Trees

Data Structures and Algorithms (094224)

Tutorial 3

Winter 2022/23

Question 1

1. Prove that every tree $T = (V, E)$ is a bipartite graph
2. Show an example of a bipartite graph which is not a forest

Recall:

- An acyclic undirected graph is called a *forest* ⟨יער⟩
- A connected forest is called a *tree* ⟨עץ⟩
- An undirected graph $G = (V, E)$ is called *bipartite* ⟨דו צדדי⟩ if:
  - V can be partitioned into $V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset$, such that $E \subseteq V_1 \times V_2$
- Let $G = (V, E)$ be some (un)directed graph. The distance from $x \in V$ to $y \in V$ = length of a shortest (x, y)-path
  - Shortest path is not necessarily unique
  - Denote distance by $\delta(x, y)$

Solution 1.1

- The proof is constructive
  - We will show how to partition the vertices of V to V_1 and V_2

- Partition V as follows:
  1. $V_1 = \emptyset, V_2 = \emptyset$
  2. Pick an arbitrary start vertex $v \in V$ and place it in V_1
  3. Every $u \in V - \{v\}$ such that $\delta(v, u)$ is even place in V_1
  4. Every $u \in V - \{v\}$ such that $\delta(v, u)$ is odd place in V_2

Solution:

- T is a tree
  - For all $u \in V$ it holds $\delta(u, v) < \infty$
- $V_1 \cup V_2 = V$
  - Every $u \in V$ will be placed in V_1 or V_2
- $V_1 \cap V_2 = \emptyset$
  - For every $u \in V - \{v\}$: $\delta(u, v)$ is either odd or even thus u cannot be in both V_1 and V_2

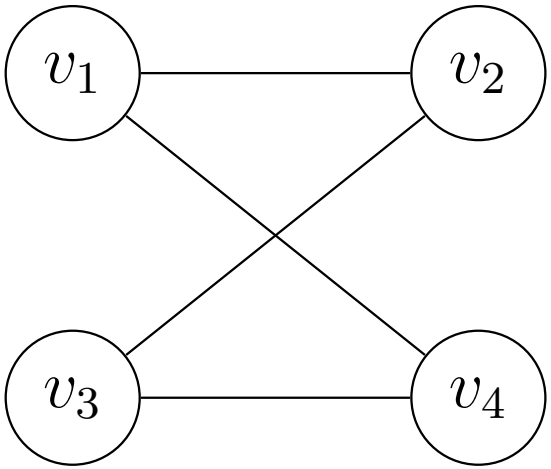
Solution 1.1 — cont.

$E \subseteq V_1 \times V_2$

- Assume by contradiction that there exists an edge $(u, w) \in E$ such that $(u, w) \notin V_1 \times V_2$
  - Either $u, w \in V_1$ or $u, w \in V_2$
- Assume, w.l.o.g, that $u, w \in V_1$, i.e., $\delta(v, u)$ and $\delta(v, w)$ are even
- T is a tree, thus there exists a unique simple path from v to w denoted by $P_w = \langle v, \ldots, w \rangle$
  - According to our assumption the length of the path is even
  - P_w is simple, hence all nodes are distinct
- If P_w contains u
  - $P_w = \langle v, \ldots, u, w \rangle$
    - $P_w = \langle v, \ldots, u, \ldots, w \rangle$ is not possible, there will be two simple (u, w)-paths
  - The (v, u)-subpath of P_w is a simple (v, u)-path of odd length, hence $\delta(v, u)$ is odd and $u \in V_2$ $(\rightarrow\leftarrow)$
- Otherwise, the path $P_v = \langle P_w, u \rangle$ is the unique simple (v, u)-path, hence $\delta(v, u)$ is odd and $u \in V_2$ $(\rightarrow\leftarrow)$

Solution 1.2

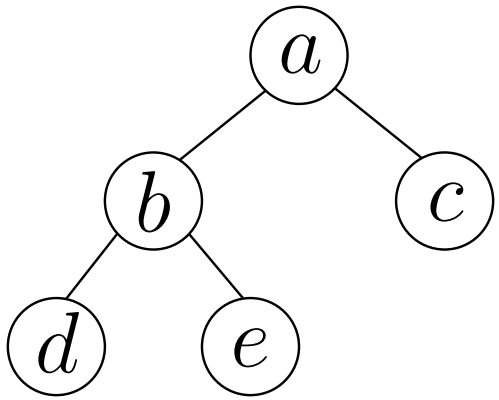
- Let G:



- $V = \{v_1, v_2, v_3, v_4\}$
- $V_1 = \{v_1, v_3\}$, $V_2 = \{v_2, v_4\}$, $E \subseteq V_1 \times V_2$
- The path $\langle v_1, v_2, v_3, v_4, v_1 \rangle$ is a cycle $\Longrightarrow$ G is not a forest

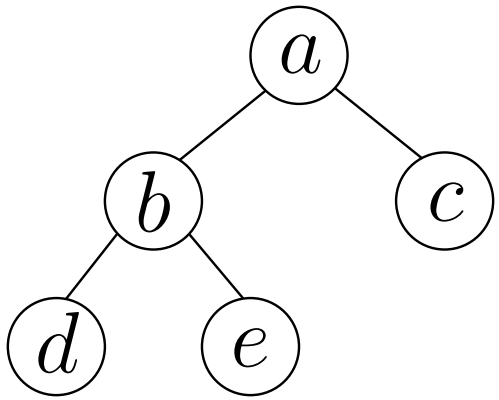
Rooted trees

- A *rooted tree* ⟨עץ מושרש⟩ is a tree with one designated vertex
  - Called the *root* ⟨שורש⟩
- Rooted tree $T = (V, E)$ with root r
- Any vertex on the unique simple path from v to r is called an *ancestor* ⟨אב קדמון⟩ of v
  - v is always an ancestor of v
- If u is an ancestor of v, then v is a *descendant* ⟨צאצא⟩ of u
- The *subtree rooted at v* ⟨תת־עץ מושרש⟩ is the tree induced by descendants of v rooted at v
- If u is the next vertex after v on the unique path from v to r, then u is the *parent* ⟨אב⟩ of v and v is the *child* ⟨בן⟩ of u
  - r is the only vertex in T with no parent



Rooted trees — cont.

