Algorithms and Data Structures

Stack, Queues, and Applications

Marius Kloft

Content of this Lecture

- Stacks and Queues
- Tree Traversal
- Towers of Hanoi

Stacks and Queues

- Recall these two fundamental ADTs

```
type stack( T)
import
  bool;
operators
  isEmpty: stack → bool;
  push:    stack x T → stack;
  pop:     stack → stack;
  top:     stack → T;
```

```
type queue( T)
import
  bool;
operators
  isEmpty: queue → bool;
  enqueue: queue x T → queue;
  dequeue: queue → queue;
  head:    queue → T;
```

- Properties
  - Stacks always add / remove the first element
    - Add and remove from right - LIFO
  - Queues always add the first element and remove the last element
    - Add from right, remove from left - FIFO

Implementation

- Which are better for implementing stacks & queues: arrays, linked lists, or double-linked lists?

	Array	**Linked list**	**Double-linked l.**
Insert	O(n)	O(n)	O(n)
InsertAfter	O(n)	O(1)	O(1)
Delete	O(n)	O(n)	O(n)
DeleteThis	O(n)	O(n)	O(1)
Search	O(n)	O(n)	O(n)
Add to start	O(n)	O(1)	O(1)
Add to end	O(1)	O(n)	O(1)

Implementation

- Stacks
  - Always add / remove at the front
  - Efficiently supported by linked lists or double-linked lists
- Queues
  - Always add at the front and remove from the back
  - Efficiently supported by double-linked lists with pointer to first and last element
  - Adding a “last” pointer to a single-linked list is also enough

	Array	**Linked list**	**Double-linked l.**
Insert	O(n)	O(n)	O(n)
InsertAfter	O(n)	O(1)	O(1)
Delete	O(n)	O(n)	O(n)
DeleteThis	O(n)	O(n)	O(1)
Search	O(n)	O(n)	O(n)
Add to start	O(n)	O(1)	O(1)
Add to end	O(1)	O(n)	O(1)

Content of this Lecture

- Stacks and Queues
- Tree Traversal
  - Application
  - Depth-First using Stacks
  - Breadth-First using Queues
- Towers of Hanoi

Application

- Information systems is a class of software systems that is concerned with managing (and analyzing) data
  - Customers of a company, calls of a telecom company, etc.
- „Managing“ means
  - Storing, being fail-safe, allowing concurrent read and write access, offering comfortable (and fast) ways of accessing the data
    - „All customers older than 55 which purchased goods worth more than 30K in the last 6 months and that did never before buy a Rolex“
  - See course on Databases
- Analyzing can mean
  - Discover interesting relationships
    - Customers who buy X, are likely to buy also Y
=> Show ad banner of Y
  - See Machine Learning course

Data Models

- Data managed within a database needs to be modeled
  - Which data do we store?
- One particularly comfortable data model is called XML
  - XML: Extended Markup Language
  - Allows to model (and define) hierarchical data structures
- Central elements: Elements and values
  - Elements are names of values or of groups of values
  - Elements have an opening and a closing tag (<x></x>)
  - Values store the actual data values

Example – Elements and Values

```
<customers>
  <customer>
    <last_name>
      Müller
    </last_name>
    <first_name>
      Peter
    </first_name>
    <age>
      25
    </age>
  </customer>
 <customer>
    <last_name>
      Meier
    </last_name>
    <first_name>
      Stefanie
    </first_name>
    <age>
      27
    </age>
  </customer>
</customers>
```

- XML is verbose …
- But can be compressed well
- Not necessarily a model for storage

Example

```
<customers>
  <customer>
    <last_name>
      Müller
    </last_name>
    <first_name>
      Peter
    </first_name>
    <age>
      25
    </age>
  </customer>
 <customer>
    <last_name>
      Meier
    </last_name>
    <first_name>
      Stefanie
    </first_name>
    <age>
      27
    </age>
  </customer>
</customers>
```

