智能软件与工程学院
School of Intelligent Software and Engineering

树
Trees

钮鑫涛
Nanjing University

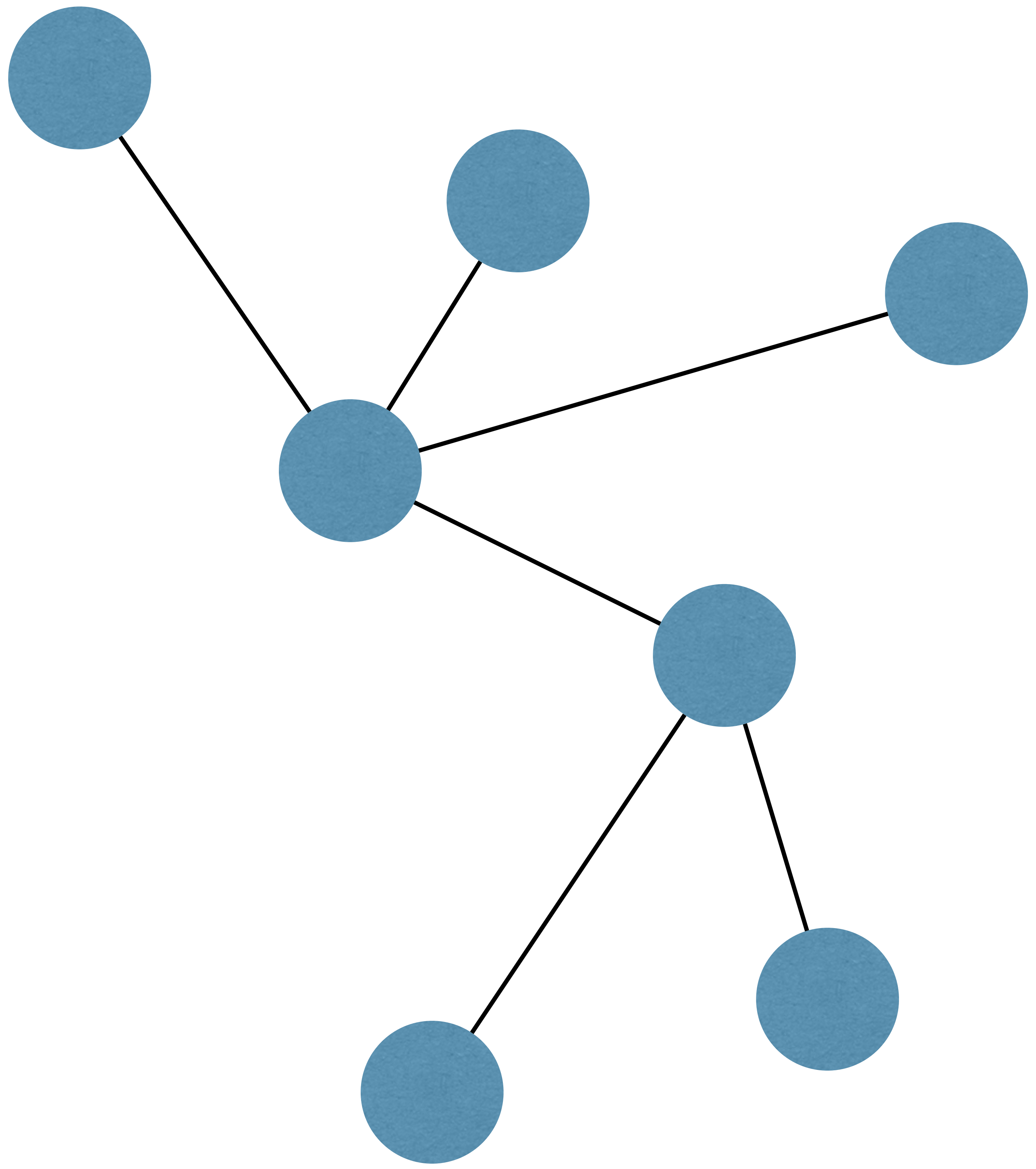
2024 Fall

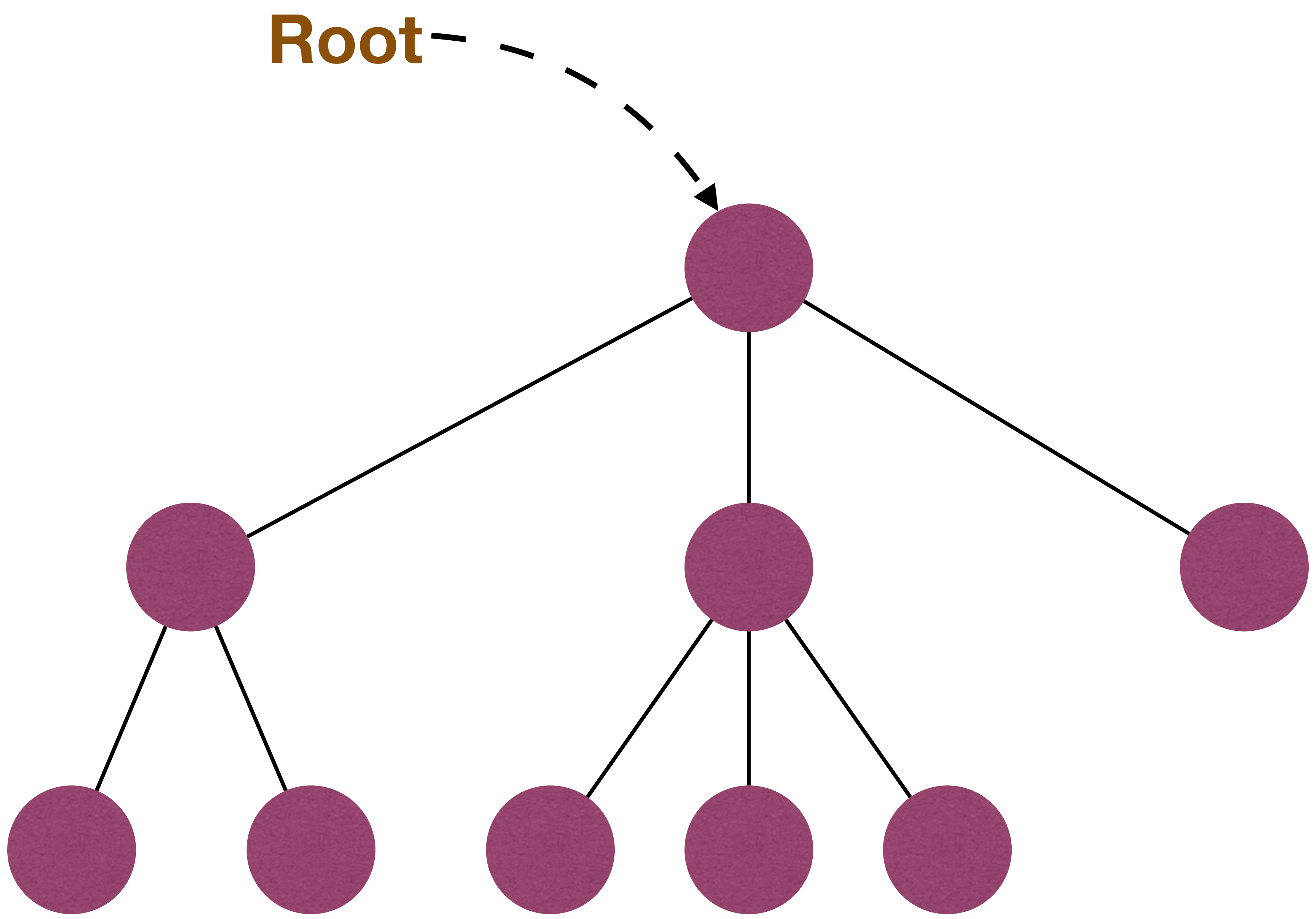
The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

Trees

A tree is a connected, acyclic undirected graph.