- A vertex with no children is a *leaf* ⟨עלה⟩
- A non-leaf is an *internal vertex* ⟨צומת פנימי⟩
- If two vertices share a parent, then they are *siblings* ⟨אחים⟩
- The *degree* ⟨דרגה⟩ of v in T=#children it has
  - Depends on whether we treat T as a graph or a rooted tree
- The *depth* ⟨עומק⟩ of v is the length of the unique path from v to r
- The *height* ⟨גובה⟩ of T is the the maximum depth of any of its leaves
- The *height* ⟨גובה⟩ of v is the height of the subtree rooted at v
- A *level* ⟨רמה⟩ of T consists of all vertices of the same depth
- The rooted tree is *ordered* ⟨מוסדר⟩ if the children of each internal vertex are ordered



Rooted trees — cont.

k-rooted-tree:

- For a constant $k \geq 1$, an rooted tree $T = (V, E)$ is a *k*-rooted-tree if for every $v \in V$ the degree of v is at most k.
- Usually we will consider a *k*-rooted-tree as an ordered rooted tree

Special Case:

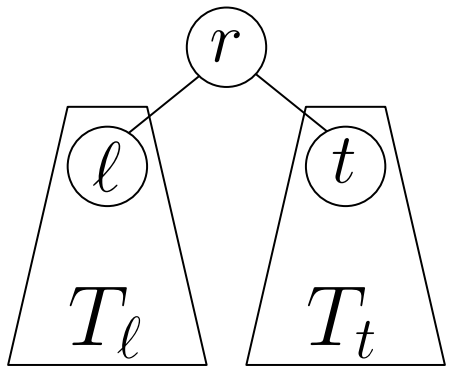
- A 2-rooted-tree is a *binary tree* ⟨עץ בינארי⟩
  - Let $T = (V, E)$ be a binary tree and $v \in V$:
    - If v has two children, then v have a right and left child
    - If node v has a single child, then it is either a left or right child
  - A *complete* ⟨שלם⟩ binary tree:
    - The degree of each internal node is exactly 2; and
    - all leaves are at the same depth

Question 2

Let $T = (V, E)$ be a binary tree with height h. Prove $|V| \leq 2^{h+1} - 1$.

Proof. by induction on h

- Base: for $h = 0$, $|V| = 1 \leq 2^{0+1} - 1 = 1$
- Hypothesis: for every binary tree with height $\leq h - 1$ it holds that $|V| \leq 2^h - 1$
- Step: consider a binary tree T with height $h \geq 1$
  - Denote by ℓ root's left child and by t root's right child
  - Let T_ℓ be the subtree rooted at ℓ and T_t the subtree rooted at t
  - The height of $T_\ell \leq h - 1$
  - The height of $T_t \leq h - 1$
  - By the induction hypothesis $|V_\ell| \leq 2^h - 1$ and $|V_t| \leq 2^h - 1$
  - $|V| = |V_\ell| + |V_t| + 1 \leq 2^h - 1 + 2^h - 1 + 1 = 2^{h+1} - 1$

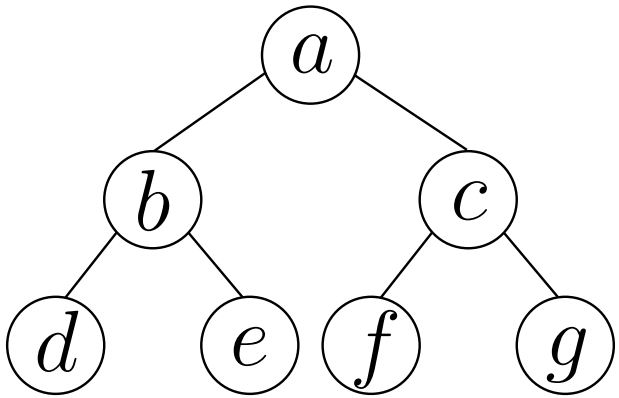


Question 3

Let $T = (V, E)$ be a complete binary tree with ℓ leaves.
Prove: #internal nodes $= \ell - 1$

Proof.

- Let $k = \#$ internal nodes in T
- $|V| = \ell + k$ thus $|E| = \ell + k - 1$
- Every internal node accounts for exactly 2 edges
  - The edges to the node's children
- Thus, $|E| = 2k$
- $\ell + k - 1 = 2k \implies k = \ell - 1$



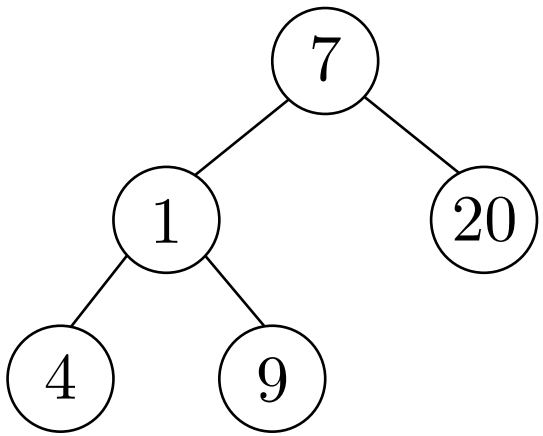
Rooted Trees - representation

- Can we represent a rooted tree with adjacency matrix? adjacency list?
- Graph algorithms time (or space) complexity may depend on the graph representation
- We will show a method for representing a rooted tree in a computer that will come in handy for future algorithms
- Every node is an object
  - For now think of C++ objects
- Objects linked together with pointers represent the rooted tree
- Node object have attributes
  - Usually the nodes have an identifier, a key
- In pseudocode we access an object by its name
  - We access object attribute by [object name].[attribute name]

Rooted Tree - Attributes Printing

Rooted tree attributes can be printed:

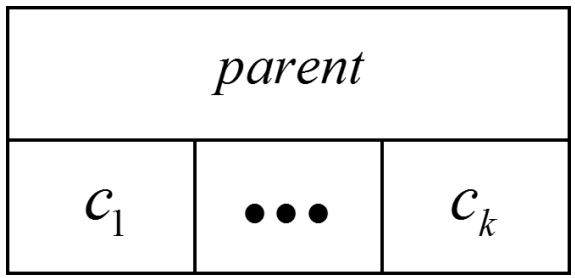
- Preorder – prints the attribute of the root before the attributes of all subtrees
- Postorder – prints the attribute of the root after the attributes of all subtrees
- Inorder (relevant only to binary trees) – prints the attribute of the root of a subtree between printing the attributes of its left subtree and printing those of its right subtree



