A tree represents a special kind of graph.

- a tree is a graph in which additional conditions must be given (similar to: every rectangle is a polygon).
- Through this, the structure and operations are simplified.

What are these conditions? (next slide)
**Definition 3.7:** The Graph $G = (V,E)$ is a **Tree** if and only if

1. $G$ is loop free
2. $G$ contains no simple edged circle
3. $G$ is associating

**Counter examples**

1. **loop**
2. **circle**
3. **Non-associating**
Analogy

leaves  
(End of the branching)

Branching  
(vertices)

twigs  
(edges)

root

Analogy

Tree as a graph

4. Data Structures

4.4 Trees
**Important:** According to later implementation in Java, as a result we must consider the edges of a tree as *directed edges*. Distinguishing the root vertex is not necessary any more.
Let $T = (V,E)$ be a tree.

- The successor of a vertex $v$ are also called **children**, or **sons** of $v$.

- The predecessor of vertex $v$ is also called **father** or **parent** of $v$.

- A vertex is the **root** of the tree when it does not have a father.

- A vertex that does not have any children is called a **leaf**.

- Vertices are called **siblings** when they have the same father.
Let $T = (V,E)$ be a tree with the root $v_0$.

- **Depth of a vertex** $v_n$
  = length of the path $\pi = (v_0, \ldots, v_n)$
  = $|\pi| = |(v_0, \ldots, v_n)|$
  = displacement from the root

- **Height of a tree**
  = maximal depth

- $T$ has the **order** $d$
  = the branching factor $d(T)$
  = Each vertex of $T$ has a maximum of $d$ children

Nomenclature:
- d-nary tree
  - i.e.: $d = 2$: binary tree
  - $d = 3$: ternary tree
In order to implement data structures based on graphs, one can use this object oriented approach.

- A vertex is an administration object with a controlled amount of child-vertices and a data object.
- An edge is created through reference to the respective child-vertices.
• The actual information is contained in the data object of the vertex
• The sum of all the data objects yields the amount of data that will be managed through the tree structure.
A binary tree represents a special case of a general tree.

- Binary = 2, in a pair, composed of 2 basic entities
- The structure of a binary tree and the operations associated with the tree are realized relatively easy on a digital computer through the tree characteristics.
Definition 3.8: The tree $T = (V,E)$ is a binary tree only if $d(T) = 2$

That means, each vertex from $T$ contains a maximum of 2 child-vertices.

examples
Binary trees are created in Java with the help of the \texttt{BNode} class.

- The \texttt{data} attribute references a data element of type \texttt{object}. (Similar to the data structure of a general tree)
- The linkage to the managing element results in the tree structure: \texttt{left} references the previous, \texttt{right} the next successor.
Binary trees: class diagrams

- **BTree**
  - root : BNode
  - isEmpty(): boolean
  - size(): int
  - contains(Object o): boolean
  - insert( Object o): void
  - remove( Object o): void

- **BNode**
  - left: BNode
  - right: BNode
  - data : Object
  - getData() : Object
  - setData( Object o)

- **Object**

2 recursive associations
Operations of a binary tree

Entire readout
- Traversing of all vertices in a certain order.

Tree organization
- Splitting of a tree into tree parts:
  \( \text{Tree[]} \) split()
- Assembly of multiple trees to one new tree:
  \( \text{Merge(Tree t)} \)

Data access
- insert: \( \text{add(Object o)} = \text{insert} \)
- delete: \( \text{remove(Object o)} = \text{delete} \)
- Search/ask: \( \text{boolean contains(Object o)} \)
Traversing process is the process that runs through each vertex of a tree-forming graph exactly once. In conjunction with binary trees, one could also be talking about a linearization.

Prevalent traversing strategies:

**Preorder:**

„WLİR“ – root, left part, right part

**Postorder:**

„LRW“ – left part, right part, root

**Inorder:**

„LWR“ – left part, root, right part
Traversing Binary trees - example

Postorder: "LRW" yields A, C, E, D, B, O, N, Z, R, P, L
Inorder: "LWR" yields A, B, C, D, E, L, N, O, P, R, Z
The value of the user data of all vertexes of the left part of the tree are smaller than the roots.

