Technische Hochschule **Rosenheim**
Technical University of Applied Sciences

Object-oriented programming
Chapter 2b – Trees

Prof. Dr Kai Höfig

Sets of elements

i.e. without duplicates

- We examined the list as a sequential data structure. This stores data in the order in which it is added. In doing so, duplicates can also be included.

- A *set* should contain each element only once, even if we try to include it again.

- The interface required could look like this:

```
public interface CharSet {
  void add(char c);           // add element
  boolean contains(char c);  // check if already included
  char remove(char c);       // remove element
  int size();                // not "length", as there is no sequence
}
```

- Since a set has no order, i.e. all the elements are "simply included" in it without index numbers, there is a `size` method instead of the `length` method. Therefore we also do not pass an index to the `remove` method, but rather a value!

Inner classes

- If we implement a set in the same way as the list, we once again need a helper class, which stores the actual data and provides the framework for the data structure. Because the class is specific to this structure and implementation, it can be created as an *inner* class:

```
class CharSetImpl1 implements CharSet {

  private Element head;

  class Element {
    char value;
        Element next;
        Element(char c, Element n) {
      value = c;
      next = n;
        }
    }
..
```

- In Java, inner classes are...
  - useful if they are used exclusively within a class, i.e. used locally.
  - declared as normal or as `static`; normal inner classes can access the variables of the outer instance, whereas static inner classes can only access static variables and methods of the outer class.
  - allocated visibility - just like variables and methods: `package` (no keyword), `private`, `public`.

Avoiding duplicates

- We can now avoid duplicates by checking whether an item is already included before adding it. To do so, we start at the front and work our way through to the back, checking each element for (value) equality.

```
public boolean contains(char c) {
  if (head == null)
    return false;

    Element it = head;
  while (it != null) {
    if (it.value == c)
      return true;
        it = it.next;
    }

  return false;
}
```

- The insertion (`add`) can then be implemented in the same way as the list, just like the methods `size`, `remove` and `toString`
- The implementation of the set as a list is highly inefficient: for each add or contains, the entire list must first be run through to check whether the element is already included or not. That's why we will now use *trees*

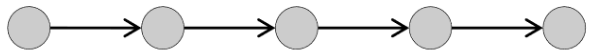
Goals:

- Getting to know the concept of the *tree* in computer science
- Being able to represent trees as chained data structures
- Understanding and being able to write recursive functions on trees

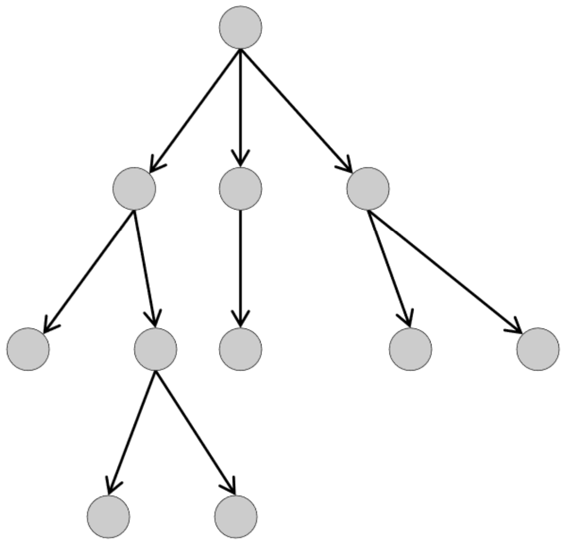
Trees

- Trees are one of the most widely used data structures in computer science
- Trees generalise lists:
  - in a list, each node has at most one successor.
  - in a tree, a node can have multiple successors.