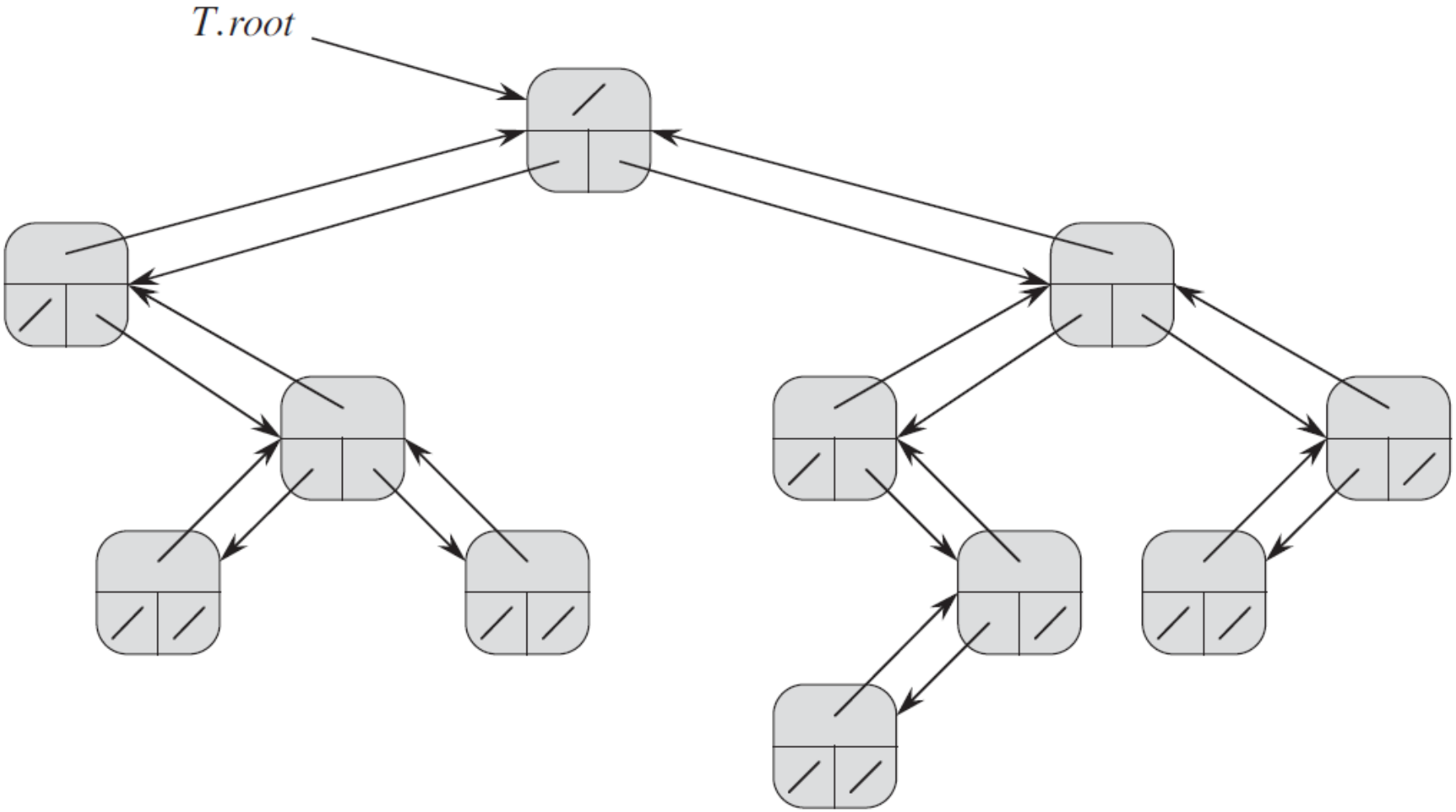
Preorder: 7,1,4,9,20
Postorder: 4,9,1,20,7
Inorder: 4,1,9,7,20

Rooted Trees with Bounded degree representation

- Objects associated with nodes of a k-rooted-tree T with degrees $\leq k$
  - k is a constant. Known at compilation time
- Data structure's attributes
  - *root*= pointer to root of T
    - If $T.root = NIL$, then the tree is empty
- Node attributes (additional):
  - *parent* = a pointer to parent in T
    - $T.root.parent = NIL$
  - c = a length k array of pointers to children in T
    - If node x has no child i, then $x.c[i] = NIL$
    - Assumption: If node x has $\ell \leq k$ children, then their pointers are stored in $c[1], \cdots, c[\ell]$
  - May hold additional attributes



Rooted Trees with Bounded degree representation - cont.



Question 4

Let $T = (V, E)$ be a k-rooted-tree. T is represented with bounded degree rooted tree representation. Each node of T has an attribute named *value*.

1. Implement a recursive algorithm for printing attribute *value* of each node, preorder. The algorithm cannot add attributes or modify existing attributes in the object representing a node. Analyze its time complexity
2. Implement an iterative algorithm for printing attribute *value* of each node, postorder. The algorithm cannot use any auxiliary data structures that depend on the number of nodes in the tree and add attributes or modify existing attributes in the object representing a node. Analyze its time complexity

