Data Structures and Algorithms

CatchUp Lecture Week 3

Recursion

- Basically speaking, an algorithm is called recursive if it calls itself

- Common examples: Fibonacci numbers, factorical, …

Fibonacci

```
public static double fibonacci(int n) throws Exception {
        if (n < 0) {
            throw new Exception("n should be positive");
        }

        if ((n == 0) || (n == 1)) {
            return 1;
        }

        return fibonacci(n - 1) + fibonacci(n - 2);
    }
```

Factorial

```
public static double factorial(int n) throws Exception {
        if (n < 1) {
            throw new Exception("n should be positive");
        }

        if (n == 1) {
            return 1;
        }

        return n * factorial(n - 1);
    }
```

Algorithm Analysis (1)

- To solve a given problem, several different algorithms may exist

- Q: Which algorithm is better?

Algorithm Analysis (2)

- To answer this question, a metric is needed:

- $\mathrm{Q}_{\mathrm{v}2}$: Which algorithm is better regarding $X*$?

$* \; X \in \{time, \; space\}$

Algorithm Analysis (3)

- Is that question good enough?

Algorithm Analysis (3)

- Is that question good enough?

- No!

Algorithm Analysis (3)

- Is that question good enough?

- No!

- Q_{v3}: Which algorithm is better regarding X*
  given problem size n?

* $X \in \{time,\ space\}$

Algorithm Analysis (4)

- The general answer to Q_{v3} generally is as follows:

The algorithms that needs less time / space.

Algorithm Analysis (4)

- The general answer to Q_{v3} generally is as follows:

The algorithms that needs less time / space.

… but how?!

Algorithm Analysis (5)

- A common approach to algorithm analysis is simply counting the steps / stored elements that occur while executing the algorithm with different input sizes n

- However, counting alone may produce strange results

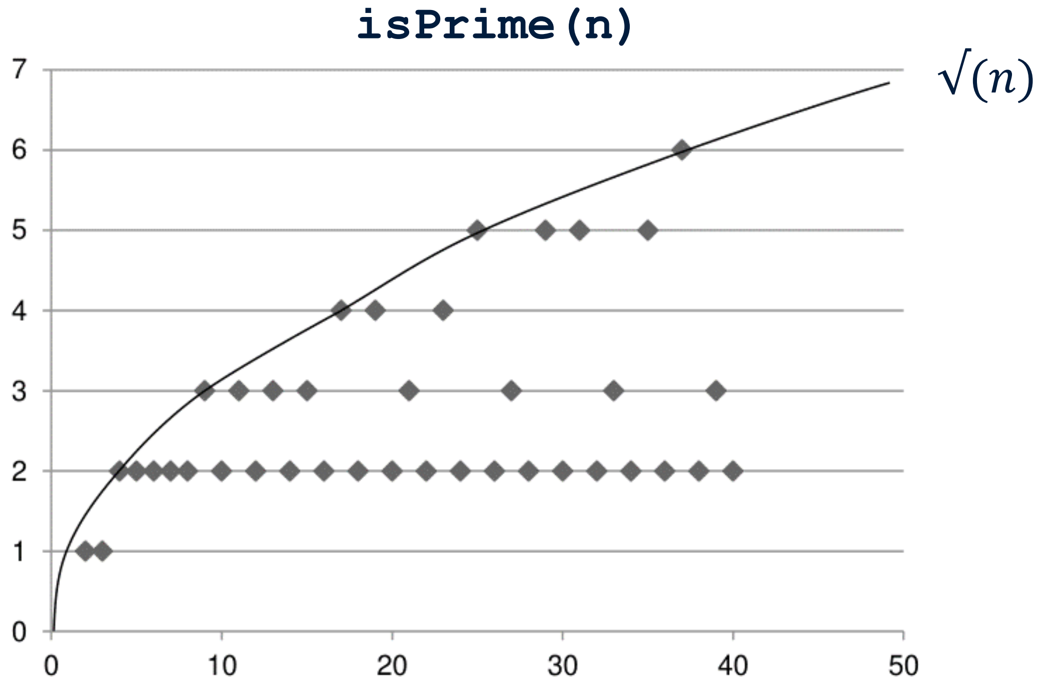
Finding Primes (1)

```
static boolean isPrime(int n) {
    // Corner case
    if (n <= 1) {
        return false;
    }

    // Check from 2 to sqrt(n)
    for (int i = 2; i <= Math.sqrt(n); i++) {
        if (n % i == 0)  {
            return false;
        }
    }

    return true;
}
```

Finding Primes (2)



Cases & Complexity Classes

- Best case, average case and worst case are distinguished

- Assign a problem to a complexity class and forget about the specific value from counting

Big-O (Landau) notation (1)

- Complexity classes are denoted using Big-O (Landau) notation:

$$O(f) = \left\{\min\left(g\colon \mathbb{N} \to \mathbb{R}^{+}\right) \mid f(n) \leq c * g(n)\ \forall\, n \in \mathbb{N}, c \in \mathbb{R}^{+}\right\}$$

Big-O (Landau) notation (2)

- Basically speaking:
  - The complexity class of a function f is the smallest function g in the set of all functions $g: \mathbb{N} \rightarrow \mathbb{R}^+$ such that $c * g(n) \geq f(n)$ for every $n \in \mathbb{N}$
  - That is called an “upper bound”

Big-O (Landau) notation (3)

Class	Name	Example
1	constant	A single instruction
$\log n$	logarithmic	Binary Search
n	linear	Unstructured search
$n * \log n$	superlinear / log linear	Efficient sorting
n^2	quadratic	Basic sorting
n^k	polynomial	Linear programming
2^n	exponential	Backtracking
$n!$	factorial	Permutations

What are ADTs? – Prerequisites (1)

Data Types

Atomic Datatypes

- basic and scalar Datatypes
- In Java: primitive Datatypes (boolean, char, byte, short, int, long, float, double)

Data Structures

- structured and composite Datatypes
- In Java: Objects
- Data structures can be homogeneous or heterogeneous

What are ADTs? – Prerequisites (2)

- Data structures adhere to the principles of object-oriented programming:

  - Encapsulation (→ next slide)
  - Inheritance: Objects acquire properties of parent objects from which they are derived
  - Polymorphism: Different classes have the same interface because of a common superclass, but may react differently

What are ADTs?

