

Lambda expressions recursive data structures

Lecture 7 (12 April 2022)

Inner classes & lambda expressions

ad hoc implementations

Java is based on classes

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

sometimes we only need operations/functions and no data

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

Java provides several solutions

1. a locally defined class (in contrast to global public 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;  
  
    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; }  
}
```

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

compare persons by their name

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

sorting a list of Persons

```
public class 00lecture7 {
```

```
    public static void main(String[] args) {  
        run(new Person("Alice",7), new Person("Dave",9),  
            new Person("Bob",2), new Person("Carol",6));  
    }
```

Java syntax for an arbitrary numbers
of arguments (of the same type)

```
    private static void run(Person ... persons){  
        List<Person> group = Arrays.asList(persons);  
        Collections.sort(group);  
        System.out.println(group);  
    }
```

these arguments are passed as an array

RUN

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

ad-hoc sorting with nested class

```
public class OOlecture7 {  
    public static void main(String[] args) {  
        run(new Person("Alice",7),new Person("Dave",9),new Person("Bob",2),new Person("Carol",6));  
    }  
    private static void run(Person ... persons){  
        List<Person> group = Arrays.asList(persons);  
        Collections.sort(group, new CompareById());  
        System.out.println(group);  
    }  
  
    private static class CompareById implements Comparator<Person> {  
        @Override  
        public int compare(Person p1, Person p2) {  
            return p1.getId() - p2.getId();  
        }  
    }  
}
```

The diagram illustrates the implementation of a custom comparator for sorting. A box containing the `Comparator` interface definition is connected by an arrow to the `CompareById` nested class, indicating that the nested class implements the interface. Another arrow points from the `new CompareById()` expression in the `run` method to the same nested class definition, showing its instantiation. A red circle highlights the `Comparator<Person>` part of the nested class declaration.

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

to sort persons on id we need a Comparator object

the nested class

RUN

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

reversed sorting with nested class

```
public class OOlecture7 {  
    public static void main(String[] args) {  
        run(new Person("Alice",7),new Person("Dave",9),new Person("Bob",2),new Person("Carol",6));  
    }  
    private static void run(Person ... persons){  
        List<Person> group = Arrays.asList(persons);  
        Collections.sort(group, new CompareById().reversed());  
        System.out.println(group);  
    }  
}
```

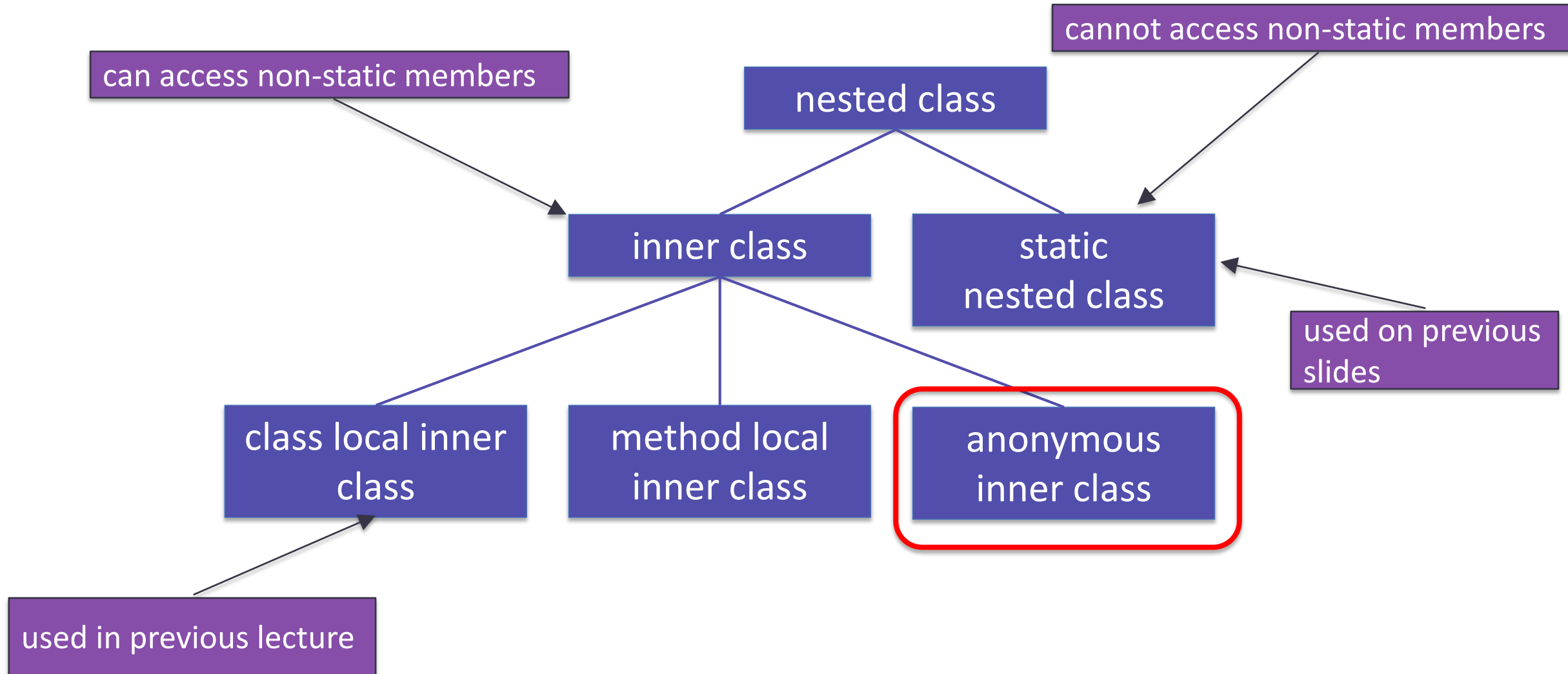
group is now sorted in descending order

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

RUN

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

kinds of nested classes in Java



ad-hoc sorting with anonymous inner class

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

