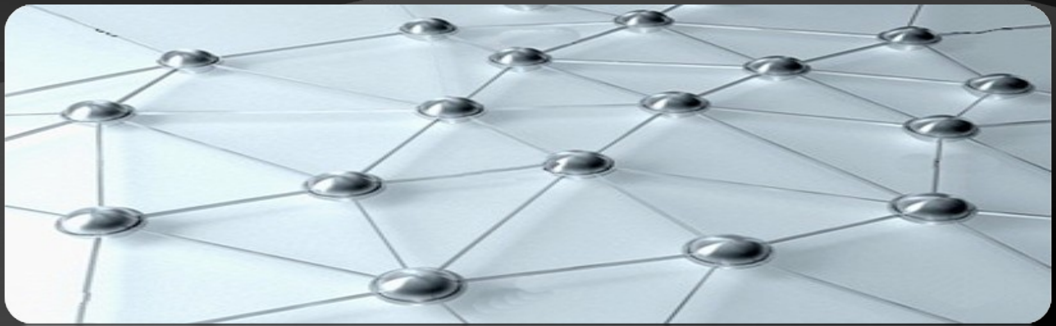
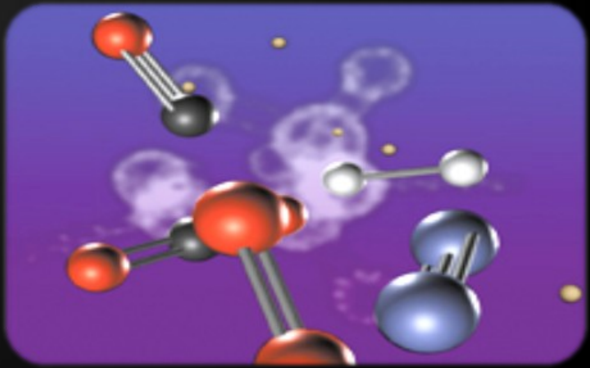


Trees and Graphs

Trees, Binary Search Trees, Balanced Trees, Graphs



Svetlin Nakov

Telerik Corporation

www.telerik.com

Table of Contents

1. Tree-like Data Structures
2. Trees and Related Terminology
3. Implementing Trees
4. Traversing Trees
5. Balanced Trees
6. Graphs

Tree-like Data Structures

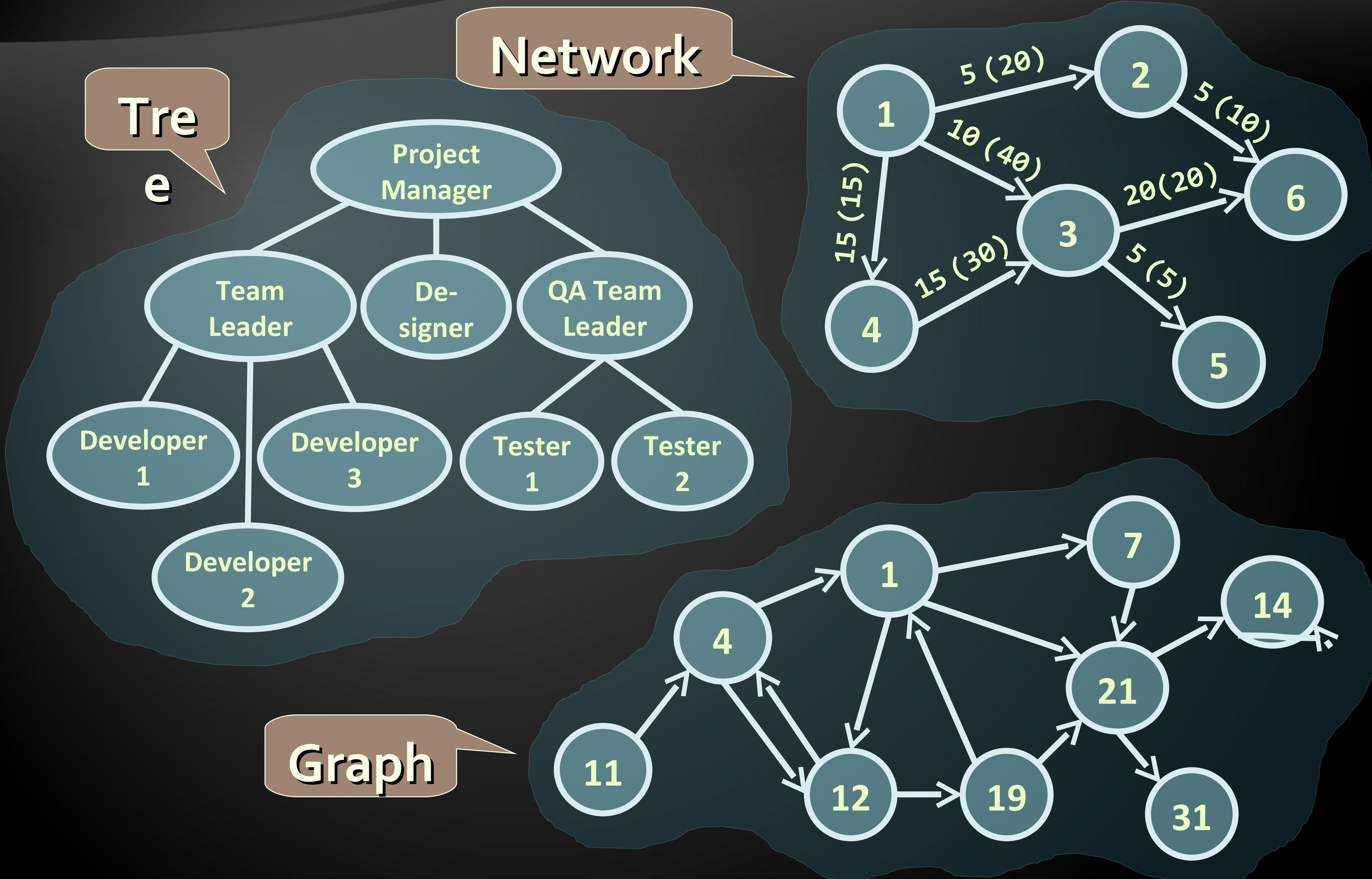
Trees, Balanced Trees, Graphs, Networks

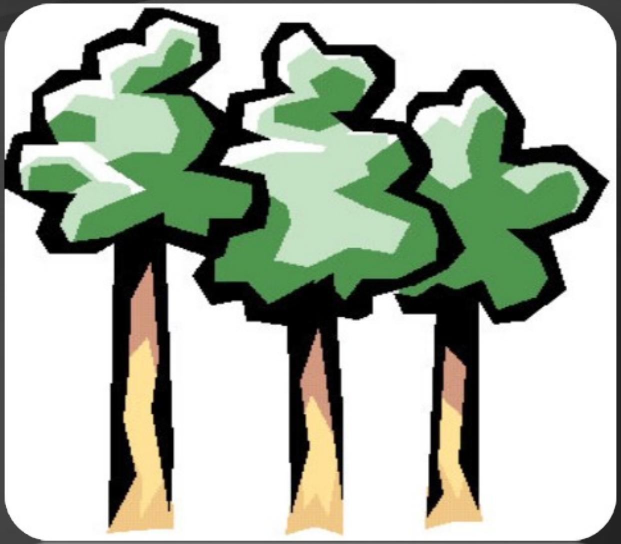
Tree-like Data Structures

- Tree-like data structures are
  - Branched recursive data structures
    - Consisting of nodes
    - Each node can be connected to other nodes
- Examples of tree-like structures
  - Trees: binary / other degree, balanced, etc.
  - Graphs: directed / undirected, weighted, etc.
  - Networks

telerik

Tree-like Data Structures



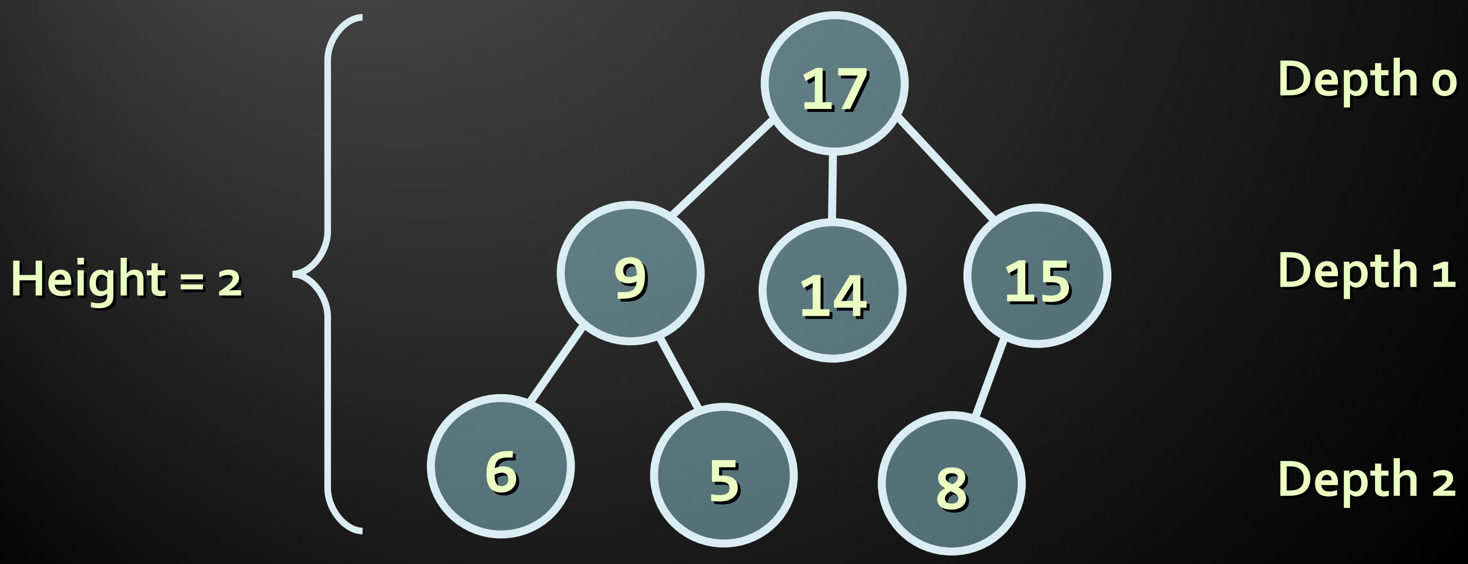


Trees and Related Terminology

Node, Edge, Root, Children, Parent, Leaf , Binary Search Tree, Balanced Tree

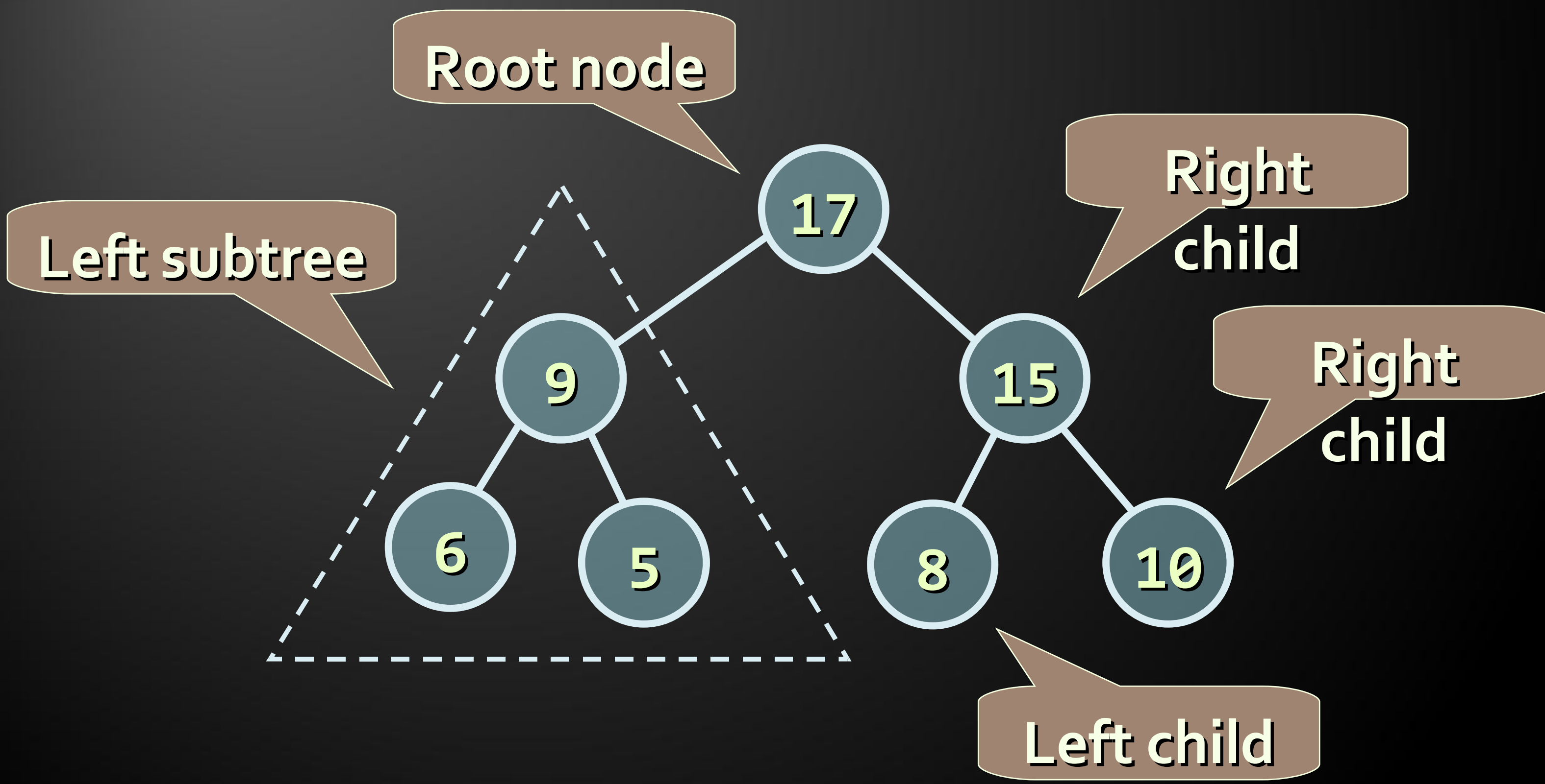
Trees

- Tree data structure – terminology
  - Node, edge, root, child, children, siblings, parent, ancestor, descendant, predecessor, successor, internal node, leaf, depth, height, subtree



Binary Trees

- Binary trees: most widespread form
  - Each node has at most 2 children

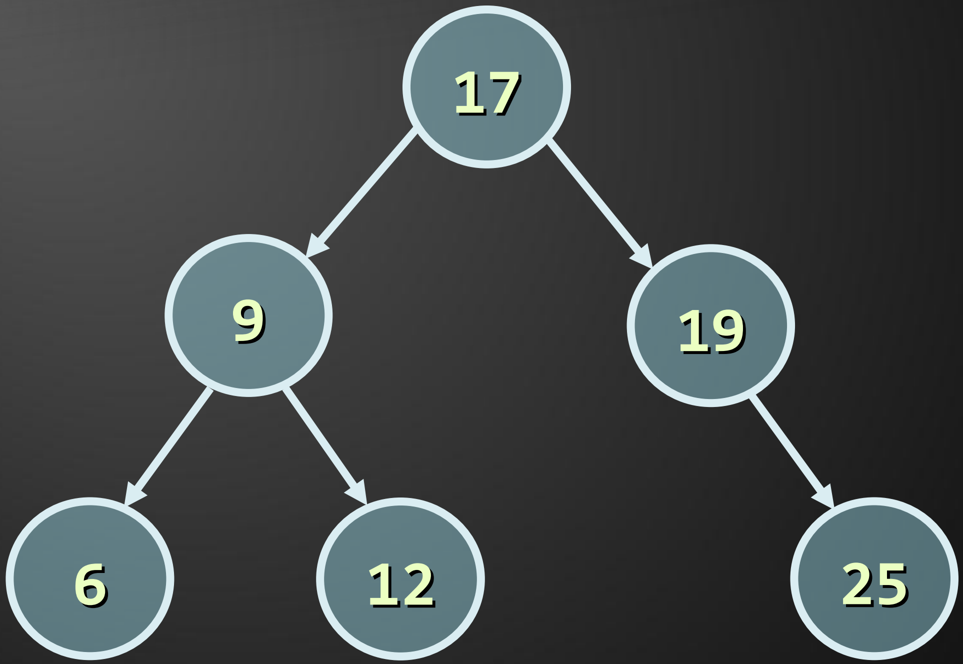


Binary Search Trees

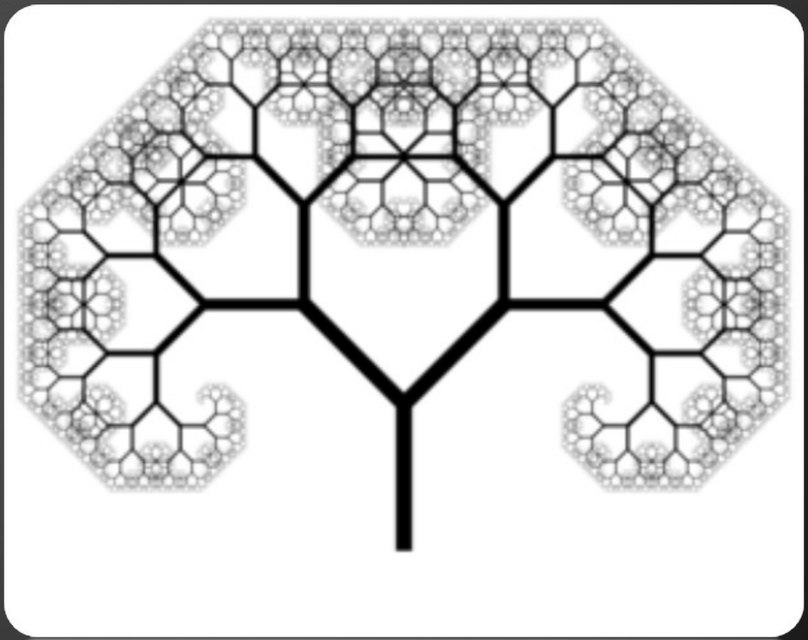
- Binary search trees are ordered
  - For each node *x* in the tree
    - All the elements of the left subtree of *x* are ≤ *x*
    - All the elements of the right subtree of *x* are > *x*
- Binary search trees can be balanced
  - Balanced trees have height of ~ $\log_2(x)$
  - Balanced trees have for each node nearly equal number of nodes in its subtrees

