

# **SD2x2.1**

# **Motivating Advanced Data Structures**

# **Chris**

# Data Structures: Common operations

---

- Add an element (possibly at a certain location)
- Remove an element (possibly from a certain location)
- Retrieve an element from a given location
- Determine whether an element is in the data structure
- Retrieve the elements in sorted order
- Retrieve the elements in the order in which they were added

# The simplest data structure: Arrays

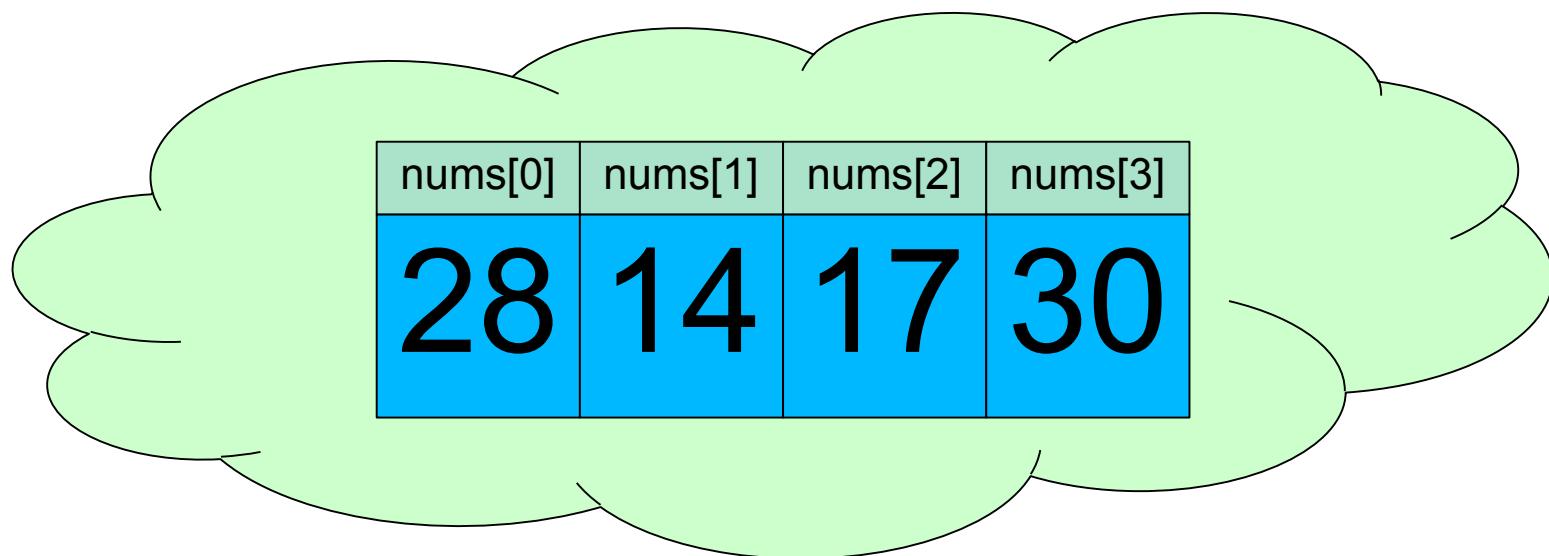
---

nums[0]	nums[1]	nums[2]	nums[3]
28	14	17	30

- Advantage:
  - simple to implement
  - $O(1)$  access to element at given position
- Disadvantages:
  - need to know number of elements in advance
  - most operations are  $O(n)$

# ArrayLists

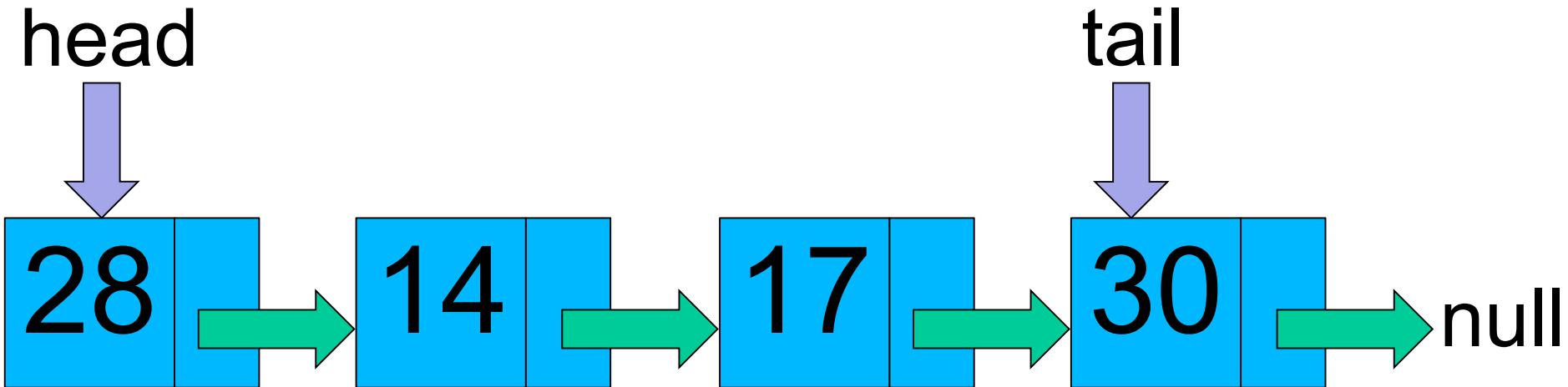
---



- Advantage:
  - automatically handle resizing of underlying array
  - still have  $O(1)$  access to element at given position
- Disadvantages:
  - most operations are still  $O(n)$

# LinkedLists

---



- Advantage:
  - many operations are  $O(1)$
  - efficient in terms of space usage
- Disadvantages:
  - now have  $O(n)$  access to element at given position

# Data Structures: Common operations

---

- Add an element (possibly at a certain location)
- Remove an element (possibly from a certain location)
- Retrieve an element from a given location
- Determine whether an element is in the data structure
- Retrieve the elements in sorted order
- Retrieve the elements in the order in which they were added

# Data Structures: Common operations

---

- Add an element (possibly at a certain location)
- Remove an element (possibly from a certain location)
- Retrieve an element from a given location
- Determine whether an element is in the data structure
- Retrieve the elements in sorted order
- Retrieve the elements in the order in which they were added

# Data Structures in Java

---

- **Lists**

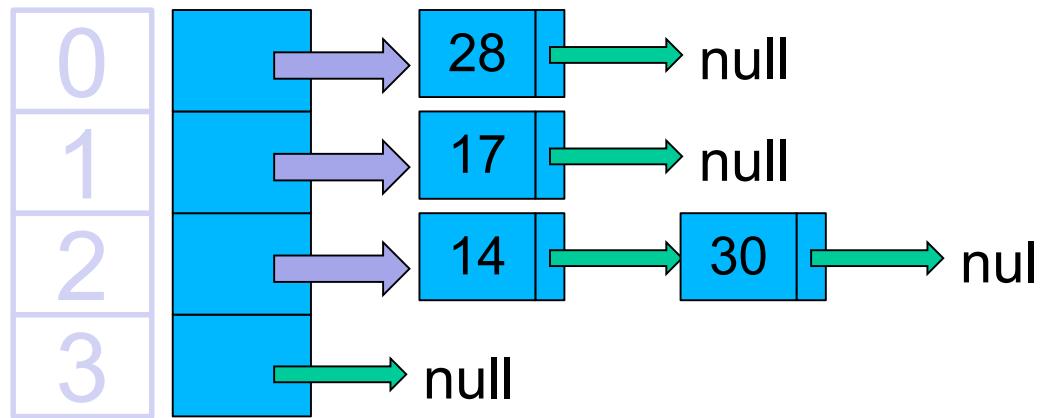
- ordered collection of elements
- allow for duplicates
- examples: ArrayList, LinkedList

- **Sets**

- (potentially) unordered collection of elements
- do not allow for duplicates
- example: HashSet

# HashSets

---



- Advantage:
  - all operations are  $O(1)$  assuming sufficient number of buckets and distribution of hash codes
- Disadvantages:
  - ordering of elements is not (necessarily) maintained
  - potentially wasted space if too many buckets

# Data Structures: Common operations

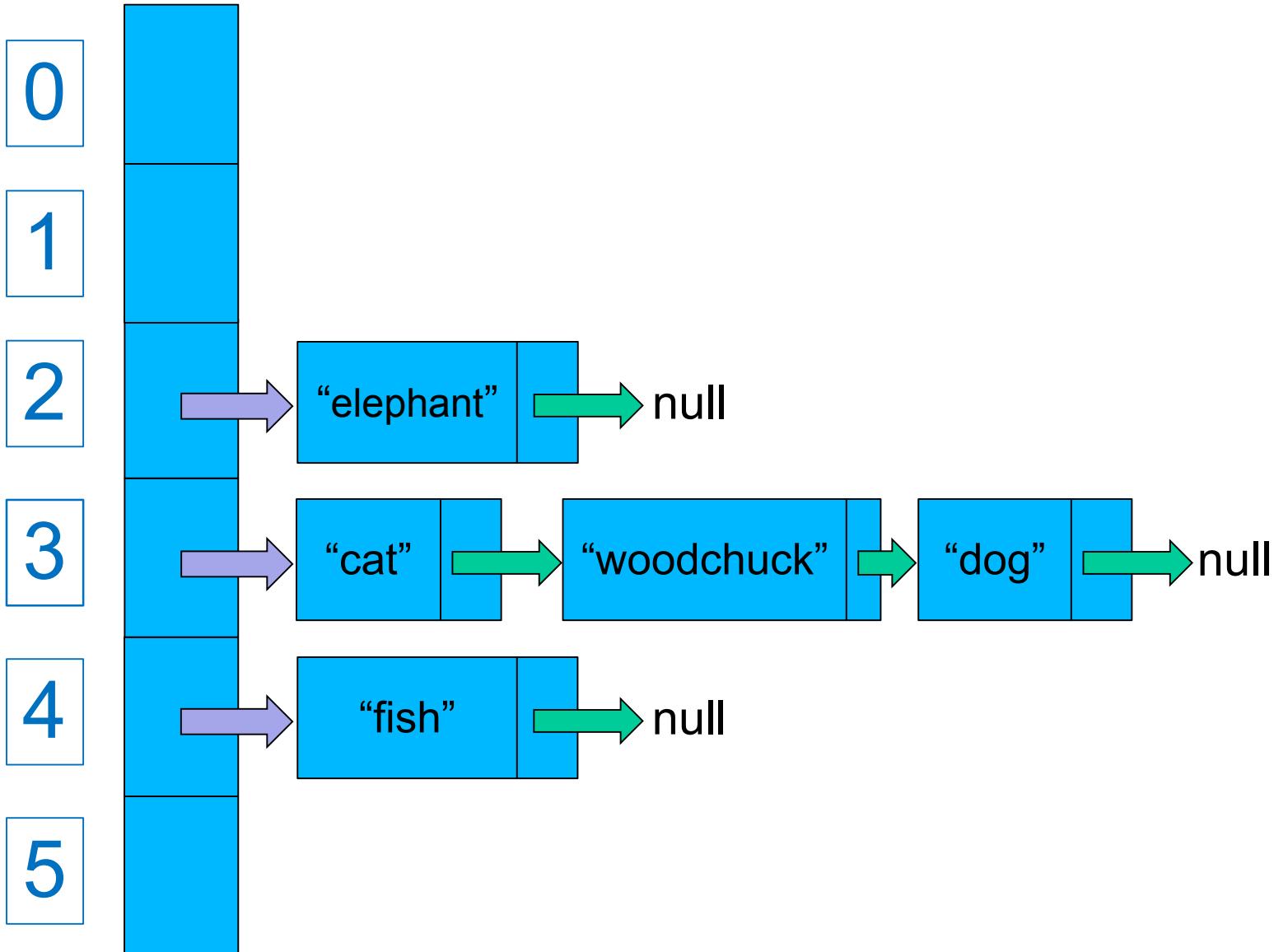
---

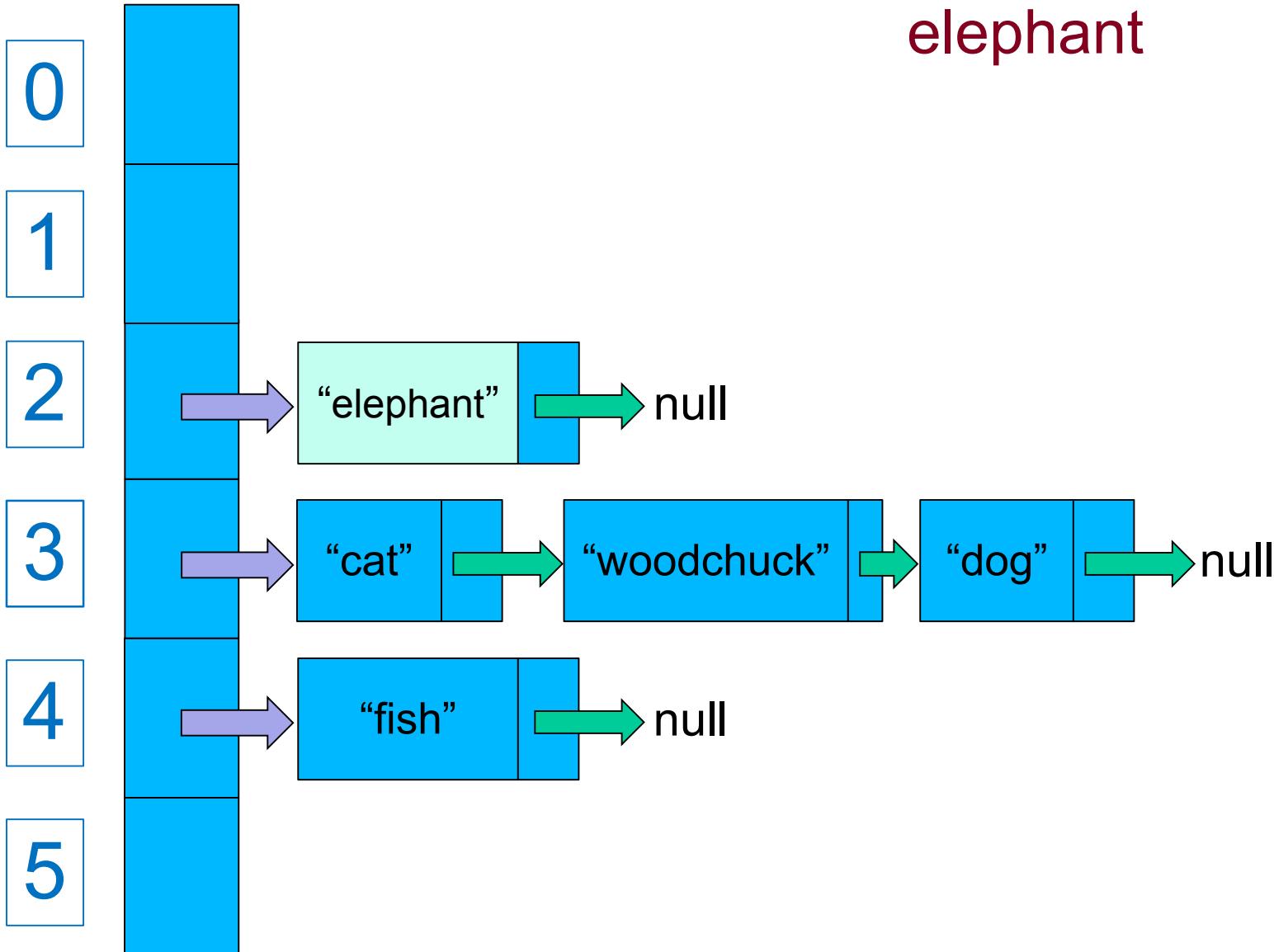
- Add an element (possibly at a certain location)
- Remove an element (possibly from a certain location)
- Retrieve an element from a given location
- Determine whether an element is in the data structure
- Retrieve the elements in sorted order
- Retrieve the elements in the order in which they were added

# Data Structures: Common operations

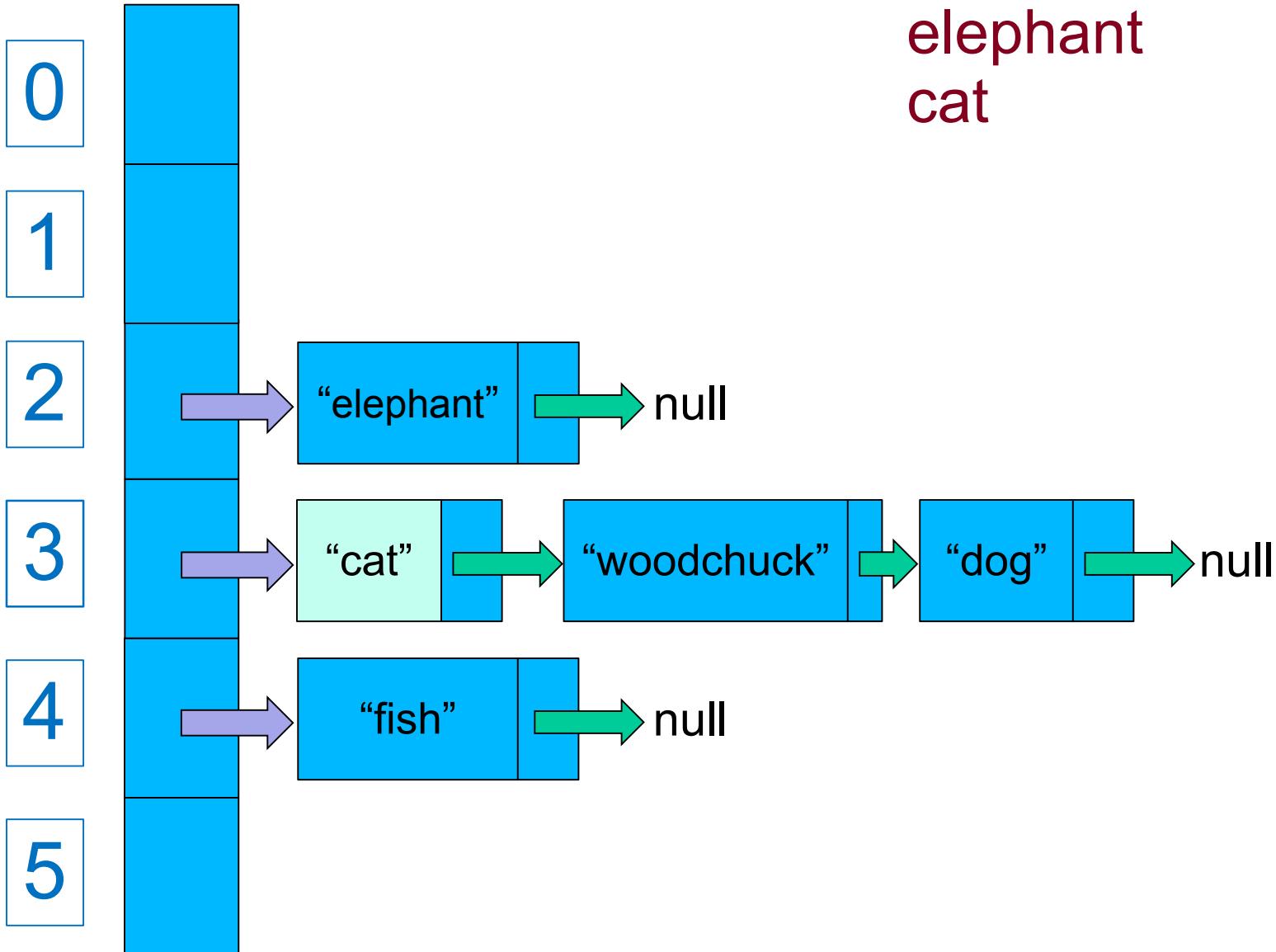
---

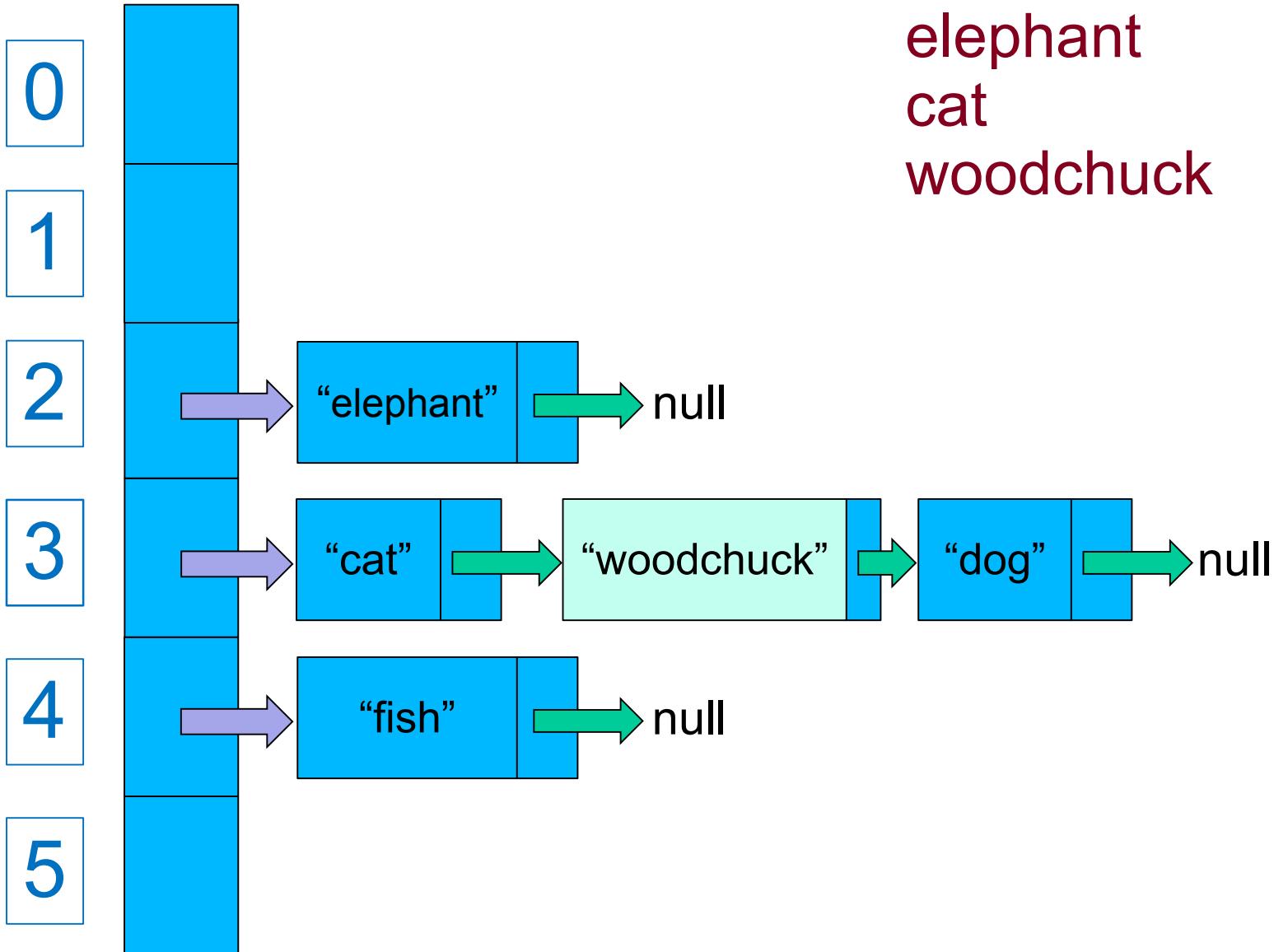
- Add an element (possibly at a certain location)
- Remove an element (possibly from a certain location)
- Retrieve an element from a given location
- Determine whether an element is in the data structure
- Retrieve the elements in sorted order
- Retrieve the elements in the order in which they were added



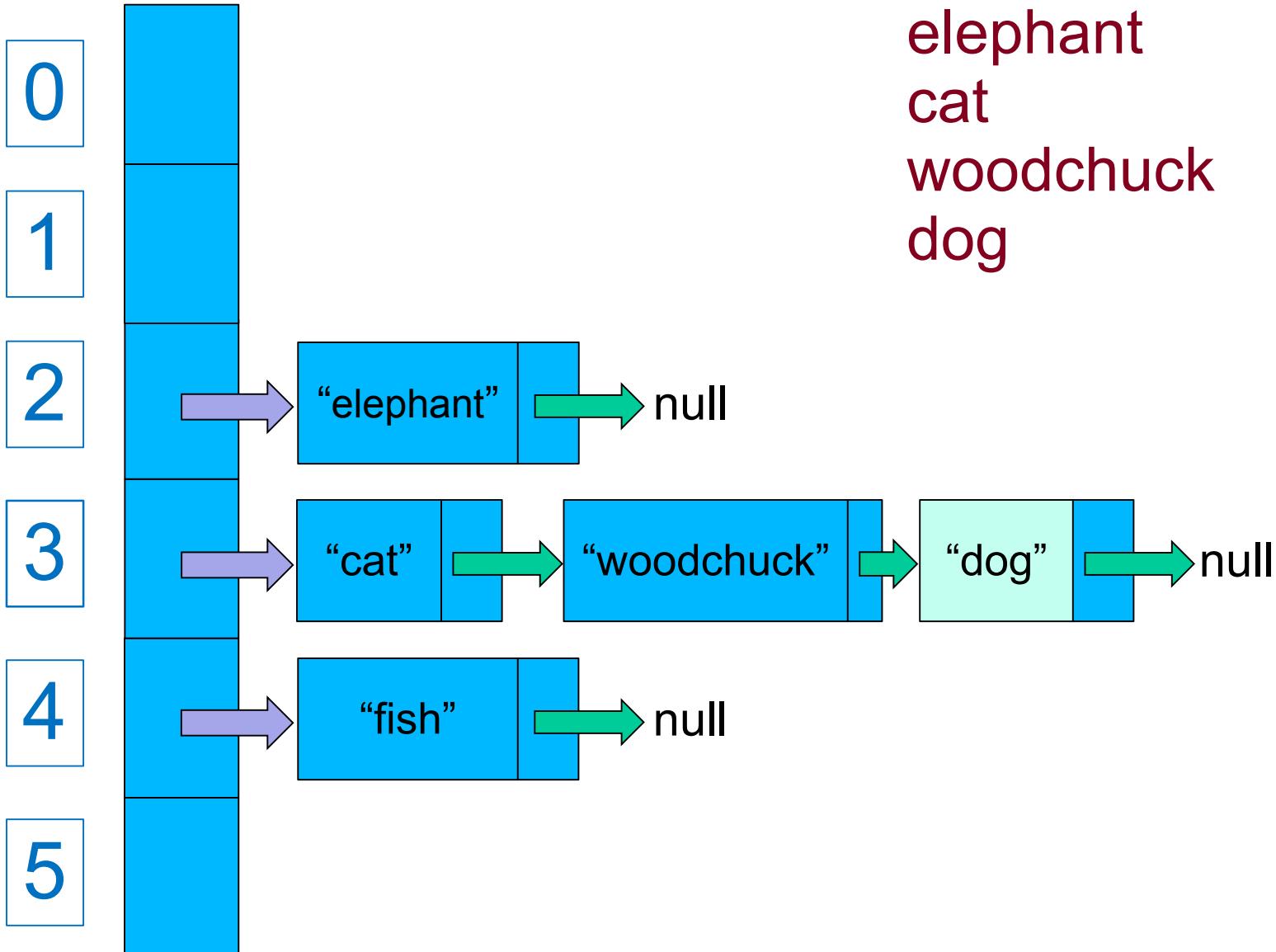


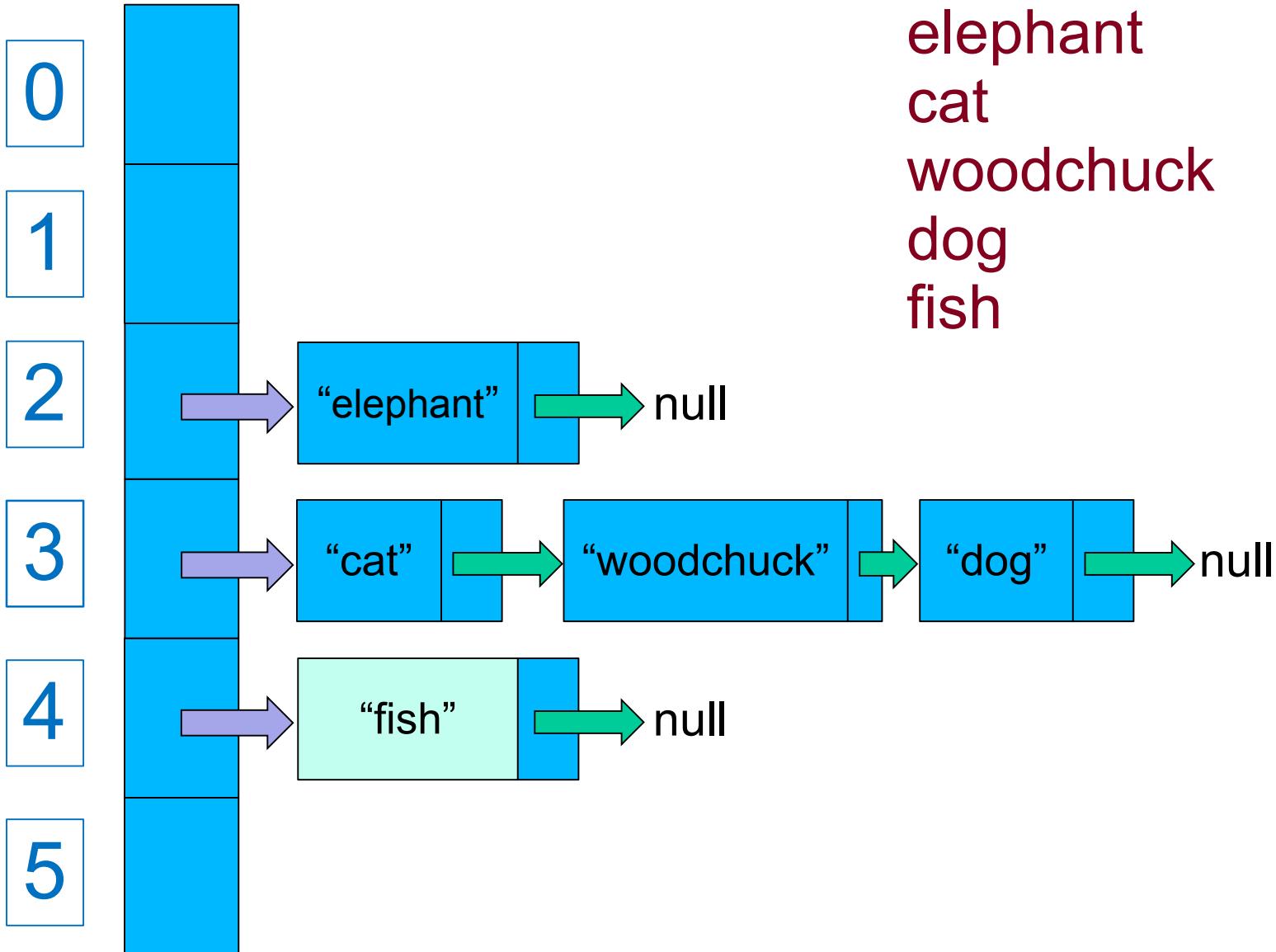
elephant

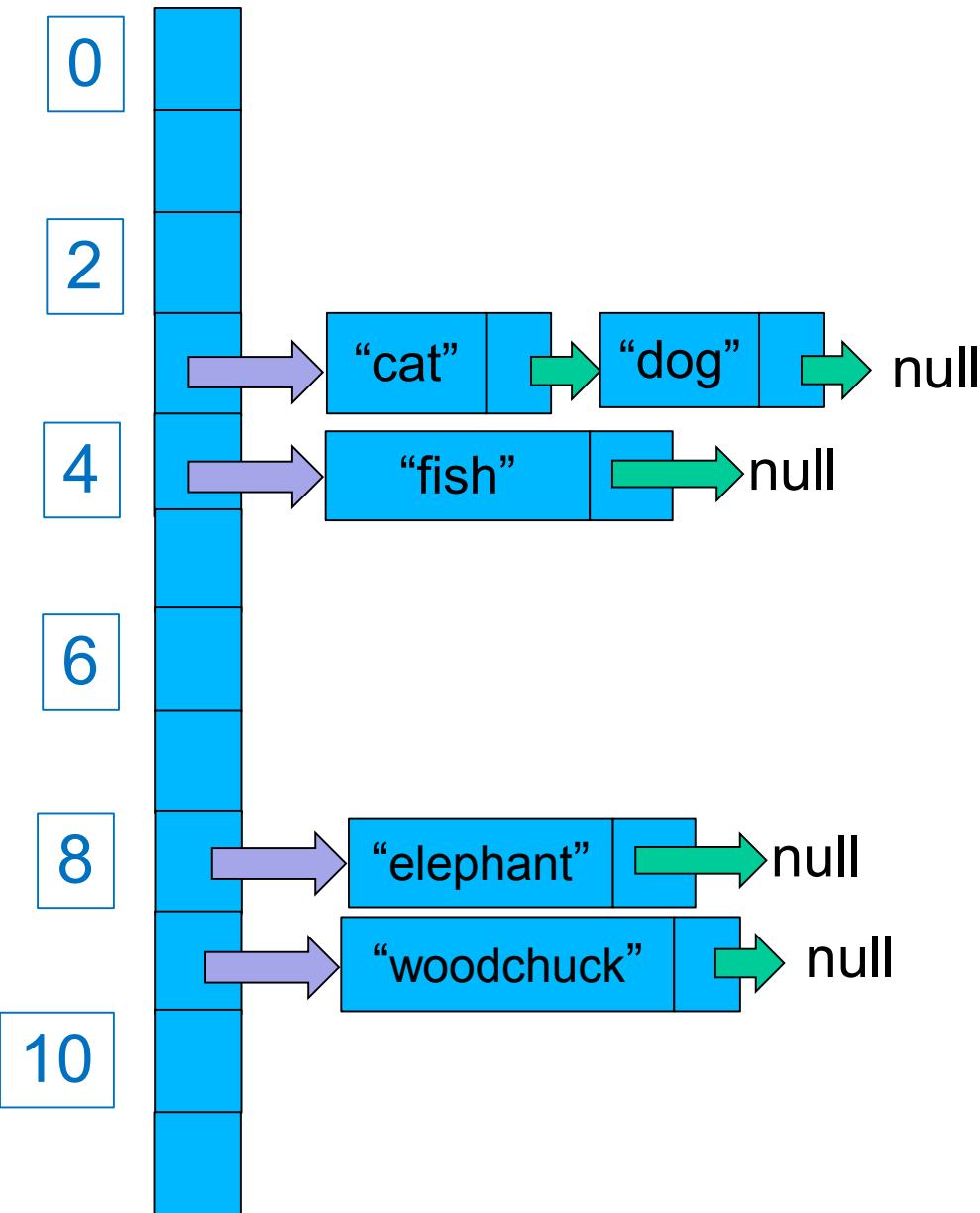


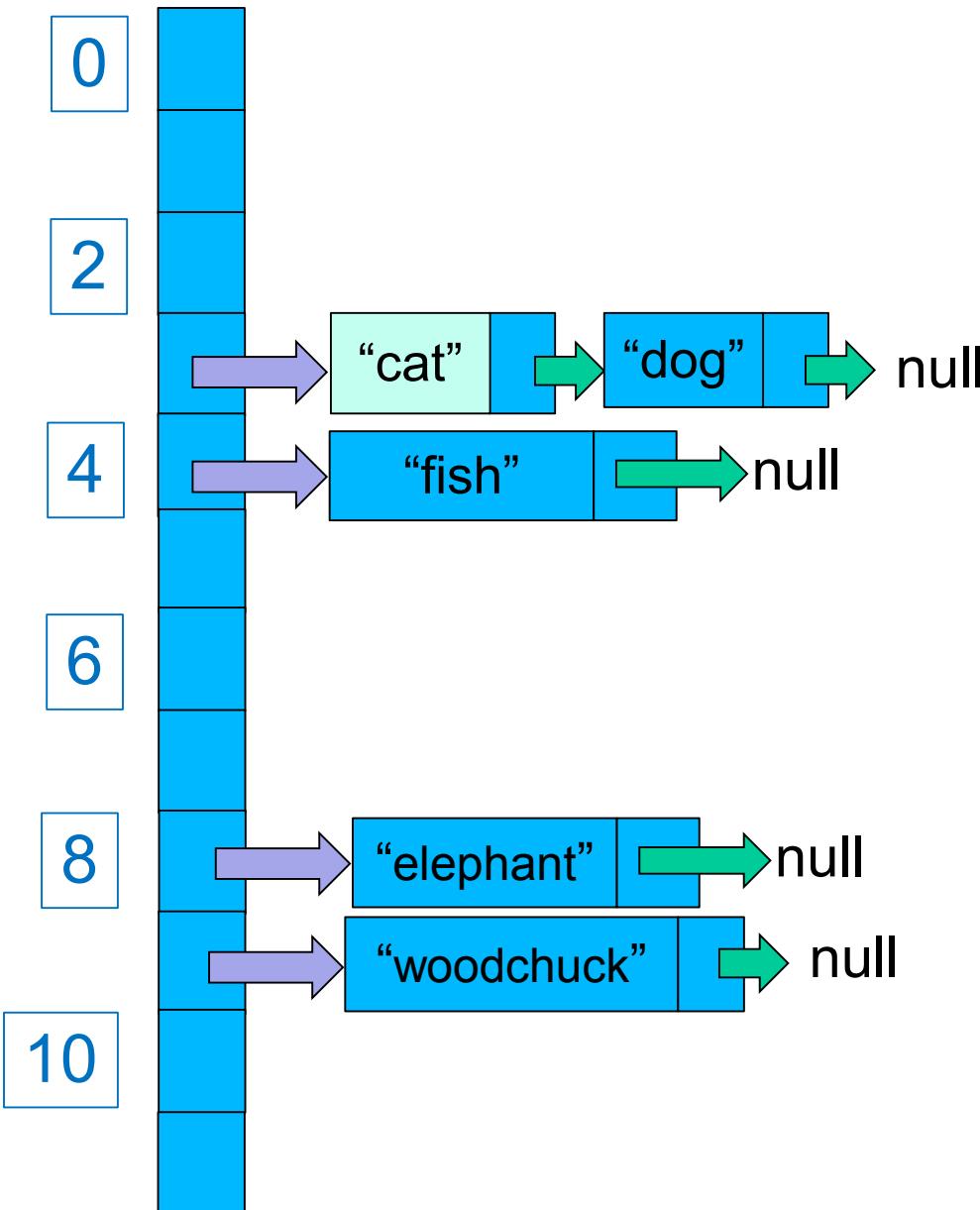


elephant  
cat  
woodchuck

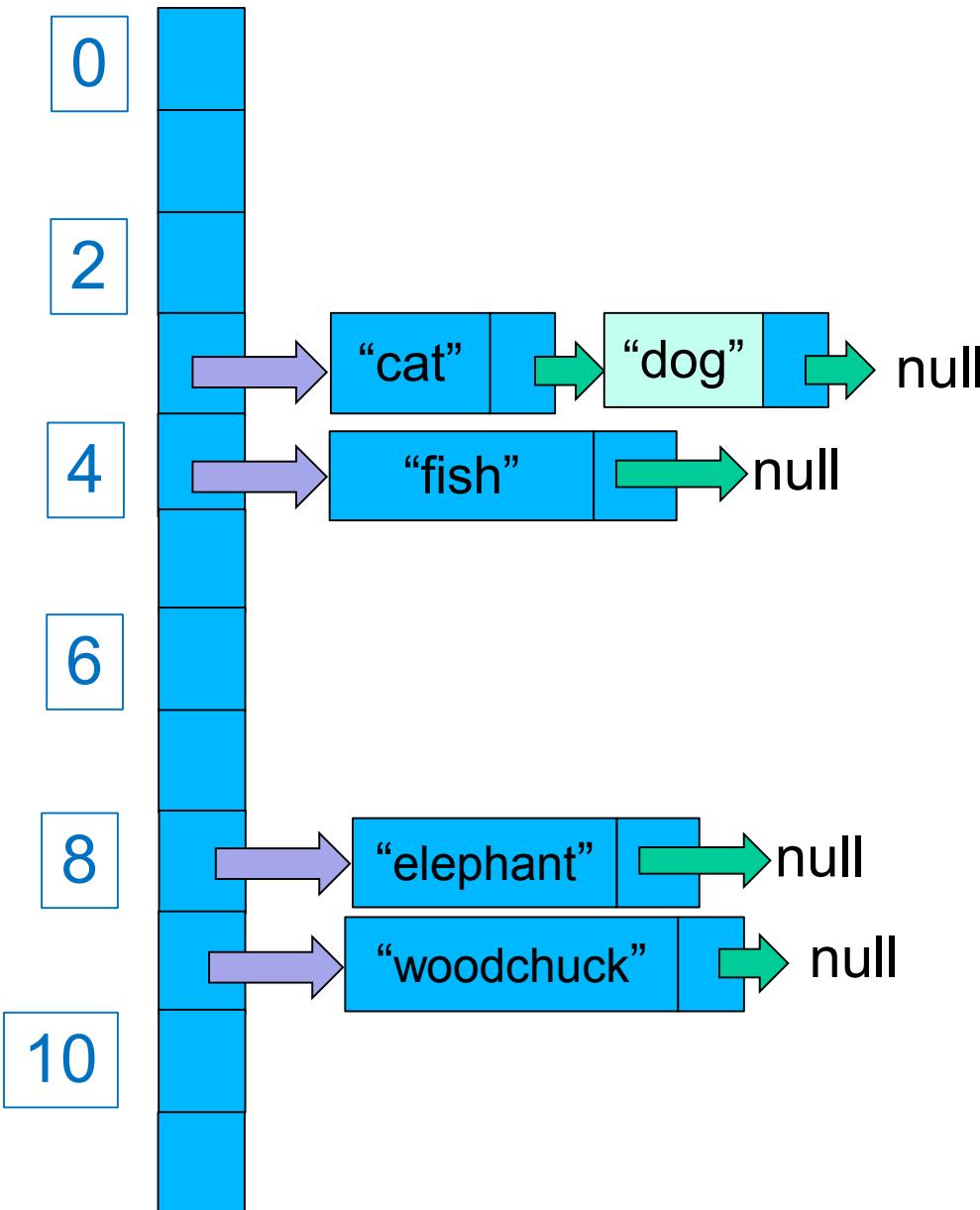




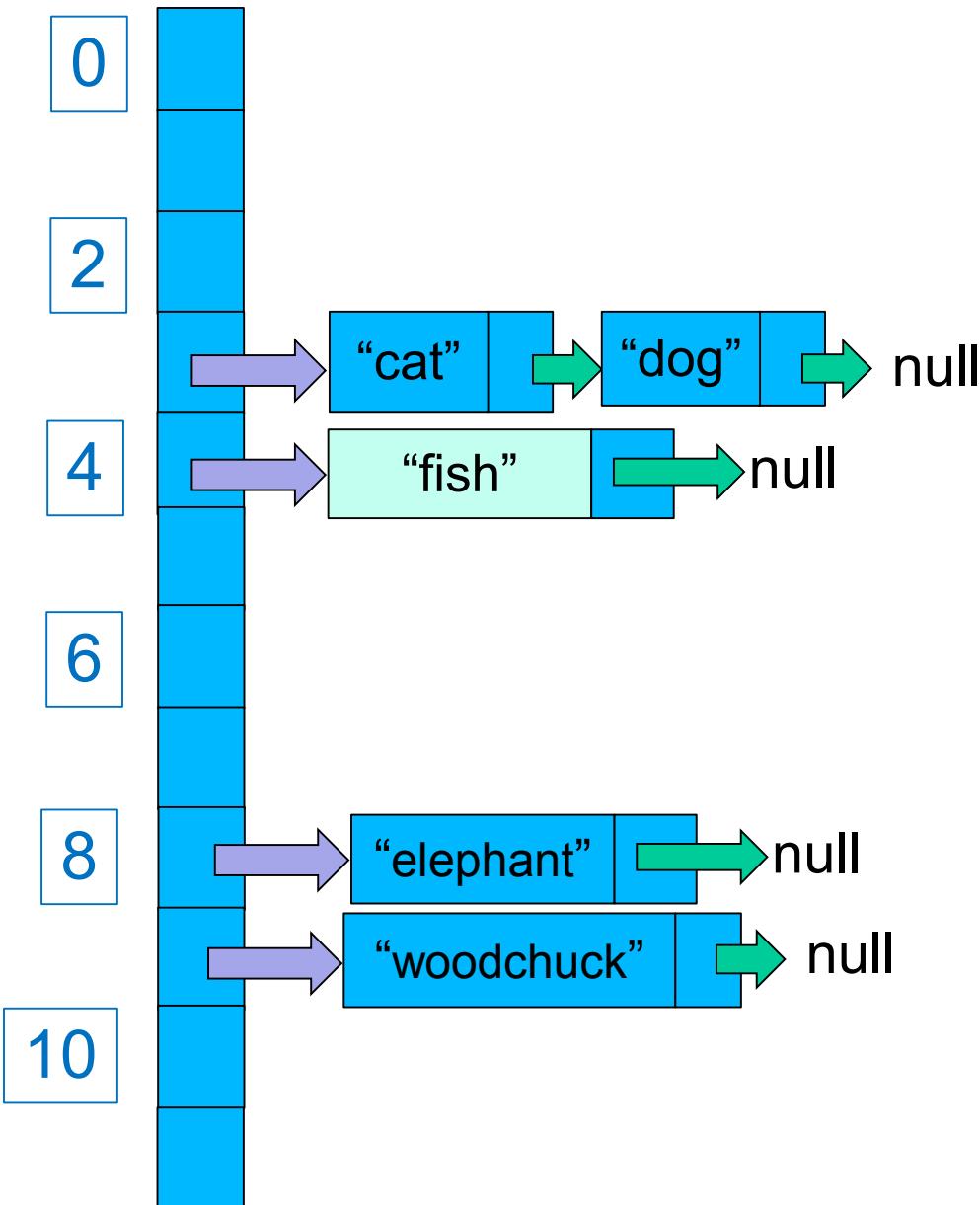




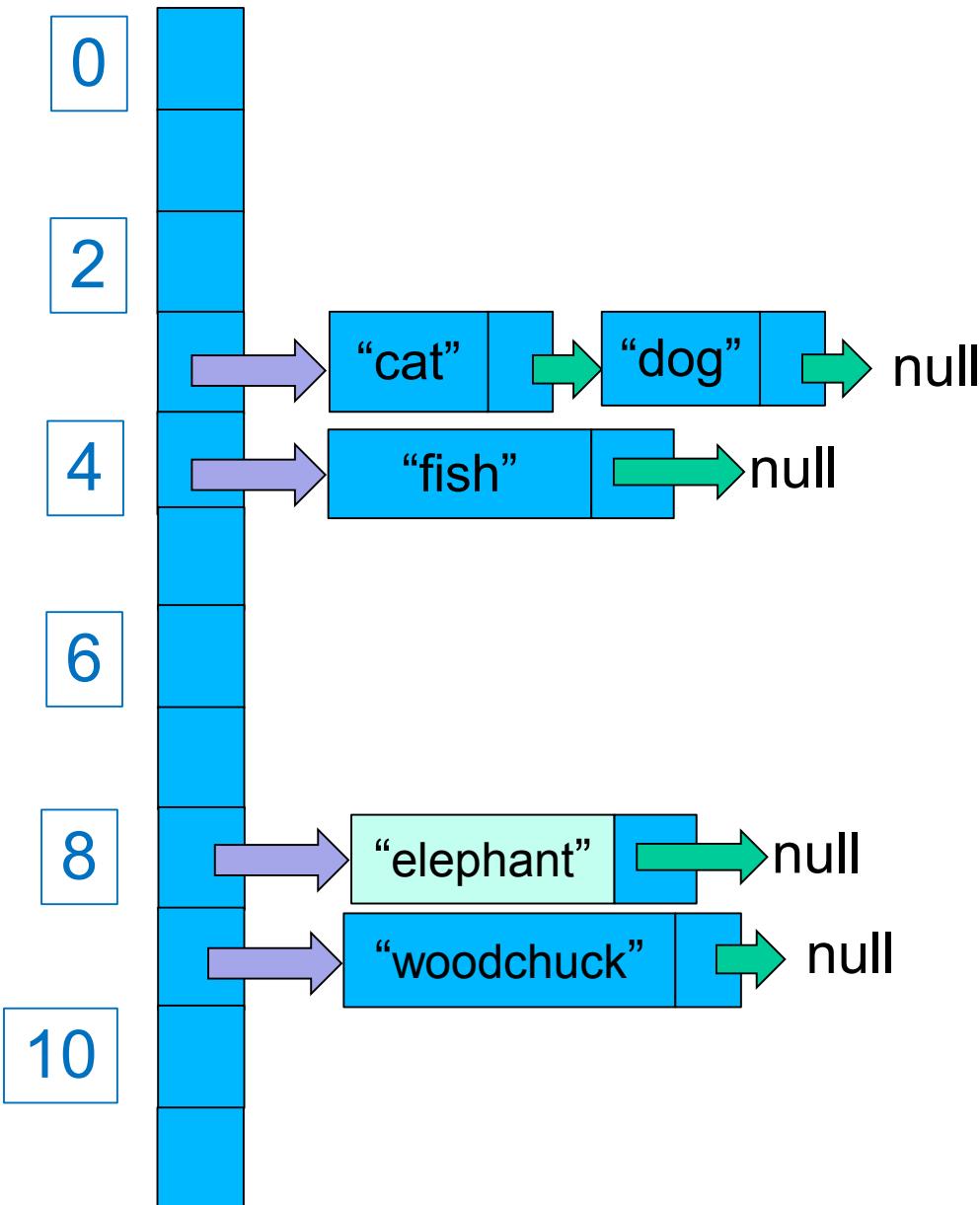
cat



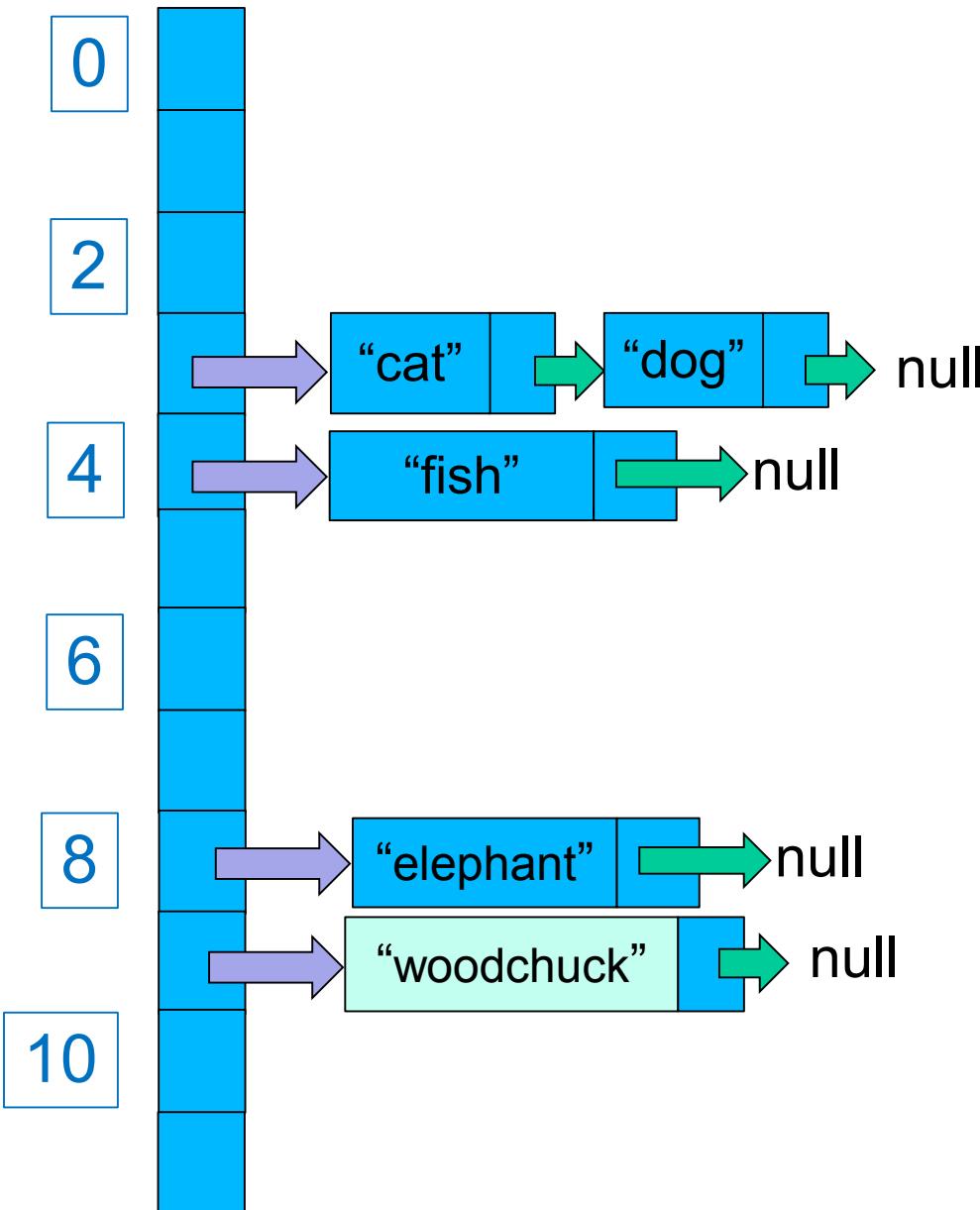
cat  
dog



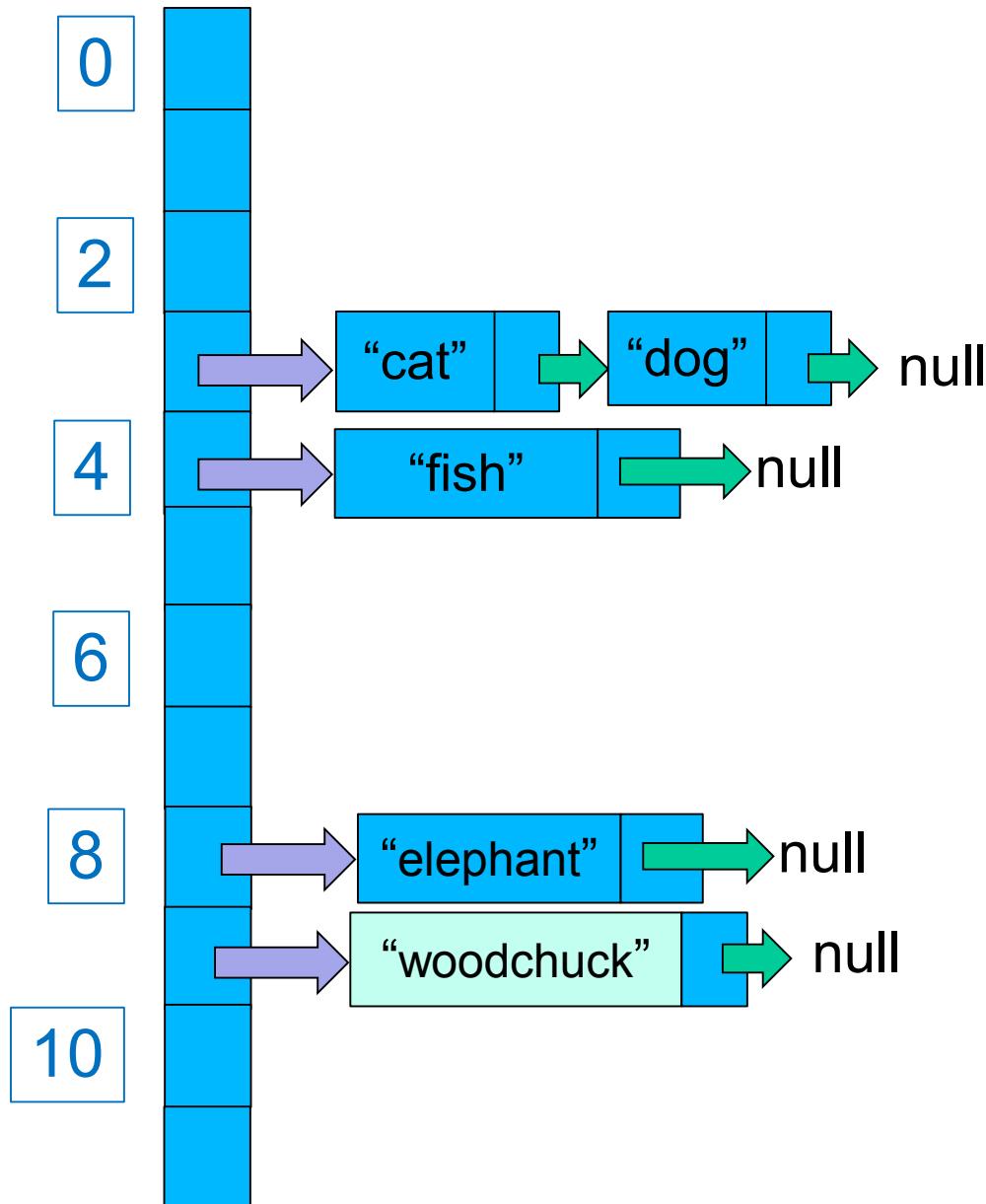
cat  
dog  
fish



cat  
dog  
fish  
elephant



cat  
dog  
fish  
elephant  
woodchuck



cat  
dog  
fish  
elephant  
woodchuck

Originally:  
elephant  
cat  
woodchuck  
dog  
fish

# What other types of problems might we want to solve?

# Other types of problems

---

- Store elements in such a way that their “natural order” is preserved
- Store elements in such a way that the largest (or smallest) element can be retrieved quickly
- Represent relationships/connections between elements in the collection

# **SD2x2.2**

# **Binary Search Trees**

# **Chris**

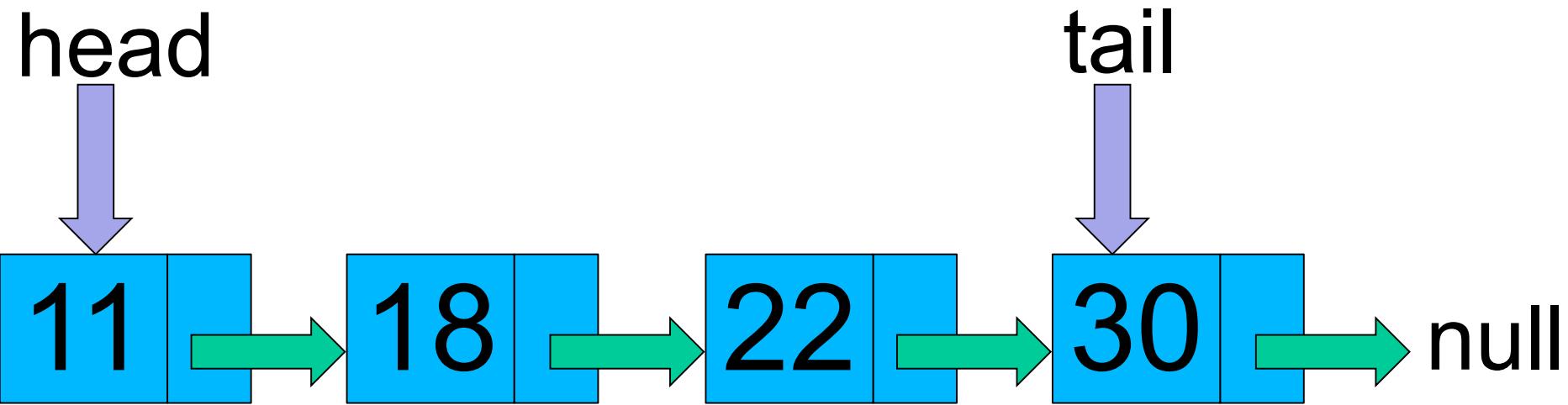
# Motivating Example

---

- Let's say we're keeping track of a collection of numbers
- How can we store them so that we can easily retrieve them (iterate over them) in **sorted** order?

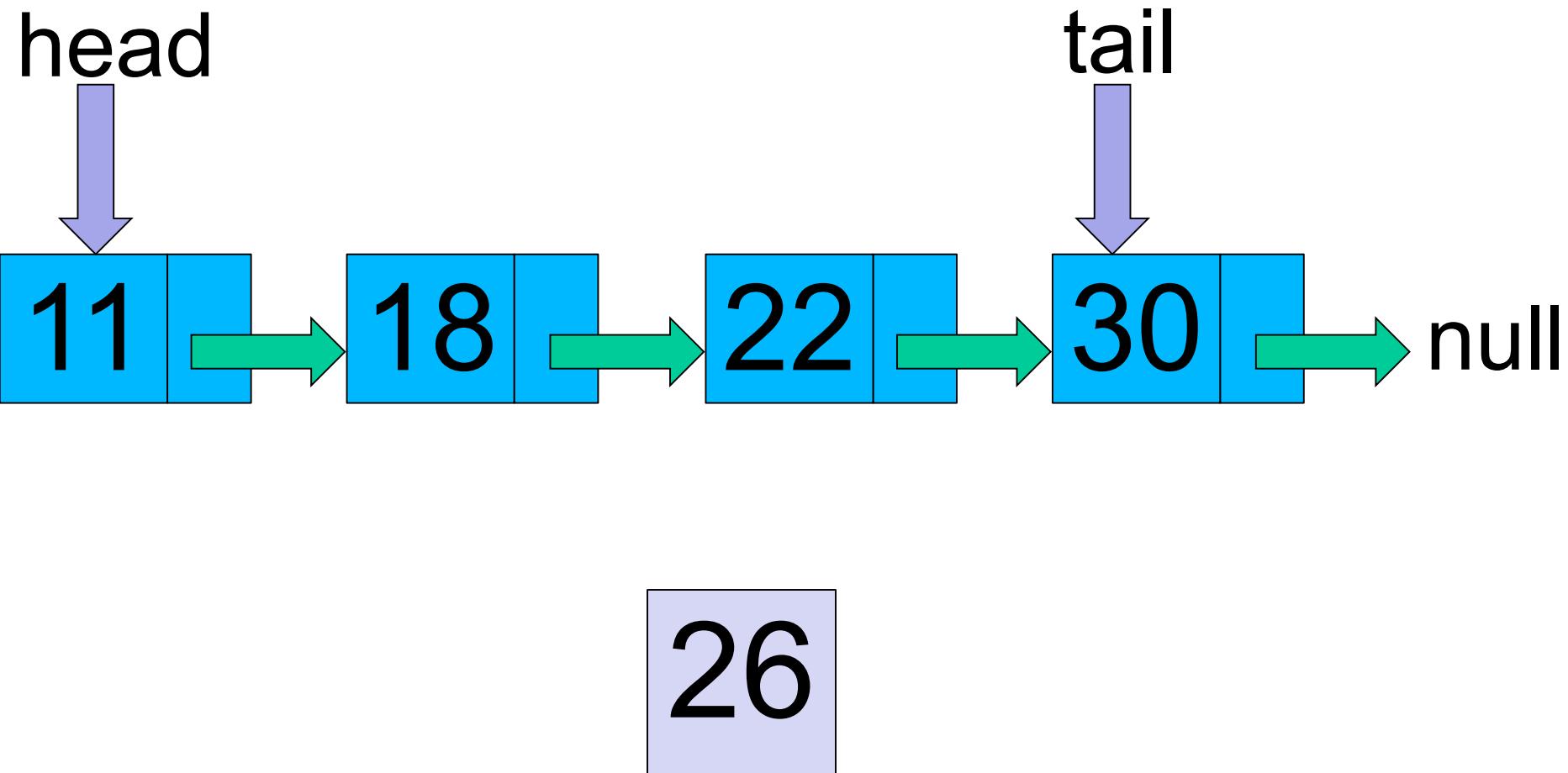
# Sorted LinkedList

---



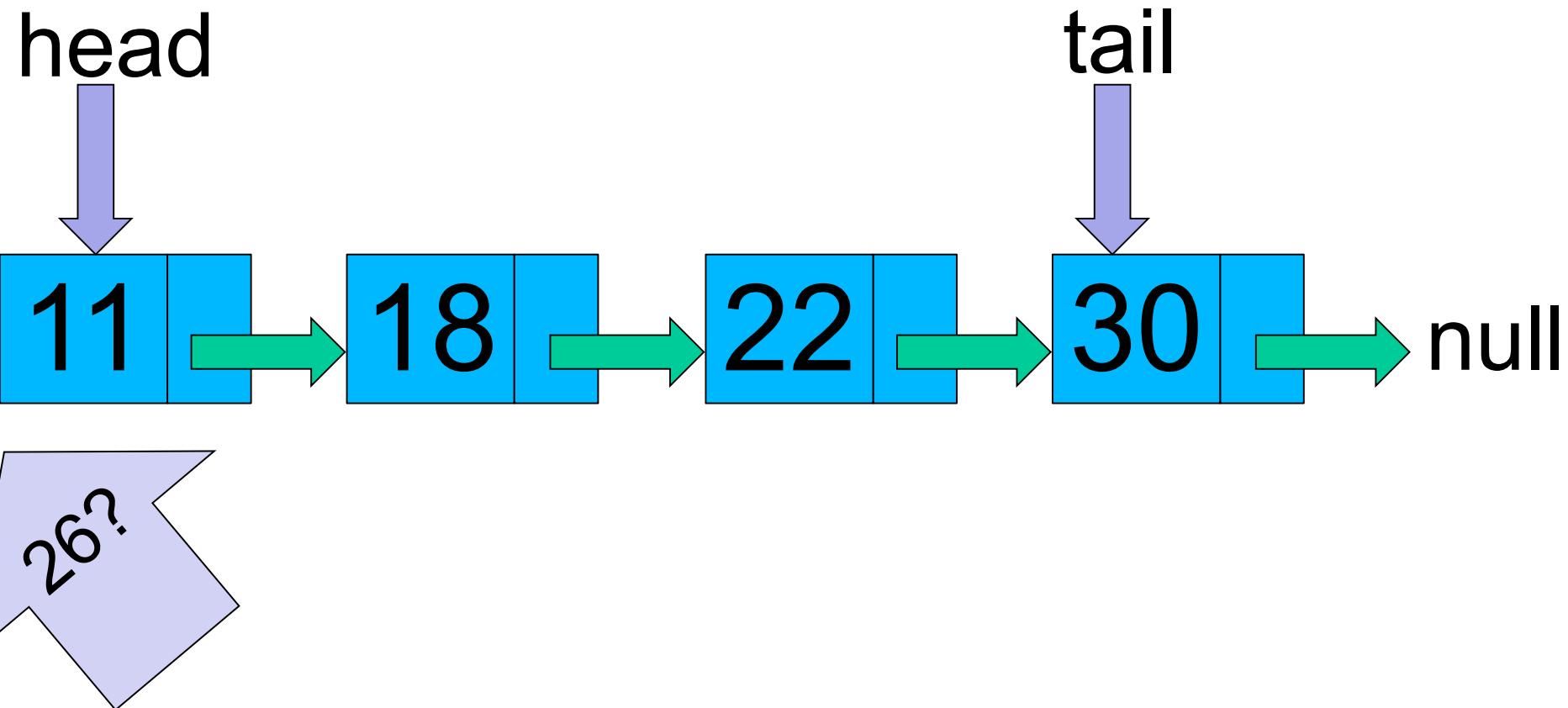
# Sorted LinkedList

---



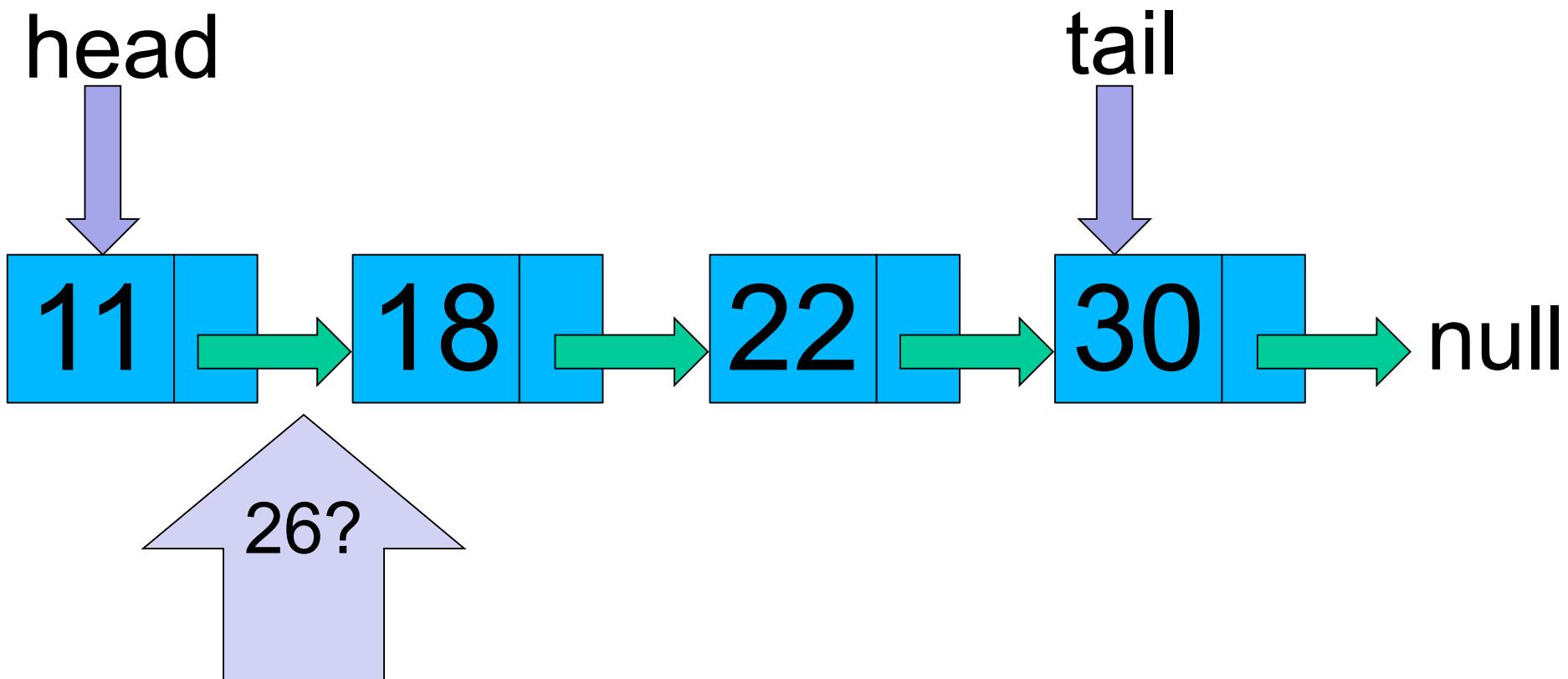
# Sorted LinkedList

---



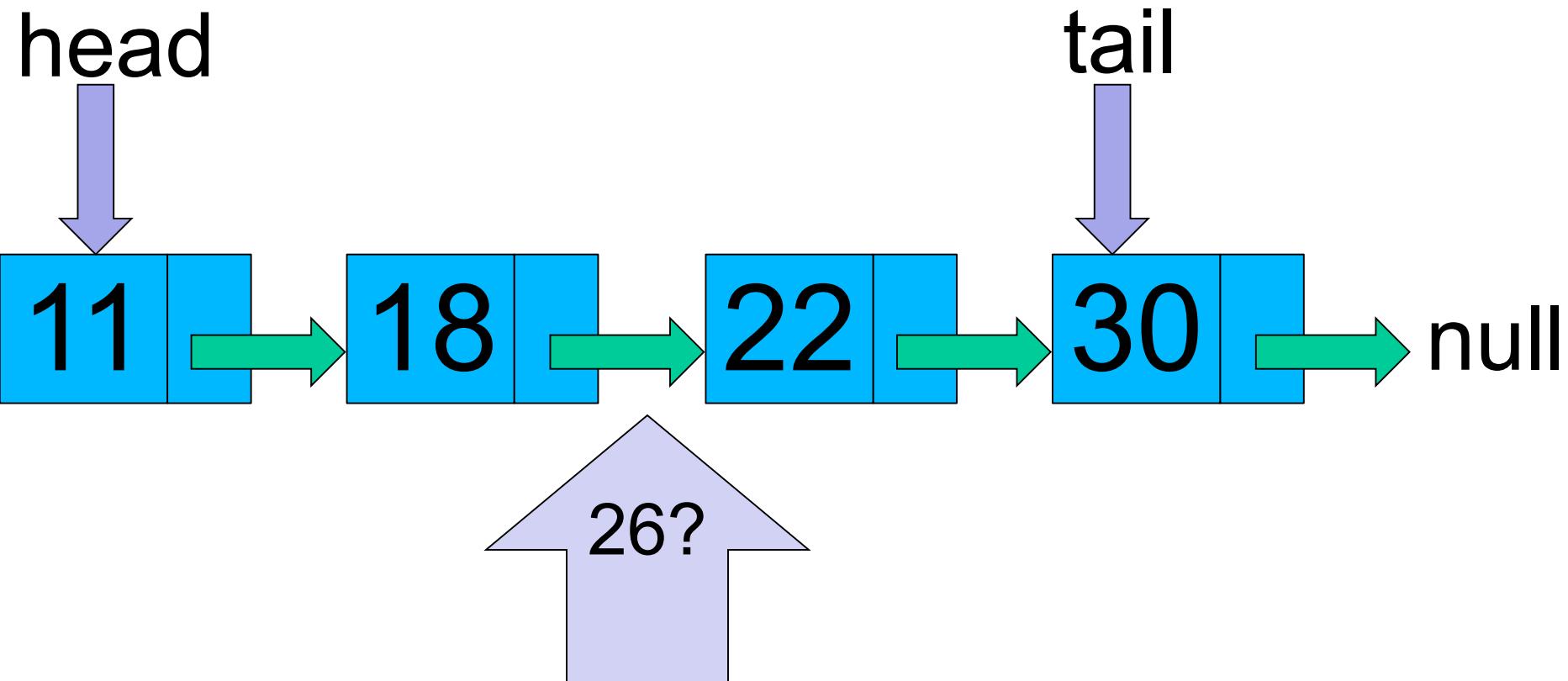
# Sorted LinkedList

---



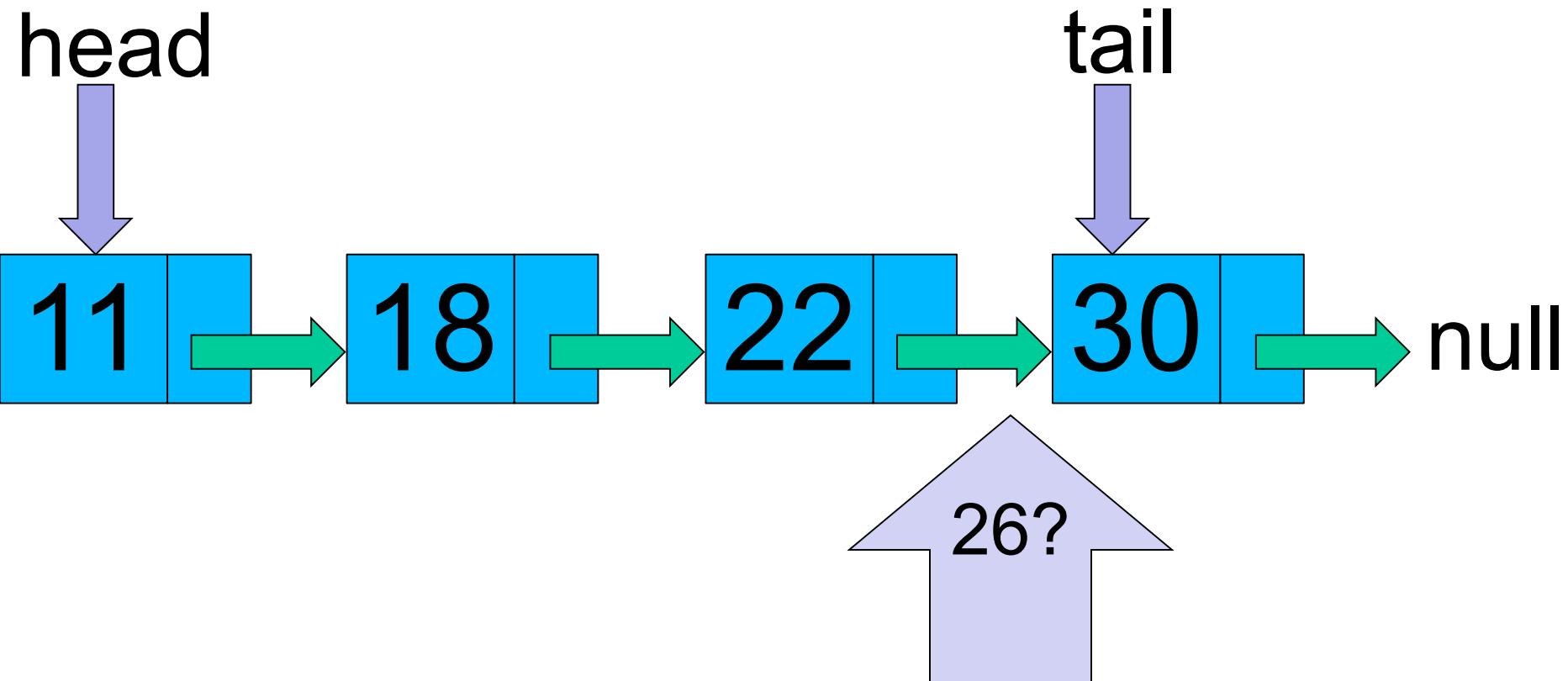
# Sorted LinkedList

---



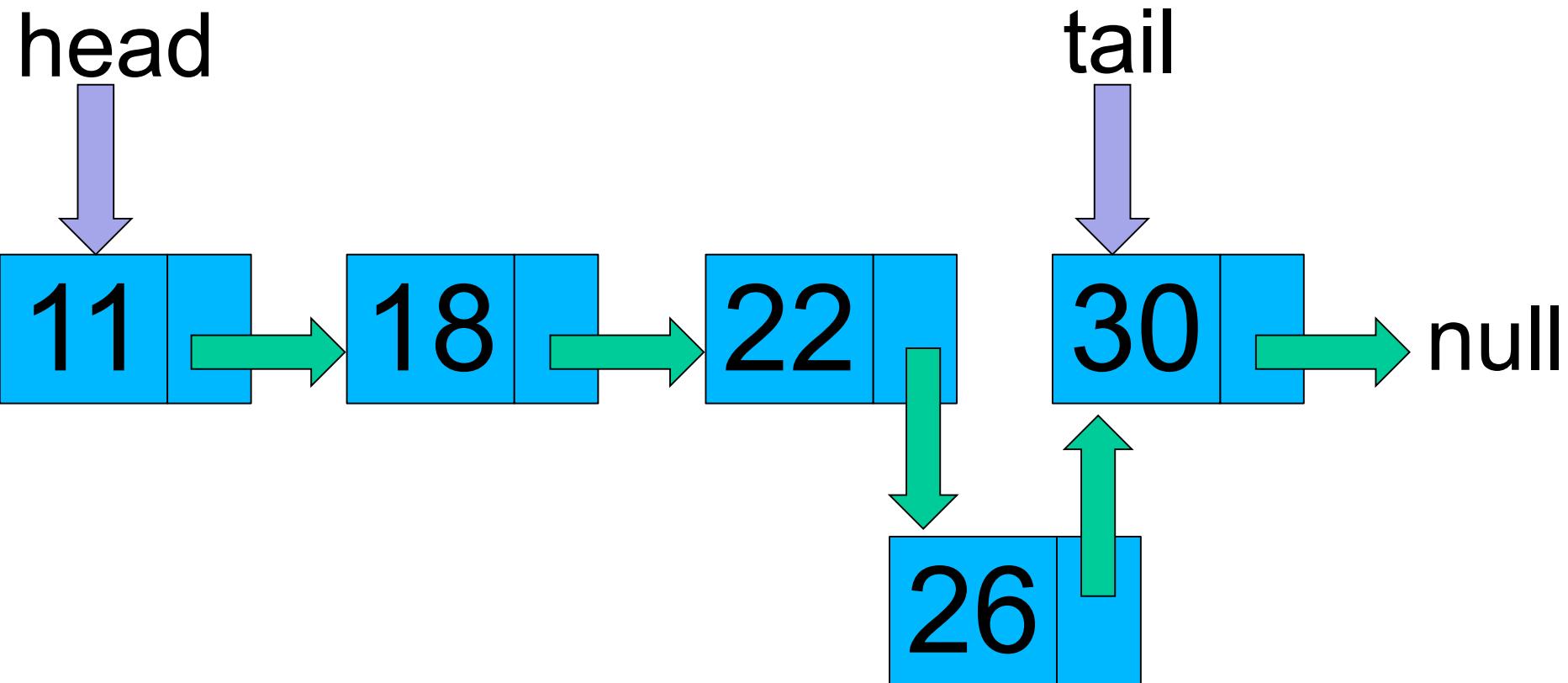
# Sorted LinkedList

---



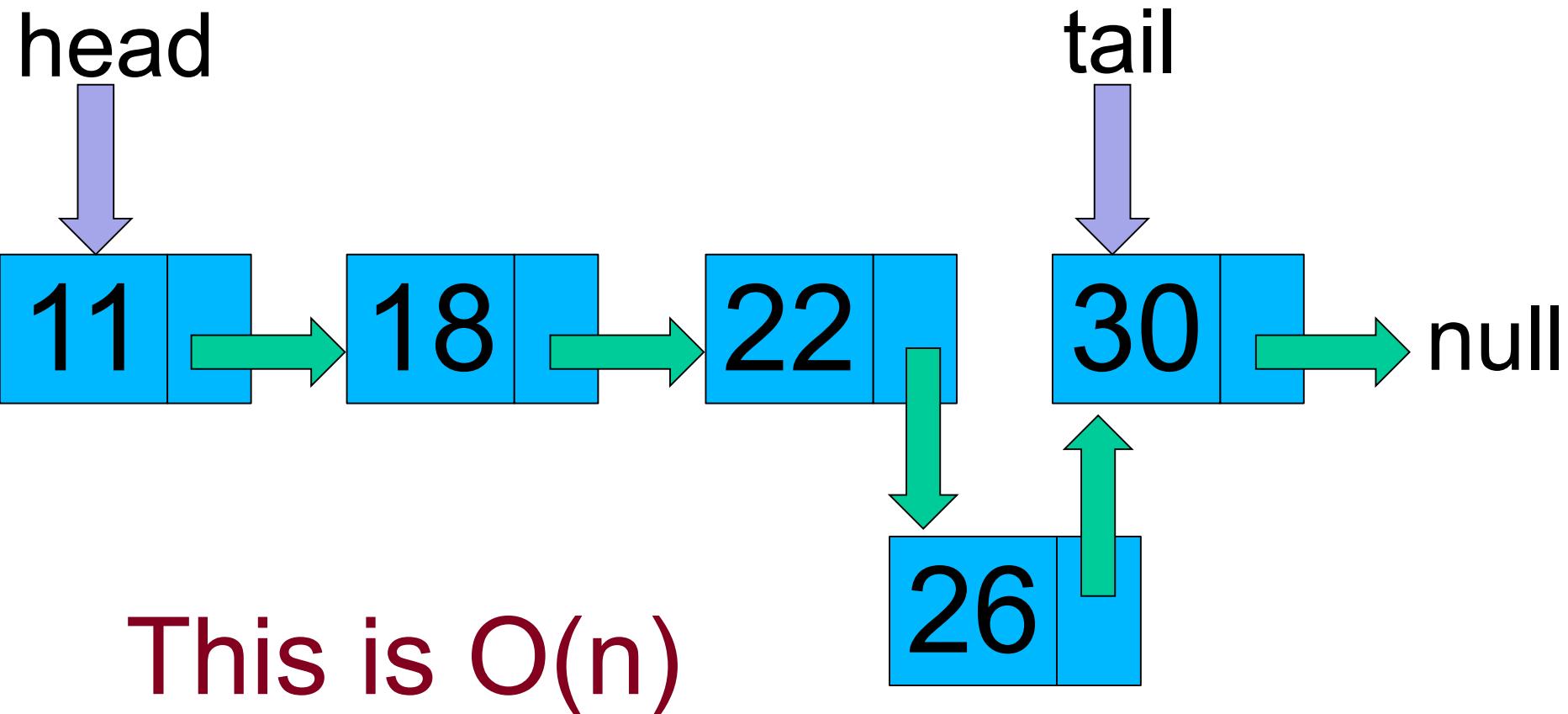
# Sorted LinkedList

---



# Sorted LinkedList

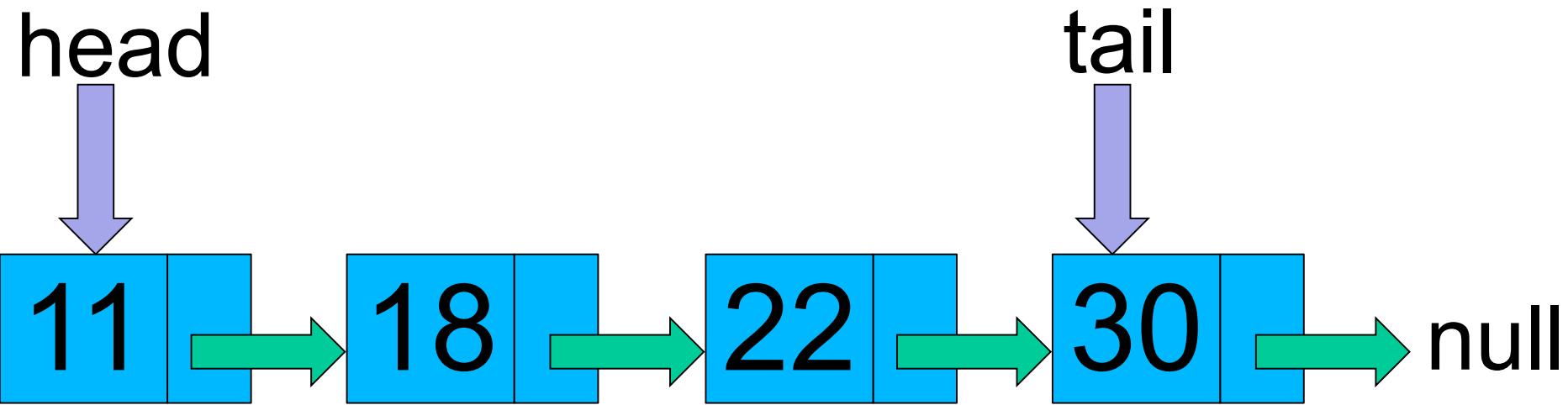
---



This is  $O(n)$

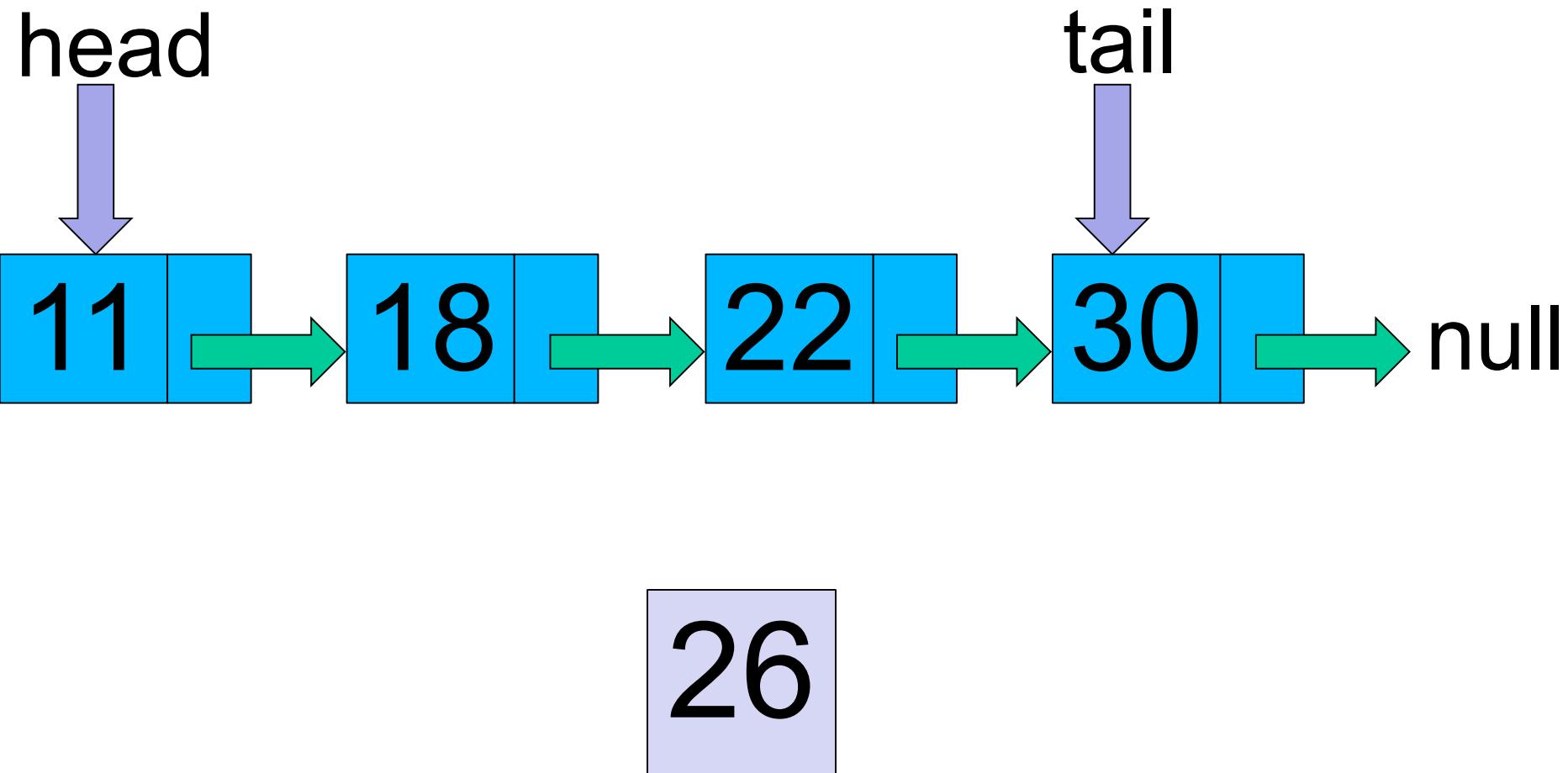
# Sorted LinkedList

---



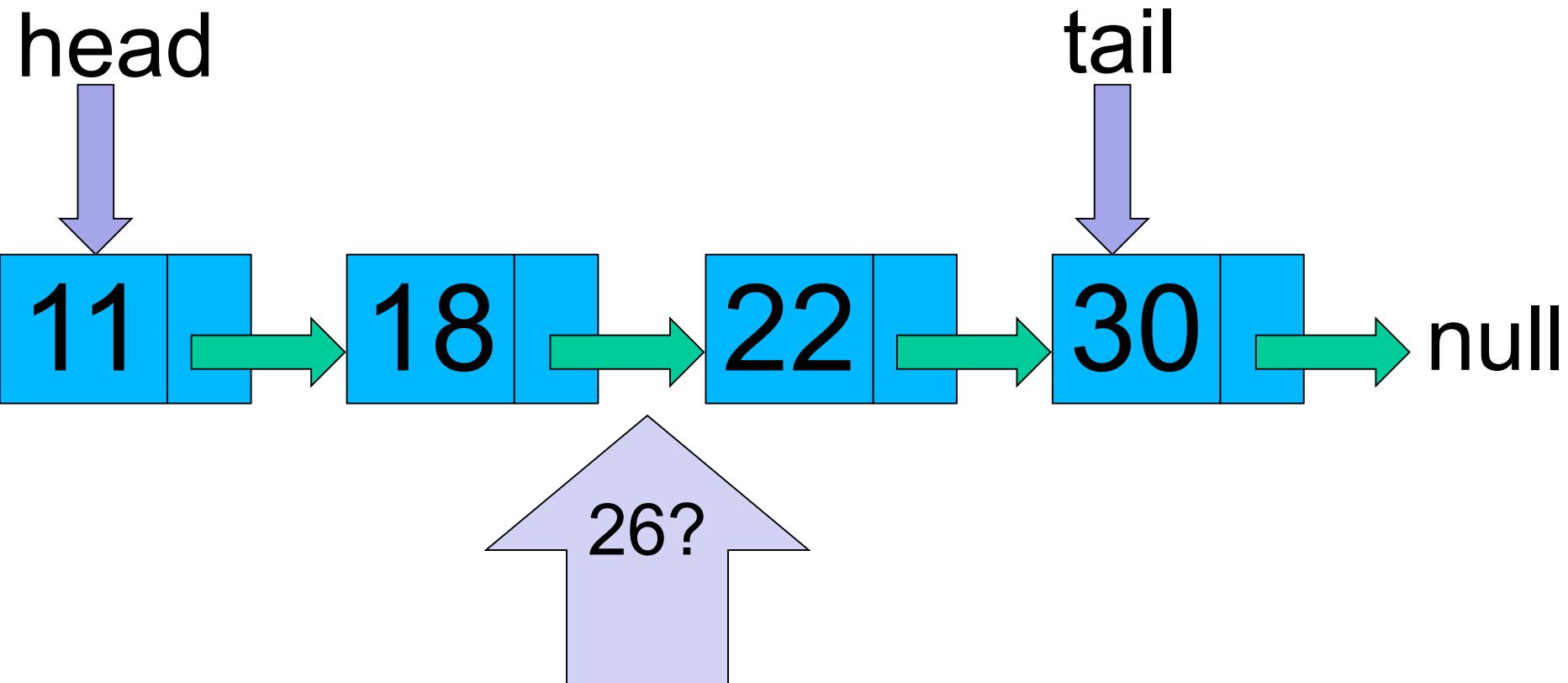
# Sorted LinkedList

---



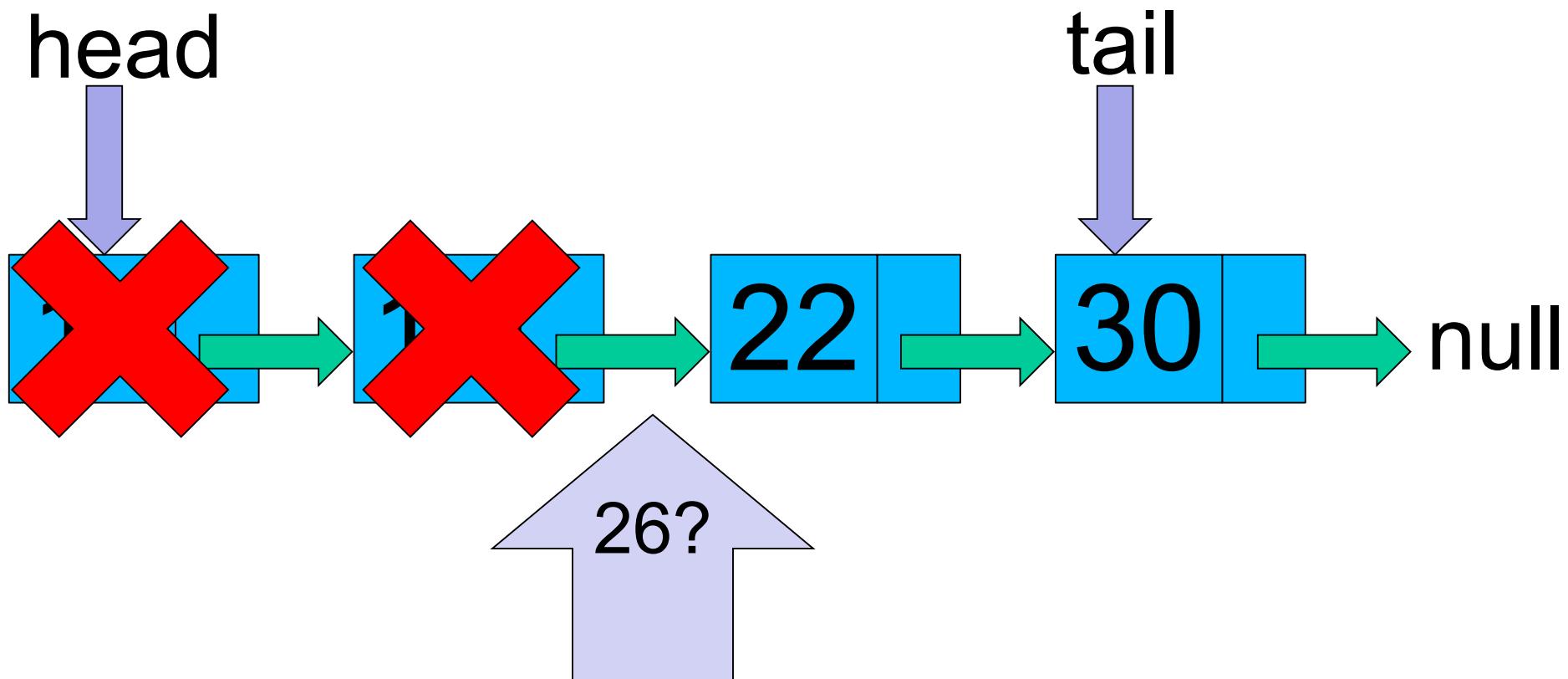
# Sorted LinkedList

---



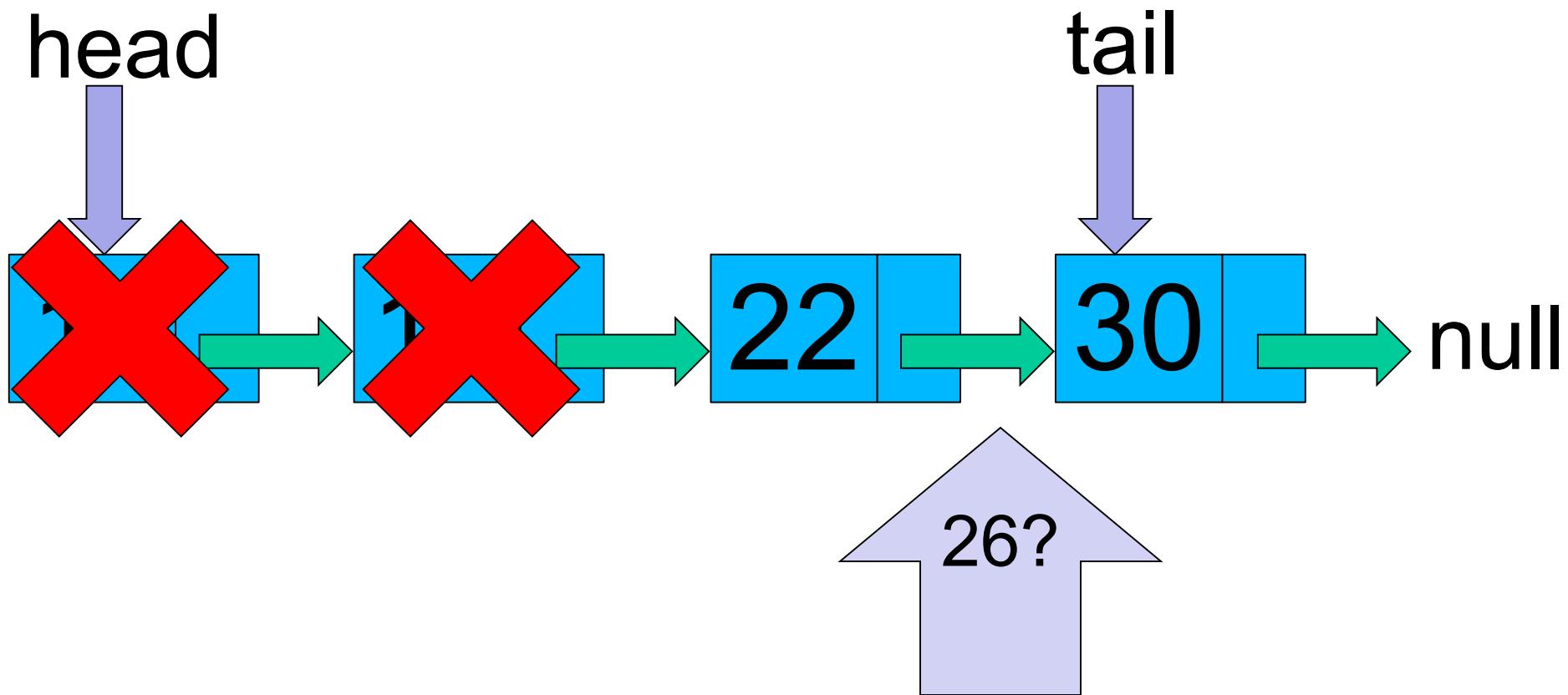
# Sorted LinkedList

---



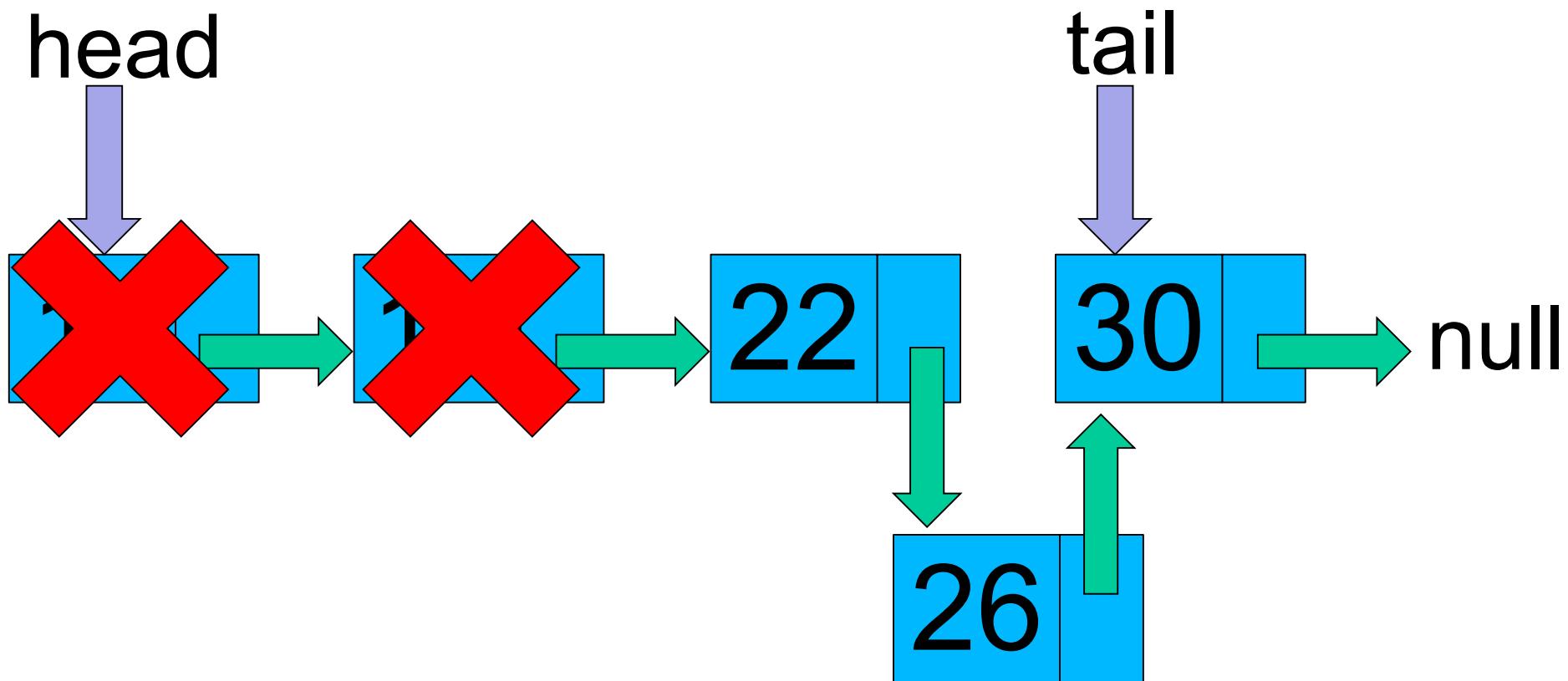
# Sorted LinkedList

---



# Sorted LinkedList

---



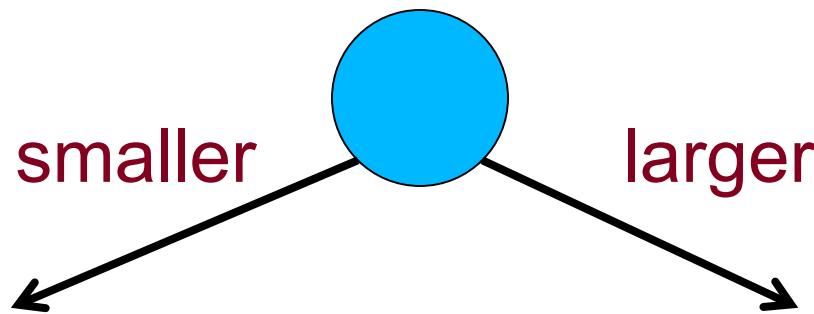
# Storing Elements in Sorted Order

---

- **Intuition:** for each element...
- Try to place it in the **middle** of the list
- If too big, try to place it in the middle of the remaining (greater) elements on the right
- If too small, try to place it in the middle of the remaining (smaller) elements on the left
- And so on...

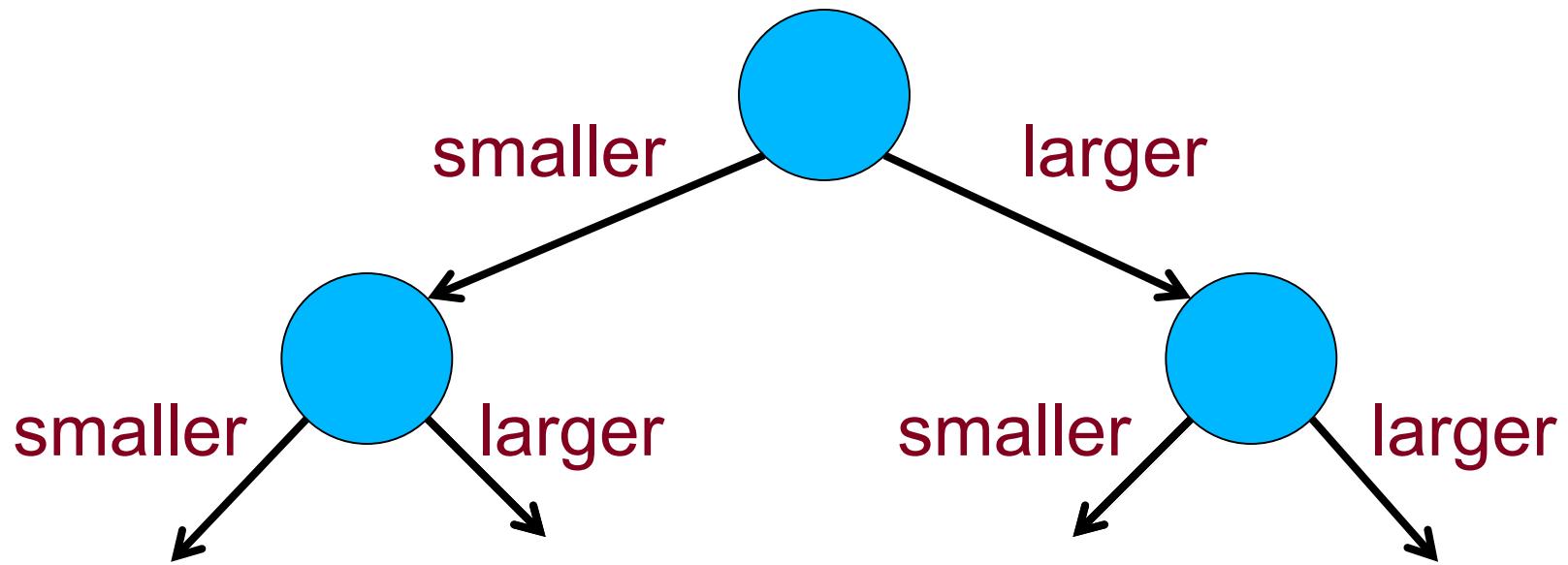
# Storing Elements in Sorted Order

---



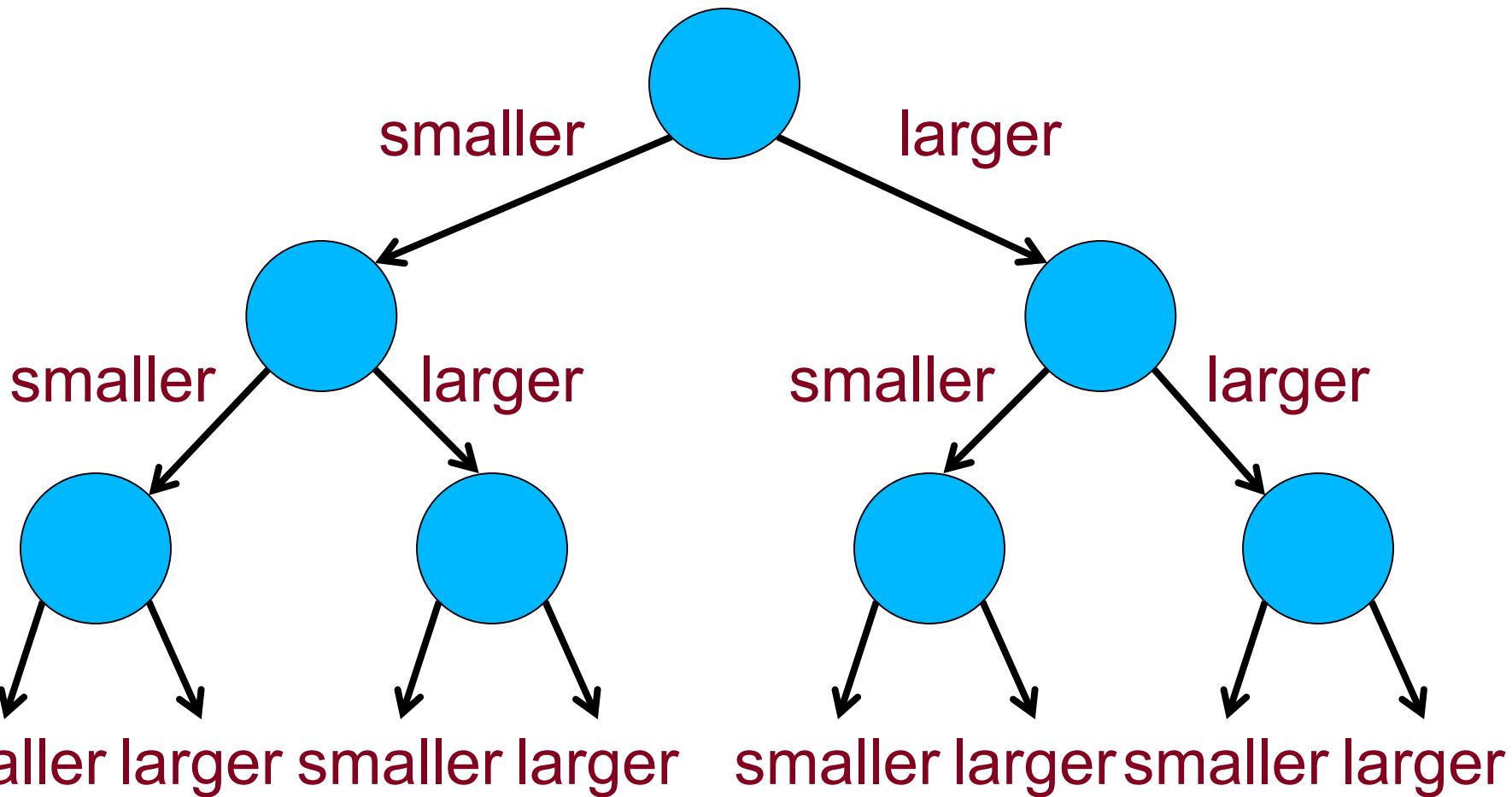
# Storing Elements in Sorted Order

---



# Storing Elements in Sorted Order

---





**Tree!**

# Trees

---

- In general, a tree is a data structure composed of zero or more **nodes**
- A node contains a **value** and links to zero or more **child nodes**
- A tree has a single **root node**

# Binary Trees

---

- A Binary Tree is a tree in which each node has links to exactly two nodes: a **left child** and a **right child**
- Either or both may be null

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node root = null;  
  
    . . .
```

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node root = null;  
  
    . . .
```

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node root = null;  
  
    . . .
```

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node root = null;  
  
    . . .
```

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node root = null;  
  
    . . .
```

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node root = null;  
  
    . . .
```

# Binary Tree class definition

---

```
public class BinaryTree {  
  
    class Node {  
        int value;  
        Node leftChild = null;  
        Node rightChild = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
}  
  
protected Node root = null;
```

...

# Binary Search Trees

---

- A Binary Search Tree is an **ordered** Binary Tree
- Every node in the tree is **greater** than each element in its left subtree
- Every node in the tree is **less** than each element in its right subtree

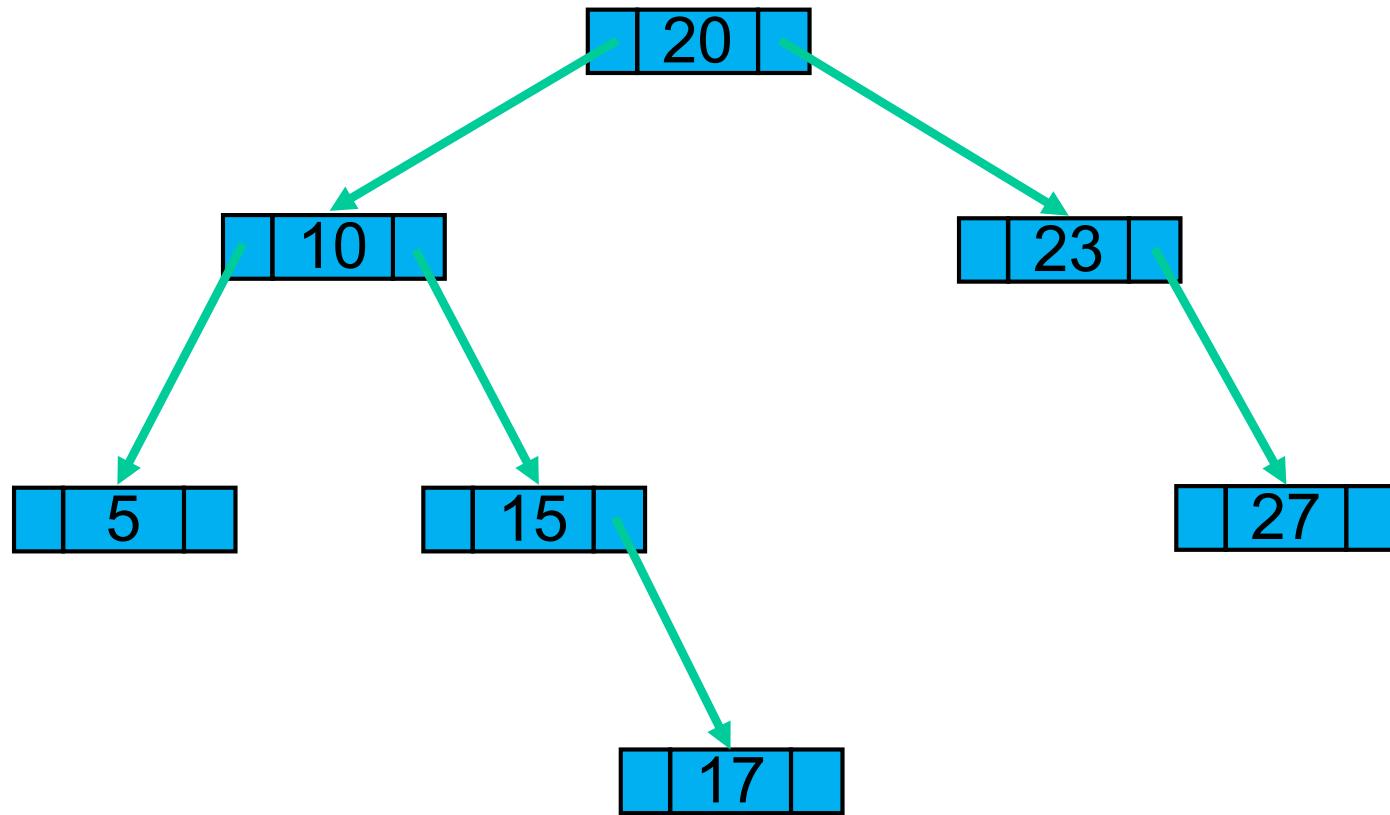
# Binary Search Tree Traversal

---

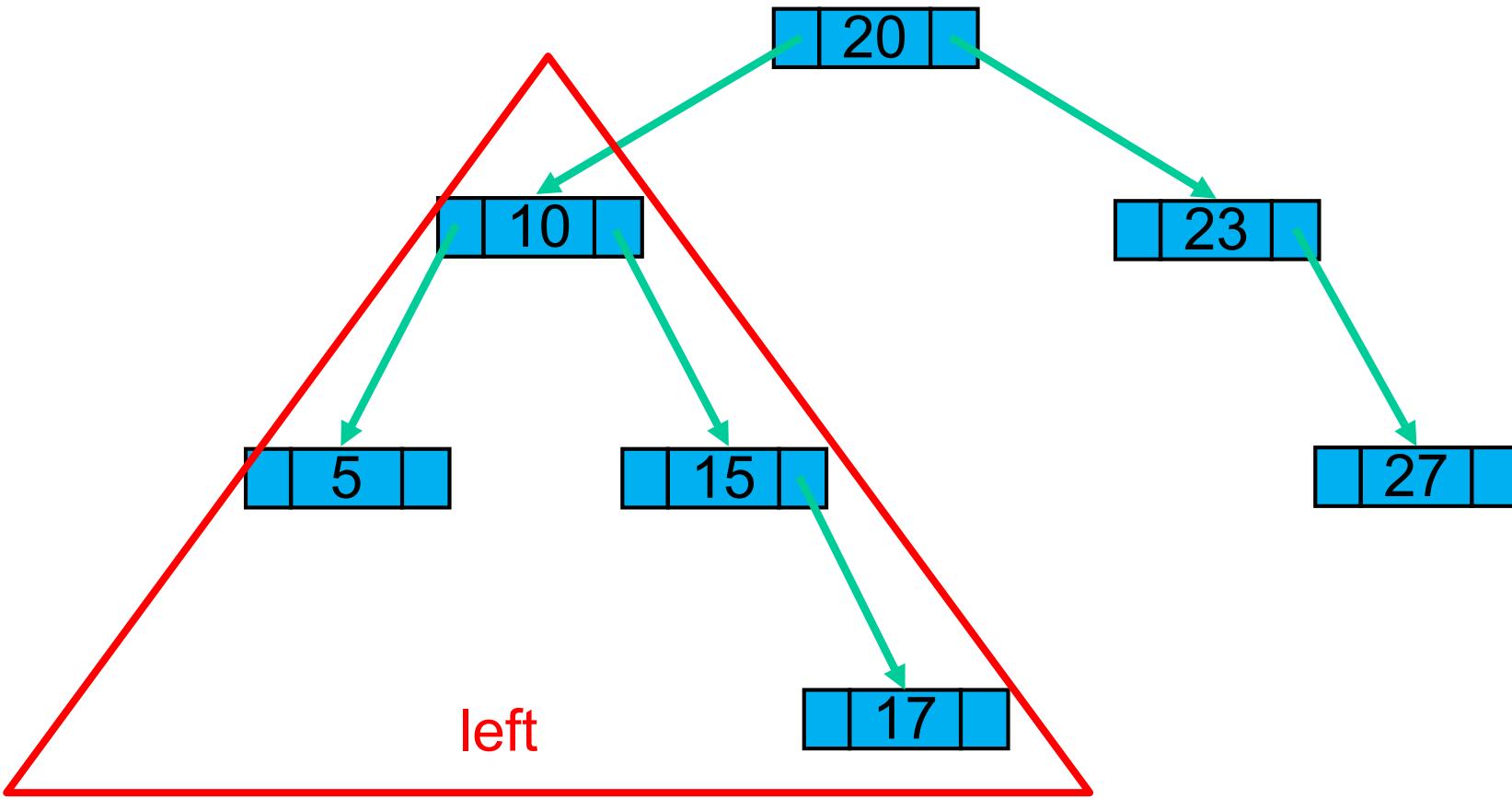
- We can perform **inorder traversal** to “visit” each node in sorted order
- Starting at the root:
  - Visit the left subtree
  - Visit the node
  - Visit the right subtree