List

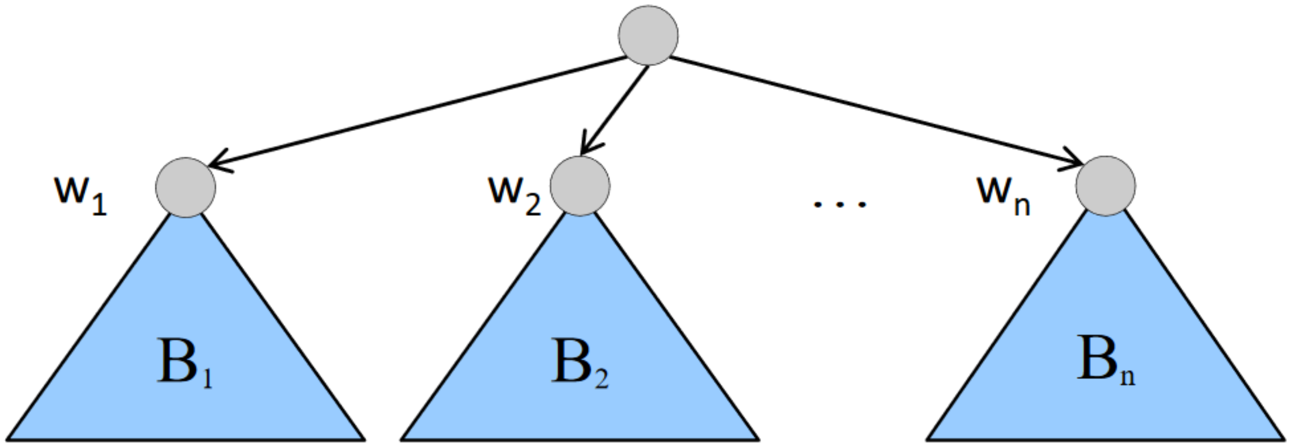


Tree



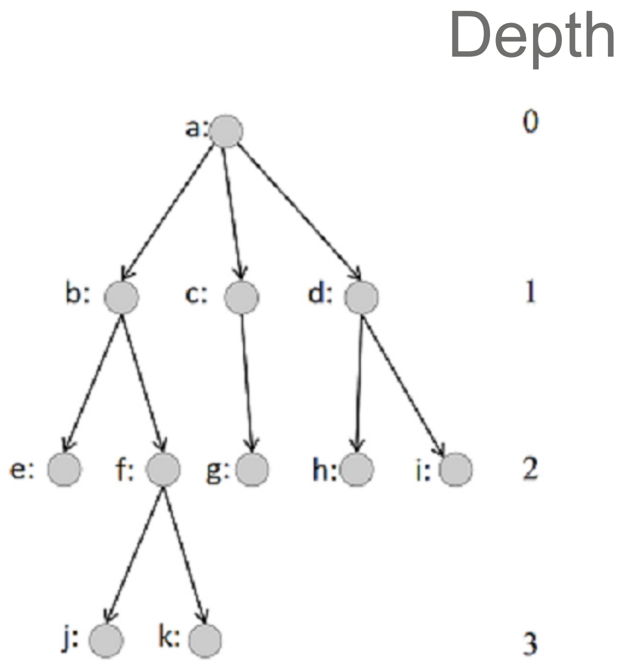
Definitions

- A **tree** consists of **nodes** connected by **edges**.
- A **root node** (*root*) is a node that no edge points to.
- Different data may be stored in a node, depending on the application.
- The set of all trees is constructed using the following rules:
  - There is an empty tree.
  - A single node without any edges is a tree (root).
  - If n > 0 and if *B1, B2, ...Bn* are trees with root nodes *w1, w2, ... wn*, we can assemble them into a larger tree by adding a new node and connecting it to w1, w2, ..., wn.
  - The new node is then the root node of the tree that has been constructed in this way.