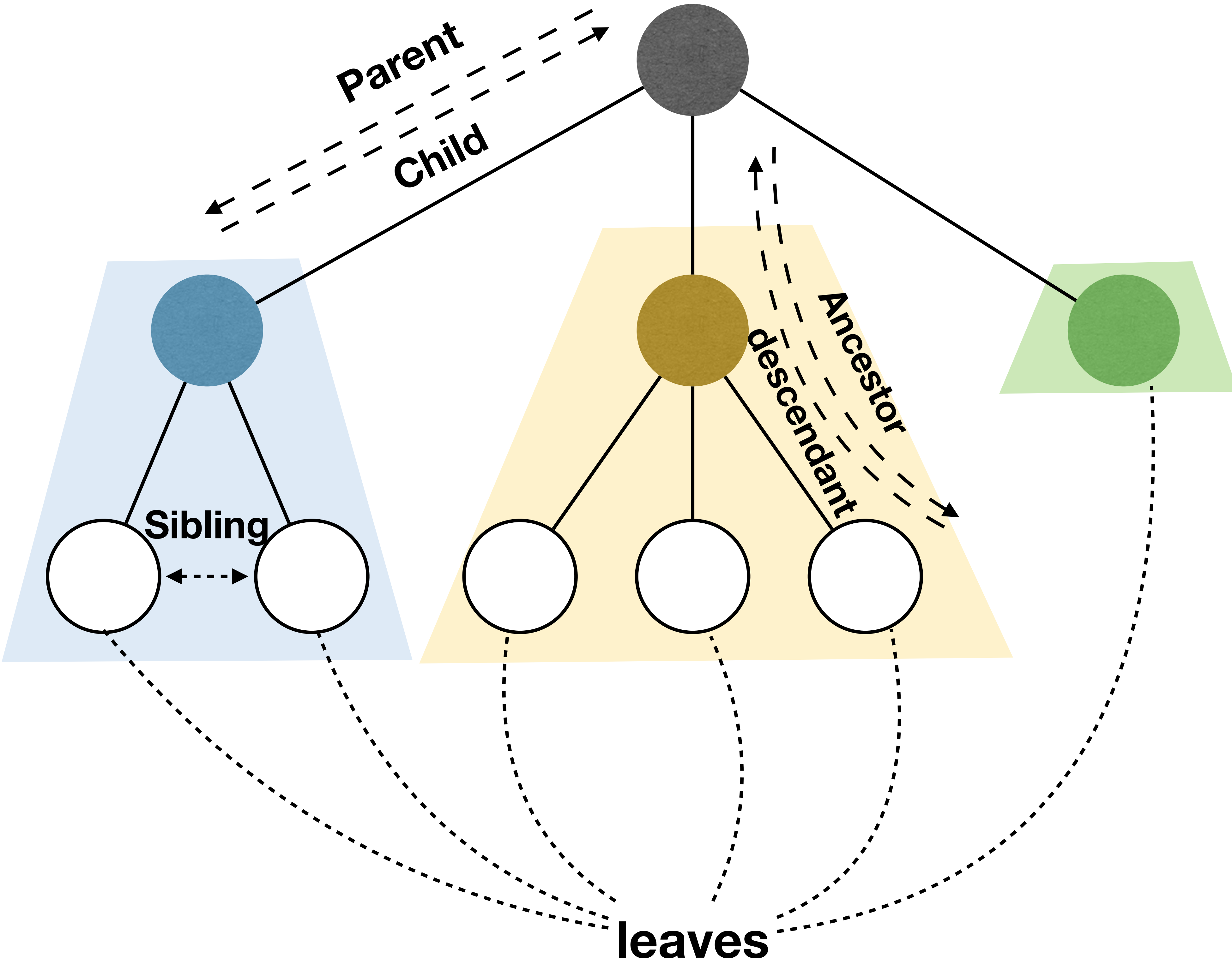
- In CS, we often study **rooted** trees





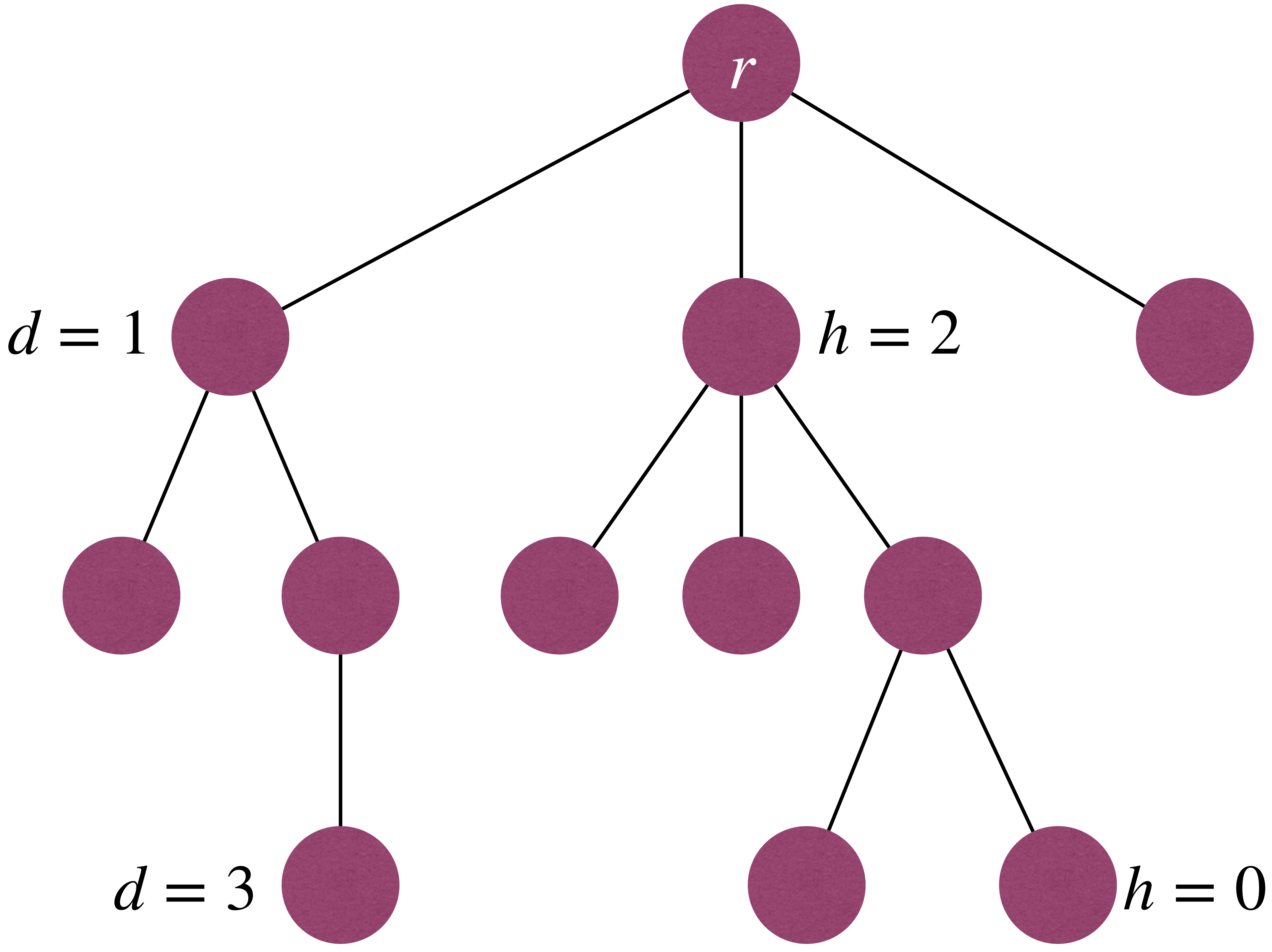
Recursive definition of trees

- A tree is either empty, or has a root r that connects to the roots of zero or more non-empty (sub)trees.
  - Root of each subtree is a **child** of r, and r is the **parent** of each subtree's root.
  - Nodes with no children are **leaves**.
  - Nodes with same parent are **siblings**.
  - If a node v is on the path from r to u, then v is an **ancestor** of u, and u is a **descendant** of v.