# Inorder Traversal: left, node, right

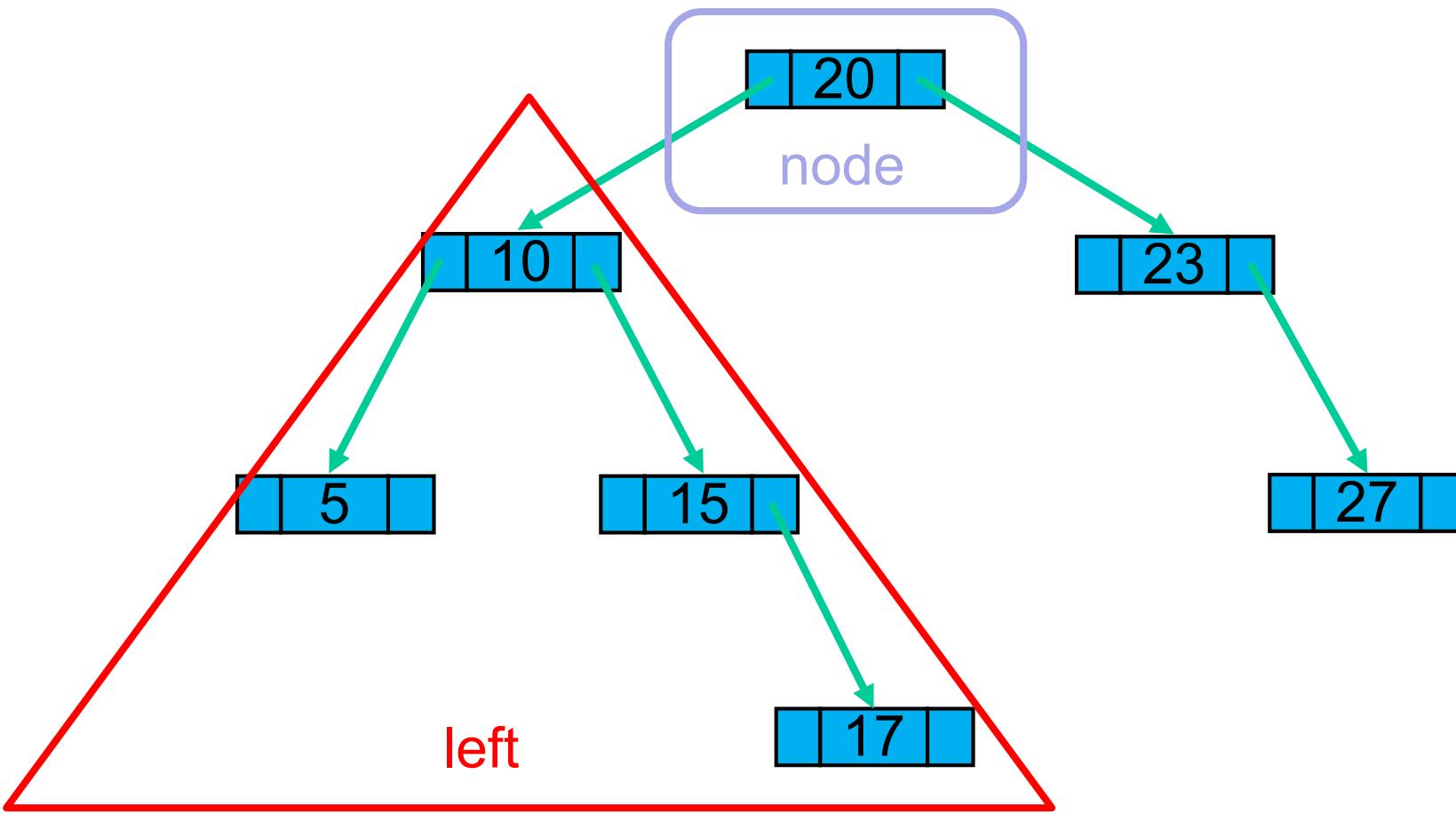
---



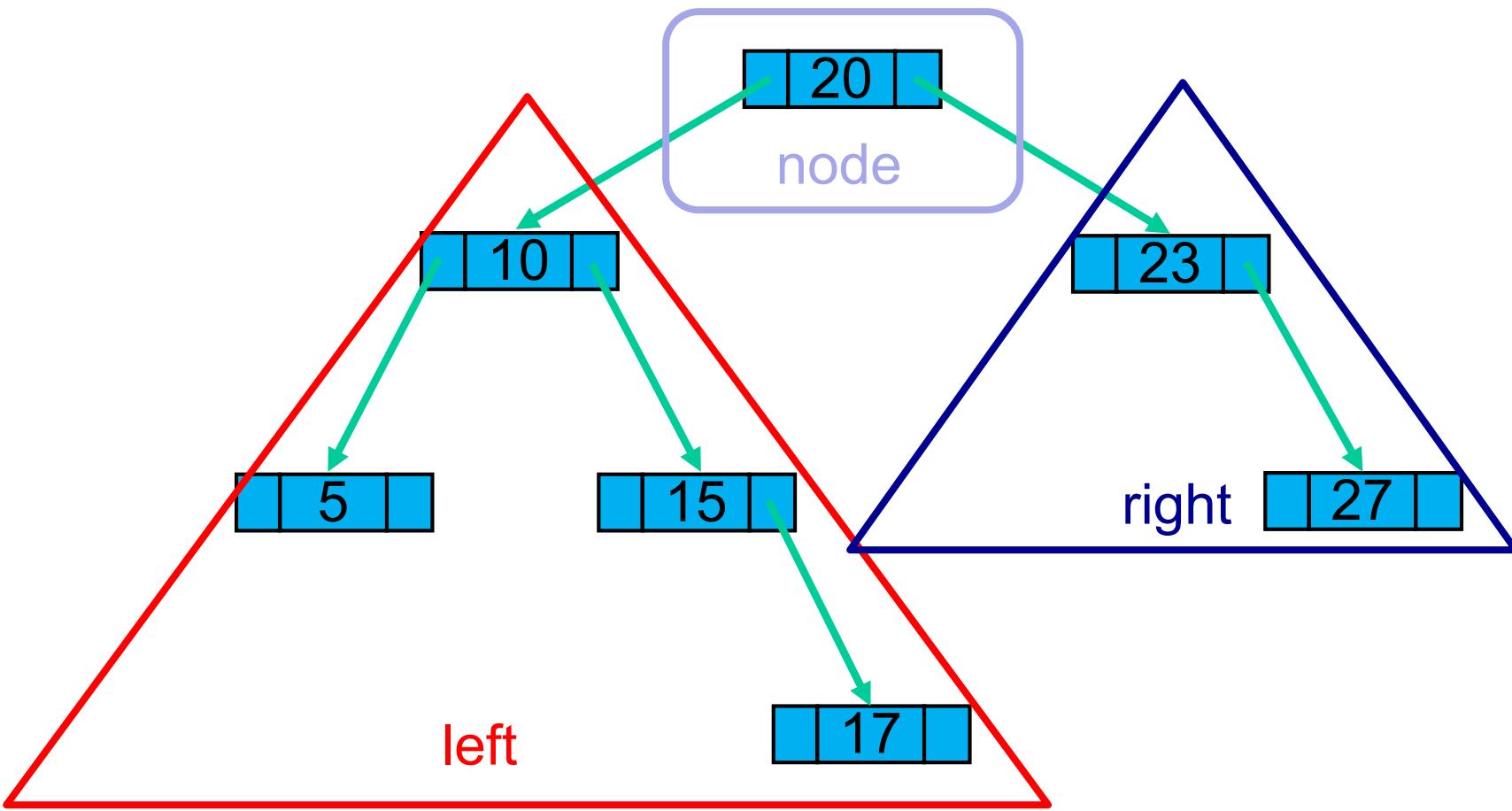
# Inorder Traversal: left, node, right



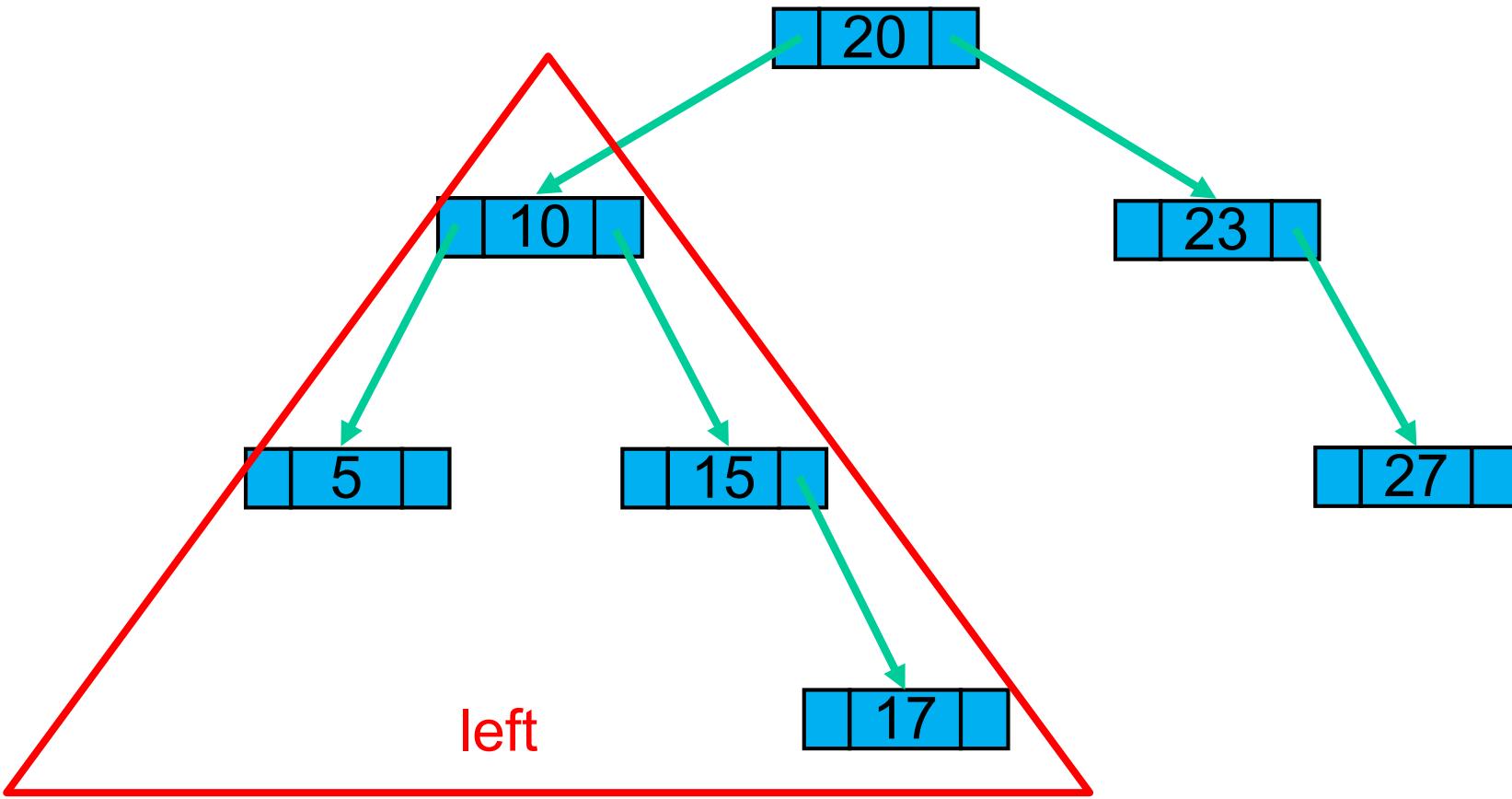
# Inorder Traversal: left, node, right



# Inorder Traversal: left, node, right

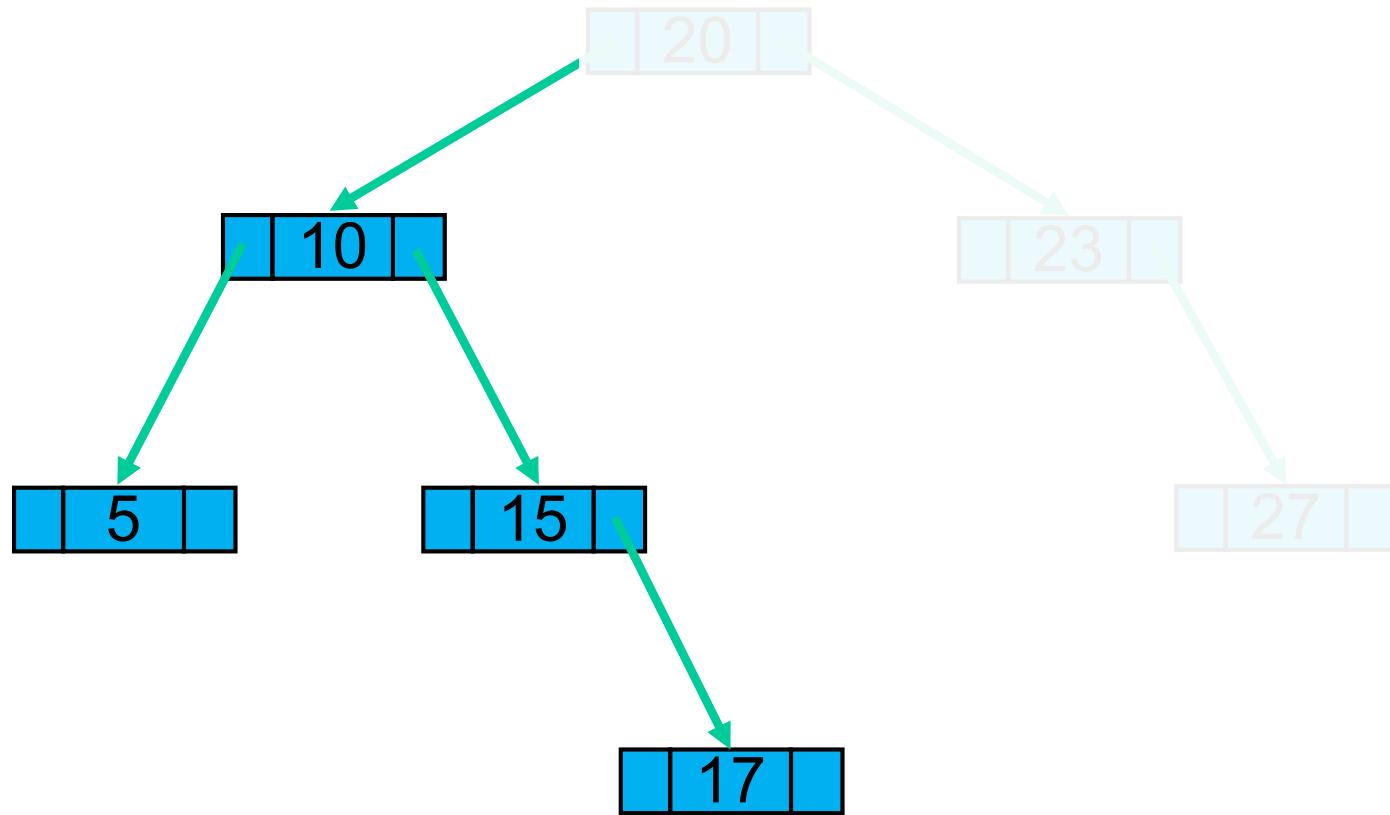


# Inorder Traversal: left, node, right



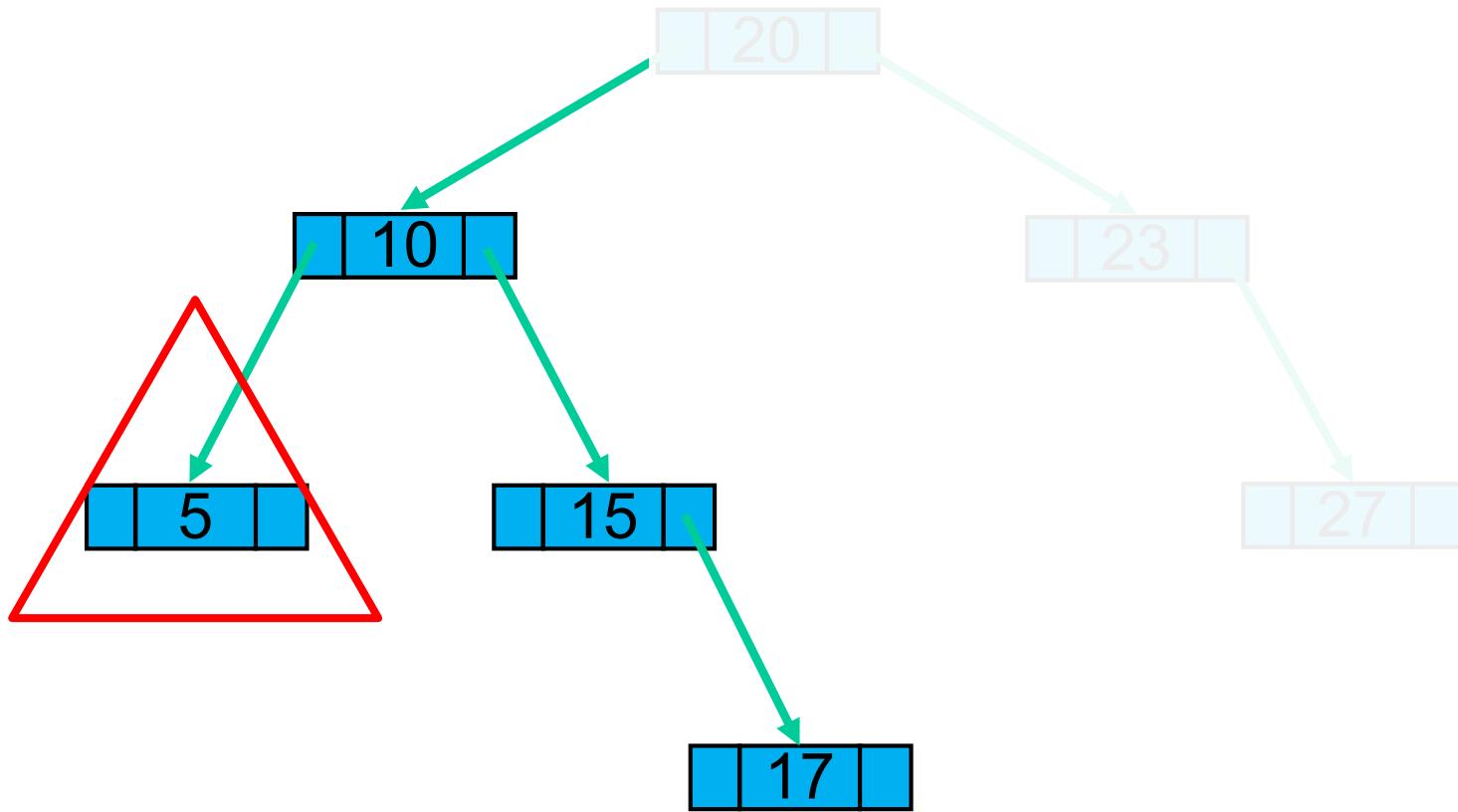
# Inorder Traversal: left, node, right

---

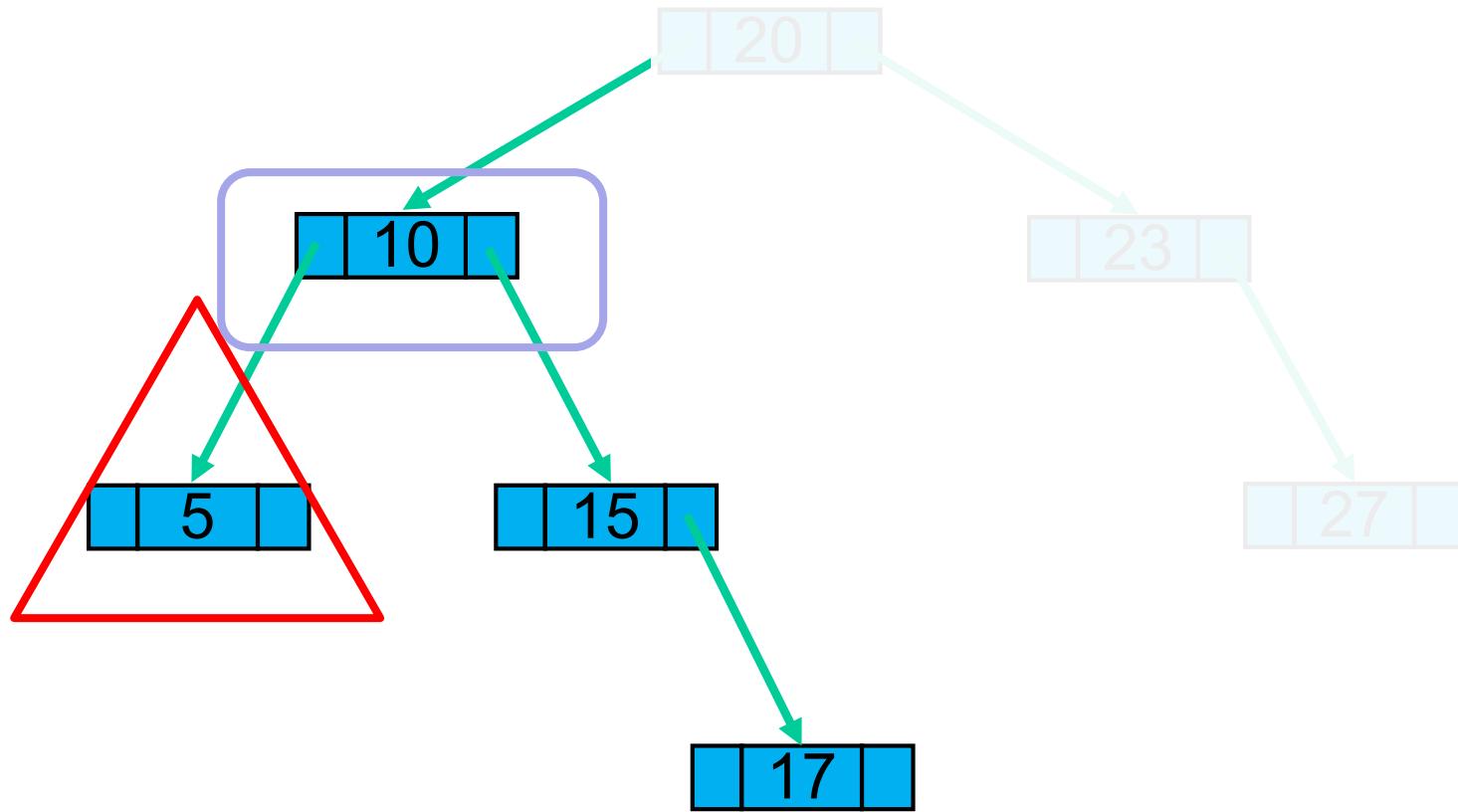


# Inorder Traversal: left, node, right

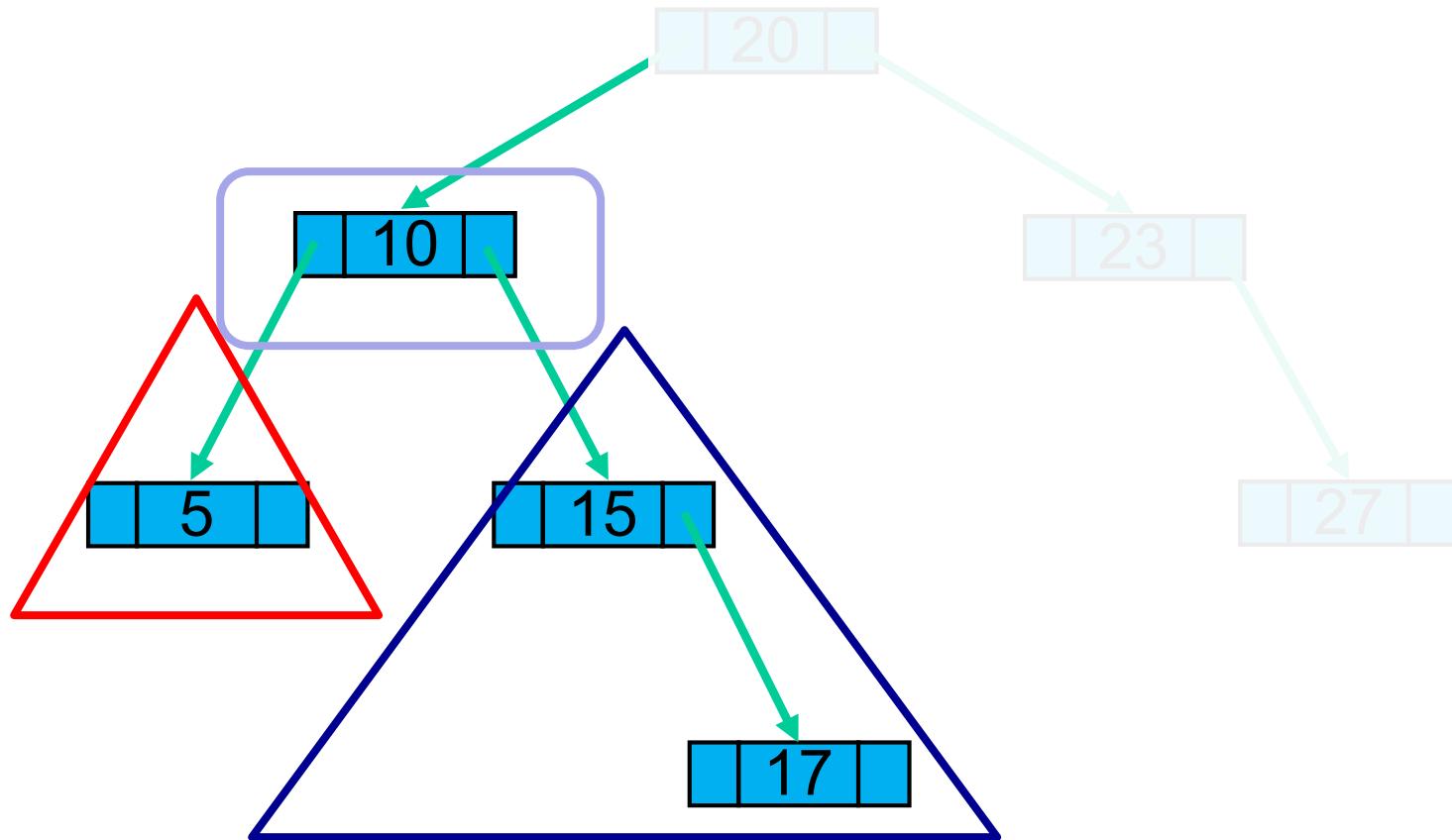
---



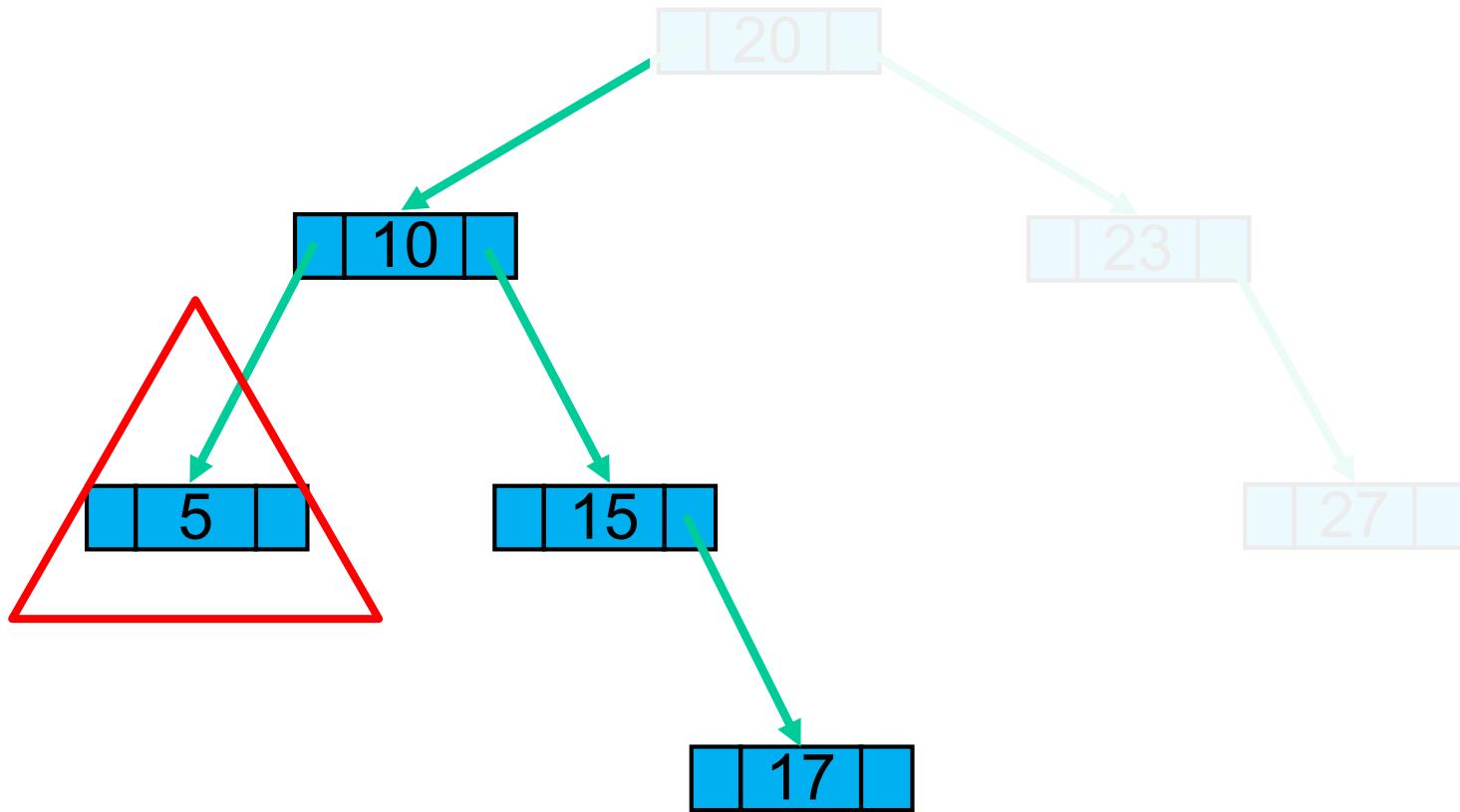
# Inorder Traversal: left, node, right



# Inorder Traversal: left, node, right

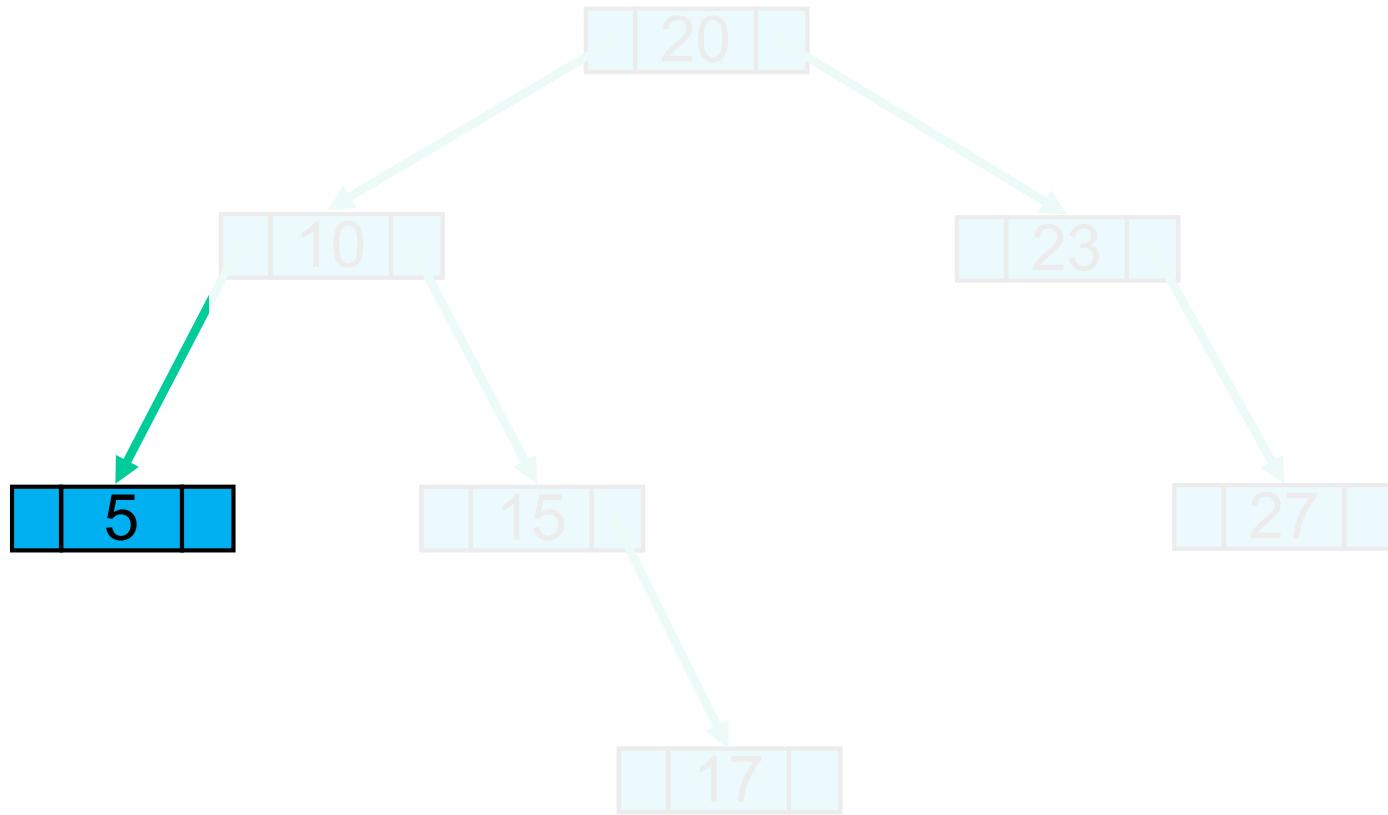


# Inorder Traversal: left, node, right



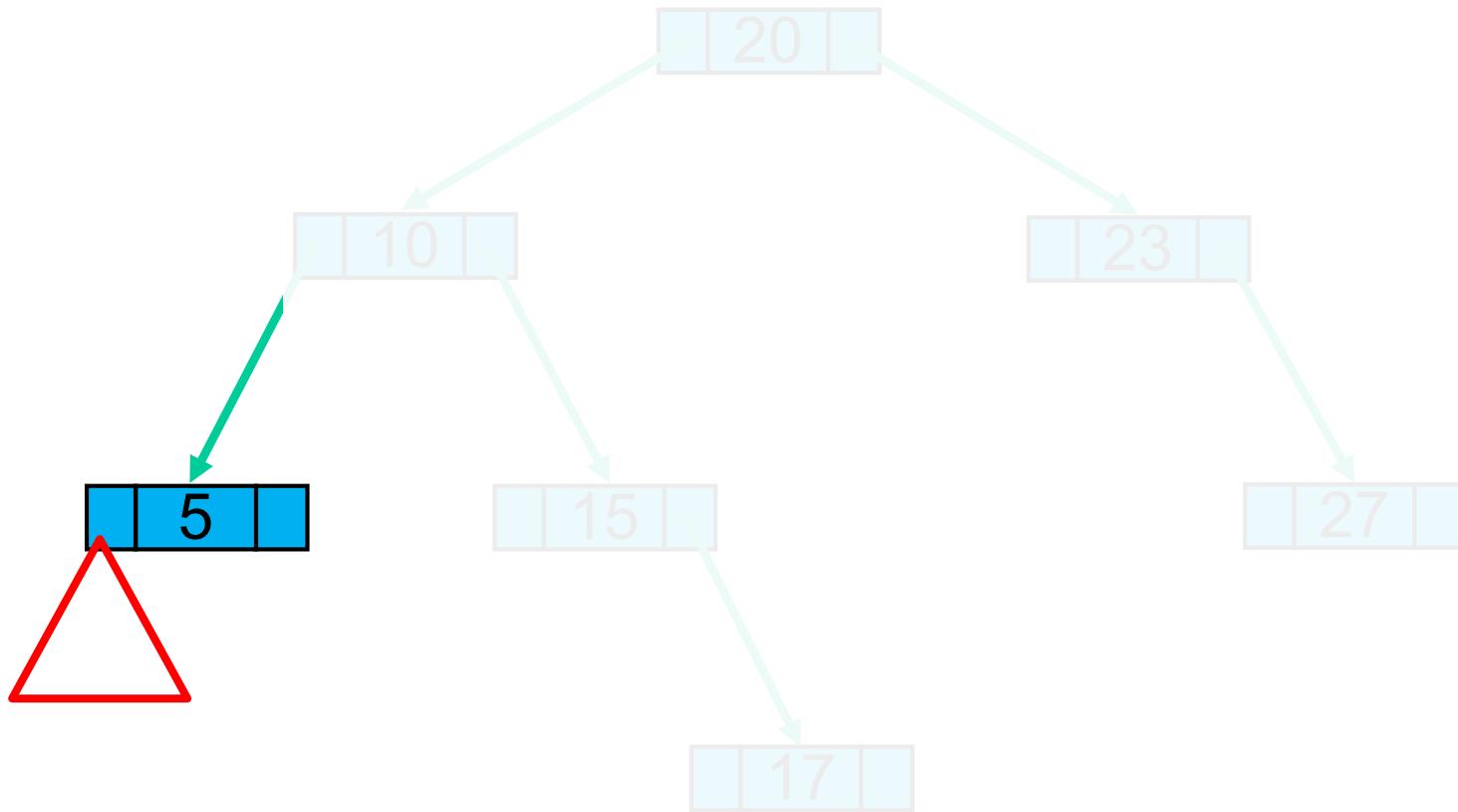
# Inorder Traversal: left, node, right

---

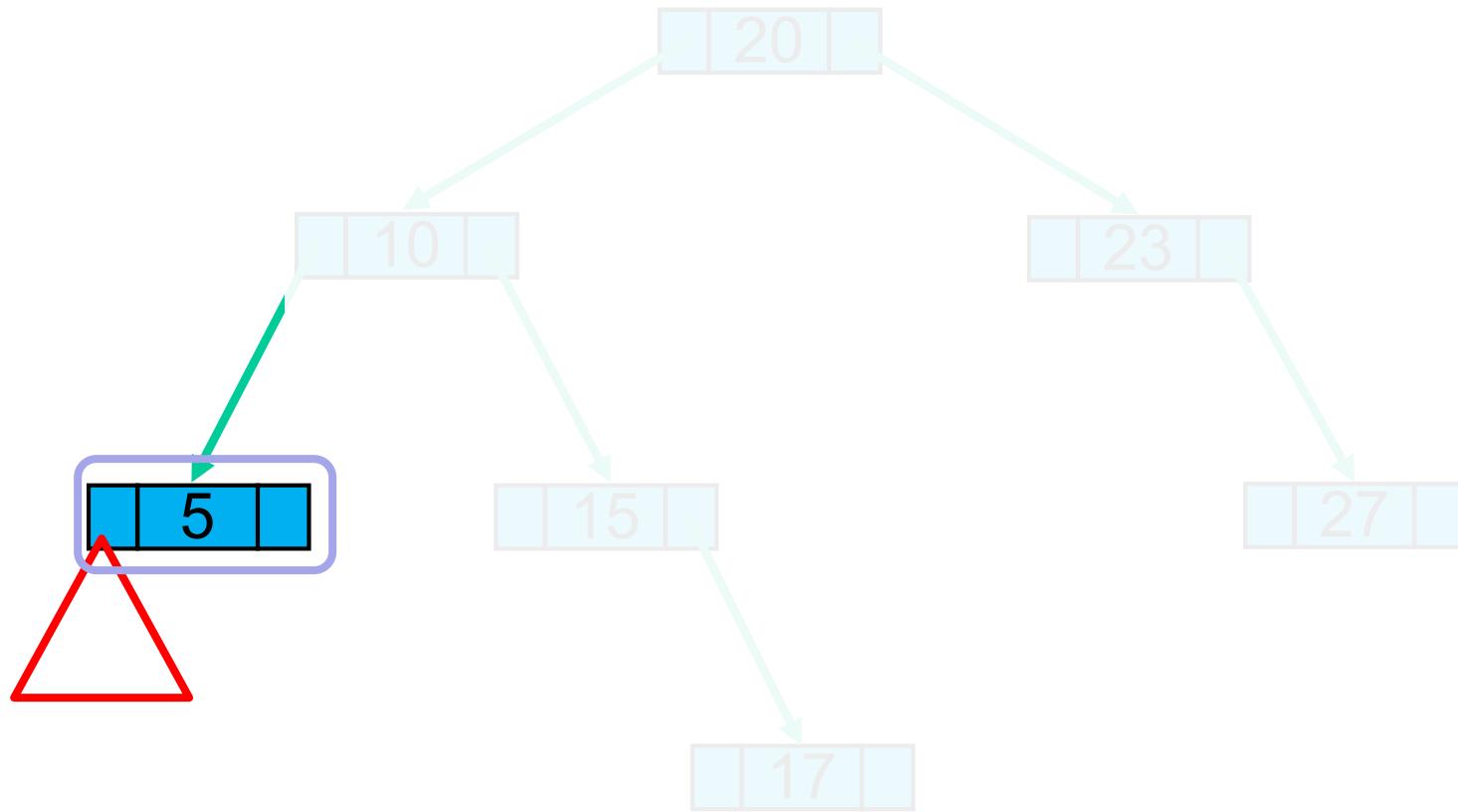


# Inorder Traversal: left, node, right

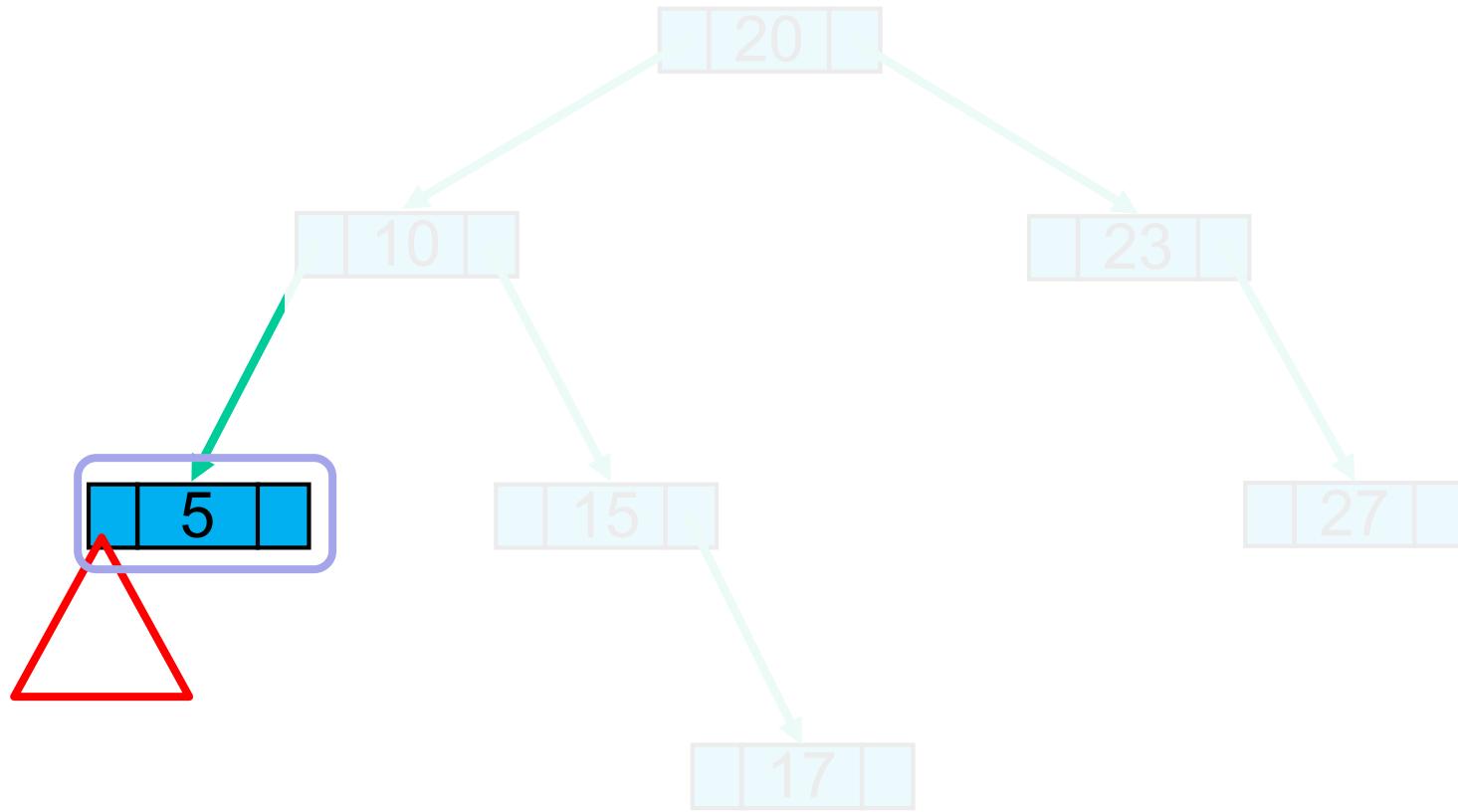
---



# Inorder Traversal: left, node, right

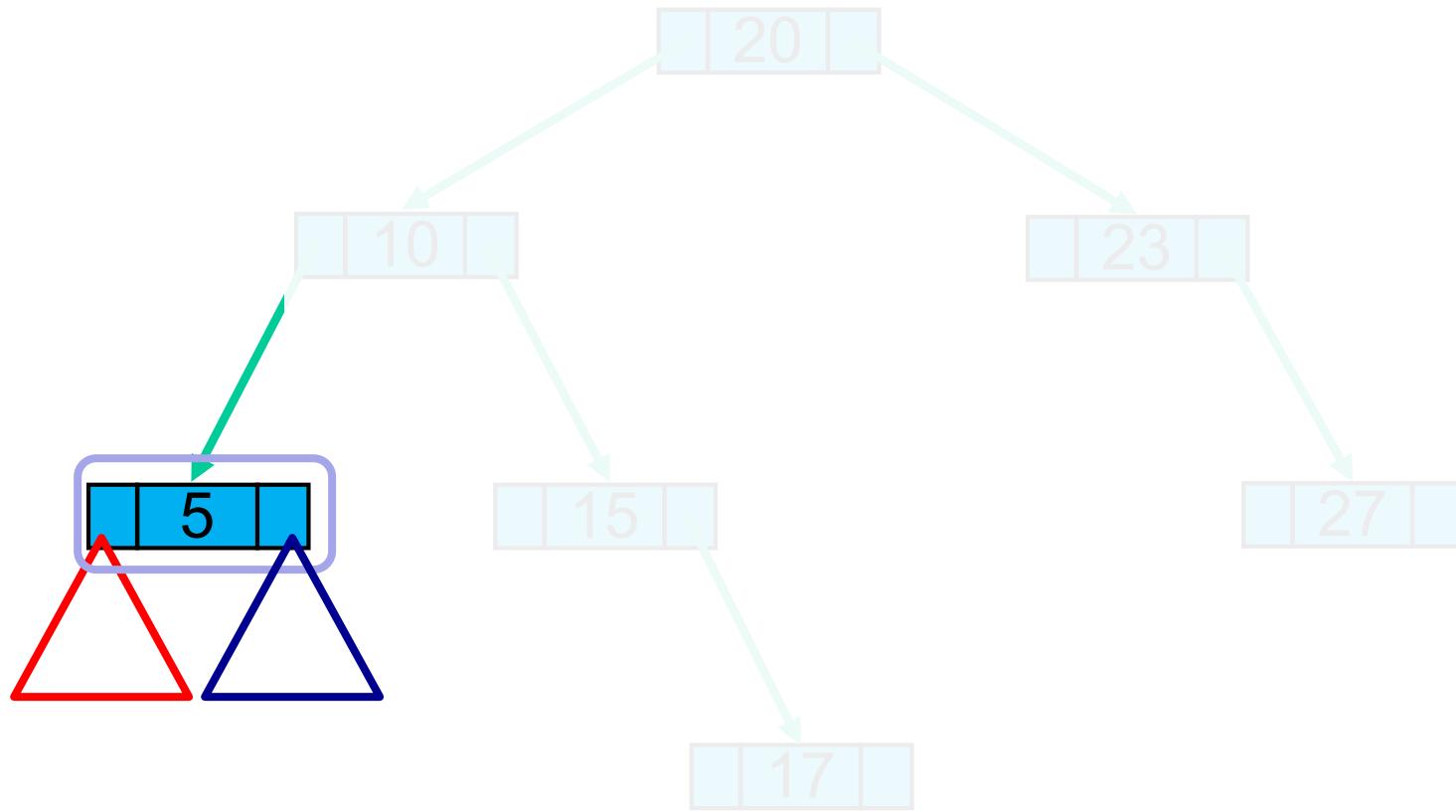


# Inorder Traversal: left, node, right



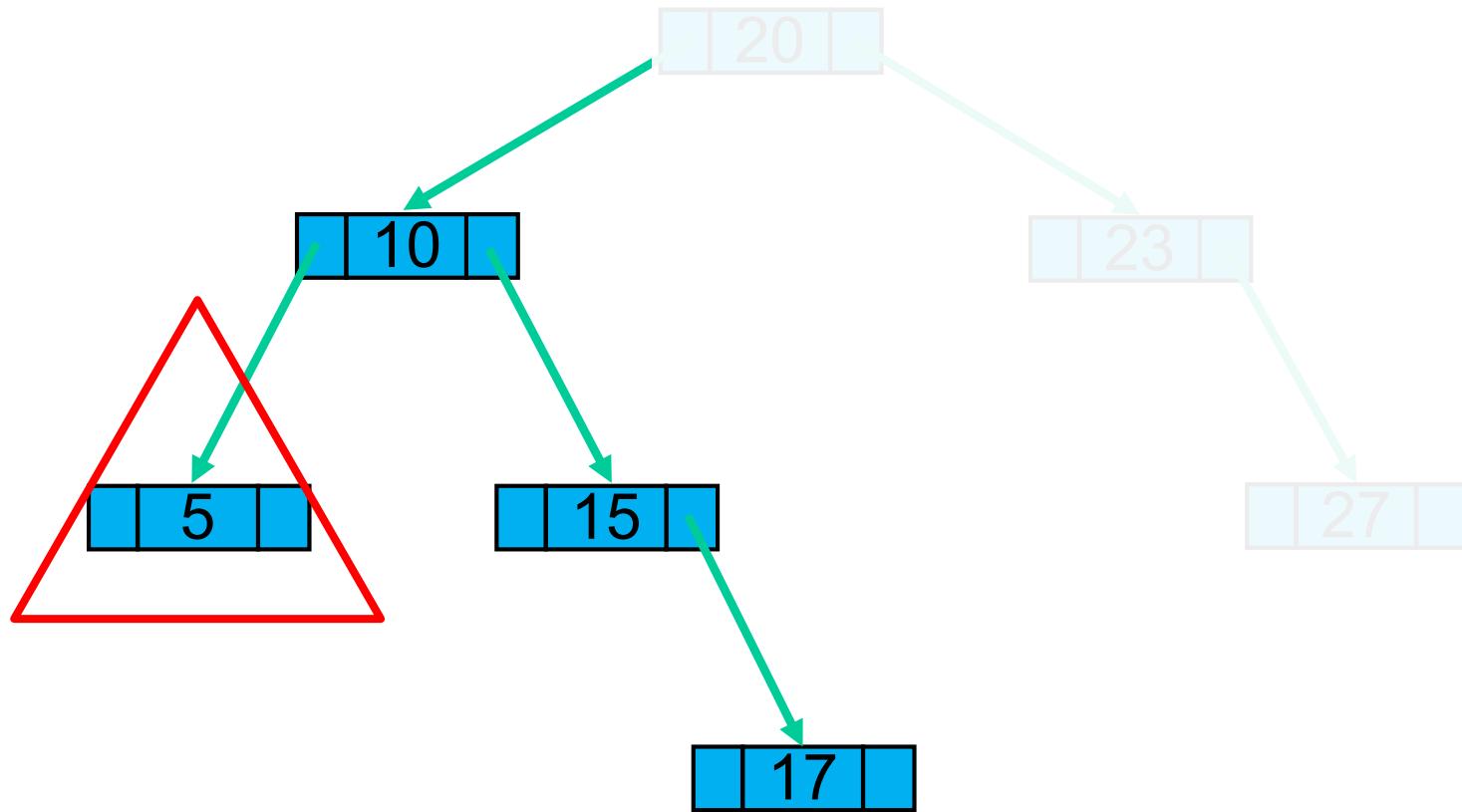
Visit: 5

# Inorder Traversal: left, node, right



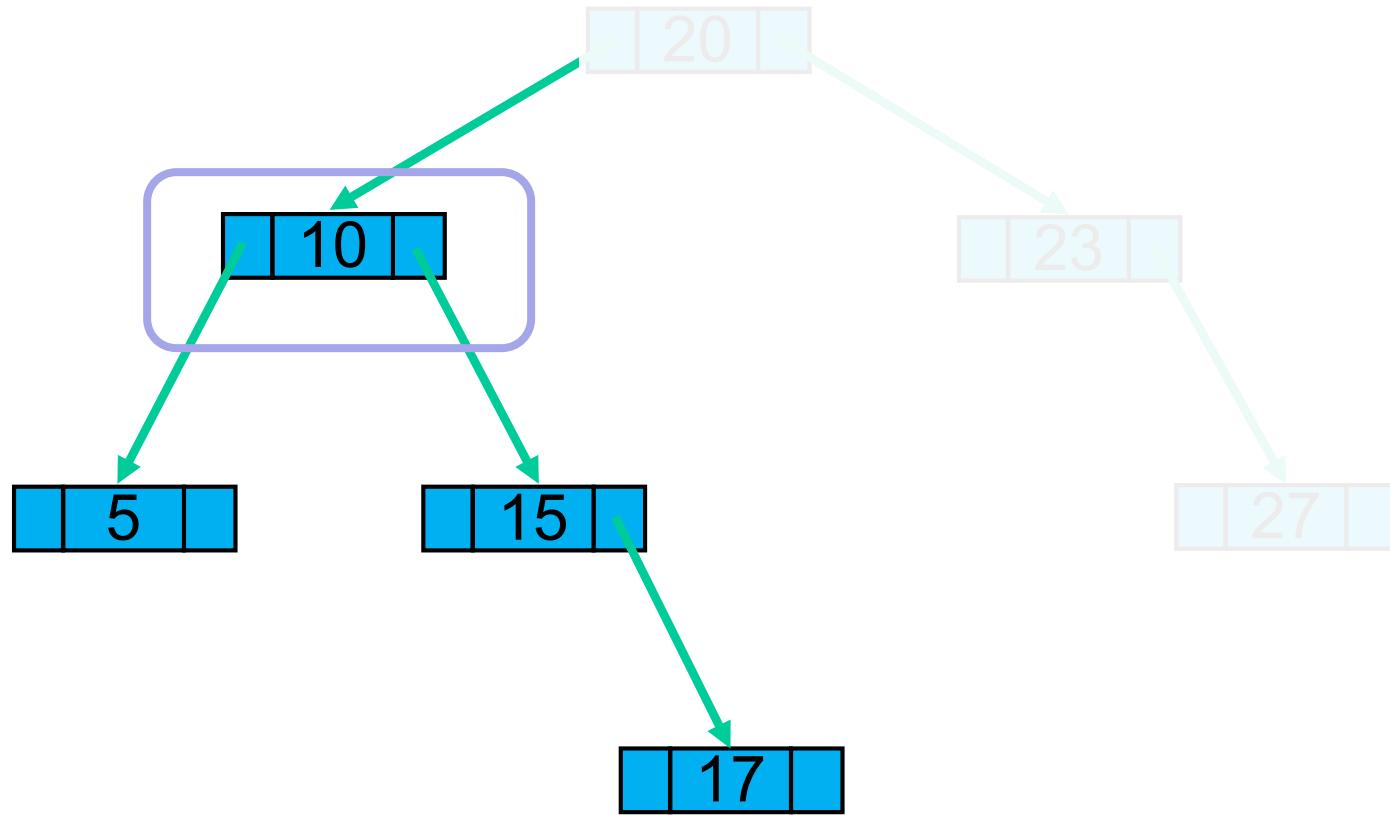
Visit: 5

# Inorder Traversal: left, node, right



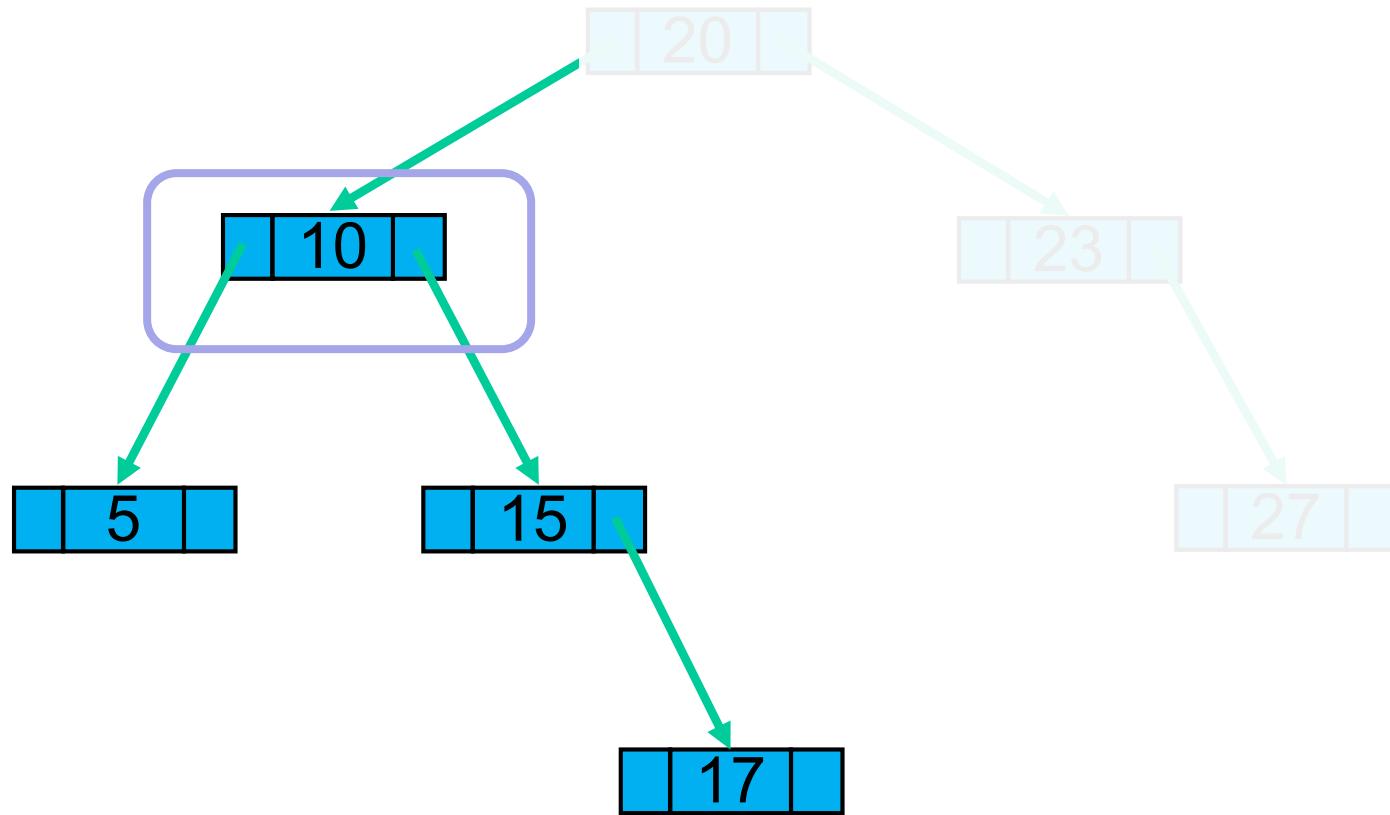
Visit: 5

# Inorder Traversal: left, node, right



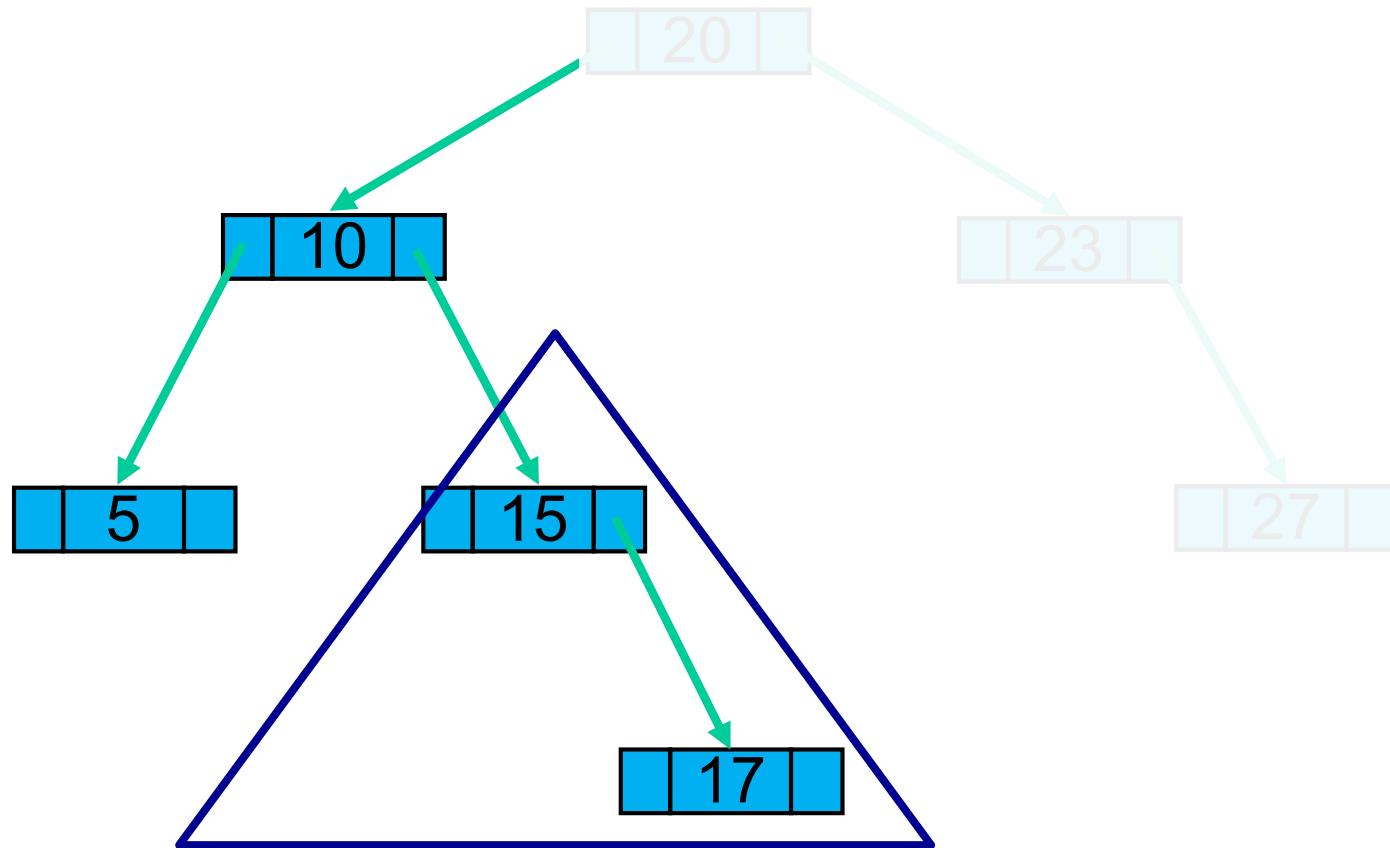
Visit: 5

# Inorder Traversal: left, node, right



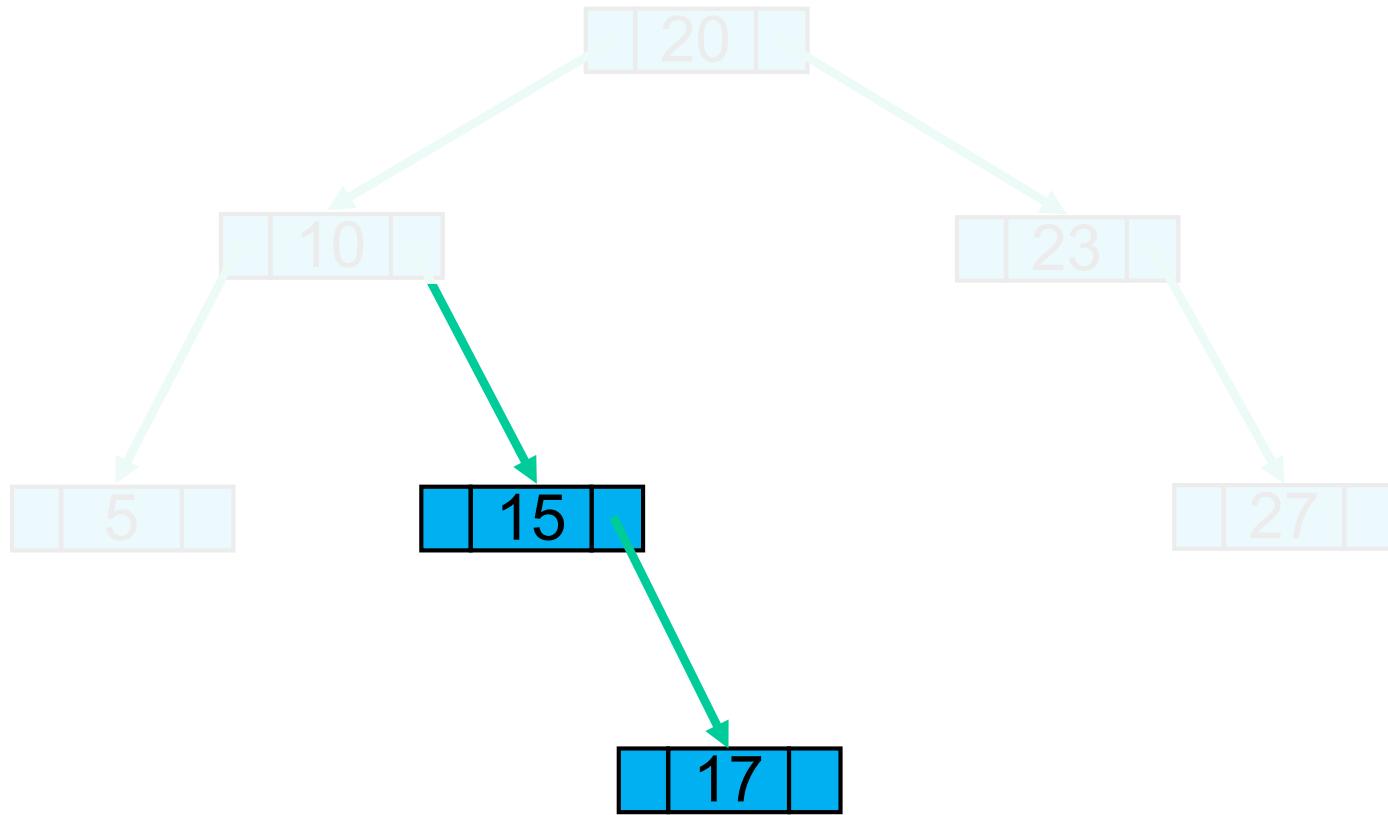
Visit: 5 10

# Inorder Traversal: left, node, right



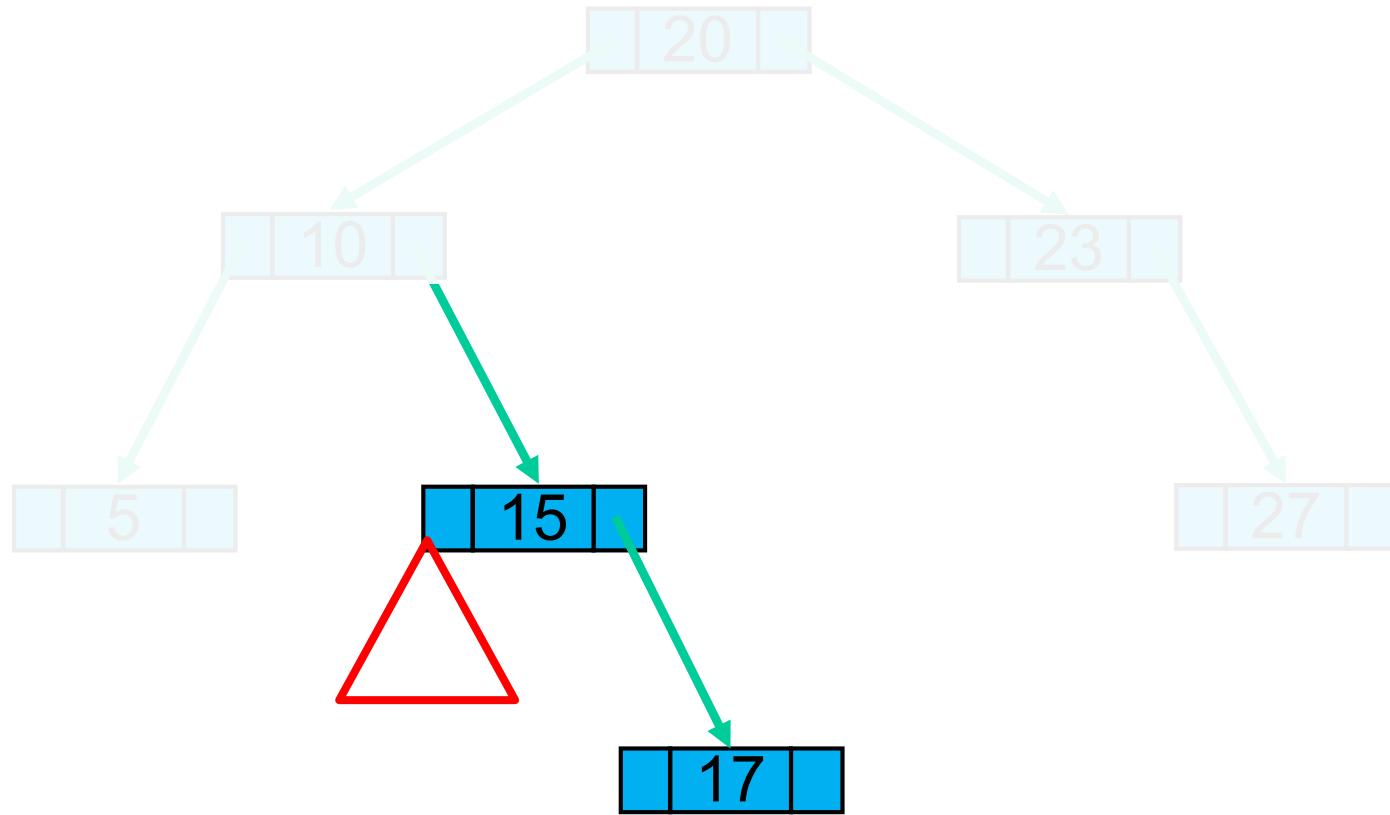
Visit: 5 10

# Inorder Traversal: left, node, right



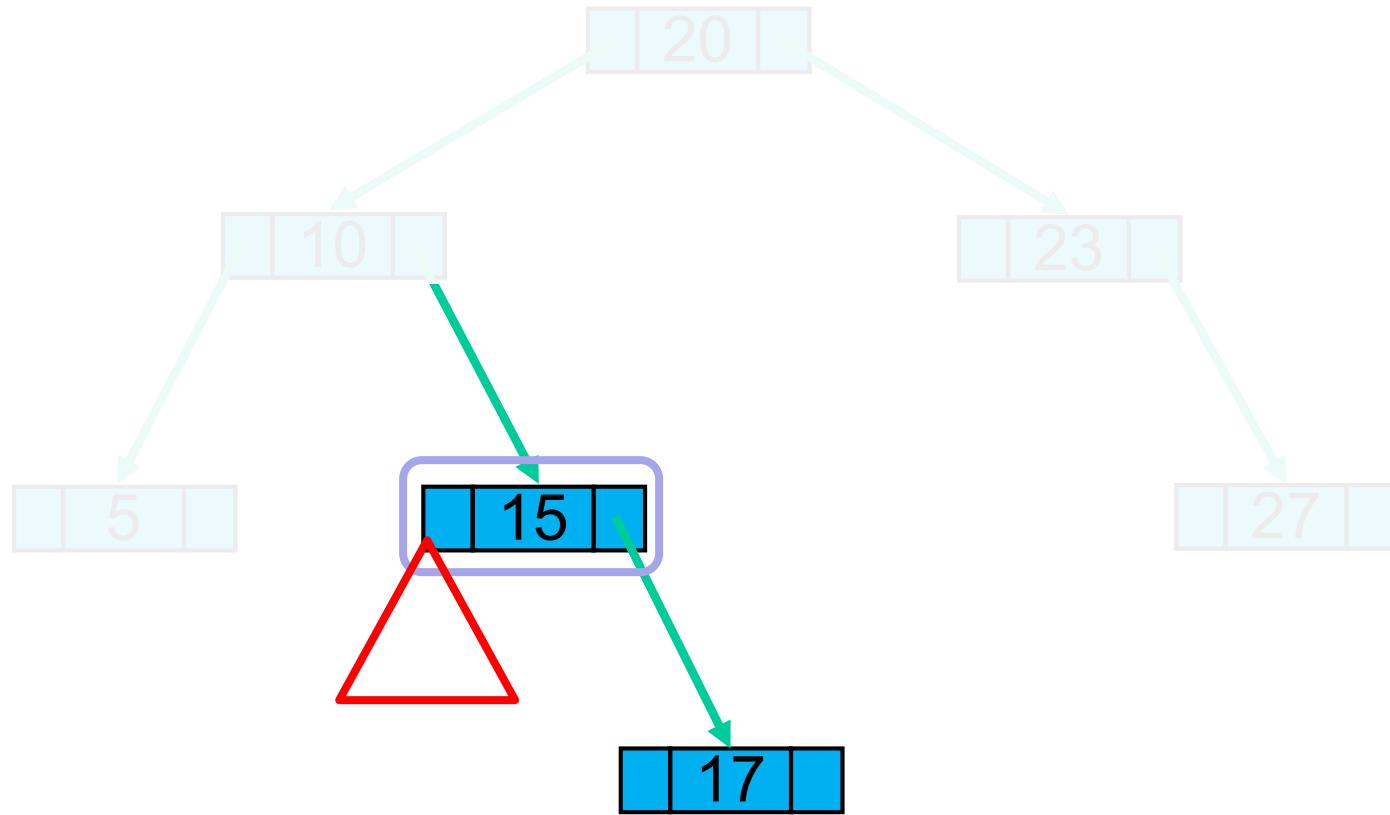
Visit: 5 10

# Inorder Traversal: left, node, right



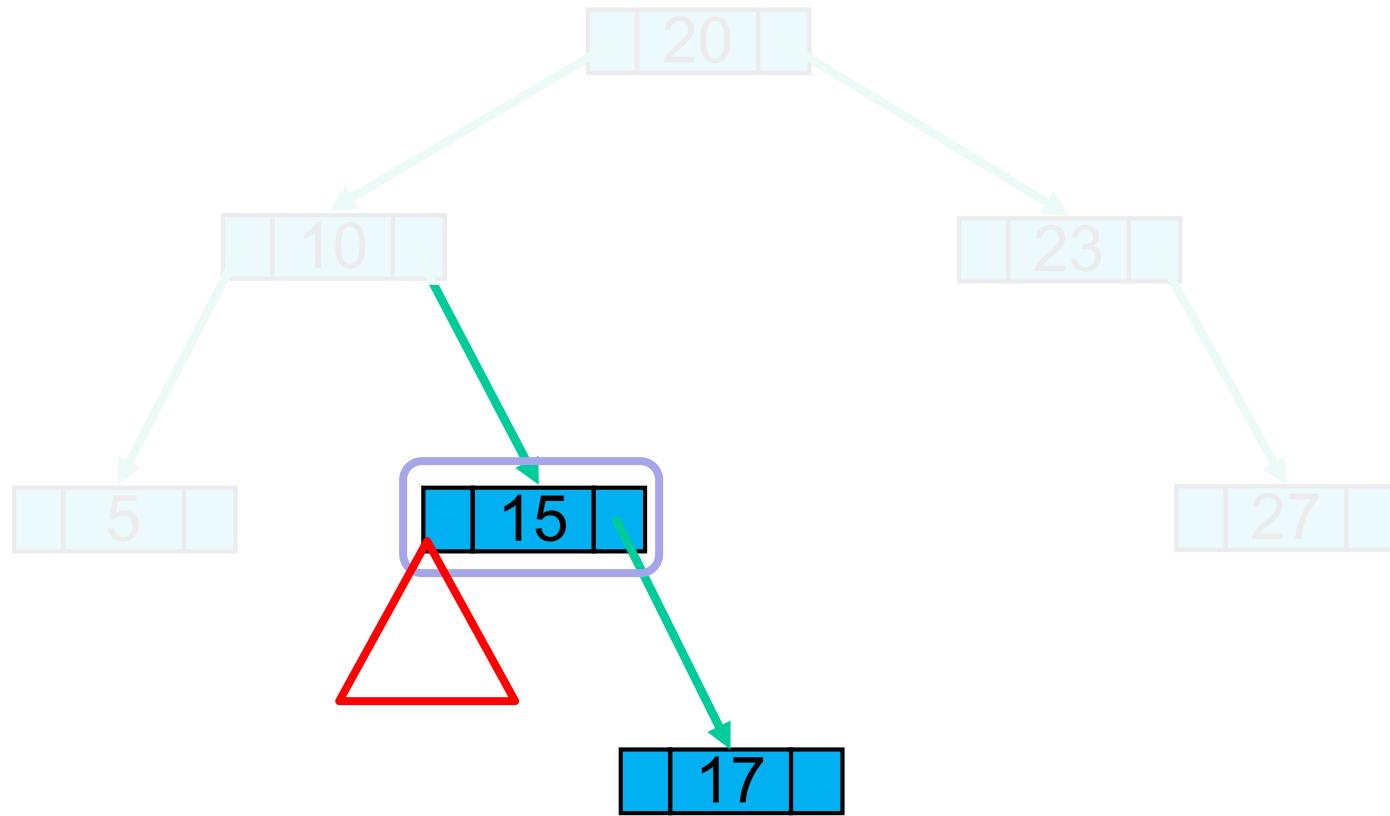
Visit: 5 10

# Inorder Traversal: left, node, right



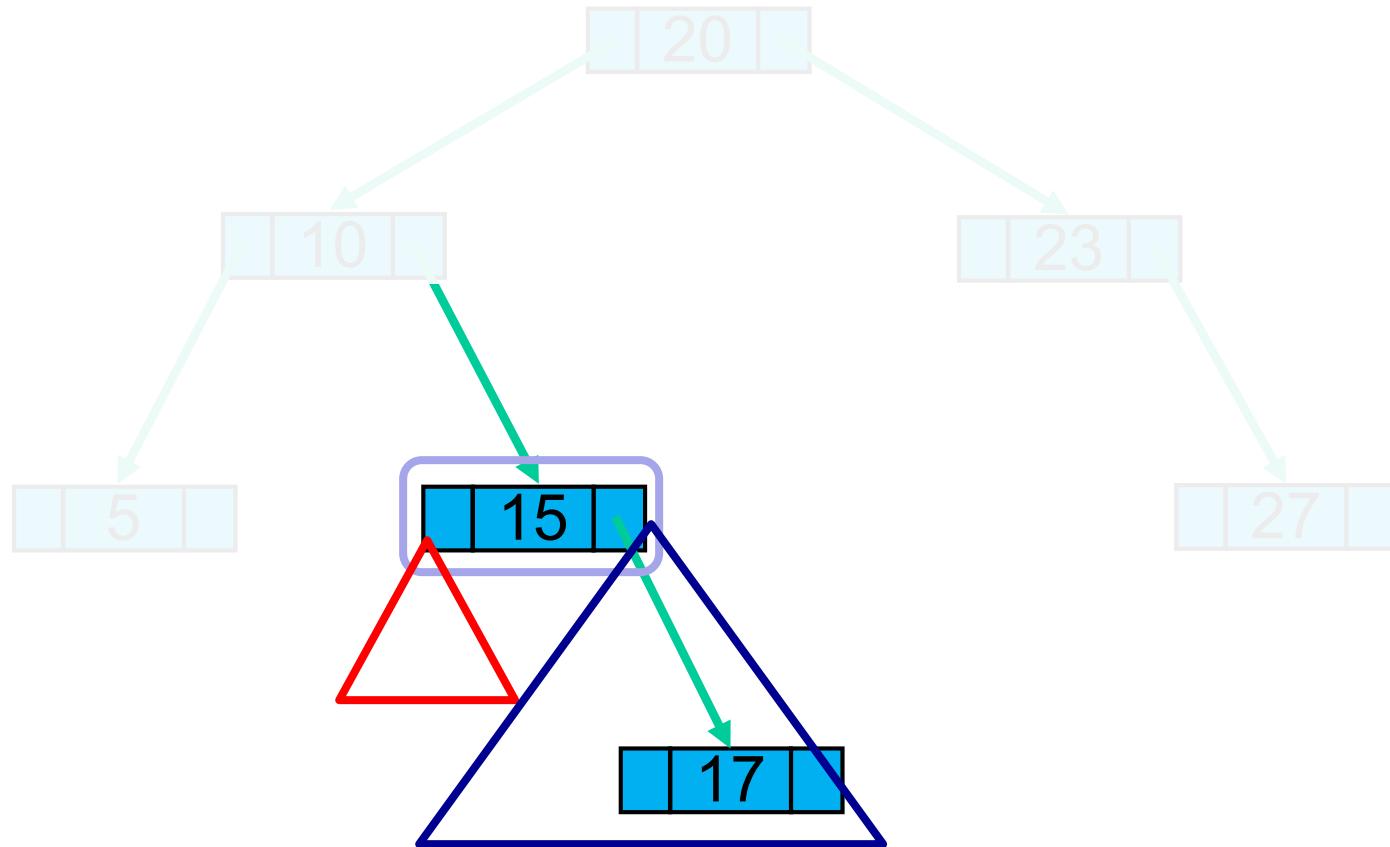
Visit: 5 10

# Inorder Traversal: left, node, right



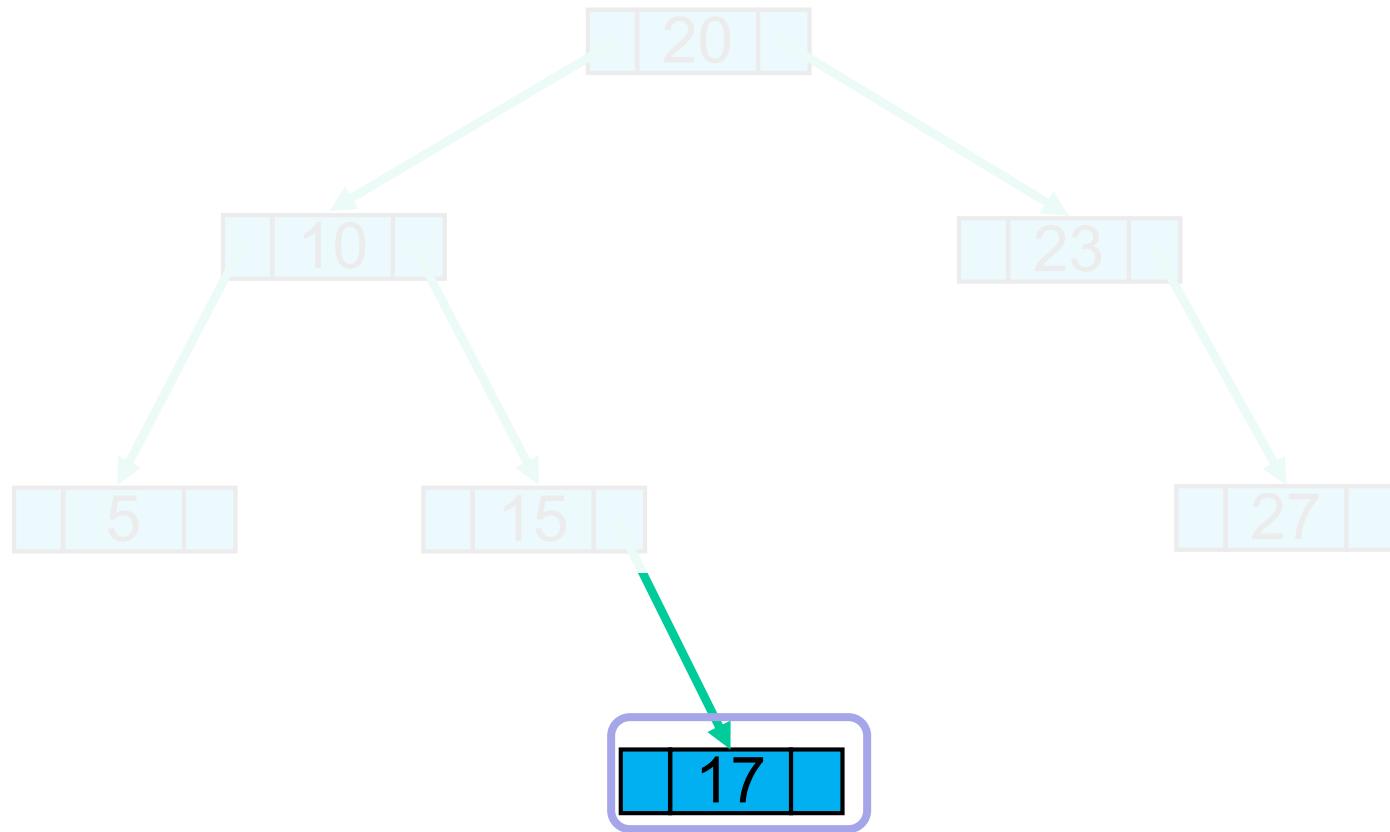
Visit: 5 10 15

# Inorder Traversal: left, node, right



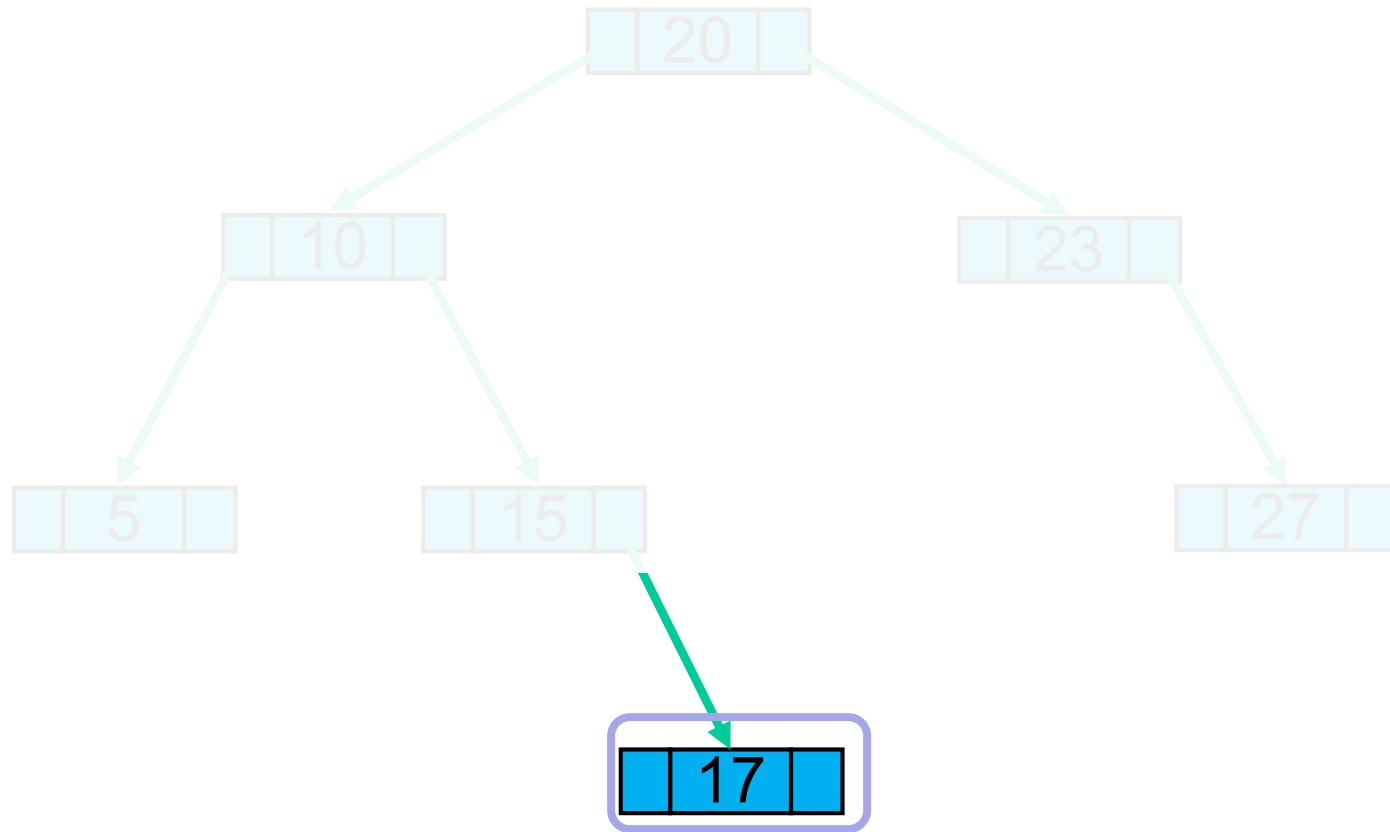
Visit: 5 10 15

# Inorder Traversal: left, node, right



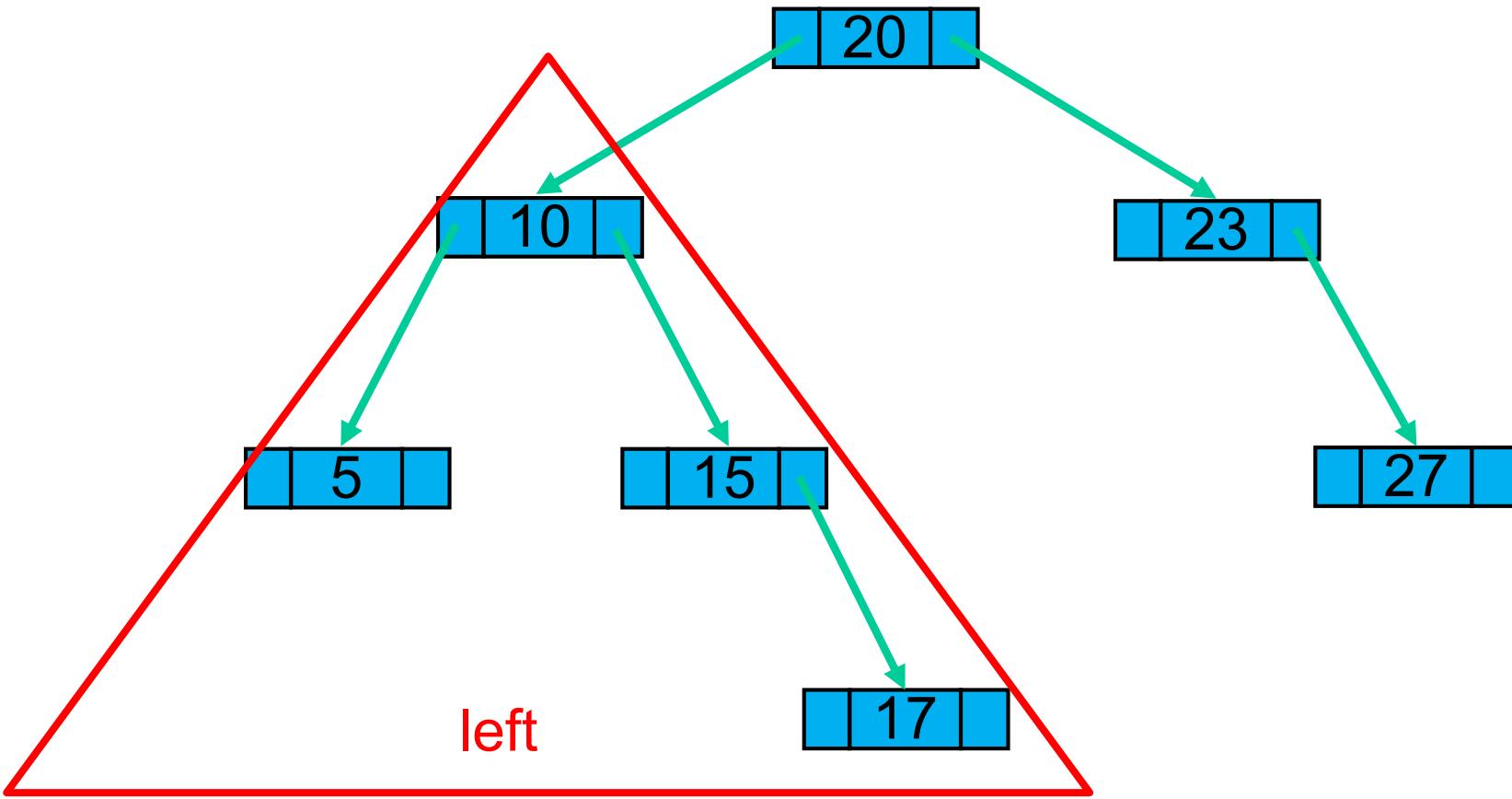
Visit: 5 10 15

# Inorder Traversal: left, node, right



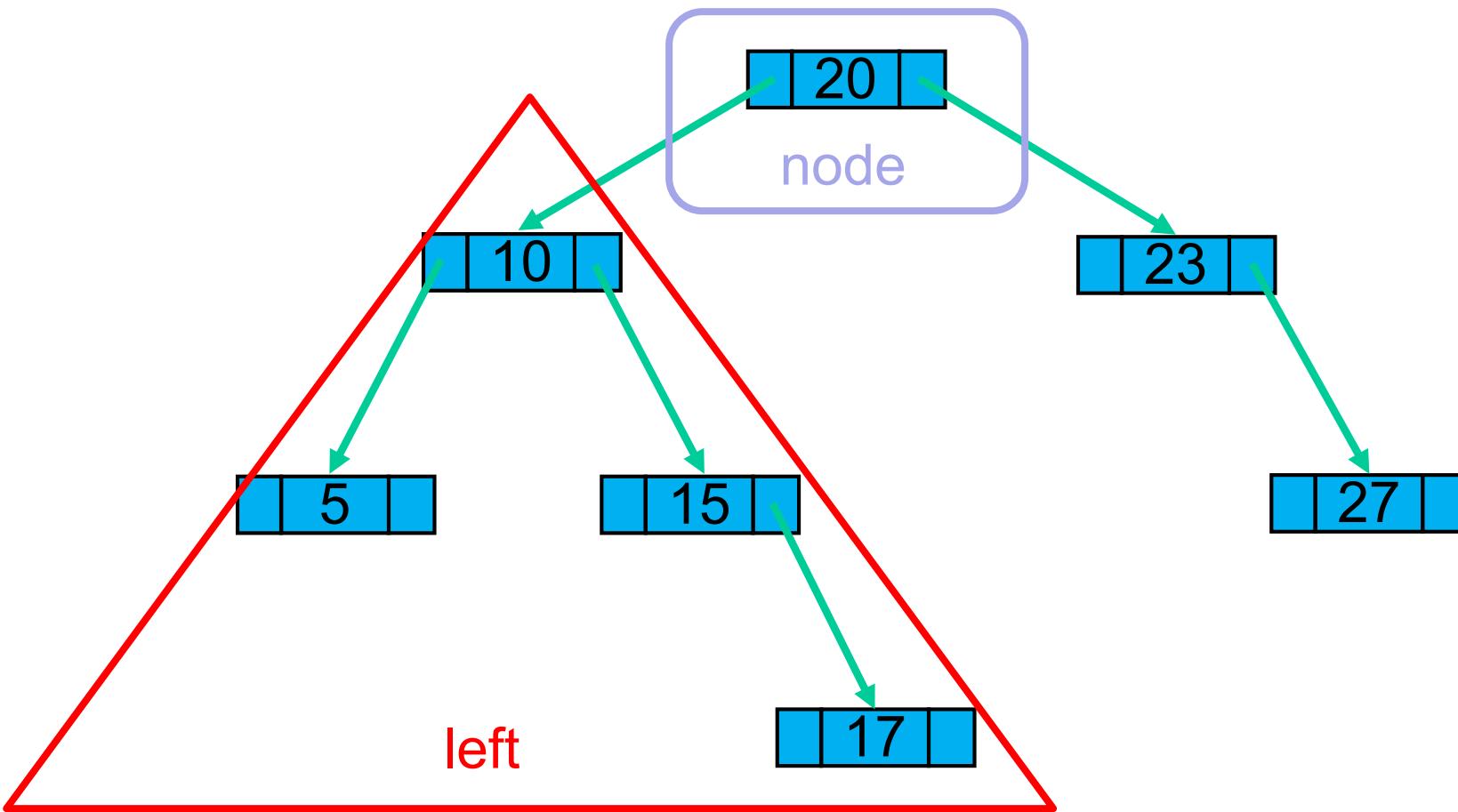
Visit: 5 10 15 17

# Inorder Traversal: left, node, right



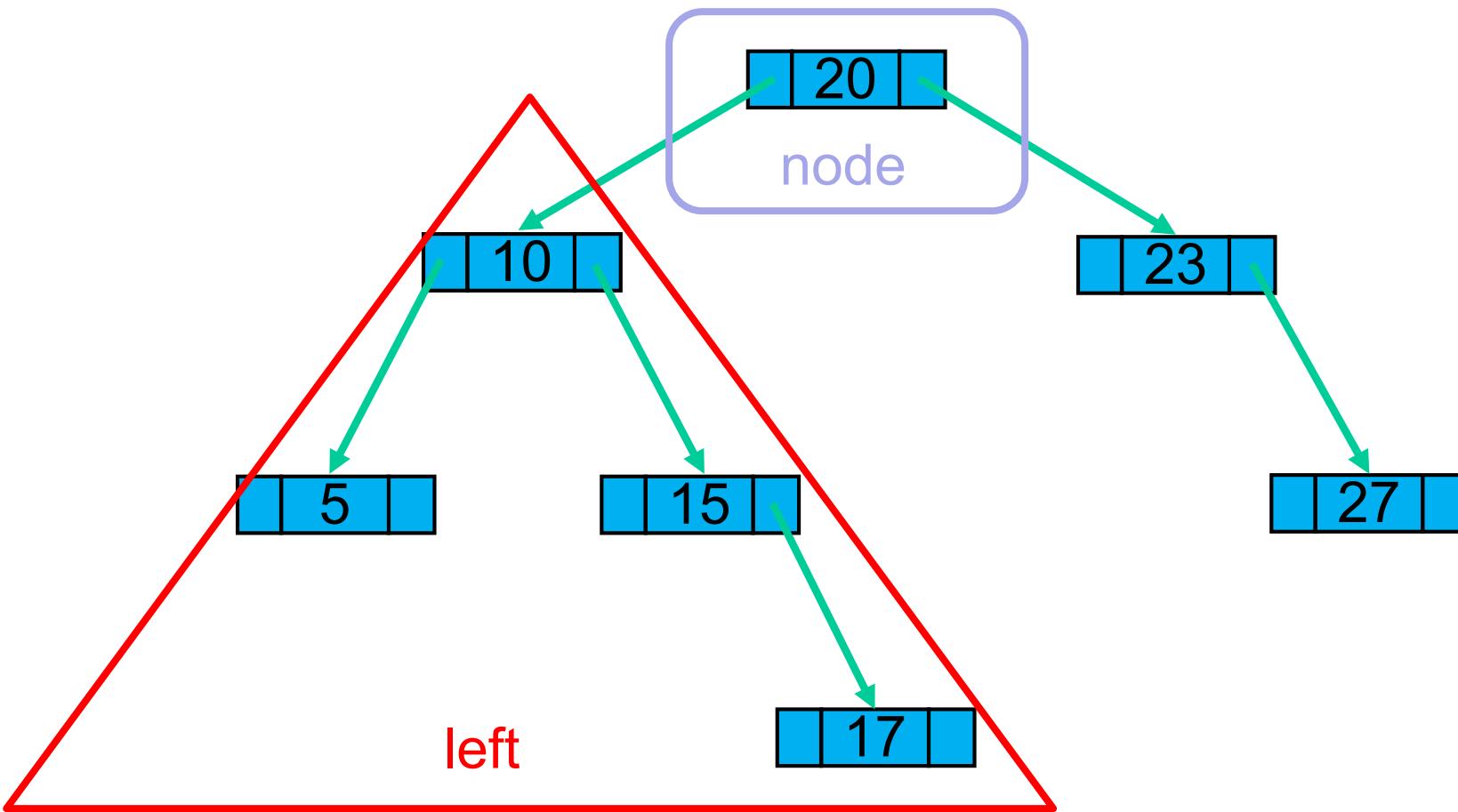
Visit: 5 10 15 17

# Inorder Traversal: left, node, right



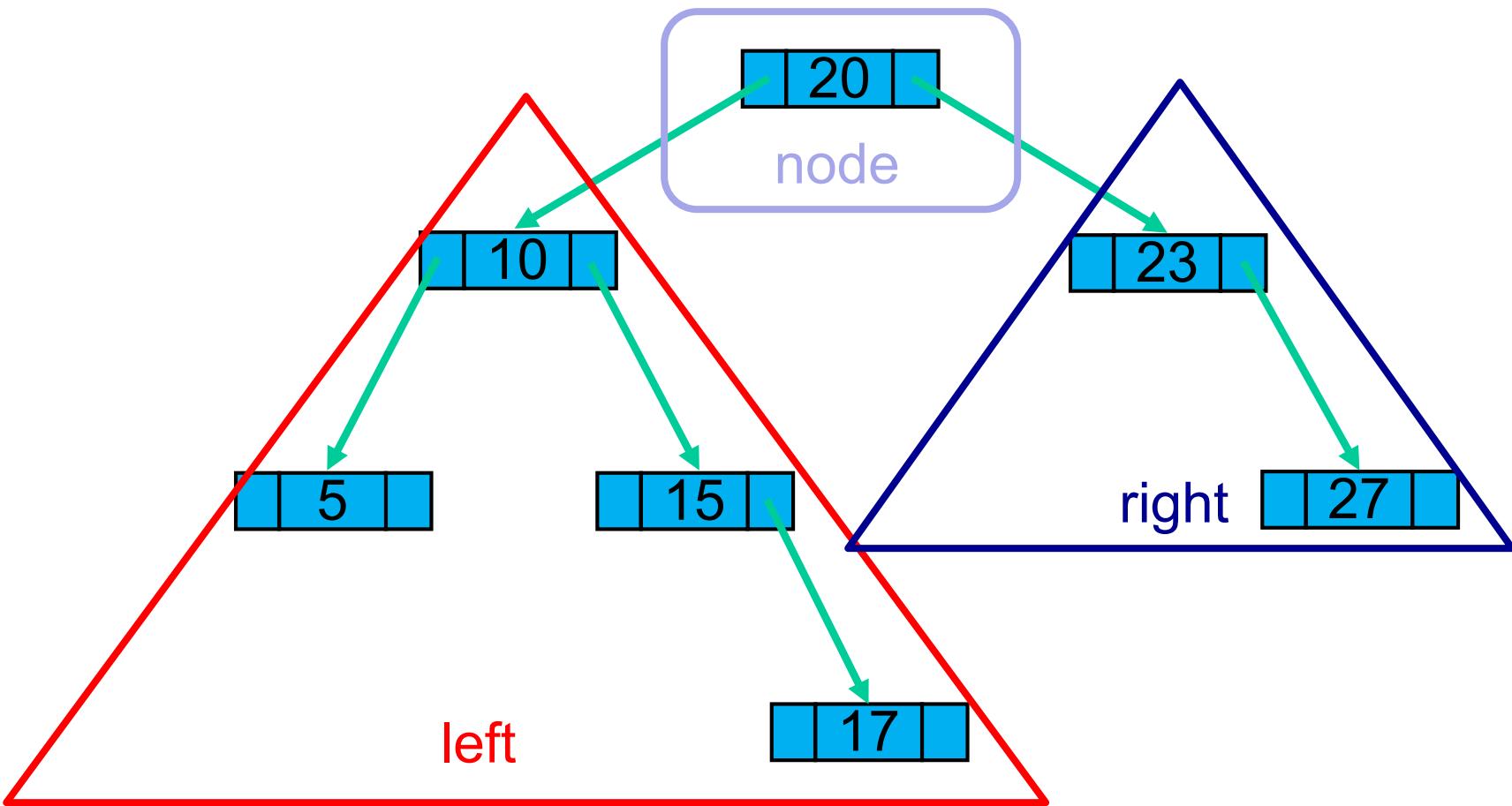
Visit: 5 10 15 17

# Inorder Traversal: left, node, right



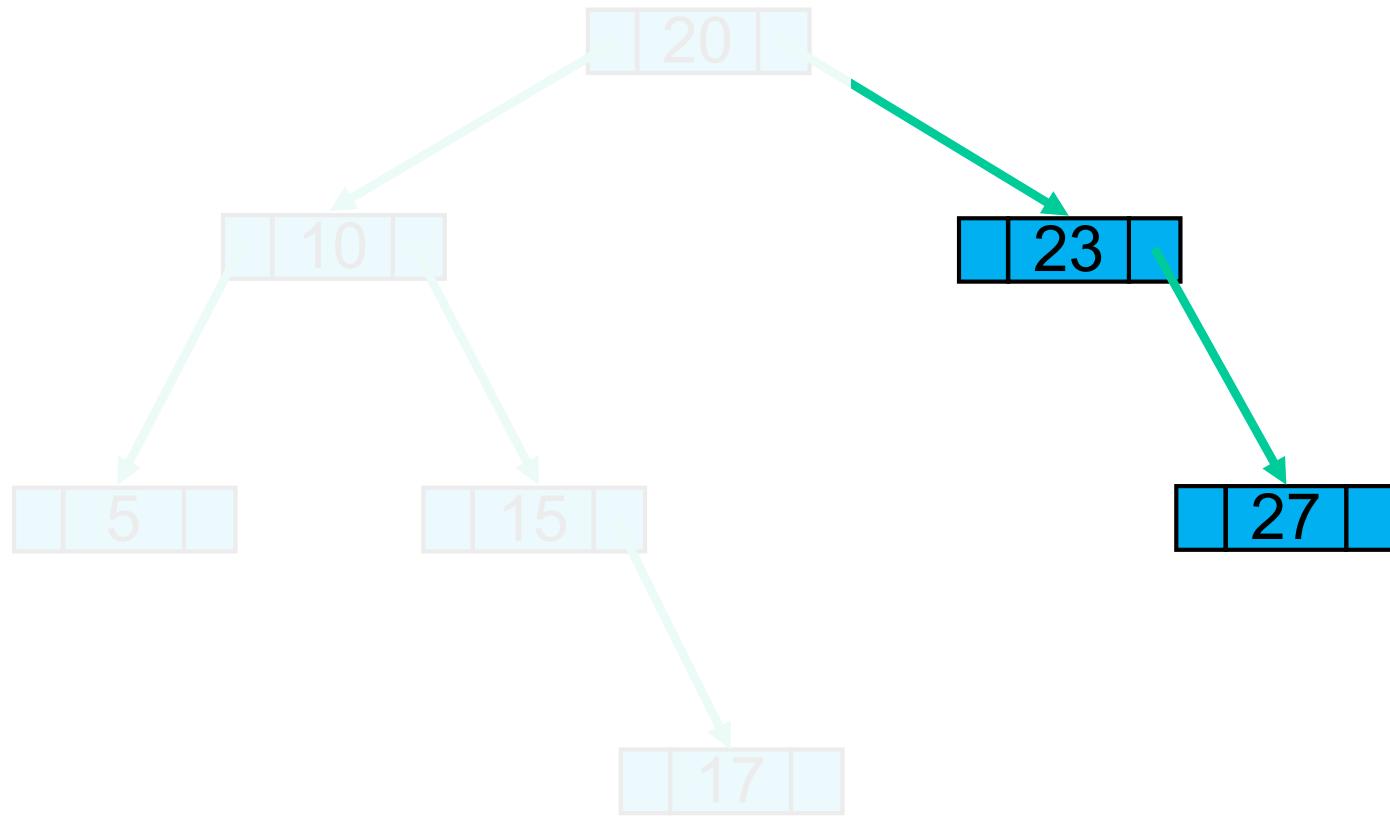
Visit: 5 10 15 17 20

# Inorder Traversal: left, node, right



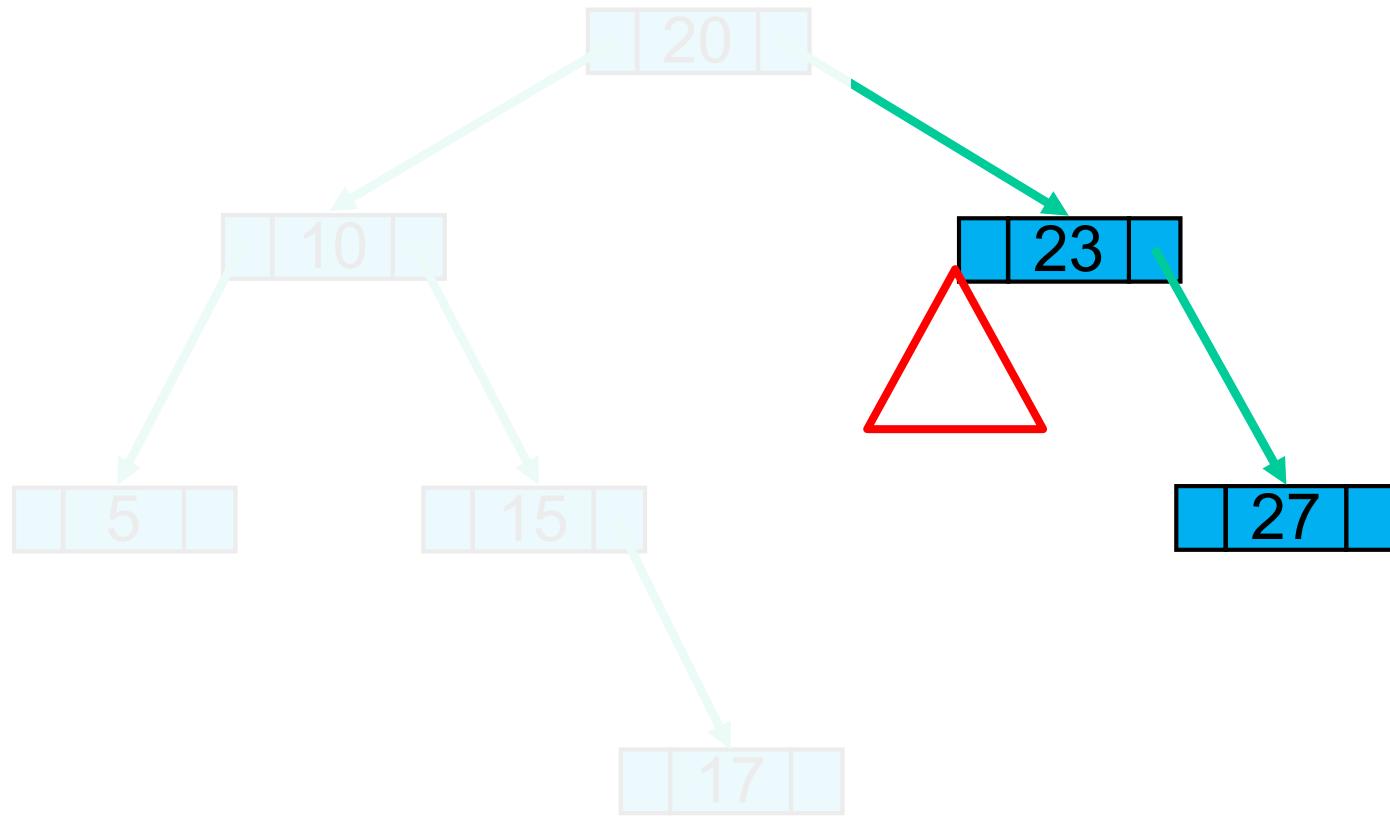
Visit: 5 10 15 17 20

# Inorder Traversal: left, node, right



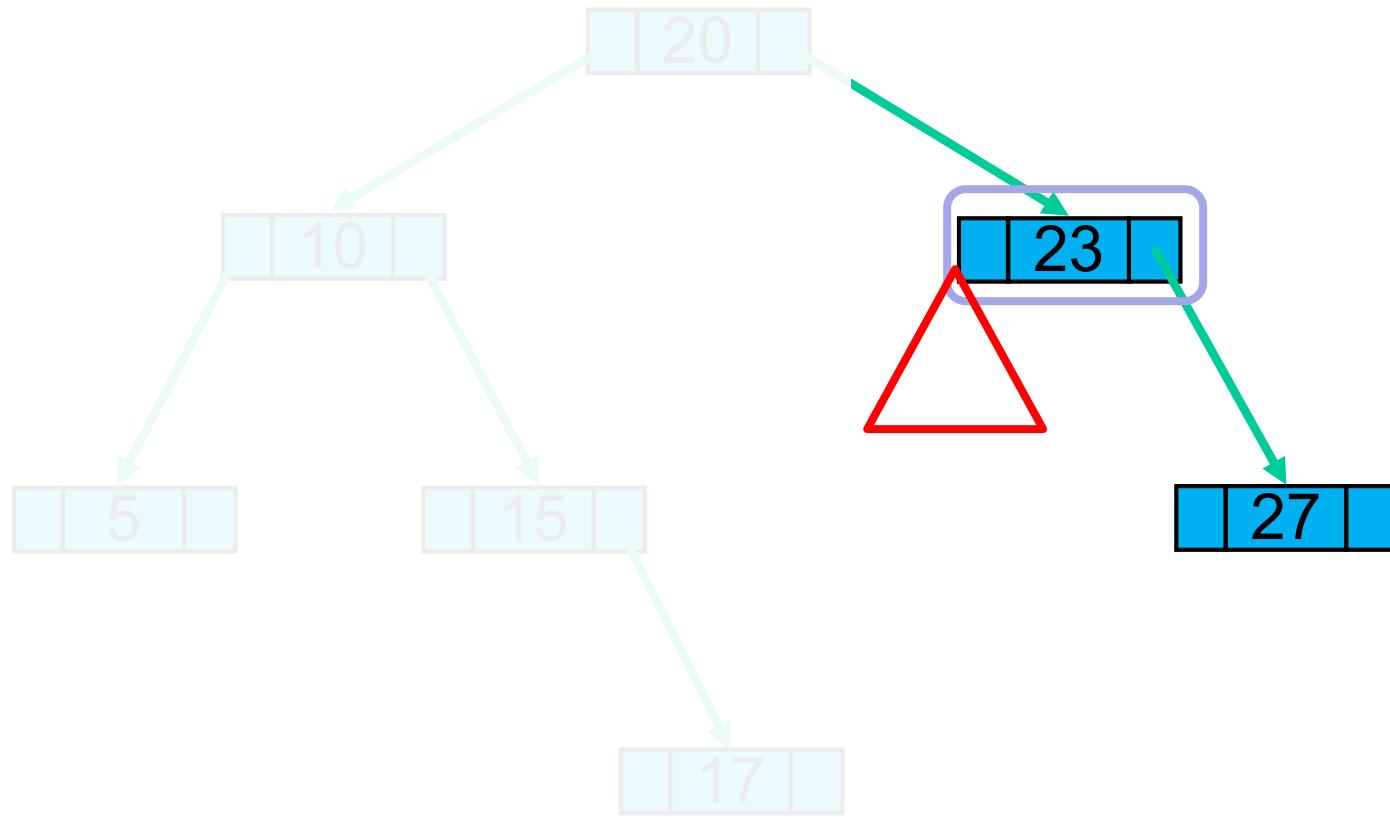
Visit: 5 10 15 17 20

# Inorder Traversal: left, node, right



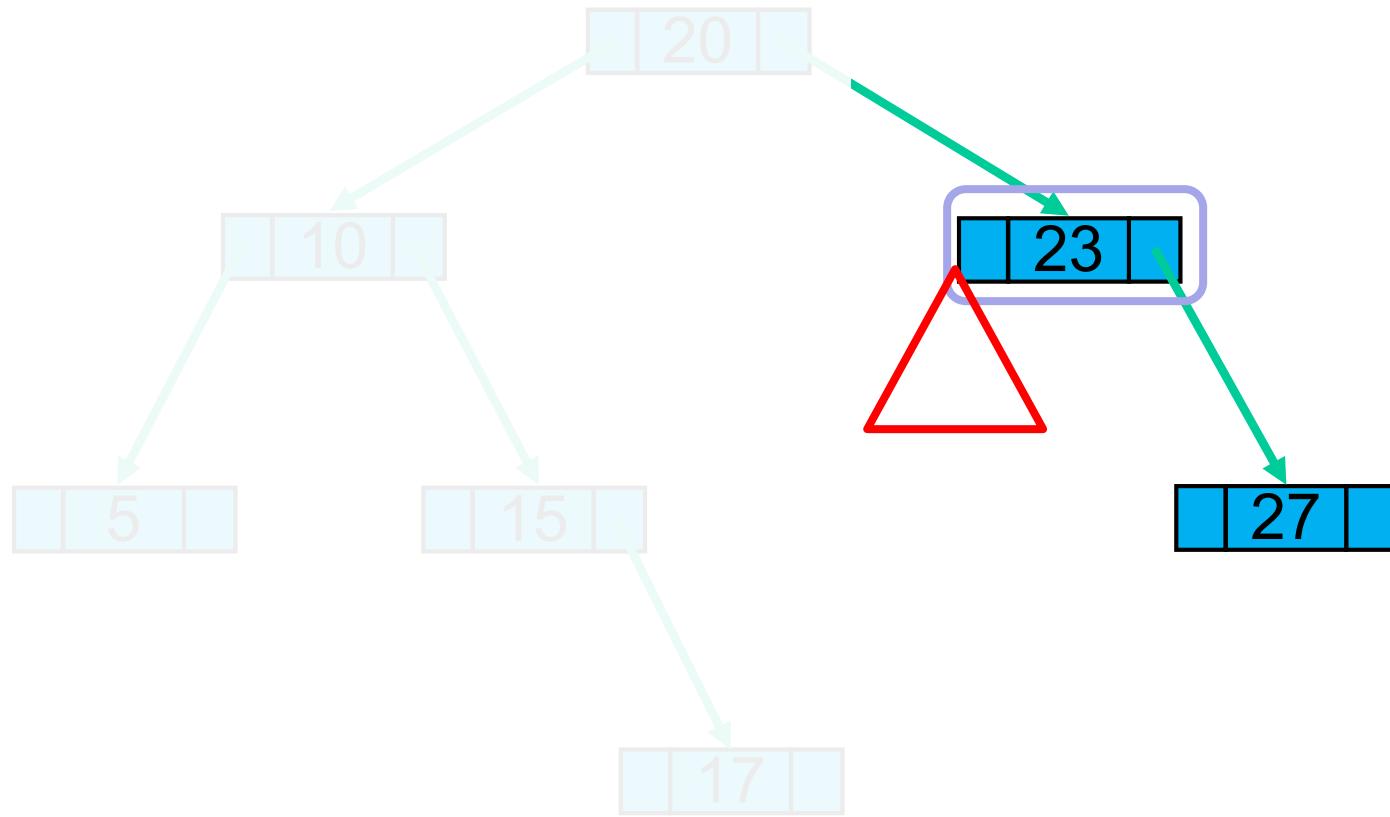
Visit: 5 10 15 17 20

# Inorder Traversal: left, node, right



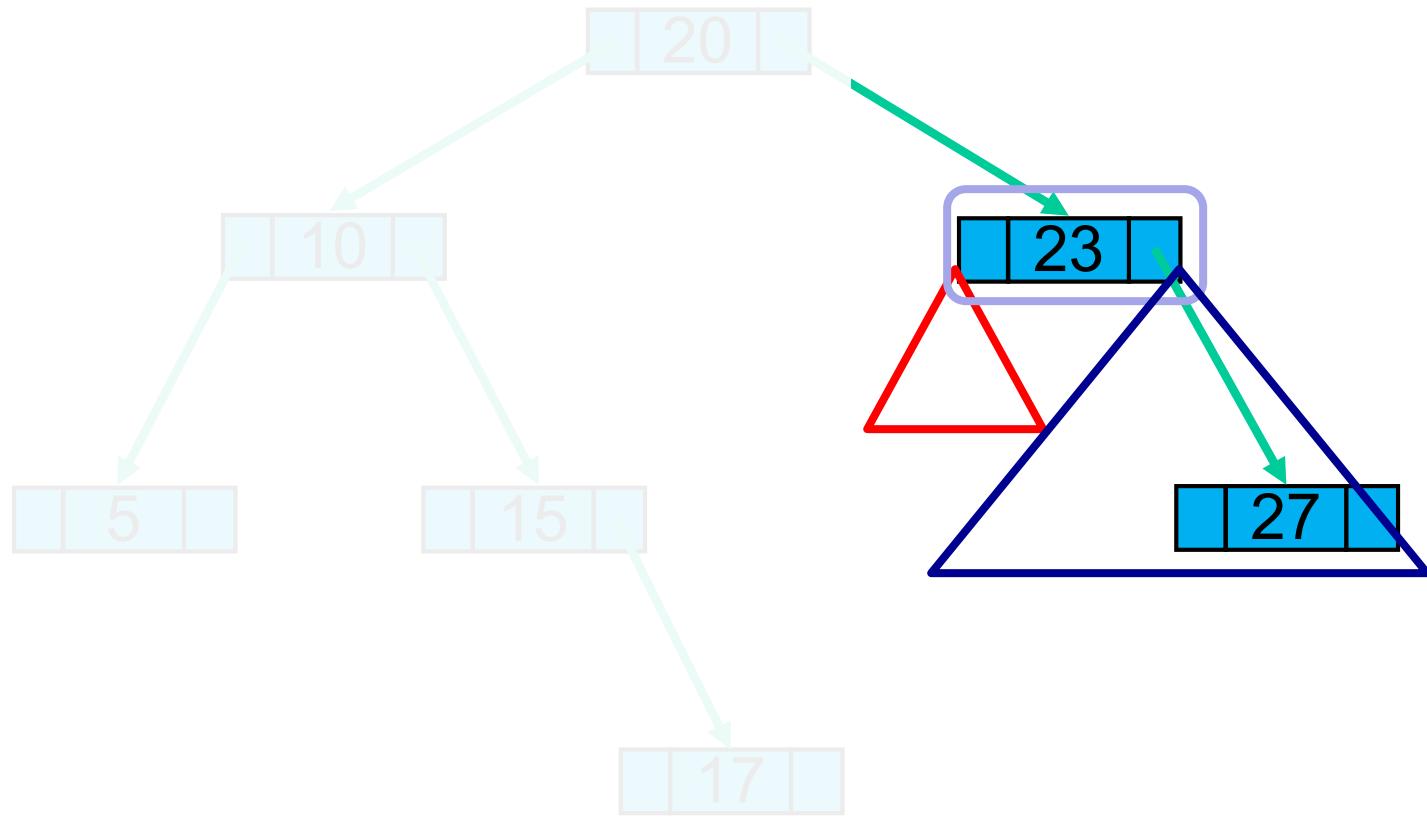
Visit: 5 10 15 17 20

# Inorder Traversal: left, node, right



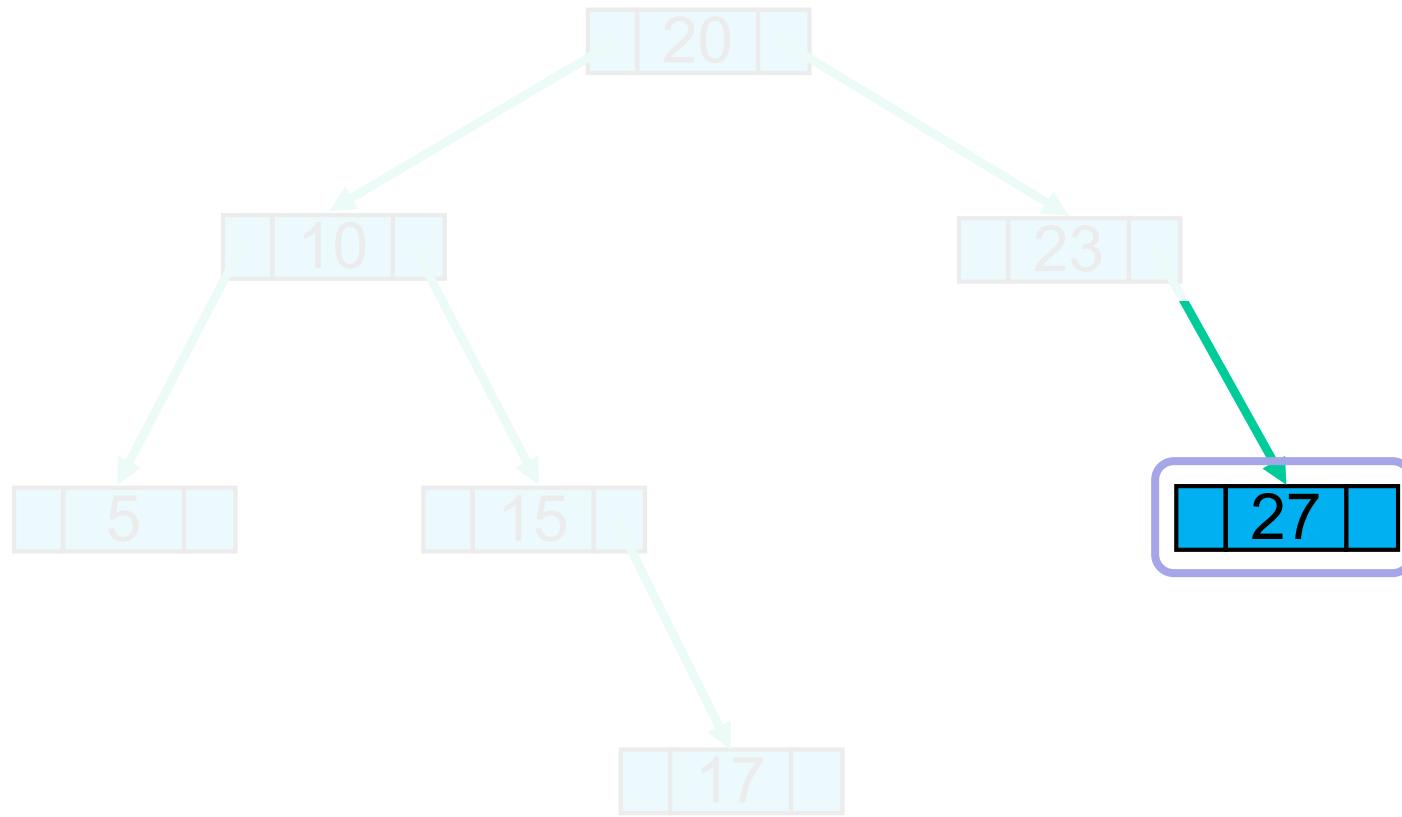
Visit: 5 10 15 17 20 23

# Inorder Traversal: left, node, right



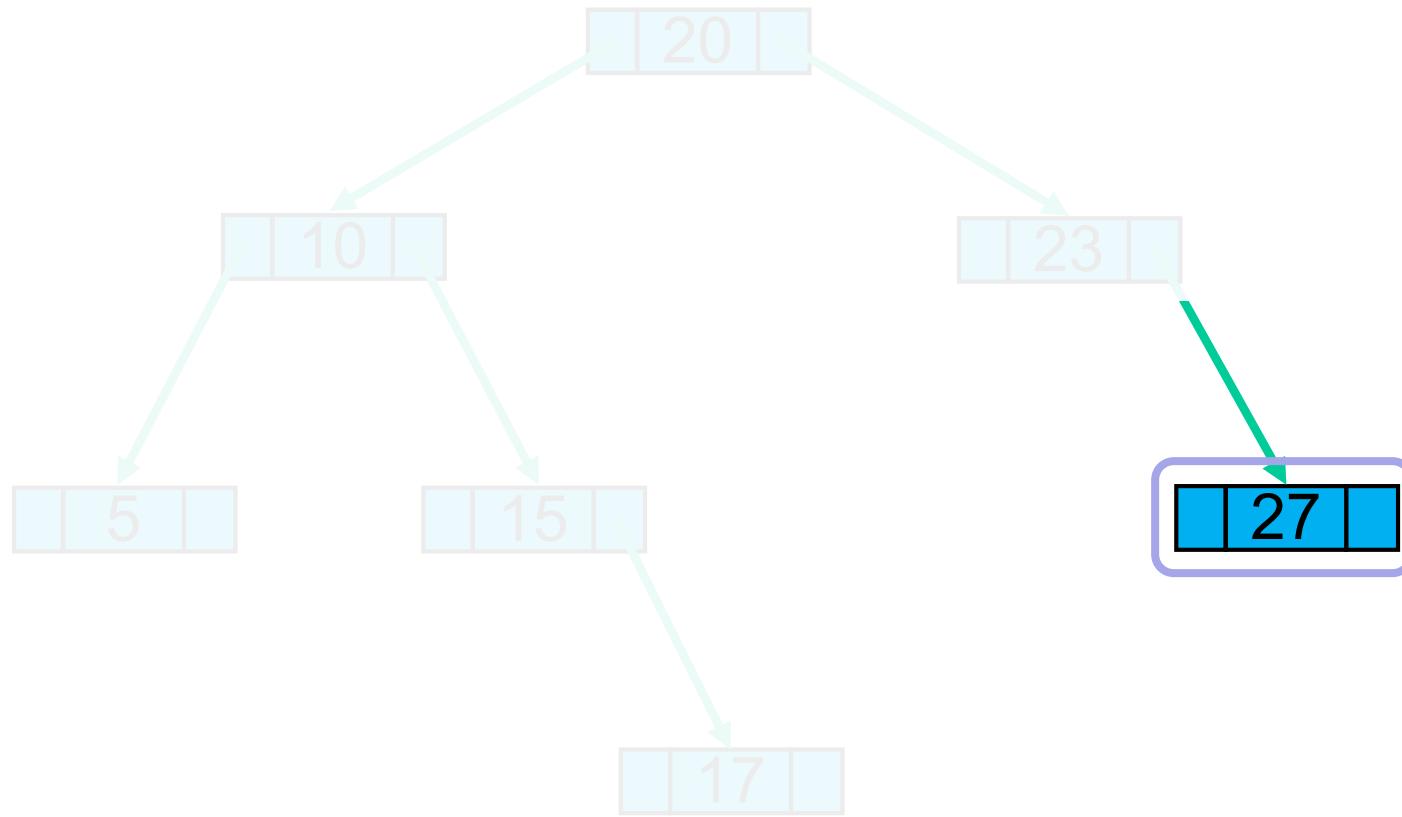
Visit: 5 10 15 17 20 23

# Inorder Traversal: left, node, right



Visit: 5 10 15 17 20 23

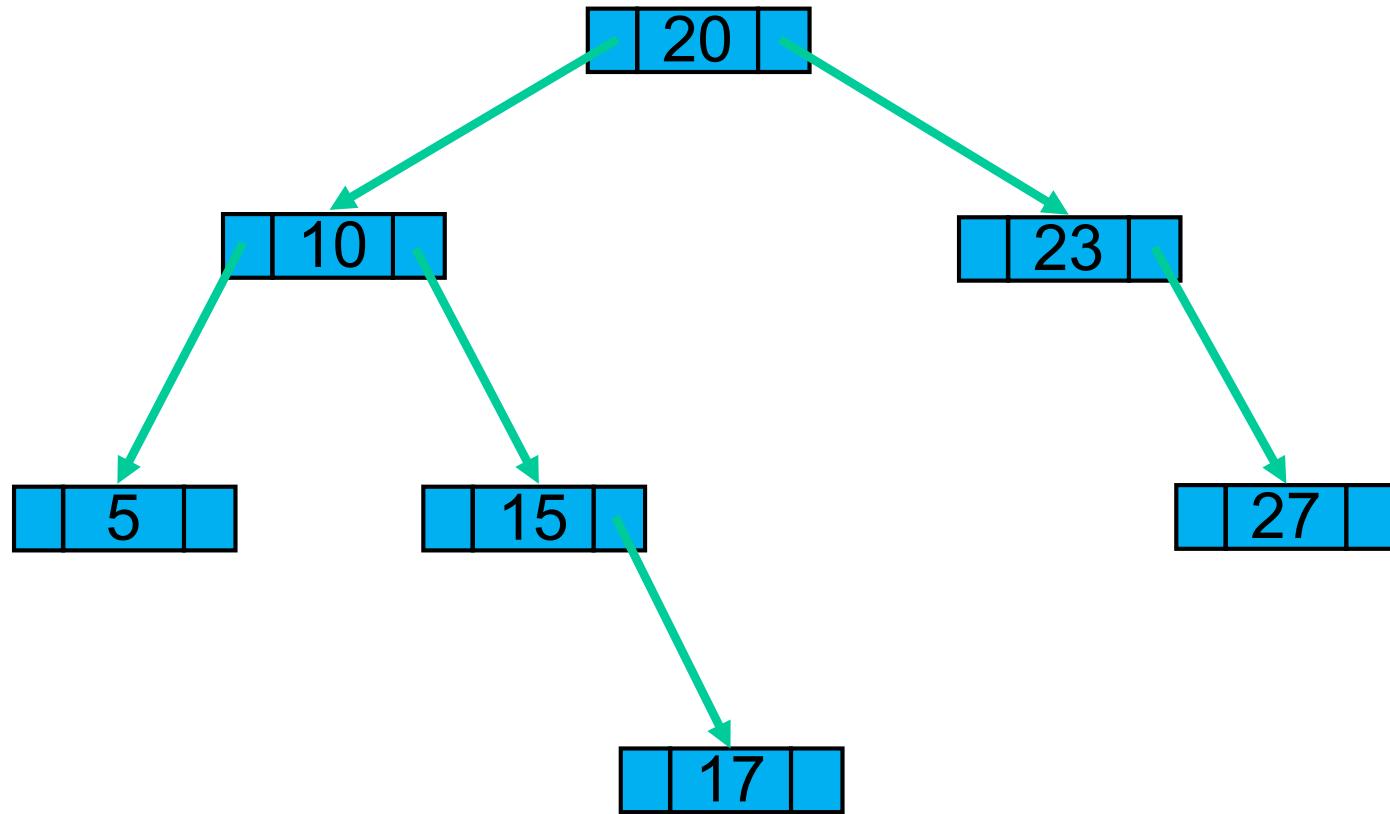
# Inorder Traversal: left, node, right



Visit: 5 10 15 17 20 23 27

# Inorder Traversal: left, node, right

---



Visit: 5 10 15 17 20 23 27

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Inorder Traversal Implementation

---

```
public void inorderTraversal(Node n) {  
    if (n == null) return;  
    inorderTraversal(n.leftChild);  
    visit(n);  
    inorderTraversal(n.rightChild);  
}  
  
public void visit(Node n) {  
    System.out.println(n.value);  
}
```

# Recap: Binary Search Trees

---

- A **Binary Search Tree** is a data structure in which each node has two child nodes
- For each node, smaller values are in its left subtree
- And larger values are in its right subtree
- By performing **inorder traversal**, we can get the values in sorted order

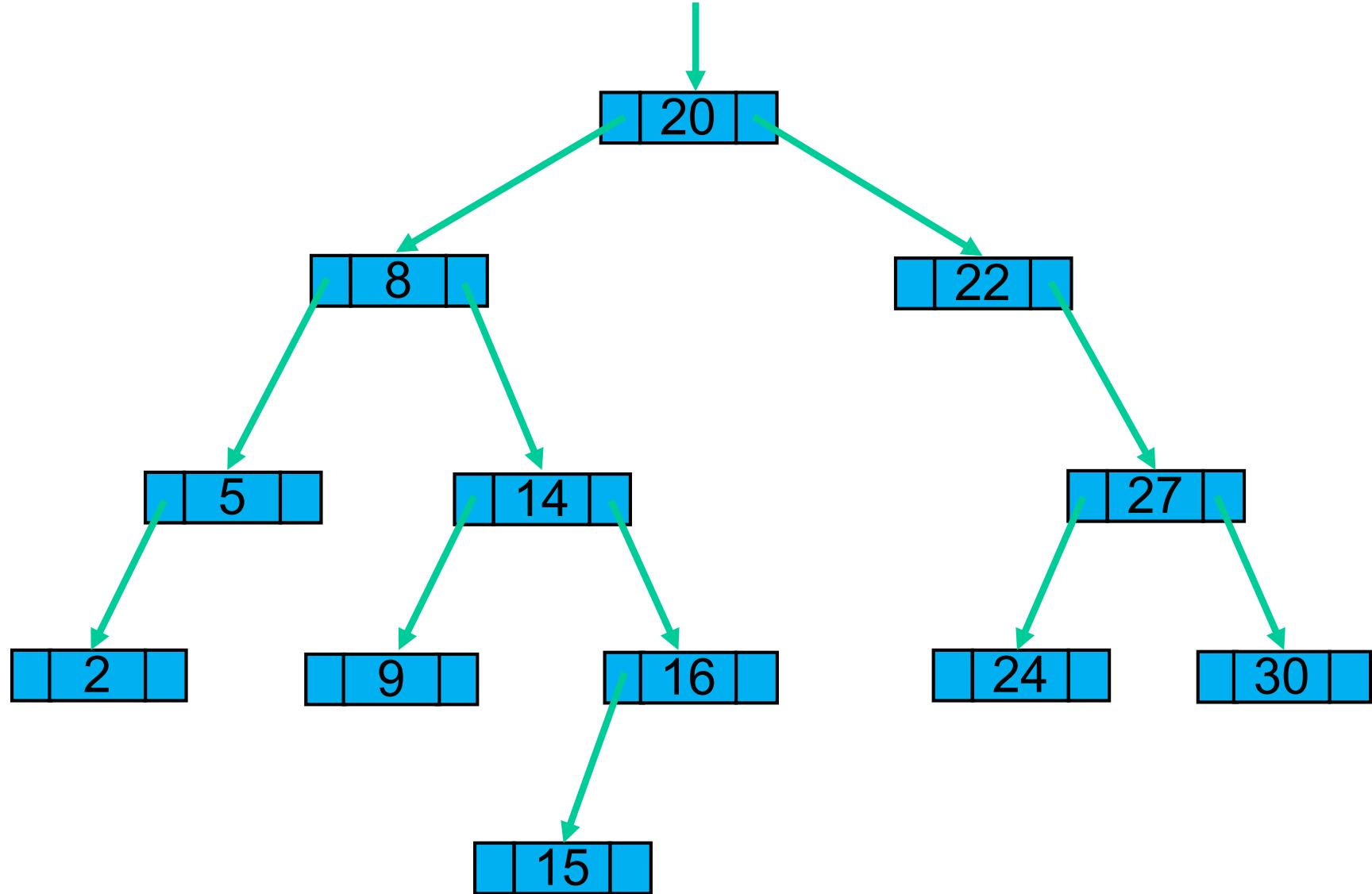
# **SD2x2.3**

# **Binary Search Trees – Contains, add**

# **Chris**

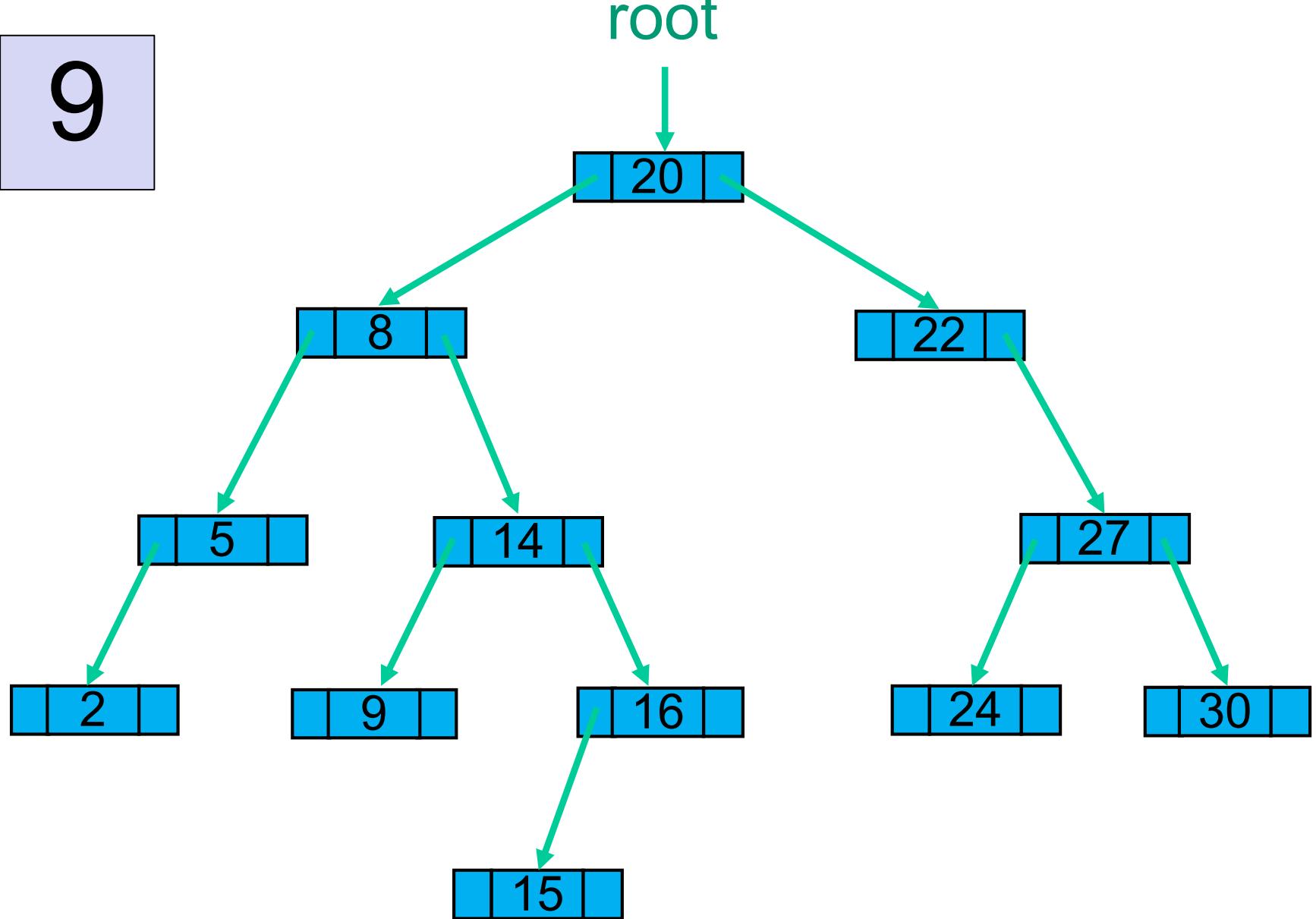
# How do we search for a value in a binary search tree?