Solution 4.1

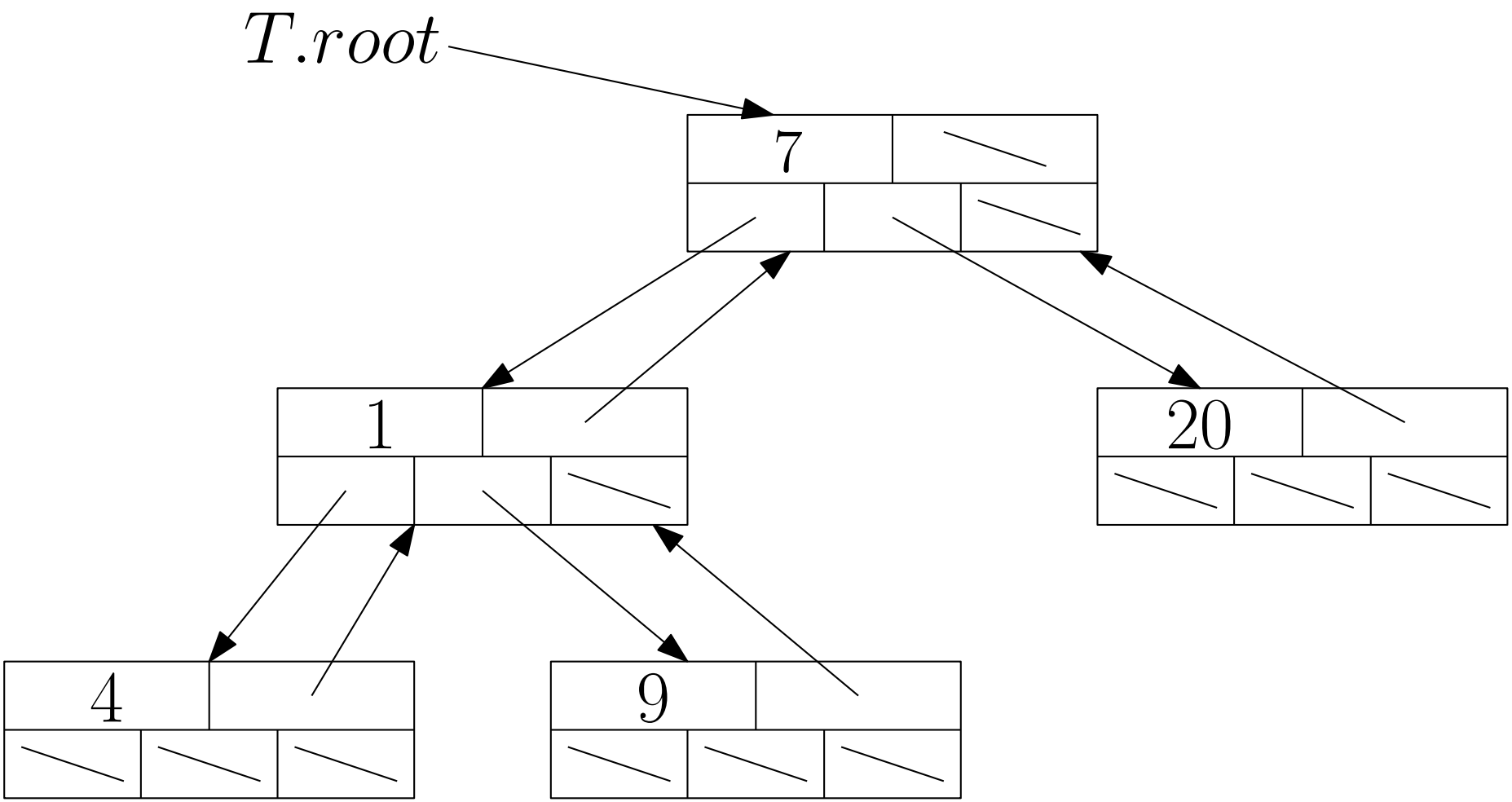
`Preorder_Print_k`(x)

```
1: if x == NIL then
2:     return
3: else
4:     print x.value
5:     for i = 1 to k do                ▷ k – maximum degree of each node
6:         Preorder_Print_k(x.c[i])
```

Complexity Analysis:

- Let n be the number of nodes in the rooted tree with root x.
- $T_{\texttt{Preorder\_Print\_k}}(n) = \Omega(n)$
  - Every node of the rooted tree is visited at least once. Each visit takes $\Omega(1)$ time
- $T_{\texttt{Preorder\_Print\_k}}(n) = O(n)$
  - For every rooted tree with n nodes, each node of the tree enters the for loop k times. Each time it performs $O(1)$ operations. Thus $T_{\texttt{Preorder\_Print\_k}}(n) = O(kn) = O(n)$ since k is a constant

Solution 4.1 — example



`Preorder_Print_k`($T.root$) : 714920

Solution 4.2

- We would like to traverse the tree and print the attribute *value* of the nodes. The challenge is to identify the next child of the current node to visit or return to the node's parent (once all of its children have been visited).
- The main idea of the solution is to hold a variable, *from*, which points, at any given time, to the previous node visited

Auxiliary function:

- `Traverse_To_k`$(x, from)$ – given that *from* holds a pointer to one of x children or parent, the function returns the index of the child to traverse to or *RETURN_TO_PARENT* if none exists

Solution 4.2 — cont

`Traverse_To_k`(x, *from*)

```
1: if from == x.parent then
2:     if x.c[1] == NIL then
3:         return RETURN_TO_PARENT
4:     else
5:         return 1
6: else
7:     for i = 1 to k − 1 do
8:         if x.c[i] == from then
9:             if x.c[i + 1] == NIL then
10:                return RETURN_TO_PARENT
11:            else
12:                return i + 1
13:    return RETURN_TO_PARENT
```