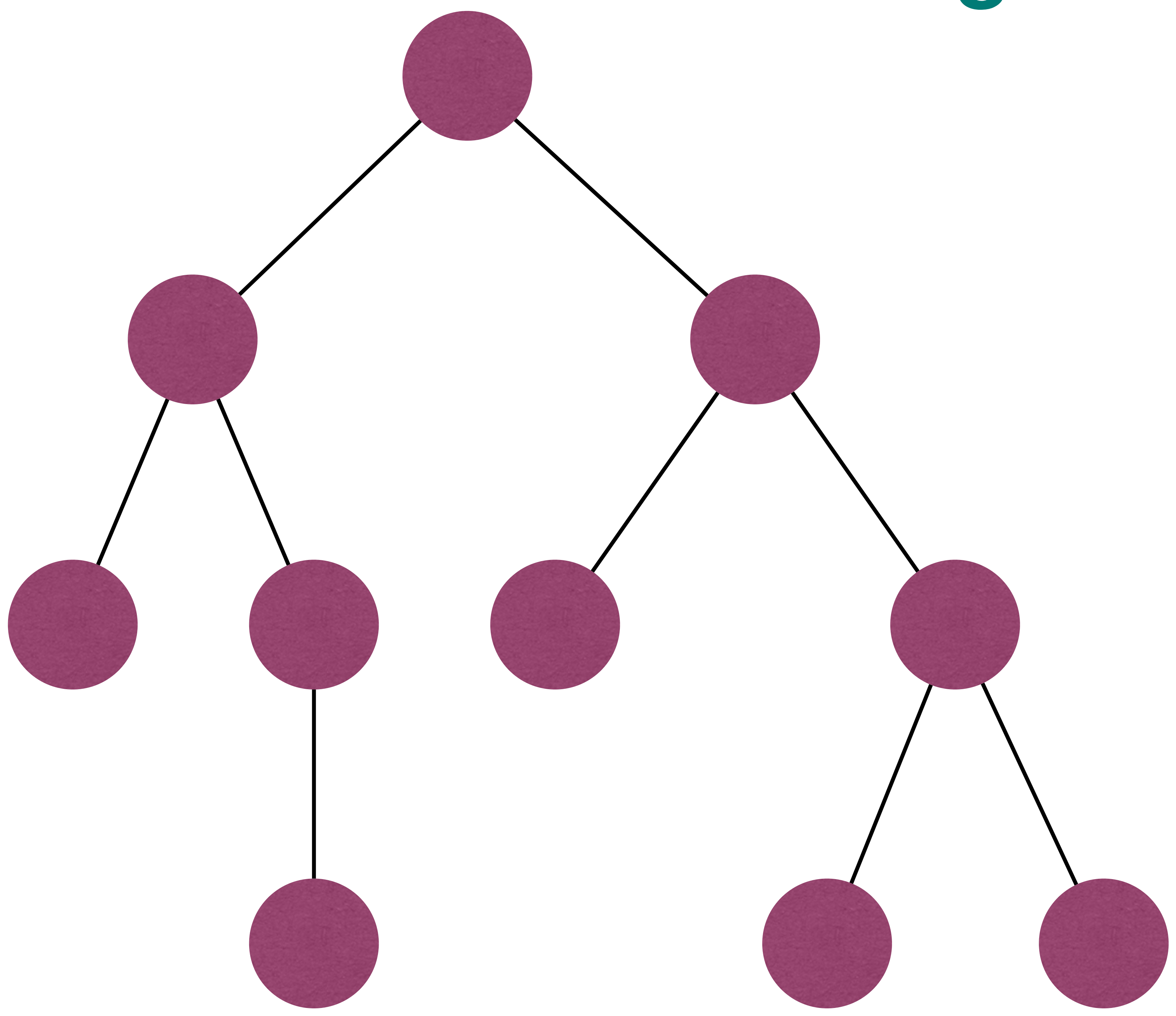
More terminology on Trees

- The **depth** of a node u is the length of the path from u to the root r.
- The **height** of a node u is the length of the longest path from u to one of its **descendants**.
    - Height of a leaf node is **zero**.
    - Height of a non-leaf node is the max **height** of its children plus **one**.



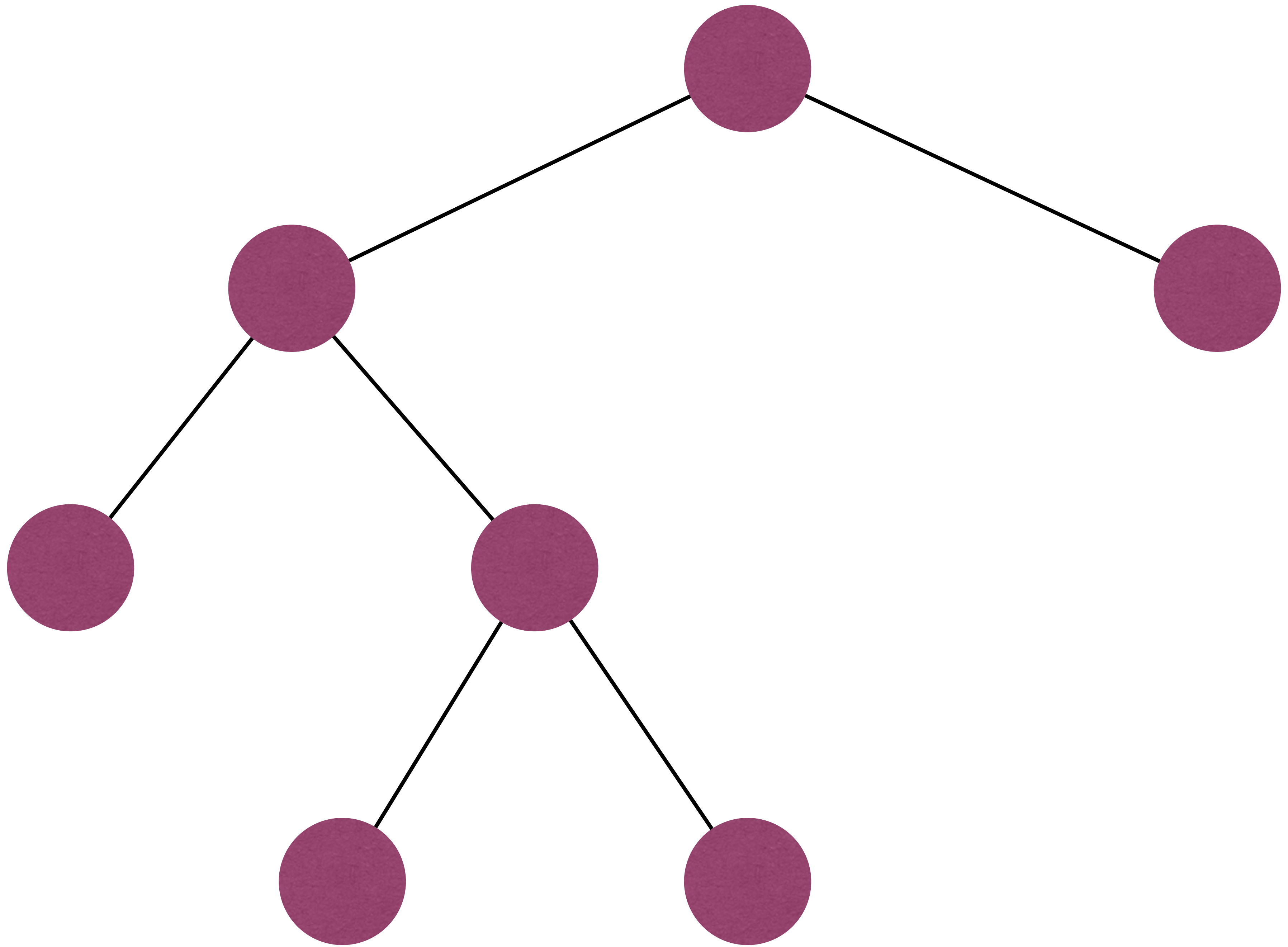
Binary Trees

- A **binary tree (二叉树)** is a tree in which each node has at most two children.
  - Often call these children as **left child** and **right child**.