root

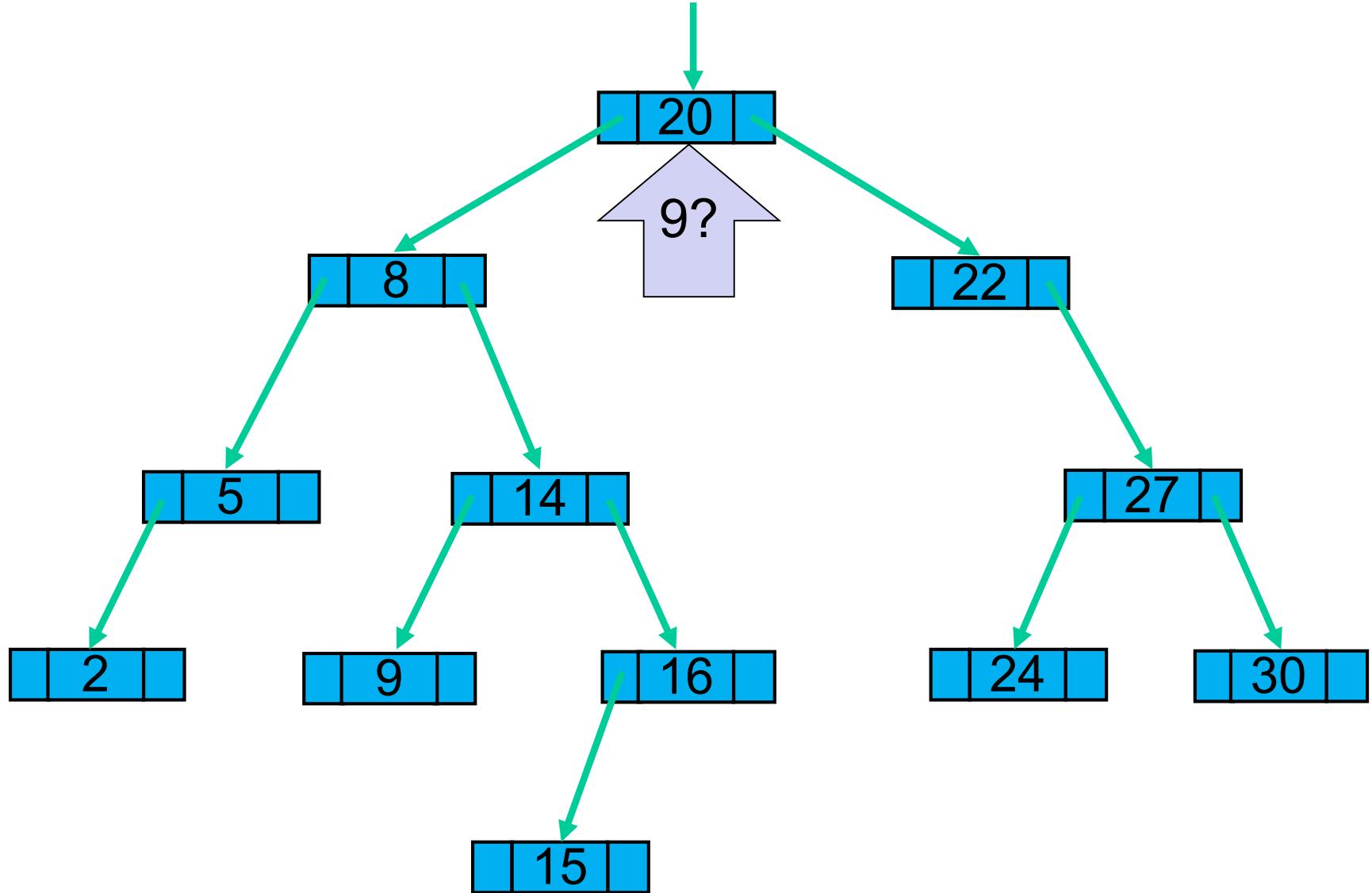


root

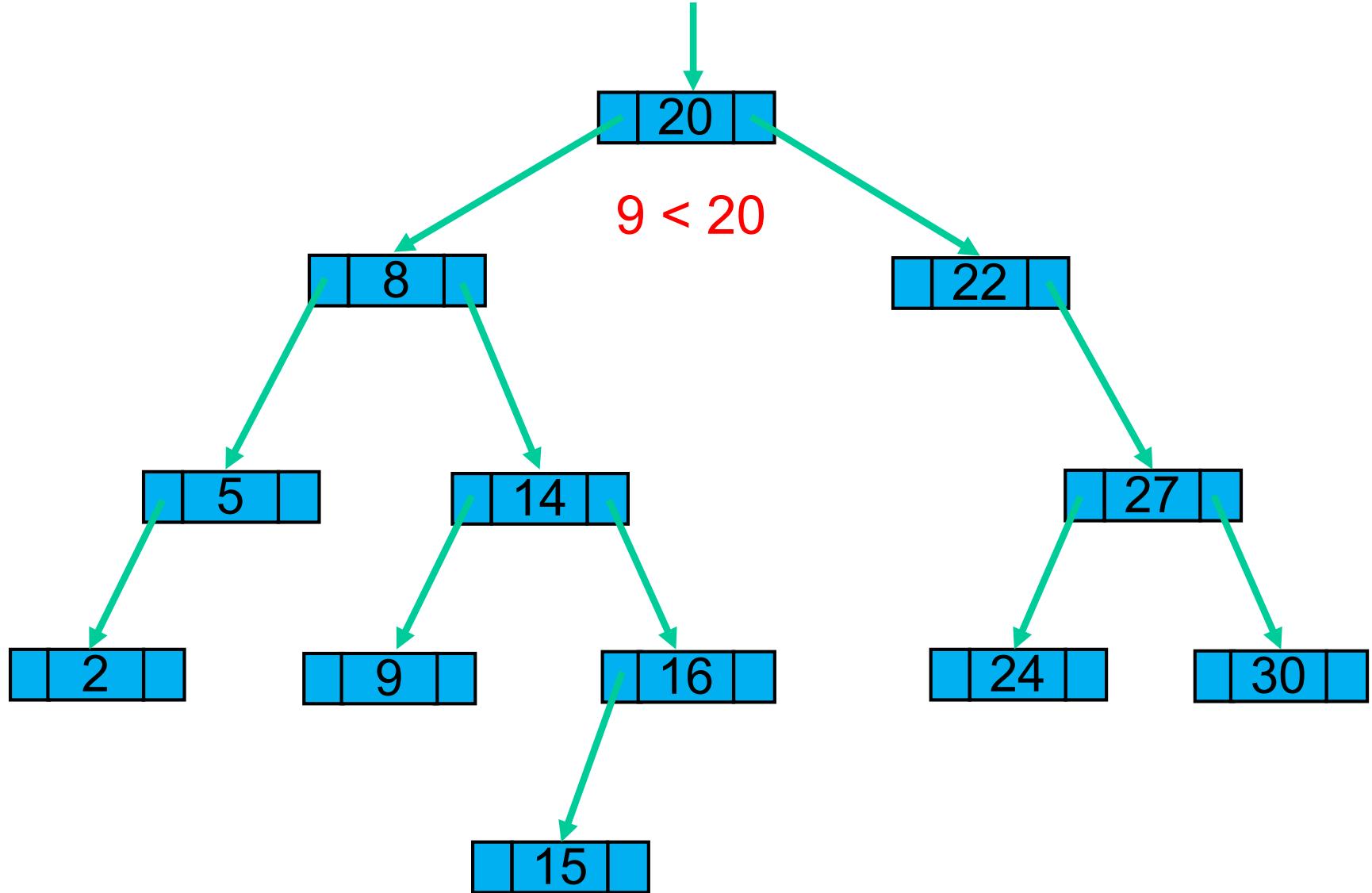
9



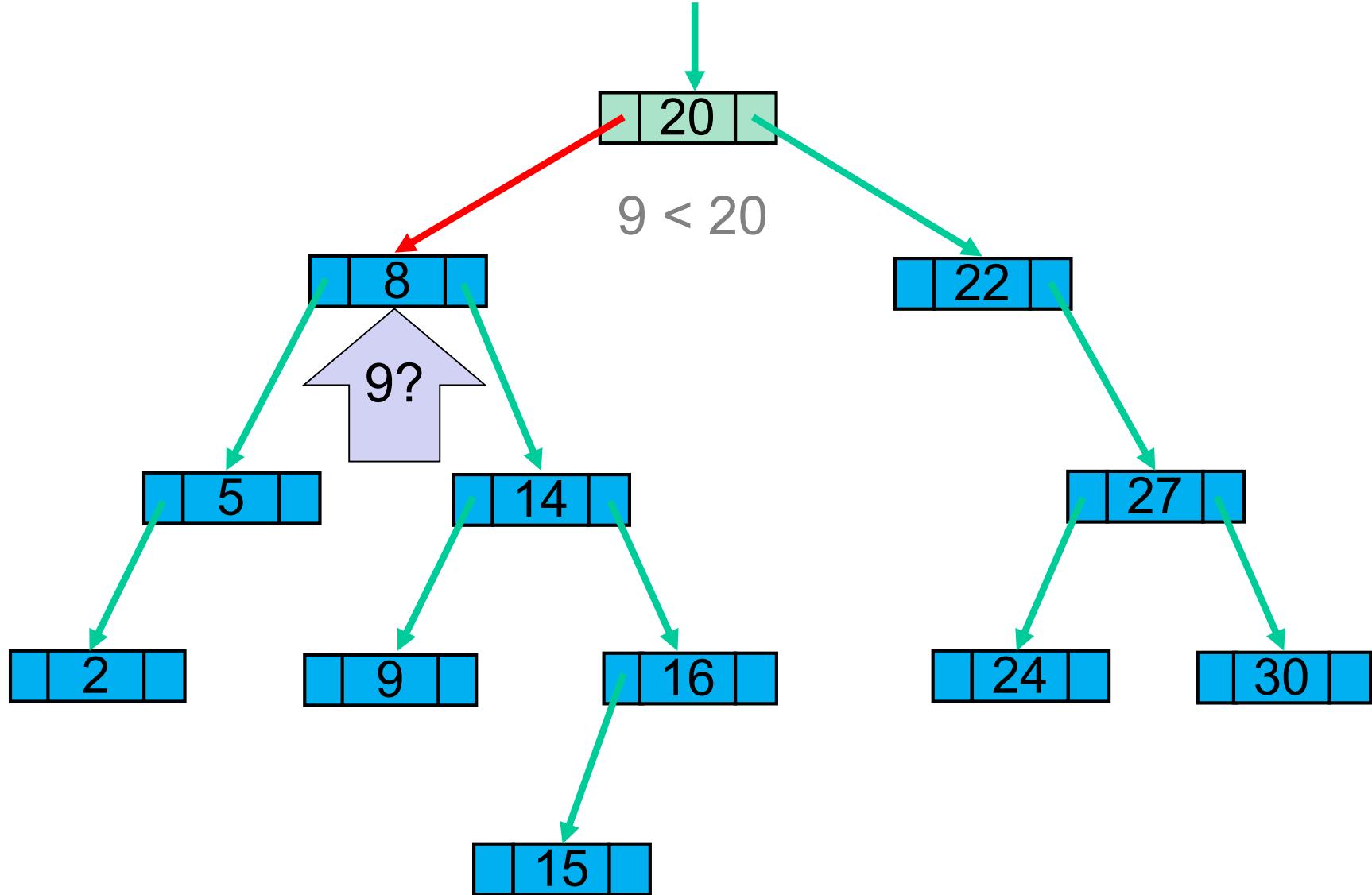
root



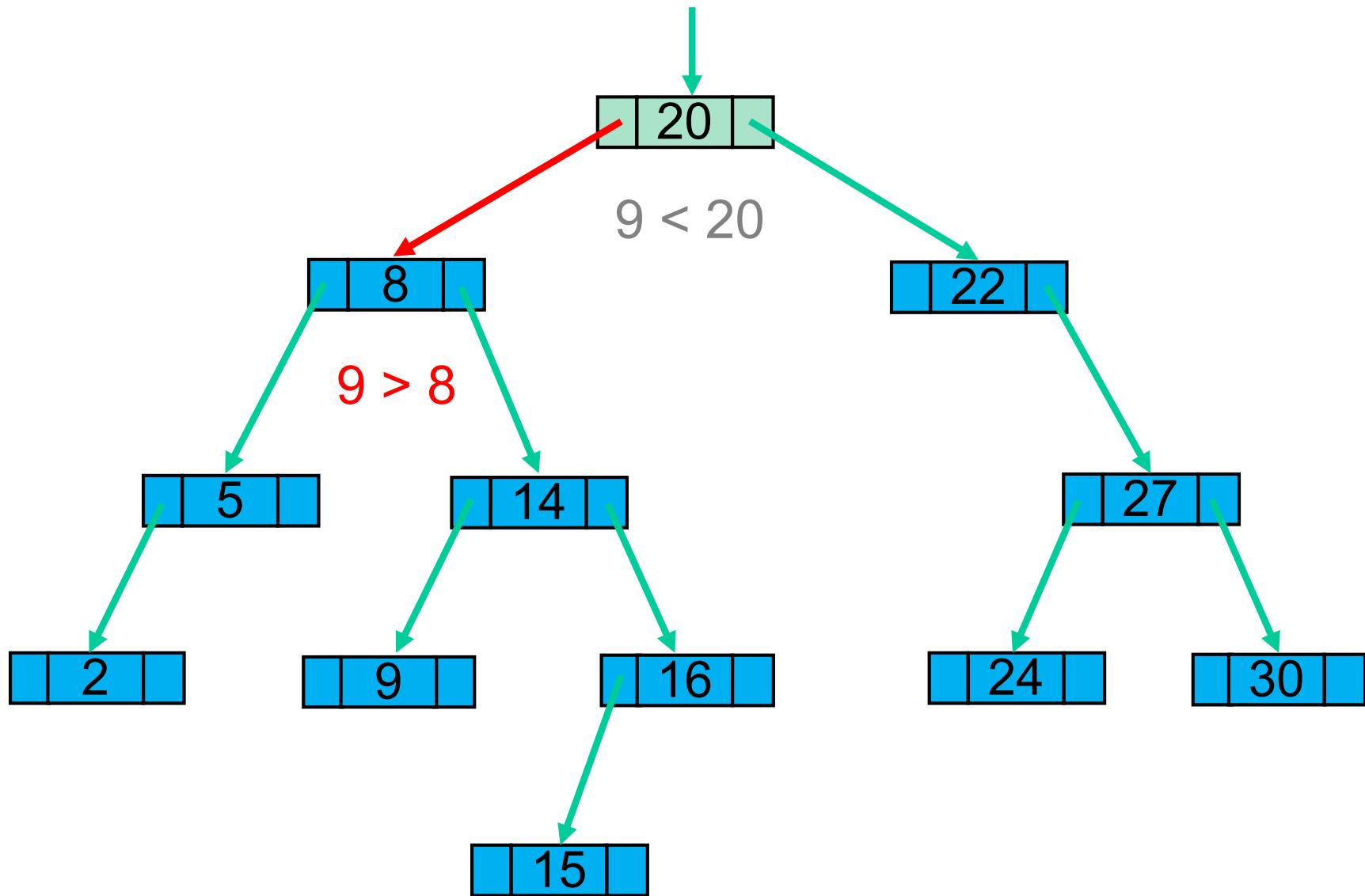
root



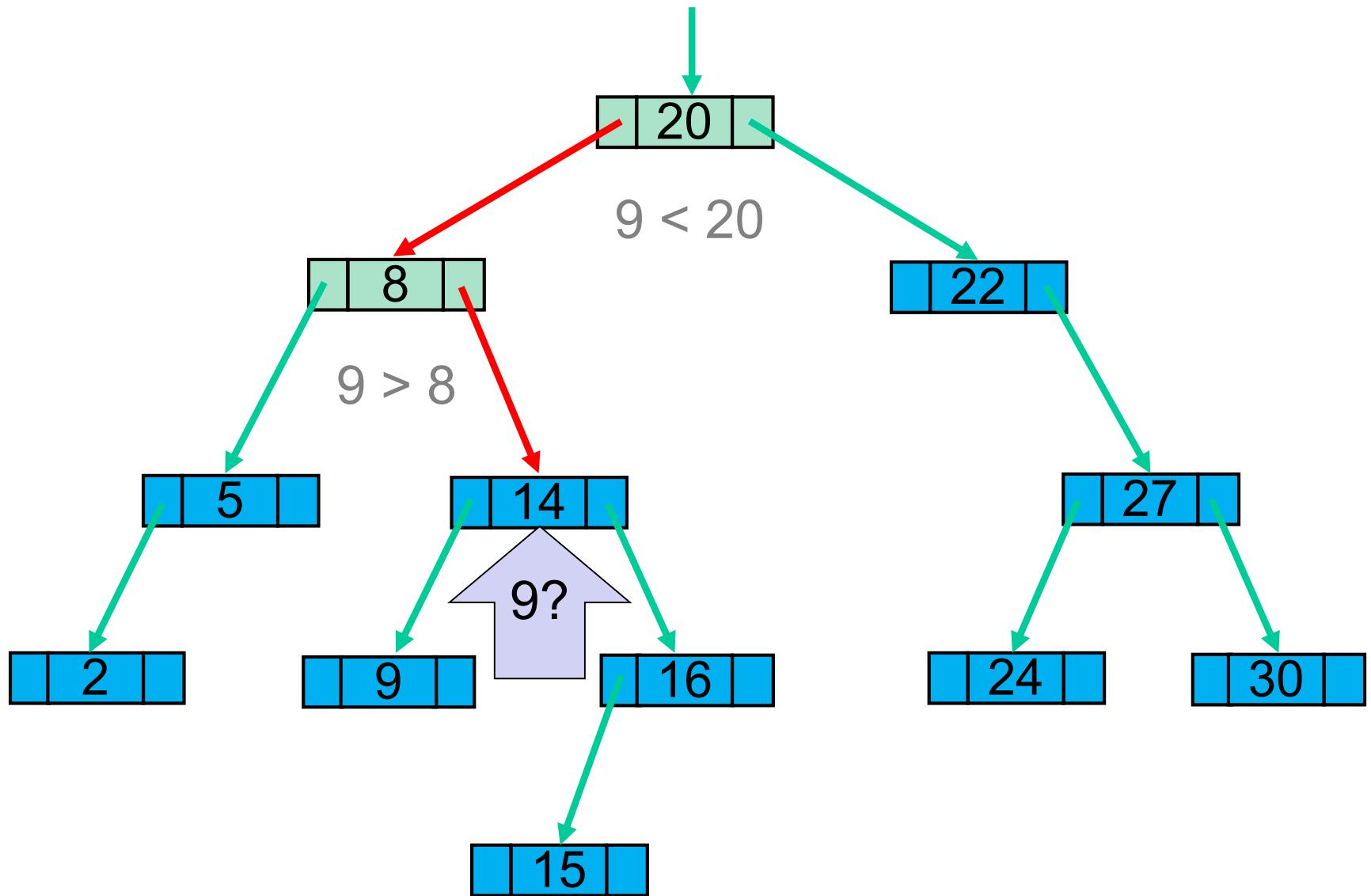
root



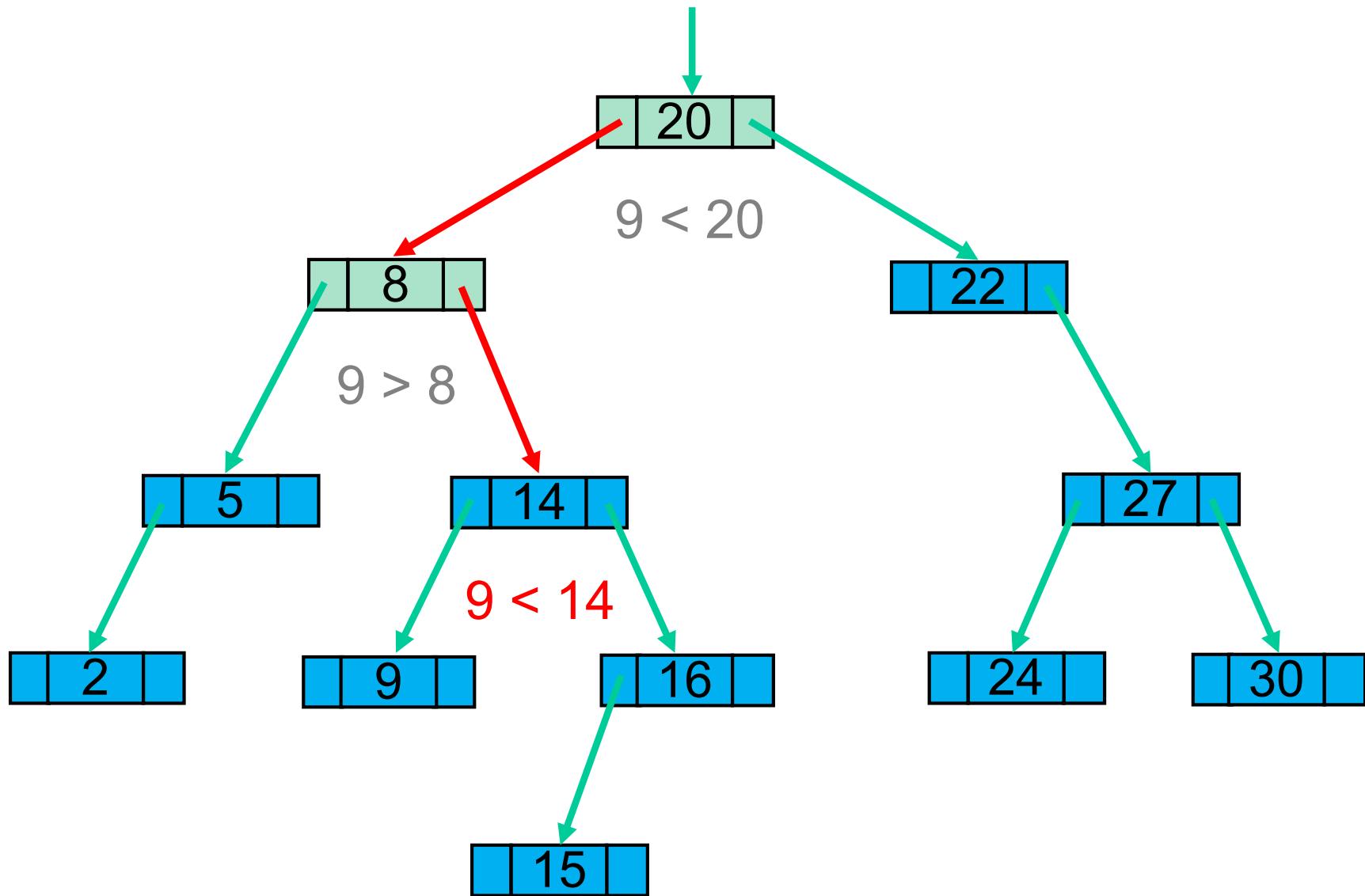
root



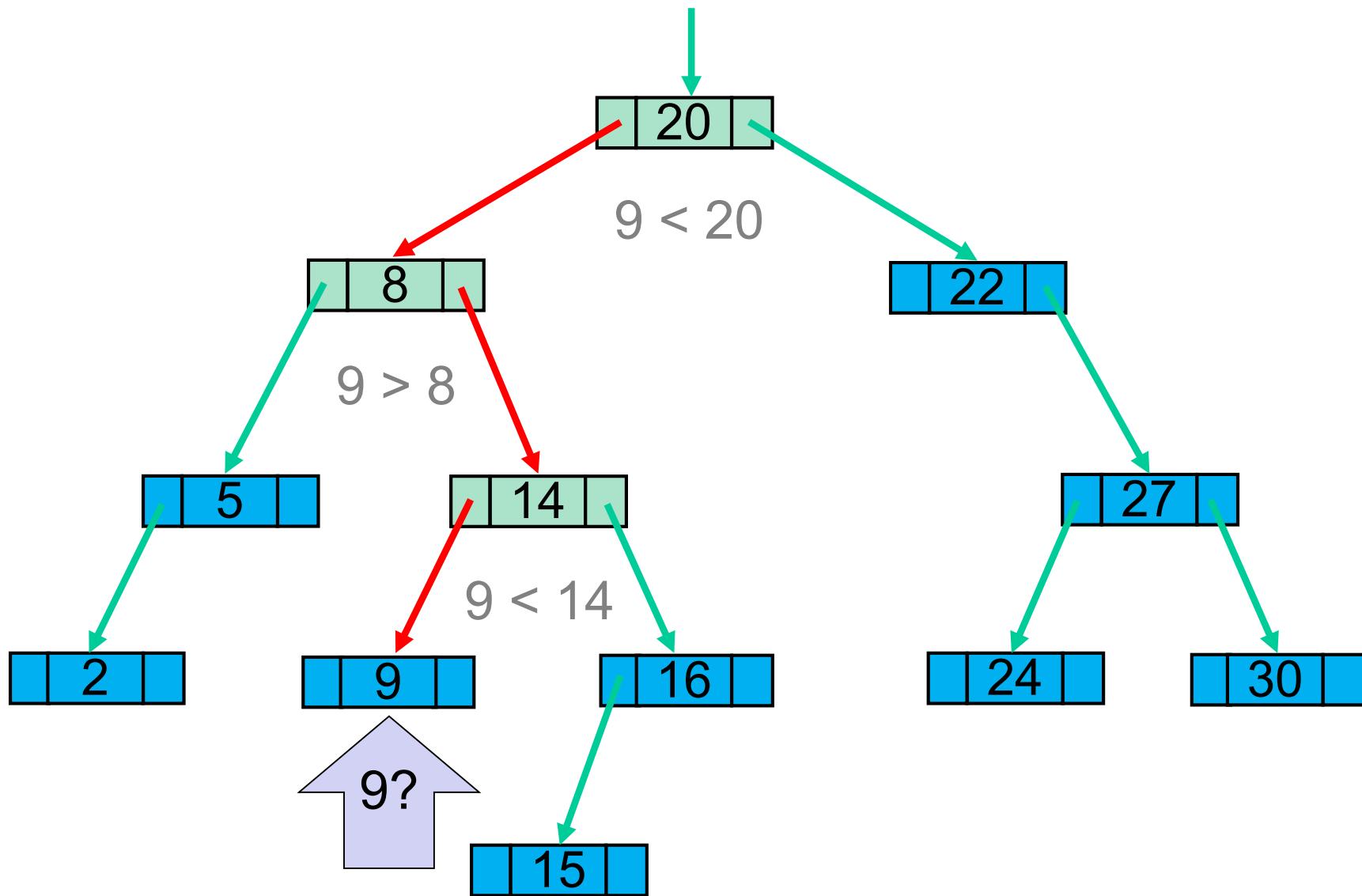
root



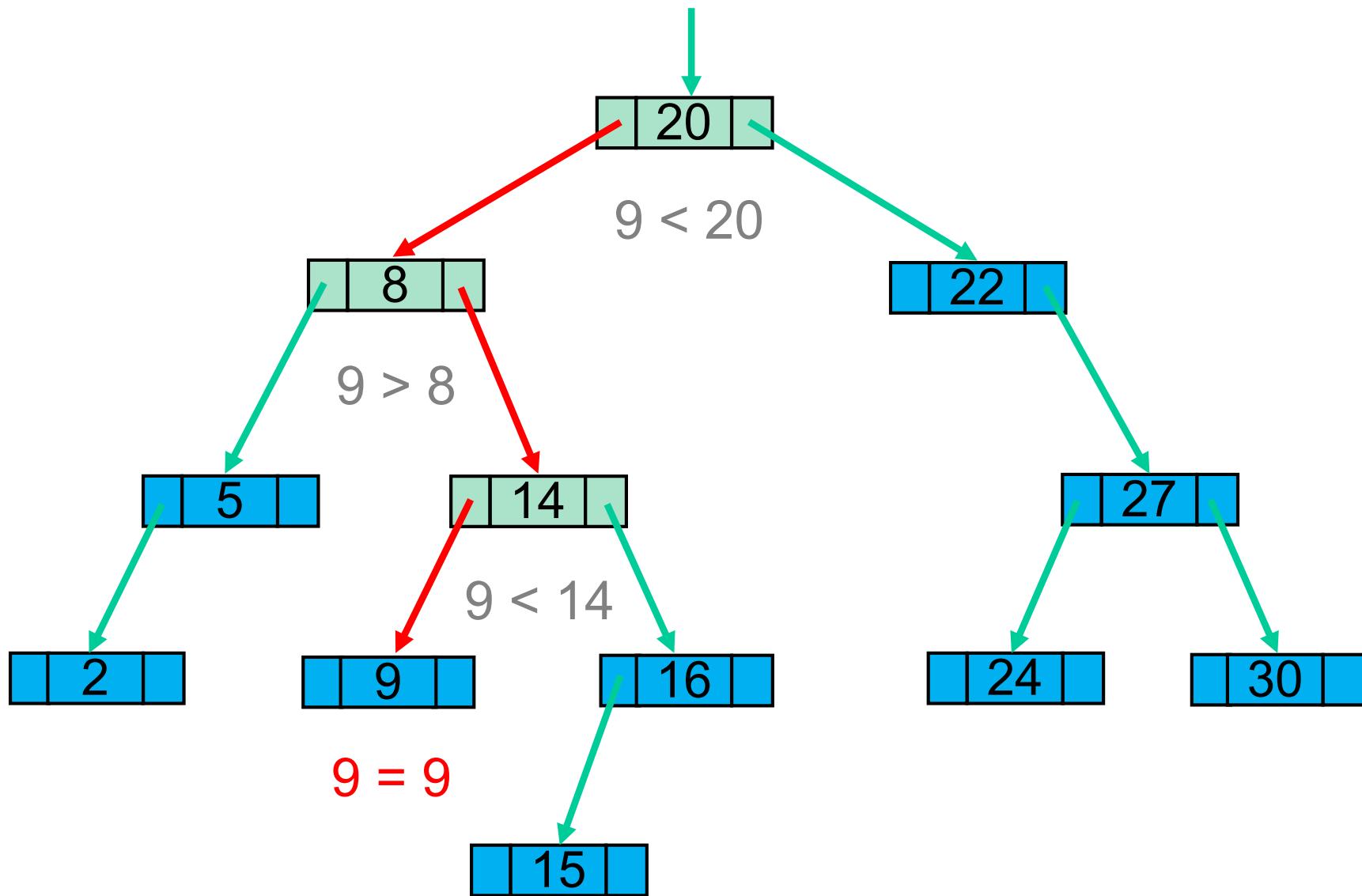
root



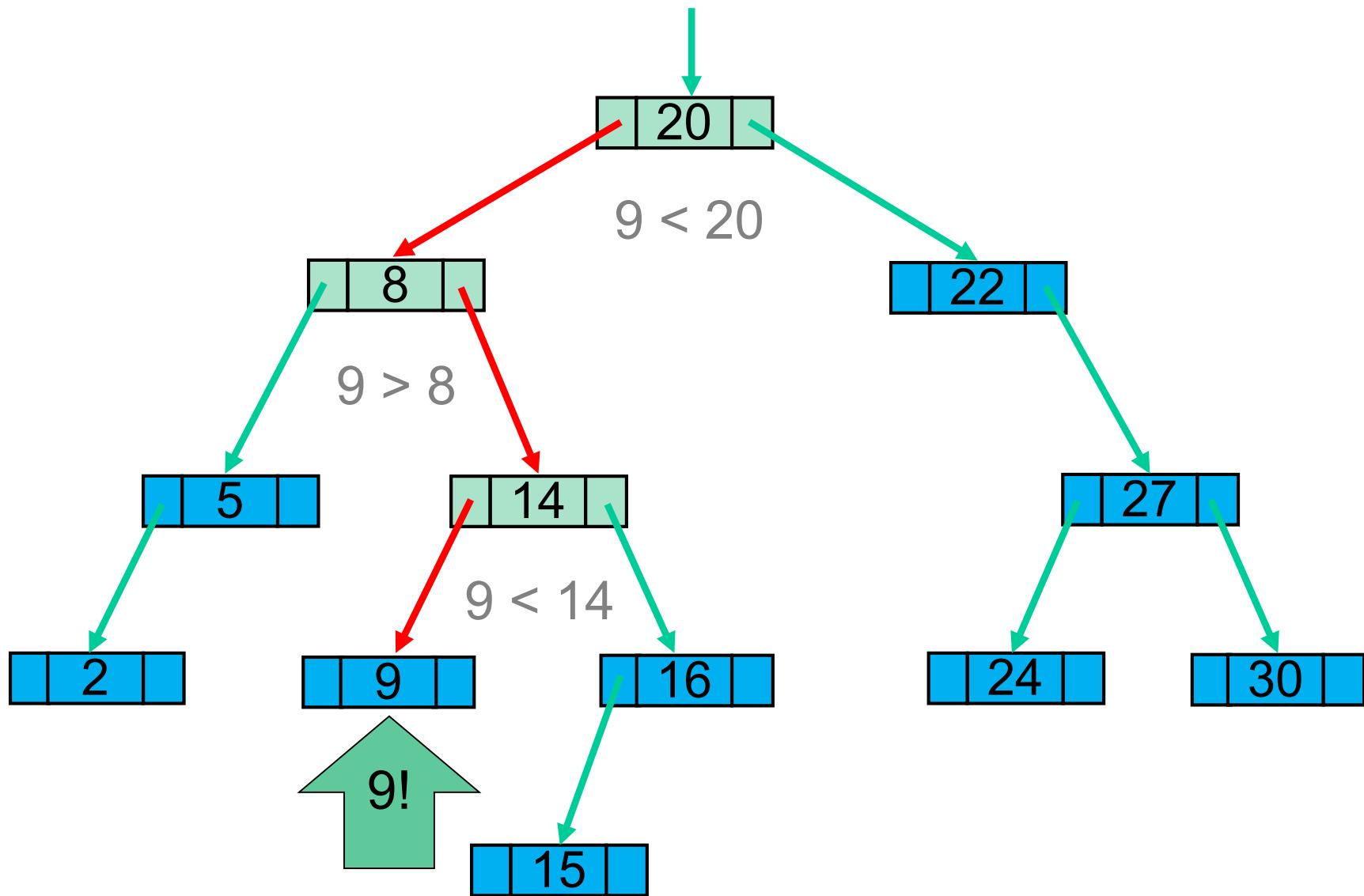
root



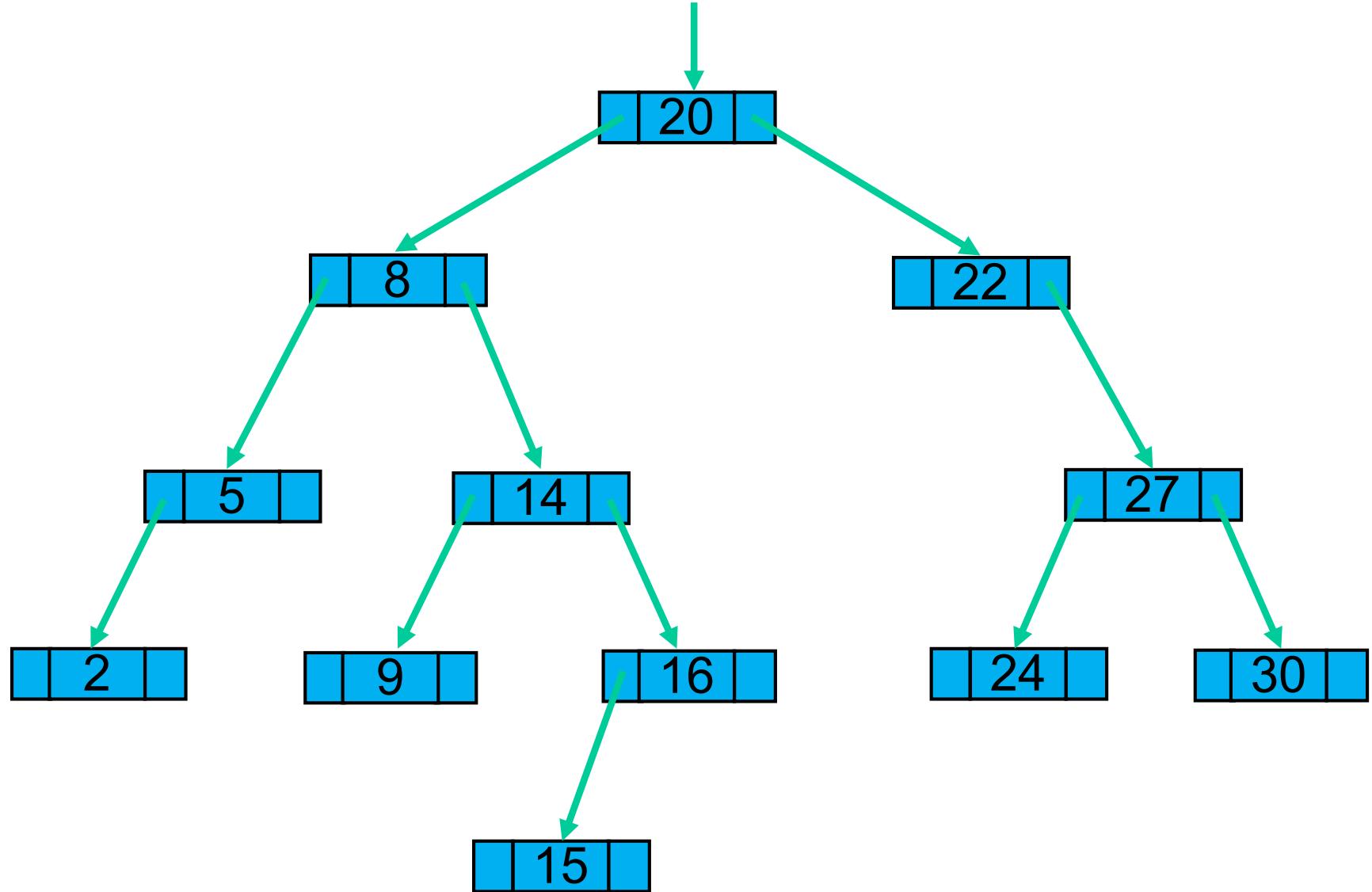
root



root

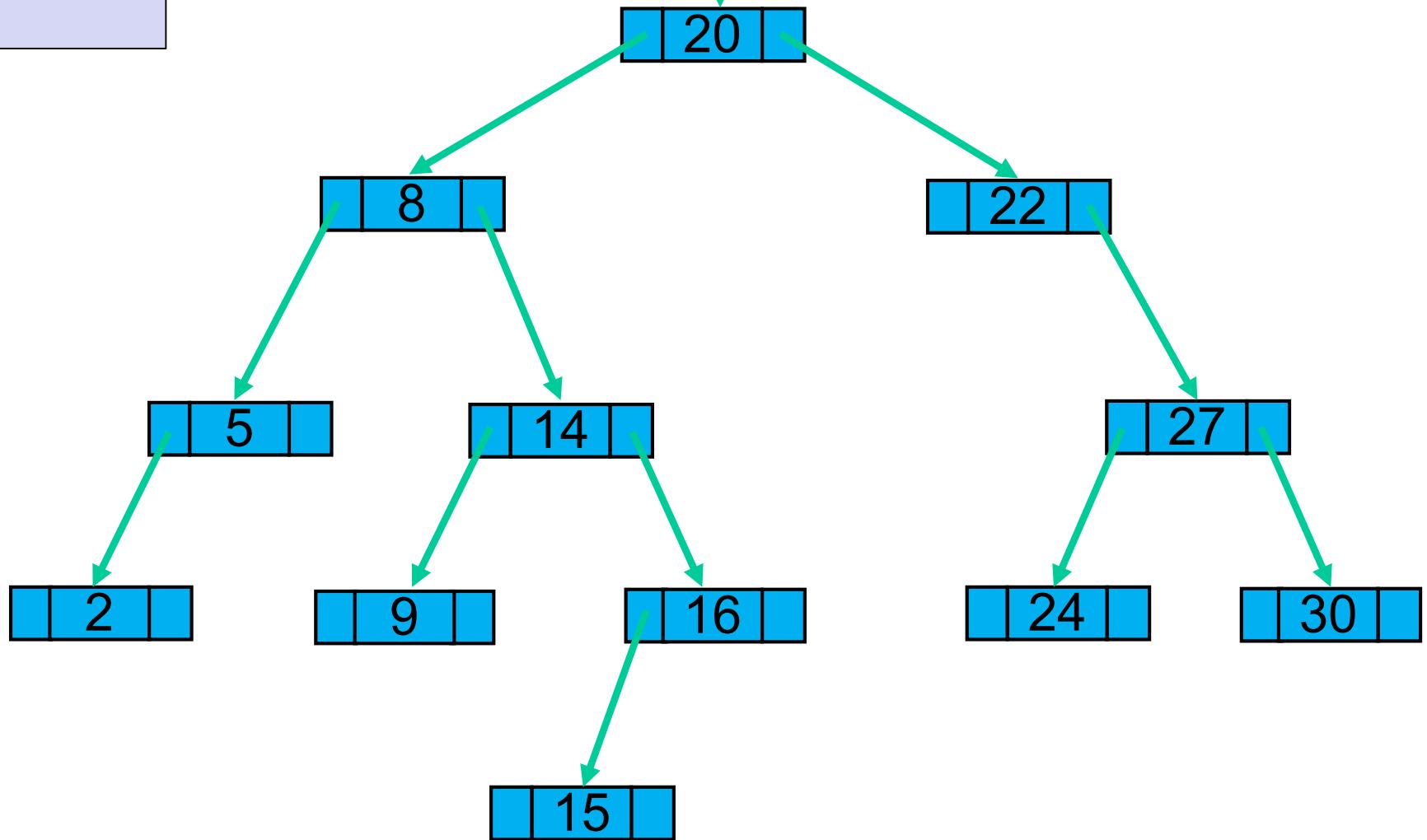


root

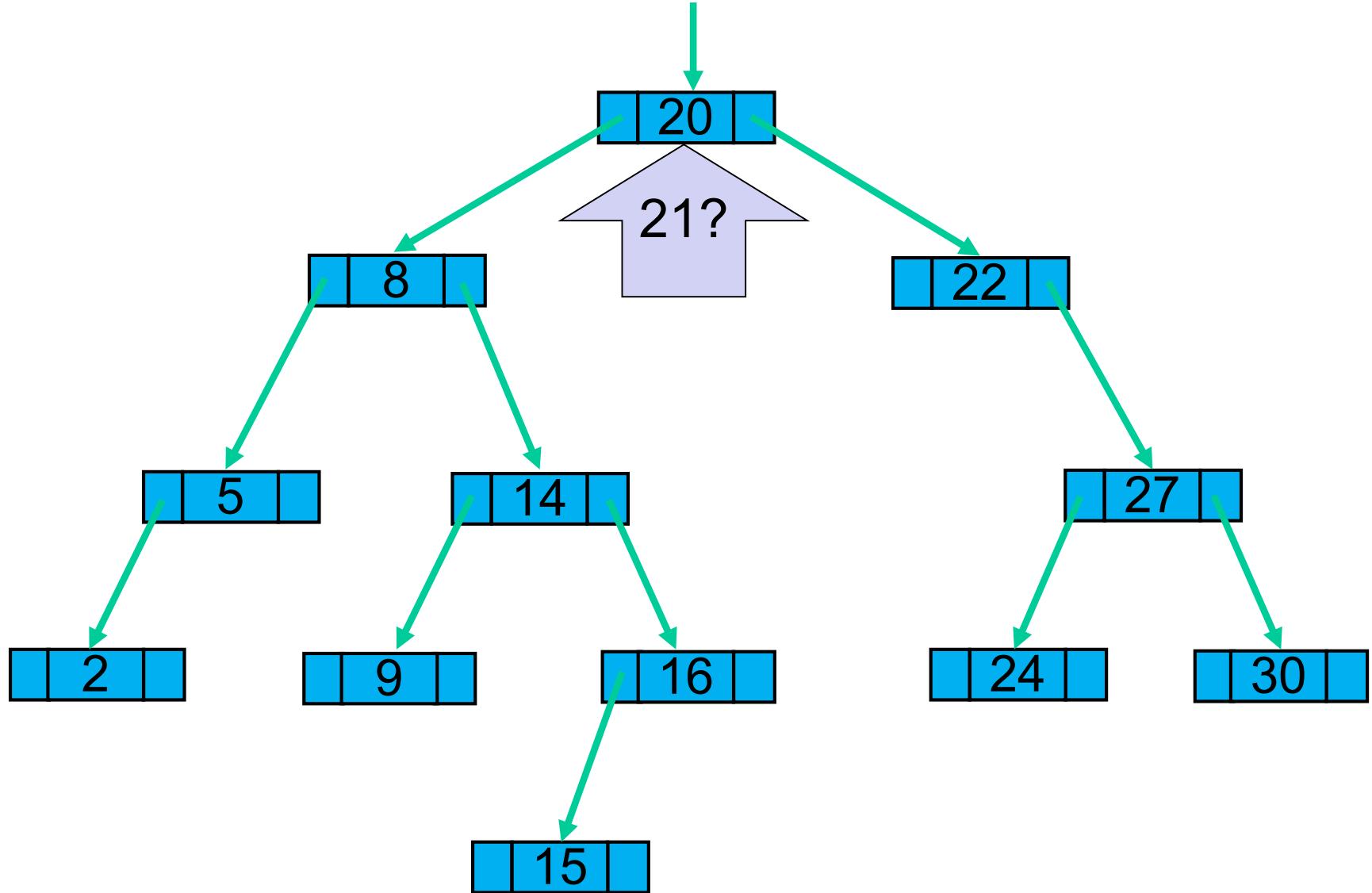


root

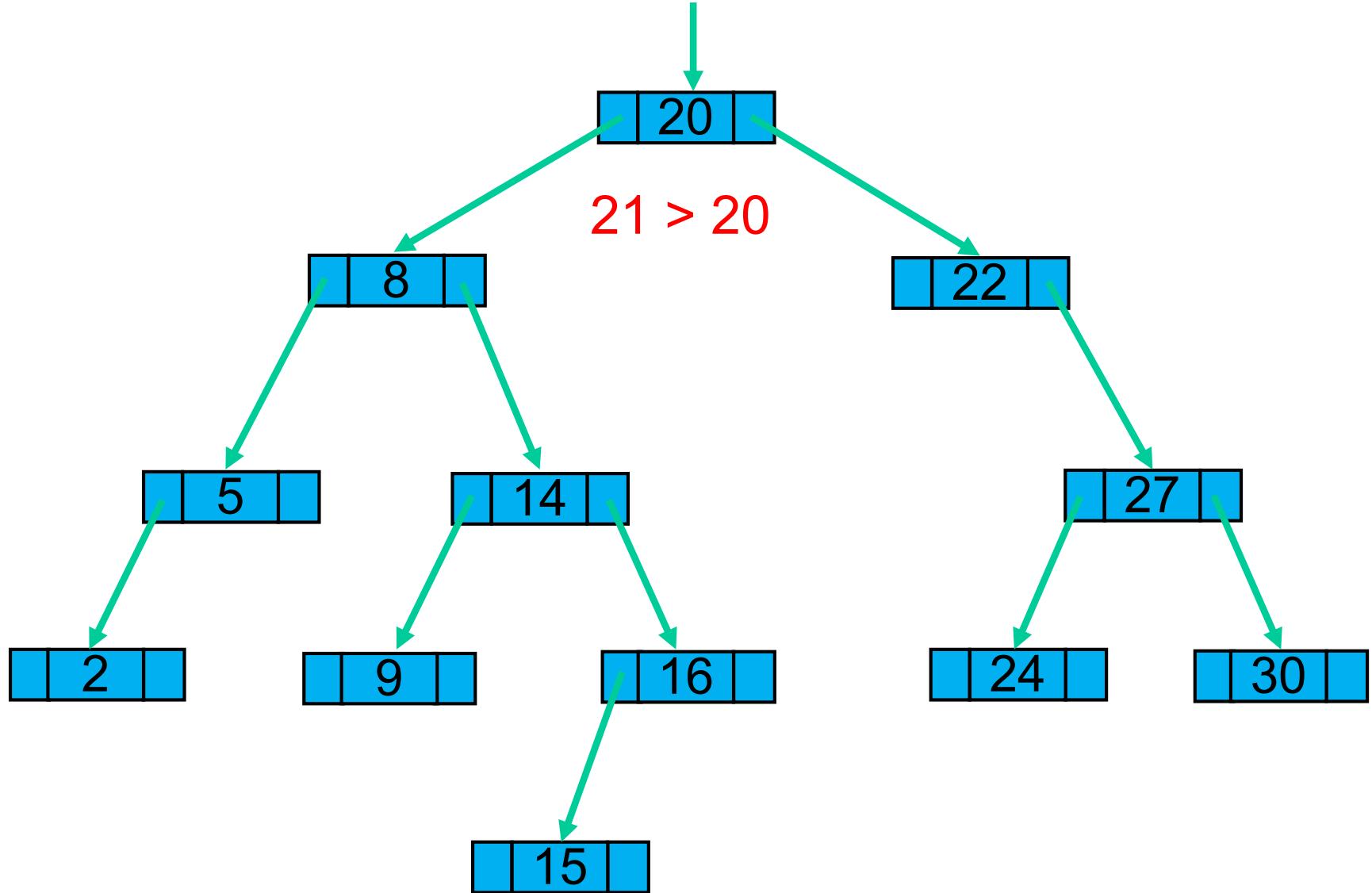
21



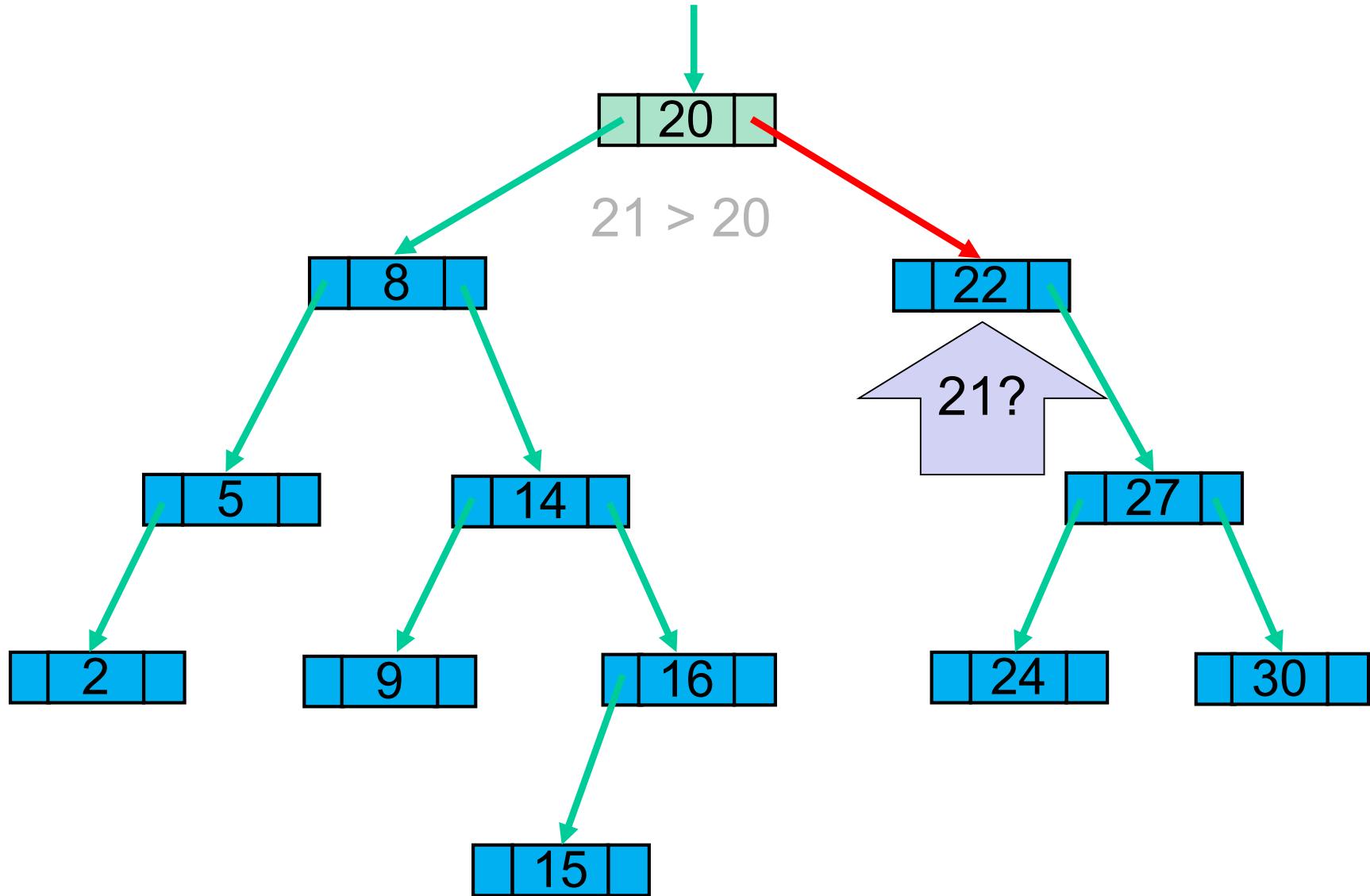
root



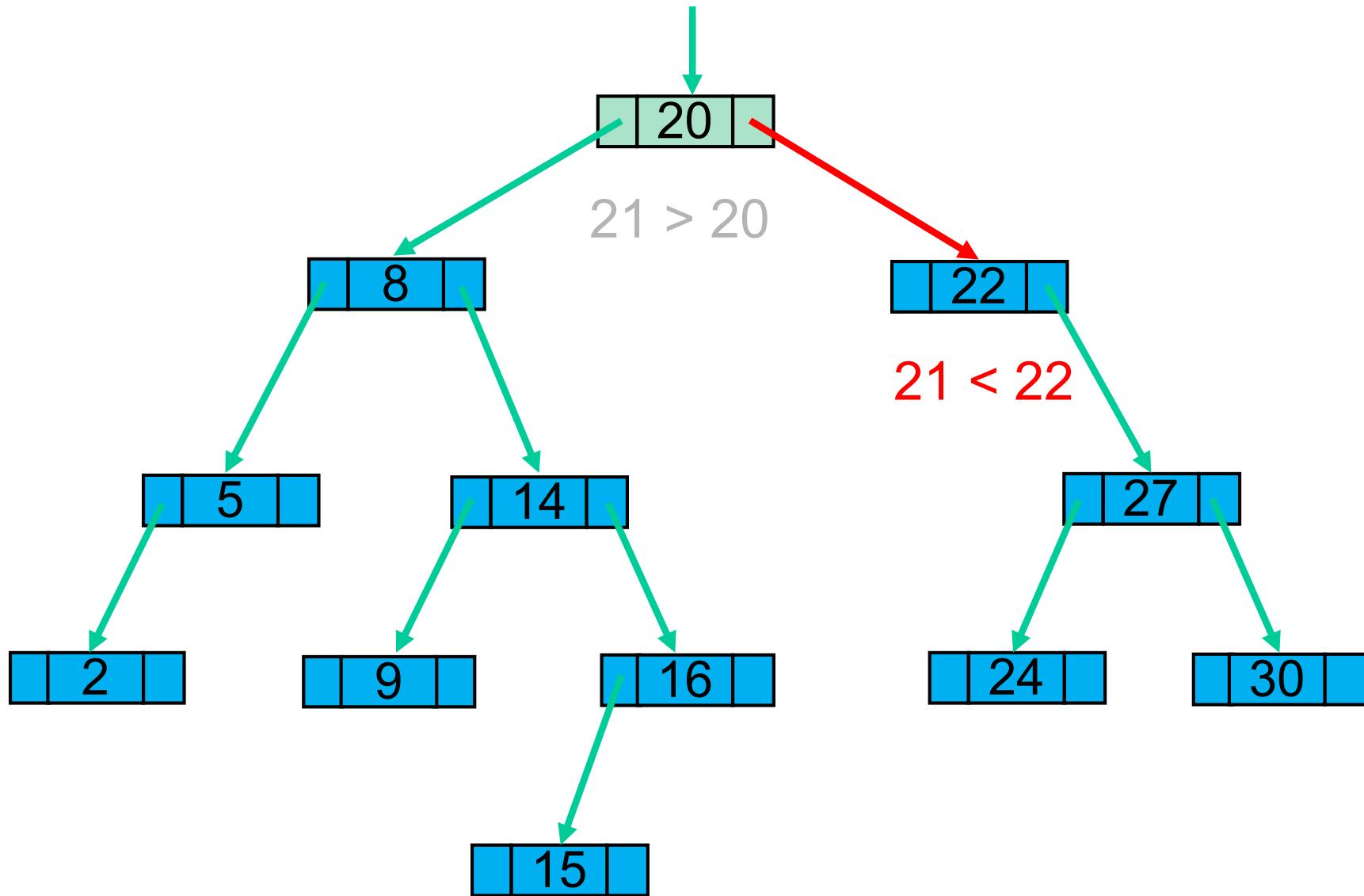
root



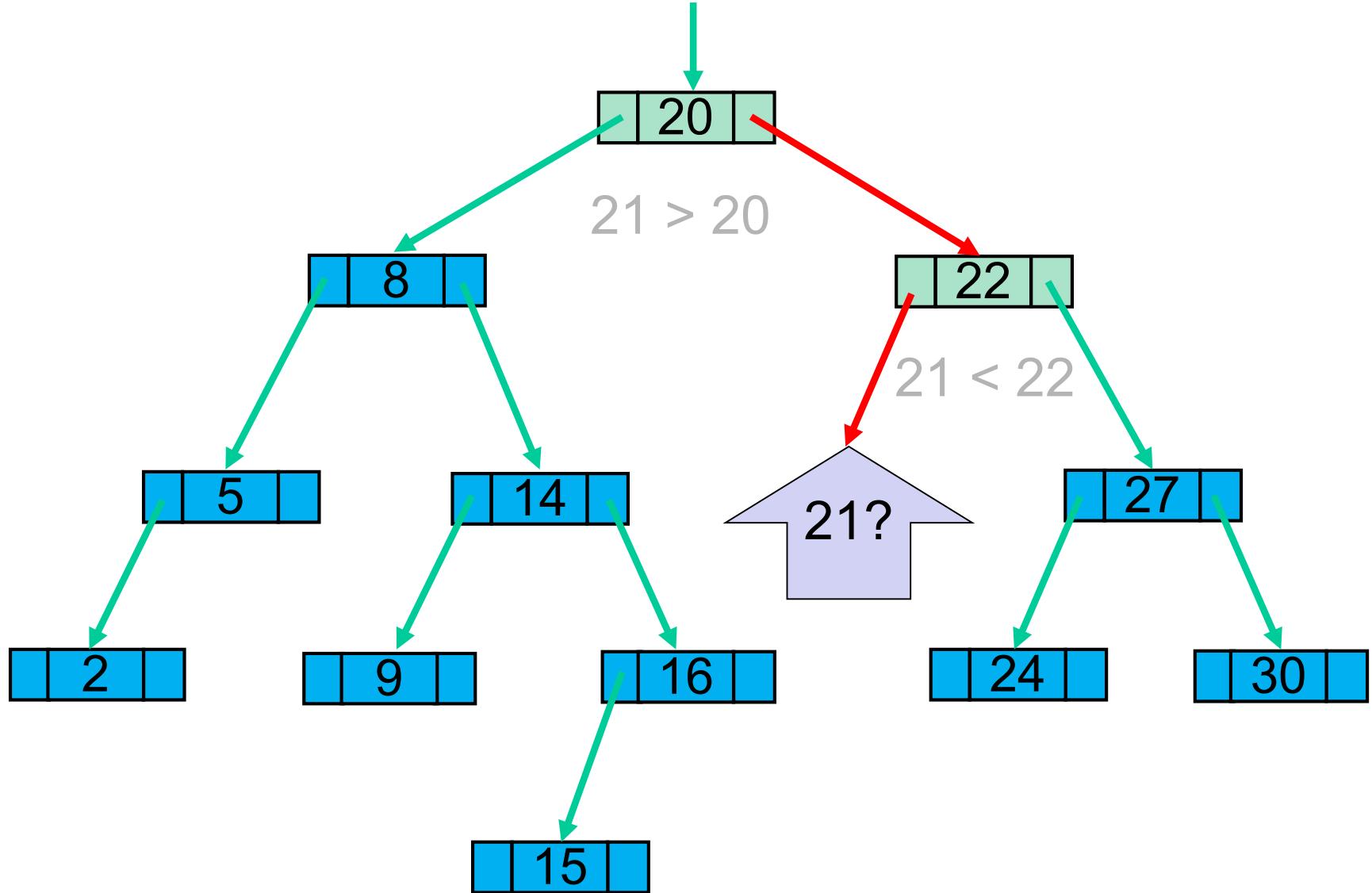
root



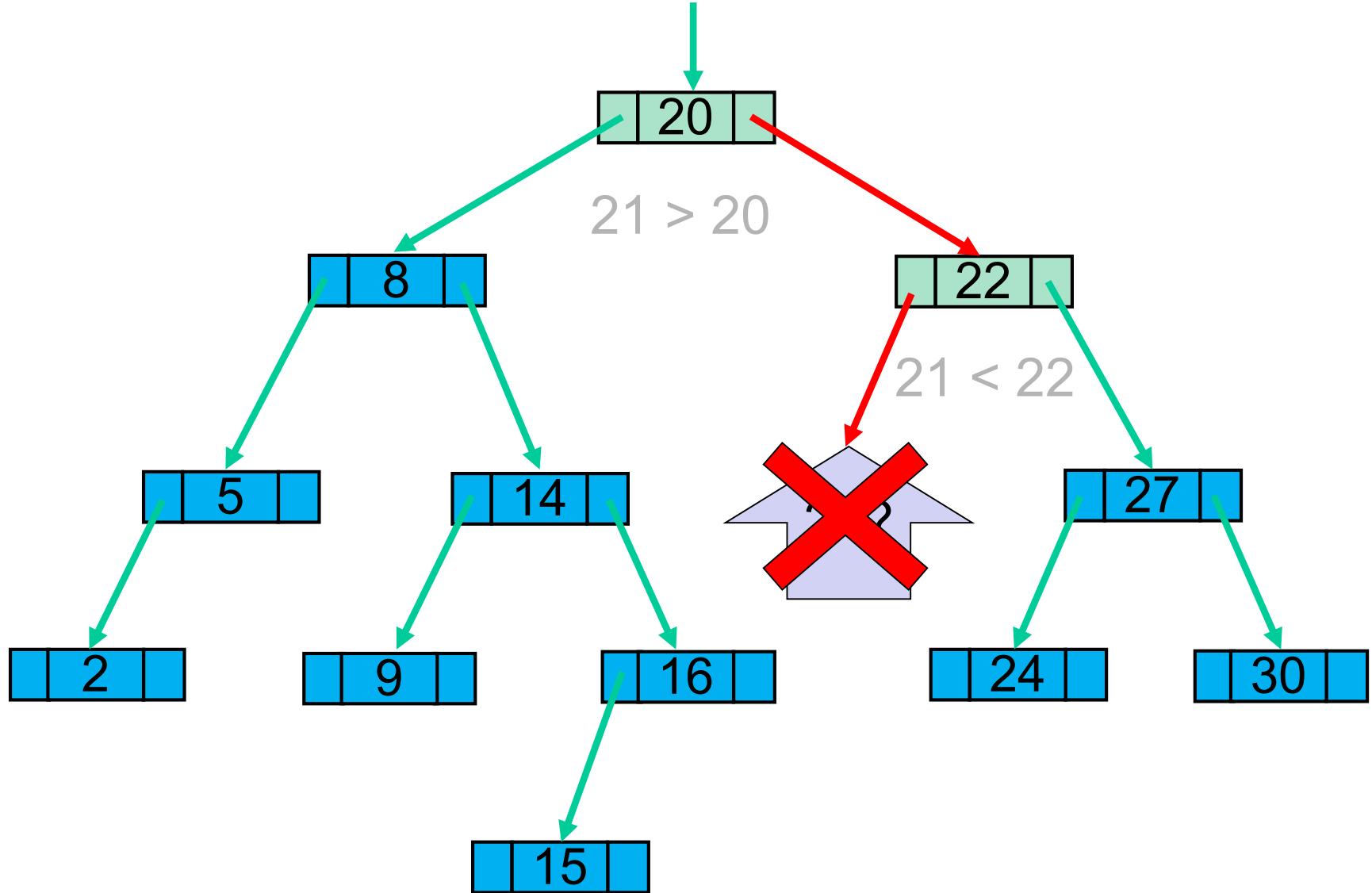
root



root



root



# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

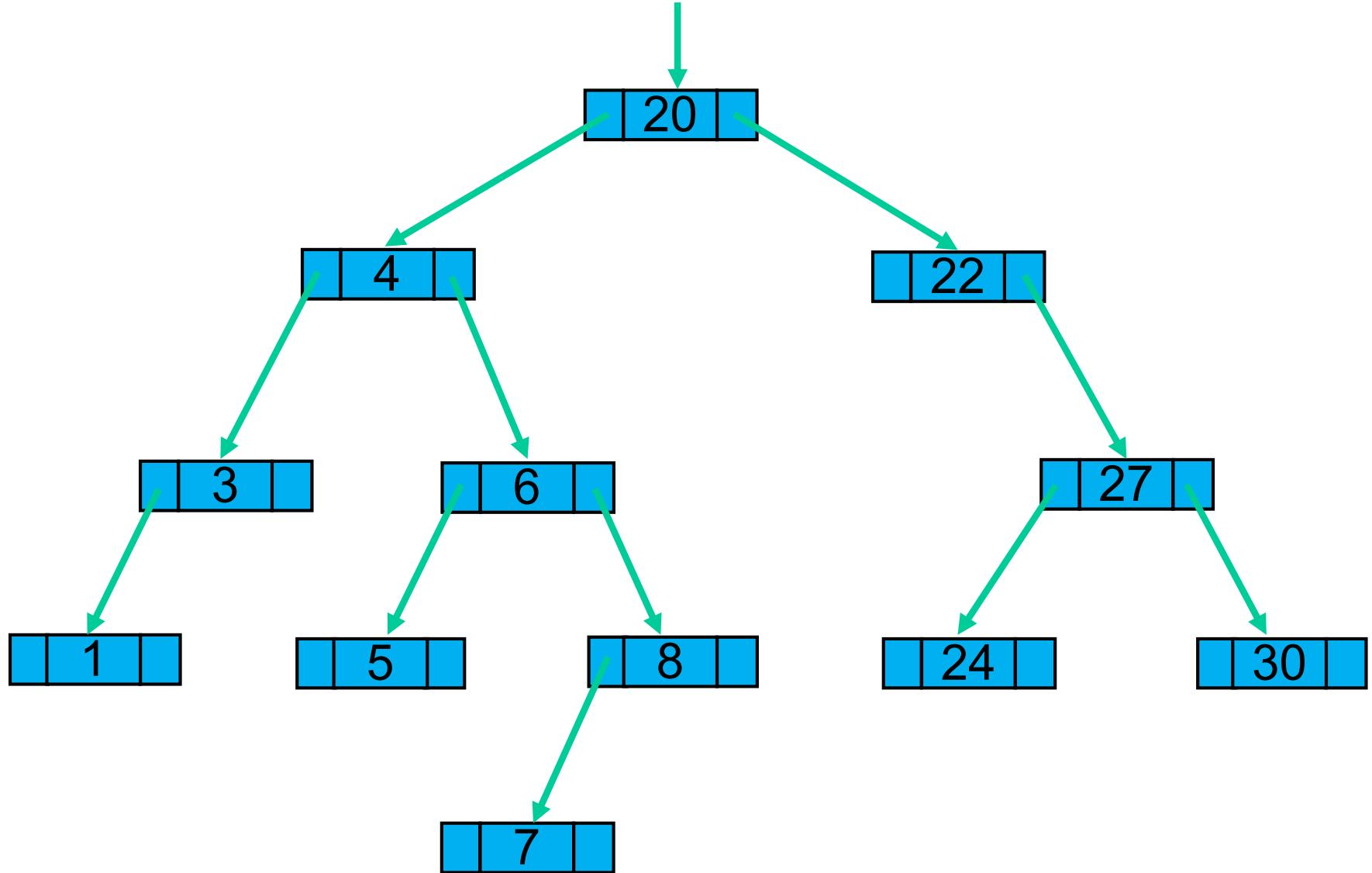
# BinaryTreeSearch

---

```
public boolean binaryTreeSearch(Node n, int val) {  
    if (n == null) return false;  
  
    if (n.value == val) {  
        return true;  
    } else if (n.value > val) {  
        return binaryTreeSearch(n.leftChild, val);  
    } else {  
        return binaryTreeSearch(n.rightChild, val);  
    }  
}
```

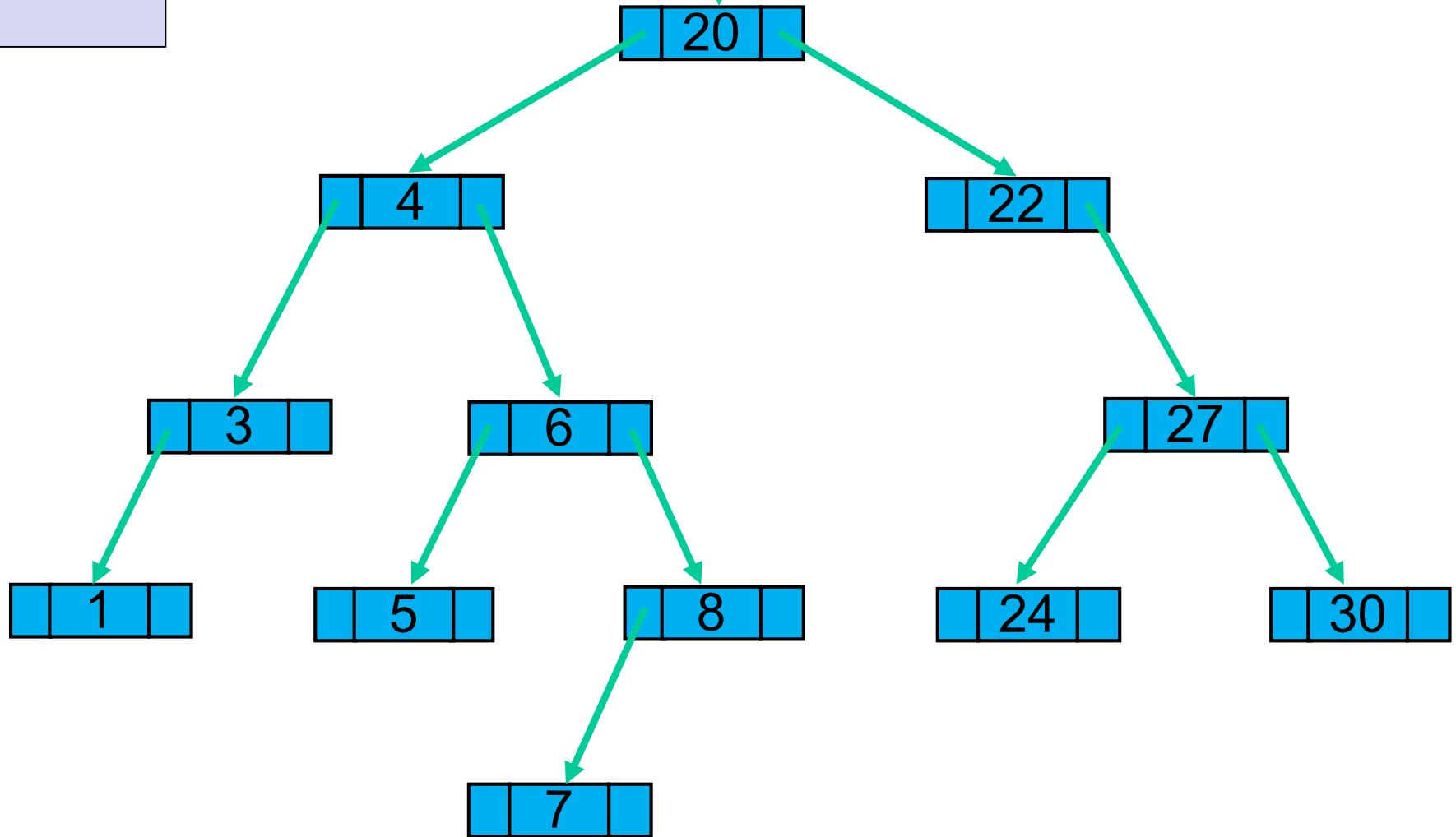
# How do we add a value to a binary search tree?

root

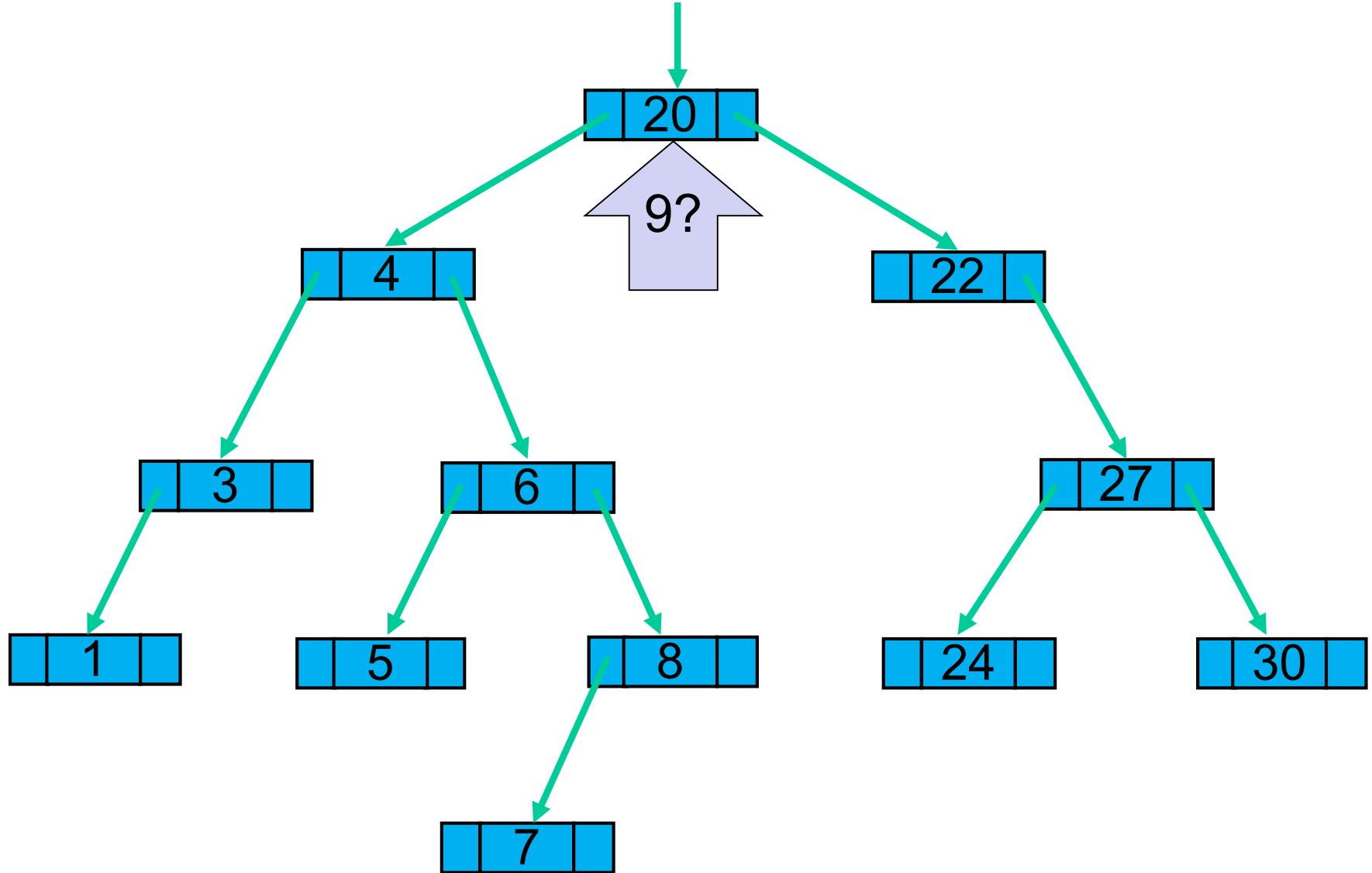


root

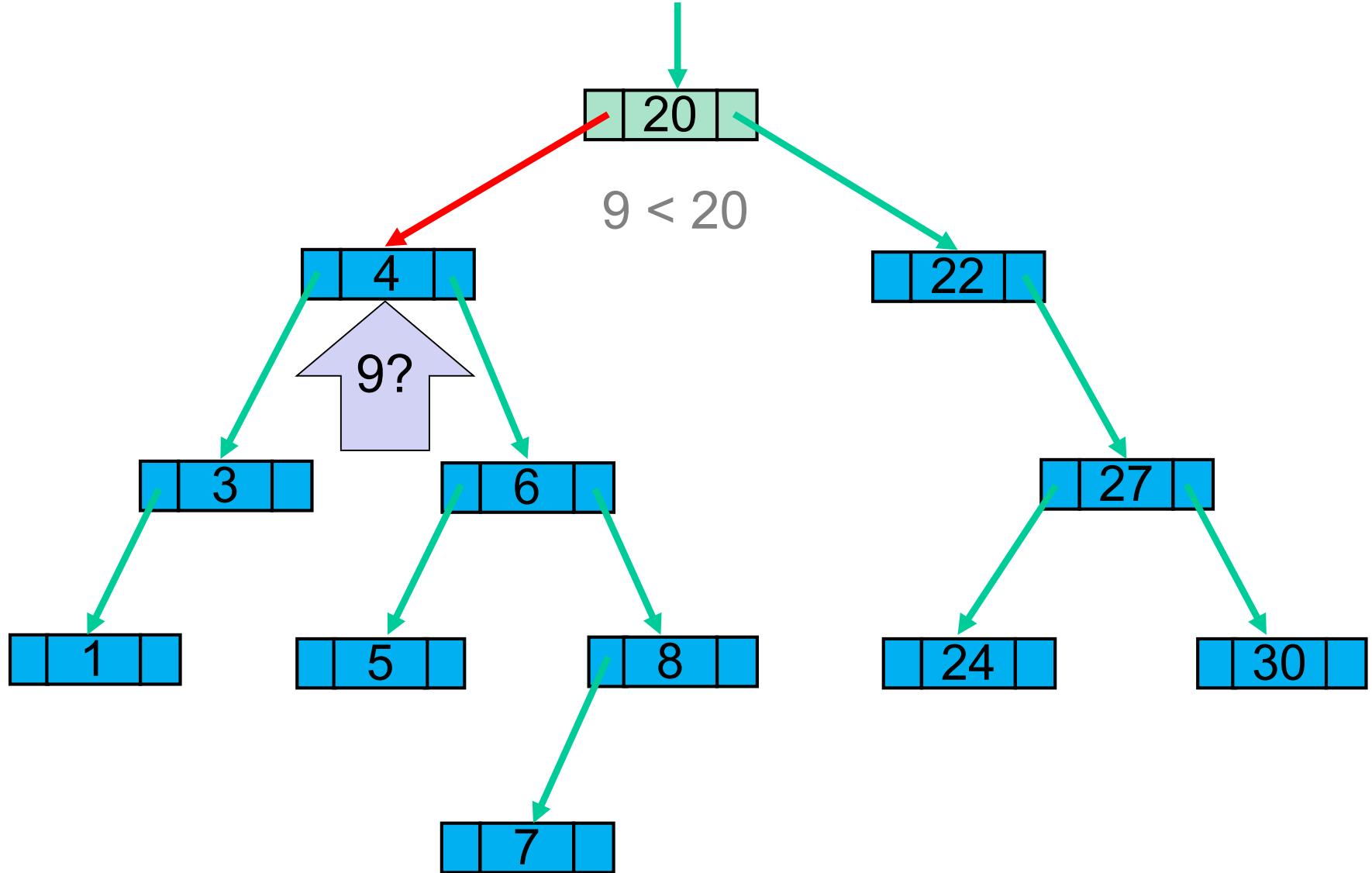
9



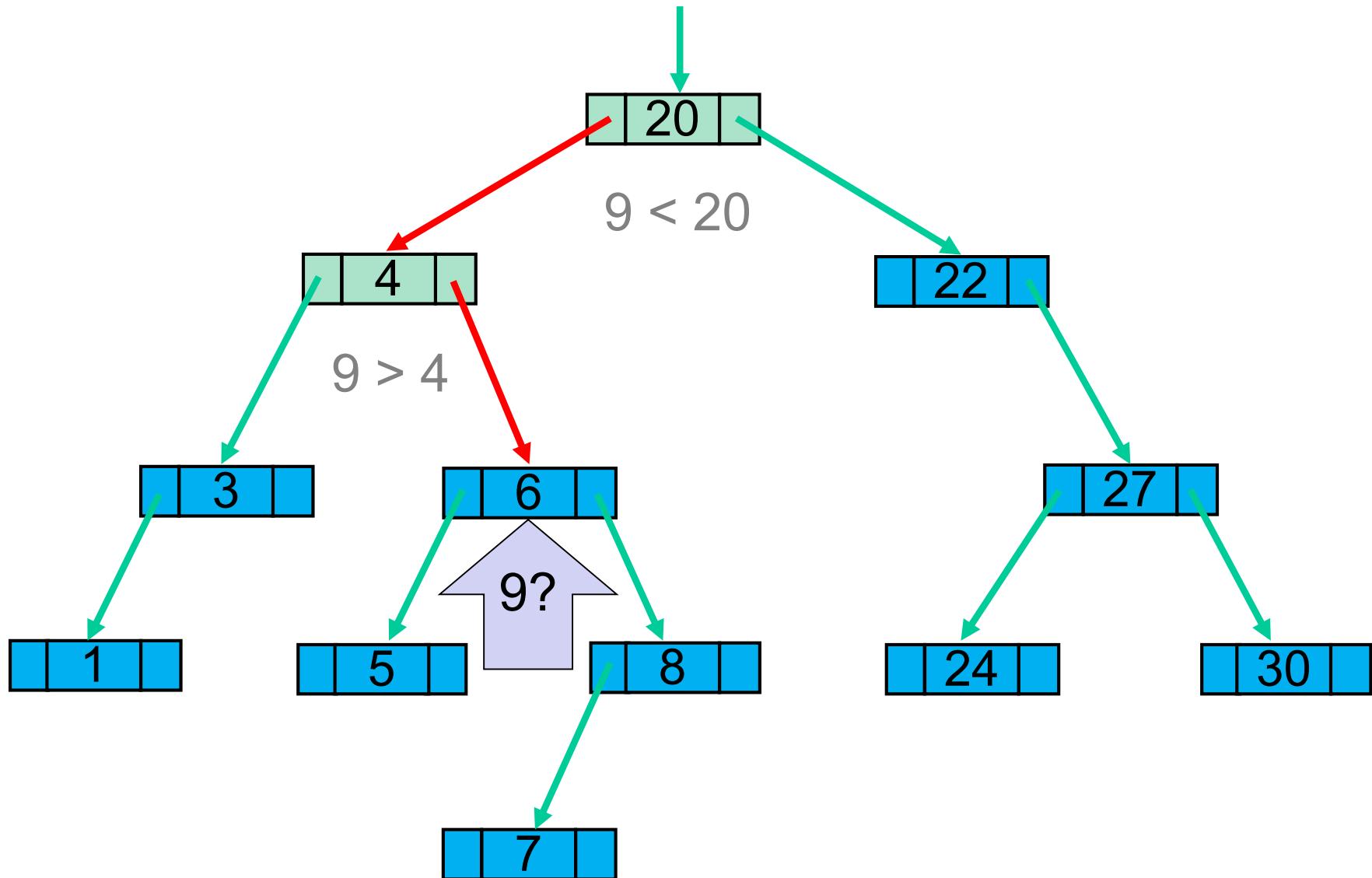
root



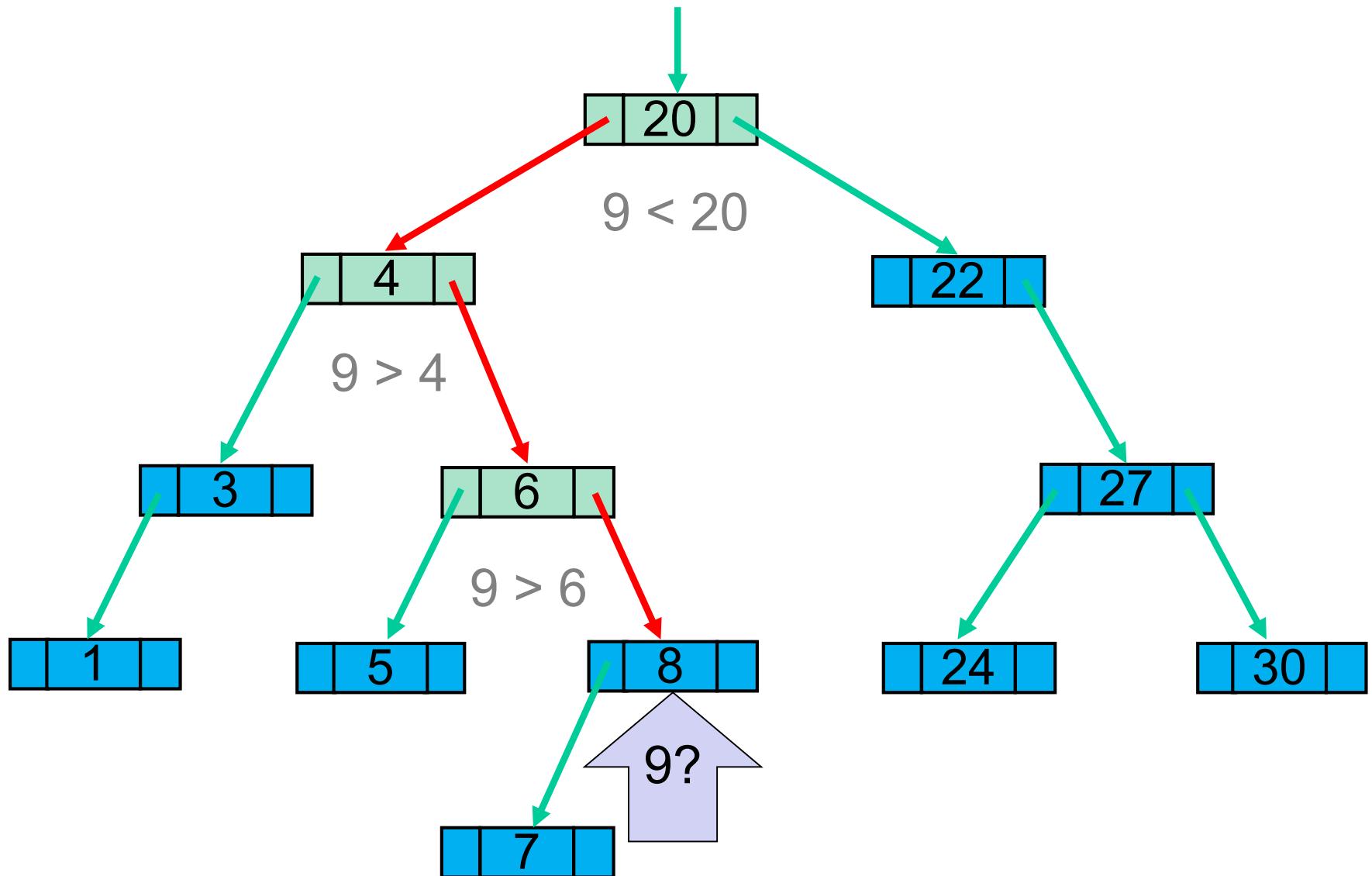
root



root



root



root

20

$9 < 20$

4

$9 > 4$

22

3

1

6

$9 > 6$

5

8

$9 > 8$

7

27

24

30

root

20

$9 < 20$

4

$9 > 4$

22

3

1

6

$9 > 6$

5

27

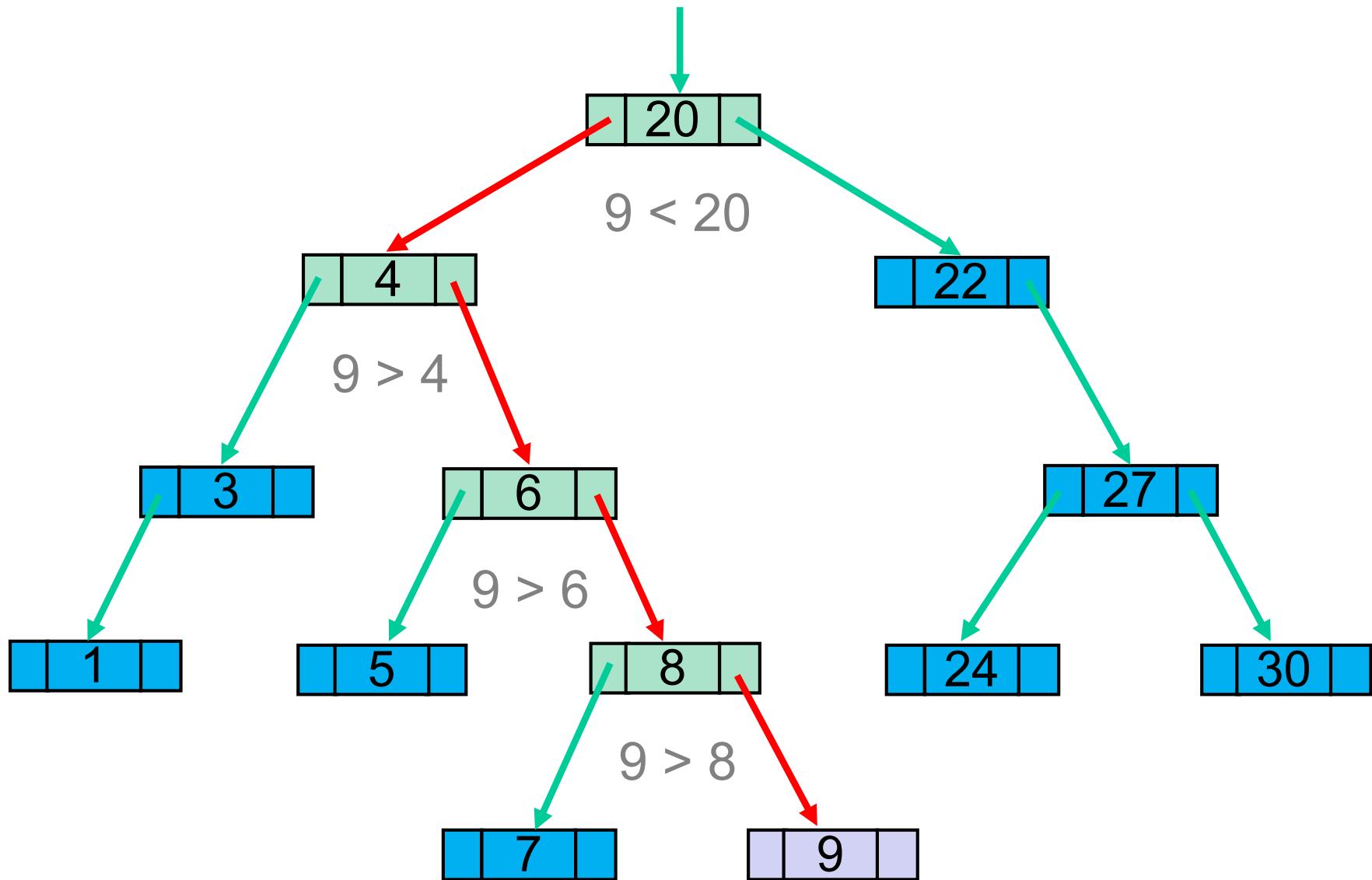
30

$9 > 8$

7

null

root



# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Binary Tree Add

---

```
public boolean add(Node n, int val) {  
    if (n.value == val) {  
        return false;  
    } else if (n.value > val) {  
        if (n.leftChild == null) {  
            n.leftChild = new Node(val);  
            return true;  
        } else {  
            return add(n.leftChild, val);  
        }  
    } else {  
        if (n.rightChild == null) {  
            n.rightChild = new Node(val);  
            return true;  
        } else {  
            return add(n.rightChild, val);  
        }  
    }  
}
```

# Recap: Binary Search Trees

---

- We can **search** for and **add** elements to a Binary Search Tree knowing that we need to keep the tree ordered
- For any node, smaller values are in its left subtree
- And greater values are in its right subtree

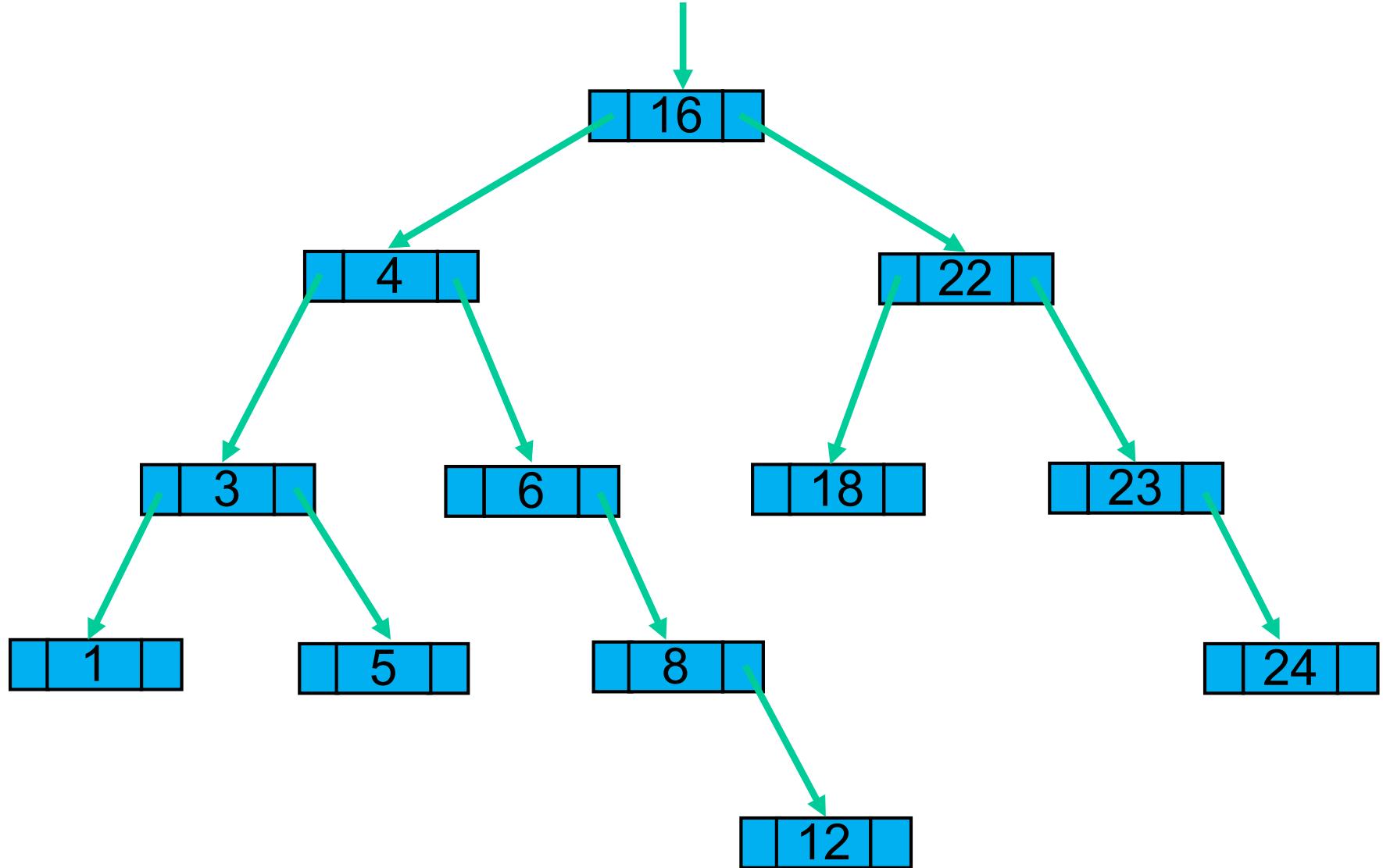
# **SD2x2.4**

# **Binary Search Trees – Remove**

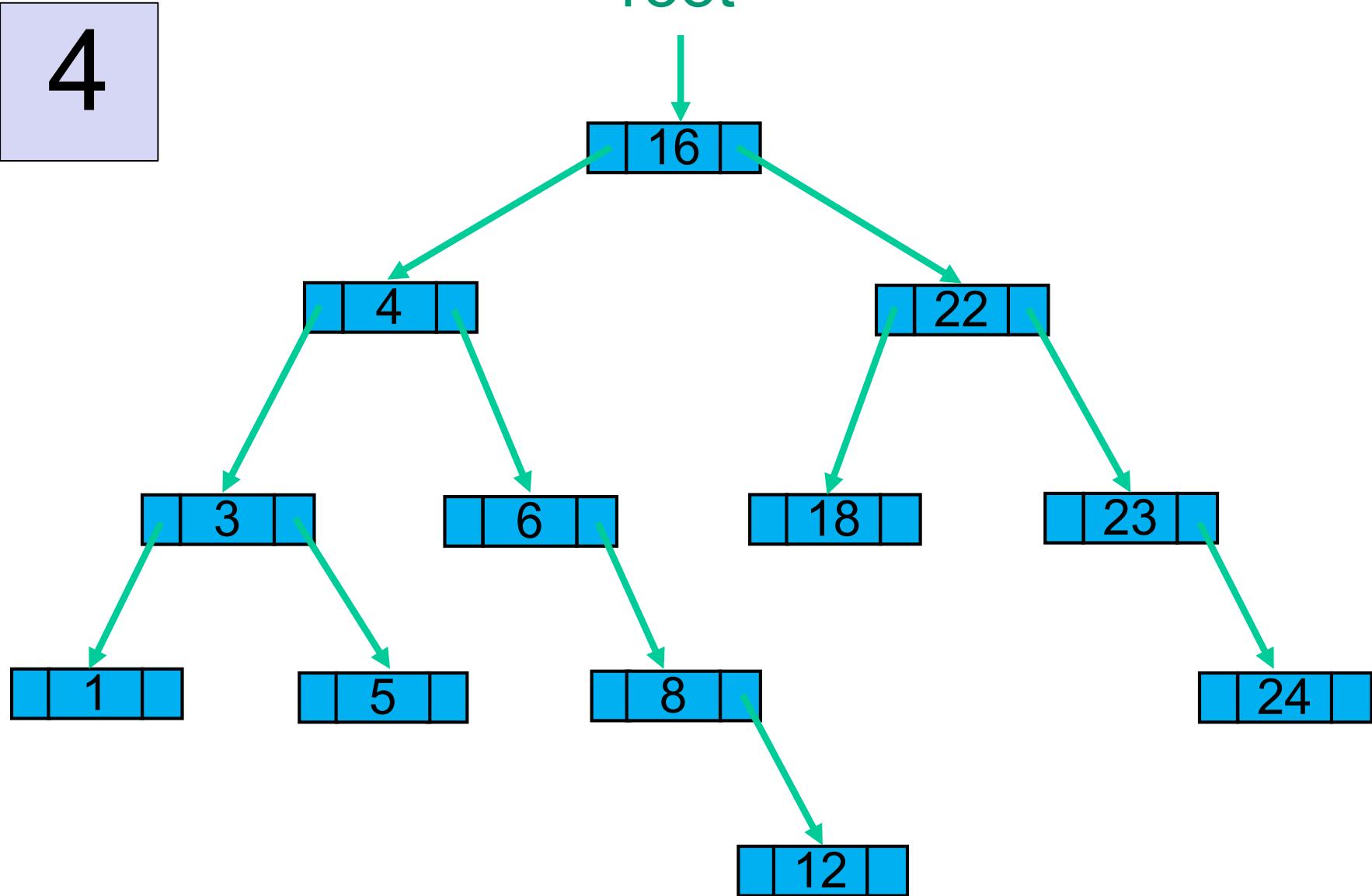
## **Chris**

# How do we remove a value from a binary search tree?

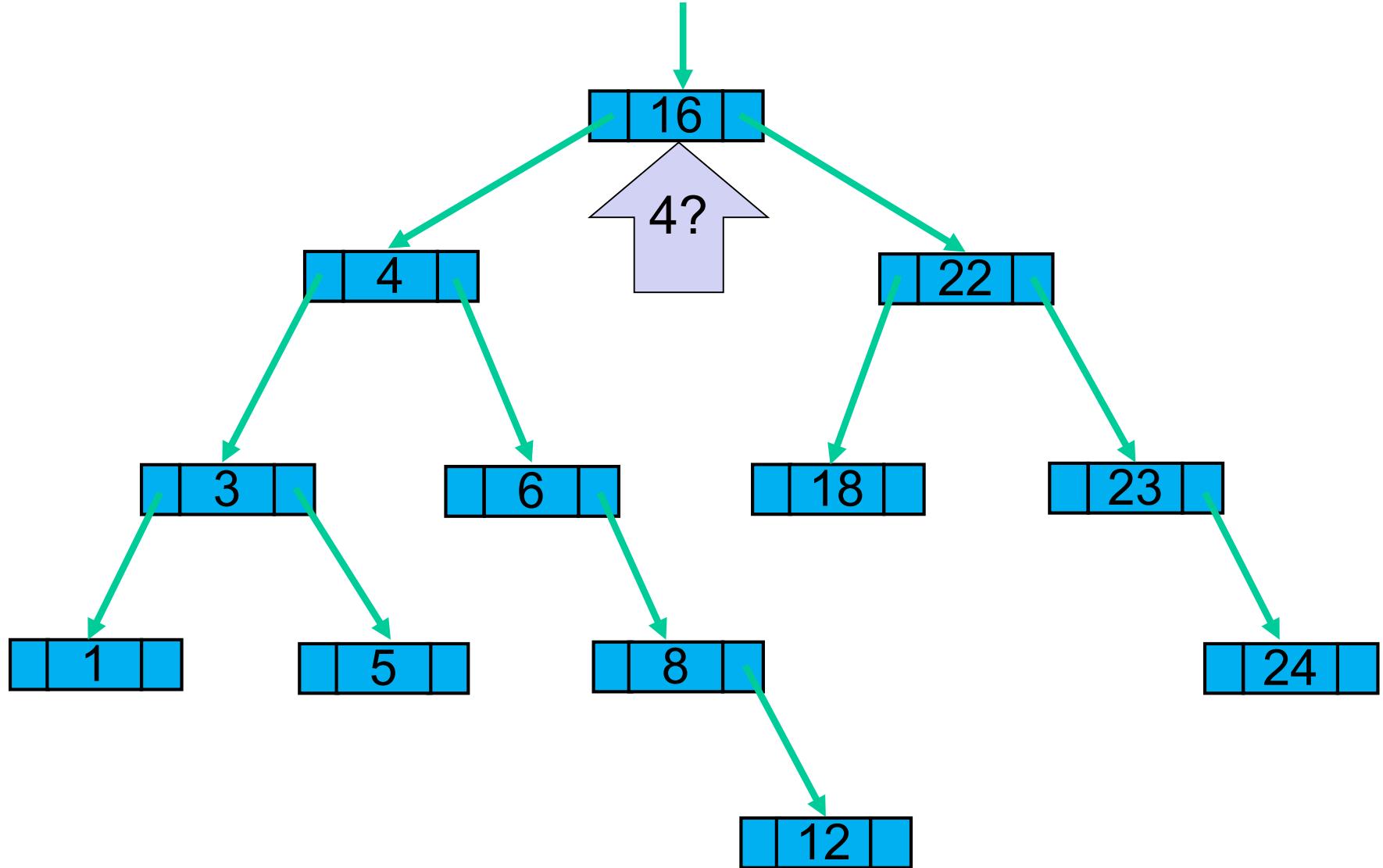
root



root

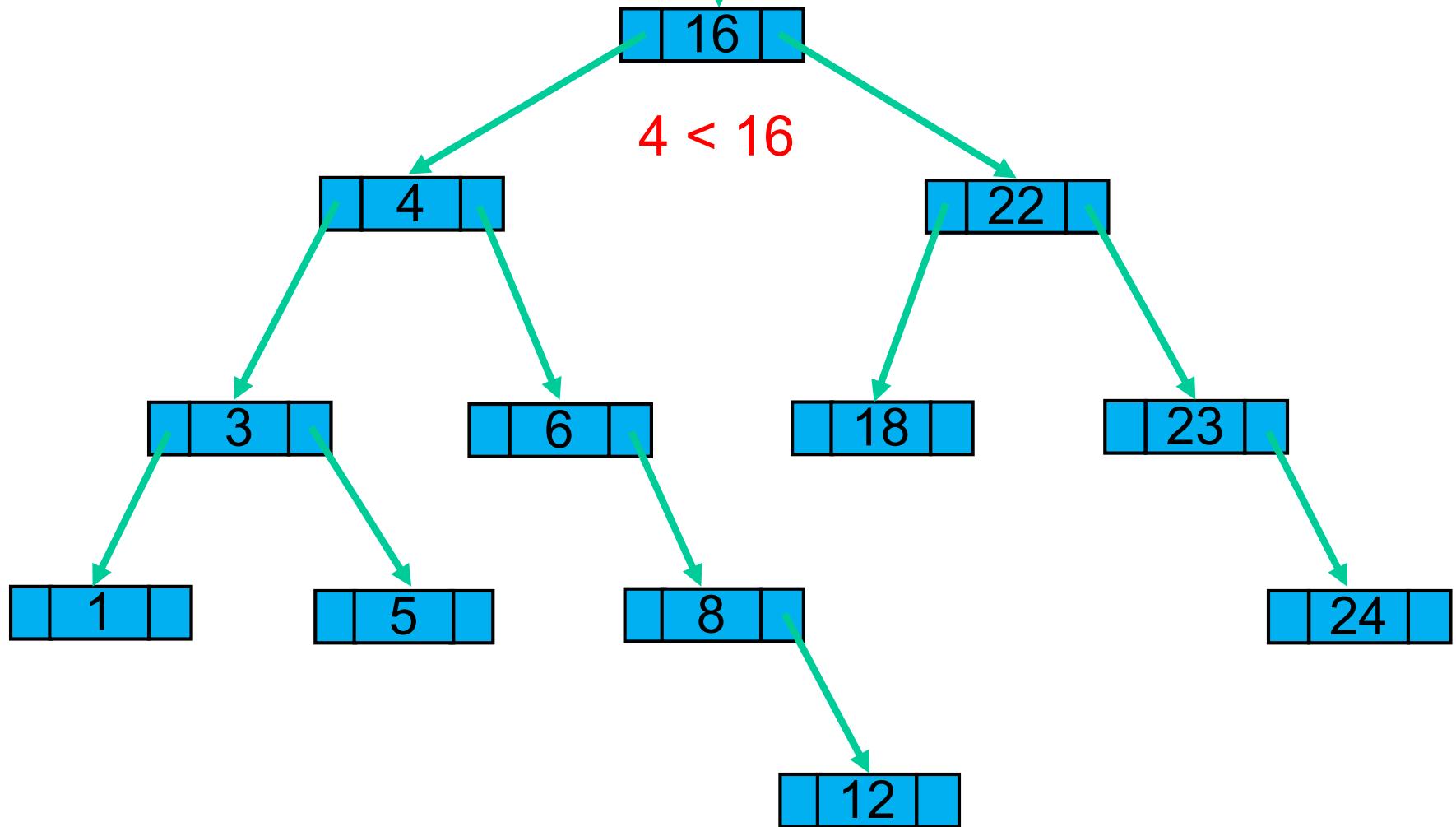


root

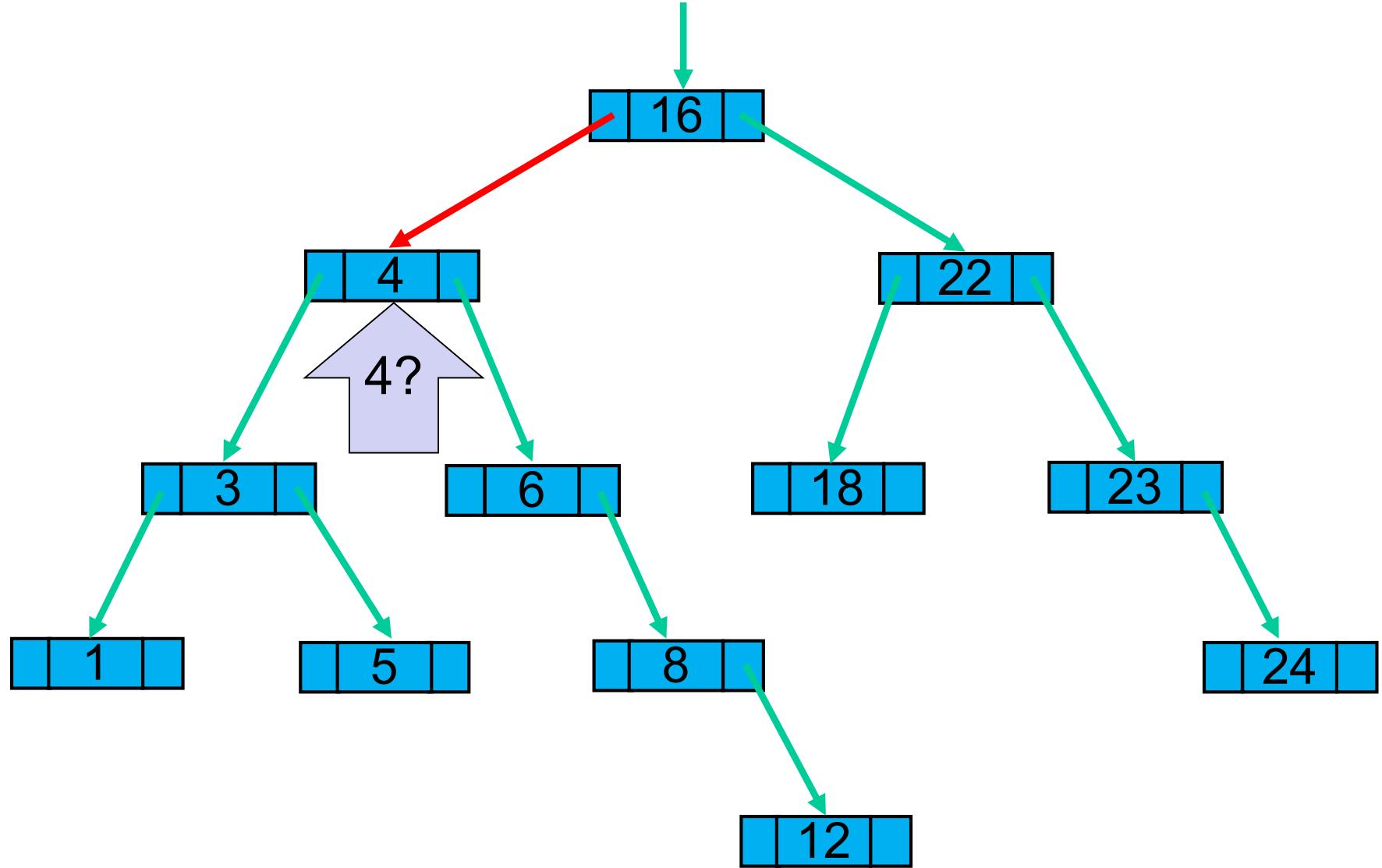


root

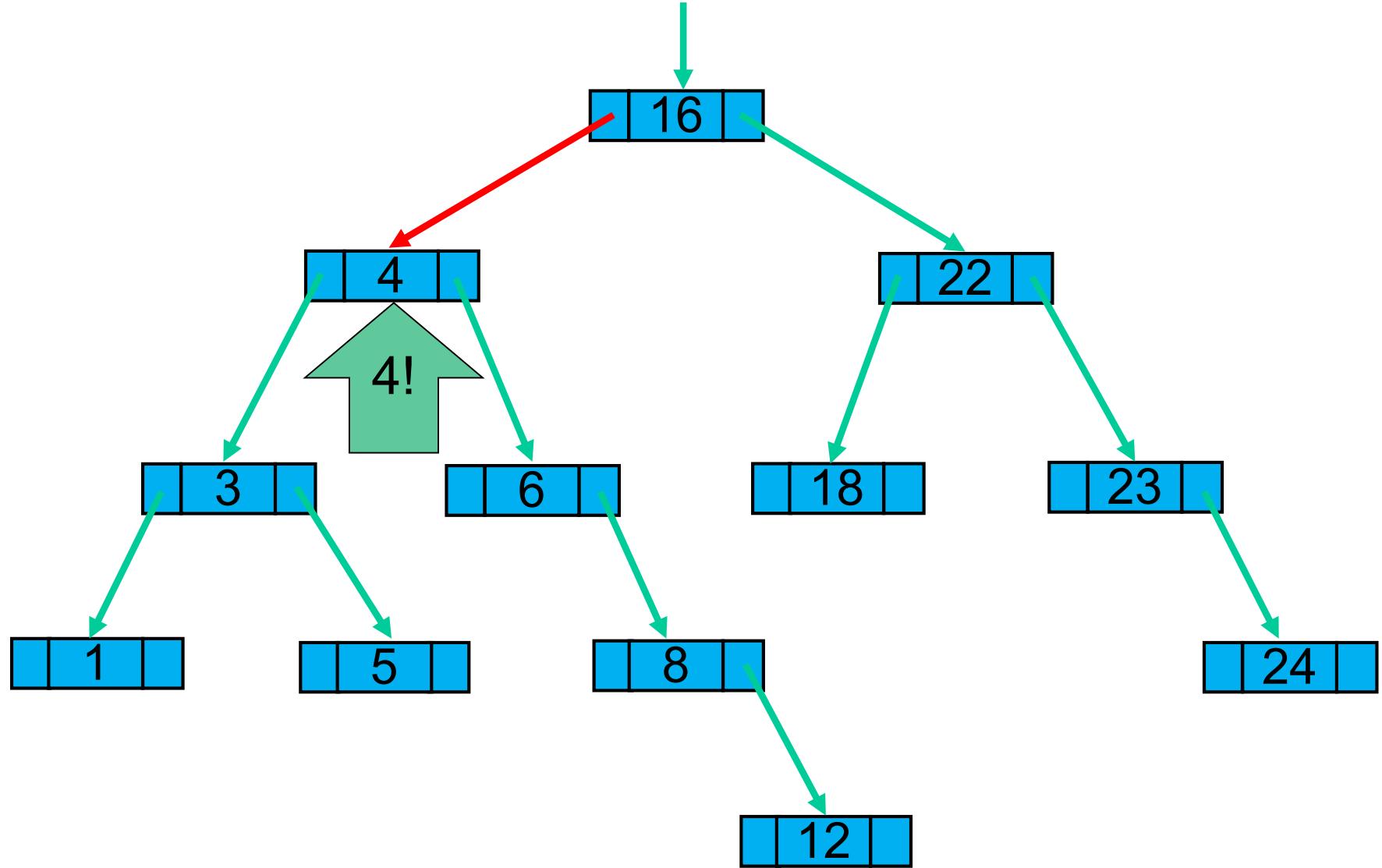
$4 < 16$



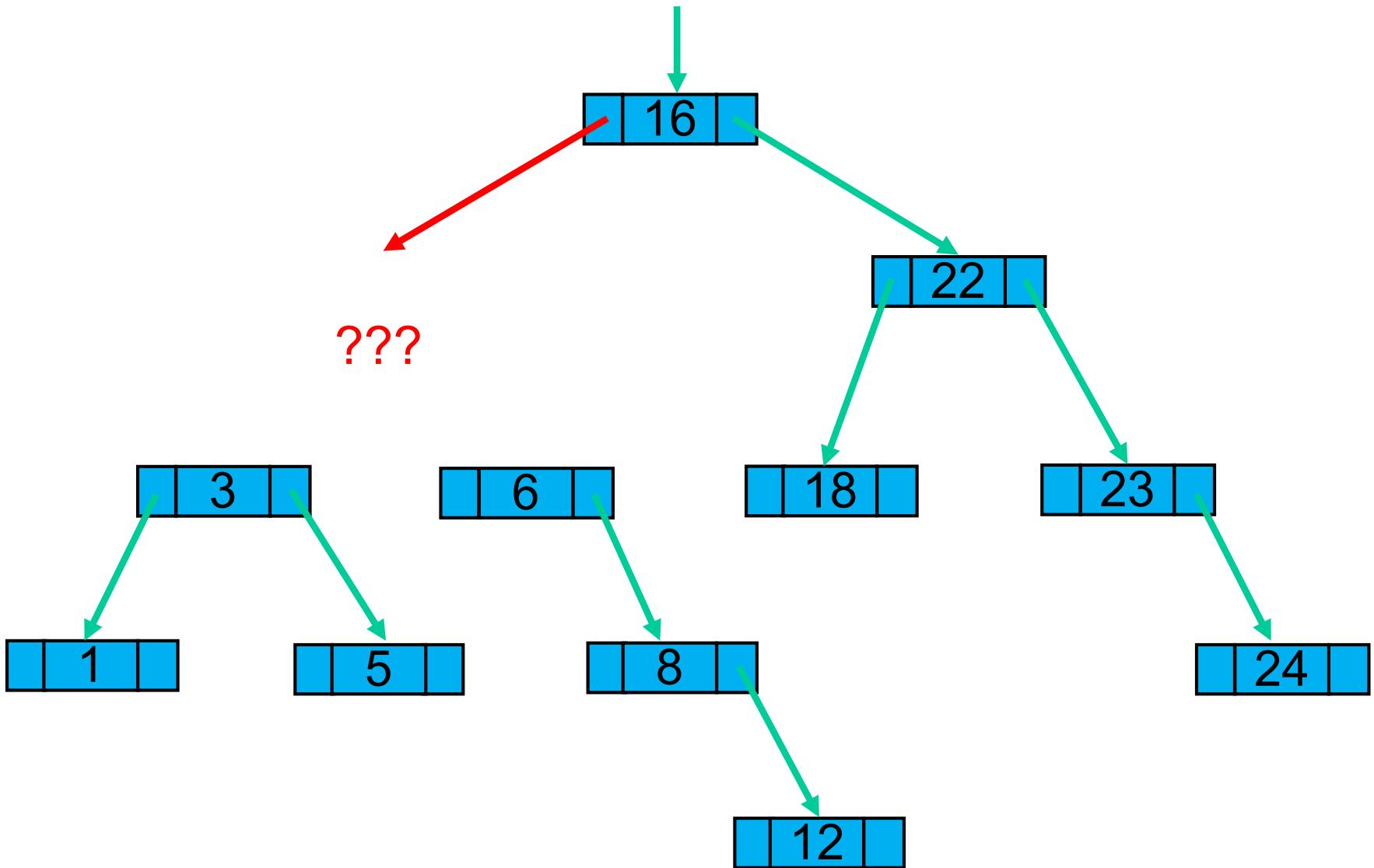
root



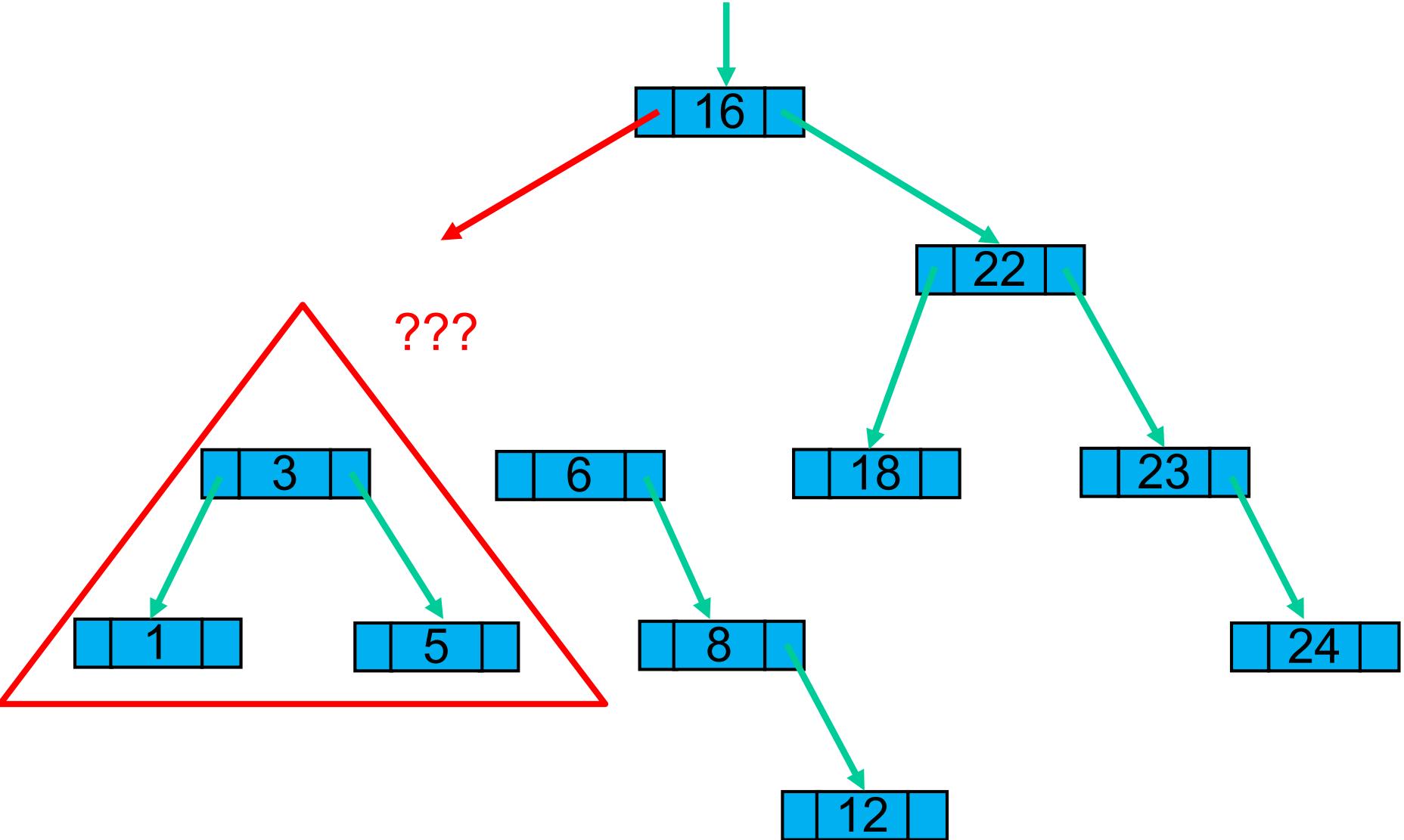
root



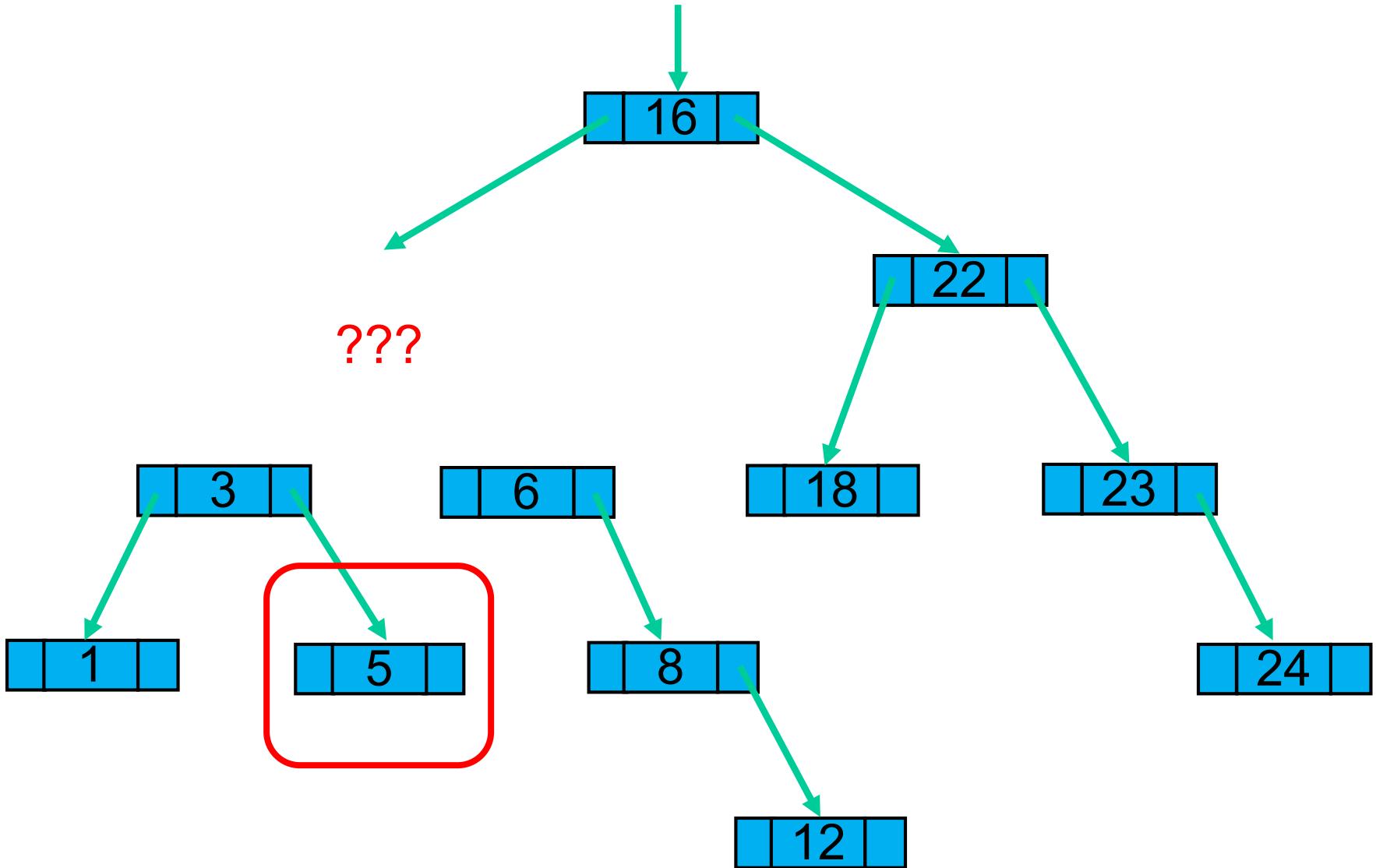
root



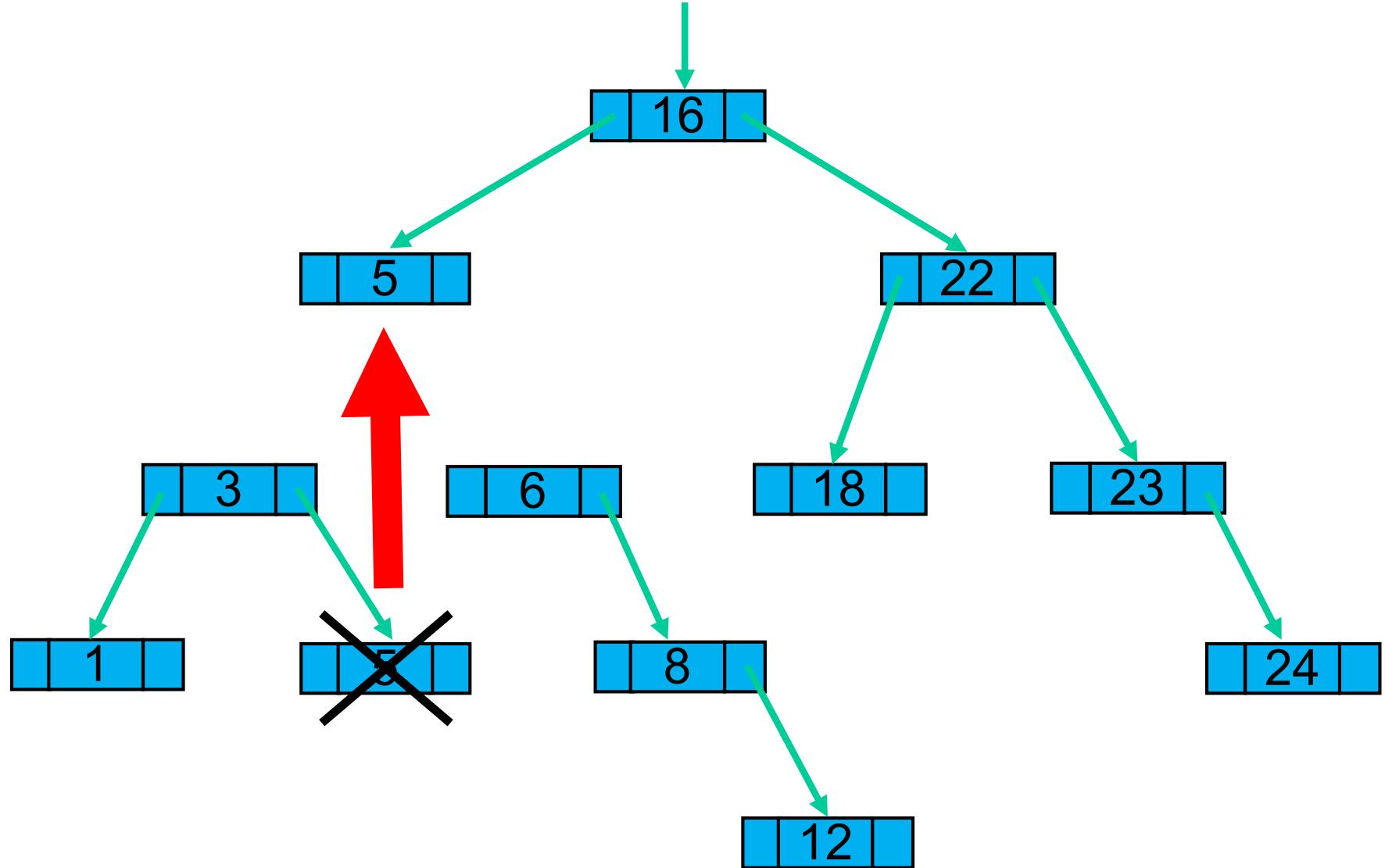
root



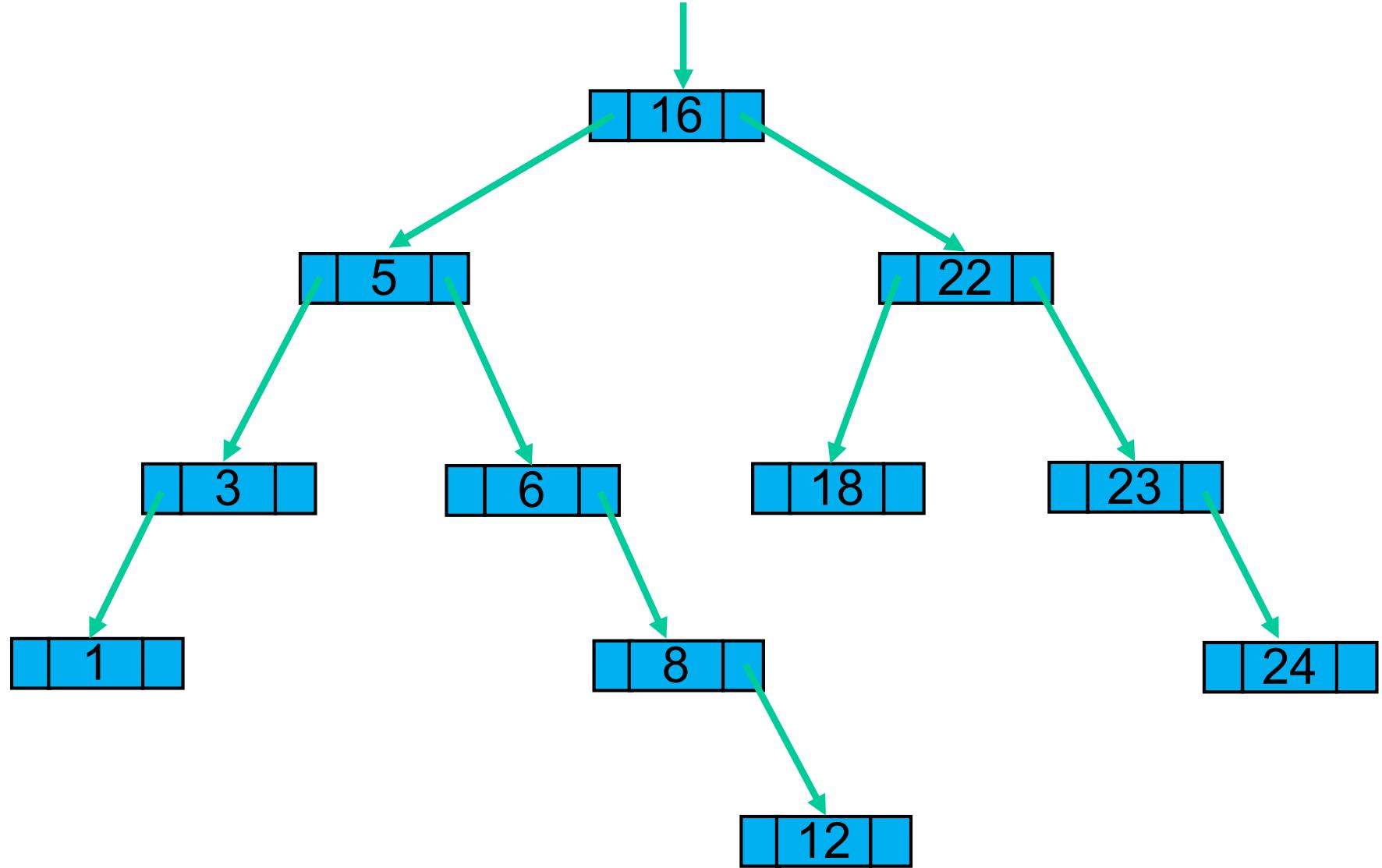
root



root



root





# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null){  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



# Binary Search Tree Remove

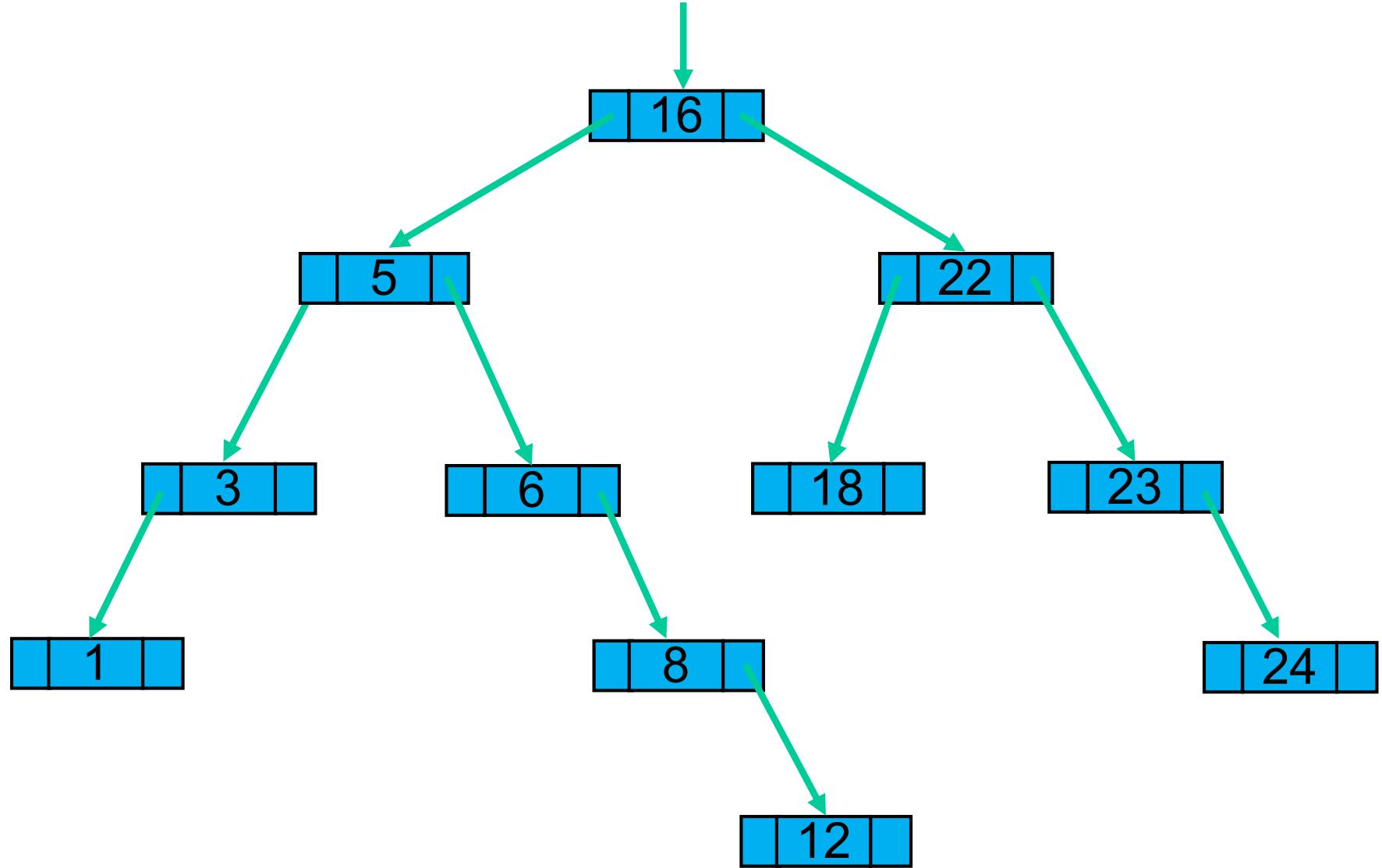
```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null) {  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n) {  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```



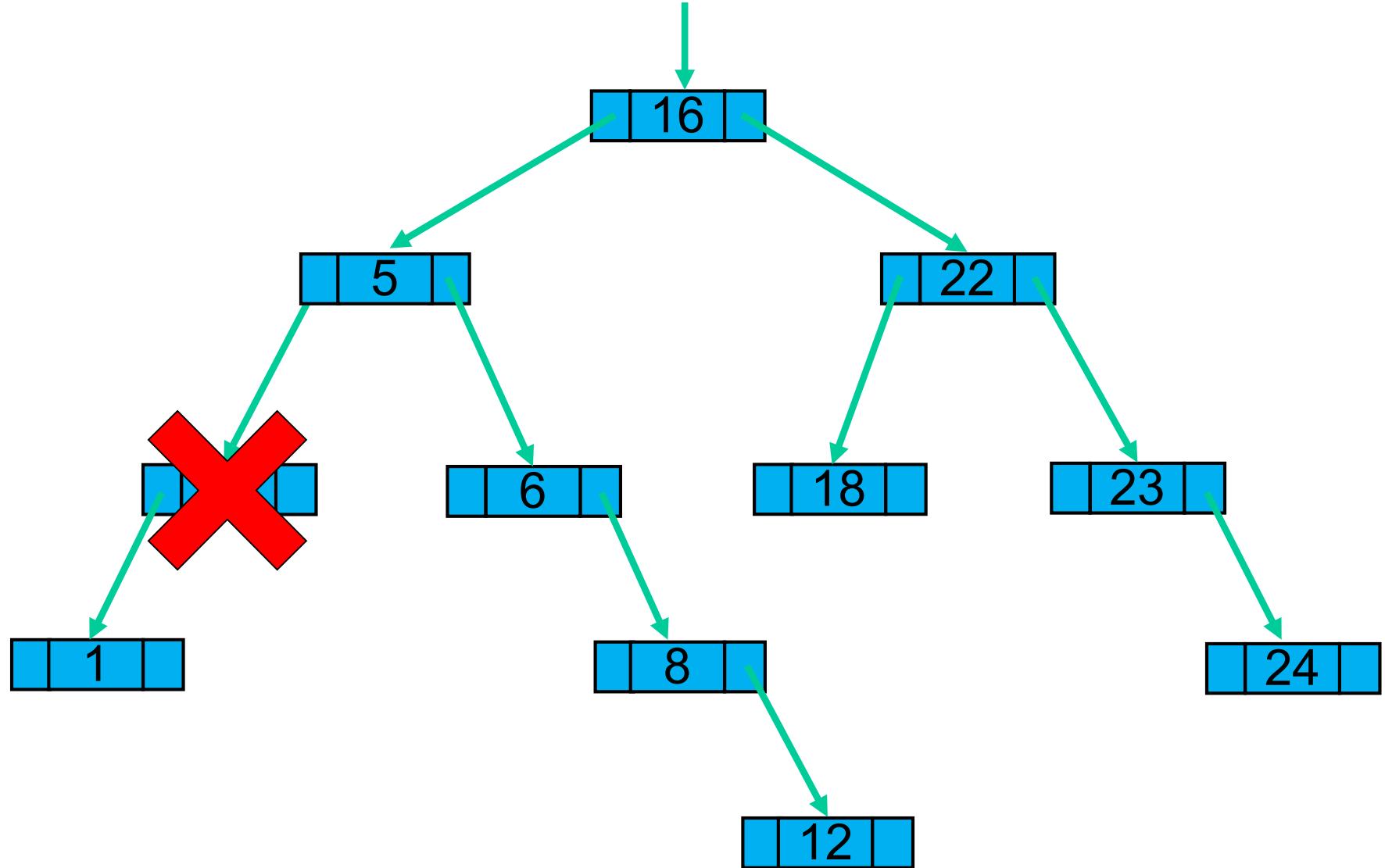
# Binary Search Tree Remove

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null) {  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n) {  
            parent.leftChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = (n.leftChild != null) ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```

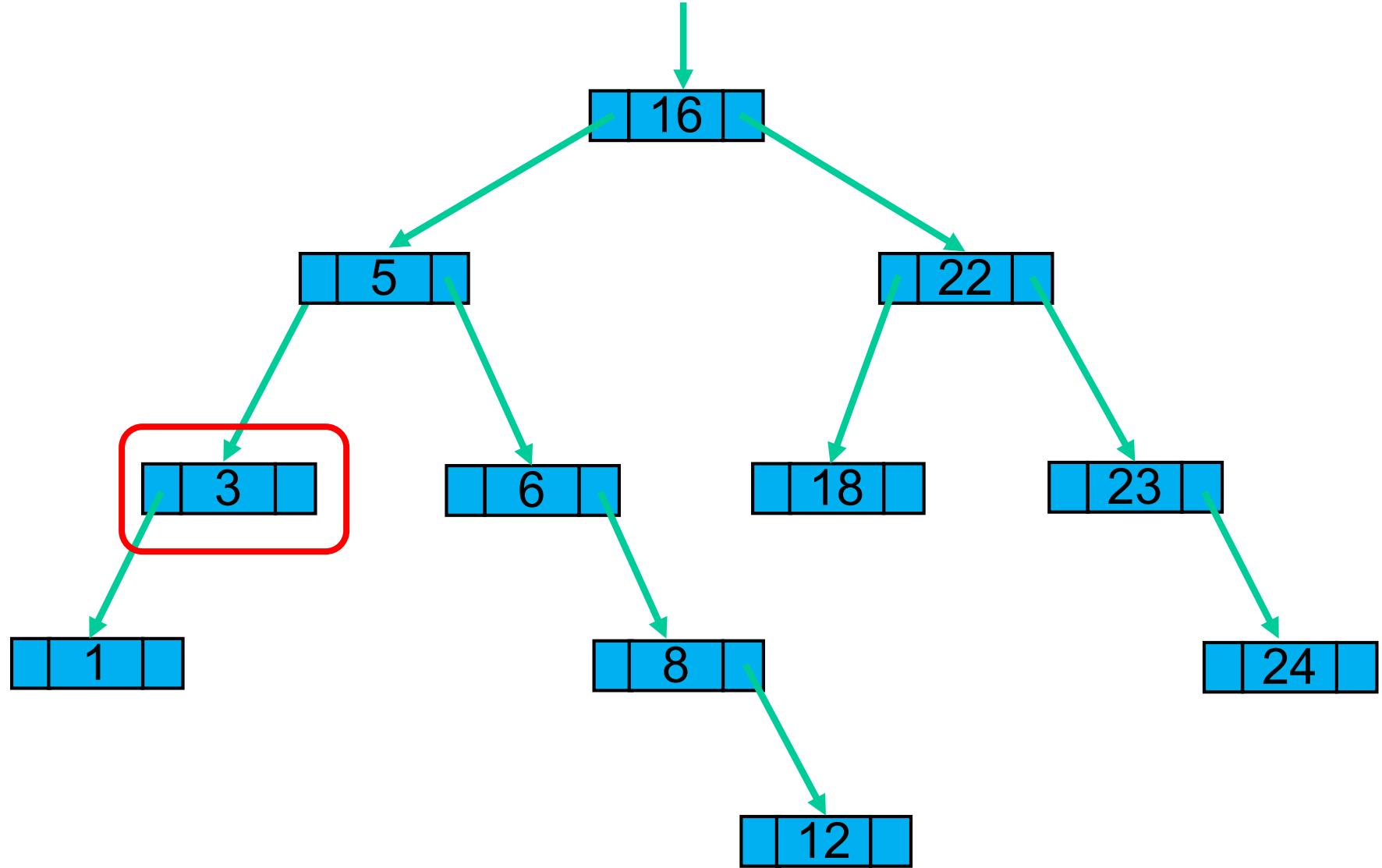
root



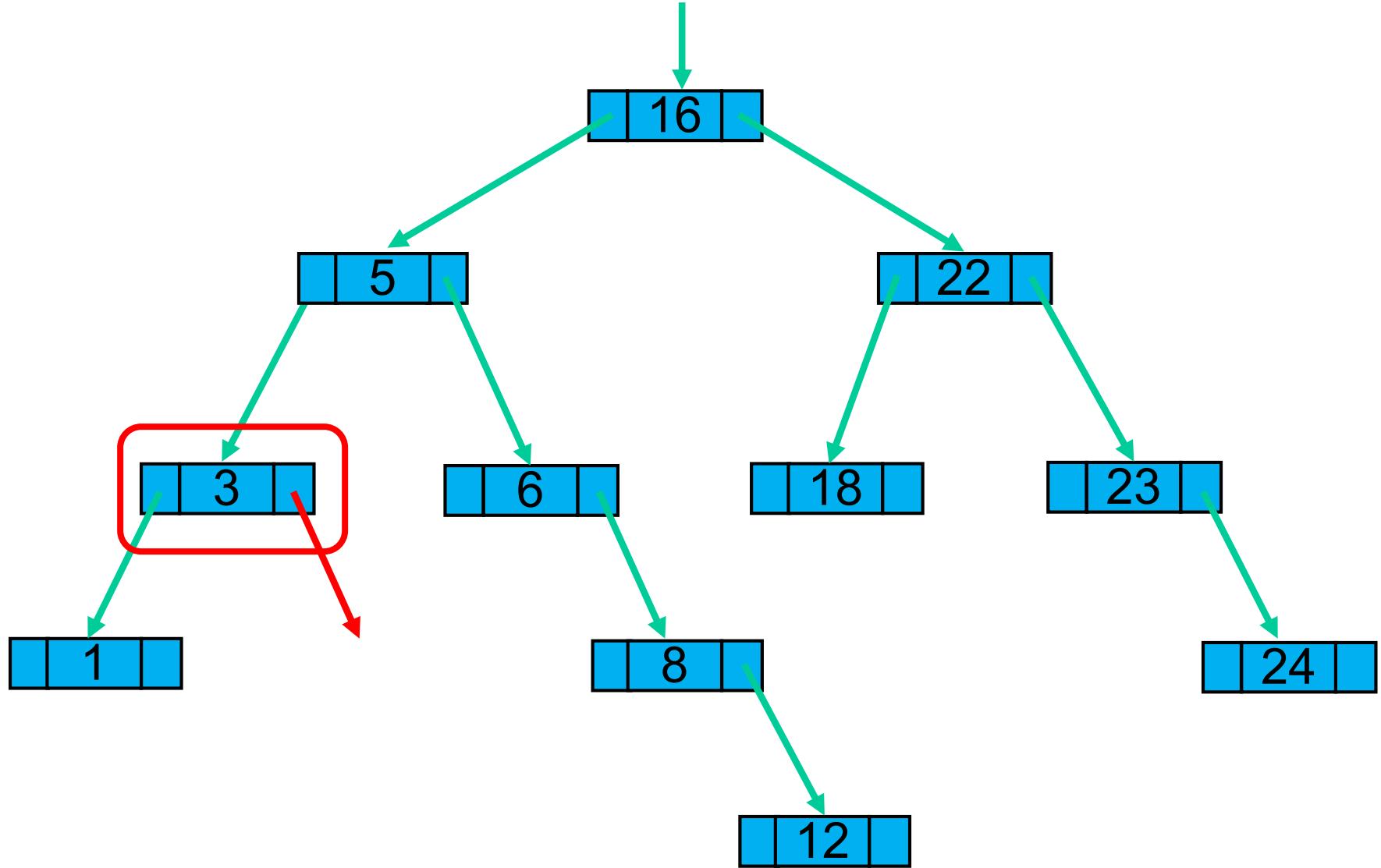
root



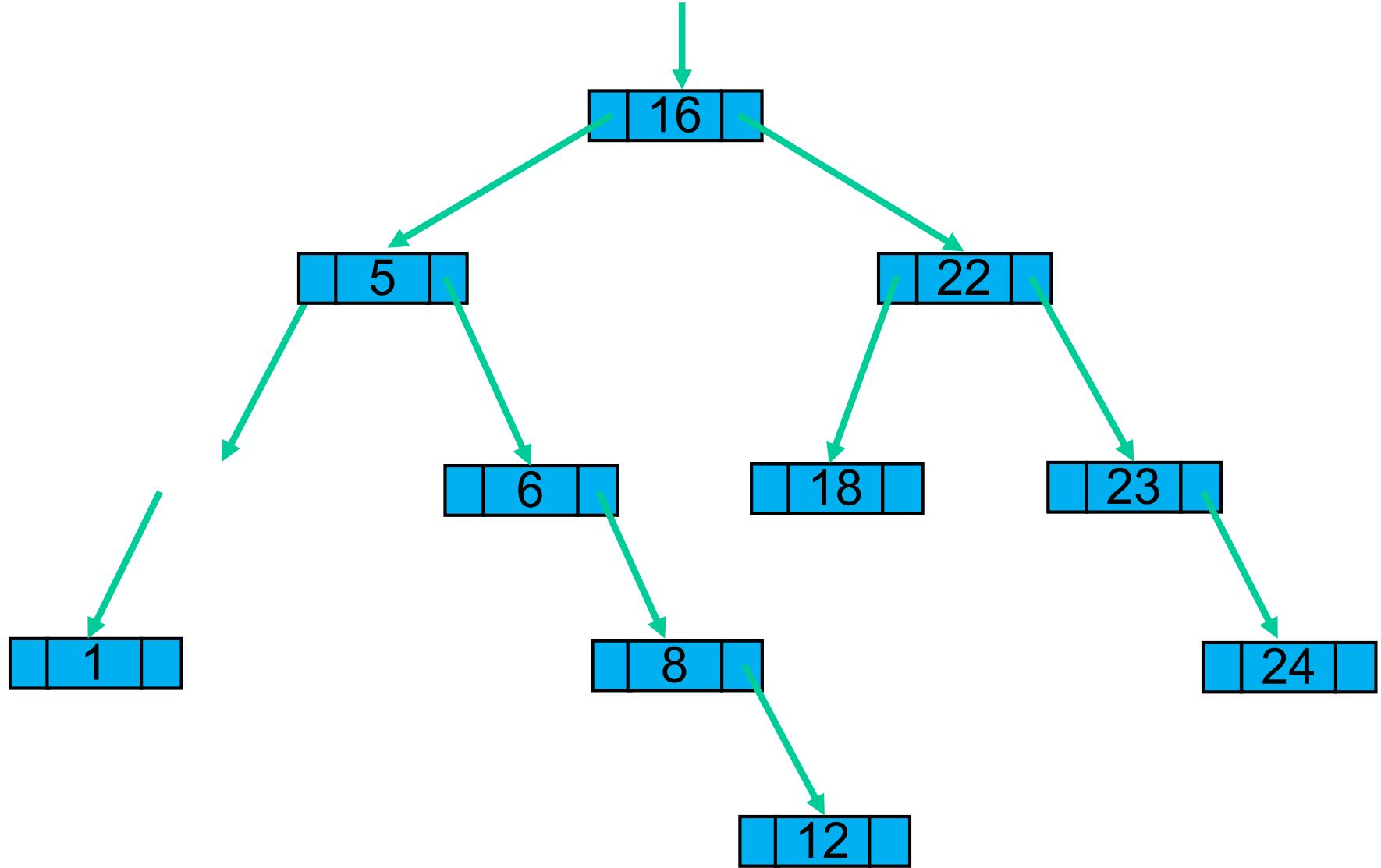
root



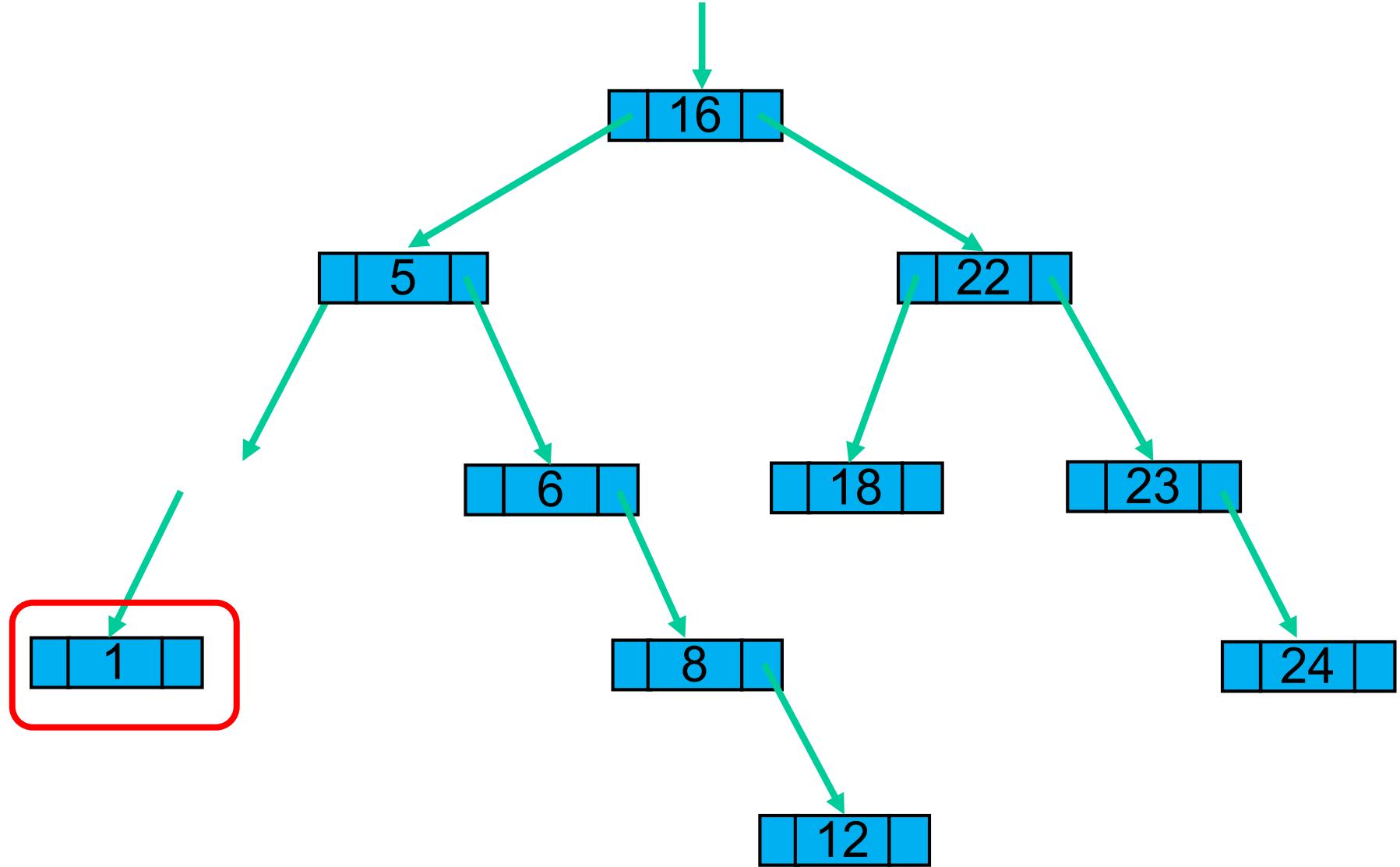
root



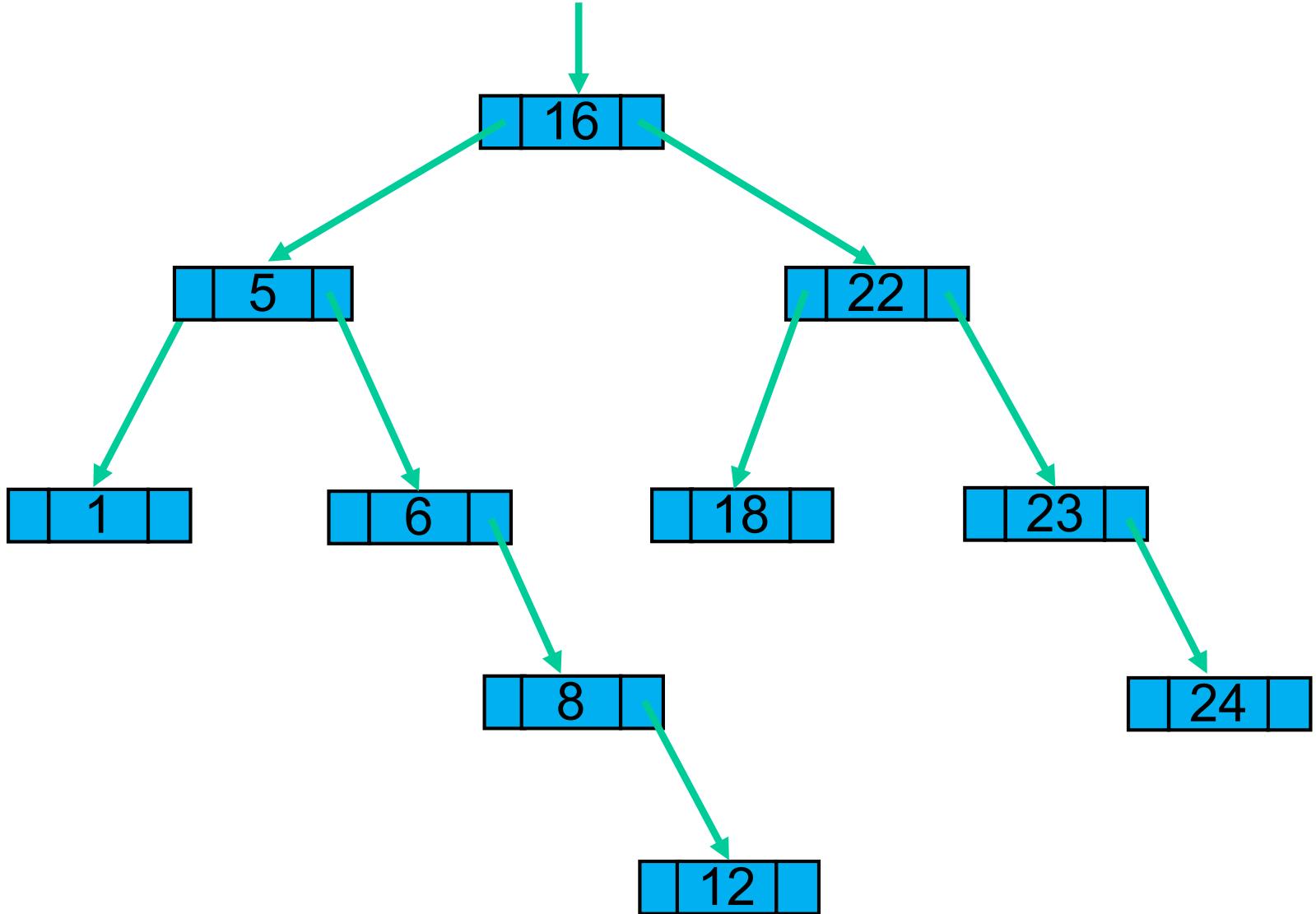
root



root



root



# Binary Search Tree Remove

---

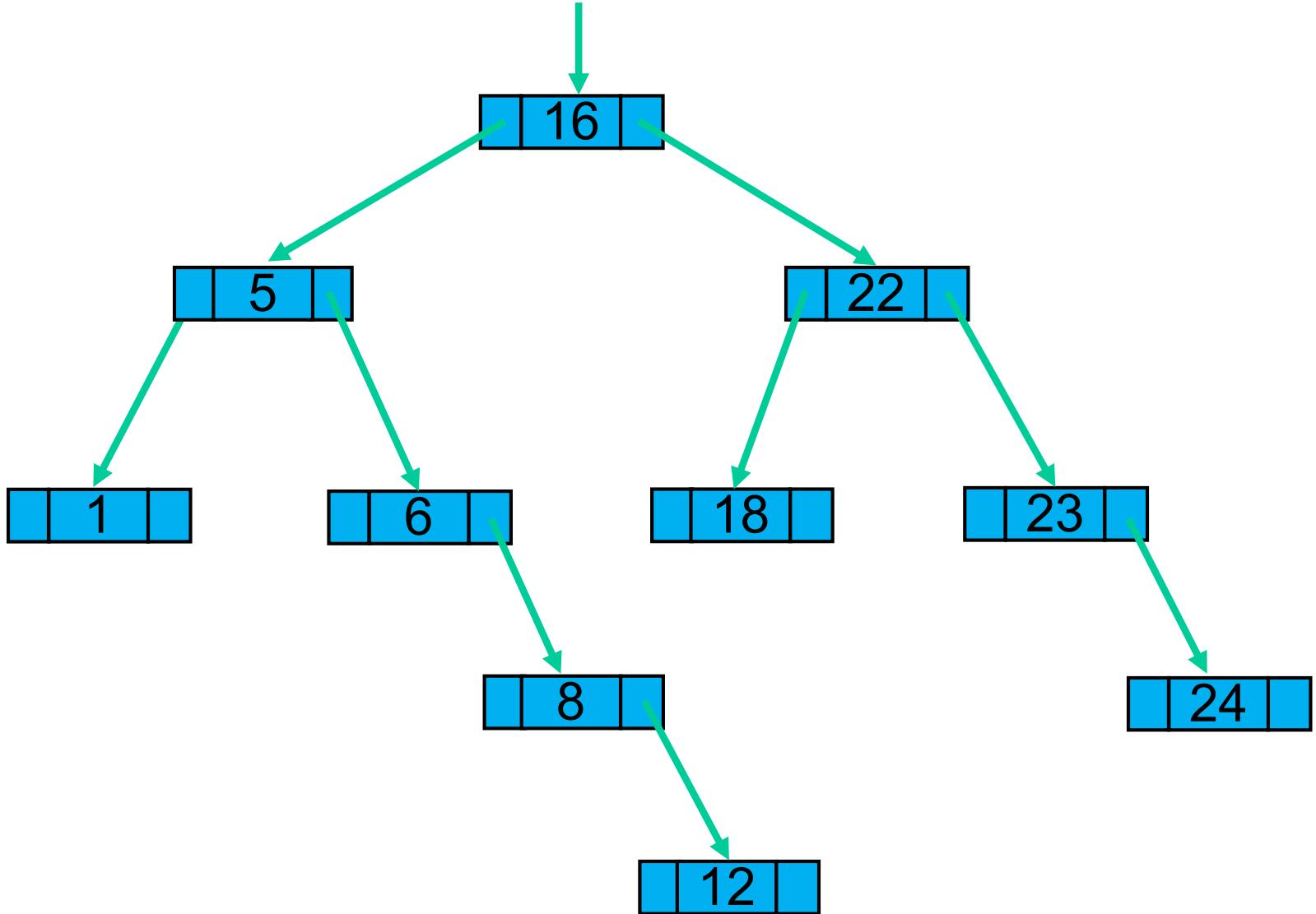
```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null) {  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n){  
            parent.leftChild = n.leftChild ? n.leftChild : n.rightChild;  
        } else {  
            parent.rightChild = n.leftChild ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```

