Datastructuren

Data Structures

Hendrik Jan Hoogeboom

Informatica – LIACS
Universiteit Leiden

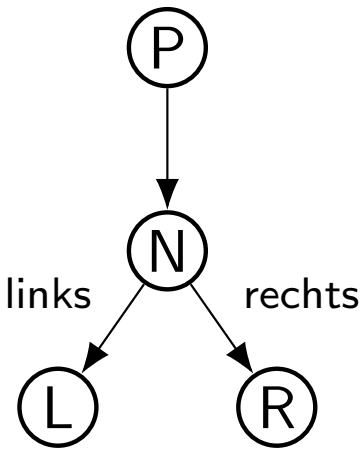
najaar 2023

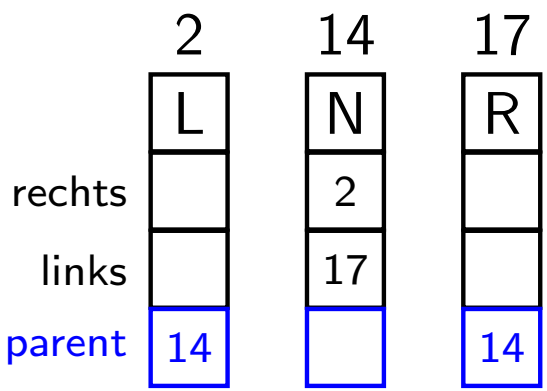
Table of Contents I

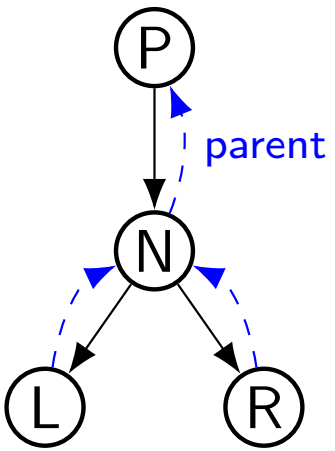
Contents

Contents

binary tree


P
N
links
rechts
L
R


2
14
17
L
N
R
rechts
2
links
17
parent
14
14


P
parent
N
L
R

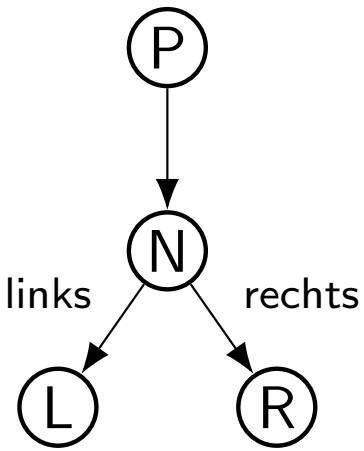
binary tree (algoritmiek)

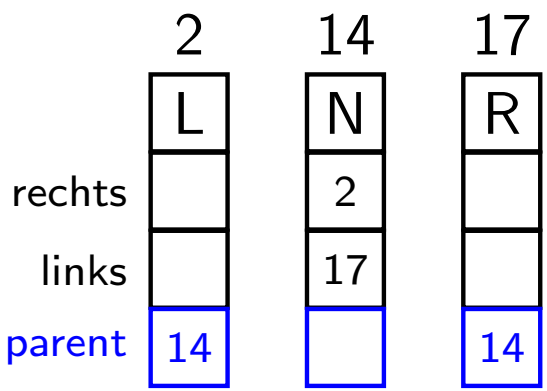
```
class knoop {   // een struct mag ook
    public:
    knoop ( ) { // constructor
        info = 0;
        links = nullptr;
        rechts = nullptr;
    }
    int info;
    knoop* links;
    knoop* rechts;
};              // knoop
```

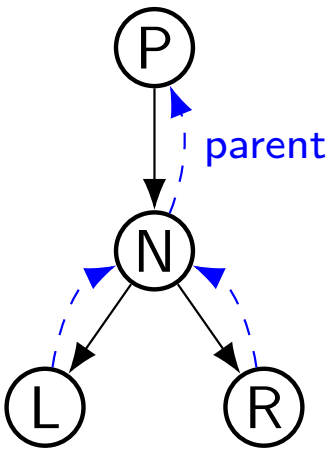
implementation binary tree

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

binary tree


P
N
links
rechts
L
R


2
14
17
L
N
R
rechts
2
links
17
parent
14
14


P
parent
N
L
R

traversal

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

orders

- breadth-first
- NLR preorder
- LNR inorder
- LRN postorder

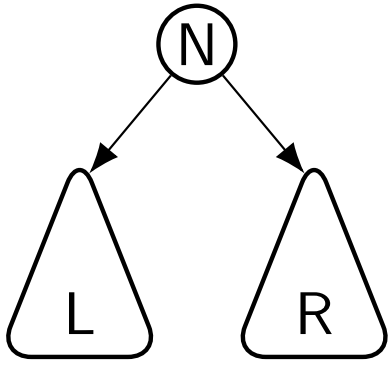
techniques

- recursion
- (parent pointer)
- iterative, with stack
- threads
- link inversion

Contents

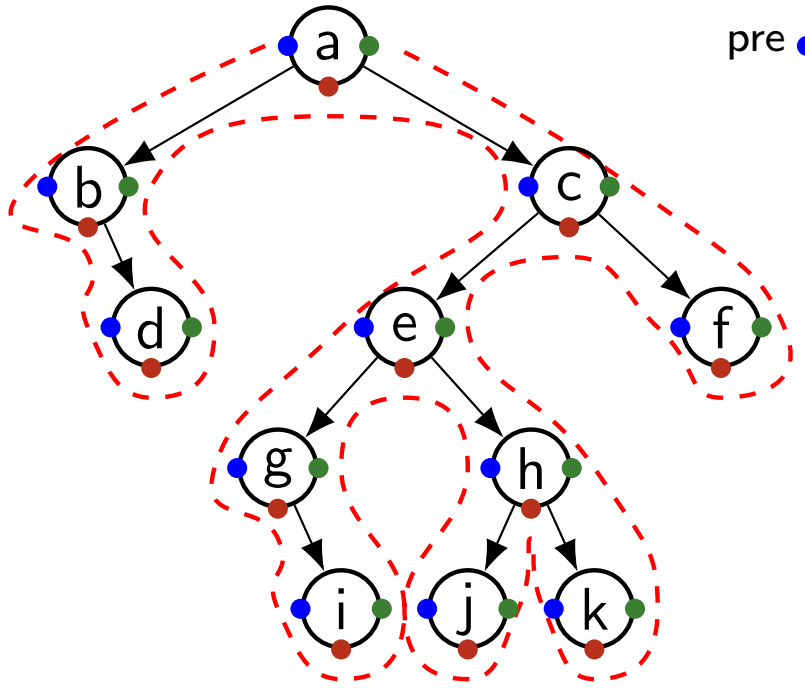
recursive tree orders

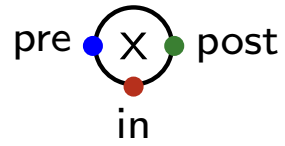
- *pre* order (NLR, node-left-right)
- *in* order, or *symmetric* order (LNR)
- *post* order (LRN)



Euler traversal

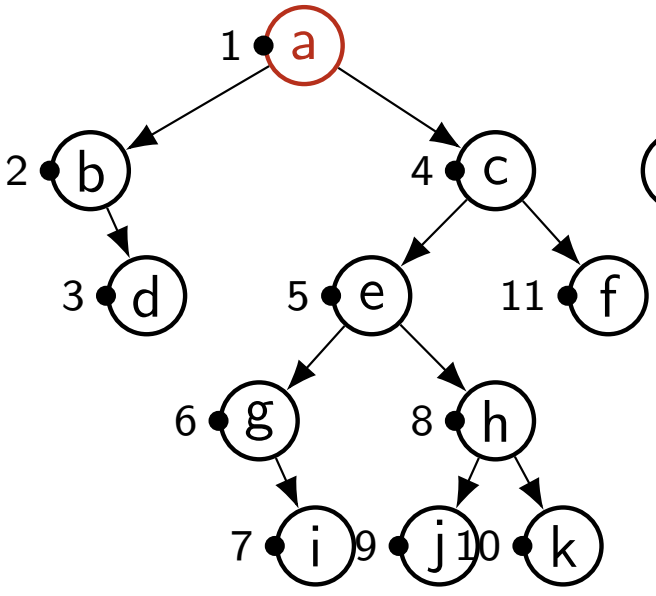
N_1 L N_2 R N_3





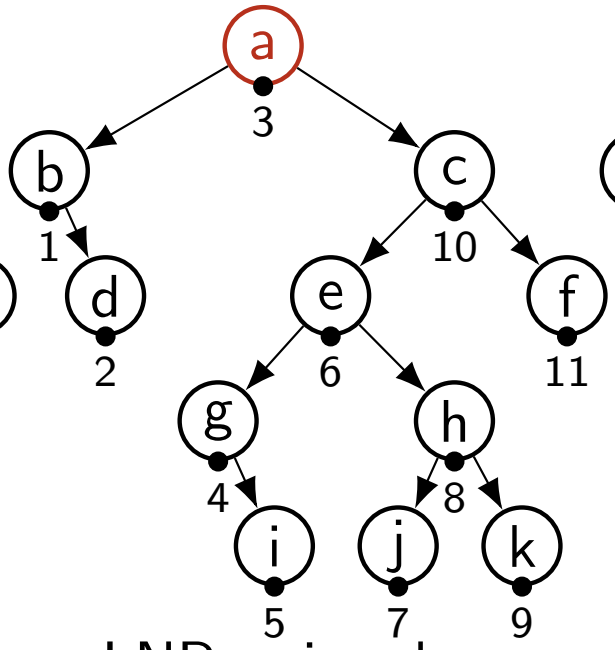
a b b d d d b a c e g g i i i g e h j j j h k k k h e c f f f c a

next: post-order (with a stack)



