Datastructuren

https://git.lumc.nl/j.k.vis/datastructuren

H.J. Hoogeboom
S. de Gouw
J.K. Vis

Universiteit Leiden

najaar 2018

Table of Contents I

Contents

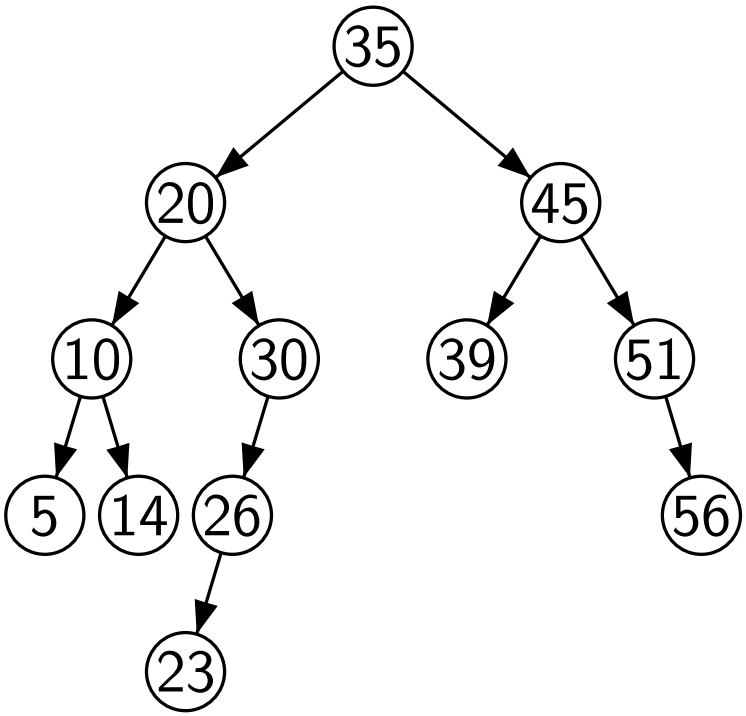
Binary trees: Recursive definition

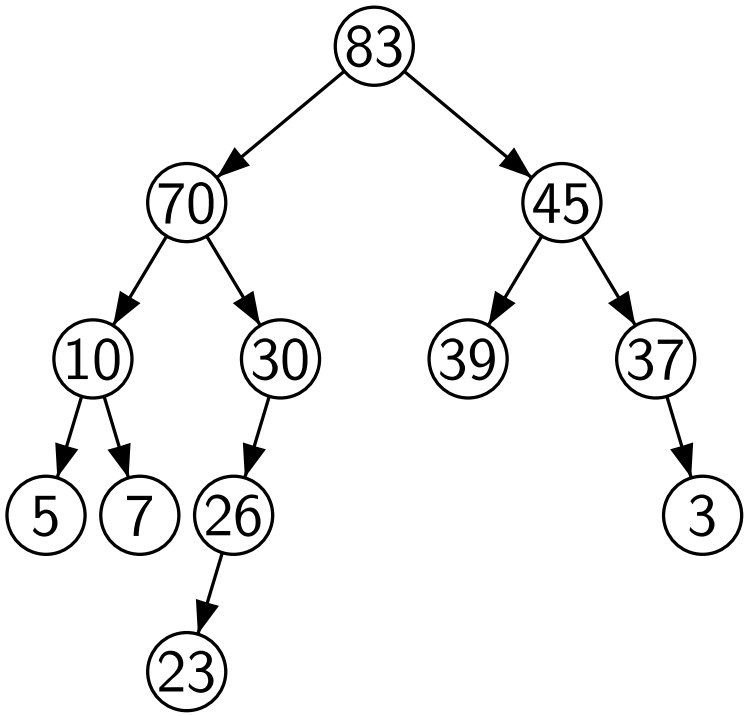
Definition (Binary Tree)

A binary tree is: an empty tree (without any nodes), or a node with two children L and R where L and R are binary trees.

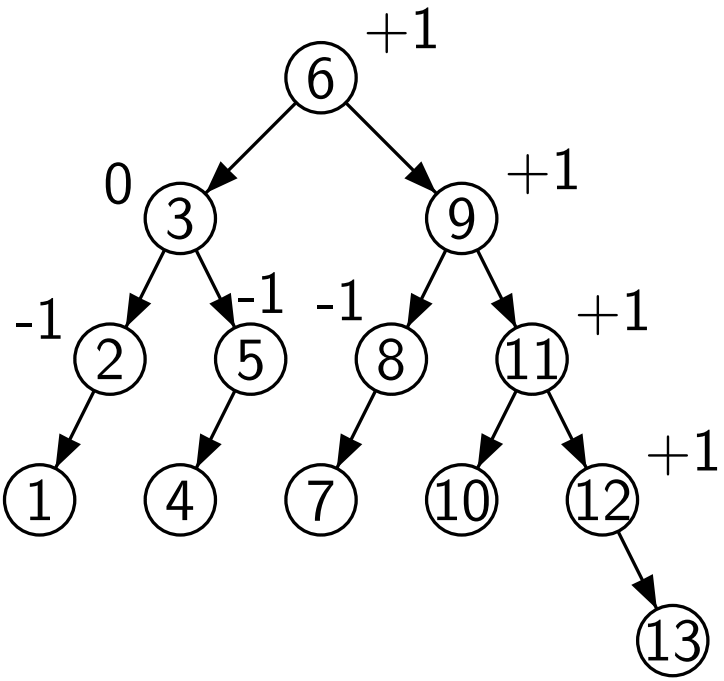
Informally: a tree where each node has ≤ 2 children.