Binary Search Trees (2)

- Example of balanced binary search tree



- If the tree is balanced, add / search / delete operations take approximately log(n) steps



Implementing Trees

Recursive Tree Data Structure

Recursive Tree Definition

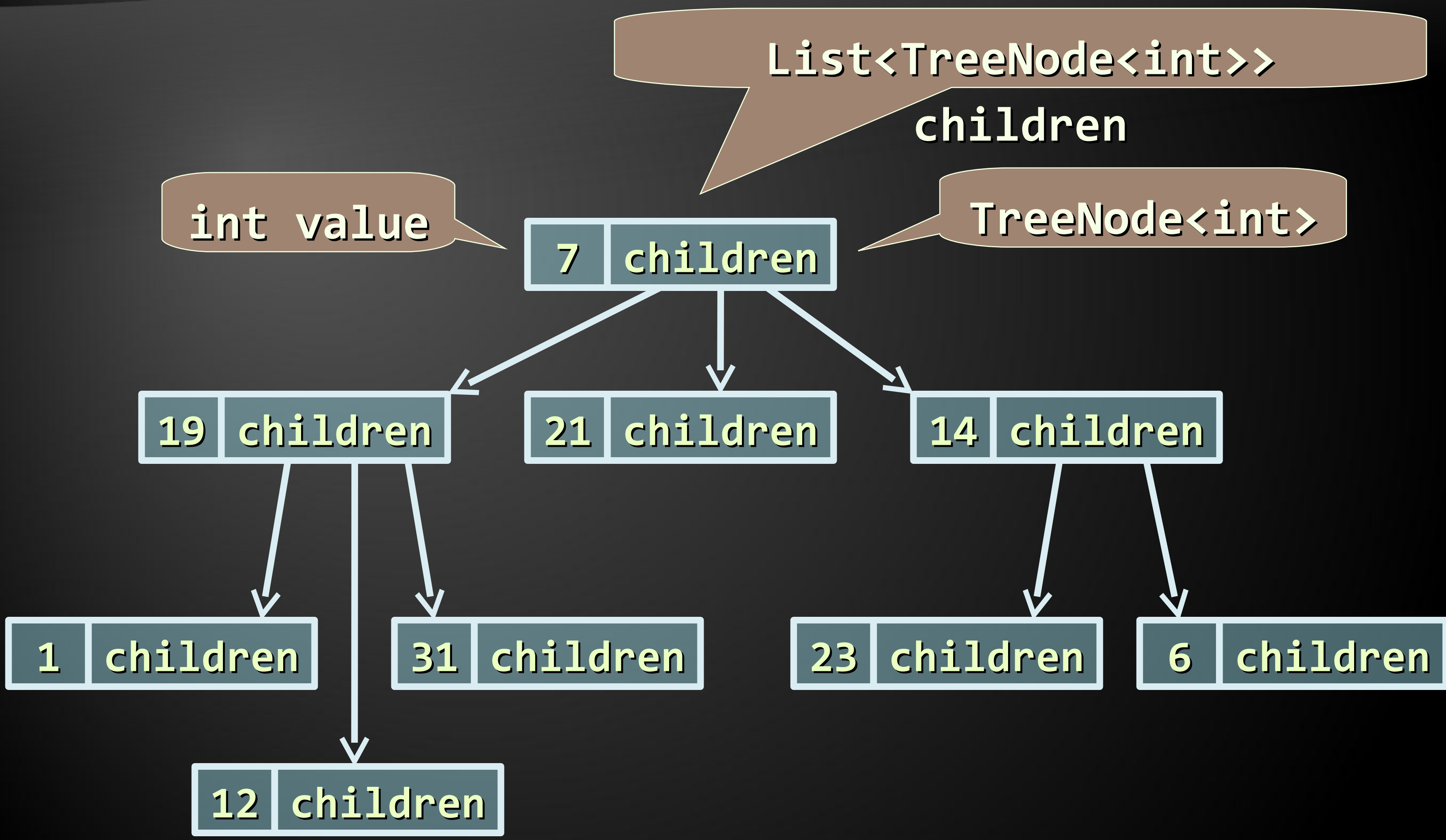
- The recursive definition for **tree** data structure:
  - A single node is tree
  - Tree nodes can have zero or multiple children that are also trees
- Tree node definition in C#

```
public class TreeNode<T>
{
  private T value;
  private List<TreeNode<T>> children;
  …
}
```

The value contained in the node

List of child nodes, which are of the same type

TreeNode<int> Structure



Implementing TreeNode<T>

```
public TreeNode(T value)
{
  this.value = value;
  this.children = new List<TreeNode<T>>();
}
public T Value
{
  get { return this.value; }
  set { this.value = value; }
}
public void AddChild(TreeNode<T> child)
{
  child.hasParent = true;
  this.children.Add(child);
}
public TreeNode<T> GetChild(int index)
{
  return this.children[index];
}
```

Implementing Tree<T>

- The class Tree<T> keeps tree's root node

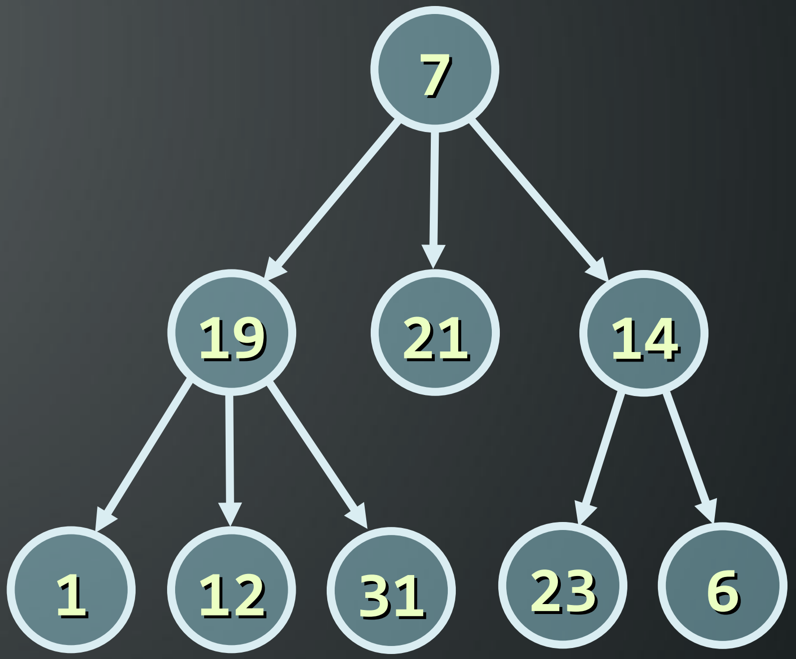
```
public class Tree<T>
{
  private TreeNode<T> root;

  public Tree(T value, params Tree<T>[] children): this(value)
  {
    foreach (Tree<T> child in children)
    {
      this.root.AddChild(child.root);
    }
  }

  public TreeNode<T> Root
  {
    get {  return this.root; }
  }
}
```

Flexible constructor for building trees

Building a Tree

- Constructing tree by nested constructors:

```
Tree<int> tree =
   new Tree<int>(7,
      new Tree<int>(19,
         new Tree<int>(1),
         new Tree<int>(12),
         new Tree<int>(31)),
      new Tree<int>(21),
      new Tree<int>(14,
         new Tree<int>(23),
         new Tree<int>(6))
);
```

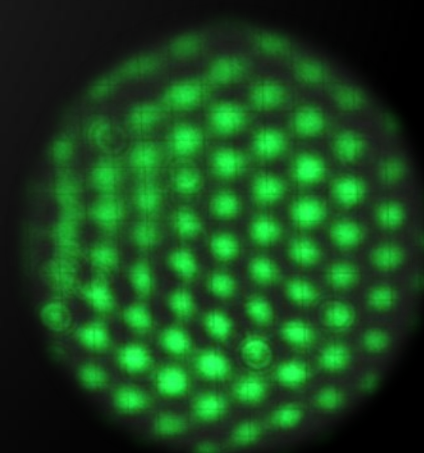


Tree Traversals

DFS and BFS Traversals

Tree Traversal Algorithms

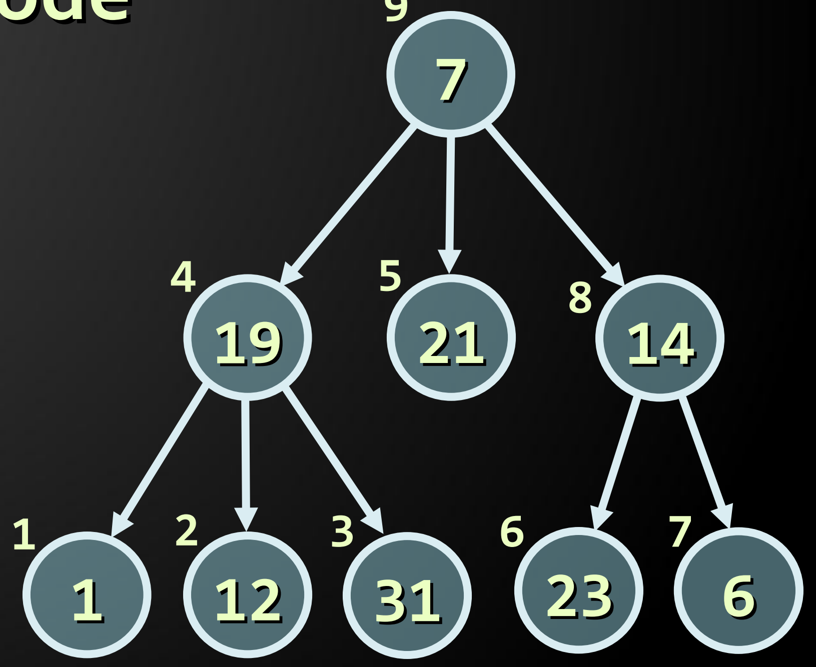
- Traversing a tree means to visit each of its nodes exactly one in particular order
  - Many traversal algorithms are known
  - Depth-First Search (DFS)
    - Visit node's successors first
    - Usually implemented by recursion
  - Breadth-First Search (BFS)
    - Nearest nodes visited first
    - Implemented by a queue



Depth-First Search (DFS)

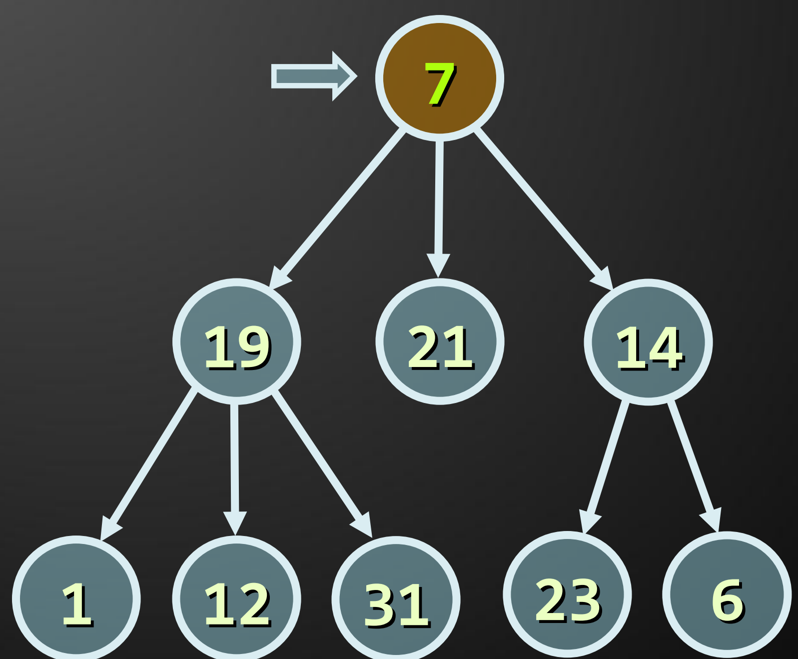
- Depth-First Search first visits all descendants of given node recursively, finally visits the node itself
- DFS algorithm pseudo code

```
DFS(node)
{
  for each child c of node
    DFS(c);
  print the current node;
}
```



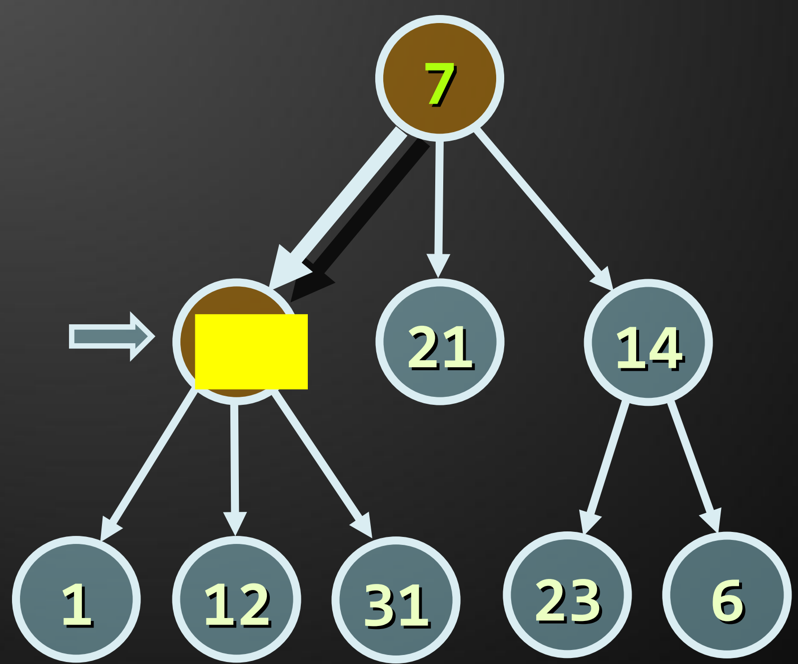
DFS in Action (Step 1)

- Stack:
- Output: (empty)



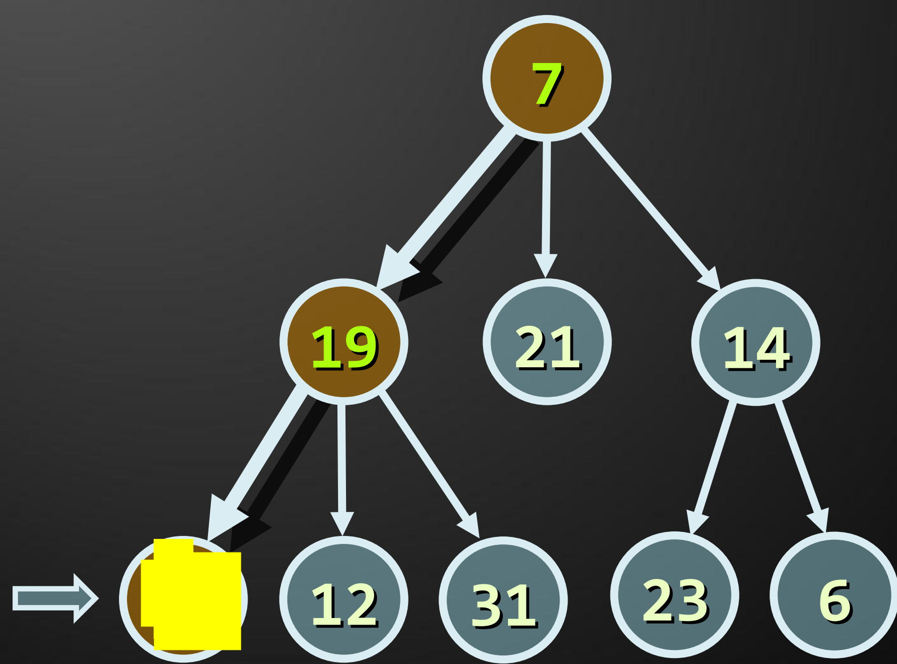
DFS in Action (Step 2)