- Production rules

  customers -> cust

  cust -> customer

  cust -> customer, cust

  customer -> last_name, first_name, age

  last_name -> *

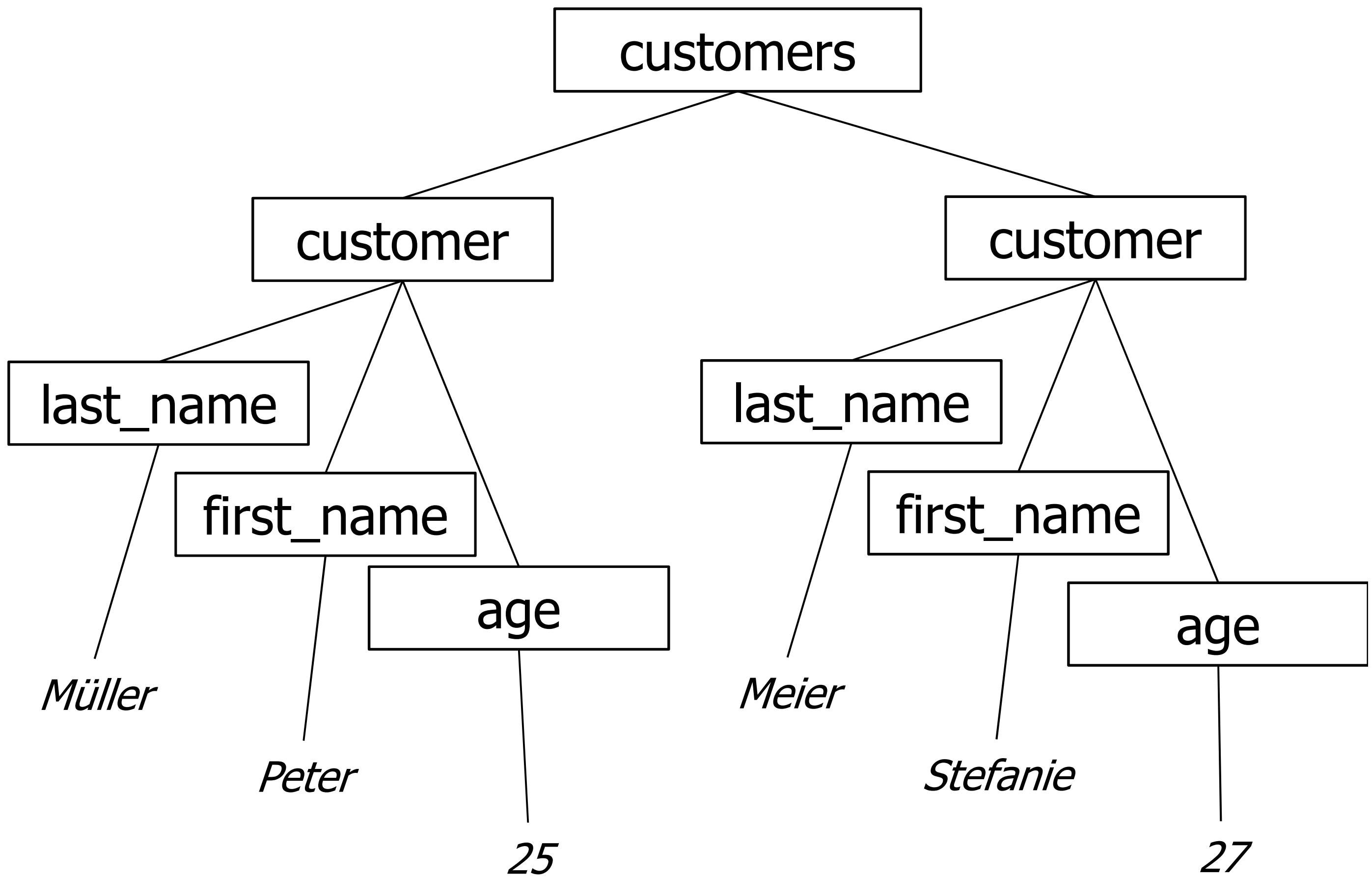
  first_name -> *

  age -> *

Data – A Tree

- The elements and values of an XML doc form a tree

```
<customers>
  <customer>
    <last_name>
      Müller
    </last_name>
    <first_name>
      Peter
    </first_name>
    <age>
      25
    </age>
  </customer>
 <customer>
    <last_name>
      Meier
    </last_name>
    <first_name>
      Stefanie
    </first_name>
    <age>
      27
    </age>
  </customer>
</customers>
```



Implementing a Tree

```
class element {
  value: String;
  children: list_of_element;
}
```

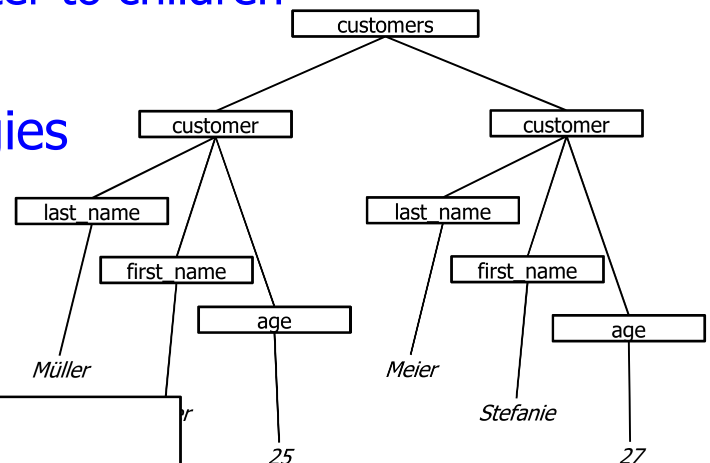
```
class XMLDoc {
  root: element;
  func void init()
  func element getRoot()
  func String printTree() {
    ? How ?
  }
}
```

```
<customers>
  <customer>
    <last_name>
      Müller
    </last_name>
    <first_name>
      Peter
    </first_name>
    <age>
      25
    </age>
  </customer>
 <customer>
    <last_name>
      Meier
    </last_name>
    <first_name>
      Stefanie
    </first_name>
    <age>
      27
    </age>
  </customer>
</customers>
```

```
                         customers
           customer                        customer
last_name first_name age       last_name first_name age
  Müller     Peter     25        Meier      Stefanie   27
```