- ADTs are data structures adhering to two principles:
  - Encapsulation: Accessing the data structure only works by using pre-defined interfaces, e.g. functions
  - Opaqueness: The internal working principles are hidden from the user

What’s the difference between … (1)

- First, you have to distinguish two things:
  - ADTs and their implementations
  - Map vs. `HashMap`, List vs. `ArrayList`, …
- Normally, the latter part is the ADT and the first part gives a hint on the implementation
- There are always multiple ways of implementing

What's the difference between … (2)

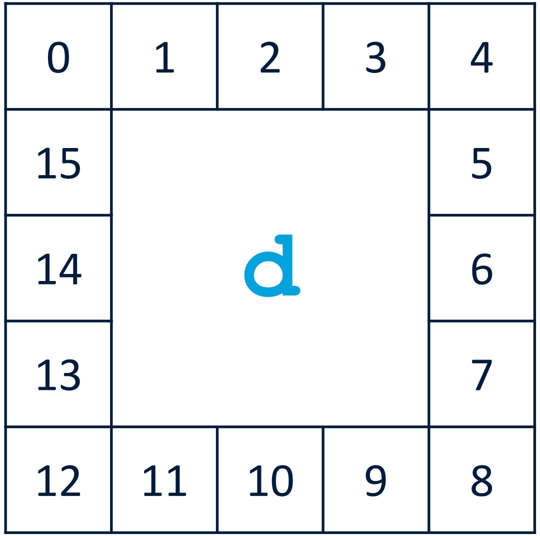
- Common ADTs:
  - (Array)
  - List (`ArrayList`, `LinkedList`)
  - Set (`HashSet`, `TreeSet`)
  - Stack
  - Queue
  - Deque
  - Map / Table (`HashMap`, `TreeMap`)

Arrays

- Arrays a composed of a fixed number of elements of the same data type

- Each element can be directly accessed
  - Access can be read or write
  - In Java: using the index operator `[]`

Circular Arrays



- Going clockwise

```
i = (i + 1) % d.length
```

- Going counter clockwise

```
i = (i - 1 + d.length) % d.length
```

Lists

- Basically speaking, a List is like an Array that can grow and shrink
- Two common implementations
  - `ArrayList`
  - `LinkedList`
- Operations: `add`, `get`, `contains`, `remove`

ArrayList (1)

- An `ArrayList` is a class based on an Array to store the data
- If the array is full (or nearly full), a new array is created an all elements are copied to the new array

ArrayList (2)

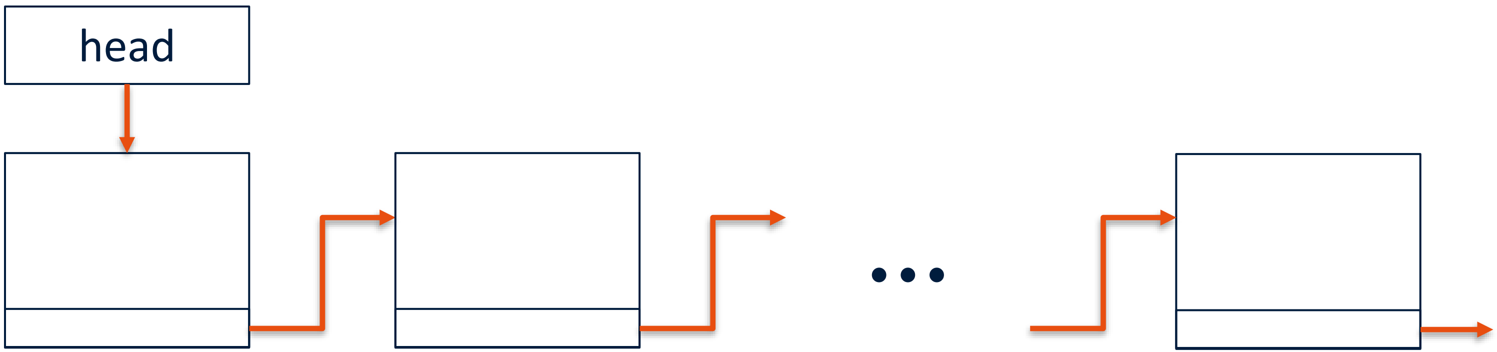
- Complexity of the operations:
  - `add`:
    - at the end: $O(1)$ (array not full), $O(n)$ (array full)
    - else: $O(n)$
  - `get`: $O(1)$
  - `contains`: $O(n)$
  - `remove`: $O(n)$

LinkedList

- A `LinkedList` is a data structure based on (doubly) linked node objects

- Each node object contains one data element and at least a link to the next node object

Single LinkedList (1)



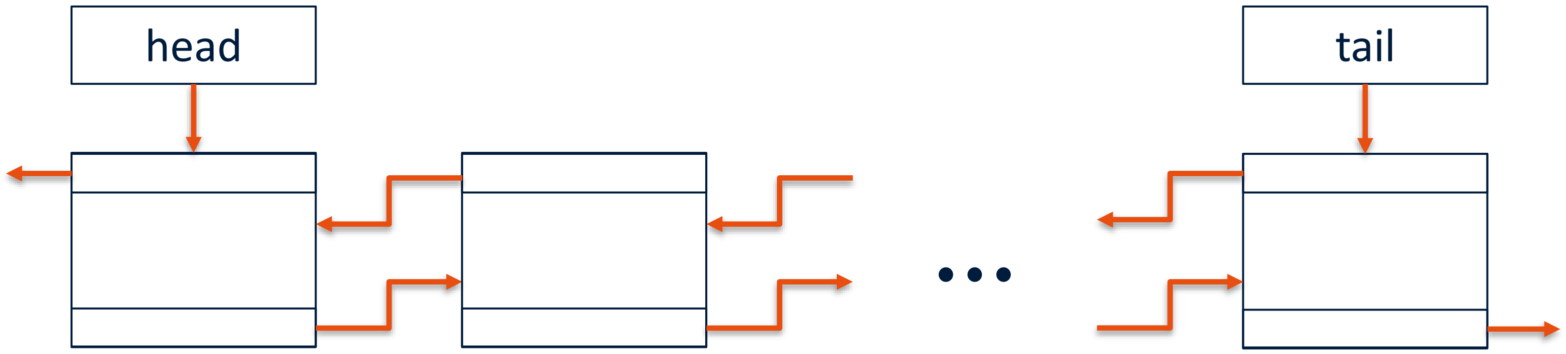

```
class List {
    Node head;
}
```

```
class Node {
    Object element;
    Node next;
}
```

Single LinkedList (2)

- Complexity of the operations:
  - `add`:
    - at the end / begin: $O(1)$
    - else: $O(n)$
  - `get`: $O(n)$
  - `contains`: $O(n)$
  - `remove`: $O(n)$

Double LinkedList (1)




```
class List {
    Node head;
    Node tail;
}
```

```
class Node {
    Object element;
    Node next;
    Node prev;
}
```

Double LinkedList (2)

- Complexity of the operations:
  - `add`, `get`, `contains`, `remove`:
    - at the begin / end: $O(1)$
    - else: $O(n)$

Iterating Arrays and Lists

- Arrays / ArrayLists

```
for (int i = 0; i < a.length; i++) {
    …
}
```

- LinkedLists

```
for (Node n = head; n != null; n = n.next) {
    …
}
```

Stacks

- Can be easily build based on an `ArrayList` or a doubly `LinkedList`
- Adding and Removing always happens at the end of the Stack (FIFO)
- Operations: `push`, `pop`, `peek`