- Stack: 7, 1
- Output: ~~(empty)~~



DFS in Action (Step 3)

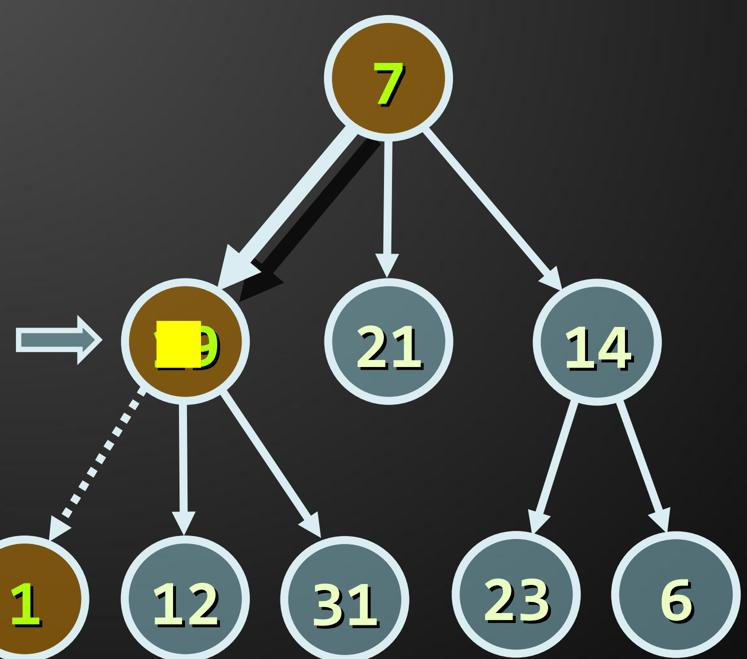
- **Stack: 7, 19, 1**
- **Output: (empty)**



telerik

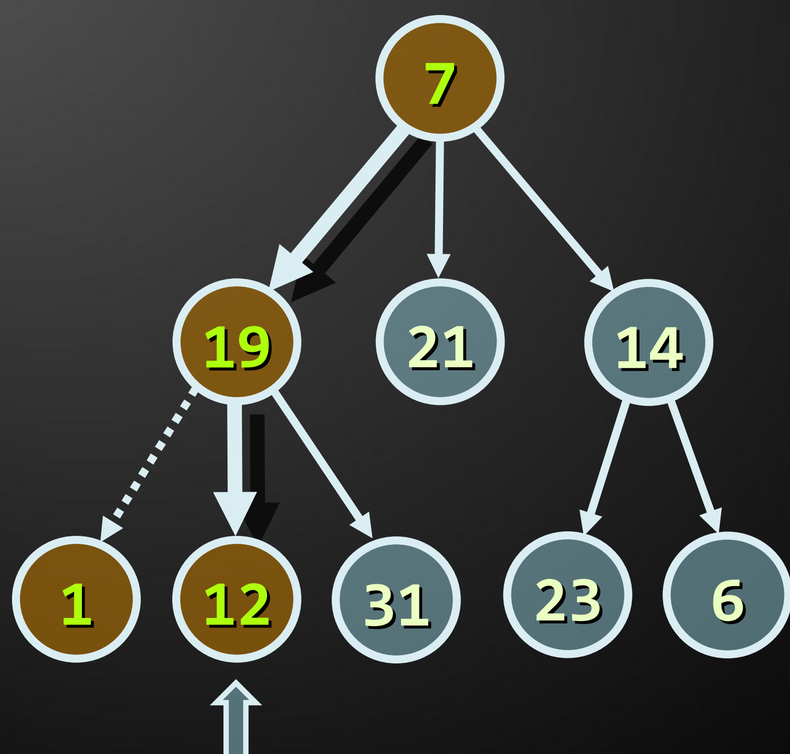
DFS in Action (Step 4)

- **Stack: 7, 19**
- **Output:**



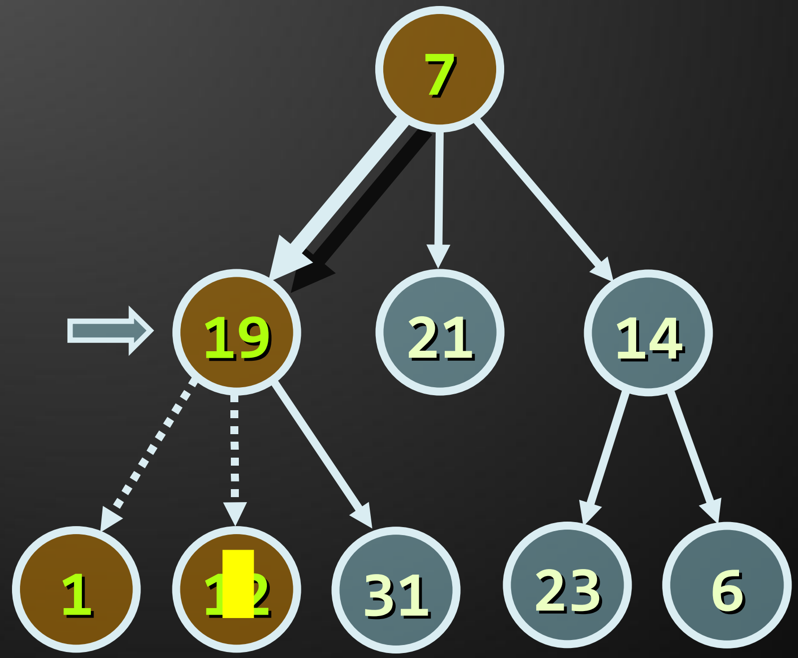
DFS in Action (Step 5)

- **Stack: 7, 19,**
- **Output: 1**



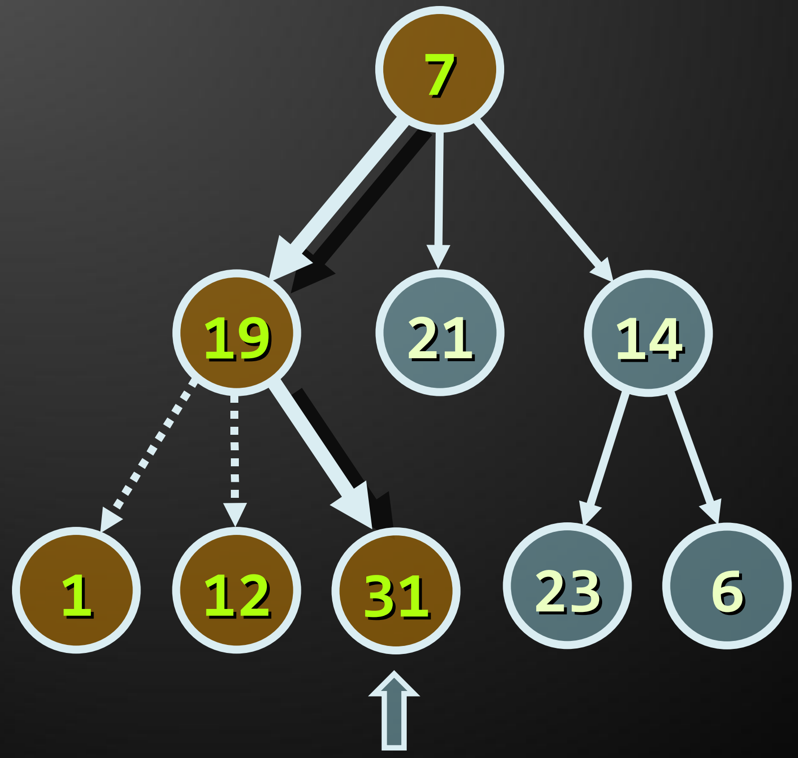
DFS in Action (Step 6)

- Stack: 7, 19
- Output: 1, 12



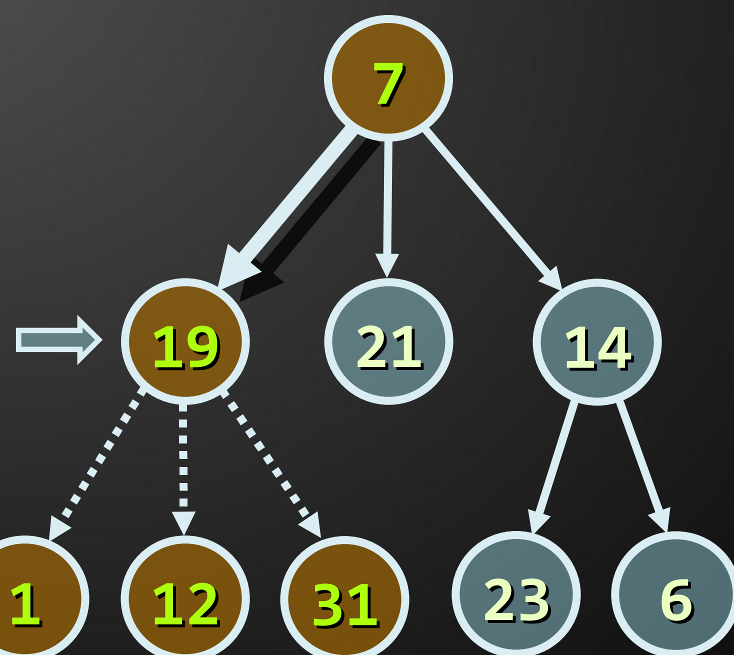
DFS in Action (Step 7)

- **Stack: 7, 19,**
- **Output: 1, 12**



DFS in Action (Step 8)

- **Stack: 7, 19**
- **Output: 1, 12, 31**

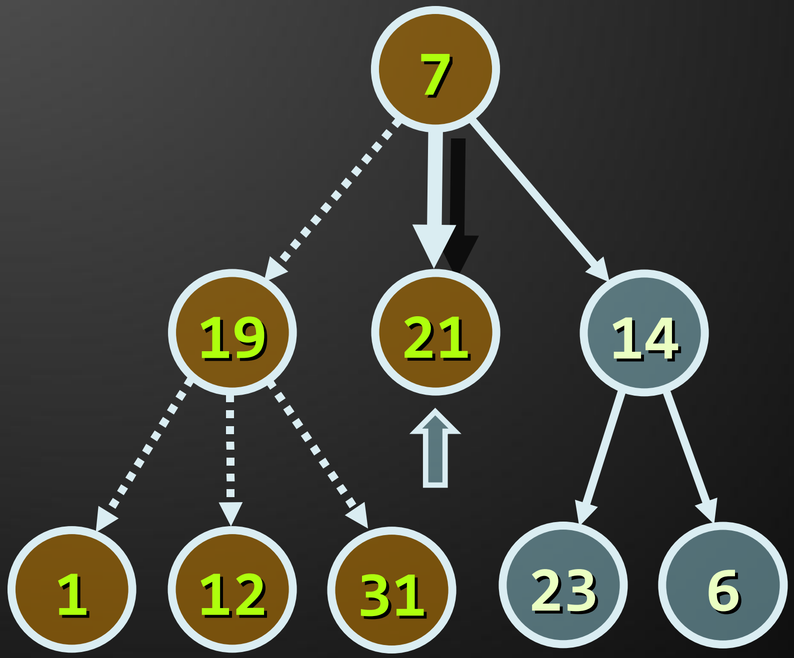


DFS in Action (Step 9)

- **Stack: 7**
- **Output: 1, 12, 31, 19**

DFS in Action (Step 10)

- **Stack: 7, 21**
- **Output: 1, 12, 31, 19**

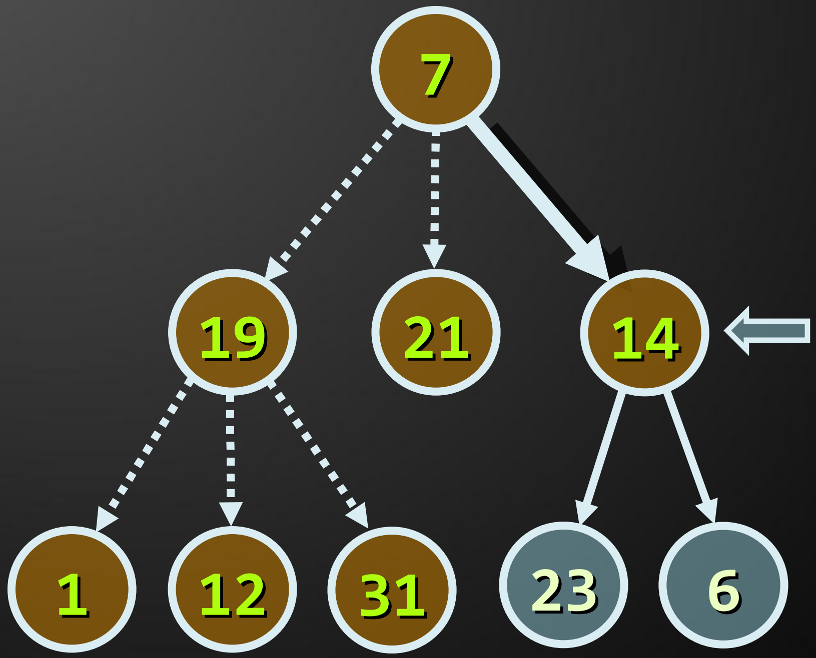


DFS in Action (Step 11)

- **Stack: 7**
- **Output: 1, 12, 31, 19, 21**

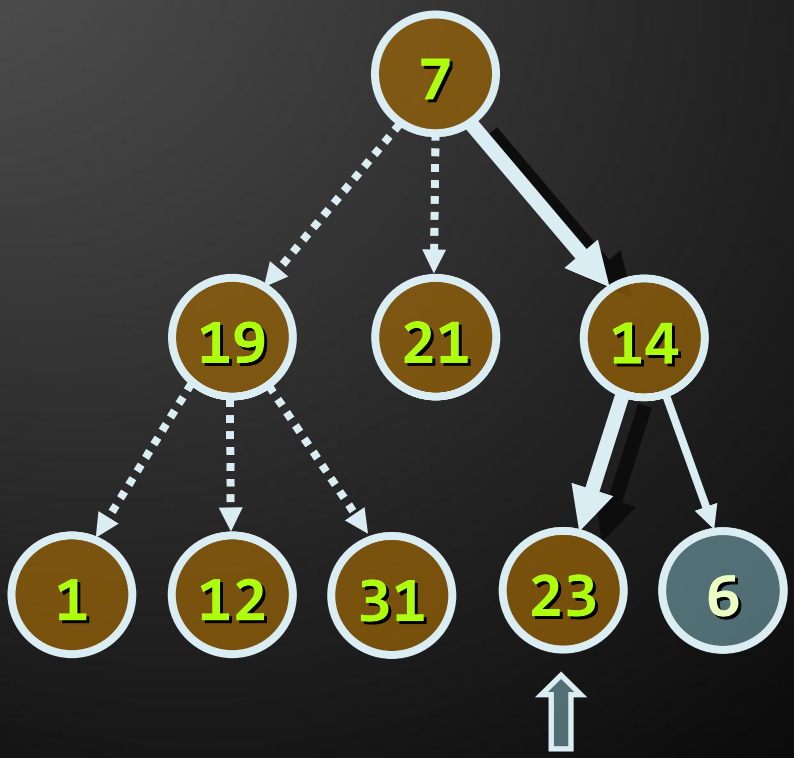
DFS in Action (Step 12)