Terminology

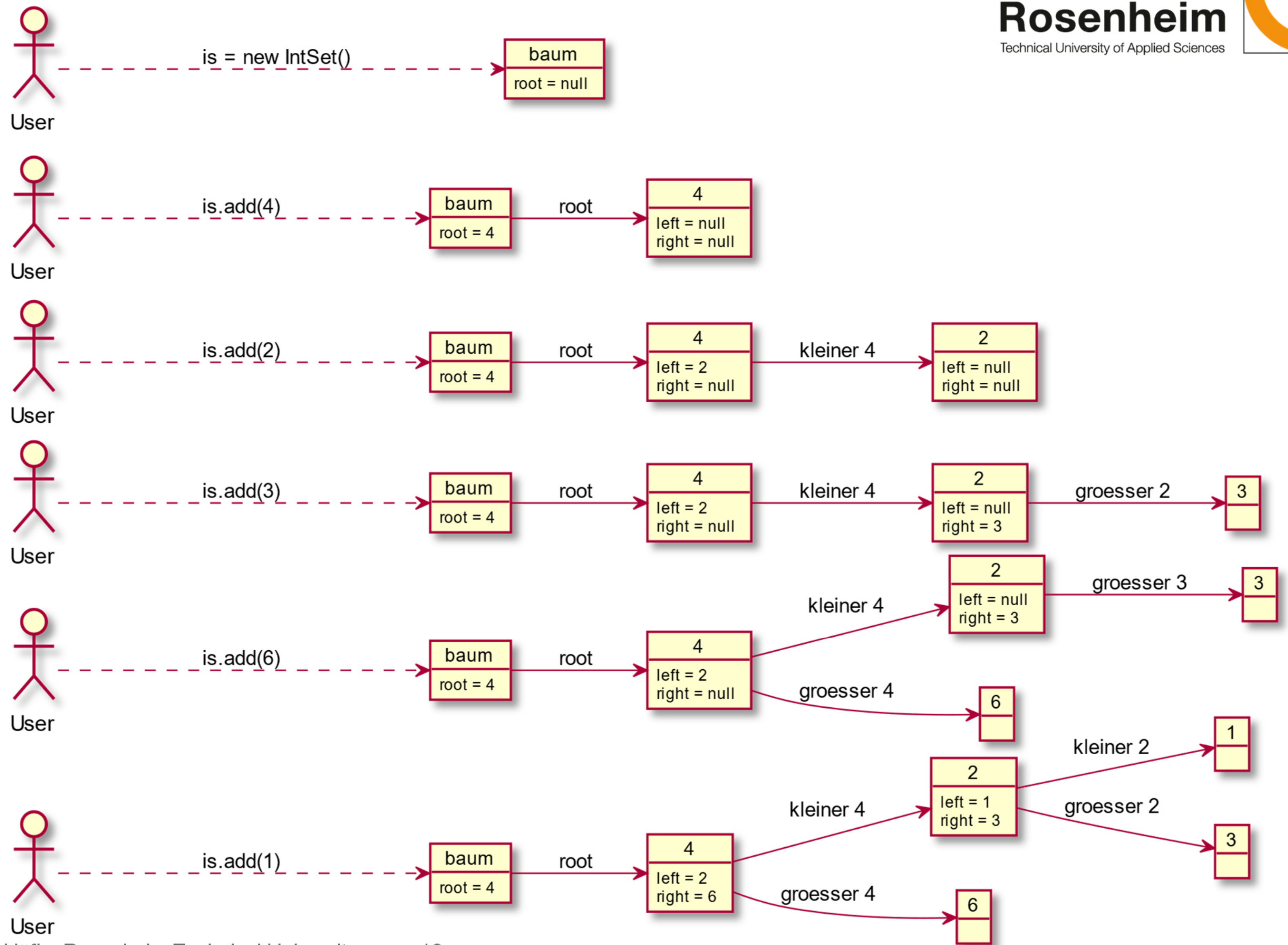
- a is the **root node** of the tree.
- h and i are the **successor nodes** or also **child nodes** of node d.
- d is the **predecessor** or **parent node** of h.
- Nodes without successors (here: e, j, k, g, h, i) are called **leaf nodes**
- The **depth of a node** in the tree is the number of steps needed to reach the node from the root.
- The **depth of the tree** is the maximum depth of all nodes of the tree. (here: 3)
- A tree is a **binary tree** if each node in it has a **maximum** of two successors.
- Each node in a tree is the root of a **subtree** of the given tree.
- The subtree with root k consists of the node k, its children, its children's children, etc.
- Examples: The subtree with root b and the one with root d. In comparison, the subtree with root a is the whole tree itself.
- The subtrees with roots b and d are both binary trees.
- The entire tree is not a binary tree.



Binary tree

- A fundamental data structure in computer science is the *binary tree*. It differs from the list in that each element has *two* successors rather than one -- hence the name: *binary*.
- Thus, an element does not point *sequentially* to the *next* element, but instead has references to a *left* and *right subtree*, which then only contain *smaller* or *larger* values respectively than the element itself.
- If we add an element, we descend from the root node (root) to the left or right, so far until we either find the value or we arrive at a point where we can insert the new value.

Example



User
is = new IntSet()
baum
root = null
User
is.add(4)
baum
root = 4
root
4
left = null
right = null
User
is.add(2)
baum
root = 4
root
4
left = 2
right = null
kleiner 4
2
left = null
right = null
User
is.add(3)
baum
root = 4
root
4
left = 2
right = null
kleiner 4
2
left = null
right = 3
groesser 2
3
User
is.add(6)
baum
root = 4
root
4
left = 2
right = null
kleiner 4
2
left = null
right = 3
groesser 3
3
groesser 4
6
User
is.add(1)
baum
root = 4
root
4
left = 2
right = 6
kleiner 4
2
left = 1
right = 3
kleiner 2
1
groesser 2
3
groesser 4
6

Implementing a binary tree

- If we now want to search for an element (contains), we start at the root node (root), check if the value already exists there, and otherwise descend to the left or right, depending on whether the value we are searching for is smaller or larger than the element that is currently being viewed.