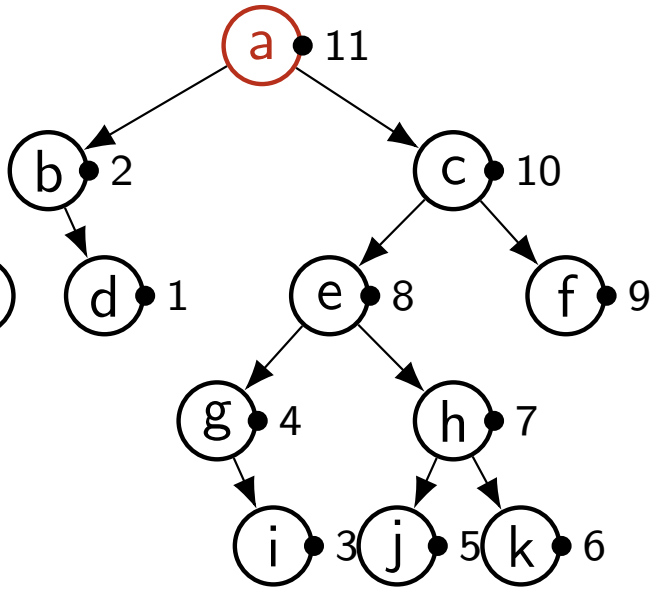
NLR = preorder

a b d c e g i h j k f



LNR = inorder

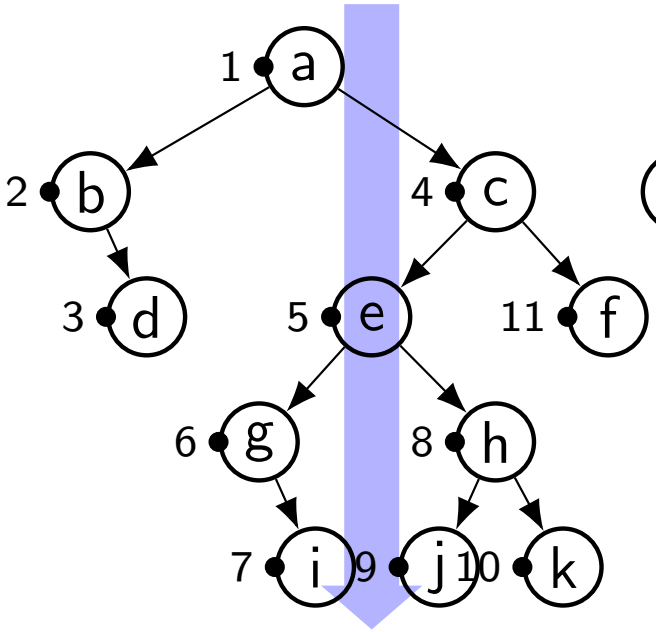
b d a g i e j h k c f



LRN = postorder

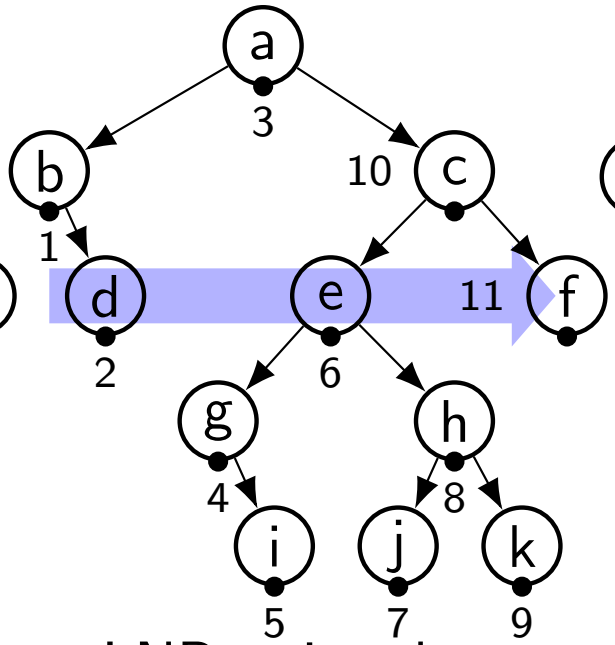
d b i g j k h e f c a

why? recursive tree orders



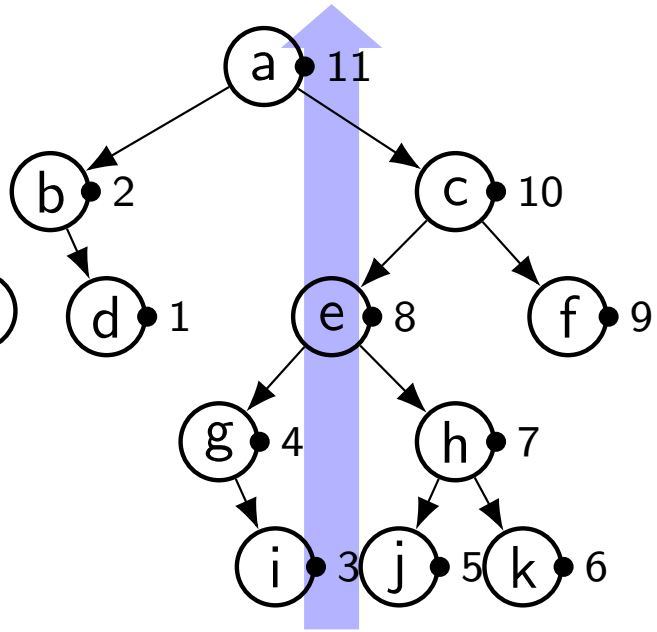
NLR = preorder

depth first



LNR = inorder

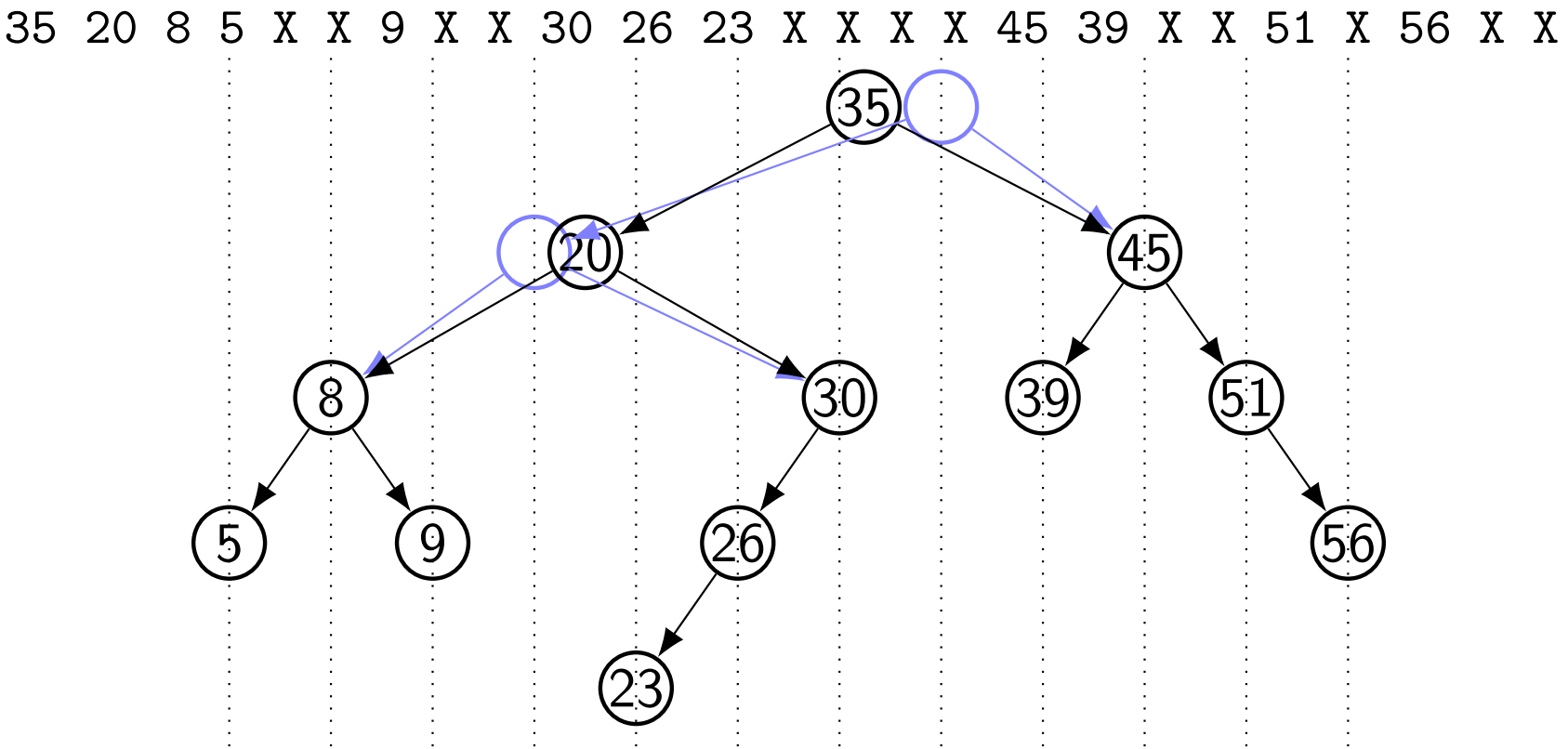
small to large



LRN = postorder

tree evaluation

drawing trees ☒



recursive tree traversal

recursive

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

Algoritmiek[1]

```
class knoop { // struct mag ook
   public:
      knoop ( ) { // constructor
      info = 0;
      links = nullptr;
      rechts = nullptr;
   }
   // maar misschien private
   int info;
   knoop* links;
   knoop* rechts;
};                    // knoop
```

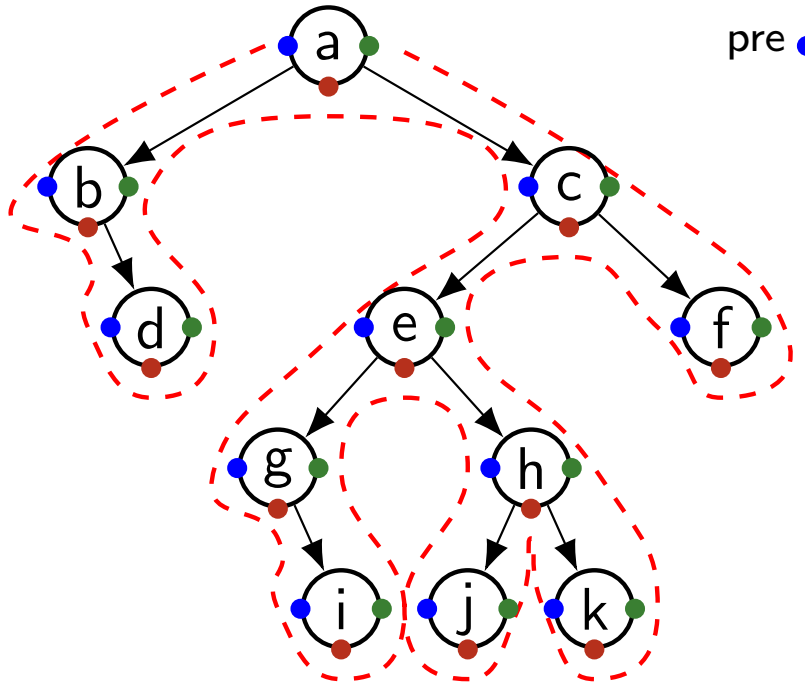
```
void preorde (knoop* root) {
   if ( root != nullptr ) {
      cout << root->info << endl;
      preorde (root->links);
      preorde (root->rechts);
   } // if
} // preorde

void symmetrisch (knoop* root) {
   if ( root != nullptr ) {
      symmetrisch (root->links);
      cout << root->info << endl;
      symmetrisch (root->rechts);
   } // if
} // symmetrisch
```

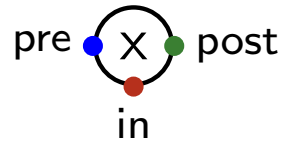
[1] ja, dat hebben we gezien

Contents

Euler traversal

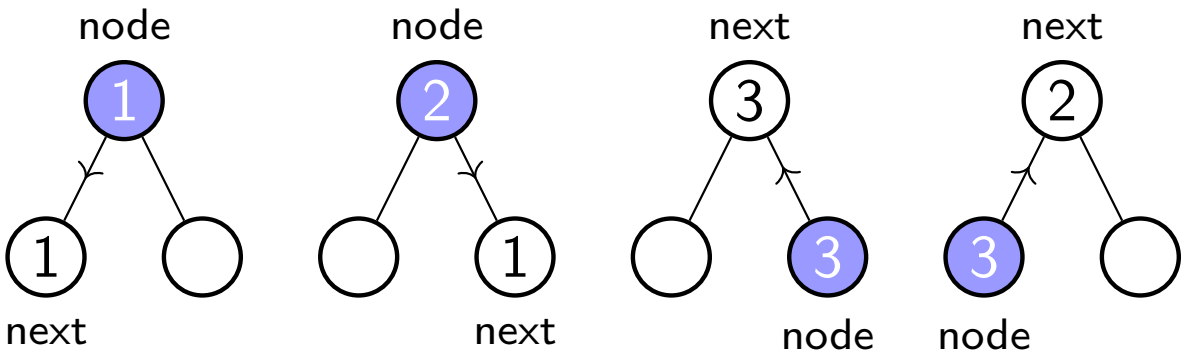
N_1 L N_2 R N_3

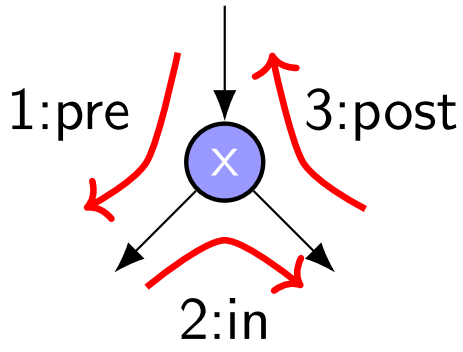




a b b d d d b a c e g g i i i g e h j j j h k k k h e c f f f c a

Euler traversal (as algorithm)





visit	test	direction	next
1	has left-child	down-left	1
		(stay)	2
2	has right-child	down-right	1
		(stay)	3
3	at left-child	up	2
	at right-child	up	3