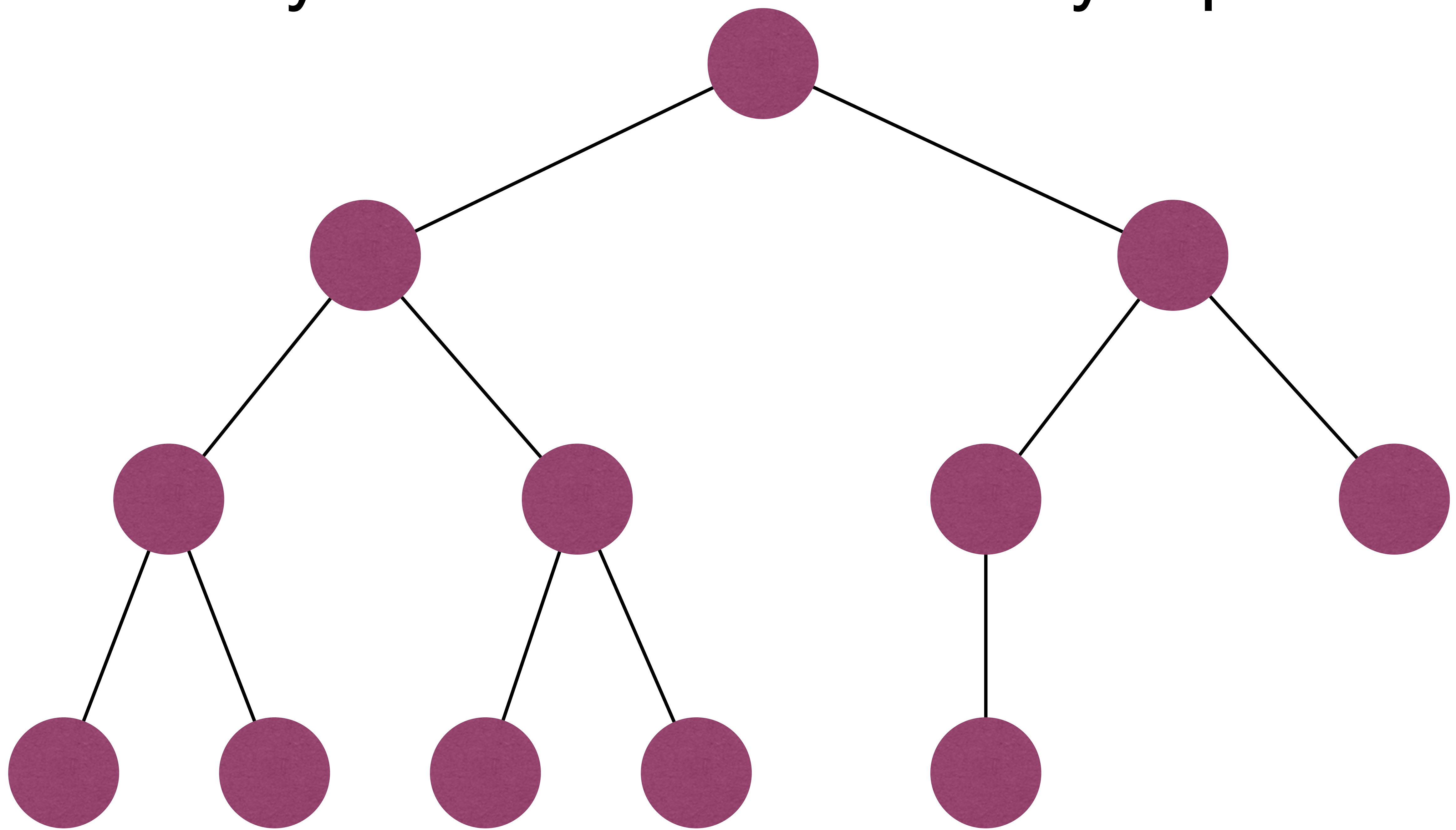
Full Binary Trees

- A **full binary tree (满二叉树)** is a binary tree where each node has either zero or two children.
    - A full binary tree is either a single node, or a tree in which the two subtrees of the root are full binary trees.



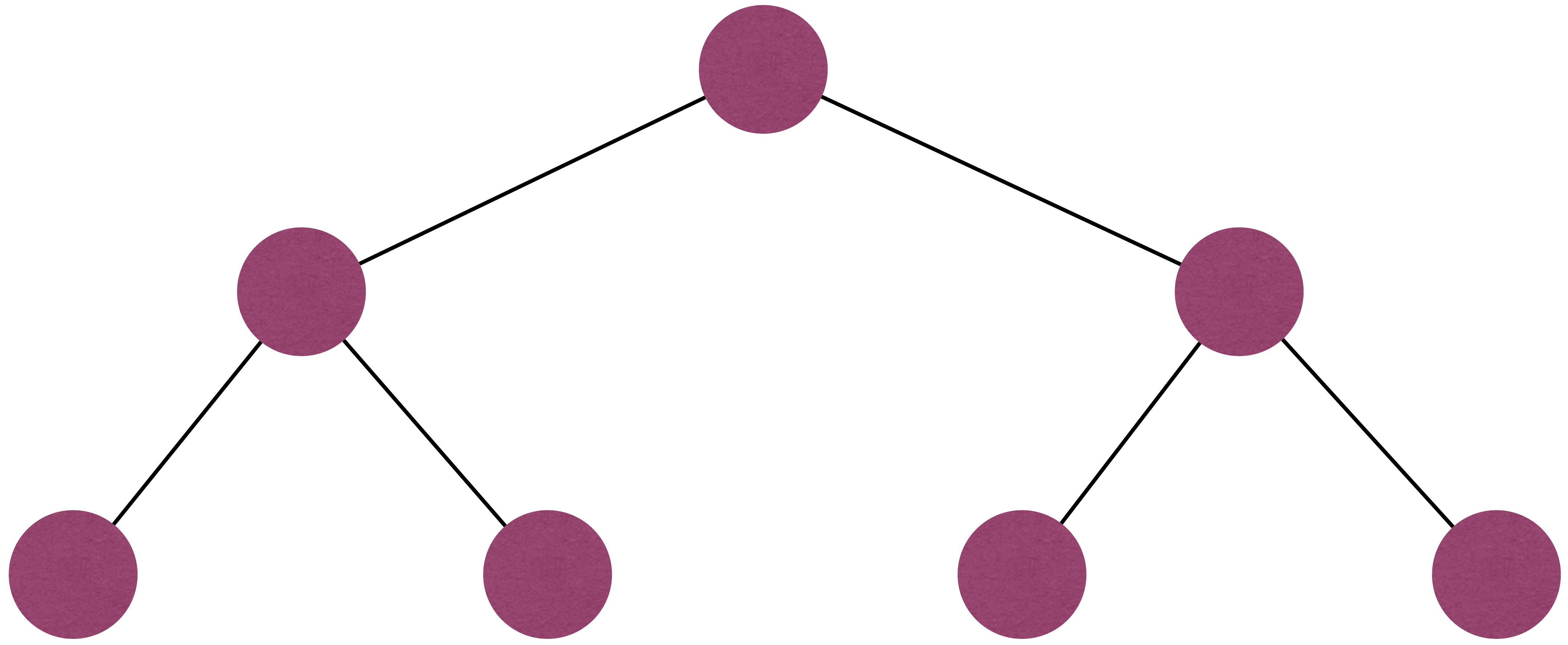
Complete Binary Trees

- A **complete binary tree (完全二叉树)** is a binary tree where every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
  - A complete binary tree can be efficiently represented using an array.