# Binary Search Tree Remove

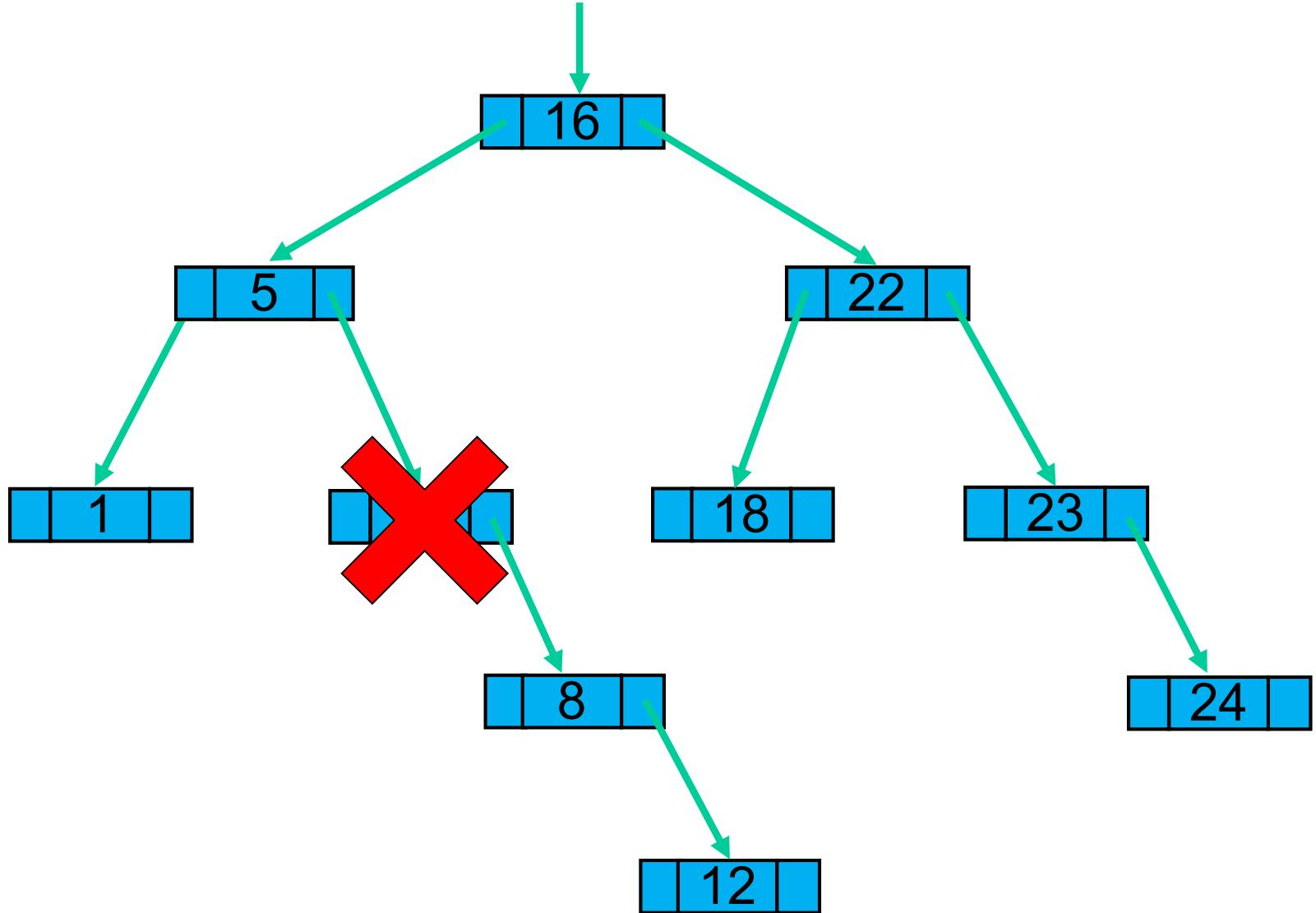
---

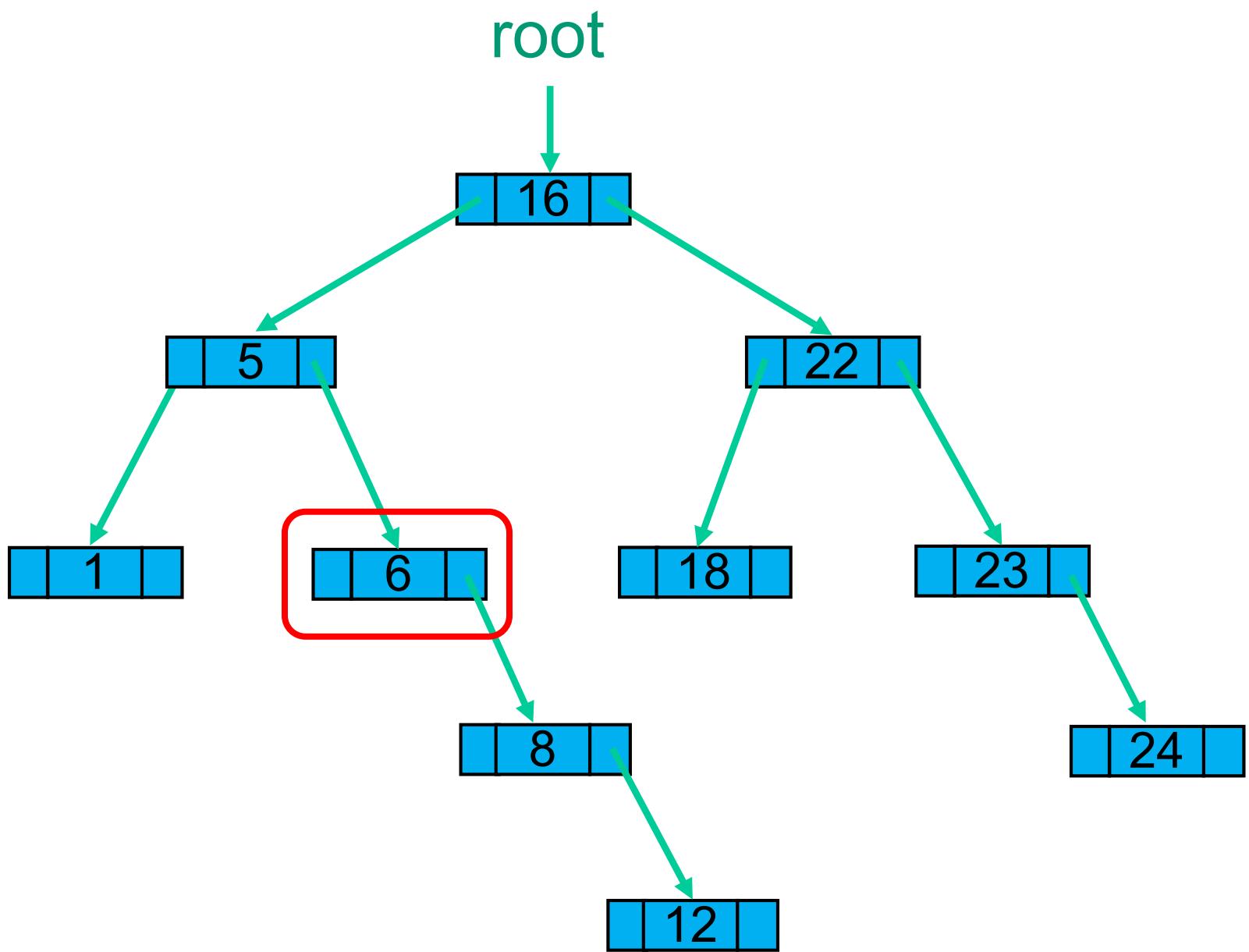
```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null) {  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n) {  
            parent.leftChild = n.leftChild ? n.leftChild : n.rightChild;  
        } else { // parent.rightChild == n  
            parent.rightChild = n.leftChild ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```

root

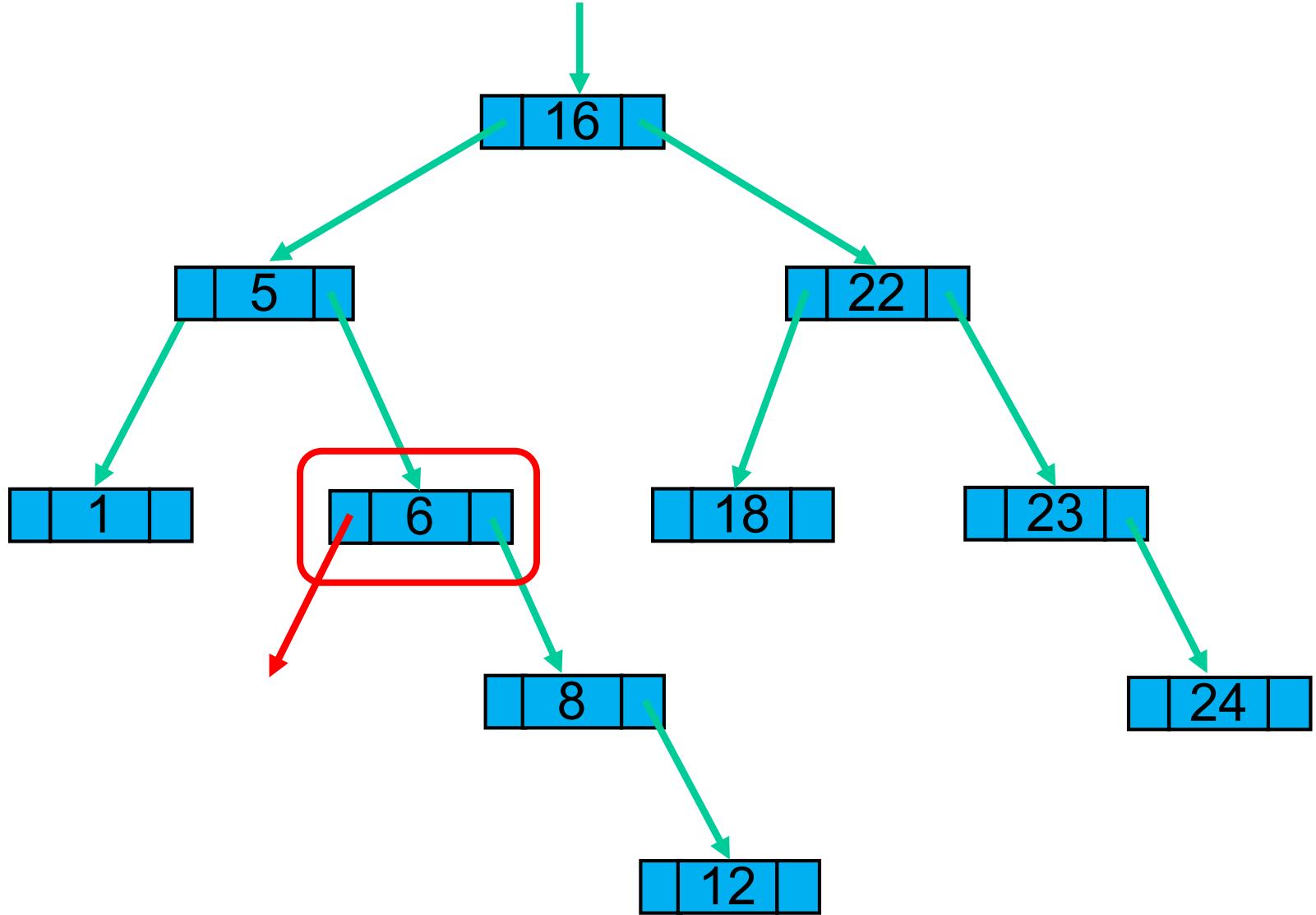


root

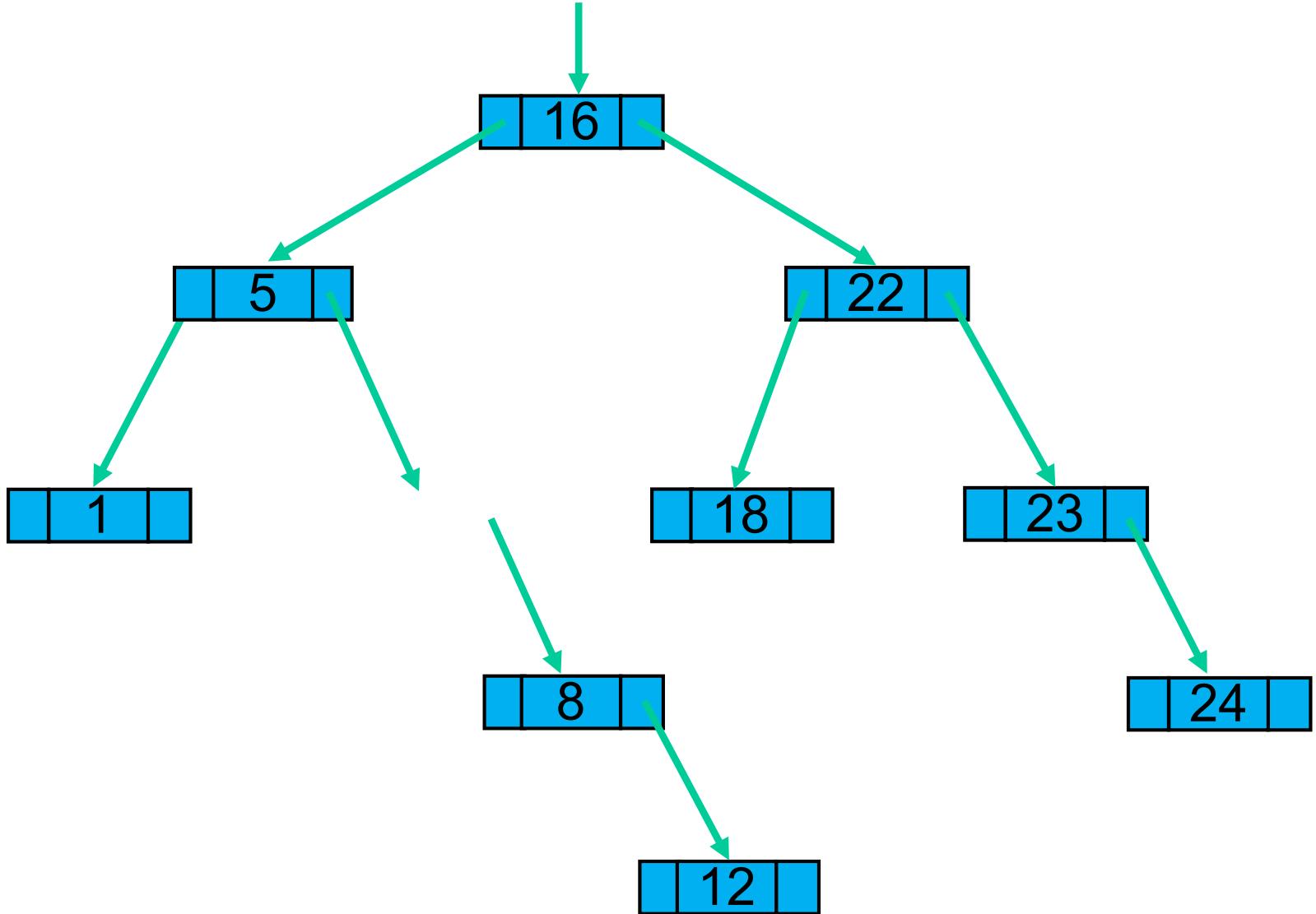




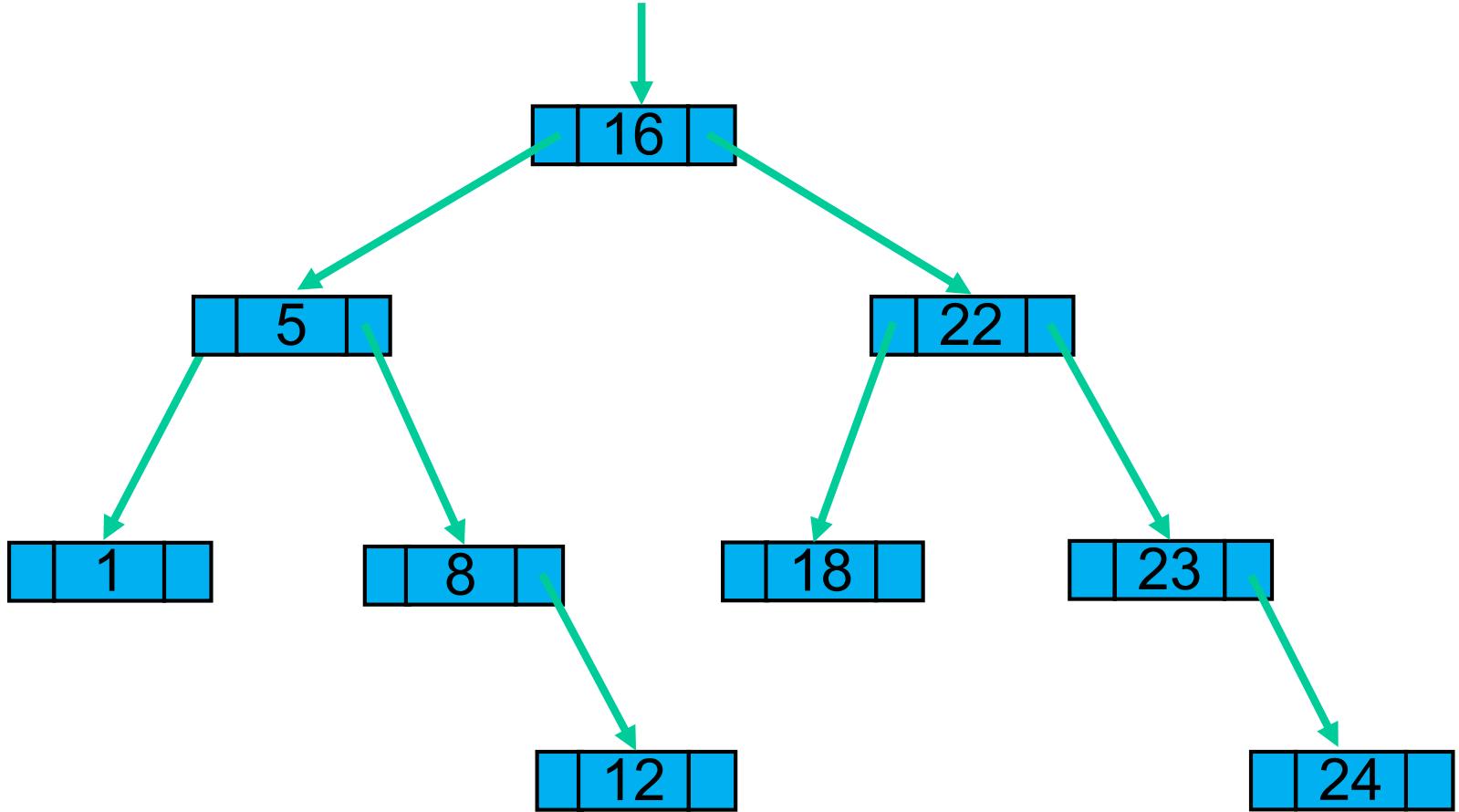
root



root



root



# Binary Search Tree Remove

---

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null) {  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n) {  
            parent.leftChild = n.leftChild ? n.leftChild : n.rightChild;  
        } else { // parent.rightChild == n  
            parent.rightChild = n.leftChild ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```

# Binary Search Tree Remove

---

```
public boolean remove(Node n, Node parent, int val) {  
    if (n == null) return false;  
  
    if (val < n.value) {  
        return remove(n.leftChild, n, val);  
    } else if (val > n.value) {  
        return remove(n.rightChild, n, val);  
    } else {  
        if (n.leftChild != null && n.rightChild != null) {  
            n.value = maxValue(n.leftChild);  
            remove(n.leftChild, n, n.value);  
        } else if (parent.leftChild == n) {  
            parent.leftChild = n.leftChild ? n.leftChild : n.rightChild;  
        } else { // parent.rightChild == n  
            parent.rightChild = n.leftChild ? n.leftChild : n.rightChild;  
        }  
        return true;  
    }  
}
```

# Recap: Removing a node from a BST

---

- When a node is removed from a Binary Search Tree, it should be replaced with the largest value in its left subtree

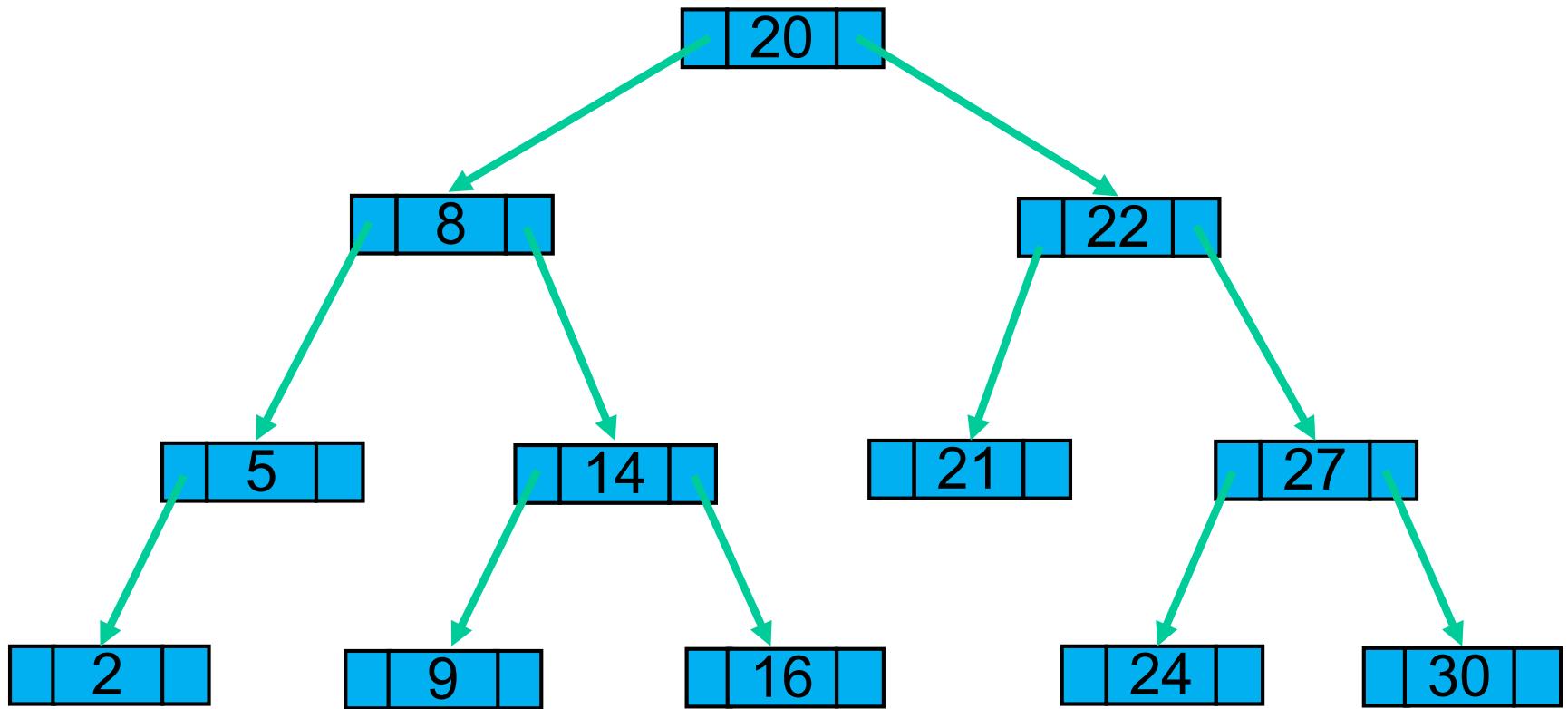
# **SD2x2.5**

# **BST Complexity**

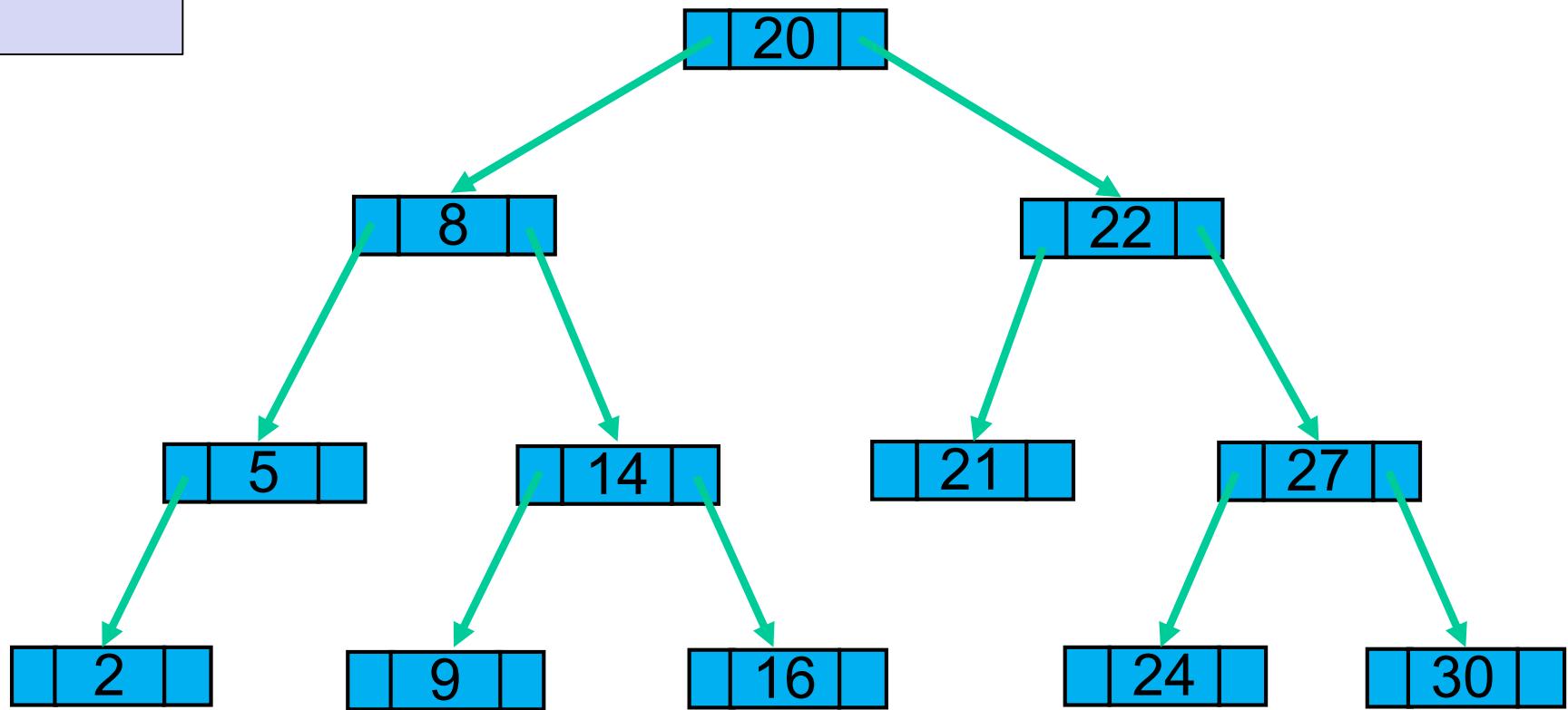
# **Red Black Trees**

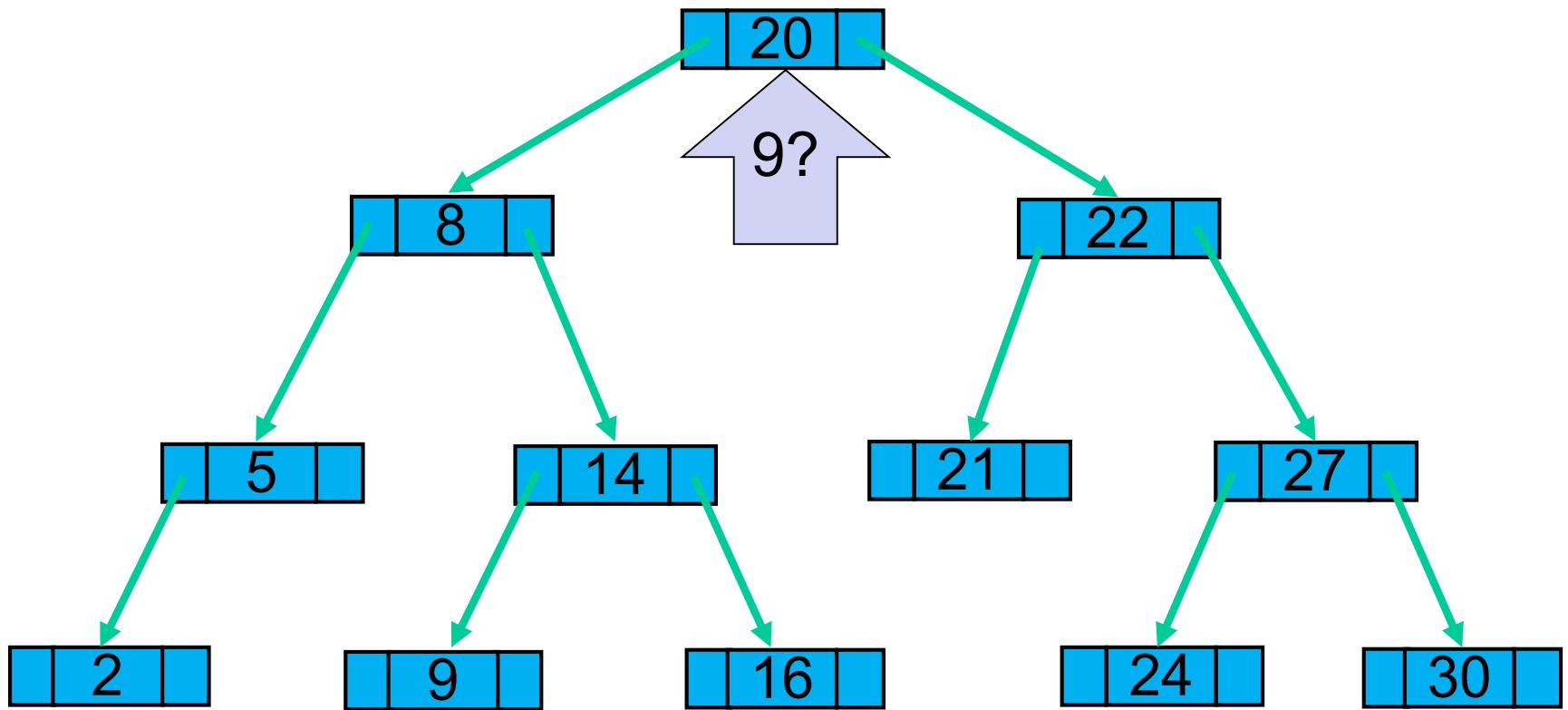
# **Chris**

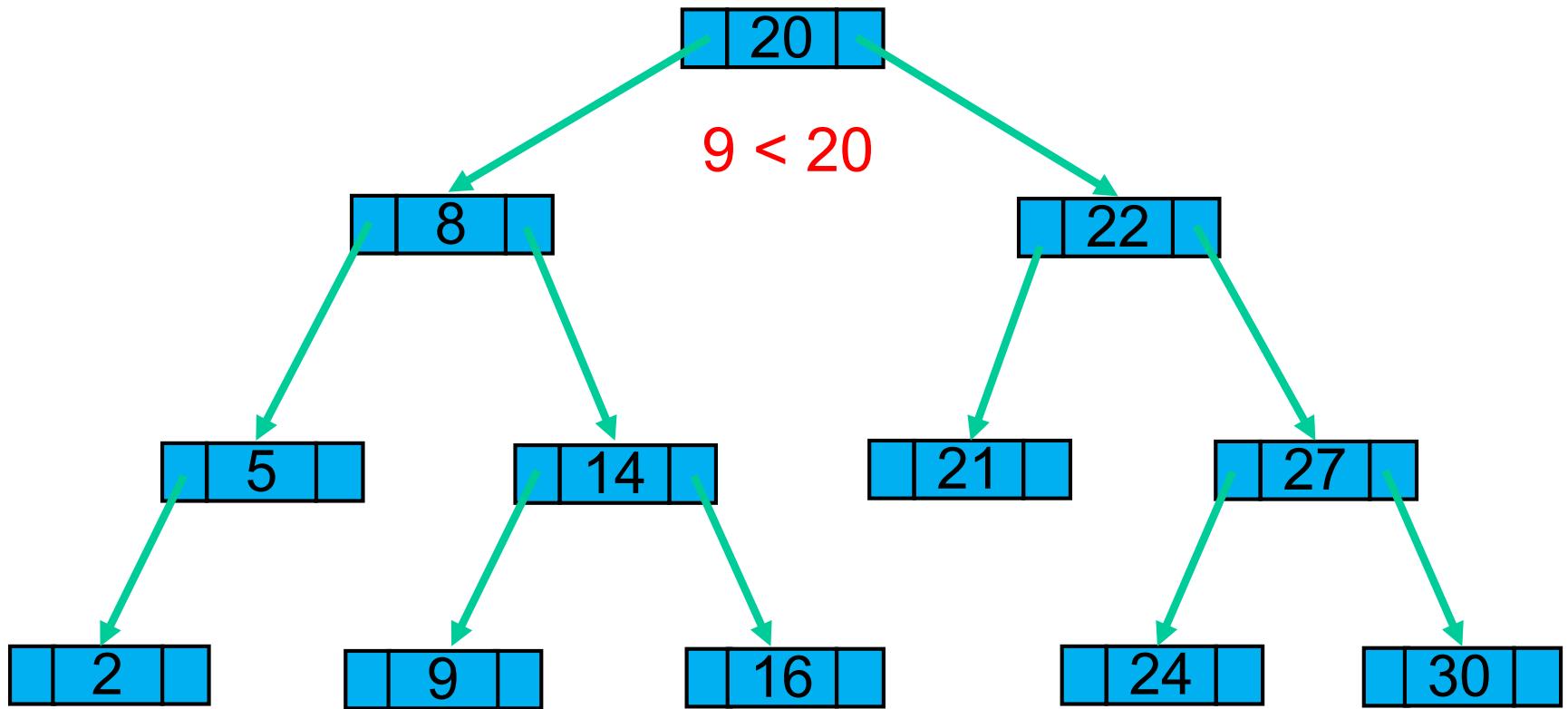
**What is the (algorithmic)  
complexity of finding an  
element in a binary search  
tree?**

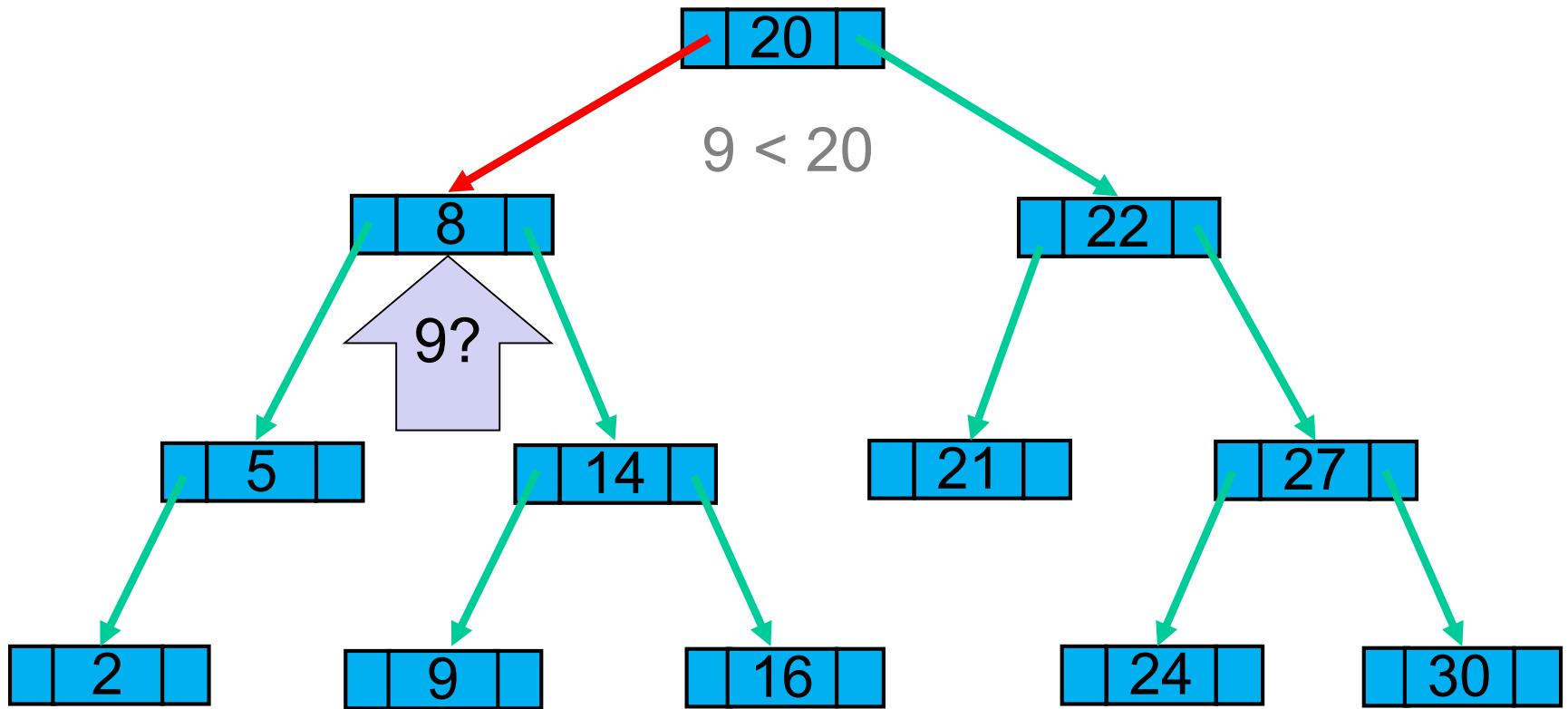


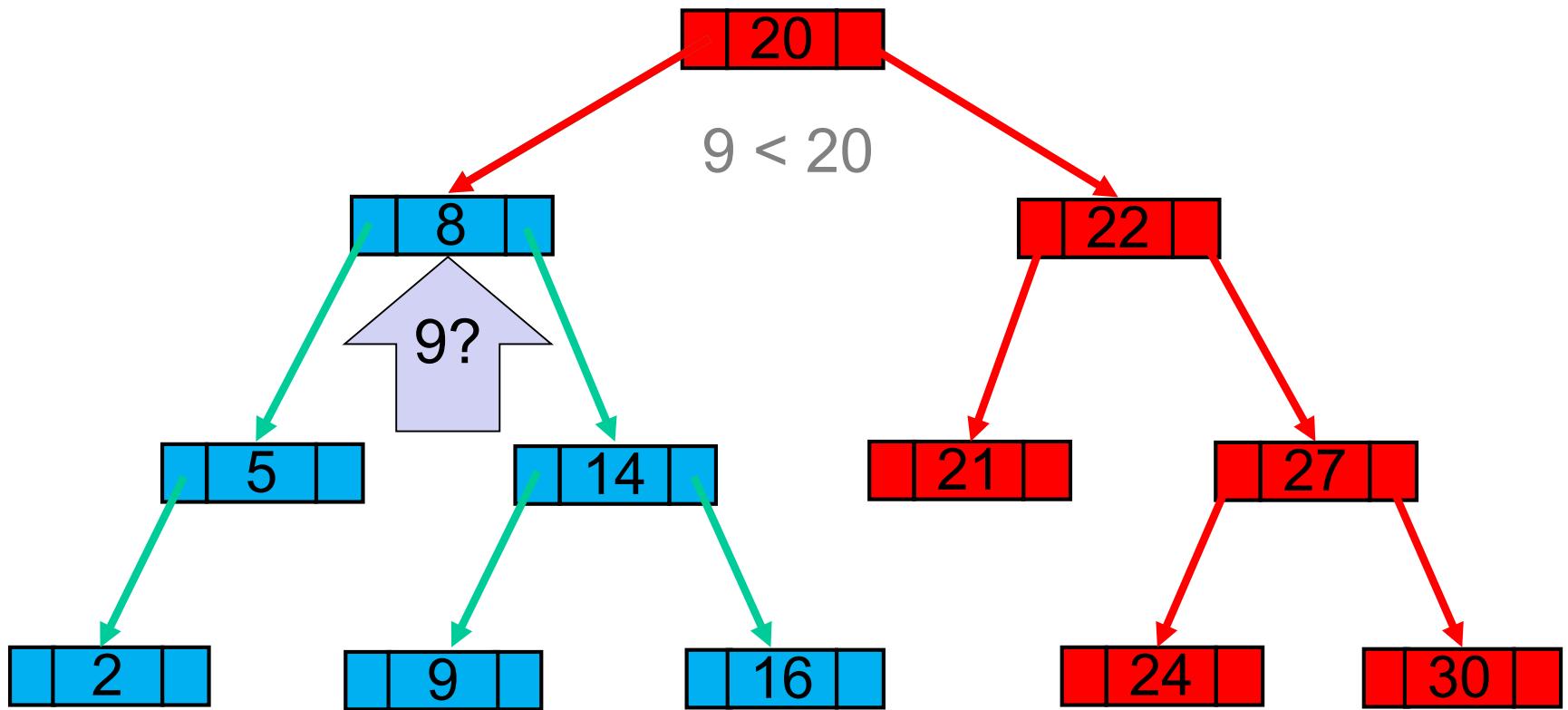
9

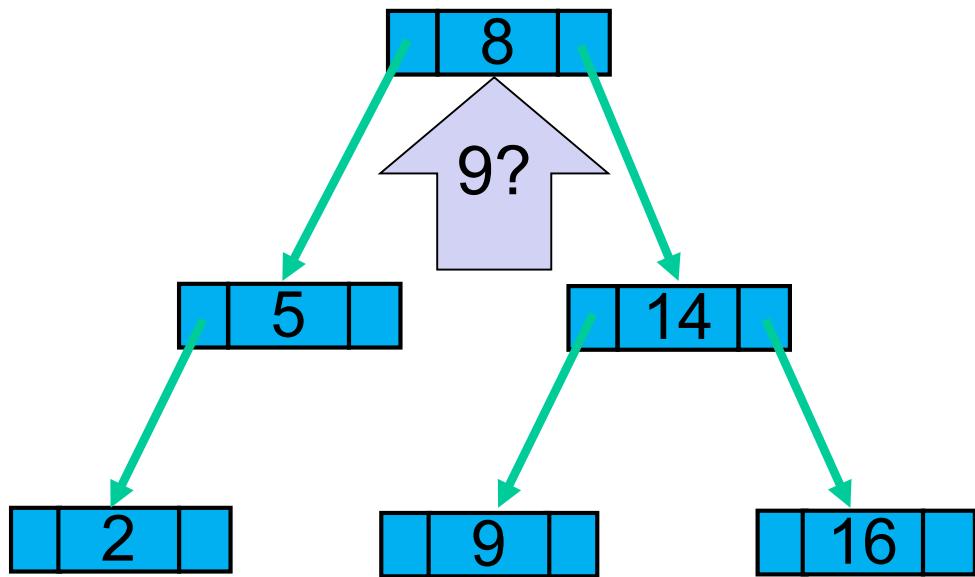


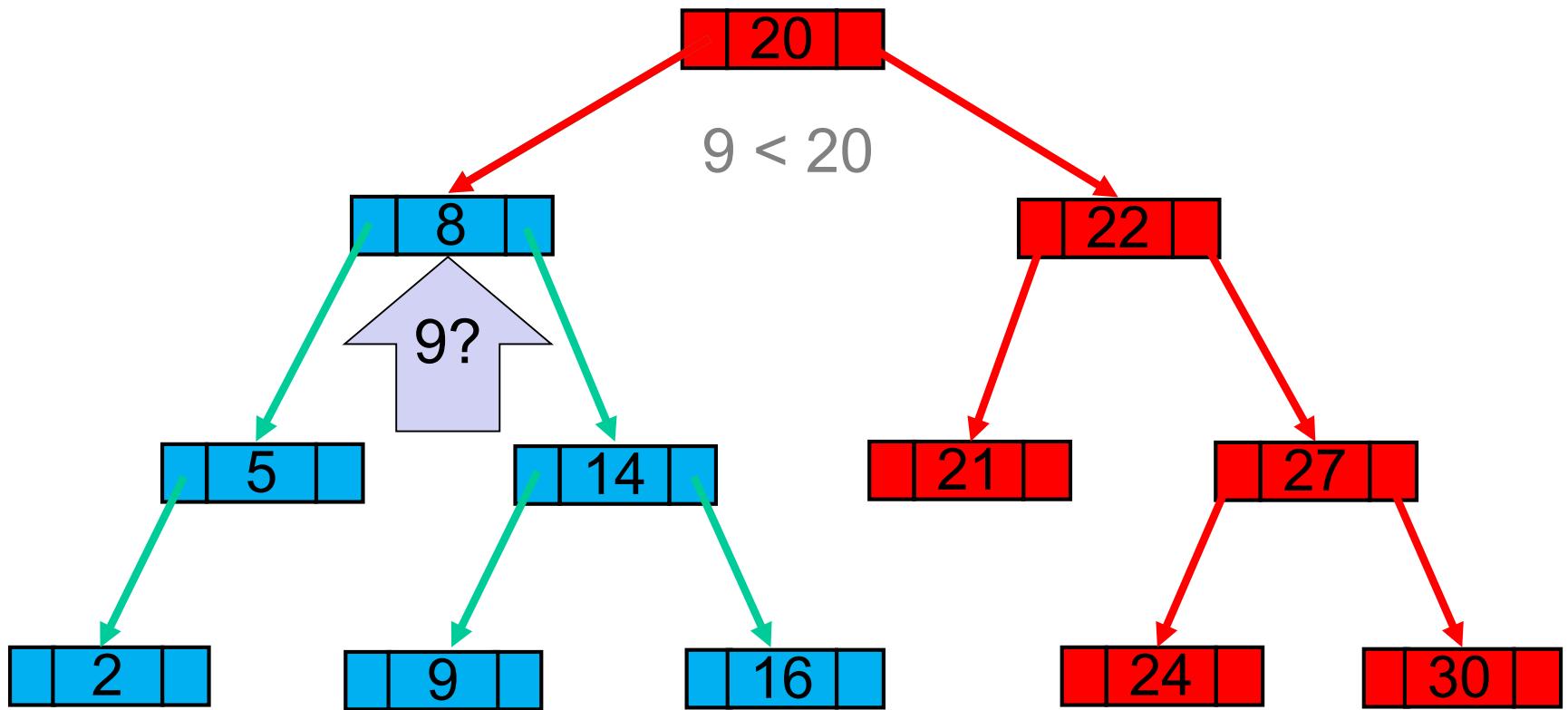


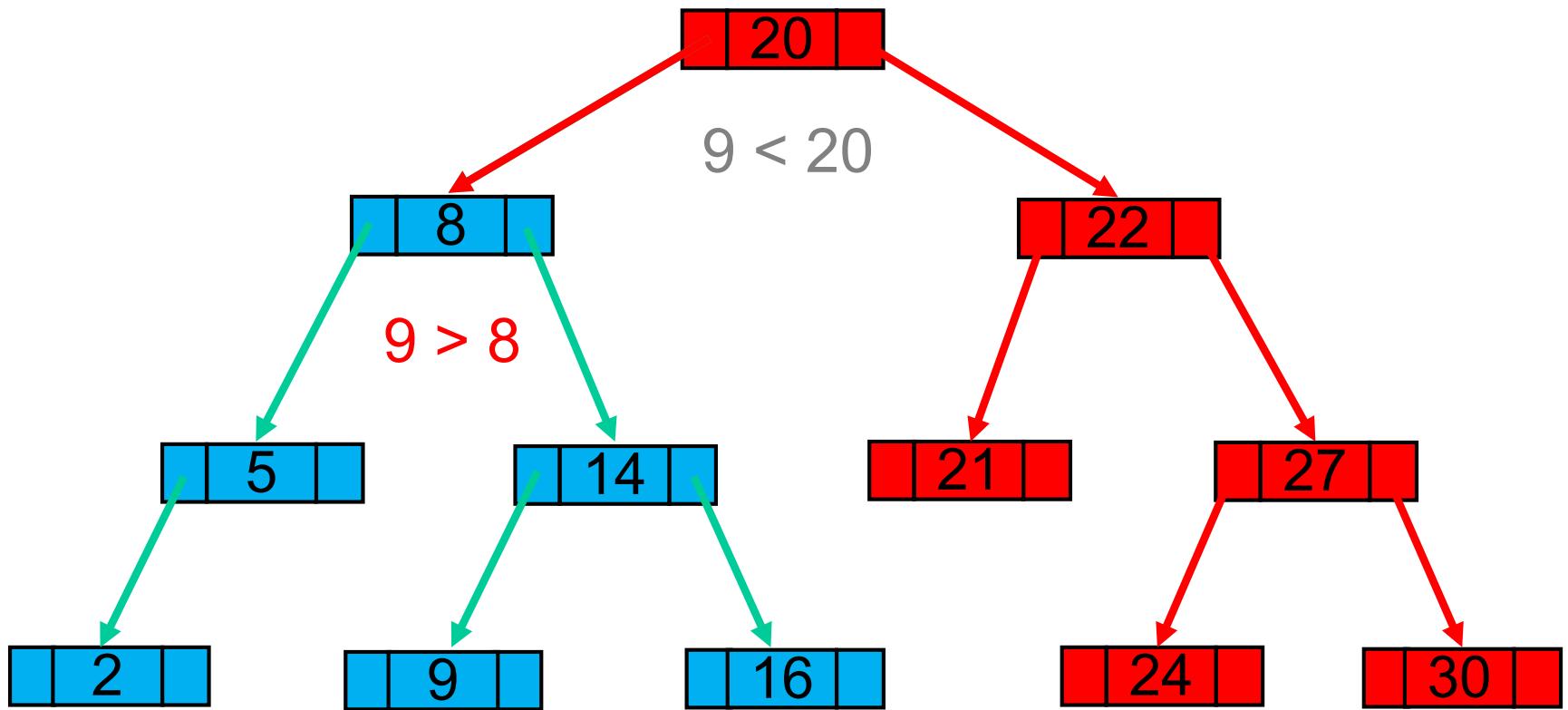


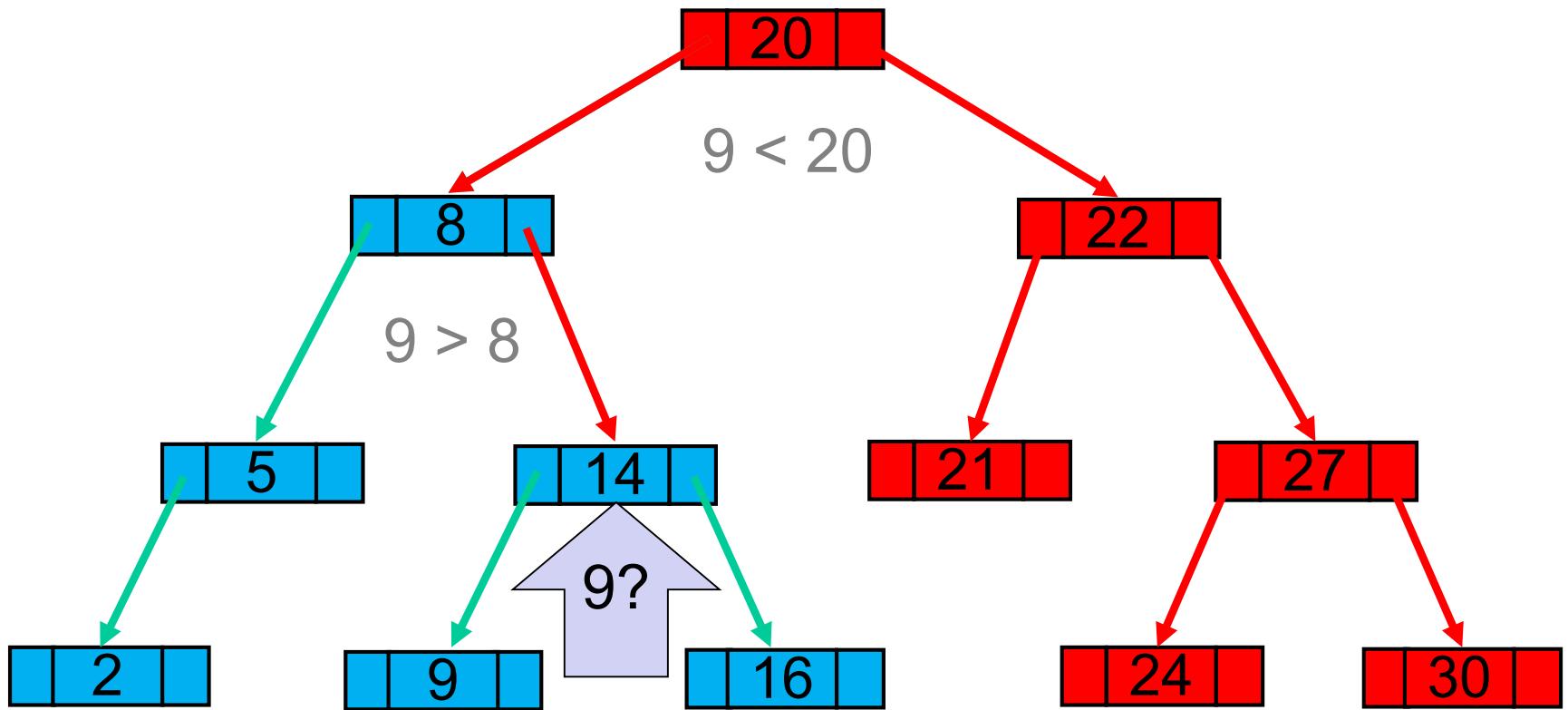


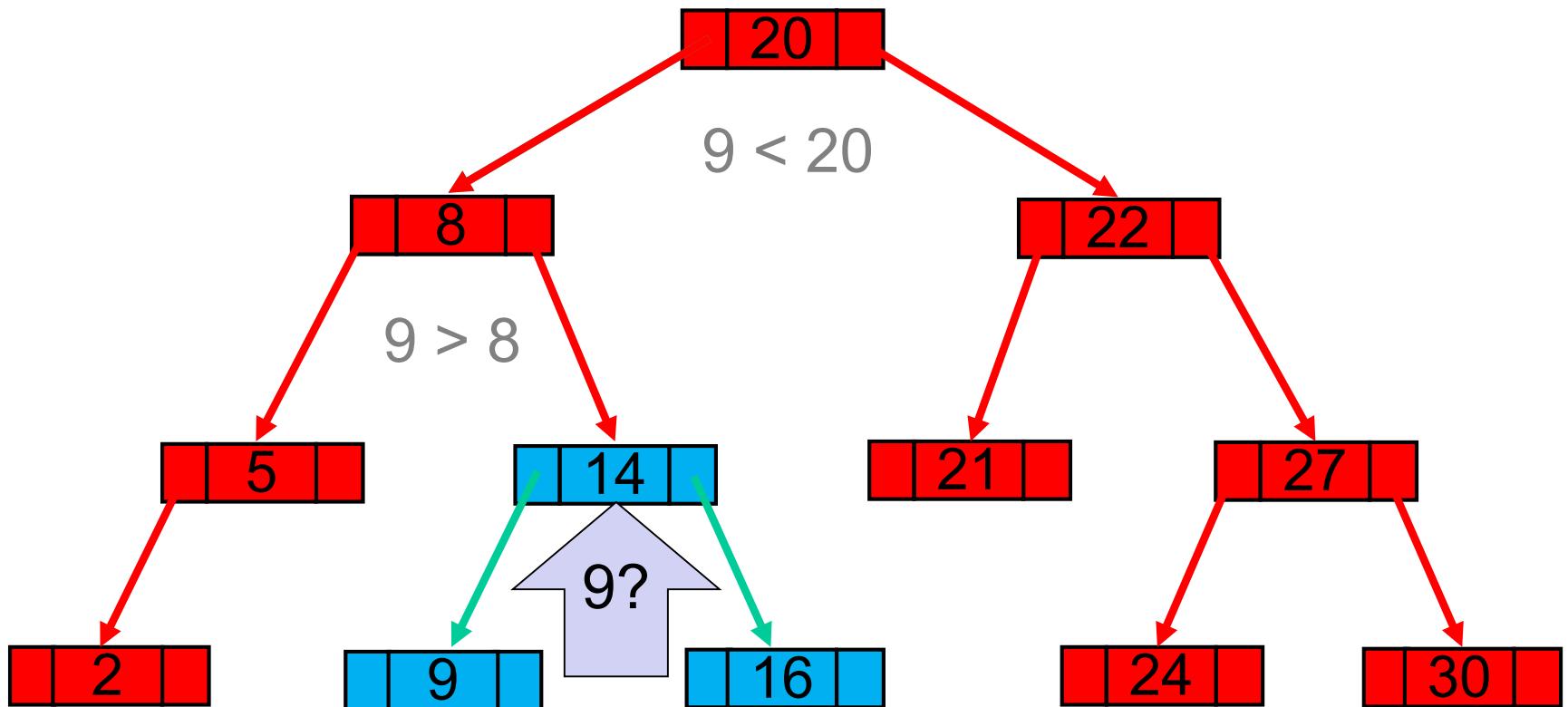


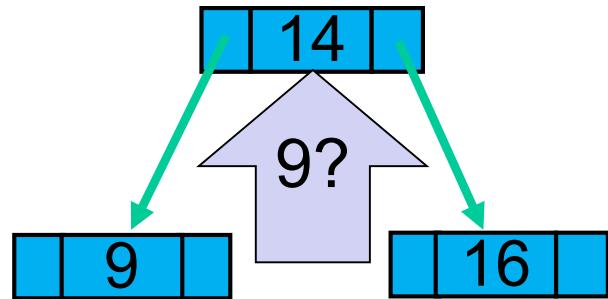


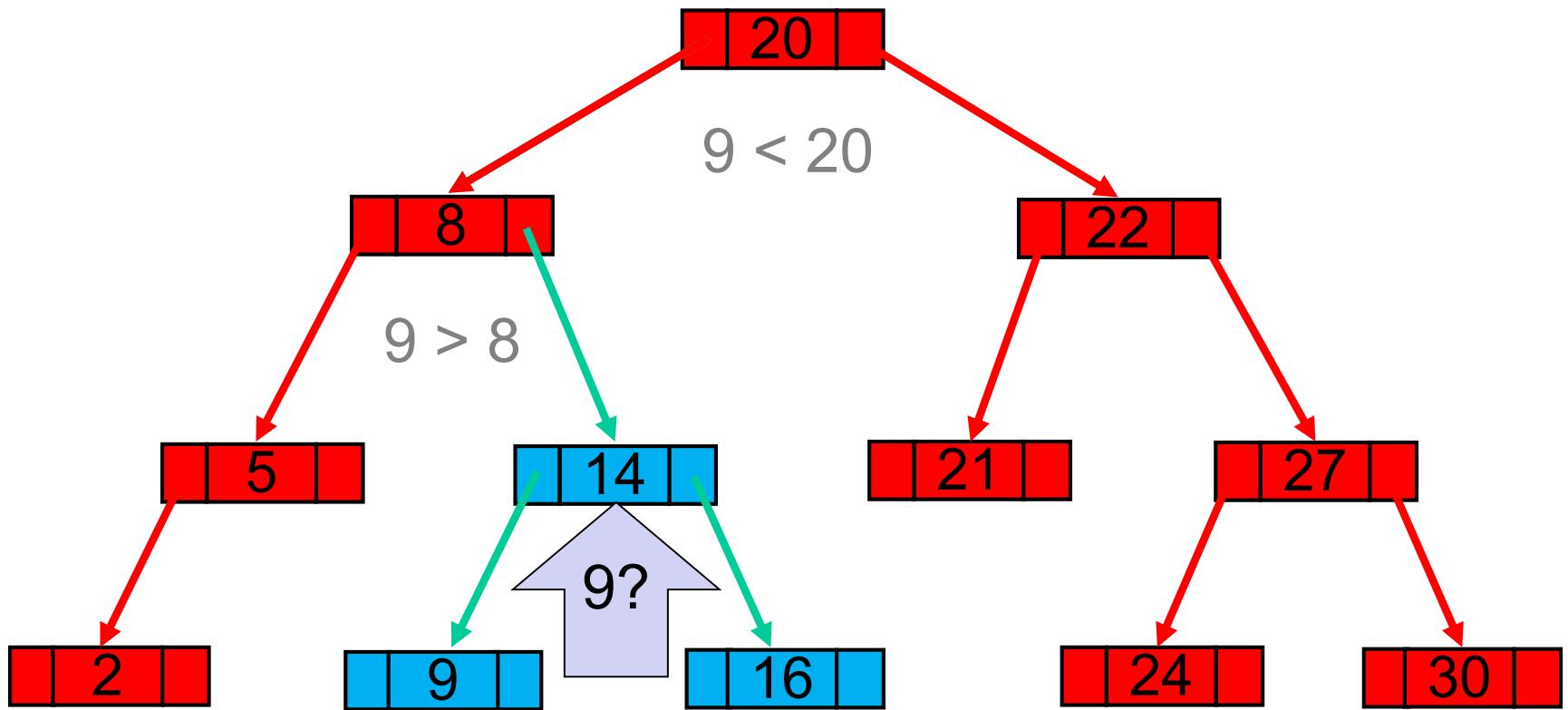


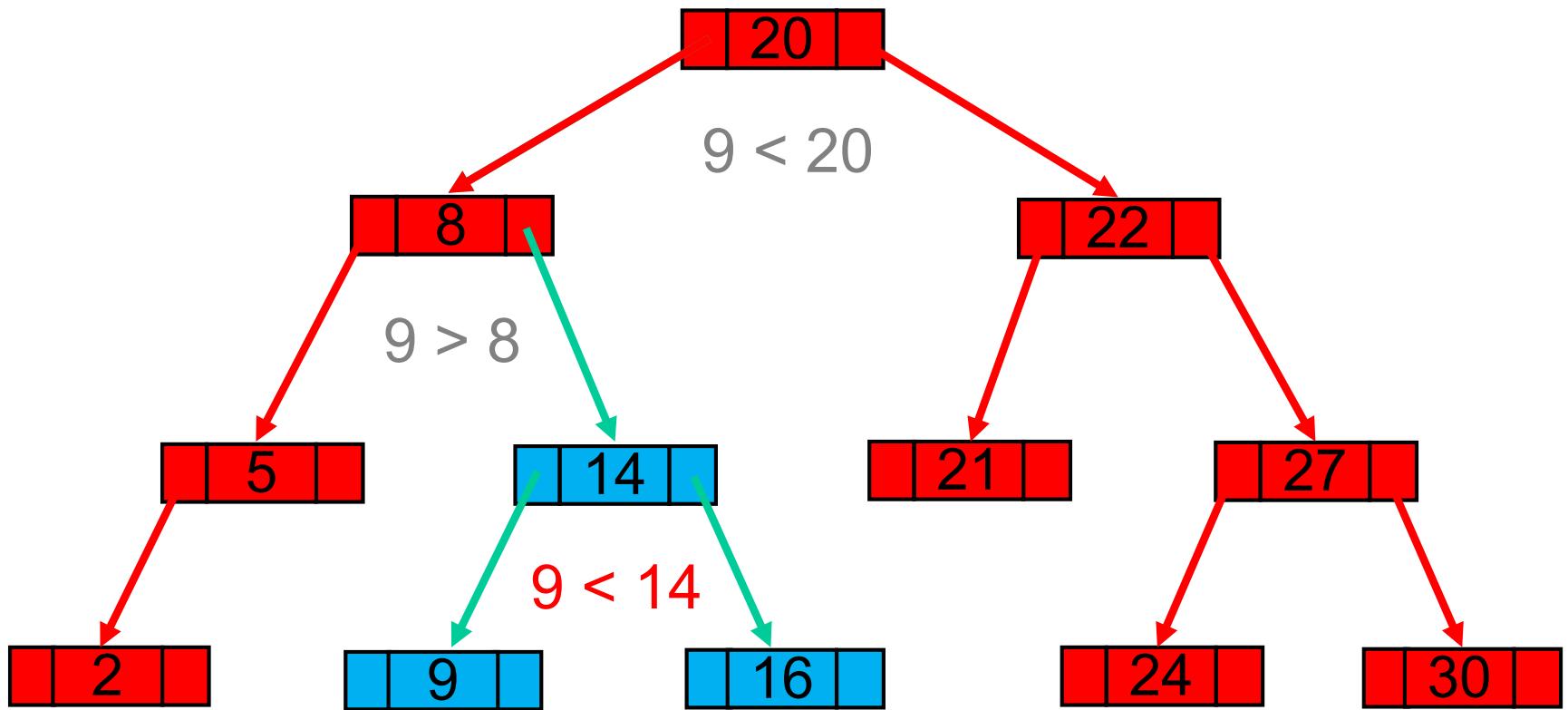


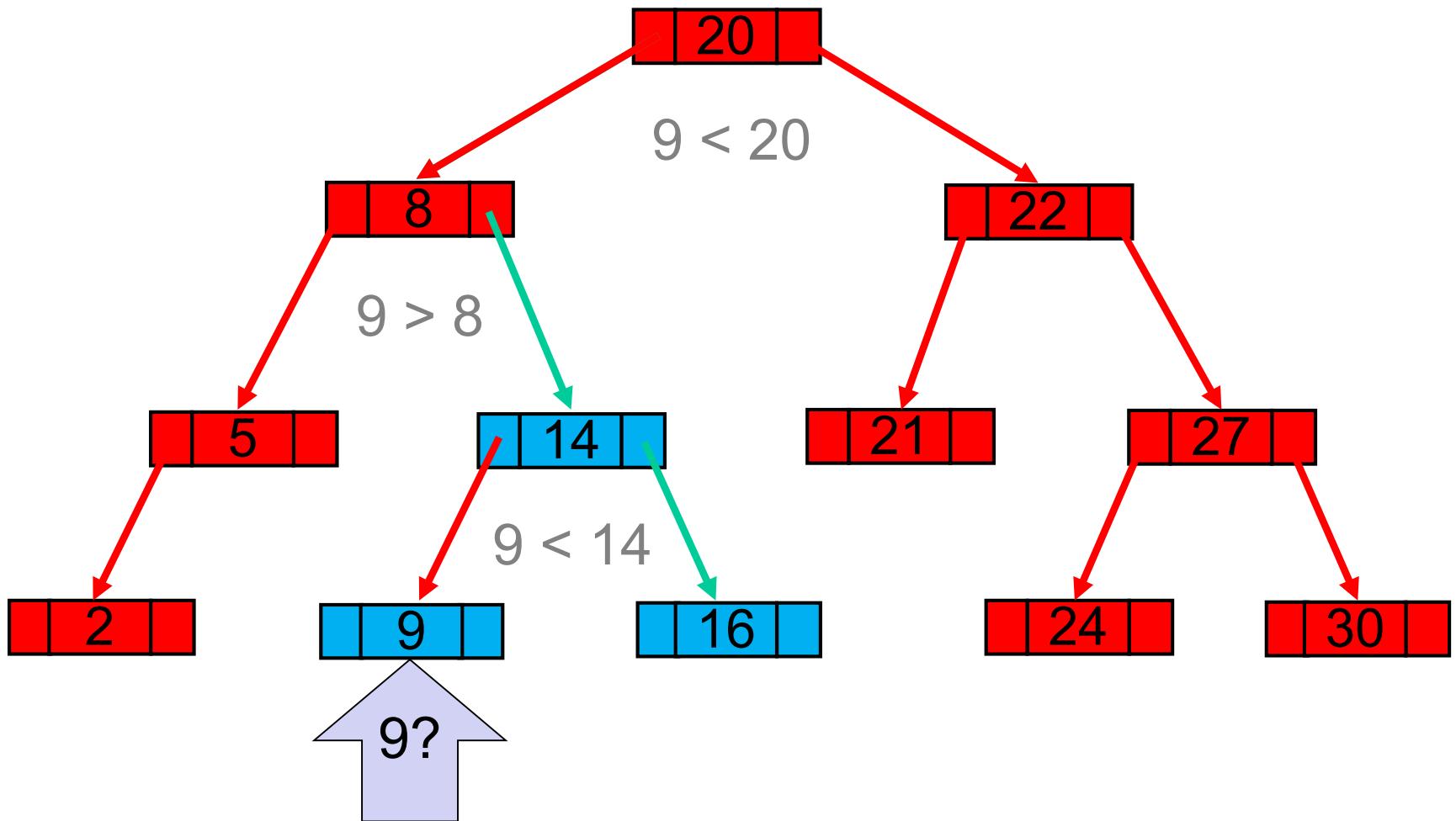


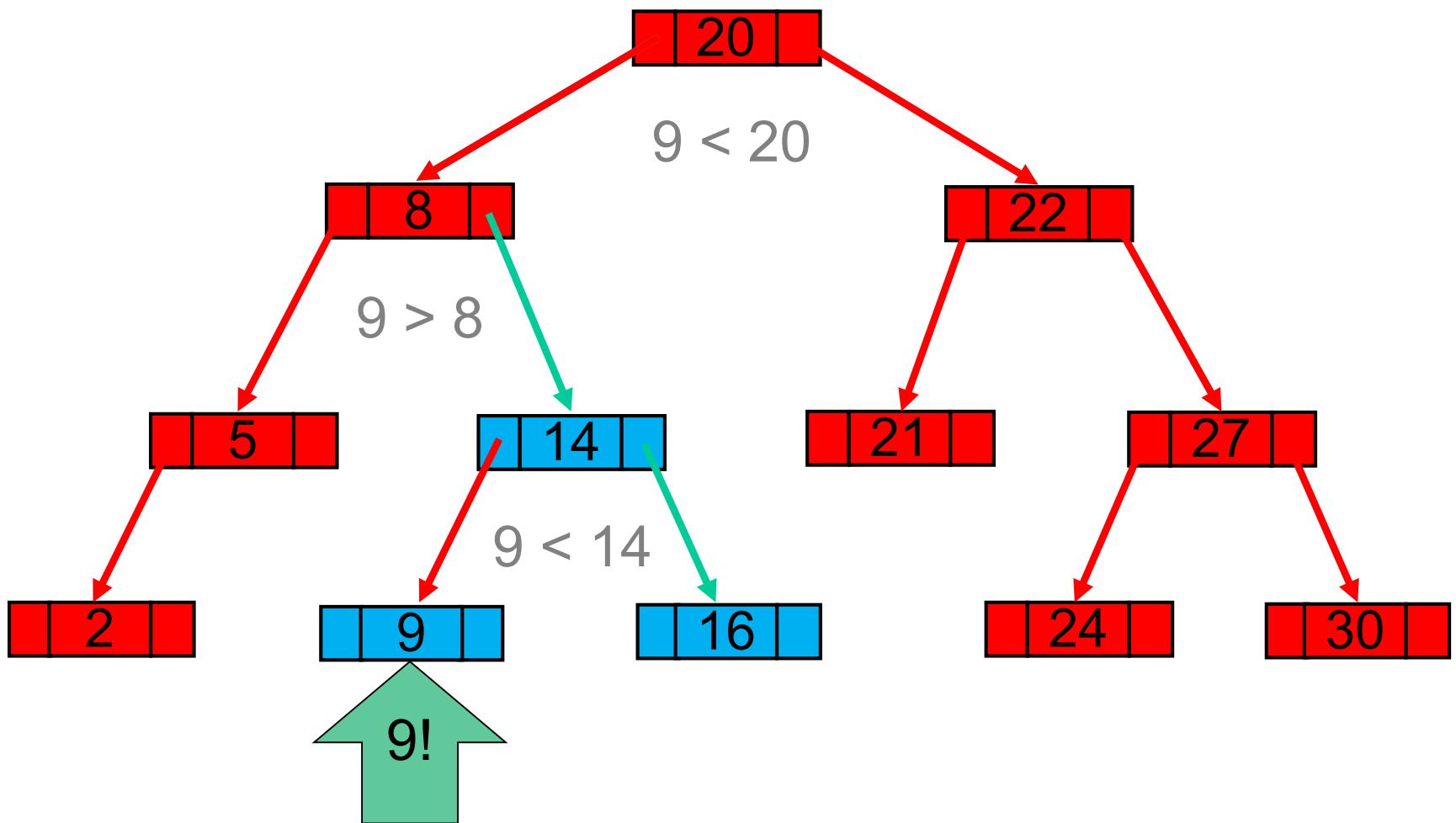












# Analysis of Binary Search Tree

---

- On each step, we reduce the number of elements we need to consider by half
  - That is, we first need to consider the whole tree
  - Then one of the subtrees, i.e. half the elements
  - Then one of **its** subtrees, i.e. half the **remaining** elements
  - And so on
- 
- **This is  $O(\log_2 n)$**

# Are these operations *always* $O(\log_2 n)$ ?

20

A diagram illustrating a linked list update. At the top, a blue rectangular box contains the number '20' in white. This box is part of a horizontal sequence of three blue boxes. A green arrow points from the right side of the '20' box down to another identical blue box at the bottom, which also contains the number '20'. This second box is also part of a similar horizontal sequence of three blue boxes. The green arrow originates from the right edge of the top '20' box and points directly to the right edge of the bottom '20' box.

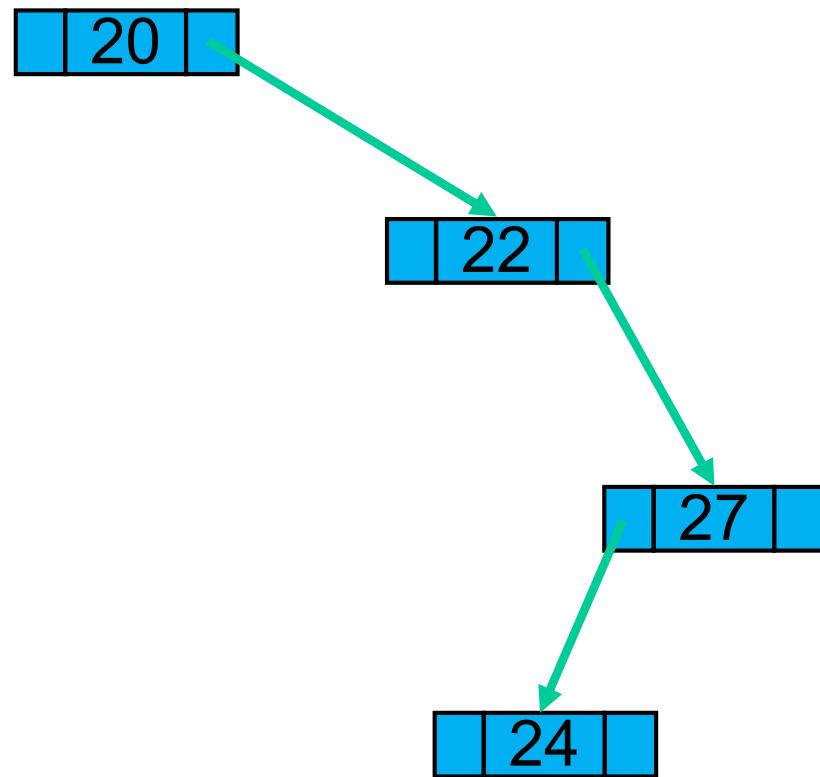
22

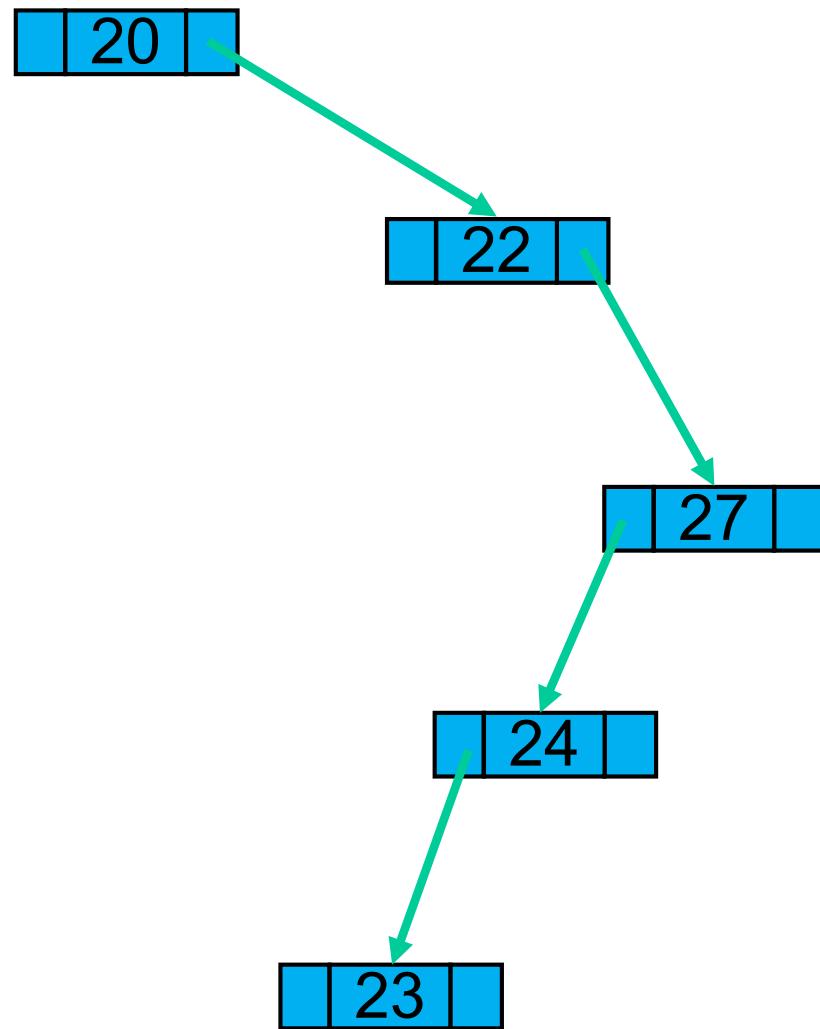
```
graph TD; N1[20] --> N2[22]; N2 --> N3[27]
```

20

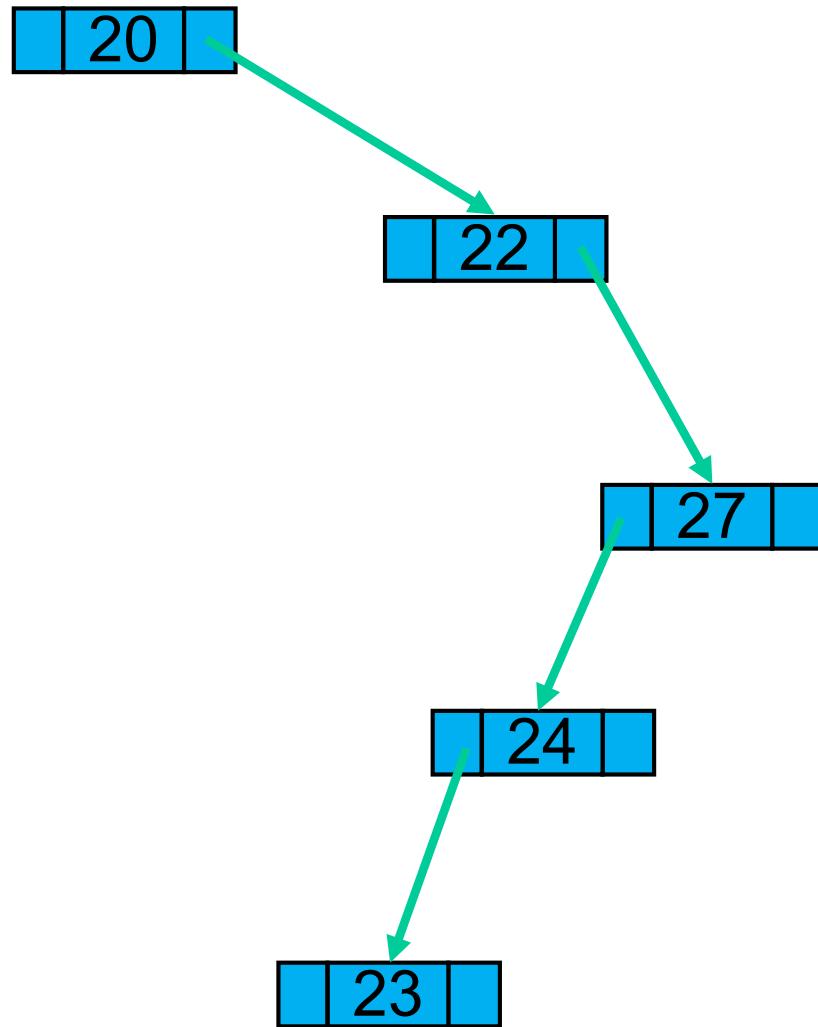
22

27

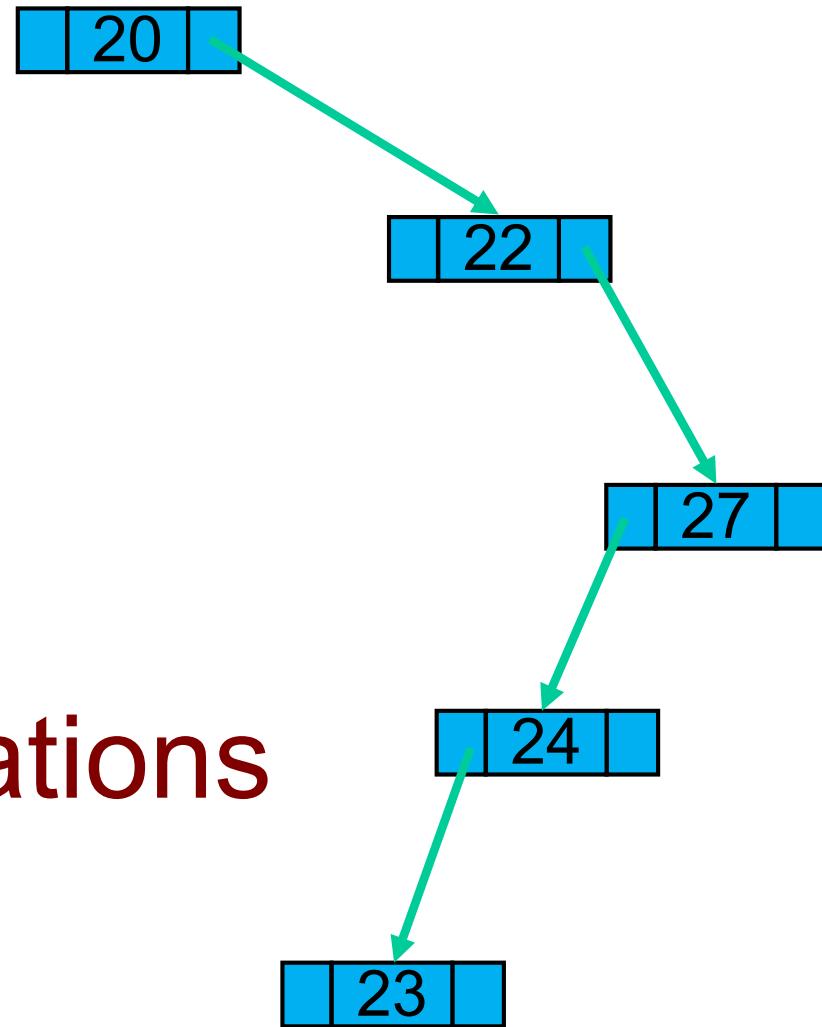




# This is just a Linked List!



This is just a  
Linked List!



Now the operations  
are  $O(n)$ .

**BAD!**

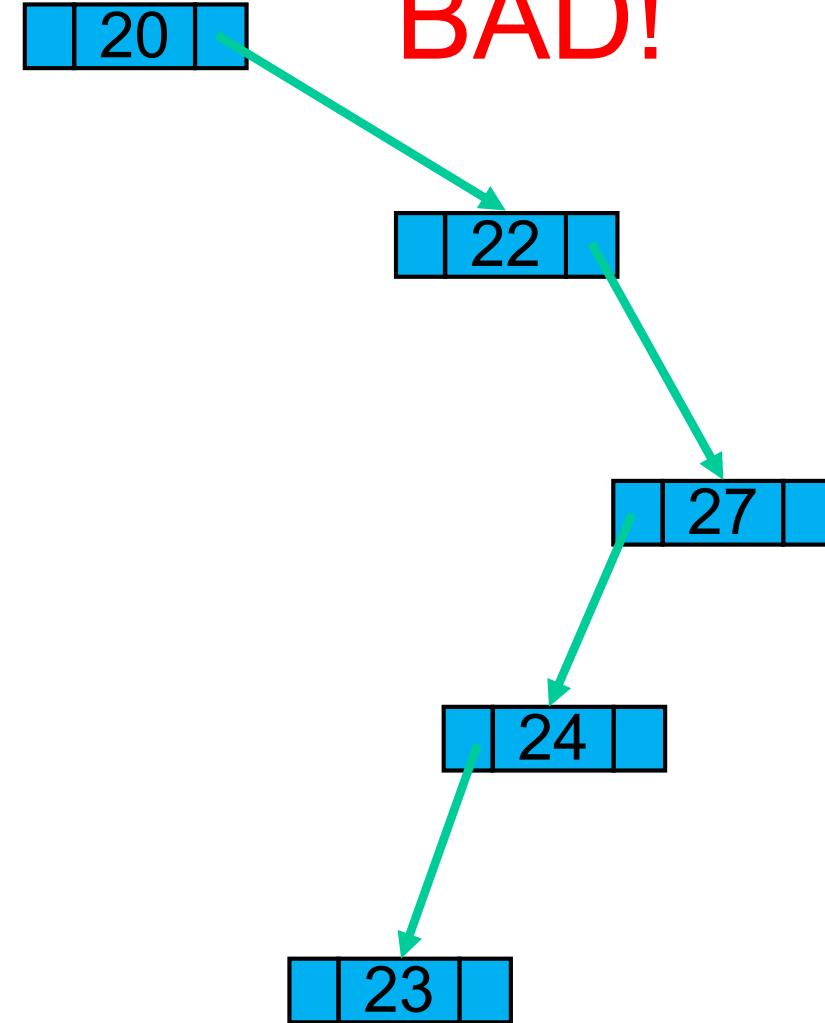
20

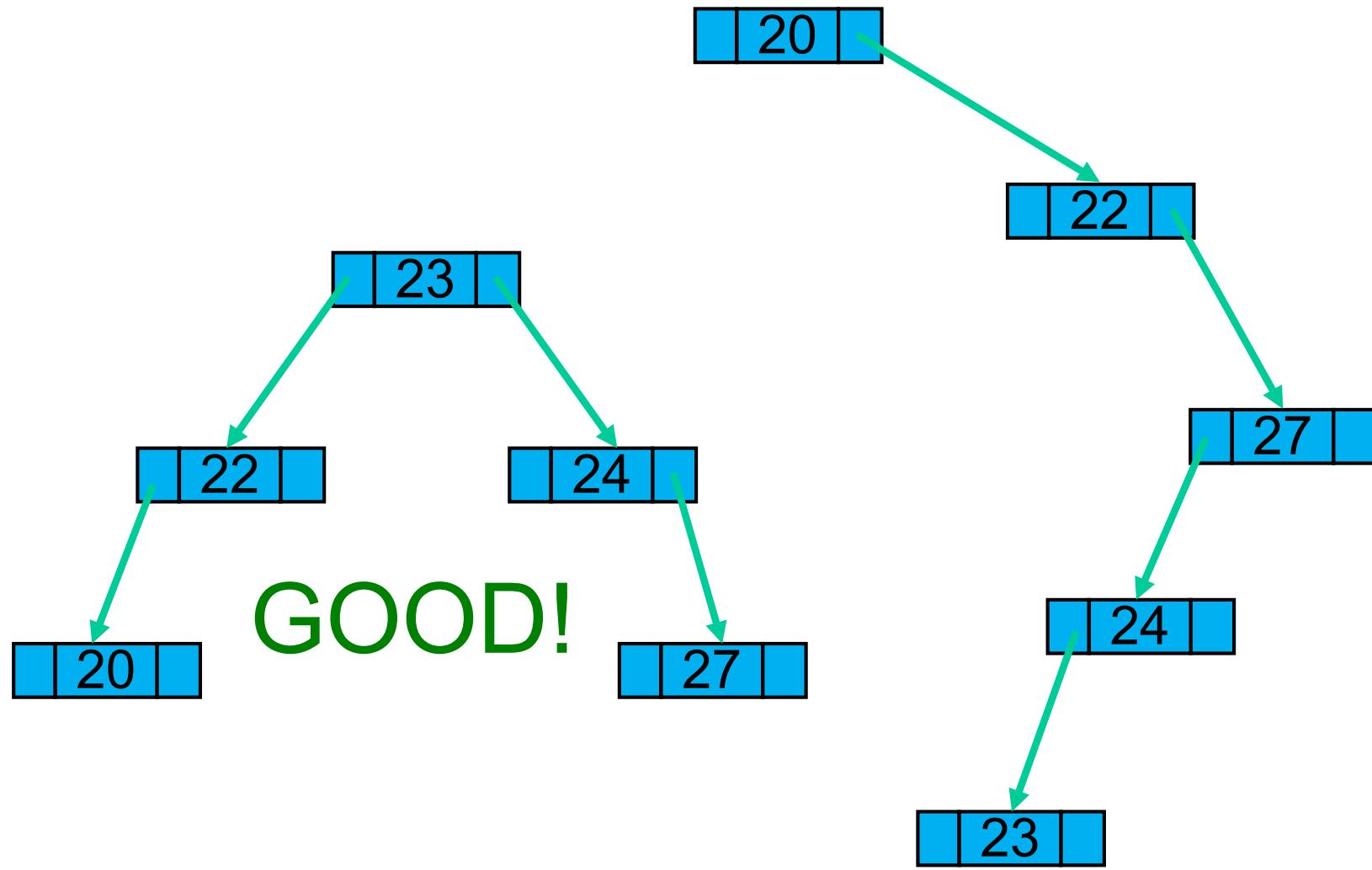
22

27

24

23

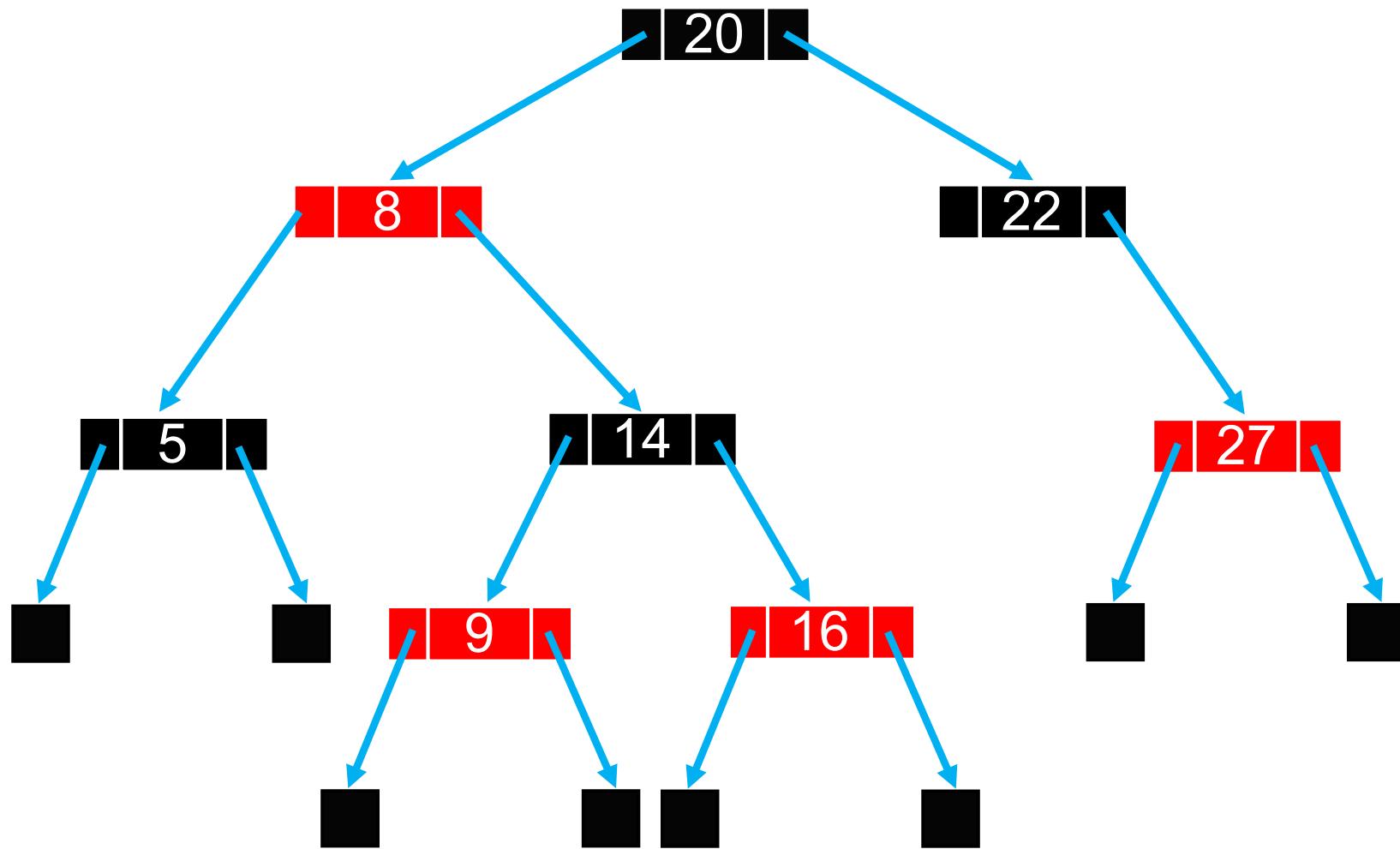




# Balanced Binary Search Tree

---

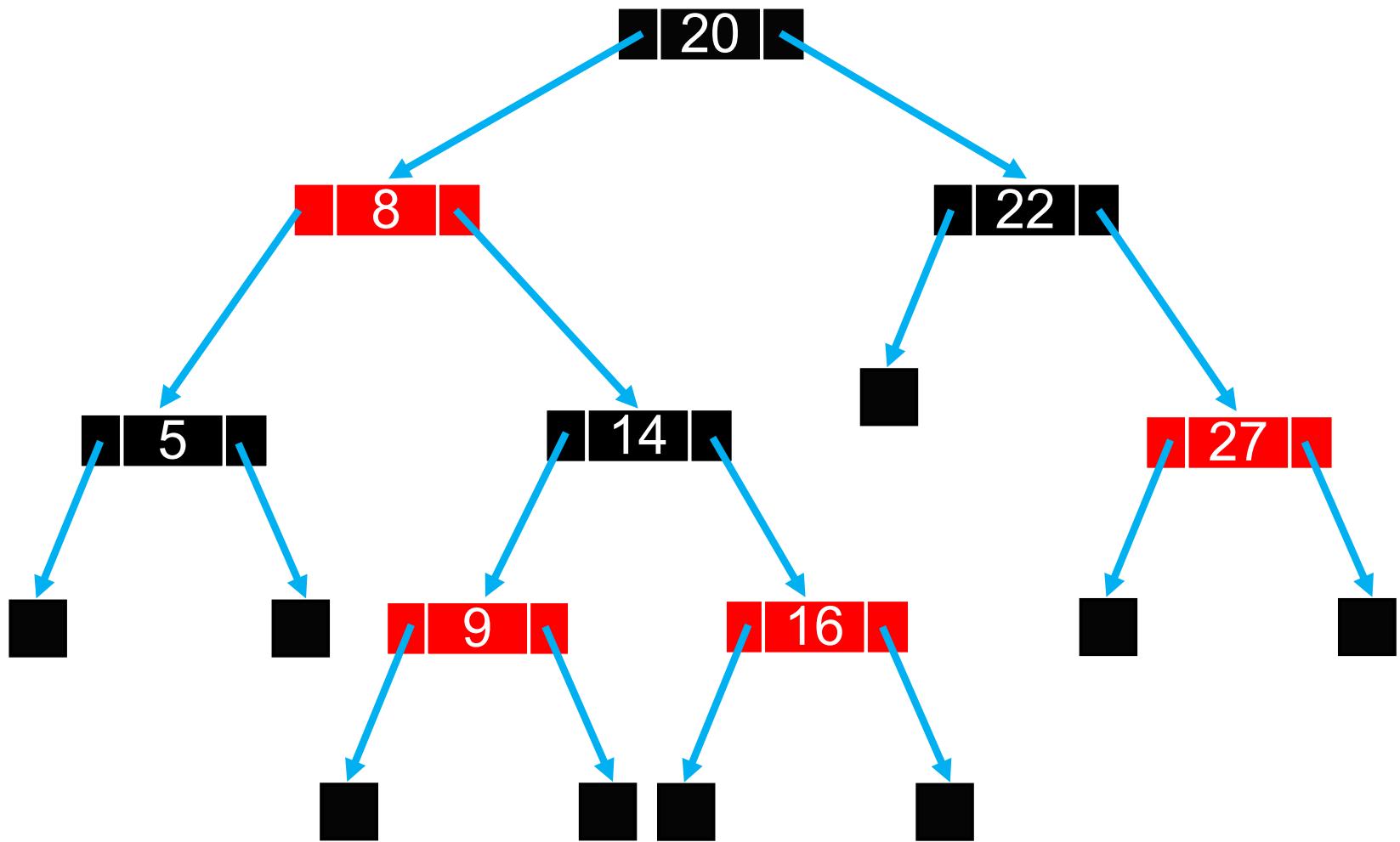
- Goal: minimize the **height** of the tree
- This can be done by making it **balanced**
- The search, add, and remove operations are  $O(\log_2 n)$  assuming that the tree is balanced
- A **self-balancing** binary search tree is one that ensures it is still balanced when an element is added or removed

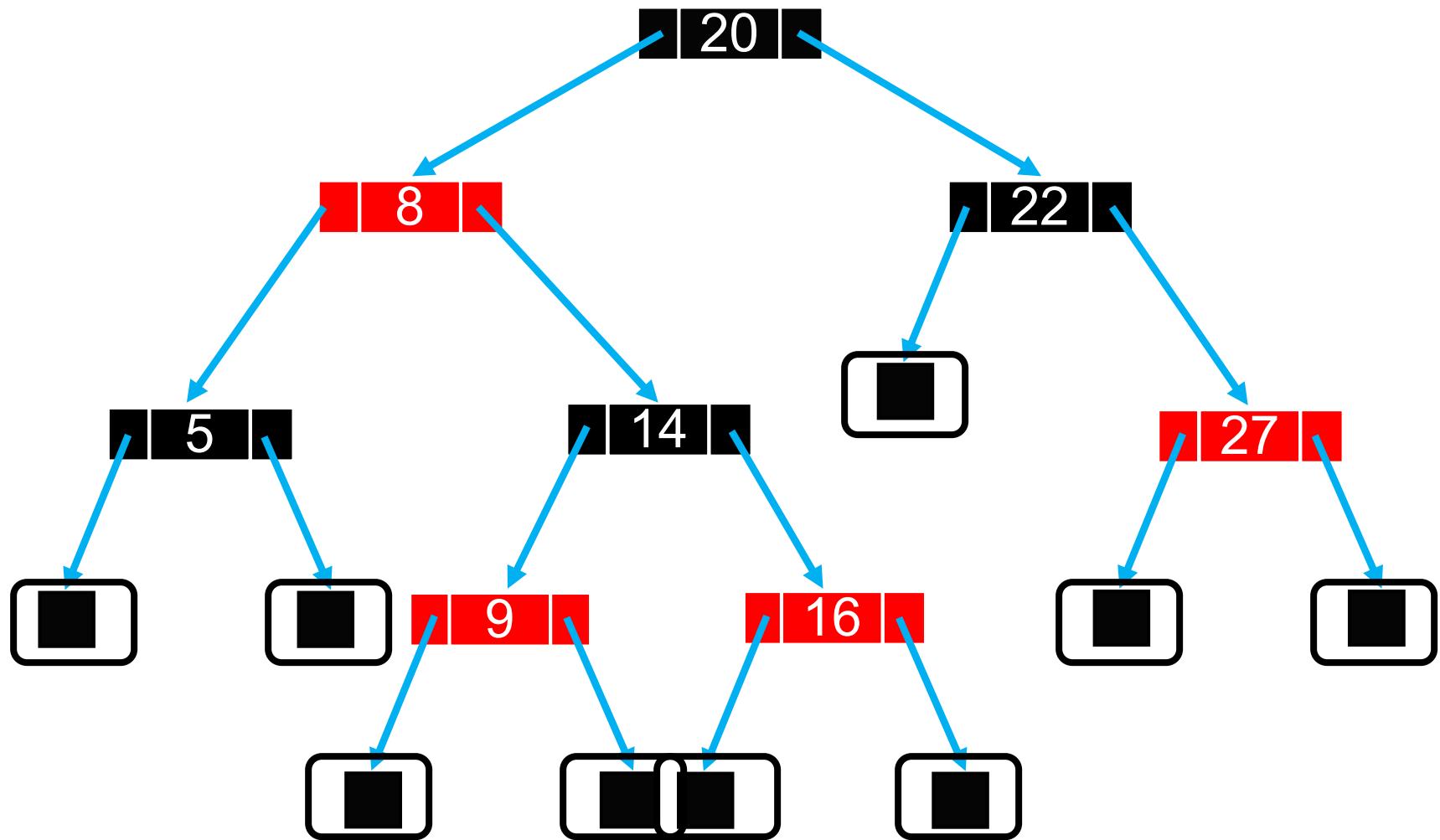


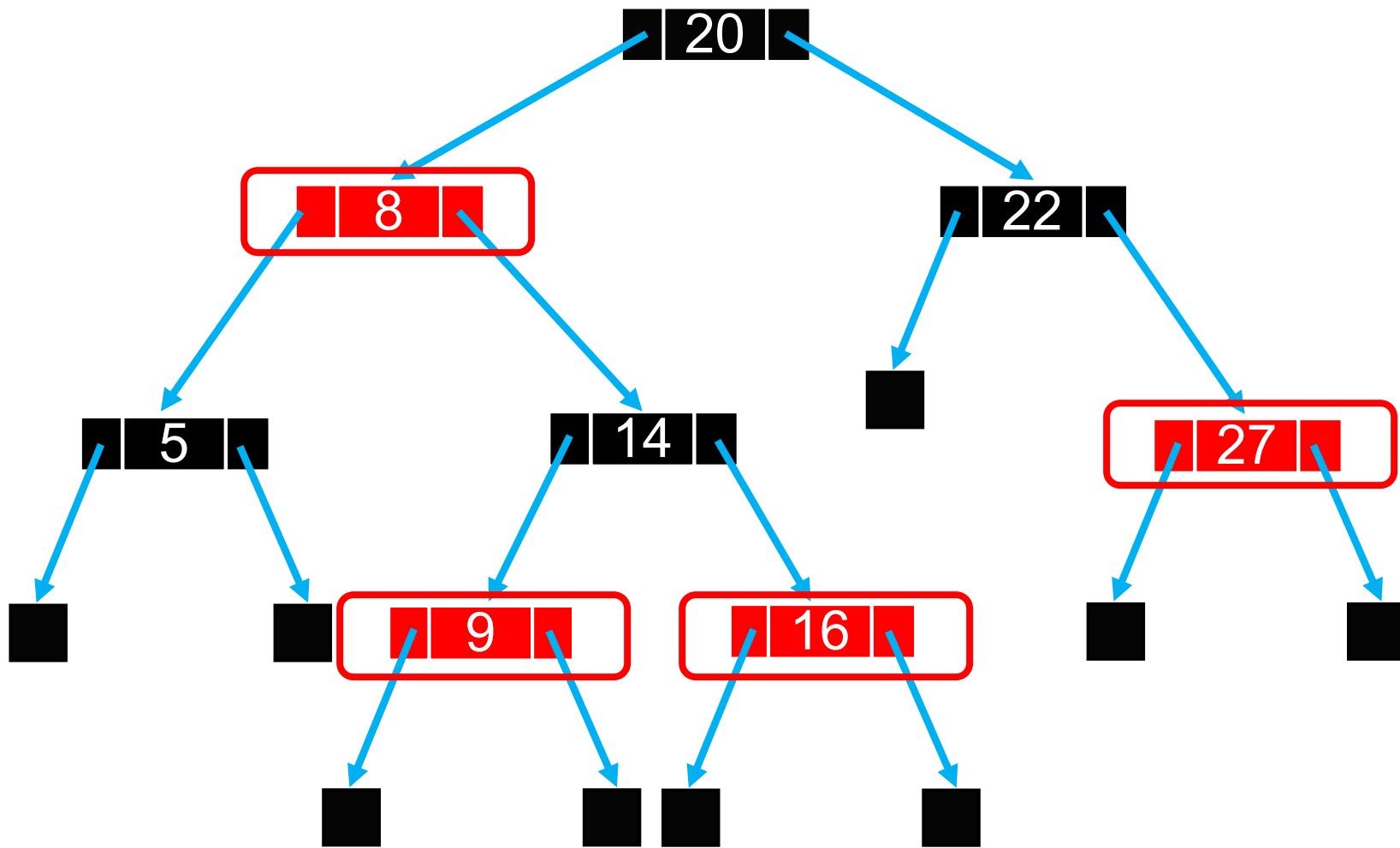
# Red-Black Tree

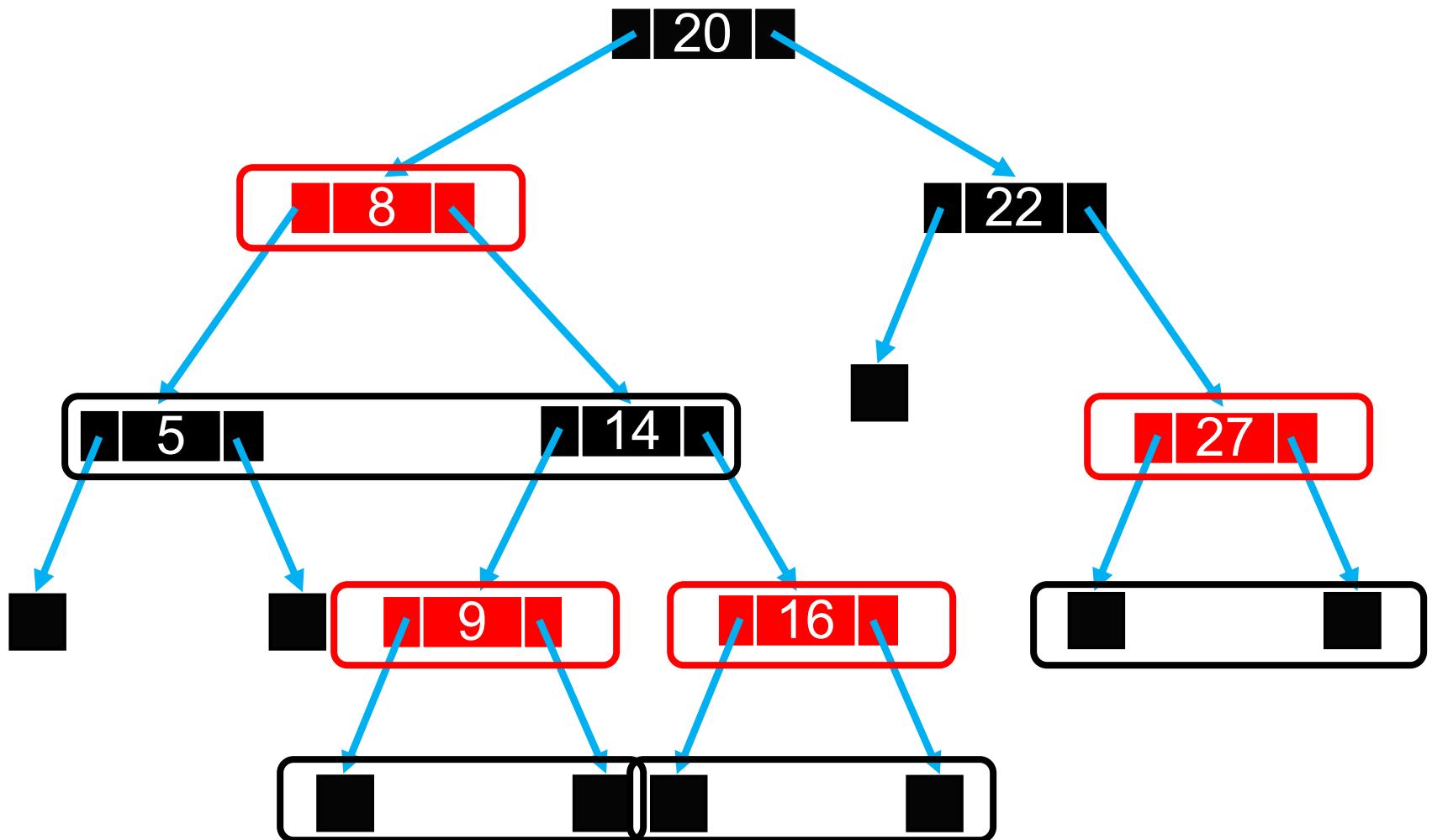
---

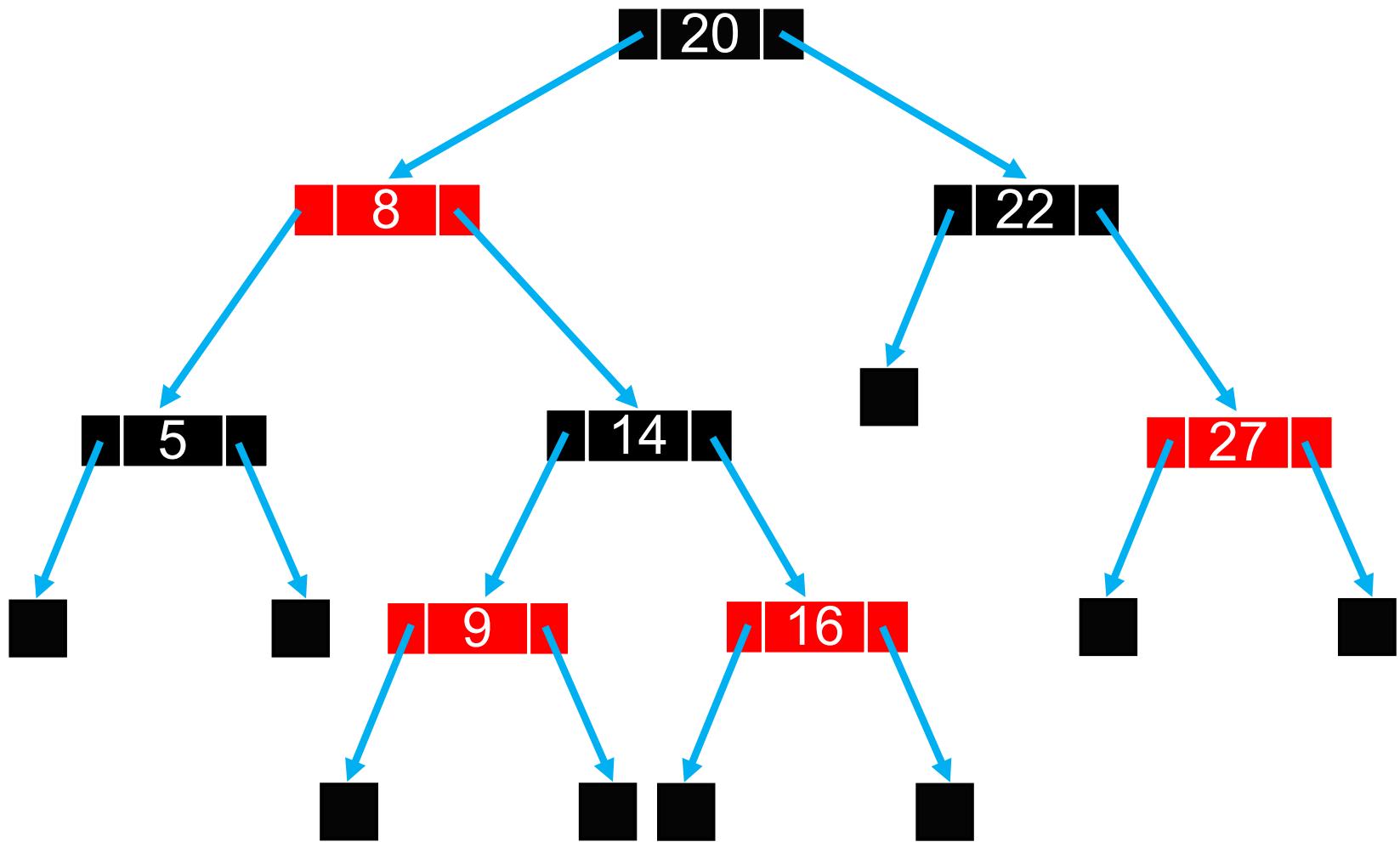
- A **red-black tree** is a self-balancing binary search tree
- Every node is colored either **red** or **black**
- Every leaf (NULL) node is **black**
- Every **red** node has 2 **black** child nodes
- Every path from a node to any leaf has the same number of black nodes (not counting the node itself)
  - This is called the **black height**

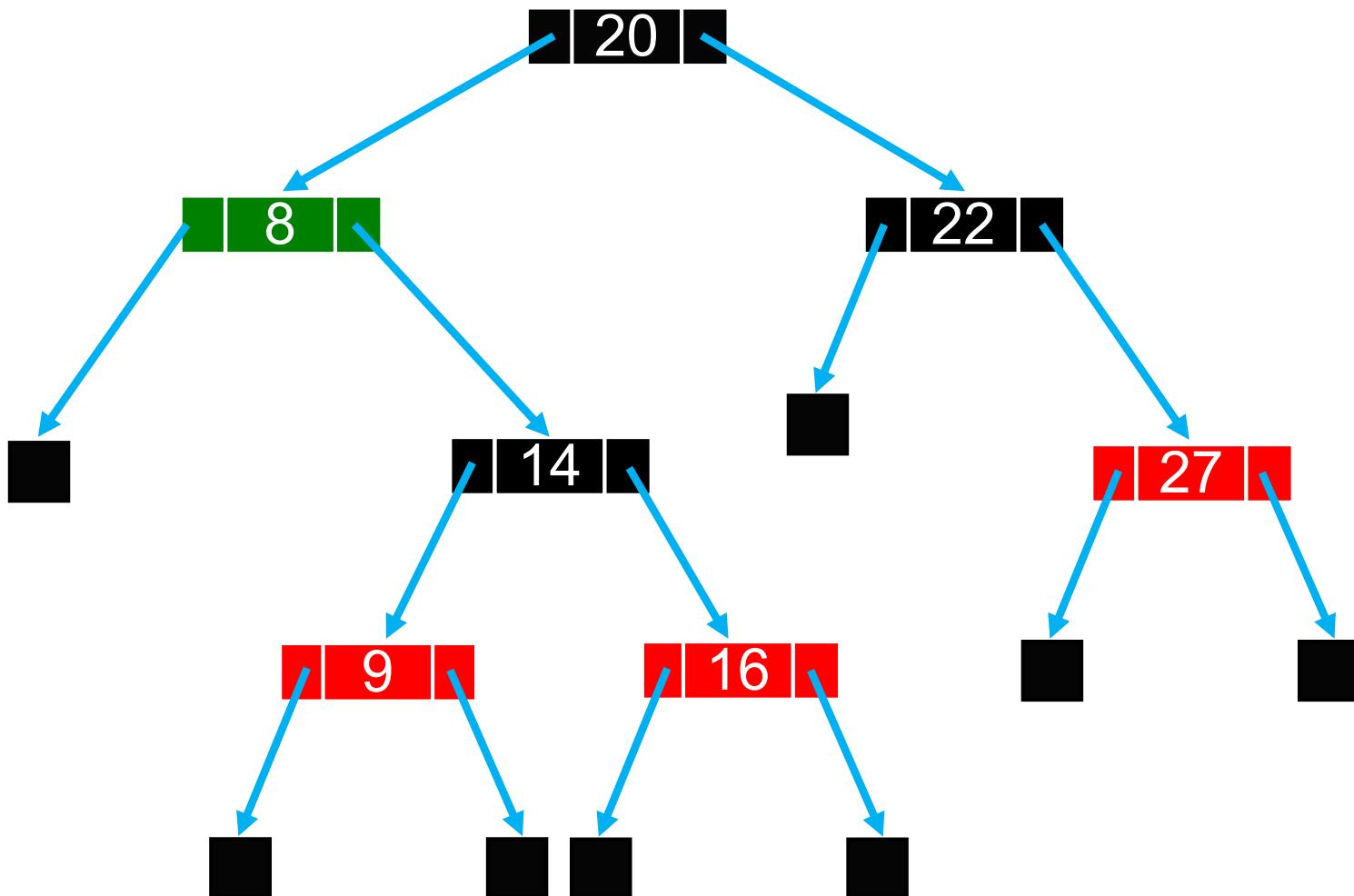


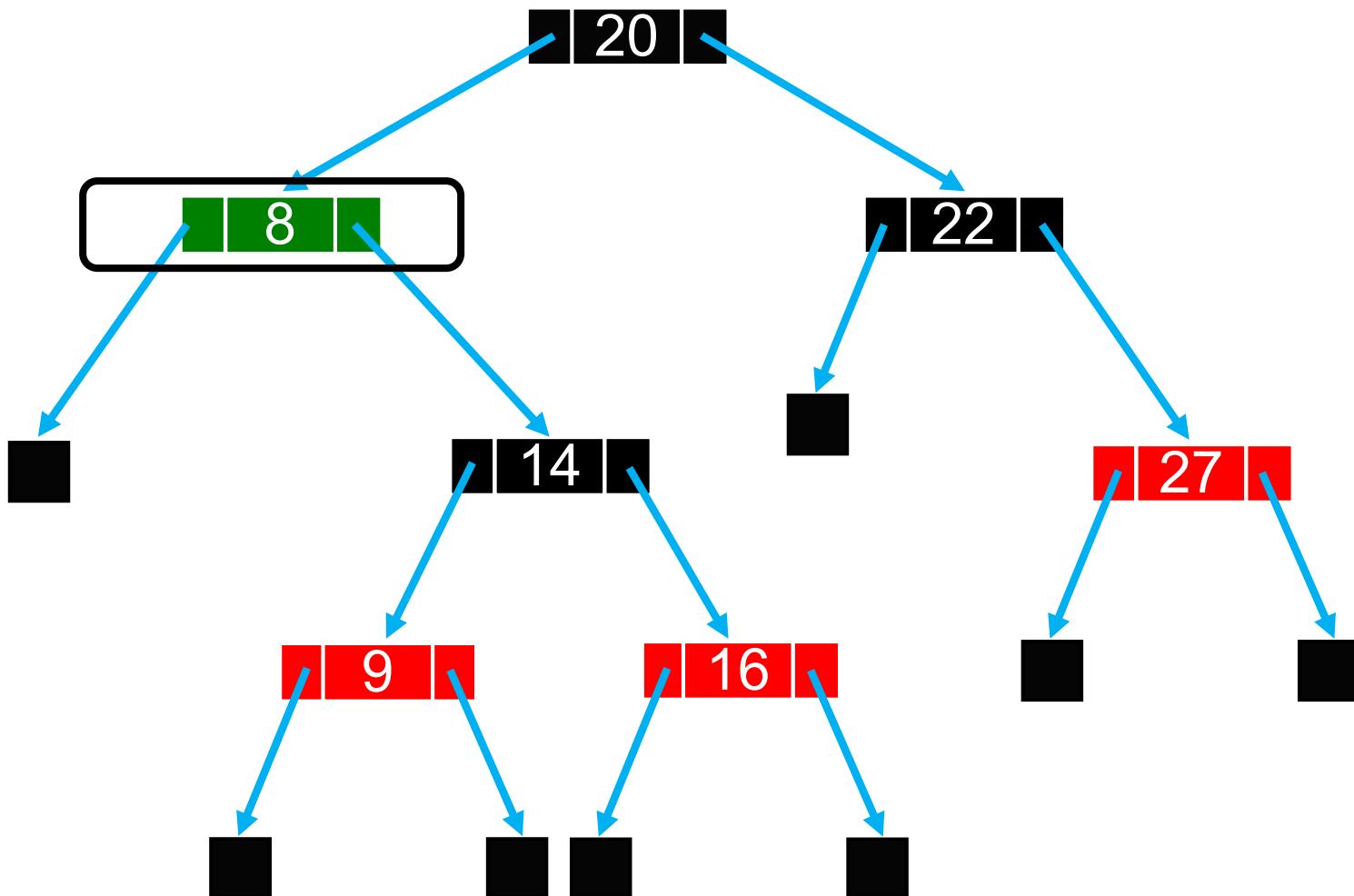


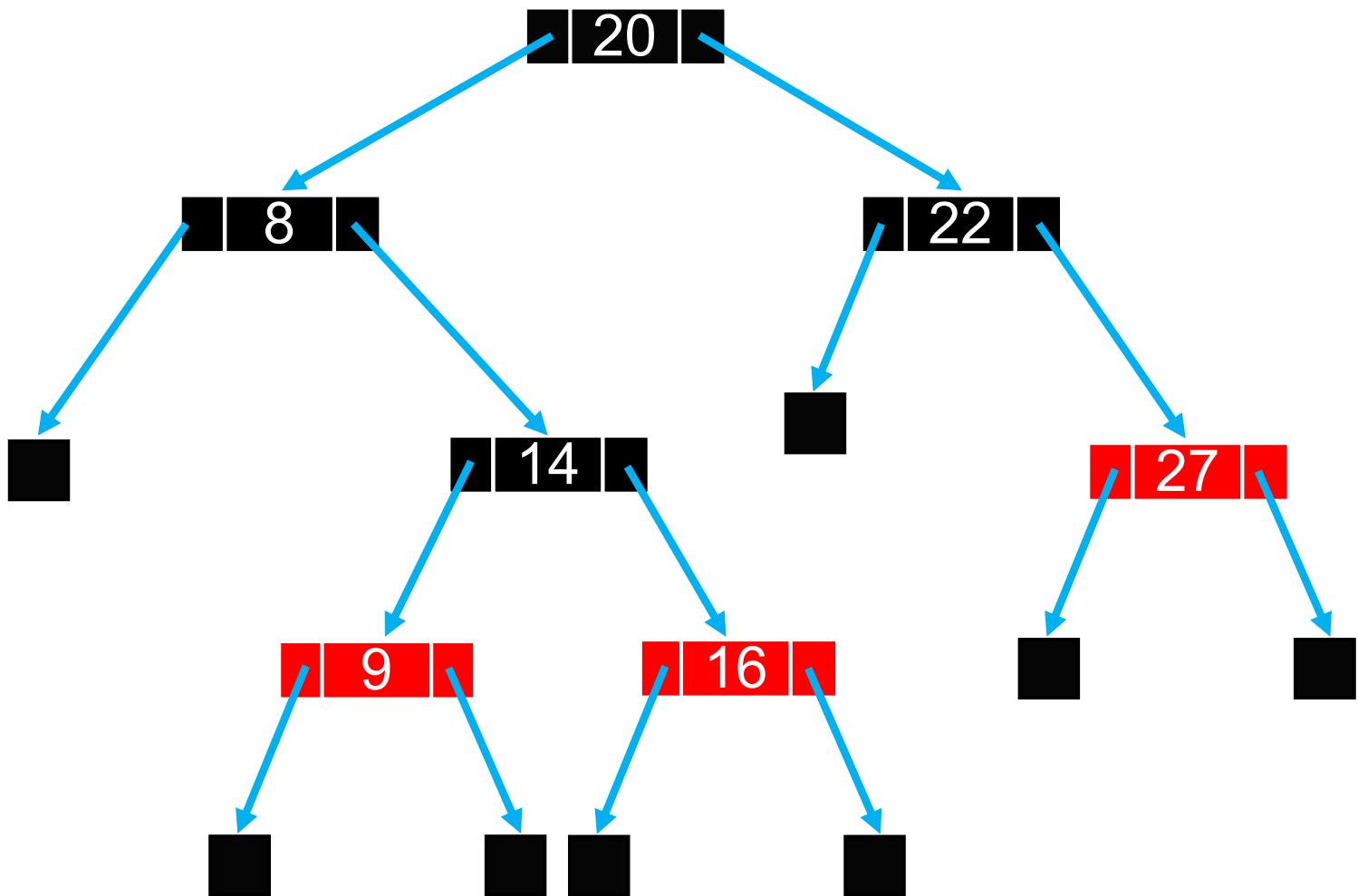


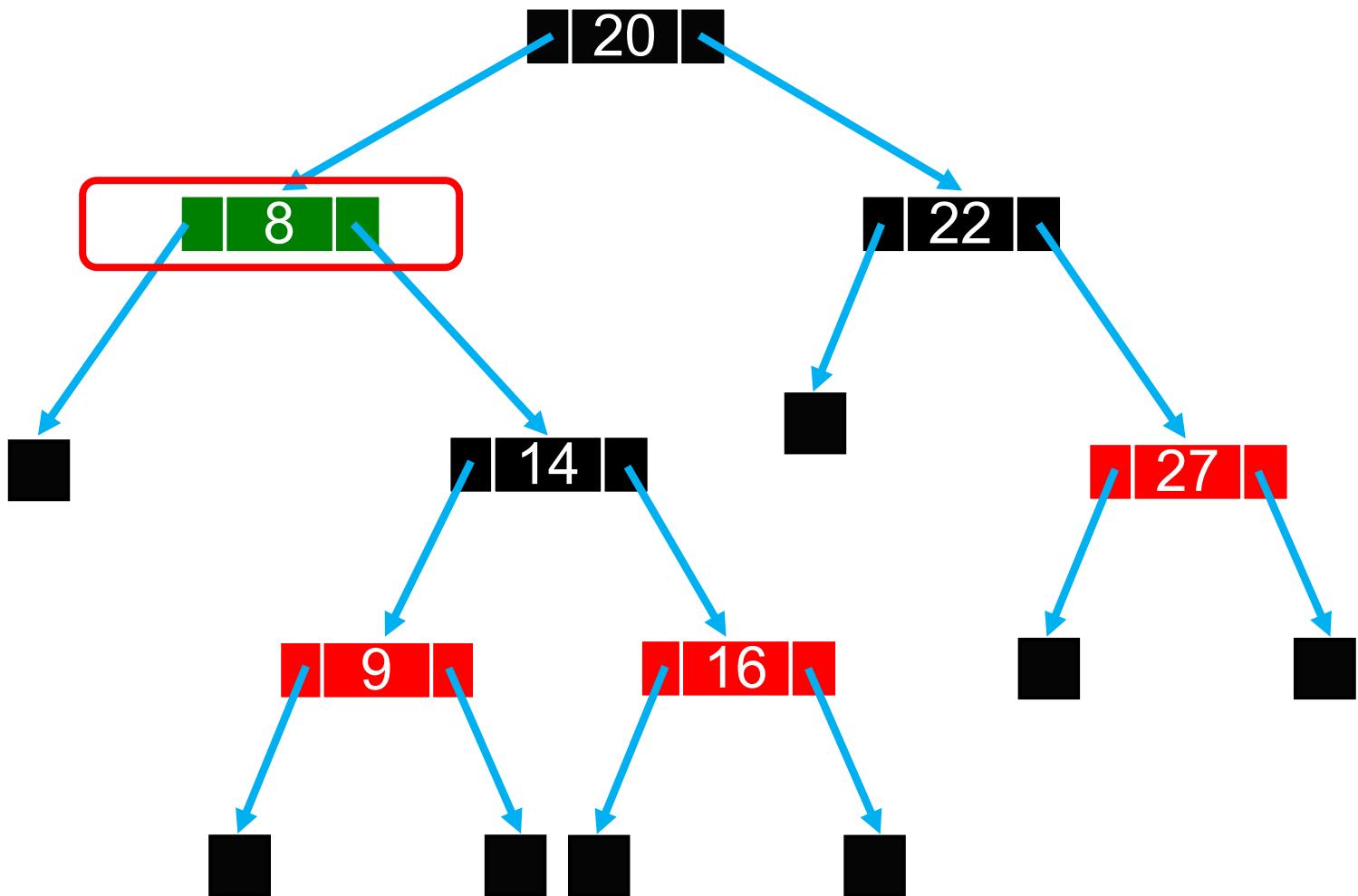


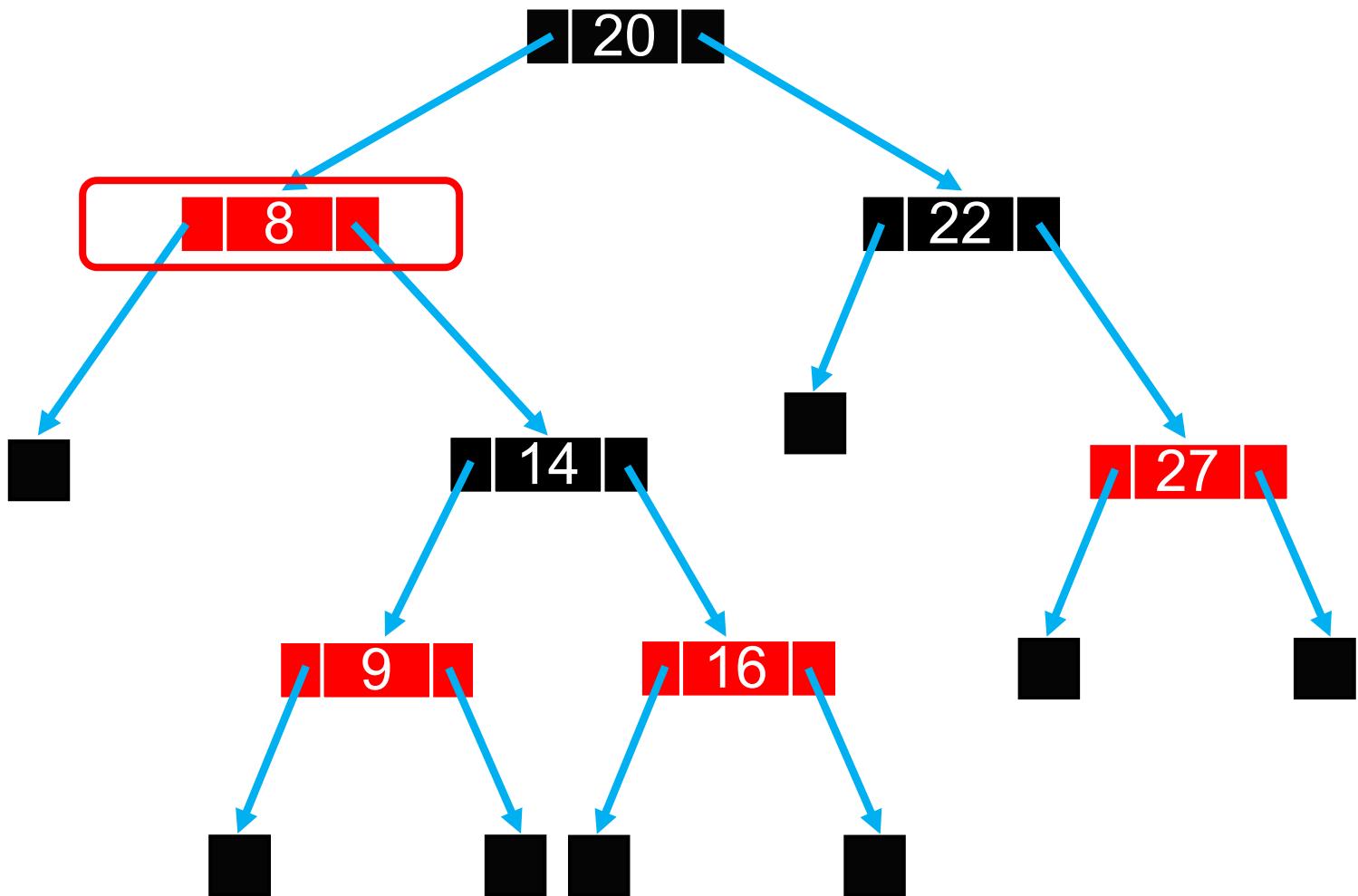


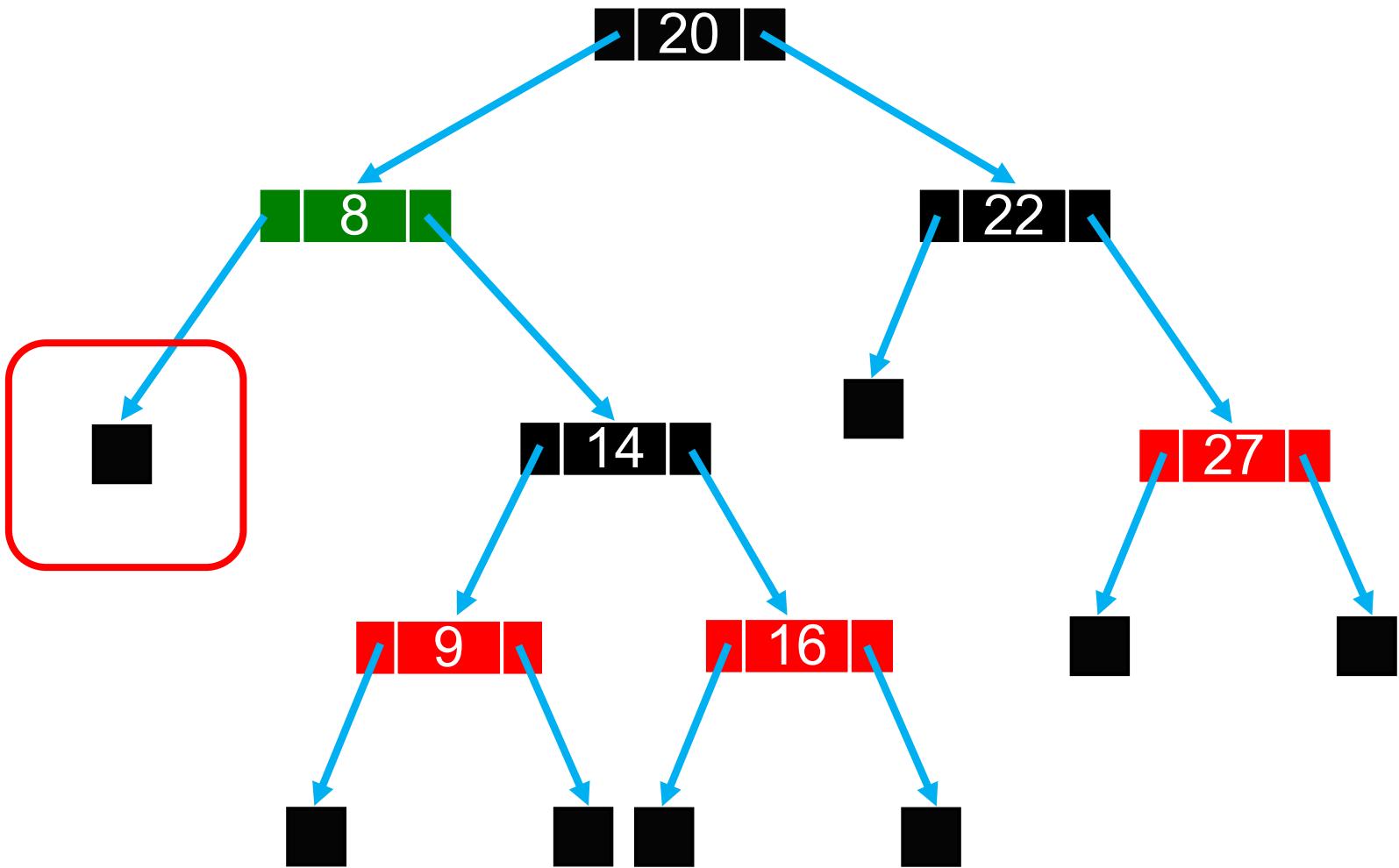




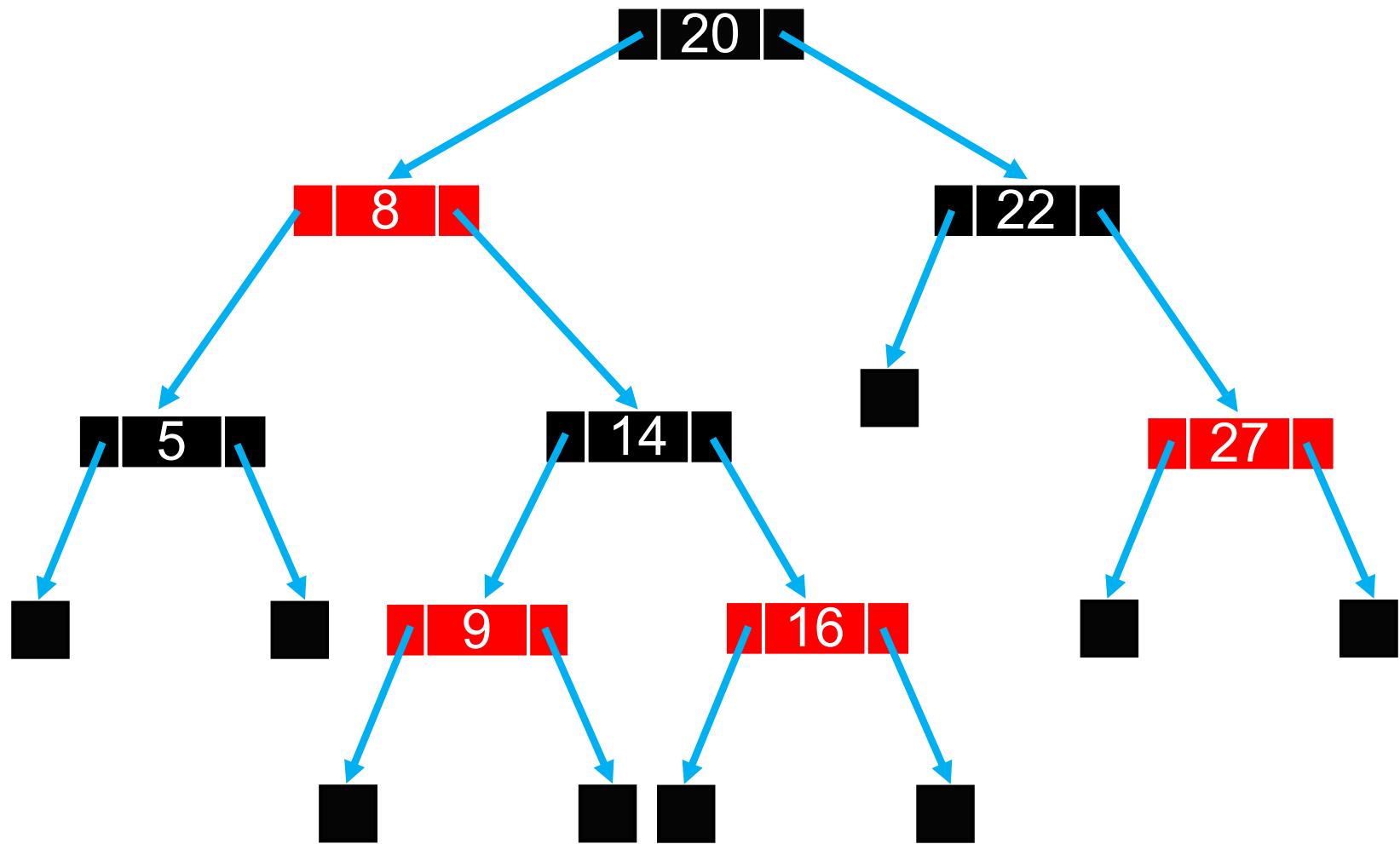




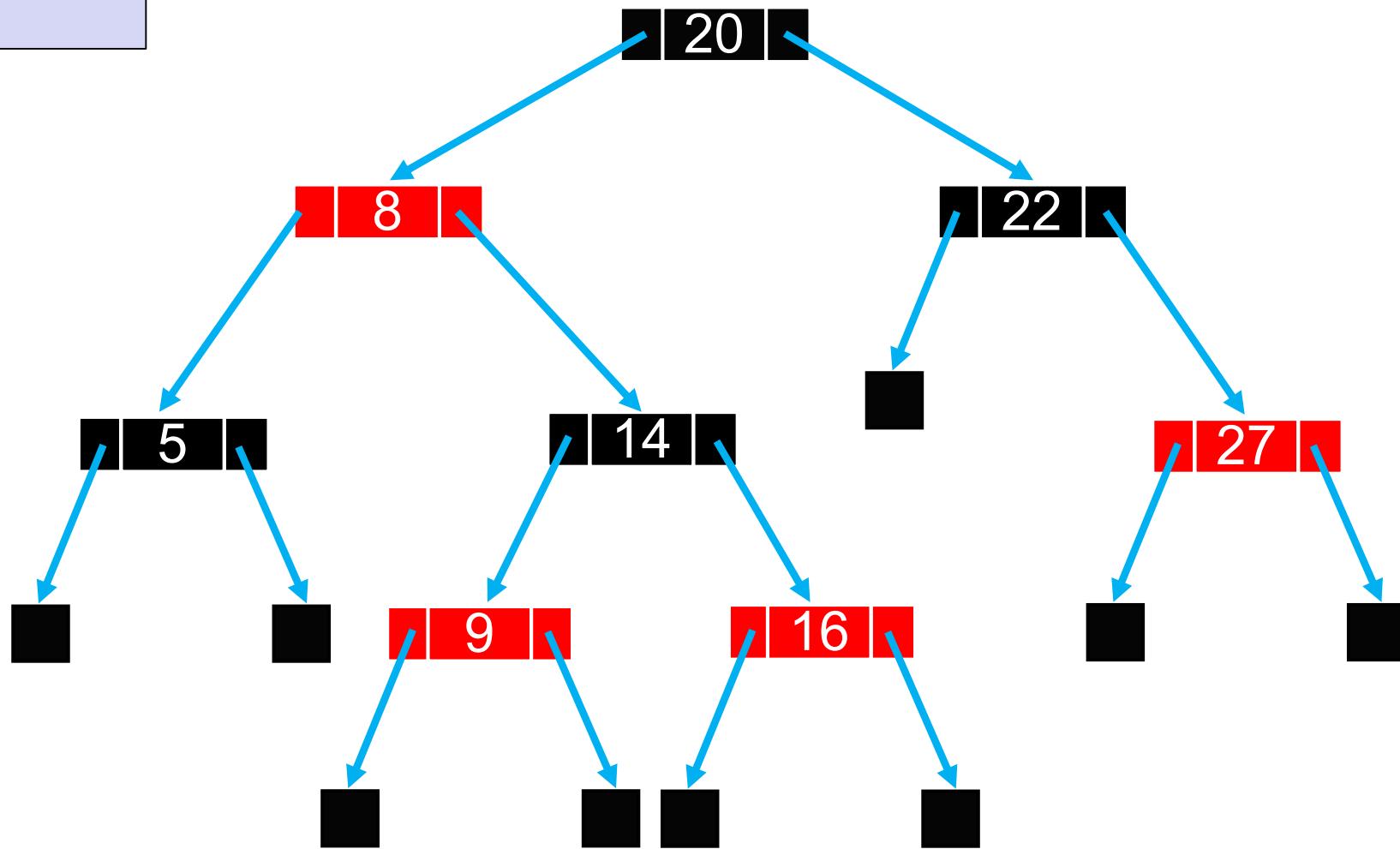




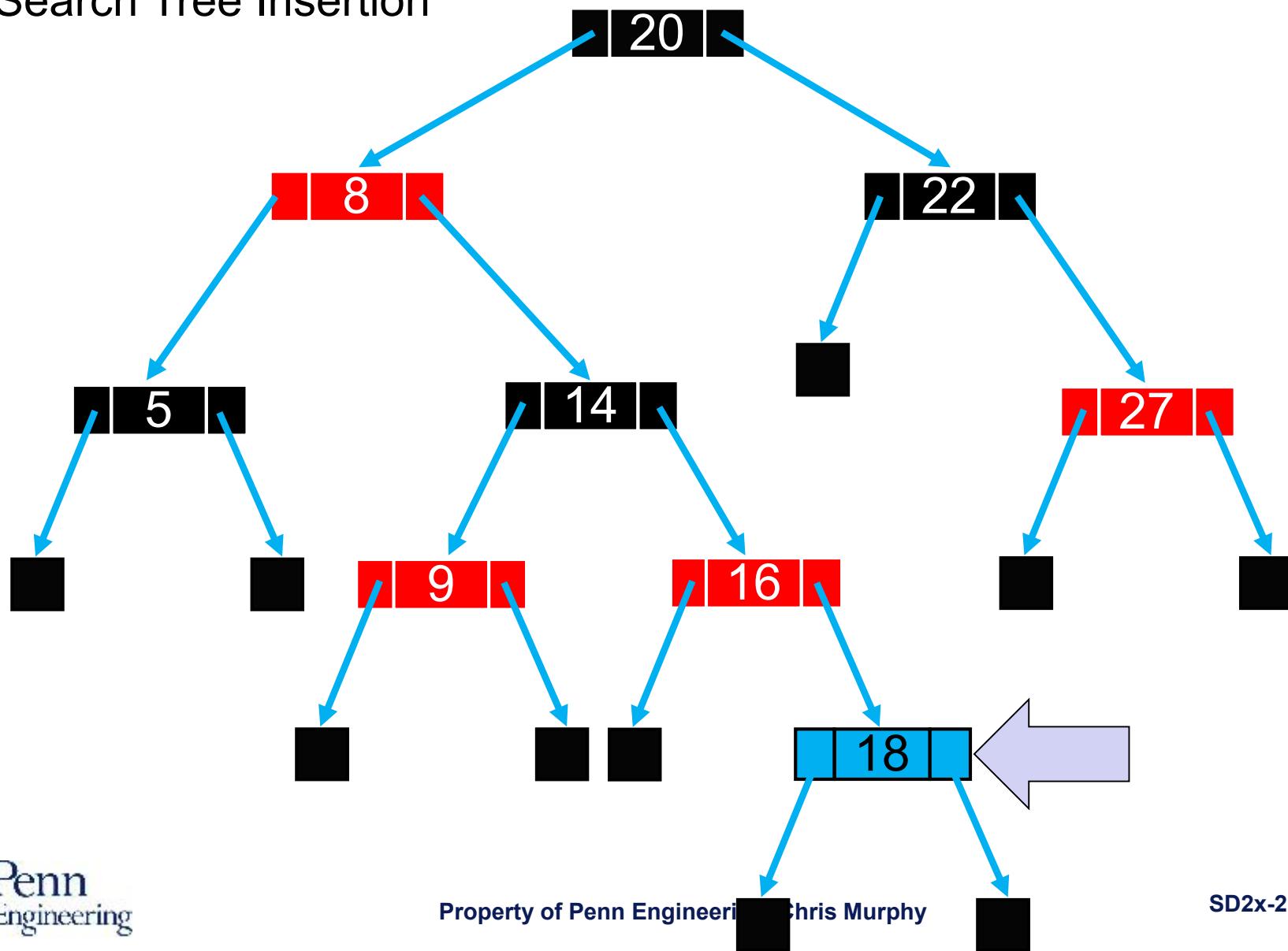
# How do we add a value to a red-black tree?



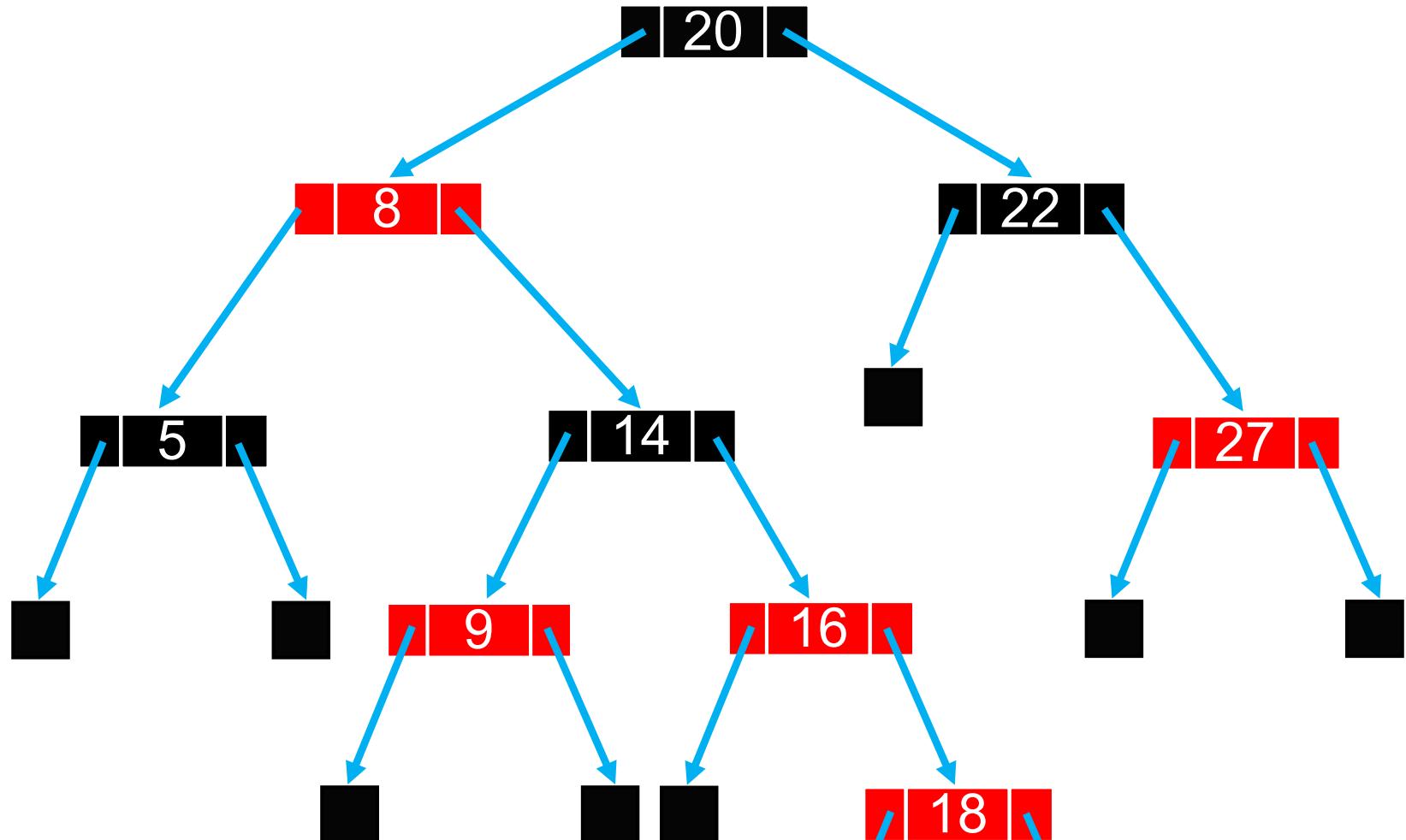
18



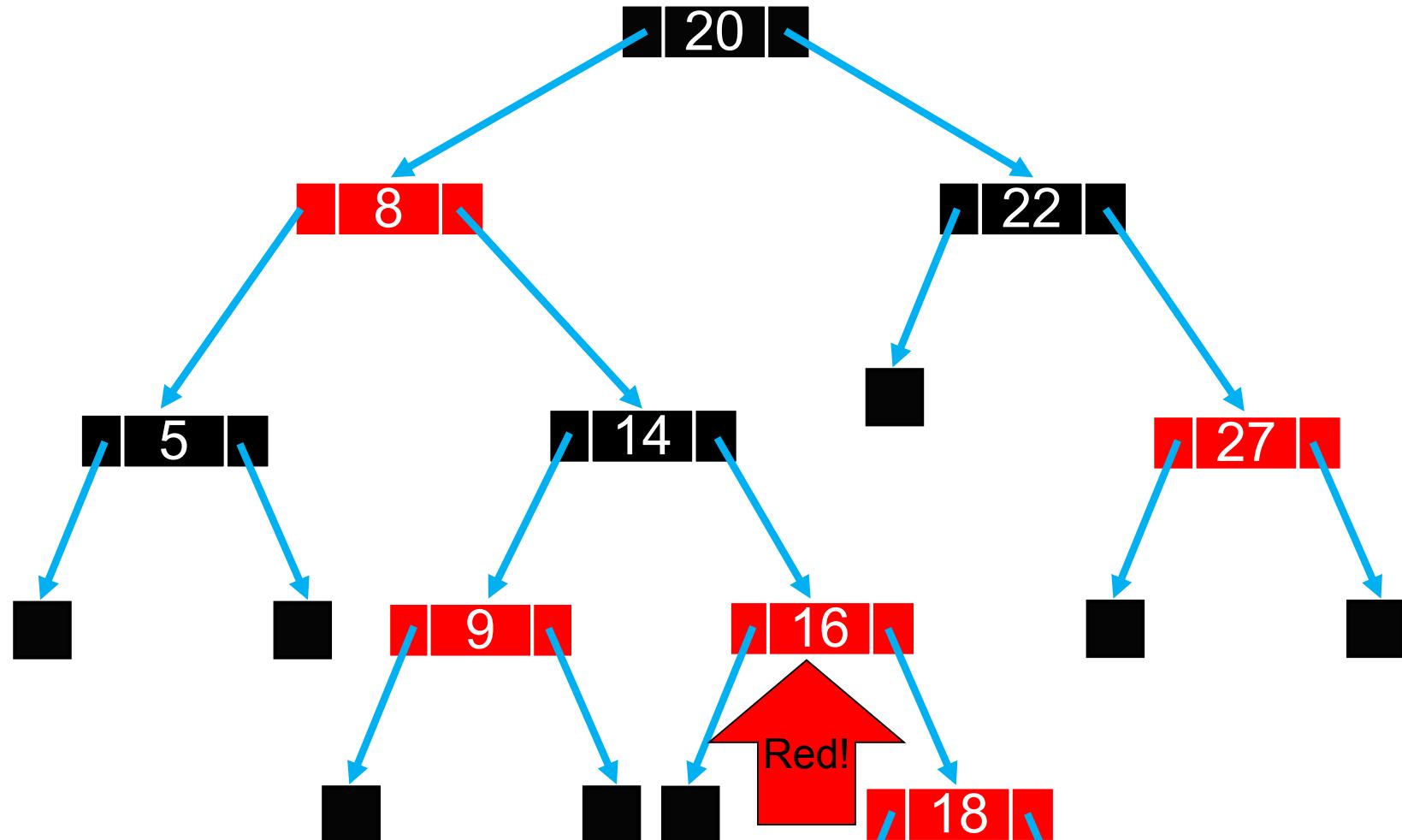
# 1. Regular Binary Search Tree Insertion



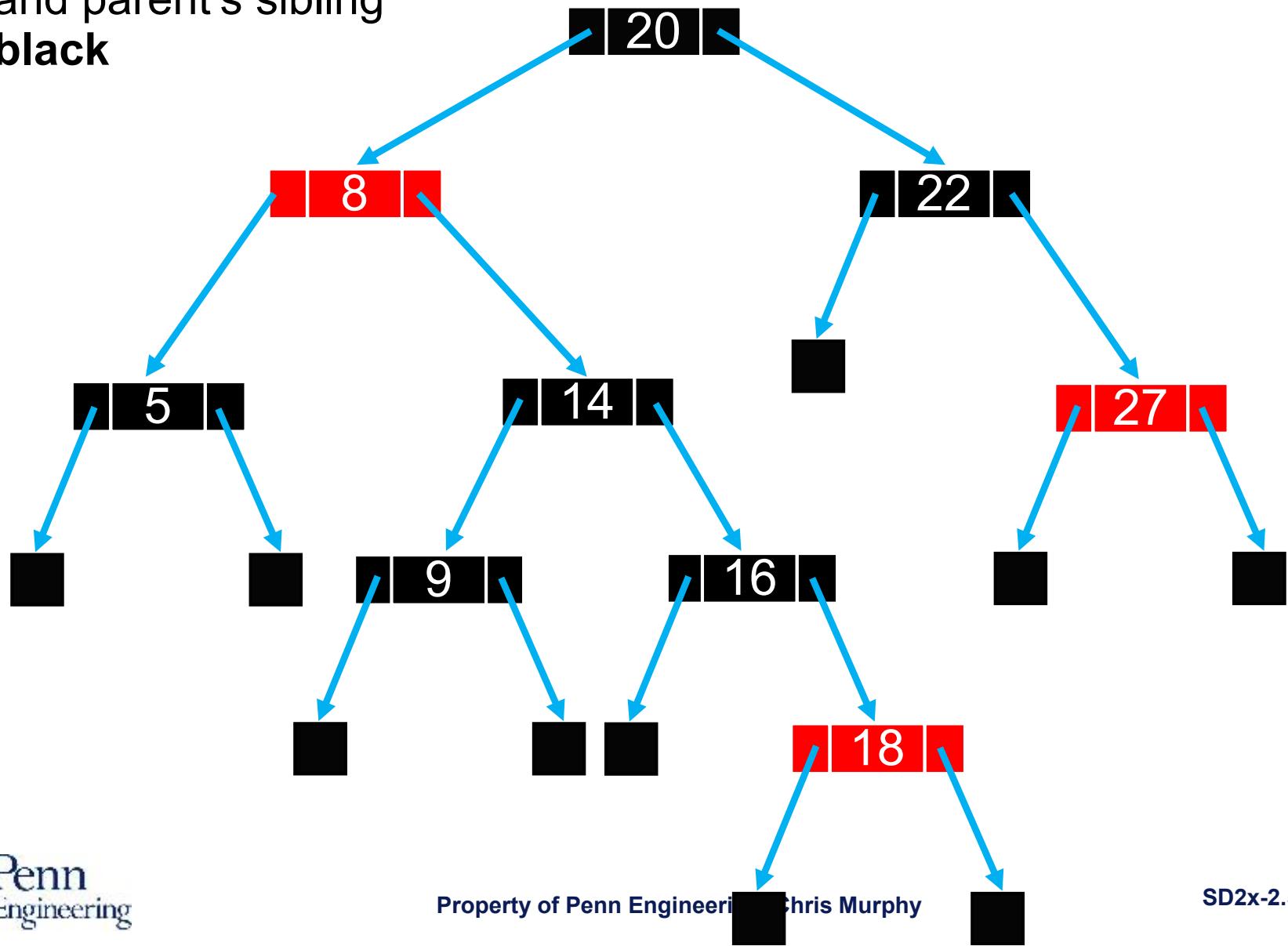
2. Color the new node red



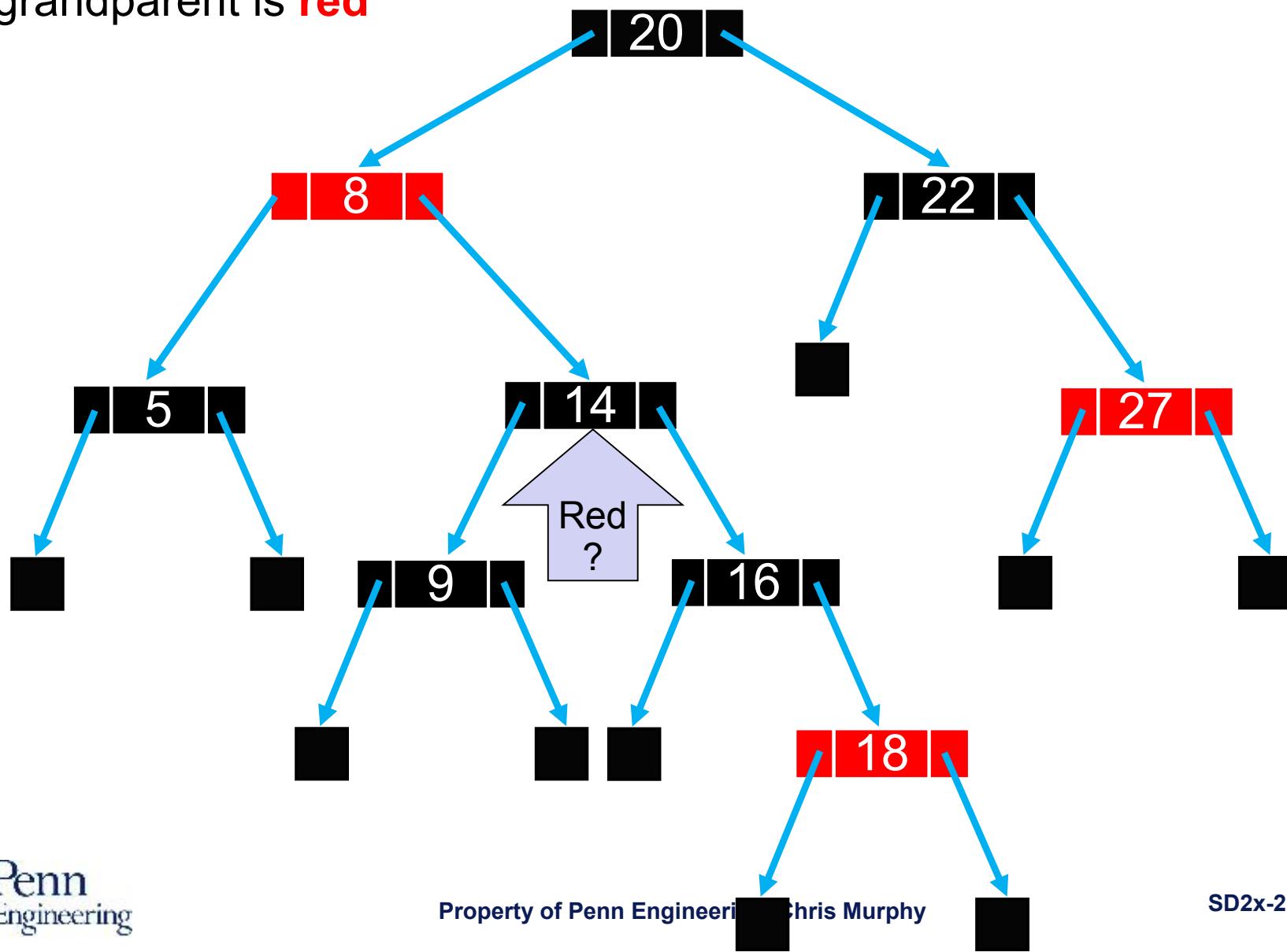
### 3. Check if parent is red



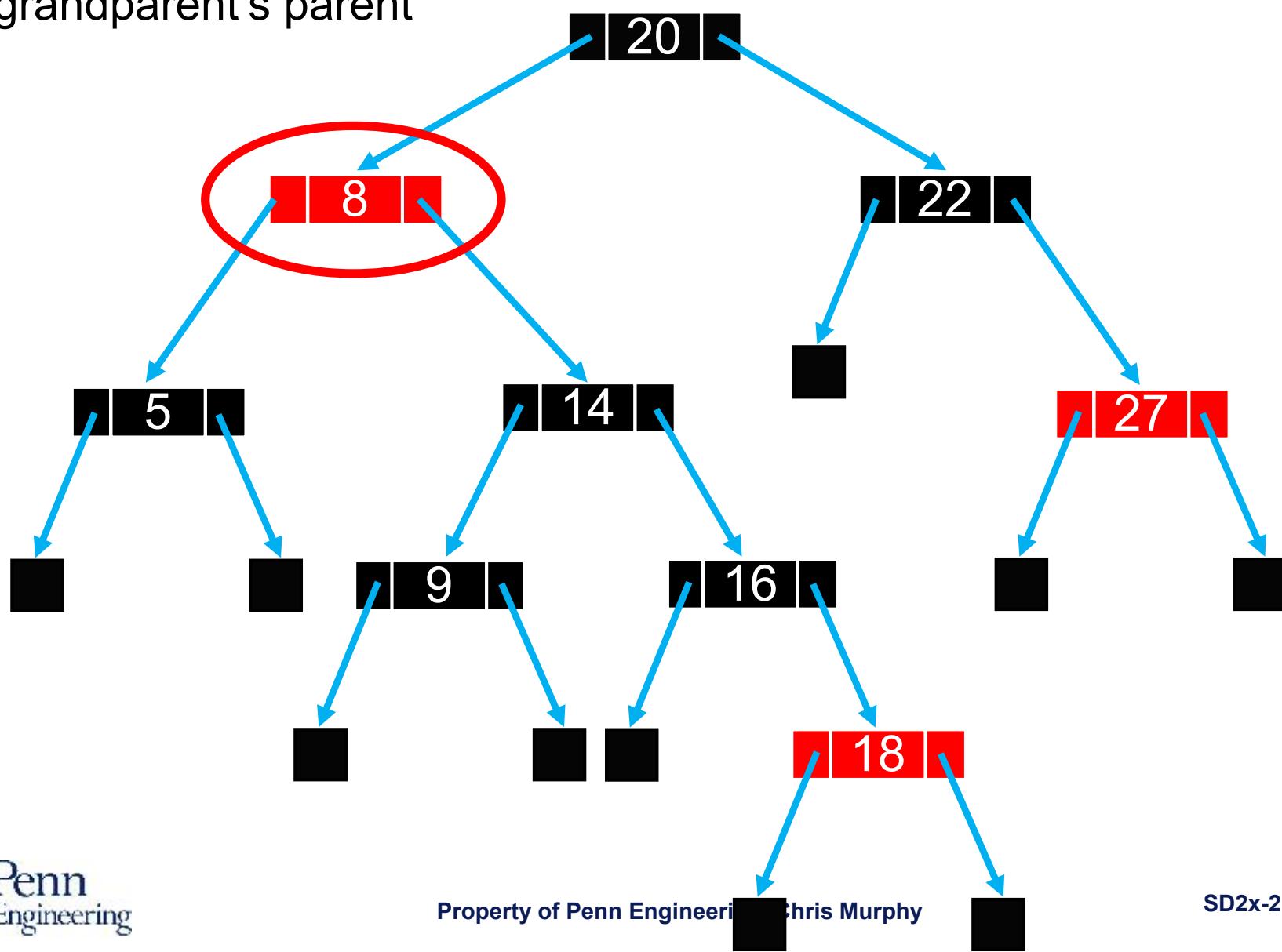
4. Color the parent  
and parent's sibling  
**black**



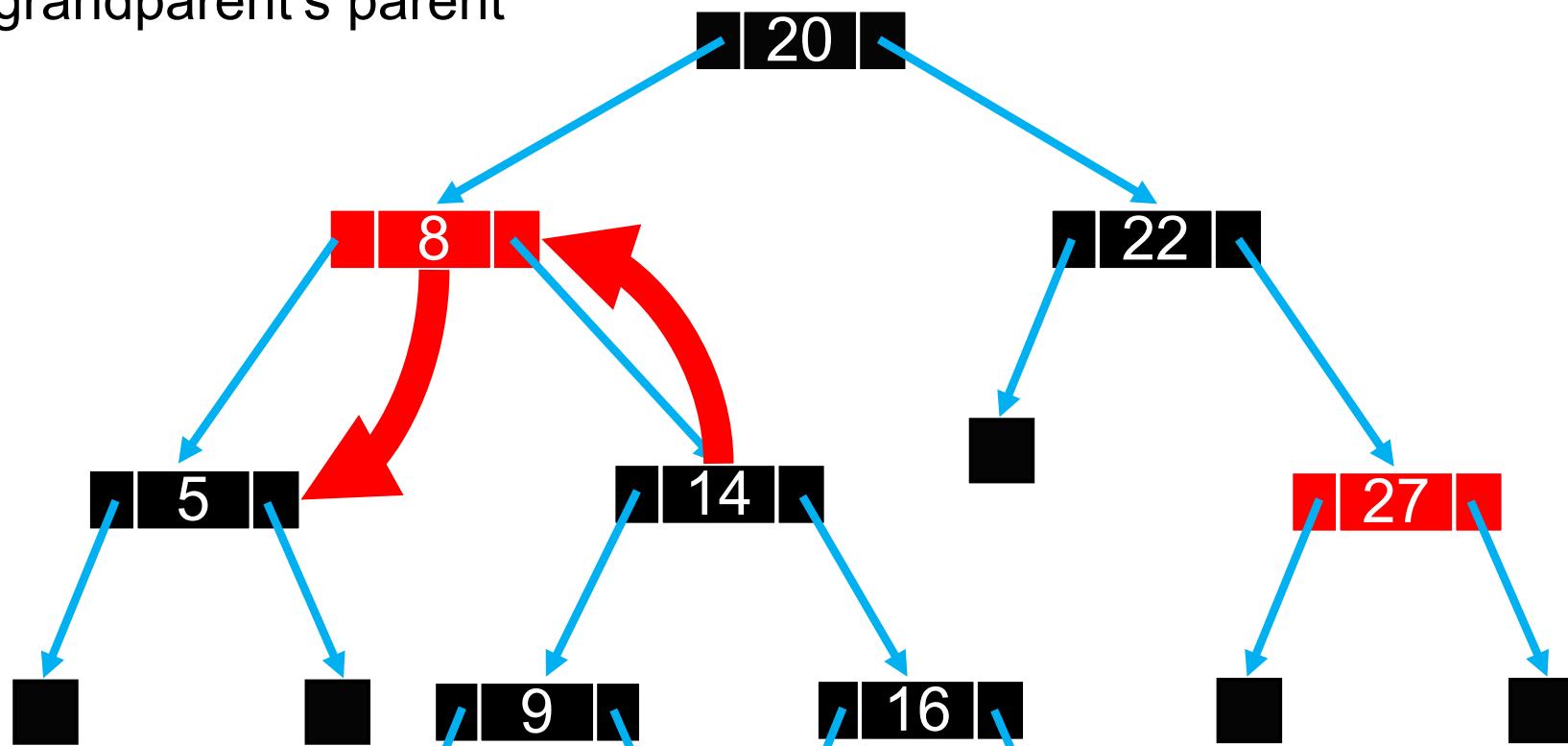
## 5. Check if the grandparent is red



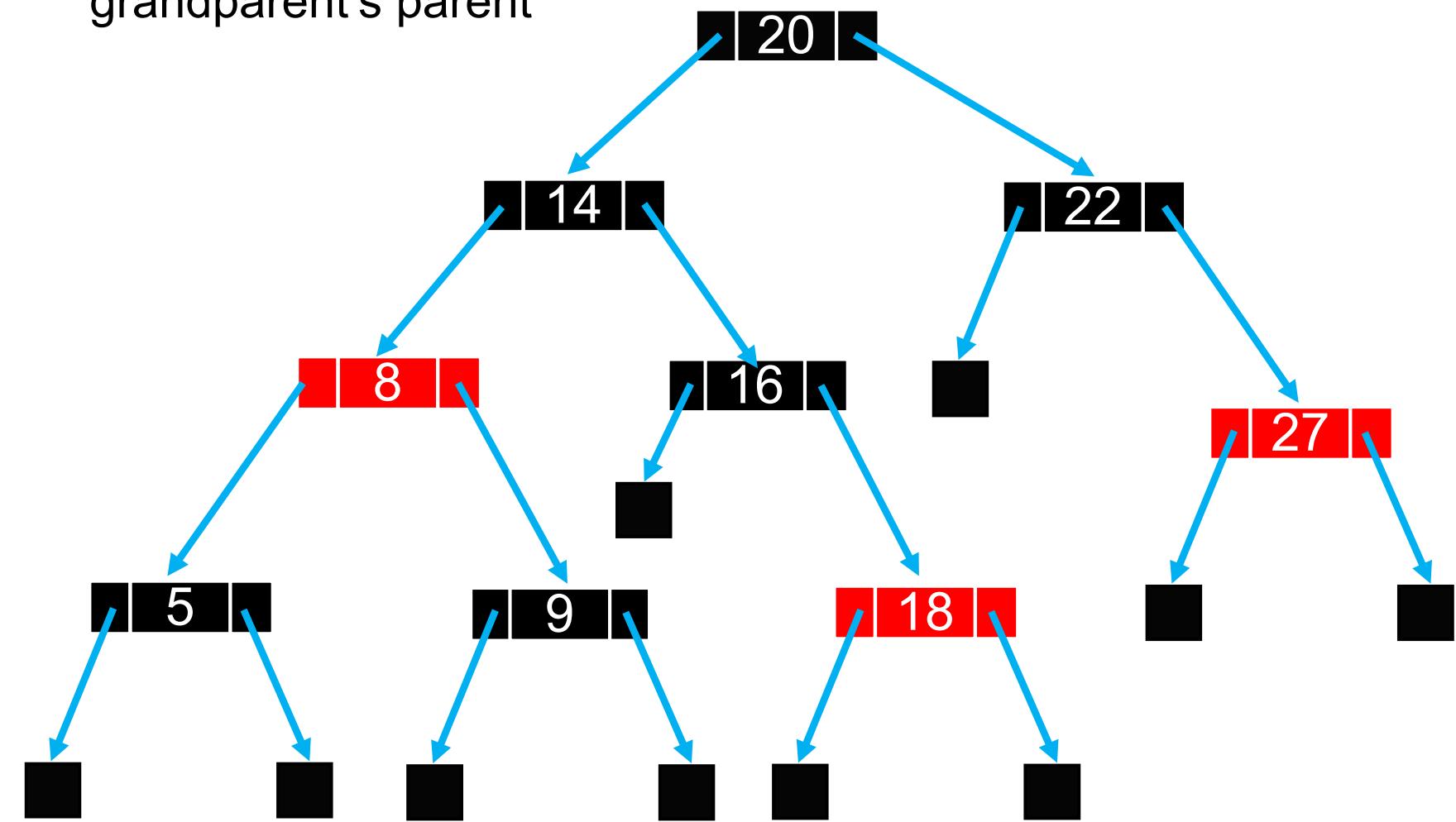
## 6. Rotate at the grandparent's parent



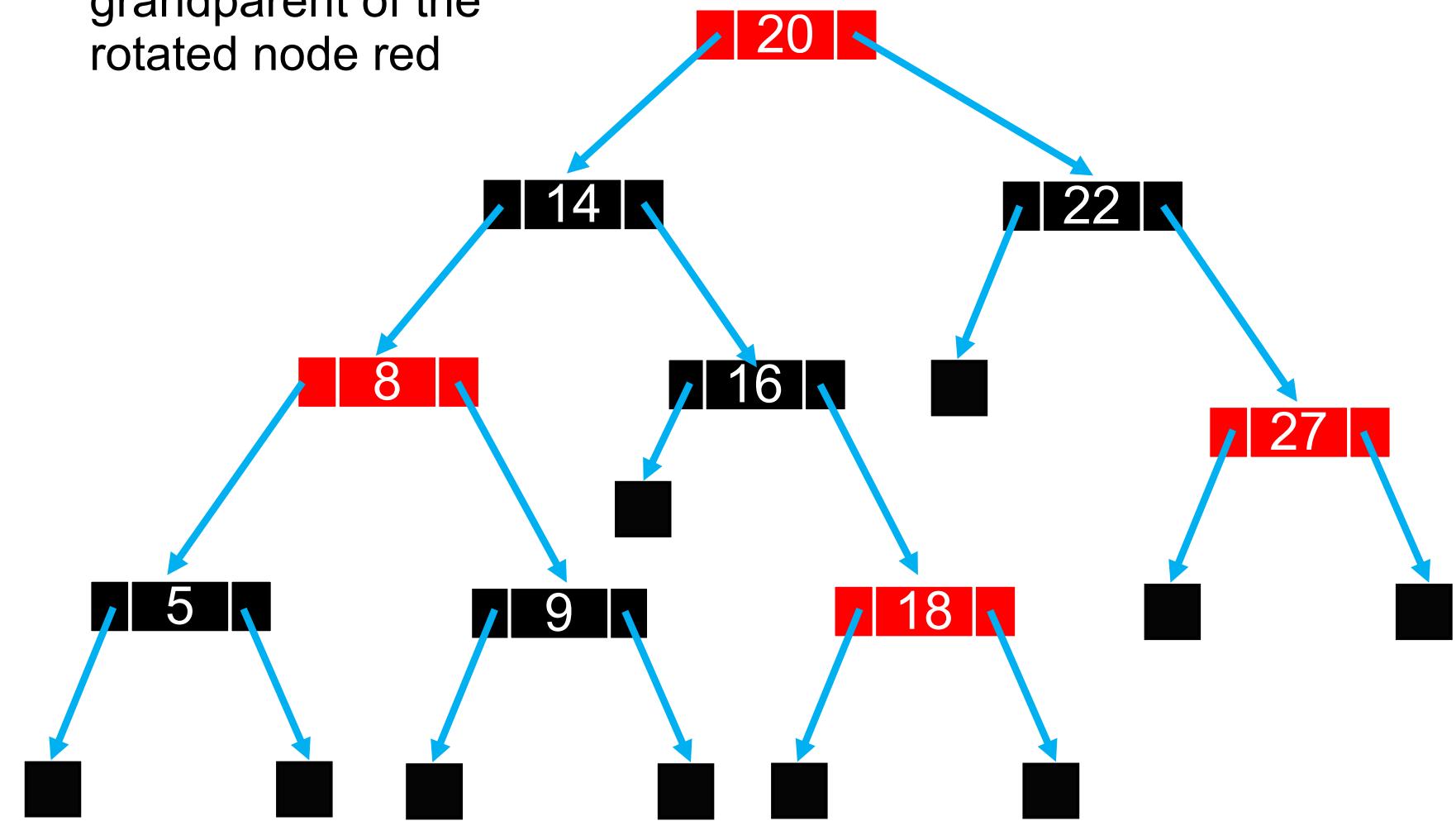
## 6. Rotate at the grandparent's parent



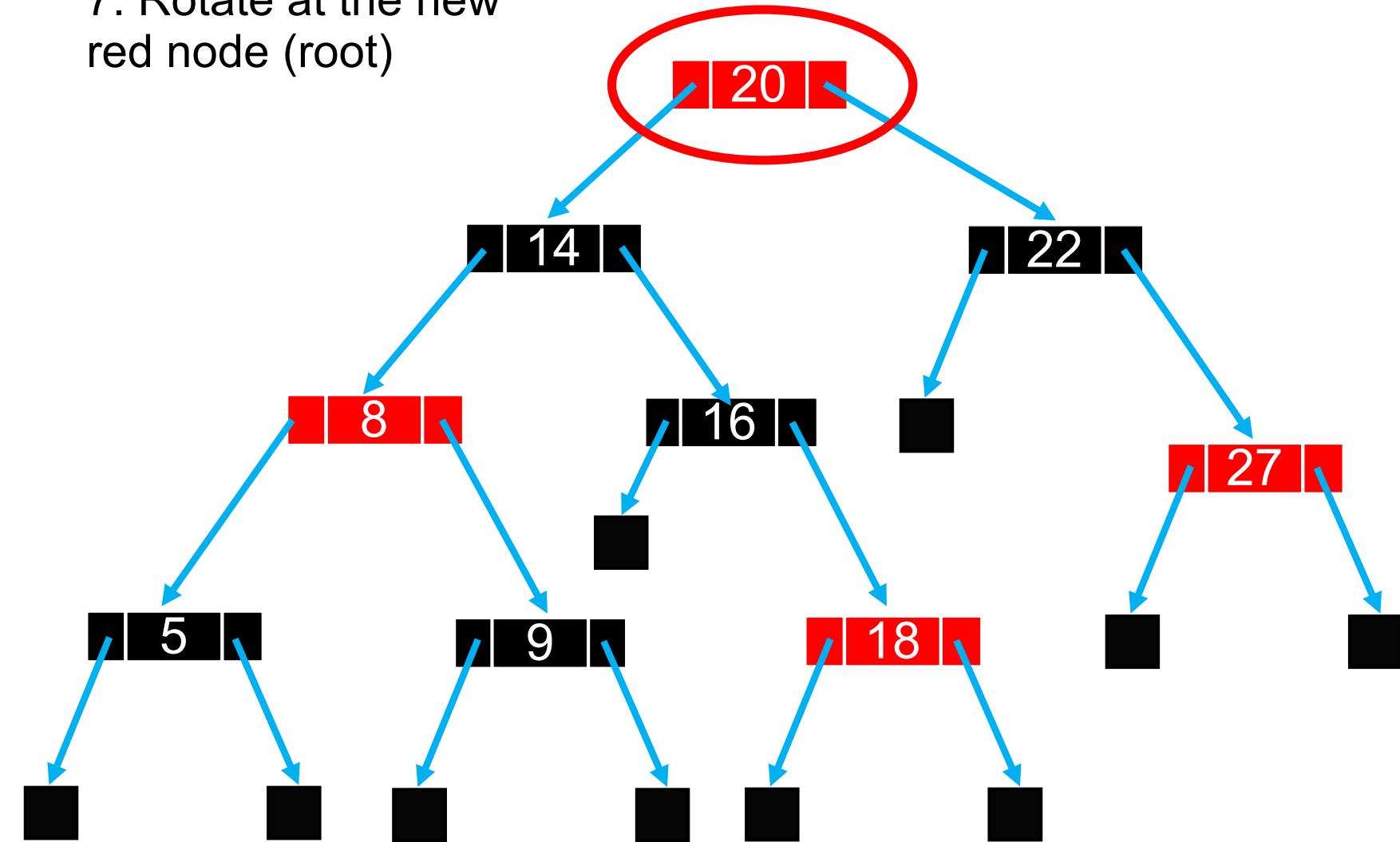
## 6. Rotate at the grandparent's parent



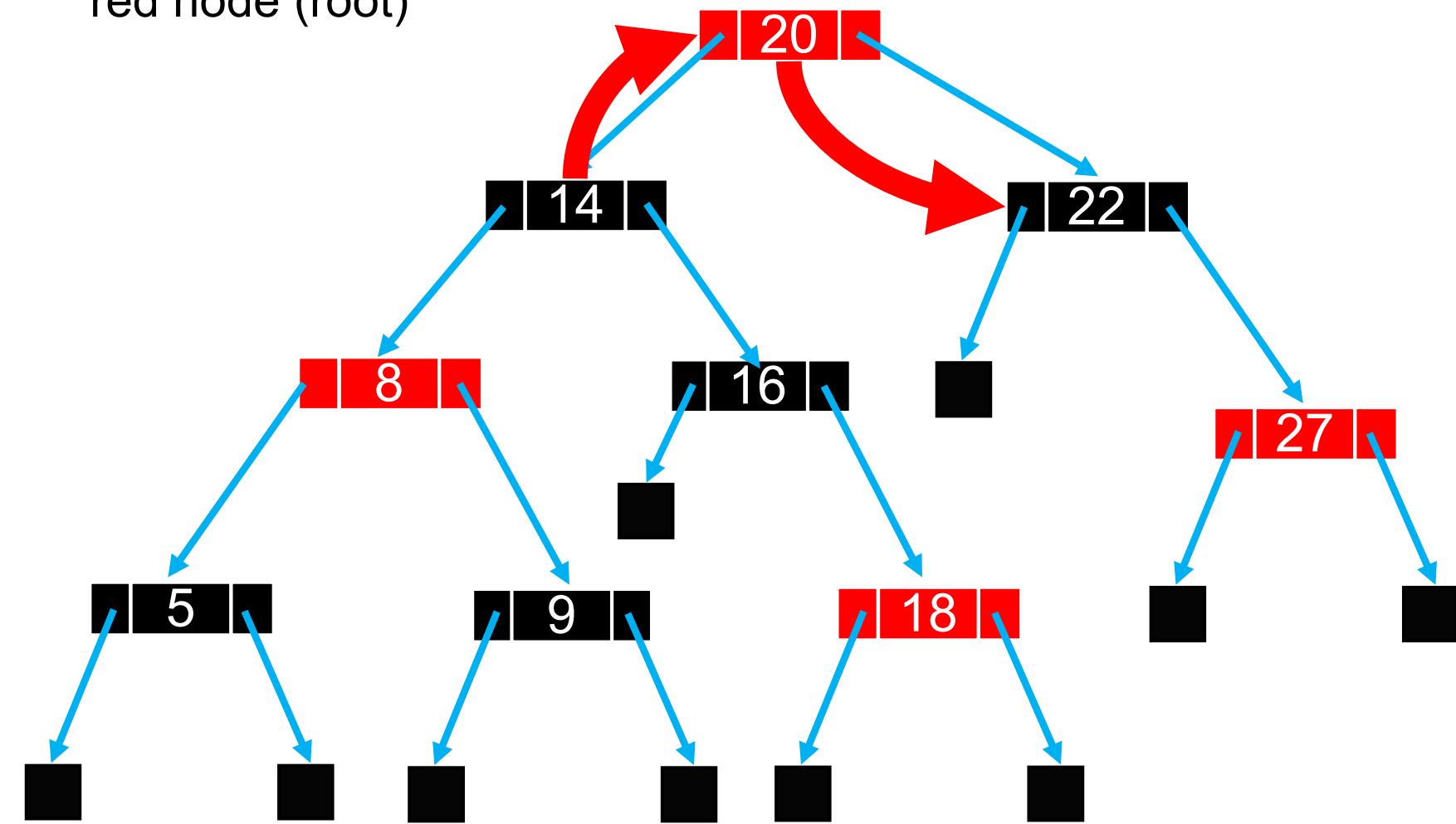
7. Color the  
grandparent of the  
rotated node red



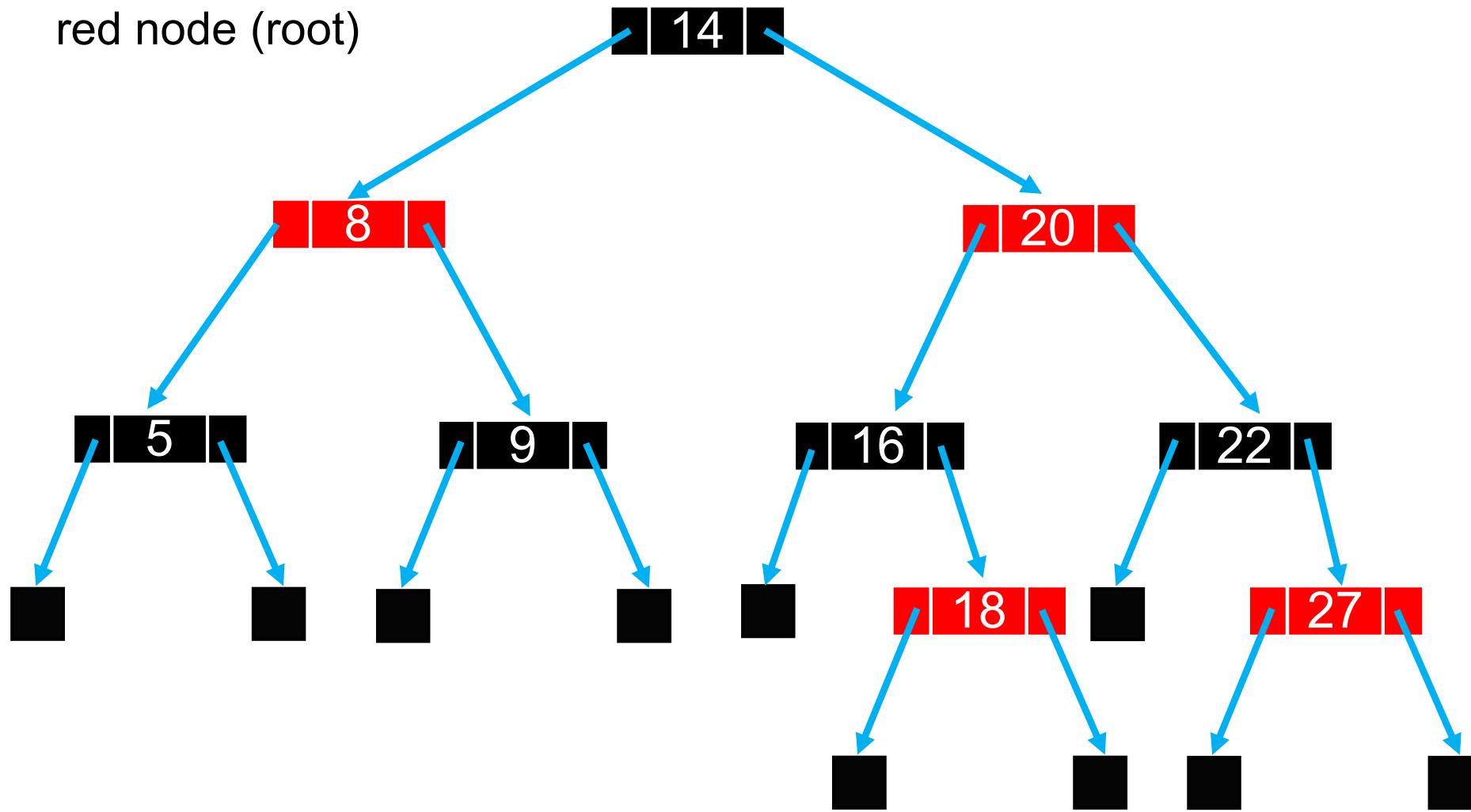
## 7. Rotate at the new red node (root)



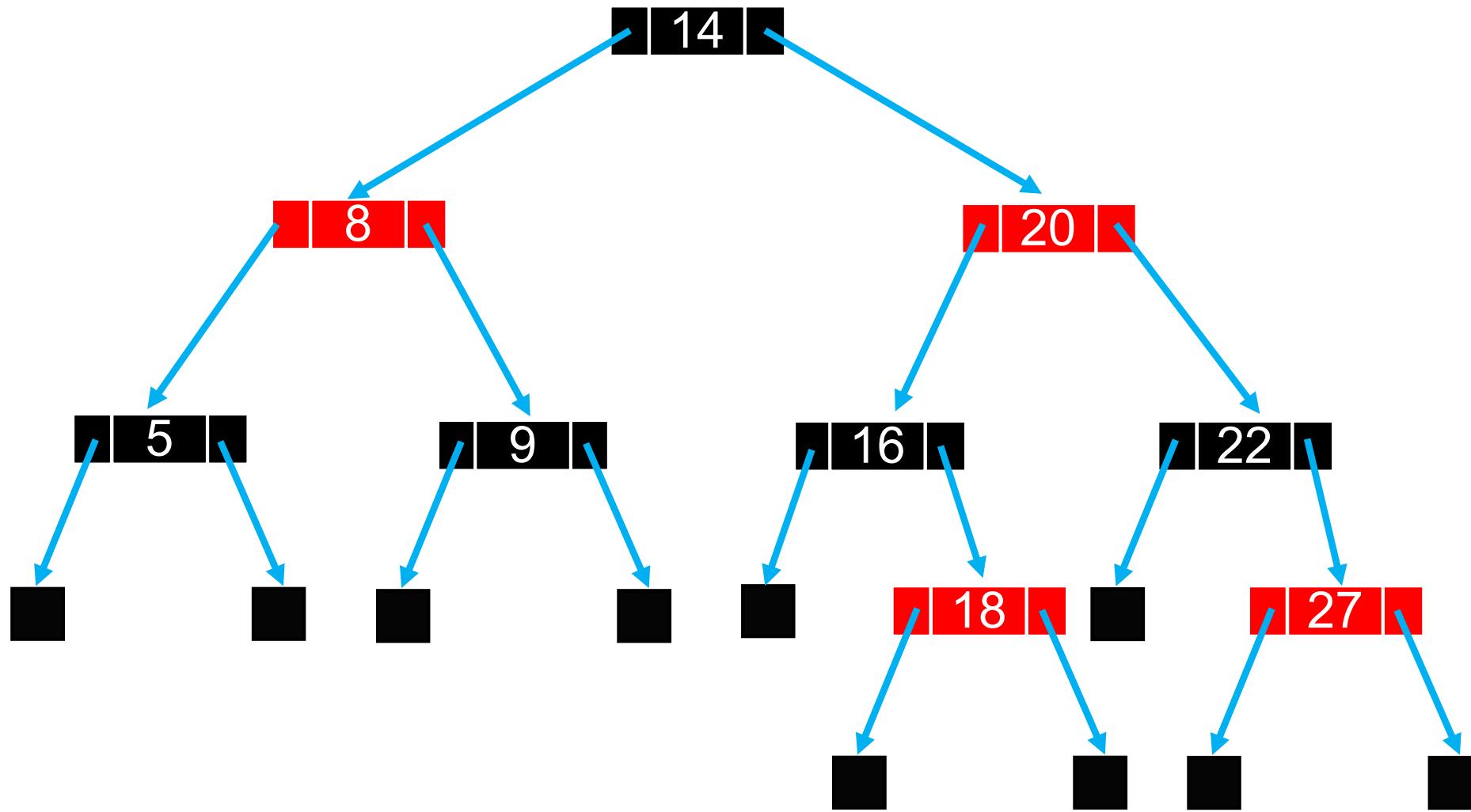
## 7. Rotate at the new red node (root)



## 7. Rotate at the new red node (root)



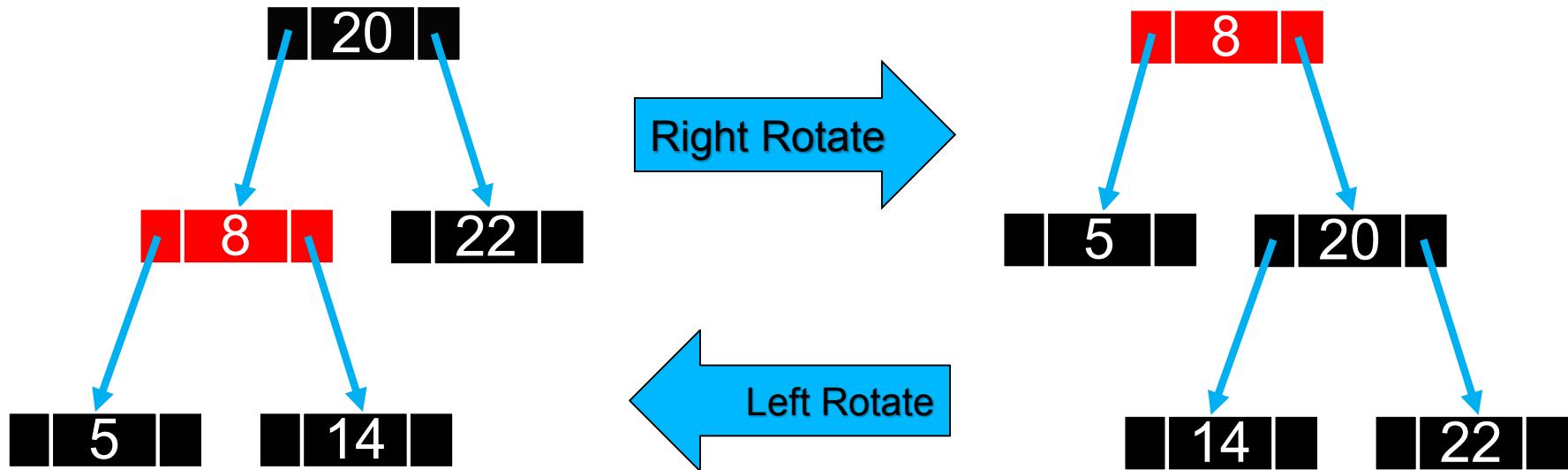
8. Done!



# Rotations

---

- An operation used in red-black trees with ***add*** and ***remove*** to help preserve order while balancing



# Recap: Binary Search Trees

---

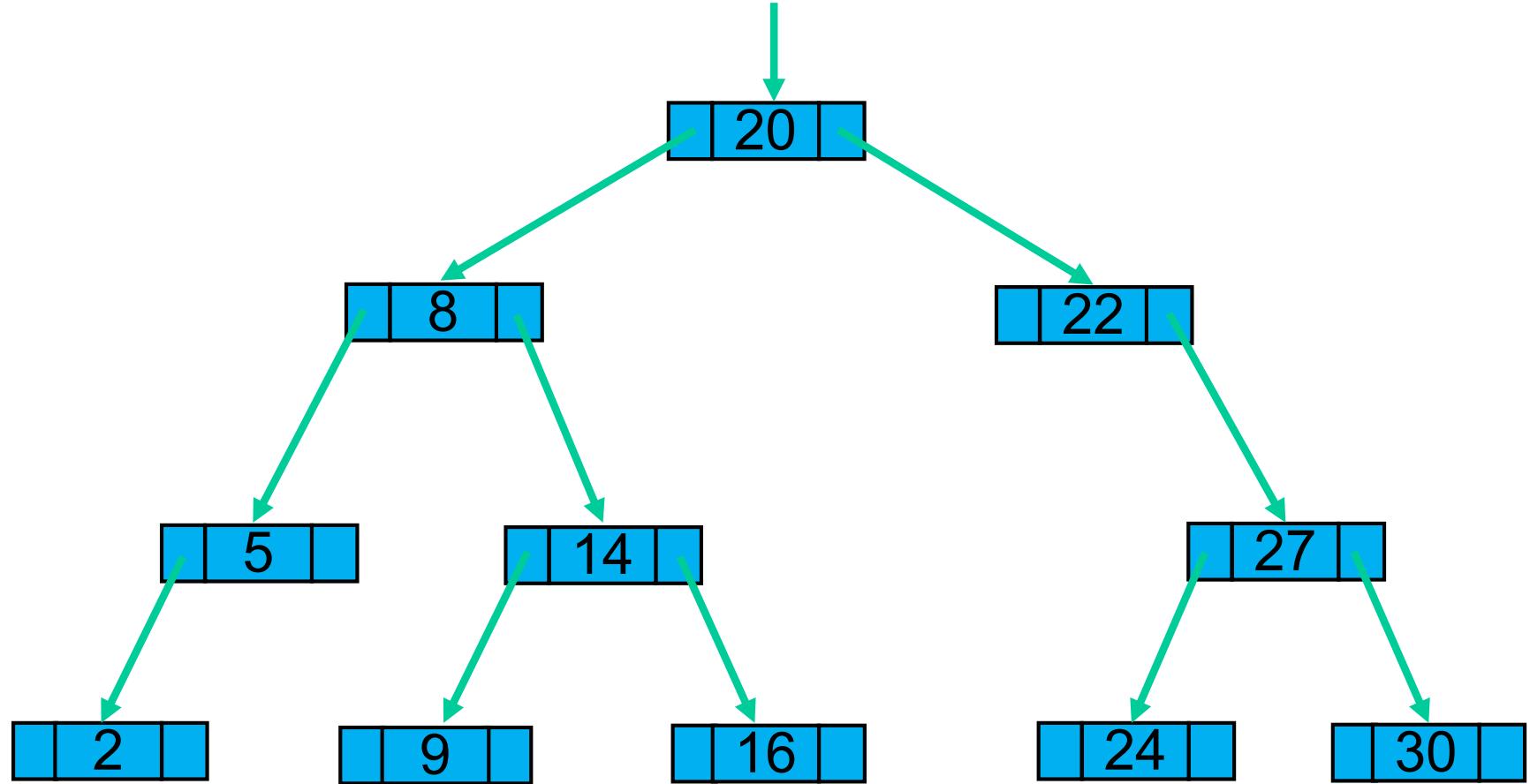
- All operations in a balanced Binary Search Tree are **O(log<sub>2</sub>n)**
- Maintaining balance is crucial!
- Red-Black Trees are a type of self-balancing Binary Search Tree that use rotation in order to maintain balance between left and right subtrees

# **SD2x2.6**

# **TreeSets in Java API; Comparing HashSets and TreeSets**

## **Chris**

root



# TreeSets in Java

---

- **java.util.TreeSet<E>**
- **add**: adds element to set if not already present
- **contains**: indicates whether set contains element
- **remove**: removes from set element if present
- **clear**: removes all elements
- **isEmpty**: indicates whether set contains no elements
- **size**: returns number of elements in set

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static TreeSet<String> getGlossaryTerms(  
                                     String manuscript) {  
    String[] tokens = getWordTokens(mscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
return glossaryTerms;  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static void main(String[] args) {  
  
    TreeSet<String> terms = getGlossaryTerms(args[0]);  
  
    Iterator<String> iter = terms.iterator();  
  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