```
public class OOlecture7 {  
    public static void main(String[] args) {  
        run(new Person("Alice",7),new Person("Dave",9),new Person("Bob",2),new Person("Carol",6));  
    }  
    private static void run(Person ... persons){  
        List<Person> group = Arrays.asList(persons);  
        Collections.sort(group, new Comparator<Person>() {  
            @Override  
            public int compare(Person p1, Person p2) {  
                return p1.getId() - p2.getId();  
            }  
        });  
        System.out.println(group);  
    }  
}
```

an interface or an (abstract) class

all methods of class or interface. Can have fields

anonymous class definition

like a constructor followed by a class body

syntax of this ***expression***:

- new operator
- name of interface or class to implement/extend
- arguments to the constructor,
an interface has no constructor: use ()
- class declaration body: method + field 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 fields of enclosing class or (final) local variables of the enclosing method

Comparable vs. Comparator

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

- Comparable is used to define the *natural* or *default* ordering of objects of a class.
 - such a class implements the Comparable interface.
- Comparator is used to define an *ad hoc* ordering on objects.

```
private static void run(){  
    Person a1 = new Person("Alice",7), da = new Person("Dave",9);  
    System.out.println(a1.compareTo(da));  
    Comparator<Person> compPerson = new Comparator<Person>() {  
        @Override  
        public int compare(Person o1, Person o2) {  
            return o1.getName().compareTo(o2.getName());  
        }  
    };  
    System.out.println(compPerson.compare(a1,da));  
}
```

natural ordering

RUN

-3
-3

ad hoc ordering

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

```
public static void main(String[] args) {  
    run(new Person("Alice",7), new Person("Dave",9),  
        new Person("Bob",2), new Person("Carol",6));  
}
```

```
private static void run( Person ... persons ) {  
    List<Person> group = Arrays.asList(persons);  
    Collections.sort(group, (p1,p2) -> p1.getId() - p2.getId() );  
    System.out.println(group);  
}
```

lambda expression



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



syntax of lambda expressions

- works only if we need exactly 1 method: functional interface
 - context should identify which class/interface is needed

1. list of parameters

- you can omit the types of the parameters
- if there is only 1 parameter without a type you can omit parentheses

2. the arrow token ->

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

3. body

- single expression
 - does not need statement braces { and }
 - does not need the return keyword
- statement block
 - needs statement braces { and }
 - multiple statements separated by ;

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

more lambda expressions 1/3

- `filter` returns a list consisting of the elements of a given list that match the given predicate.

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

```
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;
}
```



call to Predicate method test

more lambda expressions 2/3

```
public static void main(String[] args) {  
    run(new Person("Alice",7), new Person("Dave",9),  
        new Person("Bob",2), new Person("Carol",6));  
}
```

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

```
private static void run( Person ... persons ) {  
    List<Person> group = Arrays.asList(persons);  
    List<Person> group3 = filter(group, (Person p) -> p.getId() % 3 == 0);  
    System.out.println(group3);  
}
```

anonymous implementation of Predicate method test

RUN

[Dave (9), Carol (6)]

more lambda expressions 2/3

```
public static void main(String[] args) {  
    run(new Person("Alice",7), new Person("Dave",9),  
        new Person("Bob",2), new Person("Carol",6));  
}
```

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

```
private static void run( Person ... persons ) {  
    List<Person> group  = Arrays.asList(persons);  
    List<Person> group3 = filter(group, p -> { int pId = p.getId();  
                                                return pId > 4;  
                                            }  
    );  
    System.out.println(group3);  
}
```

RUN

[Alice (7), Dave (9), Carol (6)]

more lambda expressions 3/3

lambda expressions are *expressions*

- e.g. their value can be assigned to a variable

```
public static void main(String[] args) {  
    run(new Person("Alice",7), new Person("Dave",9),  
        new Person("Bob",2), new Person("Carol",6));  
}
```

a variable of type Predicate<Person>

