7SENG010W Data Structres & Algorithms

Week 5 Lecture

Trees

Overview of Week 5 Lecture: Trees

- *Tree Data Structures*
  - Definition & properties
  - Binary Trees & non-Binary Trees
- *Binary Search Trees* (BST)
  - Definition & properties
  - Creation & Insertion of a new Value
  - Deletion of a Value
- *Binary Tree Traversal*
  - *In-order*
  - *Pre-Order*
  - *Post-order*

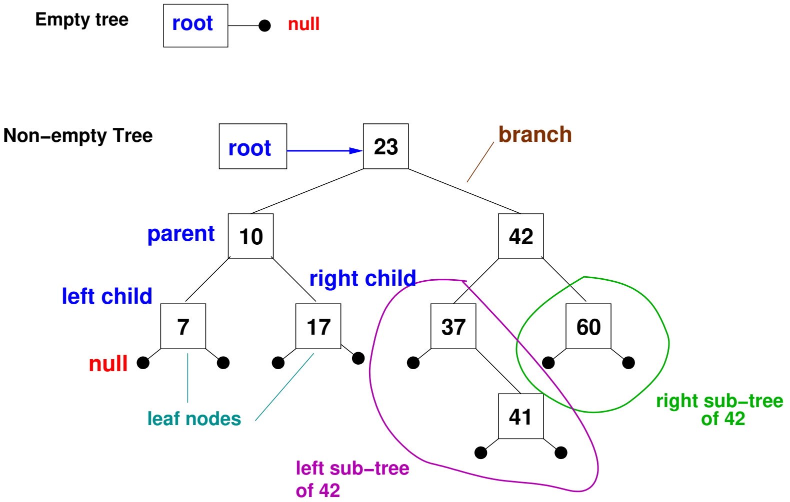
Acknowledgements: these notes are partially based on those of K. Draeger & P. Brennan.

PART I

Introduction to Tree Data Structures

Tree Data Structures (1/2)

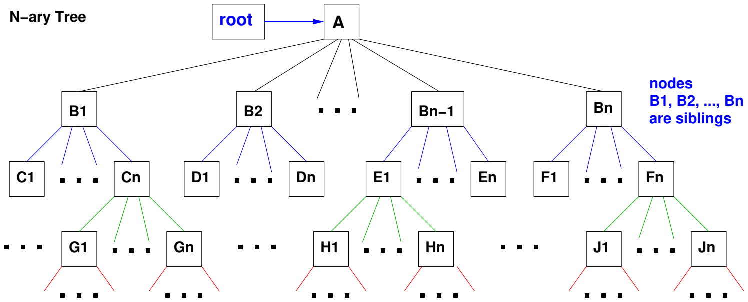
An *empty tree* & a *non-empty binary tree*.



Tree Data Structures (2/2)

A *non-empty n-ary tree*, i.e each node can have up to *n children* nodes, & hence up to n sub-trees.

All of the *child* nodes under the same *parent* node are referred to as *siblings*, e.g. the children of *A* the nodes *B1, B2, ..., Bn-1, Bn* are siblings. Similarly for the *C1, C2, ..., Cn-1, Cn*; *D1, D2, ..., Dn-1, Dn*; etc



Properties of Tree Data Structures (1/2)

Trees are *collection* data structures & are more complex that the *linear* data structures we have seen so far, e.g. lists.

Trees are very important in computing & software engineering as they are one of the most flexible & widely used data structures.

- *Non-Linear* data structures: not organised as a *sequence* of data items, but as a hierarchical "*tree*" structure.

  In other words they are not *linear* but *non-linear*.

- *Dynamic* data structures:
  - a "*tree*" structure of data items that *does not have a fixed sized*,
  - new data items can be *inserted* into a tree & existing data items in a tree can be *deleted* from it.

- *Linked* tree shaped structure of *data nodes*.