```
public static TreeSet<String> getGlossaryTerms(
                                                String manuscript) {
    String[] tokens = getWordTokens(mscript);
    TreeSet<String> glossaryTerms = new TreeSet<String>();
    for (String token : tokens) {
        glossaryTerms.add(token);
    }
    return glossaryTerms;
}
```

```
public static void main(String[] args) {
    TreeSet<String> terms = getGlossaryTerms(args[0]);
    Iterator<String> iter = terms.iterator();
    while (iter.hasNext()) {
        System.out.println(iter.next());
    }
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static void main(String[] args) {  
  
    TreeSet<String> terms = getGlossaryTerms(args[0]);  
  
    Iterator<String> iter = terms.iterator();  
  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static void main(String[] args) {  
  
    TreeSet<String> terms = getGlossaryTerms(args[0]);  
  
    Iterator<String> iter = terms.iterator();  
  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static void main(String[] args) {  
  
    TreeSet<String> terms = getGlossaryTerms(args[0]);  
  
    Iterator<String> iter = terms.iterator();  
  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

```
public static TreeSet<String> getGlossaryTerms( String manuscript) {  
    String[] tokens = getWordTokens(manuscript);  
    TreeSet<String> glossaryTerms = new TreeSet<String>();  
    for (String token : tokens) {  
        glossaryTerms.add(token);  
    }  
    return glossaryTerms;  
}
```

```
public static void main(String[] args) {  
  
    TreeSet<String> terms = getGlossaryTerms(args[0]);  
  
    Iterator<String> iter = terms.iterator();  
  
    while (iter.hasNext()) {  
        System.out.println(iter.next());  
    } // the words are sorted alphabetically!  
}
```

# Sets in Java

---

- The `java.util.HashSet` and `java.util.TreeSet` classes implement the **`java.util.Set` interface**
- This interface defines basic Set operations: add, remove, contains, etc.

# TreeMap

---

- Just as HashSets can be used to store keys in HashMaps, TreeSets can be used to store keys in **TreeMaps**
- TreeMap: a Map in which keys are stored in a TreeSet and are mapped to values
- **java.util.TreeMap<K,V>**: class that implements the java.util.Map interface

```
public class BookReport implements Comparable<BookReport> {  
    protected String bookTitle;  
    protected String studentName;  
    protected int numPages;  
  
    public BookReport(String bookTitle, String studentName,  
                      int numPages) {  
        this.bookTitle = bookTitle;  
        this.studentName = studentName;  
        this.numPages = numPages;  
    }  
  
    public String toString() {  
        return (studentName + " wrote "  
                + numPages + " pages on " + bookTitle + ".");  
    }  
  
    @Override  
    public int compareTo(BookReport otherBookReport) {  
        return numPages - otherBookReport.numPages;  
    }  
}
```

```
public class BookReport implements Comparable<BookReport> {
    protected String bookTitle;
    protected String studentName;
    protected int numPages;

    public BookReport(String bookTitle, String studentName,
                      int numPages) {
        this.bookTitle = bookTitle;
        this.studentName = studentName;
        this.numPages = numPages;
    }

    public String toString() {
        return (studentName + " wrote "
               + numPages + " pages on " + bookTitle + ".");
    }

    @Override
    public int compareTo(BookReport otherBookReport) {
        return numPages - otherBookReport.numPages;
    }
}
```

```
public class BookReport implements Comparable<BookReport> {  
    protected String bookTitle;  
    protected String studentName;  
    protected int numPages;  
  
    public BookReport(String bookTitle, String studentName,  
                      int numPages) {  
        this.bookTitle = bookTitle;  
        this.studentName = studentName;  
        this.numPages = numPages;  
    }  
  
    public String toString() {  
        return (studentName + " wrote "  
                + numPages + " pages on " + bookTitle + ".");  
    }  
  
    @Override  
    public int compareTo(BookReport otherBookReport) {  
        return numPages - otherBookReport.numPages;  
    }  
}
```

```
public class BookReport implements Comparable<BookReport> {  
    protected String bookTitle;  
    protected String studentName;  
    protected int numPages;  
  
    public BookReport(String bookTitle, String studentName,  
                      int numPages) {  
        this.bookTitle = bookTitle;  
        this.studentName = studentName;  
        this.numPages = numPages;  
    }  
  
    public String toString() {  
        return (studentName + " wrote "  
                + numPages + " pages on " + bookTitle + ".");  
    }  
  
    @Override  
    public int compareTo(BookReport otherBookReport) {  
        return numPages - otherBookReport.numPages;  
    }  
}
```

```
public static void main(String[] args) {  
    BookReport chris = new BookReport(  
        "The Cathedral and the Bazaar", "Chris", 50);  
    BookReport ada = new BookReport(  
        "The Idea Factory", "Ada", 2);  
    BookReport toby = new BookReport(  
        "Toby Tries a Taco", "Toby", 100);  
    BookReport pooh = new BookReport(  
        "The Secret Life of Bees", "Pooh", 2);  
  
    . . .
```

```
public static void main(String[] args) {  
    BookReport chris = new BookReport(  
        "The Cathedral and the Bazaar", "Chris", 50);  
    BookReport ada = new BookReport(  
        "The Idea Factory", "Ada", 2);  
    BookReport toby = new BookReport(  
        "Toby Tries a Taco", "Toby", 100);  
    BookReport pooh = new BookReport(  
        "The Secret Life of Bees", "Pooh", 2);  
  
    . . .
```

```
public static void main(String[] args) {  
    BookReport chris = new BookReport(  
        "The Cathedral and the Bazaar", "Chris", 50);  
    BookReport ada = new BookReport(  
        "The Idea Factory", "Ada", 2);  
    BookReport toby = new BookReport(  
        "Toby Tries a Taco", "Toby", 100);  
    BookReport pooh = new BookReport(  
        "The Secret Life of Bees", "Pooh", 2);
```

```
TreeMap<BookReport, Integer> reportScores =  
    new TreeMap<BookReport, Integer>();
```

... .

```
public static void main(String[] args) {  
    BookReport chris = new BookReport(  
        "The Cathedral and the Bazaar", "Chris", 50);  
    BookReport ada = new BookReport(  
        "The Idea Factory", "Ada", 2);  
    BookReport toby = new BookReport(  
        "Toby Tries a Taco", "Toby", 100);  
    BookReport pooh = new BookReport(  
        "The Secret Life of Bees", "Pooh", 2);  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    . . .
```

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
  
    . . .
```

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
        BookReport reportInfo = entry.getKey();  
        int score = entry.getValue();  
        System.out.println(reportInfo + " " + score + " pts.");  
    }  
  
    . . .
```

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
        BookReport reportInfo = entry.getKey();  
        int score = entry.getValue();  
        System.out.println(reportInfo + " " + score + " pts.");  
    }  
  
    . . .
```

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
        BookReport reportInfo = entry.getKey();  
        int score = entry.getValue();  
        System.out.println(reportInfo + " " + score + " pts.");  
    }  
    . . .
```

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
        BookReport reportInfo = entry.getKey();  
        int score = entry.getValue();  
        System.out.println(reportInfo + " " + score + " pts.");  
    }  
    . . .
```

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
        BookReport reportInfo = entry.getKey();  
        int score = entry.getValue();  
        System.out.println(reportInfo + " " + score + " pts.");  
    }  
}
```

## Output

Ada wrote 2 pages on The Idea Factory. 30 pts.

Pooh wrote 2 pages on The Secret Life of Bees. 70 pts.

Chris wrote 50 pages on The Cathedral and the Bazaar. 87 pts.

Toby wrote 100 pages on Toby Tries a Taco. 30 pts.

```
public static void main(String[] args) {  
    . . .  
  
    TreeMap<BookReport, Integer> reportScores =  
        new TreeMap<BookReport, Integer>();  
    reportScores.put(chris, 87);  
    reportScores.put(ada, 30);  
    reportScores.put(toby, 30);  
    reportScores.put(pooh, 70);  
  
    for (Map.Entry<BookReport, Integer> entry :  
        reportScores.entrySet()) {  
        BookReport reportInfo = entry.getKey();  
        int score = entry.getValue();  
        System.out.println(reportInfo + " " + score + " pts.");  
    }  
}
```

## Output

Ada wrote **2** pages on The Idea Factory. 30 pts.

Pooh wrote **2** pages on The Secret Life of Bees. 70 pts.

Chris wrote **50** pages on The Cathedral and the Bazaar. 87 pts.

Toby wrote **100** pages on Toby Tries a Taco. 30 pts.

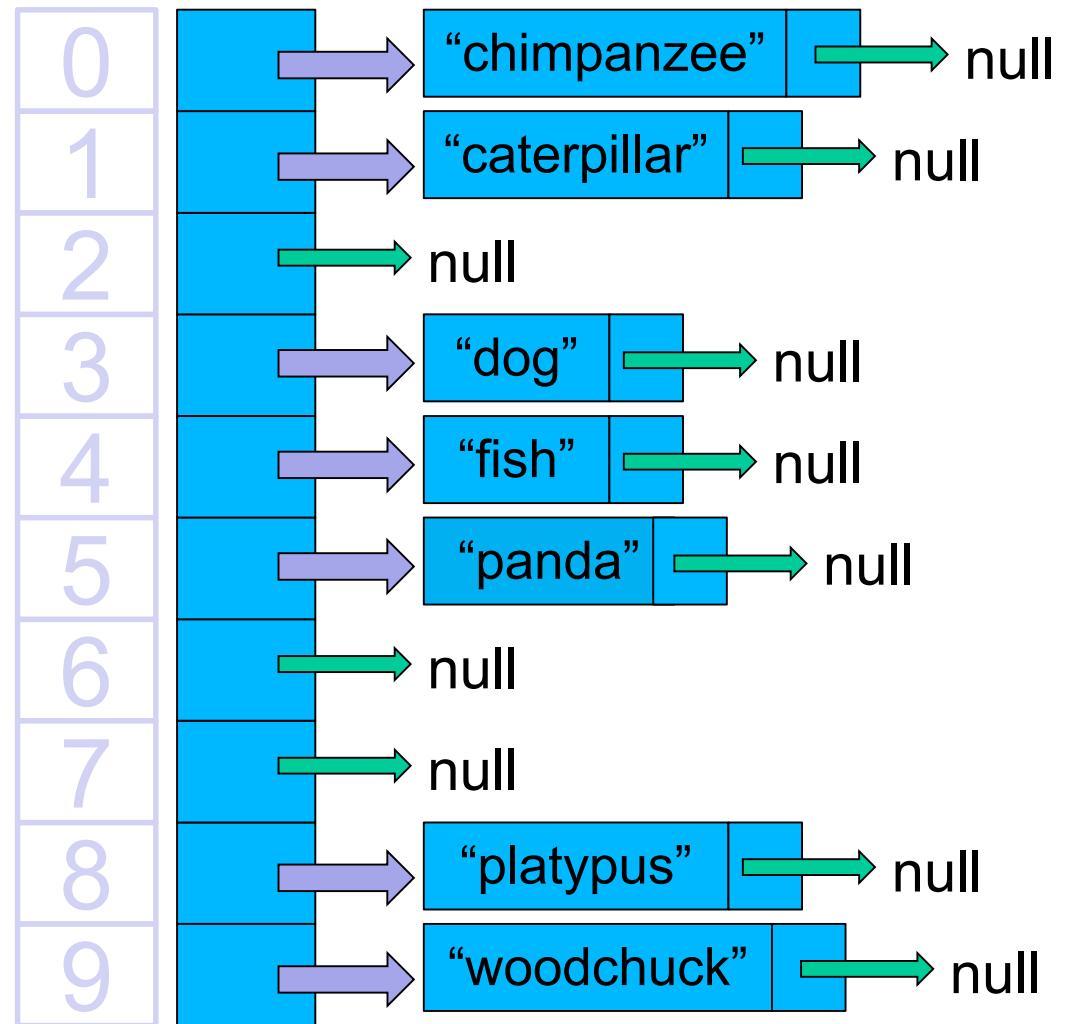
# **Set implementations: HashSet vs. TreeSet**

# Comparing HashSet vs. TreeSet

---

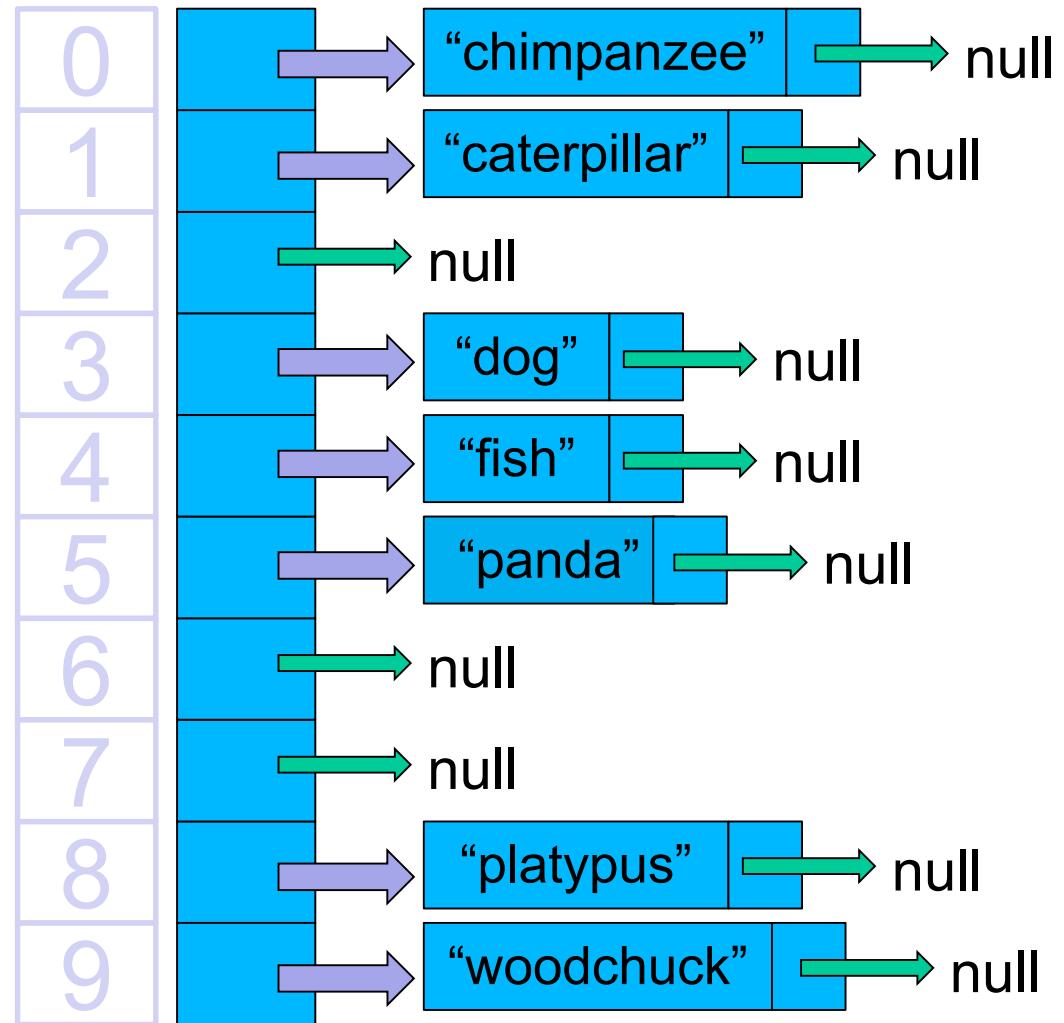
- Let's say we have a HashSet of  $n$  strings
  - And a TreeSet containing the same strings
- 
- How many steps do we need in order to determine whether the set contains a given string?

# Finding an element in HashSet

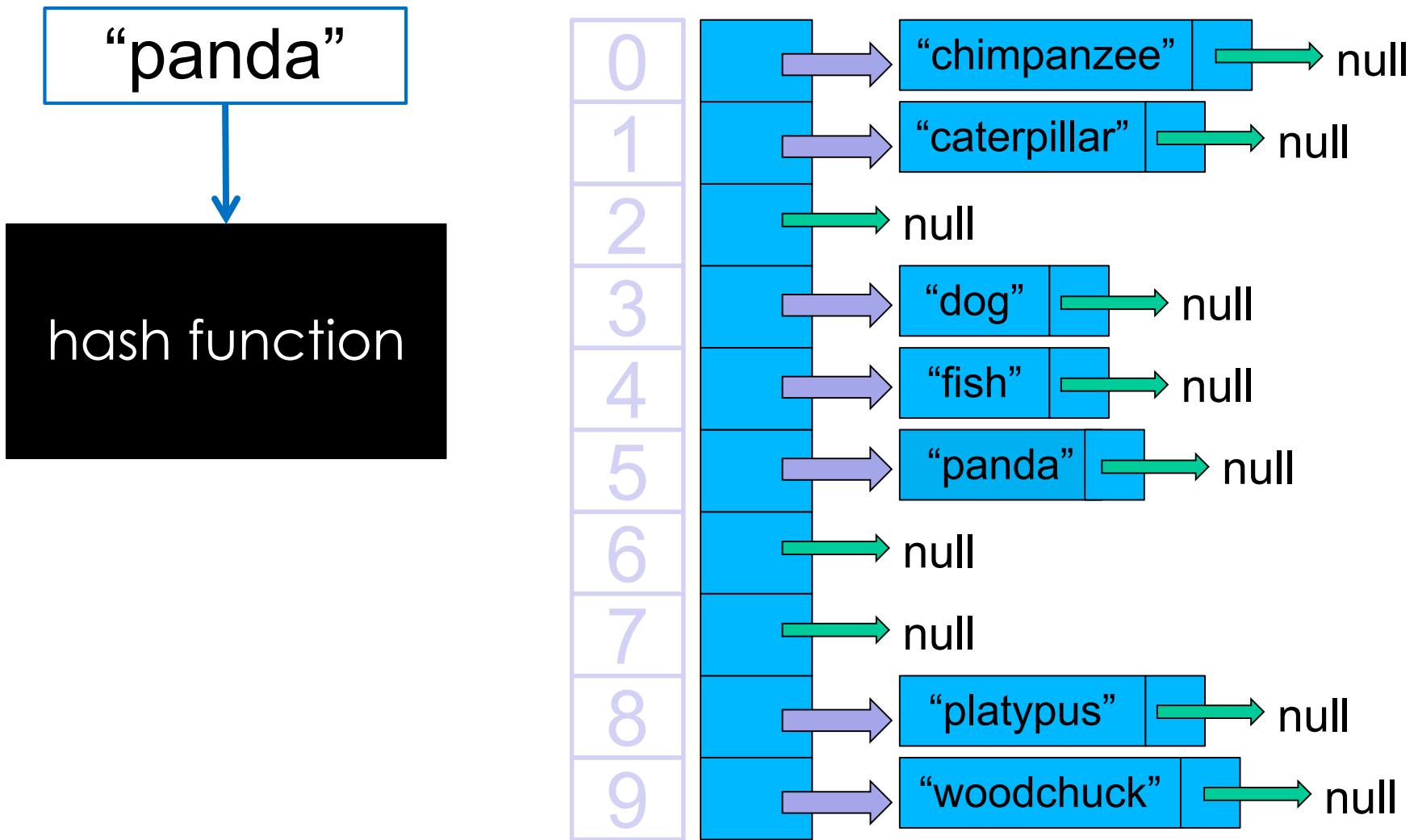


# Finding an element in HashSet

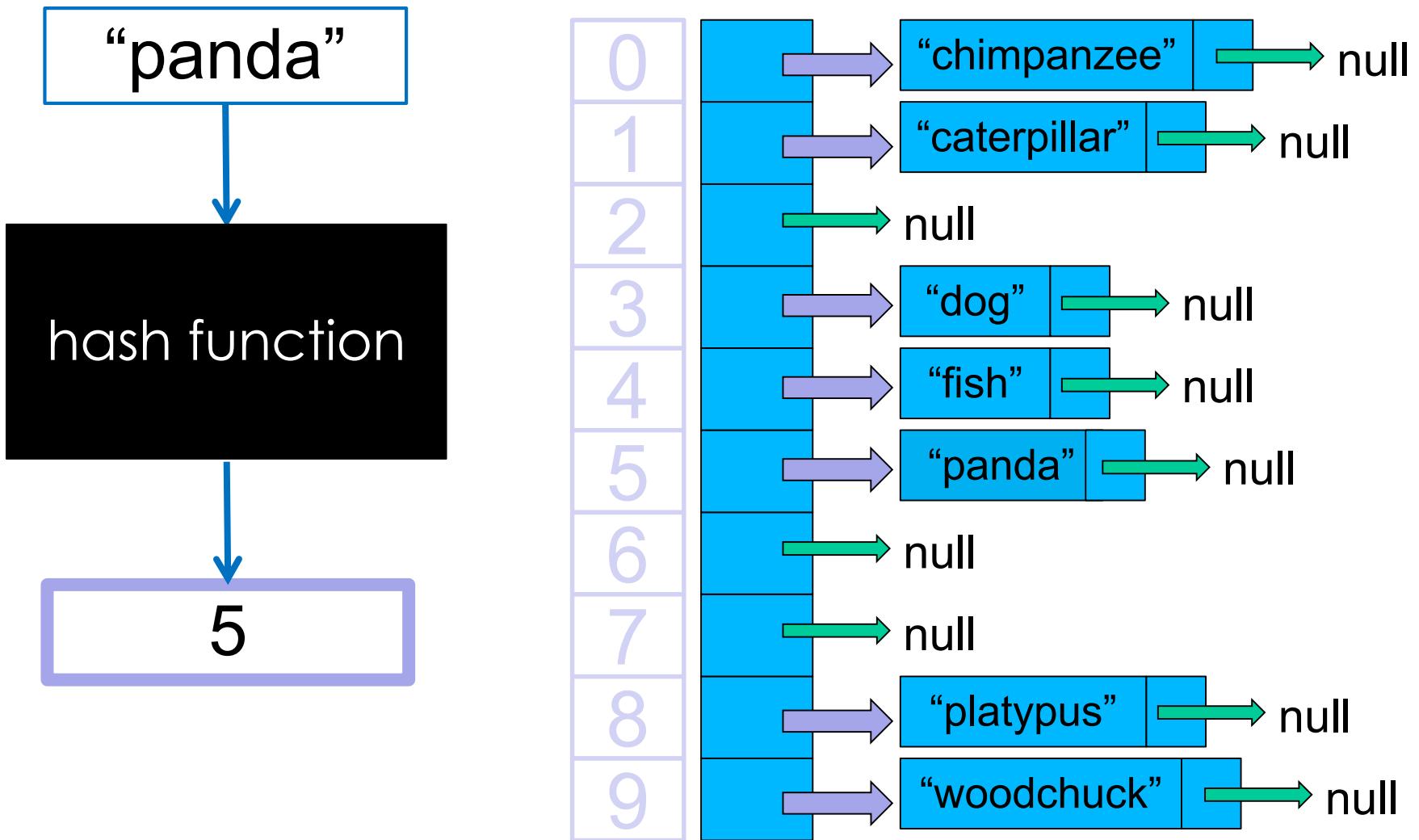
“panda”



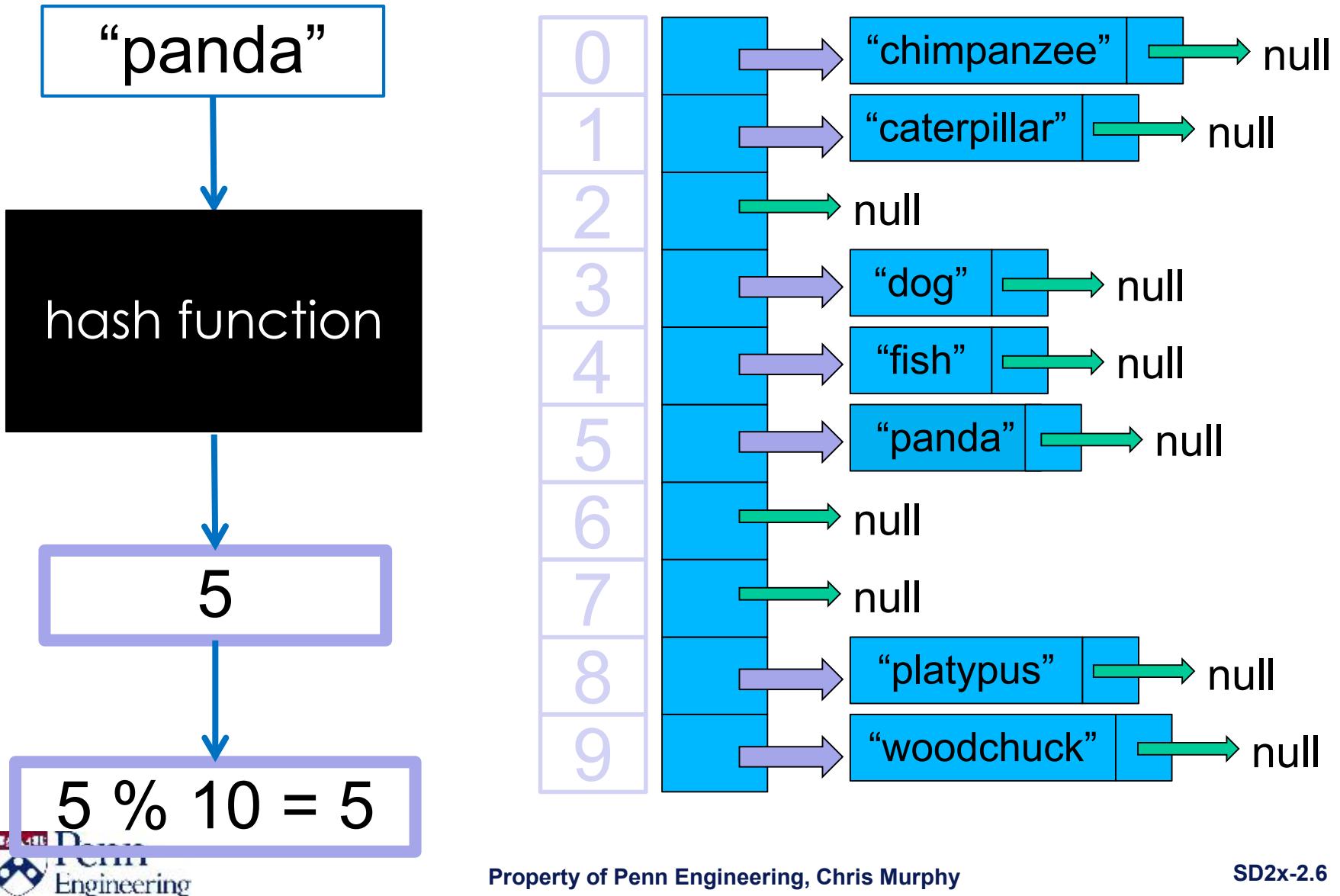
# Finding an element in HashSet



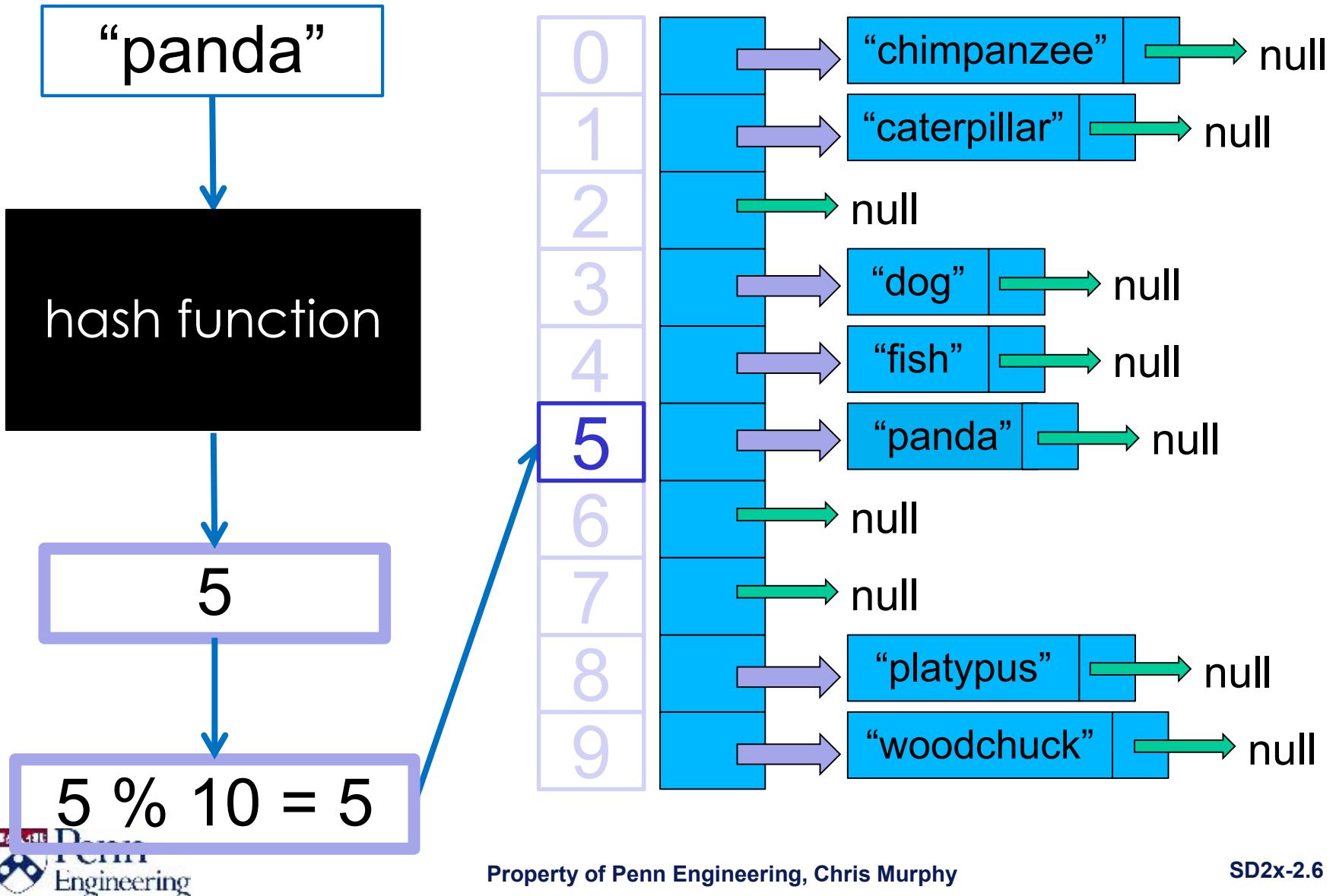
# Finding an element in HashSet



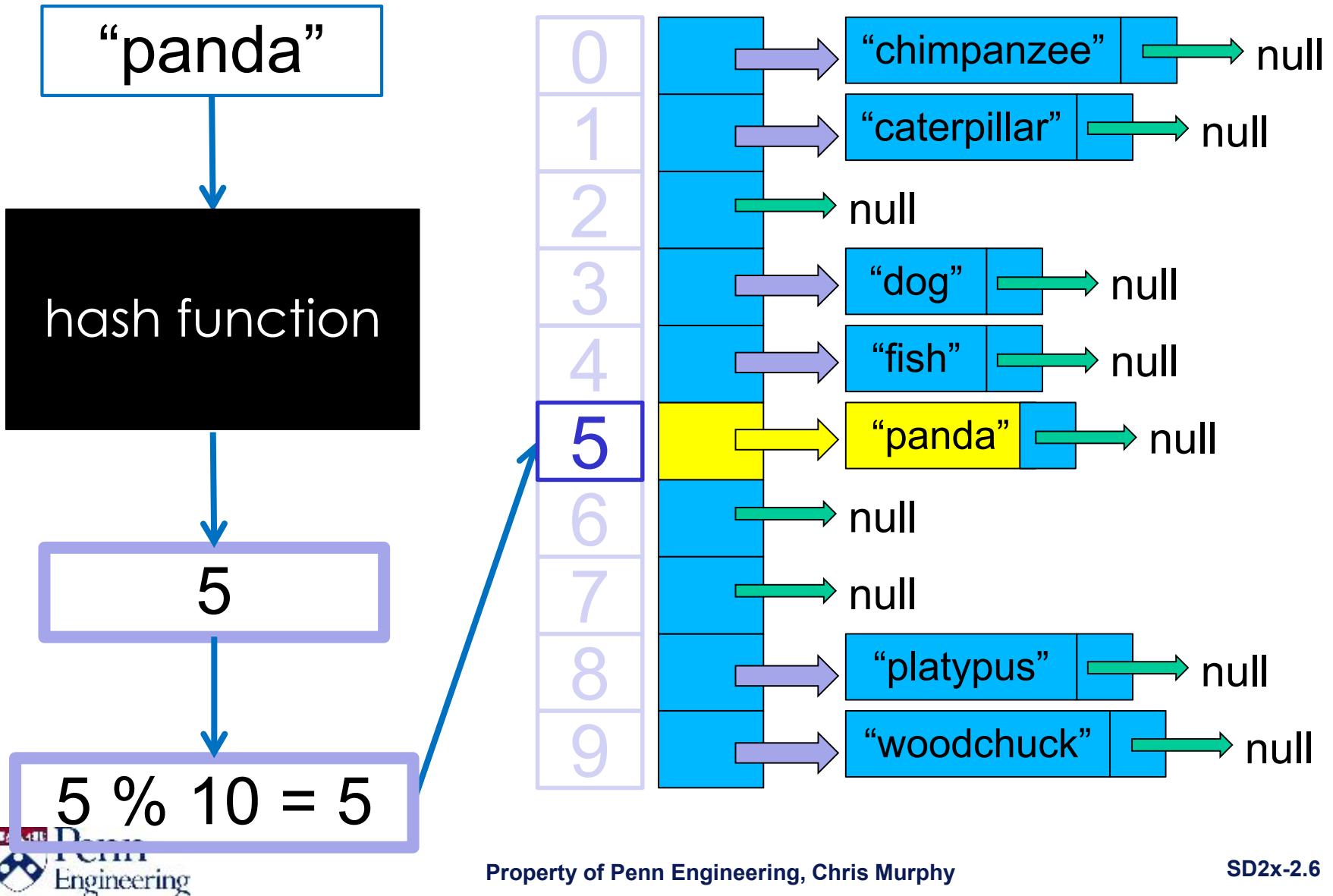
# Finding an element in HashSet



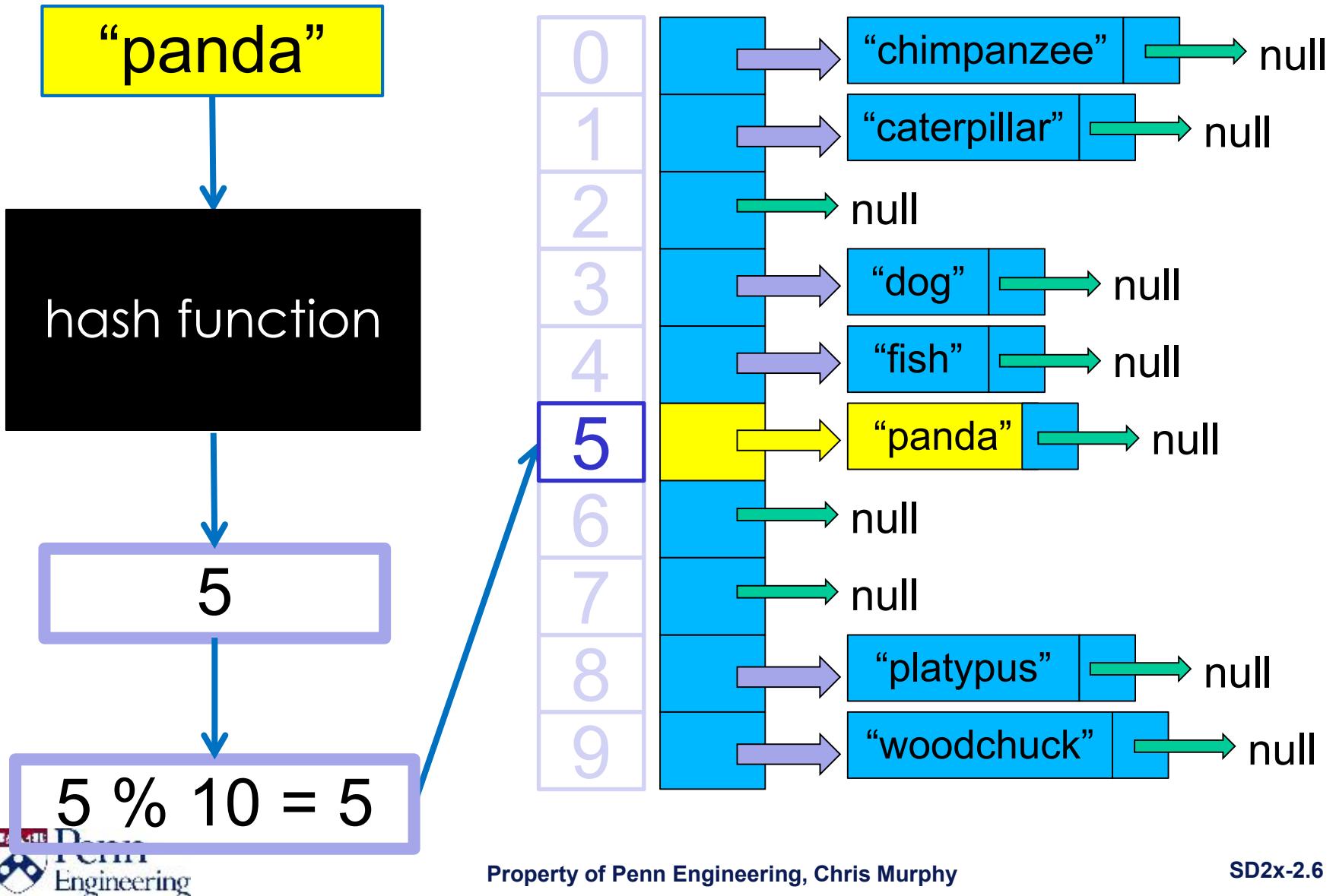
# Finding an element in HashSet



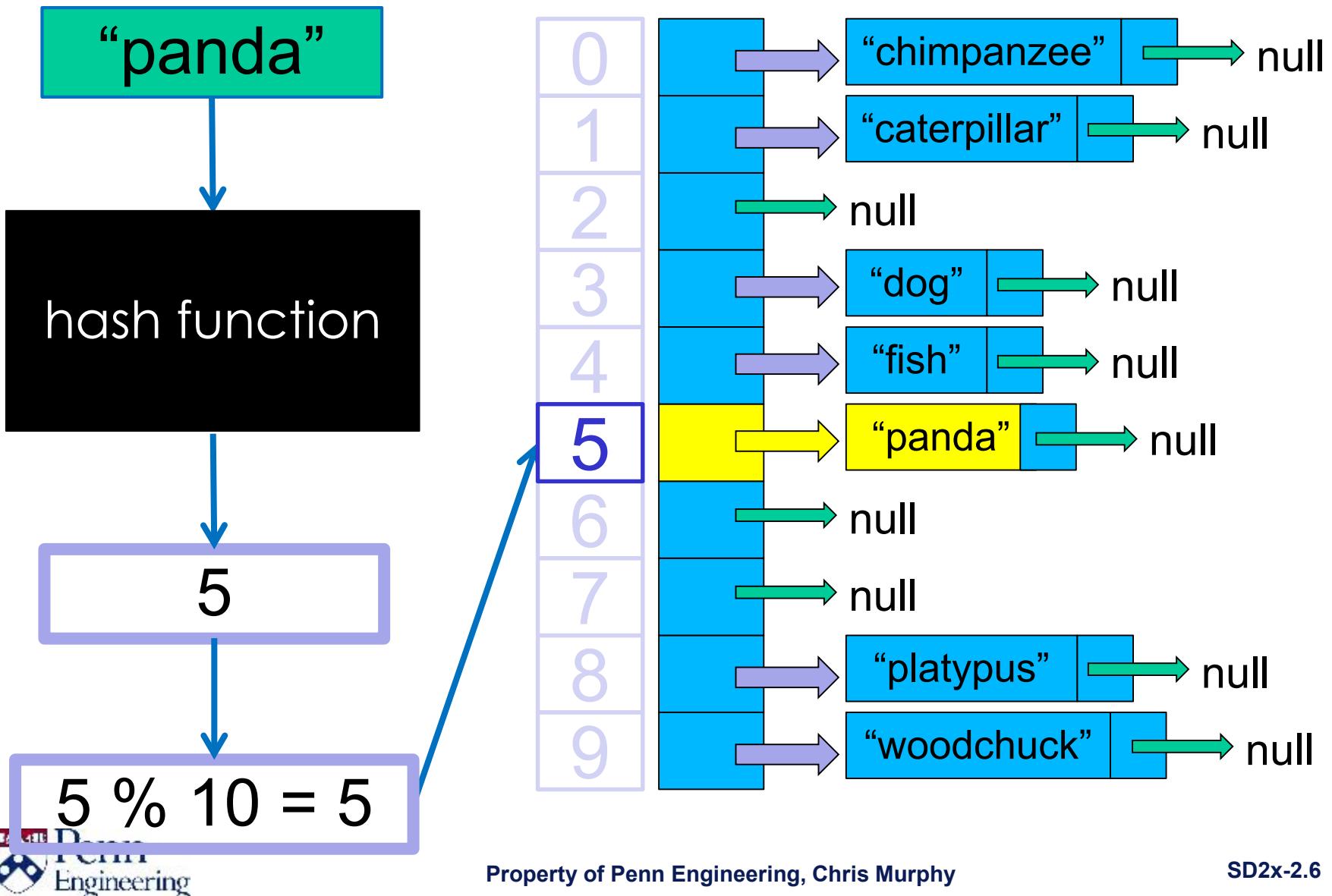
# Finding an element in HashSet



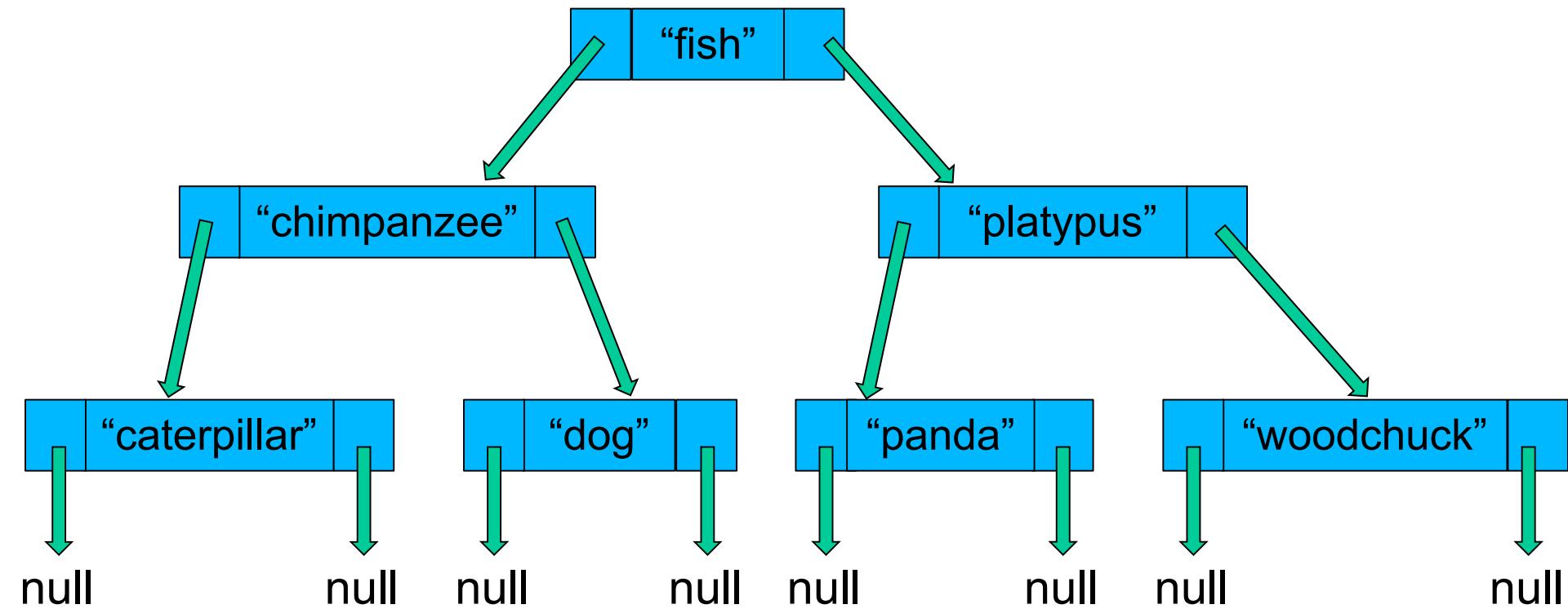
# Finding an element in HashSet



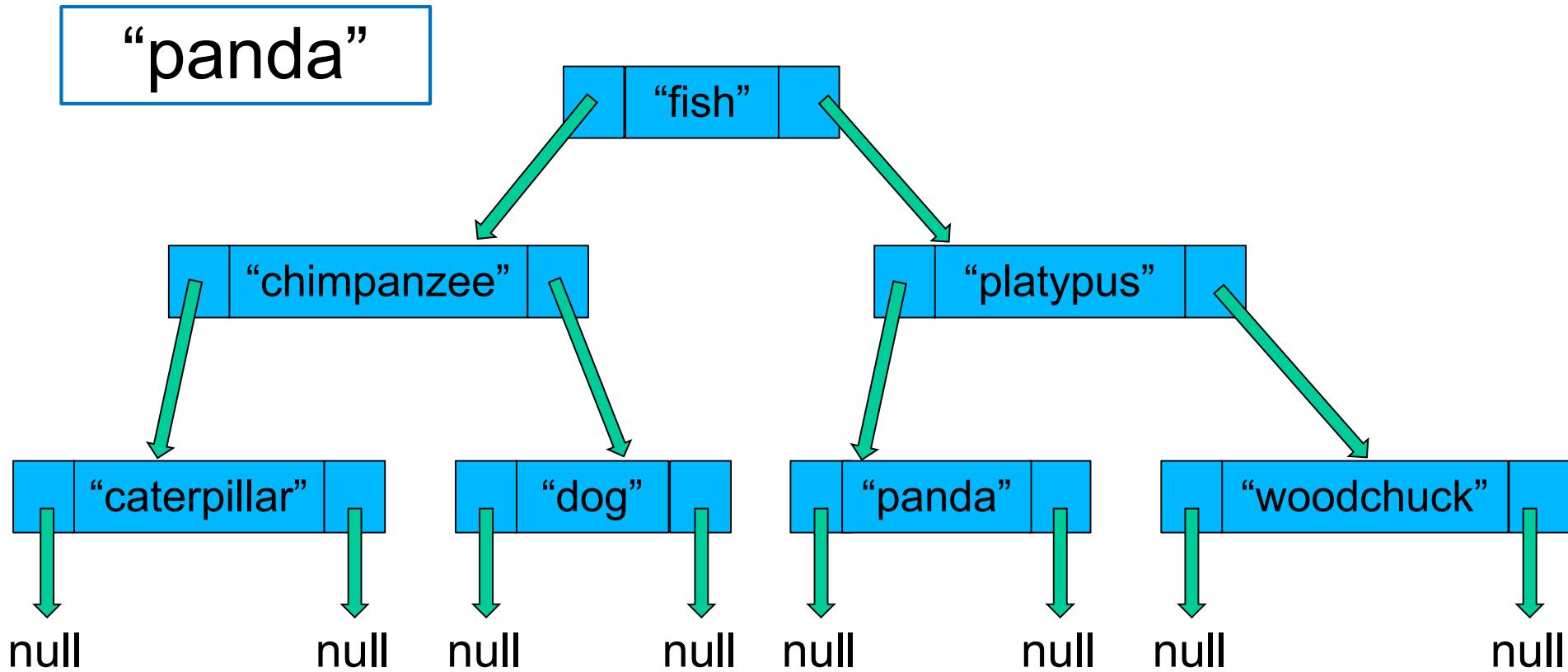
# Finding an element in HashSet



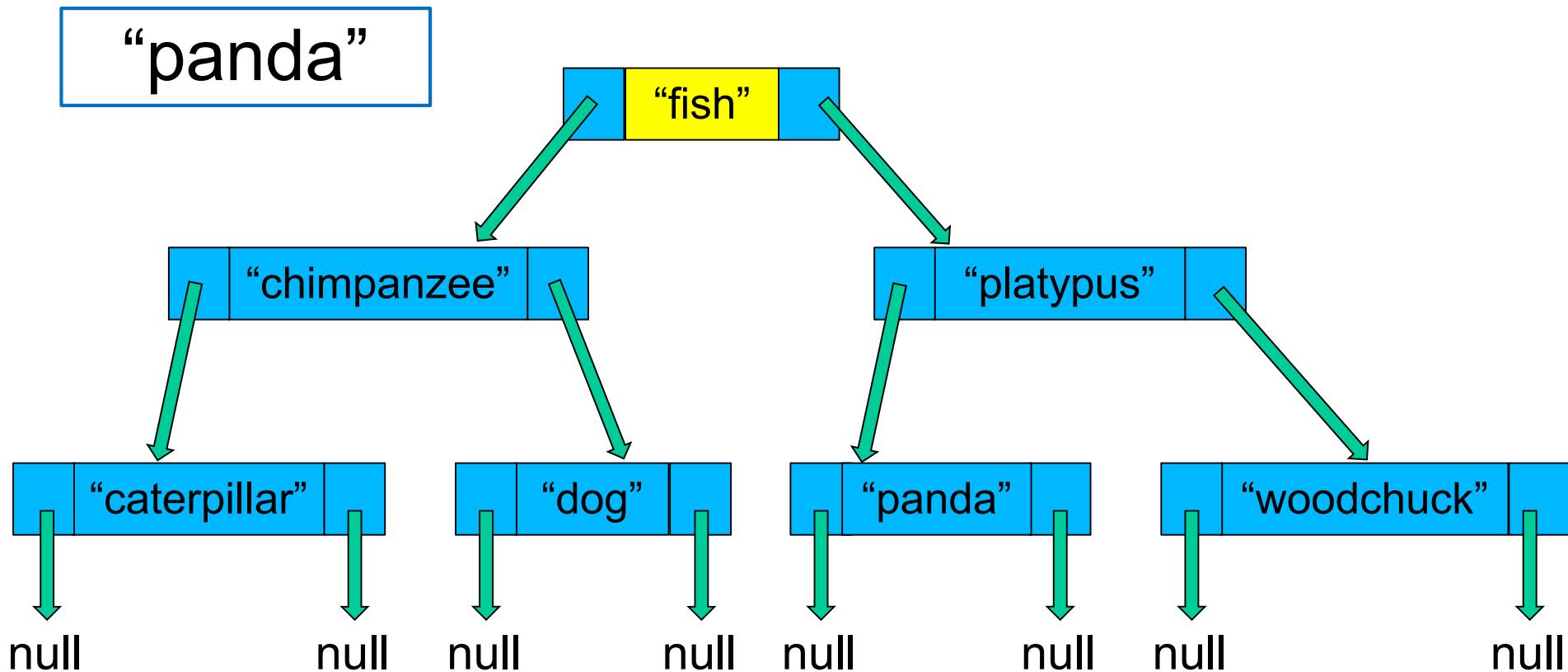
# Finding an element in TreeSet



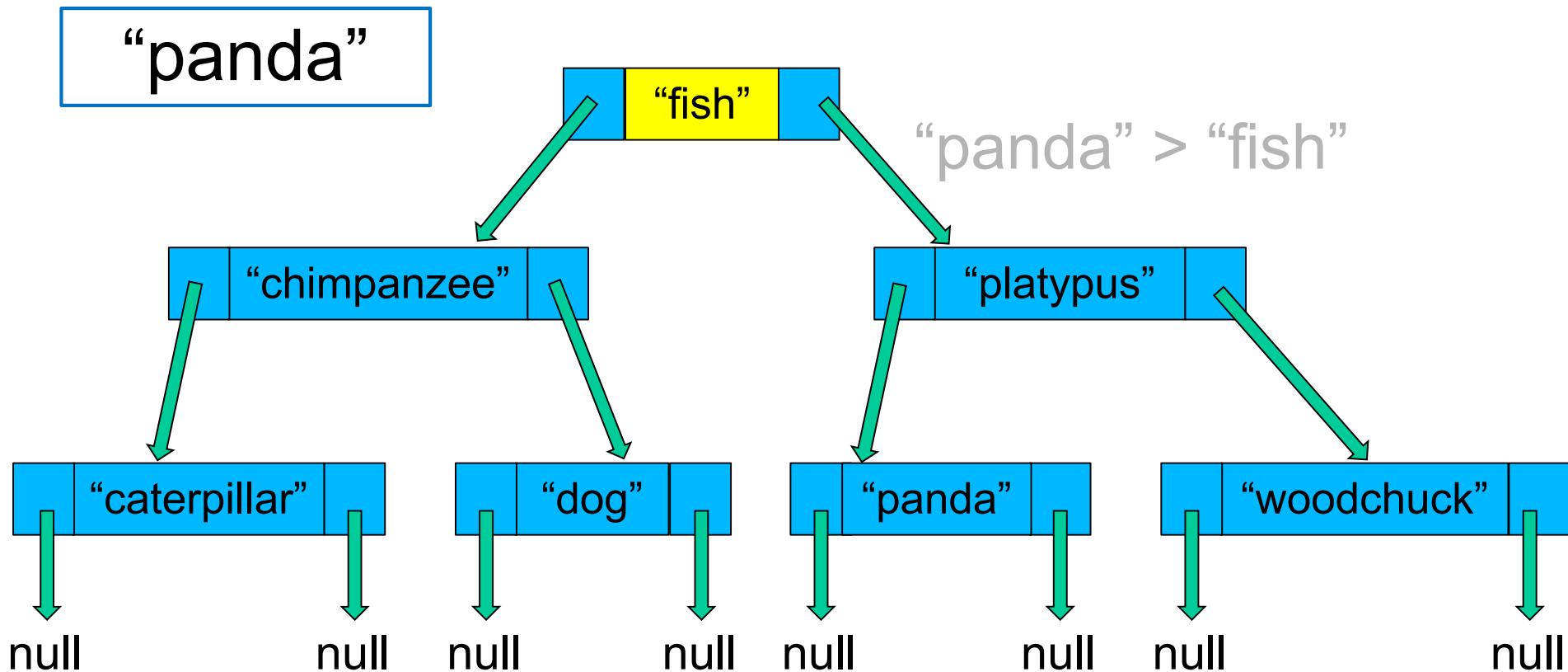
# Finding an element in TreeSet



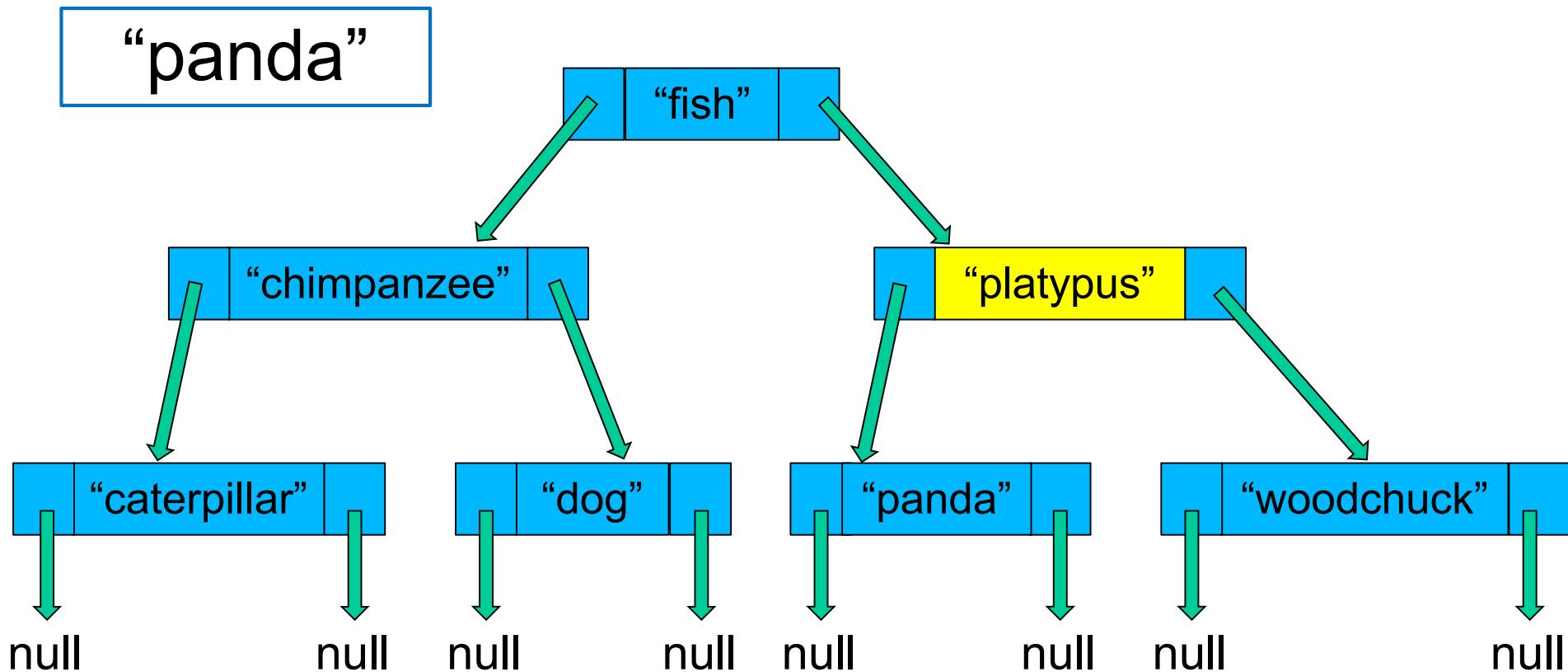
# Finding an element in TreeSet



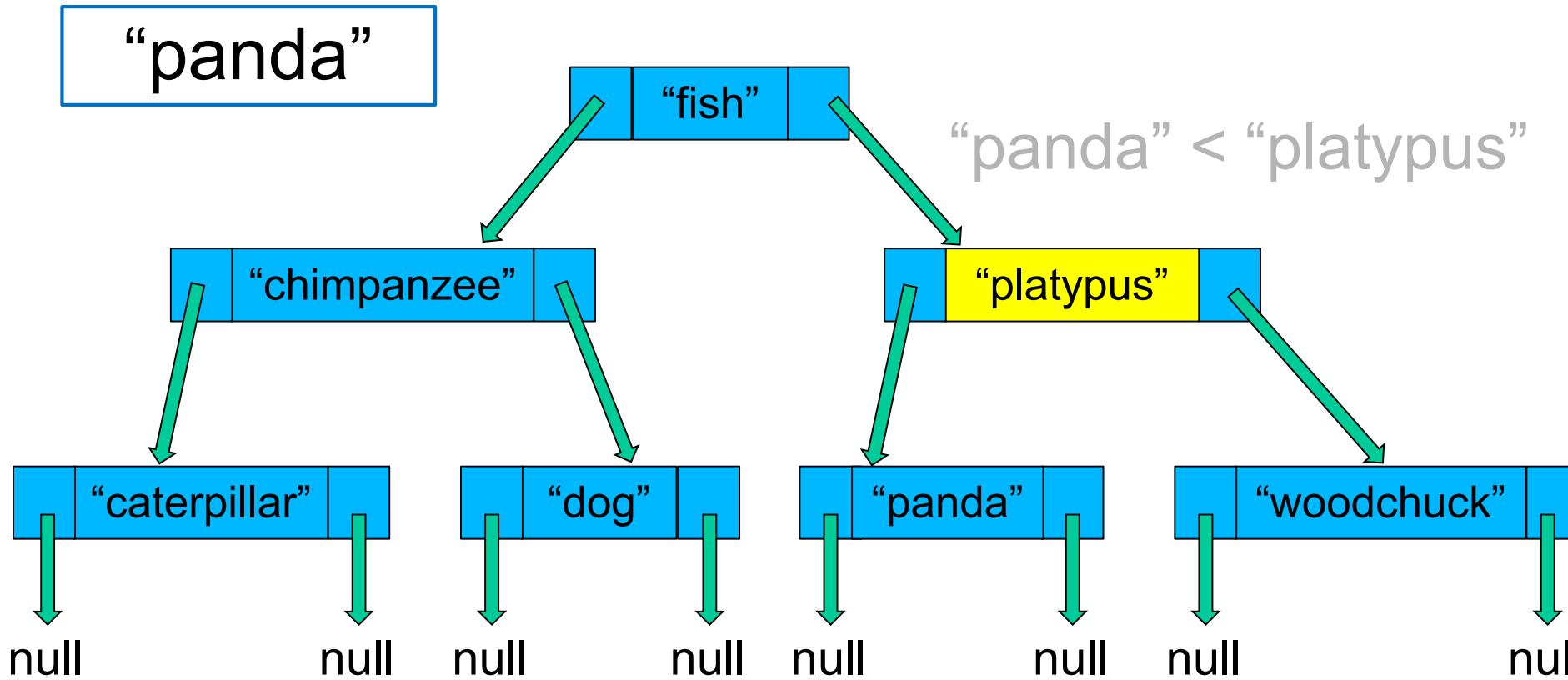
# Finding an element in TreeSet



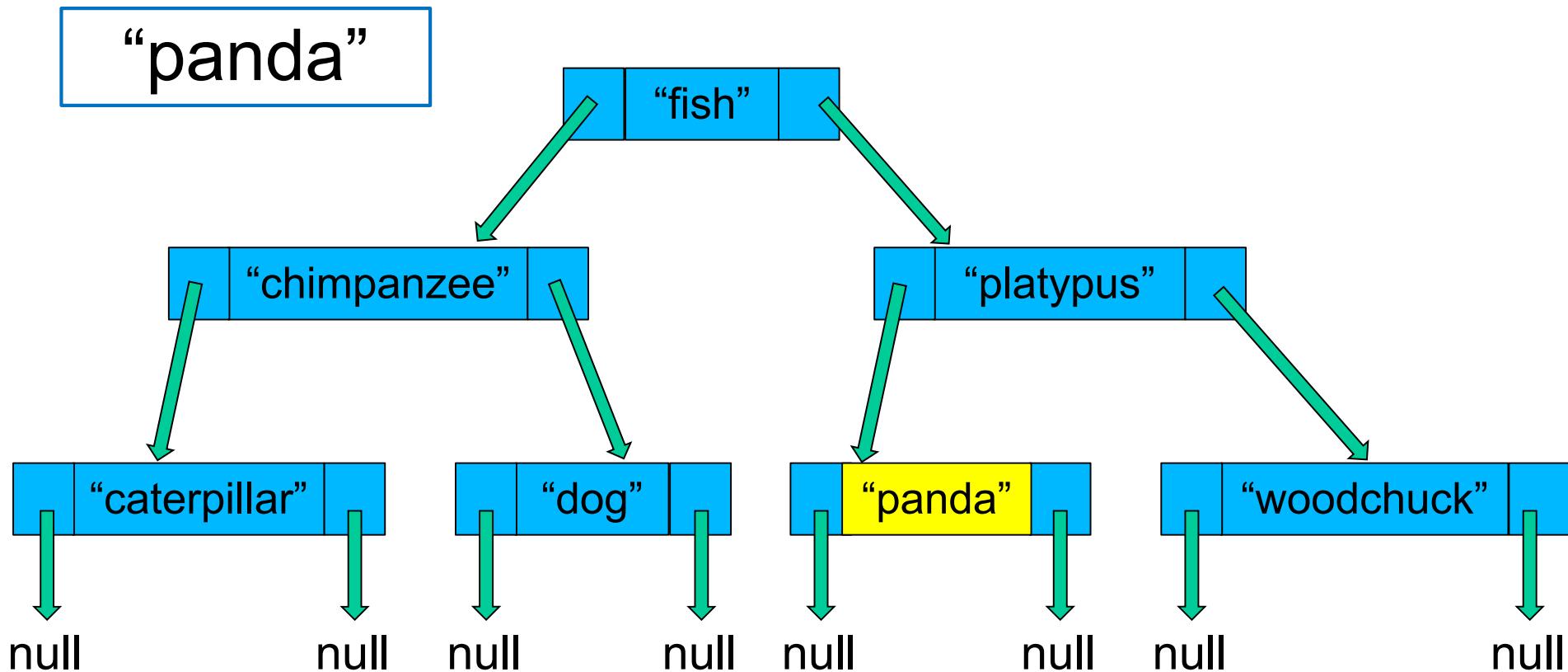
# Finding an element in TreeSet



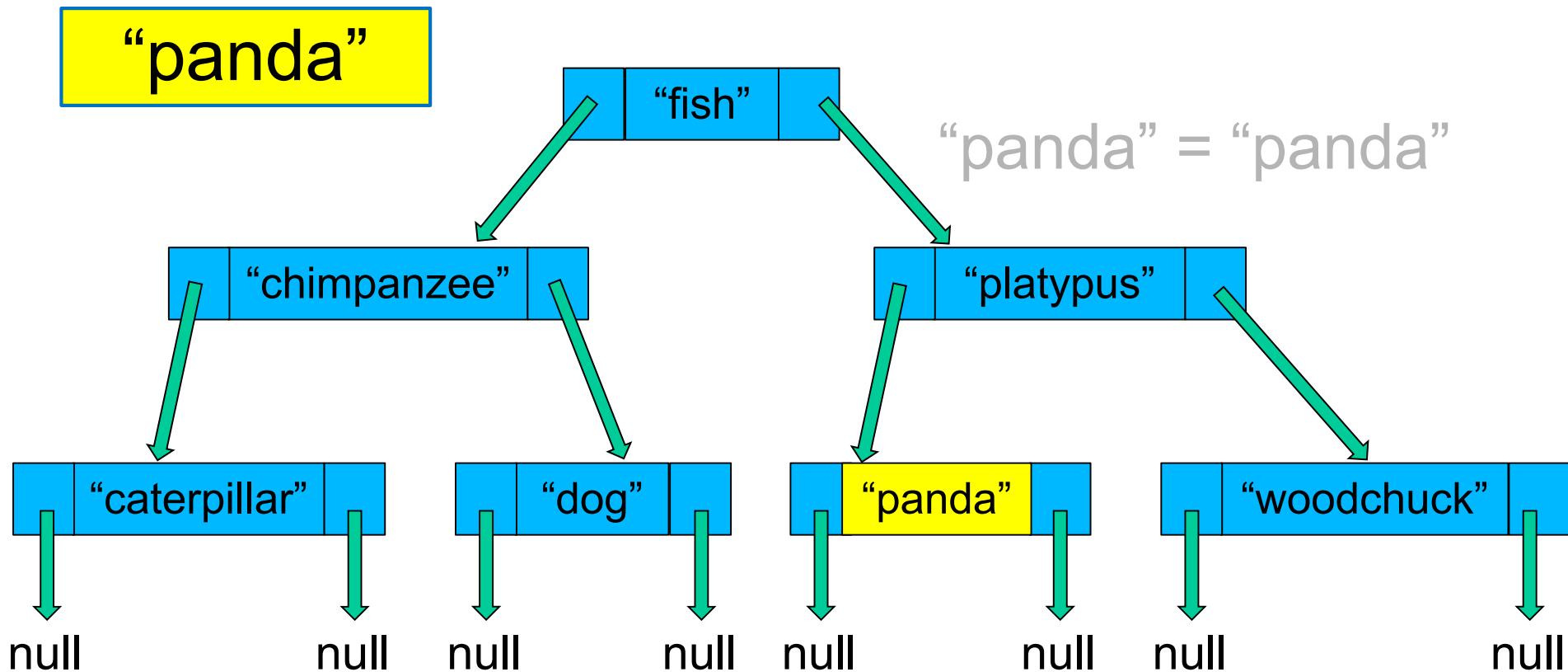
# Finding an element in TreeSet



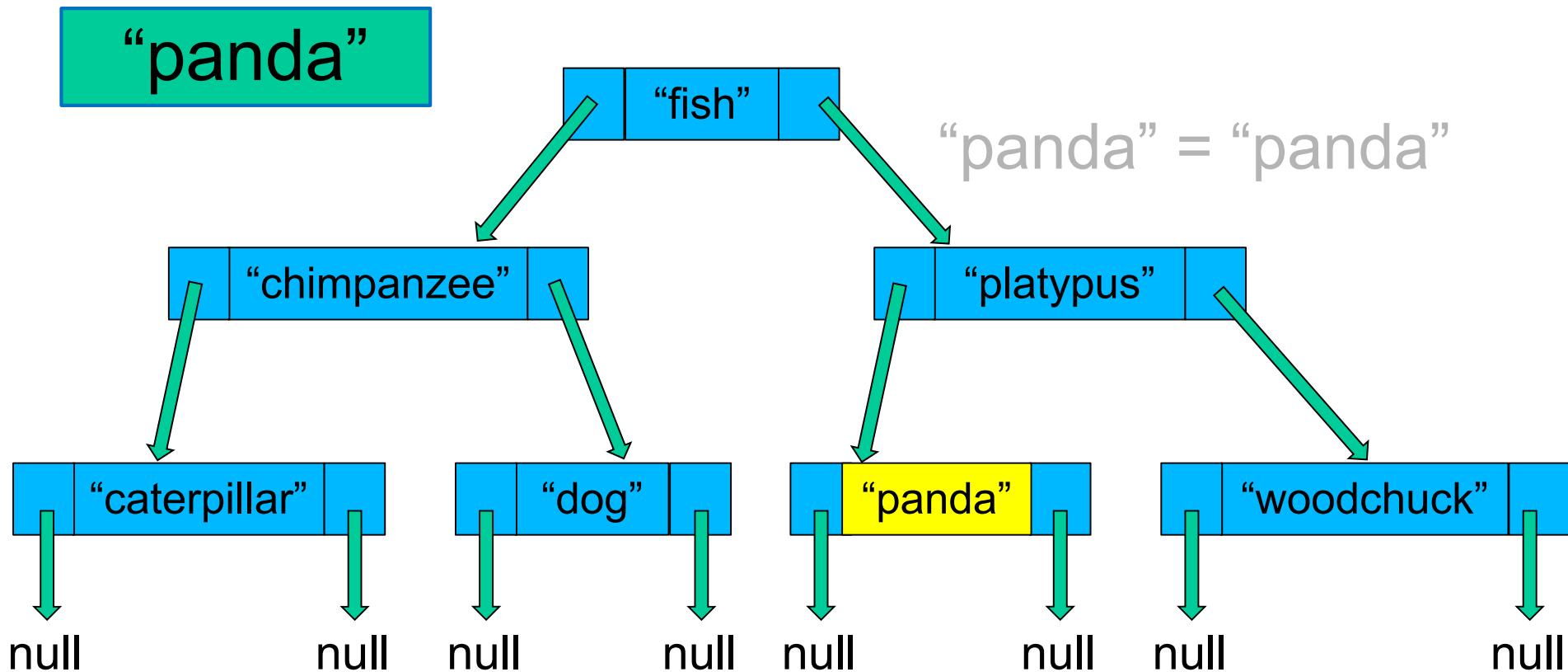
# Finding an element in TreeSet



# Finding an element in TreeSet



# Finding an element in TreeSet

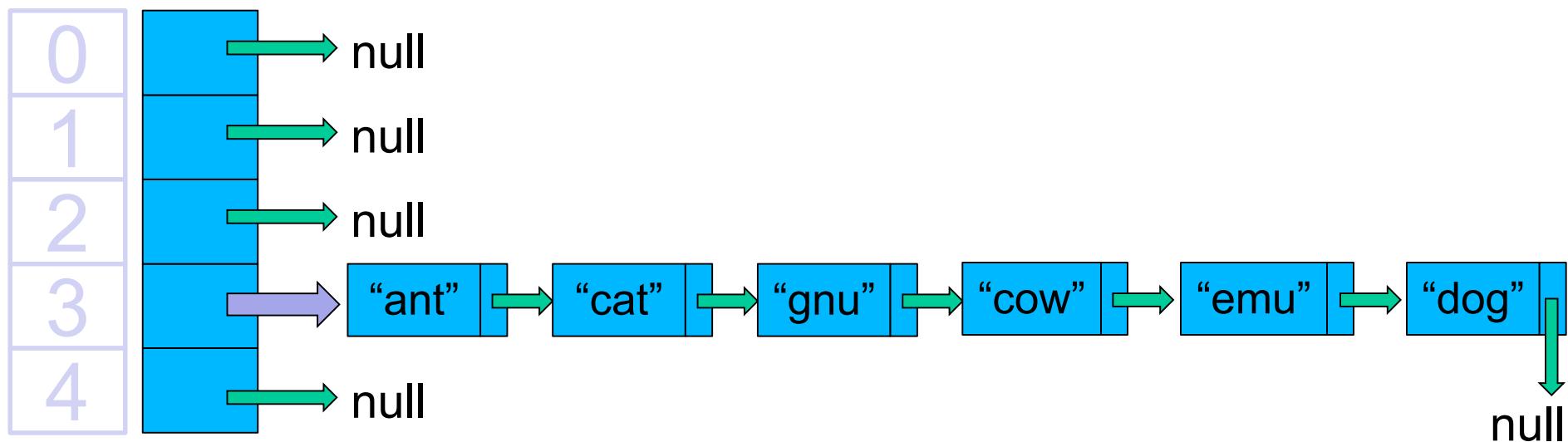


# Comparing HashSet vs. TreeSet

---

- Let's say we have a HashSet of  $n$  strings
  - And a TreeSet containing the same strings
- 
- In the **worst case**, how many steps do we need in order to determine whether the set contains a given string?
  - TreeSet:  $O(\log_2 n)$
  - HashSet:  $O(1)$
  - **But only if each bucket contains zero or one elements!**