Two Strategies

- For both cases, we need to traverse the tree
  - Start from root and recursively follow pointer to children
  - Fortunately, we cannot run into cycles
- But they require different traversal strategies
  - Depth-first: From root, always follow the left-most child until you reach a leaf; then follow second-left-most …
  - Breadth-first: From root, first look at all children, then at all grand-children, then … (always from left to right)




```
<customers>
  <customer>
    <last_name>
      Müller
    </last_name>
    <first_name>
      Peter
    </first_name>
    <age>
      25
    </age>
  </customer>
 <customer>
    <last_name>
      Meier
    </last_name>
    <first_name>
      Stefanie
    </first_name>
    <age>
      27
    </age>
  </customer>
</customers>
```

```
                         customers
           customer                          customer
last_name first_name age          last_name first_name age
  Müller     Peter    25             Meier     Stefanie   27
```

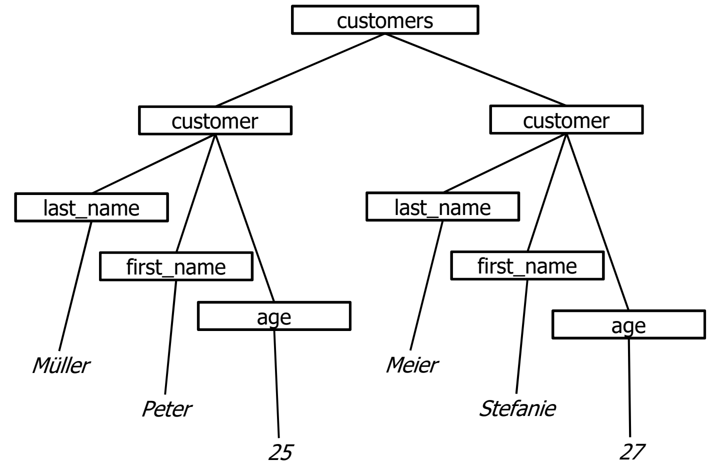
Content of this Lecture

- Stacks and Queues
- Tree Traversal
  - Application
  - Depth-First using Stacks
  - Breadth-First using Queues
- Towers of Hanoi

Depth-First Traversal (no indentation)

```
func String printDFS (t Tree) {
  s := new Stack();
  o := "";
  node : treeElement;
  s.push( t.getRoot());
  while not s.isEmpty() do
    node := s.pop();
    o := o+node.getValue()+"\lf";
    c := node.getChildren();
    foreach x in c do
      s.push( x);
    end for;
  end while;
  return o;
}
```

We assume that elements have their name as value




```
<customers>
  <customer>
    <last_name>
      Müller
    </last_name>
    <first_name>
      Peter
    </first_name>
    <age>
      25
    </age>
  </customer>
 <customer>
    <last_name>
      Meier
    </last_name>
    <first_name>
      Stefanie
    </first_name>
    <age>
      27
    </age>
  </customer>
</customers>
```

DFS-2

```
s.push( root);
while not s.isEmpty() do
  node := s.pop();
  o := o+node.getValue();
  # print s, o;
  c := node.getChildren();
  foreach x in c do
    s.push( x);
  end for;
  # print s, o;
end while;
```

Output o:

customers	customers	customers customer2	customers customer2	customers customer2 age

Stack s:

	customer2 customer1	customer1	age first_name last_name customer1	first_name last_name customer1

DFS-3

```
s.push( root);
while not s.isEmpty() do
  node := s.pop();
  o := o+node.getValue();
  # print s, o;
  c := node.getChildren();
  foreach x in c do
    s.push( x);
  end for;
  # print s, o;
end while;
```

customers customer2 age	...	customers customer2 age 25 Peter Müller	customers customer2 age 25 last_name Peter first_name Müller customer1	customers customer2 age 25 last_name Peter first_name Müller customer1	...
25 first_name last_name customer1	...	customer1		age first_name last_name	...

Adding Indentation

- We need to also store the depth of a node on the stack
  - We assume a generic, type-independent stack

```
s.push( root);
s.push( 1);
while not s.isEmpty() do
  depth := s.pop();
  node := s.pop();
  o := o+ SPACES(depth) +node.getValue();
  c := node.getChildren();
  foreach x in c do
    s.push( x);
    s.push( depth+1);
  end if;
end while;
```

```
customers
  customer2
    age
      25
    first_name
      Peter
    last_name
      Müller
  customer1
    …
```

Reverting Order

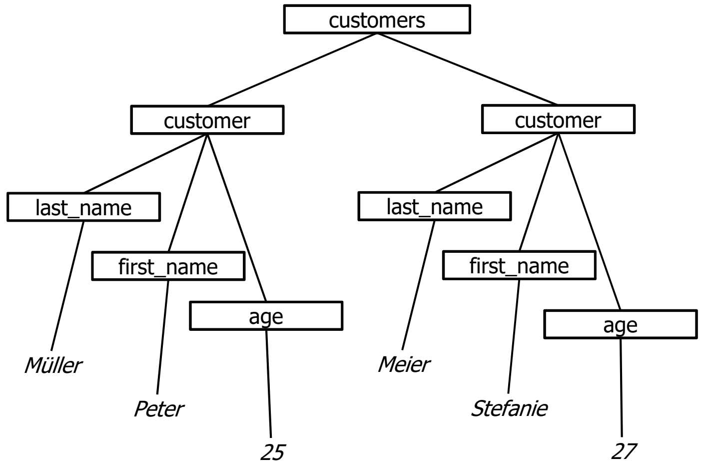
- We create customer2 … customer1 – but we wanted customer1 … customer2
- The order of children is reverted by the stack
- Remedy
  - Push children in reverted order
  - Can be achieved by a FOREACH which traverses a list in reverted order
  - Easy if a double-linked list is used