  Tree's data nodes are connected by "*branches*", i.e. links (references/pointers), to form a tree shape. (Similar to lists.)

Properties of Tree Data Structures (2/2)

- *Tree nodes:* each node has at most one "*parent*" node that links to it.
- Data nodes are *accessed* by following the *branches* (links) between the nodes.
- Data items (nodes) in a tree are *identified* or *found* by being in a "*significant*" or "*relative*" position within a tree.
- *Significant* tree position is the *unique* "*root*" node of the tree.

  The *root* node of the tree does not have a "*parent*" node, i.e. it is `null`.

  If the tree is "*empty*" then the root is equal to `null`.
- *Relative* tree positions are relative to the *current node*: its *parent* node, its "*left child*" & "*right child*" nodes.

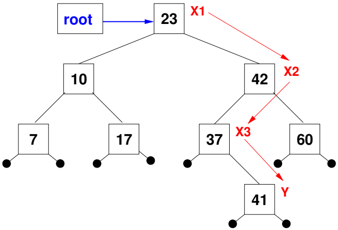
  In some types of trees its "*sibling*" nodes are also important.
- *Representations*: there are many different types of tree structures used, but the vast majority are represented by a tree structure of tree nodes.

  Some types of trees with restricted structures, e.g. a *heap*, can be implemented using arrays.

Traversing a tree

- This involves finding a *path* (sequence of branches) from the *root* to a particular node in the tree.
- Any node *X* on the **unique path** from a node *Y* to the root node is called an *ancestor* of *Y*, and *Y* is called a *descendent* of *X*.

  E.g. 41's *ancestors* are 23, 42 & 37; 23's *descendents* are 42, 37 & 41.
- If, in addition, *X* and *Y* are adjacent nodes, then *X* is said to be the *parent* of *Y*, and *Y* the *child* of *X*. E.g. 42 is the parent of 37 (& 60).
- In the tree below the *path* from the root 23 to the leaf 41 is via the branches connecting the nodes *23, 42, 37, 41*, i.e. **X1** $\rightarrow$ **X2** $\rightarrow$ **X3** $\rightarrow$ **Y**.
- So a *root* node has no parent & a *leaf* node has no children.

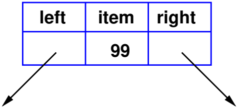


Tree Node Representations (1/2)

- Standard *Tree nodes* with a *data* item, *left child* & *right child* links have the following structure after creation & after insertion into a tree:

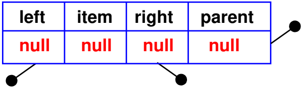
```
TreeNode  tNode  = new TreeNode();
```


```
TreeNode  tNode  = new TreeNode();
tNode .item = 99 ;
tNode .left = left_tree ;    tNode .right = right_tree;
```

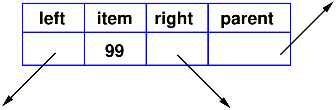


- *Tree nodes* with the added *parent* link have the following structure after creation & after insertion into a tree:

```
TreePNode  ptNode  = new TreePNode();
```

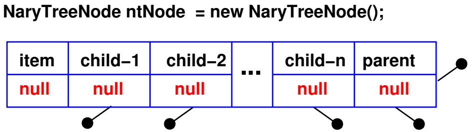


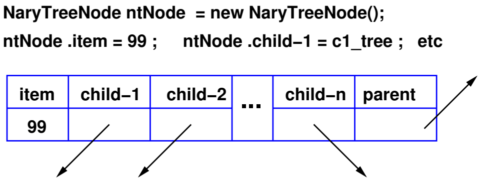

```
TreePNode  ptNode  = new TreePNode();
ptNode .item = 99 ;   etc
ptNode .parent = p_node ;
```



Tree Node Representations (2/2)

- *Non-binary* tree nodes with a *parent* link & *n-children* links have the following structure after creation & after insertion into a tree:





Note: for simplicity node's `parent`, `left child` & `right child`, etc, fields are not included in diagrams.

