DATA STRUCTURES

Binary Trees II. Traversals.

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In previous lecture

- Trees
  - Terminology
- Binary trees (I)
  - Terminology
  - Properties
  - Possible representations

Today

- Binary trees (II)
  - Traversals

Tree traversals

Traversing a tree means visiting all of its nodes.

A node is said to be *visited* when the program control arrives at the node, usually with the purpose of performing some operation on it.

For a binary tree there are 4 possible traversals:

- Preorder
- Inorder
- Postorder
- Level order (breadth first) - the same as in case of a (non-binary) tree

Binary tree - linked representation with dynamic allocation

Linked representation with dynamic allocation:

- There is one node for every element of the tree
- The structure representing a node contains:
  - the information
  - a pointer to the left child
  - a pointer to the right child
  - optionally, a pointer to the parent
- NIL denotes the absence of a node
  - $\Rightarrow$ the root of an empty tree is NIL

Binary tree representation

- In the following, we are going to use the dynamically allocated linked representation for a binary tree:

Representation of a node in a Binary Tree:

BTNode:

info: TElem

left: $\uparrow$ BTNode

right: $\uparrow$ BTNode

Representation of a Binary Tree:

BinaryTree:

root: $\uparrow$ BTNode

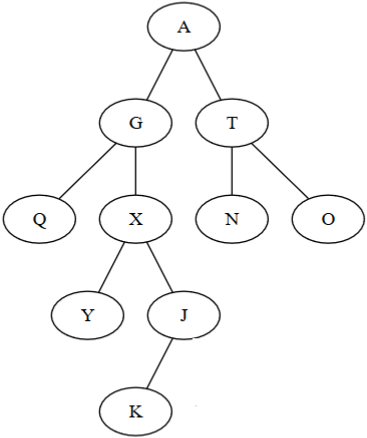
Preorder traversal

In case of a preorder traversal:

- Visit the *root* of the tree
- Traverse the left subtree - if exists
- Traverse the right subtree - if exists

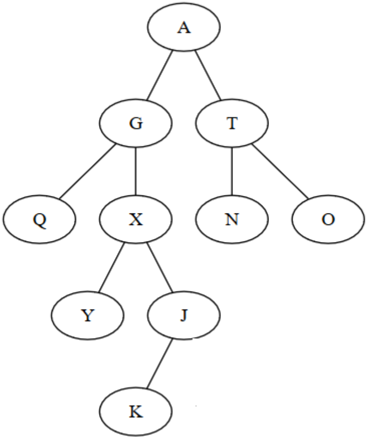
When traversing the subtrees (left or right) the same preorder traversal is applied.

Preorder traversal example



- Preorder traversal:

Preorder traversal example



- Preorder traversal: A, G, Q, X, Y, J, K, T, N, O

Preorder traversal - recursive implementation

- The simplest implementation for preorder traversal is with a recursive algorithm.

Recursive preorder traversal:

subalgorithm preorder_recursive(node) **is:**

//pre: node is a ↑ *BTNode*

Preorder traversal - recursive implementation

- The simplest implementation for preorder traversal is with a recursive algorithm.

Recursive preorder traversal:

subalgorithm preorder_recursive(node) **is:**
//pre: node is a $\uparrow$ *BTNode*
    if node $\neq$ NIL **then**
        @visit node
        preorder_recursive([node].left)
        preorder_recursive([node].right)
    end-if
end-subalgorithm

Preorder traversal - recursive implementation

We need a wrapper subalgorithm that receives a *BinaryTree* and calls the function for the root of the tree.

Recursive preorder traversal - call

subalgorithm preorderRec(tree) **is:**
//pre: tree is a BinaryTree
    preorder_recursive(tree.root)
end-subalgorithm

Preorder traversal - recursive implementation

We need a wrapper subalgorithm that receives a *BinaryTree* and calls the function for the root of the tree.

Recursive preorder traversal - call

subalgorithm preorderRec(tree) **is:**
//pre: tree is a BinaryTree
    preorder_recursive(tree.root)
end-subalgorithm

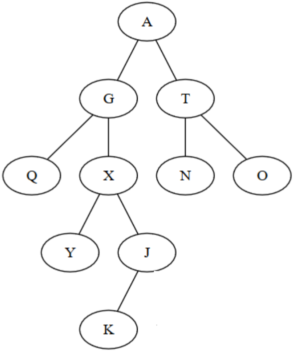
Assuming that visiting a node takes constant time, the traversal takes $\Theta(n)$ time for a tree with n nodes.

