Content of this Lecture

- Stacks and Queues
- Tree Traversal
- Towers of Hanoi
Stacks and Queues

• Recall these two fundamental ADTs

```markdown
<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>type stack( T)</td>
<td>type queue( T)</td>
</tr>
<tr>
<td>import</td>
<td>import</td>
</tr>
<tr>
<td>bool;</td>
<td>bool;</td>
</tr>
<tr>
<td>operators</td>
<td>operators</td>
</tr>
<tr>
<td>isEmpty: stack → bool;</td>
<td>isEmpty: queue → bool;</td>
</tr>
<tr>
<td>push: stack x T → stack;</td>
<td>enqueue: queue x T → queue;</td>
</tr>
<tr>
<td>pop: stack → stack;</td>
<td>dequeue: queue → queue;</td>
</tr>
<tr>
<td>top: stack → T;</td>
<td>head: queue → T;</td>
</tr>
</tbody>
</table>
```

• 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

• 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
    • How?

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Linked list</th>
<th>Double-linked l.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>InsertAfter</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>DeleteThis</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Search</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Add to start</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Add to end</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Content of this Lecture

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

• **Information systems** are a class of software systems that is concerned with managing (and analyzing) data
  – Customers of a company, calls of a telecom company, stock management, human resources, enrolled students, etc.

• „Managing“ means
  – Storing
  – Being fail-safe
  – Allowing concurrent read and write access
  – Offering comfortable (and fast) ways of accessing the data
    • „Give me all customers older than 55 which purchased goods worth more than 30K Euro in the last 6 months and they did never before buy a Rolex“

• See course on Databases
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**
  - There is much more to say about XML; we only scratch the surface
  - There are things you cannot model easily in XML, but still
    • E.g.: students enroll_to courses

• **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

- 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
  - customer -> last_name, first_name, age
  - last_name -> *
  - first_name -> *
  - age -> *
Data – A Tree

• The data items of an XML database form a tree
Implementing a Tree

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

```
customers

<table>
<thead>
<tr>
<th>last_name</th>
<th>first_name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Müller</td>
<td>Peter</td>
<td>25</td>
</tr>
<tr>
<td>Meier</td>
<td>Stefanie</td>
<td>27</td>
</tr>
</tbody>
</table>
```
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 on all grand-children, then ... (always from left to right)
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.put( t.getRoot());
    while not s.isEmpty() do
        node := s.get();
        o := o+node.getValue() + "\lf";
        c := node.getChildren();
        if (c≠null) then
            foreach x in c do
                s.put( c);
            end for;
        end if;
    end while;
    return o;
}
DFS-2

Output o:

Stack s:
DFS-3

```
s.push( root);
while not s.isEmpty() do
    node := s.get();
    o := o+node.getValue();
    # print s, o;
    c := node.getChildren();
    if (c≠null) then
        foreach x in c do
            s.put( c);
        end for;
    end if;
    # print s, 0;
end while;
```
Adding Indentation

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

```
s.put( root);
s.put( 1);
while not s.isEmpty() do
  depth := s.get();
  node := s.get();
  o := o + SPACES(depth) + node.getValue();
  c := node.getChildren();
  if (c≠null) then
    foreach x in c do
      s.put( c);
      s.put( depth+1);
    end for;
  end if;
end while;
```
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 can traverse the 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

```java
Func String printBFS (t Tree) {
    s := new Queue();
    o := "";
    node : element;
    s.put( t.getRoot());
    while not s.isEmpty() do
        node := s.get();
        o := o+node.getValue();
        c := node.getChildren();
        if (c≠null) then
            foreach x in c do
                s.put( c);
            end for;
        end if;
    end while;
}
```

```
<table>
<thead>
<tr>
<th>last_name</th>
<th>first_name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Müller</td>
<td>Peter</td>
<td>25</td>
</tr>
<tr>
<td>Meier</td>
<td>Stefanie</td>
<td>27</td>
</tr>
</tbody>
</table>
```

```
customers
  └── customer
      ├── last_name
      │    └── Müller
      │        └── Peter
      │               25
      │           └── Meier
      │                └── Stefanie
      │                           27
      └── last_name
          └── age
              └── 27
```
• 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

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

```
printBFS:
while not s.isEmpty() do
  foreach x in c do
    ...
  end for;
end while;
```
Time Complexity

- In printBFS, we do not know how often the inner loop is passed-through for a specific iteration of the outer loop
  - We could not easily 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-through 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
- In DFS, the stack holds at most \( d \) 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
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

```java
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;
```
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
1,1,3,2
1,2,3,1
2,3,2,1

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

Move 1->2
Move 1->3
Move 2->3
Move 1->2
Move 3->1
Move 3->2
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+ ... +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**

```c
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-recursivy” a recursive algorithm
  – Which doesn’t mean that the program gets easier to understand