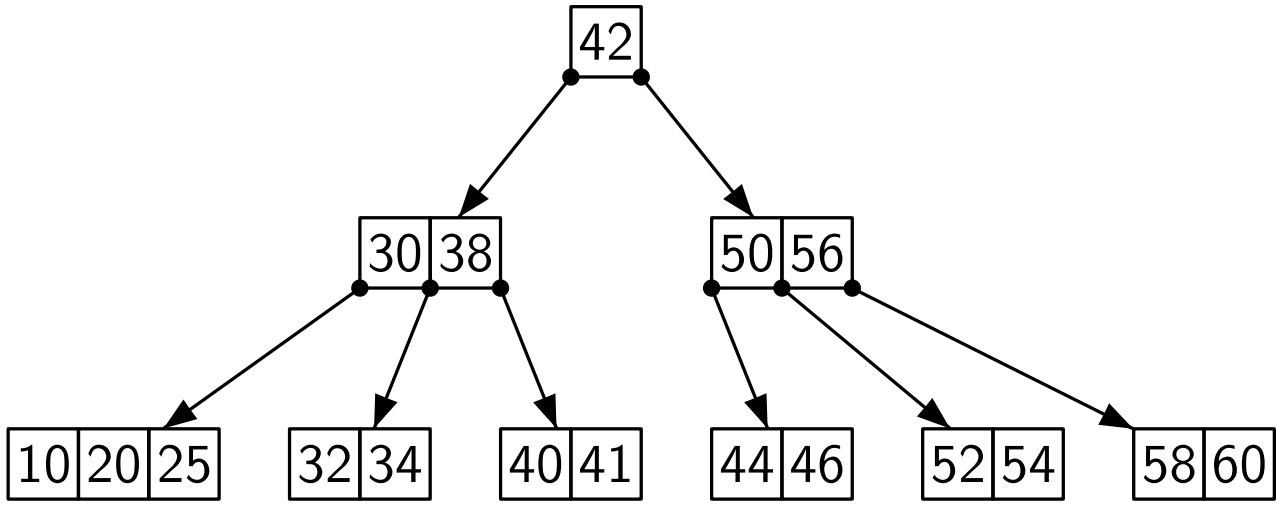
binary search tree vs heap order



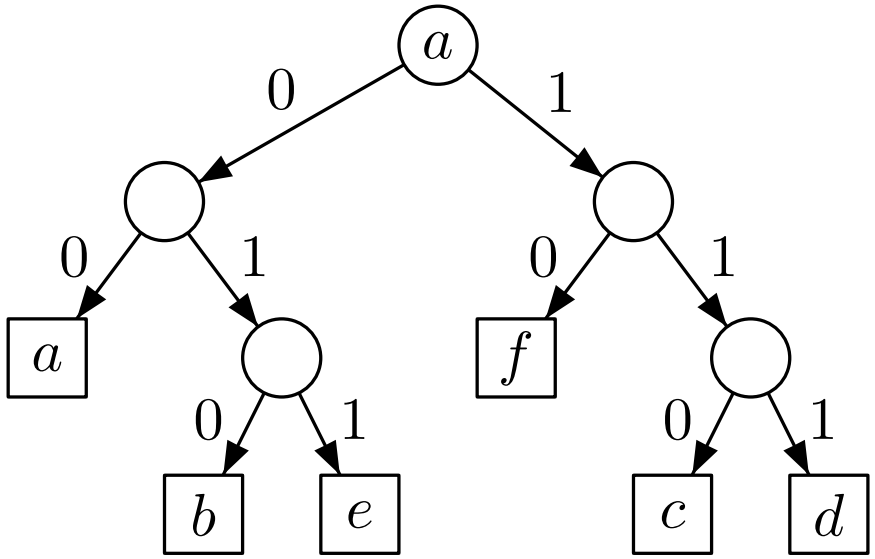


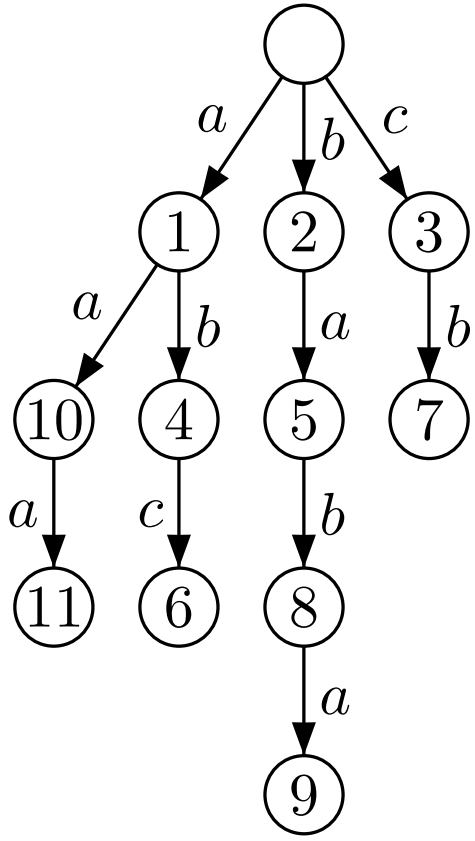
AVL-tree and B-tree



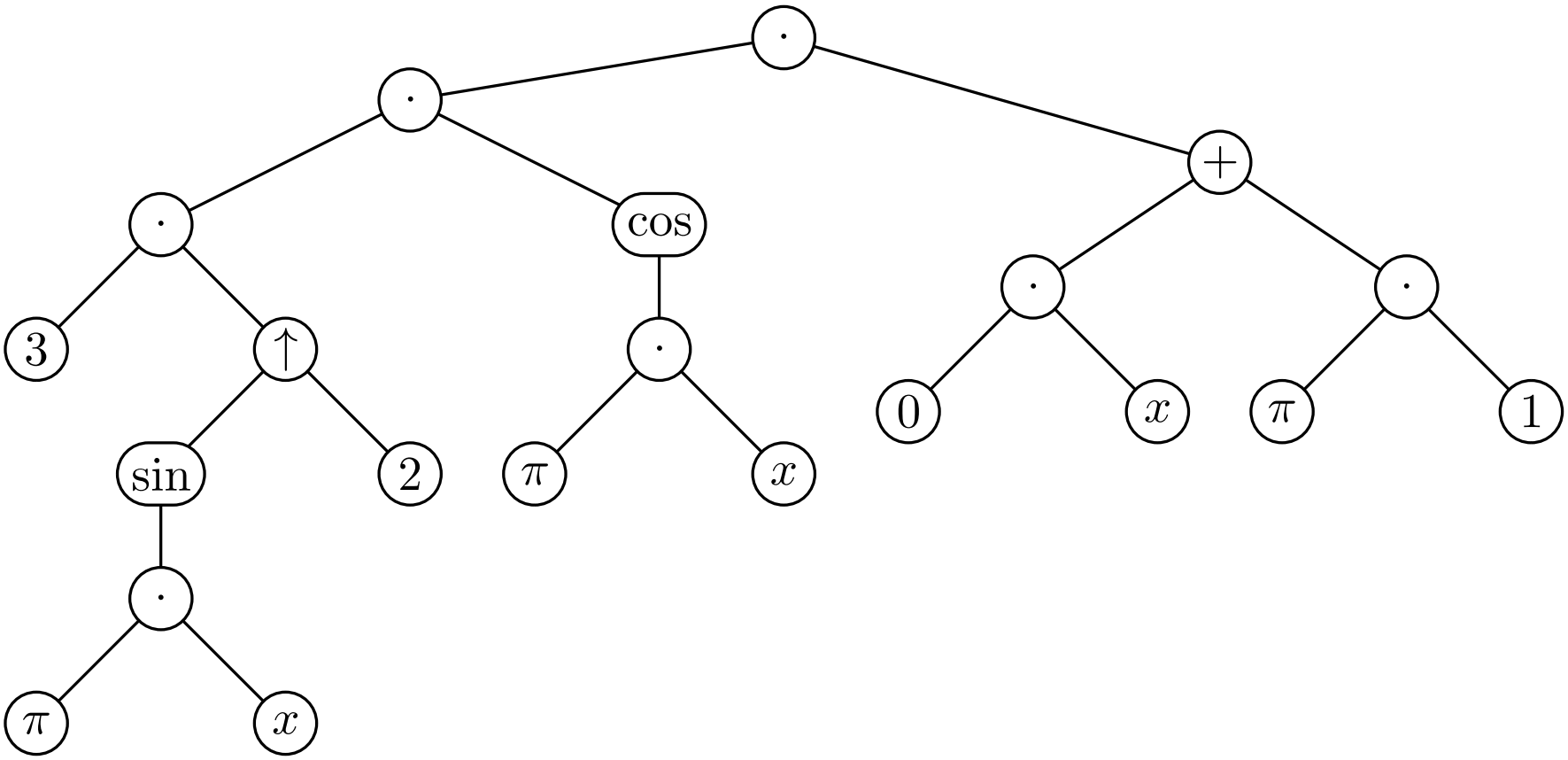


text compression Huffman & ZLW





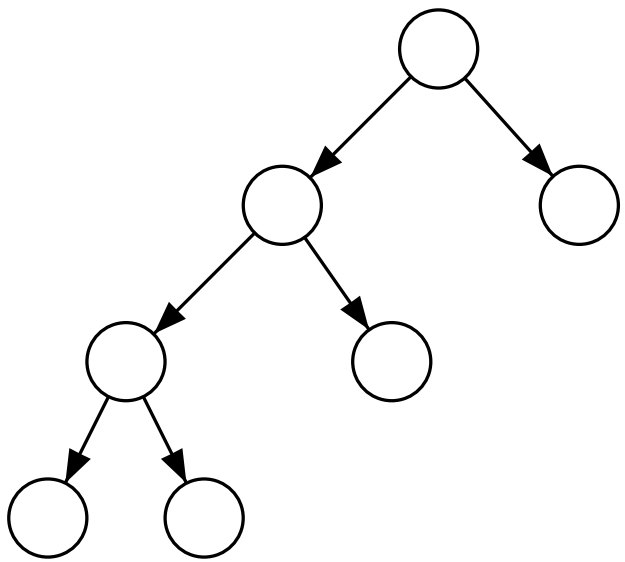
expression tree



Full binary tree

Definition (Full binary tree)

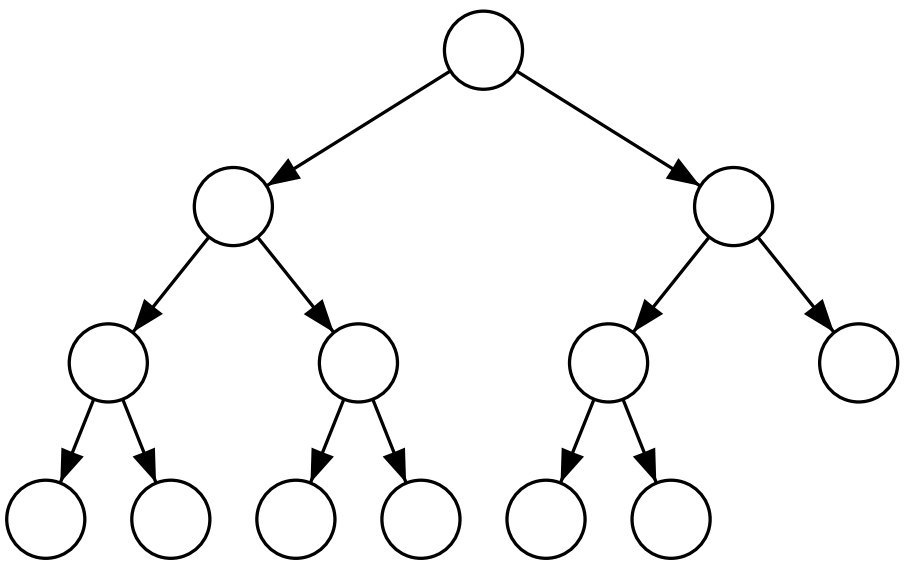