Binary Trees

- *Binary trees*, i.e. each node has at most 2 children nodes, are the most commonly used type of tree.
- For this reason we will focus on binary trees in this lecture & most of the next lecture on trees.
- In a *Binary tree* each item/node in the tree may have **at most two successors**, i.e. two sub-trees, the left & right sub-trees.
  See the tree diagrams given at the start of the lecture.
- **Definition: Binary Tree**
  A binary tree is either:

  Empty – it contains no nodes.

  Non-Empty – it contains at least one node, so it is either:

  - just one node, which is the *root* of the tree; *or*
  - the root & one disjoint binary sub-tree, either a left or right sub-tree; *or*
  - the root & two disjoint binary sub-trees, a left and right sub-tree.

Binary Tree Algorithms

- Usually tree algorithms use "*recursion*" starting from the root, as it most closely follows the "*recursive*" nature of the tree data structure.
  - Each recursive call checks that the *current* node is not `null`, then "*processes*" its data,
  - Then based on the node's data either the recursion ends or it is called again on one or both of its left & right sub-trees.
  - Terminating when either a specific node is found & processed, e.g. a search, deletion, etc, or if the *current* node is `null`.
- Some tree algorithms can also use `while`-loops to *traverse* a tree moving down it following either the left or right branches (links).
- The *meta data* is usually *references* (pointers) to the nodes linked to the *current* node: *leftChild*, *rightChild* & *parent*.
  Other data about the properties of a tree or a record of the path taken can also be used.
- The Big-O *worst-case* complexity, for most operations on a tree of *N* nodes, e.g. searching, insertion, deletion, is *Linear* – $O(N)$.
- The Big-Θ *average-case* complexity, for the above operations is *Logarithmic* – $\Theta(\log_2(N))$, & Big-Ω *best-case* is *Constant* – $\Omega(1)$.

PART II

Binary Search Trees (BST)

Binary Search Trees (BST)

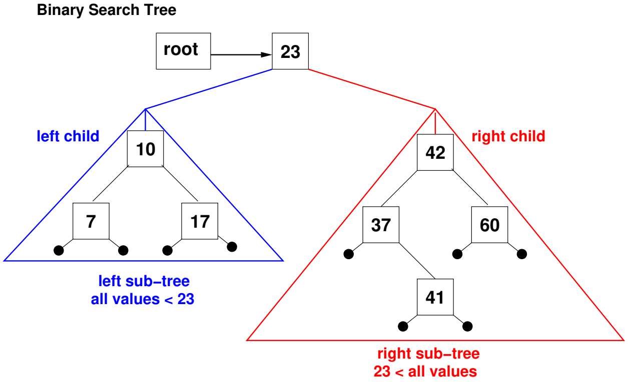
- A BST is a binary tree that represents a set of *keys* & their associated *values*[8].
- There is an additional constraint – the characteristic **BST property** that all **node keys** in the tree must satisfy:
  1. all *keys* in each node's *left sub-tree* are "*less than*" the key in the node;
  2. all *keys* in each node's *right sub-tree* are "*greater than*" the key in the node;
  3. then we **always get the keys in sorted order**.
- BST Operations: *insert* an item into the tree, *delete* an item from the tree, *search* for an item in the tree.
- The *insert* & *delete* operations **must** maintain the BST property.
- There are many different BSTs that represent the same set of keys.
- BSTs are useful when storing *ordered data* that is *dynamically changing*, & its insert & search operations are very *efficient*.

[8] For simplicity we usually treat the *key* & its *value* as the same entity, but in real examples this is usually not the case, **values** can be a large data structure.

