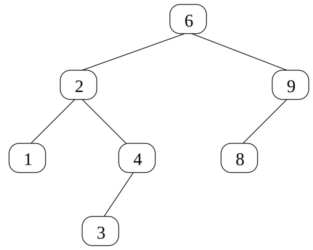
1. Consider tree:



One way to "visualize" this tree is:

```
_6_
_2_       _9
 1        _4        8
 3
```

Nodes are listed depth-vise, starting at depth 0, every depth at own row. Within one row, nodes are listed from left to right. If node x has both children, we denote $\_x\_$, if only left child we denote $\_x$, etc.

Design algorithm that given a tree does this visualization for it.

2. The following recursive algorithm does an inorder traversals to a tree

inorder-tree-walk(x)
    if x $\neq$ NIL **then**
        inorder-tree-walk(left[x])
        print key[x]
        inorder-tree-walk(right[x])

Another way to traverse through a tree is to use *preorder* traversal, where the node itself is treated before going to its children. As recursive algorithm:

preorder-tree-walk(x)
    if x $\neq$ NIL **then**
        print key[x]
        preorder-tree-walk(left[x])
        preorder-tree-walk(right[x])

- list the nodes of tree in exercise 2 in preorder
- implement preorder tree traversals without recursion
- implement inorder tree traversals without recursion

What is the time/state complexity of operations?

3. **2-3-4-tree**

Topic of Chapter 18 of the Cormen book is B-tree. A special case of B-tree, where $t = 2$, is called 2-3-4 tree (Cormen page 439). Find out how 2-3-4 tree works and denonstrate its usage when keys 41, 38, 12, 19 and 8 are added to an empty tree. Draw tree after each insertion.

Red-black tree is easy to convert to a 2-3-4 tree, and vice versa. How this is done?