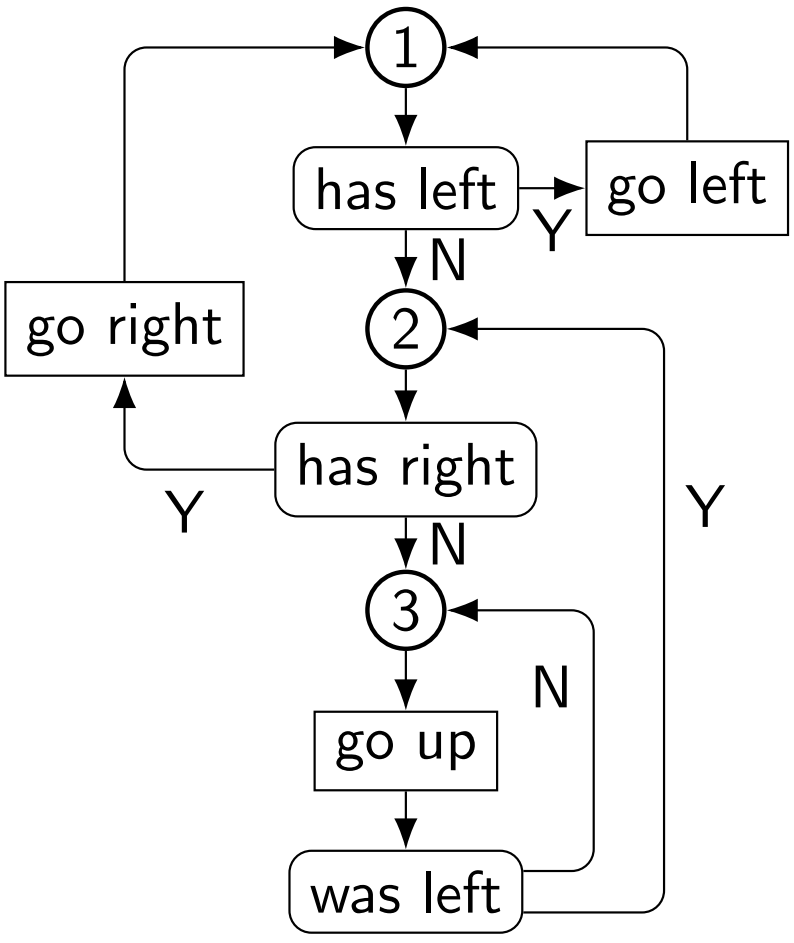
go up

hard in binary trees:

– find parent

– at left or right child?

Euler traversal ☒

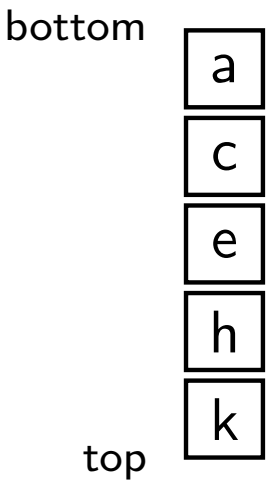


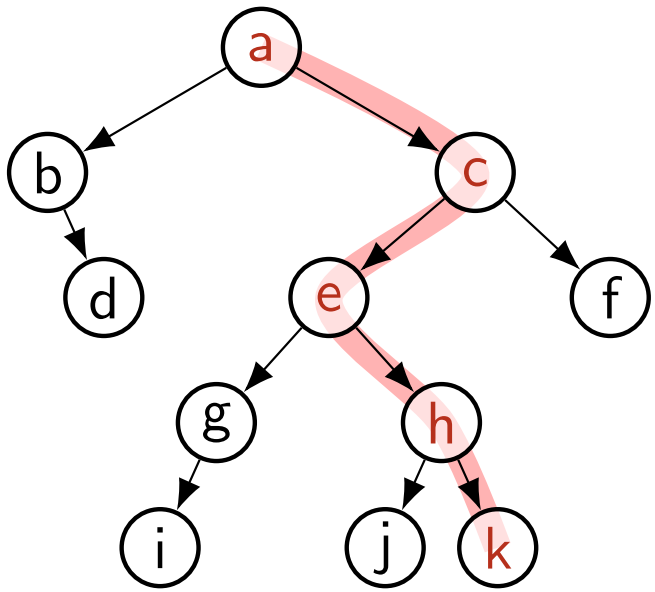
Euler traversal

```
start at root
while (node not nil)
do  pre-visit (1)
    while (has-left child)
    do  go left "push"
        pre-visit (1)
    od
    in-visit (2)
    while (not has-right)
    do  repeat
            post-visit (3)
            go up "pop"
            if nil then exit fi
        until (was-left)
        in-visit (2)
    od
    go right "push"
od
```