# A very unlucky HashSet



# **SD2x2.7**

## **Heaps properties, add, remove**

### **Kathy**

# Motivating Example

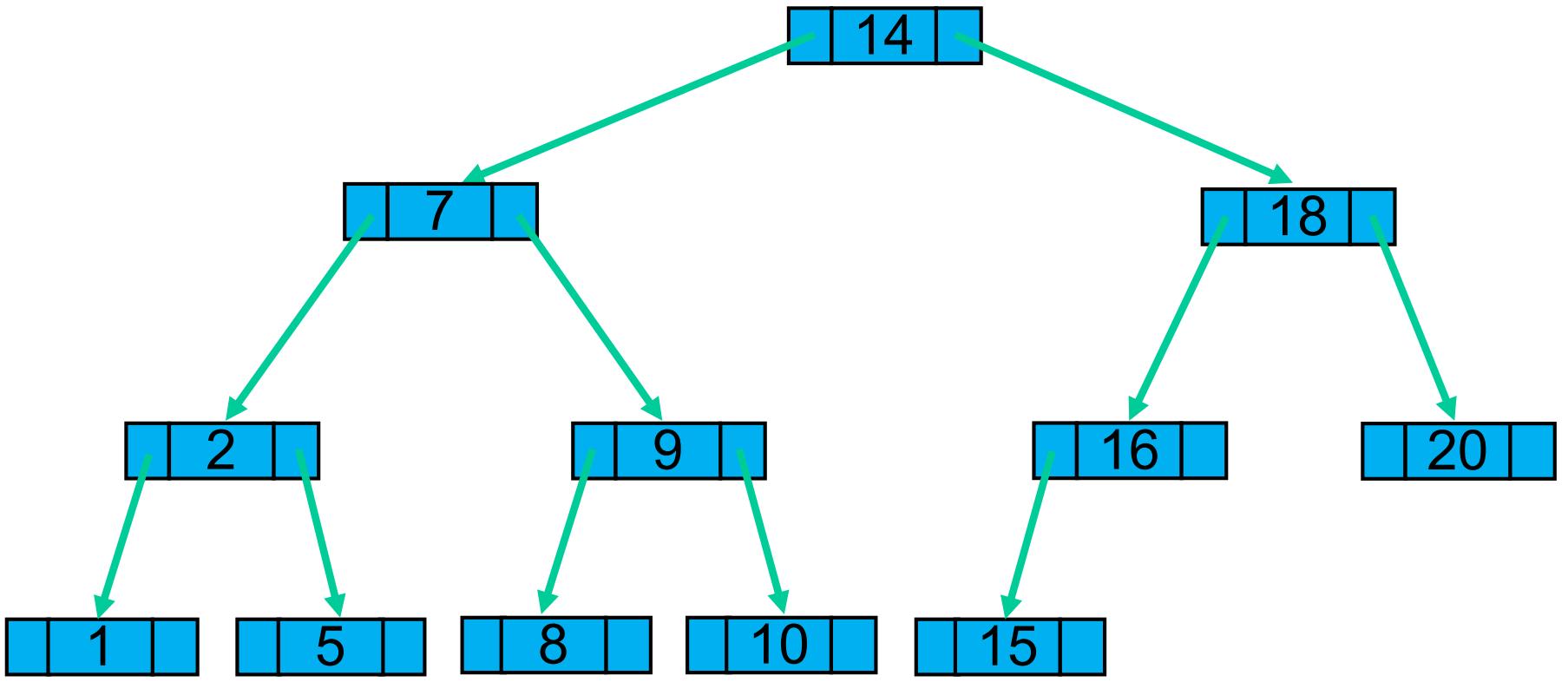
---

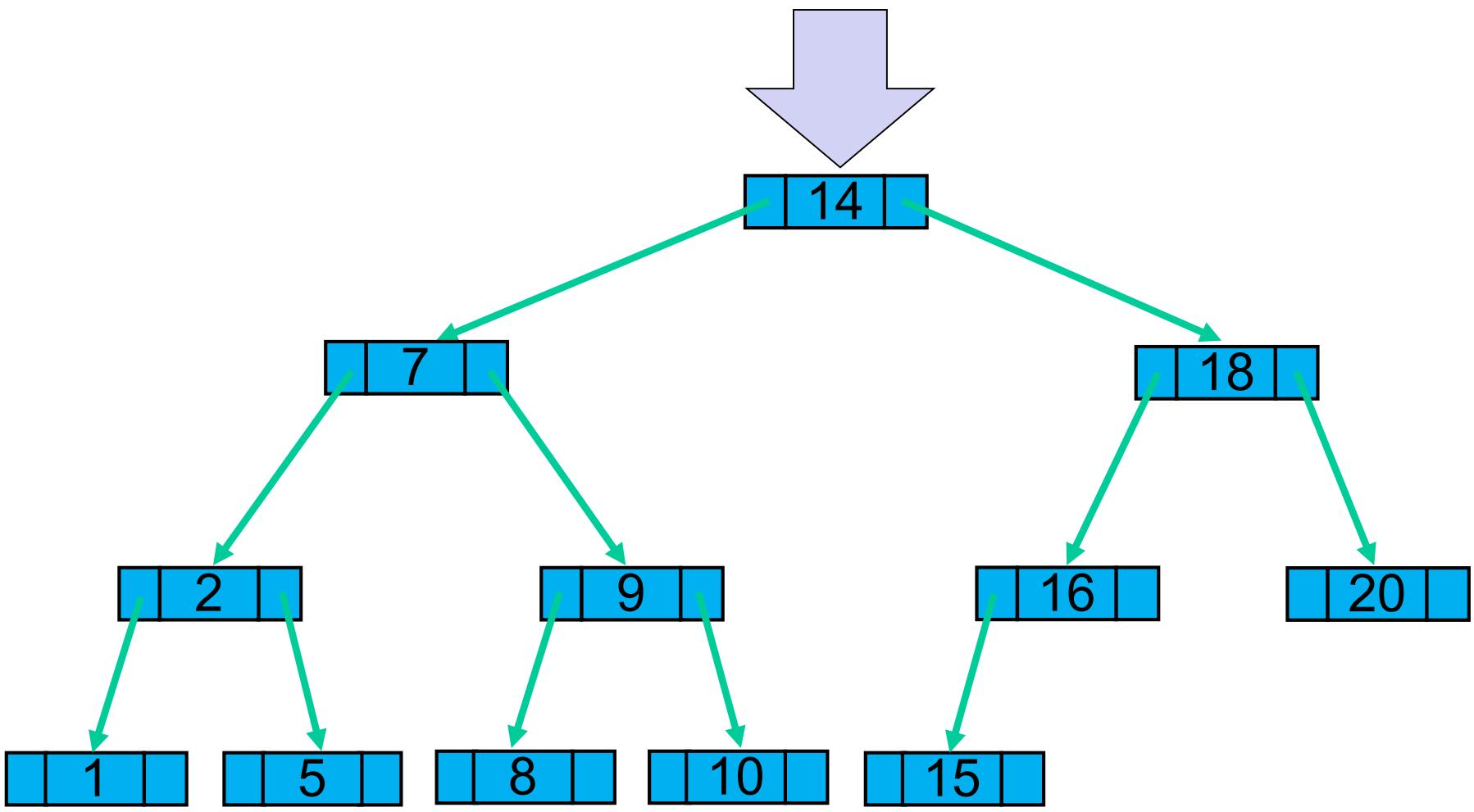
- Let's say we're keeping track of a set of numbers
- How can we quickly find the **largest** value?
- We could use a **sorted LinkedList**
  - “Tail” is always largest value:  $O(1)$
  - Inserting elements in order:  **$O(n)$**

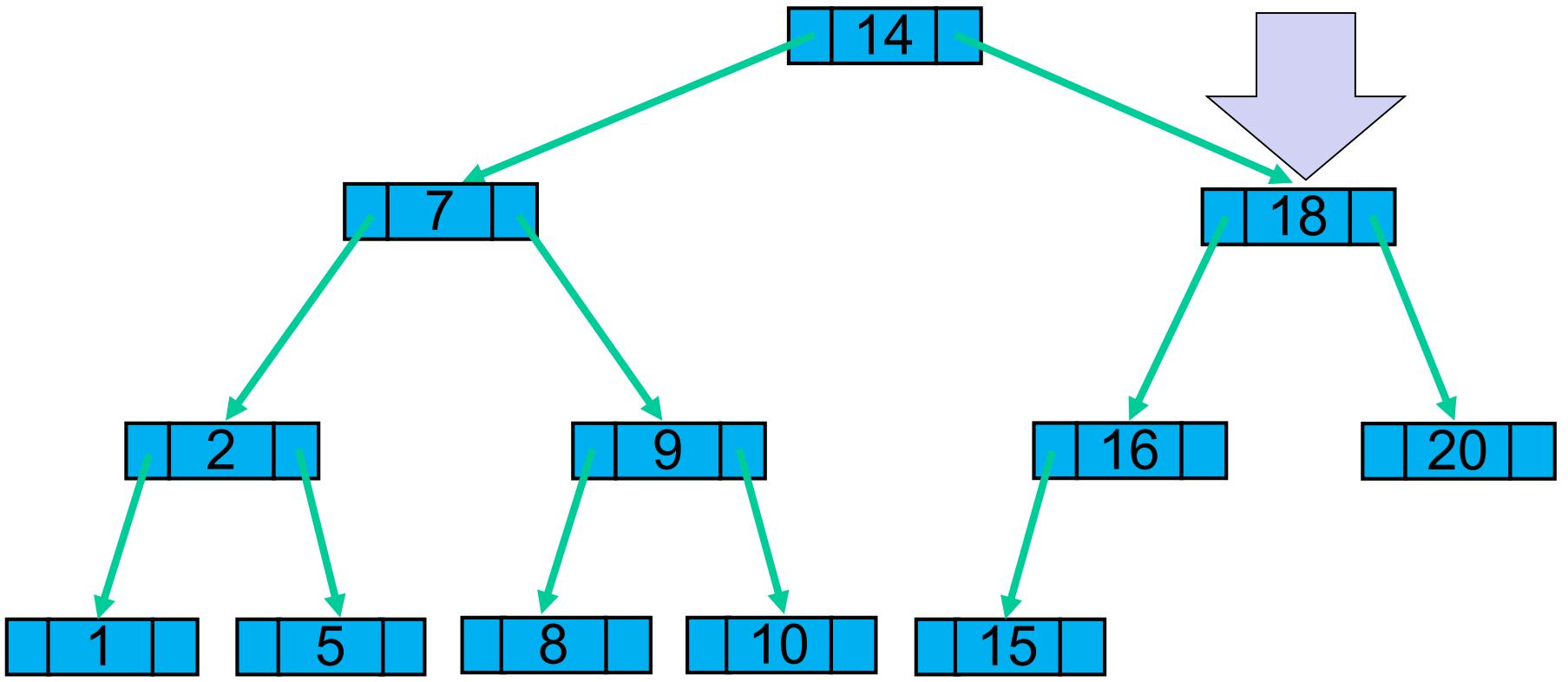
# Motivating Example

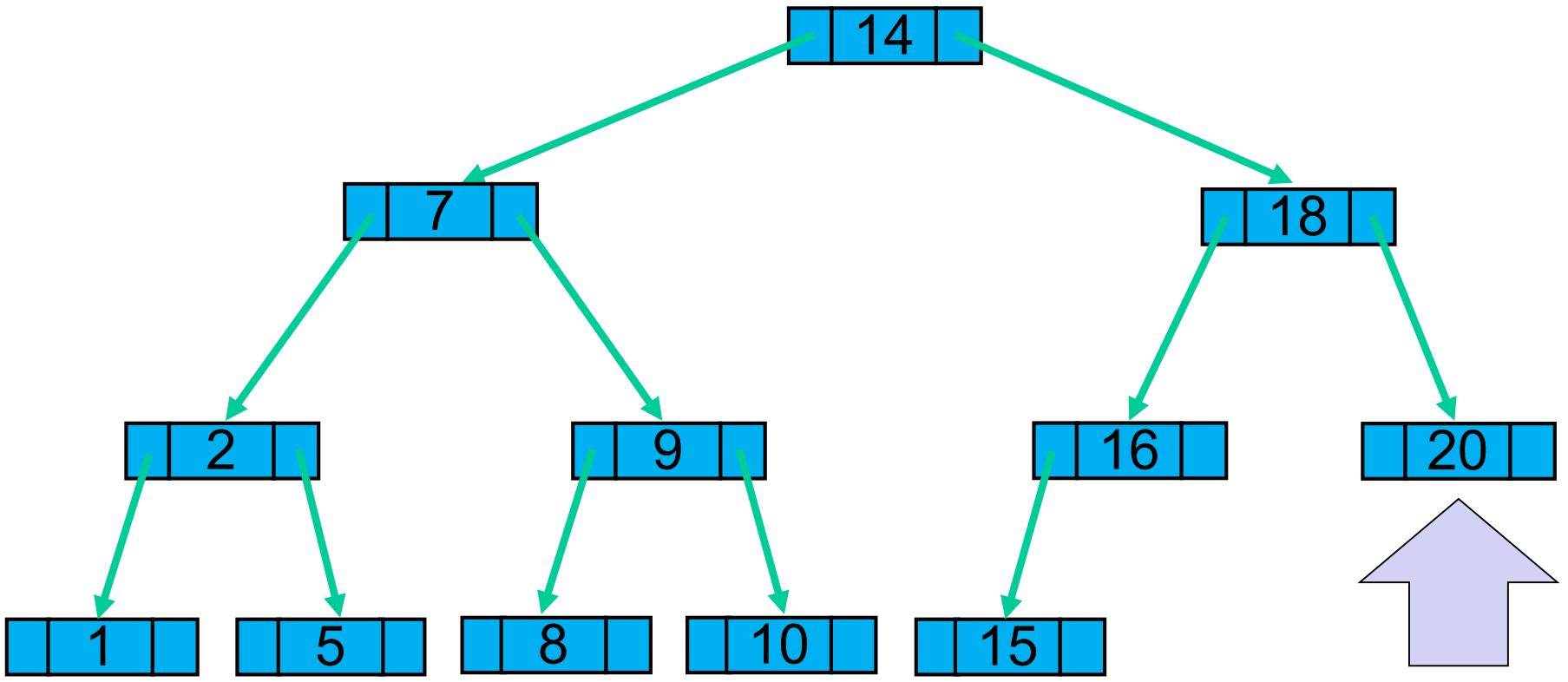
---

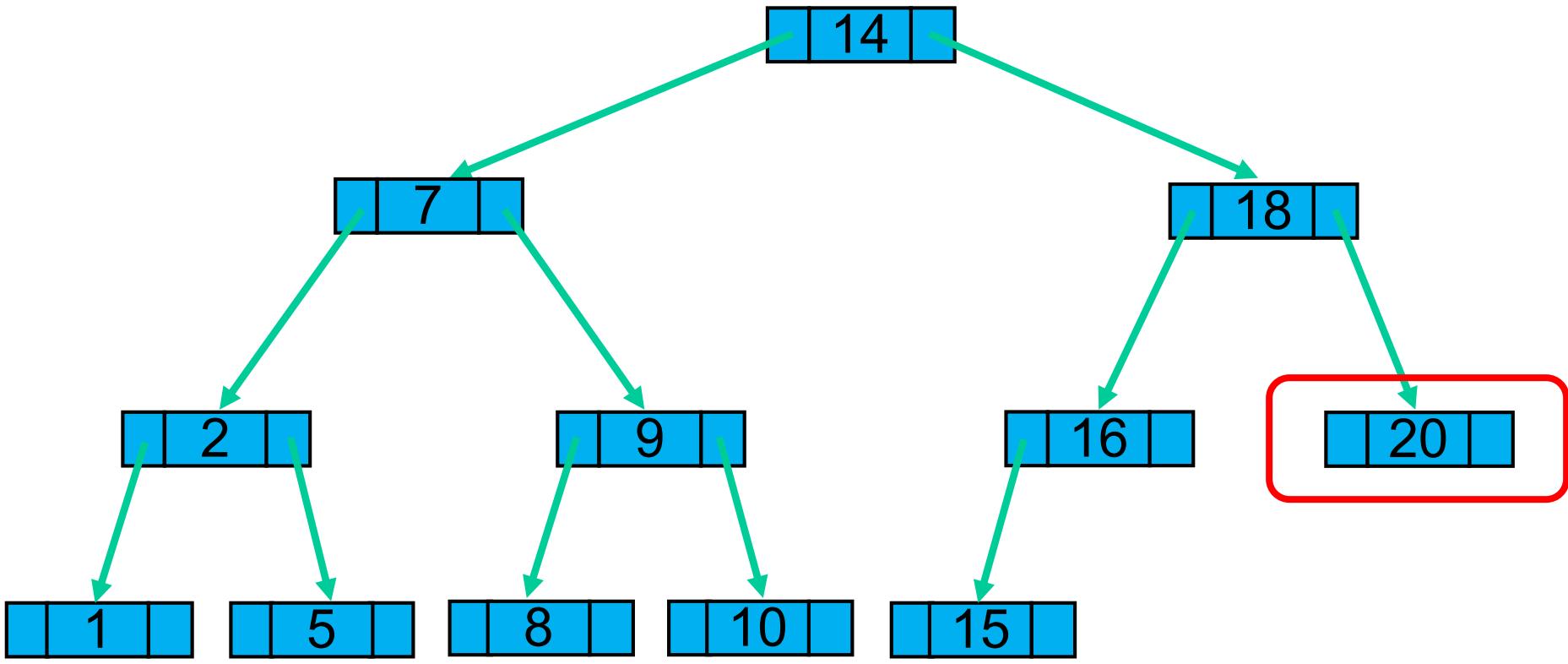
- Let's say we're keeping track of a set of numbers
- How can we quickly find the **largest** value?
- Should we use a **Binary Search Tree**?

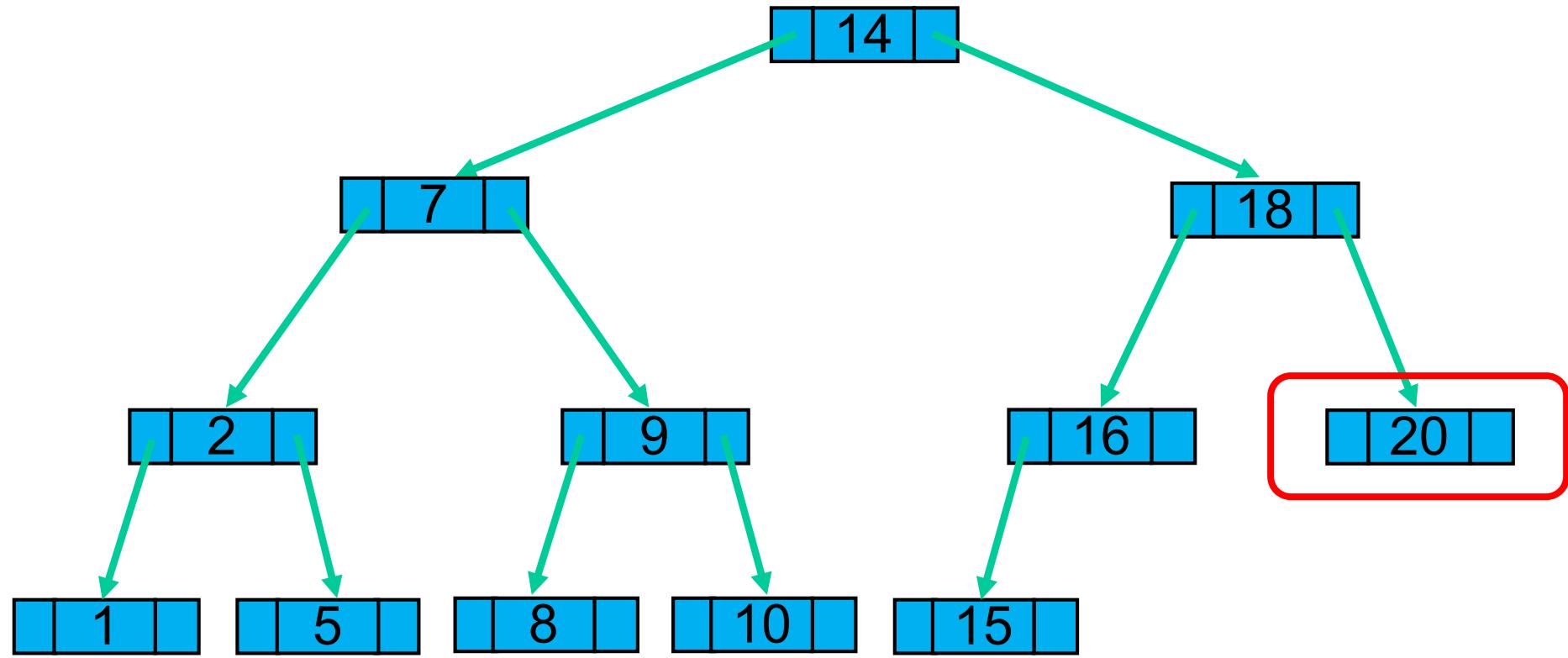




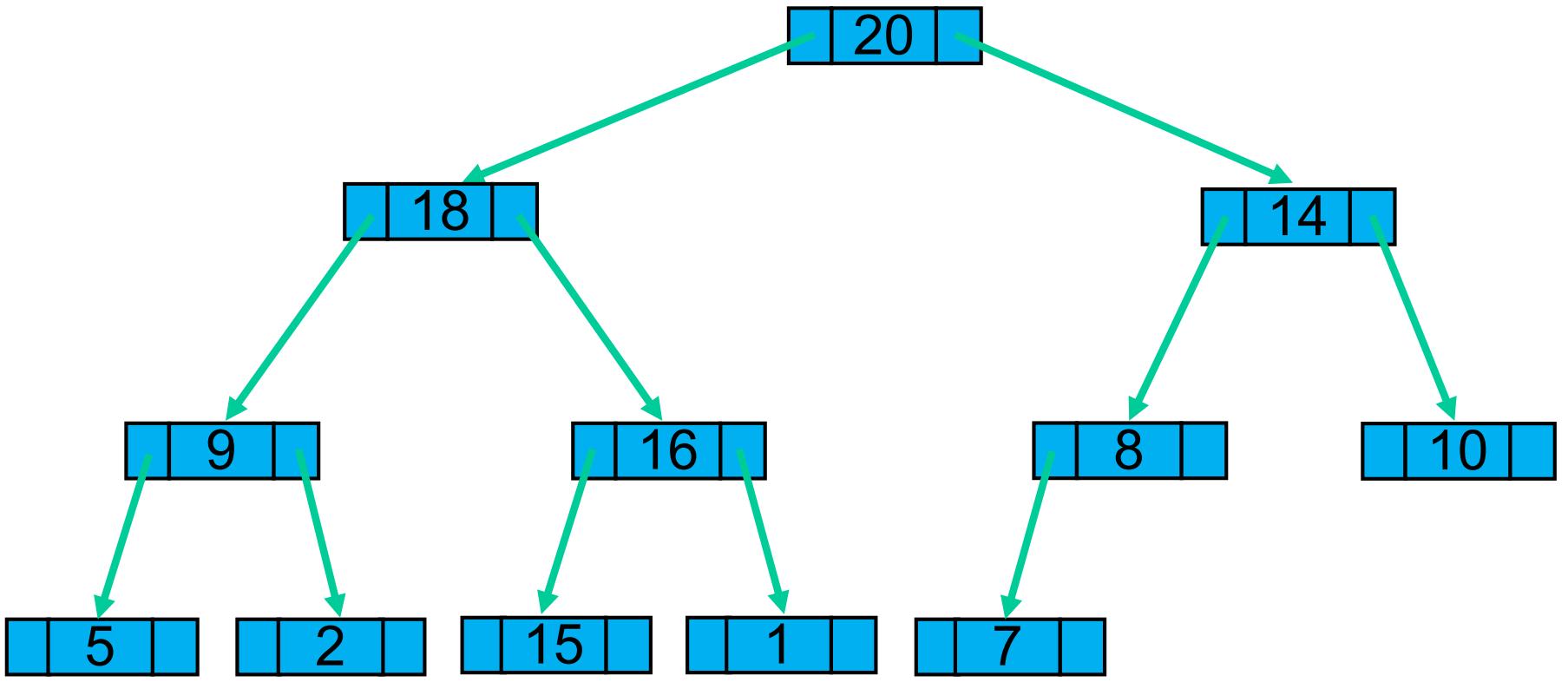


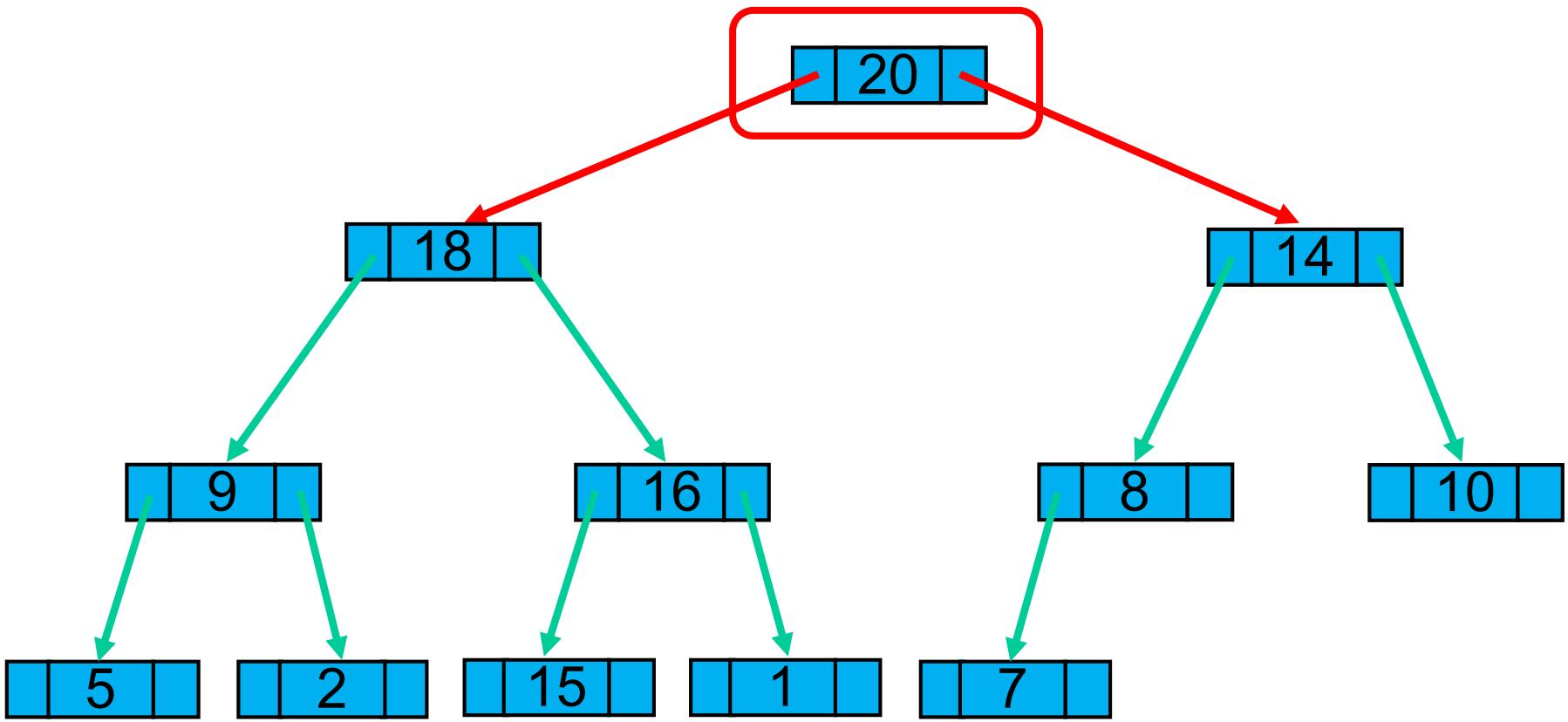


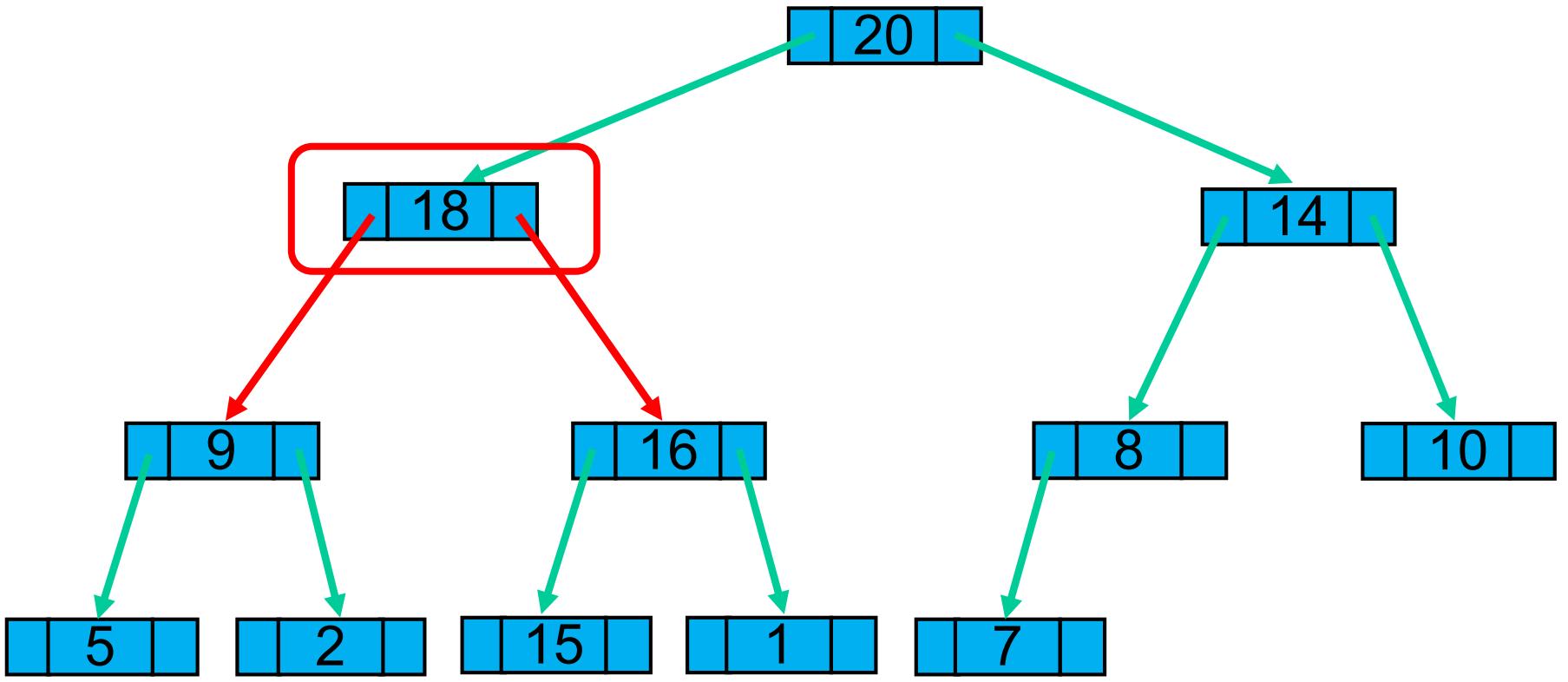


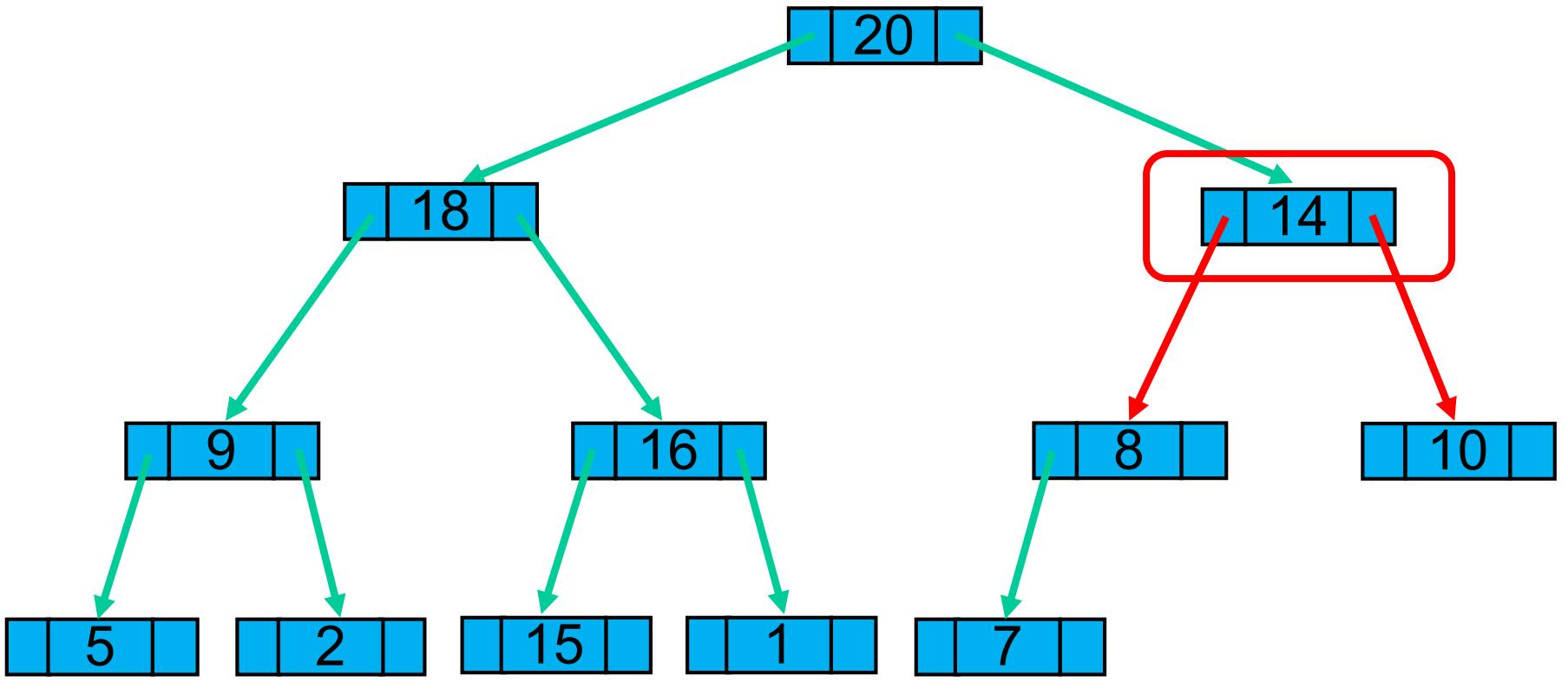


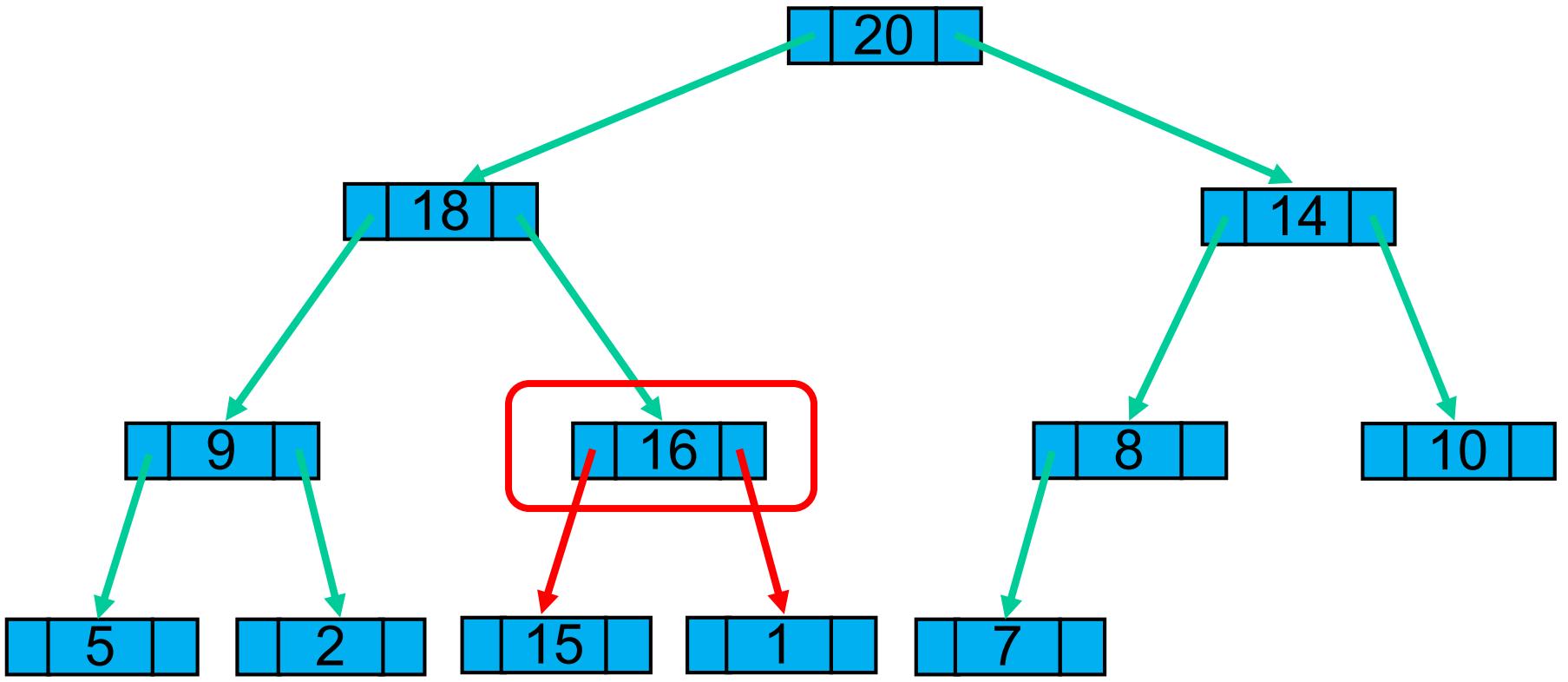
Both insert and find largest:  $O(\log_2 n)$

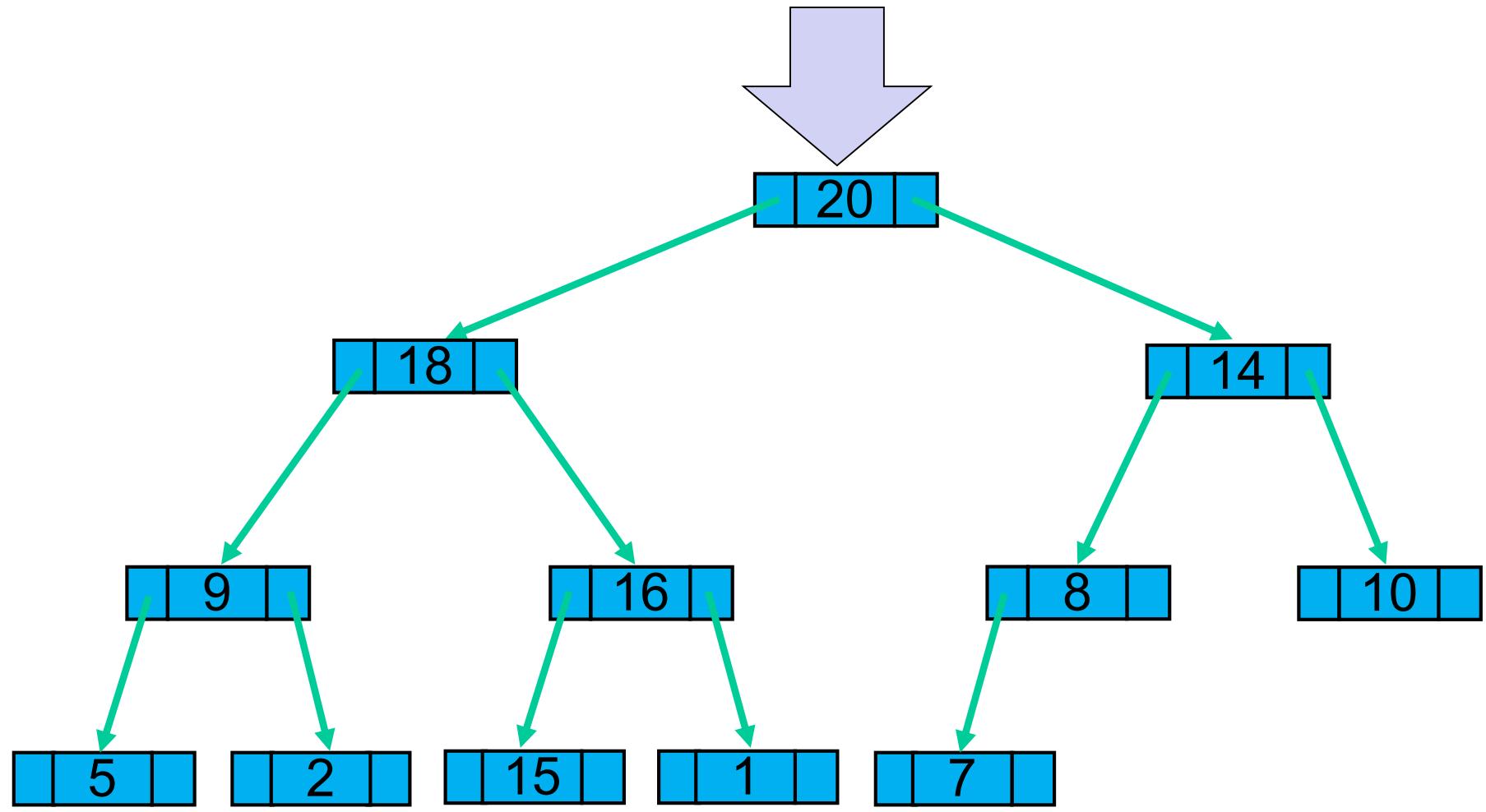


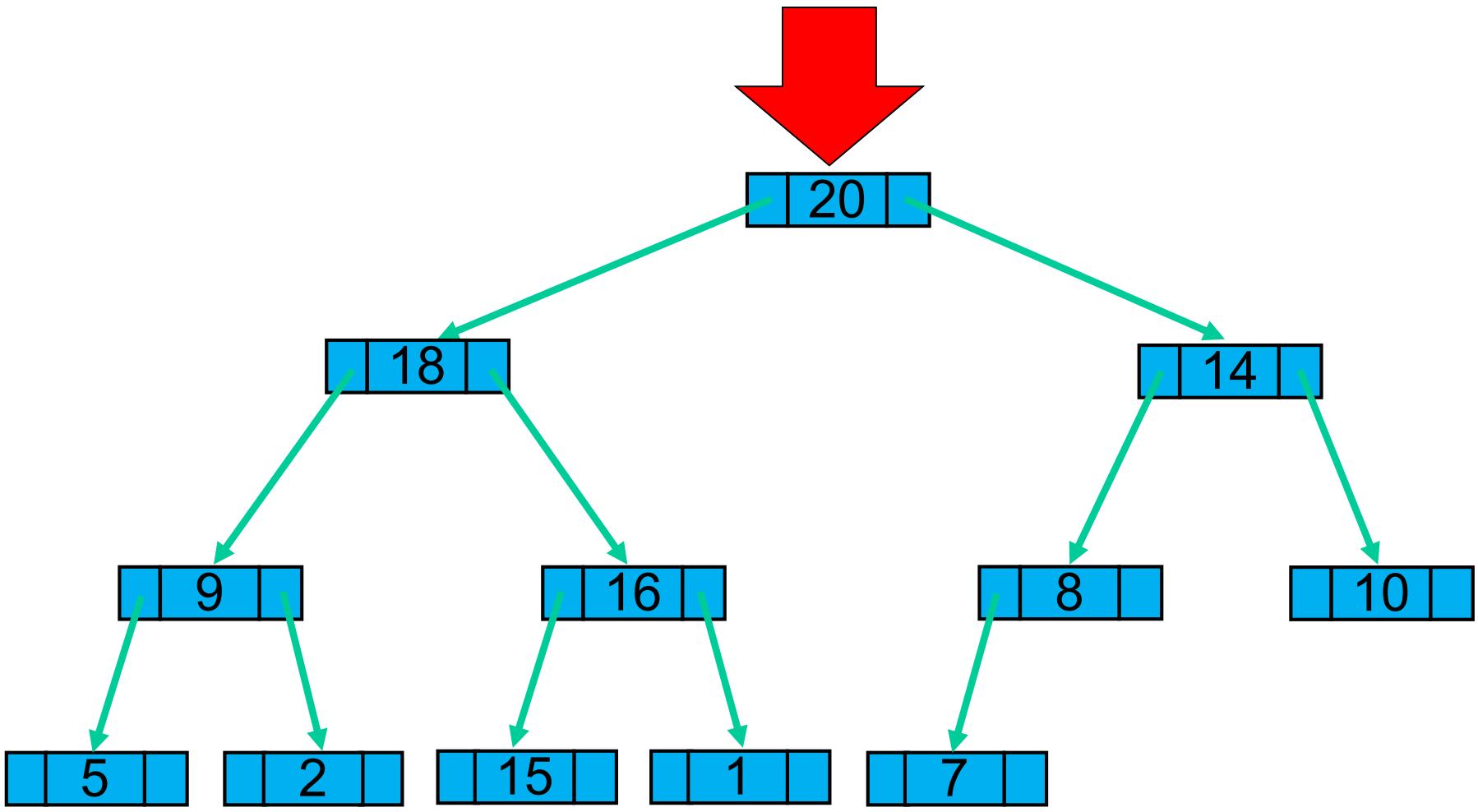


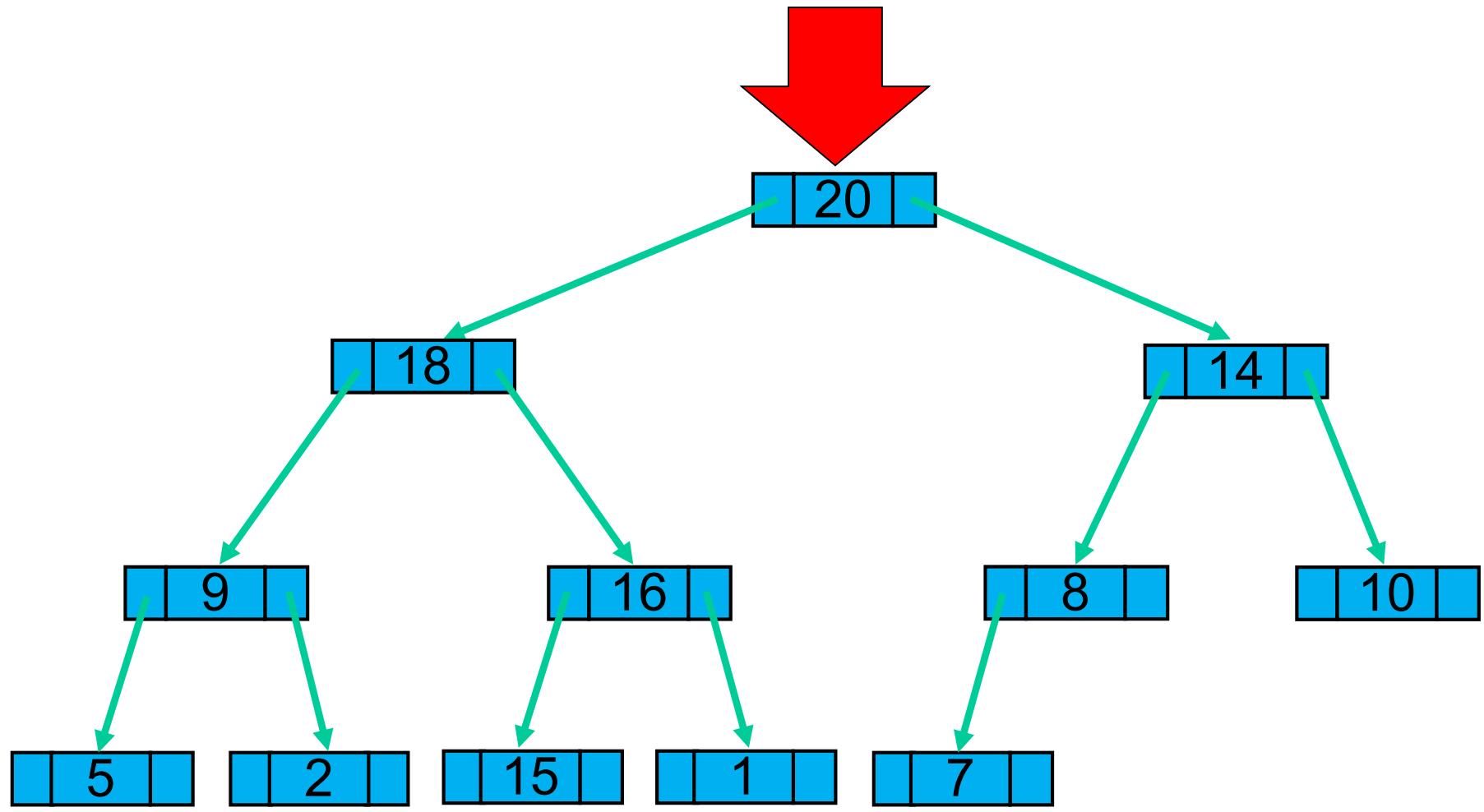




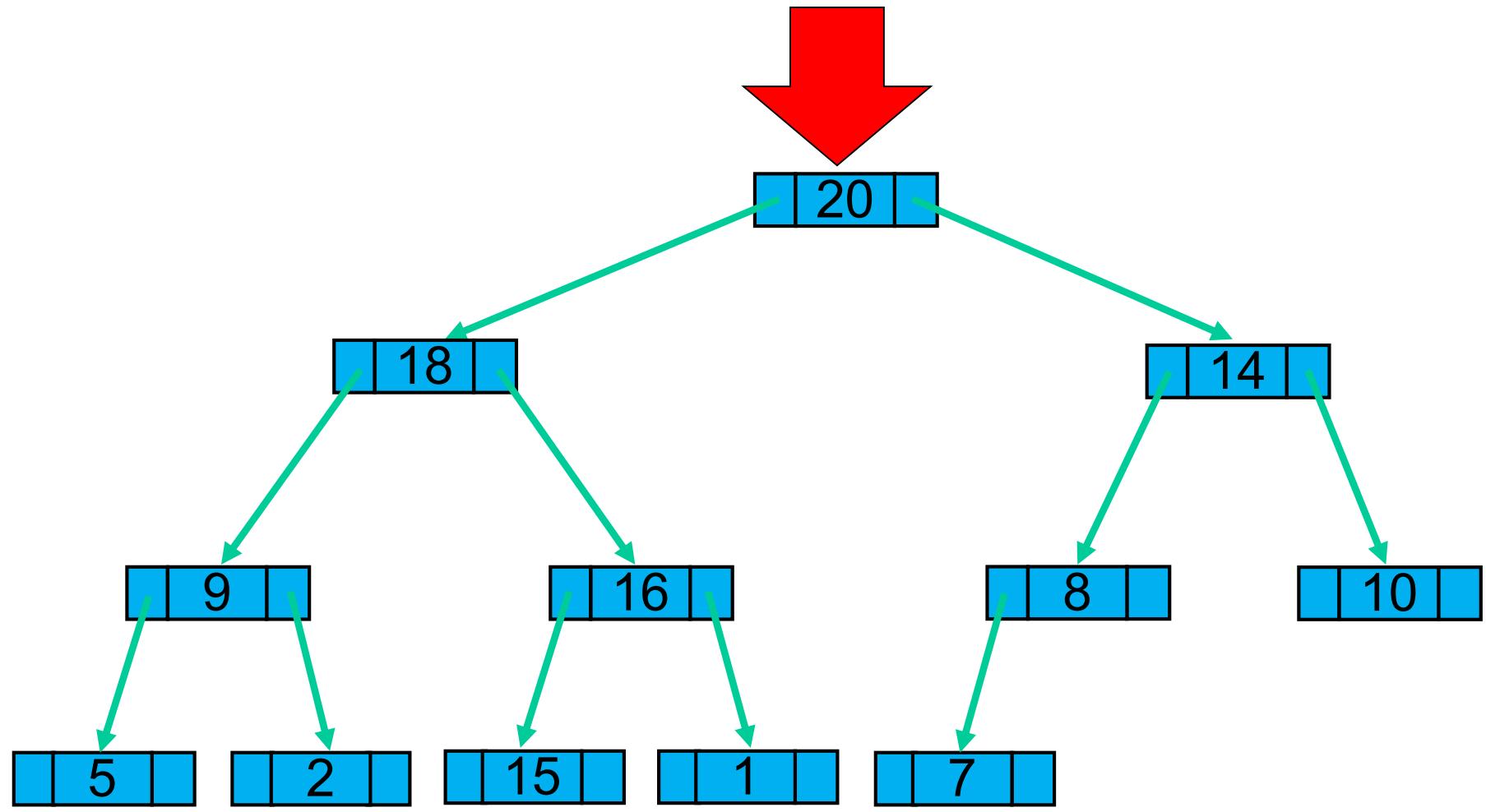








Insert:  $O(\log_2 n)$



Insert:  $O(\log_2 n)$

Find largest:  $O(1)$



# Heap!

# Heaps

---

- A heap is a **complete** binary tree
  - Each level of the tree is filled before the next
  - Each level of the tree is filled in from left to right with no gaps
- A heap satisfies the **heap property**
  - Each node has a greater value than all of its descendants
  - Therefore, a heap is considered **partially ordered**

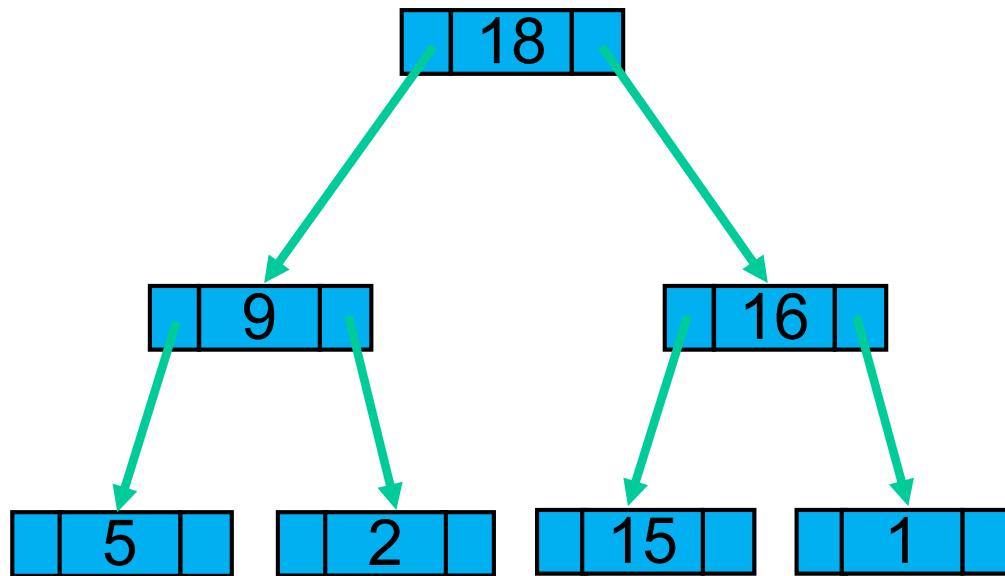
# Why Heaps?

---

- We can always get the node with the greatest value in  $O(1)$  time
- We can remove the node with the greatest value in  $O(\log_2 n)$  time
- We can add a value in  $O(\log_2 n)$  time

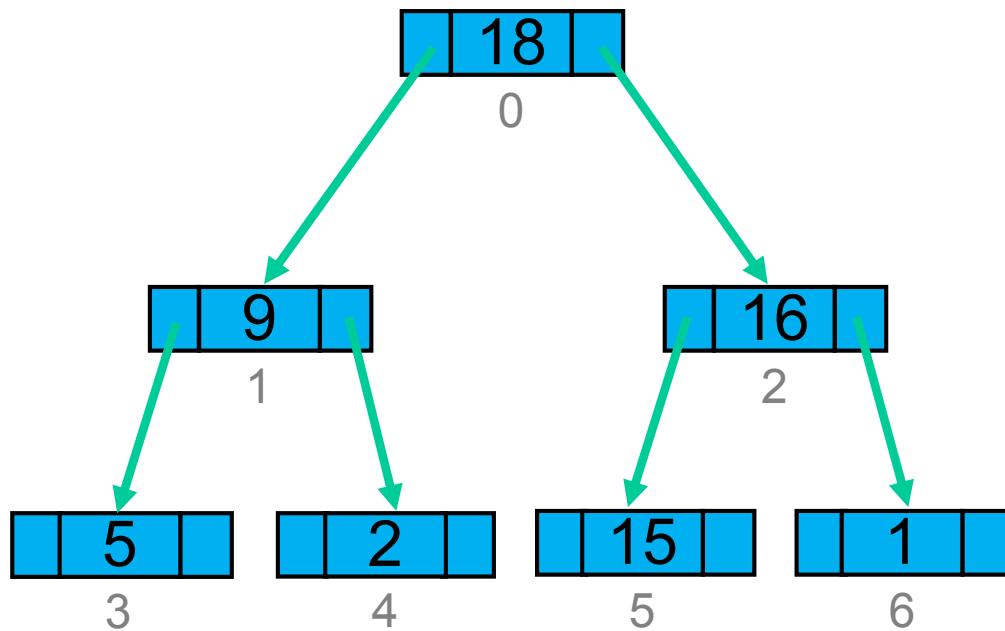
# Array Representation of Heaps

---

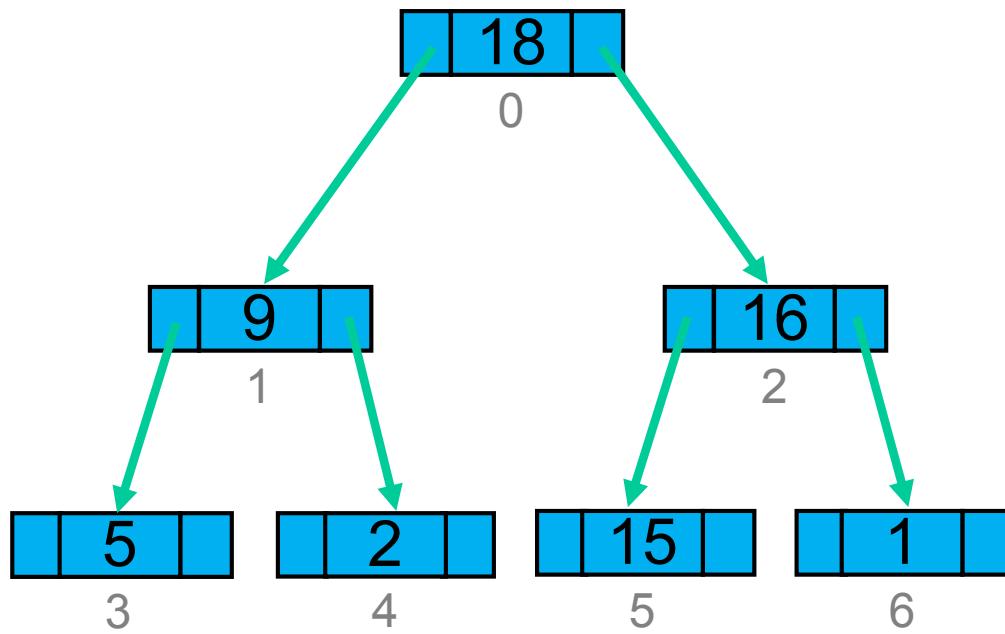


# Array Representation of Heaps

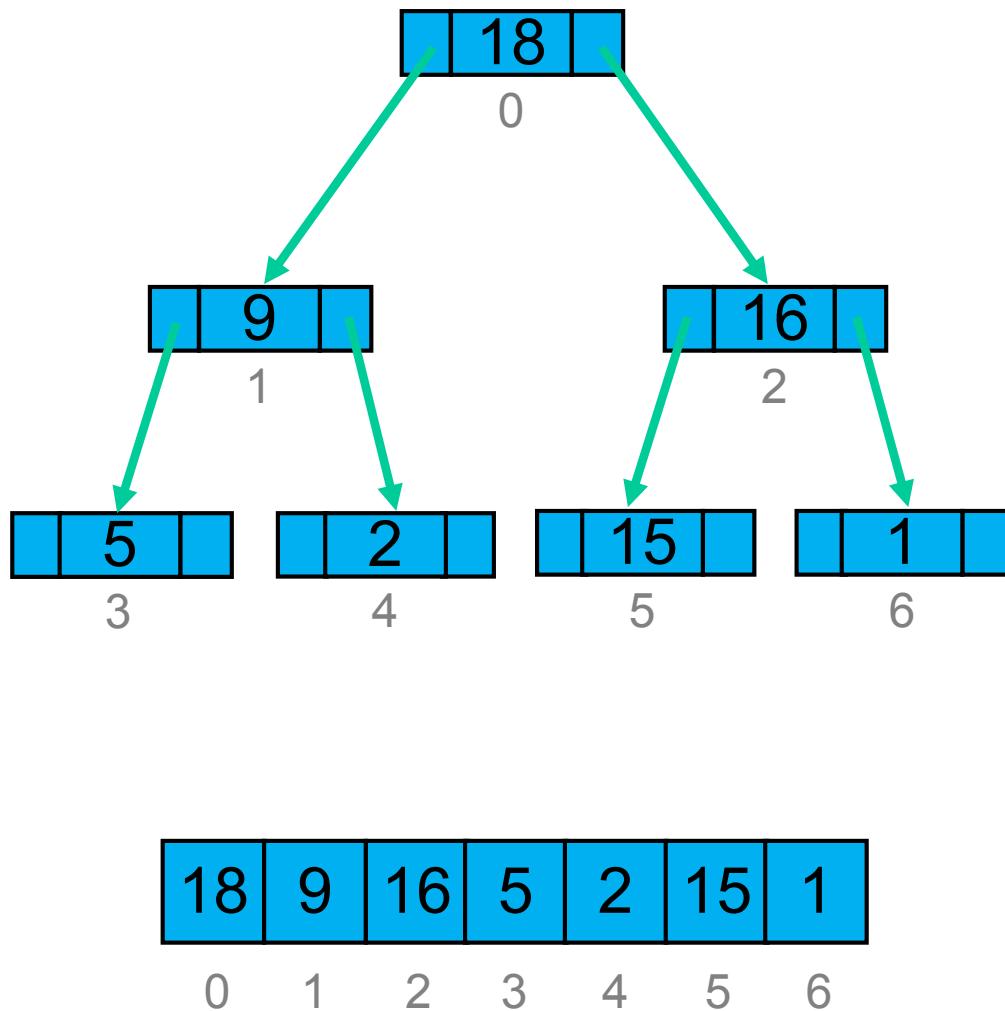
---



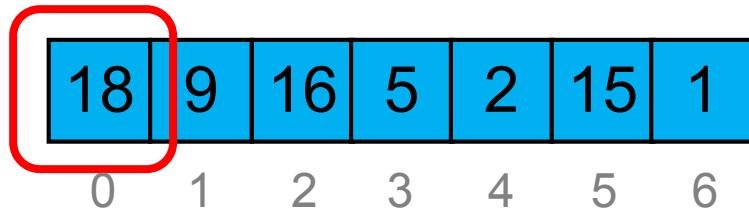
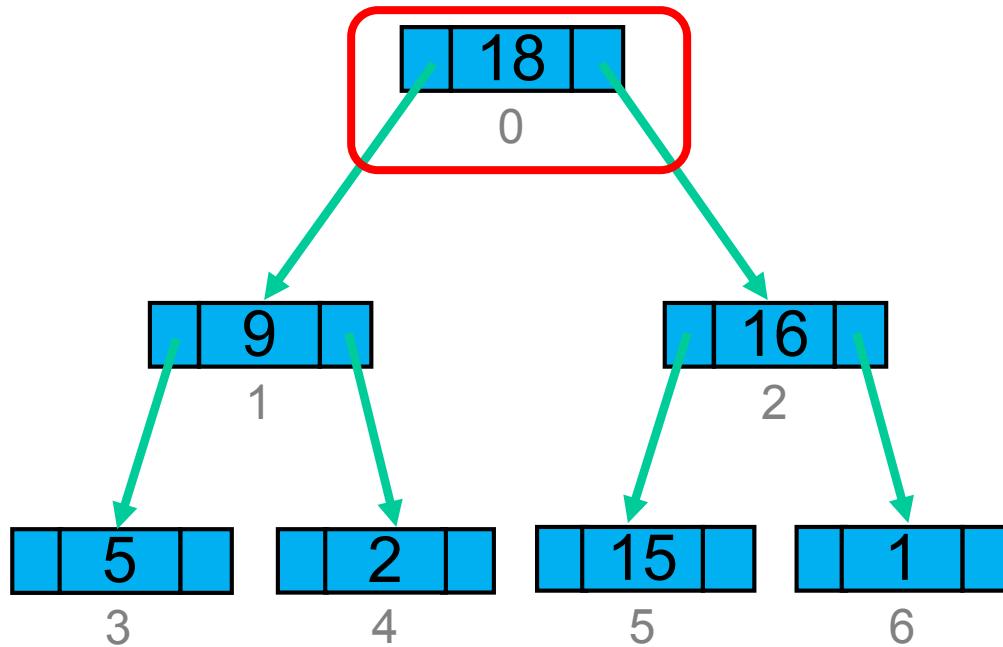
# Array Representation of Heaps



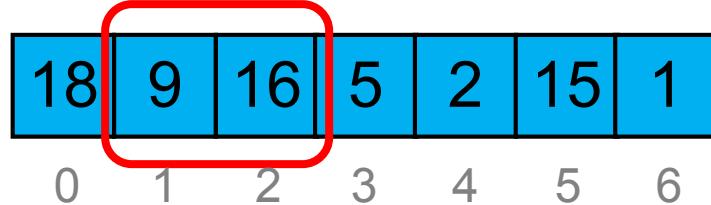
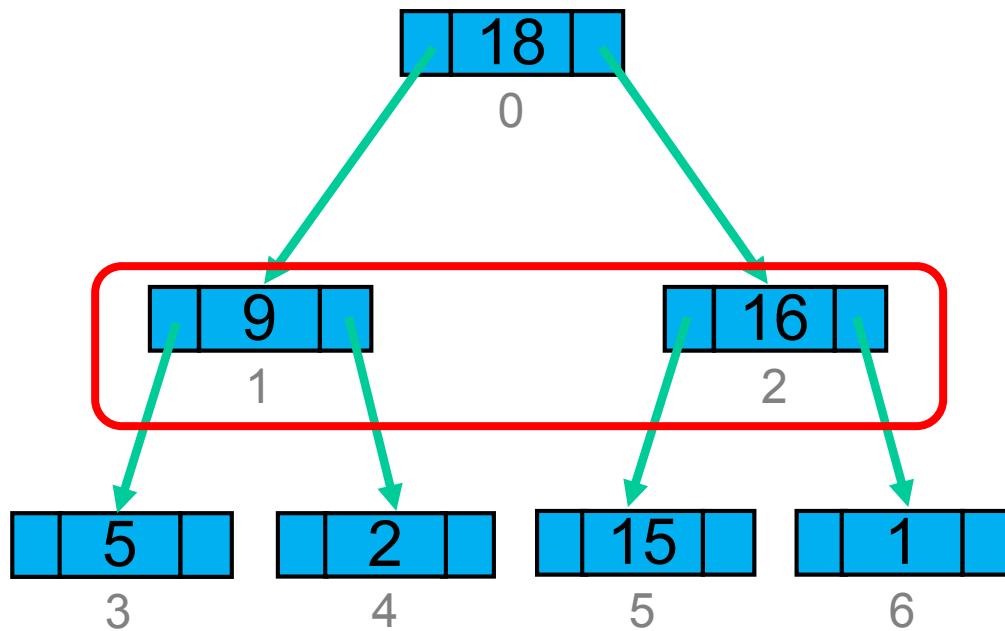
# Array Representation of Heaps



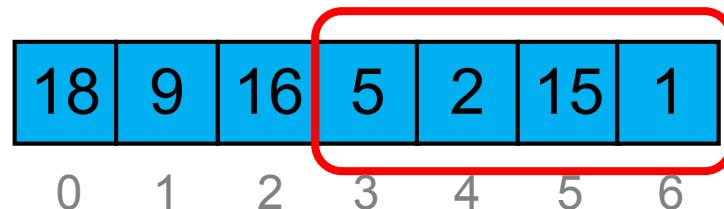
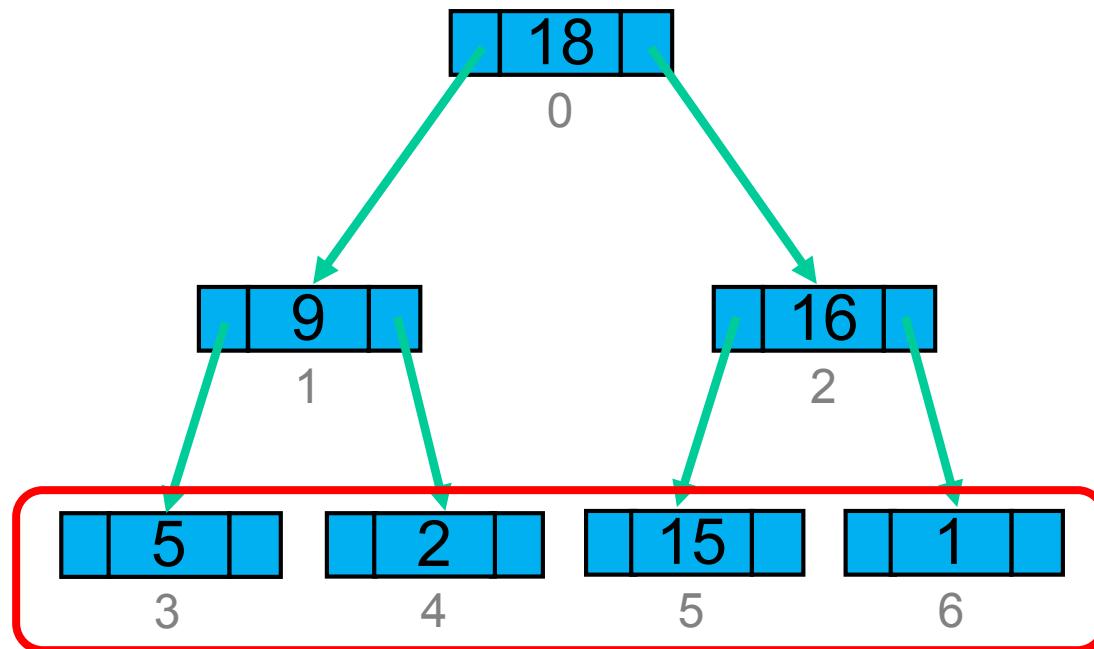
# Array Representation of Heaps



# Array Representation of Heaps



# Array Representation of Heaps



# Array Representation of Heaps

---

- A heap with max height  $n$  will have an array size of  $2^n - 1$ .
- The root of the tree is in array index 0
- If a node is at index  $k$ ...
  - Its left child is at index  $2k+1$
  - Its right child is at index  $2k+2$
  - Its parent is at index  $(k-1)/2$

# Array Representation of Heaps

---

```
public class Heap {  
  
    int[] values;  
    int size = 0;  
  
    public Heap(int maxHeight) {  
        values = new int[Math.pow(2, maxHeight)-1];  
    }  
  
    public void swapValues(int index1, int index2) {  
        int temp = values[index1];  
        values[index1] = values[index2];  
        values[index2] = temp;  
    }  
}
```

# Array Representation of Heaps

---

```
public class Heap {  
  
    int[] values;  
    int size = 0;  
  
    public Heap(int maxHeight) {  
        values = new int[Math.pow(2, maxHeight)-1];  
    }  
  
    public void swapValues(int index1, int index2) {  
        int temp = values[index1];  
        values[index1] = values[index2];  
        values[index2] = temp;  
    }  
}
```

# Array Representation of Heaps

---

```
public class Heap {  
  
    int[] values;  
    int size = 0;  
  
    public Heap(int maxHeight) {  
        values = new int[Math.pow(2, maxHeight)-1];  
    }  
  
    public void swapValues(int index1, int index2) {  
        int temp = values[index1];  
        values[index1] = values[index2];  
        values[index2] = temp;  
    }  
}
```

# Array Representation of Heaps

---

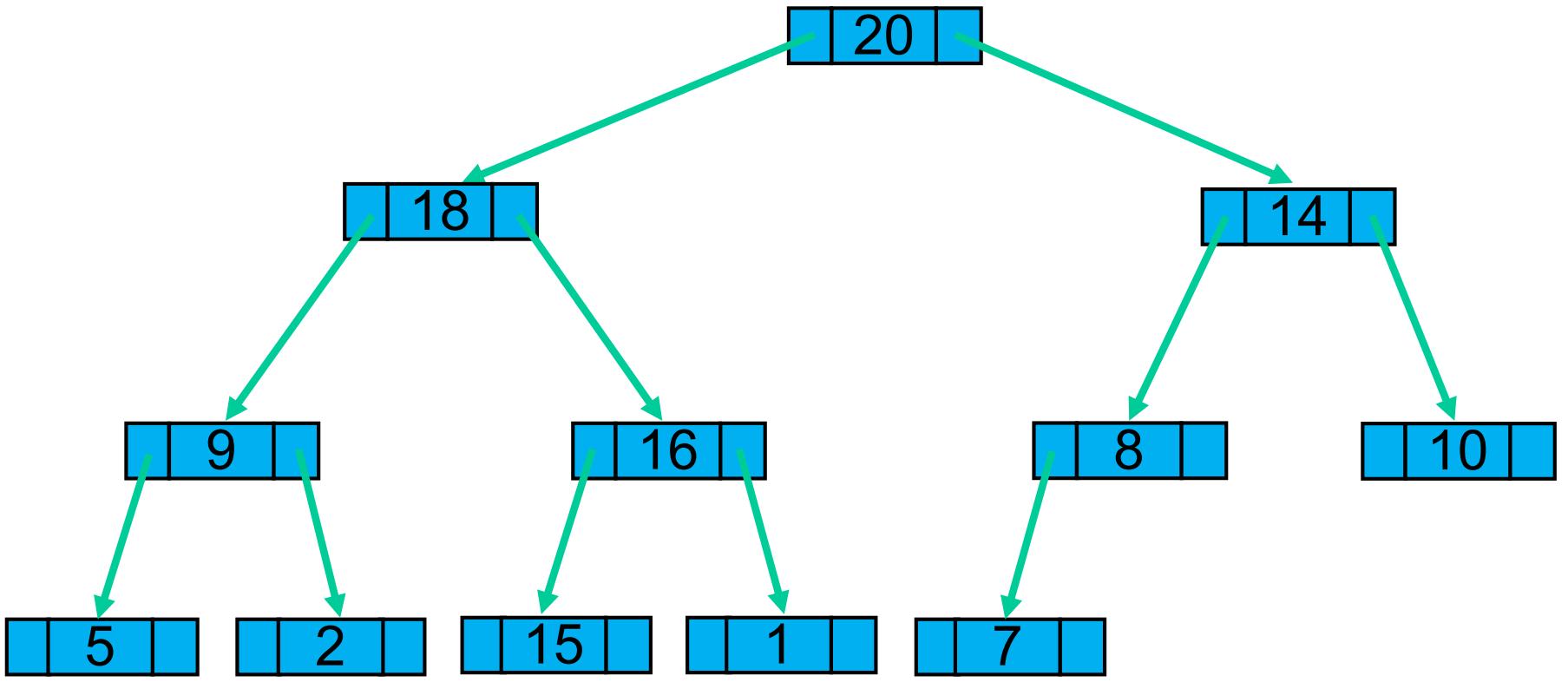
```
public class Heap {  
  
    int[] values;  
    int size = 0;  
  
    public Heap(int maxHeight) {  
        values = new int[Math.pow(2, maxHeight)-1];  
    }  
  
    public void swapValues(int index1, int index2) {  
        int temp = values[index1];  
        values[index1] = values[index2];  
        values[index2] = temp;  
    }  
}
```

# Array Representation of Heaps

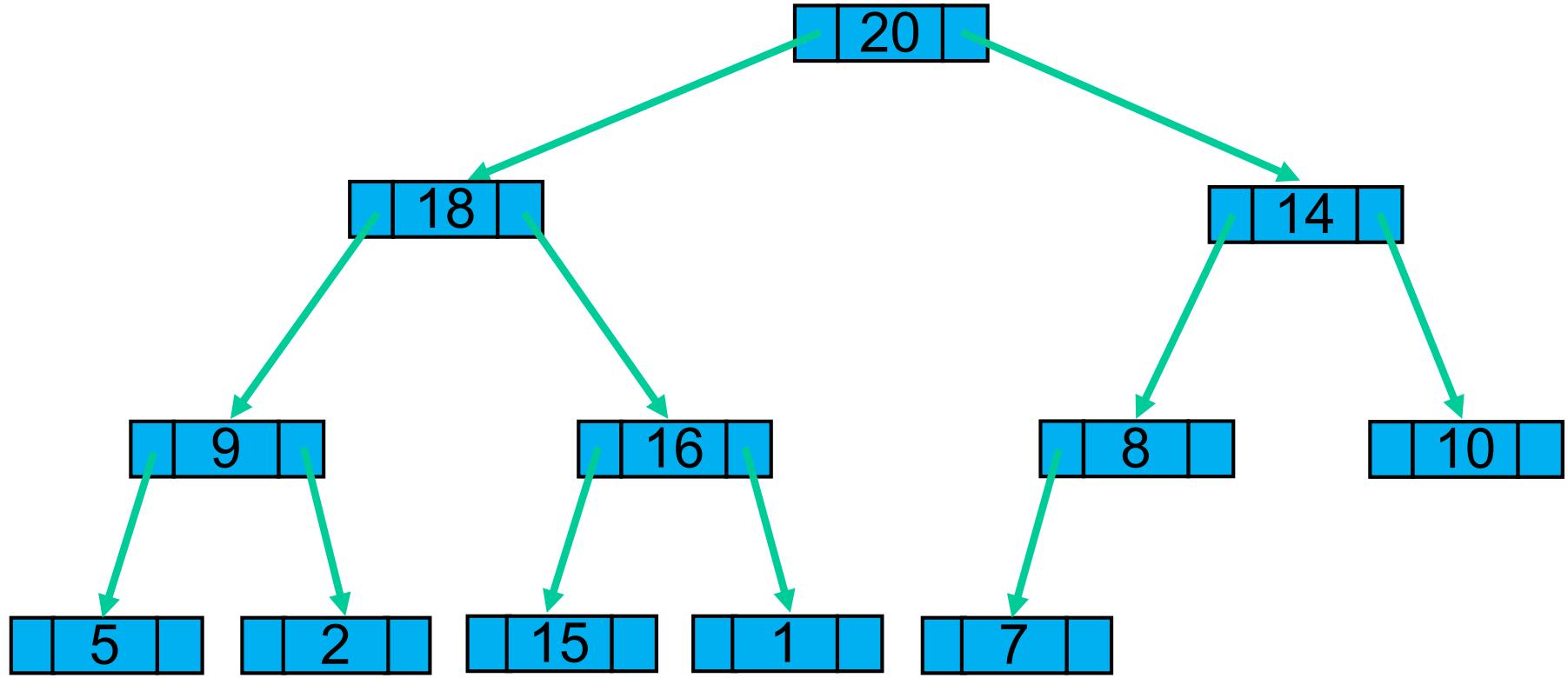
---

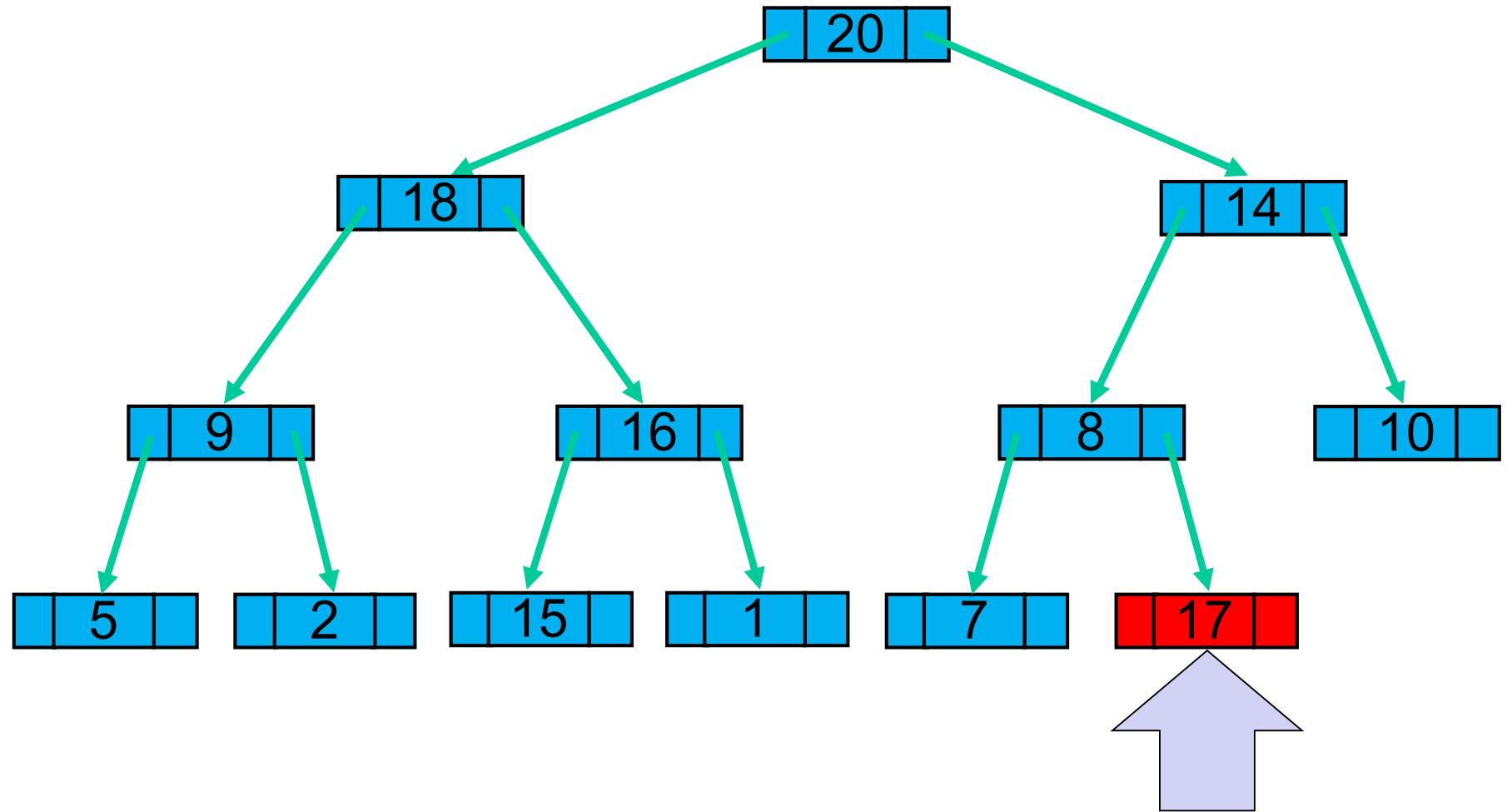
```
public class Heap {  
  
    int[] values;  
    int size = 0;  
  
    public Heap(int maxHeight) {  
        values = new int[Math.pow(2, maxHeight)-1];  
    }  
  
    public void swapValues(int index1, int index2) {  
        int temp = values[index1];  
        values[index1] = values[index2];  
        values[index2] = temp;  
    }  
}
```

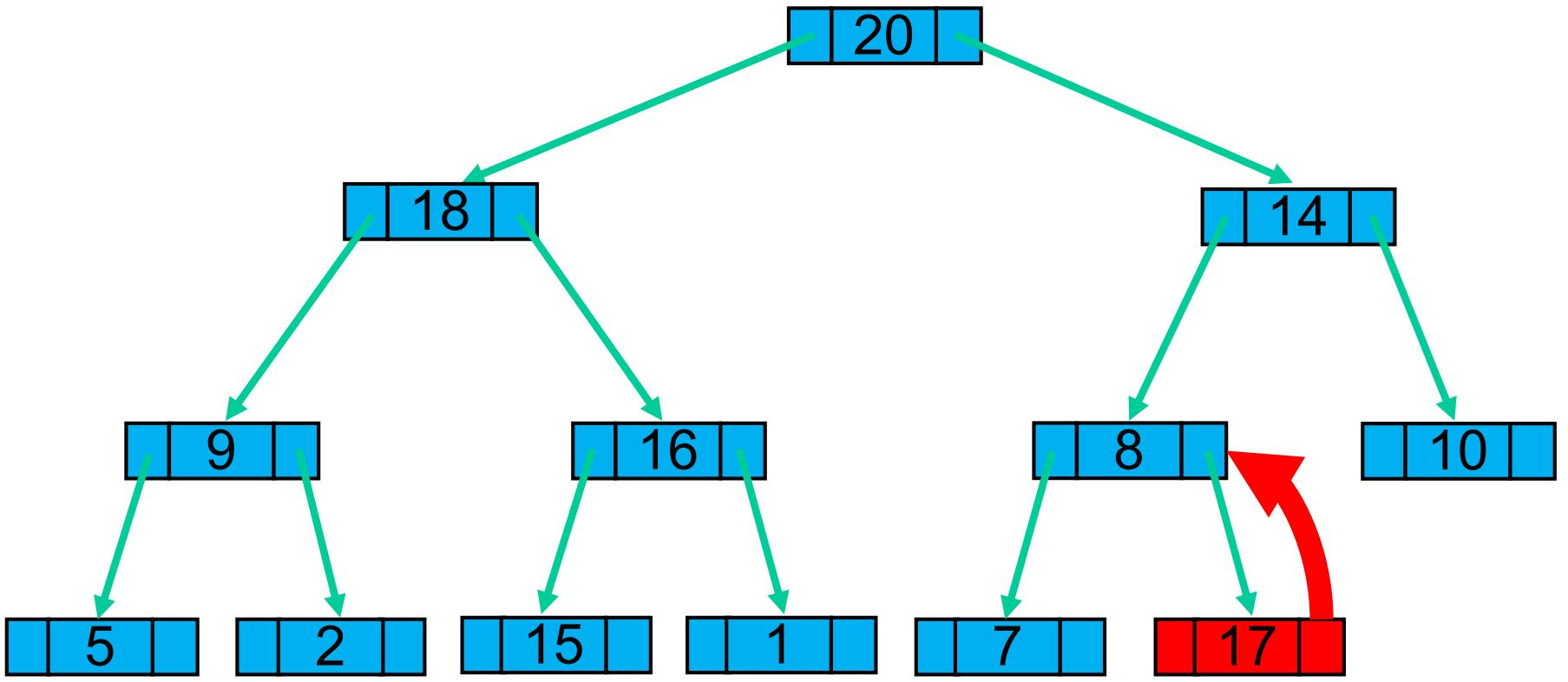
# How do we add a value to a heap?

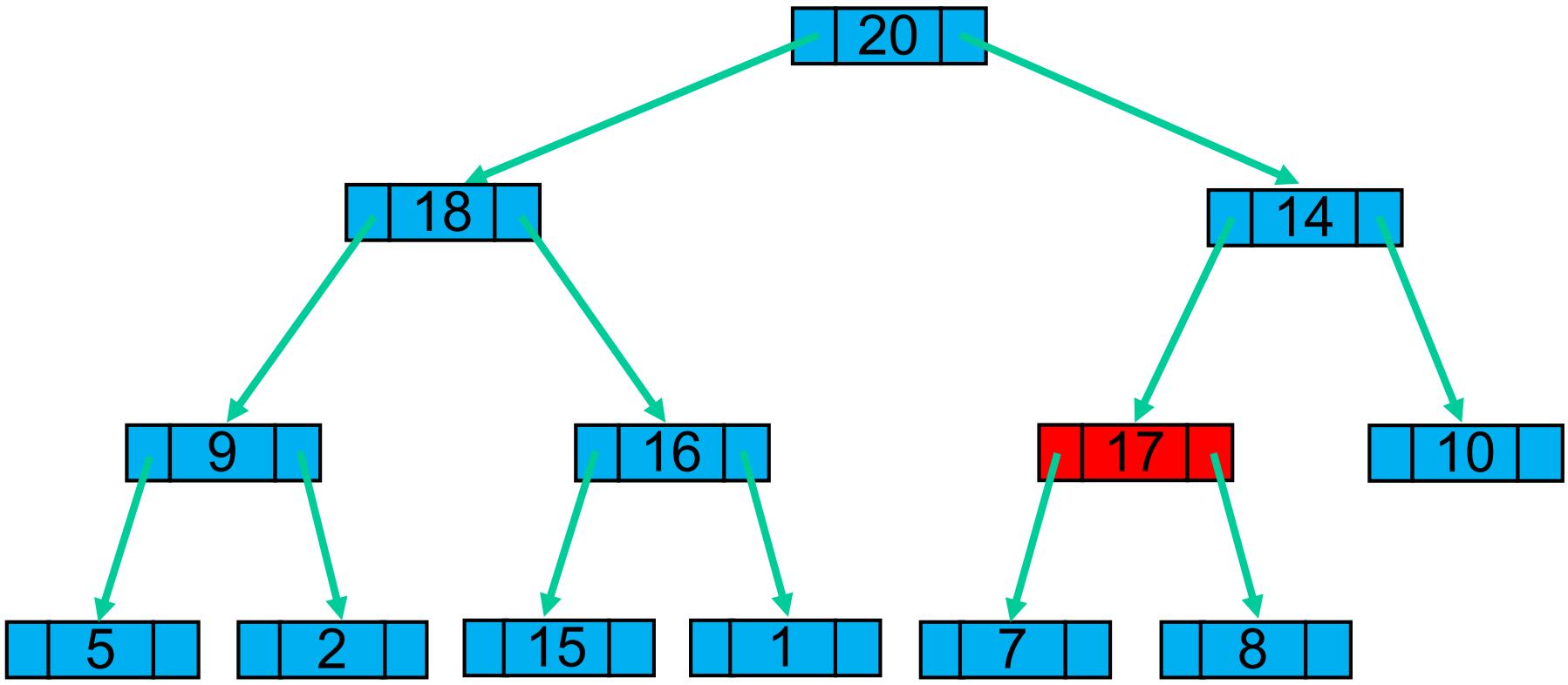


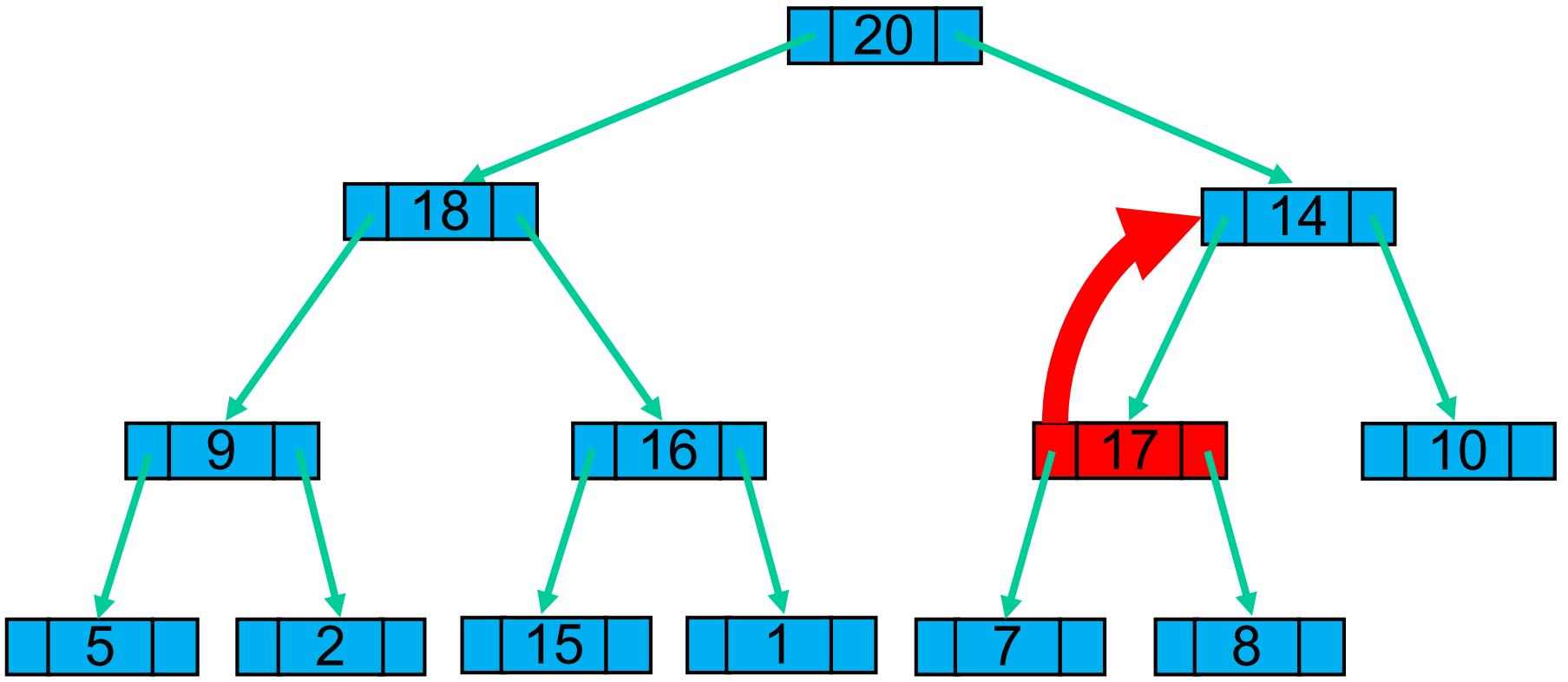
17

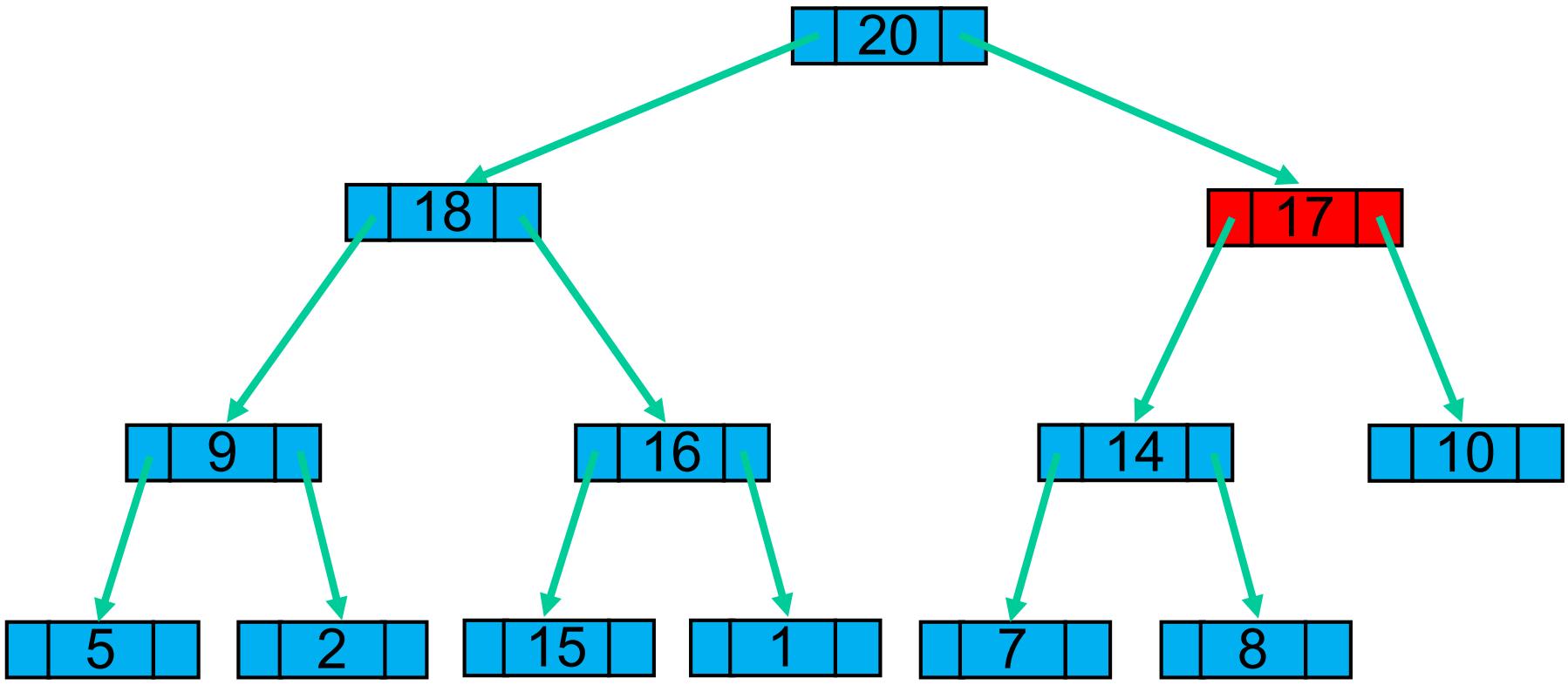


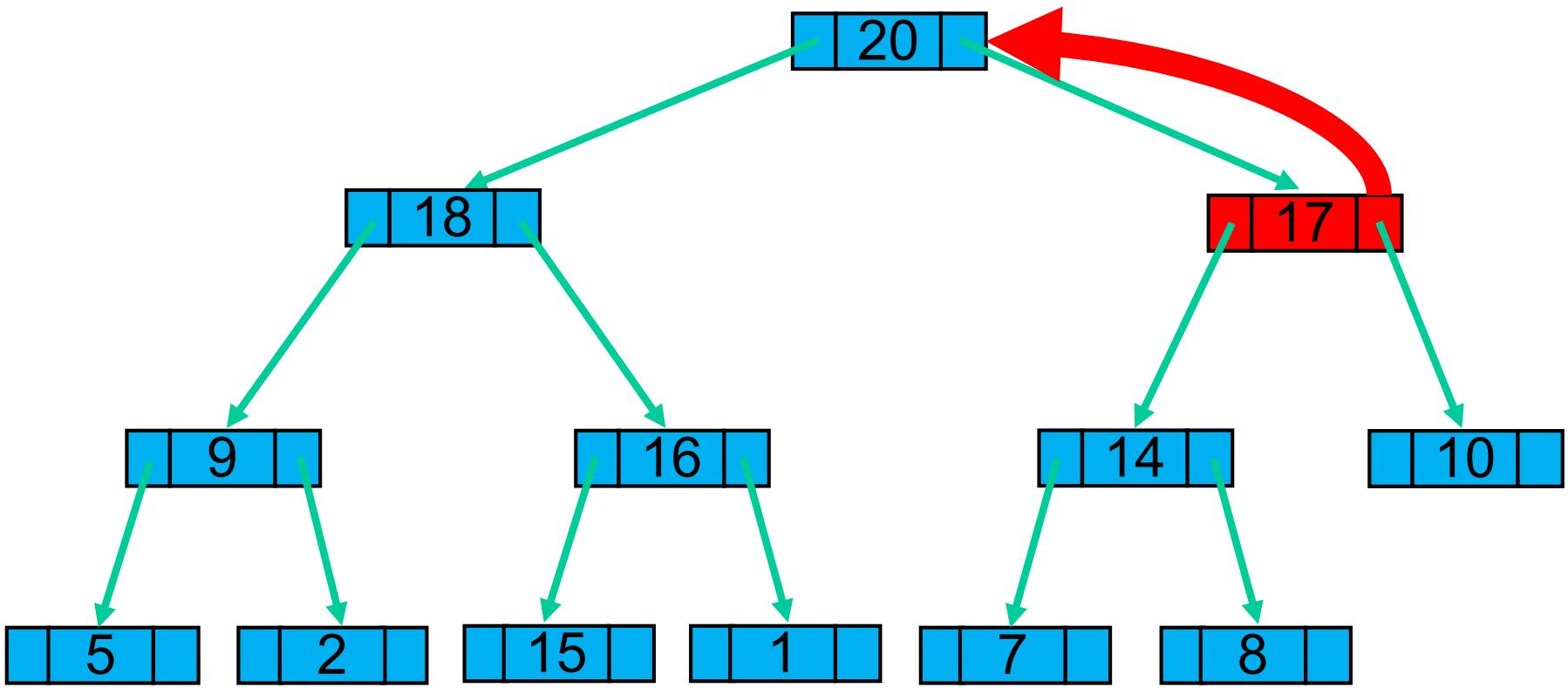


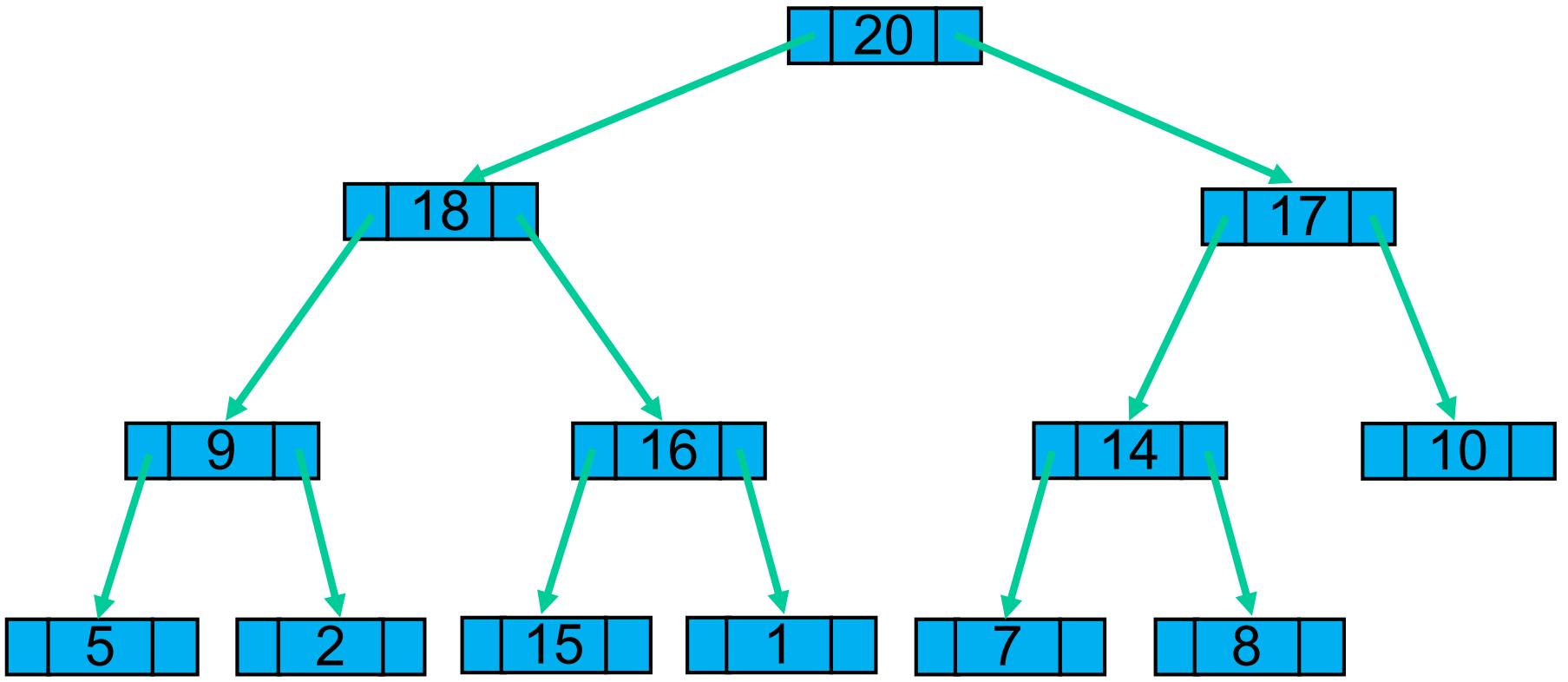


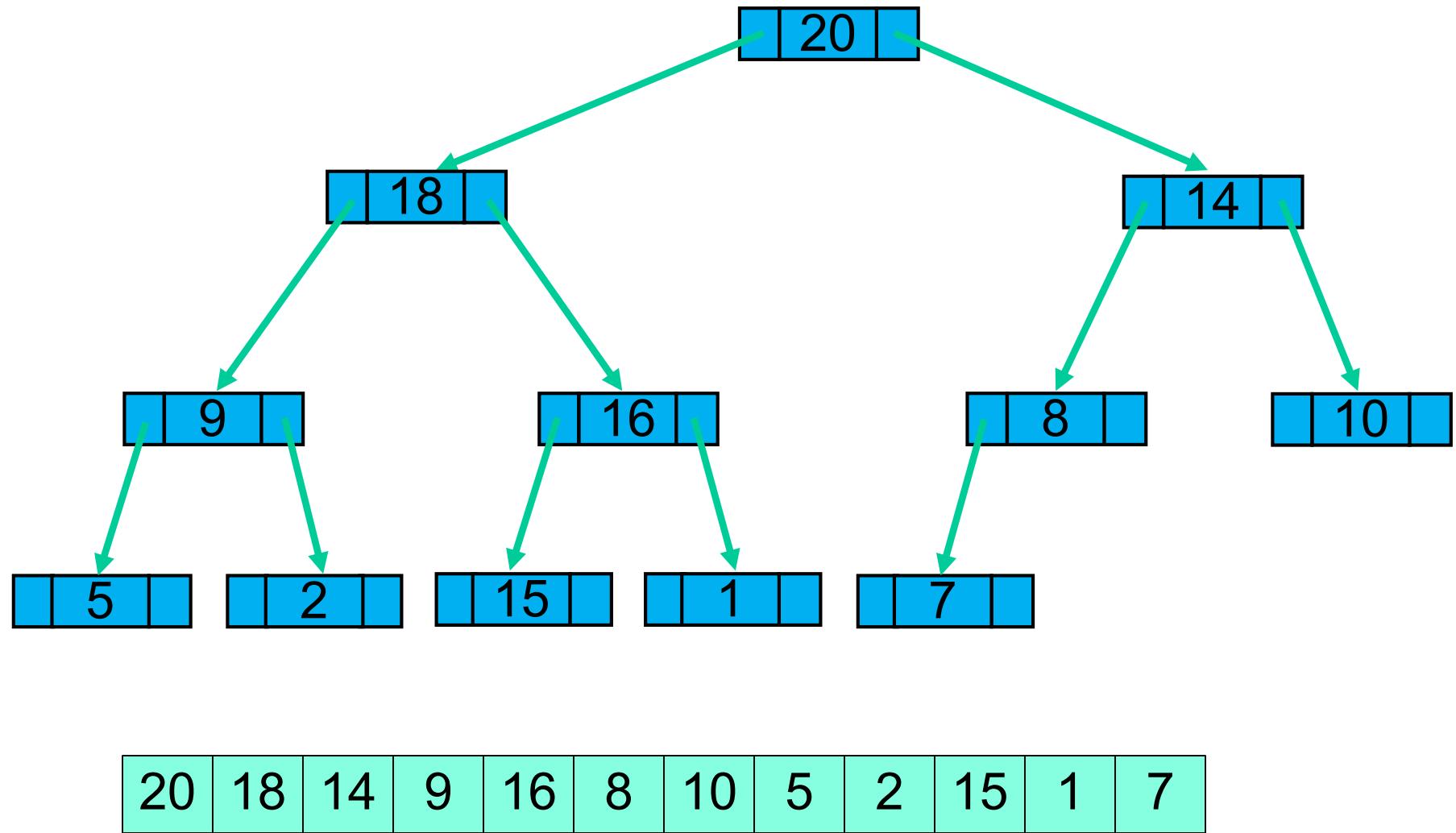


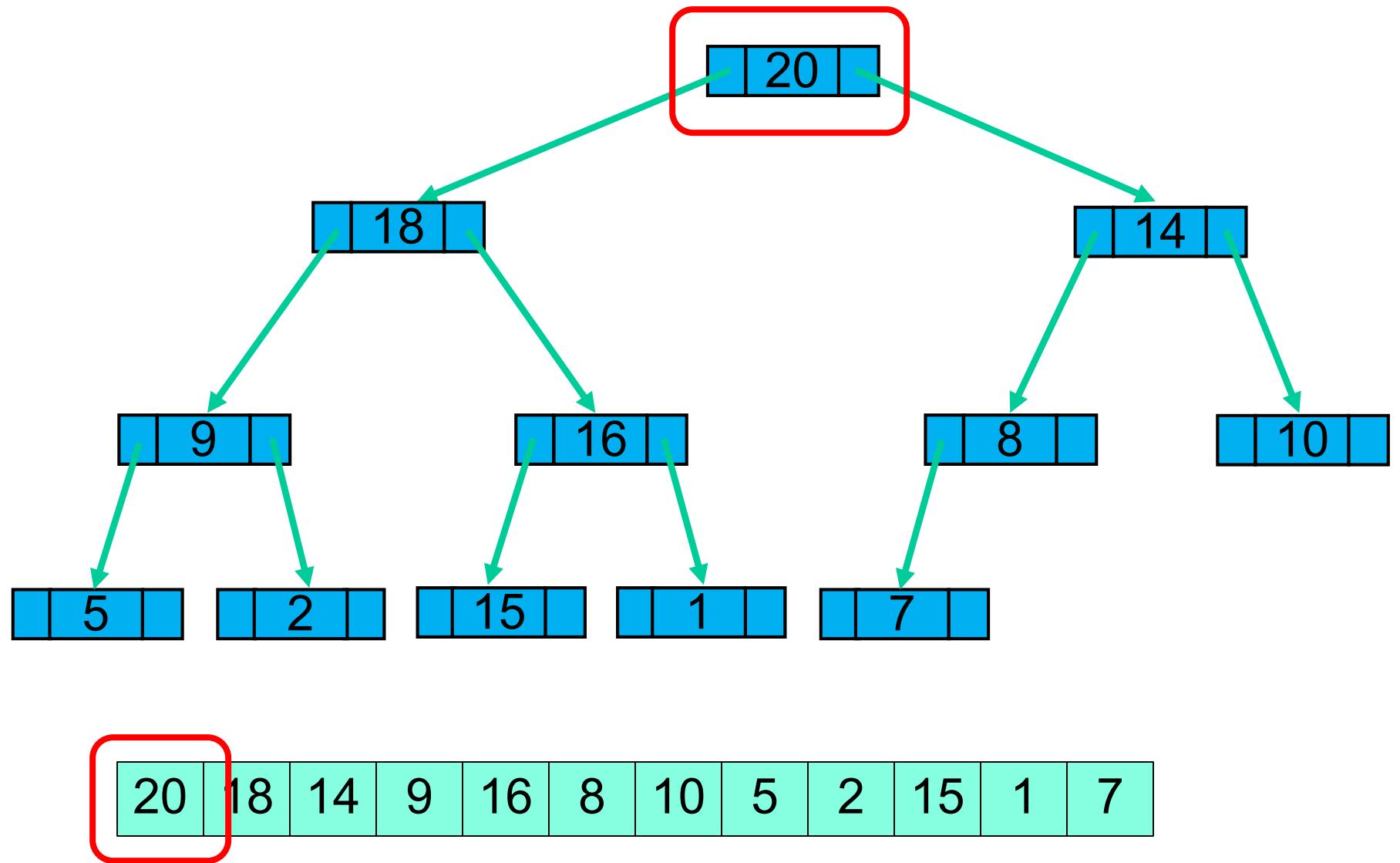


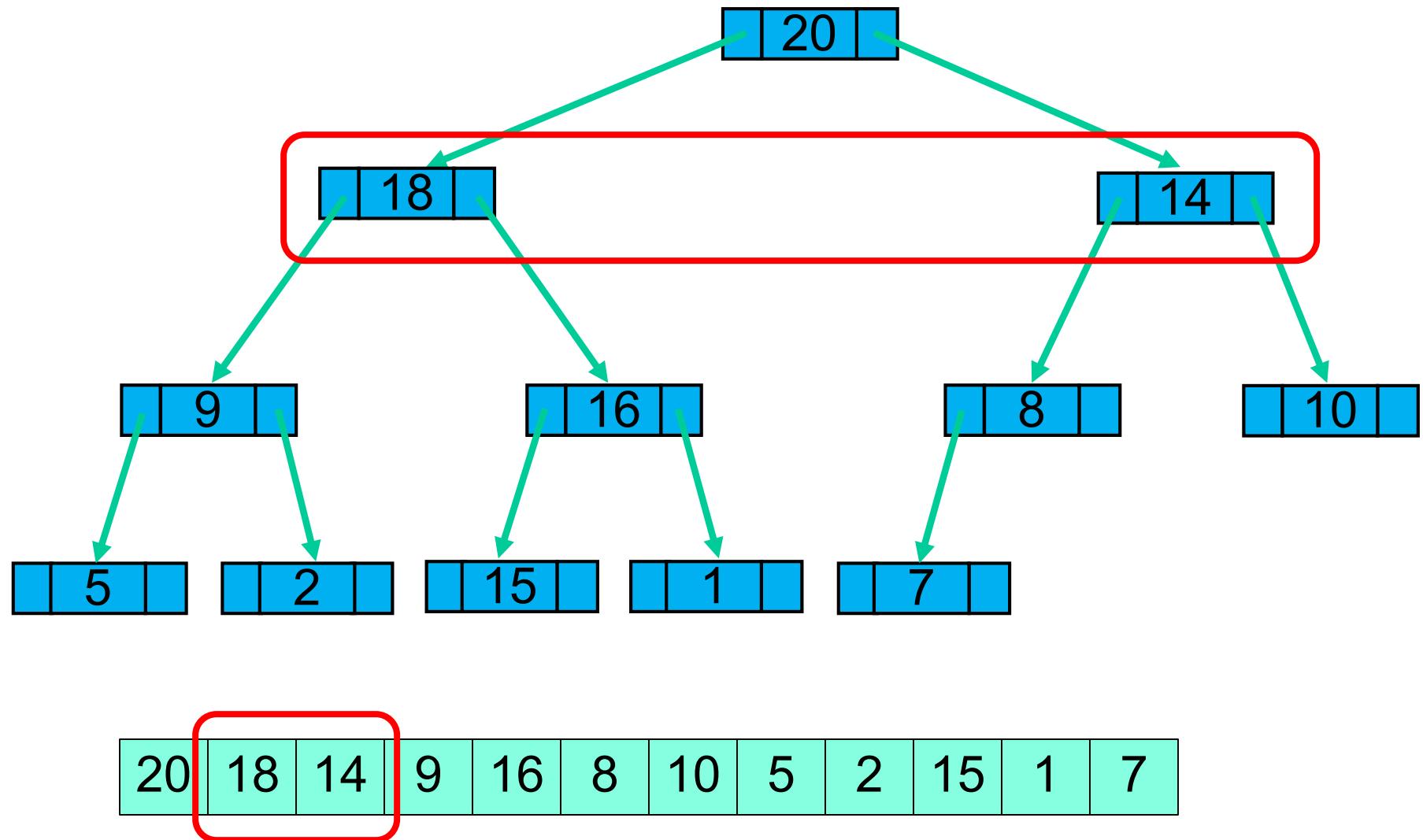


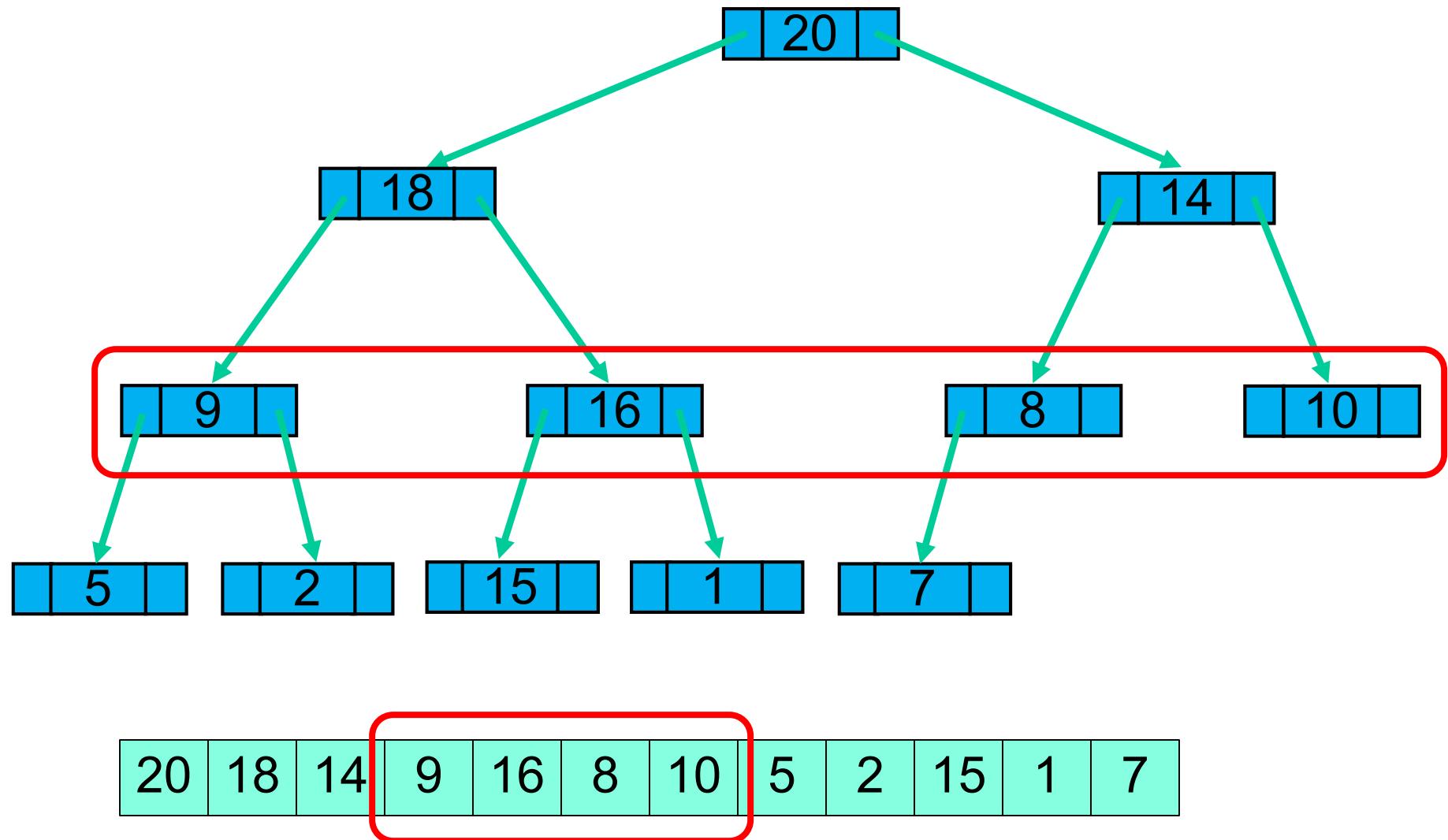


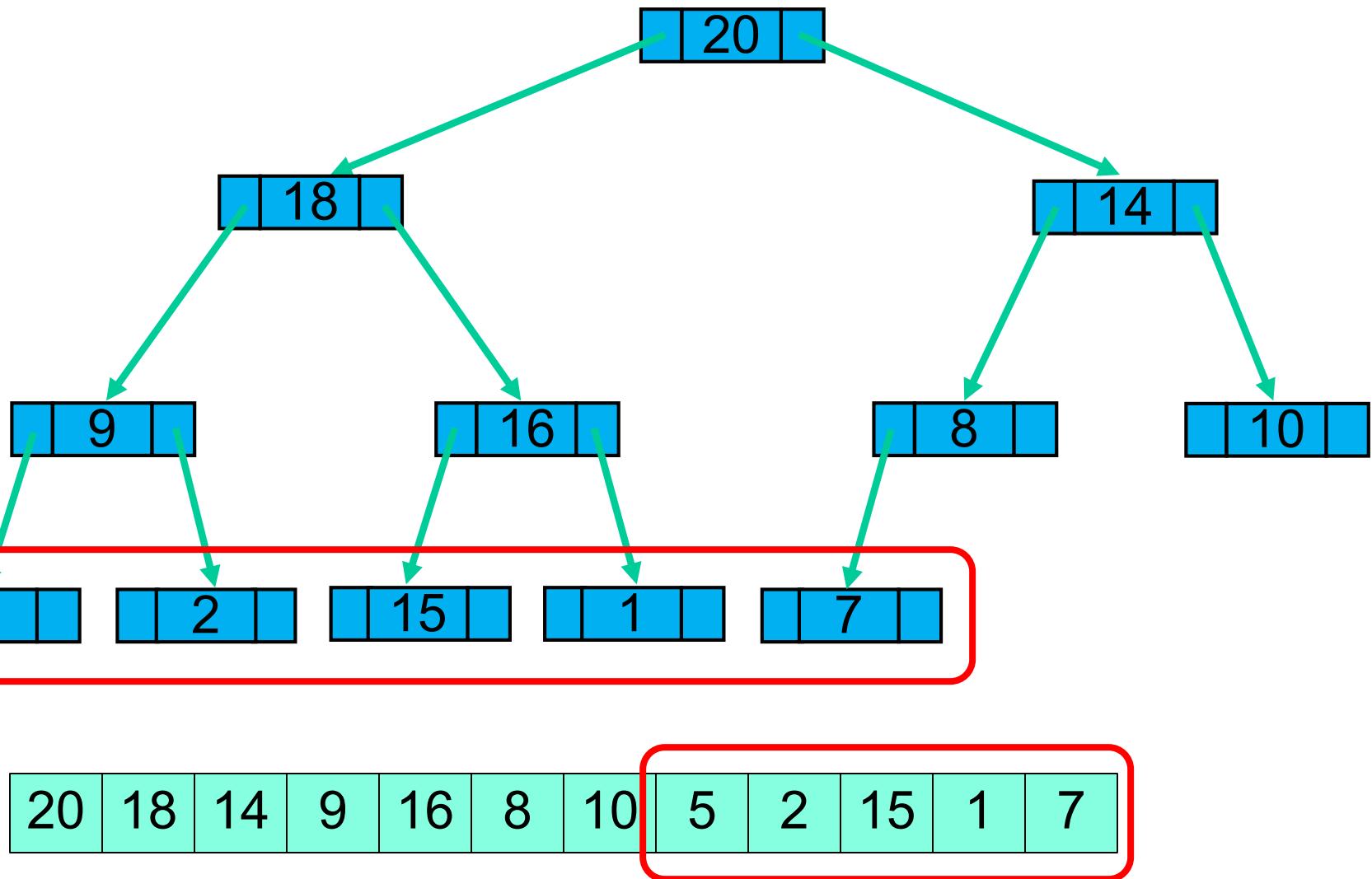


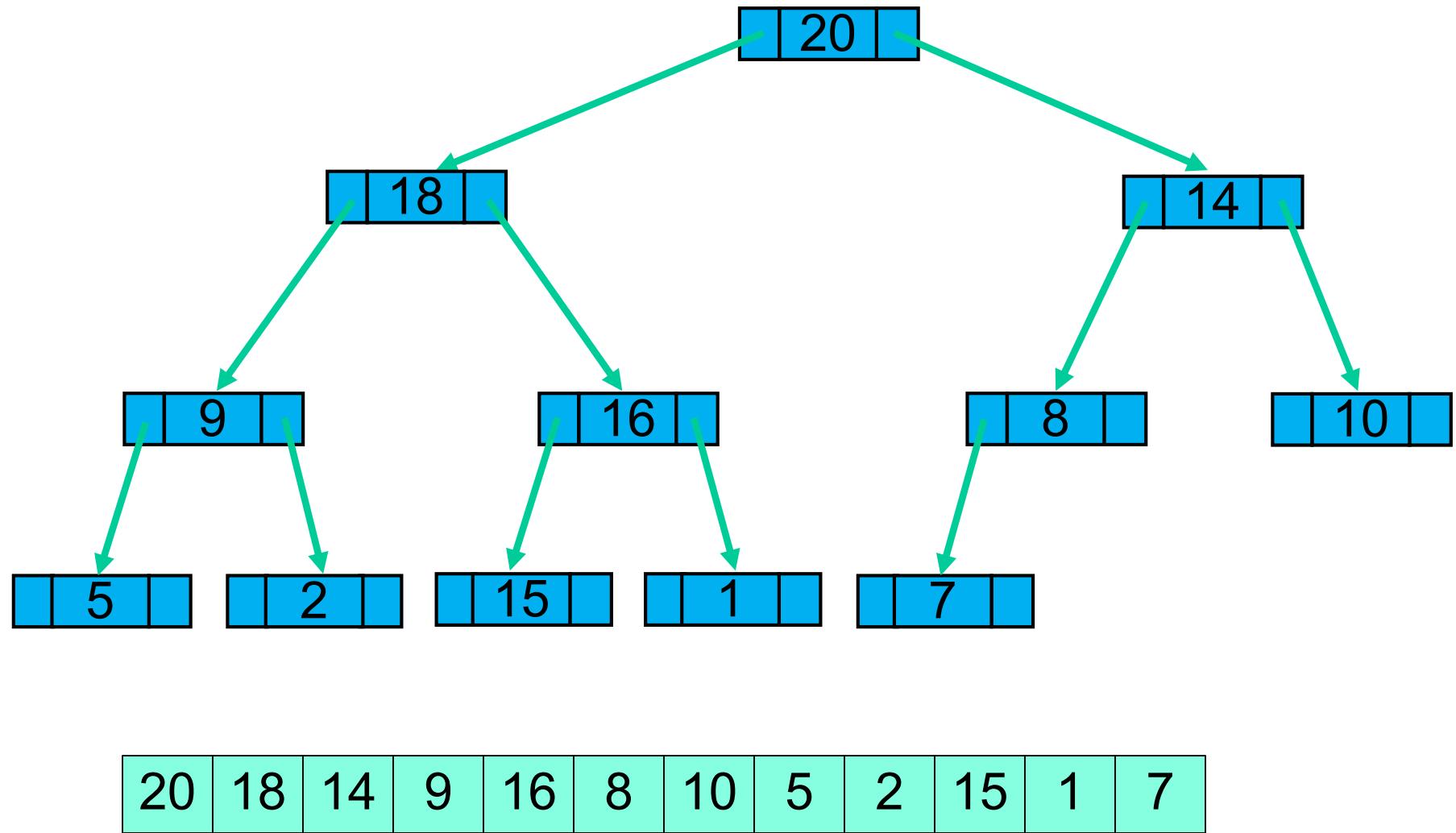




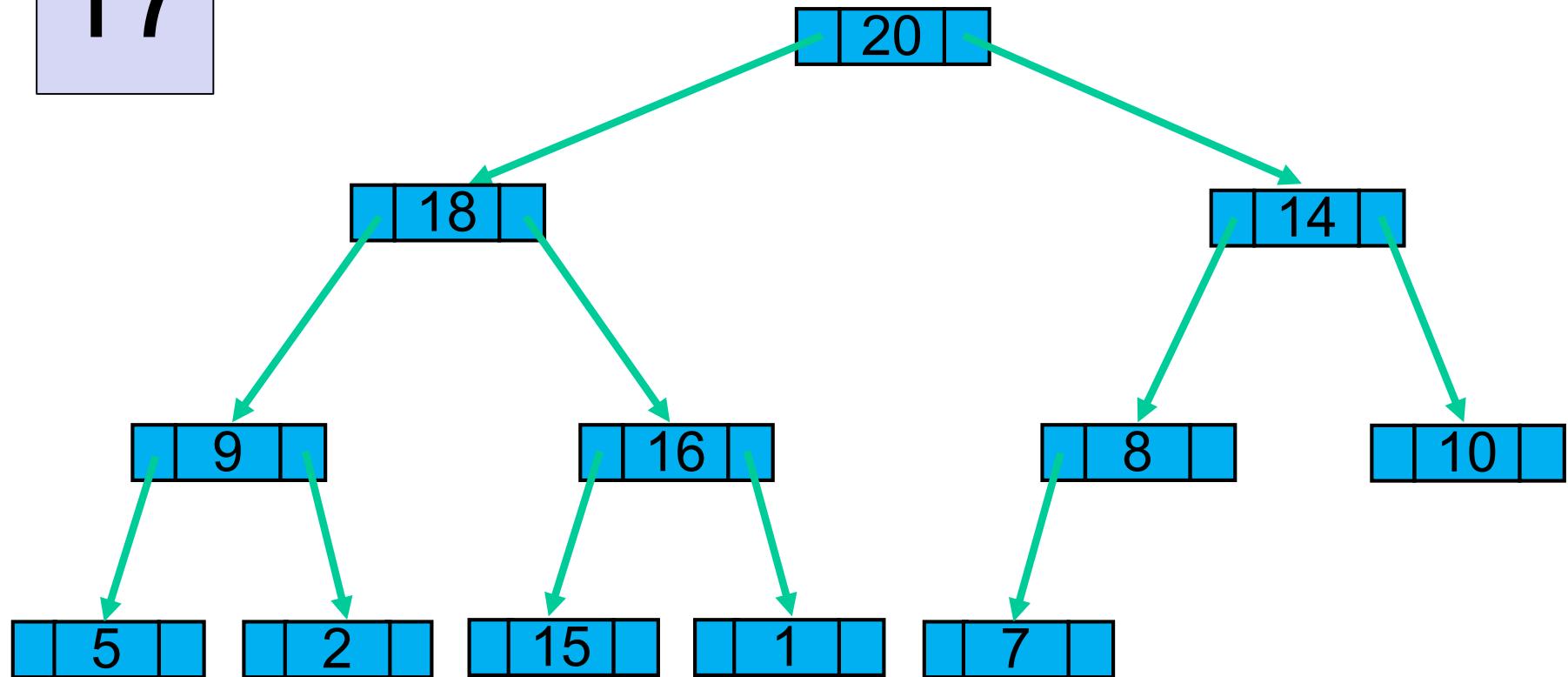




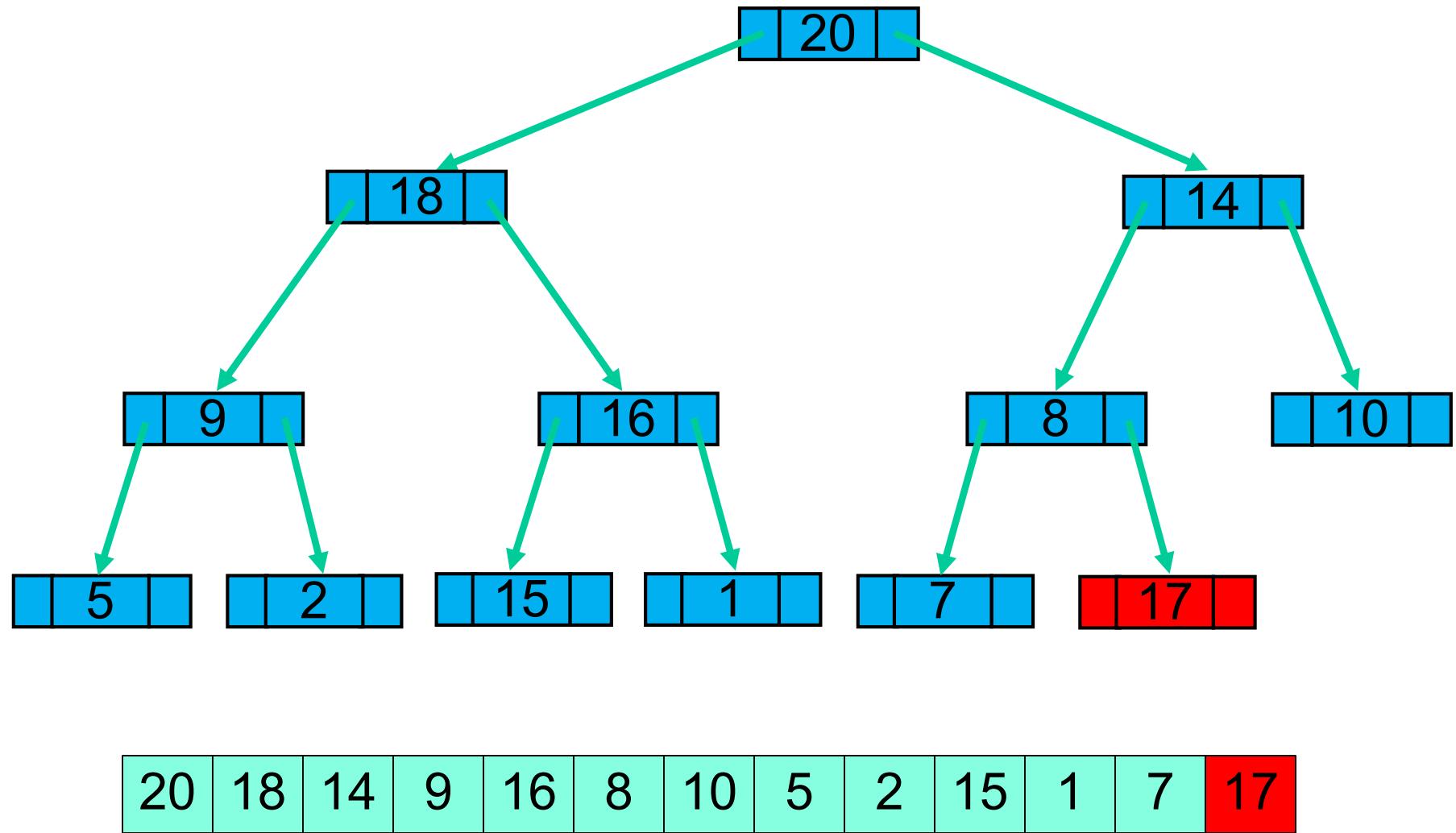


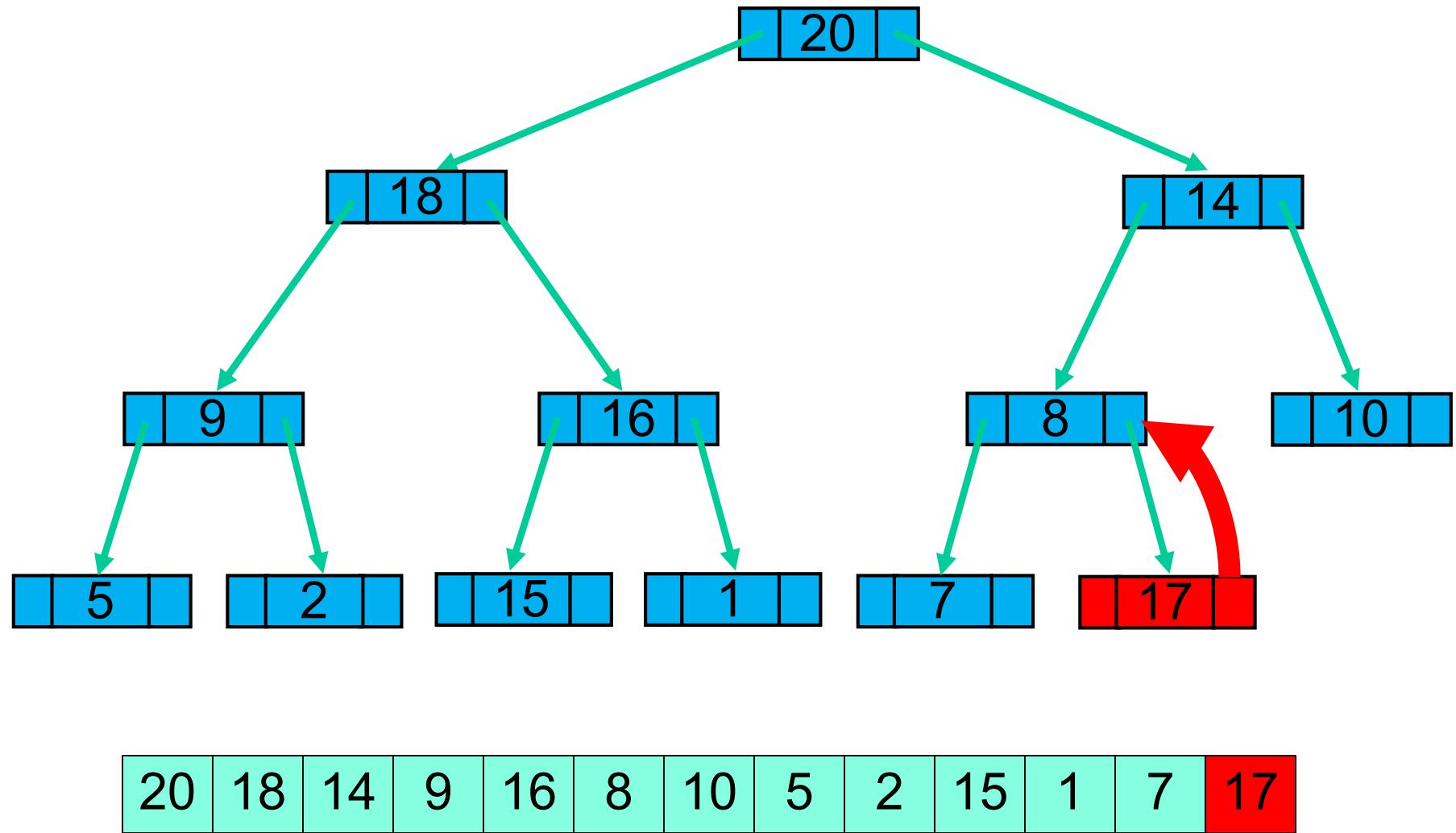


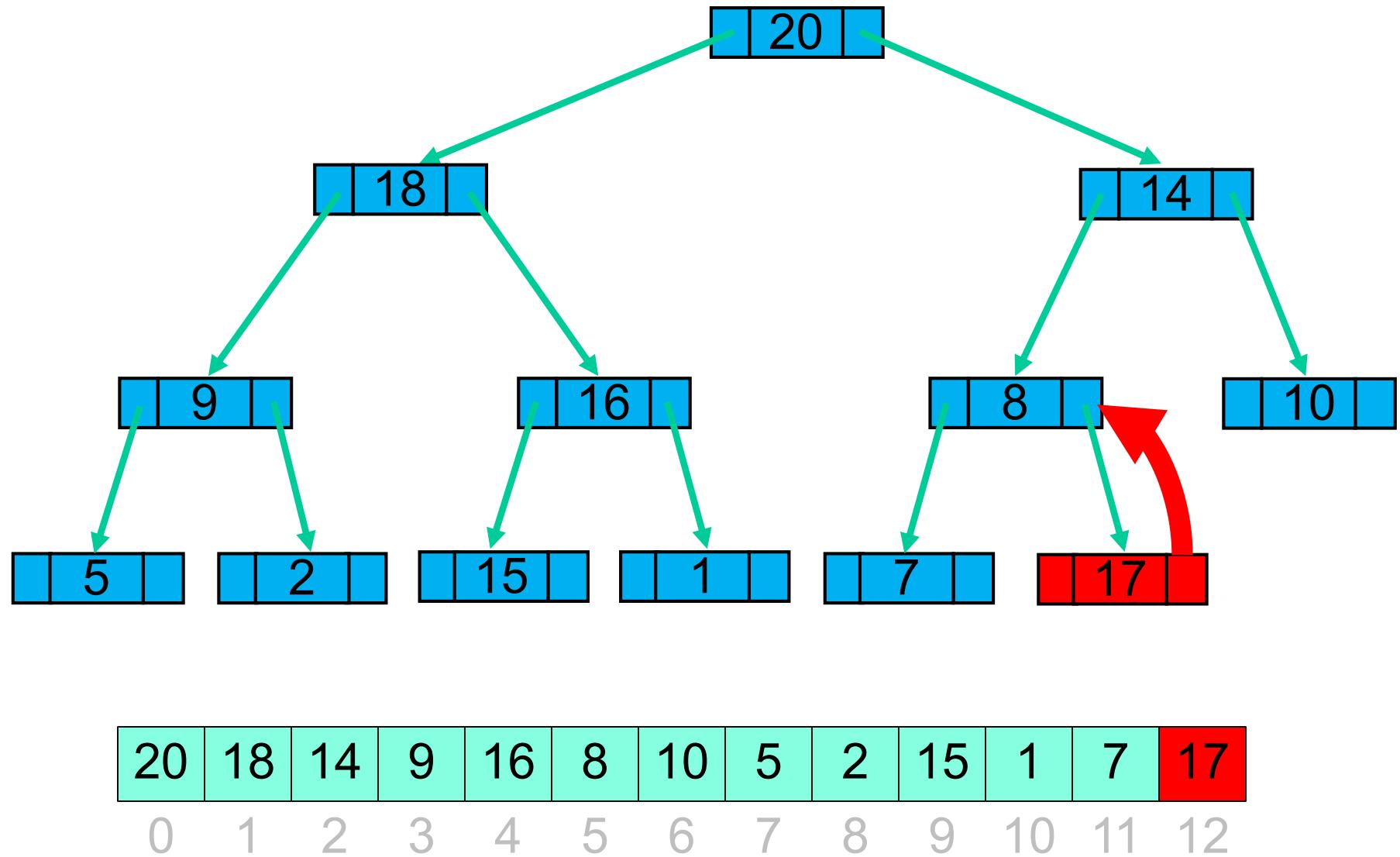
17

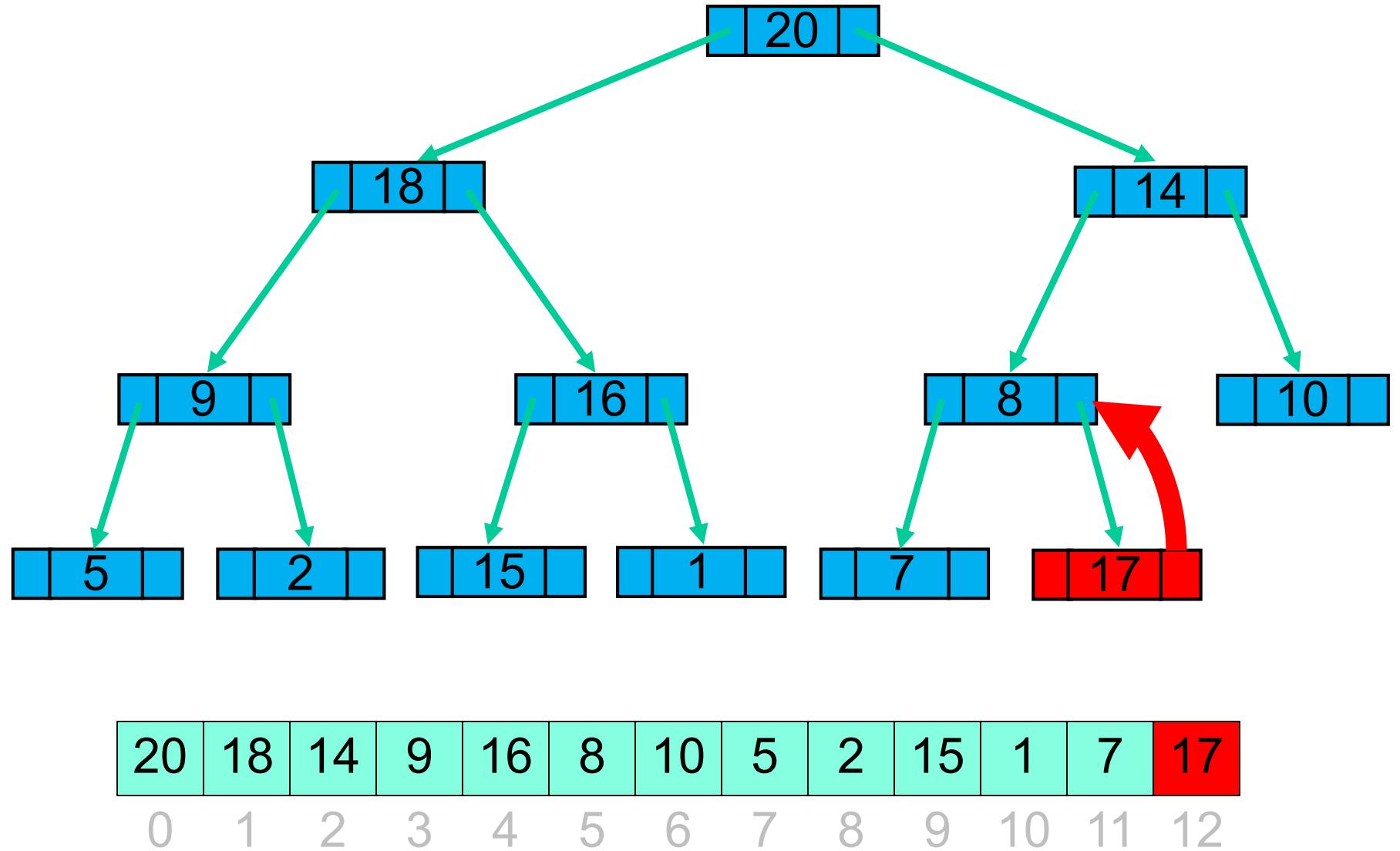


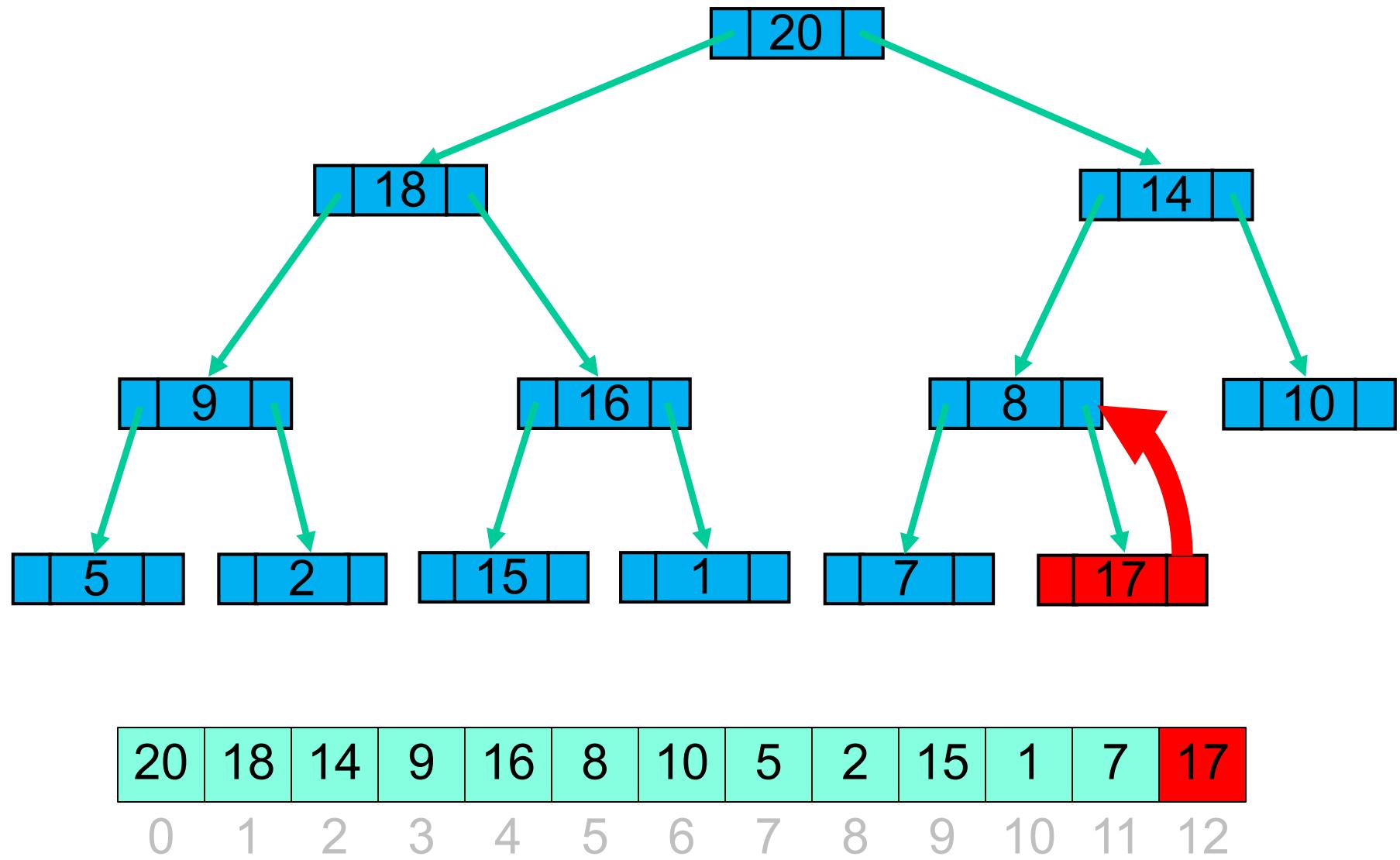
20	18	14	9	16	8	10	5	2	15	1	7
----	----	----	---	----	---	----	---	---	----	---	---