- **Stack: 7, 14**
- **Output: 1, 12, 31, 19, 21**



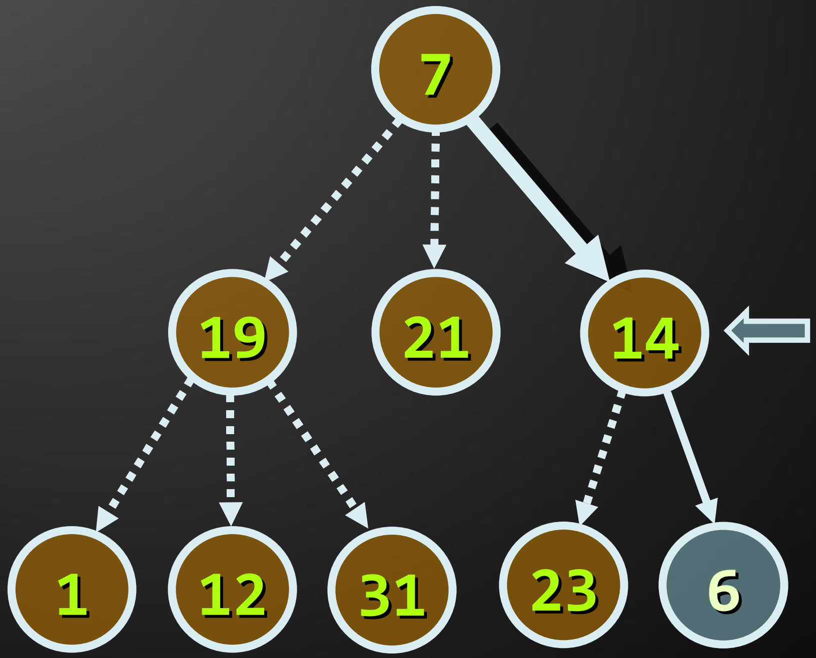
DFS in Action (Step 13)

- **Stack: 7, 14, 23**
- **Output: 1, 12, 31, 19, 21**



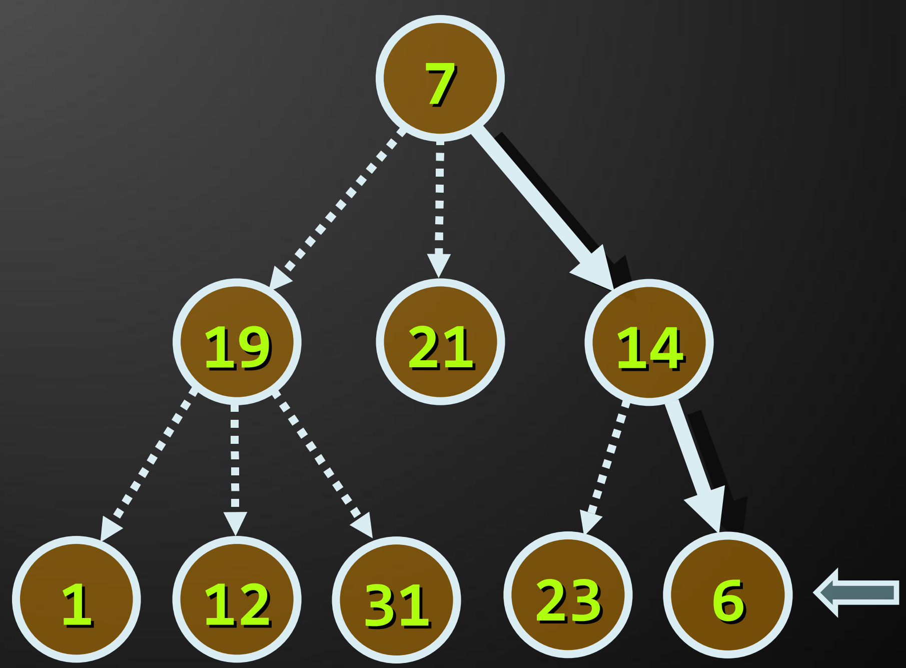
DFS in Action (Step 14)

- **Stack: 7, 14**
- **Output: 1, 12, 31, 19, 21, 23**



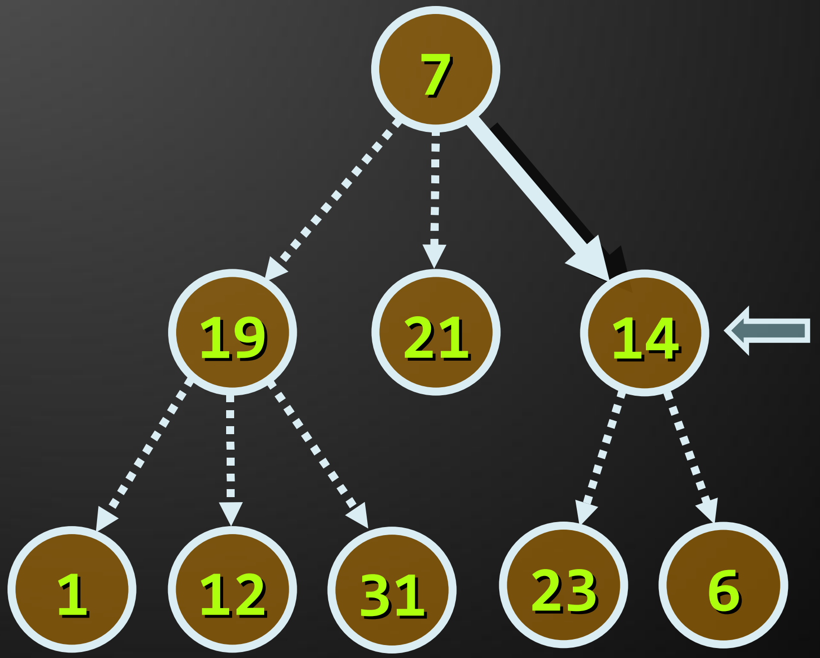
DFS in Action (Step 15)

- Stack: 7, 14, 6
- Output: 1, 12, 31, 19, 21, 23



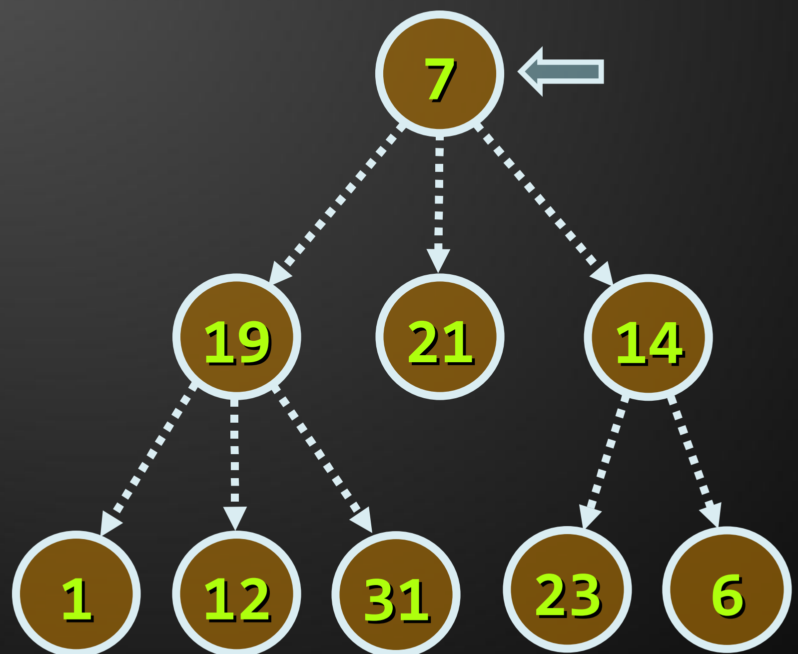
DFS in Action (Step 16)

- **Stack: 7, 14**
- **Output: 1, 12, 31, 19, 21, 23, 6**



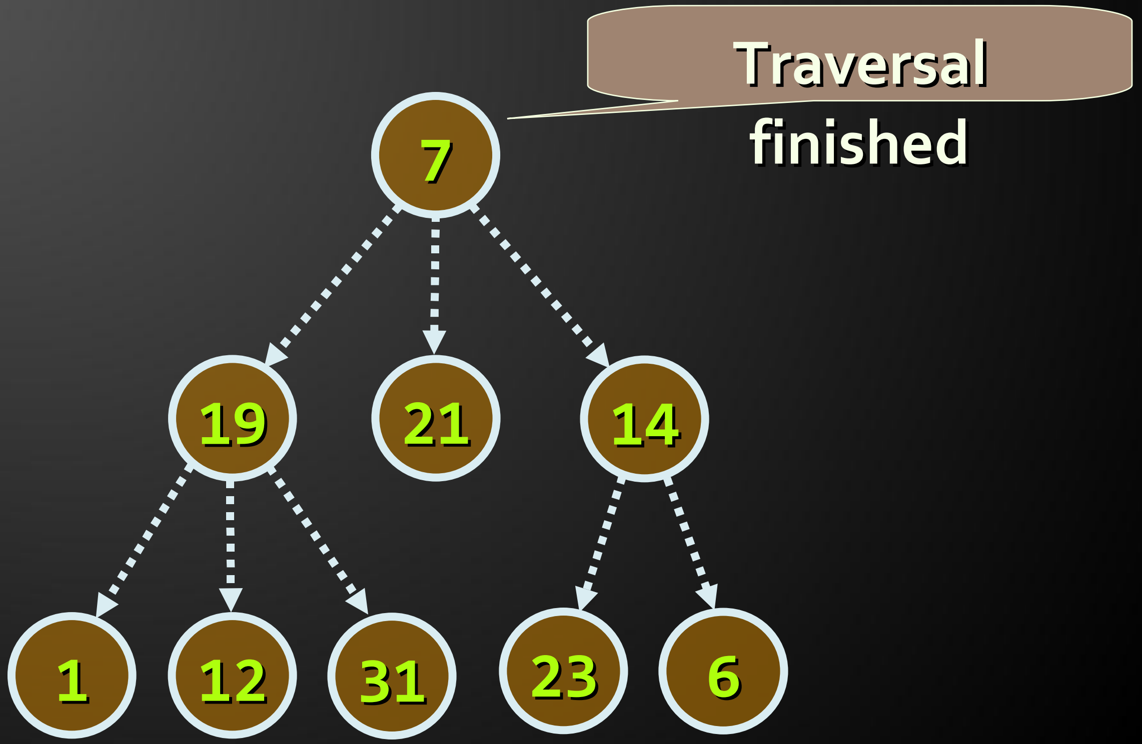
DFS in Action (Step 17)

- **Stack: 7**
- **Output: 1, 12, 31, 19, 21, 23, 6, 14**



DFS in Action (Step 18)

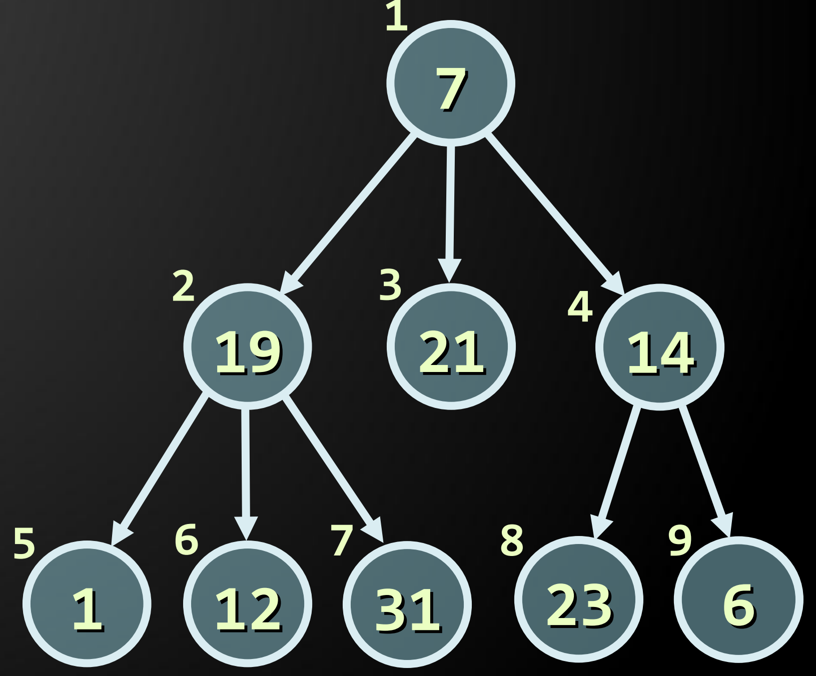
- **Stack:** **(empty)**
- **Output:** **1, 12, 31, 19, 21, 23, 6, 14, 7**



Breadth-First Search (BFS)

- Breadth-First Search first visits the neighbor nodes, later their neighbors, etc.
- BFS algorithm pseudo code

```
BFS(node)
{
  queue ← node
  while queue not empty
    v ← queue
    print v
    for each child c of v
      queue ← c
}
```



BFS in Action (Step 1)

- **Queue: 7**
- **Output: 7**

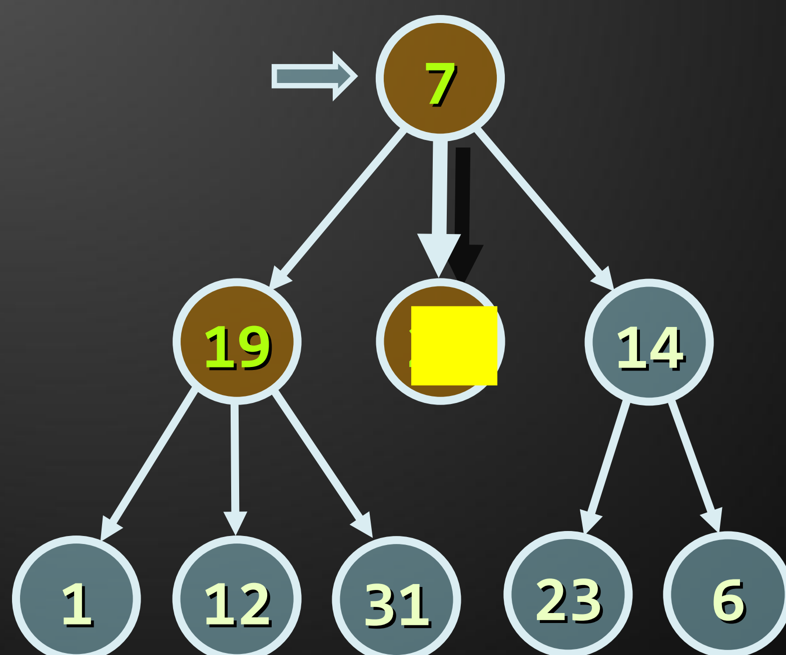
BFS in Action (Step 2)

- Queue: 7, 19
- Output: 7

telerik

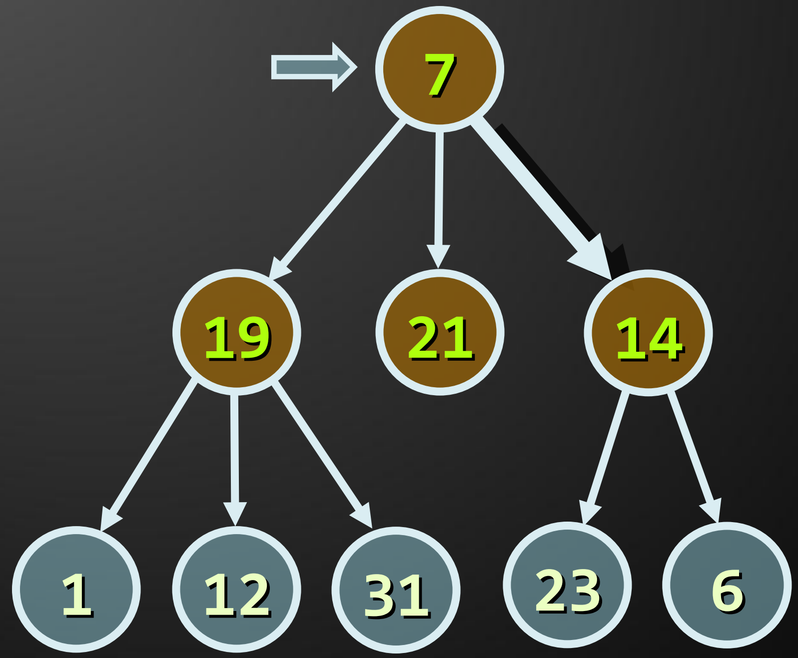
BFS in Action (Step 3)

- **Queue: 7, 19, 21**
- **Output: 7**



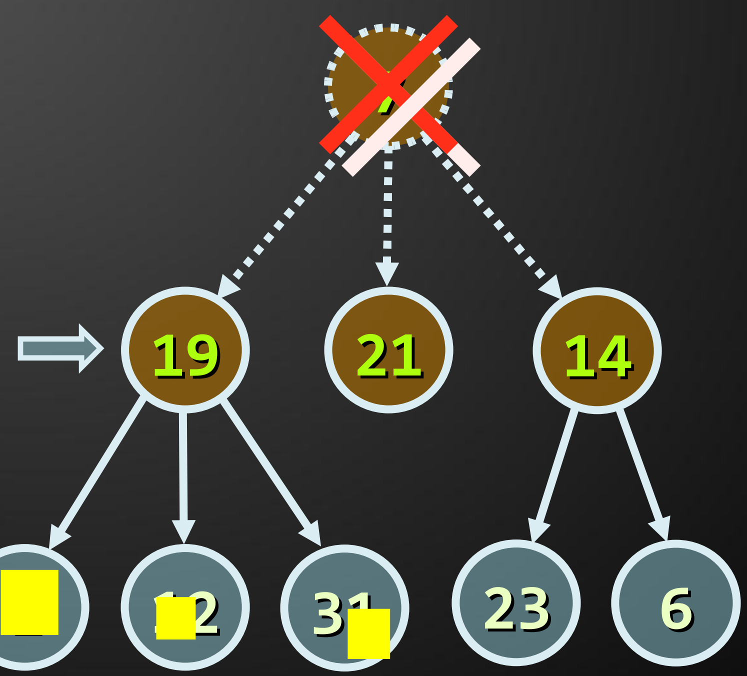
BFS in Action (Step 4)