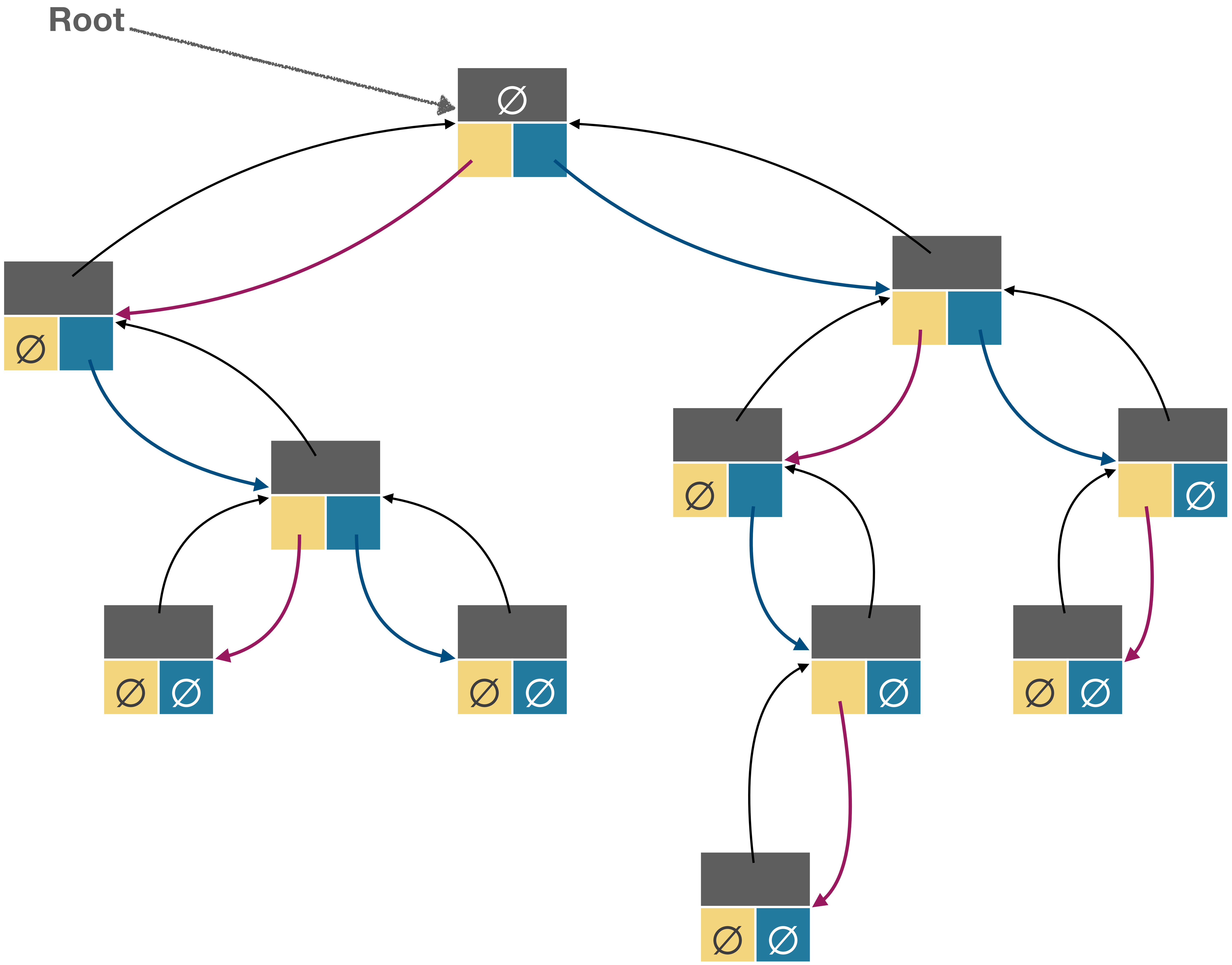
Perfect binary tree

- A **perfect binary tree (完美二叉树)** is a binary tree where all non-leaf nodes have two children and **all leaves have same depth**.



Representing Binary Trees

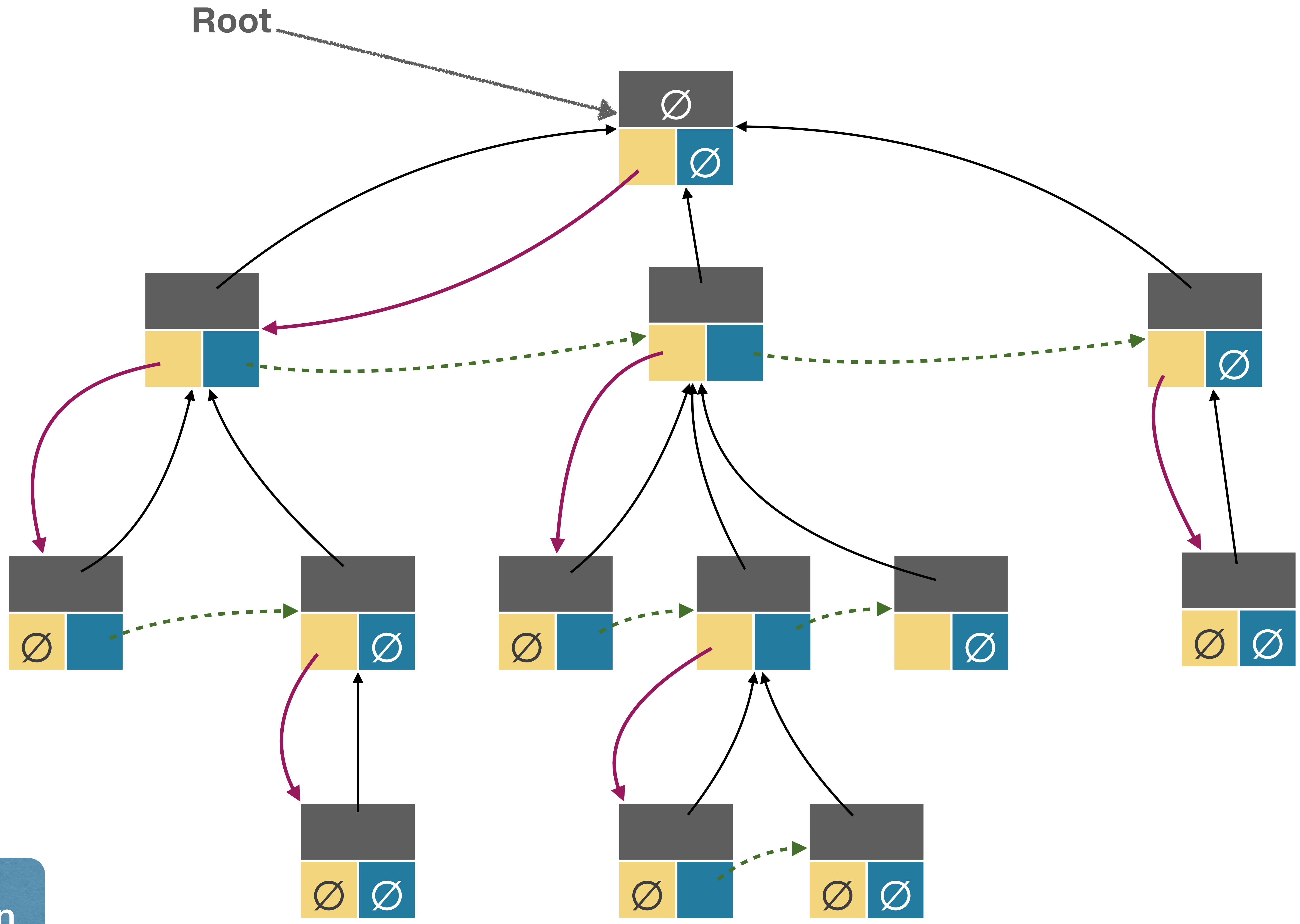
```
class Node {
    Data data
    Node paraent
    Node left
    Node right
}
```



What if nodes have more children?

Representing Binary Trees

```
class Node {
    Data data
    Node paraent
    Node firstChild
    Node nextSibling
}
```



Left-child, right-sibling representation.

Tree Traversals

- Suppose we want to visit all nodes of a tree
  - Recall the recursive definition of trees: a tree is either empty, or has a root connecting to the roots of zero or more non-empty subtrees.
- It is natural to visit the nodes in a tree recursively, but in what order?
  - **Preorder traversal (先序遍历)**: given a tree with root r, first visit r, then visit subtrees rooted at r's children, using preorder traversal.
  - **Postorder traversal (后序遍历)**: given a tree with root r, first visit subtrees rooted at r's children using postorder traversal, then visit r.
  - **Inorder traversal (中序遍历)**: given a ***binary*** tree with root r, first visit subtree rooted at $r.left$, then visit r, finally visit subtree rooted at $r.right$.

Preorder traversal

- Given a tree with root r, first visit r, then visit subtrees rooted at r's children, using preorder traversal.