Content of this Lecture

- Stacks and Queues
- Tree Traversal
  - Application
  - Depth-First using Stacks
  - Breadth-First using Queues
- Towers of Hanoi

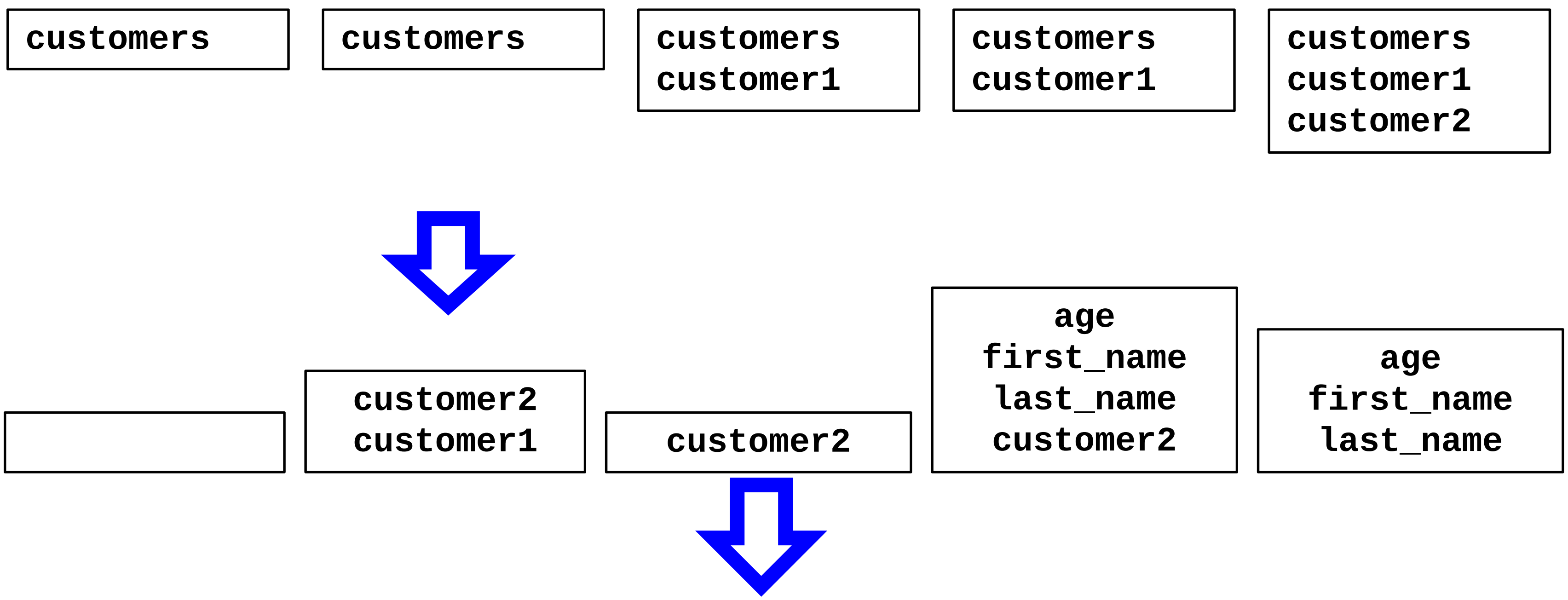
Breadth-First Traversal

```
Func String printBFS (t Tree) {
  q := new Queue();
  o := “”;
  node : element;
  q.enqueue( t.getRoot());
  while not q.isEmpty() do
    node := q.dequeue();
    o := o+node.getValue();
    c := node.getChildren();
    foreach x in c do
      q.enqueue( x);
    end if;
  end while;
}
```




```
                          customers
          customer                          customer
last_name first_name age        last_name first_name age
  Müller     Peter    25          Meier     Stefanie  27
```

BFS-2


customers
customers
customers
customer1
customers
customer1
customers
customer1
customer2
customer2
customer1
customer2
age
first_name
last_name
customer2
age
first_name
last_name

BFS-3

```
customers
customer1
customer2
```

```
customers
customer1
customer2
```

```
customers
customer1
customer2
last_name
```

```
customers
customer1
customer2
last_name
```

```
customers
customer1
customer2
last_name
first_name
```

```
age
first_name
last_name
```

```
age
first_name
last_name
age
first_name
last_name
```

```
age
first_name
last_name
age
first_name
```

```
Müller
age
first_name
last_name
age
first_name
```

```
Peter
Müller
age
first_name
last_name
age
```

- If we add information about the depth of a node, we can put elements of same depth at the same line of the output

Time Complexity

- The complexity of the traversal is O(n) in both cases
  - n = number of nodes in the tree
  - Each node is pushed (enqueued) once and popped (dequeued) once
- Thus, the foreach loop is passed by (n-1) times altogether
- The style of argument is different from what we had so far
  - Recall SelectionSort
  - We have two nested loops in both algorithms

```
printBFS:
while not q.isEmpty() do
  foreach x in c do
    …
  end for;
end while;
```

```
SelectionSort:
for i = 1..n-1 do
  for j = i+1..n do
    …
  end for;
end for;
```

Explanation

```
printBFS:
while not q.isEmpty() do
  foreach x in c do
    …
  end for;
end while;
```

```
SelectionSort:
for i = 1..n-1 do
  for j = i+1..n do
    …
  end for;
end for;
```