- Queue: 7, 19, 21, 14
- Output: 7



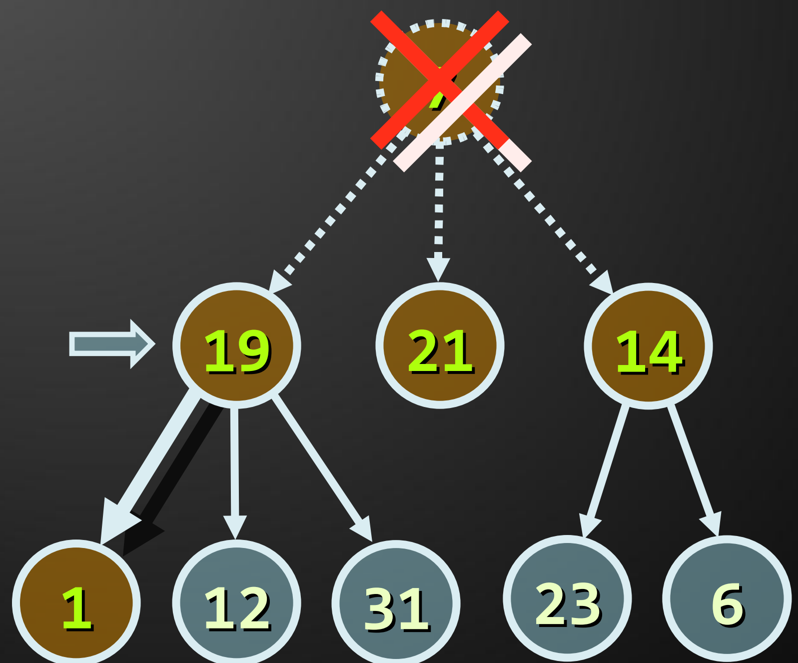
BFS in Action (Step 5)

- Queue: ~~7~~, 19, 21, 14
- Output: 7, 19



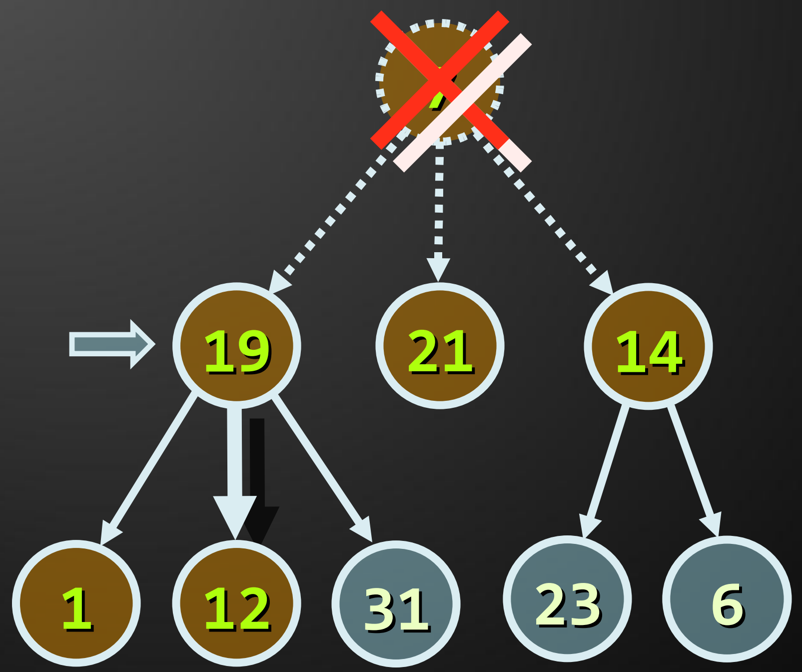
BFS in Action (Step 6)

- **Queue: ~~7~~, 19, 21, 14, 1**
- **Output: 7, 19**



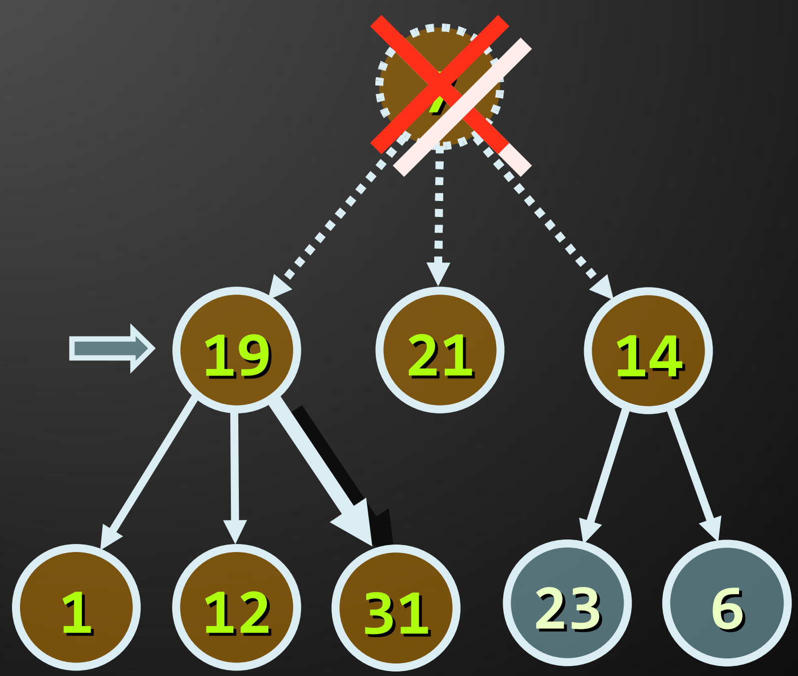
BFS in Action (Step 7)

- Queue: ~~7~~, 19, 21, 14, 1, 12
- Output: 7, 19



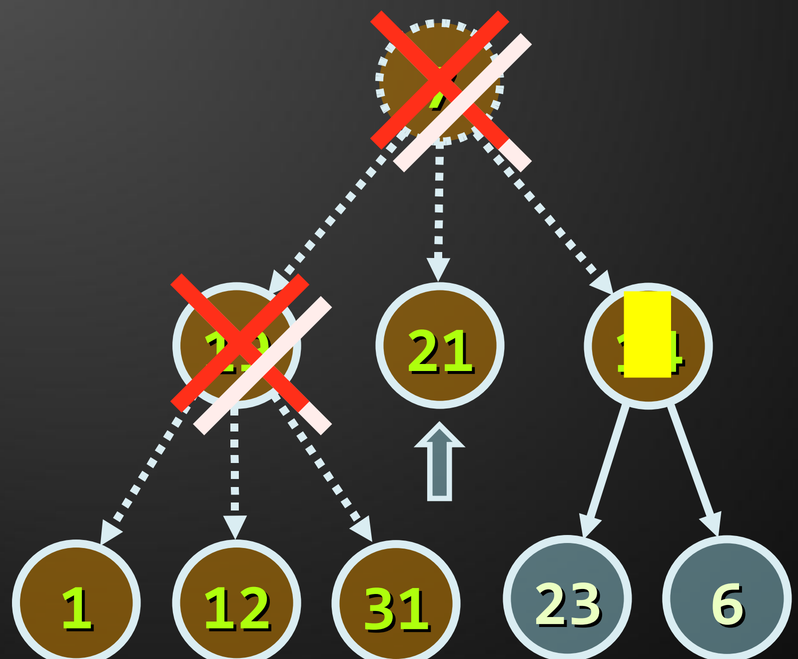
BFS in Action (Step 8)

- **Queue: ~~7~~, 21, 14, 1, 12, 31**
- **Output: 7, 19**



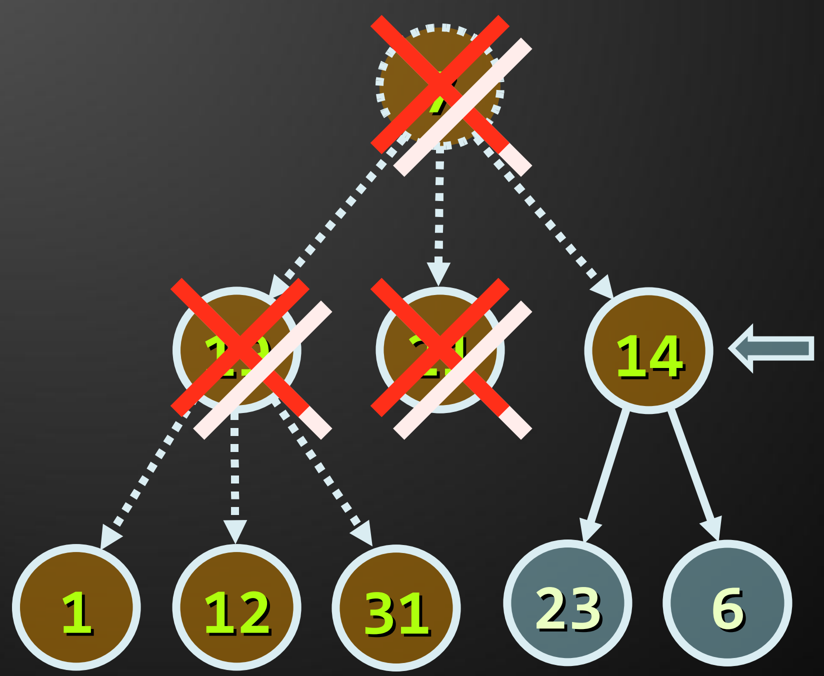
BFS in Action (Step 9)

- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12, 31
- Output: 7, 19, 21



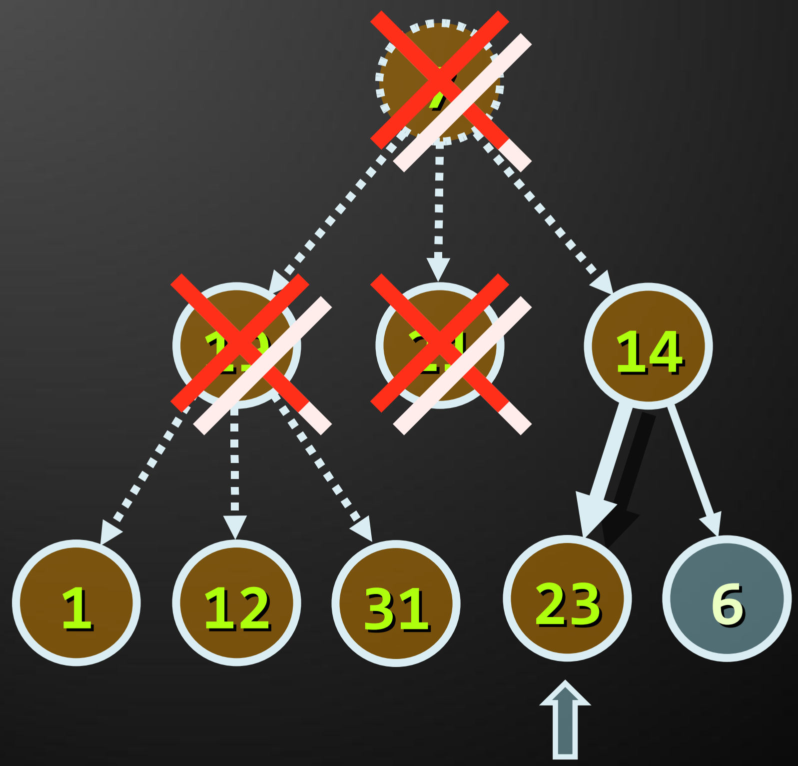
BFS in Action (Step 10)

- **Queue: ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31**
- **Output: 7, 19, 21, 14**



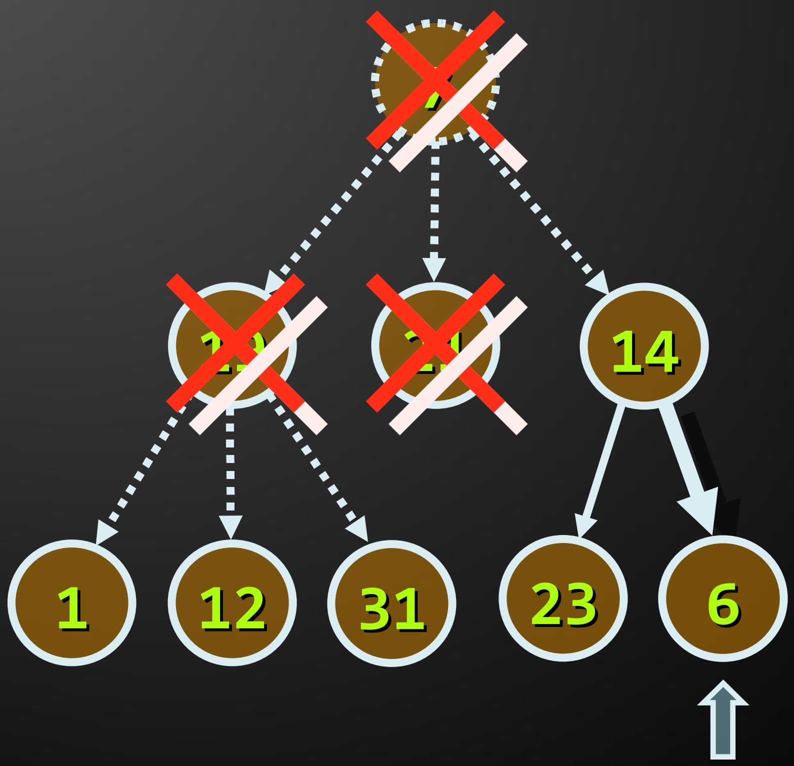
BFS in Action (Step 11)

- **Queue:** ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31, 23
- **Output:** 7, 19, 21, 14



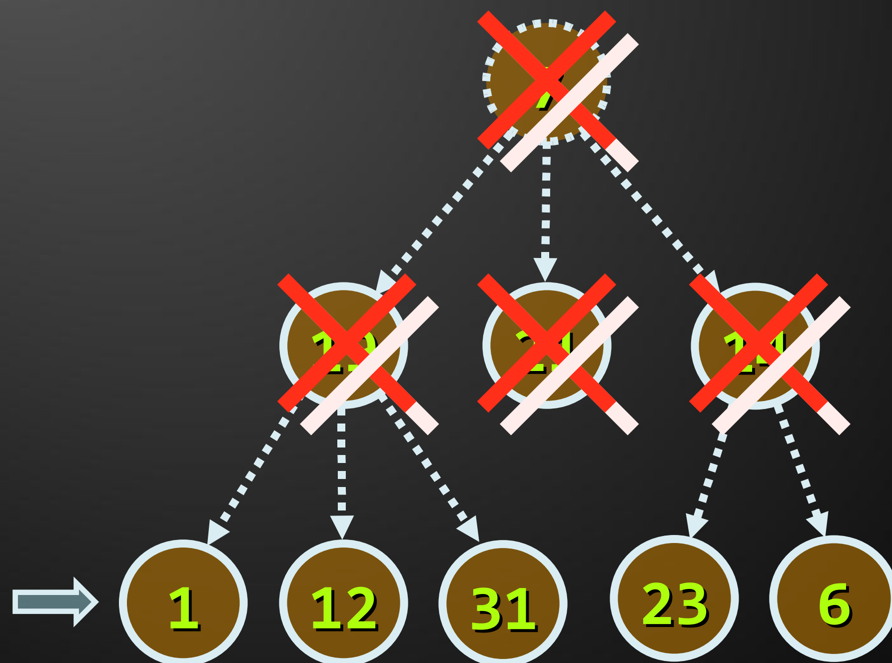
BFS in Action (Step 12)

- **Queue: ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31, 23, 6**
- **Output: 7, 19, 21, 14**