Contents

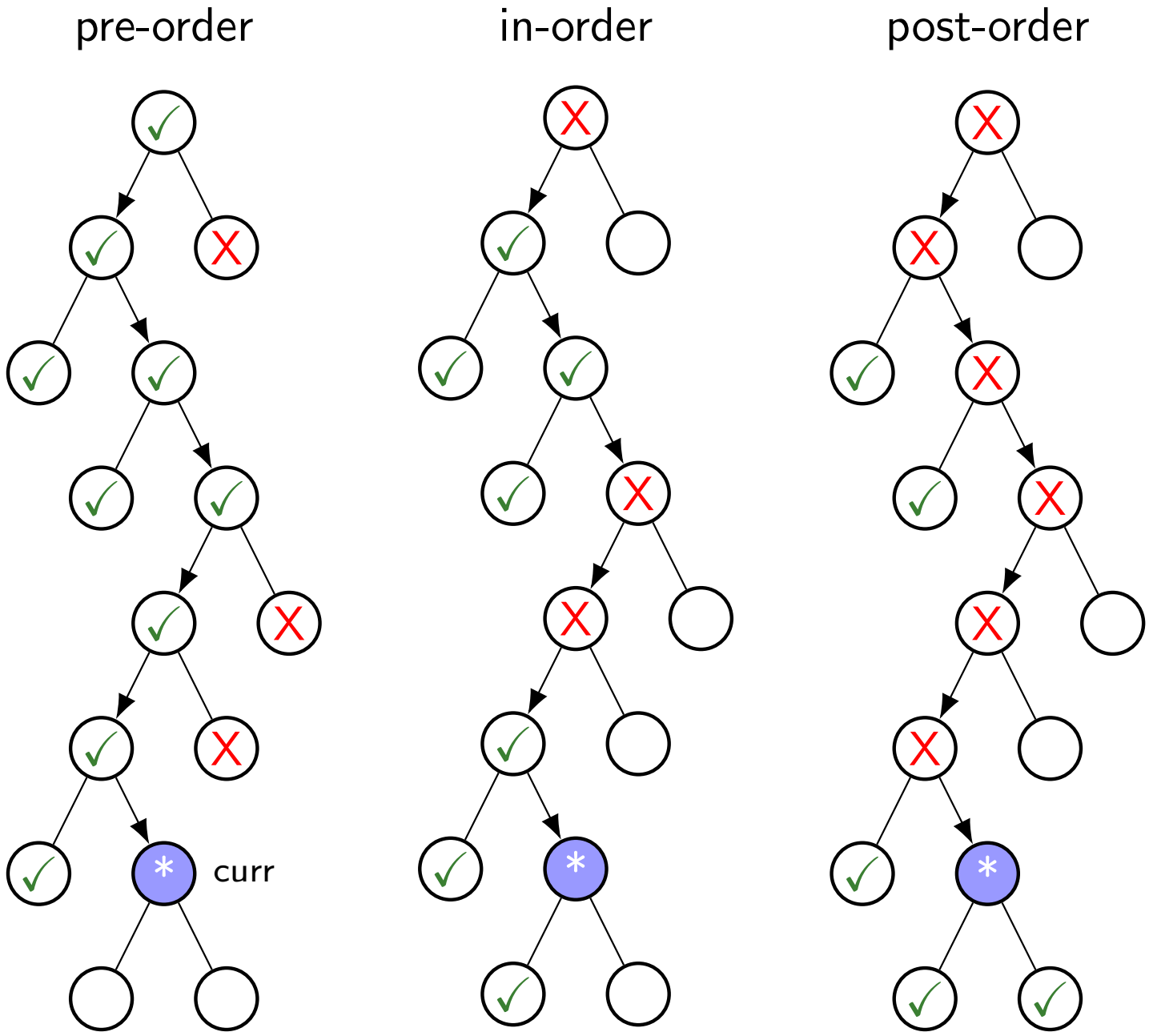
using a single stack





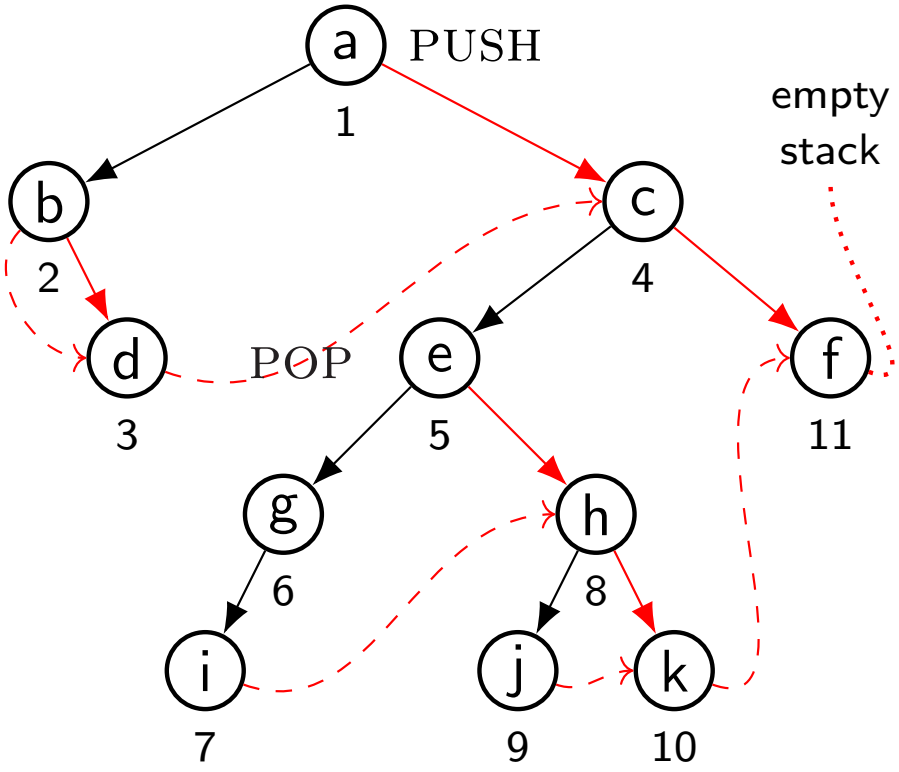
nodes stacked

* current
✓ visited
X on stack

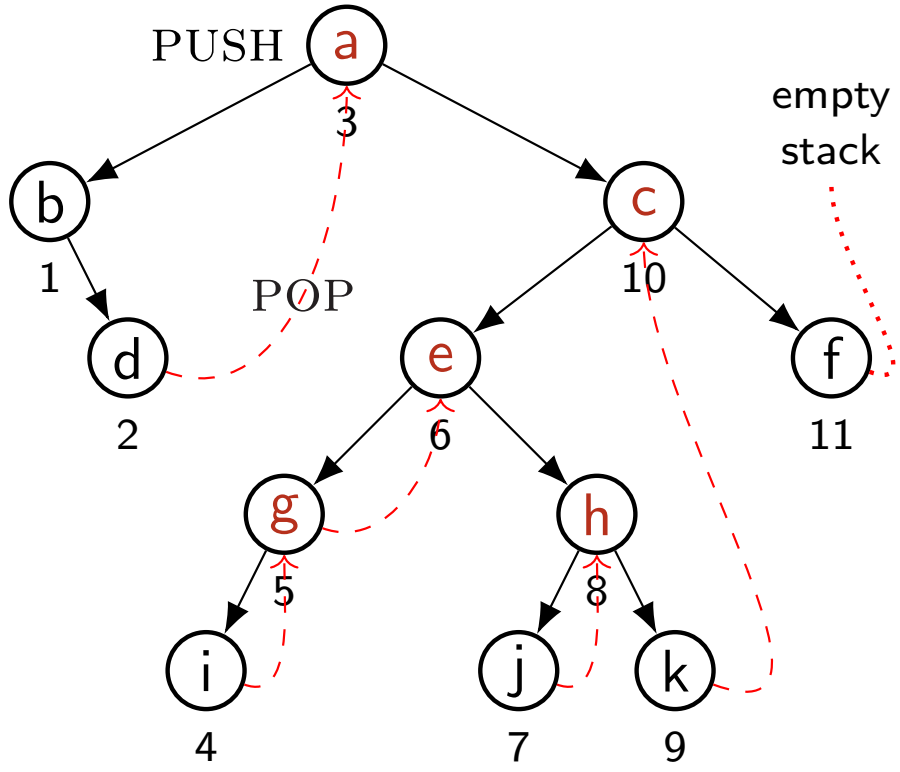


which nodes on stack

pre: right children



in: left parents



pre-order: from recursion to stack

recursive

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

pre-order

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

pre-order: removing tail recursion

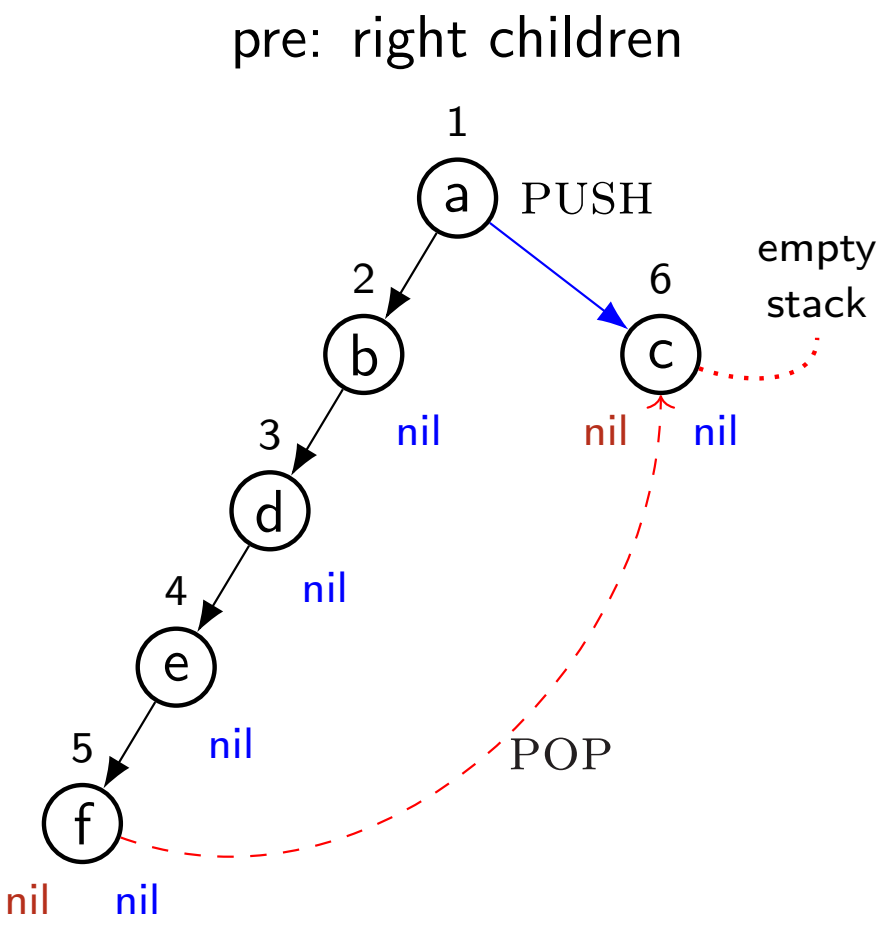
pre-order

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

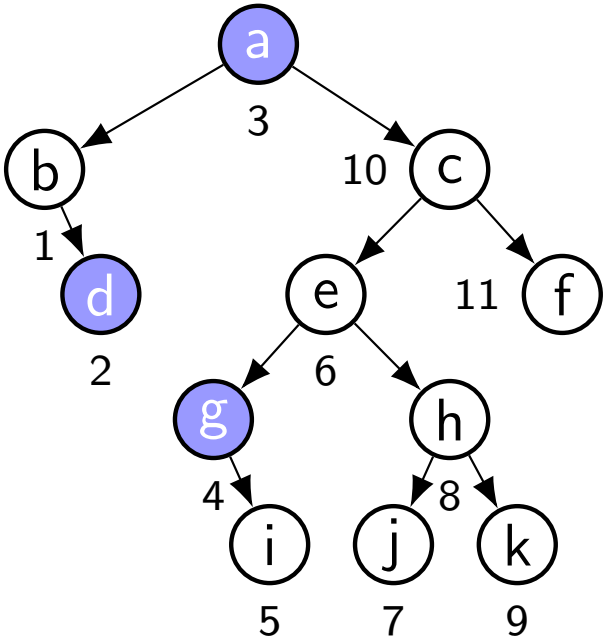
pre-order (wrapped)

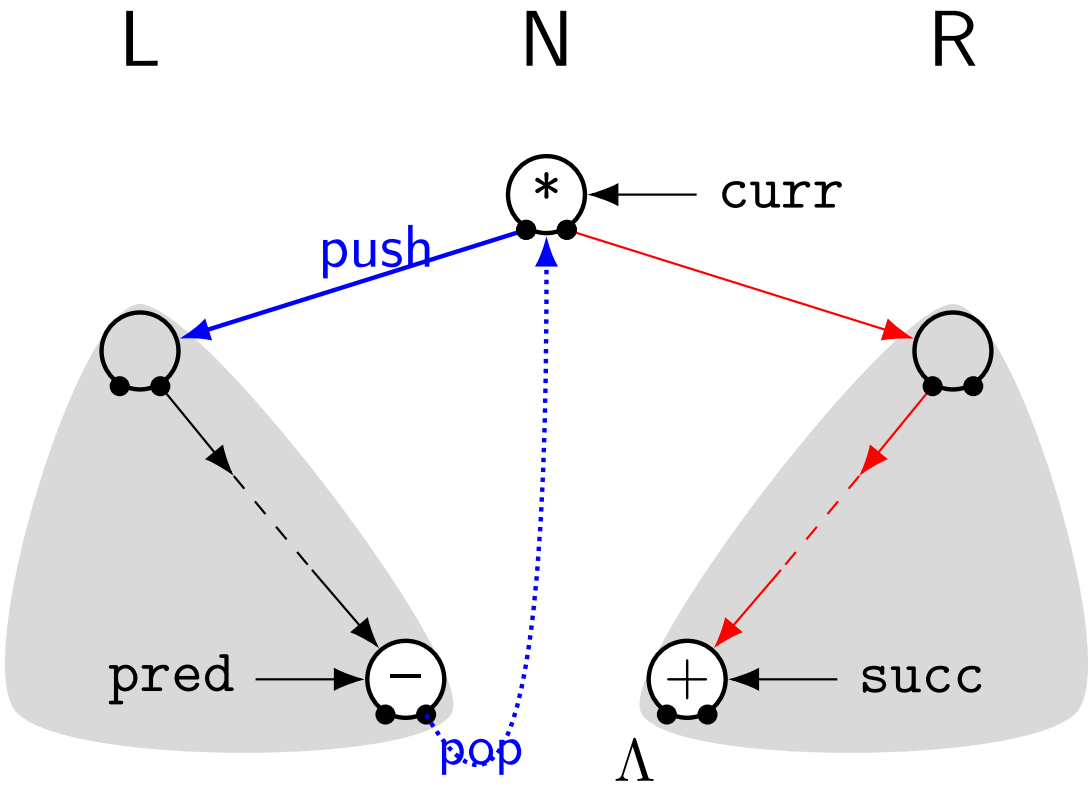
```
iterative-preorder( root )
    S : Stack
    S.create()
    S.push( root )
    while ( not S.isEmpty() )
    do  node = S.pop()
        while (node != nil)
        do  visit( node ) // pre-order
            S.push( node.right )
            node = node.left
        od
    od
end // iterative-preorder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

note: nil are 'skipped'



inorder ‘LNR’ with stack



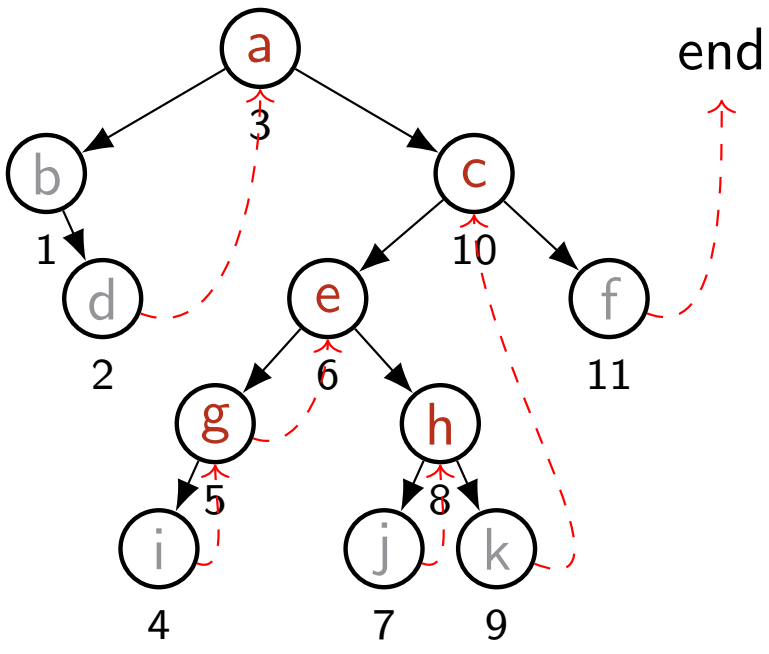


inorder traversal

in-order

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

stacked: left parents



using a function to find first-left

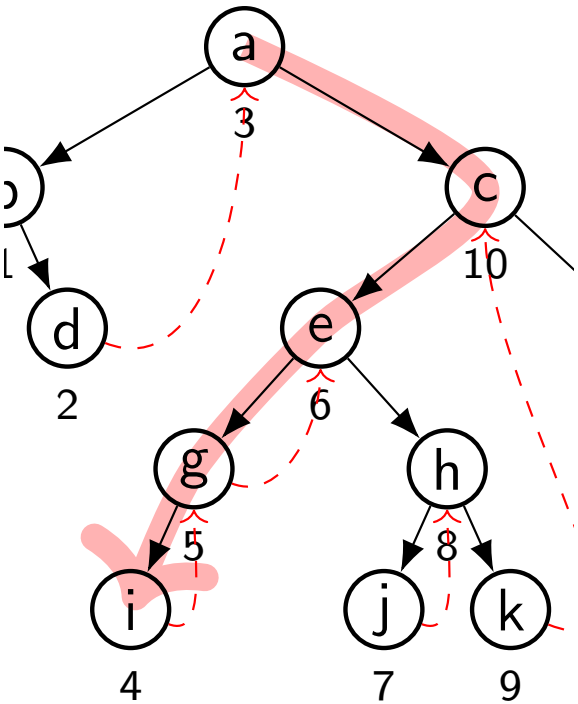
in-order

