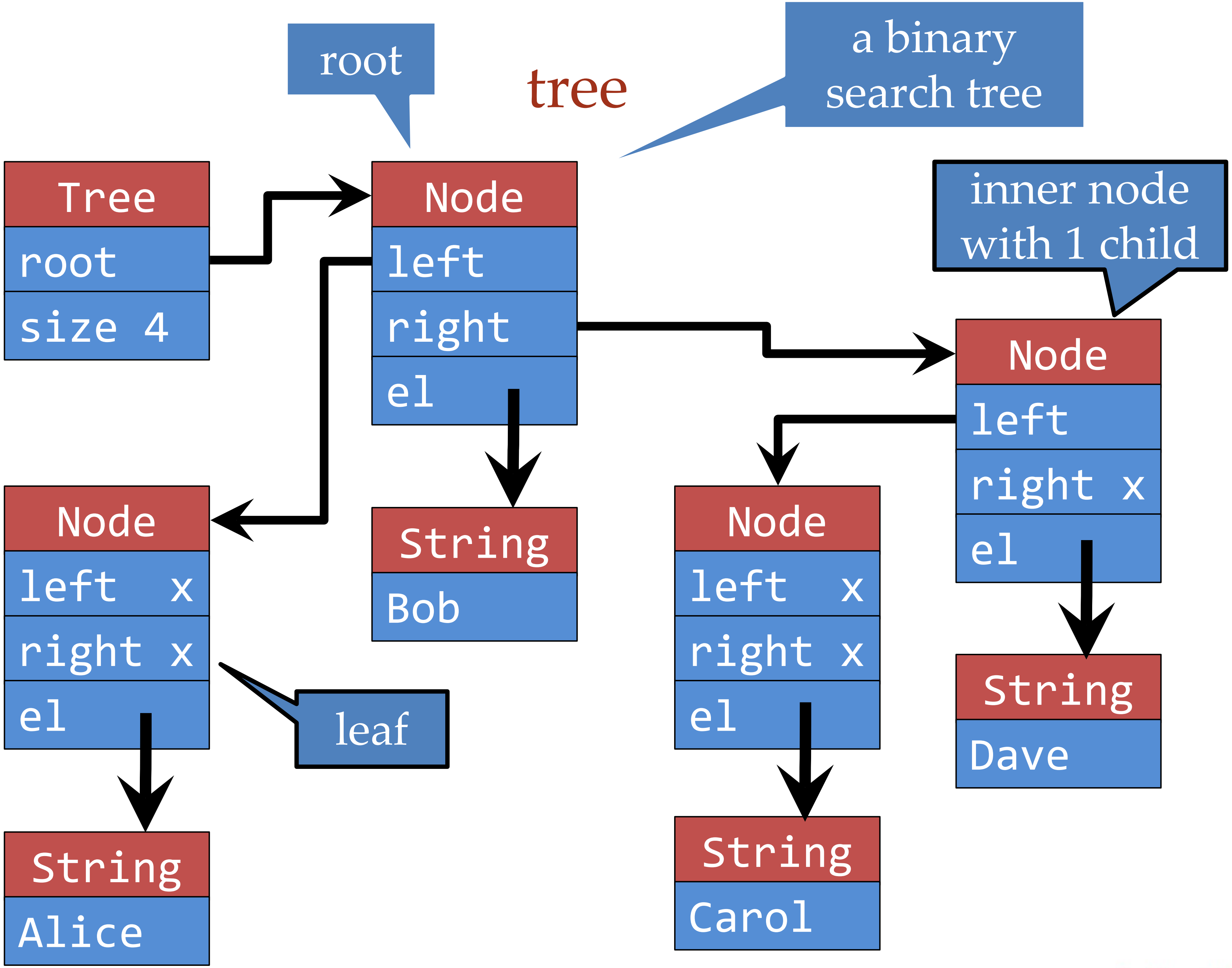

root
tree
Tree
root
size 4
Node
left
right
el
Node
left x
right x
el
String
Bob
leaf
String
Alice



root
tree
Tree
root
size 4
Node
left
right
el
Node
left
right x
el
Node
left x
right x
el
leaf
String
Bob
String
Dave
String
Alice


root
tree
Tree
root
size 4
Node
left
right
el
String
Bob
Node
left
right x
el
String
Dave
Node
left x
right x
el
String
Alice
leaf
Node
left x
right x
el
String
Carol


root
tree
Tree
root
size 4
Node
left
right
el
inner node with 1 child
Node
left
right x
el
Node
left x
right x
el
leaf
String
Bob
Node
left x
right x
el
String
Dave
String
Alice
String
Carol



root

tree

a binary search tree

inner node with 1 child

leaf

binary search tree

```
public class Tree <E extends Comparable<E>> {

    protected Node root;

    private class Node {
        E el;
        Node left, right;
        public Node (E e, Node l, Node r) {
            el = e;
            left = l;
            right = r;
        }
        public Node (E e) {
            this(e, null, null);
        }
    }
```

binary search tree

```
public class Tree <E extends Comparable<E>> {

    protected Node root;

    private class Node {
        E el;
        Node left, right;
        public Node (E e, Node l, Node r) {
            el = e;
            left = l;
            right = r;
        }
        public Node (E e) {
            this(e, null, null);
        }
    }
```

ensures comparability of elements

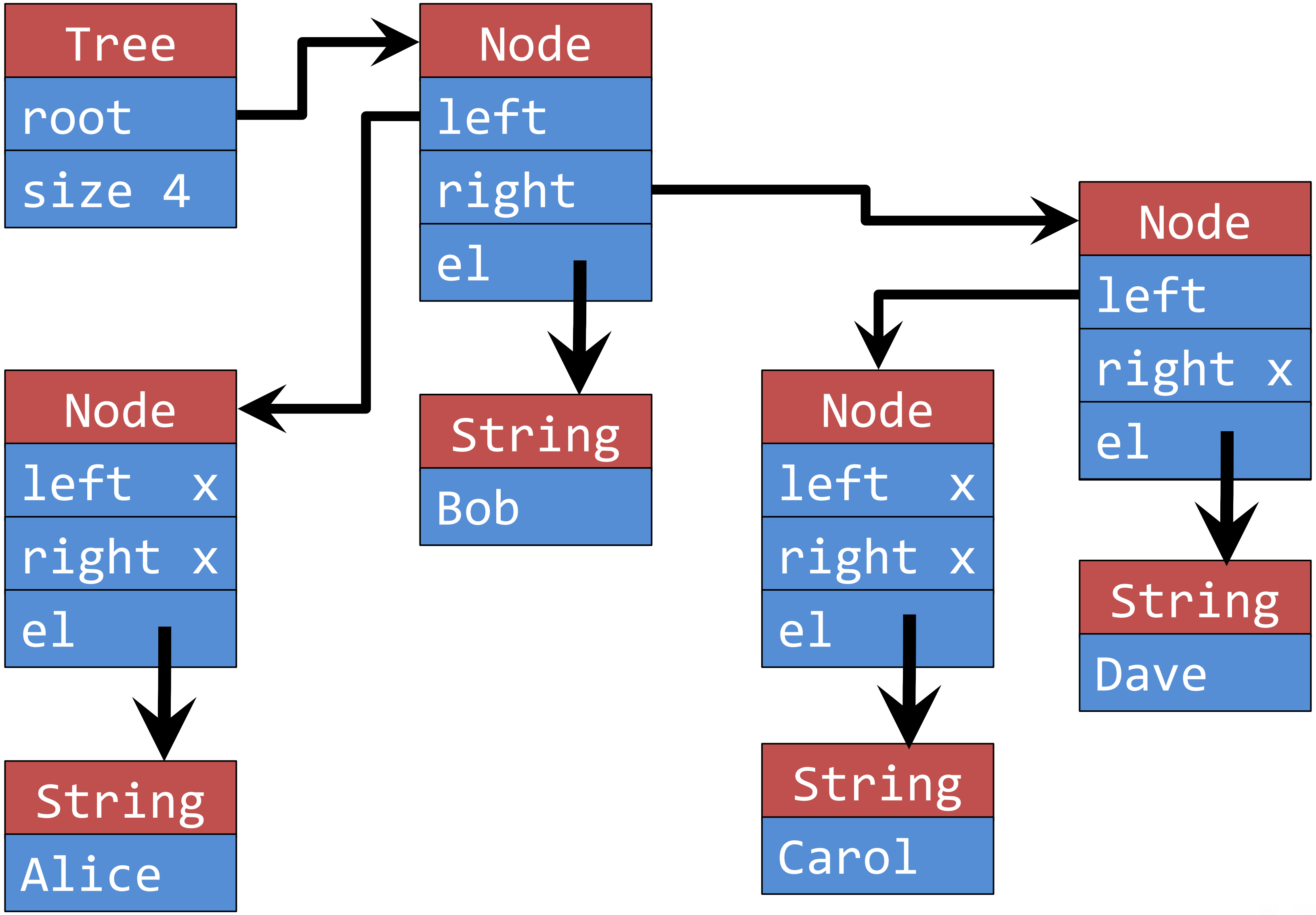
binary search tree

```
public class Tree <E extends Comparable<E>> {

    protected Node root;

    private class Node {
        E el;
        Node left, right;
        public Node (E e, Node l, Node r) {
            el = e;
            left = l;
            right = r;
        }
        public Node (E e) {
            this(e, null, null);
        }
    }
```