- In printBFS, we do not know how often the inner loop is passed-through for a specific iteration of the outer loop
  - We cannot sensibly estimate this number – depends on the number of children, not on the concrete iteration of the outer loop
  - But we can directly count how often the inner loop is passed over all iterations of the outer loop
  - This is possible because we know that no element is touched twice
- In SelectionSort, we do know how often the inner loop is passed-through for every iteration of the outer loop
  - Obviously, n-i-1 times
  - But we have no simple estimation for the number of times the inner loop is passed-through over all iterations of the outer run
  - This is because we touch elements multiple times

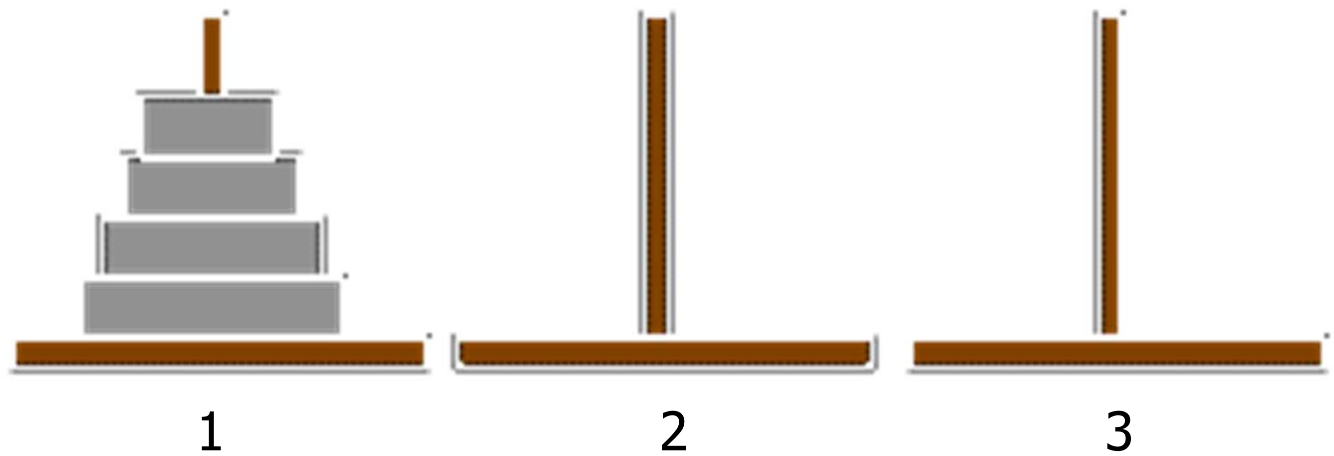
Space Complexity

- Time complexity is the same for DFS and BFS, but space complexity is different
- Let d be the depth of the tree (length of longest path)
- Let b be the breadth of the tree
  - Maximal number of nodes with same depth over all levels
- Let c be the maximal number of children of any node
- In DFS, the stack holds at most d*c elements
- In BFS, the queue holds at most b elements
- That's a big difference in typical database settings
  - Little nesting (small d), but hundreds of thousands of customers (large b)

Content of this Lecture

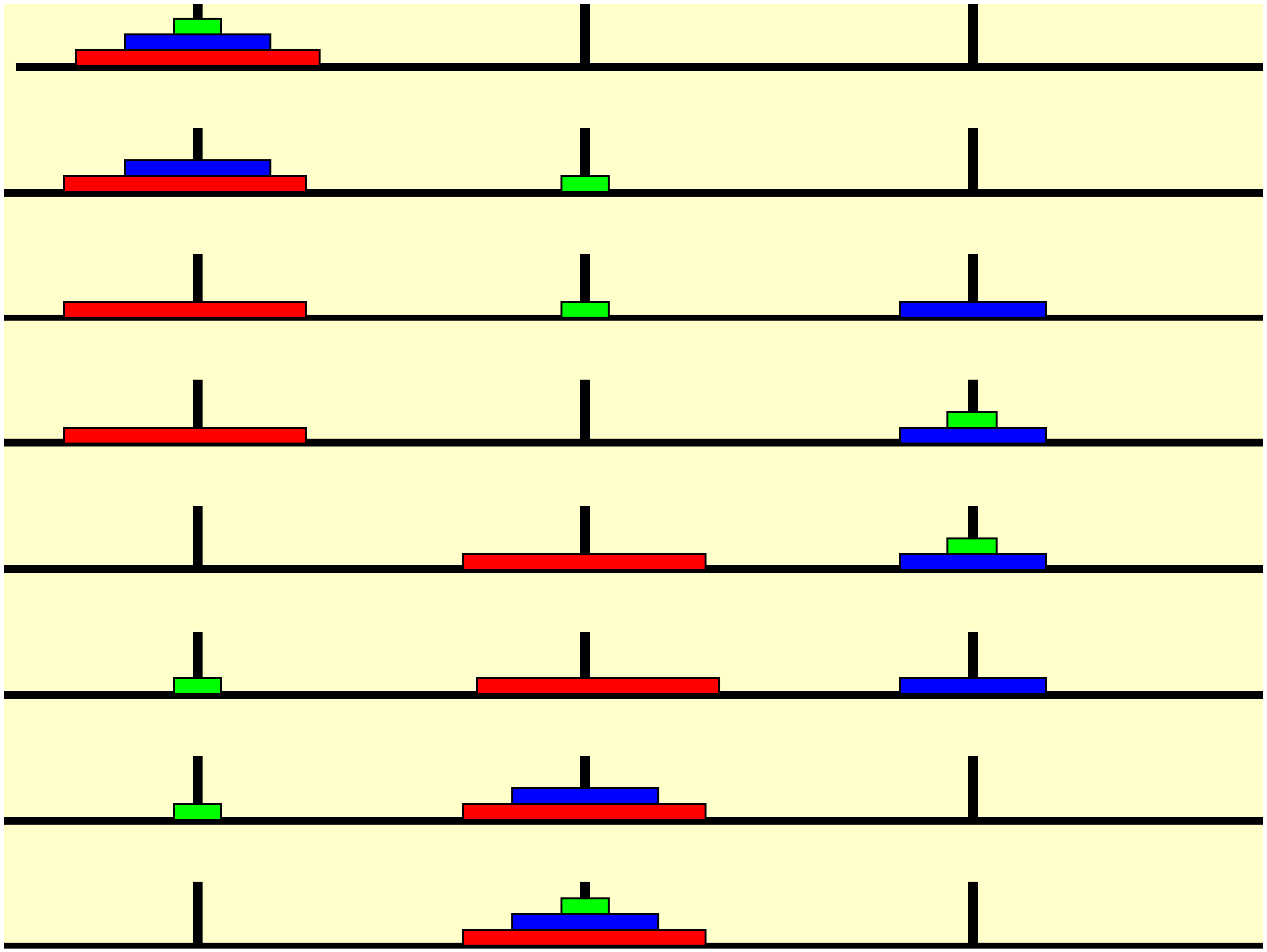
- Stacks and Queues
- Tree Traversal
- Towers of Hanoi