```
private static void run( Person ... persons ) {  
    Predicate<Person> idGT4 = (Person p) -> p.getId() > 4;  
    boolean idIsGT4 = idGT4.test(persons[2]);  
    System.out.println(idIsGT4);  
}
```

an expression of type Predicate<Person>

checks if the second person has an Id greater than 4

RUN

false

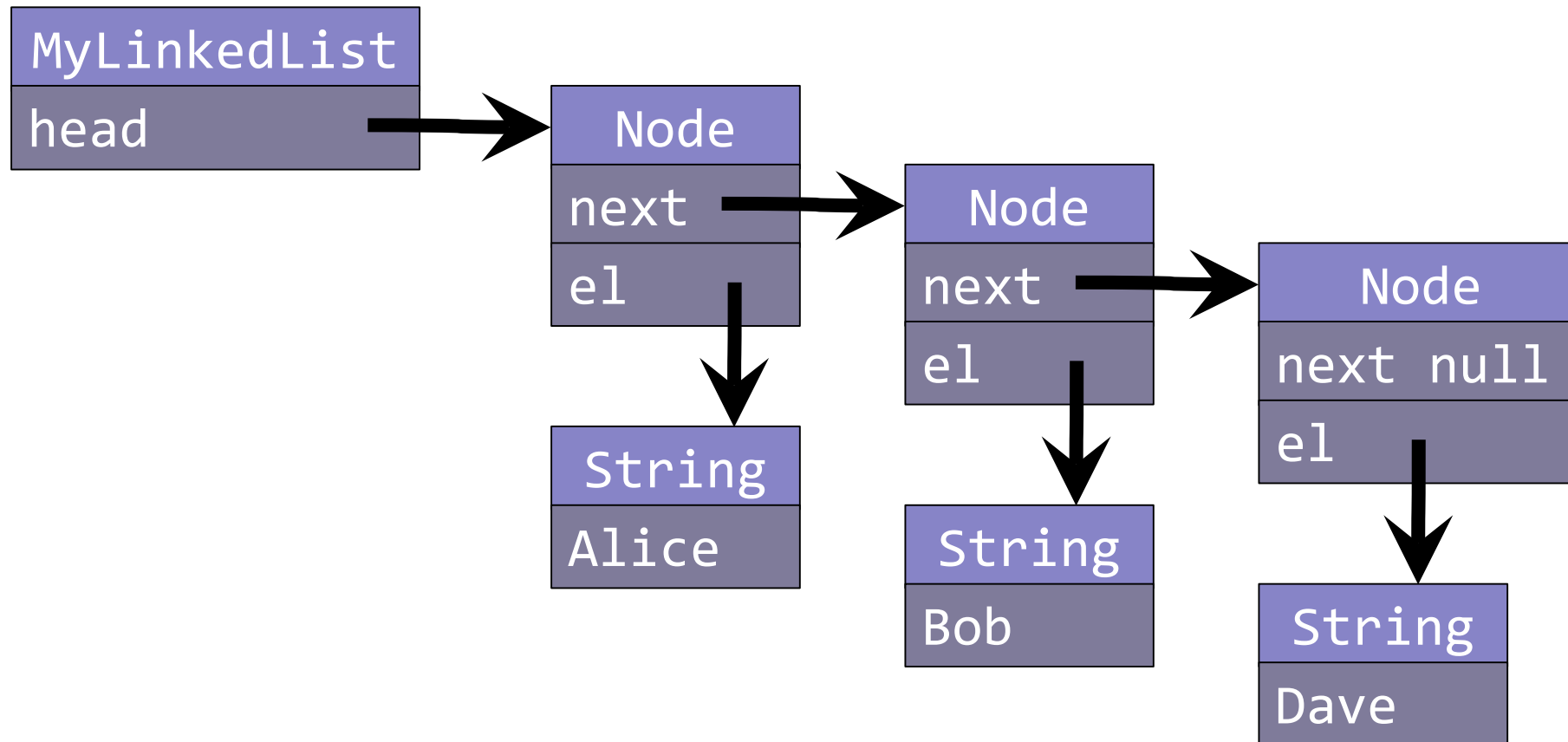
Recursive data types: trees



Generic Recursive Type with multiple children per node: **Tree**

Linked List

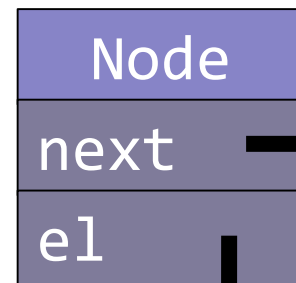
In the previous lecture we saw how we can represent the linked lists



MyLinkedList<E>: Node class

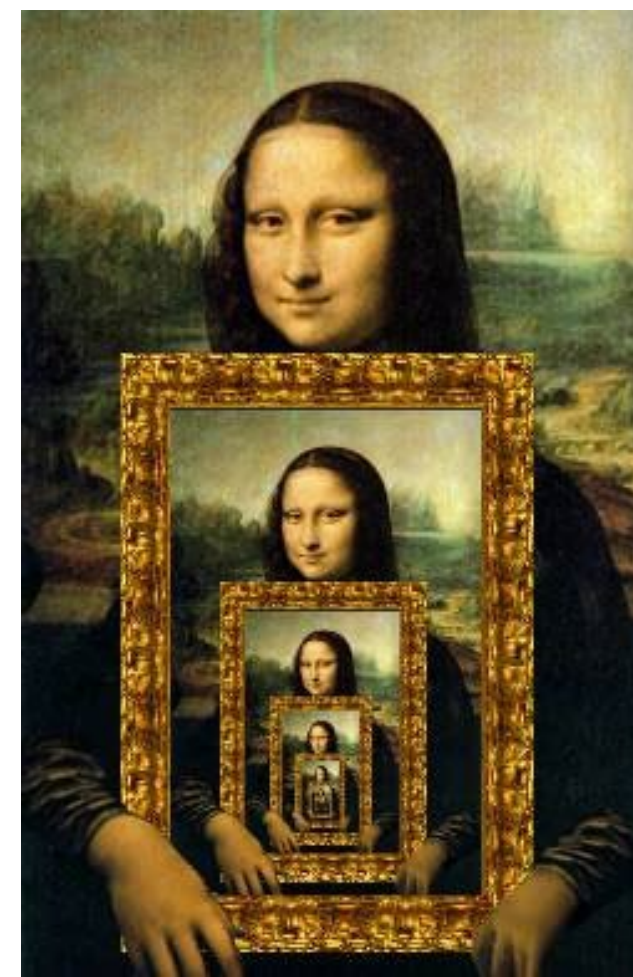
```
public class MyLinkedList<E> extends AbstractList<E> {  
    ...  
    private static class Node<A> {  
        private A e1;  
        private Node<A> next;  
        public Node(A e, Node<A> n) {  
            e1 = e;  
            next = n;  
        }  
  