```
iterative-inorder( root )
    S : Stack
    S.create()
    node = root
    // move to first node (left-mo
    while (node != nil)
    do  S.push( node )
        node = node.left
    od
    while ( not S.isEmpty() )
    do  node = S.pop()
        visit( node ) // inorder
        node = node.right
        while (node != nil)
        do  S.push( node )
            node = node.left
        od
    od
end // iterative-inorder
```

in-order (netter)

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

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

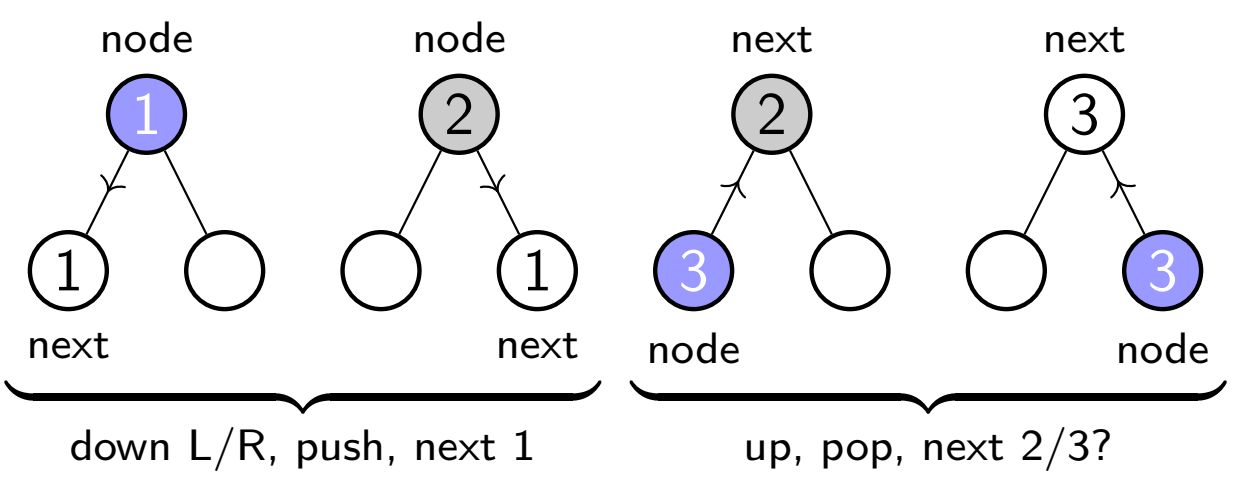
walk to successor vs one step at a time

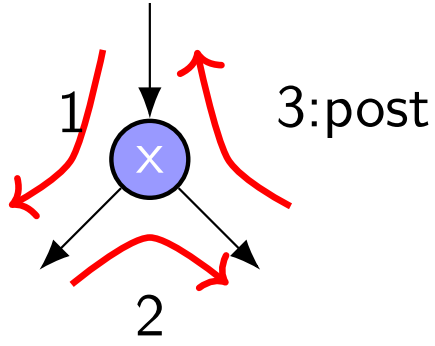


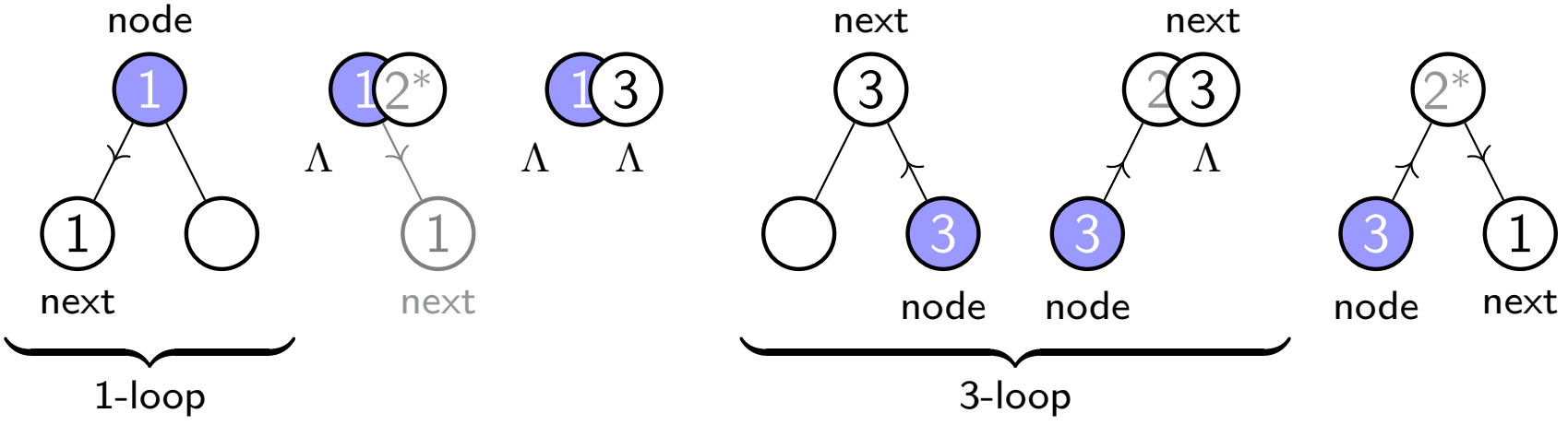
in-order (wikipedia)

```
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
```

Euler to postorder ☒

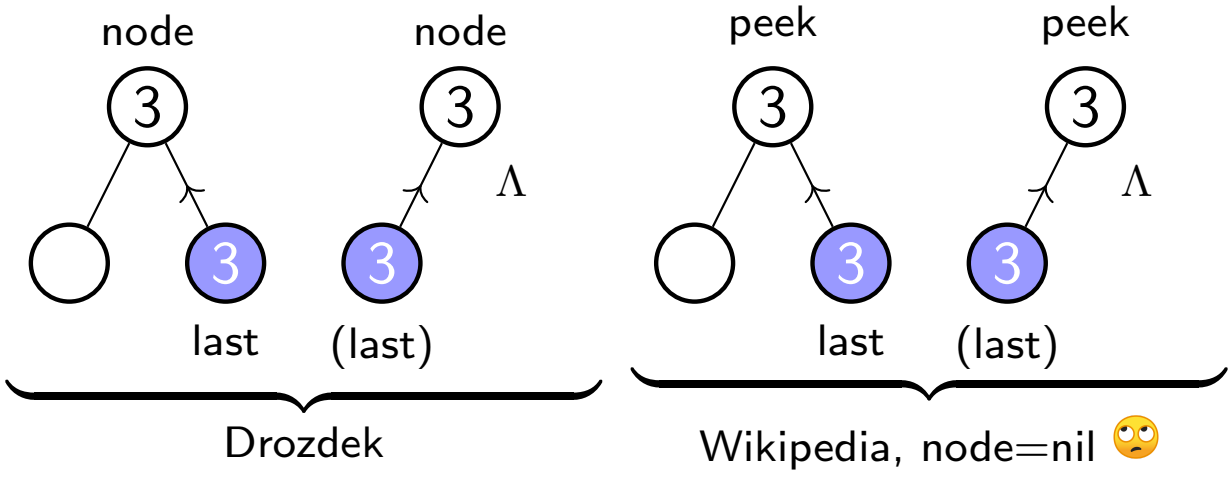






postorder implementations

- keep complete path from root to node on stack
- variant Euler traversal (only visit 3 matters):
  repeats: down (first visit: go left, push), switch, up (third visit: pop)
- node current position in tree
  last previously visited
  (next) parent = top of stack peek
- we go up while:



Drozdek

postorder (Drozdek)

```
template<class T>
void BST<T>::iterativePostorder() {
        Stack<BSTNode<T>*> travStack;
        BSTNode<T>* p = root, *q = root;
        while (p != 0) {
                for ( ; p->left != 0; p = p->left)
                travStack.push(p);
                while (p->right == 0 || p->right == q) {
                        visit(p);
                        q = p;
                        if (travStack.empty())
                        return;
                        p = travStack.pop();
                }
                travStack.push(p);
                p = p->right;
        }
}
```

Euler traversal

```
start at root
while (node not nil)
do  pre-visit (1)
    while (has-left child)
    do  go left "push"
        pre-visit (1)
    od
    in-visit (2)
    while (not has-right)
    do  repeat
            post-visit (3)
            go up "pop"
            if nil then exit fi
        until (was-left)
        in-visit (2)
    od
    go right "push"
od
```

postorder (pseudo Drozdek)

```
S : Stack
node = root, last = root
while (node != nil)
do      // go down left, 1st visit
    while (node.left != nil)
    do  S.push(node)
        node = node.left
    od
        // go up, 3rd visit
    while (node.right == nil or
           node.right == last)
    do  visit(node);
        last = node;
        if (S.isEmpty() ) then return fi
                          // exit
        node = S.pop()
    od
        // (2nd visit) go right via parent
    S.push(node)
    node = node.right
od
```

wikipedia

post-order (wikipedia 20.9'21)

```
procedure iterativePostorder( node )
    stack ← empty stack
    lastNodeVisited ← null
    while not stack.isEmpty() or node ≠ null
    do  if node ≠ null
            stack.push(node)
            node ← node.left
        else peekNode = stack.peek()
            // if right child exists and traversing node
            // from left child, then move right
            if peekNode.right ≠ null and lastNodeVisited ≠ peekNode.right
                node ← peekNode.right
            else visit(peekNode)
                lastNodeVisited ← stack.pop()
```

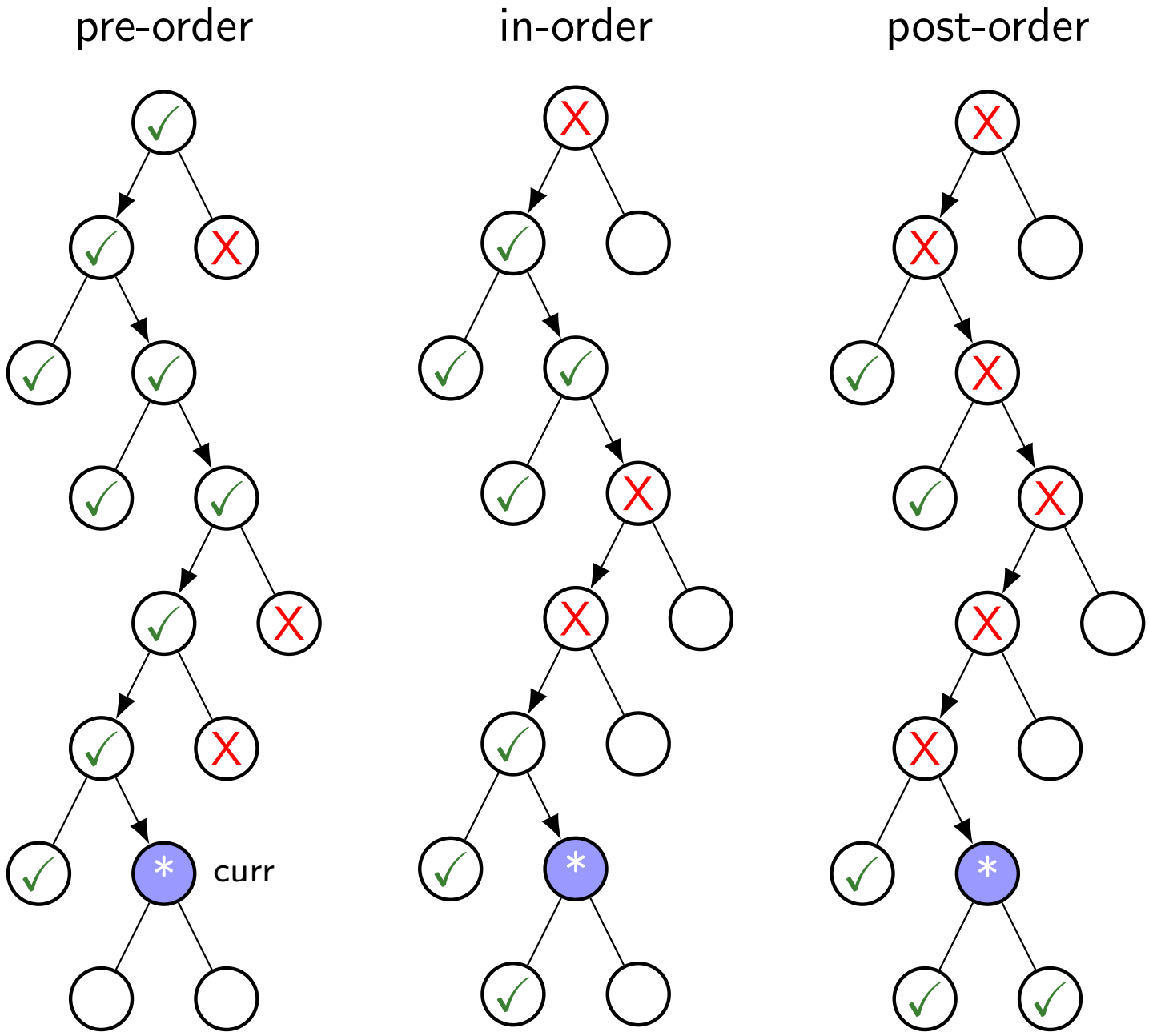
pseudo wikipedia ☒

post-order (pseudo-wikipedia)

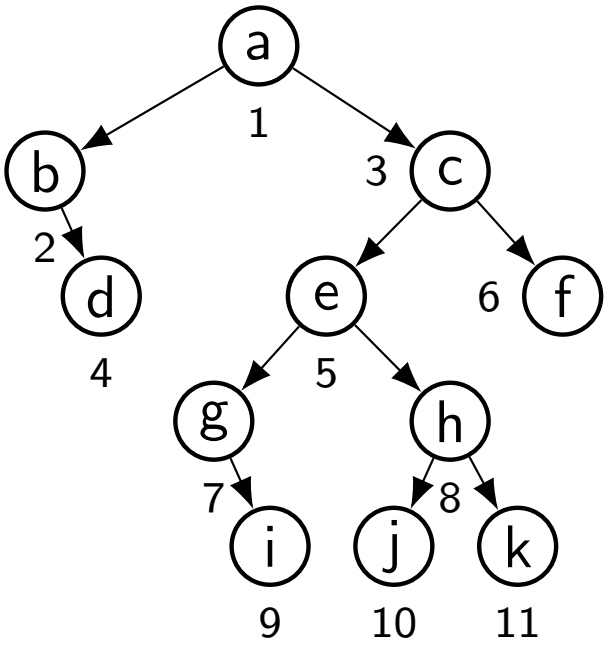
```
iterative-postorder( root )
    S : Stack // contains path from root
    S.create()
    last = nil
    node = root
    while (not S.isEmpty() or node != nil)
    do  if (node != nil)
        then S.push(node)
             node = node.left
        else // if right child exists and traversing node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