BSTs

Example BST, where all the node values satisfy the *BST property*:

1. Root: $\{7, 10, 17\} \; < \; \mathbf{23} \; < \; \{37, 41, 42, 60\}$
2. *Left sub-tree*: $\{7\} \; < \; 10 \; < \; \{17\}, \quad \{\} \; < \; 7 \; < \; \{\}, \quad \{\} \; < \; 17 \; < \; \{\}$
3. *Right sub-tree*: $\{37, 41\} \; < \; 42 \; < \; \{60\}, \quad \{\} \; < \; 37 \; < \; \{41\},$
   $\{\} \; < \; 41 \; < \; \{\}, \quad \{\} \; < \; 60 \; < \; \{\}$



BST Operations

- The operations we shall focus on are:
  - *Searching* for a value in a BST,
  - *Inserting* a value into a BST,
  - *Deleting* a value from a BST.
- Remember that for any operation that modifies a BST it **MUST** ensure that the *BST property* is maintained, e.g. the *Insertion* & *Deletion* operations.
- Searching is also required for the *deletion* of an item.
- In PART III we shall look at general tree traversal.

BST Operation: Searching for a Value

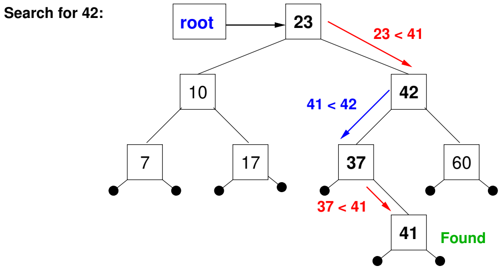
- *Searching* for a particular value in a BST, is relatively straightforward because all BSTs satisfy the *BST property*.
- So since the node values in a BST are in *ascending sorted order*, we can apply an adapted version of the *Binary Search* algorithm for an *ascending sorted array* we used in the lecture on Arrays.
- *Array Algorithm:* repeatedly comparing the value with the middle array element of a segment, then searching the left array segment if less than it, or searching the right array segment if greater than it.
- For the BST version of the algorithm we use a node's *value*, its *left child sub-tree* & its *right child sub-tree*, instead.
- So the BST *Binary Search* algorithm is:
  1. If the tree is empty then value not found & finish.
  2. Compare the value with the tree's *root* value.
  3. If equal then found value & finished.
  4. If *value is less than root value* then search in the *left sub-tree*.
  5. If *value is greater than root value* then search in the *right sub-tree*.

BST Search: Pseudo code

```
Search( TREE root, VALUE valueToFind )  // (NON-RECURSIVE Version)
BEGIN
   // define a node to use to traverse the BST, starting at the root
   currentNode <-- root

   WHILE ( currentNode is not an empty tree )
   BEGIN
        // search the tree, check root's value

        IF ( valueToFind == currentNode.value )
             RETURN currentNode                  // Found value
        ELSE

            IF( valueToFind < currentNode.value )
                // move to the left sub-tree &  search it
                currentNode <-- currentNode.leftChild
            ELSE
                IF ( currentNode.value < valueToFind )
                   //move to the right sub-tree &  search it
                   currentNode <-- currentNode.rightChild
                ENDIF
            ENDIF
        ENDIF
   ENDWHILE

   RETURN NOT_FOUND ;  // value Not found
END
```

Example: Search of BST

Search BST for *41* using algorithm's pseudo code:



1. set `currentNode` to the *root*,
2. `currentNode` is not an empty tree, i.e. `null`,
3. `valueToFind` (41) is greater than `currentNode.value` (23) so search `currentNode.`**`rightChild`** sub-tree.
4. `valueToFind` (41) is less than `currentNode.value` (42) so search `currentNode.`**`leftChild`** sub-tree.
5. `valueToFind` (41) is greater then `currentNode.value` (37) so search `currentNode.`**`rightChild`** sub-tree.
6. `valueToFind` (41) equals `currentNode.value` (41) so found.

