Computer Science for Engineers

Lecture 9

Data structures – part 3

Prof. Dr. Dr.-Ing. Jivka Ovtcharova

Dipl. Wi.-Ing. Dan Gutu

15th of January 2010
Binary trees are created in Java with the help of the `BNode` class.

- The `data` attribute references a data element of type `object`. (Similar to the data structure of a general tree)
- The linkage to the managing element results in the tree structure: `left` references the previous, `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
Entire readout

• Traversing of all vertices in a certain order.

Tree organization

• Splitting of a tree into tree parts:
  
  ```java
  Tree[] split()
  ```

• Assembly of multiple trees to one new tree:
  
  ```java
  Merge(Tree t)
  ```

Data access

• insert: `add(Object o) = insert`

• delete: `remove(Object o) = delete`

• Search/ask: `boolean contains(Object o)`
Traversing Binary trees

**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:**

„**WLR“** – root, left part, right part

**Postorder:**

„**LRW“** – left part, right part, root

**Inorder:**

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

Preorder "\\text{WLR}“ yields L, B, A, D, C, E, P, N, O, R, Z

Postorder "\\text{LRW}“ yields A, C, E, D, B, O, N, Z, R, P, L

Inorder "\\text{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 = root`: 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`:
  - `a.data == o`: `o` found!
  - `a.data > o`: search in the left part of the tree → `a = a.left`
  - `a.data < o`: search in the right part of the tree → `a = a.right`
  - `a == zero`: `o` not found in the tree!
Insert for the vertex with the data element object \( o \) in a tree. (It is supposed that there is no node in the tree by which a data object with value \( o \) is referenced.)

**Initialization:**

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

**Loop:**

As long as \( a \) is not empty

\[ v = a \quad : \quad a \text{ remember} \]

Decide for the actual node \( a \):

- \( a.data > o \) : Insert in the left subtree \( \rightarrow a = a.left \)
- \( a.data < o \) : Insert in the right subtree \( \rightarrow a = a.right \)
- \( a == \text{null} \) : see next slide
Binary Search Trees: Inserting (dealing with the case a == 0)

- **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 does not 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.
Insert a vertex with the data element object `o` with the content „C“.

Initialization:

`a = root`

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

1. Iteration

v = a

a.data = „L“

o = „C“

„L“ > „C“ : → a´ = a.left

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“.  

**Loop**

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

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

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

2. **Iteration**

\[
\begin{align*}
v &= a \\
a.\text{data} &= „B“ \\
o &= „C“ \\
„B“ < „C“: \quad \rightarrow a^\prime &= a.\text{right}
\end{align*}
\]
**Binary Search Trees: Inserting - Example (4)**

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

**3. Iteration**

- `v = a`
- `a.data = „D“`
- `o = „C“`
- „D“ > „C“: → `a` = `a.left`

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

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

**4. Iteration**

\( a == \text{zero} \rightarrow o \) 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 (\( \neq \text{zero} \))

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
Delete the vertex with the data element object o with the content „C“. (It is supposed that a node referencing the respective data object o is contained in the tree).

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 procedure for k.father.right == k)