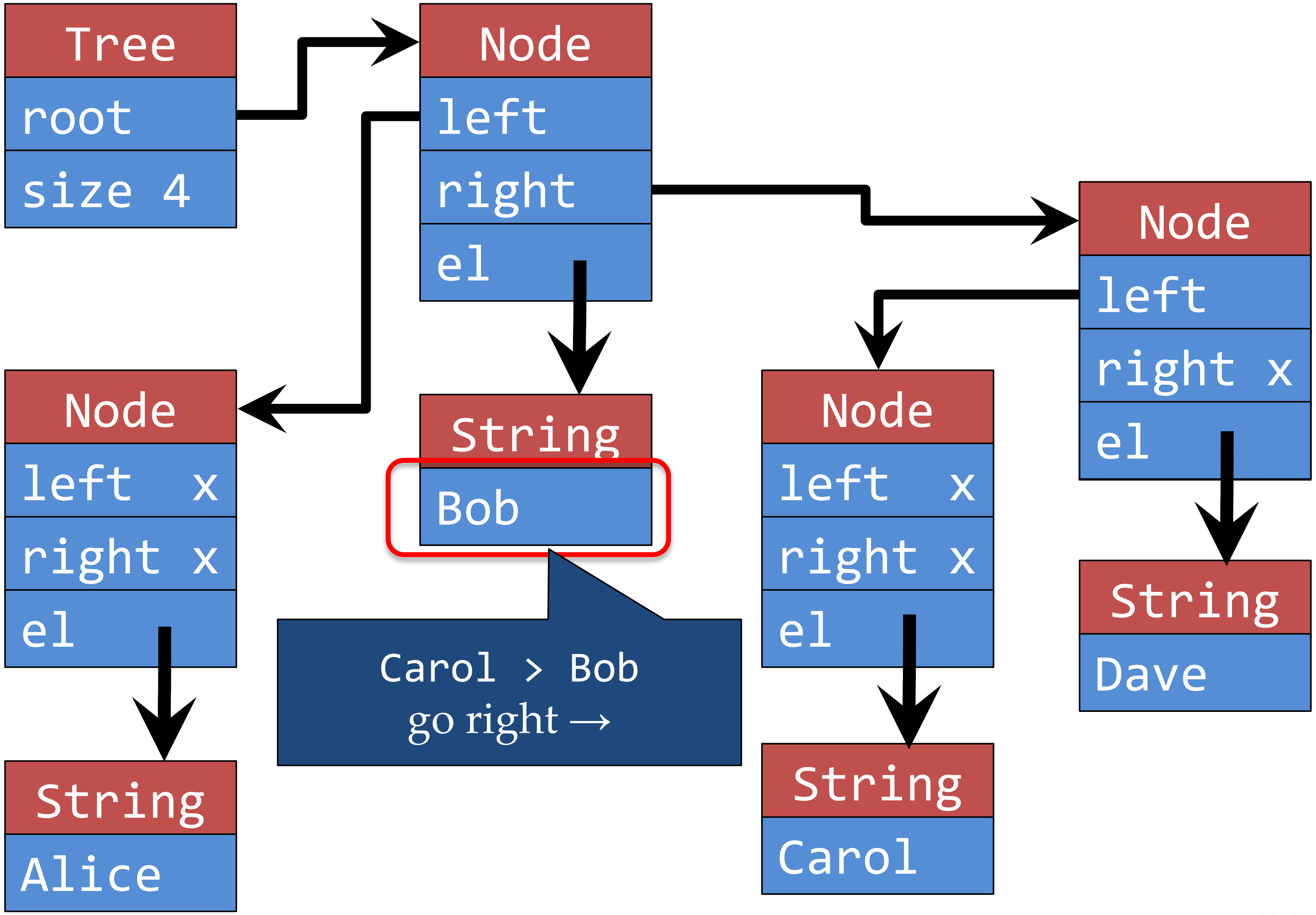
ensures comparability of elements

very similar to Linked List, only with two children

tree: search for Carol

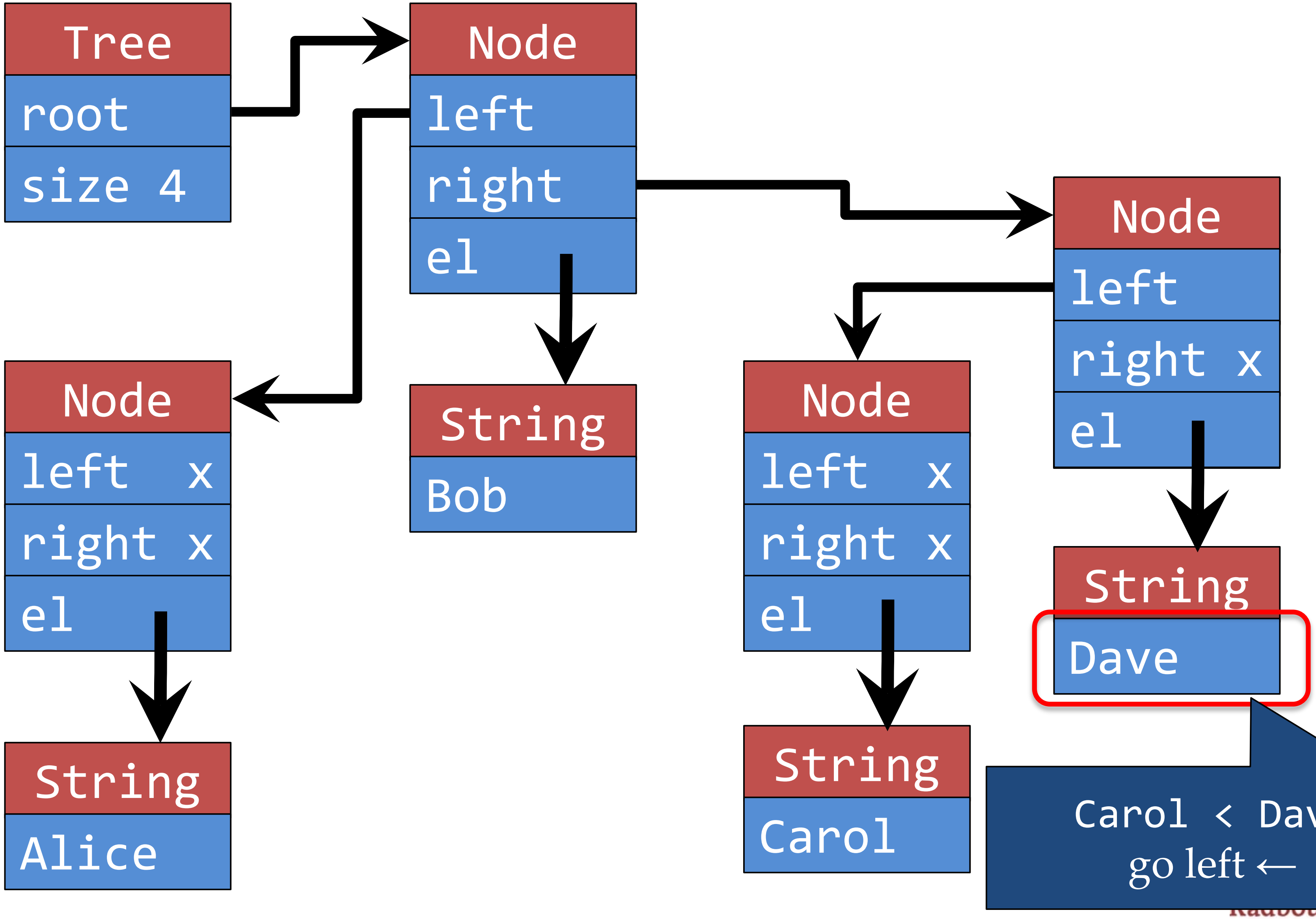


tree: search for Carol



Carol > Bob
go right →

tree: search for Carol



Tree
root
size 4

Node
left
right
el

String