The value of the user data of all vertexes of the right part of the tree are bigger than the roots.
Binary Search Trees: Search (Pseudocode)

Search for the vertex with the data element object \( o \) in a tree.

\( a \): indicates a vertex (reference in Java)

**Initialization:**

\[ a = \text{root} \quad : \quad \text{Beginning at the root} \]

**Loop:**

as long as \( o \) isn’t found and \( a \) is not empty,

the following cases are then determined for the vertex \( a \):

\[
\begin{align*}
    a\text{.data} &= o & : & \quad o \text{ found!} \\
    a\text{.data} &> o & : & \quad \text{search in the left part of the tree} \quad \rightarrow a = a\text{.left} \\
    a\text{.data} &< o & : & \quad \text{search in the right part of the tree} \quad \rightarrow a = a\text{.right} \\
    a &= \text{zero} & : & \quad o \text{ not found in the tree!}
\end{align*}
\]
Binary Search Trees: Insert (Pseudocode)

Insert for the vertex with the data element object o in a tree.

a,v : indicate a vertex (reference in Java)

Initialization :
   a = root : Beginning at the root

Loop:
   As long as a is not empty
      v = a : a remember

Decide for the actual node a:
   a.data > o : Insert in the left subtree → a = a.left
   a.data < o : Insert in the right subtree → a = a.right
   a == null : see next slide

In v we remember the actual node (≠null), such that we can access this node, if we reach with a an illegal node. a is illegal (=null), when we reached the end of the search path. At that position, the node is inserted.
• **a == zero**: this means that we've found the position where the vertex with the same data value as the data element o should stand. However, we've also realized that such an element doesn't exist (a==zero), so therefore we must create a new one and insert it here. This new element to be inserted will contain the data element o.

• At insertion, we note the current vertex a with help from the indicator v, since it becomes the father vertex of the newly inserted vertex after the insertion operation. The indicator a is placed after insertion at the newly created vertex with the data object o.

• We need both indicators, a and v, so that we can position the corresponding indicator (left or right) of the father vertex v at the newly inserted child vertex.
Binary Search Trees: Inserting - Example (1)

Insert a vertex with the data element object `o` with the content „C“.

Due to viewing space, the vertex to be inserted will not be shown on the following slides.

**Initialization:**

`a = root`
Binary Search Trees: Inserting - Example (2)

Insert a vertex with the data element object \( o \) with the content „C“.

**Loop**

as long as \( a \) is not empty, \( v = a \)
decides for the current vertex \( a \):

- \( a.data > o \) : \( a = a.left \)
- \( a.data < o \) : \( a = a.right \)
- \( a == \text{zero} \) : \( o \) insert here

1. **Iteration**

\( v = a \)

\( a.data = \text{„L“} \)

\( o = \text{„C“} \)

\( \text{„L“} > \text{„C“} : \quad a' = a.left \)

---

4. Data Structures
4.4 Trees

---

Information Management in Engineering
Prof. Dr. Dr.-Ing. Jivka Ovtcharova – CSE-Lecture – Ch. 4 - WS 08/09 - Slide 23
Insert a vertex with the data element object o with the content „C“.

loop
as long as a is not empty, v = a
decides for the current vertex a:
  a.data > o : → a = a.left
  a.data < o : → a = a.right
  a == zero : o insert here

2. Iteration
v = a
a.data = „B“
o = „C“
„B“ < „C“: → a´ = a.right

Insert a vertex with the data element object o with the content „C“.
Insert a vertex with the data element object \( o \) with the content „C“.

3. Iteration

\[ v = a \]
\[ a.data = "D" \]
\[ o = "C" \]

„D“ > „C“:
\[ a' = a.left \]

Loop

as long as \( a \) is not empty, \( v = a \)
decides for the current vertex \( a \):
- \( a.data > o \) : \( a = a.left \)
- \( a.data < o \) : \( a = a.right \)
- \( a == zero \) : \( o \) insert here
Insert a vertex with the data element object \( o \) with the content „C“.