        public Node(A e) {  
            this(e, null);  
        }  
    }  
    ...  
}
```

recursive datatype/class



object of type Node<A>

object of type A



trees

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

- or even 3 or n children

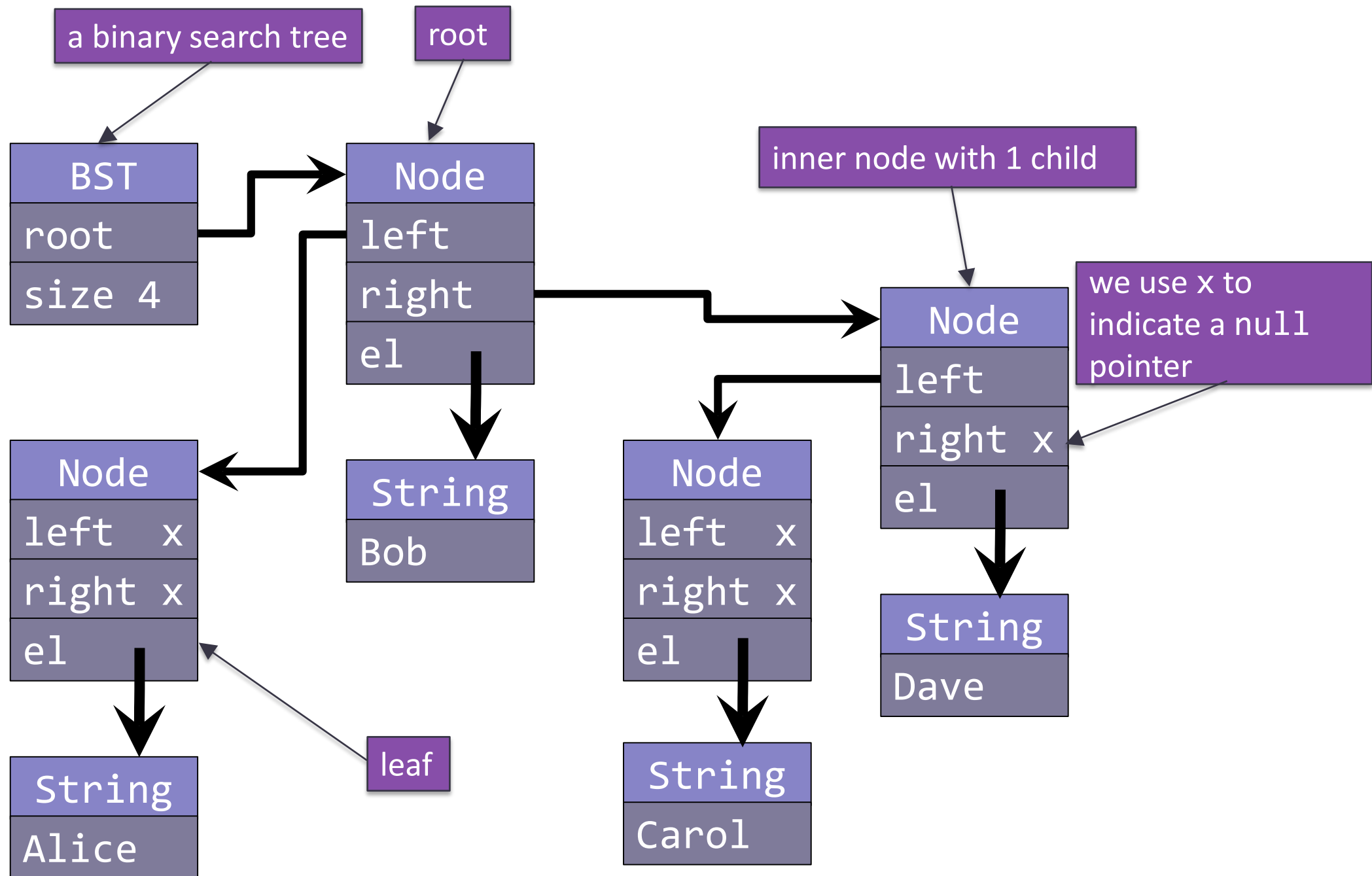
these data structures are called **trees**

- sometimes we use different kinds of nodes
e.g. Leaf (no children) and Fork (with children)
- binary trees (2 children) is most common

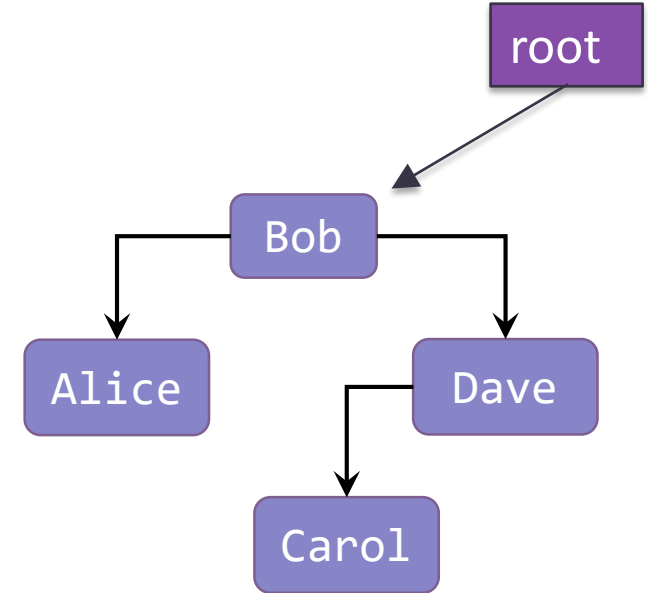