A binary tree is *full* if every node has 0 or 2 children



Complete binary tree

Definition (Complete binary tree)

A binary tree is *complete* if all levels are filled, except possible the last, and the nodes are as far to the left as possible



Representating binary trees

Representing binary tree with pointers

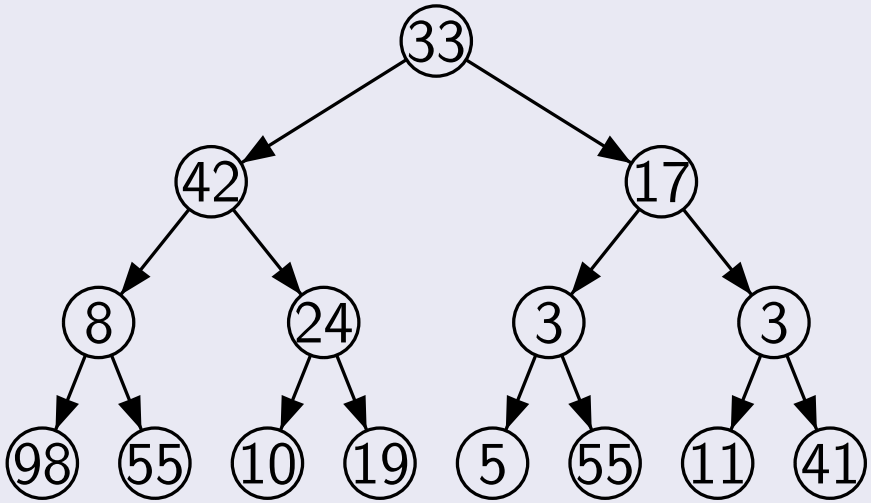
```
template <class T>
class BinKnp {
          \\ CONSTRUCTOR
    BinKnp ( const T& i,
             BinKnp<T> *l = nullptr,  \\ default
             BinKnp<T> *r = nullptr )
        : info(i)    \\ constructor of type T
        { links = l; rechts = r; }

private:   \\ DATA
    T info;
    BinKnp<T> *links, *rechts;
};
```

Representating binary trees

Representing binary tree with an array

Store root at index 1, left child of node i at index $2i$ and right child of node i at index $2i + 1$.