Preorder traversal - non-recursive implementation

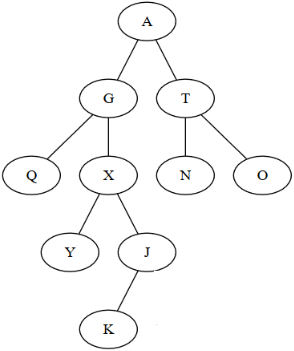
We can implement the preorder traversal algorithm without recursion, using an auxiliary *stack* to store the nodes.

- We start with an empty stack
- Push the root of the tree to the stack
- While the stack is not empty:
  - Pop a node and visit it
  - Push the node's right child to the stack
  - Push the node's left child to the stack

Preorder traversal - non-recursive implementation example



Preorder traversal - non-recursive implementation example



- Stack: A
- Visit A, push children (Stack: T G)
- Visit G, push children (Stack: T X Q)
- Visit Q, push nothing (Stack: T X)
- Visit X, push children (Stack: T J Y)
- Visit Y, push nothing (Stack: T J)
- Visit J, push child (Stack: T K)
- Visit K, push nothing (Stack: T)
- Visit T, push children (Stack: O N)
- Visit N, push nothing (Stack: O)
- Visit O, push nothing (Stack:)
- Stack is empty, traversal is complete

Preorder traversal - non-recursive implementation

Iterative preorder traversal:

subalgorithm preorder(tree) **is:**

//pre: tree is a binary tree

Preorder traversal - non-recursive implementation

Iterative preorder traversal:

subalgorithm preorder(tree) **is:**
//pre: tree is a binary tree
    init(s) *//s:Stack is an auxiliary stack*
    if tree.root $\neq$ NIL **then**
        push(s, tree.root)
    end-if
    while not isEmpty(s) **execute**
        currentNode $\leftarrow$ pop(s)
        @visit currentNode
        if [currentNode].right $\neq$ NIL **then**
            push(s, [currentNode].right)
        end-if
        if [currentNode].left $\neq$ NIL **then**
            push(s, [currentNode].left)
        end-if
    end-while
end-subalgorithm

Preorder traversal - non - recursive implementation

- The time complexity of the non-recursive traversal is:

Preorder traversal - non - recursive implementation

The time complexity of the non-recursive traversal is: $\Theta(n)$.

Extra-space complexity:

Preorder traversal - non - recursive implementation

The time complexity of the non-recursive traversal is: $\Theta(n)$.

Extra-space complexity: $O(n)$ (for the stack).

Obs: Preorder traversal is the same as *depth first traversal*.

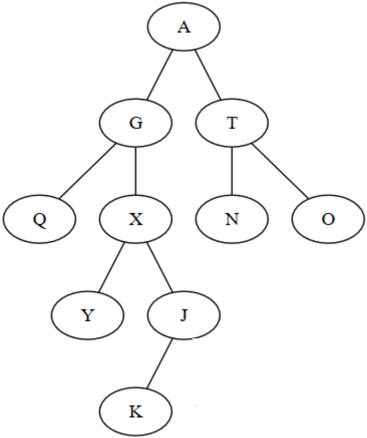
Inorder traversal

In case of *inorder* traversal:

- Traverse the left subtree - if exists
- Visit the *root* of the tree
- Traverse the right subtree - if exists

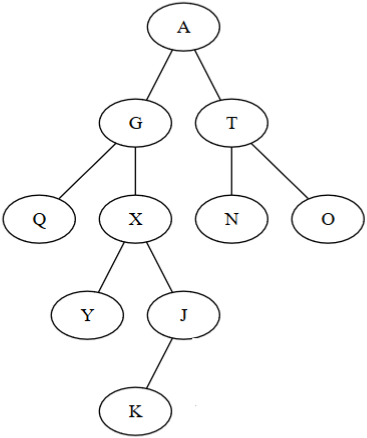
When traversing the subtrees (left or right) the same inorder traversal is applied.

Inorder traversal example



- Inorder traversal:

Inorder traversal example



- Inorder traversal: Q, G, Y, X, K, J, A, N, T, O

Inorder traversal - recursive implementation

- The simplest implementation for inorder traversal is with a recursive algorithm.

Recursive inorder traversal:

subalgorithm inorder_recursive(node) **is:**
//pre: node is a $\uparrow$ *BTNode*
    if node $\neq$ NIL **then**
        inorder_recursive([node].left)
        @visit node
        inorder_recursive([node].right)
    end-if
end-subalgorithm

We need again a wrapper subalgorithm to perform the first call to *inorder_recursive* with the root of the tree as parameter.