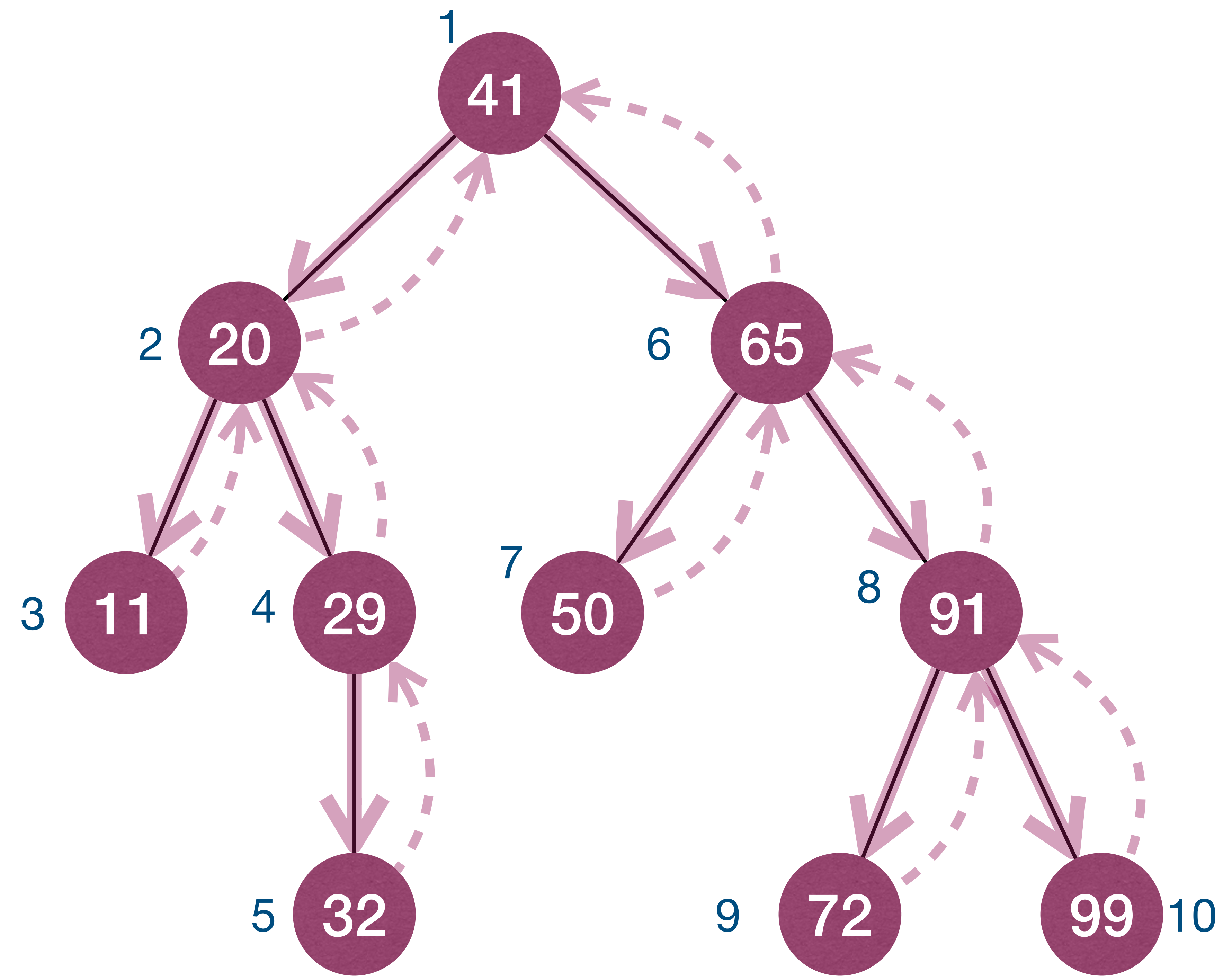
<u>PreorderTrav(r):</u>

if r != $NULL$

  $Visit(r)$

  for each child u **of** r

    $PreorderTrav(u)$

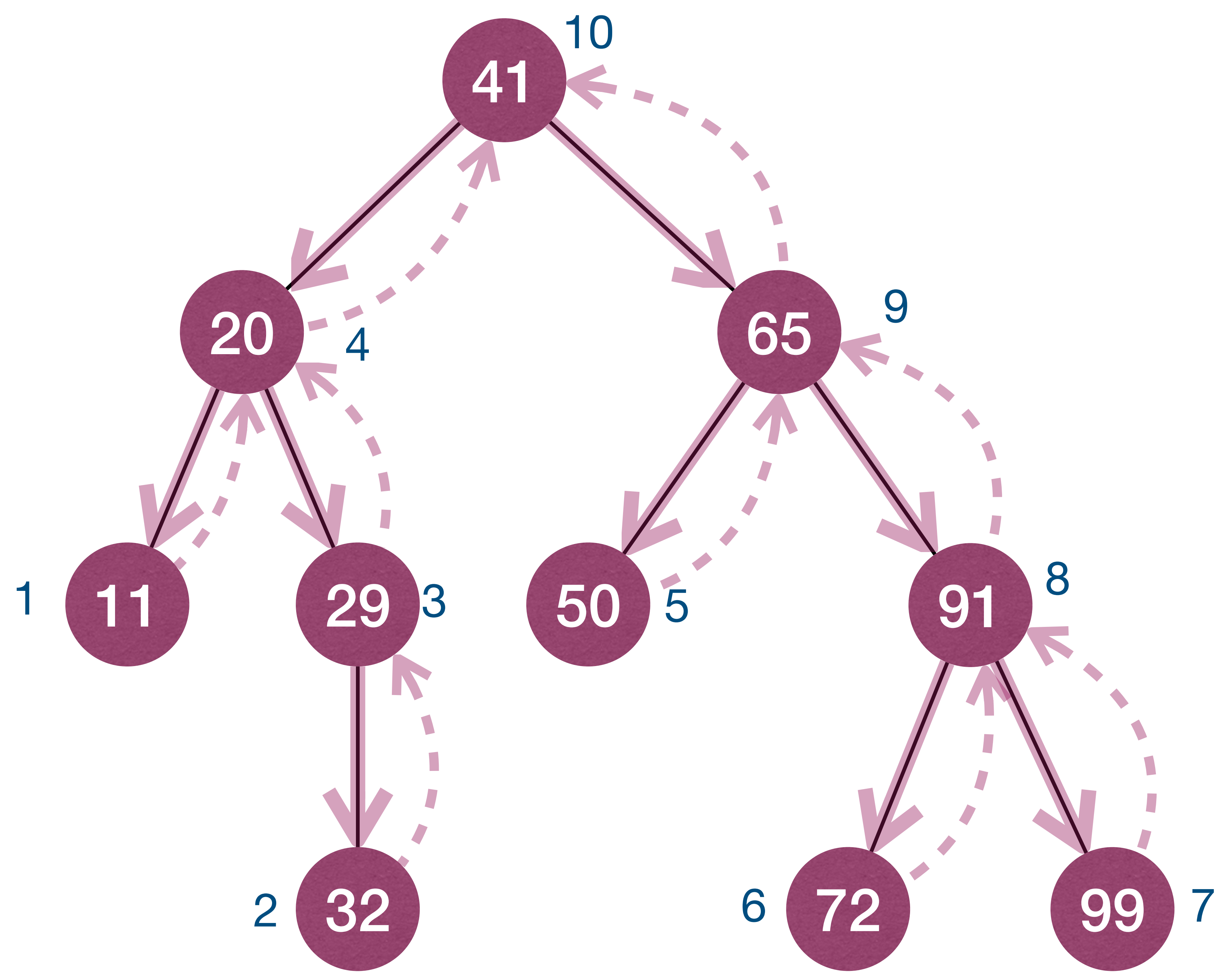


41 20 11 29 32 65 50 91 72 99

Postorder traversal

- Given a tree with root r, first visit subtrees rooted at r's children using postorder traversal, then visit r.

```
PostorderTrav(r):
if  r != NULL
    for each child u of r
            PostorderTrav(u)
    Visit(r)
```