Bob

Node
left
right x
el

String
Dave

Node
left x
right x
el

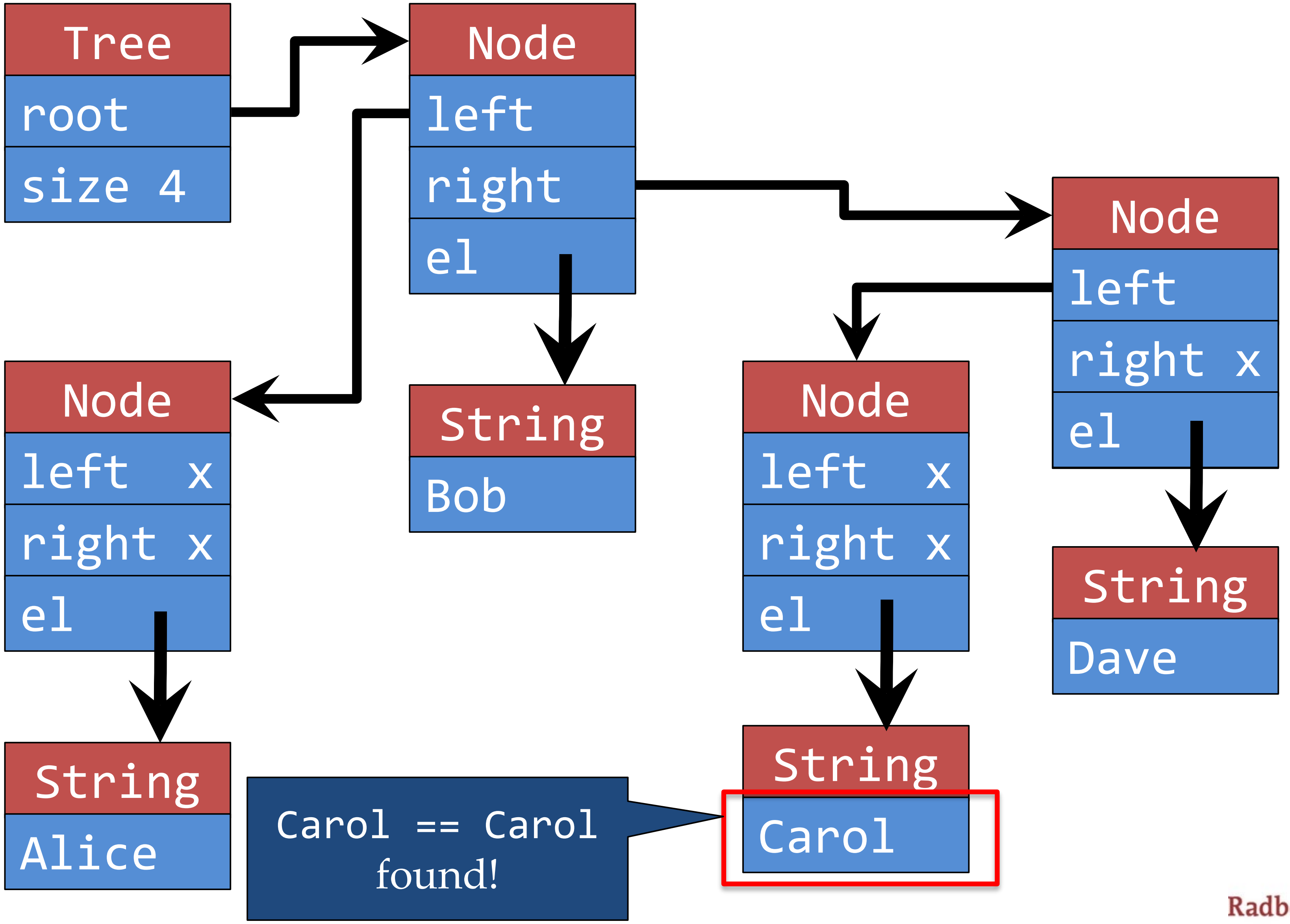
String
Alice

Node
left x
right x
el

String
Carol

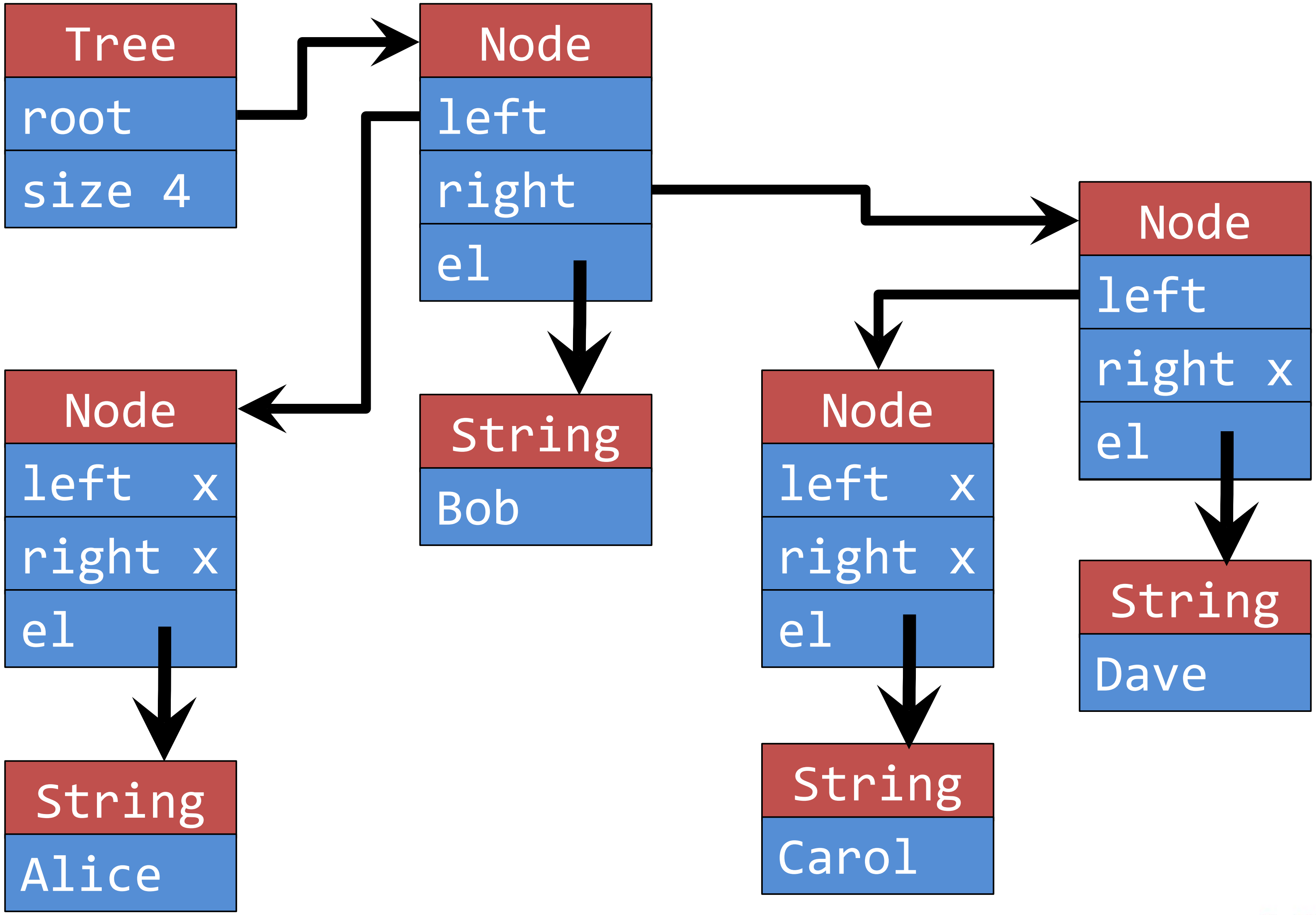
Carol < Dave
go left ←

tree: search for Carol

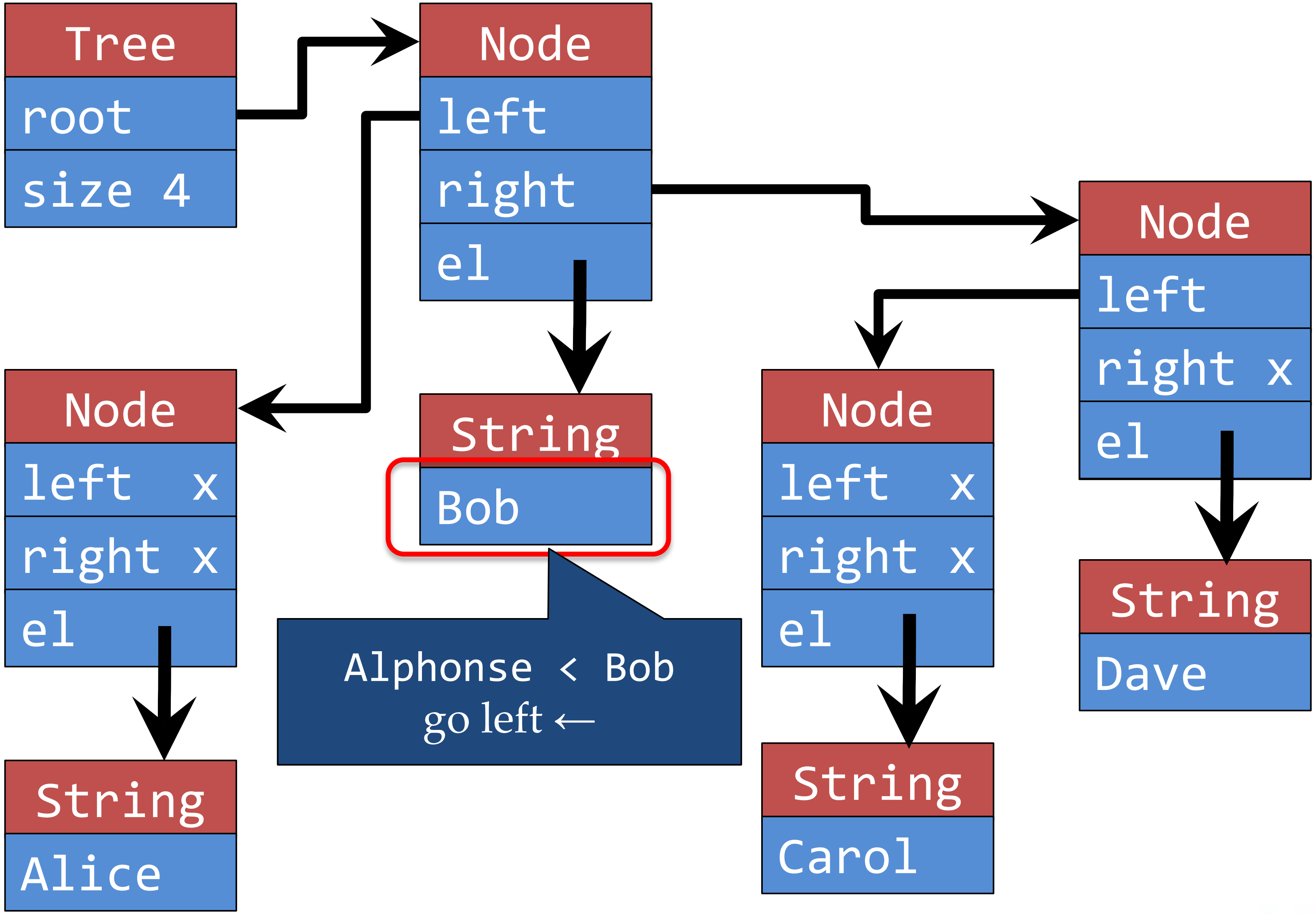


Carol == Carol
found!