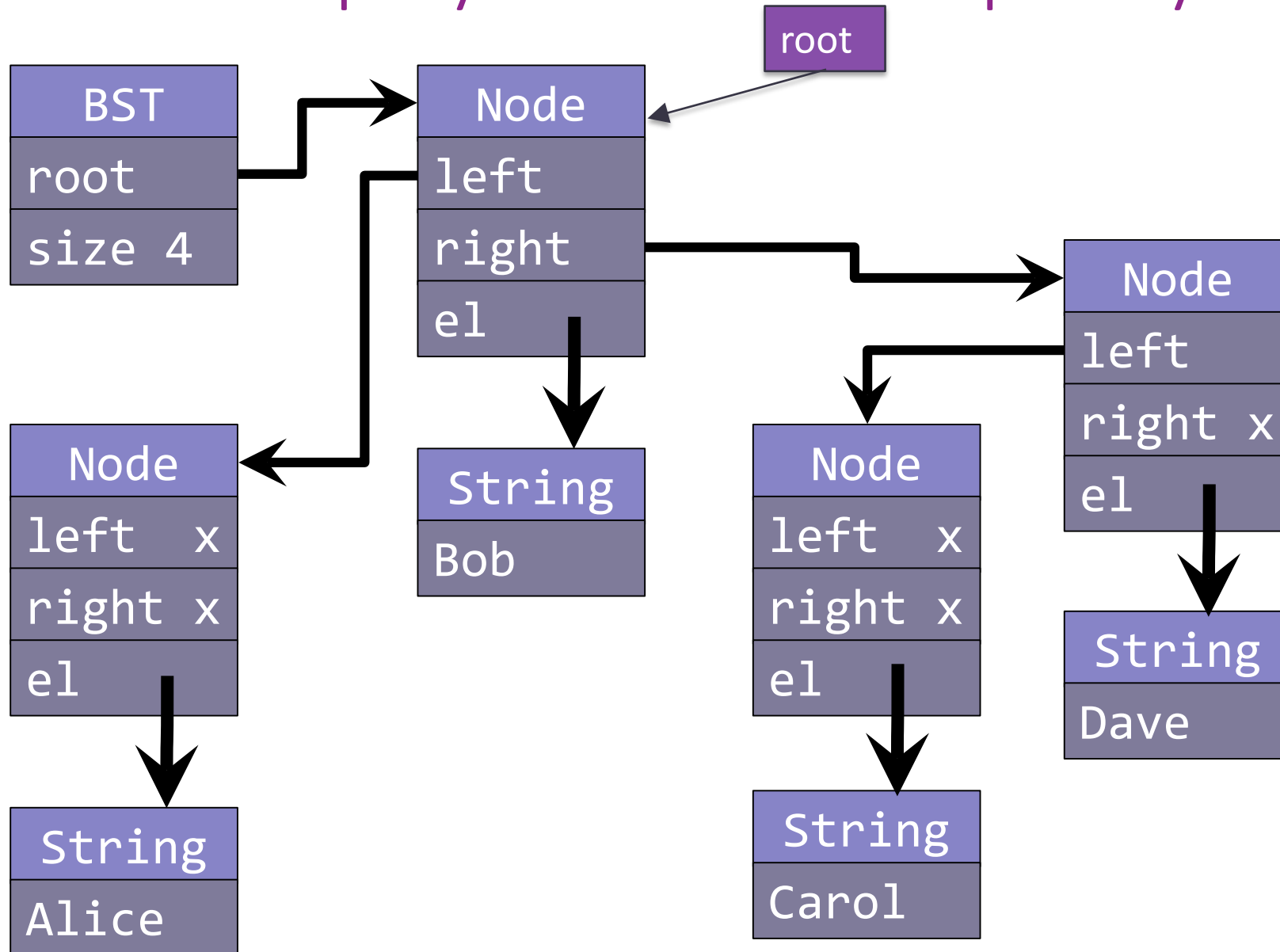
a frequently used variant is **binary search tree**

- each node has (at most) two children
- all elements in the left subtree are smaller than element in node
- all elements in right subtree are bigger

tree



tree displayed more compactly



Representing Binary Trees: class `TreeNode<E>`

A binary tree can be represented using a set of *linked nodes*. Each node contains a value and two links named *Left* and *right* that reference the left child and right child.