Rules of the Game



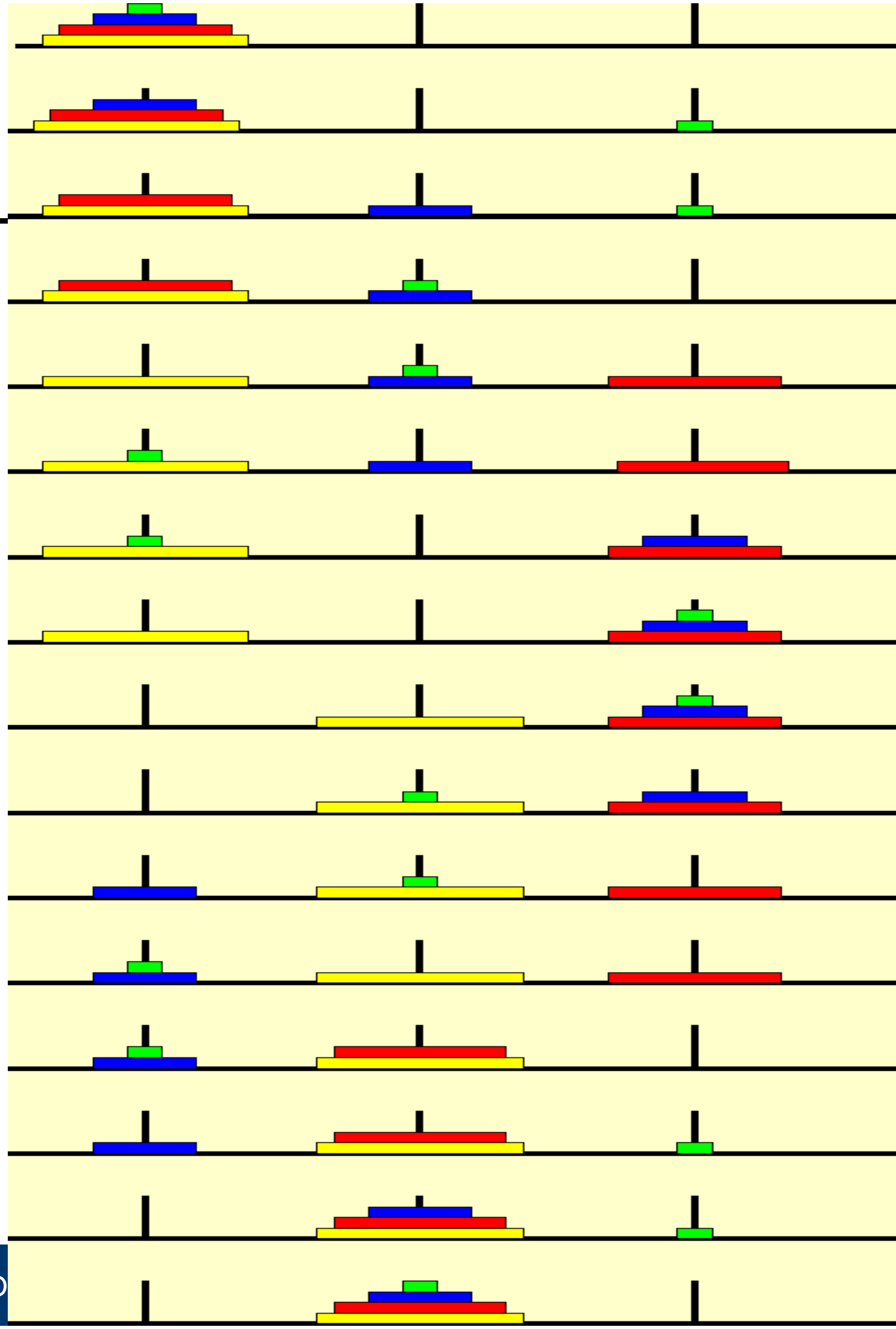
- Move stack from stick 1 to stick 2
- Always move only one disc at a time
- Never place a larger disc on a smaller one