nodes stacked

* current
✓ visited
X on stack



ps. level-order ∼ BFS with queue

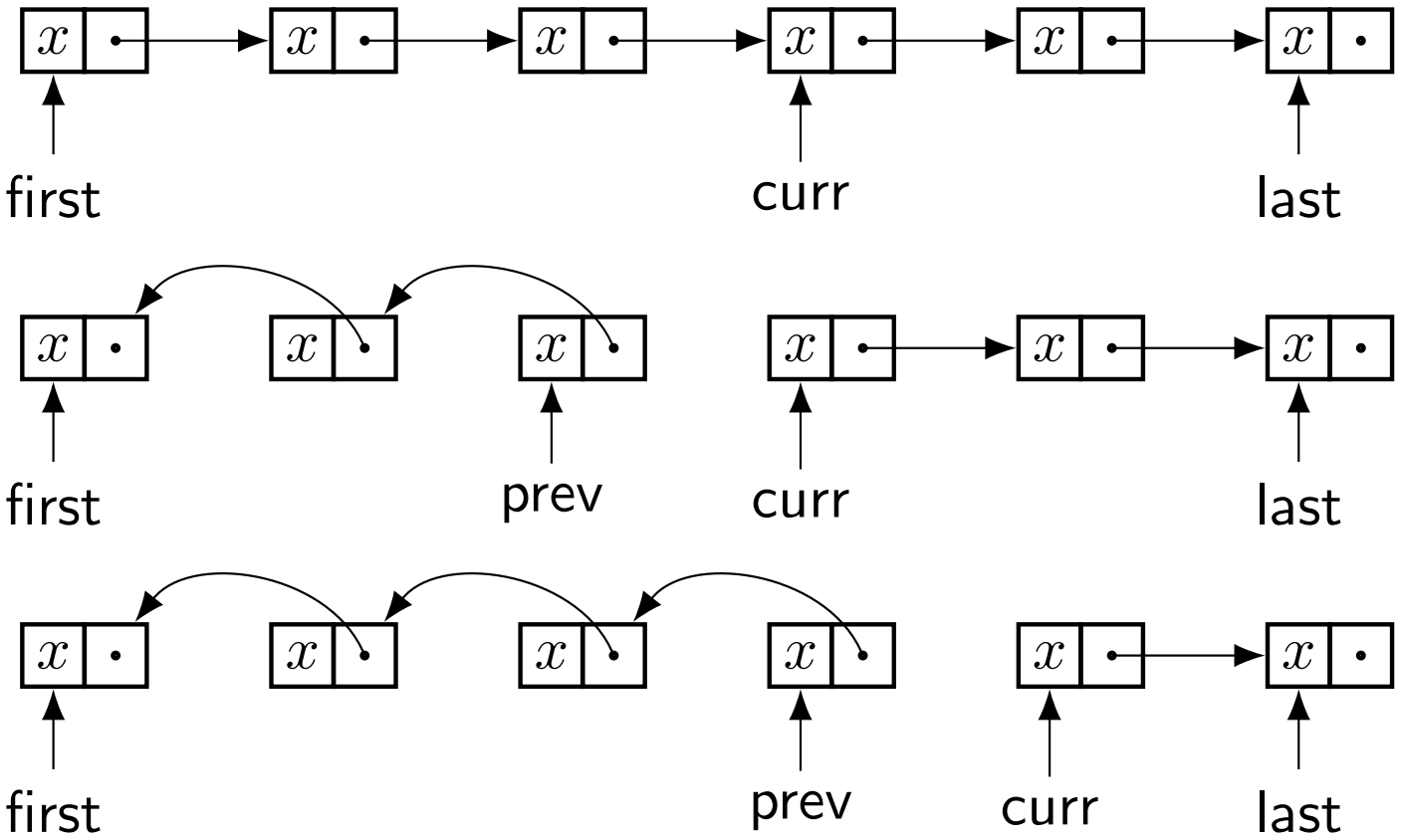


level-order

```
iterative-preorder( root )
    Q : Queue
    Q.create()
    Q.enqueue( root )
    while ( not Q.isEmpty() )
    do  node = Q.dequeue()
        visit( node )
        S.enqueue( node.left )
                unless (node.left==nil)
        S.enqueue( node.right )
                unless (node.right==nil)
    od
end // level-order
```

Contents

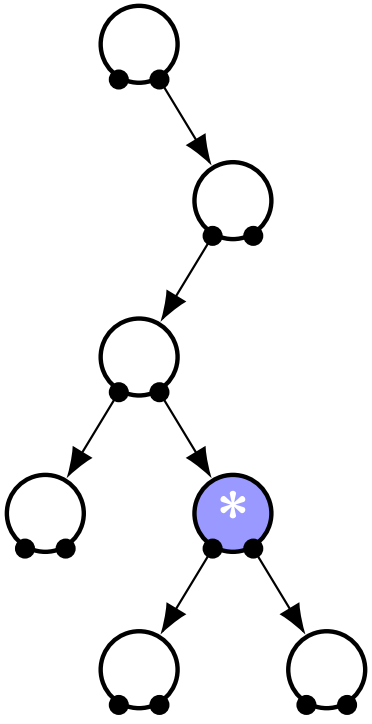
link inversion in linear list



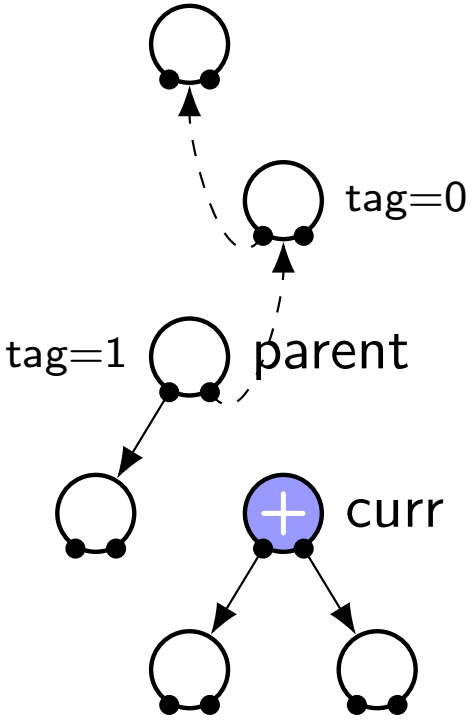
link inversion

- *Euler style*: global visit counter $1, 2, 3$
  can be used in pre-order, in-order or post-order fashion.
- no *external* stack
- path to visited node *link-inverted*, to function as stack
- *tag*: one bit per node (along the inverted path) to distinguish left/right children on inverted path
- keep pointer to parent (gap)
- structure is disturbed: only a single traversal at a time