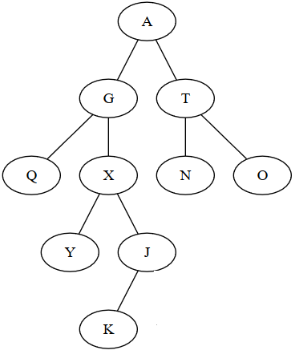
The traversal takes $\Theta(n)$ time for a tree with n nodes.

Inorder traversal - non-recursive implementation

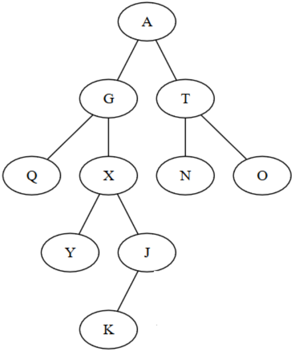
We can implement the inorder traversal algorithm without recursion, using an auxiliary stack to store the nodes.

- We start with an empty stack and an auxiliary current (pointer to) node (*currentNode*) is set to the root
- While *currentNode* is not NIL, push it to the stack and set it to its left child
- While stack not empty:
  - Pop a node and visit it
  - Set *currentNode* to the right child of the popped node
  - While *currentNode* is not NIL, push it to the stack and set it to its left child

Inorder traversal - non-recursive implementation example



Inorder traversal - non-recursive implementation example



- currentNode: A (Stack:)
- currentNode: NIL (Stack: A G Q)
- Visit Q, currentNode NIL (Stack: A G)
- Visit G, currentNode X (Stack: A)
- currentNode: NIL (Stack: A X Y)
- Visit Y, currentNode NIL (Stack: A X)
- Visit X, currentNode J (Stack: A)
- currentNode: NIL (Stack: A J K)
- Visit K, currentNode NIL (Stack: A J)
- Visit J, currentNode NIL (Stack: A)
- Visit A, currentNode T (Stack:)
- currentNode: NIL (Stack: T N)
- ...

Inorder traversal - non-recursive implementation

Iterative inorder traversal:

subalgorithm inorder(tree) **is:**

//pre: tree is a BinaryTree

Inorder traversal - non-recursive implementation

Iterative inorder traversal:

```
subalgorithm inorder(tree) is:
//pre: tree is a BinaryTree
    init(s) //s:Stack is an auxiliary stack
    currentNode ← tree.root
    while currentNode ≠ NIL execute
        push(s, currentNode)
        currentNode ← [currentNode].left
    end-while
    while not isEmpty(s) execute
        currentNode ← pop(s)
        @visit currentNode
        currentNode ← [currentNode].right
        while currentNode ≠ NIL execute
            push(s, currentNode)
            currentNode ← [currentNode].left
        end-while
    end-while
end-subalgorithm
```

Inorder traversal - non-recursive implementation

Time complexity:

Inorder traversal - non-recursive implementation

Time complexity: $\Theta(n)$

Extra space complexity:

Inorder traversal - non-recursive implementation

Time complexity: $\Theta(n)$

Extra space complexity: $O(n)$

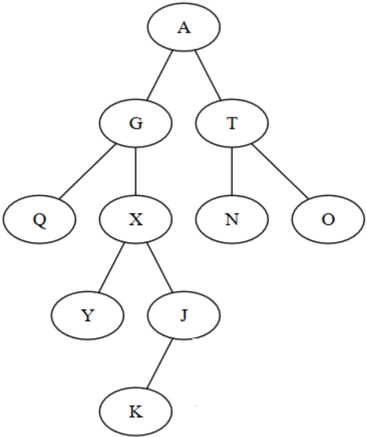
Postorder traversal

In case of *postorder* traversal:

- Traverse the left subtree - if exists
- Traverse the right subtree - if exists
- Visit the *root* of the tree

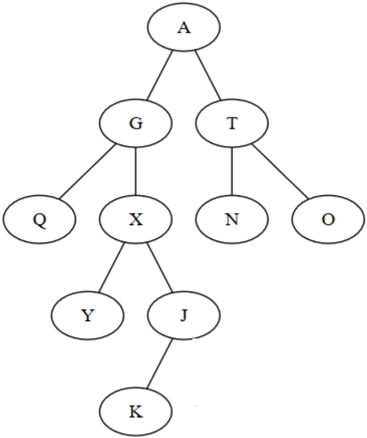
When traversing the subtrees (left or right) the same postorder traversal is applied.