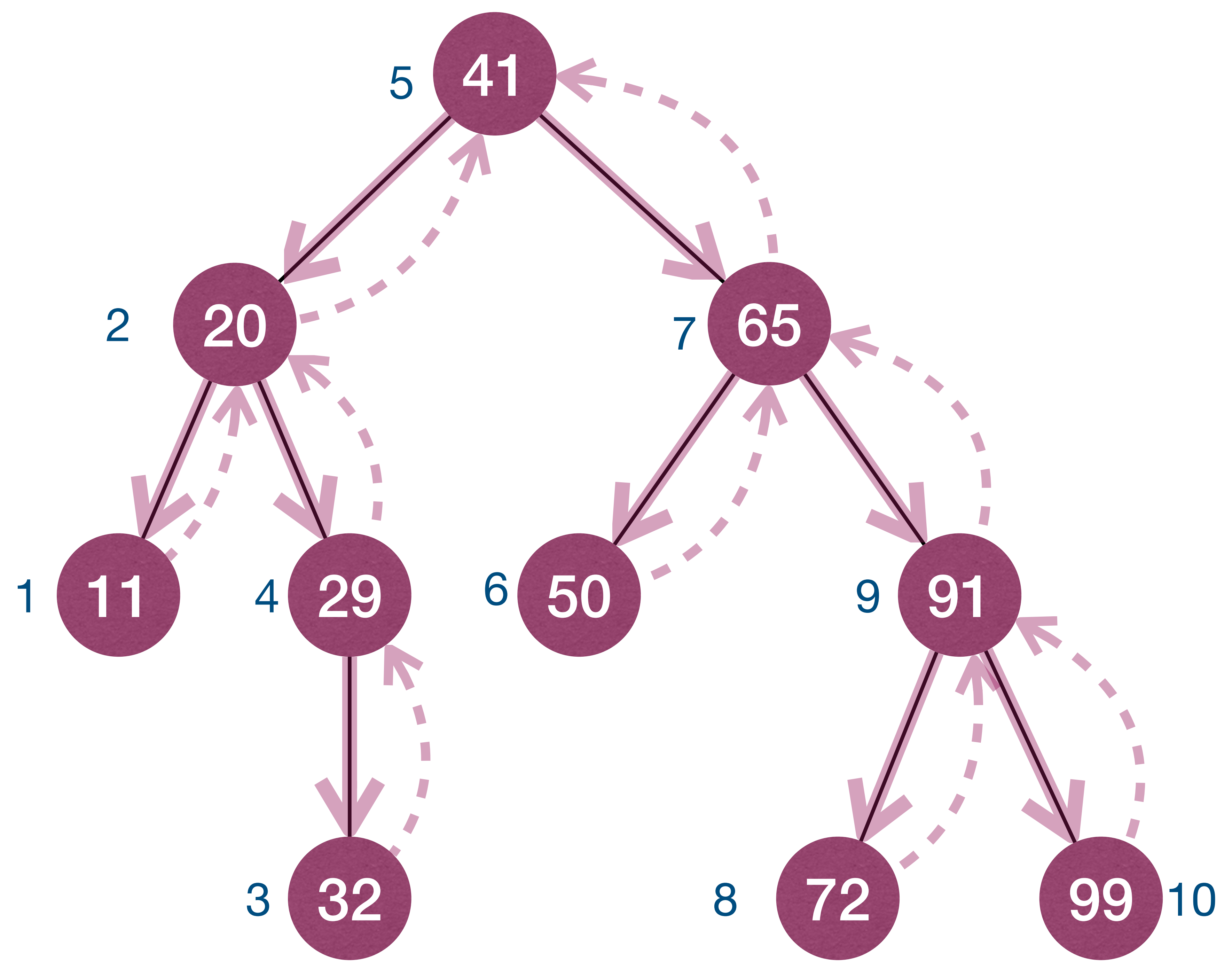
11 32 29 20 50 72 99 91 65 41

Inorder traversal

- Given a ***binary*** tree with root r, first visit subtree rooted at $r.left$, then visit r, finally visit subtree rooted at $r.right$.

<u>InorderTrav(r):</u>

if r != $NULL$
    $InorderTrav(r.left)$
    $Visit(r)$
    $InorderTrav(r.right)$



11 20 32 29 41 50 65 72 91 99

Complexity of recursive traversal

PreorderTrav(r):

if r != $NULL$
    $Visit(r)$
    for each child u **of** r
        $PreorderTrav(u)$

PostorderTrav(r):

if r != $NULL$
    for each child u **of** r
        $PostorderTrav(u)$
    $Visit(r)$

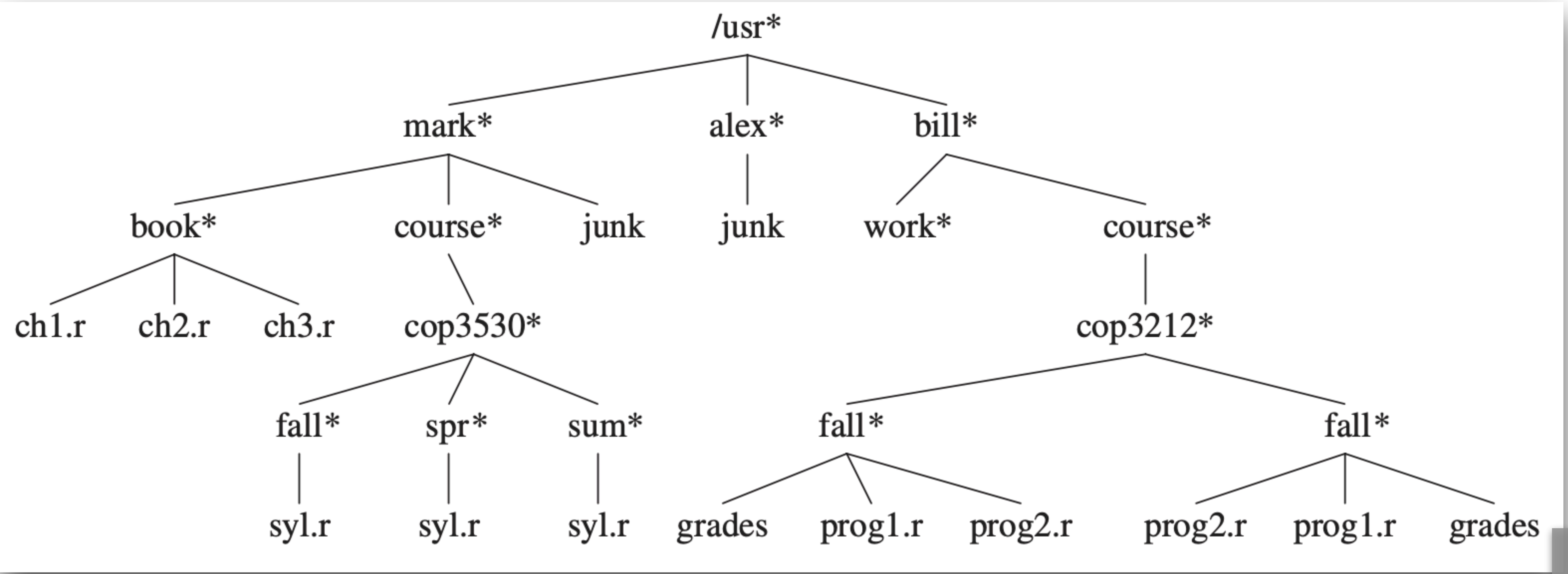
InorderTrav(r):

if r != $NULL$
    $InorderTrav(r.left)$
    $Visit(r)$
    $InorderTrav(r.right)$

- Time complexity for a size n tree?
  - $\Theta(n)$ as processing each node takes $\Theta(1)$.
- Space complexity for a size n tree?
  - $O(n)$ as worst-case **call stack** depth is $\Theta(n)$.

Sample application of preorder traversal

- Directory Listing




```
ListDir(obj, depth):
if obj != NULL
    PrintName(obj, depth)
    if IsDirectory(obj)
        for each subobj in obj
            ListDir(subobj, depth + 1)
```

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Iterative tree traversal

- Basic idea: simulate the recursive process with the help of a stack.

```
PreorderTrav(r):
if r != NULL
    Visit(r)
    for each child u of r
        PreorderTrav(u)
```

```
class Frame {
    Node node
    bool visit
    Frame(Node n, bool v) {
        node := n
        visit := v
    }
}
```

Visit `node` or the subtree rooted at `node`.

```
PreorderTravIter(r):
Stack s
s.push(Frame(r, false))
while !s.empty()
    f = s.pop()
    if f.node != NULL
        if f.visit
            Visit(f.node)
        else
            for each child u of f.node
                s.push(Frame(u, false))
            s.push(Frame(f.node, true))
```