- How is such a tree implemented, and how do we descend to the left or right? Let's start with the structure; an element now doesn't have only one successor next, but two: left and right.

```
public class BinaryTreeCharSetImpl implements CharSet{
  class Element {
    char value;
        Element left, right;
        Element(char c, Element le, Element re) {
      value = c;
      left = le;
      right = re;
        }
    }
    Element root;
..
```

The `contains` method

- With the contains method, we iterate in a similar way to the list, but instead of it = it.next, a distinction must be made as to whether we want to descend to the left or right:

```
public boolean contains(char t) {
  if (root == null)
    return false;

    Element it = root;
  while (it != null) {
    if (t == it.value)
      return true;
    else if (t < it.value) {
            it = it.left;
        } else {
            it = it.right;
        }
    }

  // not found!
  return false;
}
```

The `add` method

- The insertion procedure is now similar, except that we must look left or right here, instead of always just looking one element ahead:

```
public void add(char c) {
    Element e = new Element(c, null, null);

  if (root == null) {
    root = e;
    return;
    }

    Element it = root;  // begin at the root
  while (it != null) {
    // already present -> finished!
    if (it.value == e.value)
      return;
      // descend left?
    else if (e.value < it.value) {
      // we have reached the bottom -> insert!
      if (it.left == null) {
        it.left = e;
        return;
            } else
        it = it.left;
        } else {
      // similarly
      if (it.right == null) {
        it.right = e;
        return;
            } else
        it = it.right;
        }
    }
}
```

The `size` method

with recursion

```
class Element {
  char value;
    Element left, right;
  int size() {
    int s = 1;  // at least a minimum of one element.

        // is there a left subtree? Then add its size.
    if (left != null) s += left.size();

    // similarly.
    if (right != null) s += right.size();

    return s;
    }

..
```

```
@Override
public int size() {
  if (root == null) return 0;
  else return root.size();
}
```

The `toString` method

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Once again, this method is also split into a part for `Element`

```
public String toString() {
    String s = "" + value;  // at least this element

    // is there a left subtree? Then add this string
  if (left != null) s += ", " + left.toString();

  // similarly.
  if (right != null) s += ", " + right.toString();

  return s;
}
```

- ..and a part for our tree set implementation:

```
public String toString() {
  if (root == null) return "[]";
  else return "[" + root.toString() + "]";
}
```

Deleting elements

- With the **list**, deleting an item was easy: first, we always look one element ahead, to find the element we want to delete; so we stop one element in front of the one to be deleted. The actual deletion is achieved by changing the link so that the element before the one to be deleted then instead points to the element after the one to be deleted.

- If we want to delete an element from the set or **tree**, it is a bit more complicated since we have to look to the right and left when searching, similar to `contains`.

- We break this difficult problem up into simpler sub-problems. First, there are three cases to consider:

  - The tree is already empty; a NoSuchElementException should be thrown here.
  - The element to be deleted is the root element.
  - The element to be deleted is an inner node or leaf.