tree: search for Alphonse

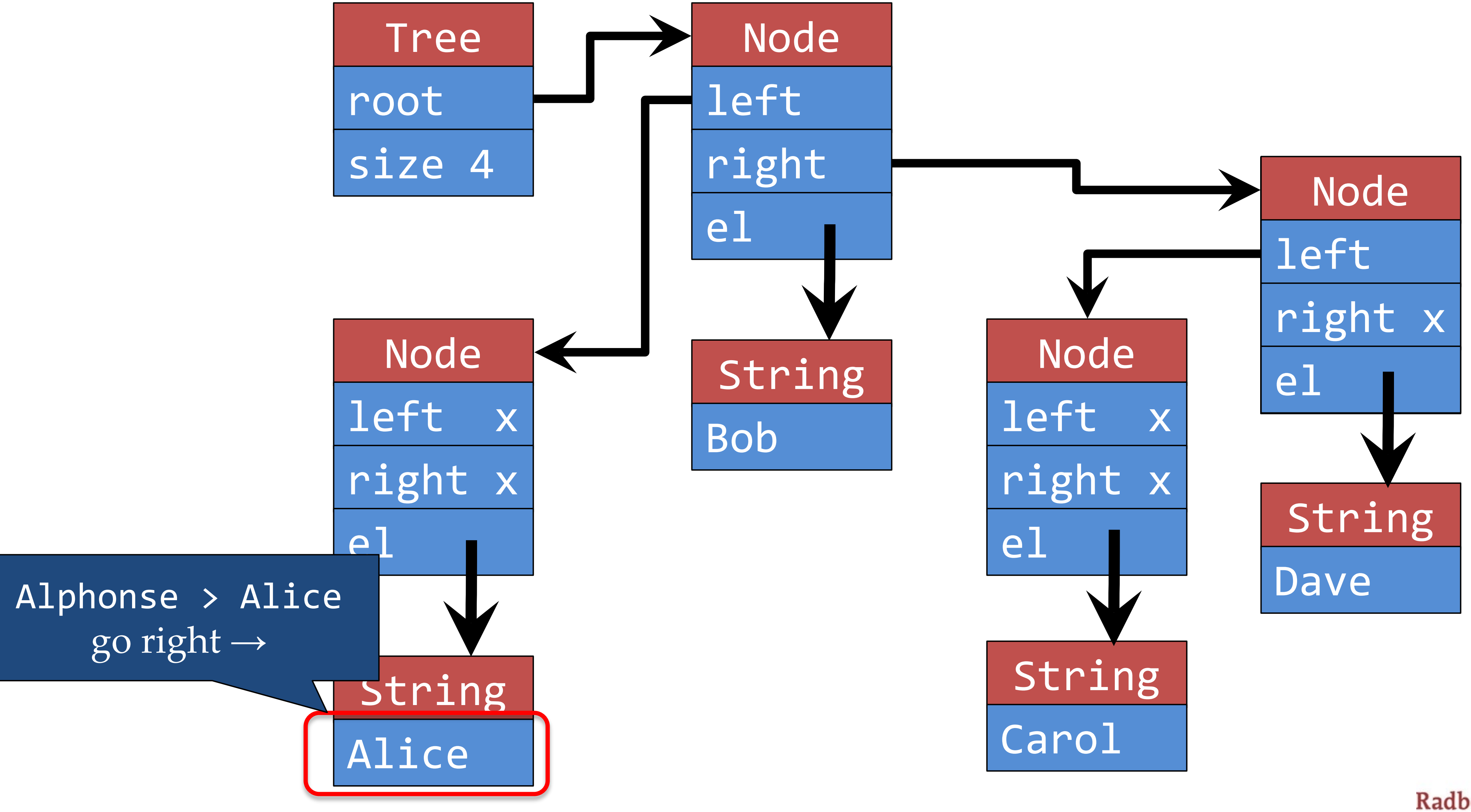


tree: search for Alphonse



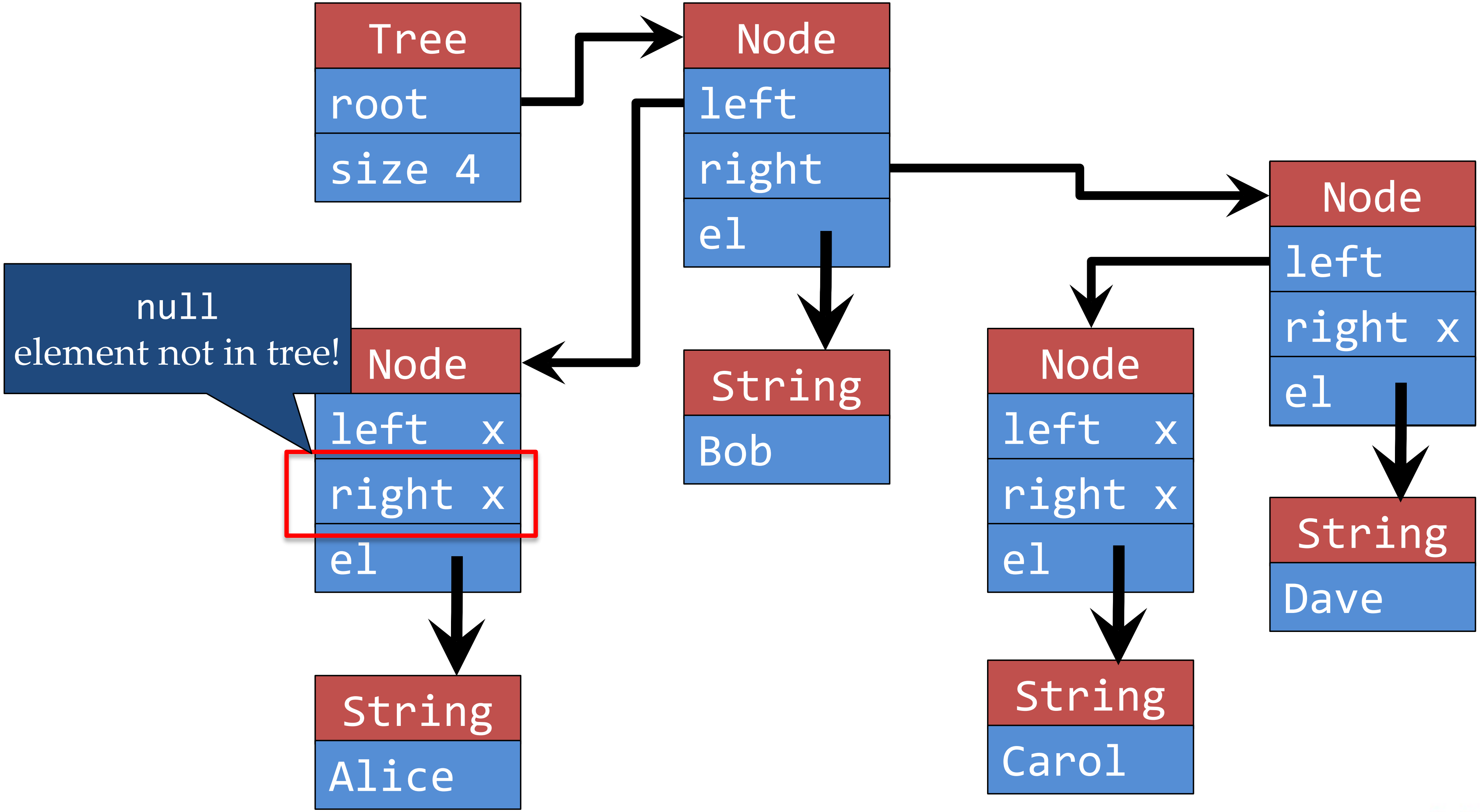
Alphonse < Bob
go left ←

tree: search for Alphonse



Alphonse > Alice
go right →

tree: search for Alphonse



search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
```

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
```

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
```

common pattern:
helper method with
reference to node in tree

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
```

common pattern:
helper method with
reference to node in tree

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
```

common pattern:
helper method with
reference to node in tree

empty subtree: element
does not occur

search in search tree

```java
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
    } else {
        int comp = e.compareTo(n.el);
        if (comp < 0) {
            return has (n.left, e);
```

common pattern:
helper method with
reference to node in tree

empty subtree: element
does not occur

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
    } else {
        int comp = e.compareTo(n.el);
        if (comp < 0) {
            return has (n.left, e);
```

common pattern:
helper method with
reference to node in tree

empty subtree: element
does not occur

smaller: search
left subtree

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
    } else {
        int comp = e.compareTo(n.el);
        if (comp < 0) {
            return has (n.left, e);
        } else if (comp == 0) {
            return true;
```

common pattern:
helper method with
reference to node in tree

empty subtree: element
does not occur

smaller: search
left subtree

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
    } else {
        int comp = e.compareTo(n.el);
        if (comp < 0) {
            return has (n.left, e);
        } else if (comp == 0) {
            return true;
```

common pattern:
helper method with
reference to node in tree

empty subtree: element
does not occur

smaller: search
left subtree

equal: found

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
    } else {
        int comp = e.compareTo(n.el);
        if (comp < 0) {
            return has (n.left, e);
        } else if (comp == 0) {
            return true;
        } else { // comp > 0
            return has (n.right, e);
        }
    }
}
```

common pattern:
helper method with
reference to node in tree

empty subtree: element
does not occur

smaller: search
left subtree

equal: found

search in search tree