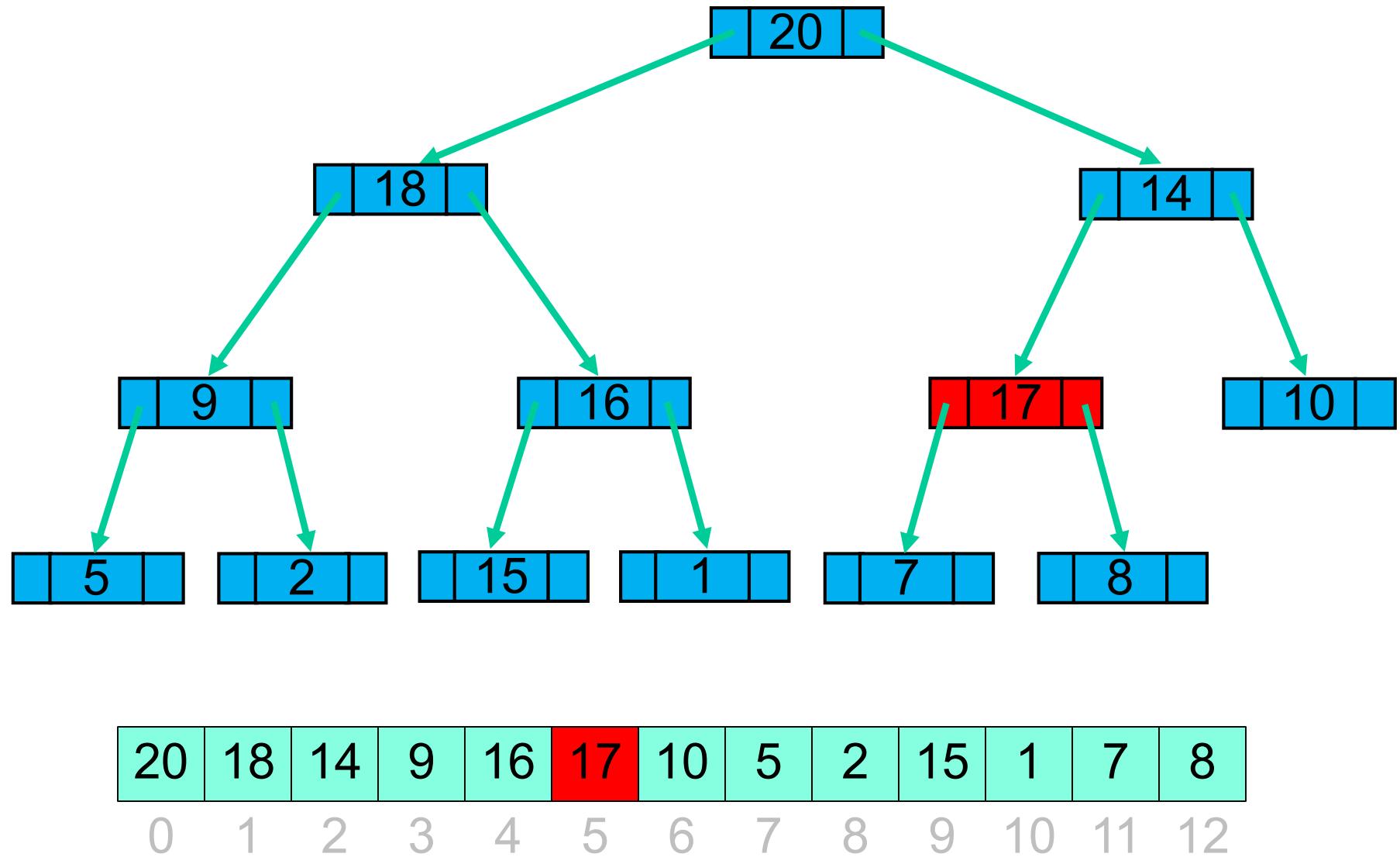


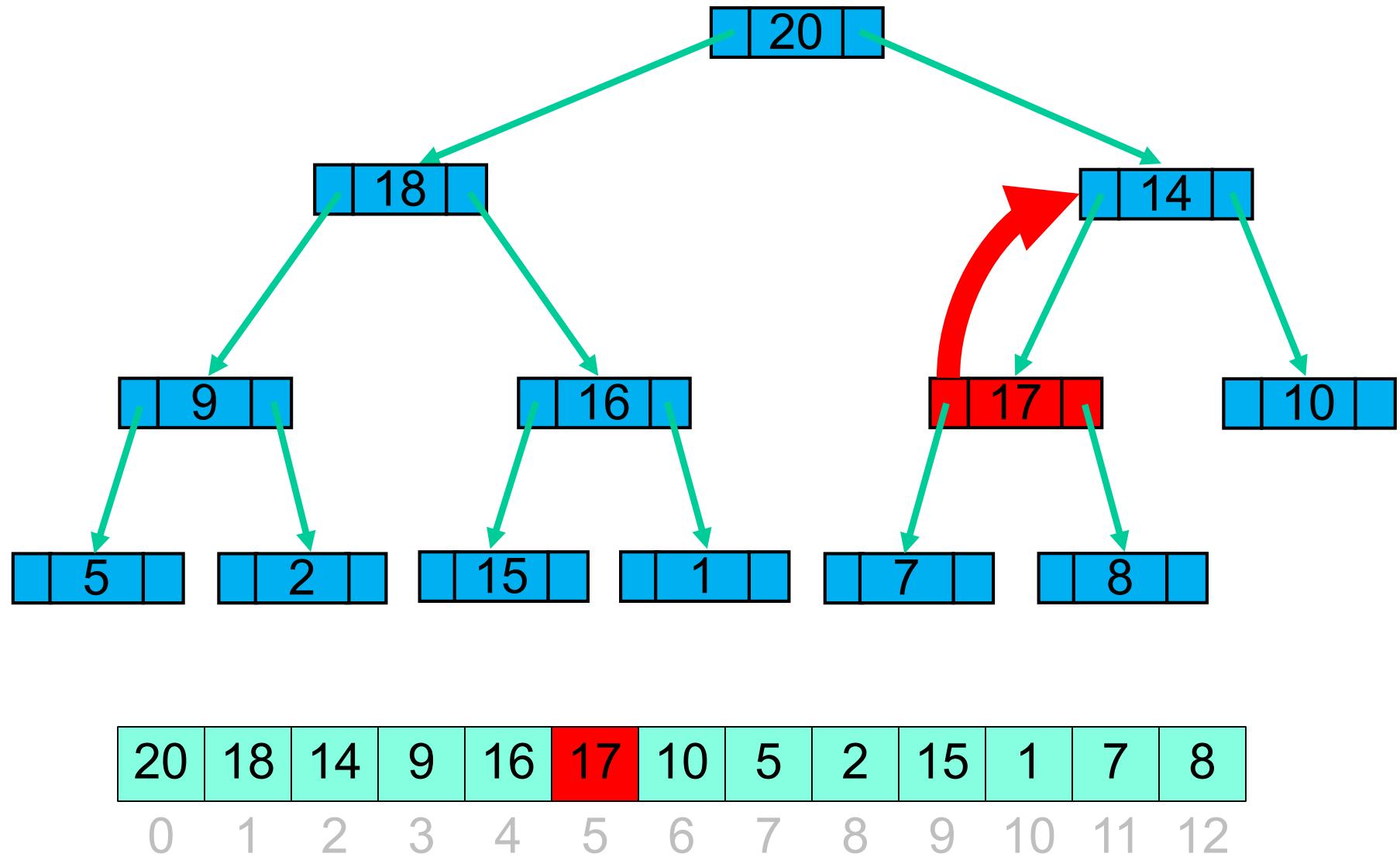


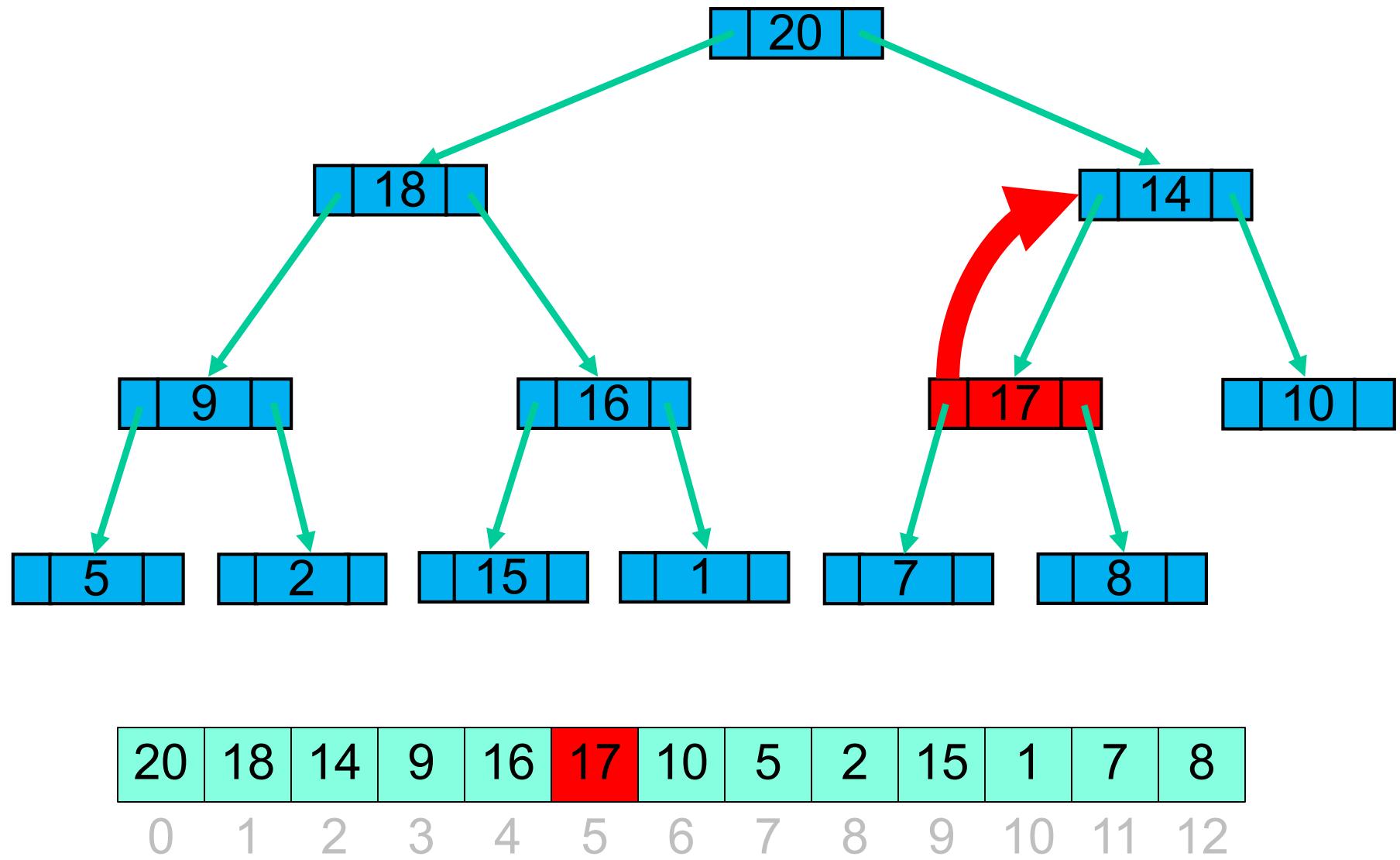


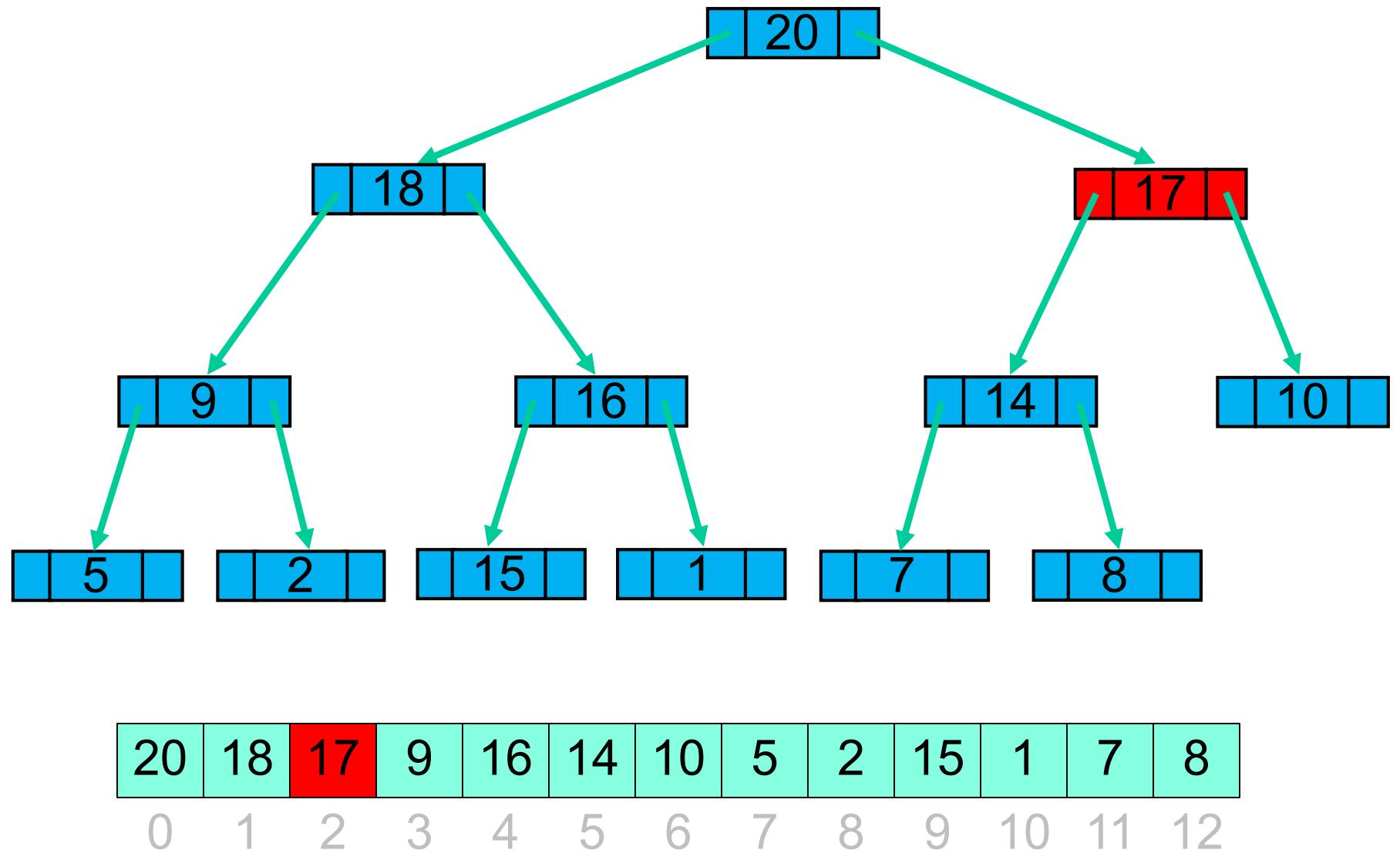


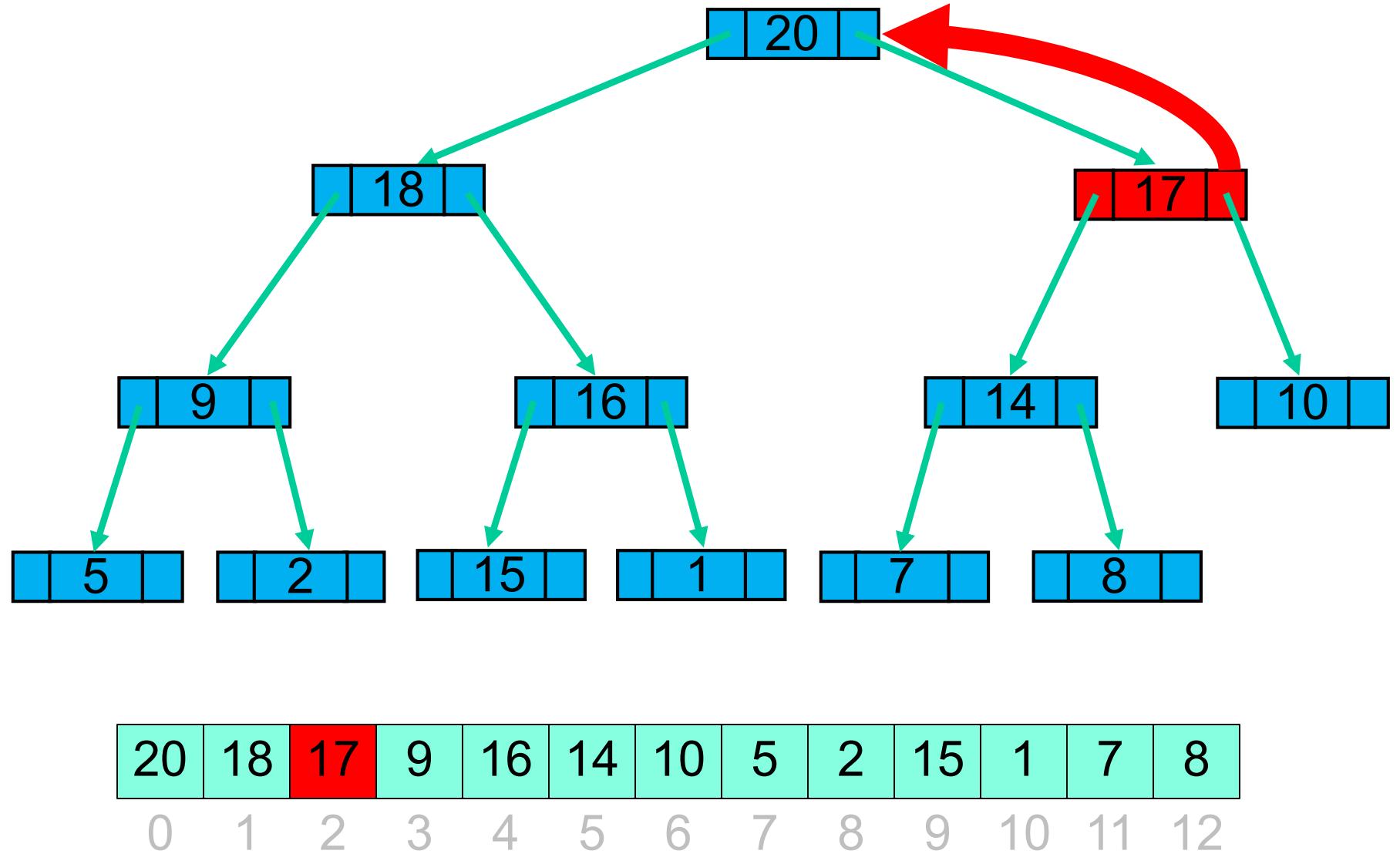


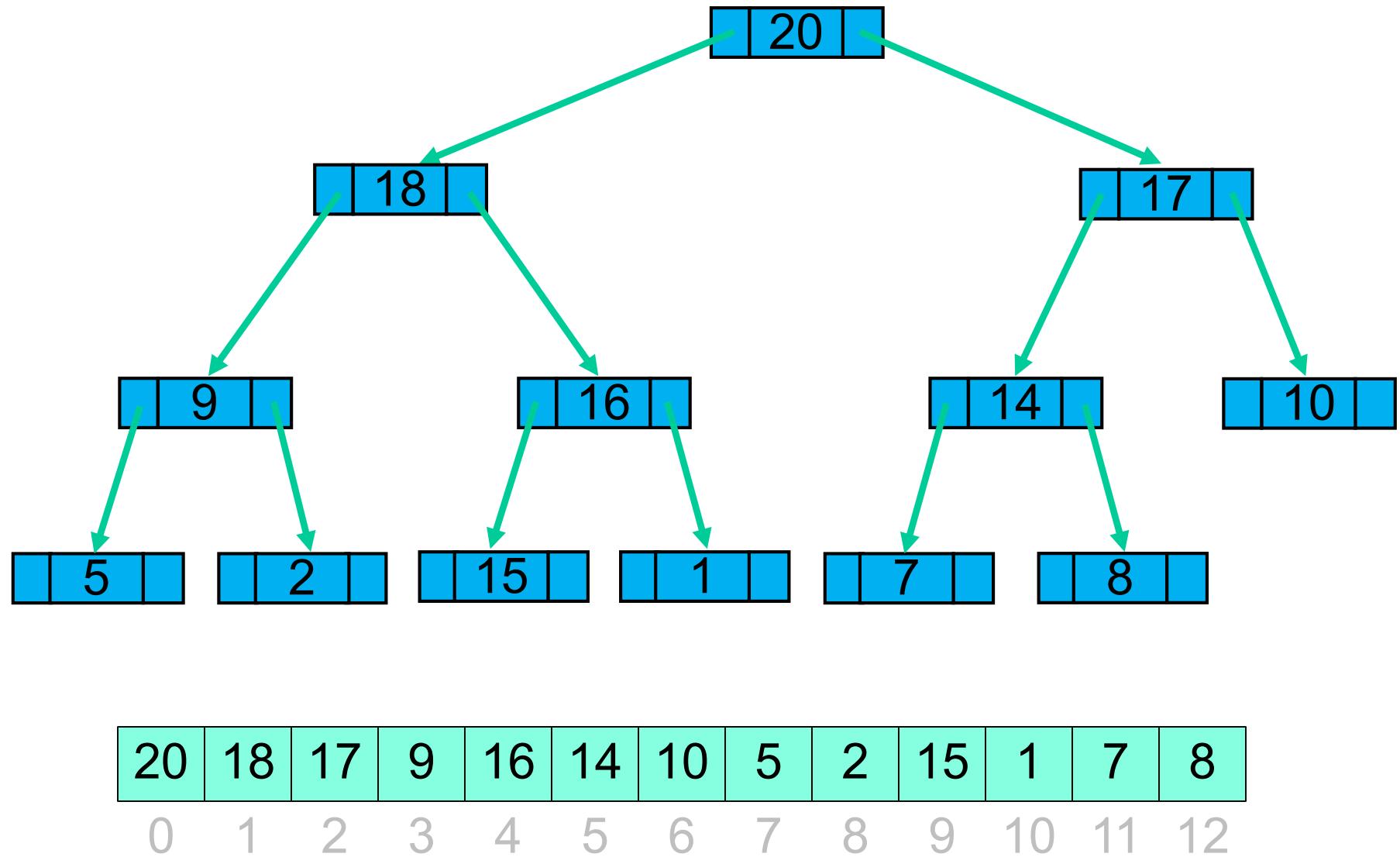












# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Adding to a Heap

---

```
public void add(int value) {  
    if(size == values.length) {  
        throw new HeapException();  
    }  
    values[size] = value;  
    bubbleUp(size);  
    size++;  
}  
  
public void bubbleUp(int index) {  
    if(index == 0) return;  
    int parentIndex = (index - 1) / 2;  
    if(values[parentIndex] < values[index]) {  
        swapValues(index, parentIndex);  
        bubbleUp(parentIndex);  
    }  
}
```

# Recap: Heaps

---

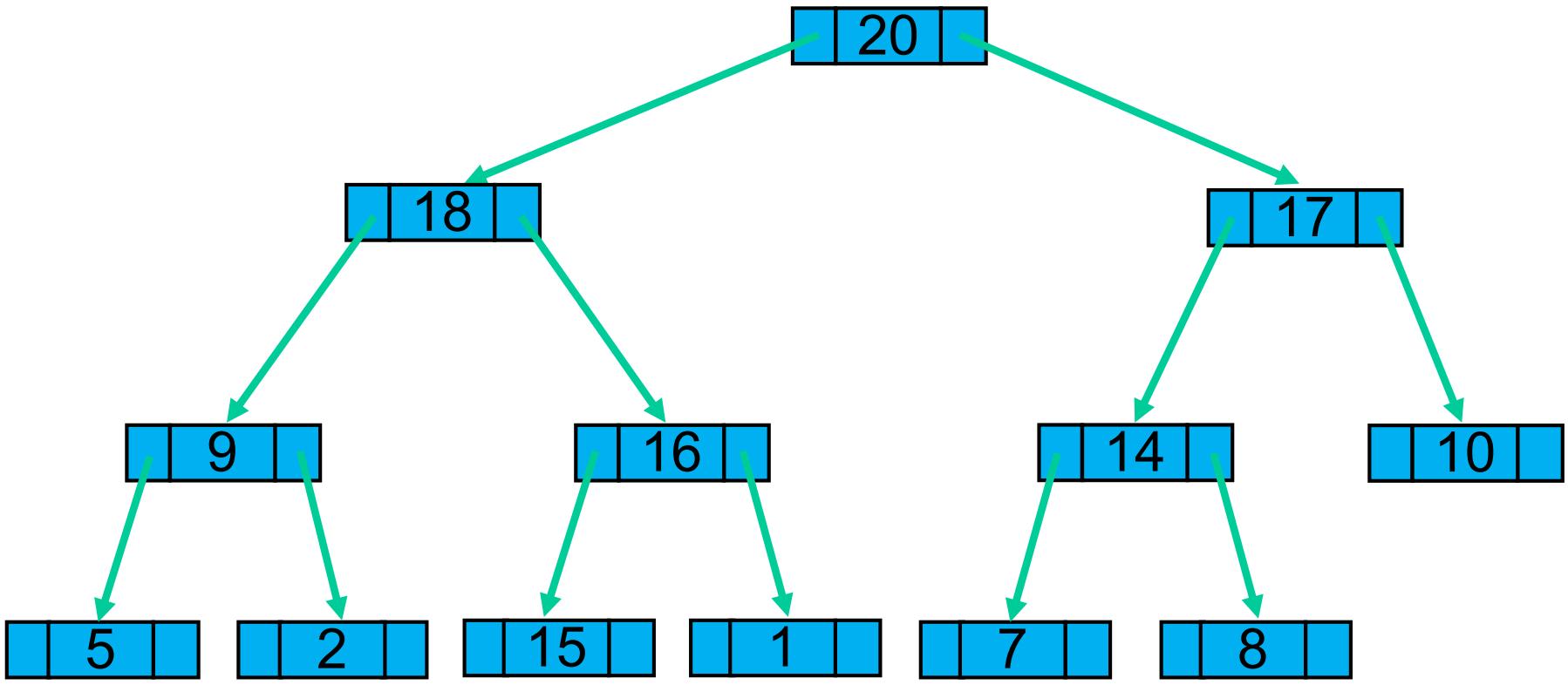
- A (max-)heap is a binary tree in which the root node is greater than its descendants
- The root is always the largest value, so finding it is **O(1)**
- Adding an element is **O(log<sub>2</sub>n)**

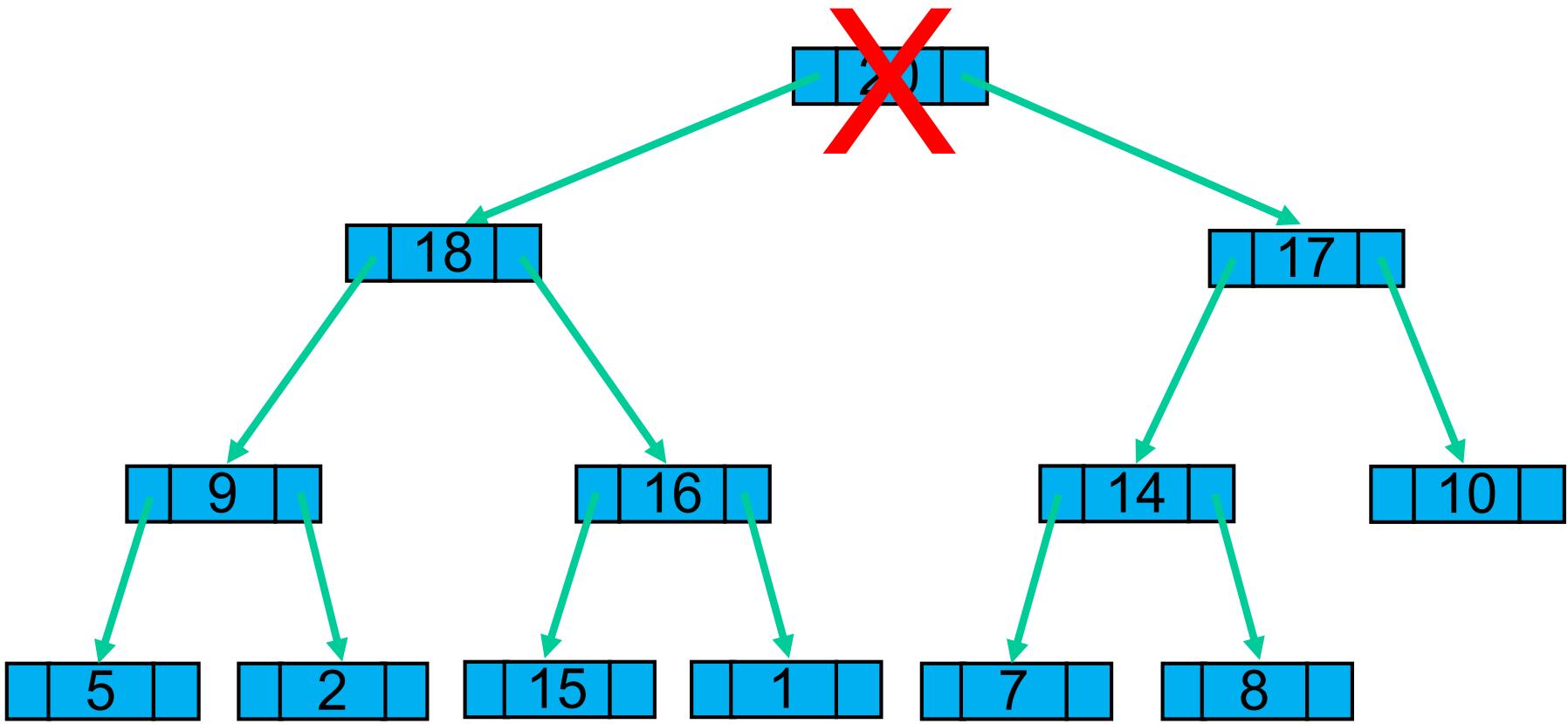
# **SD2x2.8**

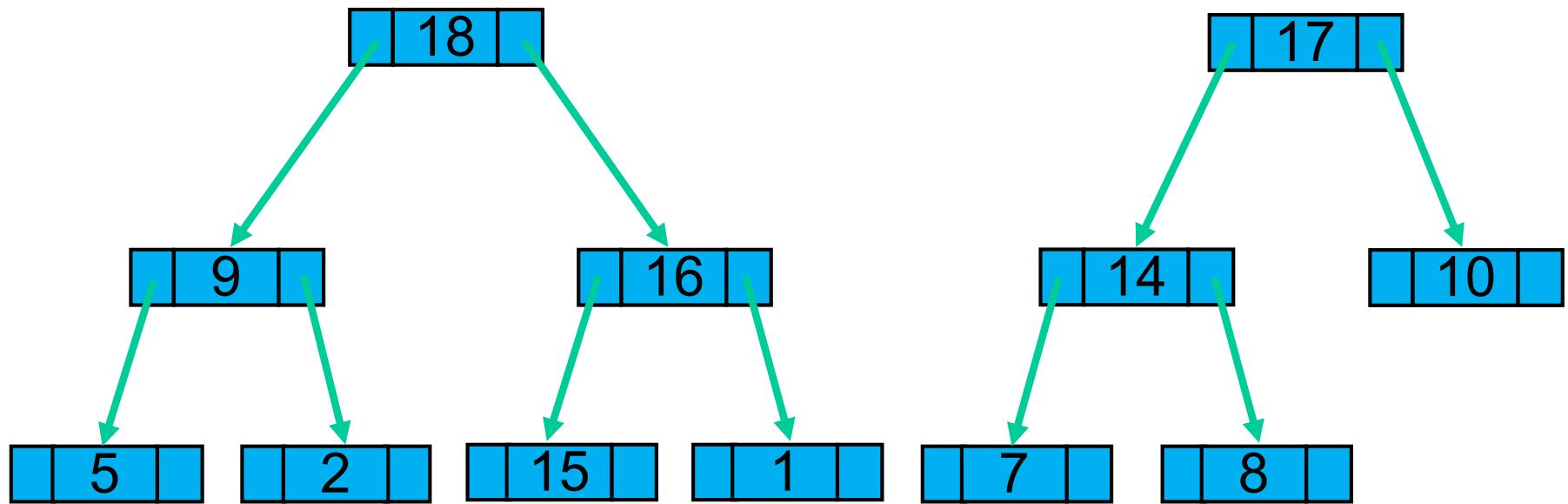
## **Heaps: remove**

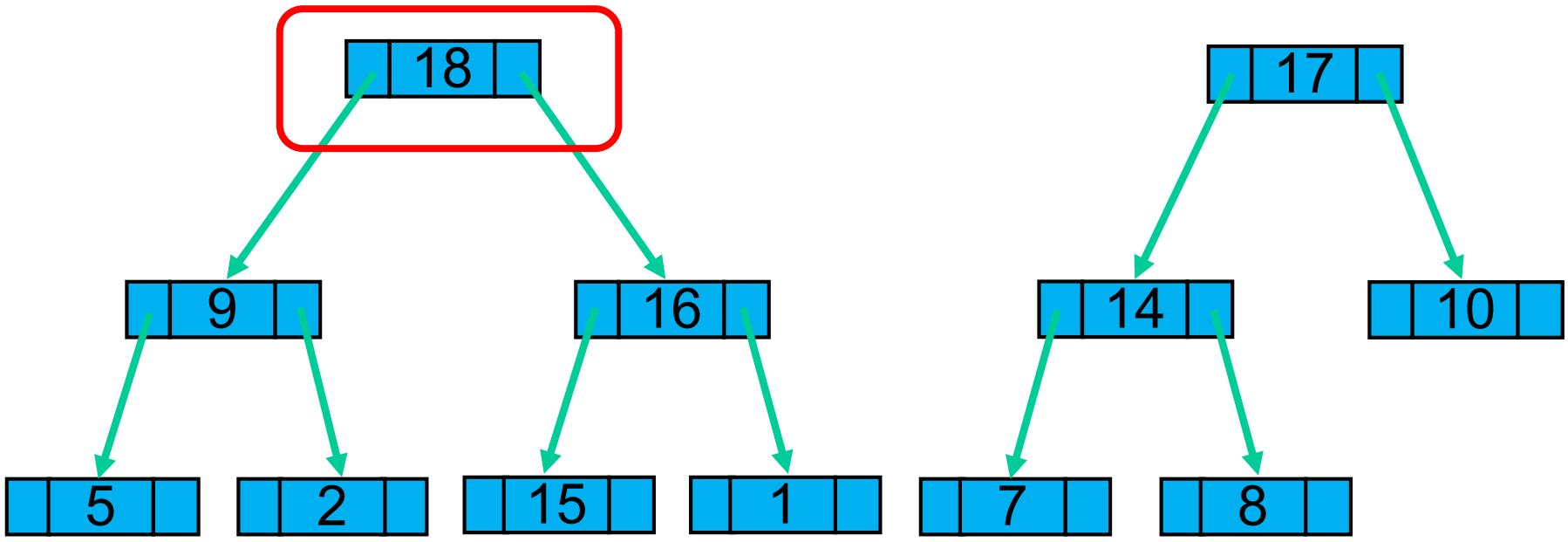
### **Kathy**

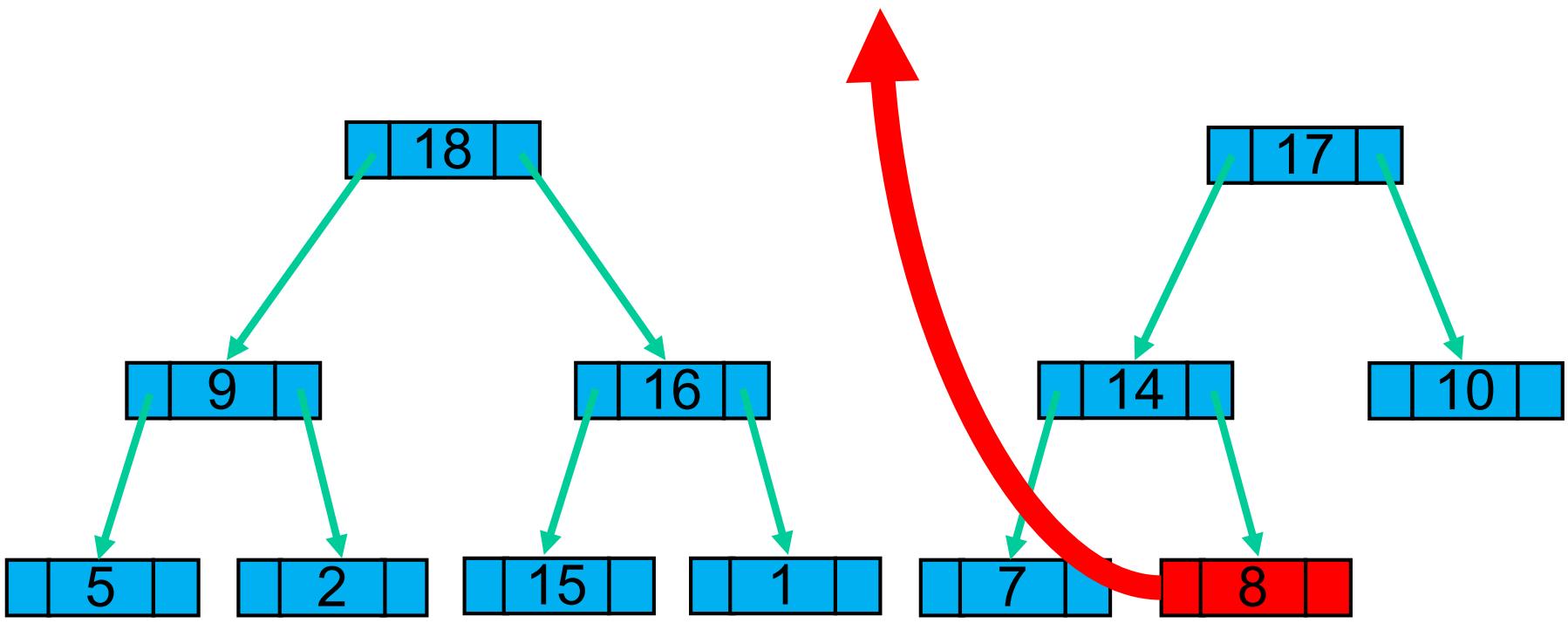
# How do we extract the largest value of a heap?

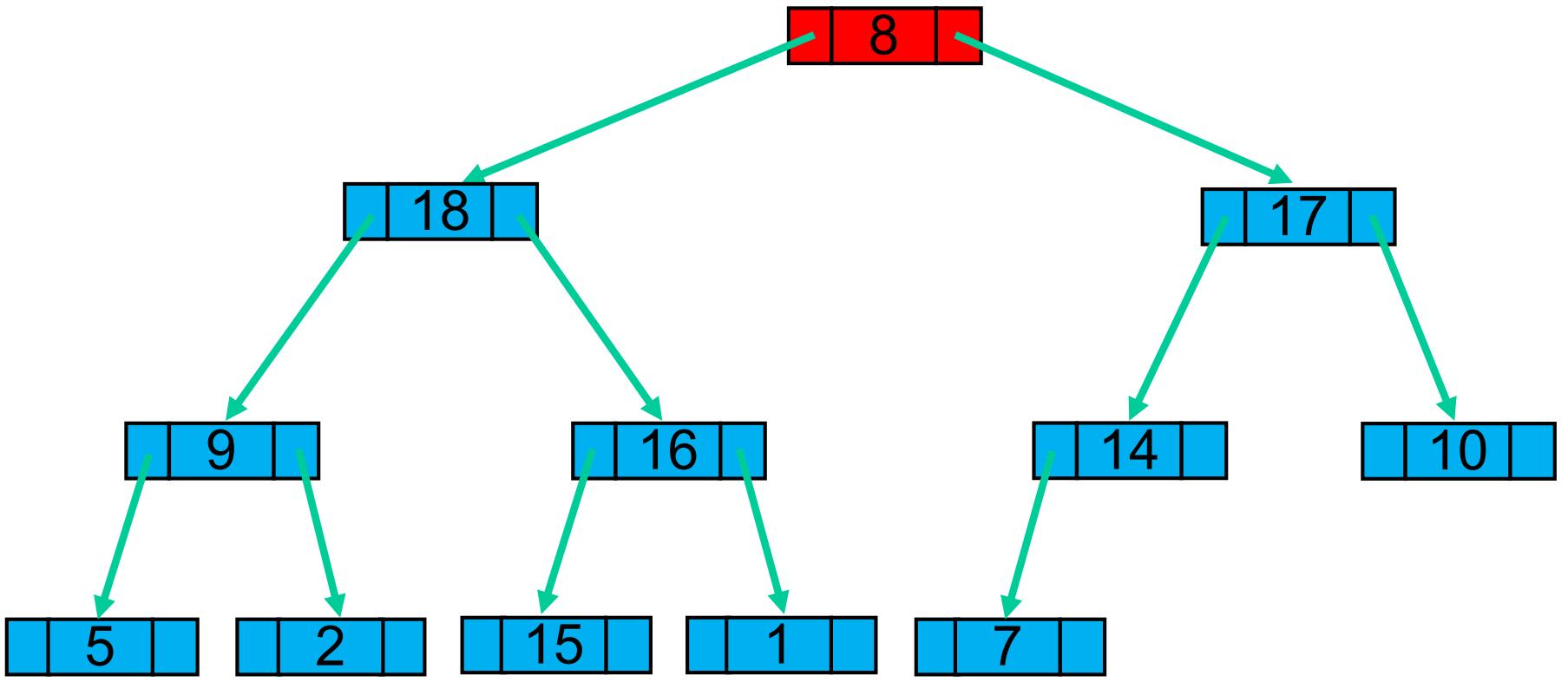


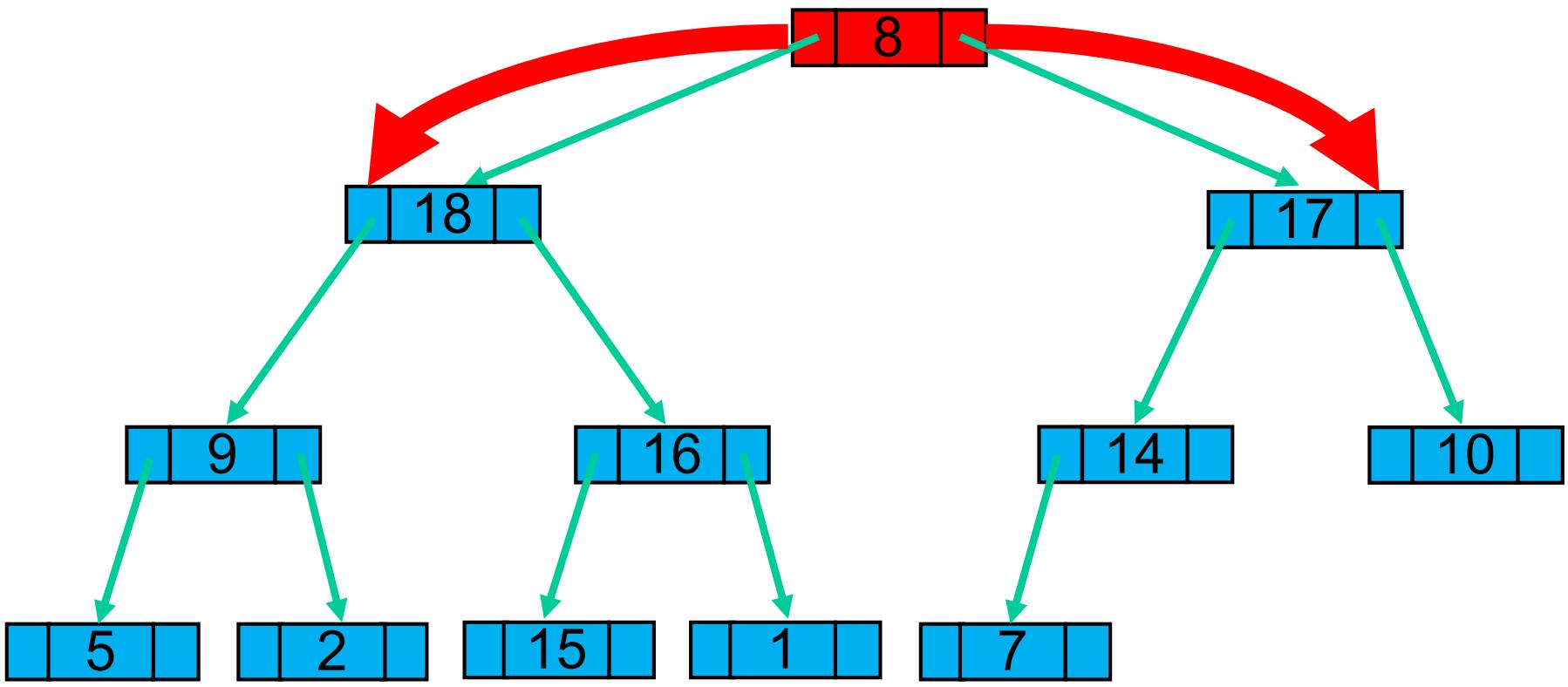


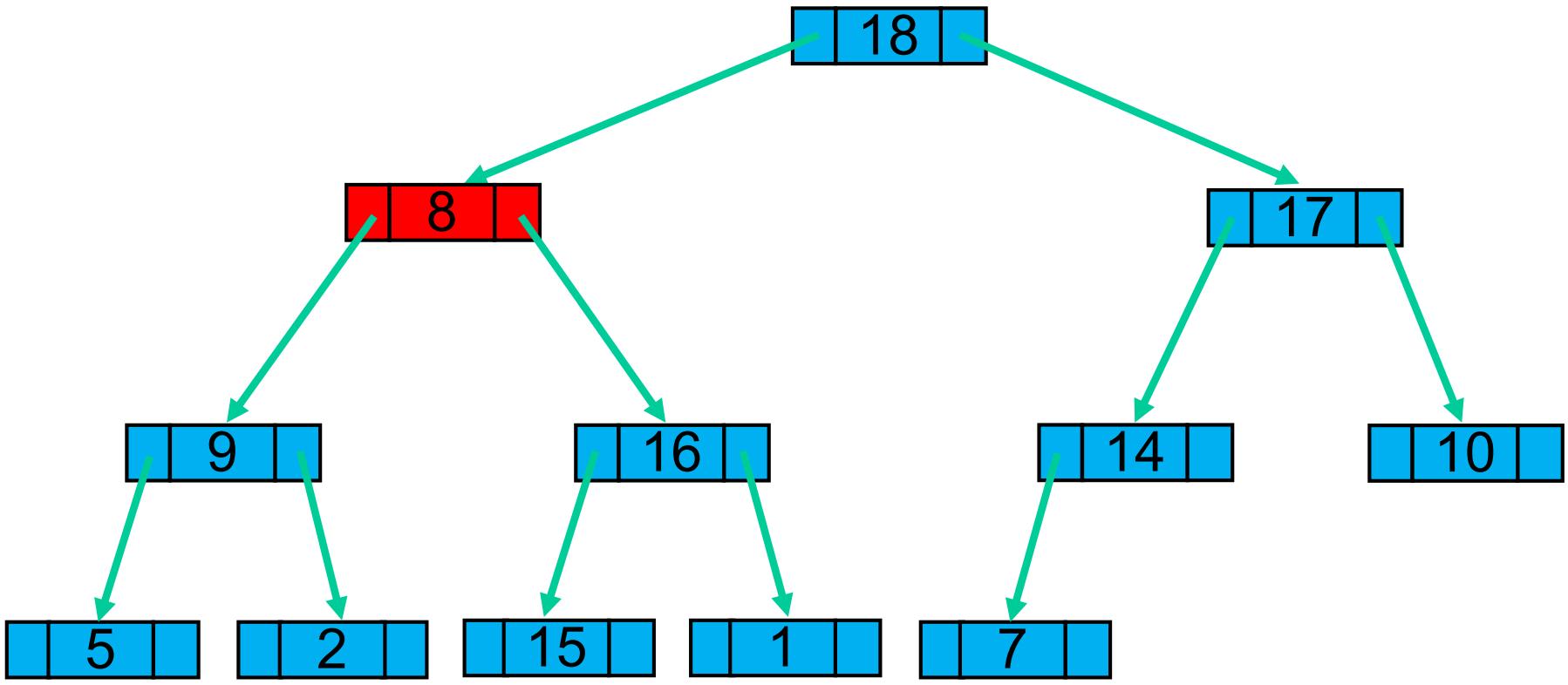


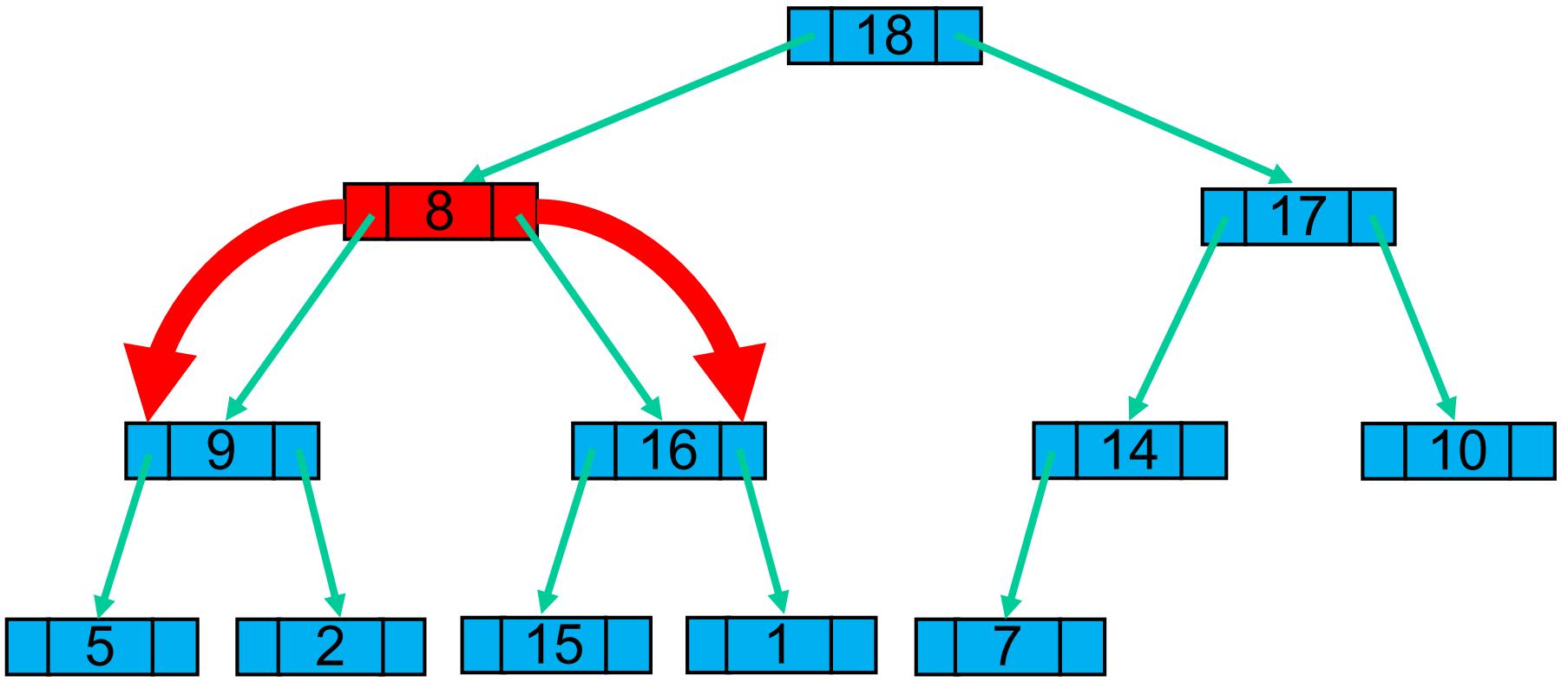


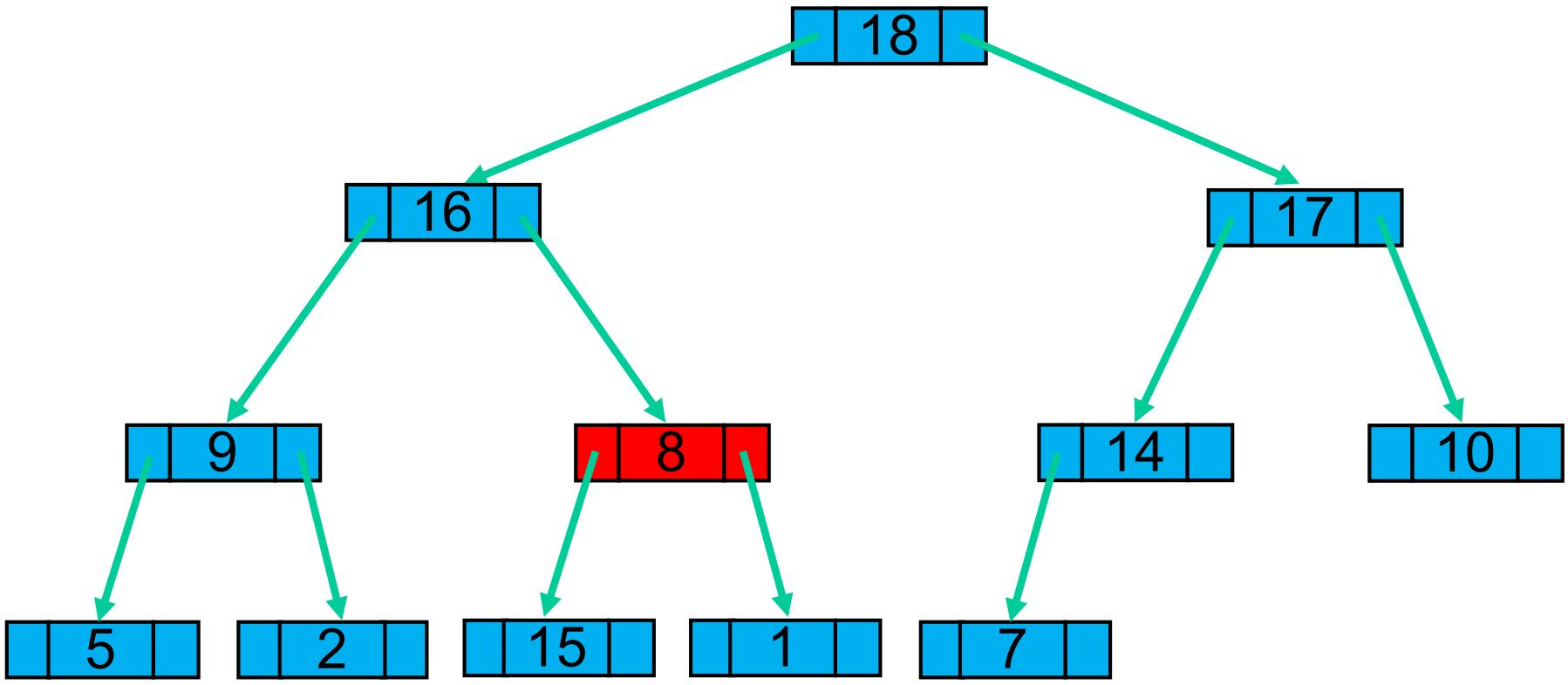


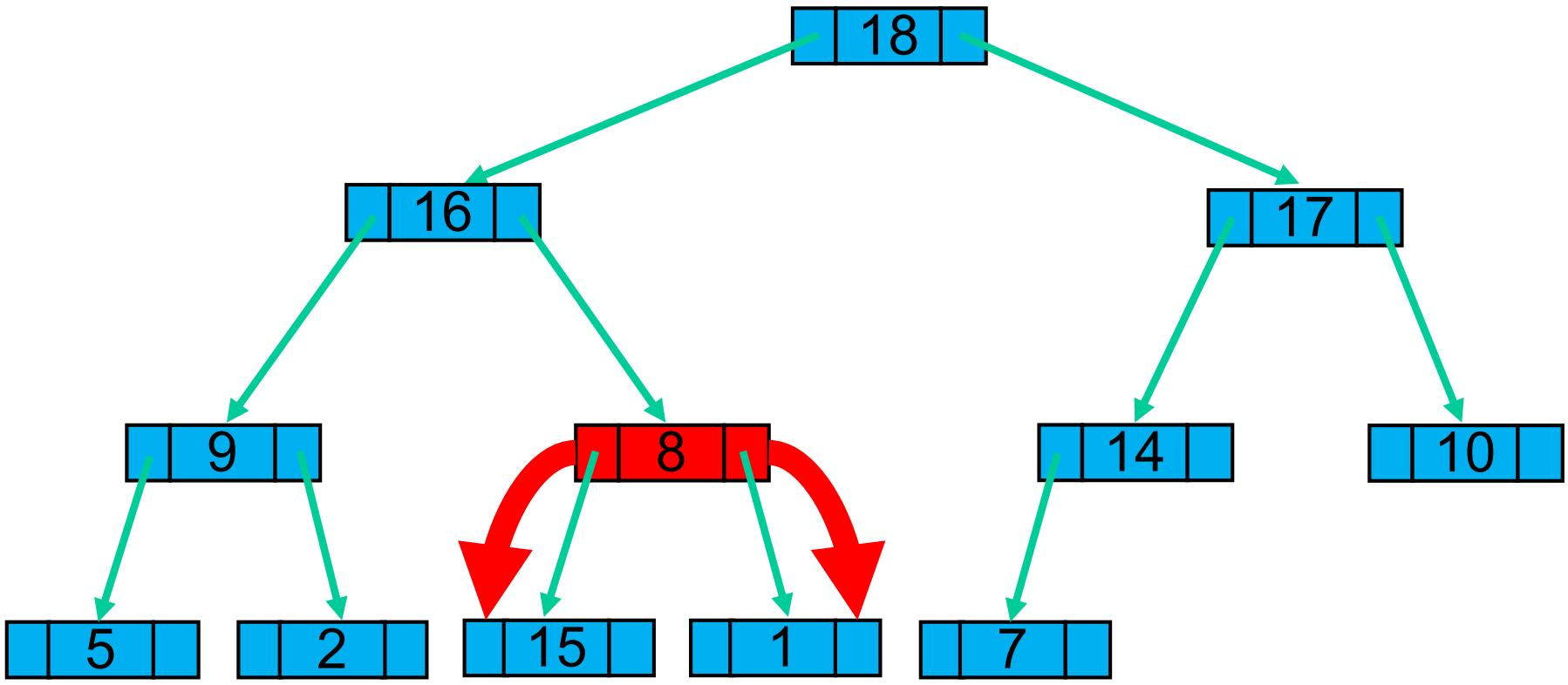


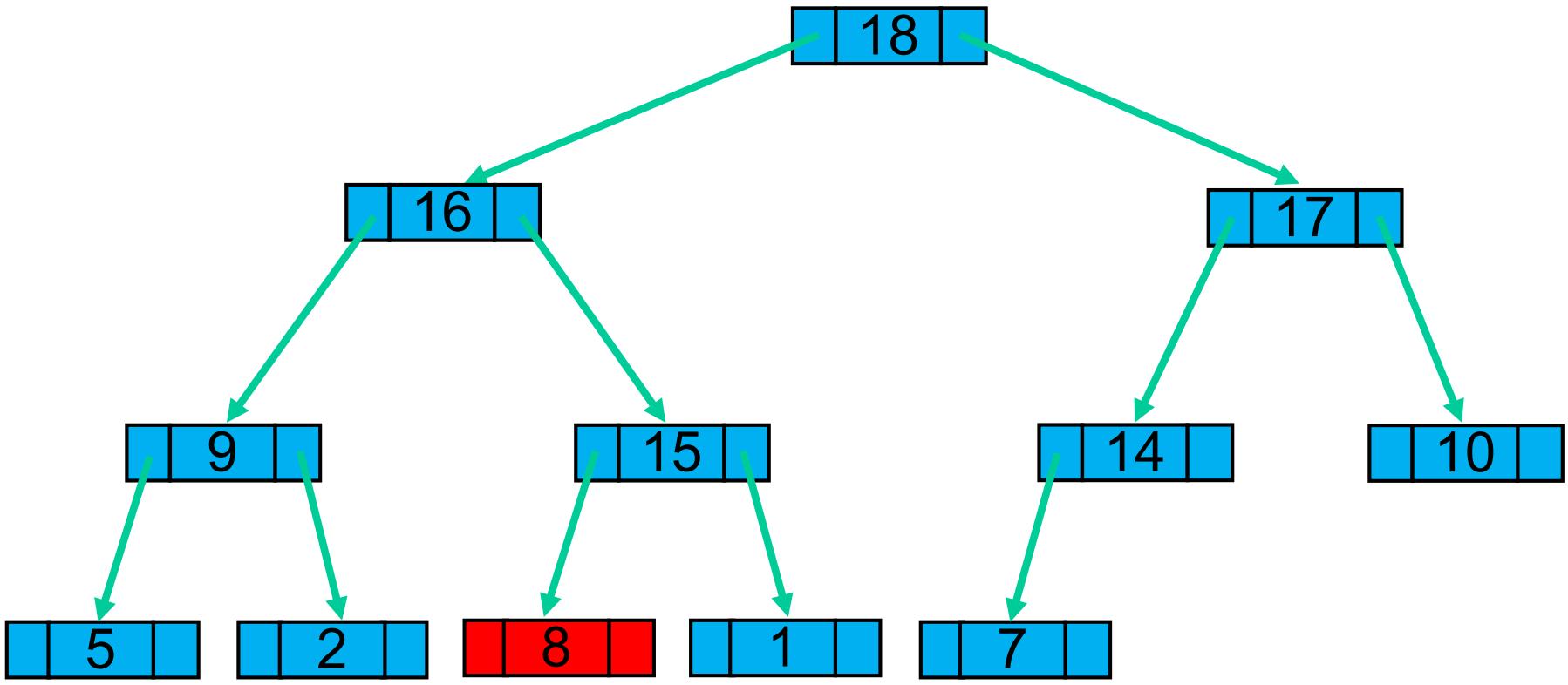


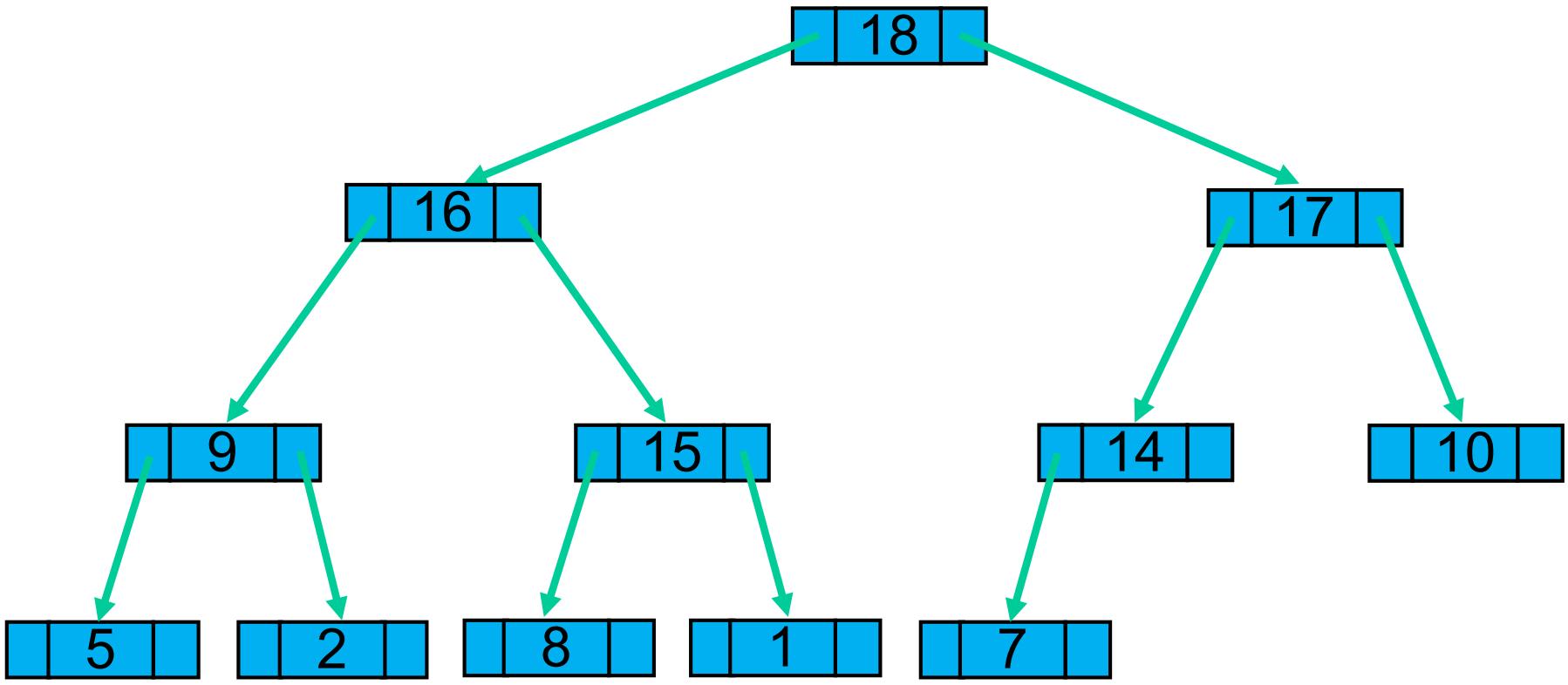












# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {
    if (size == 0) {
        throw new HeapException();
    }
    int extractMax = values[0];
    values[0] = values[size - 1];
    bubbleDown(0);
    size -= 1;
    return extractMax;
}

public void bubbleDown(int index) {
    int greaterChild;
    int leftIndex = (index * 2) + 1;
    int rightIndex = (index * 2) + 2;
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {
        greaterChild = rightIndex;
    } else if (leftIndex < size) {
        greaterChild = leftIndex;
    } else return;
    if (values[index] < values[greaterChild]) {
        swapValues(index, greaterChild);
        bubbleDown(greaterChild);
    }
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}
```

```
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size){  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Extracting from a Heap

---

```
public int extract() {  
    if (size == 0) {  
        throw new HeapException();  
    }  
    int extractMax = values[0];  
    values[0] = values[size - 1];  
    bubbleDown(0);  
    size -= 1;  
    return extractMax;  
}  
  
public void bubbleDown(int index) {  
    int greaterChild;  
    int leftIndex = (index * 2) + 1;  
    int rightIndex = (index * 2) + 2;  
    if (rightIndex < size && values[rightIndex] > values[leftIndex]) {  
        greaterChild = rightIndex;  
    } else if (leftIndex < size) {  
        greaterChild = leftIndex;  
    } else return;  
    if (values[index] < values[greaterChild]) {  
        swapValues(index, greaterChild);  
        bubbleDown(greaterChild);  
    }  
}
```

# Heaps in Java: Priority Queues

---

- **java.util.PriorityQueue<E>**
- **add(Element):** inserts element into **min-heap**
- **remove():** retrieves and removes root of the heap (**minimum** value; head of the queue)
- **peek():** retrieves but does not remove root of the heap (head of the queue)

```
public class Mentee implements Comparable<Mentee> {  
    protected int year;  
    protected String name;  
  
    public Mentee(int year, String name) {  
        this.year = year;  
        this.name = name;  
    }  
  
    public int compareTo(Mentee otherMentee) {  
        return year - otherMentee.year;  
    }  
}
```

```
public class Mentee implements Comparable<Mentee> {  
    protected int year;  
    protected String name;  
  
    public Mentee(int year, String name) {  
        this.year = year;  
        this.name = name;  
    }  
  
    public int compareTo(Mentee otherMentee) {  
        return year - otherMentee.year;  
    }  
}
```

```
public class Mentee implements Comparable<Mentee> {  
    protected int year;  
    protected String name;  
  
    public Mentee(int year, String name) {  
        this.year = year;  
        this.name = name;  
    }  
  
    public int compareTo(Mentee otherMentee) {  
        return year - otherMentee.year;  
    }  
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
return menteesInProgram;
}
```

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

**Prospective mentee list      can accept 3**

- Kathy, year 2
- Jane, year 1
- Eliana, year 3
- Yujie, year 5
- Swapneel, year 4

```
public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}
```

**Prospective mentee list      can accept 3**

- Kathy, year 2
- Jane, year 1
- Eliana, year 3
- Yujie, year 5
- Swapneel, year 4

```

public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}

```

**Prospective mentee list      can accept 2**

- Kathy, year 2
- Jane, year 1      **<- 1. accept into program**
- Eliana, year 3
- Yujie, year 5
- Swapneel, year 4

```

public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}

```

**Prospective mentee list      can accept 1**

- Kathy, year 2    <- 2. accept into program
- Jane, year 1    <- 1. accept into program
- Eliana, year 3
- Yujie, year 5
- Swapneel, year 4

```

public static String[] acceptMenteesIntoProgram(
    int numMentorsAvailable,
    PriorityQueue<Mentee> interestedMentees) {

    int numToAccept = Math.min(interestedMentees.size(),
                               numMentorsAvailable);

    String[] menteesInProgram = new String[numToAccept];
    for (int i = 0; i < numToAccept; i++) {
        Mentee mentee = interestedMentees.remove();
        menteesInProgram[i] = mentee.name;
    }
    return menteesInProgram;
}

```

**Prospective mentee list      can accept 0**

- Kathy, year 2      <- 2. accept into program
- Jane, year 1      <- 1. accept into program
- Eliana, year 3      <- 3. accept into program
- Yujie, year 5
- Swapneel, year 4

# Recap: Heaps

---

- Removing the largest element of the heap is **O(log<sub>2</sub>n)**
- Min-Heaps are implemented in the Java API using the **PriorityQueue** class

# **SD2x2.9**

## **Graph terminology**

### **Chris**

# Motivating Example

---

- How can we represent relationships between elements?
- The links in LinkedLists, Trees, etc. are for structure, but not for relationships
- Maps only allow a single relationship from a Key to a Value

# Graphs

---

- **Model relationships between objects**
- Example: We're planning a road trip and we want to visit multiple cities in the U.S.
- How could we represent information about driving distances between cities?

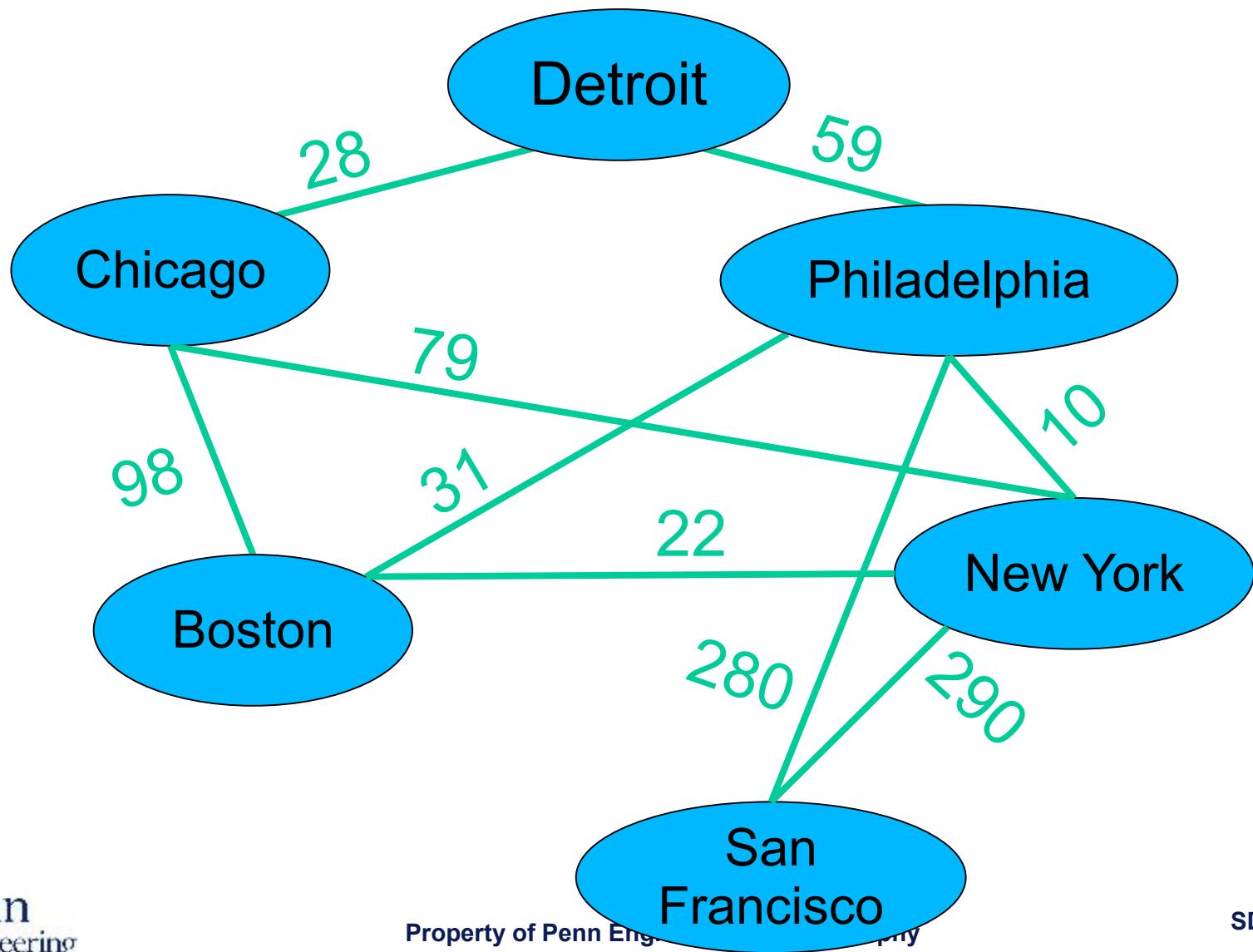
# Driving Distances between cities

---

- Detroit is 280 miles away from Chicago.
- Detroit is 590 miles away from Philadelphia.
- Philadelphia is 100 miles away from New York City.
- Philadelphia is 2,800 miles away from San Francisco.
- Chicago is 980 miles away from Boston.  
. . . and so on.

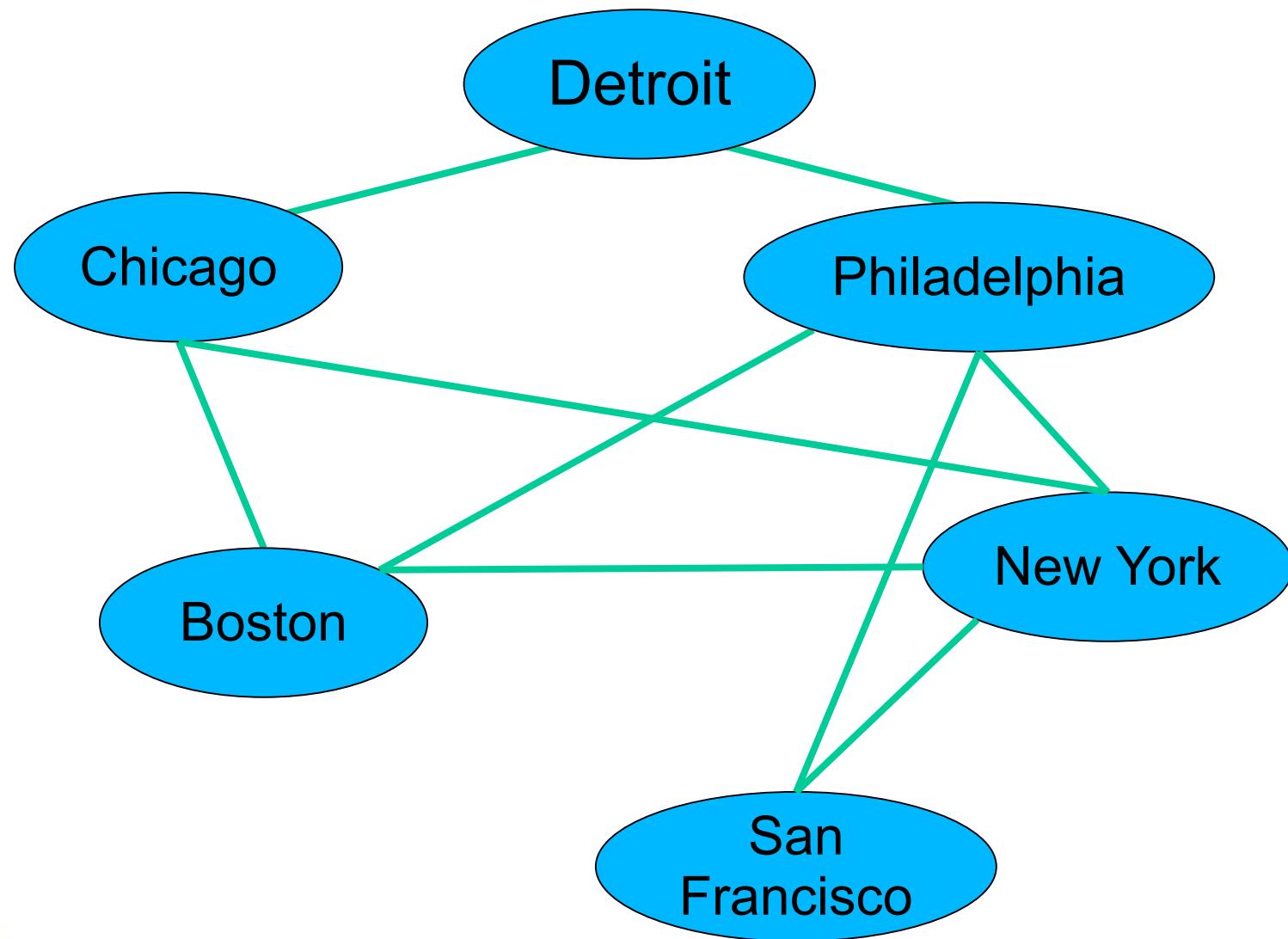
# Driving Distances between cities (x 10 miles)

---

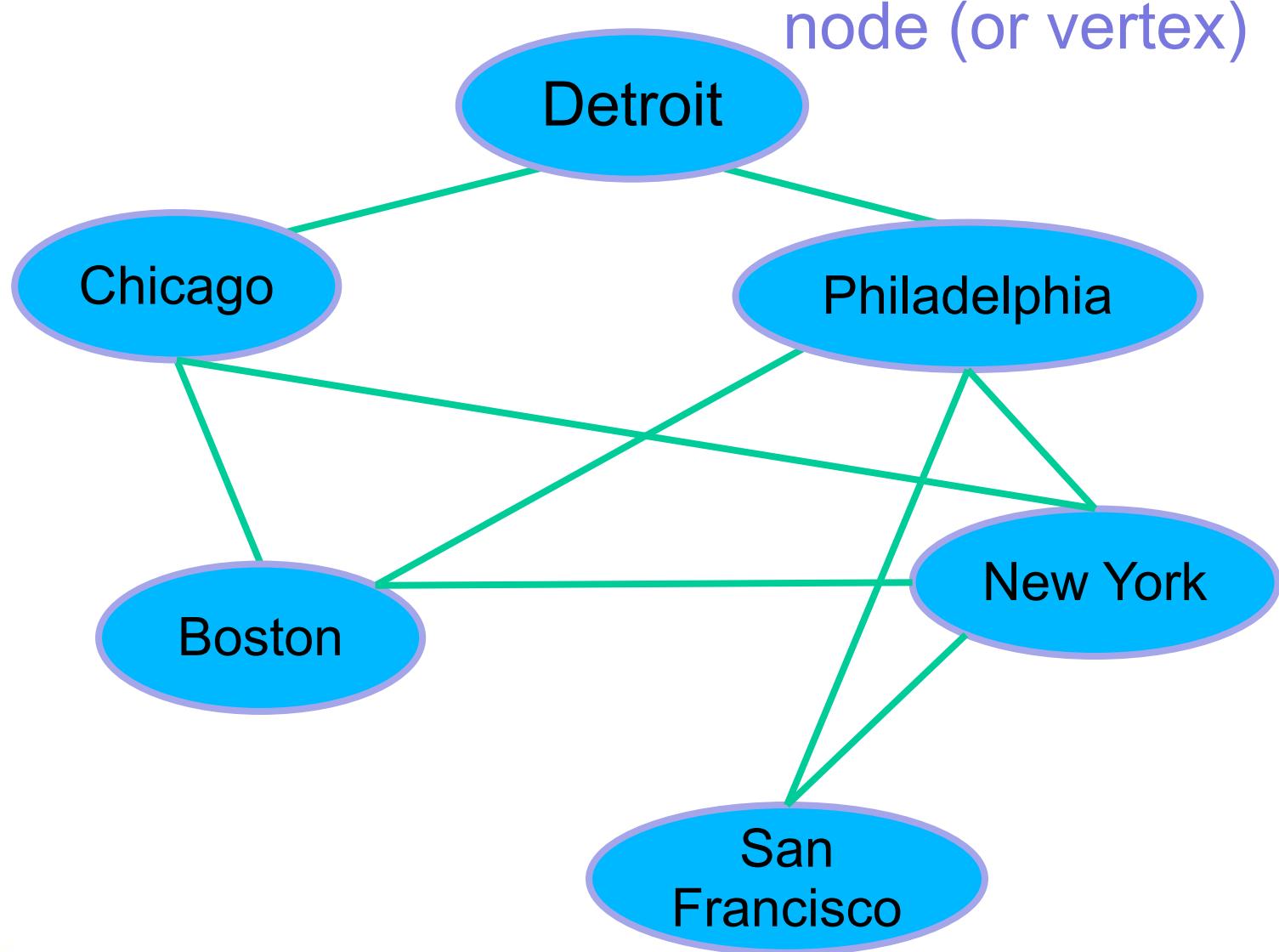


# Graph Terminology

---



# General Graph Terminology



# General Graph Terminology

---



node

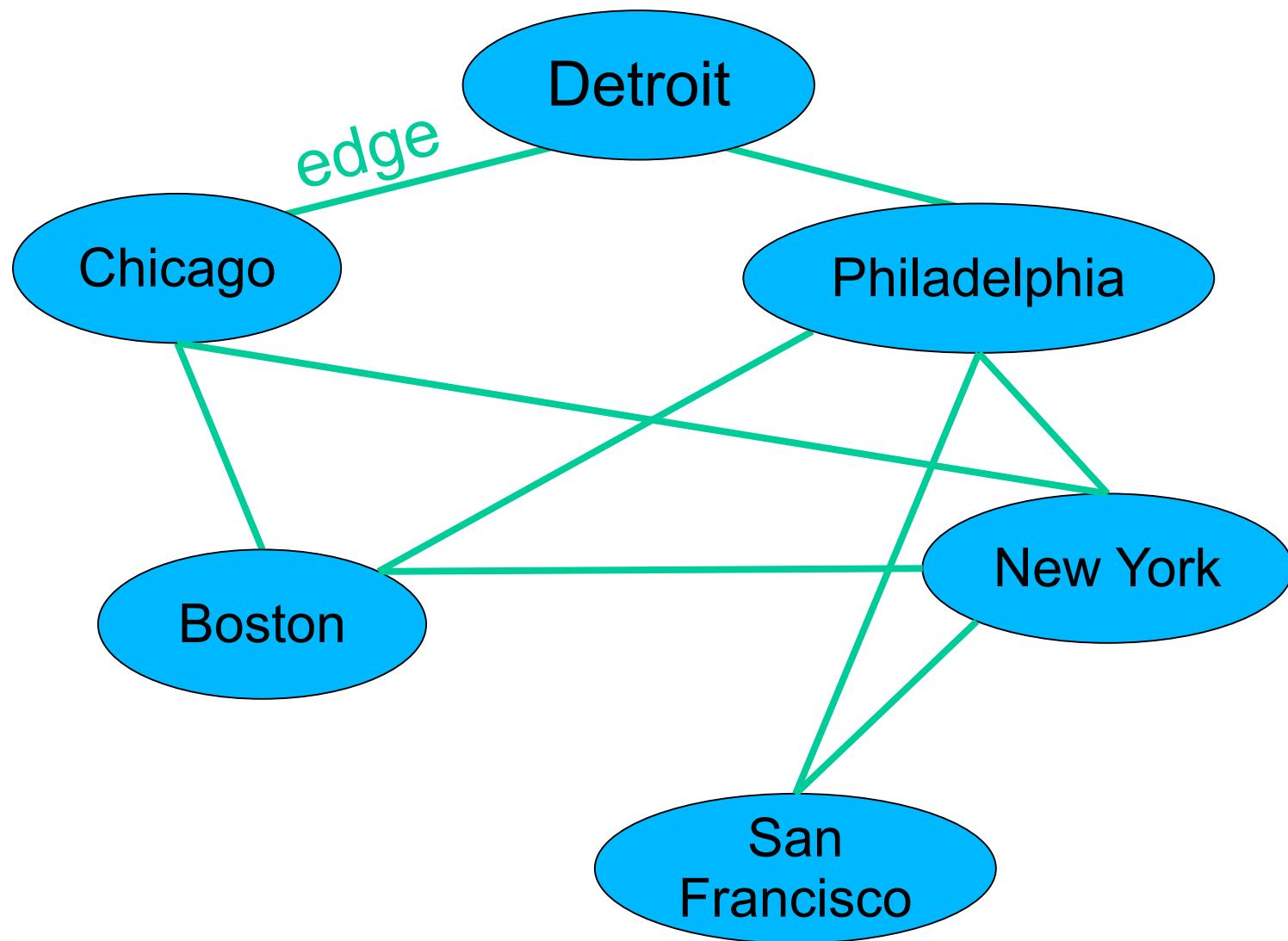
# General Graph Terminology

---



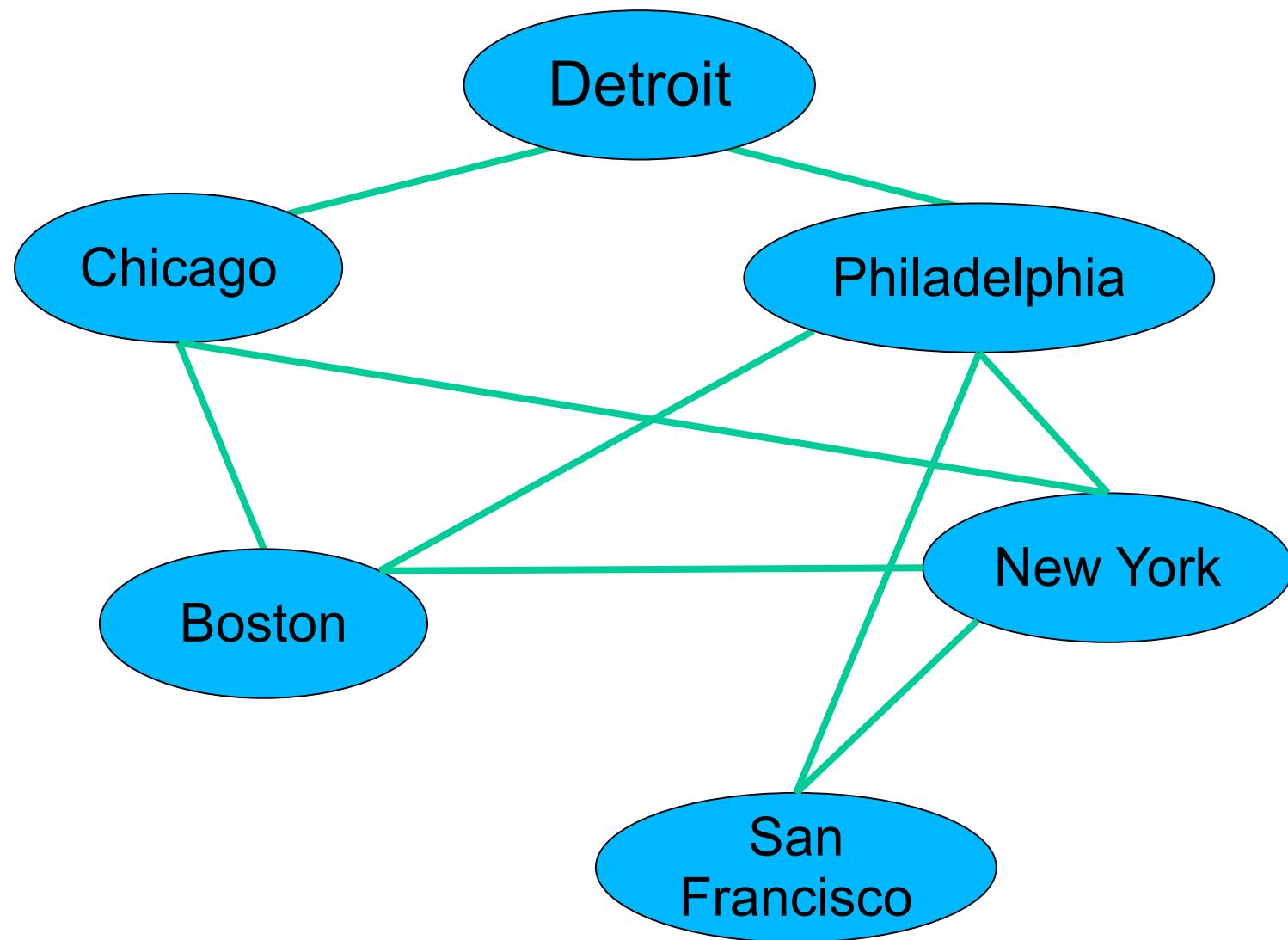
# General Graph Terminology

---



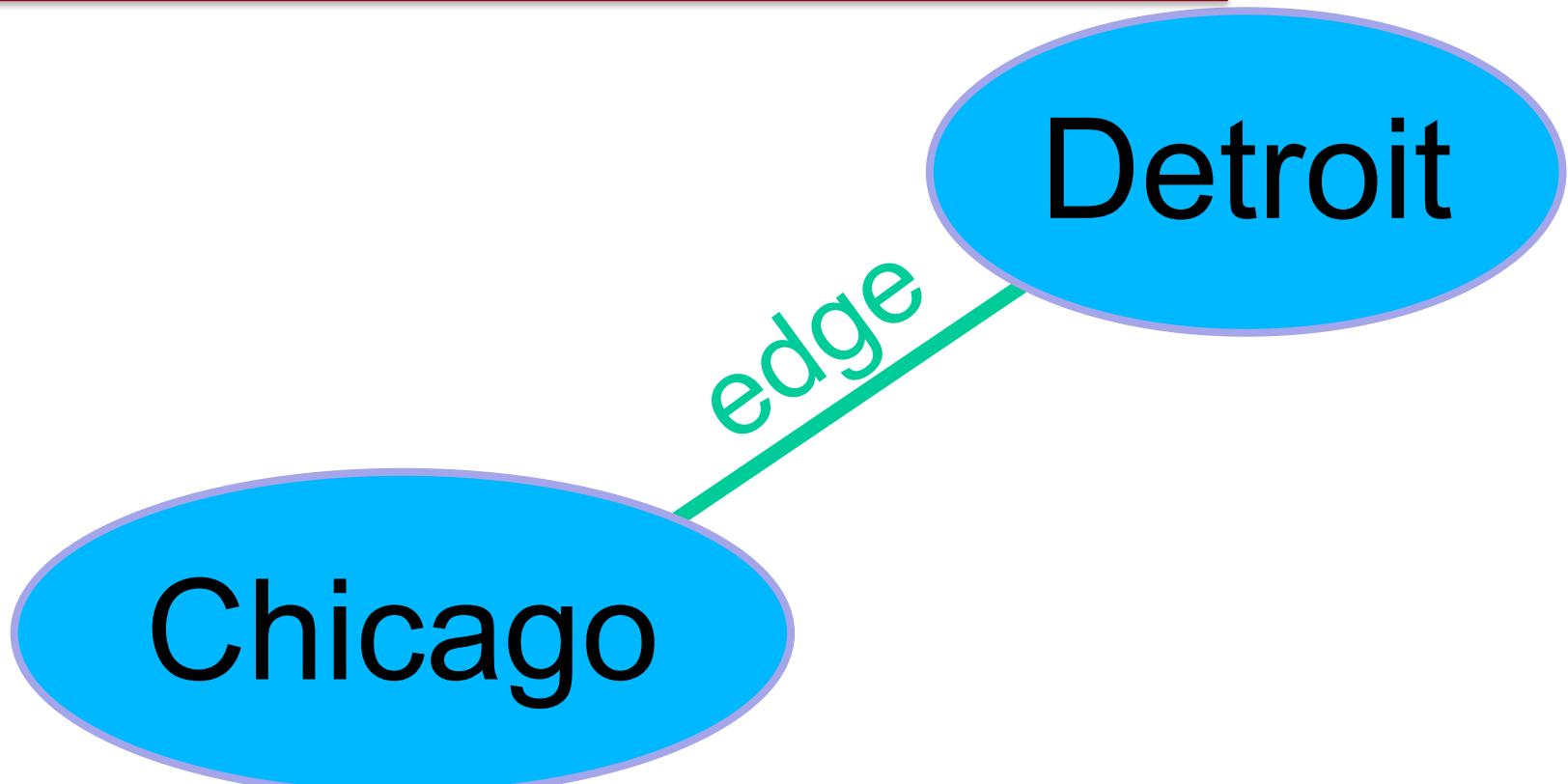
# Undirected Graph

---



# General Graph Terminology

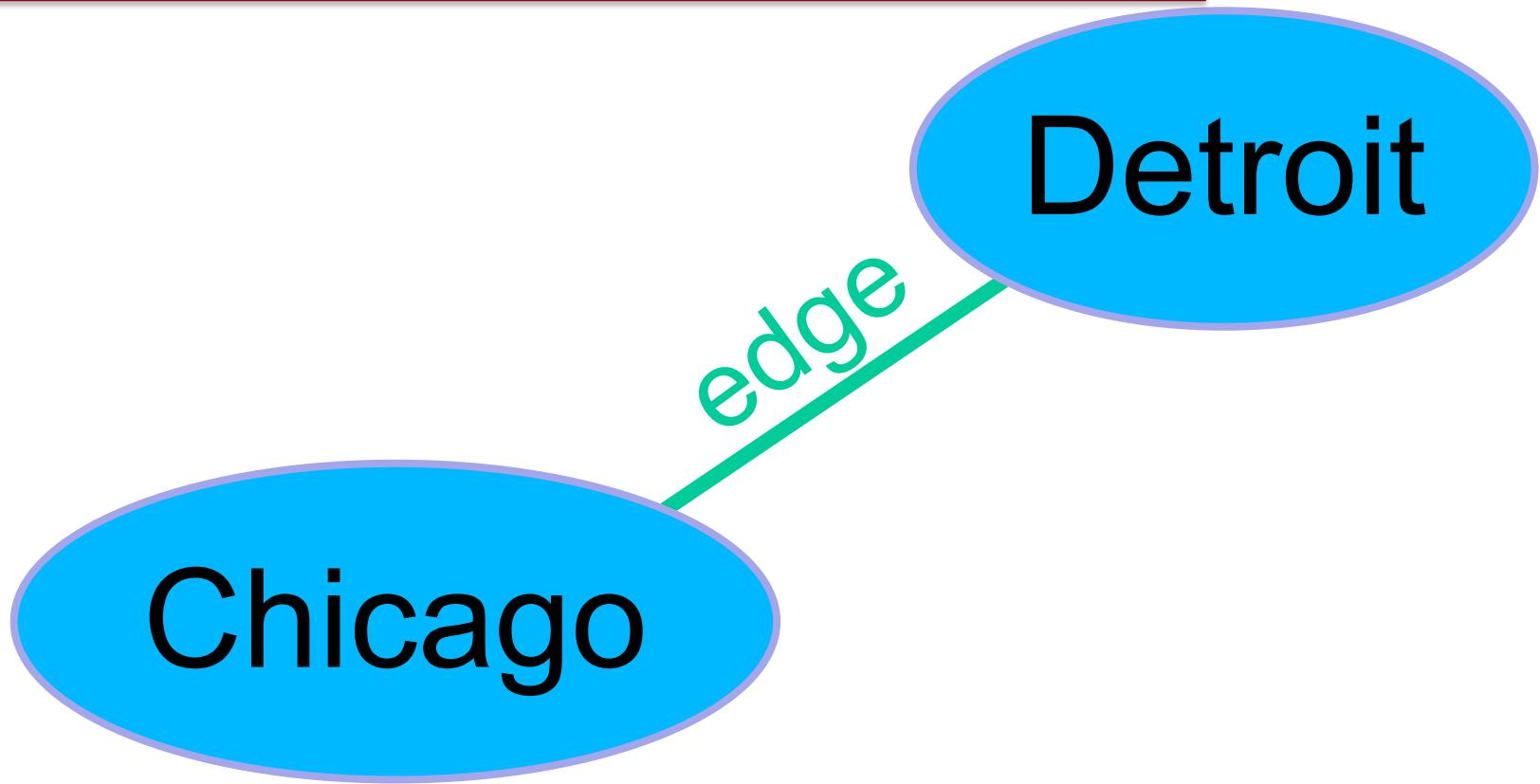
---



nodes *share* an edge

# General Graph Terminology

---



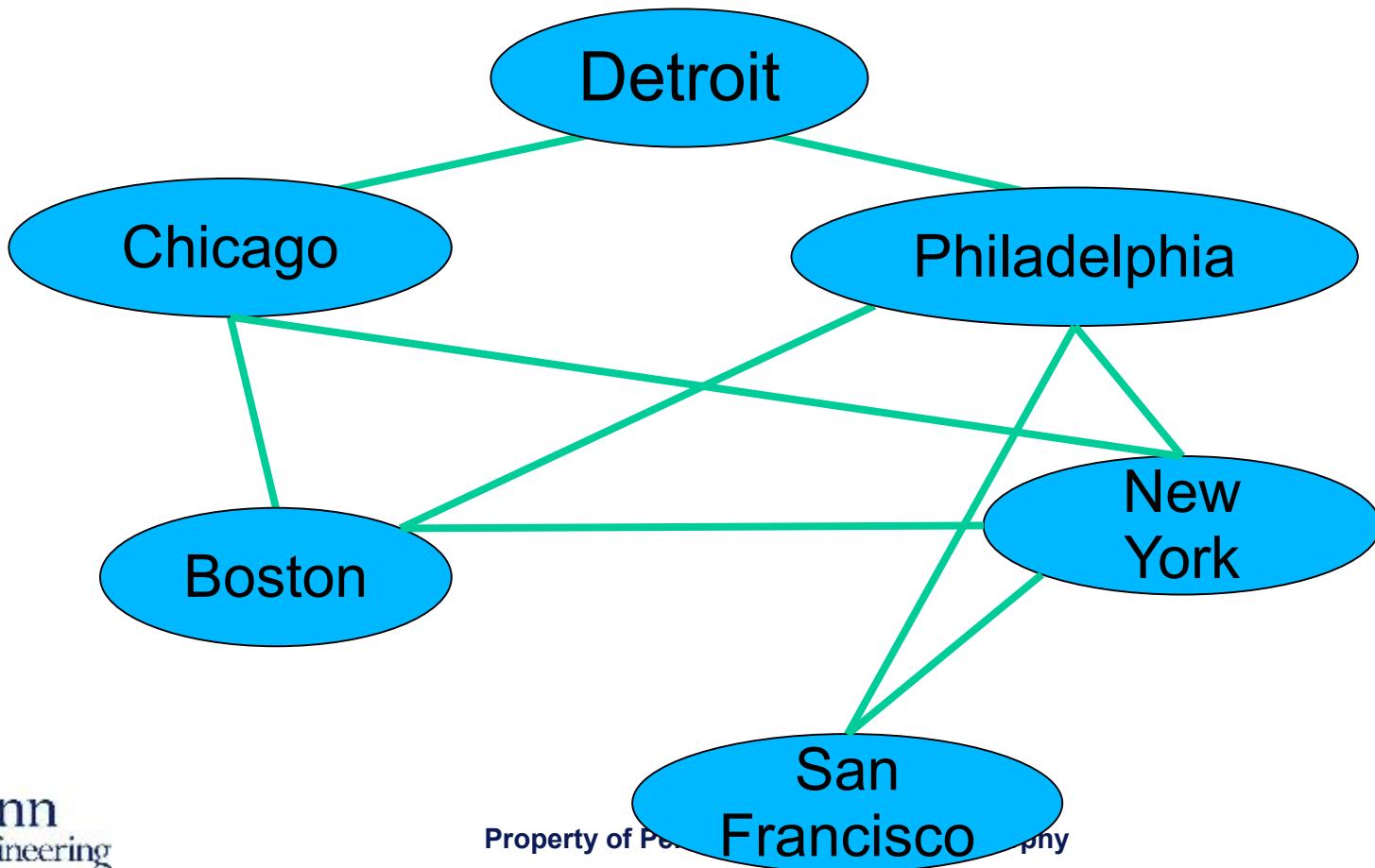
nodes *share* an edge

an edge connects two nodes

# General Graph Terminology

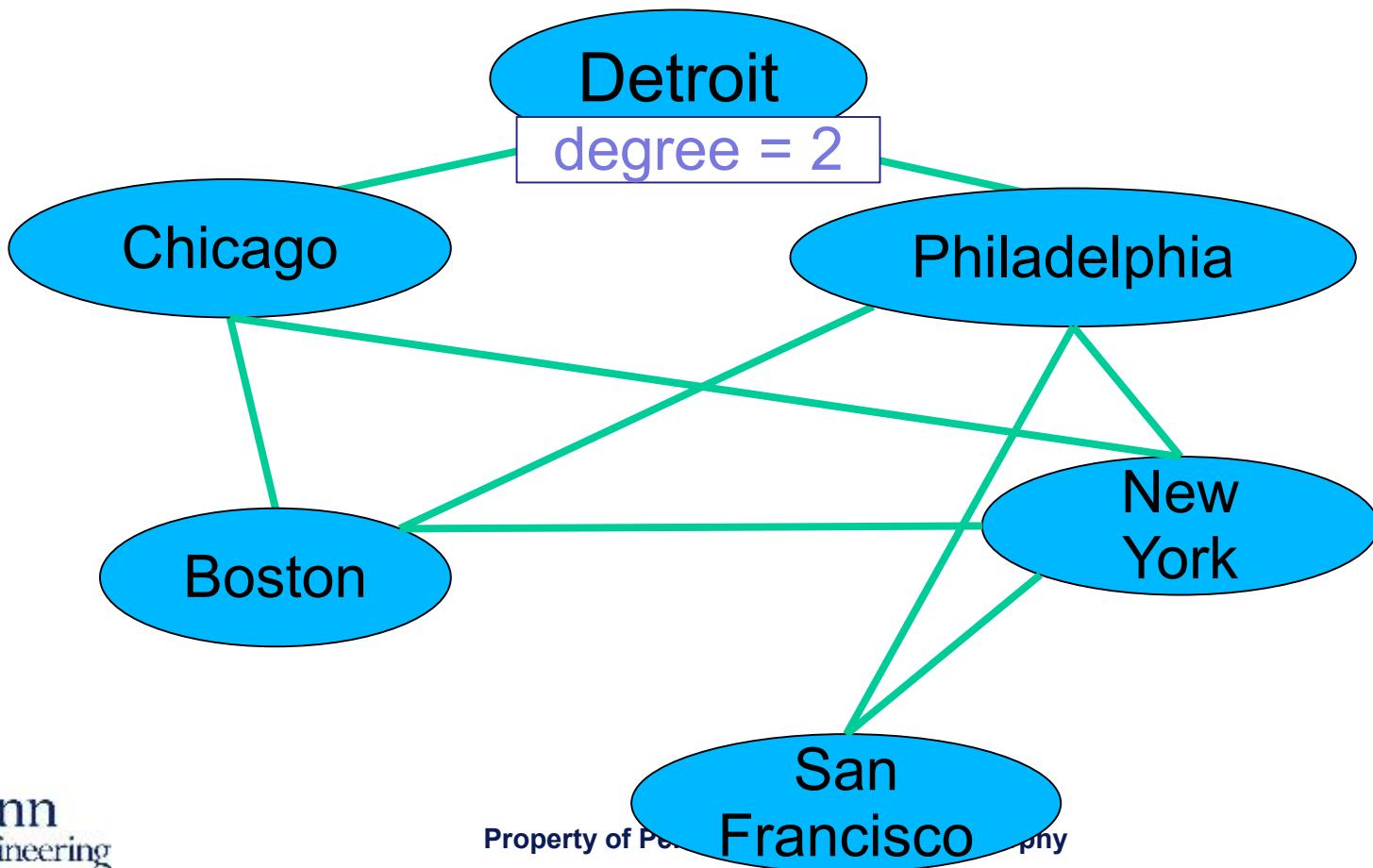
---

The **degree** of a node is the number of edges it has.



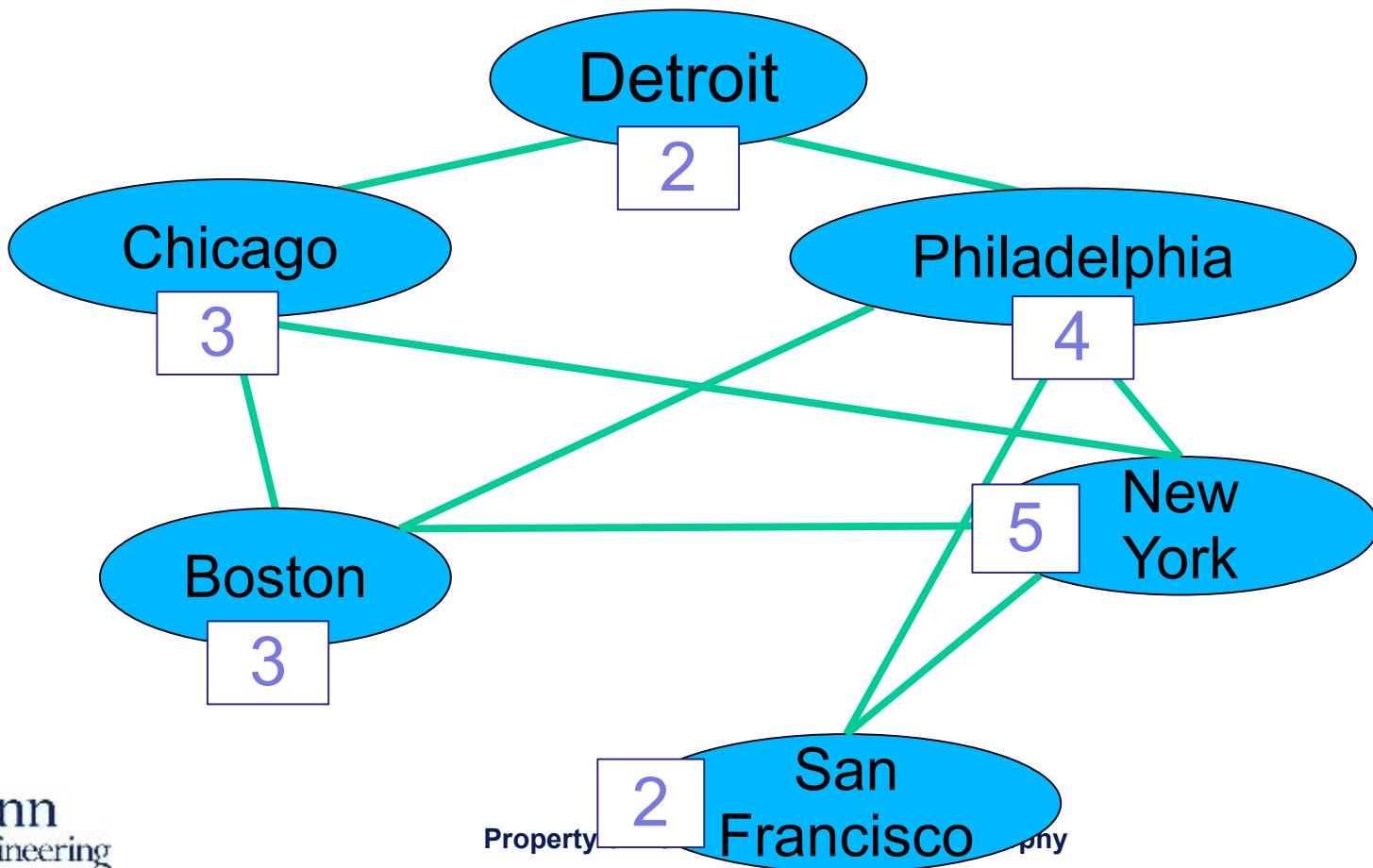
# General Graph Terminology

The **degree** of a node is the number of edges it has.



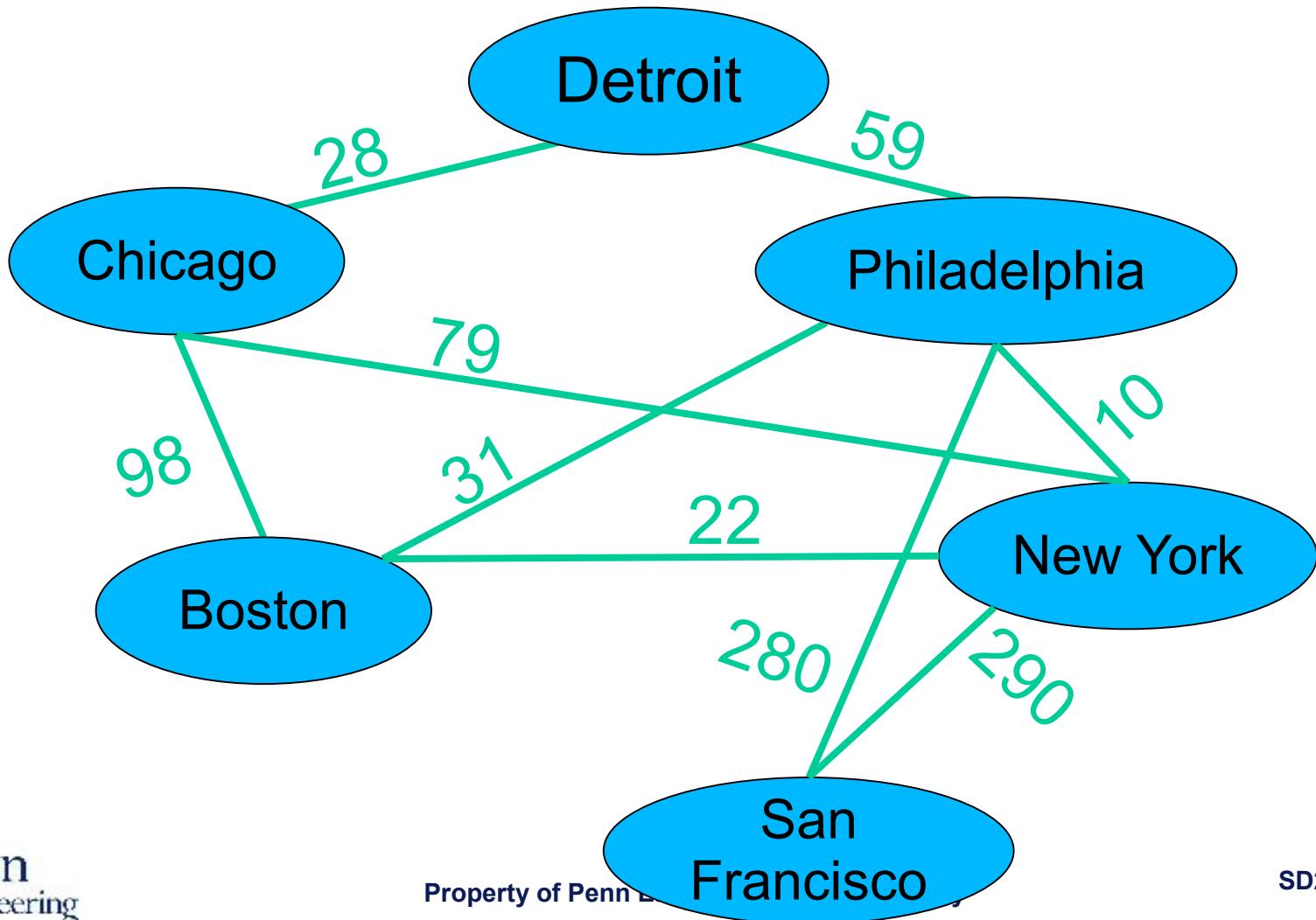
# General Graph Terminology

The **degree** of a node is the number of edges it has.



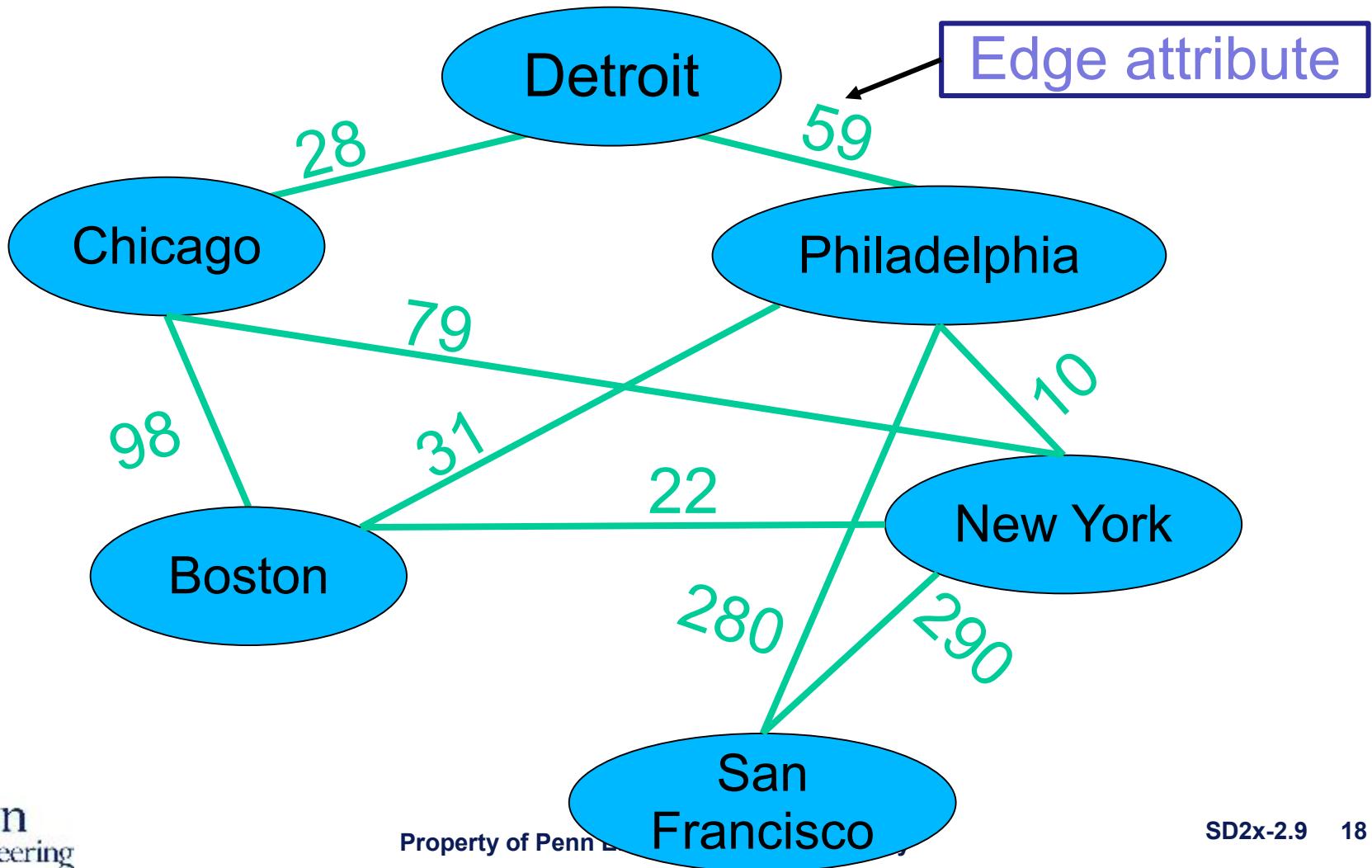
# General Graph Terminology

Graphs can be **weighted**.



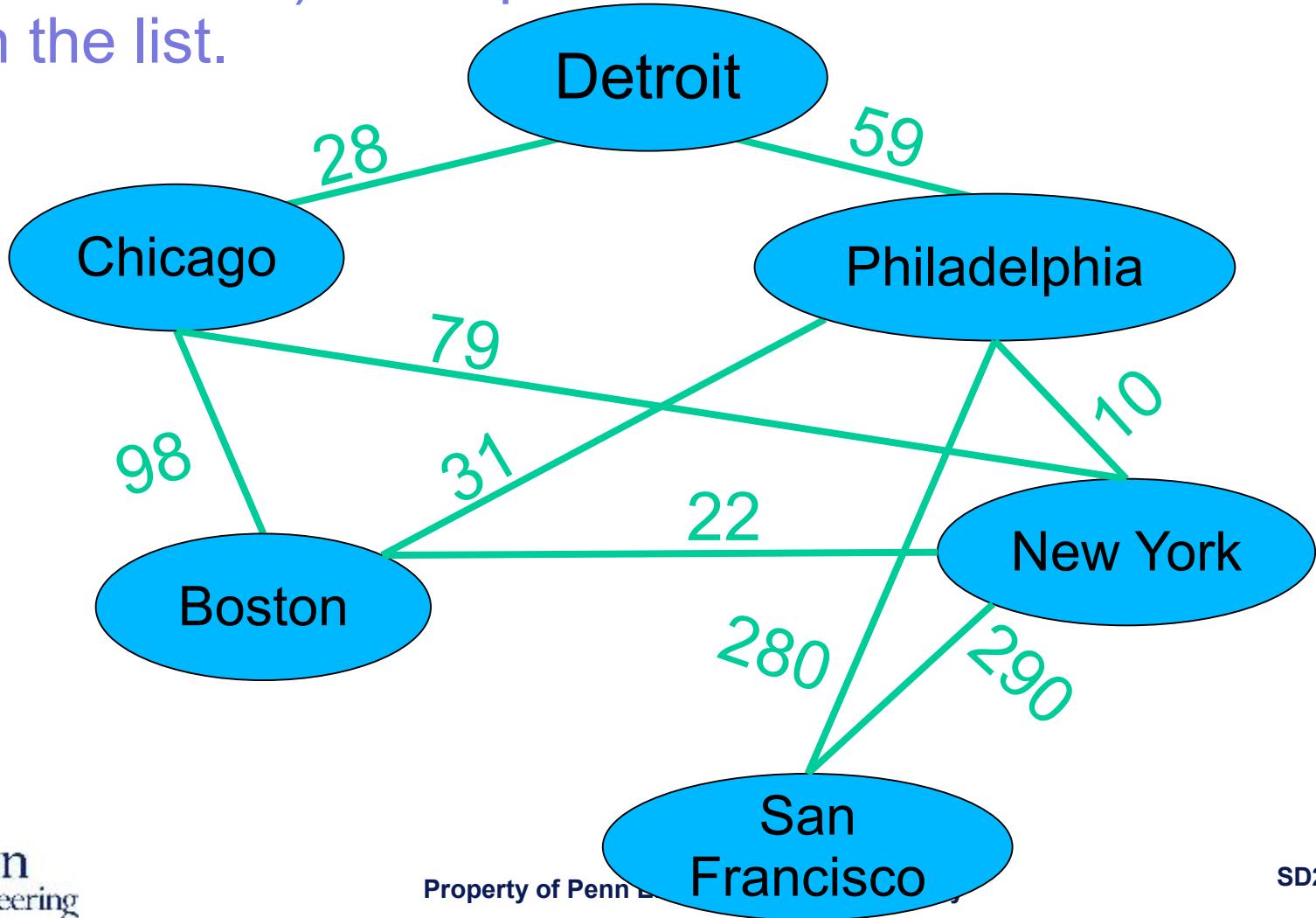
# General Graph Terminology

Graphs can be **weighted**.