Deleting elements

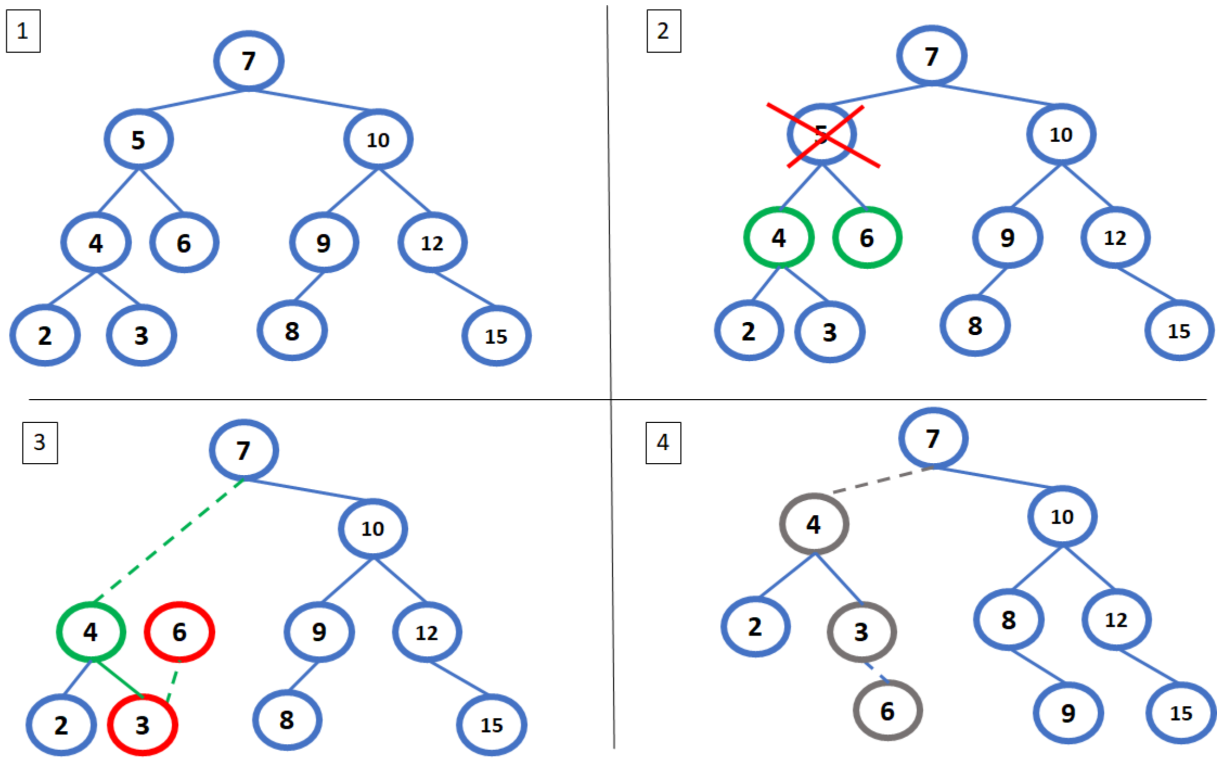
The tree is empty

- The tree is already empty; a NoSuchElementException should be thrown here.

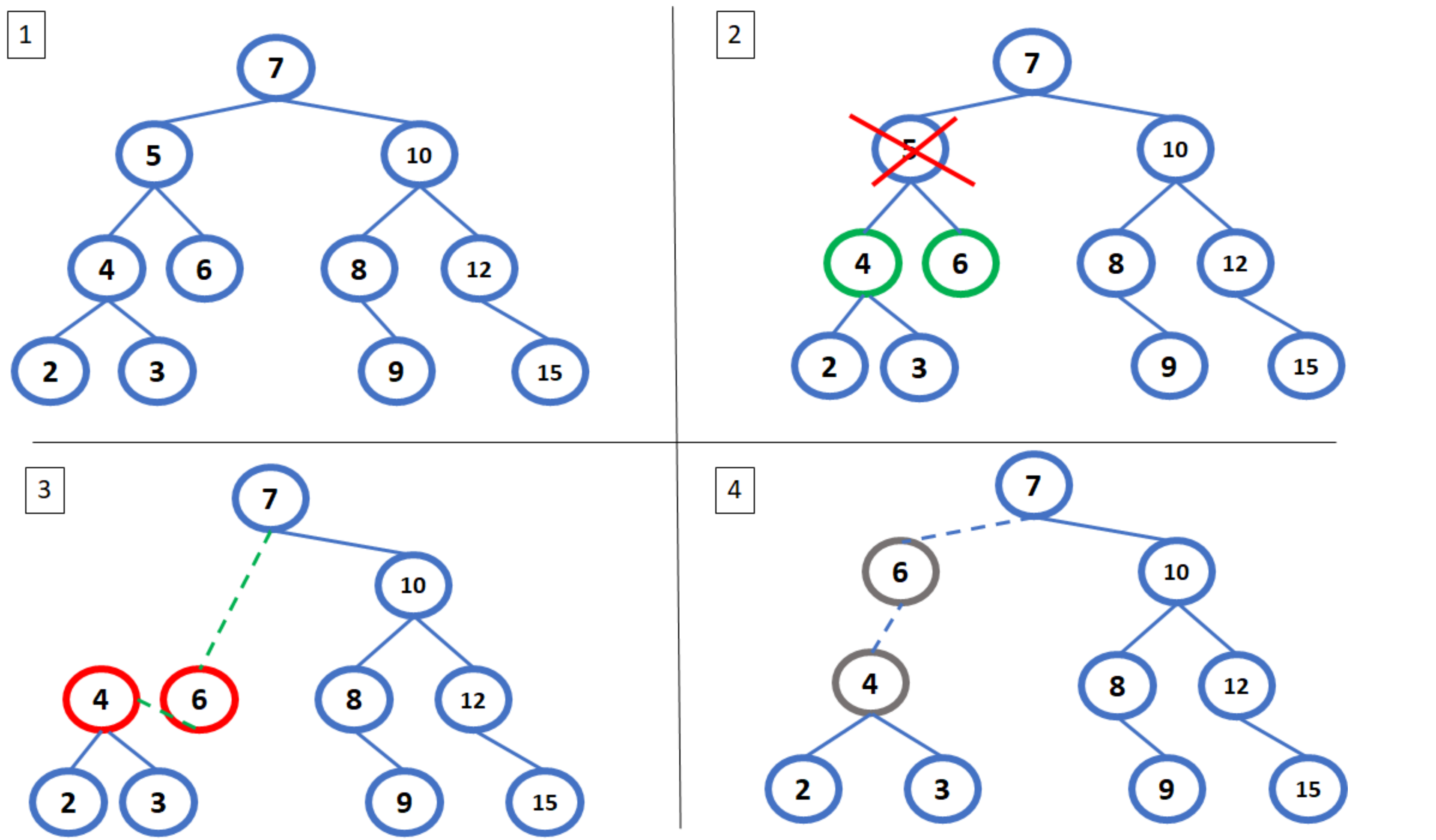
```
public char remove(char c) {
  // check for trivial case
  if (root == null)
    throw new NoSuchElementException();
```

Procedure for deleting

- If we delete an element, we must ensure that any subtrees (left and right) are inserted back into the tree again. To do this, we create an `addElement` helper method that works in a similar way to the existing `add` method, but inserts an element (with any subtrees) instead of just a value.

