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“. 
Binary Search Trees: Deleting, case c (1)

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 objects 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
Scene graph

4. Data Structures
4.4 Trees

root

stateSwitch

coordinate₁ coordinate₂ coordinateₙ

component₁ component₂ componentₘ

appearance₁ indexedShape₁
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>
    <!-- and so on  -->
  </musicians>
</my_band>
```
<table>
<thead>
<tr>
<th>参考文献</th>
<th>作者和书名</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUP05</td>
<td>Vorlesung „Formale Methoden und Programmierung“, 2005</td>
</tr>
<tr>
<td>Schn98</td>
<td>H.-J. Schneider (Hrsg.), „Lexikon der Informatik“, Oldenbourg Verlag, 1998</td>
</tr>
<tr>
<td>Wid02</td>
<td>Ottmann, Widmayer, „Algorithmen und Datenstrukturen“, Spektrum Akademischer Verlag, 2002</td>
</tr>
<tr>
<td>WaWa86</td>
<td>Waldschmidt, Walter, „Grundzüge der Informatik“ – Band 1 &amp; 2, Spektrum Akademischer Verlag, 1998</td>
</tr>
<tr>
<td>Wald87</td>
<td>Waldschmidt, „Einführung in die Informatik für Ingenieure“, Oldenbourg Verlag, 1987</td>
</tr>
</tbody>
</table>
[Sen05] Sen, S.: „Bondgraphs – a formalism for modeling physical systems“, School of Computer Science, 2005


The structuring of product data also concerns the organization of manufacturing. The product is thus separated into small chunks:

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