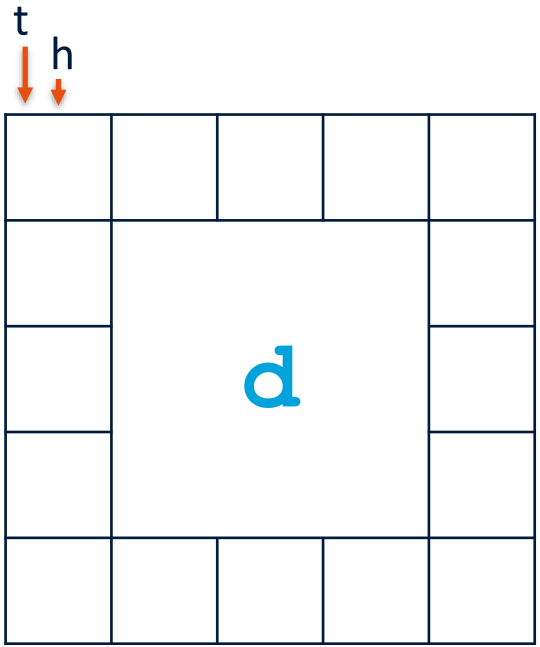
Queues (1)

- Can be easily build based on a circular Array doubly `LinkedList`

- Adding happens at the end of the queue, removing always at the beginning (LIFO)

- Operations: `put`, `get`, `peek`

Queues (2)

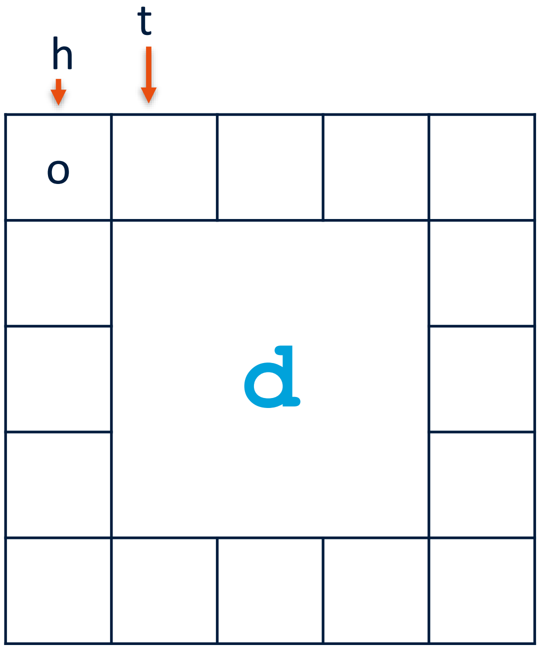



```
public boolean isEmpty() {
    return head == tail;
}


public void put(Object o) {
    data[tail] = o;
    tail = (tail + 1) % data.length;

    if (head == tail) {
        resize();
    }
}
```

Queues (3)

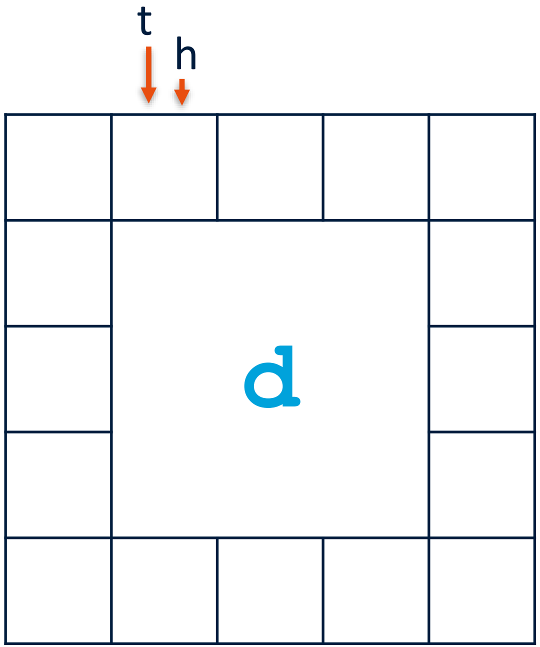



```
public Object get() {
     if (isEmpty()) {
          return null;
     }

     Object r = data[head];
     data[head] = null;
     head = (head + 1) % data.length;

     return r;
}
```

Queues (3)




```
public Object get() {
    if (isEmpty()) {
        return null;
    }

    Object r = data[head];
    data[head] = null;
    head = (head + 1) % data.length;

    return r;
}
```

Sets and Maps (1)

- A set is an ADT that represents a collection of elements of the same data type
  - Sets are unordered
  - Each element can occur only once in the set

- Operations: `add`, `contains`

Sets and Maps (2)

- A map is an ADT that represents a collection of elements of the same data type accessible by a unique key
  - → The keys of a map form a set

- Operations: `add`, `get`, `contains`

Sets and Maps (3)

- A map can be used as a set when only adding dummy data

- Implementations using Hashes and Trees* are common (`HashSet` & `HashSet` vs. `TreeSet` & `TreeMap`)

* We'll come to that later!

Sets and Maps (4)

- Complexity of the operations of a Set:
  - `HashSet` / `TreeSet`
    - `contains`: $O(1)$ / $O(\log n)$
    - `add`:
      - If is full: $O(n)$ / $O(\log n)$
      - else: $O(1)$ / $O(\log n)$

Sets and Maps (5)

- Complexity of the operations of a Map:
  - **`HashMap`** / **`TreeMap`**
    - `contains`: $O(1)$ / $O(\log n)$
    - `get`: $O(1)$ / $O(\log n)$
    - `add`:
      - If is full: $O(n)$ / $O(\log n)$
      - else: $O(1)$ / $O(\log n)$

Iterating Sets

```
Set<T> s = …;

…

Iterator<T> it = s.iterator();

while (it.hasNext()) {
    T t = it.next();
    …
}
```

Iterating Maps

```java
Map<K, V> m = …;

…

Set<Map.Entry<K,V>> s = m.entrySet();
Iterator<Map.Entry<K,V>> it = s.iterator();

while (it.hasNext()) {
    Map.Entry<K, V> kv = it.next();
    K key = kv.getKey();
    V val = kv.getValue();
    …
}
```

Hashing

- A hash function $h(x)$ is a function that maps an arbitrary input x to an output of fixed-size
- Hash functions are generally not injective

ADTs based on Hashing (1)