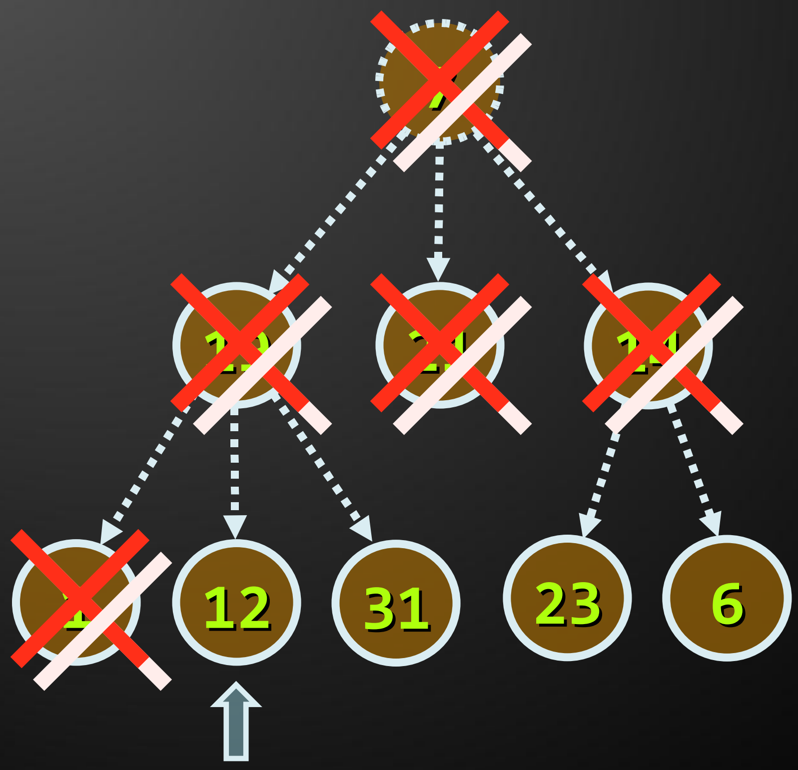
BFS in Action (Step 13)

- **Queue:** ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31, 23, 6
- **Output:** 7, 19, 21, 14, 1



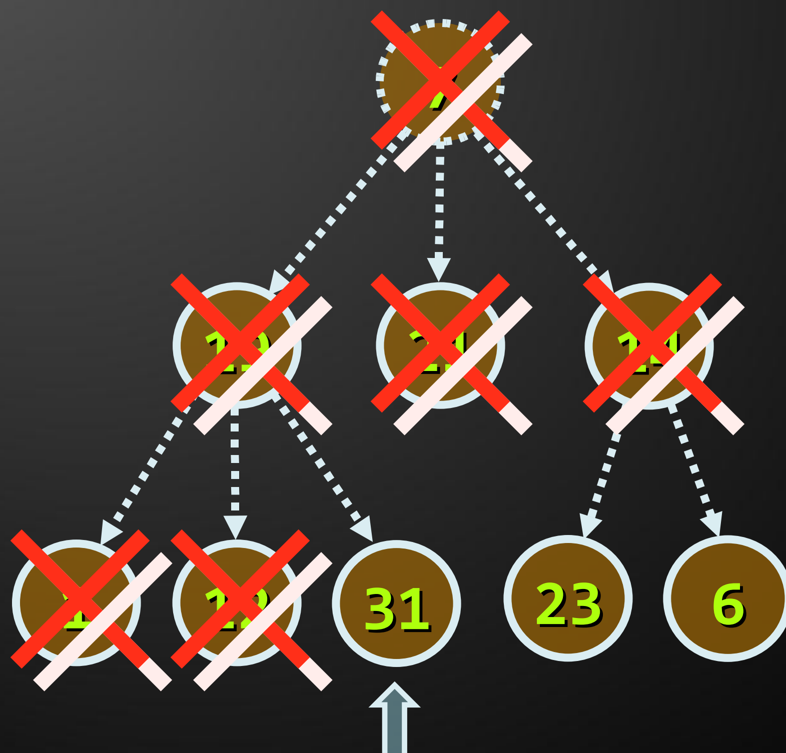
BFS in Action (Step 14)

- **Queue:** ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, 12, 31, 23, 6
- **Output:** 7, 19, 21, 14, 1, 12



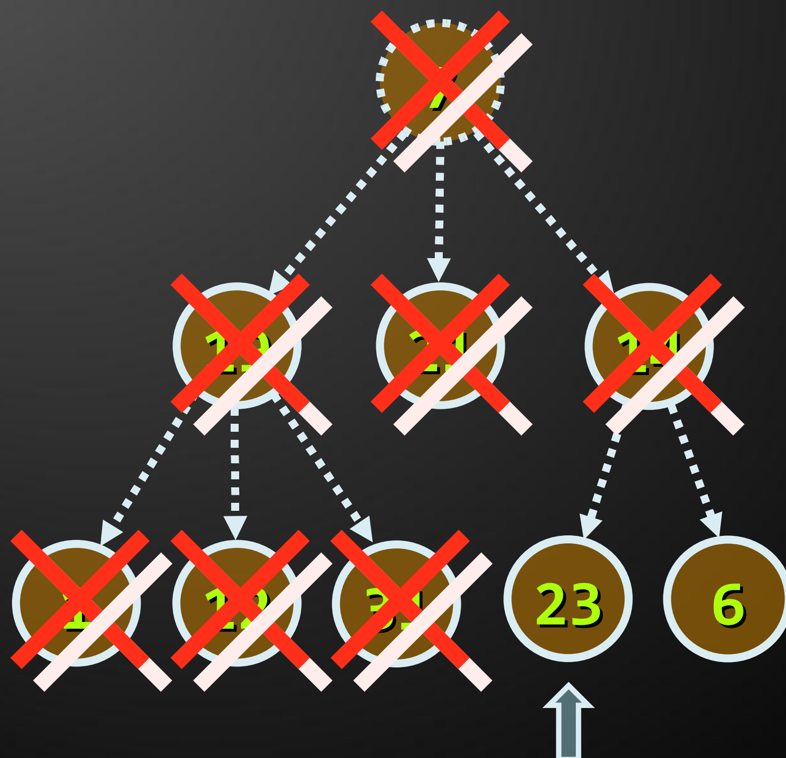
BFS in Action (Step 15)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, 31, 23, 6
- Output: 7, 19, 21, 14, 1, 12, 31



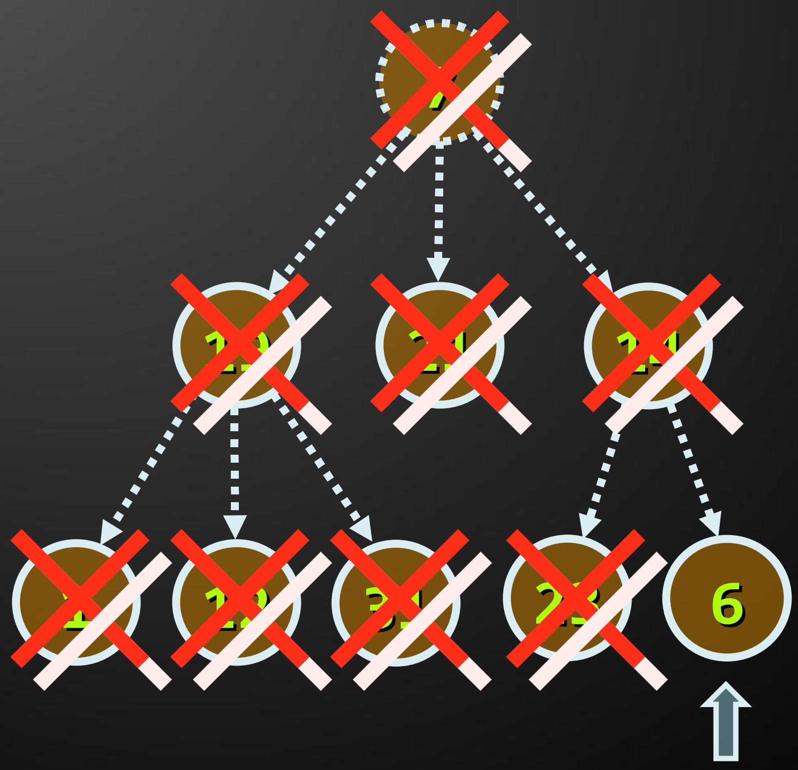
BFS in Action (Step 16)

- **Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, 23, 6**
- **Output: 7, 19, 21, 14, 1, 12, 31, 23**



BFS in Action (Step 16)

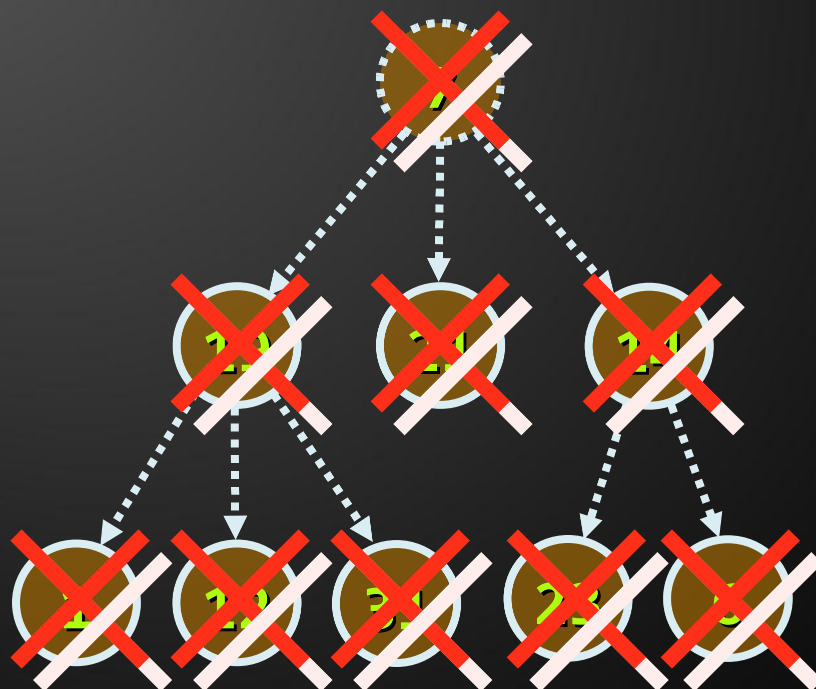
- **Queue:** ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, 6
- **Output:** 7, 19, 21, 14, 1, 12, 31, 23, 6



BFS in Action (Step 17)

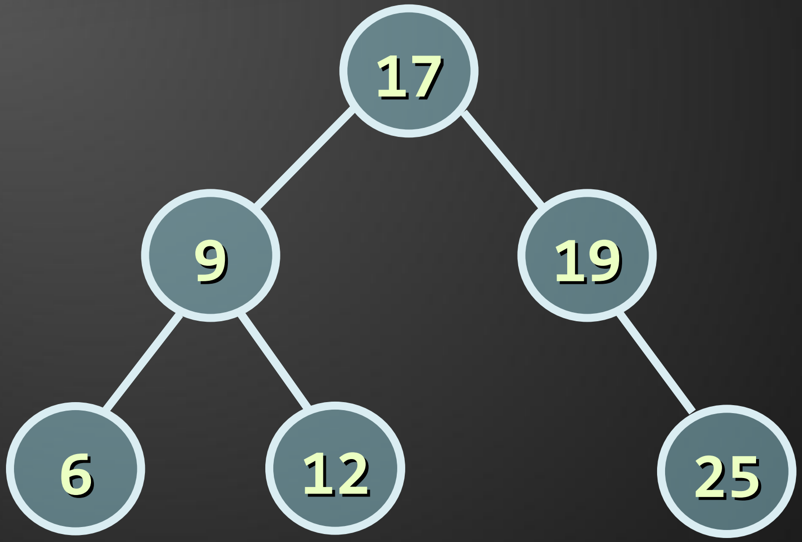
- **Queue:** ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~
- **Output:** 7, 19, 21, 14, 1, 12, 31, 23, 6

The queue is empty → stop



Binary Trees DFS Traversals

- DFS traversal of binary trees can be done in pre-order, in-order and post-order



  - Pre-order: left, root, right → 6, 9, 12, 17, 19, 25
  - In-order: root, left, right → 17, 9, 6, 12, 19, 25
  - Post-order: left, right, root → 6, 12, 9, 25, 19, 17

Iterative DFS and BFS

- What will happen if in the Breadth-First Search (BFS) algorithm a stack is used instead of queue?
  - An iterative Depth-First Search (DFS) – in-order

```
BFS(node)
{
  queue ← node
  while queue not empty
    v ← queue
    print v
    for each child c of v
      queue ← c
}
```

```
DFS(node)
{
  stack ← node
  while stack not empty
    v ← stack
    print v
    for each child c of v
      stack ← c
}
```

Trees and Traversals

Live Demo

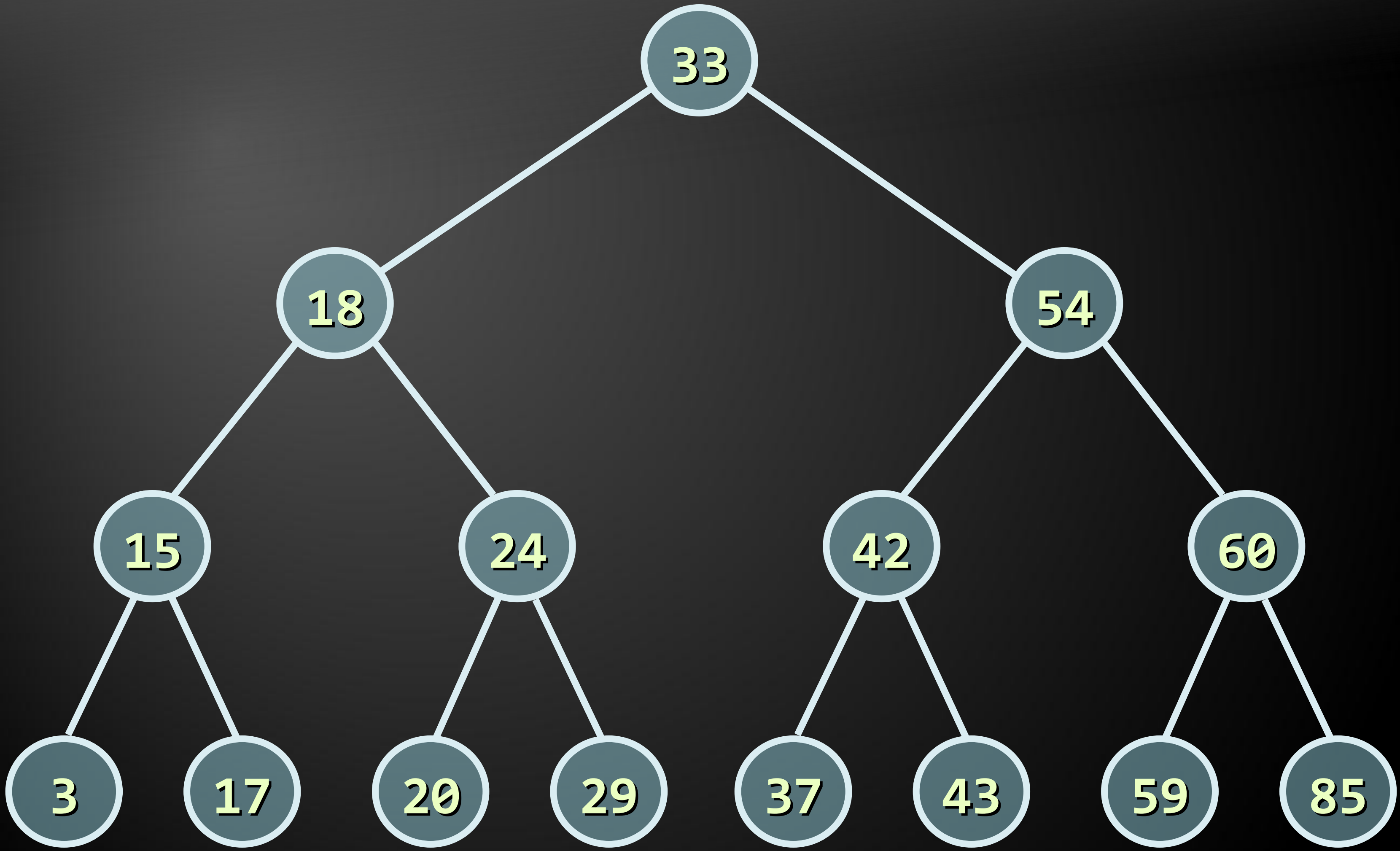
Balanced Search Trees

AVL Trees, B-Trees, Red-Black Trees, AA-Trees

Balanced Binary Search Trees

- Ordered Binary Trees (Binary Search Trees)
  - For each node x the left subtree has values ≤ x and the right subtree has values > x
- Balanced Trees
  - For each node its subtrees contain nearly equal number of nodes → nearly the same height
- Balanced Binary Search Trees
  - Ordered binary search trees that have height of $\log_2(n)$ where n is the number of their nodes
  - Searching costs about $\log_2(n)$ comparisons

