

UE5 Fundamentals of Algorithms

Lecture 9: Binary trees traversals

Ecole Centrale de Lyon, Bachelor of Science in Data Science for Responsible Business

Romain Vuillemot



```
In [24]: import sys
import os
from graphviz import Digraph
from IPython.display import display
from utils import draw_binary_tree
```

Outline

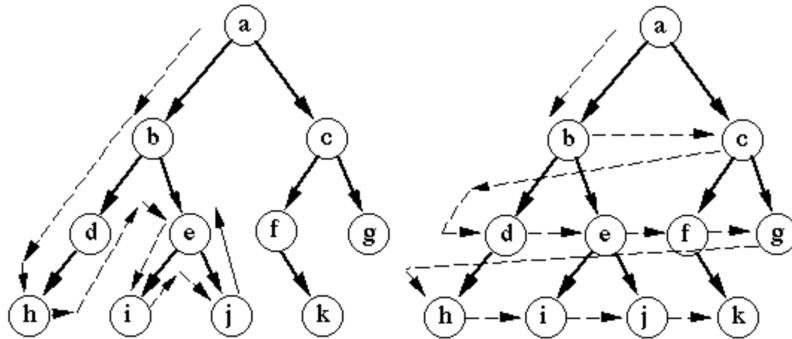
- Traversal methods
- Depth first
- Breadth first

Binary trees traversal methods

Methods to explore and process nodes in a tree (or a graph).

- Because Trees are non-linear, there are multiple possible paths
- Can be applied to the whole tree or until a certain condition is met
- Traversals methods will provide very different results

Two main traversal strategies:



1. Depth-First search (DFS):

- visiting a node (starting with the root)
- then recursively traversing as deep as possible
- then explore another branch.

2. Breadth-First search (BFS):

- visiting a node (starting with the root)
- explore all its neighbors (children)
- then move to the children.

Depth-first search (or traversal)

***Depth-first search (DFS)** is a traversal method that visits all the leaves first in a tree (or a graph).*

1. Place the source node in a **stack**.
2. Remove the node from the top of the stack for processing.
3. Add all unexplored neighbors to the stack (at the top).
4. If the stack is not empty, go back to step 2.

Depth-first search (or traversal)

```
In [25]: def dfs(tree, start):  
         stack = [start]  
         while stack:  
             vertex = stack.pop()  
             print(vertex, end = ' ') # traitement  
             stack.extend(tree[vertex])
```

```
In [26]: tree = {'A': set(['B', 'C']),  
                 'B': set(['D', 'E', 'F']),  
                 'C': set([]),  
                 'D': set([]),  
                 'E': set([]),  
                 'F': set([])  
                 }
```

```
In [27]: dfs(tree, 'A') # A B D F E C
```

A B F E D C

Depth-first search: pre-order, in-order, and post-order.

For **depth-first search**, there are different types of processing: *pre-order*, *in-order*, and *post-order*, based on when the processing is done (before/after exploring the root or the children). Notation :

- R = Root
- D = Right subtree
- G = Left subtree

There are three (main) types of traversal:

- **Pre-order:** R G D
- **In-order:** G R D
- **Post-order:** G D R

Depth-first traversal: pre-order, in-order, and post-order.

Implementation of the strategies:

```
def preorder(R):  
    if not empty(R):  
        process(R)           # Root  
        preorder(left(R))    # Left  
        preorder(right(R))   # Right  
  
def inorder(R):  
    if not empty(R):  
        inorder(left(R))     # Left  
        process(R)           # Root  
        inorder(right(R))    # Right  
  
def postorder(R):  
    if not empty(R):  
        postorder(left(R))   # Left  
        postorder(right(R))  # Right  
        postorder(R)         # Root
```