33	42	17	8	24	3	3	98	55	10	19	5	55	11	41
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Works well for *complete binary trees*, but leads to unused array elements for arbitrary binary trees (due to “missing” nodes)

Traversal

Is the process of *visiting* each node (precisely once) in a systematic way:

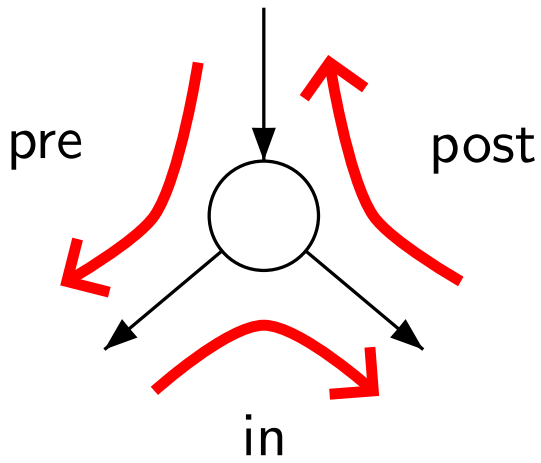
- breadth-first search;
- preorder;
- inorder;
- postorder.

- recursion;
- iterative;
- threaded.

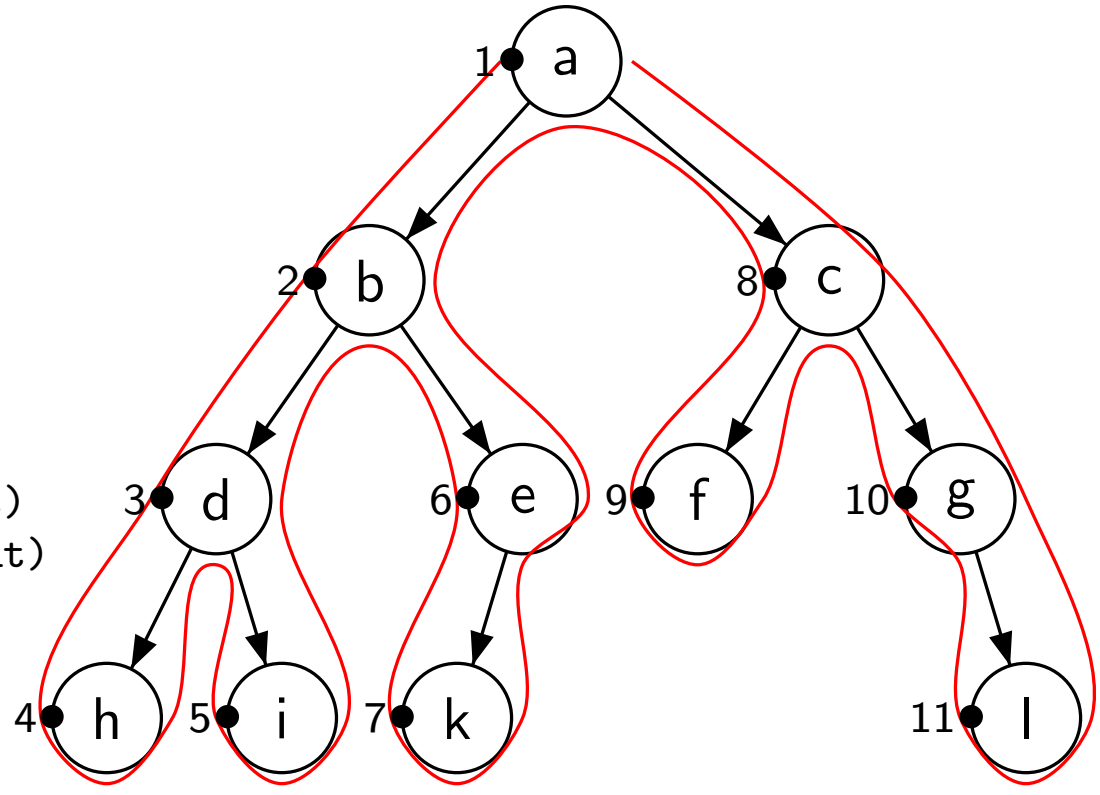
recursie

recursive

```
traversal( node )
     if (node != nil) then
          // pre-visit(node)
          traversal(node.left)
          // in-visit(node)
          traversal(node.right)
          // post-visit(node)
     fi
end // traversal
```