Balanced Binary Search Tree – Example



Balanced Binary Search Trees

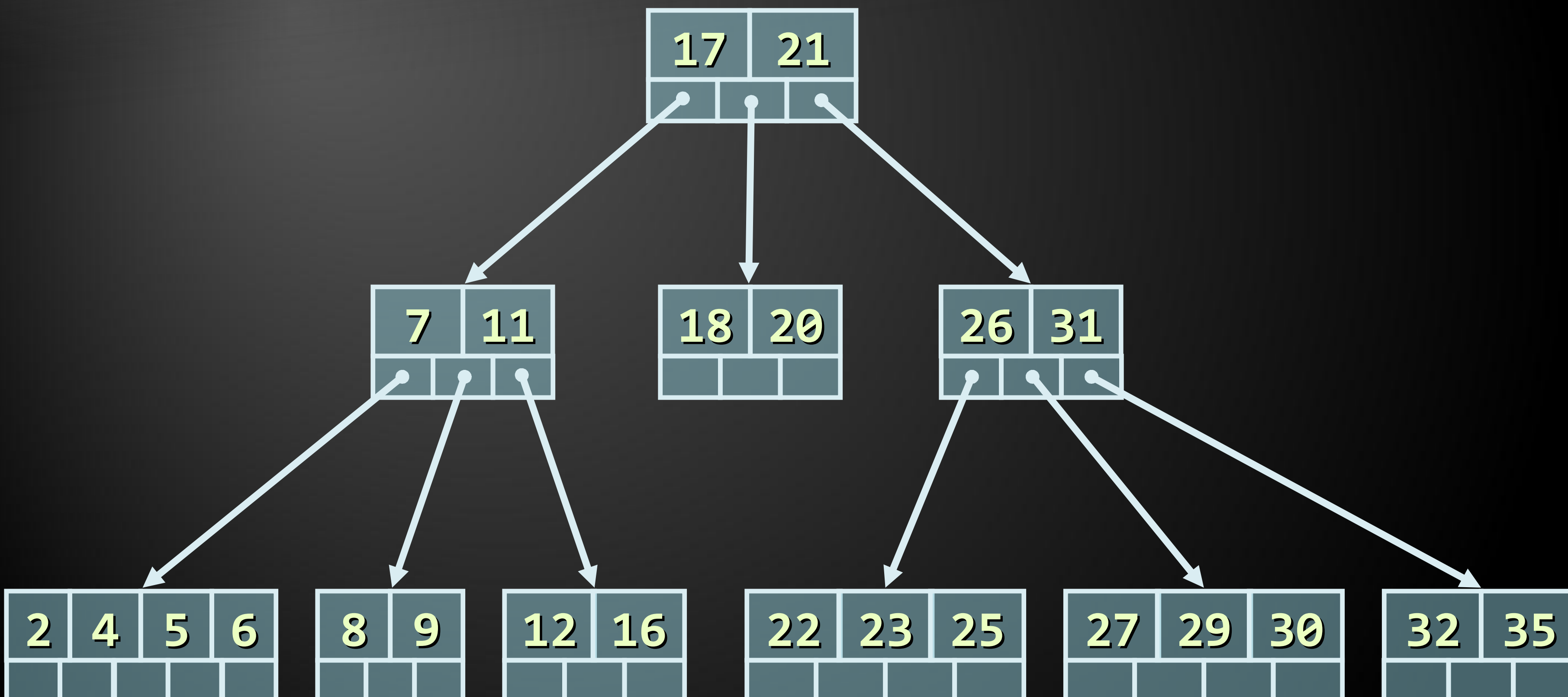
- Balanced binary search trees are hard to implement
  - Rebalancing the tree after insert / delete is complex
- Well known implementations of balanced binary search trees
  - AVL trees – ideally balanced, very complex
  - Red-black trees – roughly balanced, more simple
  - AA-Trees – relatively simple to implement
- Find / insert / delete operations need $\log_2(n)$ steps

B-Trees

- B-trees are generalization of the concept of ordered binary search trees
  - B-tree of order d has between d and 2*d keys in a node and between d+1 and 2*d+1 child nodes
  - The keys in each node are ordered increasingly
  - All keys in a child node have values between their left and right parent keys
- If the b-tree is balanced, its search / insert / add operations take about log(n) steps
- B-trees can be efficiently stored on the disk

B-Tree – Example

- B-Tree of order 2, also known as 2-3-4-tree:



Balanced Trees in .NET

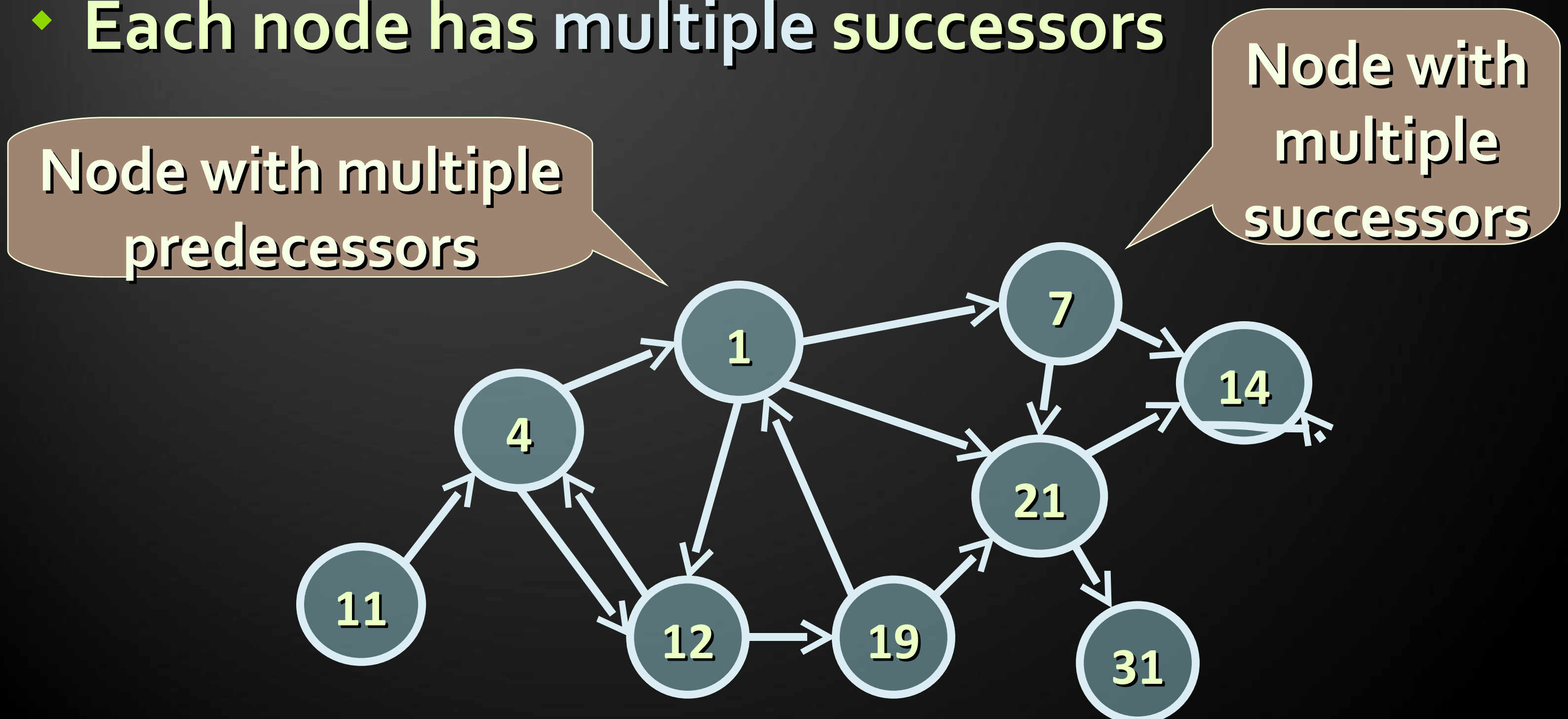
- .NET Framework has several built-in implementations of balanced search trees:
  - `SortedDictionary<K,V>`
    - Red-black tree based map of key-value pairs
  - `OrderedSet<T>`
    - Red-black tree based set of elements
- External libraries like "Wintellect Power Collections for .NET" are more flexible
  - http://powercollections.codeplex.com

Graphs

Definitions, Representation, Traversal Algorithms

Graph Data Structure

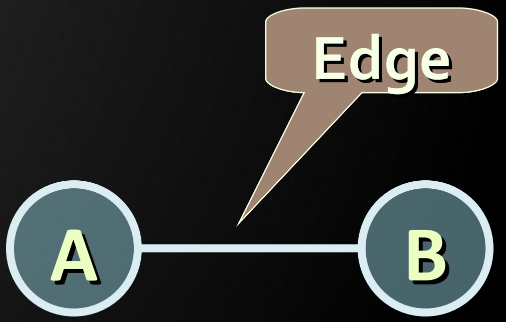
- Set of nodes with many-to-many relationship between them is called graph
  - Each node has multiple predecessors
  - Each node has multiple successors



Graph Definitions

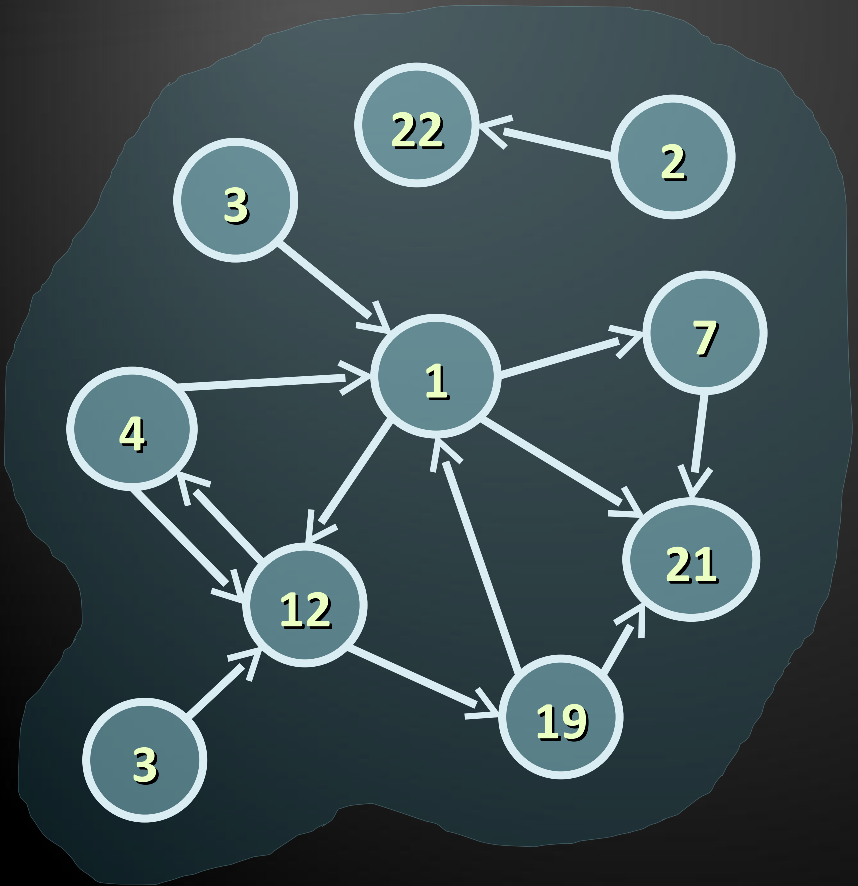
- Node (vertex)
  - Element of graph
  - Can have name or value
  - Keeps a list of adjacent nodes

- Edge
  - Connection between two nodes
  - Can be directed / undirected
  - Can be weighted / unweighted
  - Can have name / value

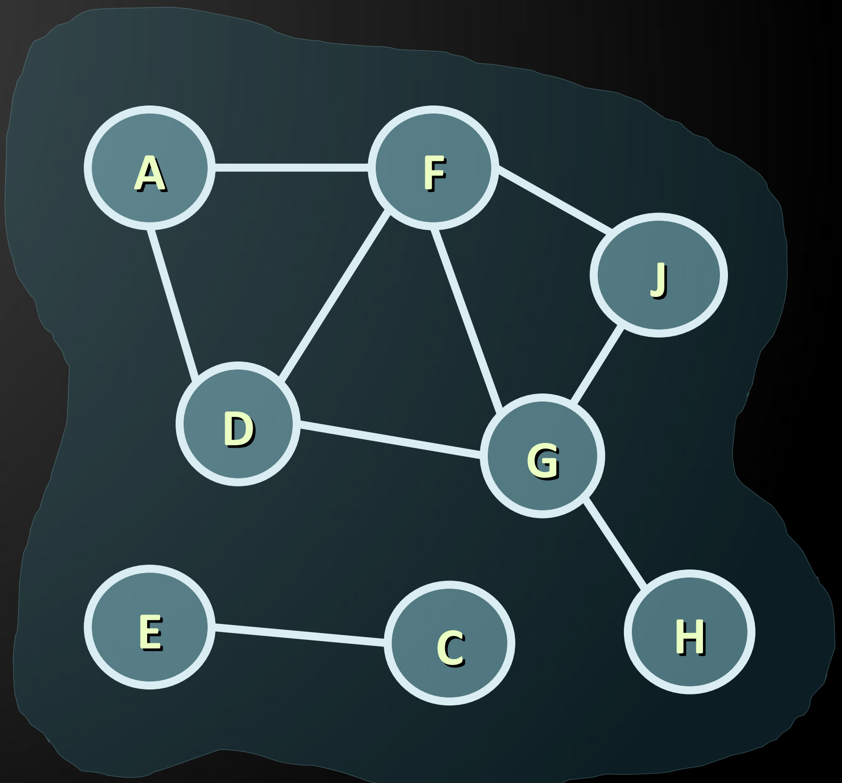


Graph Definitions (2)

- Directed graph
  - Edges have direction

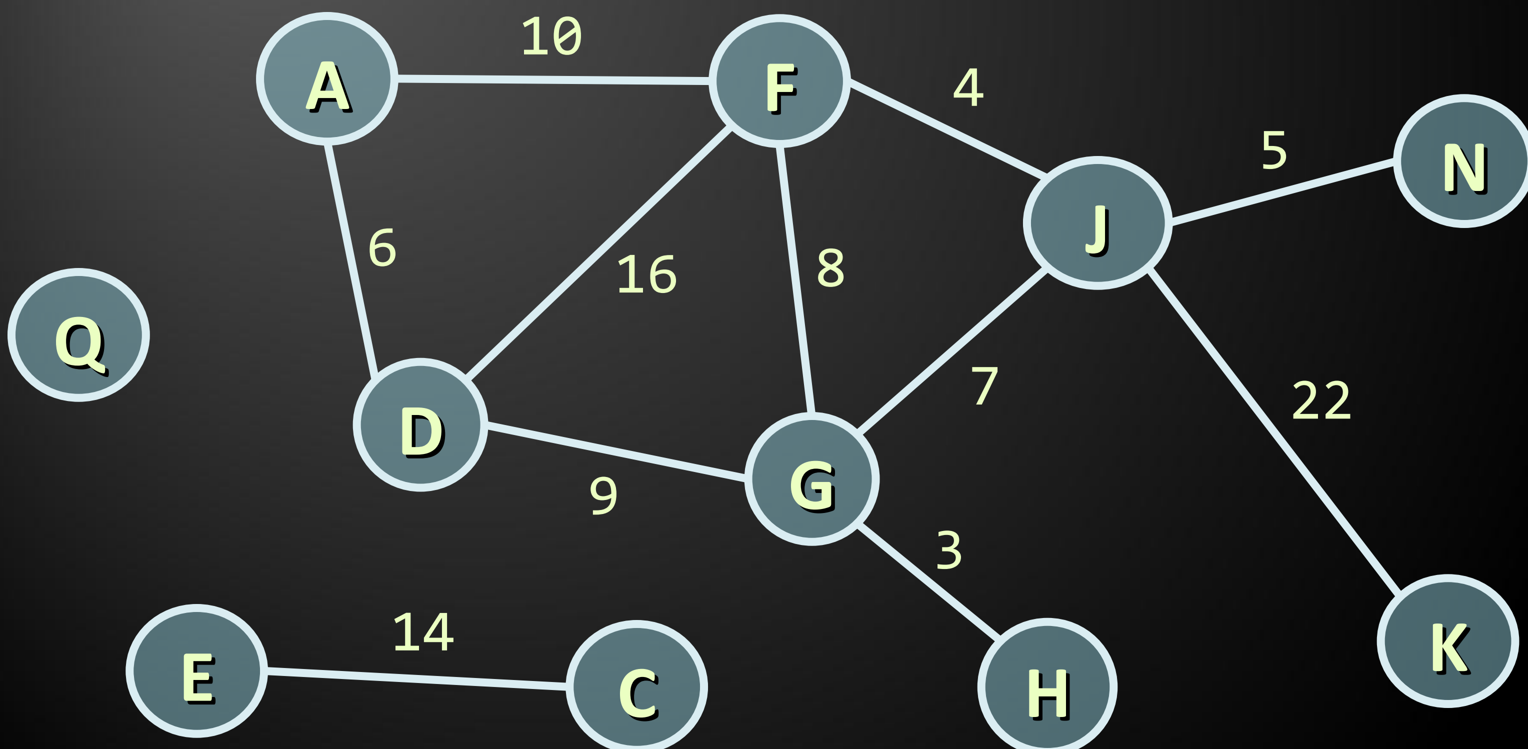


- Undirected graph
  - Undirected edges



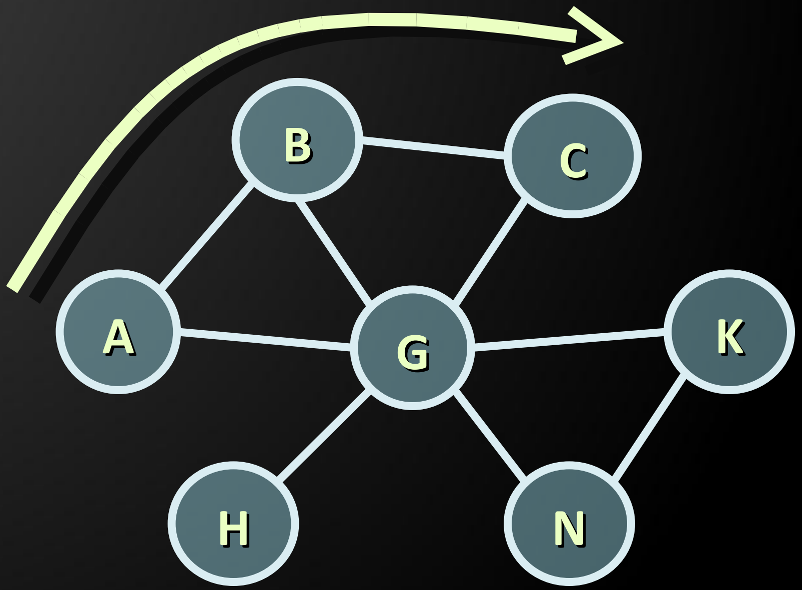
Graph Definitions (3)