BST Operation: Insert a Value

- Suppose we want to *insert new data in to a BST*.
- First decide whether *duplicate* values are allowed or only *unique* values are allowed in the BST.
- BSTs are often used to implement a container that implements a *set* of values, & sets do not have duplicate values.
- So we will adopt this approach & not have duplicate values in our BST.
- If duplicates were allowed then simply add an *occurrences counter* to each BST node, this would then not represent a *set* container, but a "*bag*" (or "*multi-set*") container.
- BST *Insert* algorithm (assuming no duplicates) is:
  1. If the tree is empty (root is `null`) then replace root with a new node containing the new value & finished.
  2. Otherwise, compare the new value with the tree's *root* value.
  3. If *new value is less than root value* then recursively insert it in the *left sub-tree*.
  4. If *new value greater than root value* then recursively insert it in the *right sub-tree*.

BST Insertion: Pseudo code (1/2)

The *insertion* operation is defined using two methods:

- `Insert( VALUE newValue )` – initially called to deal with a possibly empty BST & creating a new root node.

  If the BST is not empty then call the recursive method to insert the value into a non-empty BST.

- `InsertInTree( TREE root, VALUE newValue )` – called to insert the new node into a non-empty BST at the "*bottom*" of the BST, ensuring it maintains the *BST-property*.

  It uses recursion to traverse the BST from its root to the correct location to insert the new node as a *leaf* node.

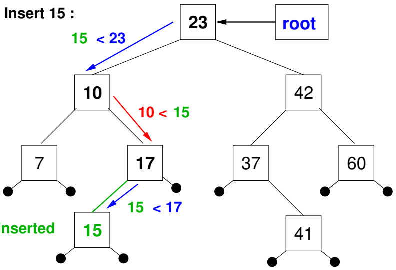
```
// Initially called Insert method
Insert( VALUE newValue )
BEGIN
     IF ( root is an empty tree )           // Check for an empty BST
            root <-- Node( newValue )
     ELSE
            InsertInTree( root, newValue )  // Insert into tree
     ENDIF
END
```

BST Insertion: Pseudo code (2/2)

```
// Use recursion for non-empty trees
InsertInTree( TREE root, VALUE newValue )
BEGIN
     IF ( newValue < root.value )
          // insert the new node in the left sub-tree

          IF ( root.leftChild is an empty tree )
               // insert the new node here
               root.leftChild  <-- Node( newValue )
          ELSE
               // recursively call this method on the leftChild
               InsertInTree( root.leftChild, newValue )
          ENDIF
     ELSE
          IF ( root.value < newValue )
              // insert the new node in the right sub-tree

              IF ( root.rightChild is an empty tree )
                   // insert the new node here
                   root.rightChild  <-- Node( newValue )
              ELSE
                   // recursively call this method on the rightChild
                   InsertInTree( root.rightChild, newValue )
               ENDIF
          ELSE
               SKIP  // newValue equals root.value(), already in BST
          ENDIF
     ENDIF
END
```

Example: Insertion into a BST

Insert **15** into our BST using the pseudo code algorithm:



1. `root` is not an empty tree, so call recursive method to do insertion.
2. `newValue` (**15**) is less than `root.value` (23) so insert into `root.`**`leftChild`** sub-tree (10).
3. `newValue` (**15**) is greater then `root.value` (10) so insert into `root.`**`rightChild`** sub-tree (17).
4. `newValue` (**15**) is less than `root.value` (17) & `root.`**`leftChild`** sub-tree is an empty tree (`null`) so insert (**15**) as new `root.`**`leftChild`** sub-tree.

BST Operation: Delete a Value

- *Deletion* is the most complex of the BST operations as there are 4 cases to consider.

- Start by *searching* the BST for the value to be deleted.

  There are 4 possible outcomes in relation to the “*location/type*” of node the delete value is in:

  1. **not** in the BST, or
  2. a *leaf* node, (i.e. both left & the right sub-trees are `null`), or
  3. a *non-leaf* node with *just 1 non-null sub-tree*, either the left or the right, or
  4. a *non-leaf* node with *both* the *left sub-tree* & the *right sub-tree* being *non-null*.