Solution for 3 Discs



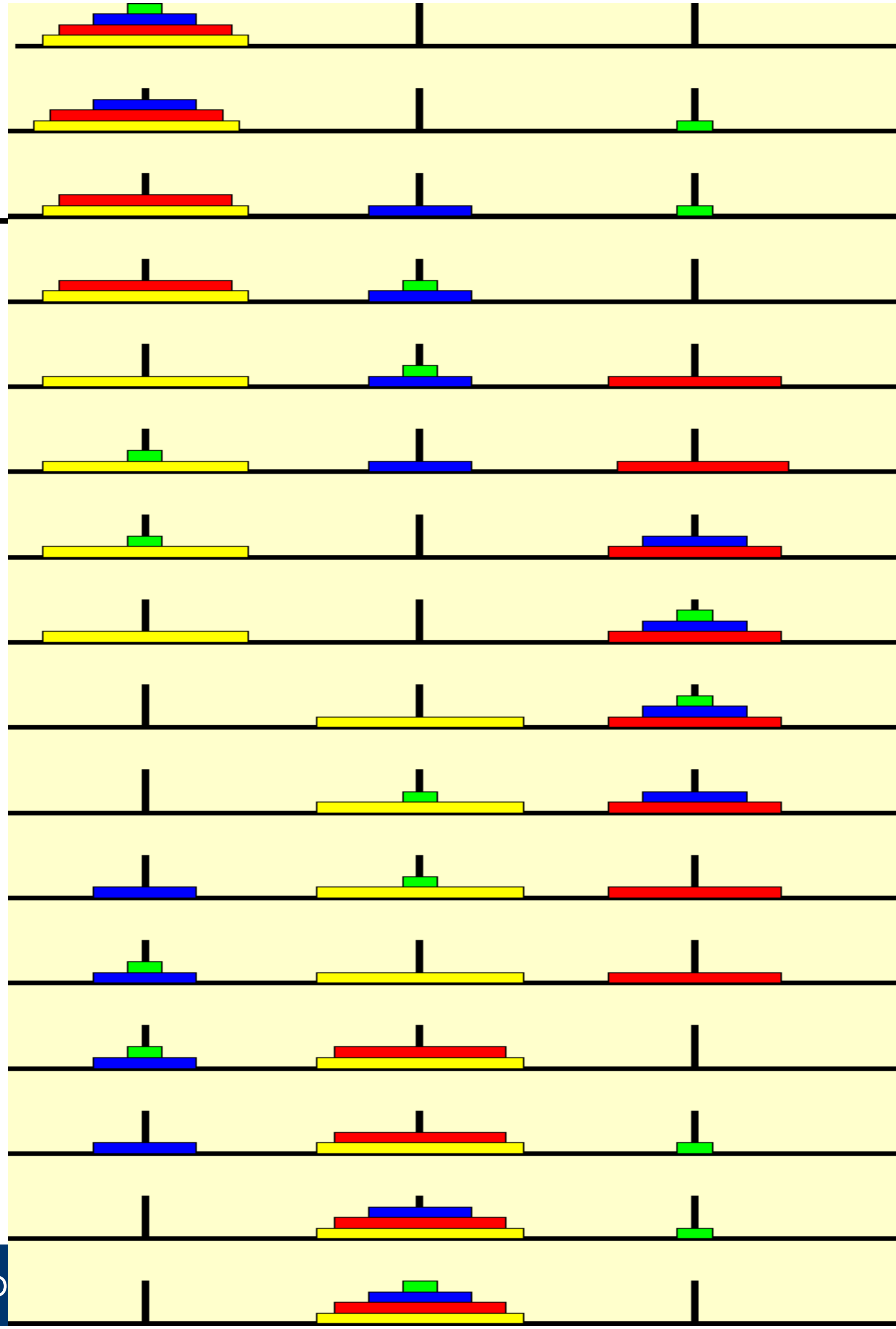
Source: Informatik Didaktik, U Potsdam

4 Discs



We have seen this part before

4 Discs

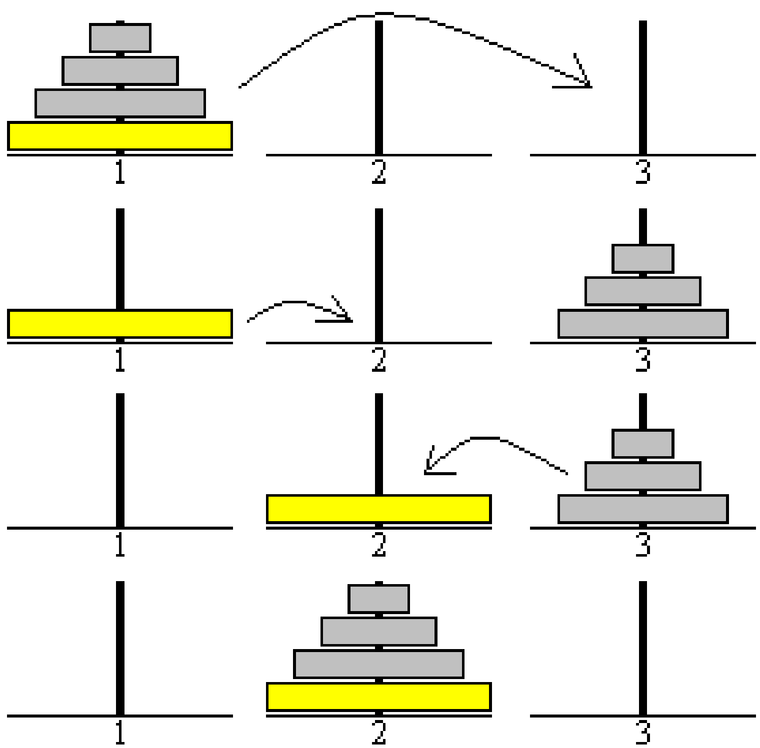


And this part as well

We have seen this part before

Idea

- The problem can be solved “easily” (with little program code) using the following observations
  – Suppose you know how to solve the problem for n-1 discs
  – Then solving it for n discs is simple
    - 1. Move the (n-1) top-part of the tower to stick 3
    - 2. Move the n’th (largest) disc to stick 2
    - 3. Move the (n-1) tower from stick 3 to stick 2
  – Furthermore, we know how to solve the problem for n=1
  – Done



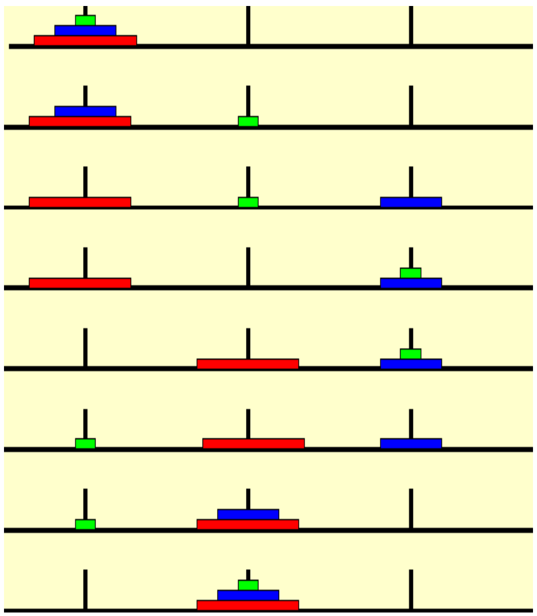
Algorithm

- We want an algorithm which prints the series of moves that solve the problem for size n
- We encode a move as a quadruple (n, a, b, c) which means: “Move n discs from stick a to b using c”
- We build a stack of tasks
- When we pop a task from the stack, we can do either
  - Task is easy (n=1): Print next move
  - Task is difficult (n>1): Push three new tasks

```
s: stack;
s.push( n, 1, 2, 3);
while not s.isEmpty() do
  (n, a, b, c) := s.pop();
  if (n=1) then
    print “Move “+a+”->“+b;
  else
    s.push( n-1, c, b, a);
    s.push( 1, a, b, c);
    s.push( n-1, a, c, b);
  end if;
end while;
```

Example

```
s: stack;
s.push( n, 1, 2, 3);
while not s.isEmpty() do
  (n, a, b, c) := s.pop();
  if (n=1) then
    print “Move “+a+”->“+b;
  else