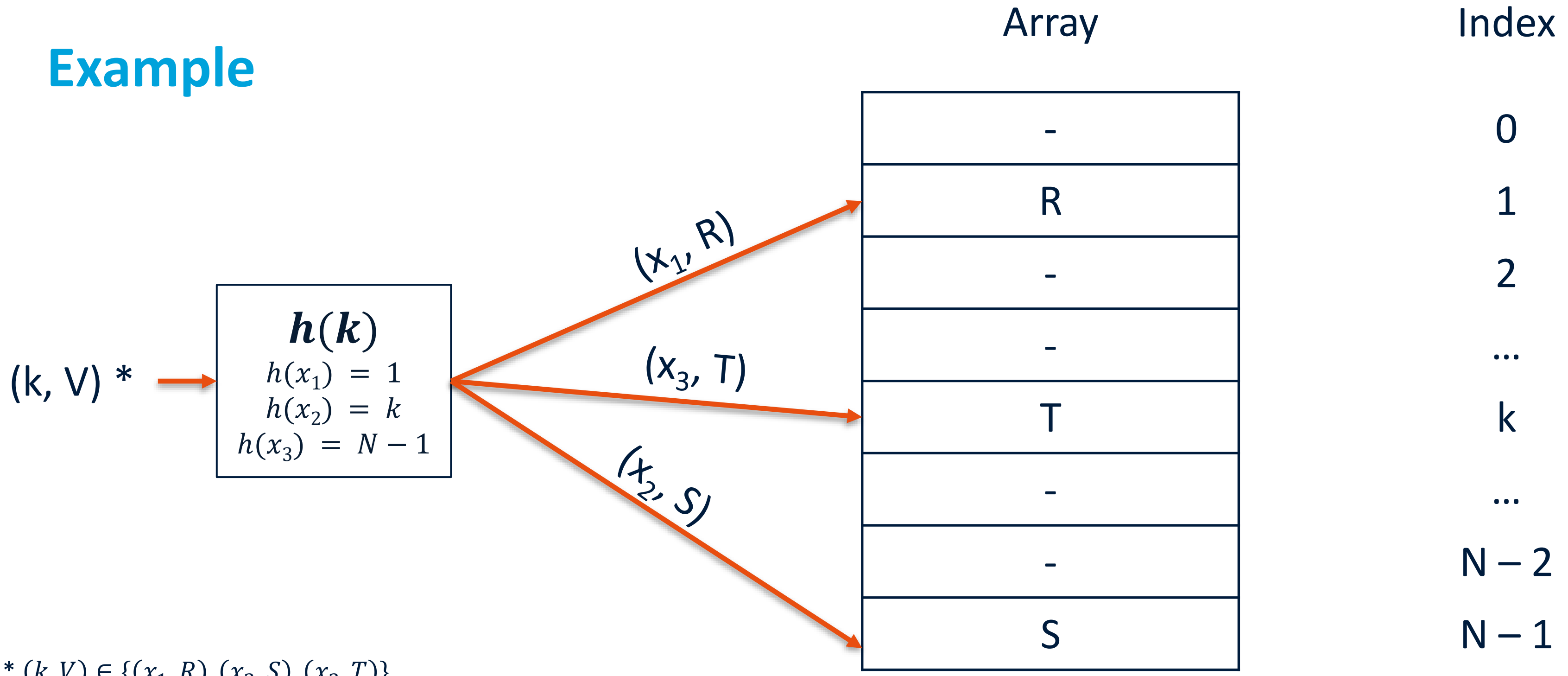
- Approach: Hashing with modulo on arrays

$$h(x) = h'(x)\ mod\ N$$

- That is: $h(x)$ is the result of another function $h'(x)$ modulo N, where N is the size of the underlying array

ADTs based on Hashing (2)

Example



* $(k, V) \in \{(x_1, R), (x_2, S), (x_3, T)\}$

ADTs based on Hashing (3)

- Problem: There will be collisions
  → Need to store key-value pairs either way, to be able to uniquely identify stored values

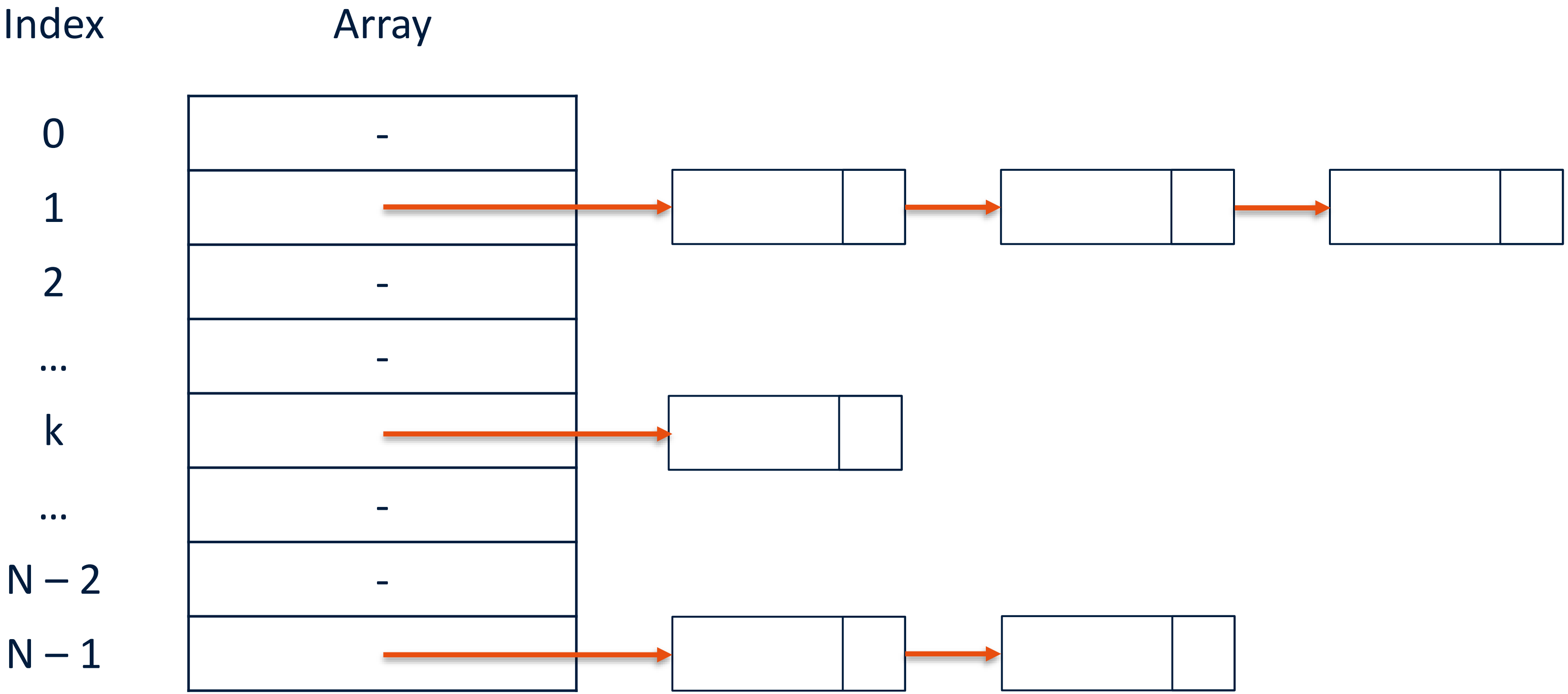
- Possible solutions:
  - Separate Chaining
  - Open Adressing