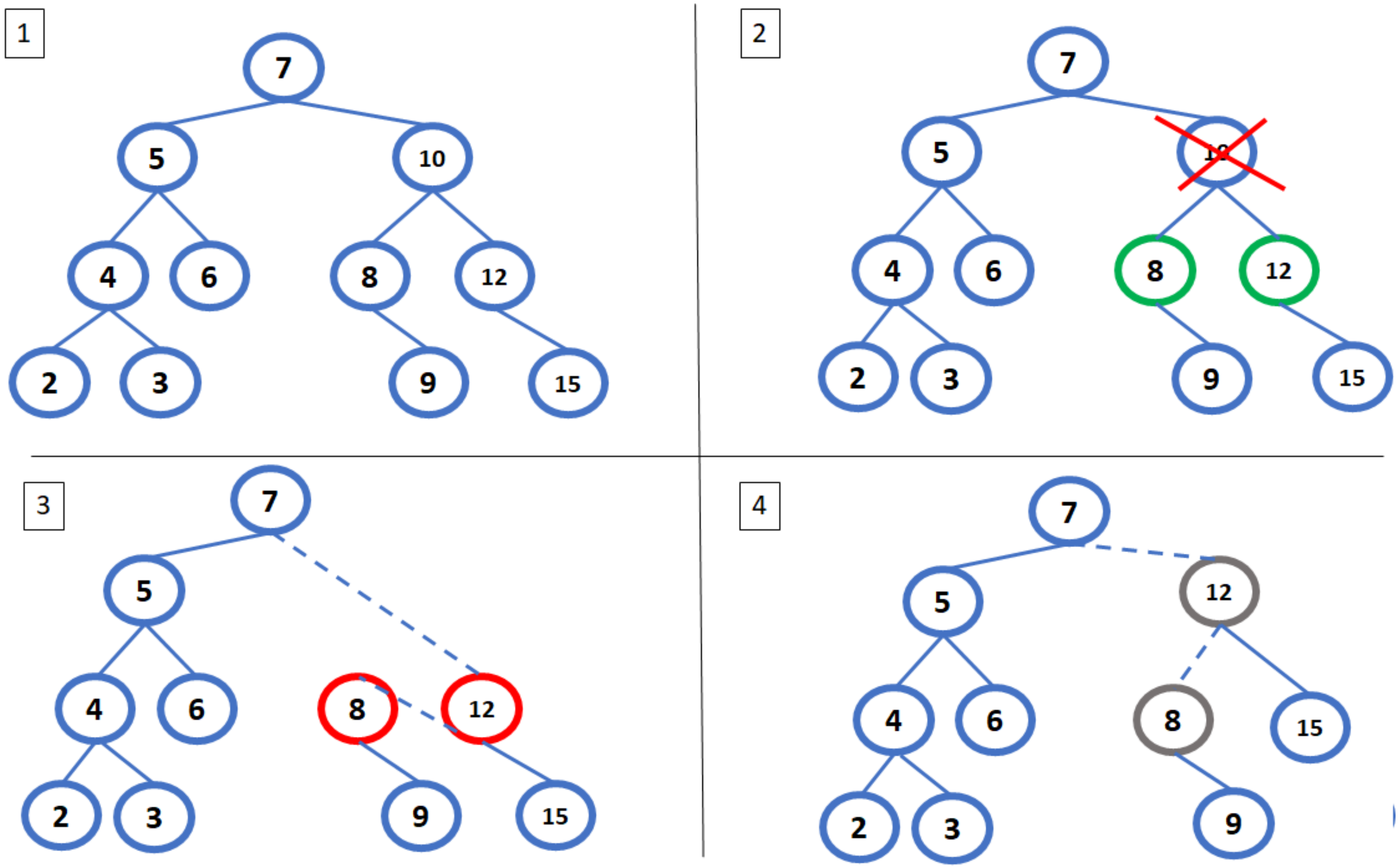


Option 2

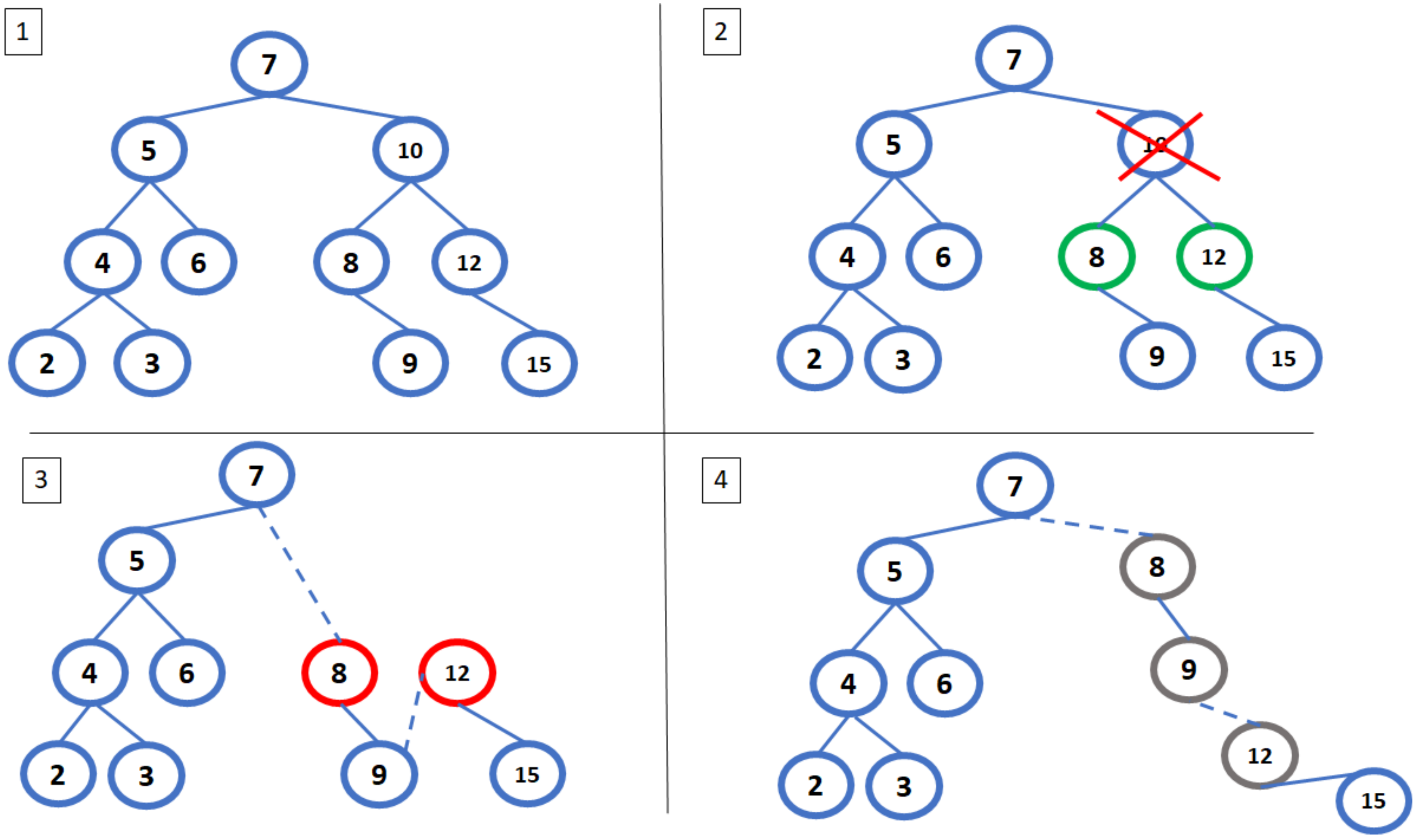


1
7
5
10
4
6
8
12
2
3
9
15
2
7
5
10
4
6
8
12
2
3
9
15
3
7
10
4
6
8
12
2
3
9
15
4
7
6
10
4
8
12
2
3
9
15

Option 3


1
7
5
10
4
6
8
12
2
3
9
15
2
7
5
10
4
6
8
12
2
3
9
15
3
7
5
4
6
8
12
2
3
9
15
4
7
5
12
4
6
8
15
2
3
9

Option 4


1
7
5
10
4
6
8
12
2
3
9
15
2
7
5
10
4
6
8
12
2
3
9
15
3
7
5
4
6
8
12
2
3
9
15
4
7
5
8
4
6
9
2
3
12
15