- Remember that if we alter the structure of a BST by deleting a value (node) then we have to make sure that **the resultant BST satisfies the BST-property**, i.e. maintains the *key* ordering.

  This sometimes means that we have to *restructure* the BST by moving parts of the tree around.

- We shall now consider the above 4 cases of deleting a node form a BST.

BST Deletion: Cases 1 - 3 Pseudo Code Steps

Steps for *deleting* a value – `deleteValue` from a BST are:

1. Find the `node` containing `deleteValue`:
   - If there is no `node` with `node.value` equal to `deleteValue`, then `deleteValue` is not in the BST, so finished.

2. If there is a `node` with `node.value` equal to `deleteValue` & the `node` is a *leaf node*, then delete it.

   Achieved by setting its parent node's leftChild or rightChild to `null` depending on which one the `node` was, & finished.

3. If the `node` is a *non-leaf node* with *just one non-`null` child* then:
   - *Restructure* the BST by: attaching the `node`'s child (left or right) to the `node`'s parent instead.
   - Delete the `node`.

BST Deletion: Case 4 Pseudo Code Steps

4. If the `node` containing `deleteValue` is a *non-leaf node* and *both* its leftChild & rightChild are non-`null` then:

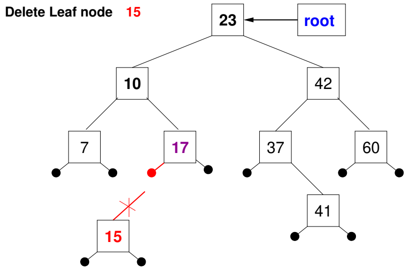
   - The BST has to be *restructured* to maintain the *BST property* by replacing `deleteValue` by the maximum value in the `node`'s left sub-tree & then deleting the maximum value's node.
   - Find the `maxLeftSubTreeNode` containing the `node`'s left sub-tree's *maximum value* by:
   - Starting from `node.leftChild` sub-tree, going down to the right as far as possible.
   - Use `maxLeftSubTreeNode`'s maximum value – `maxLeftSubTreeNode.value` to replace the value in `node.value`.
   - Attach `maxLeftSubTreeNode.leftChild` sub-tree (if non-`null`) to `maxLeftSubTreeNode`'s parent.

     Note there is **no rightChild sub-tree** because have traversed as far right as possible, i.e. it must be `null`.
   - Finally delete `maxLeftSubTreeNode`.

For the full details see the example code given in the tutorial exercises.

Example: Deletion of a BST Leaf Node

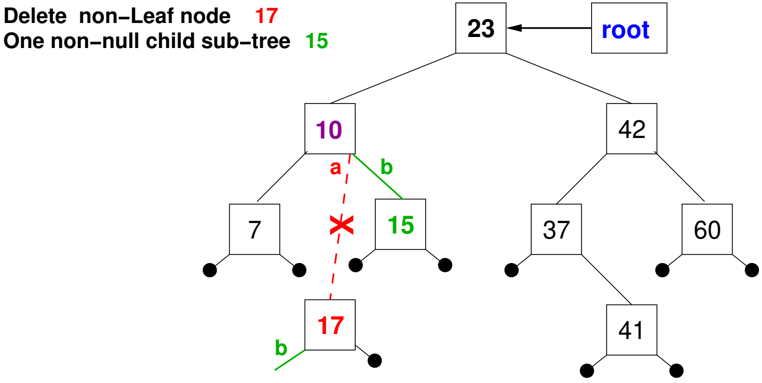
Deleting the *leaf node* **15** from our BST using the pseudo code algorithm:



1. `root` is not an empty tree, so search for the node containing the `deleteValue` **15**.
2. `deleteValueNode` (**15**) is the `leftChild` of its `parentNode` (**17**).
3. `deleteValueNode` (**15**) is a *leaf* node, i.e. both left & right children are `null`, so can be simply deleted.
4. Delete by setting `deleteValueNode`'s (**15**) `parentNode.leftChild` to `null`.