```
public boolean has(E e) {
    return has(root, e);
}
private boolean has (Node n, E e) {
    if (n == null) {
        return false;
    } else {
        int comp = e.compareTo(n.el);
        if (comp < 0) {
            return has (n.left, e);
        } else if (comp == 0) {
            return true;
        } else { // comp > 0
            return has (n.right, e);
        }
    }
}
```

common pattern:
helper method with
reference to node in tree

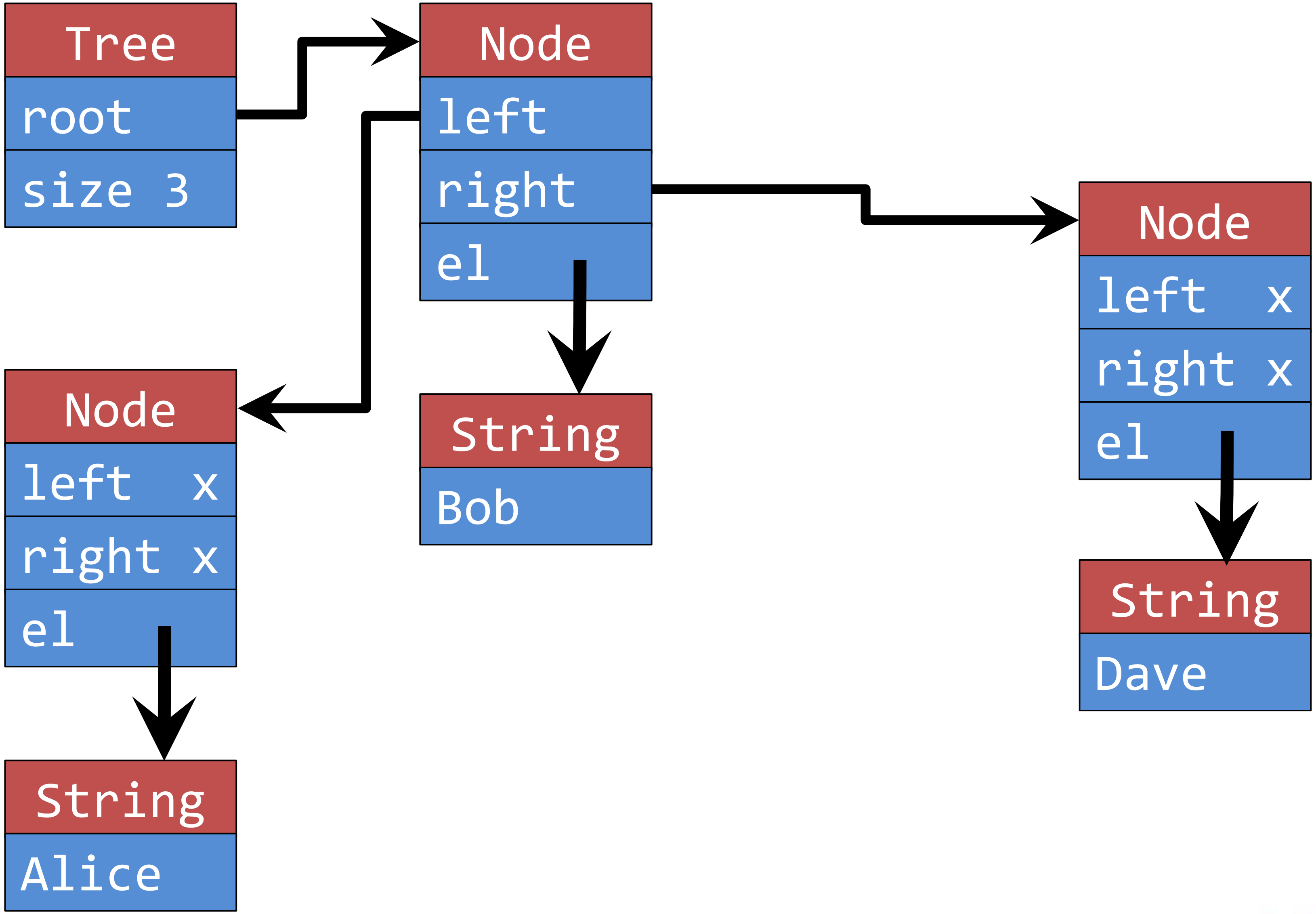
empty subtree: element
does not occur

smaller: search
left subtree

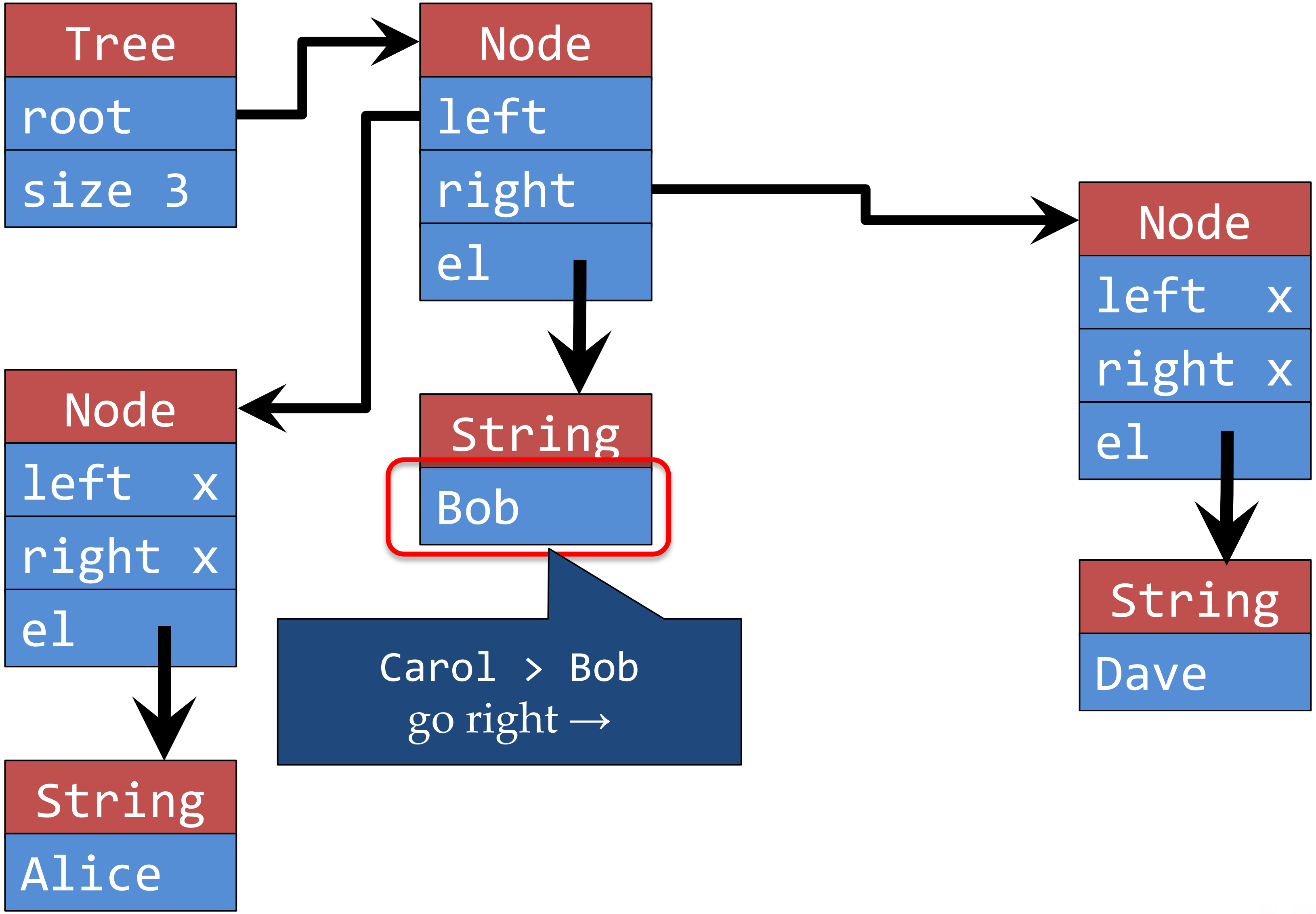
equal: found

bigger: search
right subtree

tree: adding Carol

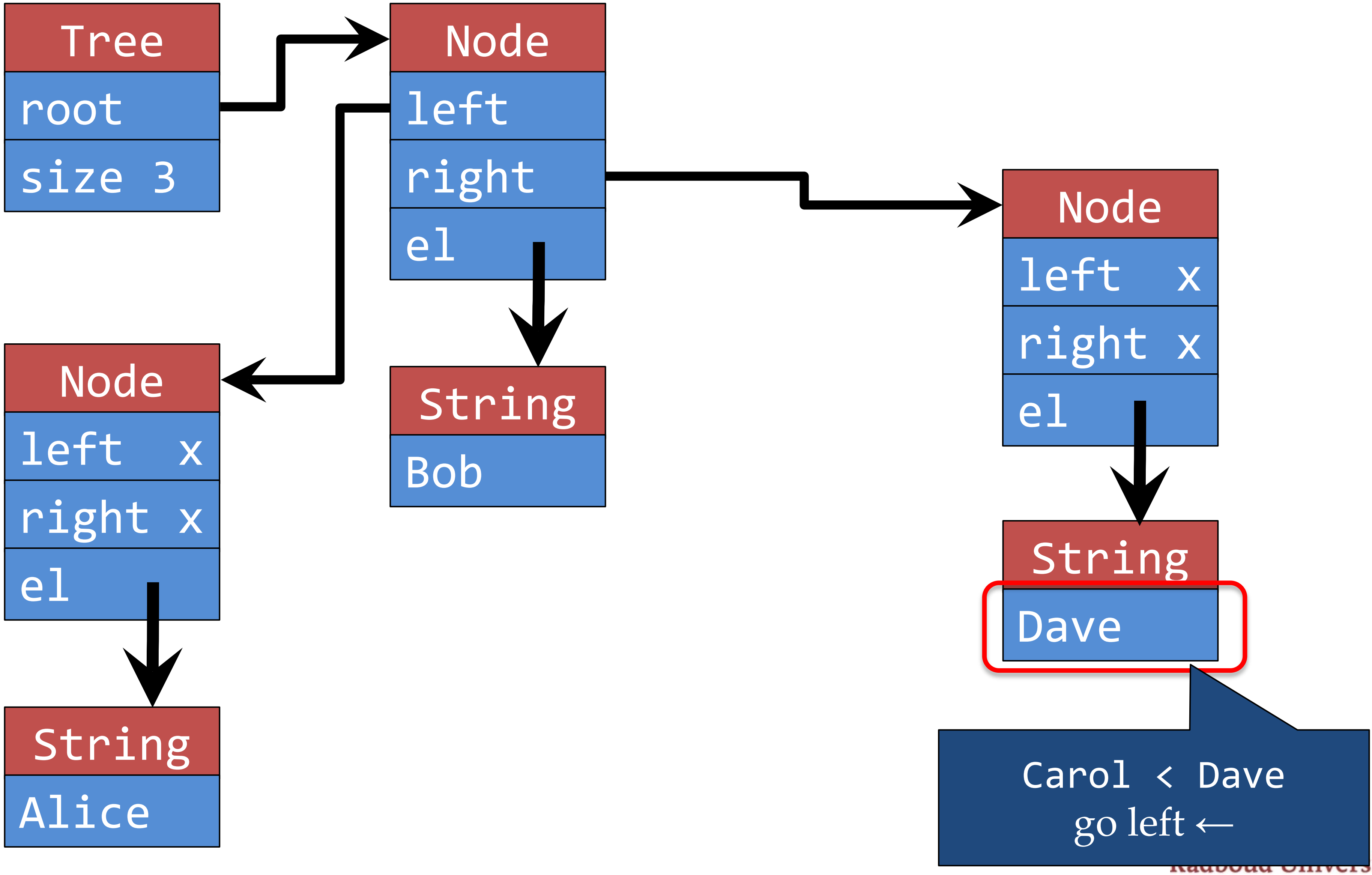


tree: adding Carol



Carol > Bob
go right ⟶

tree: adding Carol



Tree
root
size 3

Node
left
right
el

String
Bob

Node
left x
right x
el

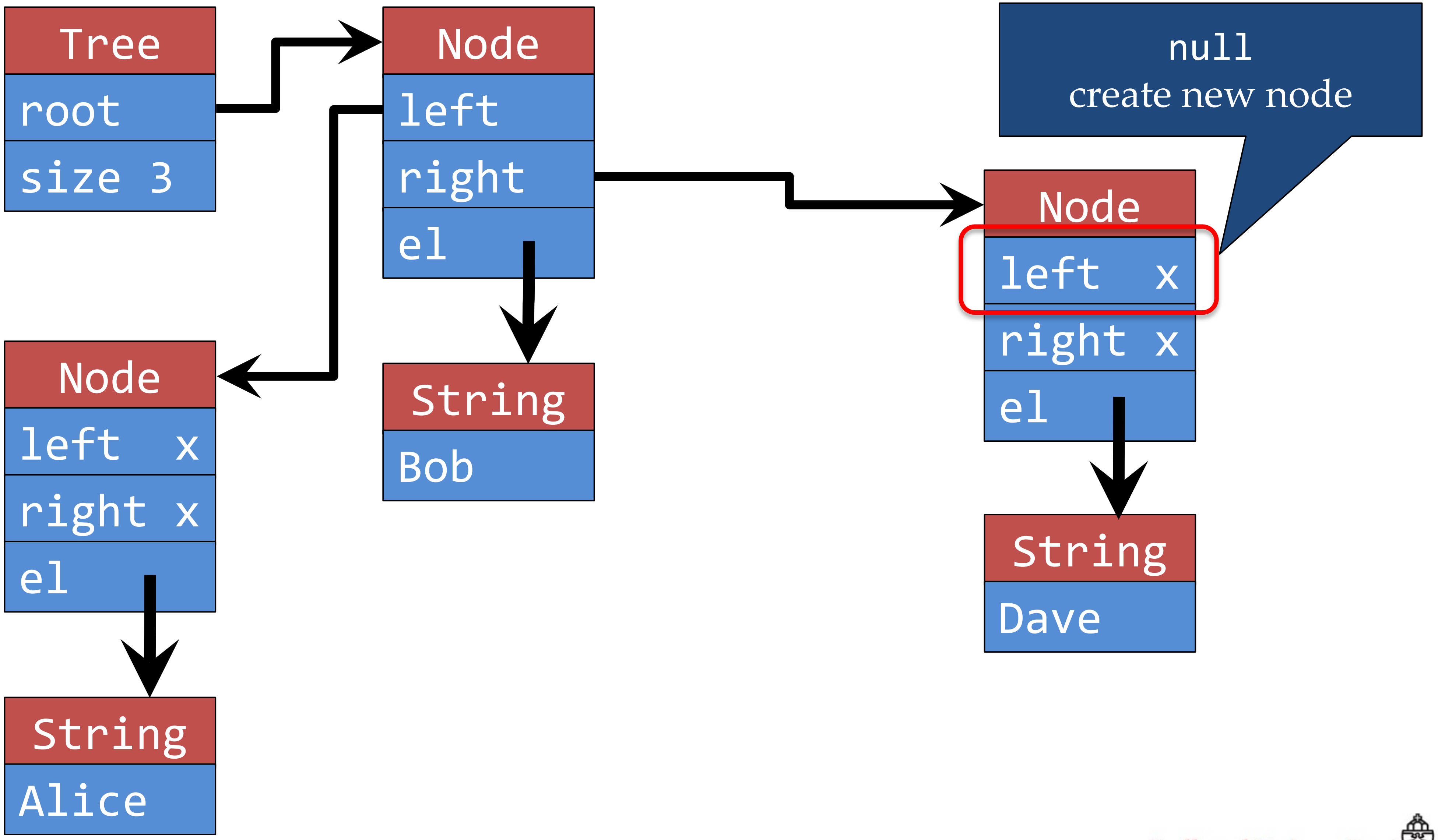
String
Alice

Node
left x
right x
el