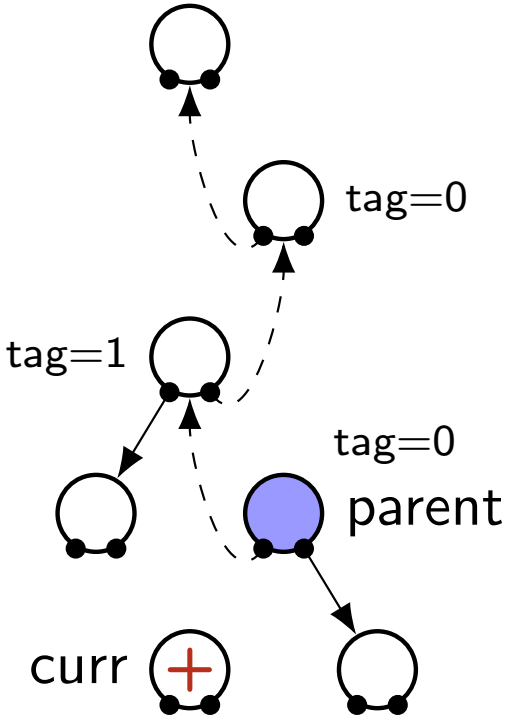
link inversion



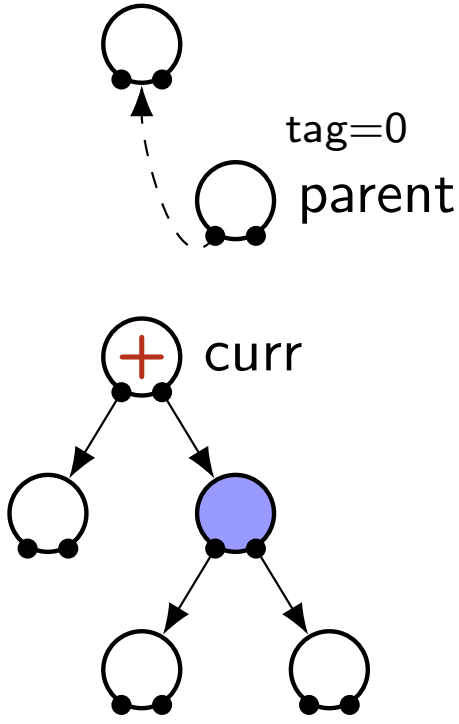
binary tree



visits at $*$



after 1st visit



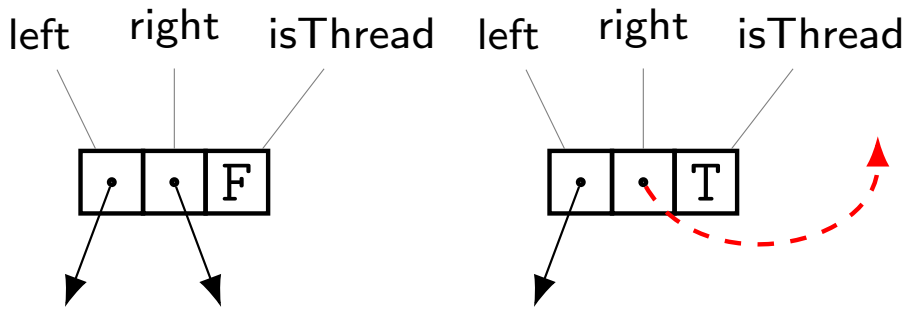
after 3rd visit

Contents

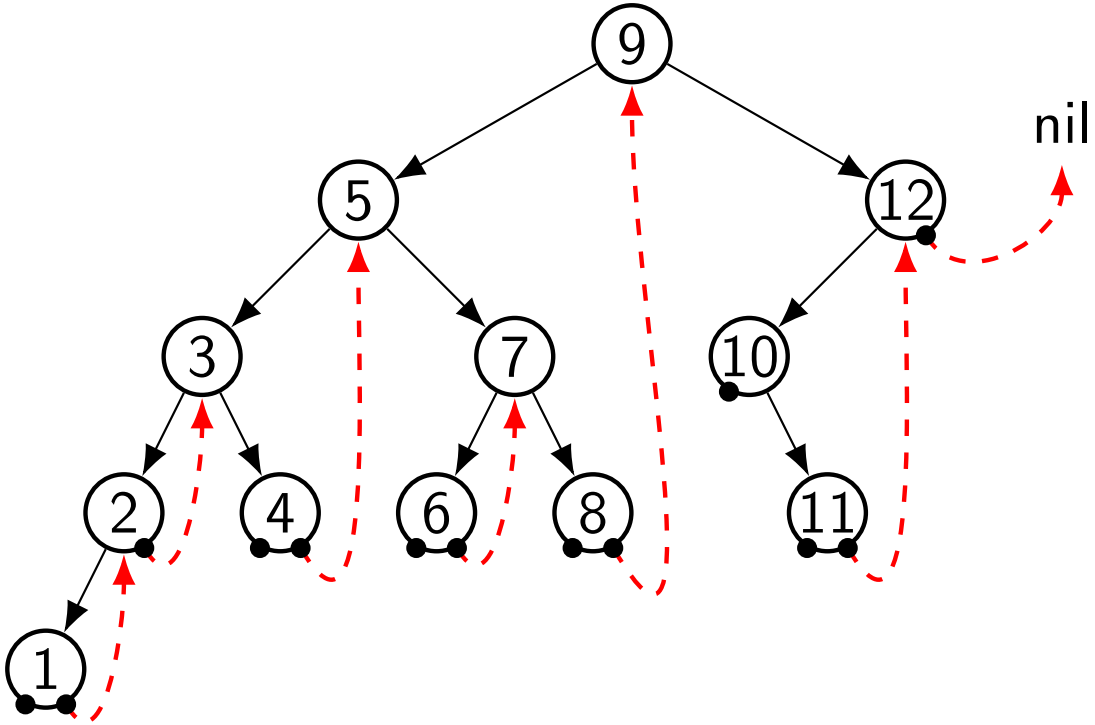
inorder threads

- *threads*:
  replace nil-pointers to indicate *inorder successors*
- can be used to perform stack-less traversal
- need one bit [boolean] per node to mark thread
- adding/deleting nodes: also update threads!

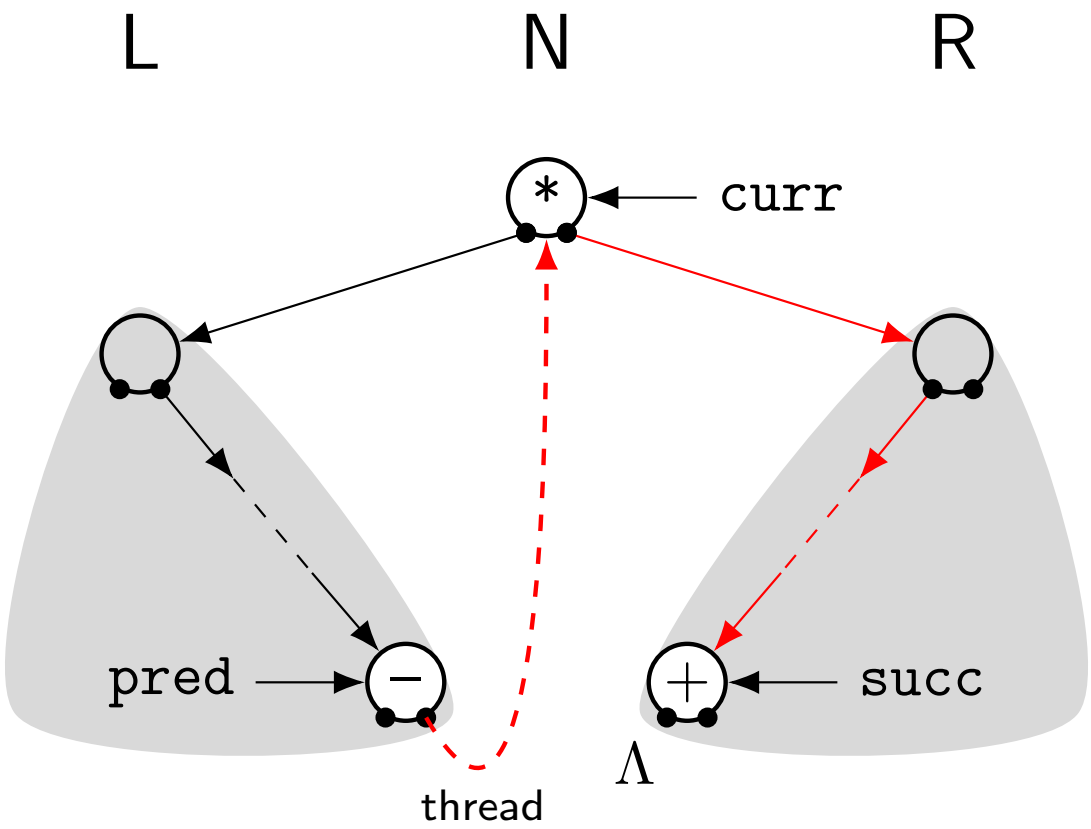
- *Morris-variant*: temporary threads, no extra bit
- structure is disturbed: only a single traversal at a time