```
private static class TreeNode<E> {  
    private E element;  
    private TreeNode<E> left, right;
```

very similar to Linked List, only with two children



```
    public TreeNode(E e) {  
        this(e, null, null);  
    }
```

```
    public TreeNode(E element, TreeNode<E> left, TreeNode<E> right) {  
        this.element = element;  
        this.left     = left;  
        this.right    = right;  
    }  
}
```

Representing Binary Trees: class BST

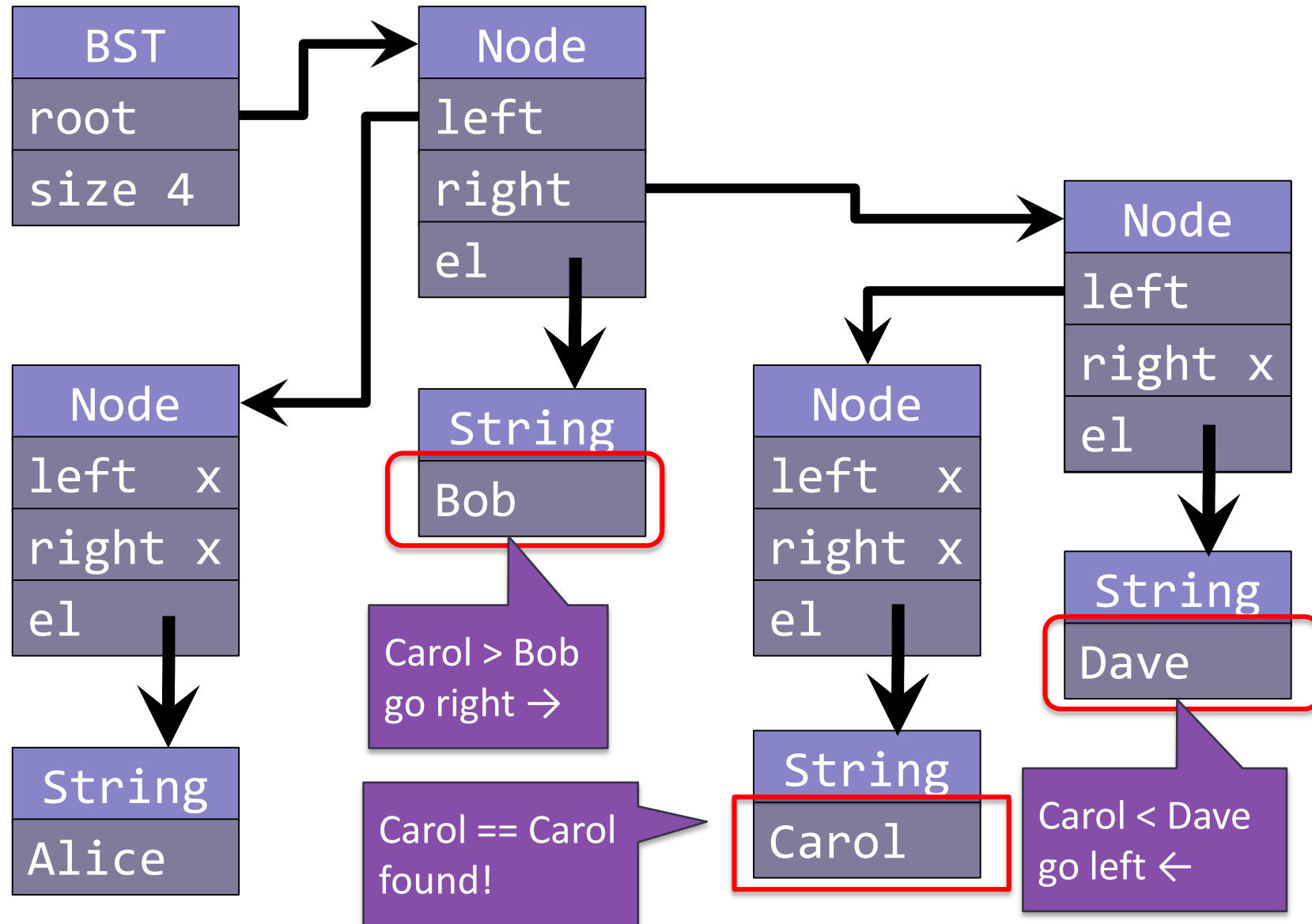
Analogous to the list implementations, we don't want to give users direct access to the tree structure itself.

- For that reason, we are introducing a wrapper class BST that hides the internal structure.

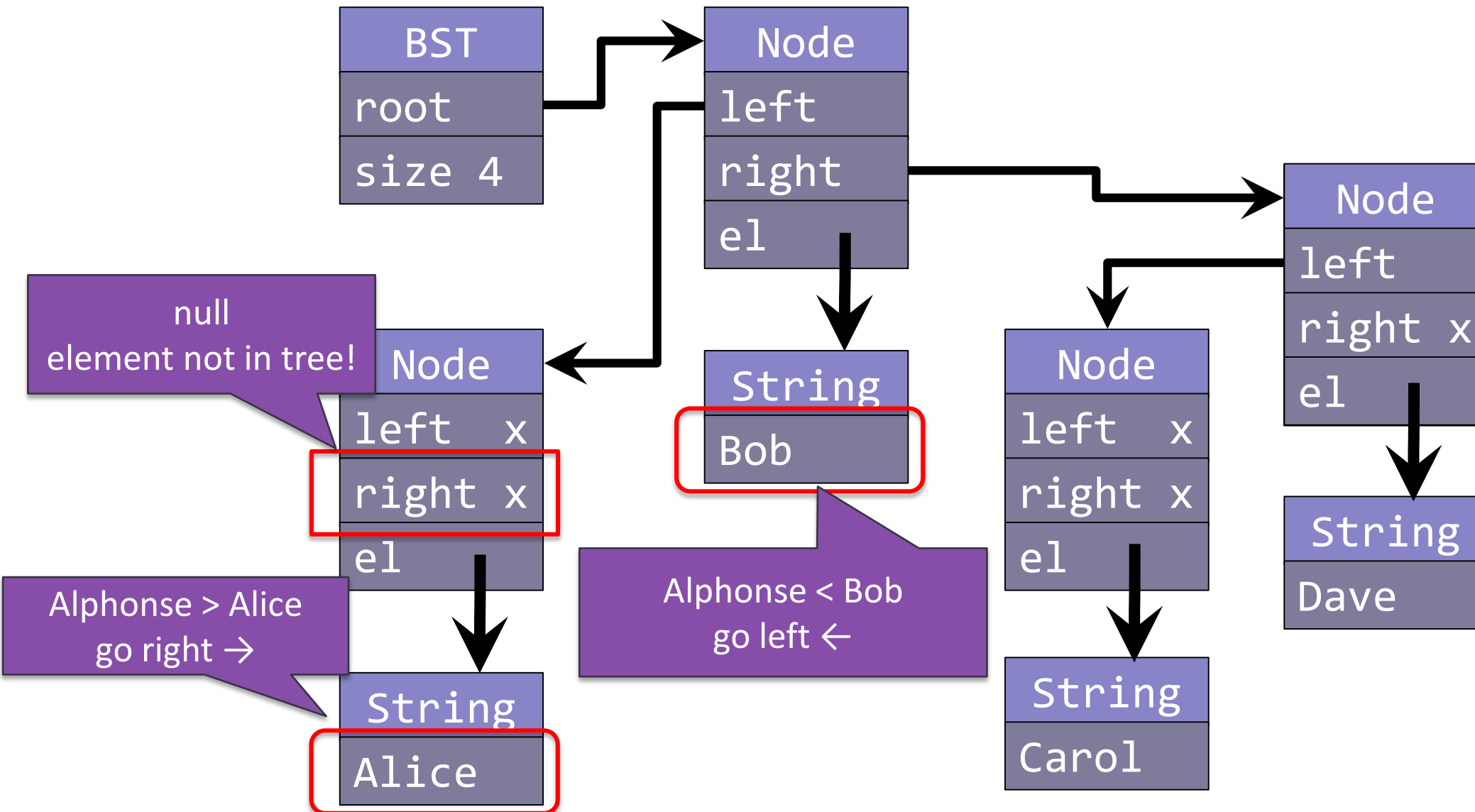
ensures comparability of elements