Separate Chaining

- Use the underlying array not to store the data, but to store lists

- Each list contains all elements that were mapped to the same index

Separate Chaining



Open Addressing

- Resort to another hash function in case a collision occurs

- Can be done n-fold

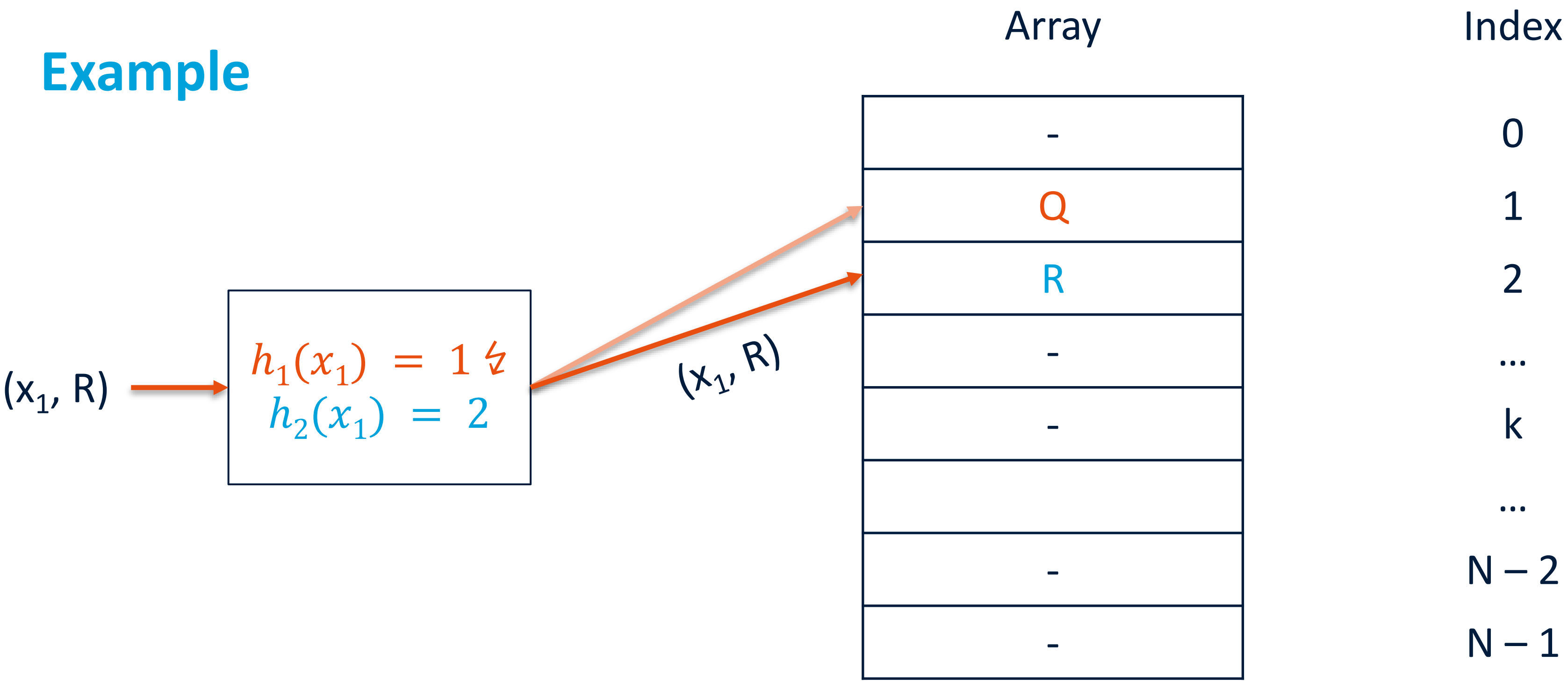
Open Addressing

- Different common approaches
  - Linear probing: $h_{2(x)} = h_{1(x)} + k$
  - Quadratic probing: $h_{2(x)} = h_{1(x)} + k^2$

k starts at 1 and is incremented by one for each step

Open Addressing

Example

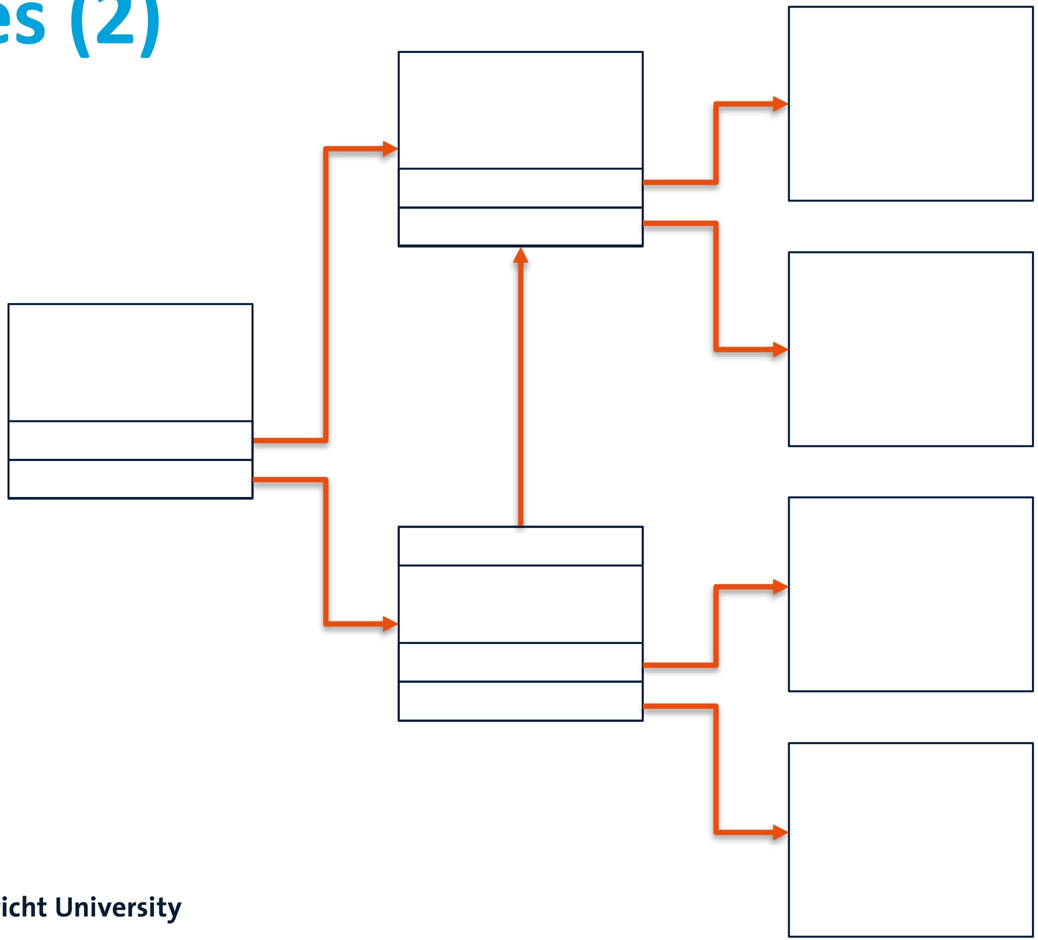


Trees

- If in a `LinkedList` there is not only one next node, but multiple next nodes, then this is called a Tree