Postorder traversal example



- Postorder traversal:

Postorder traversal example



- Postorder traversal: Q, Y, K, J, X, G, N, O, T, A

Postorder traversal - recursive implementation

- The simplest implementation for postorder traversal is with a recursive algorithm.

Recursive postorder traversal:

subalgorithm postorder_recursive(node) **is:**
//pre: node is a $\uparrow$ *BTNode*
    if node $\neq$ NIL **then**
        postorder_recursive([node].left)
        postorder_recursive([node].right)
        @visit node
    end-if
end-subalgorithm

We need again a wrapper subalgorithm to perform the first call to *postorder_recursive* with the root of the tree as parameter.

The traversal takes $\Theta(n)$ time for a tree with n nodes.

Postorder traversal - non-recursive implementation

We can implement the postorder traversal iteratively, but it is slightly more complicated than preorder and inorder traversals.

Postorder traversal - non-recursive traversal with one stack

- We start with an empty stack and a current node (*currentNode*) set to the root of the tree
- While *currentNode* is not NIL, push to the stack its right child, the *currentNode* and then set *currentNode* to its left child.
- While the stack is not empty:
  - Pop a node from the stack (*currentNode*)
  - If it has a right child, the stack is not empty and contains the right child on top of it, then pop the right child, push *currentNode* and set *currentNode* to its right child.
  - Otherwise, visit *currentNode* and set it to NIL
  - While *currentNode* is not NIL, push to the stack its right child, the *currentNode* and then set *currentNode* to its left child.

Postorder traversal - non-recursive implementation

Iterative postorder traversal:

subalgorithm postorder(tree) **is:**
//pre: tree is a BinaryTree

Postorder traversal - non-recursive implementation

Iterative postorder traversal:

```
subalgorithm postorder(tree) is:
//pre: tree is a BinaryTree
    init(s) //s: Stack is an auxiliary stack
    currentNode ← tree.root
    while currentNode ≠ NIL execute
        if [currentNode].right ≠ NIL then
            push(s, [currentNode].right)
        end-if
        push(s, currentNode)
        currentNode ← [currentNode].left
    end-while
    while not isEmpty(s) execute
        currentNode ← pop(s)
        if [currentNode].right ≠ NIL and (not isEmpty(s)) and [currentNode].right =
top(s) then
            pop(s)
            push(s, currentNode)
            currentNode ← [currentNode].right
//continued on the next slide
```

Postorder traversal - non-recursive implementation

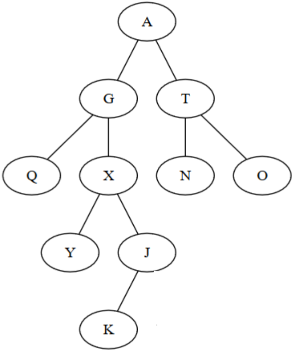
Iterative postorder traversal:

```
        else
            @visit currentNode
            currentNode ← NIL
        end-if
        while currentNode ≠ NIL execute
            if [currentNode].right ≠ NIL then
                push(s, [currentNode].right)
            end-if
            push(s, currentNode)
            currentNode ← [currentNode].left
        end-while
    end-while end-subalgorithm
```

Time complexity: $\Theta(n)$

Extra space complexity: $O(n)$

Postorder traversal - non-recursive implementation example

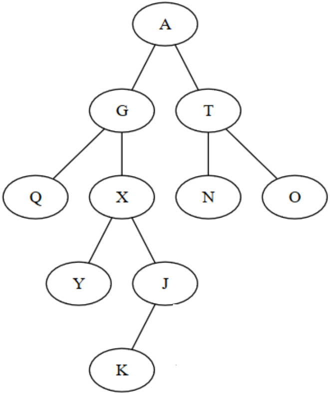


- currentNode: A (Stack:)
- currentNode: NIL (Stack: T A X G Q)
- Visit Q, currentNode NIL (Stack: T A X G)
- currentNode: X (Stack: T A G)
- currentNode: NIL (Stack: T A G J X Y)
- Visit Y, currentNode: NIL (Stack: T A G J X)
- currentNode: J (Stack: T A G X)
- currentNode: NIL (Stack: T A G X J K)
- Visit K, currentNode: NIL (Stack: T A G X J)
- Visit J, currentNode: NIL (Stack: T A G X)
- Visit X, currentNode: NIL (Stack: T A G)
- Visit G, currentNode: NIL (Stack: T A)
- currentNode: T (Stack: A)
- currentNode: NIL (Stack: A O T N)

...