```
public class BST <E extends Comparable<E>> {  
    private Node<E> root;  
    private int size;  
    ...  
}
```

tree: search for Carol



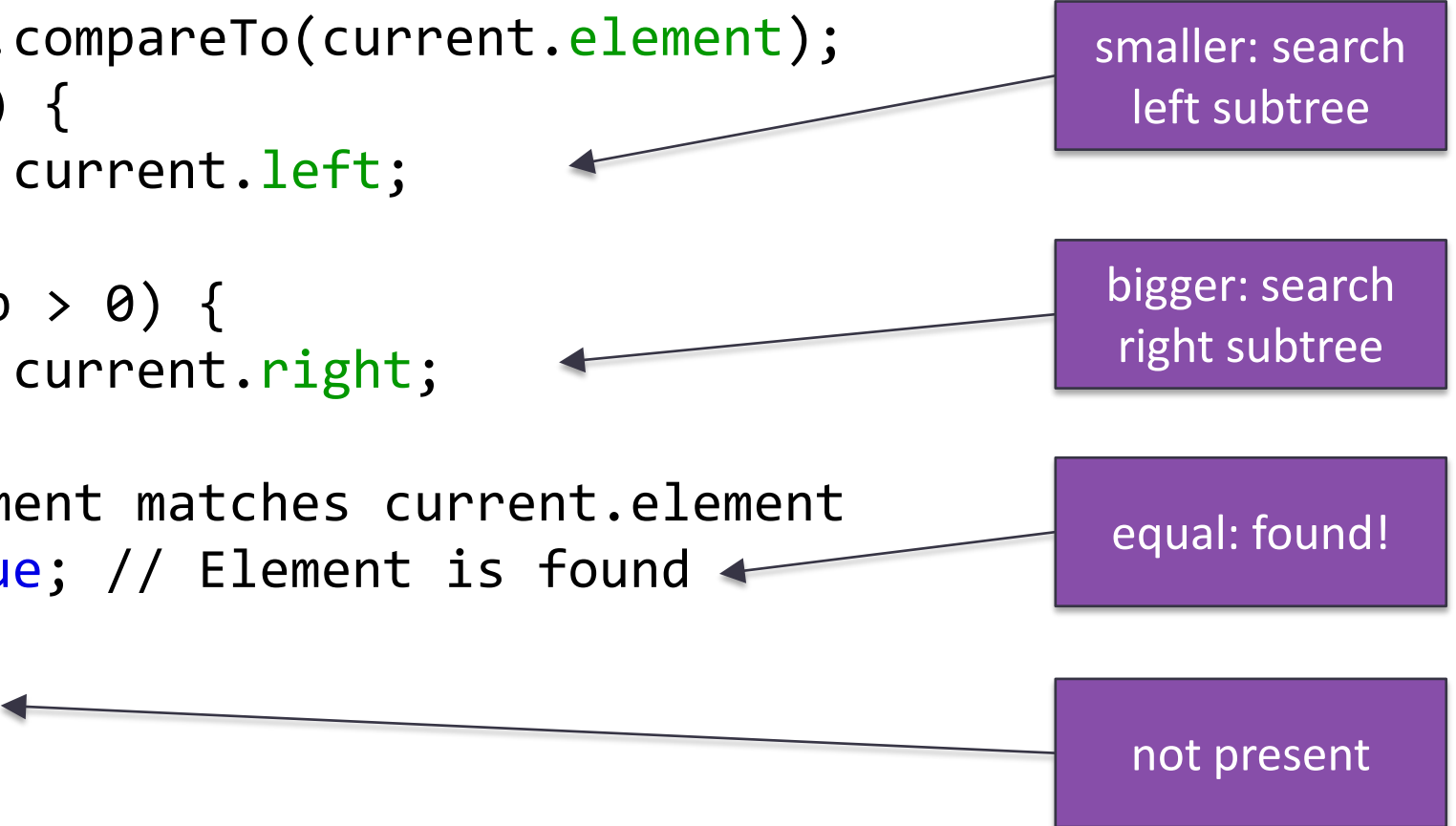
tree: search for Alphonse



search in search tree (iteratively)

```
public boolean search(E e) {  
    TreeNode<E> current = root; // Start from the root  
  