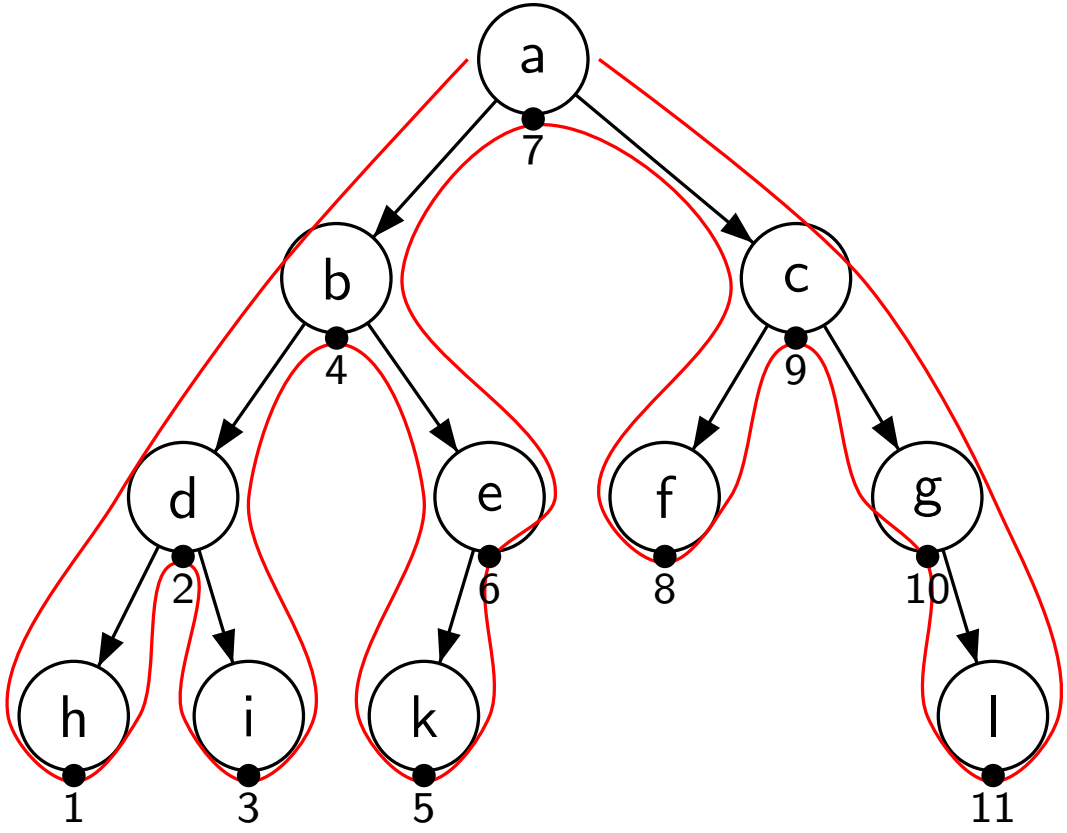
```
pre-traversal( node )
    if (node != nil) then
        pre-visit(node)
        pre-traversal(node.left)
        pre-traversal(node.right)
    fi
end
```



preorder

a b d h i e k c f g l

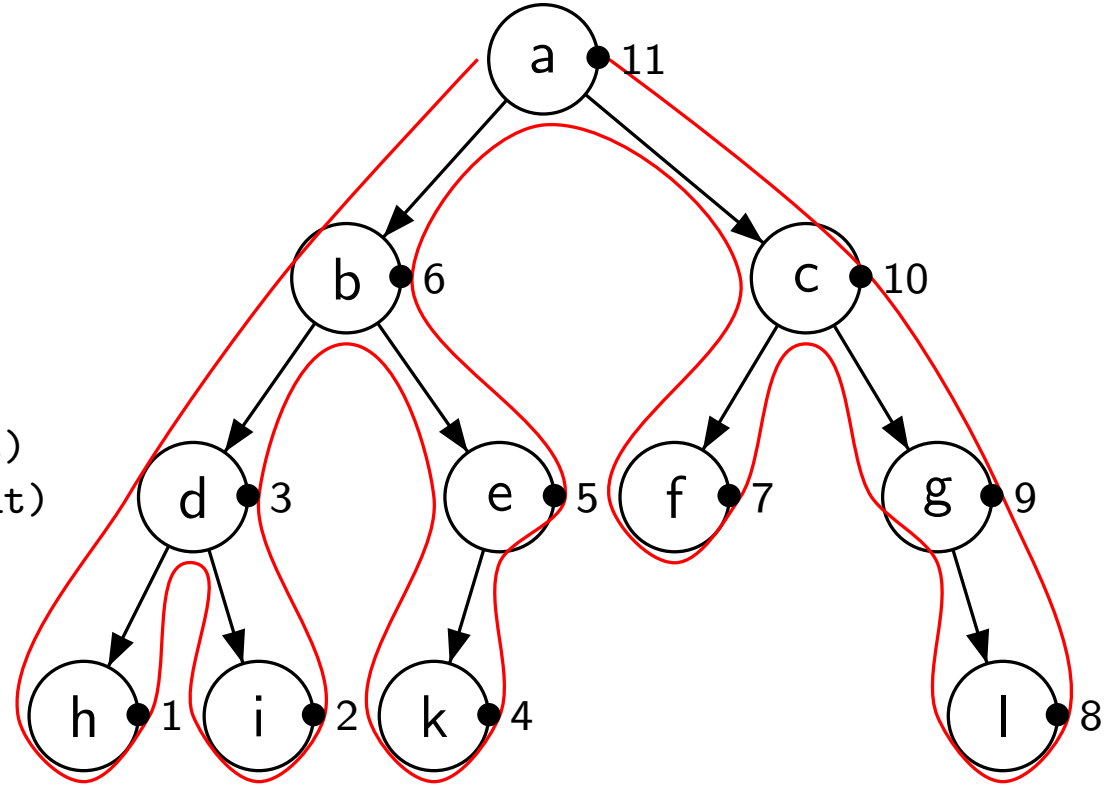
```
in-traversal( node )
    if (node != nil) then
        in-traversal(node.left)
        in-visit(node)
        in-traversal(node.right)
    fi
end
```