- **Weighted graph**
  - **Weight (cost) is associated with each edge**



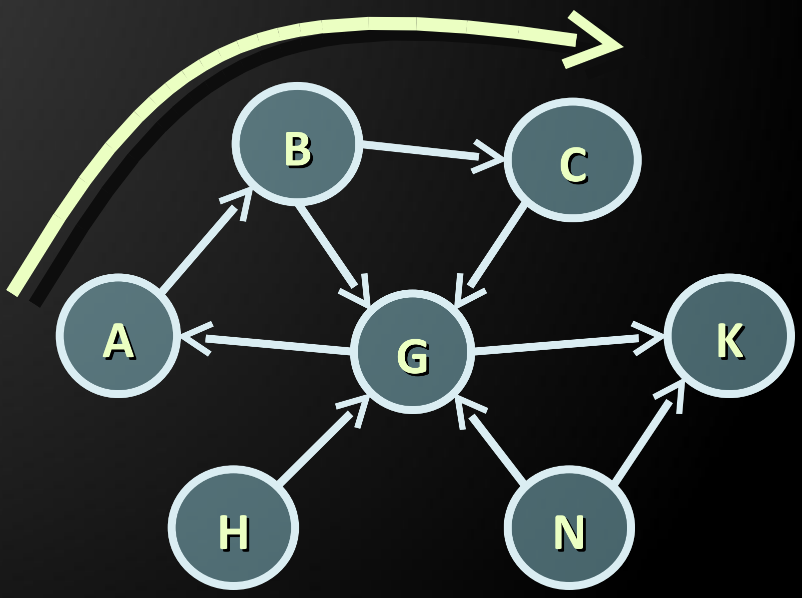
Graph Definitions (4)

- Path (in undirected graph)
  - Sequence of nodes n_1, n_2, … n_k
  - Edge exists between each pair of nodes n_i, n_{i+1}
  - Examples:
    - A, B, C is a path
    - H, K, C is not a path



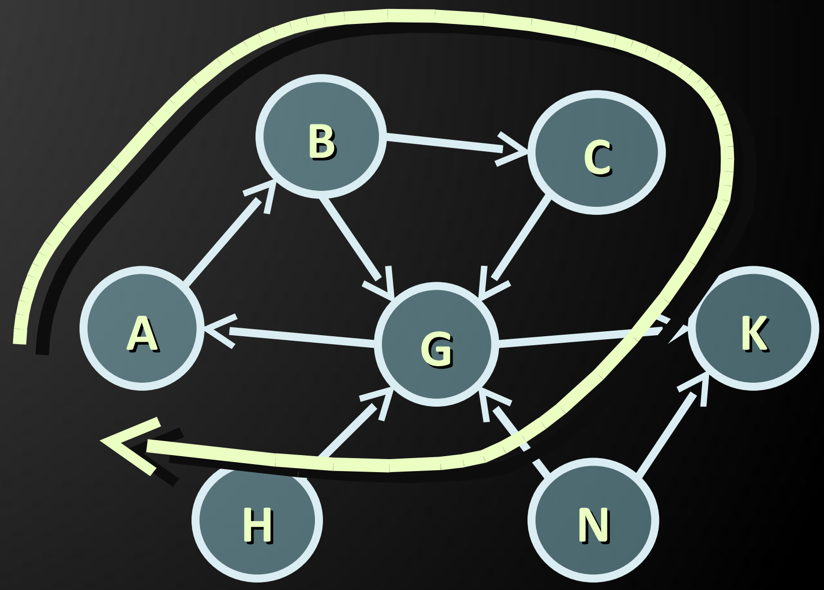
Graph Definitions (5)

- Path (in directed graph)
  - Sequence of nodes $n_1, n_2, \ldots n_k$
  - Directed edge exists between each pair of nodes n_i, n_{i+1}
  - Examples:
    - A, B, C is a path
    - A, G, K is not a path



Graph Definitions (6)

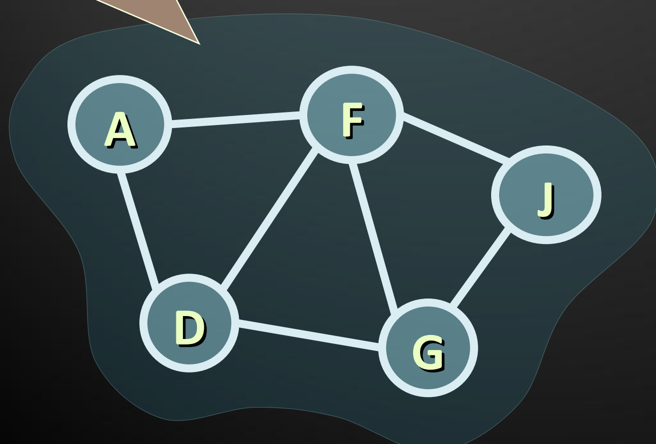
- Cycle
  - Path that ends back at the starting node
  - Example:
    - A, B, C, G, A
- Simple path
  - No cycles in path
- Acyclic graph
  - Graph with no cycles
  - Acyclic undirected graphs are trees



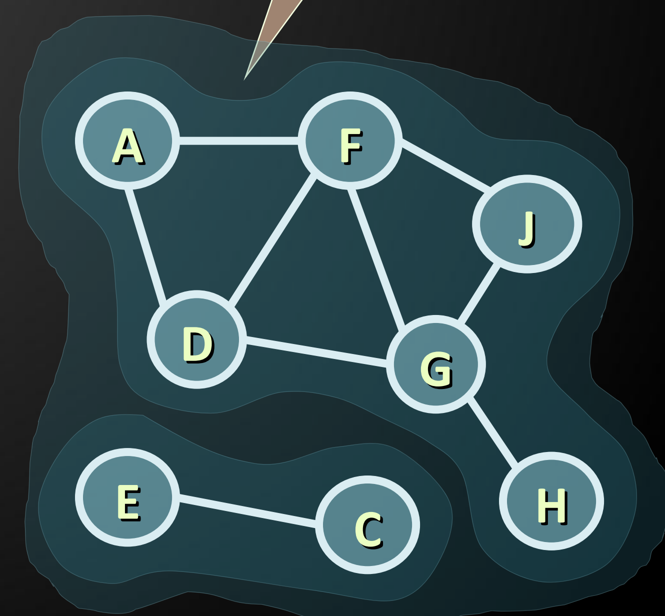
Graph Definitions (8)

- Two nodes are reachable if
  - Path exists between them
- Connected graph
  - Every node is reachable from any other node

Connected graph

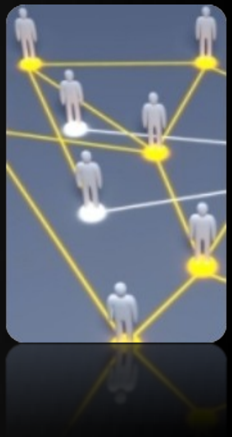


Unconnected graph with two connected components



Graphs and Their Applications

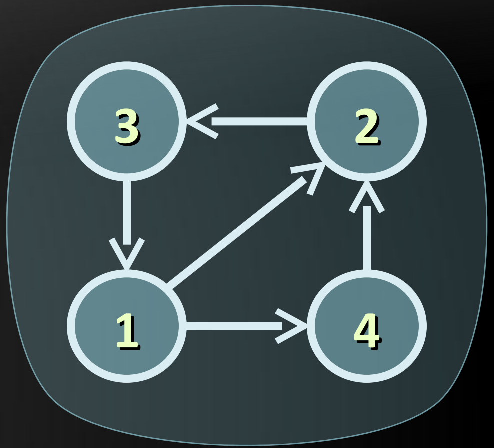
- Graphs have many real-world applications
  - Modeling a computer network like Internet
    - Routes are simple paths in the network
  - Modeling a city map
    - Streets are edges, crossings are vertices
  - Social networks
    - People are nodes and their connections are edges
  - State machines
    - States are nodes, transitions are edges



Representing Graphs

- Adjacency list
  - Each node holds a list of its neighbors

```
1 → {2, 4}
2 → {3}
3 → {1}
4 → {2}
```



- Adjacency matrix
  - Each cell keeps whether and how two nodes are connected

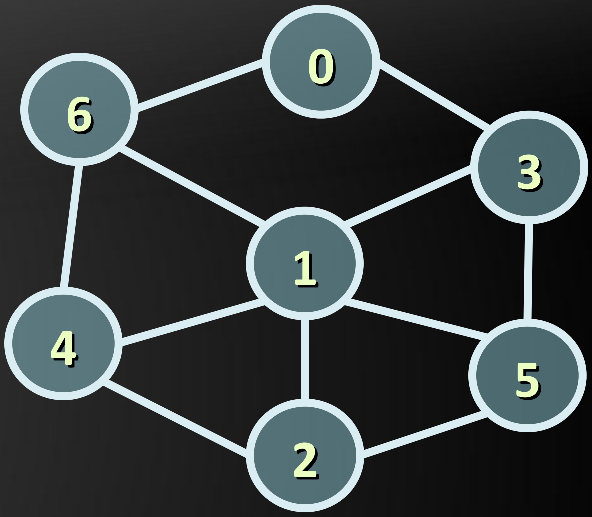
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	1	0	0	0
4	0	1	0	0

- Set of edges

```
{1,2} {1,4} {2,3} {3,1} {4,2}
```

Representing Graphs in C#

```
public class Graph
{
  int[][] childNodes;
  public Graph(int[][] nodes)
  {
    this.childNodes = nodes;
  }
}
```




```
Graph g = new Graph(new int[][] {
  new int[] {3, 6}, // successors of vertice 0
  new int[] {2, 3, 4, 5, 6}, // successors of vertice 1
  new int[] {1, 4, 5}, // successors of vertice 2
  new int[] {0, 1, 5}, // successors of vertice 3
  new int[] {1, 2, 6}, // successors of vertice 4
  new int[] {1, 2, 3}, // successors of vertice 5
  new int[] {0, 1, 4}  // successors of vertice 6
});
```

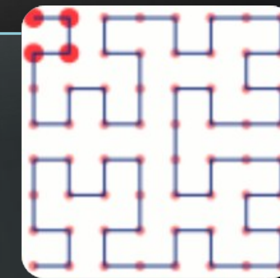
Graph Traversal Algorithms

- **Depth-First Search (DFS) and Breadth-First Search (BFS) can traverse graphs**
  - **Each vertex should be is visited at most once**

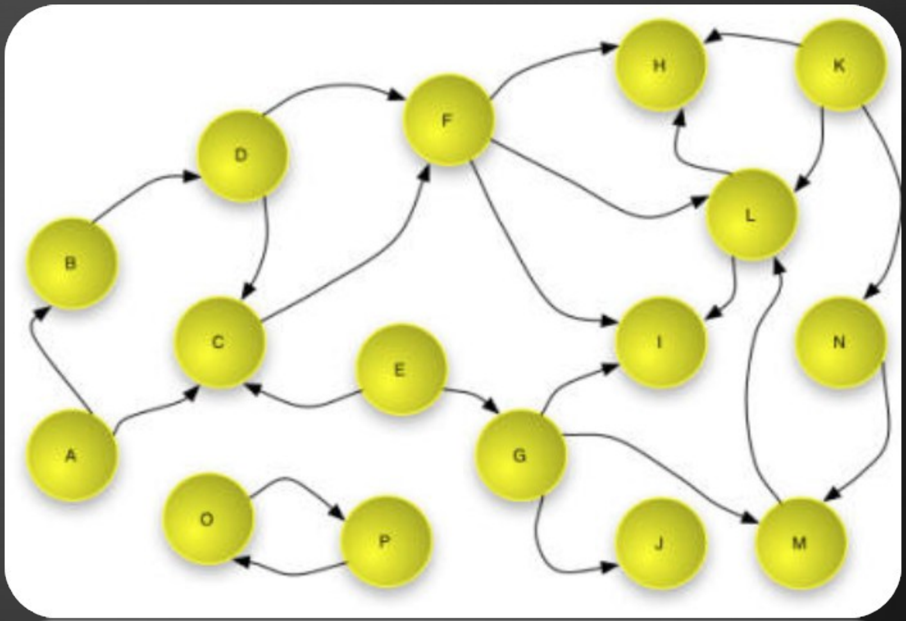
```
BFS(node)
{
  queue ← node
  visited[node] = true
  while queue not empty
    v ← queue
    print v
    for each child c of v
      if not visited[c]
        queue ← c
        visited[c] = true
}
```

```
DFS(node)
{
  stack ← node
  visited[node] = true
  while stack not empty
    v ← stack
    print v
    for each child c of v
      if not visited[c]
        stack ← c
        visited[c] = true
}
```

Recursive DFS Graph Traversal



```
void TraverseDFSRecursive(node)
{
  if (not visited[node])
  {
    visited[node] = true
    print node
    foreach child node c of node
    {
       TraverseDFSRecursive(c);
    }
  }
}

vois Main()
{
  TraverseDFS(firstNode);
}
```



Graphs and Traversals

Live Demo

Summary

- Trees are recursive data structure – node with set of children which are also nodes
- Binary Search Trees are ordered binary trees
- Balanced trees have weight of log(n)
- Graphs are sets of nodes with many-to-many relationship between them
  - Can be directed/undirected, weighted / unweighted, connected / not connected, etc.
- Tree / graph traversals can be done by Depth-First Search (DFS) and Breadth-First Search (BFS)

Questions?

Exercises

1. Write a program to traverse the directory `C:\WINDOWS` and all its subdirectories recursively and to display all files matching the mask `*.exe`. Use the class `System.IO.Directory`.
2. Define classes `File { string name, int size }` and `Folder { string name, File[] files, Folder[] childFolders }` and using them build a tree keeping all files and folders on the hard drive starting from `C:\WINDOWS`. Implement a method that calculates the sum of the file sizes in given subtree of the tree and test it accordingly. Use recursive DFS traversal.

Exercises (2)

1. Implement the recursive Depth-First-Search (DFS) traversal algorithm. Test it with the sample graph from the demonstrations.
2. Implement the queue-based Breath-First-Search (BFS) traversal algorithm. Test it with the sample graph from the demonstrations.
3. Write a program for finding all cycles in given undirected graph using recursive DFS.
4. Write a program for finding all connected components of given undirected graph. Use a sequence of DFS traversals.

Exercises (3)

- Write a program for finding the shortest path between two vertices in a weighted directed graph. Hint: Use the Dijkstra's algorithm.
- We are given a set of N tasks that should be executed in a sequence. Some of the tasks depend on other tasks. We are given a list of tasks { t_i, t_j} where t_j depends on the result of t_i and should be executed after it. Write a program that arranges the tasks in a sequence so that each task depending on another task is executed after it. If such arrangement is impossible indicate this fact.

  Example: {1, 2}, {2, 5}, {2, 4}, {3, 1} → 3, 1, 2, 5, 4