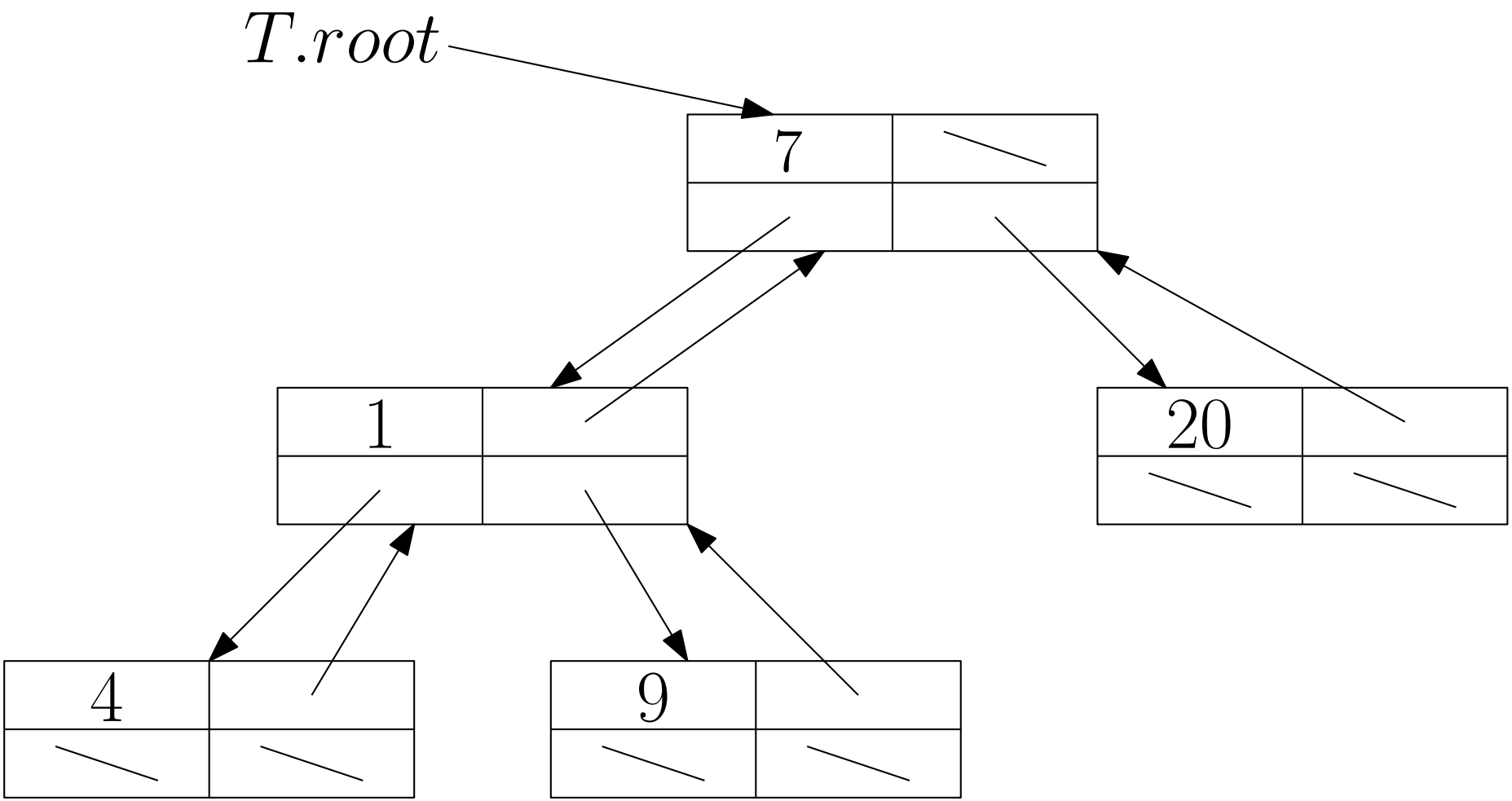
Complexity Analysis:

- $T_{\texttt{Traverse\_To\_k}}(x, \mathit{from}) = \Theta(1)$ since k is a constant

Solution 4.2 — cont

```
Postorder_Print_k(T)
 1: from = NIL
 2: x = T.root
 3: while x ≠ NIL do
 4:     to = Traverse_To_k(x, from)
 5:     from = x
 6:     if to == RETURN_TO_PARENT then
 7:         print x.value
 8:         x = x.parent
 9:     else
10:         x = x.c[to]
```

Solution 4.2 — example



`Postorder_Print_k`(T) : 4921207

Solution 4.2 — cont

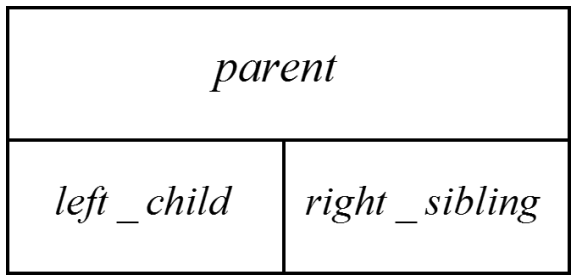
Complexity Analysis:

- Let $n = |T|$
- Auxiliary function takes $\Theta(1)$ time
- $T_{\texttt{Postorder\_Print\_k}}(n) = \Omega(n)$
  - Every node of the rooted tree is visited at least once. Each visit takes $\Omega(1)$ time
- $T_{\texttt{Postorder\_Print\_k}}(n) = O(n)$
  - Each iteration of the *while* loop takes $O(1)$ time. For every rooted tree with n nodes we visit each node at most $k+1$ times thus $T_{\texttt{Postorder\_Print\_k}}(n) = O((k+1)n) = O(n)$ since k is a constant

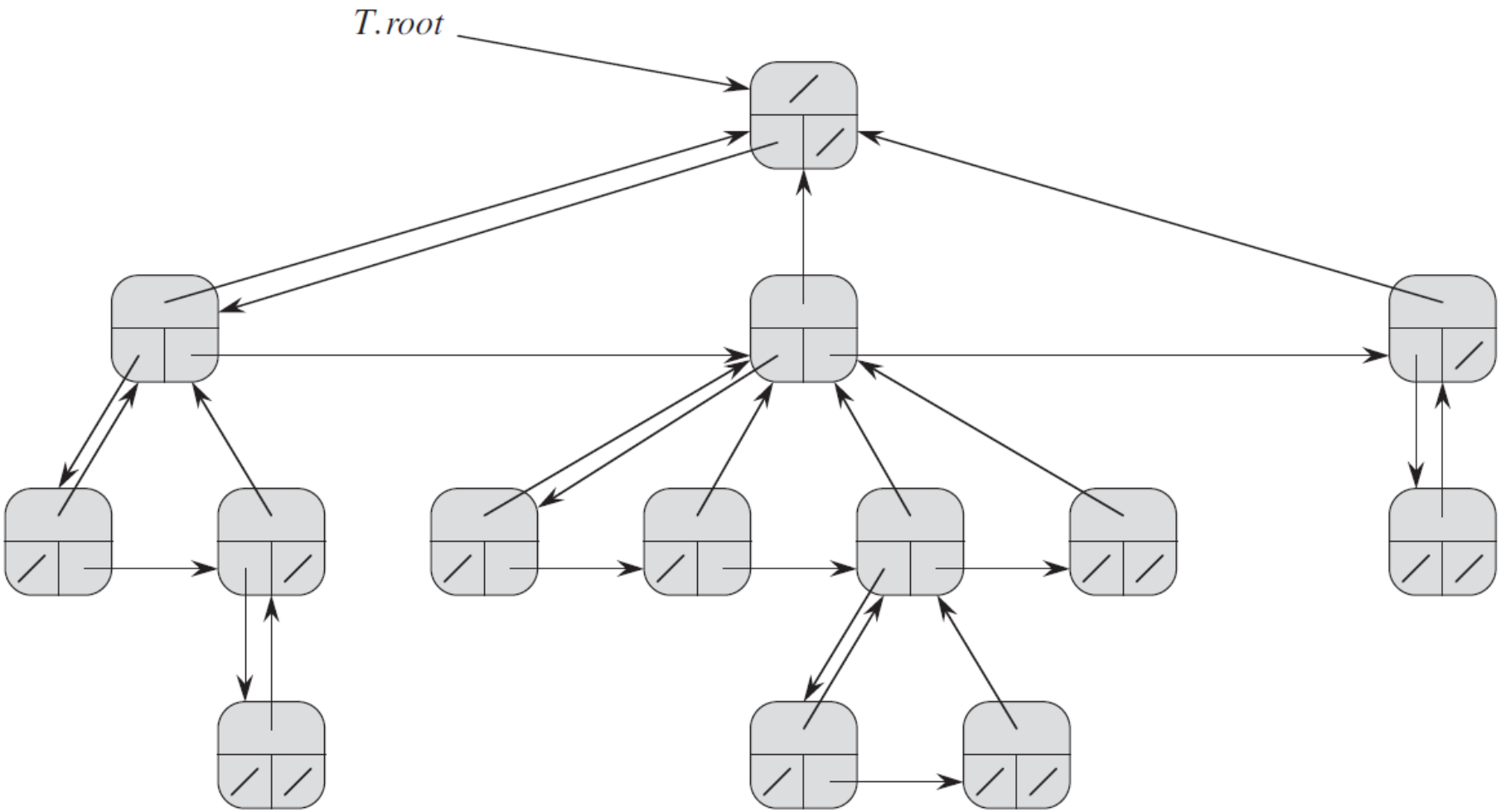
Rooted Trees with Unbounded degree representation