Example: Deletion of a BST Non-Leaf Node (1 Child)

Deleting the *non-leaf node* **17** that has just one non-`null` child **15** from our BST using the pseudo code algorithm:



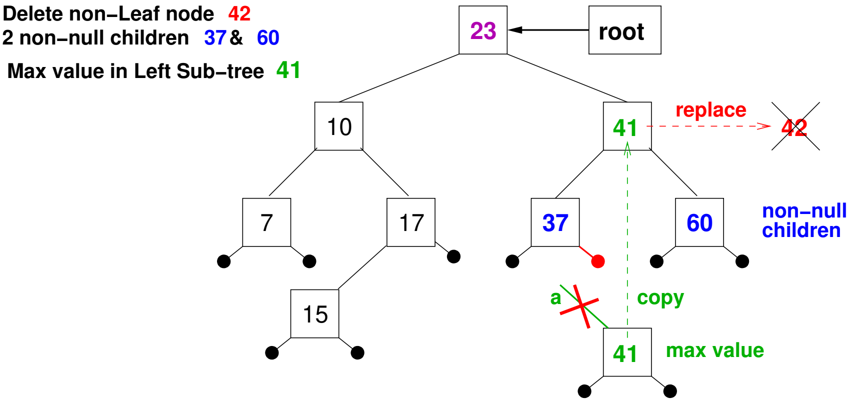
Example: Deletion Steps

1. `root` is not an empty tree, so search for the node containing the `deleteValue` **17**.
2. `deleteValueNode` (**17**) is the `rightChild` (**a**) of its `parentNode` (**10**).
3. `deleteValueNode` (**17**) is a *non-leaf* node as its left child is non-`null`, i.e. branch **b** to node **15**.
4. So must *restructure* the BST by:

   replacing **17**'s `parentNode` **10**'s `rightChild` with **17**'s `leftChild` **15**.
5. That is by setting `deleteValueNode.parentNode.rightChild` to `deleteValueNode.leftChild`, i.e. branch **b** replaces branch **a** in node **10**.
6. Delete node `deleteValueNode` **17**.

Example: Deletion of a BST Non-Leaf Node (2 Children)

Deleting the *non-leaf node* **42** that has 2 non-`null` child nodes *37* & *60* from our BST using the pseudo code algorithm:



Example: Deletion Steps

1. `root` (**23**) is not an empty tree, so search for the node containing the `deleteValue` (**42**).
2. `deleteValueNode` (**42**) is the `rightChild` of its `parentNode` (**23**).
3. `deleteValueNode` (**42**) is a *non-leaf* node, with both its left child (**37**) & right child (**60**) being non-`null`.
4. So to maintain the *BST-property*, must find the **maximum value** in **42**'s left sub-tree.
5. Starting from `deleteValueNode.leftChild` sub-tree (**37**), going down to the right as far as possible, which is `maxLeftSubTreeNode.value` (**41**).
6. Use `maxLeftSubTreeNode.value` (**41**) to replace the `deleteValueNode.value` (**42**), thus deleting it.
7. The `maxLeftSubTreeNode.leftChild` sub-tree is `null`, so nothing needs to be attached to `maxLeftSubTreeNode.parent` (**37**) `rightChild`. (If there was see previous Case steps 3 - 6.)
8. Finally delete `maxLeftSubTreeNode` (**41**), by setting (**37**) `rightChild` to `null`, i.e. replace branch **a** with `null`.

PART I

Tree Traversal

Tree Traversal

Tree traversal is the act of visiting all the nodes of a tree in a certain order to process the data contained in the tree.

For example, if you want to print all the numbers stored in a BST in *ascending* or *descending* order, or count them, etc.