**Loop**

as long as \( a \) is not empty, \( v = a \)

*decides* for the current vertex \( a \):

\[
\begin{align*}
  \text{if } a.\text{data} > o & : \quad a = a.\text{left} \\
  \text{if } a.\text{data} < o & : \quad a = a.\text{right} \\
  \text{if } a == \text{zero} & : \quad o \text{ insert here}
\end{align*}
\]

**4. Iteration**

\( a == \text{zero} \) → \( o \) as the child of \( v \) inserted

\( v \) – pointer on the father of the new vertex, was already placed in iteration 3 at the last valid vertex of the search path (\( \neq \text{zero} \))
Delete the vertex with the data element object o with the content „C“.

1. Search vertex k with k.data = o.
   assuming k is the left son of the father, that means k.father.left == k

2. Case differentiation on the number of successors of k:
   a: k has no successors  (case a)
      → delete vertex k
   b: k has exactly 1 successor  (case b)
      → replace k with a son vertex
   c: k has exactly 2 successors  (case c)
      → replace vertex k with the largest vertex, g, of the left side of the tree

(similar to k.father.right == k)
Delete the vertex with the data element object `o` with the content „C“.

**case a:** vertex is a leaf, that means, it has **no successor**.

make the corresponding child vertex (here: `D.left`)
of the belonging father vertex `k.father` (here: `D`) **zero**.
Delete the vertex with the data element object \( o \) with the content „C“.
Delete the vertex with the data element object o with the content „N“.

**case b:** vertex k has exactly 1 successor

place the corresponding child vertex indicator (here: P.left) of the belonging father vertex k.father (here: P) on the non-empty successor of the vertex k to be deleted (here N.right = 0).
Delete the vertex with the data element object \( o \) with the content „N“.
Delete the vertex with the data element object o with the content „L“.

**case c:** vertex k has **exactly 2 successors**

replace the vertex to be deleted k (here: L) with the largest vertex from the left side of the tree (here: E, on the right side)

correct references (here: D.right = zero)
Delete the vertex with the data element object \( o \) with the content „L“.
Examples for trees

- **Scene graphs**
  - Representation of 3D-scenes
  - Starting from one node, the geometry of all object is represented in a tree structure
  - Geometries are subdivided into sub-geometries
  - The branches can be manipulated. Ex., positioning, orientation, …
  - Each geometry has its position in the scene, an orientation and a behavior
  - With a camera object that can be navigated, the entire scene can be made visible
Example XML