left_child, *right_sibling* representation.

- Objects associated with nodes of a rooted tree T
- Data structure's attribute:
  - *root* = pointer to root of T
    - If $T.root = NIL$, then tree is empty
- Node attributes:
  - *parent* = pointer to parent in T
    - $T.root.parent = NIL$
  - *left_child* = pointer to left most child in T
  - *right_sibling* = pointer to right sibling in T
    - If node x has no children, then $x.left\_child = NIL$
    - If node x has no right sibling then $x.right\_sibling = NIL$
  - May hold additional attributes



Rooted Trees with Unbounded degree representation - cont.



Question 5

Let $T = (V, E)$ be a rooted tree. T is represented with *left_child*, *right_sibling* representation. Each node of T has an attribute named *value*.

1. Implement an iterative algorithm for printing attribute *value* of each node. The algorithm cannot use any auxiliary data structures that depend on the number of nodes in the tree and add attributes or modify existing attributes in the object representing a node. Analyze its time complexity
2. Implement a recursive algorithm for calculating the number of nodes in T. The algorithm cannot add attributes or modify existing attributes in the object representing a node. Analyze its time complexity

Solution 5.1

- As in the iterative algorithm presented for postorder printing of bounded degree rooted trees representation, we should identify the node which we came from

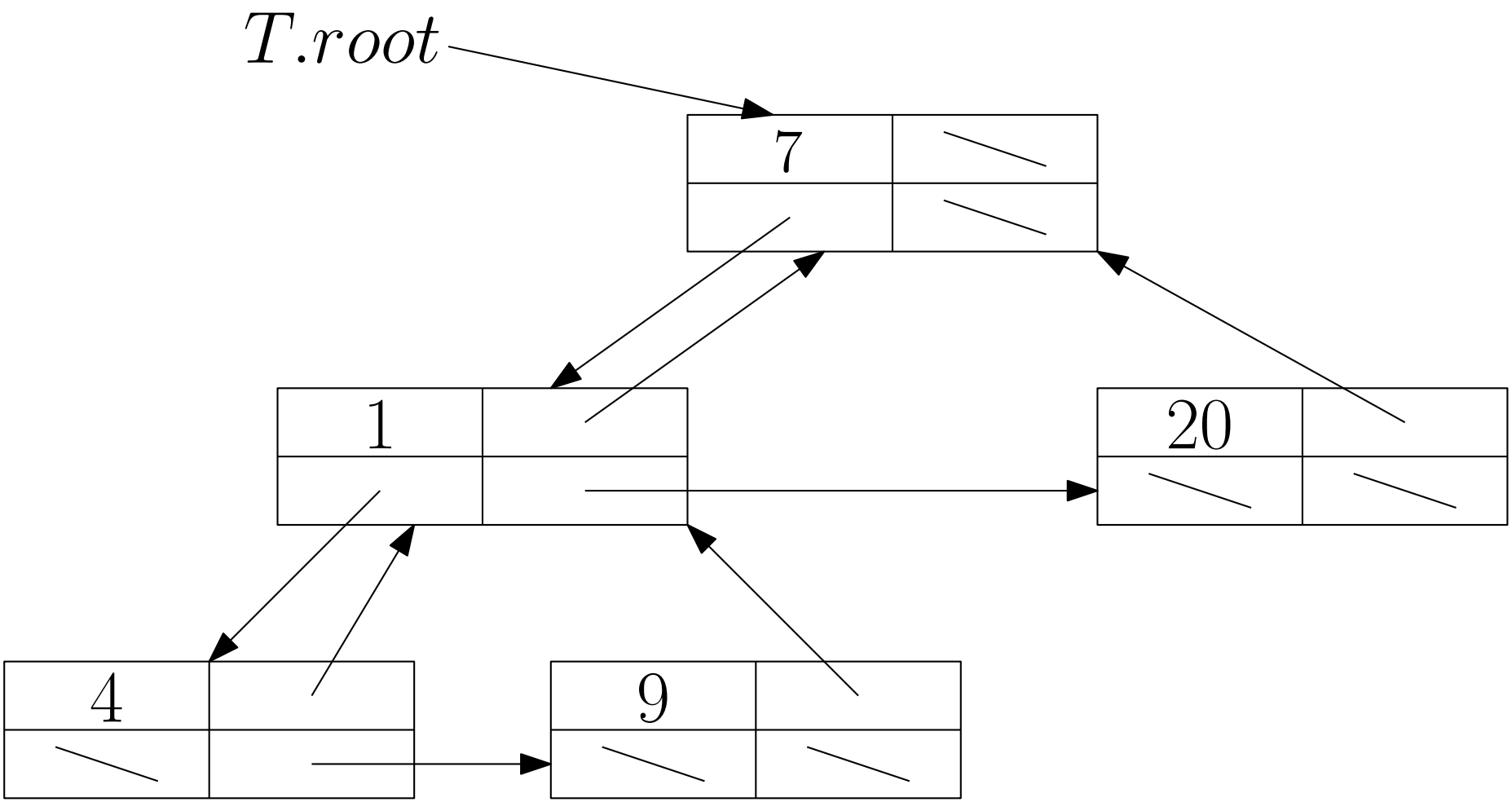
Definitions:

- $from = CHILD$ - came from child
- $from = PARENT\_SIBLING$ - came from parent or left sibling

Solution 5.1 — cont.

```
Print(T)
 1: from = PARENT_SIBLING, x = T.root
 2: while x ≠ NIL do
 3:     if from == PARENT_SIBLING then
 4:         print x.value
 5:         if x.left_child ≠ NIL then
 6:             x = x.left_child
 7:         else
 8:             if x.right_sibling ≠ NIL then
 9:                 x = x.right_sibling
10:             else
11:                 from = CHILD
12:                 x = x.parent
13:     else
14:         if x.right_sibling ≠ NIL then
15:             from = PARENT_SIBLING
16:             x = x.right_sibling
17:         else
18:             x = x.parent
```

Solution 5.1 — cont.



`Print`(T) : 714920 (preorder)

Solution 5.1 — cont.

Complexity Analysis:

- Let $n = |T|$
- $T_{\texttt{Print}}(n) = \Omega(n)$
  - Every node of the rooted tree is visited at least once. Each visit takes $\Omega(1)$ time
- $T_{\texttt{Print}}(n) = O(n)$
  - Each iteration of the *while* loop takes $O(1)$ time. For every rooted tree with n nodes we visit each node at most 2 times thus $T_{\texttt{Print}}(n) = O(n)$

Solution 5.2

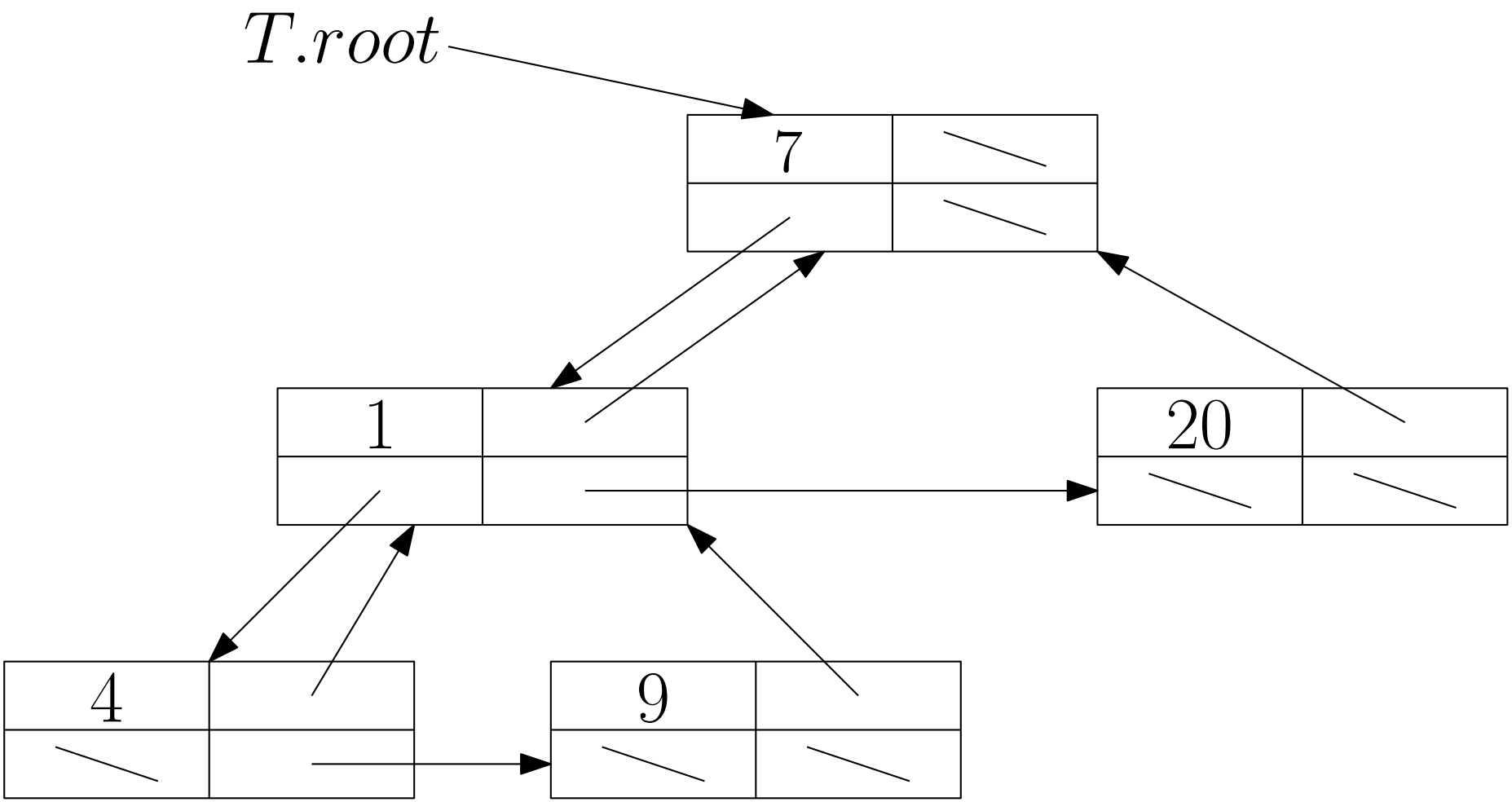
`Count`(x)

1: **if** $x == NIL$ **then**
2: return 0
3: return `Count`($x.left\_child$) + `Count`($x.right\_sibling$) + 1

Complexity Analysis:

- Let n be the number of nodes in the rooted tree with root x
- $T_{\texttt{Count}}(n) = \Omega(n)$
  - Every node of the rooted tree is visited at least once. Each visit takes $\Omega(1)$ time
- $T_{\texttt{Count}}(n) = O(n)$
  - For every rooted tree, each node of the tree is visited 2 times. Each visit takes $O(1)$ time, thus $T_{\texttt{Count}}(n) = \Theta(n)$

Solution 5.2 — example



$\texttt{Count}(T.root) : 5$

Question 6

Let $T = (V, E)$ be a k-rooted tree. The tree T is represented with bounded degree rooted tree representation. Each node of T has an attribute named *height*. Implement an $O(n)$ recursive algorithm that updates the attribute *height* of each node to the node's height in T. The algorithm cannot add attributes or modify existing attributes in the object representing a node except the attribute *height*.