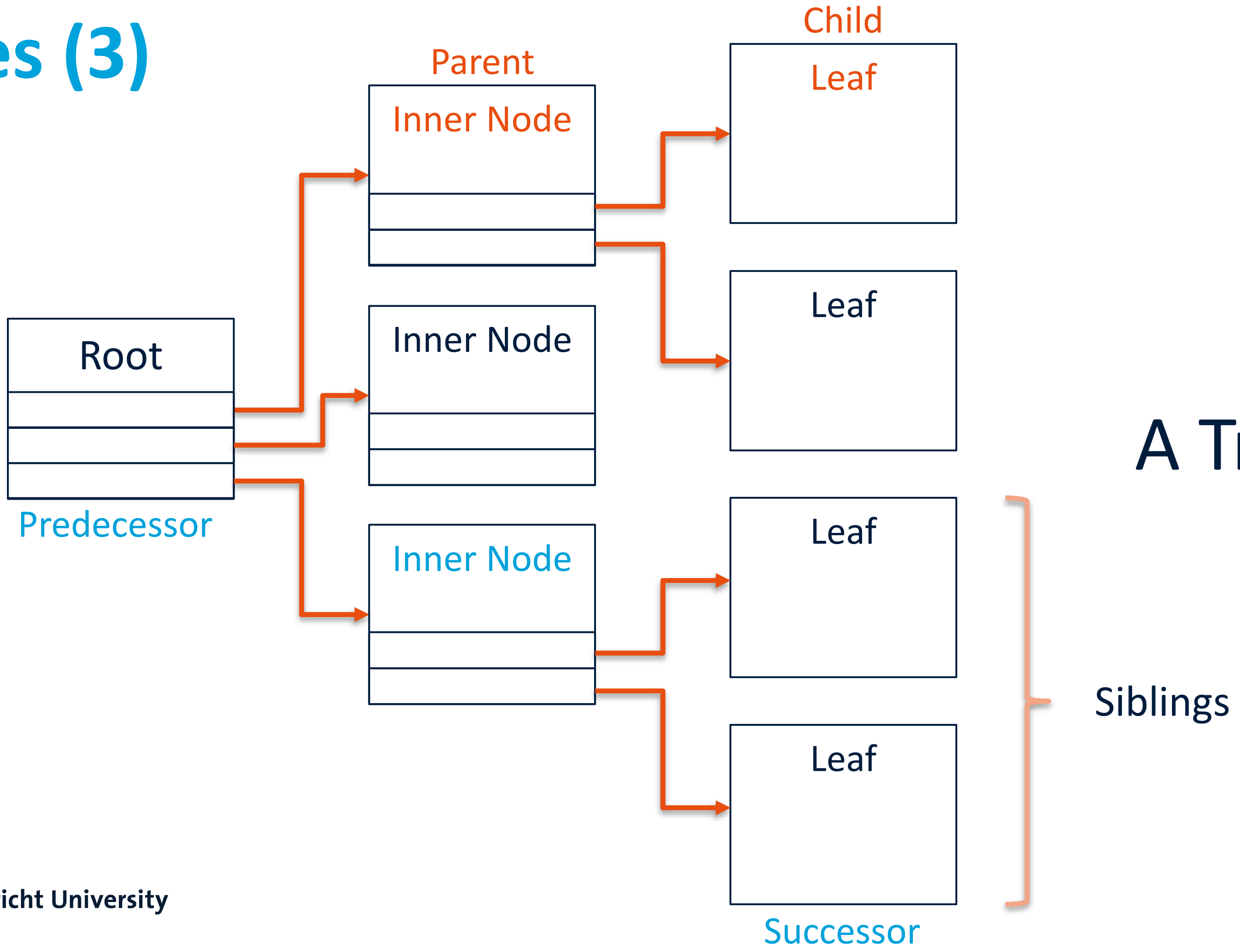
- Two distinct nodes in the Tree are connected by exactly one path, otherwise it is a Graph

Trees (2)



Not a Tree

Trees (3)



A Tree

Trees (3)

- Each node in a Tree has a level
  - The root has level 0
  - Each other node has the level of the parent node plus 1

- The height of a tree is equal to the maximum level of any node

Binary Trees (1)

- A binary tree is a tree in which every node has two children at maximum

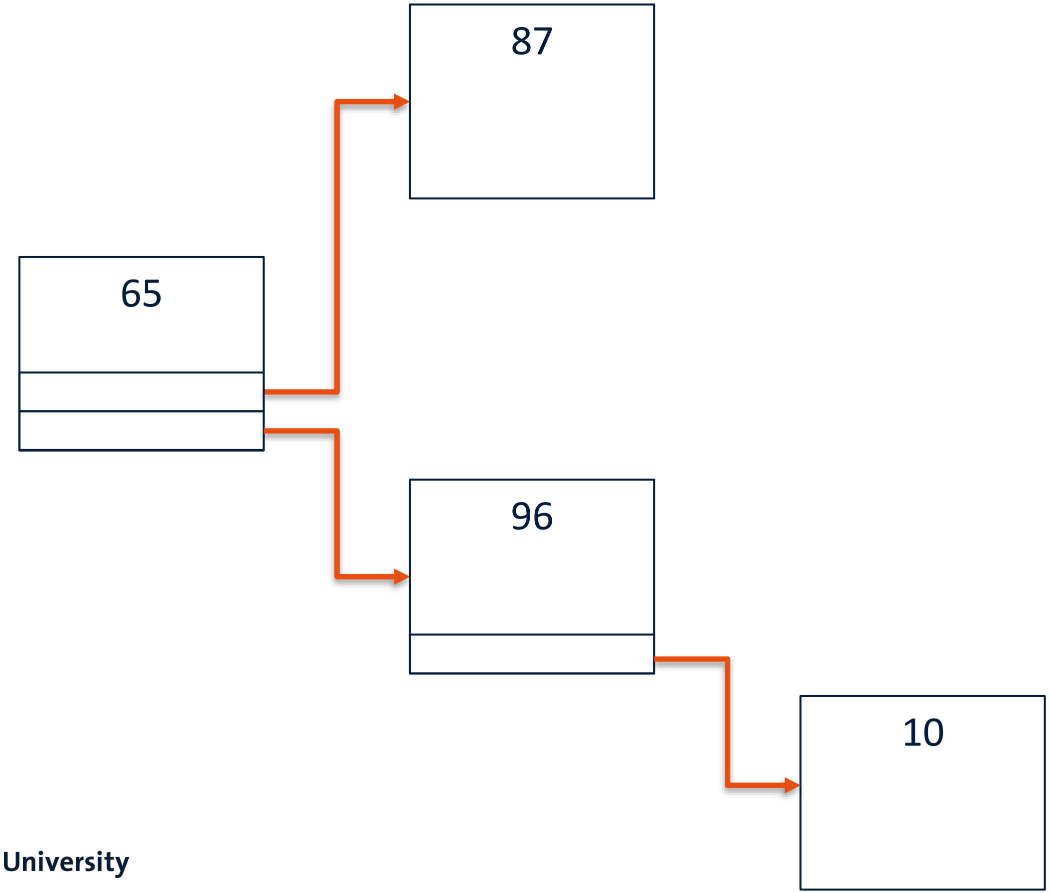
- In a balanced binary tree, all leaf nodes have the same level