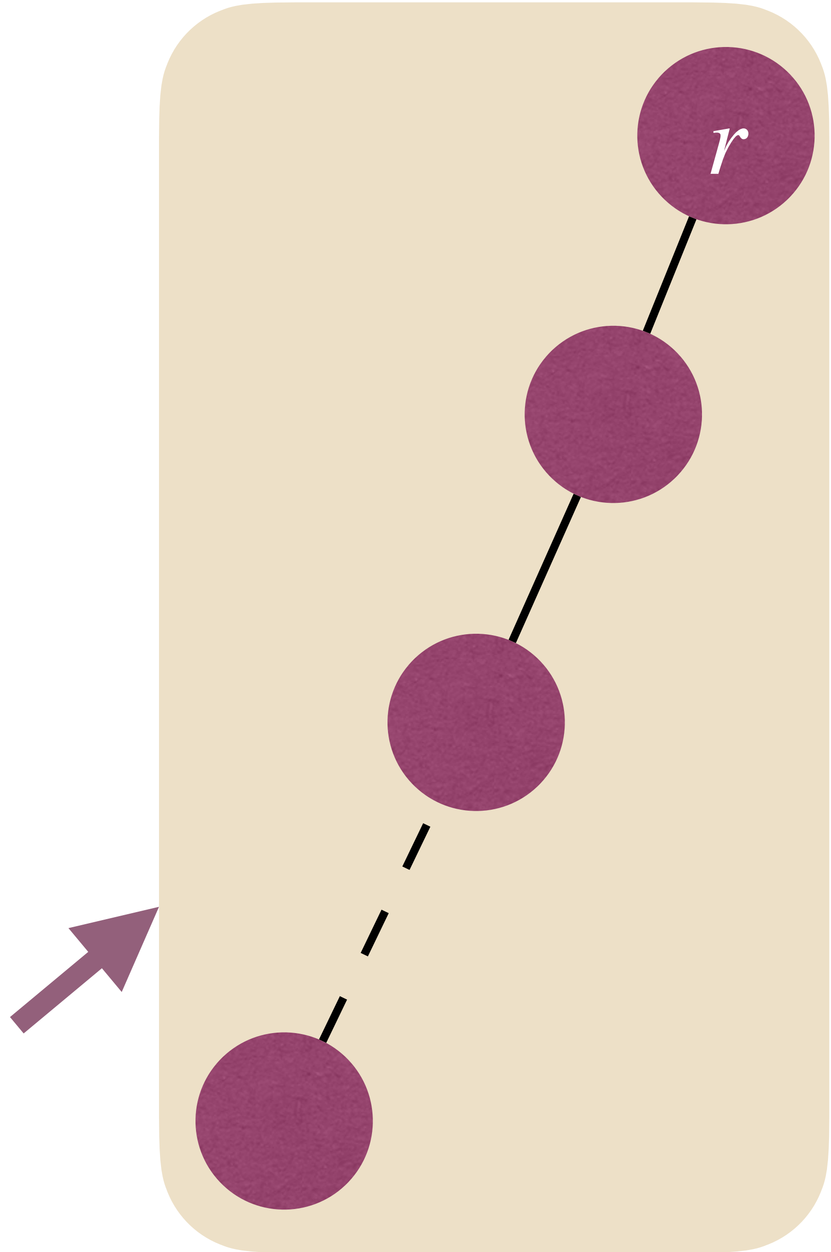
Exchange for postorder traversal

What about inorder traversal?

Iterative inorder tree traversal

```
InorderTravIter(r):
Stack s
s.push(Frame(r, false))
while !s.empty()
    f = s.pop()
    if f.node != NULL
        if f.visit
            Visit(f.node)
        else
            s.push(Frame(f.node.right, false))
            s.push(Frame(f.node, true))
            s.push(Frame(f.node.left, false))
```

- What is the time complexity?
  - $\Theta(n)$
- What is the space complexity?
  - $O(n)$
- When do we need $\Theta(\mathrm{n})$ space?

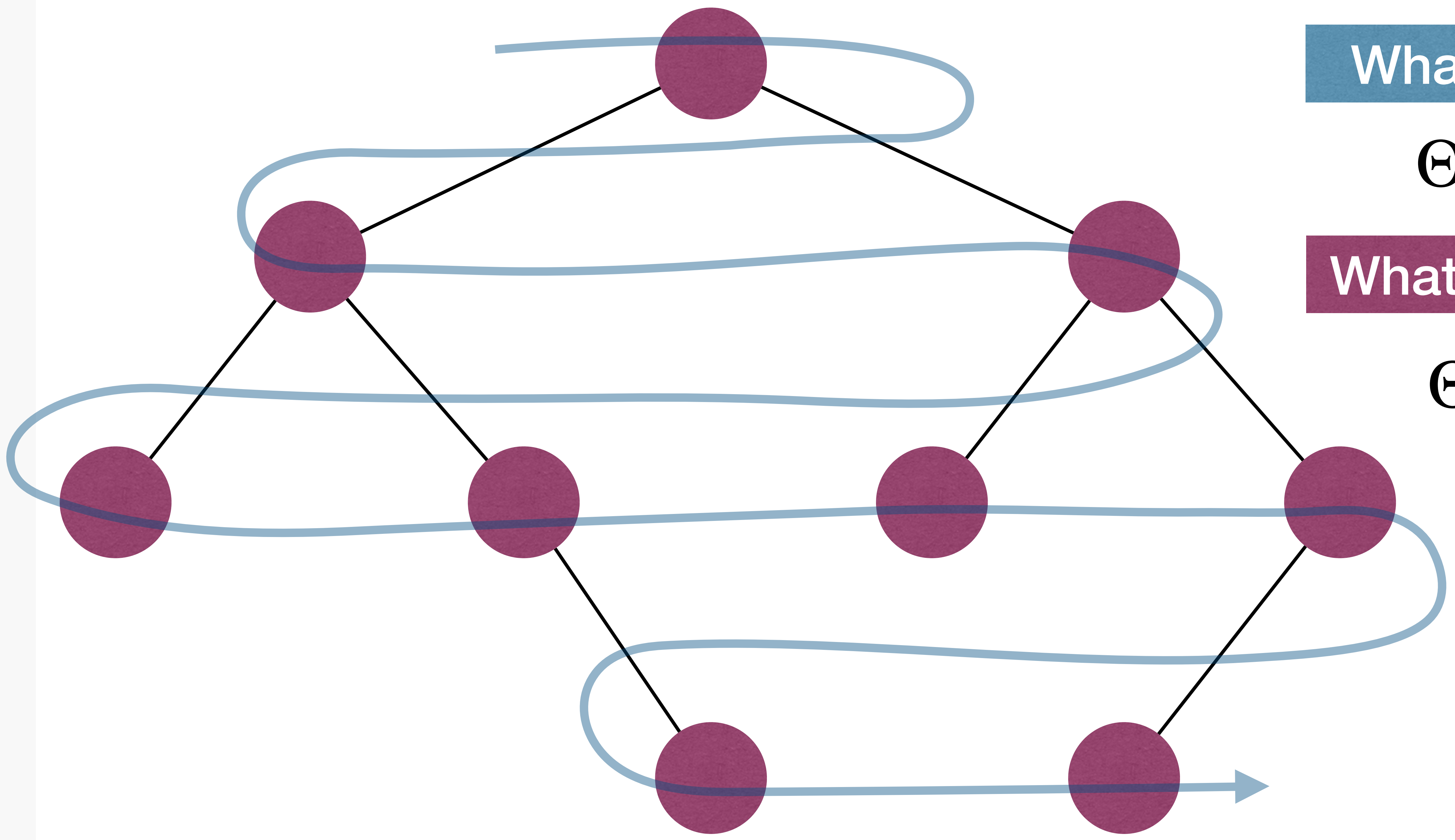


- Can we have better space complexity?
  - *Morris inorder tree traversal

Level-order traversal of trees

- A special kind of traversal is ***breadth-first traversal***. (Previous methods are all **depth-first** traversal.)
  - In a breadth-first traversal, the nodes are visited level-by-level starting at the root and moving down, visiting the nodes at each level from left to right.

```
LevelorderTrav(r):
if r != NULL
Queue q
q.add(r)
while !q.empty()
    node := q.remove()
    if node != NULL
        Visit(node)
        q.add(node.left)
        q.add(node.right)
```



What is the time complexity?

$\Theta(n)$

What is the space complexity?

$\Theta(n)$ in the worst-case

Further reading

- [CLRS] Ch.10 (10.4)
- [Weiss] Ch.4 (4.1-4.2)
- [Morin] Ch.6 (6.1)