Example

We will use this data structure

```
In [28]: class Node:
          def __init__(self, value):
              self.value = value
              self.left = None
              self.right = None

          def get_value(self):
              return self.value

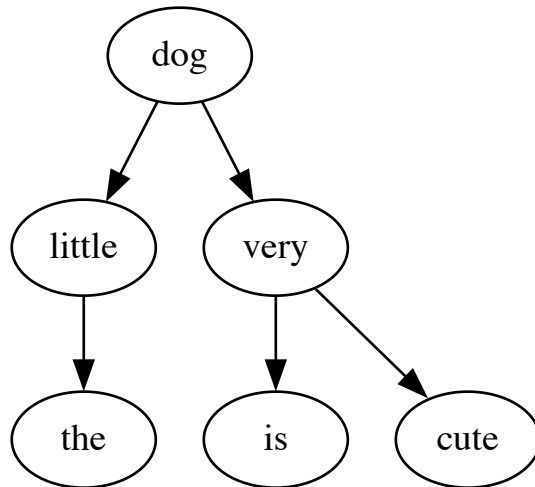
          def set_value(self, v = None):
              self.value = v
```

```
In [29]: root = Node("dog")
          root.left = Node("little")
          root.left.left = Node("the")
          root.right = Node("very")
          root.right.left = Node("is")
          root.right.right = Node("cute")
```

Example

How to get the sentence in the correct order?

```
In [30]: draw_binary_tree(root)
```



Depth-first traversal pre-order (OOP + iterative)

```
In [31]: def iterative_inorder_traversal(node):  
         stack = [node]  
         while stack:  
             current_node = stack.pop()  
             print(current_node.value)  
             if current_node.right:  
                 stack.append(current_node.right)  
             if current_node.left:  
                 stack.append(current_node.left)
```

```
In [32]: iterative_inorder_traversal(root)
```

```
dog  
little  
the  
very  
is  
cute
```

Depth-first traversal pre-order (dict + recursive)

Recursive implementation using a dictionary data structure.

Depth-first traversal pre-order (dict + recursive)

Recursive implementation using a dictionary data structure.

```
In [33]: TT = {"dog": ["little", "very"],  
              "little": ["the"],  
              "the": [],  
              "very": ["is", "cute"],  
              "is": [],  
              "cute": []  
            }
```

```
In [34]: def preorder(T, node):  
          if node is not None:  
              print(node)  
              if len(T[node]) > 0:  
                  preorder(T, T[node][0])  
              if len(T[node]) > 1:  
                  preorder(T, T[node][1])
```

```
In [35]: preorder(TT, "dog")
```

```
dog  
little  
the  
very
```

is
cute

Iterative version.

```
In [36]: def preorder_traversal(T, node):  
         stack = [node]  
  
         while stack:  
             current_node = stack.pop()  
             print(current_node)  
  
             if len(T[current_node]) > 1:  
                 stack.append(T[current_node][1])  
  
             if len(T[current_node]) > 0:  
                 stack.append(T[current_node][0])
```

```
In [37]: preorder_traversal(TT, "dog")
```

```
dog  
little  
the  
very  
is  
cute
```

Solution: inorder traversal

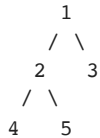
```
In [38]: def inorder(T, node):  
         if node is not None:  
             if len(T[node]) > 0:  
                 inorder(T, T[node][0])  
             print(node)  
             if len(T[node]) > 1:  
                 inorder(T, T[node][1])
```

```
In [39]: inorder(TT, "dog")
```

```
the  
little  
dog  
is  
very  
cute
```

Breadth-first search (or traversal)

Breadth-first search (BFS) is a traversal method that visits all the nodes in a tree (or a graph) level by level.



The main difference will be that we use a Queue instead of a Stack

```
In [40]: def bfs_print(node):  
    if node is None:  
        return  
  
    queue = [node]  
  
    while queue:  
        current_node = queue.pop(0)  
        print(current_node.value, end=' ')  
  
        if current_node.left:  
            queue.append(current_node.left)
```

```
if current_node.right:  
    queue.append(current_node.right)
```

```
In [41]: root = Node(1)
         root.left = Node(2)
         root.right = Node(3)
         root.left.left = Node(4)
         root.left.right = Node(5)
```

```
In [42]: bfs_print(root)
```

1 2 3 4 5