    while (current != null) {  
        int cmp = e.compareTo(current.element);  
        if (cmp < 0) {  
            current = current.left;  
        }  
        else if (cmp > 0) {  
            current = current.right;  
        }  
        else // element matches current.element  
            return true; // Element is found  
    }  
    return false;  
}
```

smaller: search
left subtree



bigger: search
right subtree

equal: found!

not present

search in search tree (recursively)

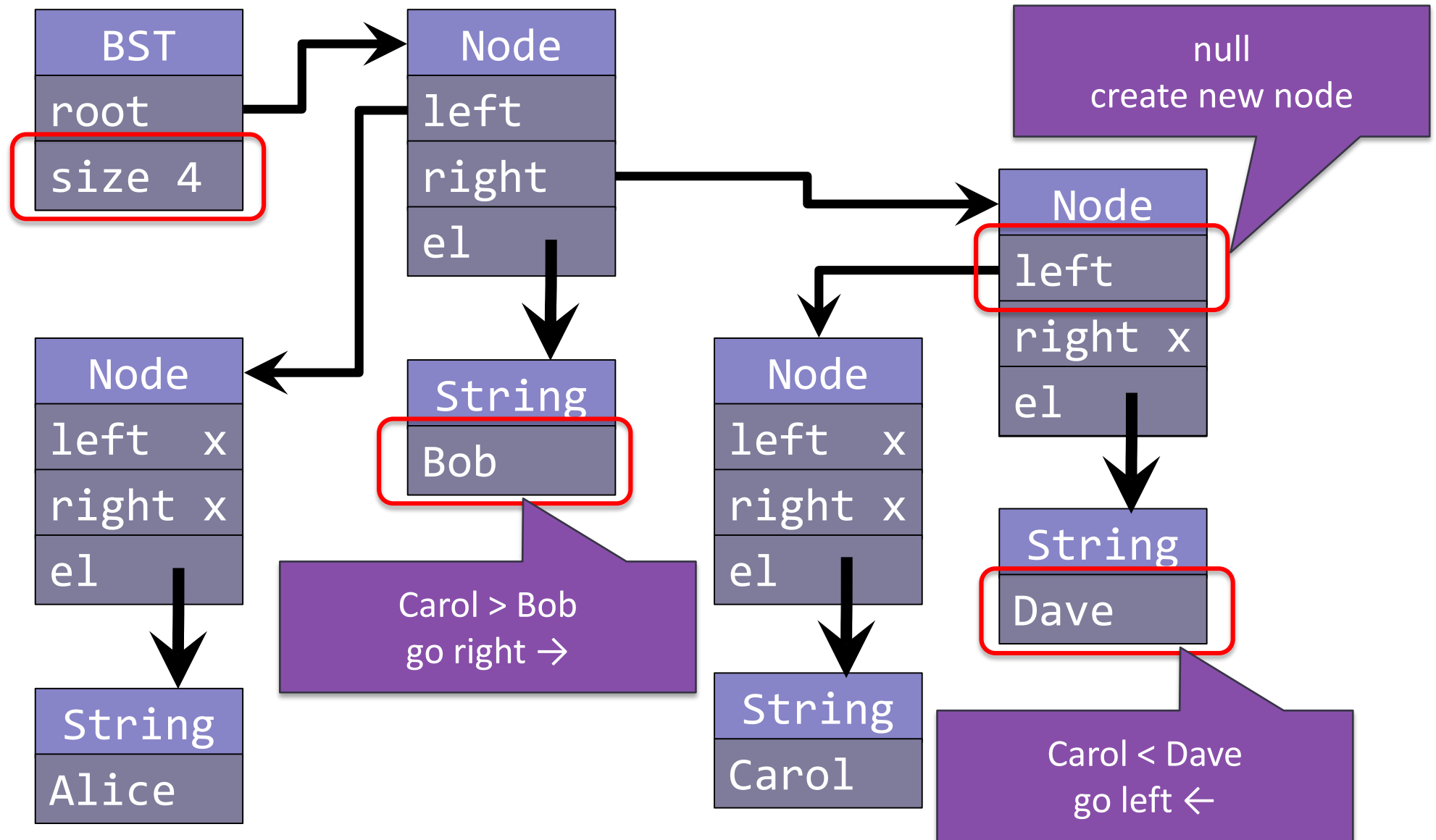
```
public boolean search(E e) {  
    return search(root, e);  
}
```

```
private static <E extends Comparable<E>> boolean search (TreeNode<E> n, E e) {  
    if (n == null) {  
        return false;  
    } else {  
        int comp = e.compareTo(n.element);  
        if (comp < 0) {  
            return search(n.left, e);  
        } else if (comp > 0) {  
            return search(n.right, e);  
        } else { // comp == 0  
            return true;  
        }  
    }  
}
```

static recursive helper method

search is called recursively

tree: adding Carol



add to a search tree (iteratively)

```
public void insert(E e) {  
    if ( root == null ) {  
        root = new TreeNode<>(e);  
        size = 1;  
    } else {  
        TreeNode<E> previous = null, current = root;  
        while (current != null) {  
            int cmp = e.compareTo(current.element);  
            if (cmp < 0) {  
                previous = current;  
                current = current.left;  
            }  
            else if (cmp > 0) {  
                previous = current;  
                current = current.right;  
            }  
            else  
                return;  
        }  
        if ( e.compareTo(previous.element) < 0 ) {  
            previous.left = new TreeNode<>(e);  
        } else {  
            previous.right = new TreeNode<>(e);  
        }  
        size++;  
    }  
}
```

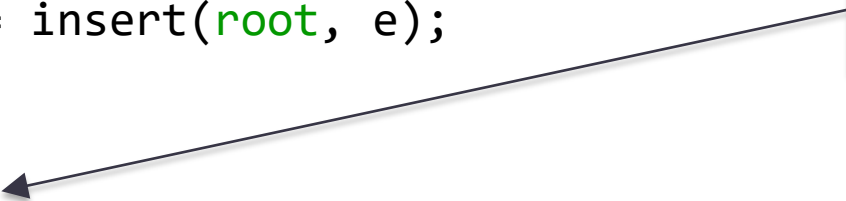


no duplicates

add to a search tree (recursively)

```
public void insert(E e) {  
    root = insert(root, e);  
}
```

not static because the size is
possibly adjusted



```
private TreeNode<E> insert(TreeNode<E> n, E e) {  
    if (n == null) {  
        size++;  
        return new TreeNode<>(e);  
    } else {  
        int comp = e.compareTo(n.element);  
        if (comp < 0) {  
            n.left = insert(n.left, e);  
        } else if (comp > 0) {  
            n.right = insert(n.right, e);  
        }  
        return n;  
    }  
}
```

converting a BST to a list (recursively)

```
public List<E> toList() {  
    List<E> list = new LinkedList<>();  
    toList(root, list);  
    return list;  
}
```

can be static again



```
private static <E> void toList(TreeNode<E> n, List<E> list) {  
    if (n != null) {  
        toList(n.left, list);  
        list.add(n.element);  
        toList(n.right, list);  
    }  
}
```

in-order traversal



testing BST

```
private static void run(){  
    int[] items = { 1, 5, 2, 8, 3, 12, 2 };  
    BST<Integer> bst = new BST();  
    for (int it: items) {  
        bst.insert(it);  
    }  
    List<Integer> elems = bst.toList();  
    System.out.println(elems);  
    System.out.println(bst.getSize());  
}
```

RUN

[1, 2, 3, 5, 8, 12]

6

recursive data-structure implementation pattern

there are many different recursive data-structures

- they might differ in complexity of operations

there is a main (wrapper) class providing 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 is often used to indicate that there is no other Node
- class Node is never exposed to ensure integrity of constraints: encapsulation

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

NEXT WEEK

Lecture 8: GUI programming (JavaFX)