Binary Trees (2)

- A complete binary tree is a tree, in which all levels of the tree are filled

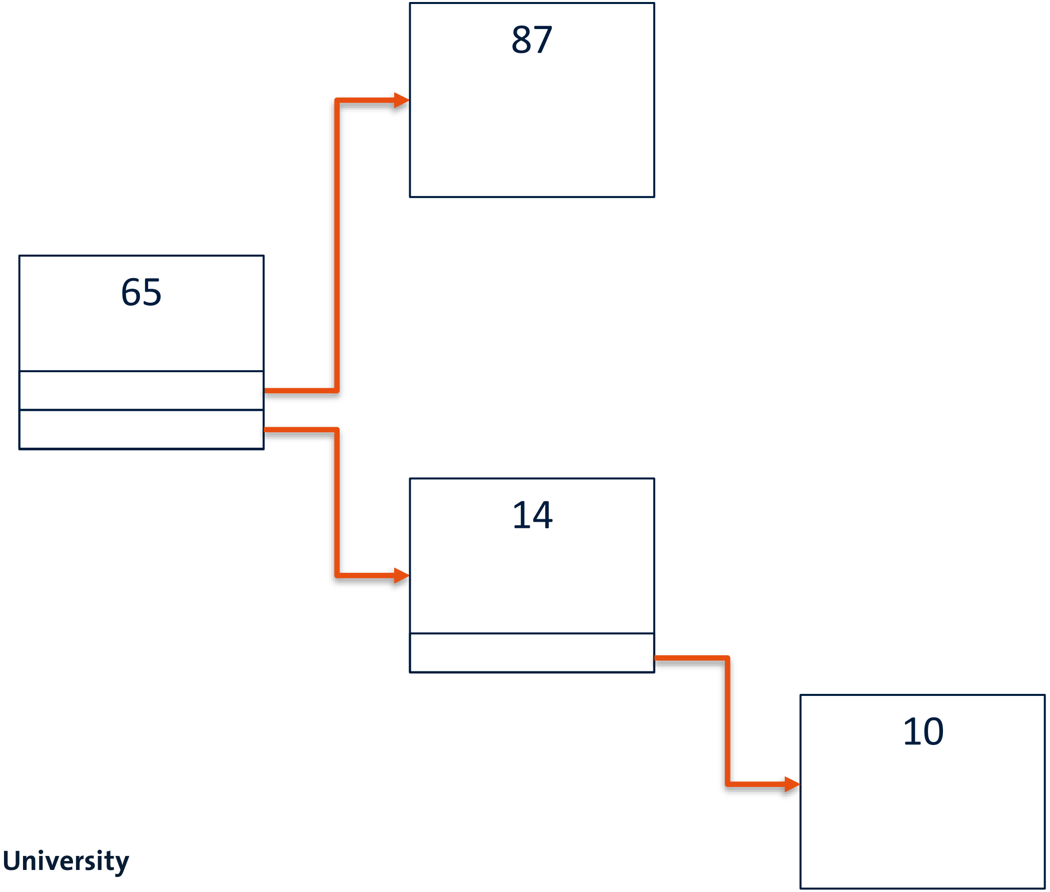
- A binary search tree is a tree, in which all children to the left have a lower value than the parent and all children to the right have a higher value than the parent

Binary Trees (3)



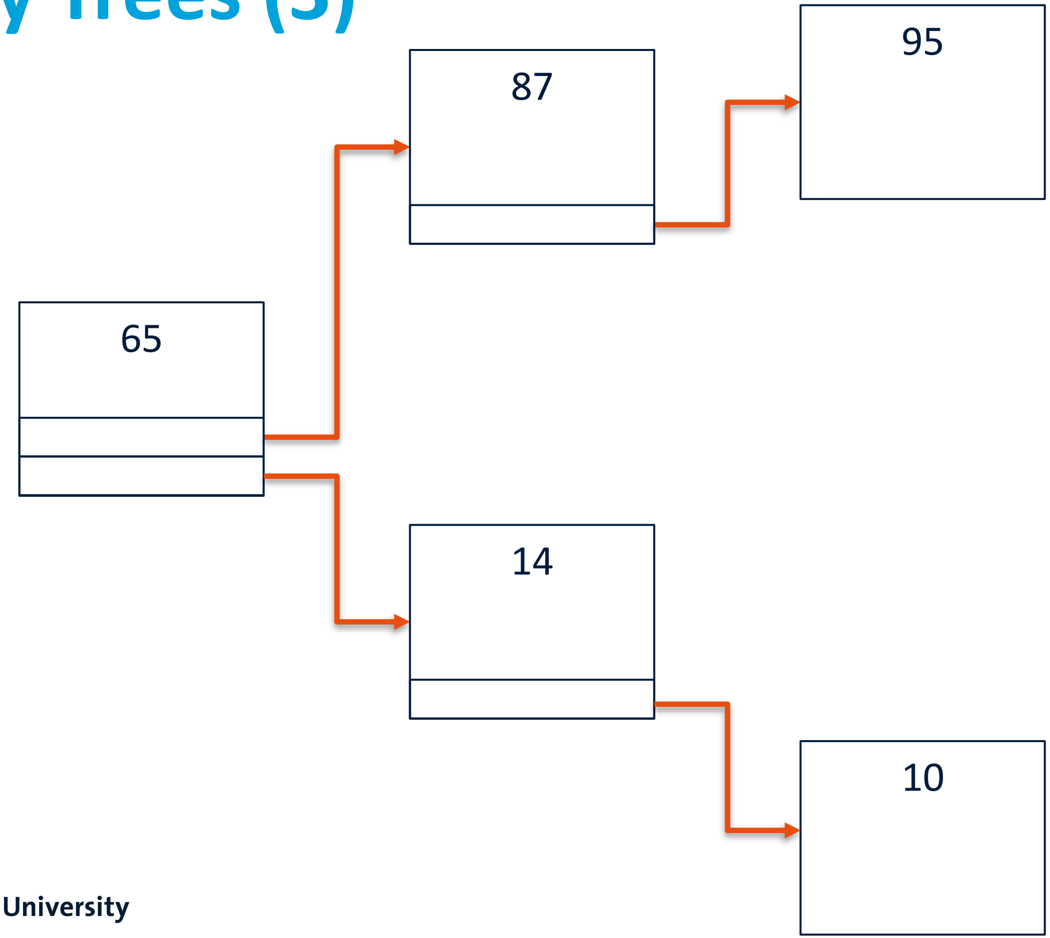
A
binary
tree

Binary Trees (3)