String
Dave

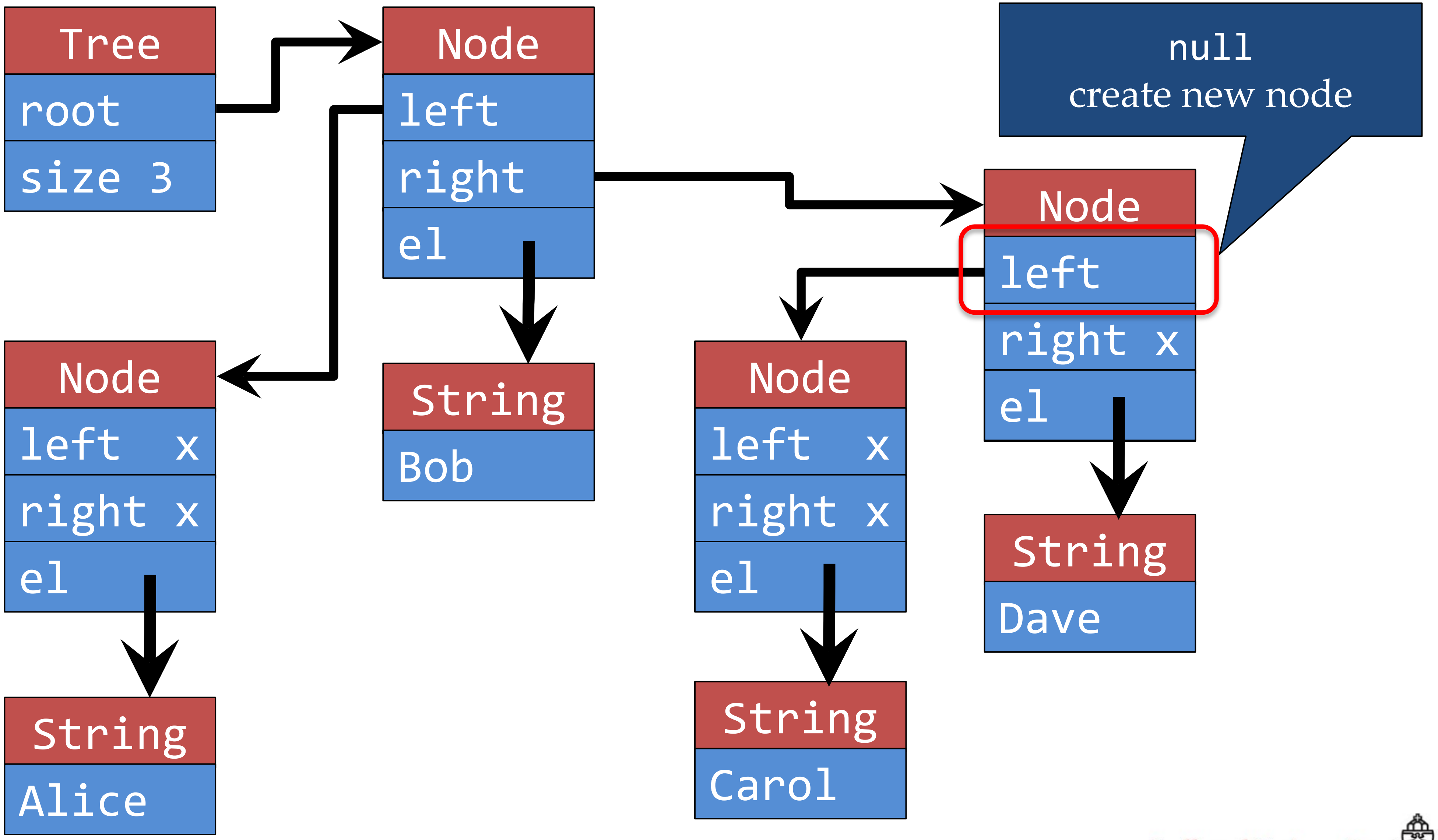
Carol < Dave
go left ←

tree: adding Carol



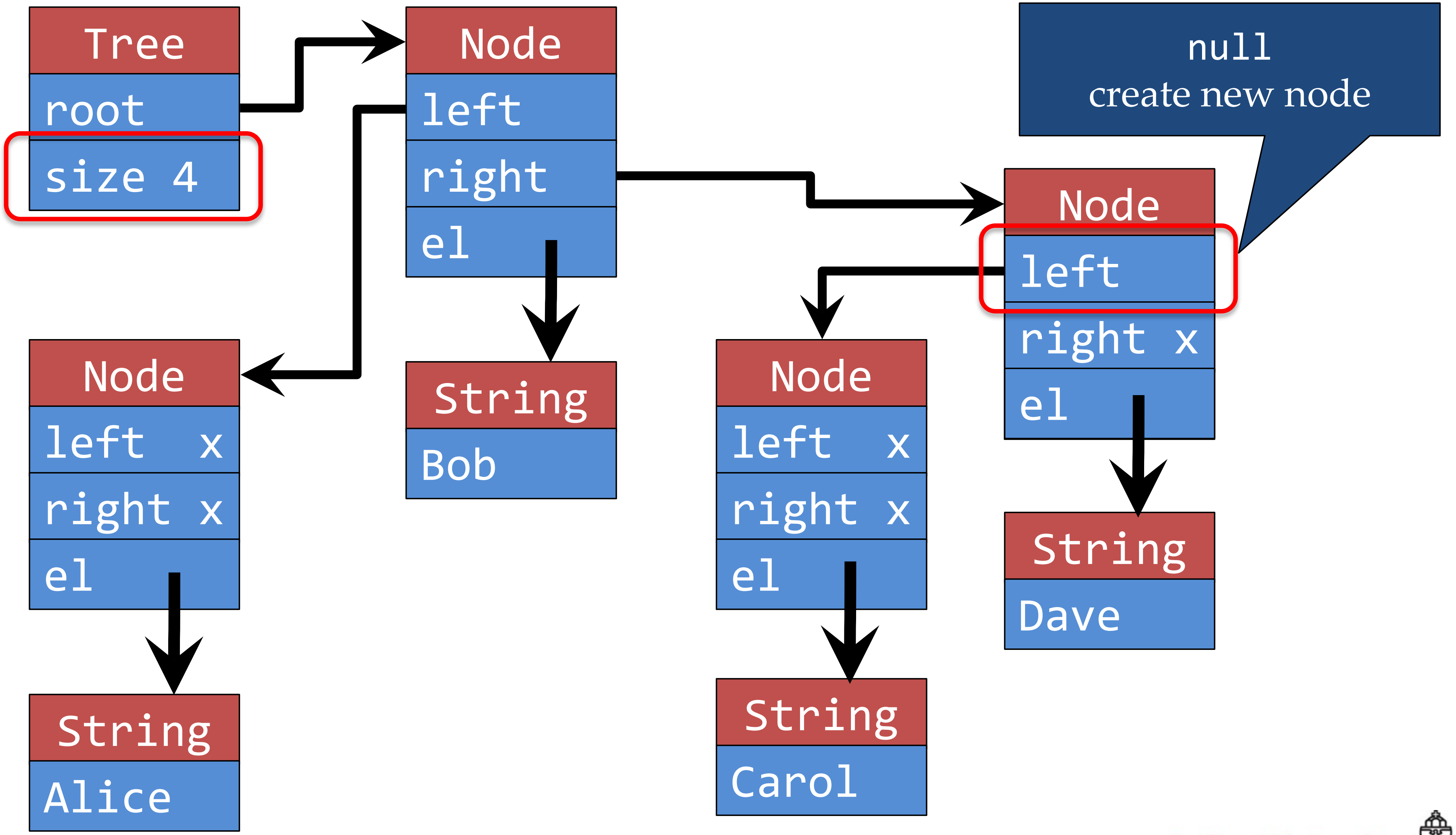
null
create new node

tree: adding Carol



null
create new node

tree: adding Carol



Tree
root
size 4

Node
left
right
el

Node
left x
right x
el

String
Alice

String
Bob

Node
left x
right x
el

String
Carol

Node
left
right x
el

String
Dave

null
create new node

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
```