`addElement` helper method

```java
private void addElement(Element e) {
  if (e == null)
    return;
  if (root == null) {
    root = e;
    return;
    }
    Element it = root;
  while (it != null) {
    if (it.value == e.value)
      return;
    else if (e.value < it.value) {
      if (it.left == null) {
        it.left = e;
        return;
            } else
        it = it.left;
        } else {
      if (it.right == null) {
        it.right = e;
        return;
            } else
        it = it.right;
        }
    }
}
```

Deleting elements

Deleting the root

- The element to be deleted is the root element.

```
private char removeRoot() {
    Element e = root;
  if (e.left == null && root.right == null) {
    // no subtrees, so the tree is now empty
    root = null;
    } else if (e.left == null) {
    // only right subtree, so set this as root
    root = e.right;
    } else if (e.right == null) {
    // ditto, for left
    root = e.left;
    } else {
    // there are both subtrees; left becomes the new root,
        // right is inserted as an element.
    root = e.left;
        addElement(e.right);
    }
```

Deleting elements

Deleting an inner node or leaf

- The element to be deleted is an inner node or leaf.

```
// Start at the root node
Element it = root;
while (it != null) {
  if (c < it.value) {
    // if the value searched for is smaller, we must
        // look at the left, and if necessary delete the left
        // successor element; delegate
    if (it.left != null && it.left.value == c)
      return removeElement(it, it.left);
        it = it.left;
    } else {
    // similarly on the right.
    if (it.right != null && it.right.value == c)
      return removeElement(it, it.right);
        it = it.right;
    }
}
```

`removeElement` helper method

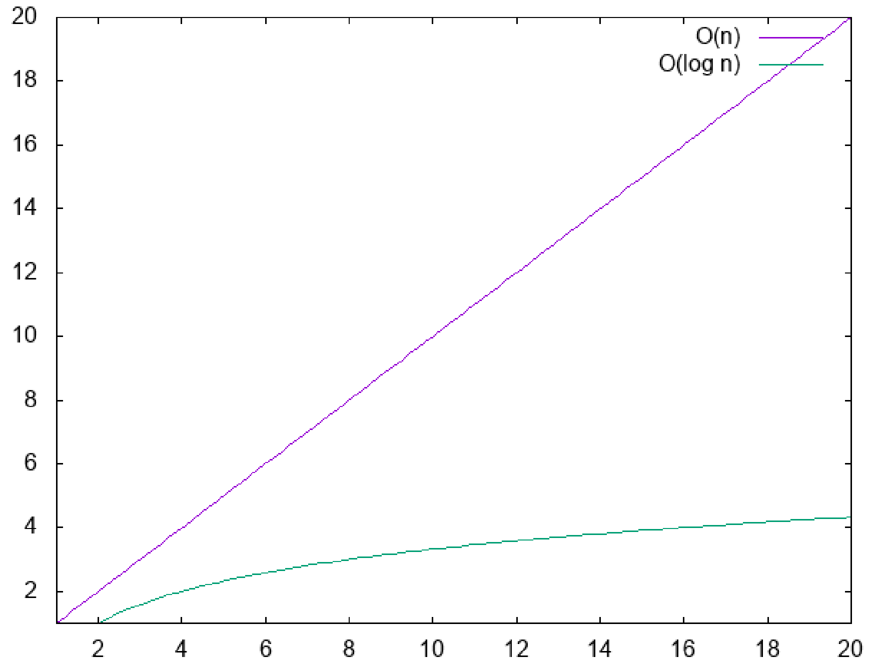
```
private char removeElement(Element parent, Element element) {
  // do we want to delete the left element?
  if (element == parent.left) {
        parent.left = null;
    } else {
        parent.right = null;
    }

  // reinsert any subtrees into the tree
  addElement(element.left);
    addElement(element.right);

  return element.value;
}
```

Complexity

- Why make the whole effort of using a tree if a list would be so much easier?
- We can prove with combinatorial mathematics that the average effort (complexity) for searching or inserting in a binary tree is O(log n); if the tree is *balanced* (i.e. the branching is balanced), then the effort is O(log n) even in the *worst case*.
- If we remember the introduction to this chapter: the naive implementation of a set as a list had the complexity O(n). This doesn't seem very different when reading it, but here too, a picture says more than 1000 words:
- We can see: with a linear increase in n (x-axis), the logarithmic slope very quickly remains far below the linear slope.
  So the tree structure is much more efficient!



Summary

- In contrast with a list, a set is free of duplicates; `add`, `remove`, `contains` and `size` are usually supported.
- Since a set does not have a sequence (order), there is no access via an index.
- A binary tree is a data structure in which elements have two successors; the smaller (left) and larger (right) elements are stored in these subtrees.
- Although a set can be implemented with a list, a binary tree is much more efficient.
- In the case of a complicated structure, recursion can often be an elegant solution; in this case a method calls itself again.