- Documents are stored in a tree structure
- Tags (like in HTML) can be defined
- A meta-language defines the formal aspects of the tags
- Prescribed: opening tag, closing tag, in between content, attributes.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<my_band>
  <Bandname>Nero's Delight</Bandname>
  <musicians>
    <musician>
      <name>Michael</name>
      <instrument>Trumpet</instrument>
      <lives_in>
        <Town>Bonn</Town>
      </lives_in>
    </musician>
    <musician>
      <name>Heidrun</name>
      <instrument>Trombone</instrument>
      <lives_in>
        <Town>Köln</Town>
      </lives_in>
    </musician>
  </musicians>
</my_band>
```

4. Data Structures
4.4 Trees
Literature


[FMUP05] Vorlesung „Formale Methoden und Programmierung“, 2005

[Schn98] H.-J. Schneider (Hrsg.), „Lexikon der Informatik“, Oldenbourg Verlag, 1998

[Wid02] Ottmann, Widmayer, „Algorithmen und Datenstrukturen“, Spektrum Akademischer Verlag, 2002


[Wald87] Waldschmidt, „Einführung in die Informatik für Ingenieure“, Oldenbourg Verlag, 1987
[Sen05] Sen, S.: „Bondgraphs – a formalism for modeling physical systems“, School of Computer Science, 2005


Outline

Lecture Content

1. Preface
2. Basics
3. Object orientation
4. Data Structures
   4.1. Introduction
   4.2. Graphs
   4.3. Use of Graphs
   4.4. Trees
   4.5. Use of Trees
   4.6. Linked Lists
   4.7. Queues and Stacks
5. Algorithms
The structuring of product data also concerns the organization of manufacturing. The product is thus separated into small chunks:

- Single parts
- Assembly groups
- Sub assembly groups
- Graphs
- Standard parts
- Documentation
In practice, application has established a hierarchical organization for manufacturing:

- **Product**
  - **Assembly group 1**
    - part 1
    - part 3
  - **Assembly group 2**
    - part 2
    - part 4
  - **Assembly group 3**
    - part 5
    - part 6
CATIA – Example for an Assembly Group

„Hydraulic Pump“

Group (CATProduct)

Single Parts (CATPart)

Constraints

4. Data structures
4.5 Use of trees
Pro/ENGINEER – Example for Choosing a Standard Part from a Parts Library

4. Data structures
4.5 Use of trees
The construction of a complex assembly part is created by functional and constructional coherences in a Constructive Solid Geometry (CSG) tree (constructive solid body geometry).

- By this way, the geometry of complex assembly parts can be generated from primitive bodies such as cubes, cylinders, prisms, spheres, or rings - that are linked with operations.
- Usually, the common boolean operations on sets are used for this purpose: union, difference and intersection, as the example shows.
The product structures in CATIA contain, next to the geometric data, also functional data and physical characteristics, such as volume, mass, surface, version (work-up edition), etc. These are dependent on one another and represent complex networked structures.
• For the illustration of such dependence in the product data structure, graphs are used.

• By means of graphs, the „desktop“ in CATIA shows the coherences between the graphical, functional, and physical characteristics of a product.
Use of Graphs in CATIA (2)

4. Data structures

4.5 Use of trees
A simple linked list represents a special case of a tree.

- The definition of a simple linked list is easily derived from the definition of a binary tree, since a simple linked list is just a limitation on the tree: each vertex has at the most exactly 1 successor.
Definition 3.10: the binary tree $T = (V,E)$ is a simple linked list when exactly $d(T) = 1$

That means, each vertex of $T$ has exactly one child vertex.

Characteristics:
- There is exactly one list element without a predecessor (first list element)
- There is exactly one list element without a successor (last list element)

Example:
Simple linked lists are achieved in Java through the following:

- Construction of the list from 2 types of **head elements**:
  - The vertex (**Node**) as a head element knows its successor and a data element.
  - a **List-Element** (**List**) knows the first vertex.

**List illustration:**

**Implementation:**
Simple Linked List as a Dynamic Data Structure (2)

example: simply linked, with additional indicators at the end (UML)

- classes:
  - **List** – list class, provides methods
  - **Node** – head vertex
  - **Object** – data object

<table>
<thead>
<tr>
<th>Class</th>
<th>Method/Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>+ head : Node</td>
</tr>
<tr>
<td></td>
<td>+ tail : Node</td>
</tr>
<tr>
<td></td>
<td>- currentNode : Node</td>
</tr>
<tr>
<td></td>
<td>+ isEmpty() : boolean</td>
</tr>
<tr>
<td></td>
<td>+ size() : int</td>
</tr>
<tr>
<td></td>
<td>+ contains(Object o) : boolean</td>
</tr>
<tr>
<td></td>
<td>+ get(int index) : Node</td>
</tr>
<tr>
<td></td>
<td>+ add(int index, Object o) : void</td>
</tr>
<tr>
<td></td>
<td>+ remove(Object o) : void</td>
</tr>
<tr>
<td></td>
<td>+ remove(Node n) : void</td>
</tr>
</tbody>
</table>

Recursive Association of the “Node” class
Operations for Simple Linked Lists

**insert**
- At the start: `void insertAtHead(Object o)`
- At position \( i \): `void insertAt(int i, Object o)`

**read**
- Testing of content: `boolean contains(Object o)`
- Read the first element: `Object getFirst()`
- Read at position \( i \): `Object get(int i)`

**delete**
- Delete the first element: `void removeAtHead()`
- Delete at position \( i \): `void removeAt(int i)`
Insert the vertex $v_k$ with the data element object $o$ in a non-empty simple linked list at position $i$.

$c$ : indicates the vertex

$j$ : Integer

**Initialization:**

$c = \text{head}, j = 0$ : start of the first element

**loop:**

as long as $j < i - 1$ : search of the insertion position

$j = j + 1$

$c = c.\text{next}$

**Insert operation:**

$k.\text{next} = c.\text{next}$ : insert the vertex $k$

$c.\text{next} = k$
Simple Linked Lists: Inserting - Example (1)

Insert the vertex $v_k$ at position $i$.

Initialization:

$c = \text{head}, j = 0$

![Diagram showing insertion of vertex $v_k$ at position $i$.]
Simple Linked Lists: Inserting - Example (2)

Insert the vertex $v_k$ at position $i$.

**loop:**
- as long as $j < i - 1$
  - $j = j + 1$
  - $c = c.next$

(head)

$j = index - 1$

(current state after running the loop)
Simple Linked Lists: Inserting - Example (3)

Insert the vertex \( v_k \) at position \( i \).

Insert operation:

\[
\begin{align*}
    k.\text{next} &= c.\text{next} \\
    c.\text{next} &= k
\end{align*}
\]

\[ j = \text{index} - 1 \]
Simple Linked Lists: Inserting - Example (4)

Insert the vertex \( v_k \) at position \( i \).

Insert operation:

\[
\begin{align*}
k & . \text{next} = c . \text{next} \\
c & . \text{next} = k
\end{align*}
\]

\( j = \text{index} - 1 \)
The insertion of the vertex $v_k$ in an empty simple linked list is a special case that must be dealt with.

Since the list is empty, the vertex $v_k$ can be inserted at no further expense:

```
head = k
k.next = zero
```
Delete the vertex $v_i$ with the data element object $o$ in a non-empty simple linked list at position $i$.

**c, k**: indicates a vertex (k – temporary indicator)

**j**: Integer

**Initialization:**

$c = \text{head}, j = 0$ : starts with the first element

**Loop:**

as long as $j < i - 1$ : search for the position to be deleted

$j = j + 1$

$c = c\.next$

**Delete operation:**

$k = c\.next$ : delete the vertex

$c\.next = k\.next$

delete $k$
Delete the vertex $v_i$ at position $i$.

Initialization:
$c = \text{head}, j = 0$
Delete the vertex $v_i$ at position $i$.

**loop:**
- as long as $j < i - 1$
  - $j = j + 1$
  - $c = c.next$

$j = index - 1$

*(current state after running the loop)*
Delete the vertex \( v_i \) at position \( i \).

Delete operation:

\[
\begin{align*}
    k &= c.\text{next} \\
    c.\text{next} &= k.\text{next} \\
    \text{delete } k
\end{align*}
\]

\[
    j = \text{index} - 1
\]
Delete the vertex $v_i$ at position $i$.

Delete operation:

1. $k = c.next$
2. $c.next = k.next$
3. delete $k$

$j = index - 1$
Delete the vertex $v_i$ at position $i$.

Delete operation:

$k = c.next$

$c.next = k.next$

`delete k`

$j = index - 1$
The **special case** of the element to be deleted $v_i$, that is the **first** in the list ($i = 0$), must be dealt with.

Since the element is the first in the list, the vertex $v_i$ can be removed from the list without further expense and later be deleted with:

```
head = head.next
```
Double linked lists

The simple linked list is supplemented with the following references:

• Each vertex contains a reference to his predecessor.
• tail references the end of the list.

advantages:

• Insertion operations in front and behind are fast to perform, since one must not run through the list.
• One can run through the list in both directions.