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
```

a very similar structure: helper method with reference in tree

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
private boolean add(Node n, E e) {
    int comp = e.compareTo(n.el);
    if (comp < 0) {
```

a very similar structure: helper method with reference in tree

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
private boolean add(Node n, E e) {
    int comp = e.compareTo(n.el);
    if (comp < 0) {
        if (n.left == null) {
            n.left = new Node (e);
            return true;
        } else
            return add(n.left, e);
```

a very similar structure: helper method with reference in tree

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
private boolean add(Node n, E e) {
    int comp = e.compareTo(n.el);
    if (comp < 0) {
        if (n.left == null) {
            n.left = new Node (e);
            return true;
        } else
            return add(n.left, e);
    } else if (comp == 0) {
        return false;
```

a very similar structure: helper method with reference in tree

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
private boolean add(Node n, E e) {
    int comp = e.compareTo(n.el);
    if (comp < 0) {
        if (n.left == null) {
            n.left = new Node (e);
            return true;
        } else
            return add(n.left, e);
    } else if (comp == 0) {
        return false;
```

a very similar structure: helper method with reference in tree

no duplicates

add to a search tree

```
public boolean add(E e) {
    if (root == null) {
        root = new Node(e);
        return true;
    } else
        return add(root, e);
}
private boolean add(Node n, E e) {
    int comp = e.compareTo(n.el);
    if (comp < 0) {
        if (n.left == null) {
            n.left = new Node (e);
            return true;
        } else
            return add(n.left, e);
    } else if (comp == 0) {
        return false;
    } else { // comp > 0
        if (n.right == null) {
            n.right = new Node (e);
            return true;
        } else
            return add (n.right, e);
    }
}
```

a very similar structure: helper method with reference in tree

no duplicates

recursive data-structure implementation pattern

there are many different recursive data-structures

- they differ in complexity of operations

there is a main (wrapper) class with a set of operations

- operations: access, search, insert, delete, ..
- generics to allow different type of elements
- 1 (or more) local (recursive) `class Node`
- `Node` contains (has references to) one or more other nodes
  - ➢ `null` if there is no other `Node`
- `Node` is never exposed to ensure integrity of constraints: **encapsulation**

recursive data-structure implementation pattern

there are many different recursive data-structures

- they differ in complexity of operations

there is a main (wrapper) class with a set of operations

- operations: access, search, insert, delete, ..
- generics to allow different type of elements
- 1 (or more) local (recursive) `class Node`
- `Node` contains (has references to) one or more other nodes
    - ➢ `null` if there is no other `Node`
- `Node` is never exposed to ensure integrity of constraints: **encapsulation**

there is a separate course on algorithms & data-structures: NWI-IBC027

recap

recursive structures are very useful

- recursion: `Node` that contains one (or more) `Node` (or `null`)
- flexible
- efficient manipulations

recap

recursive structures are very useful

- recursion: `Node` that contains one (or more) `Node` (or `null`)
- flexible
- efficient manipulations

lists are very standard and provided in standard libraries

- arrayList
  - ➢ efficient `get(i)`: O(1) ☺
  - ➢ `add(e, i), remove(i)` are O(N) ☹

recap

recursive structures are very useful

- recursion: `Node` that contains one (or more) `Node` (or `null`)
- flexible
- efficient manipulations

lists are very standard and provided in standard libraries

- arrayList
  - ➢ efficient `get(i)`: O(1) ☺
  - ➢ `add(e, i)`, `remove(i)` are O(N) ☹
- LinkedList
  - ➢ `get(i)` is O(i) ☹
  - ➢ `add` and `remove` O(1) (when we know the place)☺
  - ➢ `ListIterator` for efficient access ☺

recap

recursive structures are very useful

- recursion: `Node` that contains one (or more) `Node` (or `null`)
- flexible
- efficient manipulations

lists are very standard and provided in standard libraries

- arrayList
  - efficient `get(i)`: O(1) ☺
  - `add(e, i), remove(i)` are O(N) ☹
- LinkedList
  - `get(i)` is O(i) ☹
  - `add` and `remove` O(1) (when we know the place)☺
  - `ListIterator` for efficient access ☺

more than one successor: Tree

- typically you construct your own tailor-made tree

Lecture 7: Testing with JUnit