There are 3 common methods for traversing a tree:

- *In-order*
- *Pre-order*
- *Post-order*

In-Order Tree Traversal

The **in-order** traversal method can be defined using the following pseudo code *recursive* algorithm:

```
InOrderTraverse( Node node ):
    IF ( node equals NULL)
    THEN
          return
    ELSE
          InOrderTraverse( node.leftSubTree )
          ProcessNode( node )
          InOrderTraverse( node.rightSubTree )
    END
```

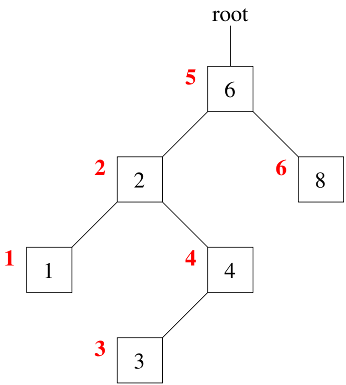
Visit the **node** (root of this tree) *in between* visiting the left & right sub-trees.

The **left sub-tree** is always visited first, then the **node**, then the **right sub-tree**.

NOTE: Traversing a standard **ordered binary tree** using the in-order algorithm means that all the nodes will be visited in **sorted order**, i.e. ascending key order.

So if “processing the node” was just printing and its values were just numbers then the output would be the numbers in the nodes in ascending order.

Traversing a Tree using In-Order



The numbers **1** to **6** indicate the order in which the nodes are processed.

In-order traversal: 1, 2, 3, 4, 6, 8.

Pre-Order Tree Traversal

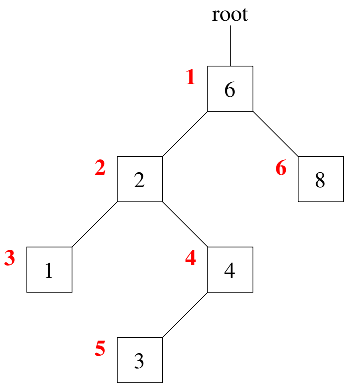
The **pre-order** traversal method can be defined using the following pseudo code *recursive* algorithm:

```
PreOrderTraverse( Node node ):
    IF ( node equals NULL)
    THEN
          return
    ELSE
          ProcessNode( node )
          PreOrderTraverse( node.leftSubTree )
          PreOrderTraverse( node.rightSubTree )
    END
```

So, in pre-order traversal:

- we visit the **node** *before* traversing the **left sub-tree**,
- and then finally traverse the **right sub-tree**.

Traversing a Tree using Pre-Order



The numbers **1** to **6** indicate the order in which the nodes are processed.

Pre-order traversal: 6, 2, 1, 4, 3, 8.

Post-Order Tree Traversal

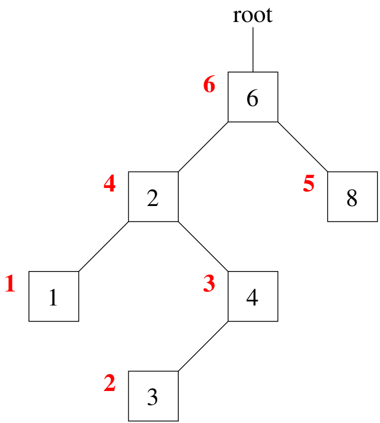
The **post-order** traversal method can be defined using the following pseudo code *recursive* algorithm:

```
PostOrderTraverse( Node node ):
    IF ( node equals NULL)
    THEN
          return
    ELSE
          PostOrderTraverse( node.leftSubTree )
          PostOrderTraverse( node.rightSubTree )
          ProcessNode( node )
    END
```

So, in post-order traversal:

- first traversing the **left sub-tree**,
- then traverse the **right sub-tree**.
- and then finally visit the **node**.

Traversing a Tree using Post-Order



The numbers **1** to **6** indicate the order in which the nodes are processed.

Post-order traversal: 1, 3, 4, 2, 8, 6.