Level order traversal

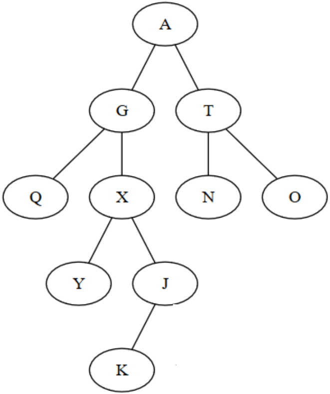
In case of level order traversal, we first visit the root, then the children of the root, then the children of the children, etc.



Level order traversal:

Level order traversal

In case of level order traversal, we first visit the root, then the children of the root, then the children of the children, etc.



Level order traversal: A, G, T, Q, X, N, O, Y, J, K

Preorder traversal - non-recursive implementation

Iterative level order traversal:

subalgorithm levelOrder(tree) **is:**
//pre: tree is a binary tree

Preorder traversal - non-recursive implementation

Iterative level order traversal:

```
subalgorithm levelOrder(tree) is:
//pre: tree is a binary tree
    init(q) //q:Queue is an auxiliary queue
    if tree.root ≠ NIL then
        push(q, tree.root)
    end-if
    while not isEmpty(q) execute
        currentNode ← pop(q)
        @visit currentNode
        if [currentNode].left ≠ NIL then
            push(q, [currentNode].left)
        end-if
        if [currentNode].right ≠ NIL then
            push(q, [currentNode].right)
        end-if
    end-while
end-subalgorithm
```

Binary tree iterator

The interface of the binary tree contains the *iterator* operation, which should return an iterator.

- This operation receives a parameter that specifies what kind of traversal we want to do with the iterator (*preorder*, *inorder*, *postorder*, *level order*).

The traversal algorithms discussed so far traverse all the elements of the binary tree at once, but an iterator has to do element-by-element traversal.

For defining an iterator, we have to divide the code into the functions of an iterator: *init*, *getCurrent*, *next*, *valid*.

Inorder binary tree iterator

Representation of an inorder iterator:

<u>InorderIterator:</u>

bt: BinaryTree

s: Stack

currentNode: ↑BTNode

Inorder binary tree iterator - init

The constructor of an inorder iterator over a binary tree:

subalgorithm init (it, bt) **is:**
//pre: it - is an InorderIterator, bt is a BinaryTree
    it.bt $\leftarrow$ bt
    init(it.s)
    node $\leftarrow$ bt.root
    while node $\neq$ NIL **execute**
        push(it.s, node)
        node $\leftarrow$ [node].left
    end-while
    if not isEmpty(it.s) **then**
        it.currentNode $\leftarrow$ top(it.s)
    else
        it.currentNode $\leftarrow$ NIL
    end-if
end-subalgorithm

Inorder binary tree iterator - getCurrent

The function for getting the current element of an inorder iterator over a binary tree:

function getCurrent(it) **is:**
    if not valid(sit) **then**
        @throw an exception
    end-if
    getCurrent ← [it.currentNode].info
end-function

Inorder binary tree iterator - valid

The function for checking the validity of an inorder iterator over a binary tree:

function valid(it) **is:**
    if it.currentNode = NIL **then**
        valid ← false
    else
        valid ← true
    end-if
end-function

Inorder binary tree iterator - next

The operation for advancing to the next element of an inorder iterator over a binary tree:

```
subalgorithm next(it) is:
    node ← pop(it.s)
    if [node].right ≠ NIL then
        node ← [node].right
        while node ≠ NIL execute
            push(it.s, node)
            node ← [node].left
        end-while
    end-if
    if not isEmpty(it.s) then
        it.currentNode ← top(it.s)
    else
        it.currentNode ← NIL
    end-if
end-subalgorithm
```

Preorder, Inorder, Postorder

How to remember the difference between traversals?

- Left subtree is always traversed before the right subtree.
- The visiting of the root is what changes:
  - **PRE**order - visit the root **before** the left and right
  - **IN**order - visit the root **between** the left and right
  - **POST**order - visit the root **after** the left and right

Binary Trees- Applications

Real-word applications of binary tree data structure:

Machine Learning

- Representing Decision Trees

Working with Morse Code

- The organization of Morse code is done in the form of a binary tree

Binary Expression Trees

- For evaluating arithmetic expressions: operators are stored in the internal nodes, while the operands are stored in the leaves

Routing tables

- A routing table is used to link routers in a network. It is usually implemented with a variation of a binary tree.

Databases

- Indexing (using B-trees)

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you