inorder

h d i b k e a f c g i

```
post-traversal( node )
    if (node != nil) then
        post-traversal(node.left)
        post-traversal(node.right)
        post-visit(node)
    fi
end
```



postorder

h i d k e b f l g c a

pre-order

```
iterative-preorder( root )
    S : Stack
    S.create()
    S.push( root )
    while ( not S.isEmpty() ) do
        node = S.pop()
        if (node != nil) then
            visit( node )
            S.push( node.right )
            S.push( node.left )
        fi
    do
end // iterative-preorder
```

pre-order (2)

```
iterative-preorder( root )
    S : Stack
    S.create()
    S.push( root )
    while ( not S.isEmpty() ) do
        node = S.pop()
        while (node != nil) do
            visit( node )
            S.push( node.right )
            node = node.left
        od
    do
end // iterative-preorder [bis]
```

in-order

```
iterative-inorder( root : Node)
    S : Stack
    S.create()
    // move to first node (left-most)
    walkLeft( root, S )
    while ( not S.isEmpty() ) do
        node = S.pop()
        visit( node )
        walkLeft( node.right, S )
    do
end // iterative-inorder

walkLeft( node : Node, S : Stack)
    while (node != nil) do
        S.push( node )
        node = node.left
    od
end // walkLeft
```

in-order (2)

```
iterative-inorder( root )
    S : Stack
    S.create()
    node = root;
    while (node != nil or
           not S.isEmpty() ) do
        if (node != nil) then
            S.push( node )
            node = node.left
        else
            node = S.pop()
            visit( node )
            node = node.right
        fi
    od
end // iterative-inorder [bis]
```

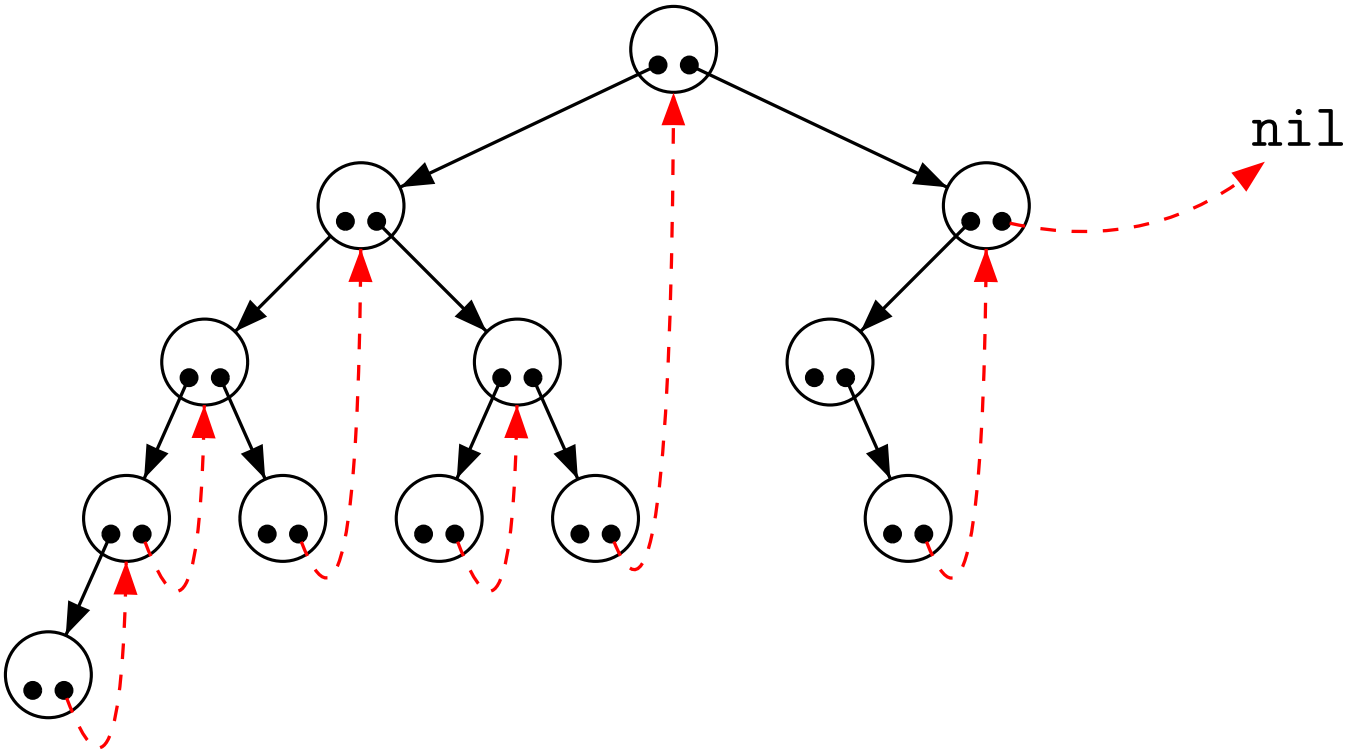
post-order

```
iterative-postorder( root )
    S : Stack;      // contains path from root
    S.create();
    last = nil
    node = root
    while (not S.isEmpty() or node != nil) do
        if (node != nil) then
            S.push(node)
            node = node.left
        else
            peek = S.top()
            if (peek.right != nil and last != peek.right) then
            // right child exists AND traversing from left, move right
                 node = peek.right
            else
                 visit(peek)
                 last = S.pop()
            fi
        fi
    od
end  // iterative-postorder
```