inorder threads



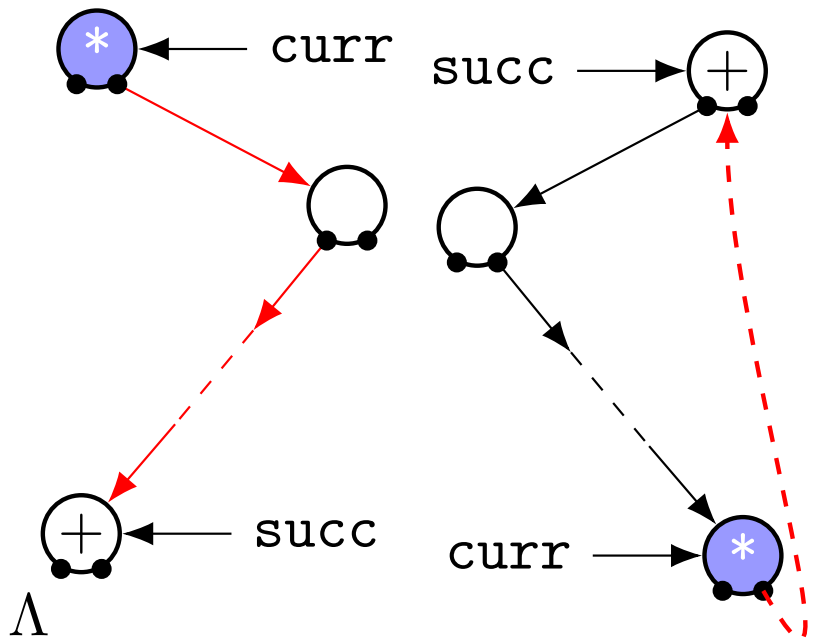
traversal with inorder threads



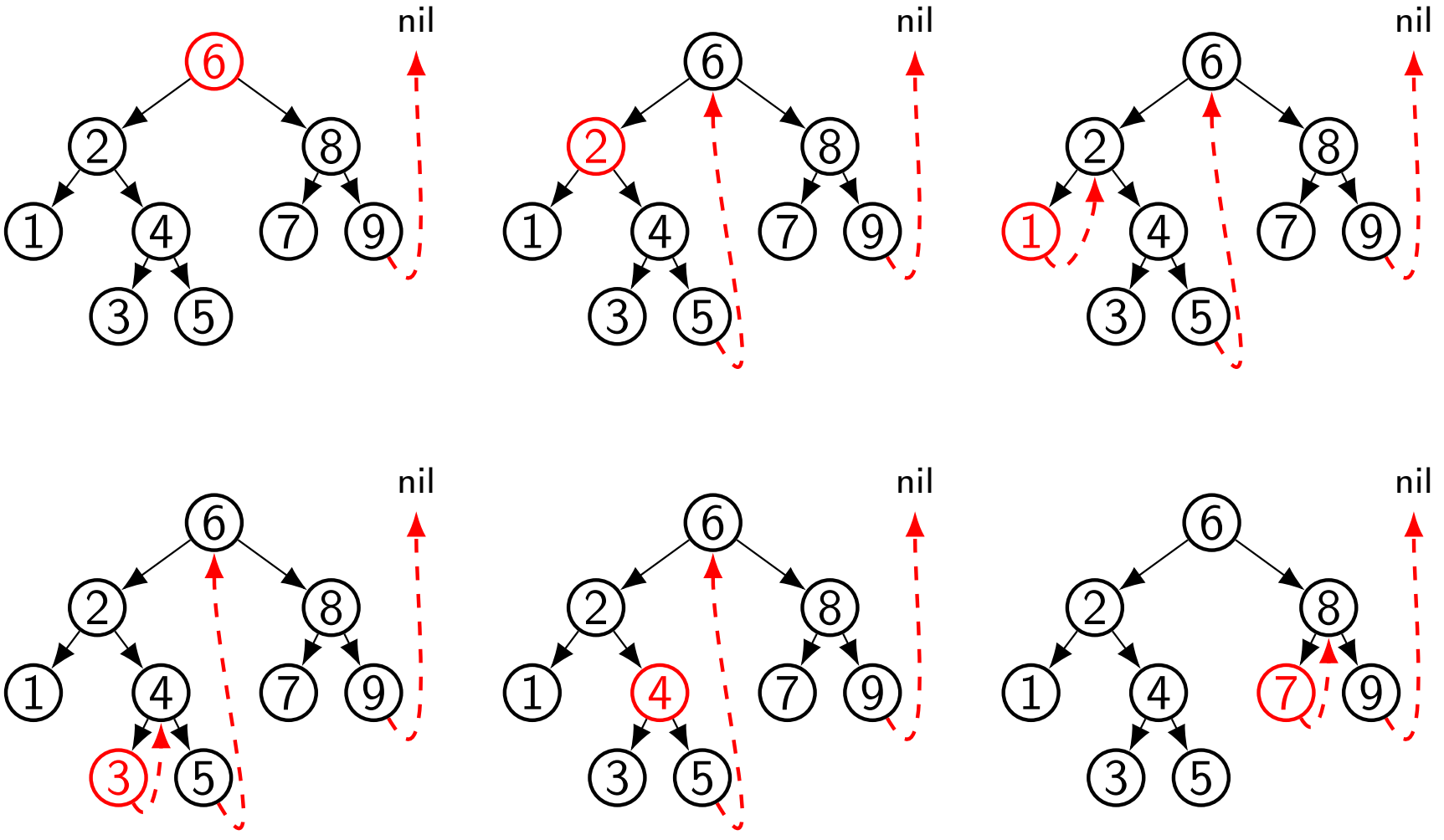
threaded traversal – fixed threads

inorder threads

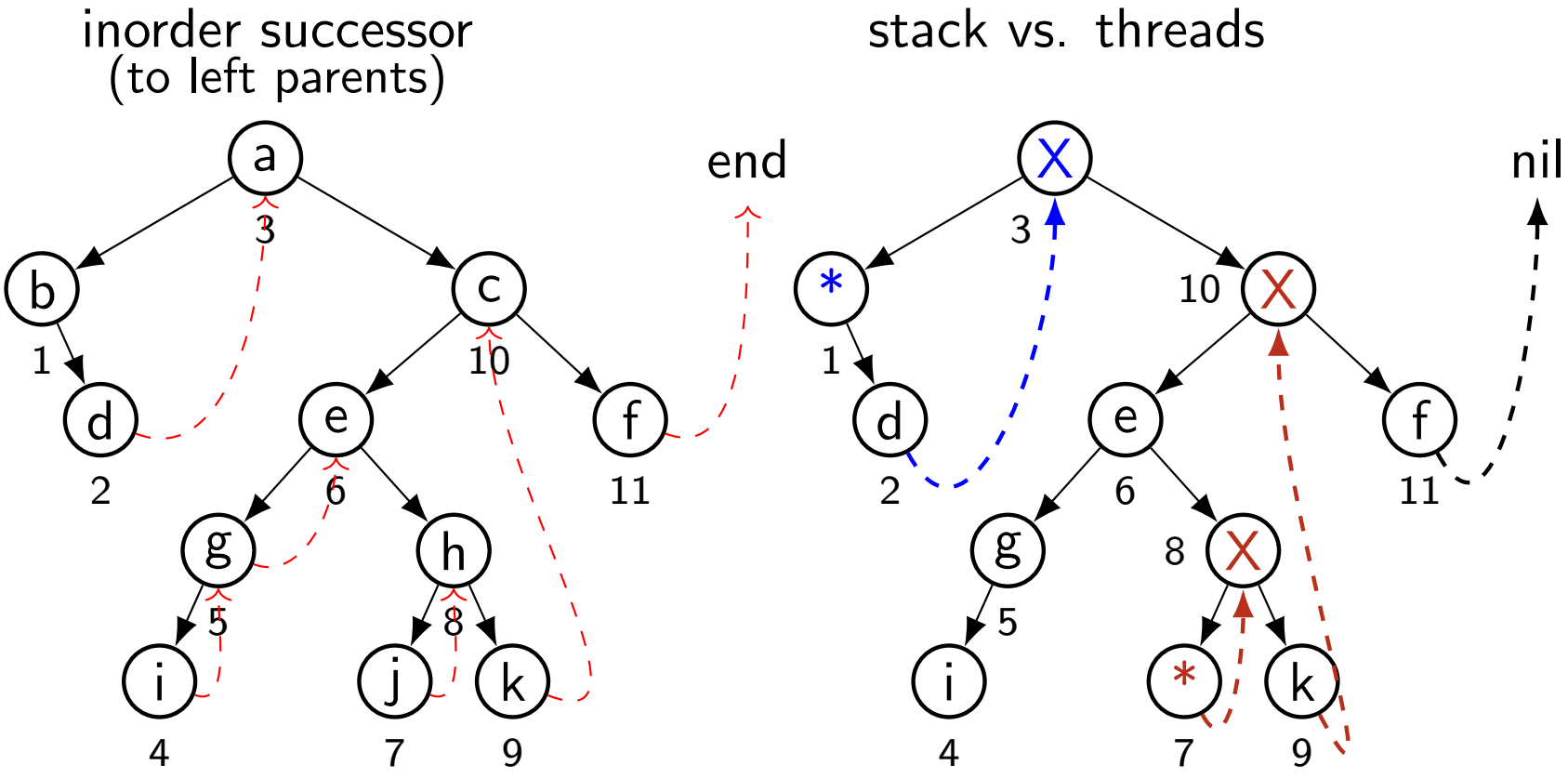
```
    // assuming root != nil
    // to first in inorder
curr = root;
walkLeft( root );
while (curr != nil)
do  inOrderVisit( curr );
    if (curr.isThread)
    then    // follow thread
        curr = curr.right;
    else    // first in subtree
        curr = curr.right;
        walkLeft (curr)
    fi
od

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



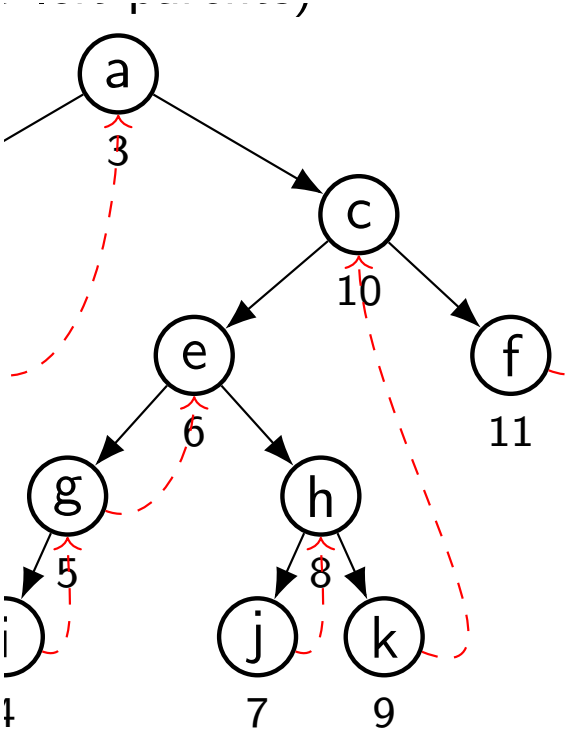
Morris: threads on the fly



Morris: temporary threads form stack



Morris: basics

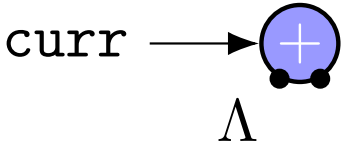


two visits

1. (pre-order)
   arrive from parent
   via child-link (left or right)
   add thread to current node
2. (inorder)
   leave subtree, via thread
   remove thread

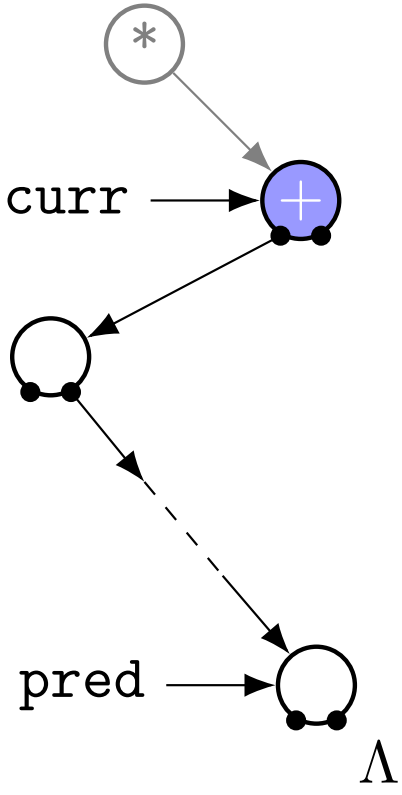
catch: algorithm *does not know threads*
so does not know which visit
but will check *afterwards*!

Morris traversal

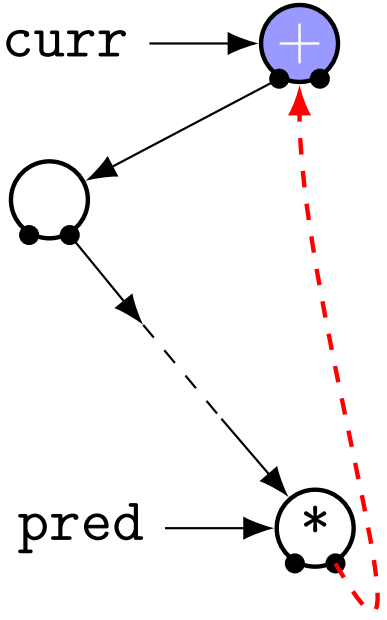


no left subtree:

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



new subtree: *1st visit*
construct thread
go left (at curr)



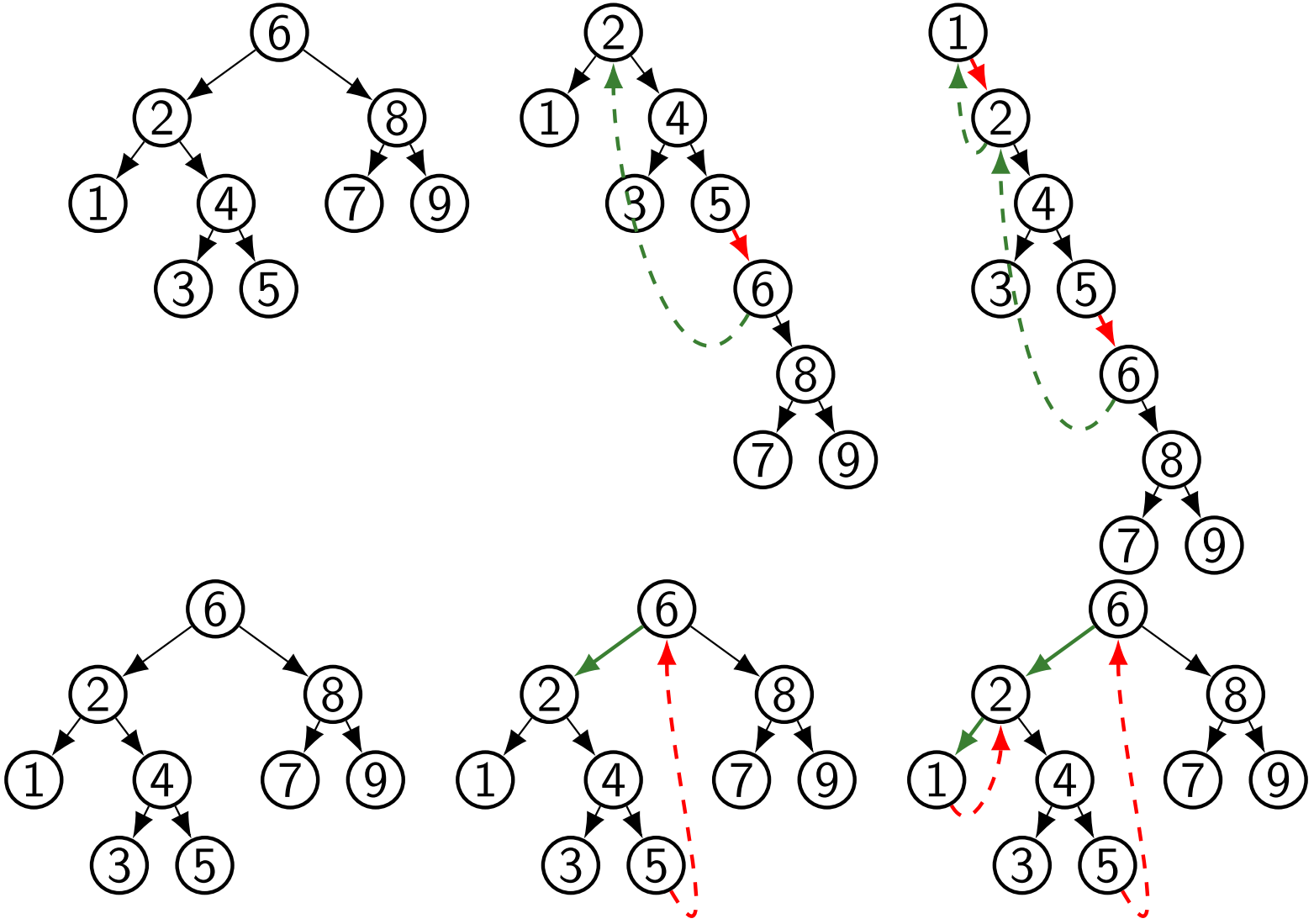
been there: *2nd visit*
delete thread
go right (at curr)

Morris, the algorithm

Morris traversal

```
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 and 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
od
```

alternative view: tree transformation ☒



end.