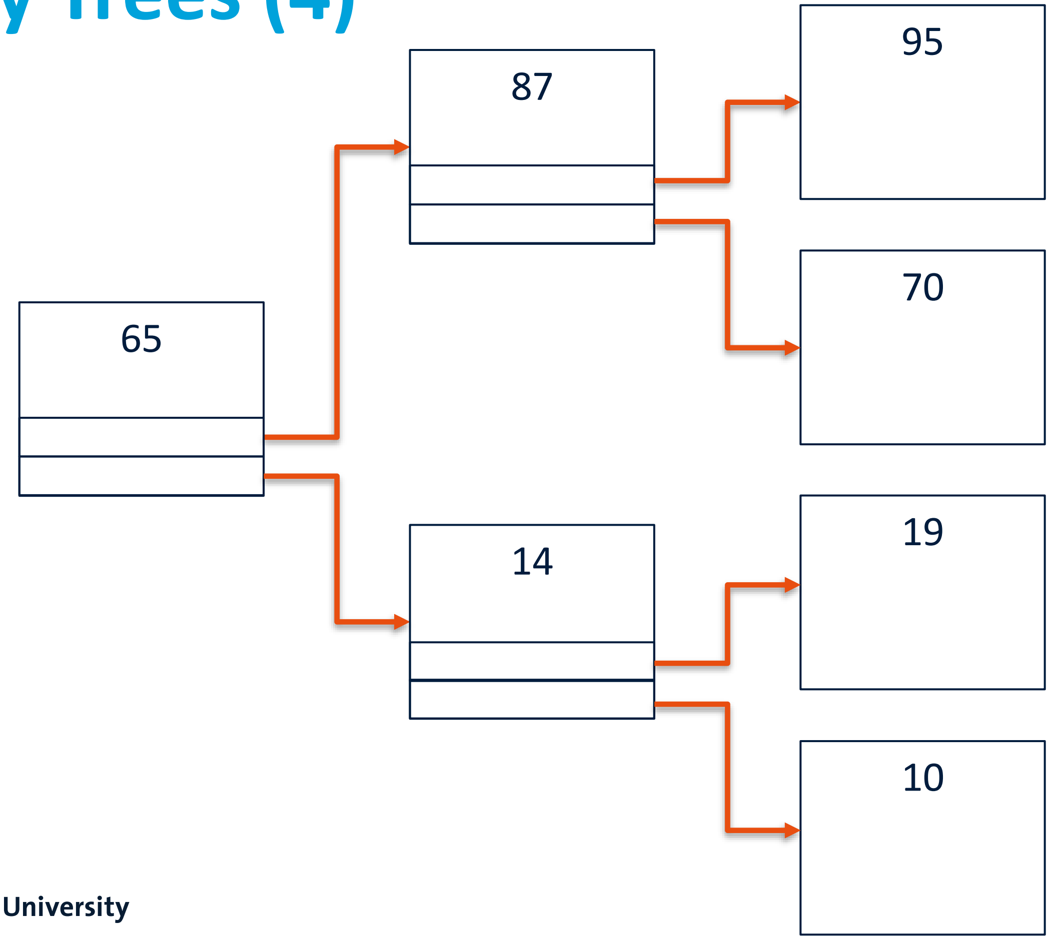
A binary search tree

Binary Trees (3)



A balanced binary search tree

Binary Trees (4)



A balanced complete binary search tree

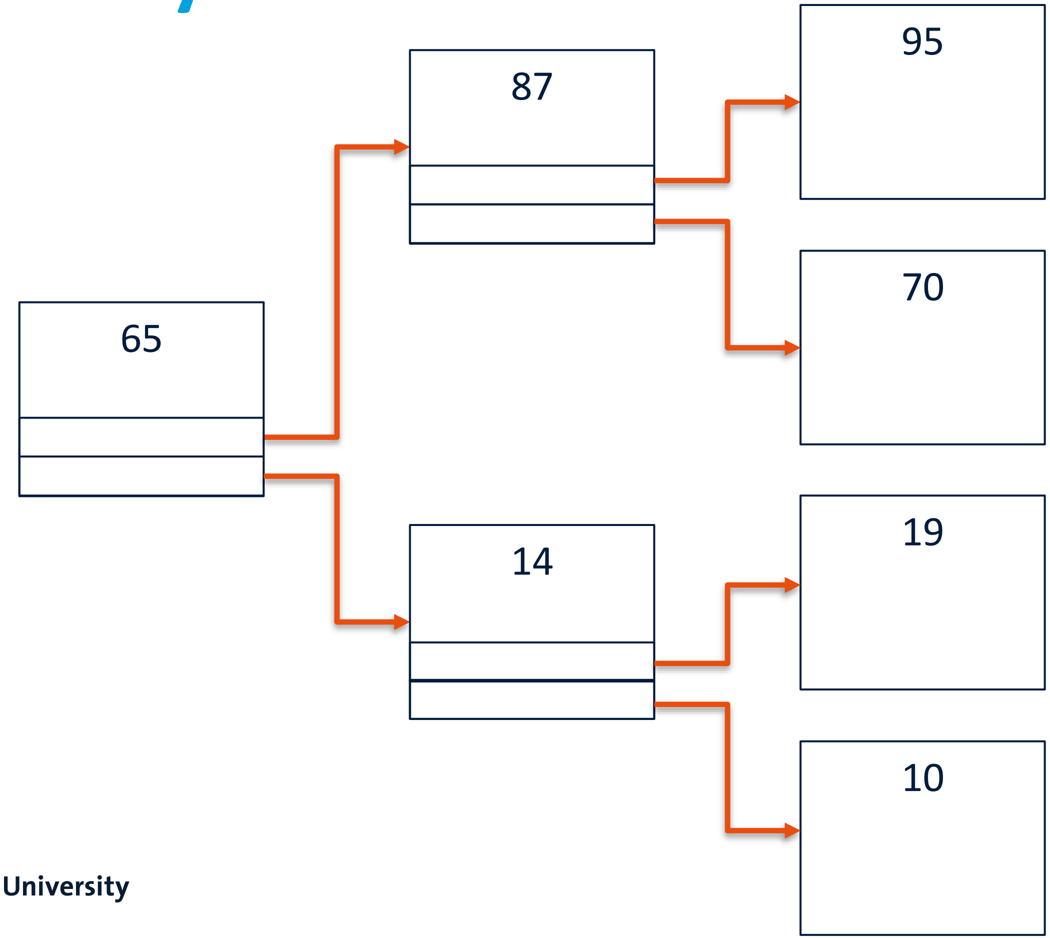
Complexity of a Tree

- In a complete binary search tree with N nodes, you will have to traverse $\log N$ nodes on average to find a node

- You have to find a node to delete it

- You have to "not find" a node to insert it

Complexity of a Tree



$$N = 6$$
$$\log N \cong 2.58$$