```
<table>
<thead>
<tr>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly group 1</td>
</tr>
<tr>
<td>part 1</td>
</tr>
<tr>
<td>Assembly group 2</td>
</tr>
<tr>
<td>part 3</td>
</tr>
<tr>
<td>part 4</td>
</tr>
<tr>
<td>Assembly group 3</td>
</tr>
<tr>
<td>part 5</td>
</tr>
<tr>
<td>part 6</td>
</tr>
<tr>
<td>part 2</td>
</tr>
</tbody>
</table>
```
CATIA – Example for an Assembly Group

"Hydraulic Pump"

Group (CATProduct)

Single Parts (CATPart)

Constraints
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

Ein Objekt oder einen Befehl auswählen
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.
Pro/ENGINEER – Example for Choosing a Standard Part from a Parts Library

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

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

### Classes

**List**
- `head : Node`
- `tail : Node`
- `currentNode : Node`

**Node**
- `isEmpty() : boolean`
- `size() : int`
- `contains(Object o) : boolean`
- `get(int index) : Node`
- `add(int index, Object o) : void`
- `remove(Object o) : void`
- `remove(Node n) : void`

**Object**
- `getData() : Object`
- `setData(Object o)`

Recursive Association of the "Node" class
Operations for Simple Linked Lists

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

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

**delete**
- Delete the first element: \( \text{void removeAtHead()} \)
- Delete at position \( i \): \( \text{void removeAt(int i)} \)
Simple Linked Lists: Inserting (Pseudocode)

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

\[
\begin{align*}
  j & = j + 1 \\
  c & = c.\text{next}
\end{align*}
\]

**Insert operation:**

\[
\begin{align*}
  k.\text{next} & = c.\text{next} \quad : \text{insert the vertex } k \\
  c.\text{next} & = k
\end{align*}
\]
Insert the vertex $v_k$ at position $i$.

Initialization:

$c = \text{head}, j = 0$
Insert the vertex $v_k$ at position $i$.

**loop:**

as long as $j < i - 1$

\[
j = j + 1
\]

\[
c = c.next
\]

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

Insert operation:

- $k$.next = $c$.next
- $c$.next = $k$

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

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

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

\[
j = \text{index} - 1
\]
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:

$$
\text{head} = k \\
\text{k.next} = \text{zero}
$$

\[\text{Diagram: head} \quad \text{zero} \quad \text{head} \quad v_k \quad \text{zero} \]

\[\text{Diagram: k} \quad v_k \quad \text{head} \quad \text{zero} \]
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
\( j = j + 1 \)
\( c = c.\text{next} \)

**Delete operation:**
\( k = c.\text{next} \) : delete the vertex
\( c.\text{next} = k.\text{next} \)
delete \( k \)
Simple Linked List: Deleting - Example (1)

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 $j < i - 1$
  - $j = j + 1$
  - $c = c.next$

(central state after running the loop)

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

Delete operation:

$k = c.next$

c.next = k.next

delete k

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

Delete operation:

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

\[j = \text{index} - 1\]
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:

\[
\text{head} = \text{head}.\text{next}
\]
Double linked lists

The simple linked list is supplemented with the following references:

- Each vertex contains a reference to his predecessor.
- \textit{tail} references the end of the list.

\begin{itemize}
  \item Insertion operations in front and behind are fast to perform, since one must not run through the list.
  \item One can run through the list in both directions.
\end{itemize}
The characteristics of the double linked lists allow for the implementation of 2 important data structures.

- **Queues**: **FIFO** – „First in First out“
  Elements are extracted in the order in which they enter the queue.
  Everyday examples: Bank counter, supermarket, waiting room at the doctor

- **Stacks**: **LIFO** – „Last in First out“
  Elements are extracted in reverse order of how they enter the line.
  Everyday examples: plate sequence, overcrowded busses, recursive problems
Queues

Operations:
- The first element is „served“
- A new element is added behind

Application:
example: Administration of print orders or processes
Queues: Operations

Minimum Operations:
- **write**
  insert (add) elements at the end      \( \text{void enqueue(Object \ o)} \)
- **delete**
  remove an element from the beginning \( \text{void dequeue()} \)
- **Read and search**
  returns the beginning element, without removal \( \text{Object front()} \)

Optional Operations:
- **Number of elements** \( \text{int size()} \)
- **Queue empty?** \( \text{boolean empty()} \)
### Queues - Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Content of Queue</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Queue()</td>
<td>()</td>
<td>-</td>
</tr>
<tr>
<td>enqueue (1)</td>
<td>( 1 )</td>
<td>-</td>
</tr>
<tr>
<td>enqueue (3)</td>
<td>( 1, 3 )</td>
<td>-</td>
</tr>
<tr>
<td>front ()</td>
<td>( 1, 3 )</td>
<td>1</td>
</tr>
<tr>
<td>enqueue (7)</td>
<td>( 1, 3, 7 )</td>
<td>-</td>
</tr>
<tr>
<td>dequeue ()</td>
<td>( 3, 7 )</td>
<td>-</td>
</tr>
<tr>
<td>front ()</td>
<td>( 3, 7 )</td>
<td>3</td>
</tr>
<tr>
<td>dequeue ()</td>
<td>( 7 )</td>
<td>-</td>
</tr>
<tr>
<td>front ()</td>
<td>( 7 )</td>
<td>7</td>
</tr>
</tbody>
</table>
Stacks

4. Data structures
4.7 Lines and sequence

**Operations:**

- **Adding is only possible at one end (the back).** The oldest elements are at the front.
- **Removal at the same end**

**Applications:**

**Examples:**

- Calculations of multi-body systems and simulations (sequence of transformation matrices)
- Saving of local variables at function calls
Stacks: Operations

Minimum Operations:

- **write**
  inserts an element at the end
  
  ```java
  void push(Object o)
  ```

- **delete**
  removes an element from the end
  
  ```java
  void pop()
  ```

- **Read and search**
  returns the last element without deleting
  
  ```java
  Object peek()
  ```

Optional Operations:

- **Number of elements**
  
  ```java
  int size()
  ```

- **Stack empty?**
  
  ```java
  boolean empty()
  ```
### Stacks: Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Content of Stack</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Stack()</td>
<td>()</td>
<td>-</td>
</tr>
<tr>
<td>push (1)</td>
<td>(1)</td>
<td>-</td>
</tr>
<tr>
<td>push (3)</td>
<td>(1, 3)</td>
<td>-</td>
</tr>
<tr>
<td>peek ()</td>
<td>(1, 3)</td>
<td>3</td>
</tr>
<tr>
<td>push (7)</td>
<td>(1, 3, 7)</td>
<td>-</td>
</tr>
<tr>
<td>pop ()</td>
<td>(1, 3)</td>
<td>-</td>
</tr>
<tr>
<td>peek ()</td>
<td>(1, 3)</td>
<td>3</td>
</tr>
<tr>
<td>pop ()</td>
<td>(1)</td>
<td>-</td>
</tr>
<tr>
<td>peek ()</td>
<td>(1)</td>
<td>1</td>
</tr>
<tr>
<td>Reference</td>
<td>Title</td>
<td>Author(s)</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>[FMUP05]</td>
<td>Vorlesung „Formale Methoden und Programmierung“, 2005</td>
<td></td>
</tr>
<tr>
<td>[Wald87]</td>
<td>Waldschmidt, „Einführung in die Informatik für Ingenieure“, Oldenbourg Verlag, 1987</td>
<td>Waldschmidt</td>
</tr>
<tr>
<td>Reference</td>
<td>Author and Title</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Sen05</td>
<td>Sen, S.: „Bondgraphs – a formalism for modeling physical systems“, School of Computer Science, 2005</td>
<td></td>
</tr>
<tr>
<td>Math06</td>
<td>Mathews, J.: „Intelligent powertrain design“, Mississippi State University, 2006</td>
<td></td>
</tr>
</tbody>
</table>