symmetric threads

Create thread in all nodes with `nil` right child to inorder successor

- Link rightmost node in left-subtree to current node
- Use boolean `IsThread` in each node to mark threaded nodes



traversal with symmetric threads

inorder threads

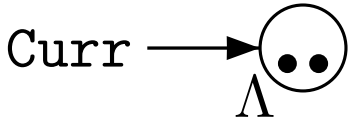
```
// assuming Root != nil, find first position in inorder
Curr = walkLeft( Root );
while (Curr != nil) do
     inOrderVisit( Curr );
     if (Curr.IsThread) then
         Curr = Curr.right; // to inorder successor
     else
         Curr = walkLeft (Curr.right)
     fi
od

walkLeft( node : Node)
    while (node.left != nil) do
        node = node.left
    od
    return node
end // walkLeft
```

Morris traversal - Pseudocode

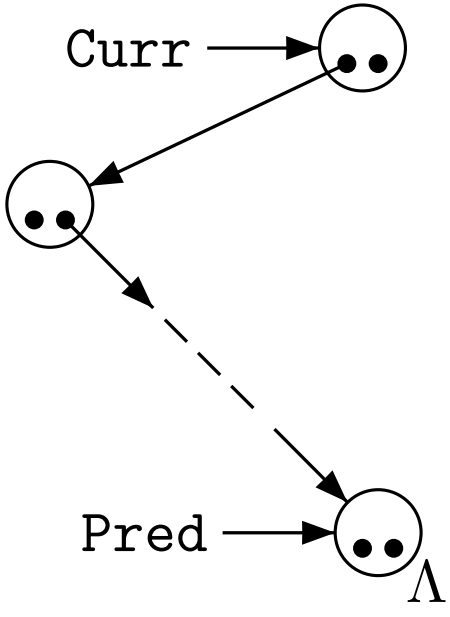
```
MorrisInorder()
    while not finished
        if node has no left descendant
            visit it
            go to the right
        else make this node the right child of the
            rightmost node in its left descendant
            go to this left descendant
```

Morris traversal - basic idea

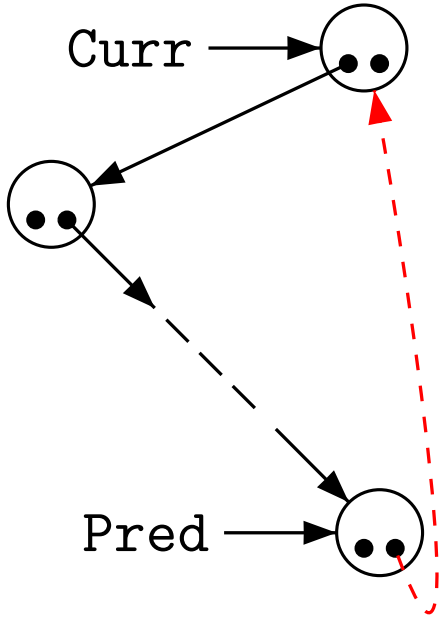


no left subtree:

1st ánd 2nd visit
go right
(by edge or by thread)



new subtree: 1st visit
construct thread
go left



been there: 2nd visit
delete thread
go right

Morris traversal - algorithm

morris-algo

```
Curr = Root;
while (Curr != nil) do
    if (Curr.left = nil) then
        inOrderVisit( Curr )
        Curr = Curr.right
    else
        // find predecessor
        Pred = Curr.left
        while (Pred.right != Curr && Pred.right != nil) do
            Pred = Pred.right
        od
        if  (Pred.right=nil) then
            // no thread: subtree not yet visited
            Pred.right = Curr
            Curr = Curr.left
        else
             // been there, remove thread
             Pred.right = nil
             inOrderVisit( Curr )
             Curr = Curr.right
        fi
    fi
```