    s.push( n-1, c, b, a);
    s.push( 1, a, b, c);
    s.push( n-1, a, c, b);
  end if;
end while;
```



3,1,2,3

2,1,3,2
1,1,2,3
2,3,2,1

1,1,2,3	Move 1->2
1,1,3,2	Move 1->3
1,2,3,1	Move 2->3
1,1,2,3	Move 1->2
2,3,2,1	

1,3,1,2	Move 3->1
1,3,2,1	Move 3->2
1,1,2,3	Move 1->2

Complexity

- How often do we pop from the stack?
  - For a task of size n, we pop once and create two tasks of size n-1 and one task of size 1
  - For a task of size 1, we pop once and create no further task
  - This gives $1+2+1+4+1+8+1+ \ldots +2^{n-1} = O(2^n)$ tasks altogether
    - Recall that $\sum 2^i = 2^{n+1}-1$
- The algorithm has complexity $O(2^n)$

Optimality

- We can also derive: For solving a problem of size n, the algorithm creates 2^n-1 moves
  - As every pop yields one move
- As no algorithm can create 2^n-1 moves in less than 2^n-1 operations, the algorithm is optimal for such sequences
- Question: Is there a shorter sequence of moves that also solves the problem?
  - Answer: No
- Second example of an exponential problem

Recursion

- Doesn‘t this fiddling around with a stack look overly complex?
- Recursive formulation

```
func void solve( n, a, b, c) {
  if (n=1) then
    print “Move “+a+”->“+c;
  else
    solve( n-1,a, c, b);
    solve( 1, a, b, c);
    solve( n-1, c, b, a);
  end if;
}
```

- This program will create more or the less the same stack - on the program stack
- A stack can be used to “de-recursify” a recursive algorithm
  – Which doesn’t mean that the program gets easier to understand