Solution:

- By definition, the height of a node is the maximum height of its children plus 1

Solution

`Calc_Height_k`(x)

```
1:  if x == NIL then
2:      return -1
3:  else
4:      max_height = 0
5:      for i = 1 to k do
6:          cur_height = Calc_Height_k(x.c[i]) + 1
7:          if max_height < cur_height then
8:              max_height = cur_height
9:  x.height = max_height
10: return x.height
```

Complexity Analysis:

- For every rooted tree with n nodes, each node of the tree enters the for loop k times. Each time it performs $O(1)$ operations. Thus $T_{\texttt{Calc\_Height\_k}}(n) = O(n)$

Question 7

Let $T = (V, E)$ be a rooted tree. The tree T is represented with *left_child*, *right_sibling* representation. Implement an $O(n)$ iterative algorithm that returns the number of internal nodes that all of their children are leaves. The algorithm cannot use any auxiliary data structures that depend on the number of nodes in the tree and add attributes or modify existing attributes in the object representing a node.

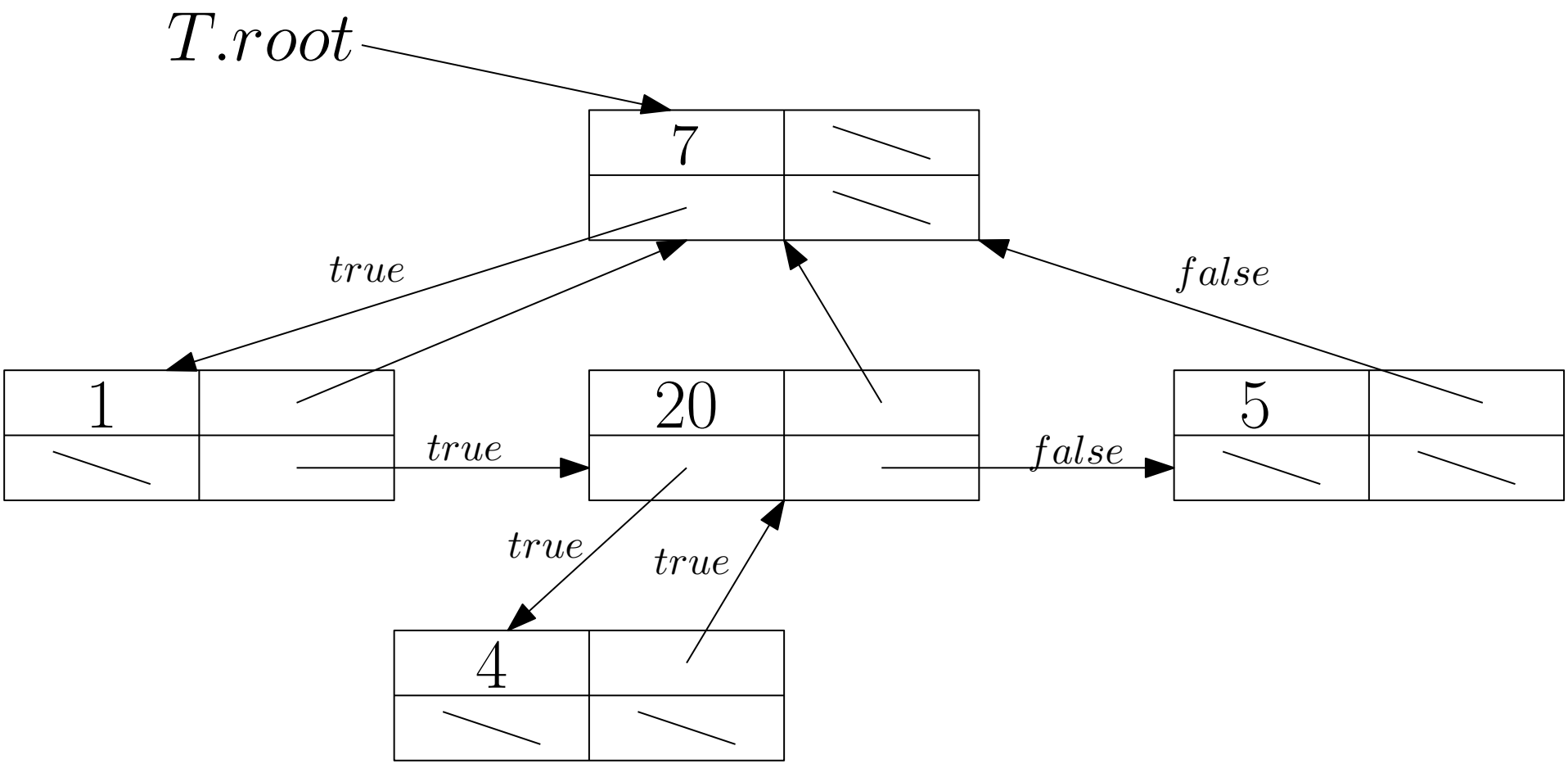
Solution:

- Basic concept – if a node is a leaf, then $x.left\_child = NIL$. If we return to a node from a child, then it is not a leaf
- Use a boolean variable *all_leaves* to notify node x if some left sibling is a leaf
- If x is also a leaf, than the value of *all_leaves* doesn't change
- If x is not a leaf, than notify it using $all\_leaves = false$
  - To right sibling that some left sibling is not a leaf; or
  - To parent that not all of his children are leaves

Solution

```
Count_All_Leaves(T)
 1: from = PARENT_SIBLING, x = T.root, counter = 0, all_leaves =
    True
 2: while x ≠ NIL do
 3:     if from == PARENT_SIBLING then
 4:         if x.left_child ≠ NIL then
 5:             x = x.left_child
 6:             all_leaves = True
 7:         else
 8:             if x.right_sibling ≠ NIL then
 9:                 x = x.right_sibling
10:             else
11:                 from = CHILD
12:                 x = x.parent
13:     else
14:         if all_leaves == True then
15:             counter = counter + 1
16:         all_leaves = False
17:         if x.right_sibling ≠ NIL then
18:             from = PARENT_SIBLING
19:             x = x.right_sibling
20:         else
21:             x = x.parent
22: return counter
```

Solution 7 — example



`Count_All_Leaves`$(T) : 1$