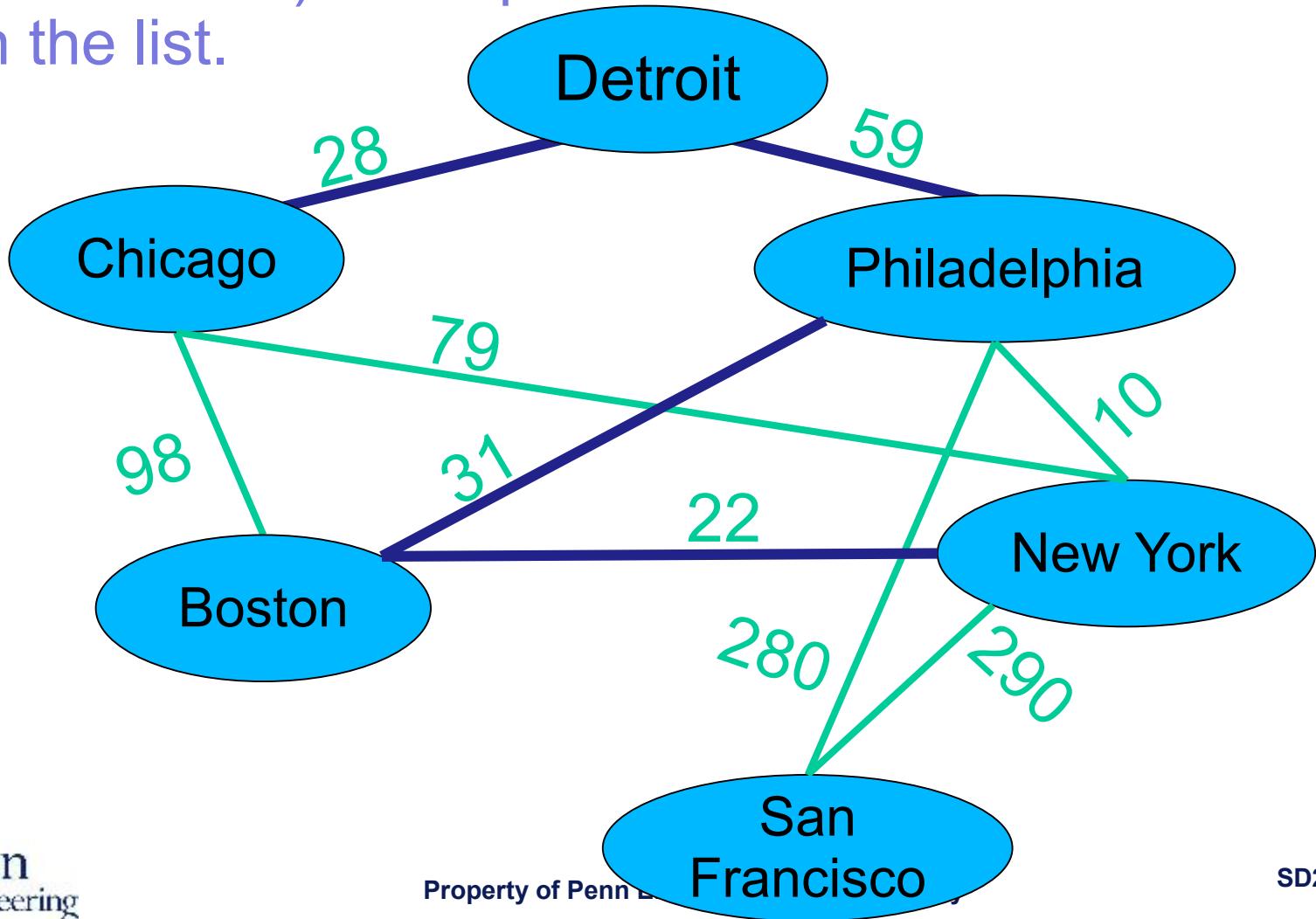
# General Graph Terminology

A **path** is a list of edges such that each node (but the last) is the predecessor of the next node in the list.



# General Graph Terminology

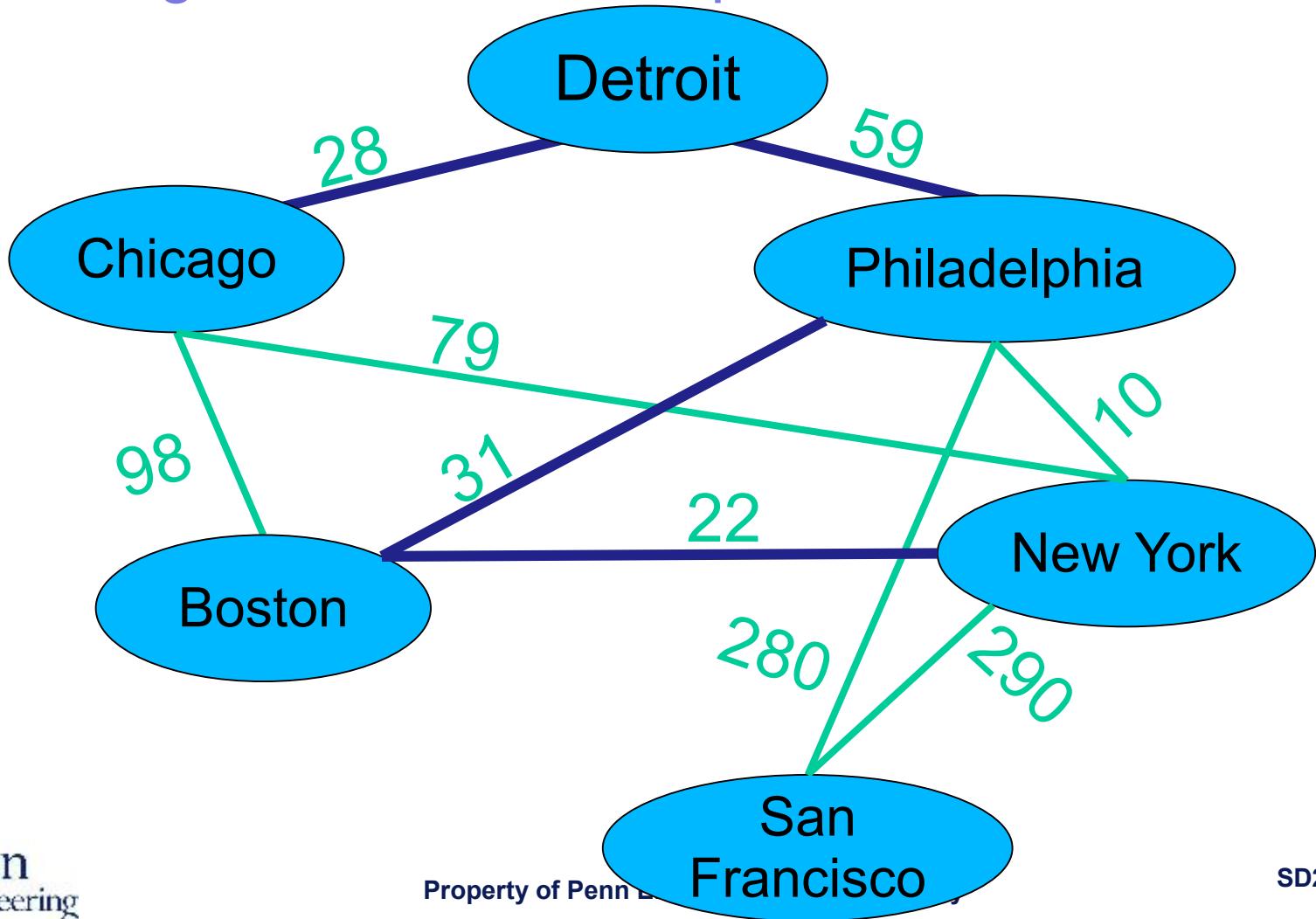
A **path** is a list of edges such that each node (but the last) is the predecessor of the next node in the list.



# General Graph Terminology

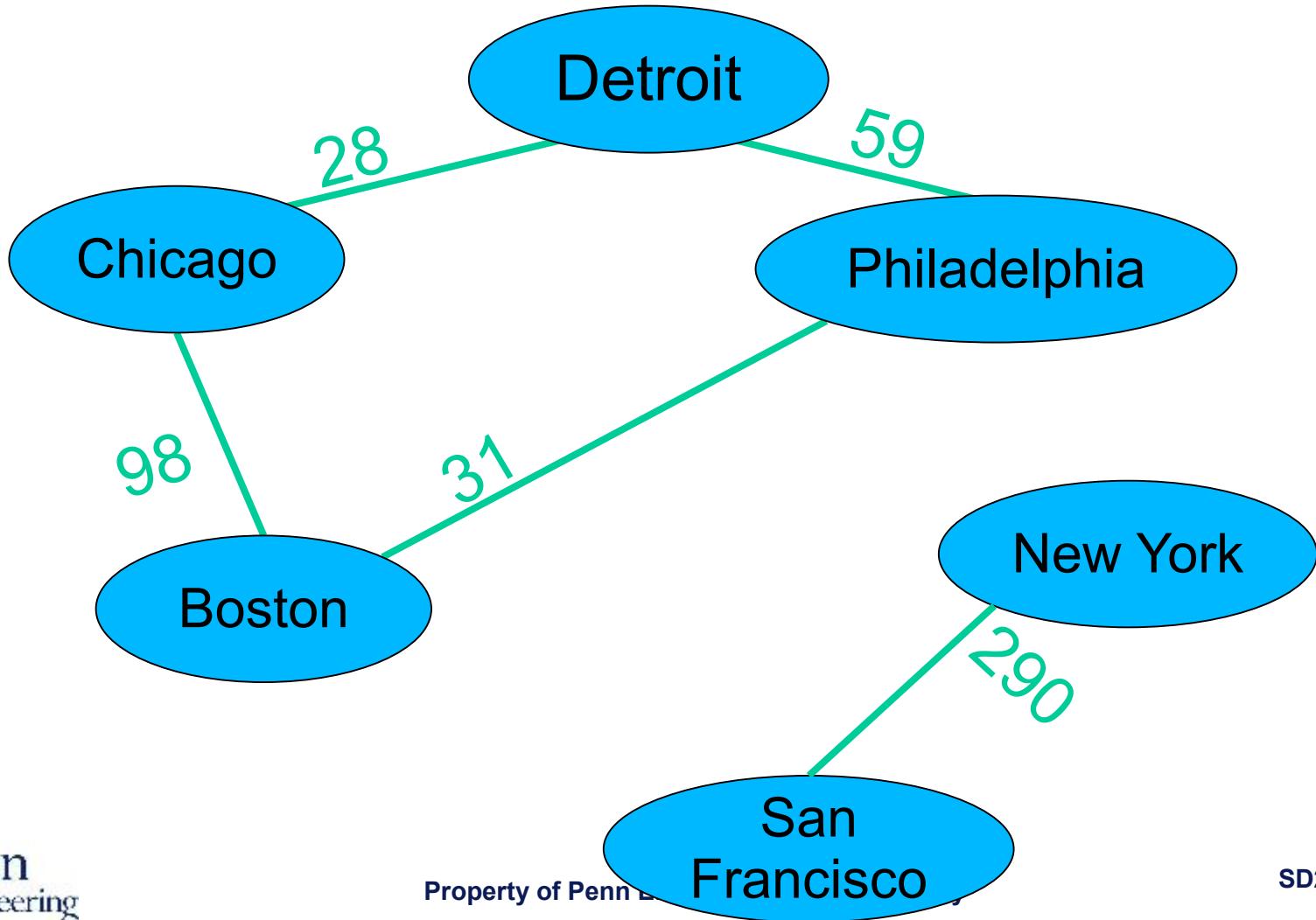
Example:

[Chicago, Detroit, Philadelphia, Boston, New York]



# General Graph Terminology

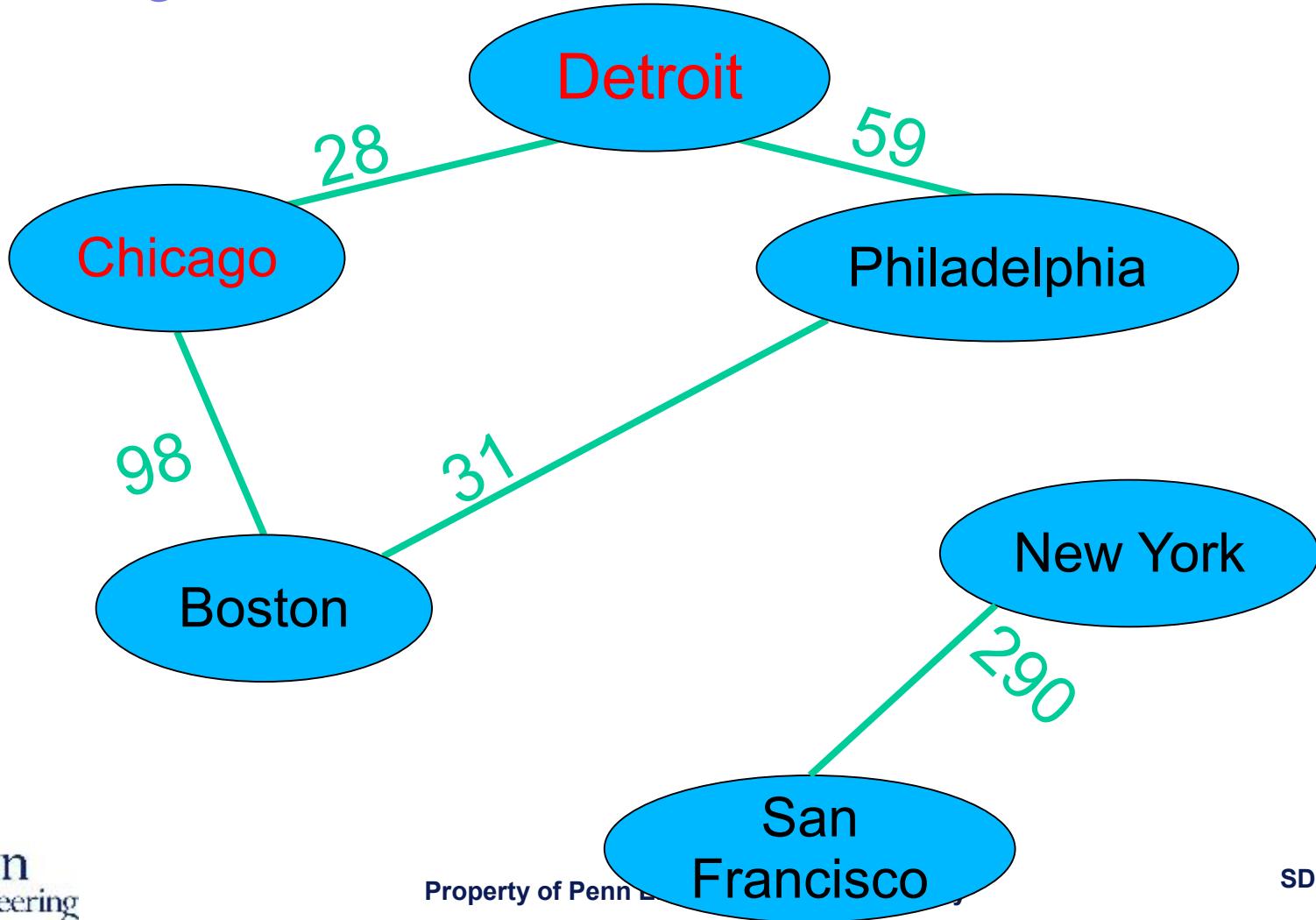
A node X is **reachable** from a node Y if there is a path from X to Y.



# General Graph Terminology

Example:

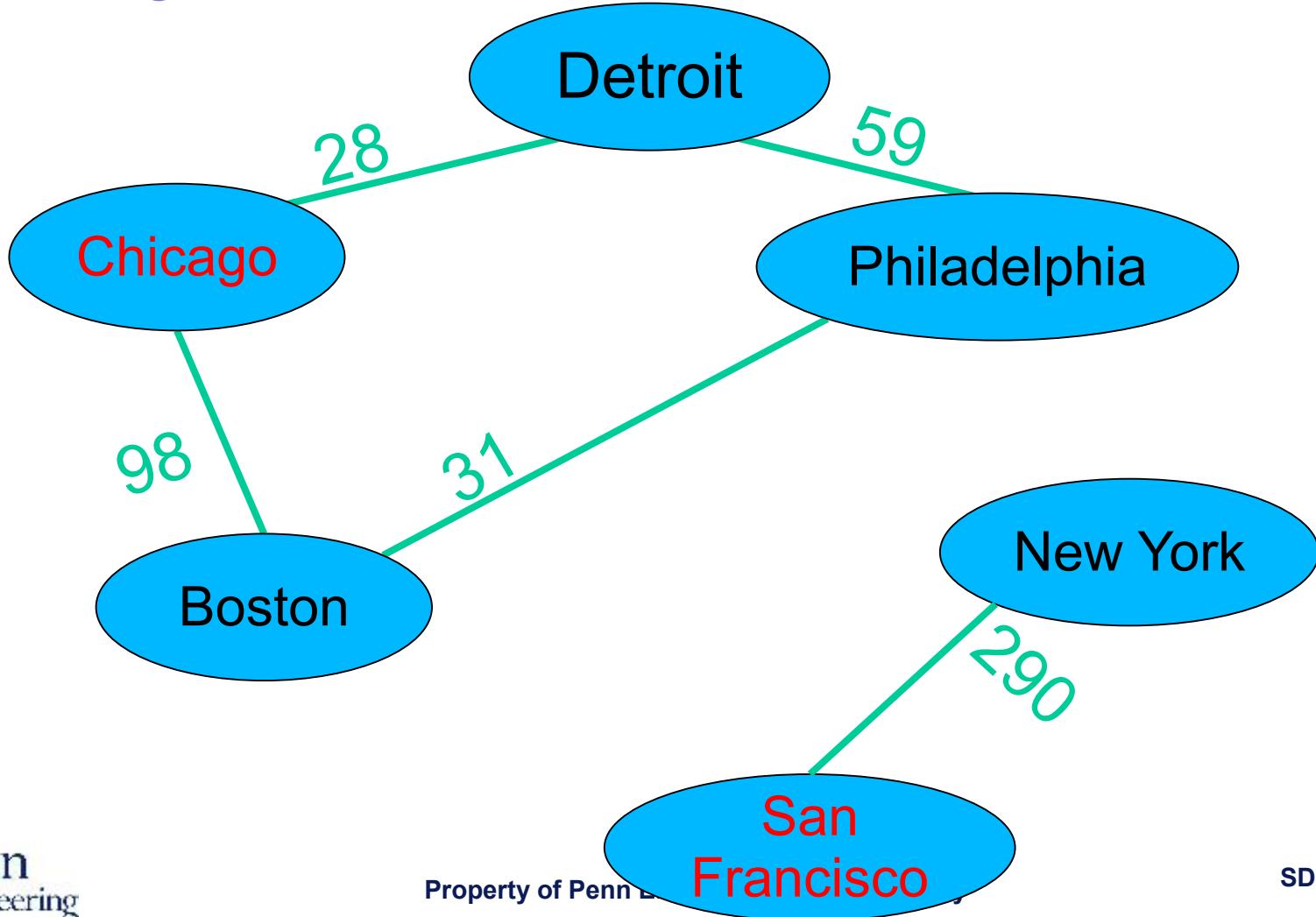
Chicago is reachable from Detroit.



# General Graph Terminology

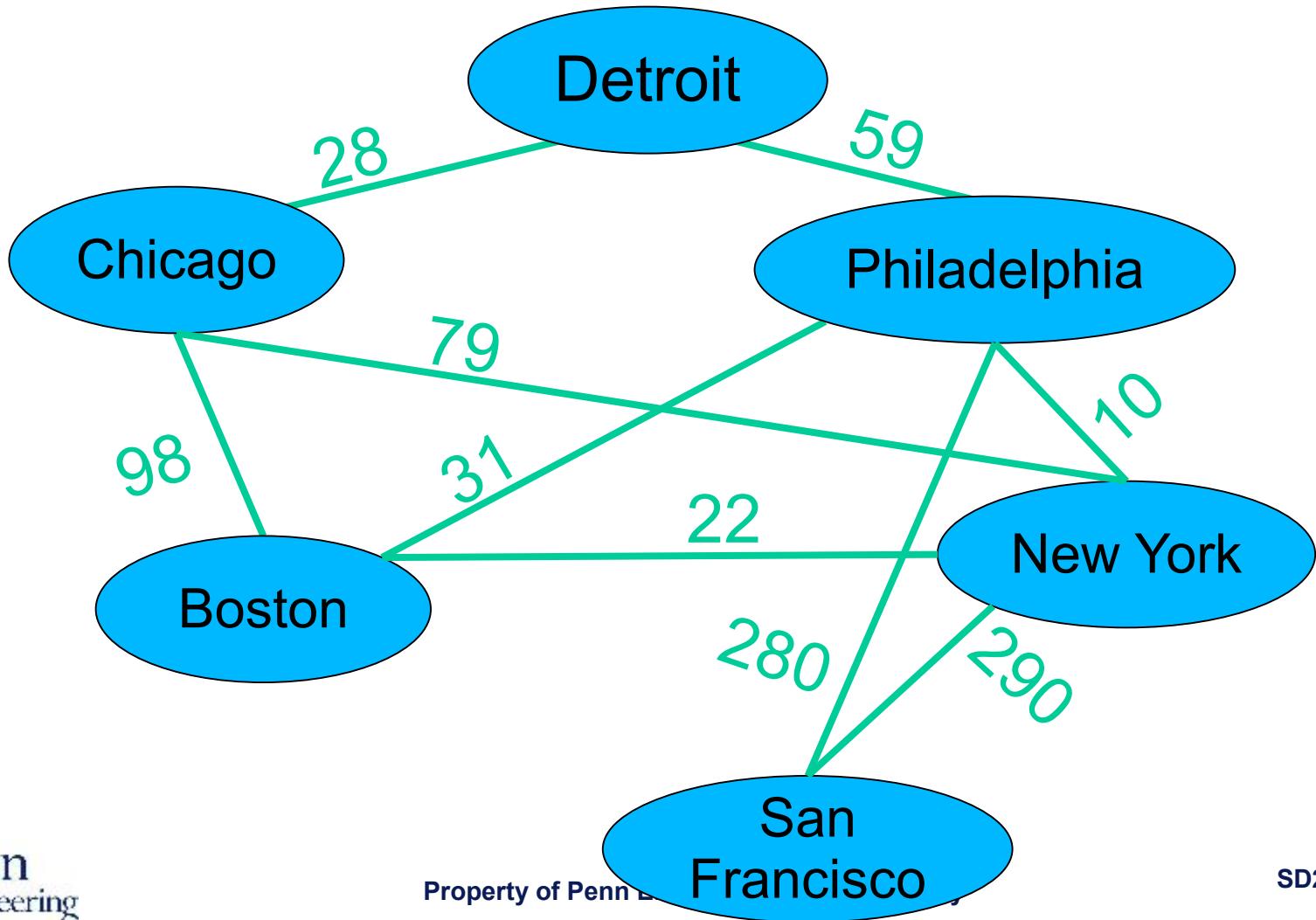
Example:

Chicago is *not* reachable from San Francisco



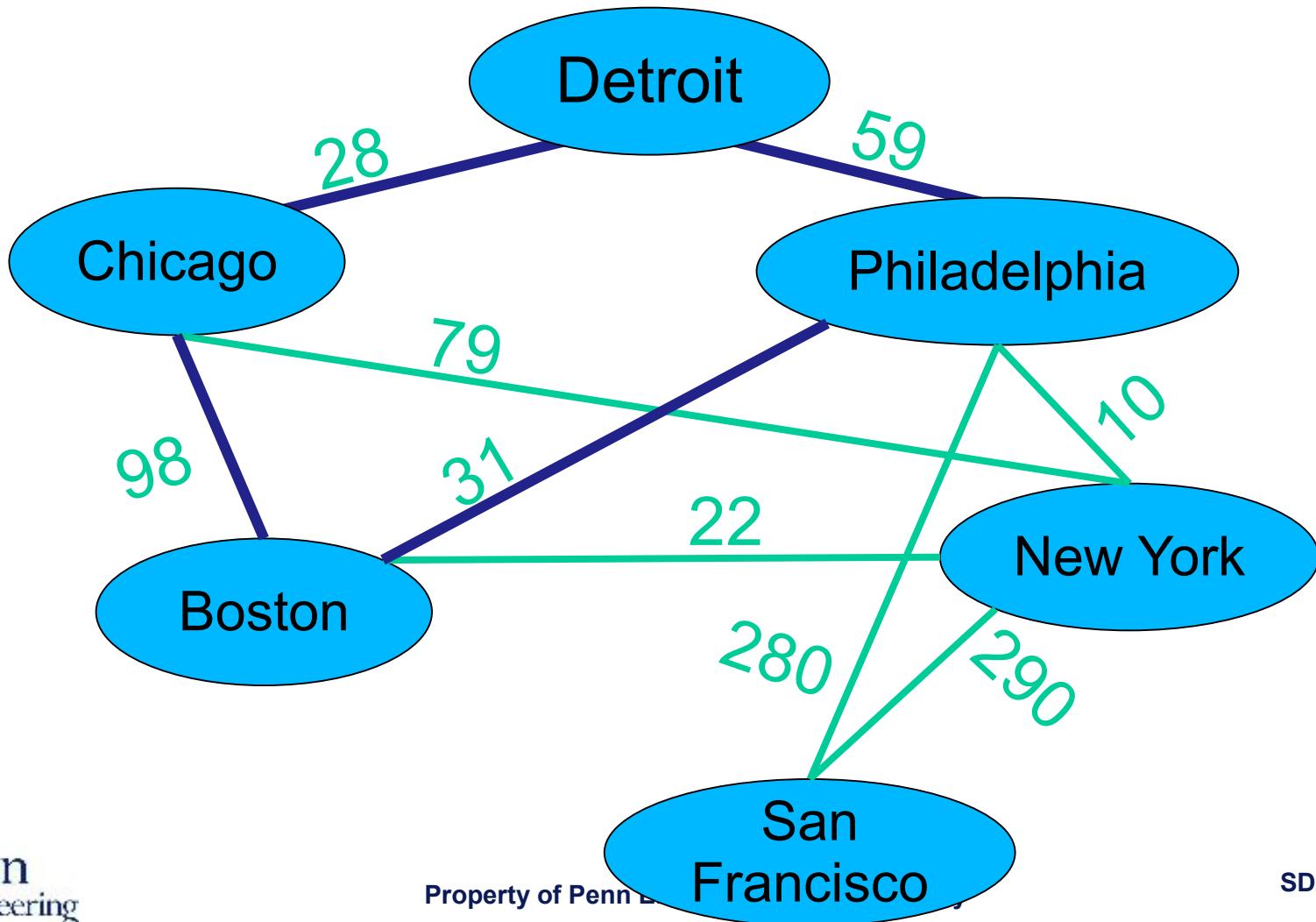
# General Graph Terminology

A **cycle** is a path whose first and last nodes are the same.



# General Graph Terminology

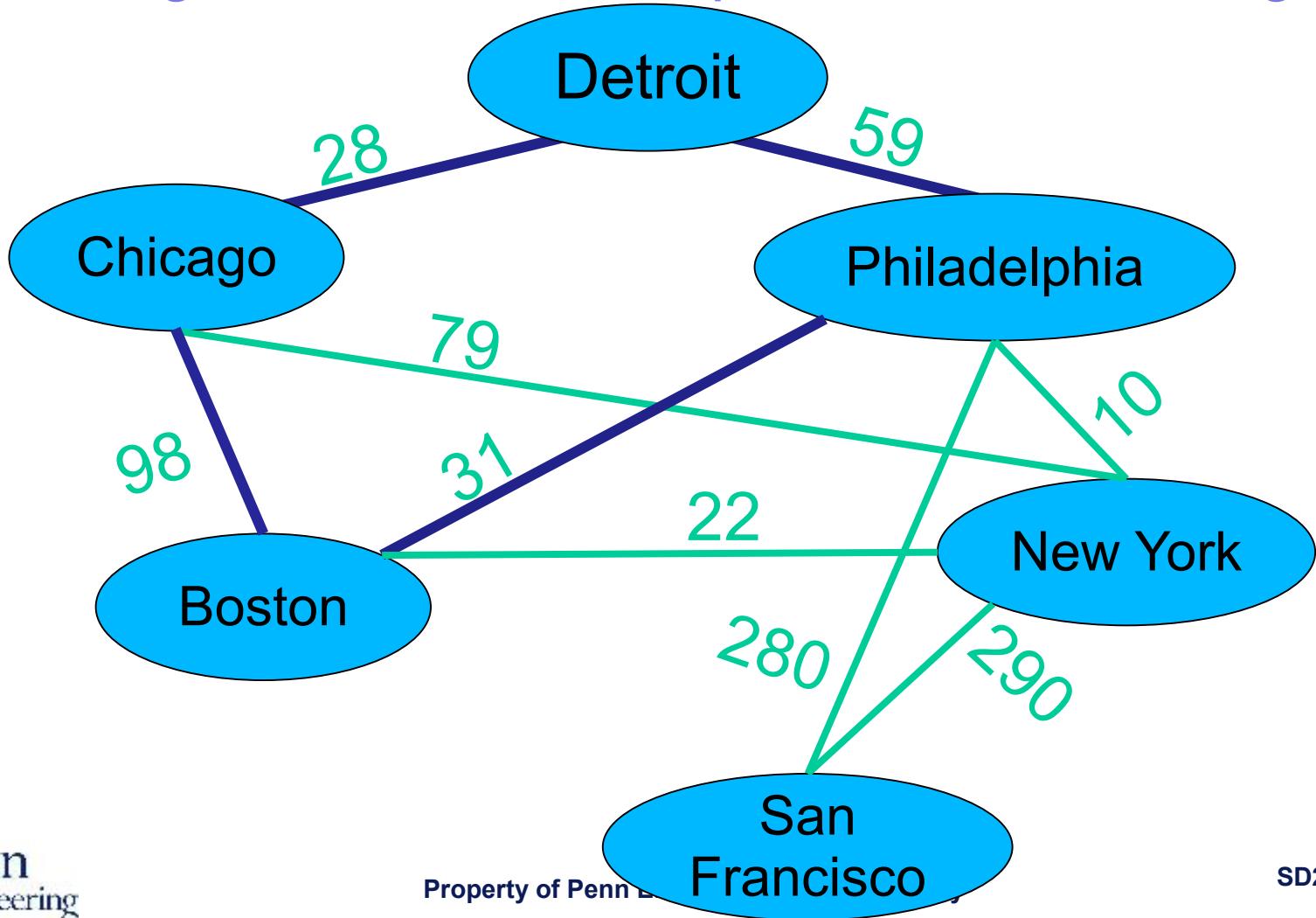
A **cycle** is a path whose first and last nodes are the same.



# General Graph Terminology

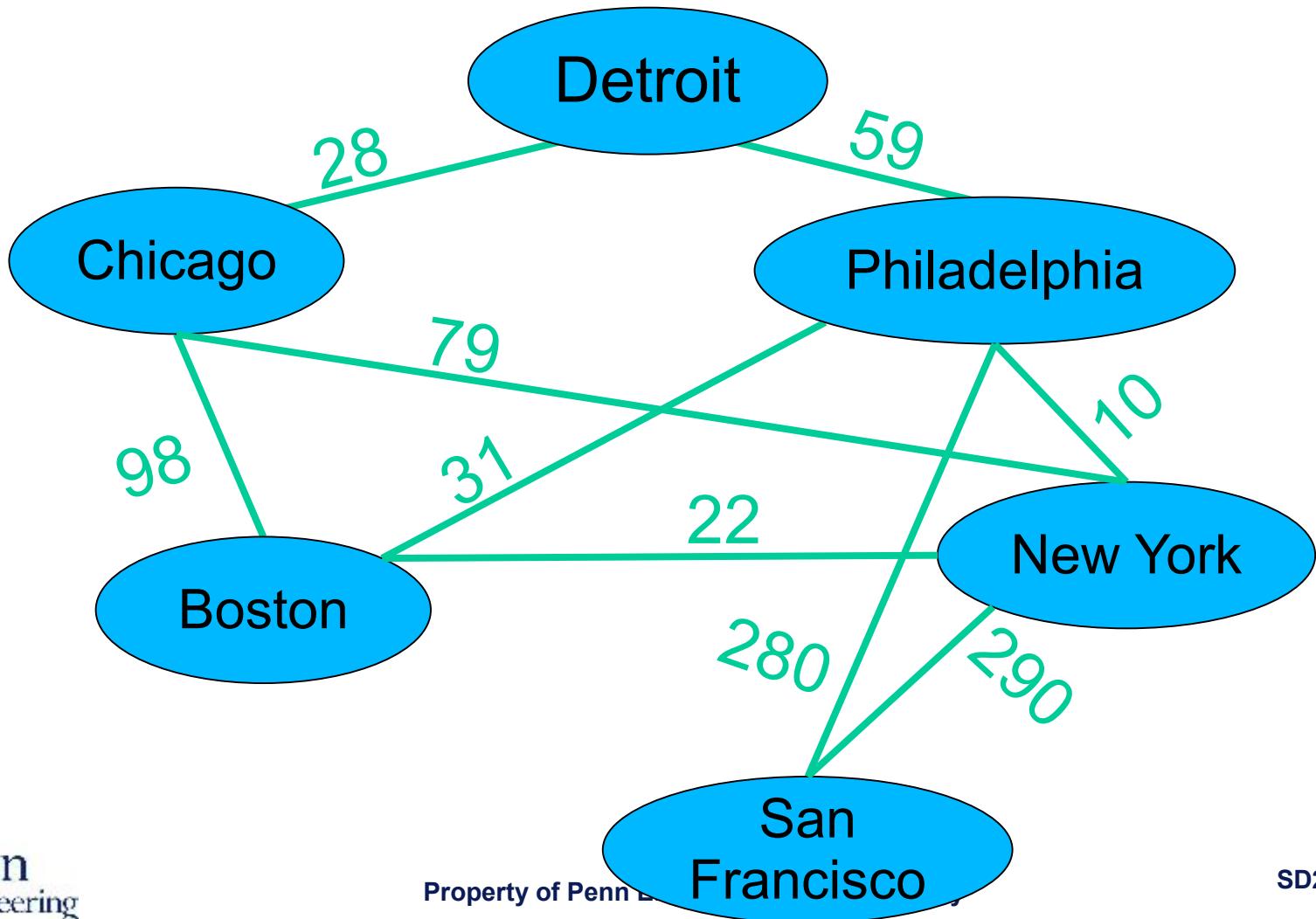
Example:

[Chicago, Detroit, Philadelphia, Boston, Chicago]



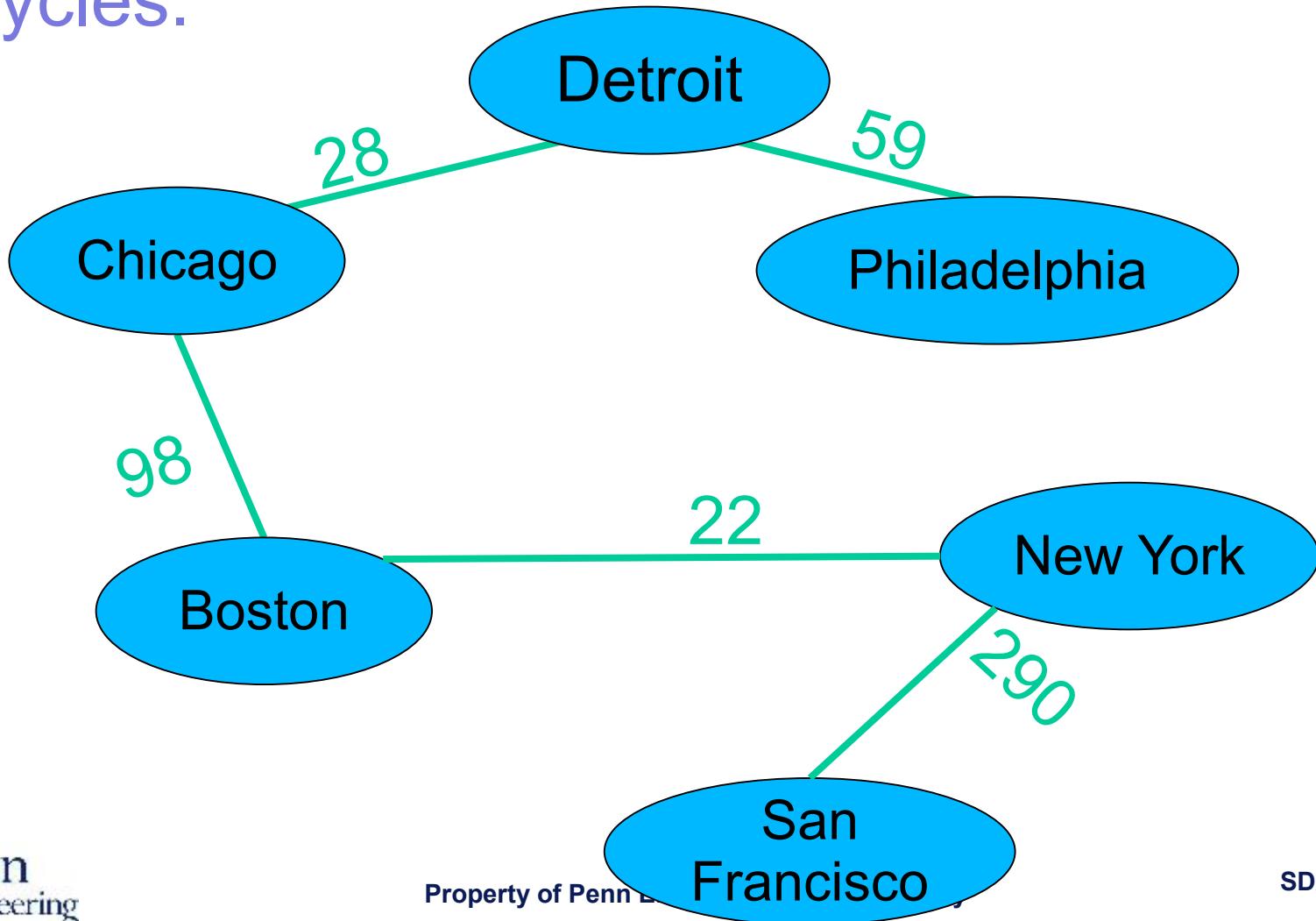
# General Graph Terminology

A **cyclic** graph contains at least one cycle.



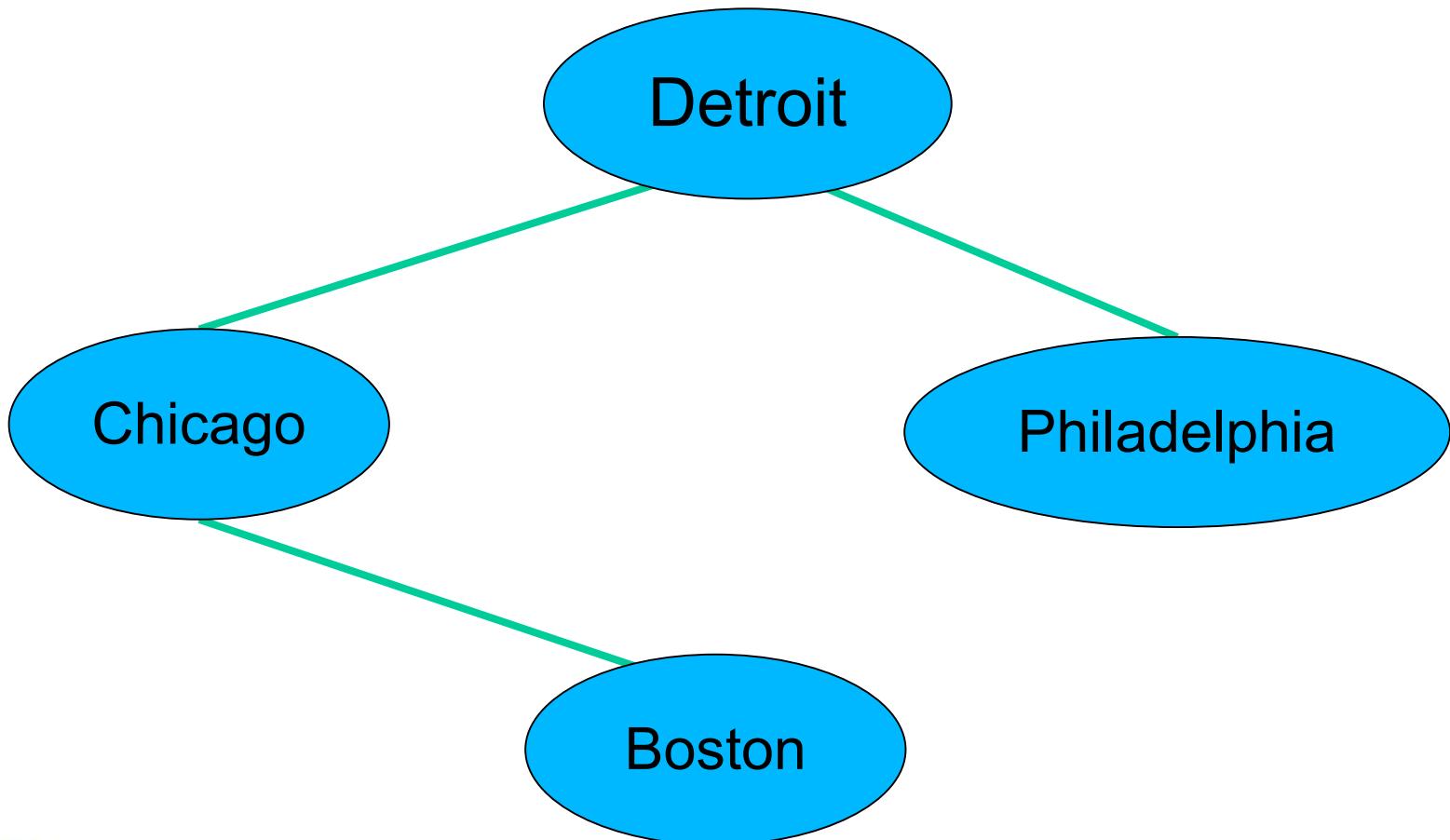
# General Graph Terminology

An **acyclic** graph does not contain any cycles.



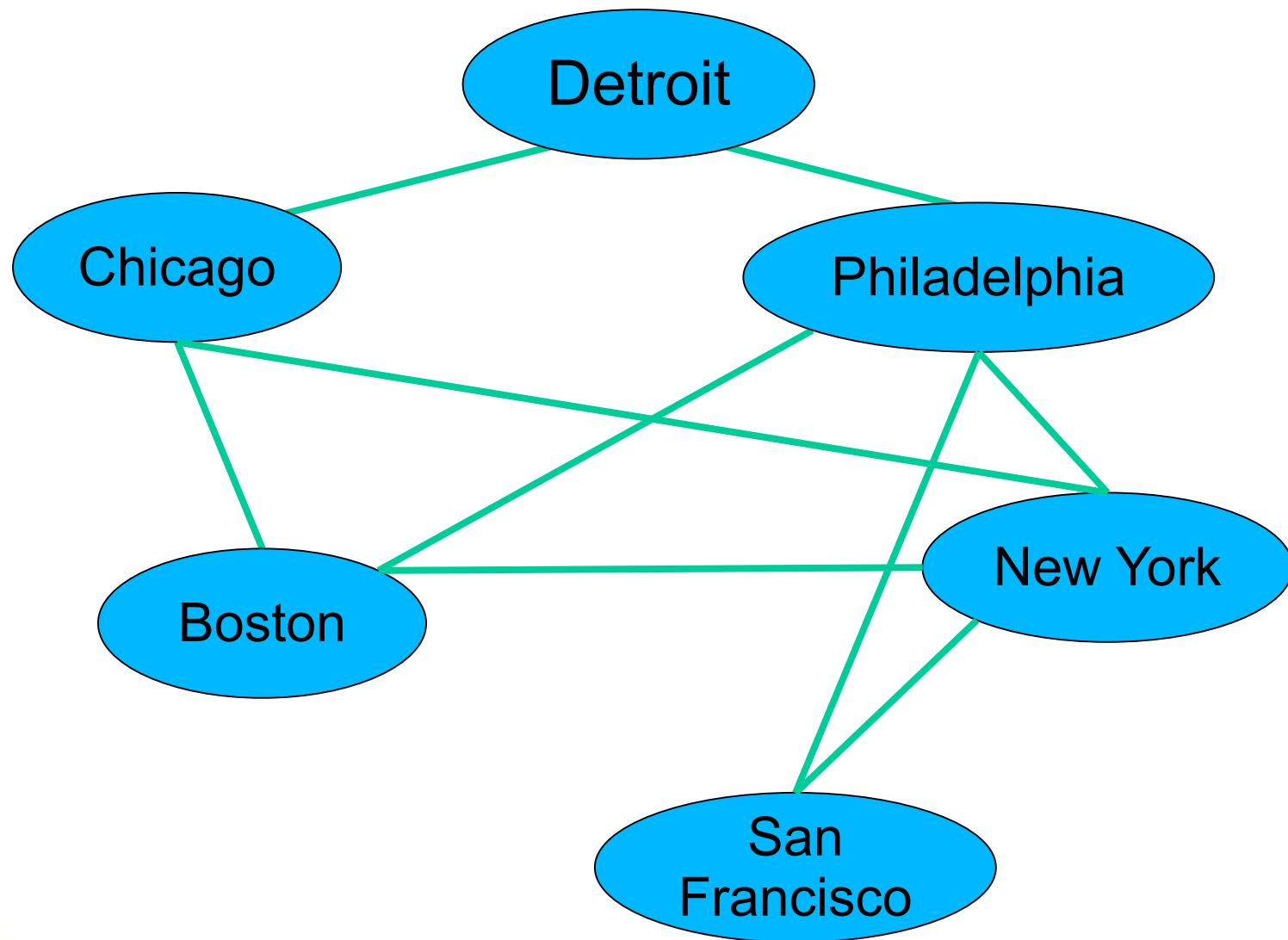
# General Graph Terminology

A graph is ***connected*** if there is a path from every node to every other node.



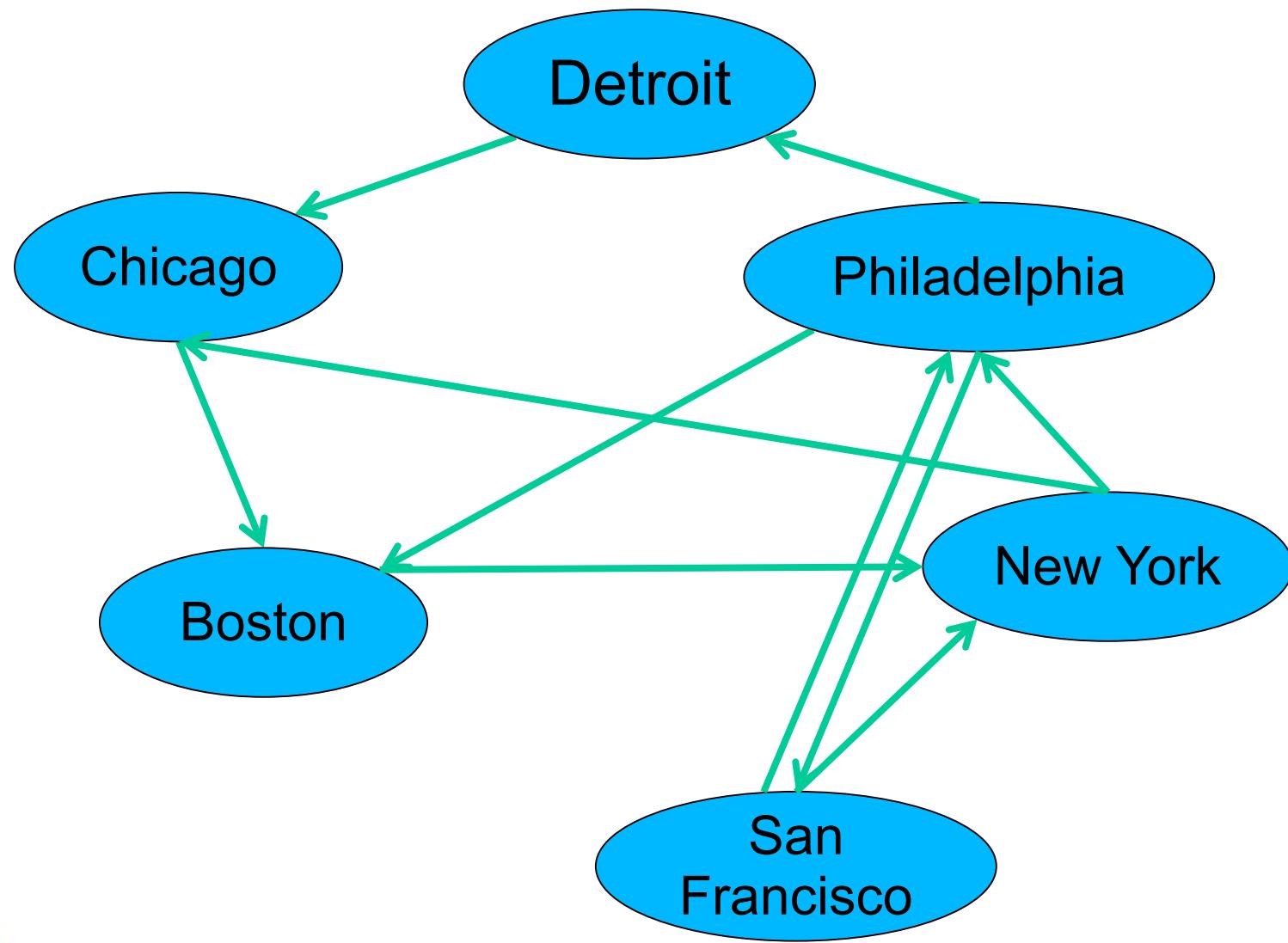
# Undirected Graph

---



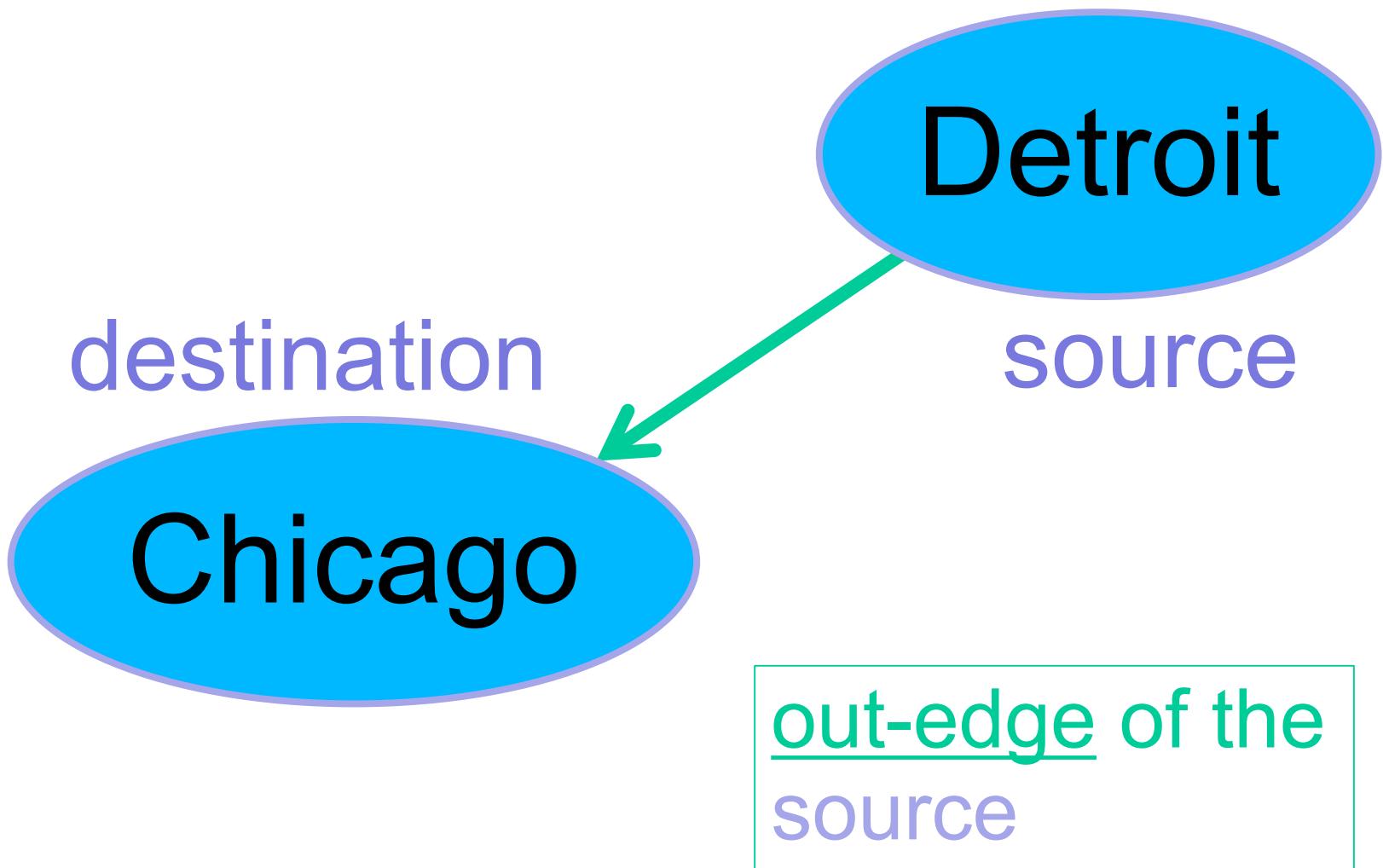
# Directed Graph

---



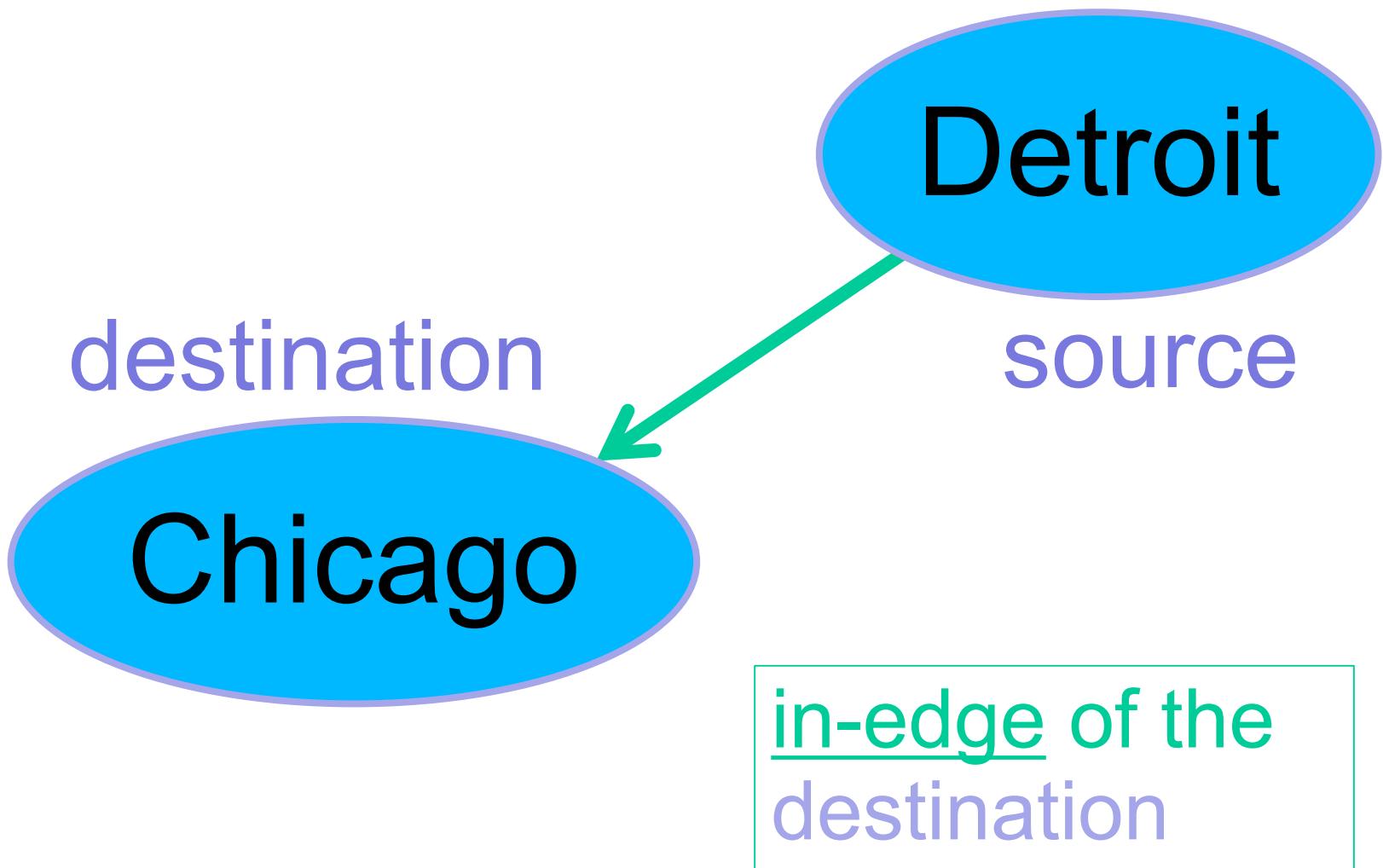
# Directed Graph Terminology

---



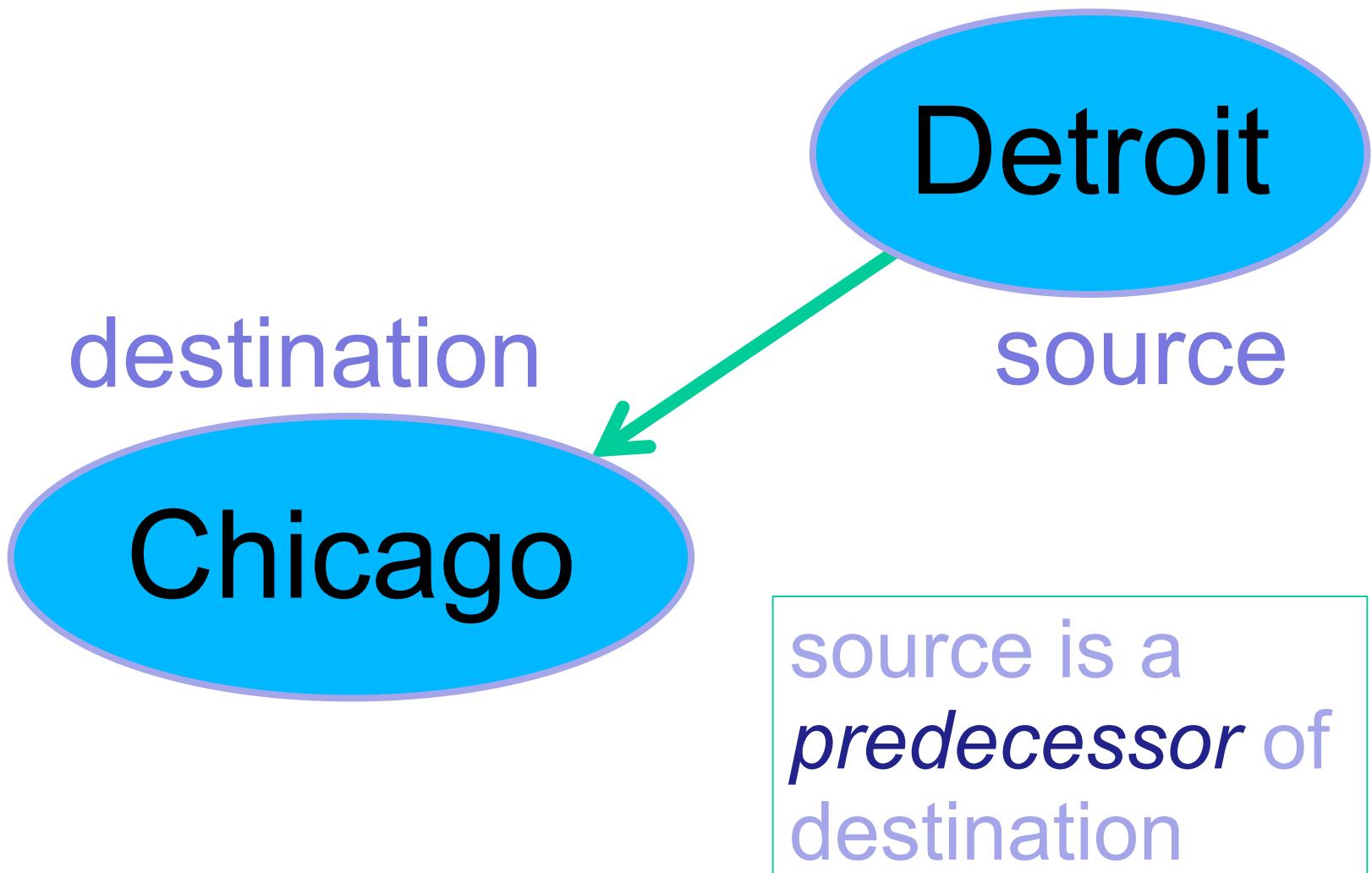
# Directed Graph Terminology

---



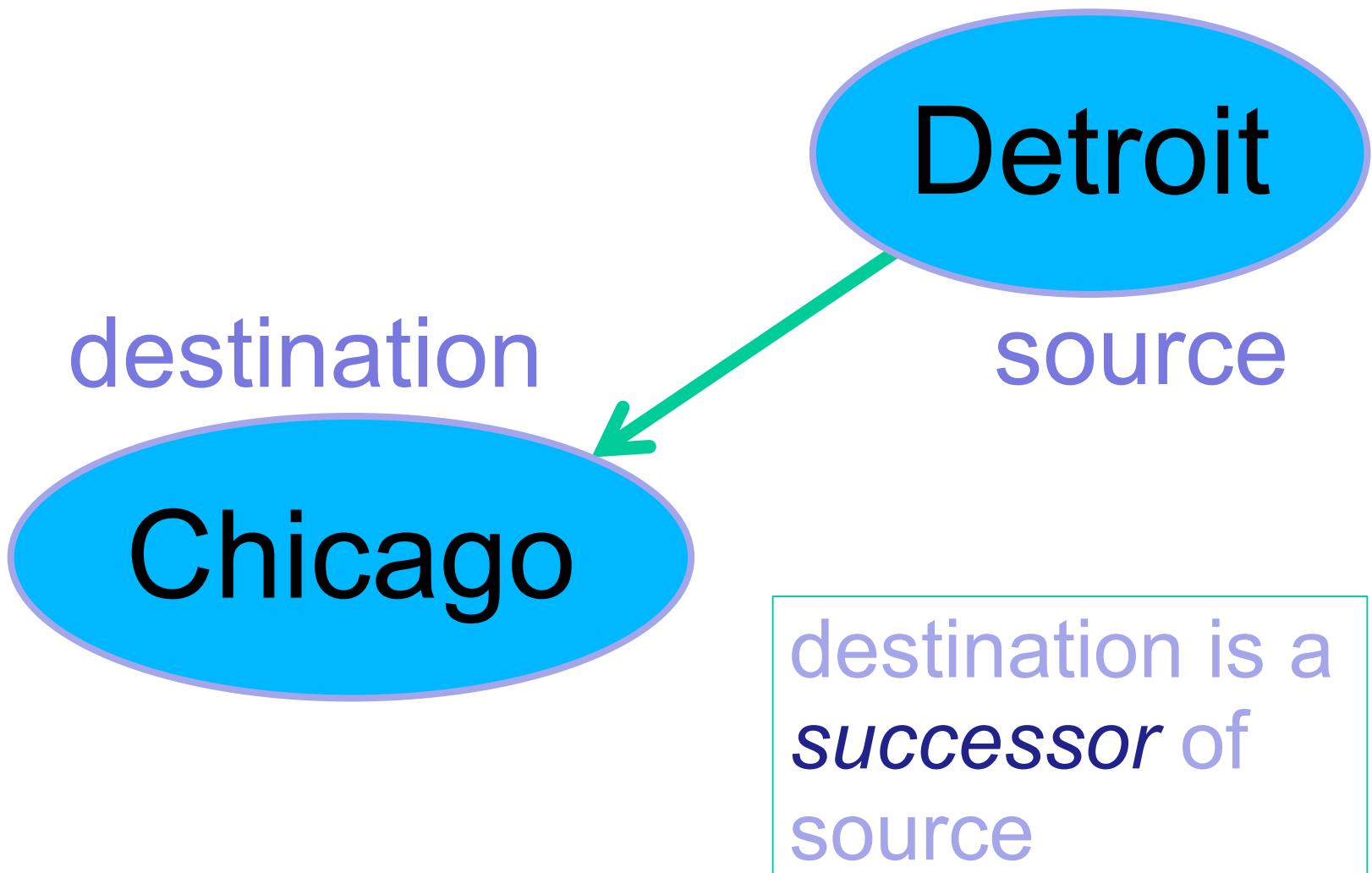
# Directed Graph Terminology

---



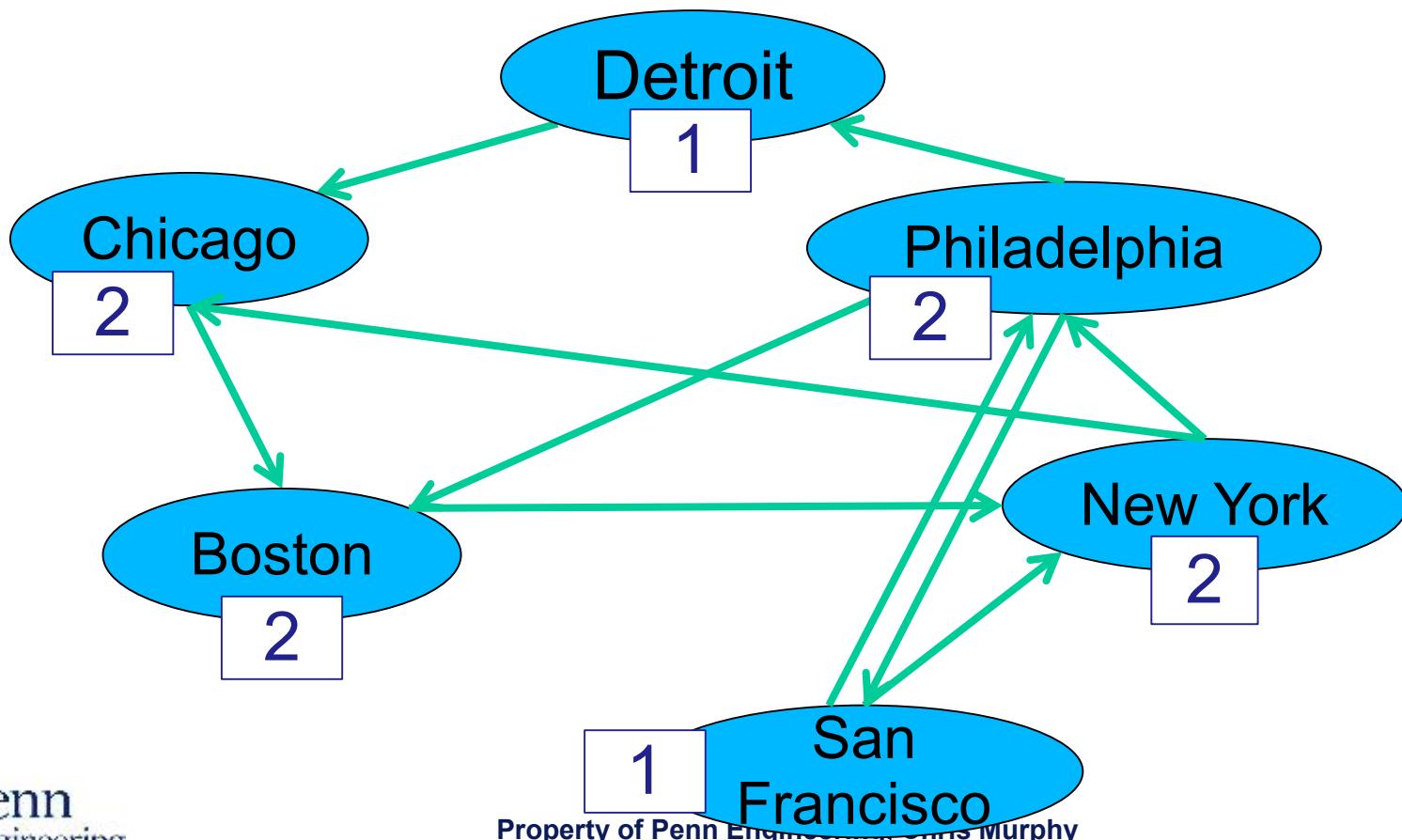
# Directed Graph Terminology

---



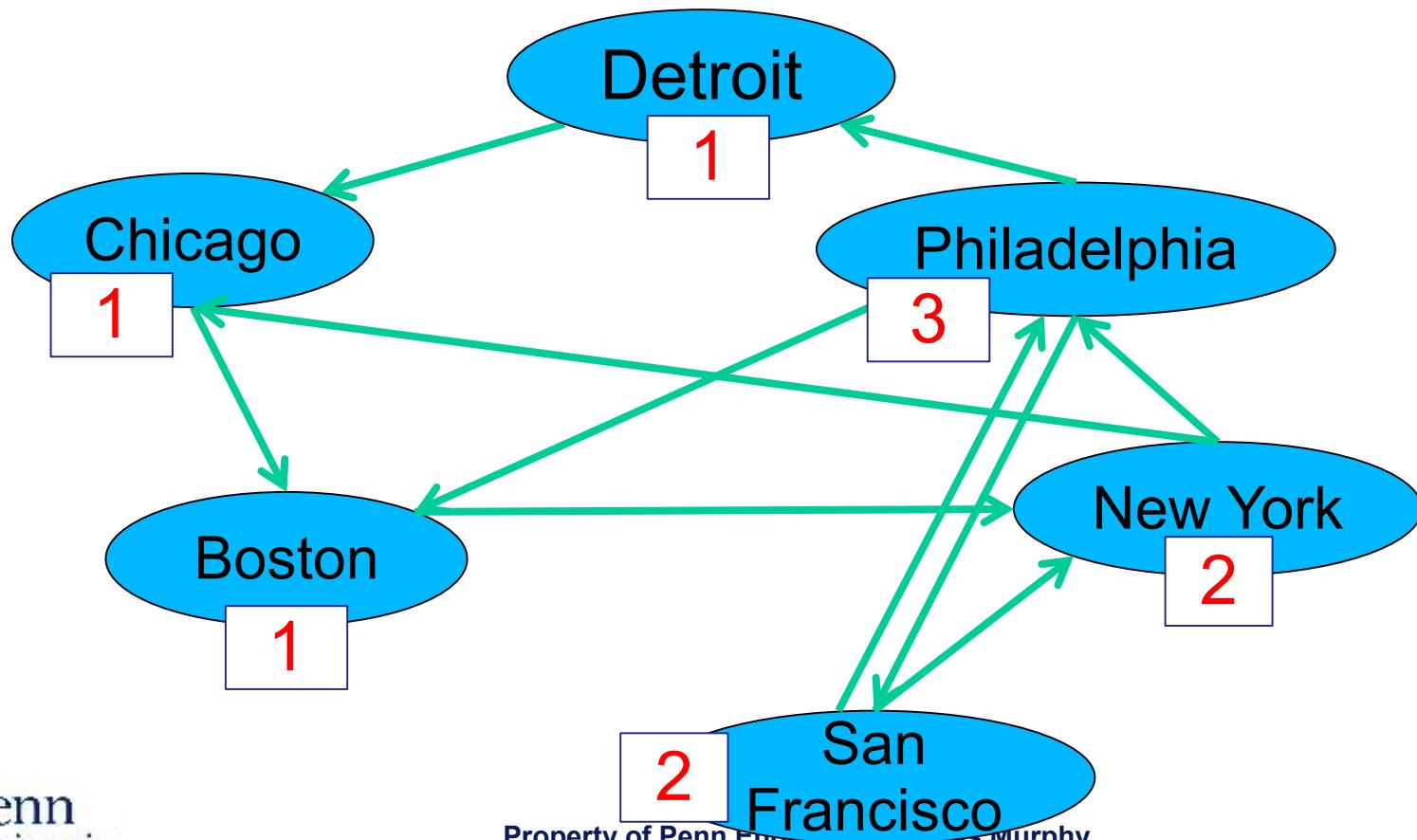
# Directed Graph Terminology

The ***in-degree*** of a node is the number of in-edges it has.



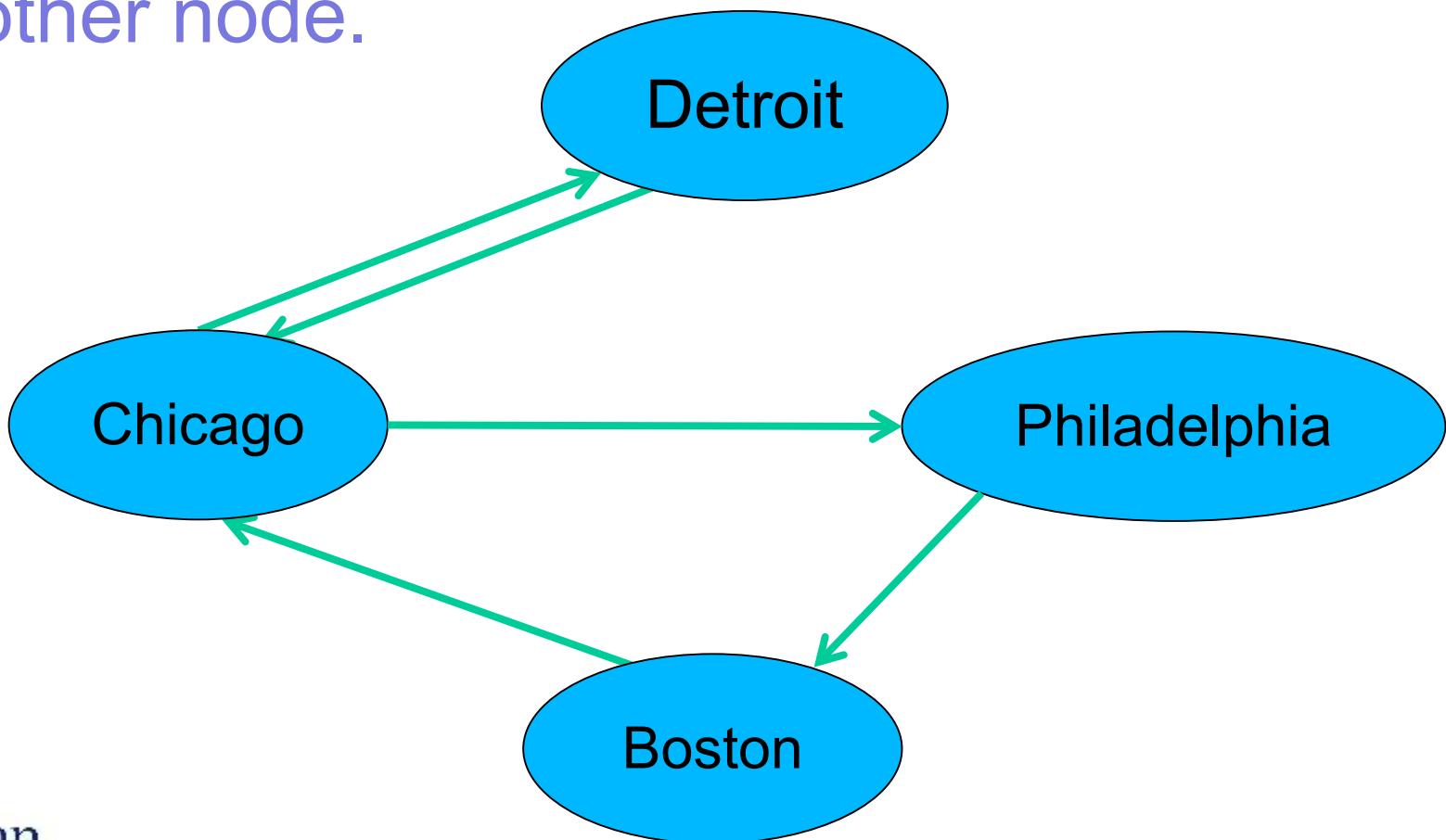
# Directed Graph Terminology

The **out-degree** of a node is the number of out-edges it has.



# Directed Graph Terminology

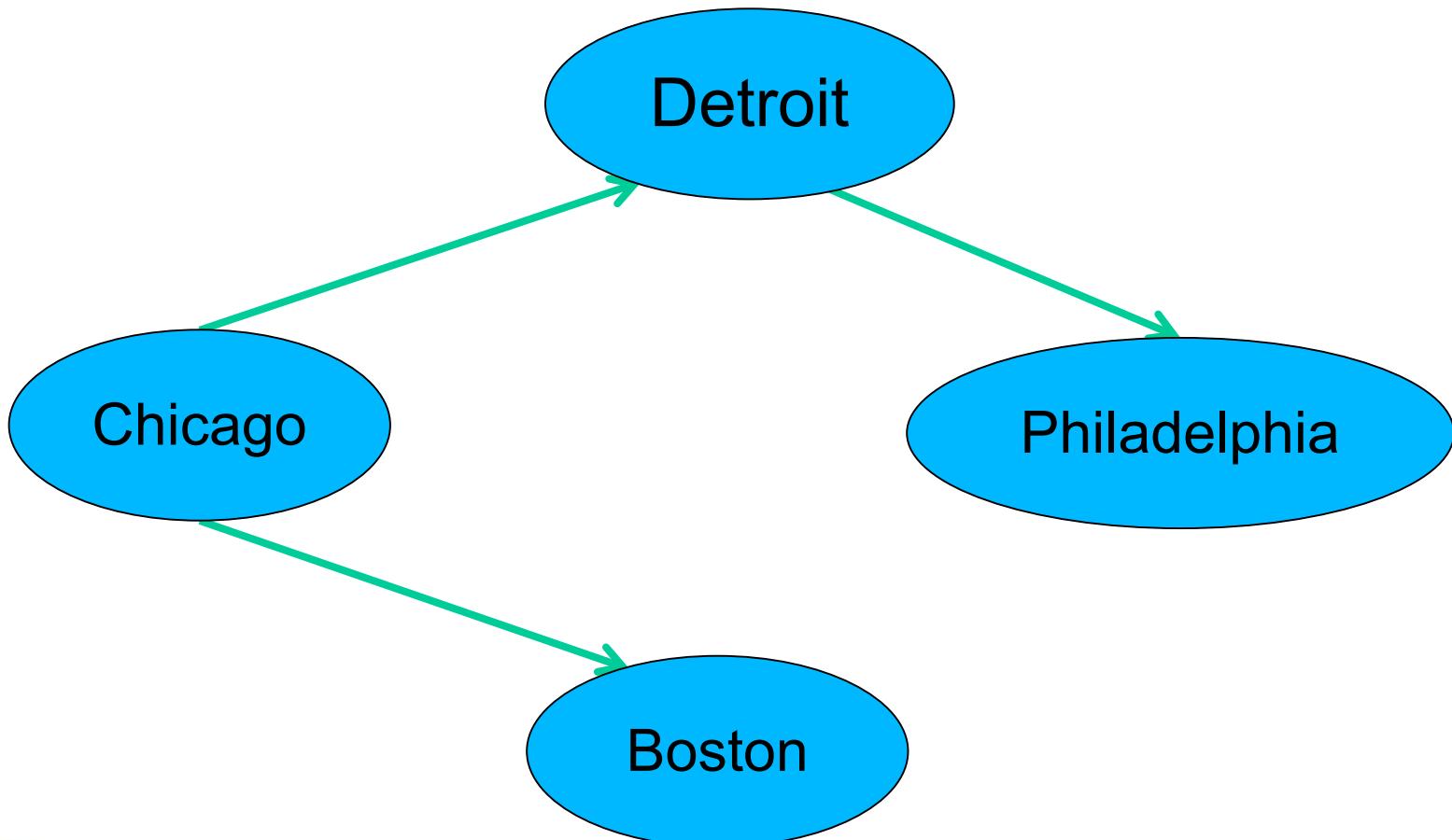
A directed graph is ***strongly connected*** if there is a path from every node to every other node.



# Directed Graph Terminology

---

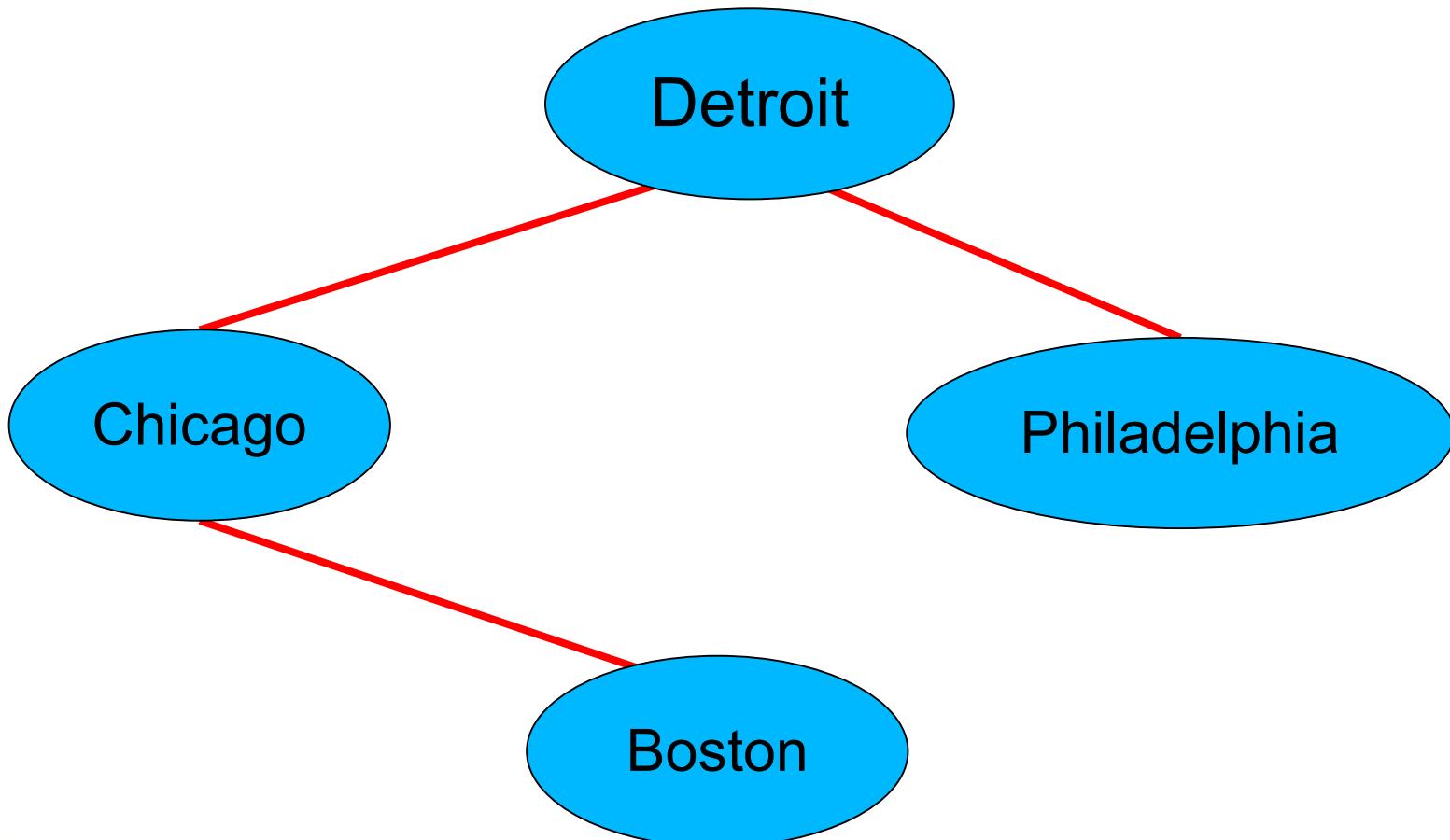
A directed graph is **weakly connected** if the underlying undirected graph is connected.



# Directed Graph Terminology

---

A directed graph is ***weakly connected*** if the underlying undirected graph is connected.



# Recap: Graphs

---

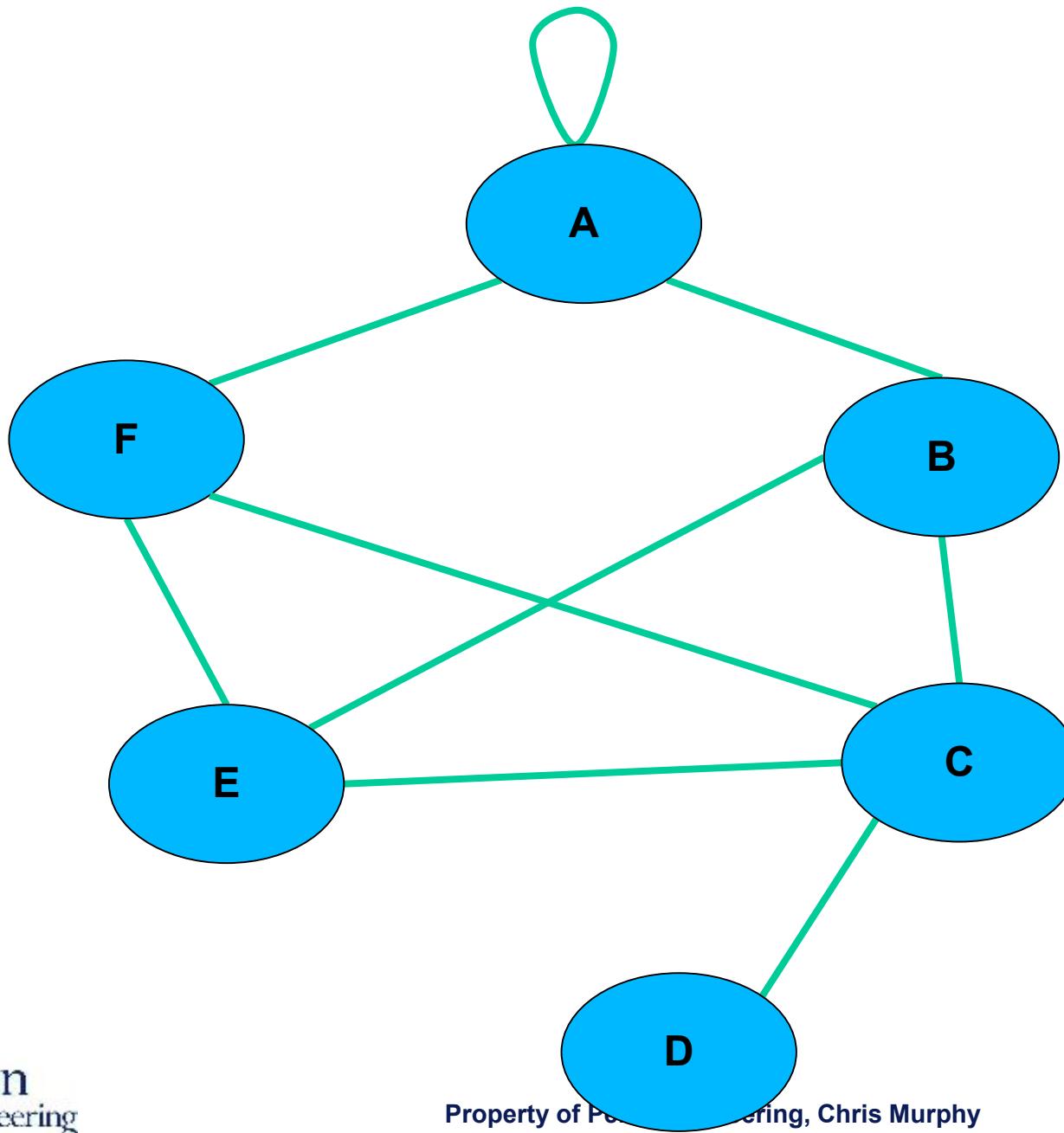
- A collection of **nodes** that hold values
- Nodes are connected by **edges** to represent relationships
- Edges can be **directed** and/or **weighted**

# **SD2x2.10**

# **Graph representations**

# **Chris**

# How can we represent a graph?



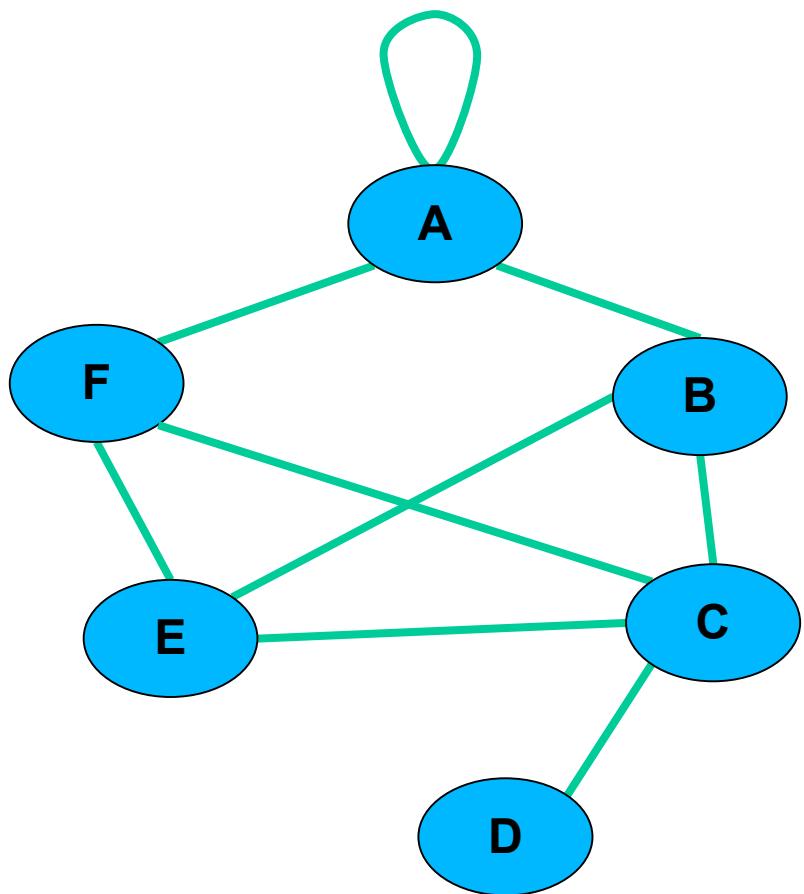
# Graph Representation

---

- **Adjacency Matrix**: 2-dimensional array in which values represent weights
- **Edge Set**: set of all edges
- **Adjacency Set**: mapping of nodes to sets of edges

# Adjacency Matrix Representation

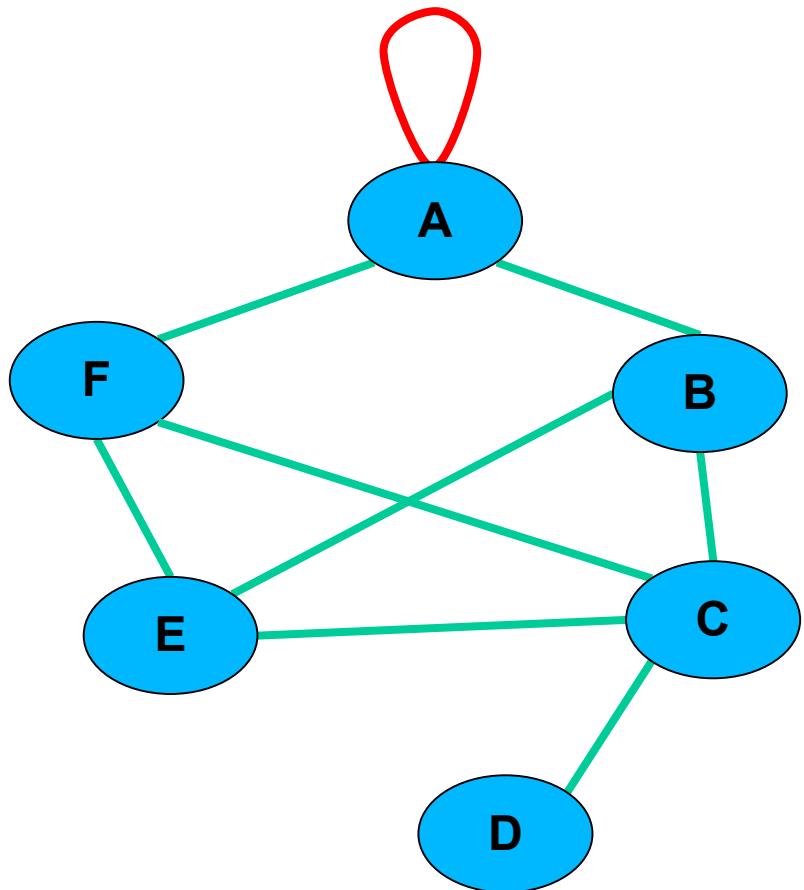
---



	A	B	C	D	E	F
A	1	1	0	0	0	1
B	1	0	1	0	1	0
C	0	1	0	1	1	1
D	0	0	1	0	0	0
E	0	1	1	0	0	1
F	1	0	1	0	1	0

# Adjacency Matrix Representation

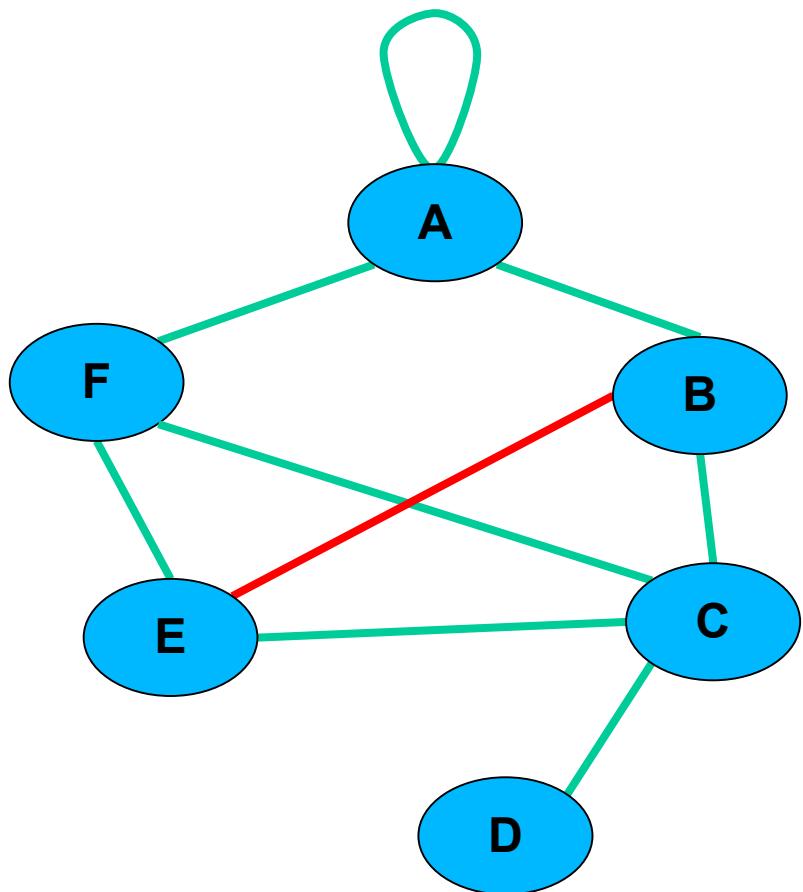
---



	A	B	C	D	E	F
A	1	1	0	0	0	1
B	1	0	1	0	1	0
C	0	1	0	1	1	1
D	0	0	1	0	0	0
E	0	1	1	0	0	1
F	1	0	1	0	1	0

# Adjacency Matrix Representation

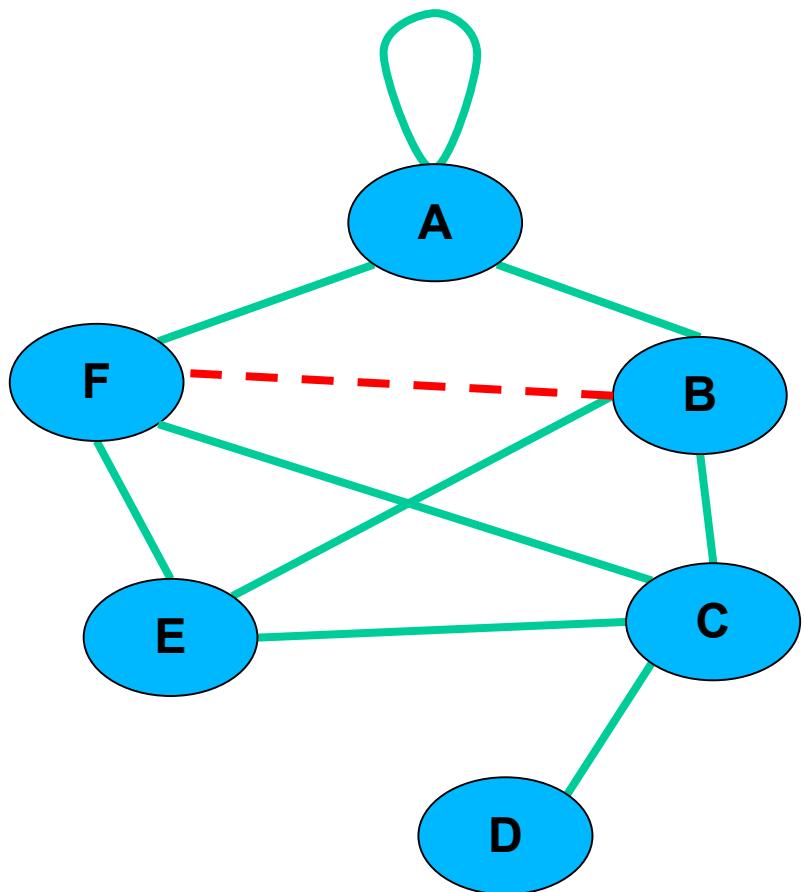
---



	A	B	C	D	E	F
A	1	1	0	0	0	1
B	1	0	1	0	1	0
C	0	1	0	1	1	1
D	0	0	1	0	0	0
E	0	1	1	0	0	1
F	1	0	1	0	1	0

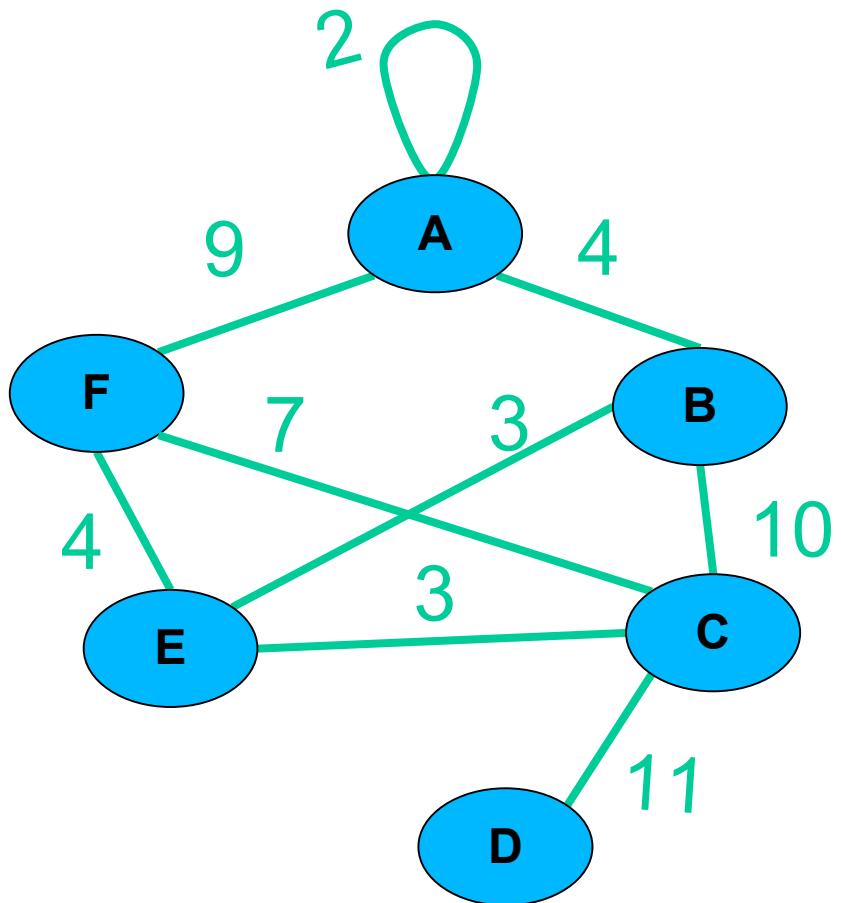
# Adjacency Matrix Representation

---



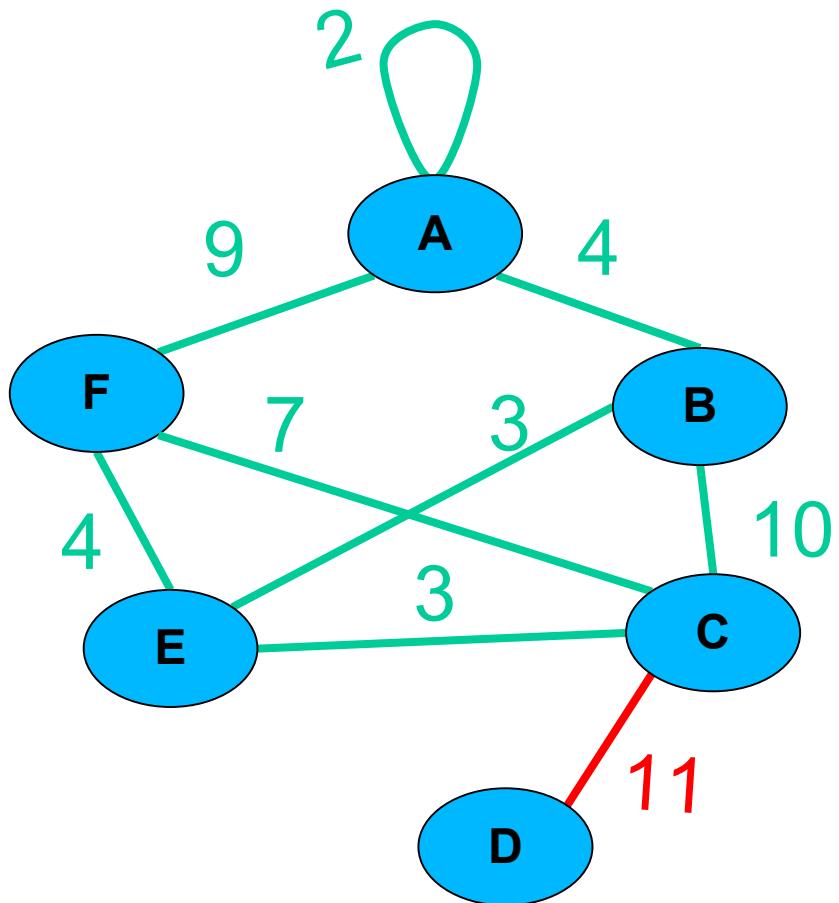
	A	B	C	D	E	F
A	1	1	0	0	0	1
B	1	0	1	0	1	0
C	0	1	0	1	1	1
D	0	0	1	0	0	0
E	0	1	1	0	0	1
F	1	0	1	0	1	0

# Adjacency Matrix Representation



	A	B	C	D	E	F
A	2	4	0	0	0	9
B	4	0	10	0	3	0
C	0	10	0	11	3	7
D	0	0	11	0	0	0
E	0	3	3	0	0	4
F	9	0	7	0	4	0

# Adjacency Matrix Representation

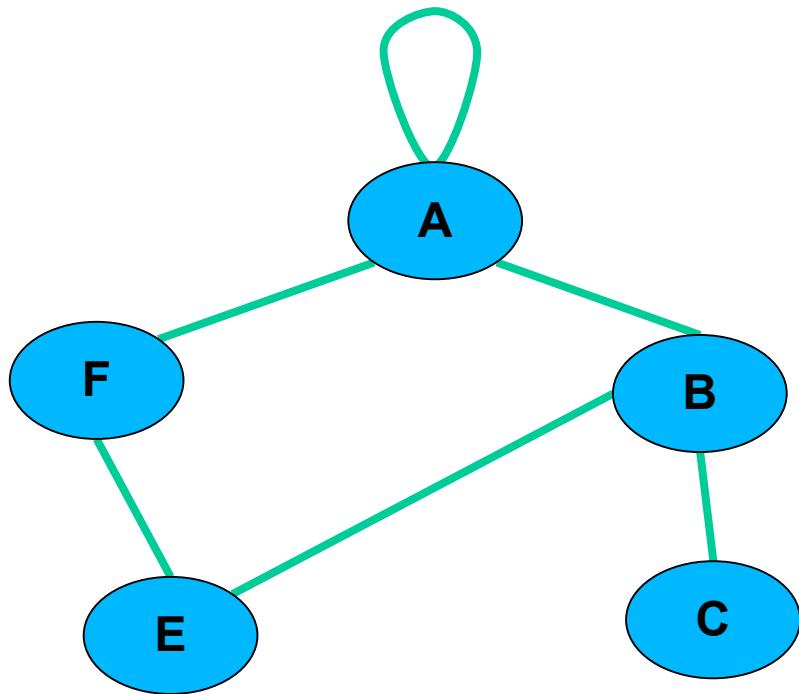


	A	B	C	D	E	F
A	2	4	0	0	0	9
B	4	0	10	0	3	0
C	0	10	0	11	3	7
D	0	0	11	0	0	0
E	0	3	3	0	0	4
F	9	0	7	0	4	0

# Edge set

# Edge Set Representation

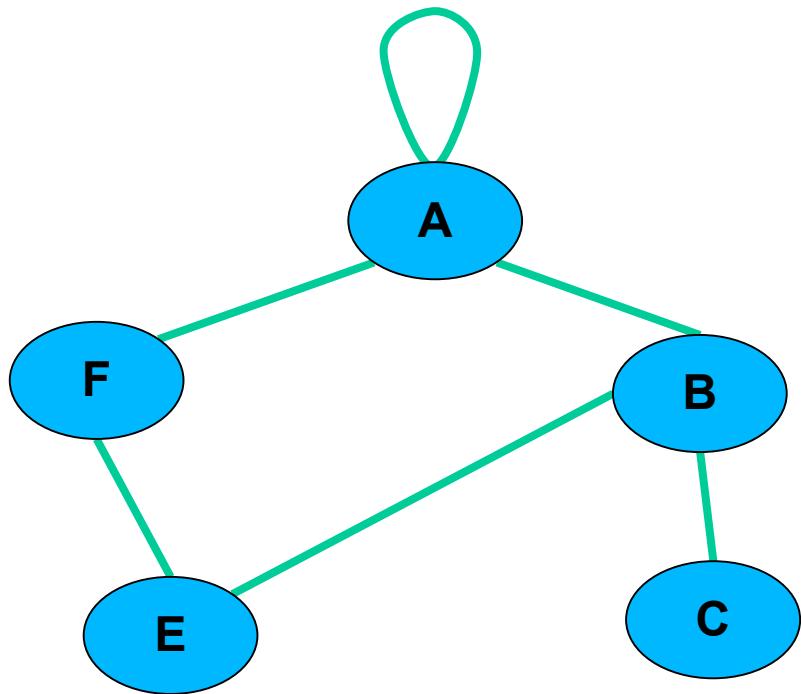
---



The graph is stored  
as a set of edges.

# Edge Set Representation

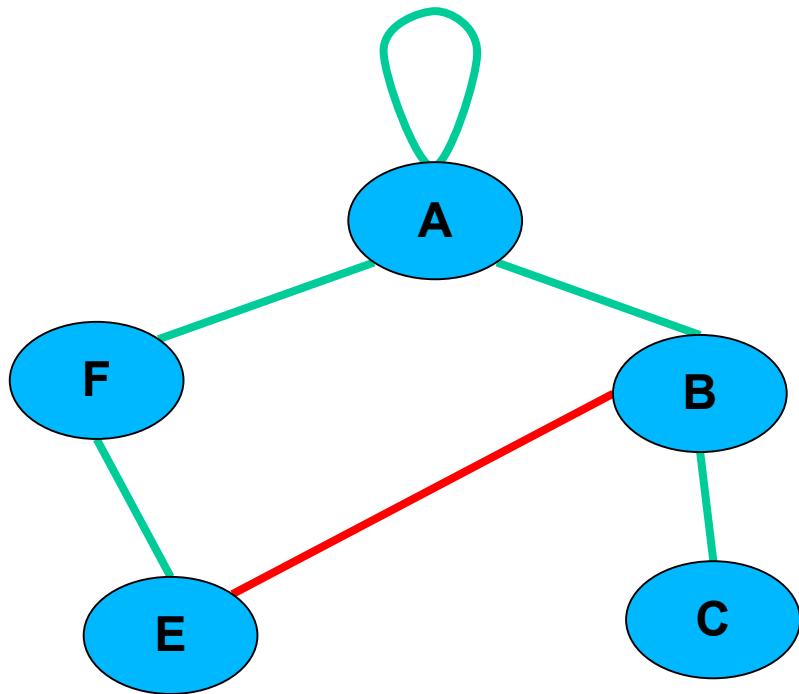
---



edges = {  
  (A, A), (A, B), (B, C),  
  (B, E), (A, F), (F, E)}  
Each Edge object  
contains references  
to the two Nodes

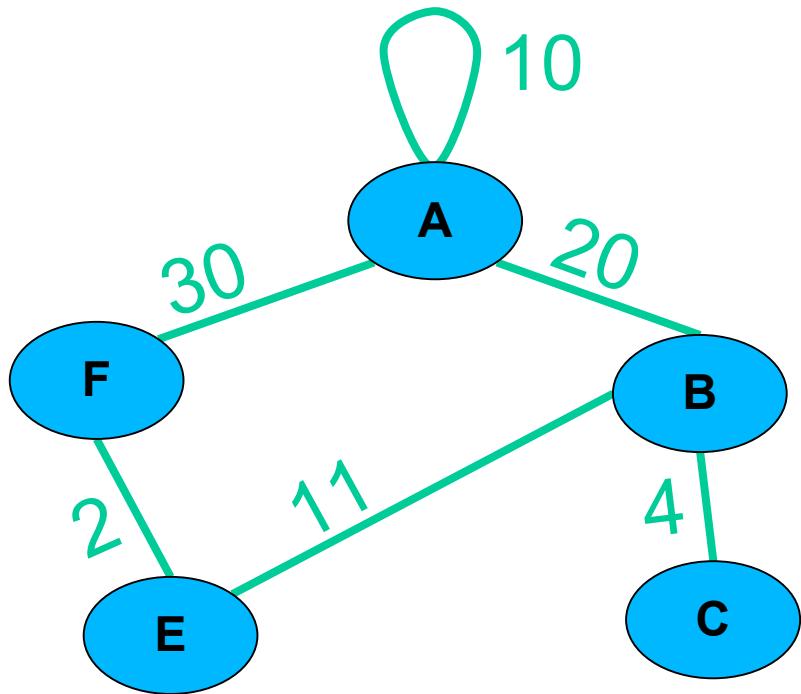
# Edge Set Representation

---



edges = {  
  (A, A), (A, B), (B, C),  
  (B, E), (A, F), (F, E)}  
Each Edge object  
contains references  
to the two Nodes

# Edge Set Representation (Weighted)

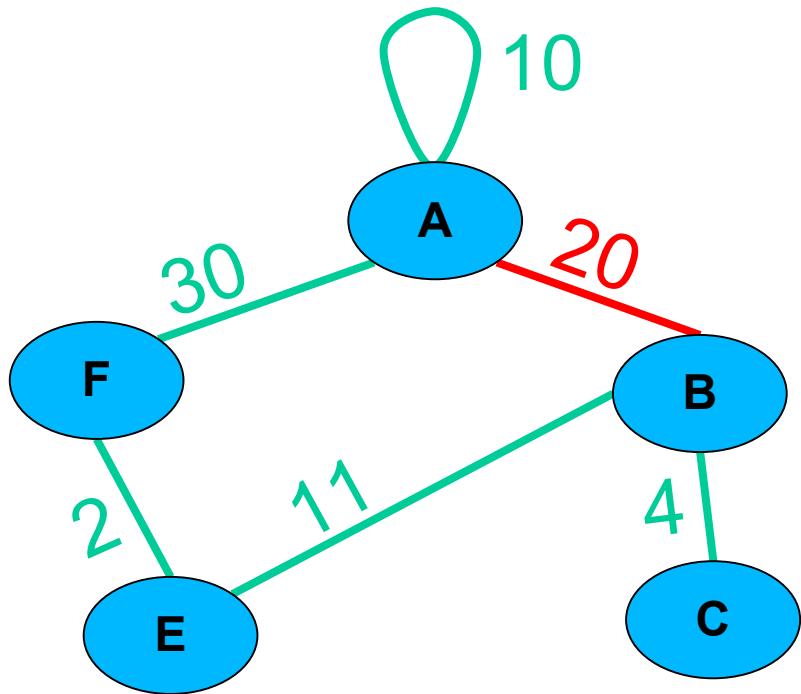


edges =

Each Edge object contains references to the two Nodes and its attribute (if any)

```
{(A, A, 10), (A, B, 20),  
(B, C, 4), (B, E, 11),  
(A, F, 30), (F, E, 2)}
```

# Edge Set Representation (Weighted)

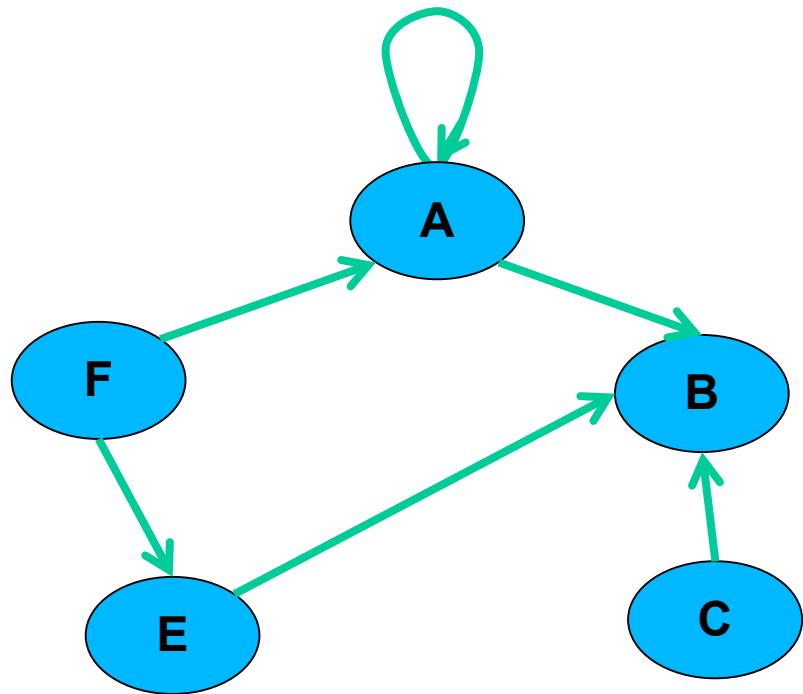


edges =

Each Edge object contains references to the two Nodes and its attribute (if any)

```
{(A, A, 10), (A, B, 20),  
(B, C, 4), (B, E, 11),  
(A, F, 30), (F, E, 2)}
```

# Edge Set Representation (Directed)



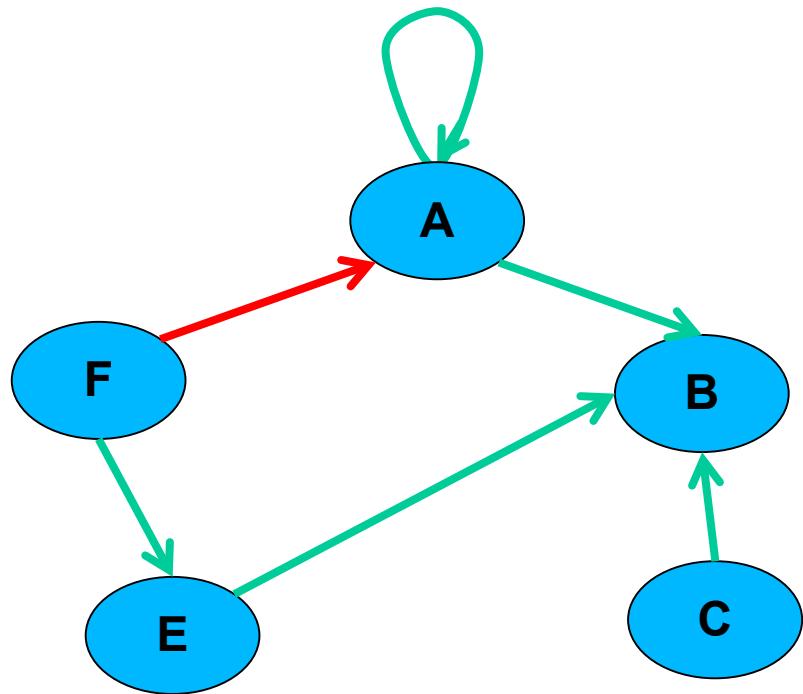
In a **directed** graph, each Edge object contains references to a source Node and destination Node

**edges =**

```
{(A, A), (A, B), (C, B),  
(E, B), (F, A), (F, E)}
```

(and its attribute, if any)

# Edge Set Representation (Directed)



In a **directed** graph, each `Edge` object contains references to a source Node and destination Node

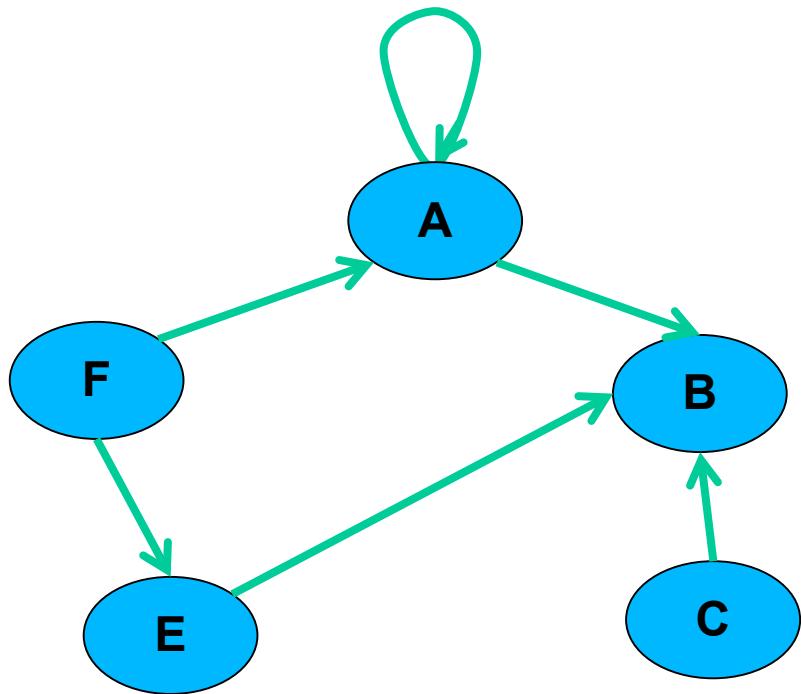
`edges =`

$\{(A, A), (A, B), (C, B), (E, B), (F, A), (F, E)\}$

# Adjacency set

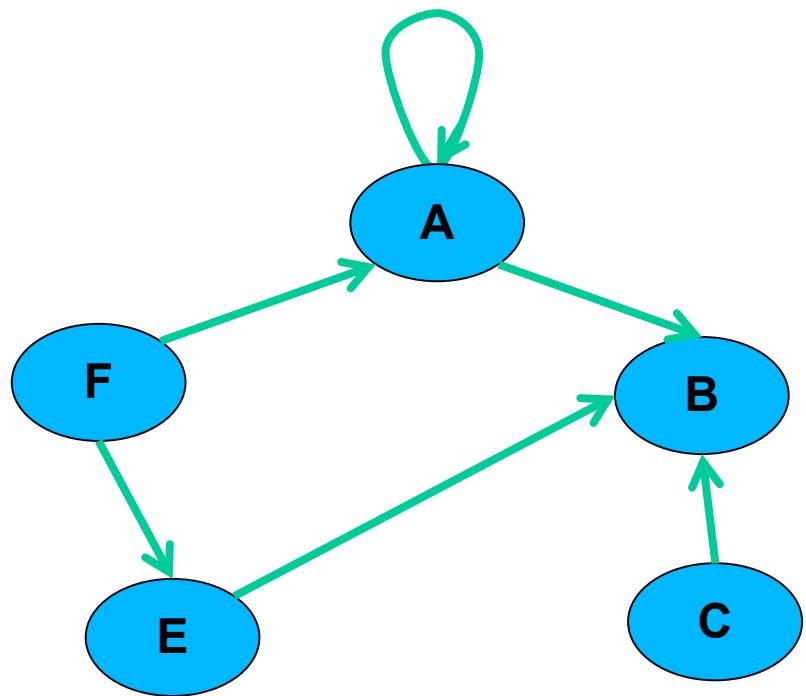
# Adjacency Set Representation

---



The graph is stored as a set of nodes, where each node is mapped to a set of its edges.

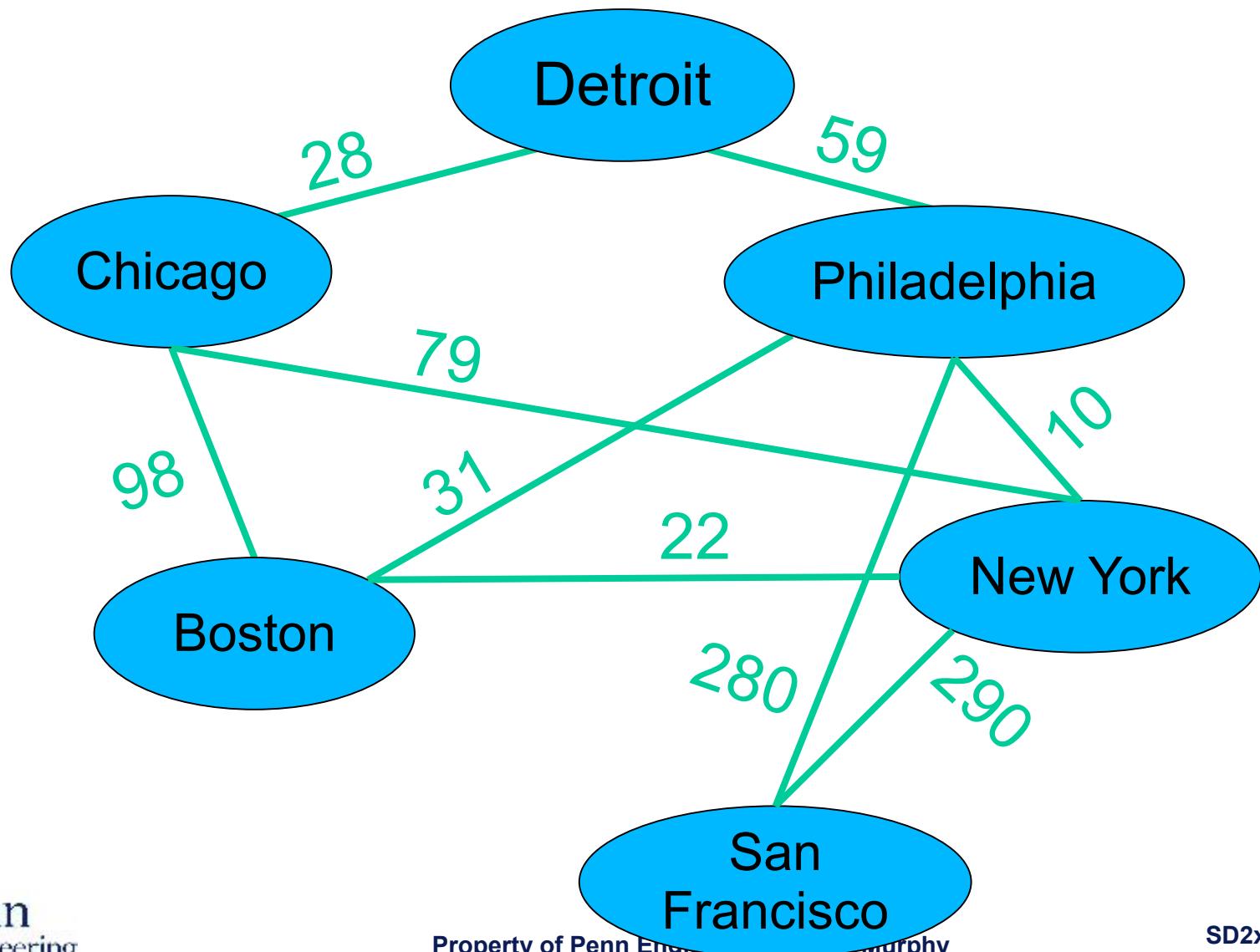
# Adjacency Set Representation (Directed)



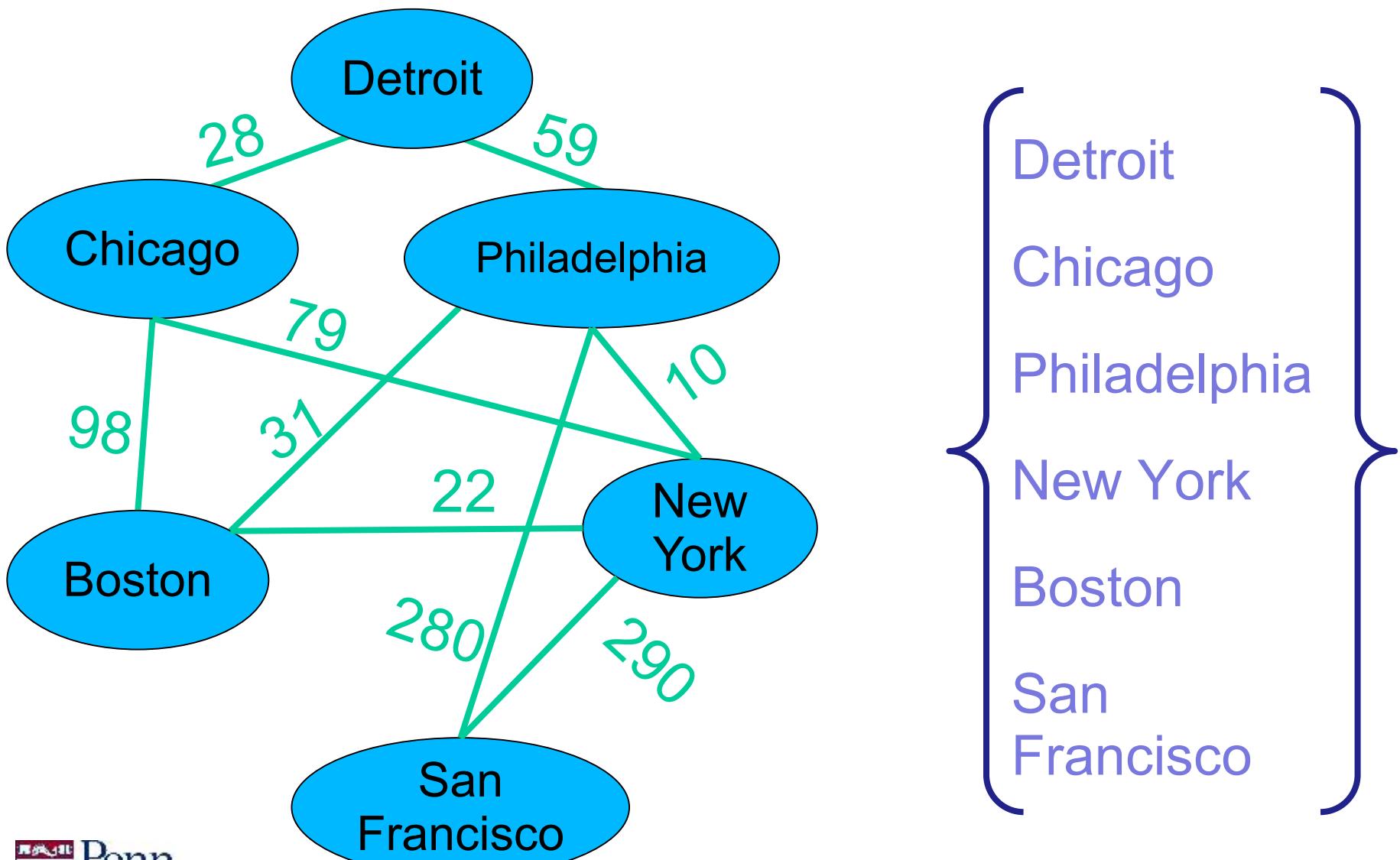
A	→	$\{(A,A), (A,B)\}$
B	→	{ }
C	→	$\{(C,B)\}$
E	→	$\{(E,B)\}$
F	→	$\{(F,A), (F,E)\}$

# Driving Distance between cities (x 10 miles)

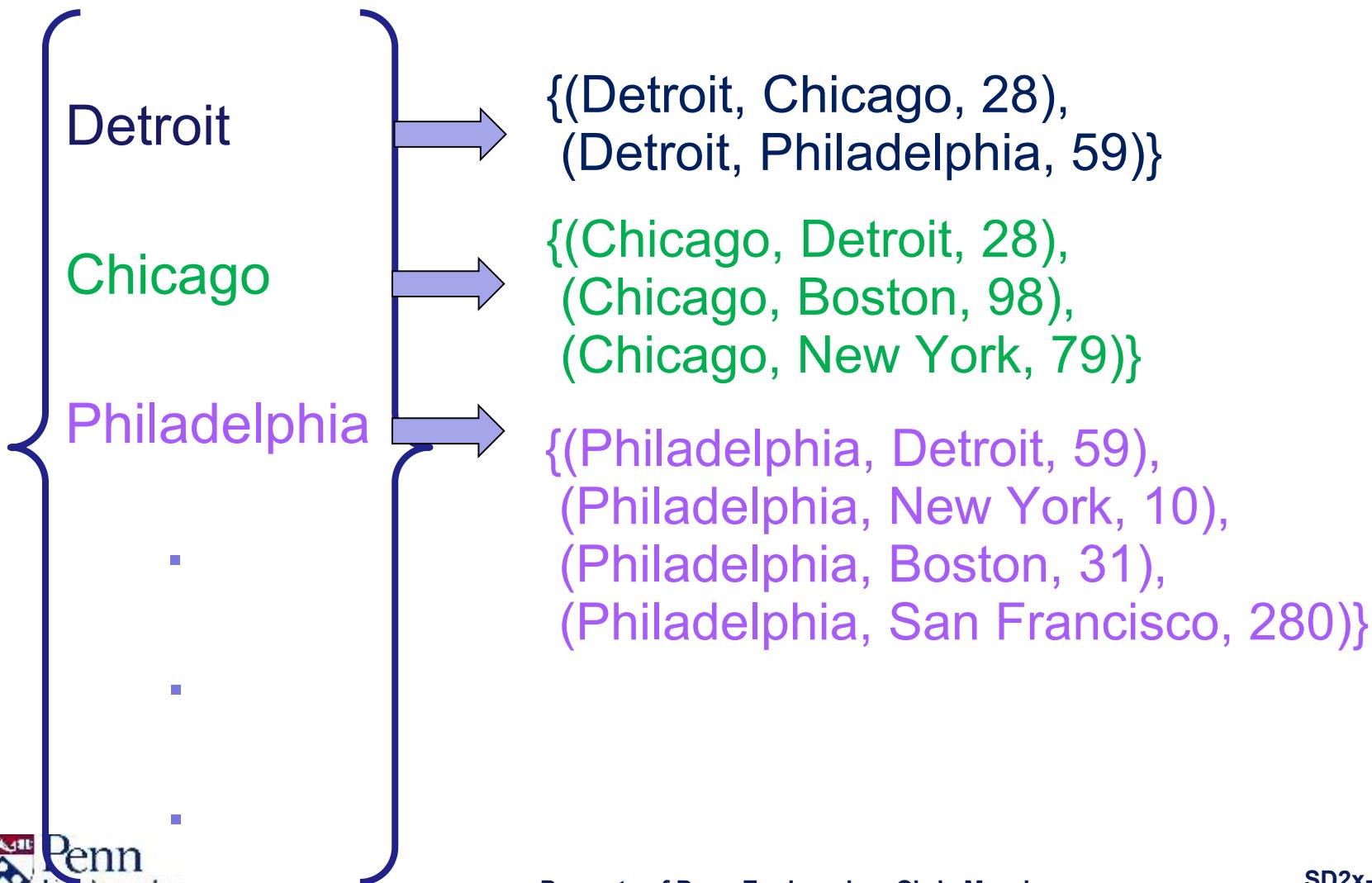
---



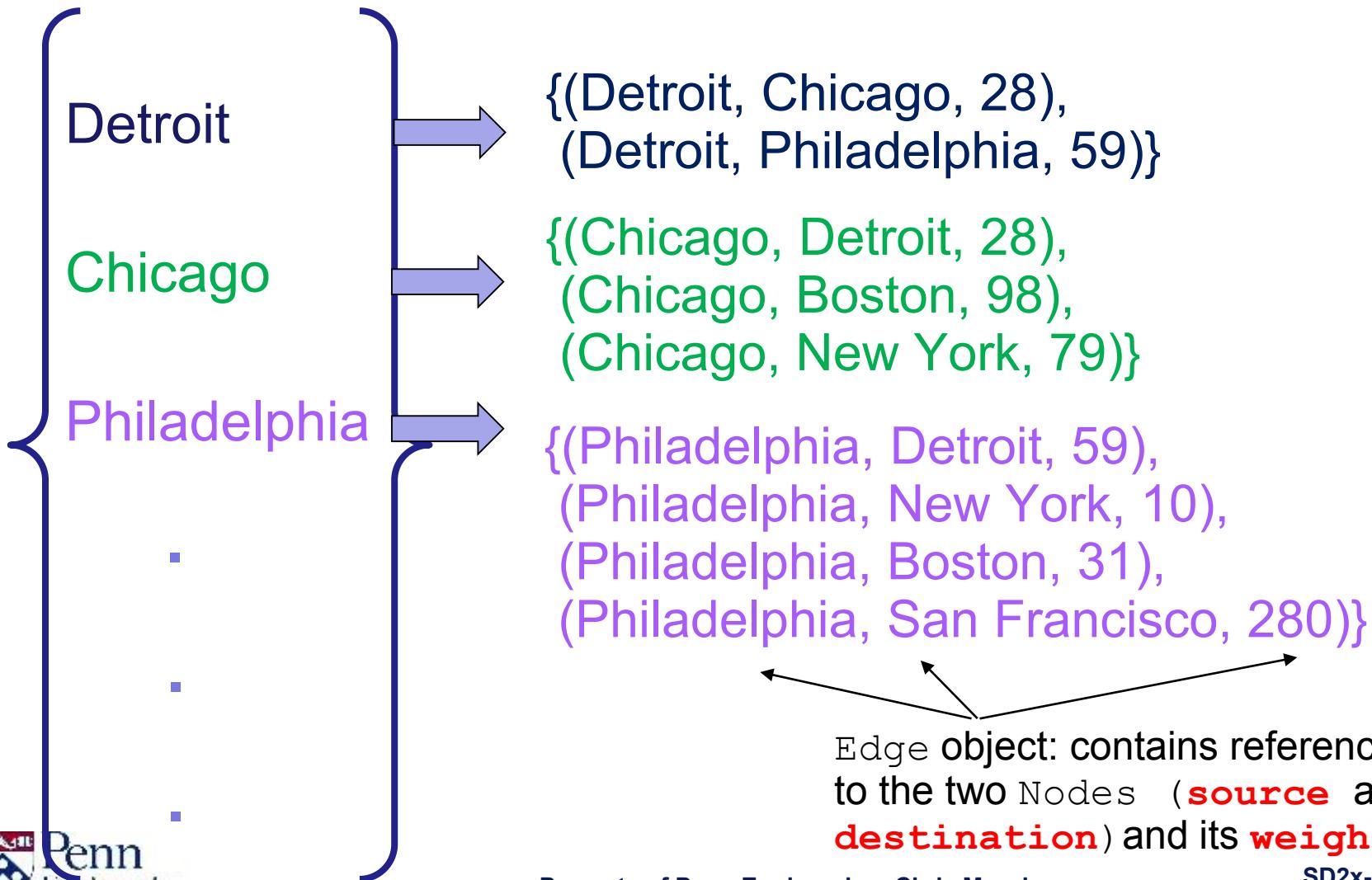
# Weighted, undirected graph



# Adjacency Set Representation



# Adjacency Set Representation



```
public class Node {  
  
    private String element;  
  
    public Node(String element) {  
        this.element = element;  
    }  
  
    public String getElement() {  
        return element;  
    }  
}
```

```
public class Node {  
  
    private String element;  
  
    public Node(String element) {  
        this.element = element;  
    }  
  
    public String getElement() {  
        return element;  
    }  
}
```

```
public class Node {  
  
    private String element;  
  
    public Node(String element) {  
        this.element = element;  
    }  
  
    public String getElement() {  
        return element;  
    }  
  
}
```

```
public class Edge {  
  
    private final Node source;  
    private final Node destination;  
    private final int weight;  
  
    public Edge(Node source, Node destination, int weight) {  
        this.source = source;  
        this.destination = destination;  
        this.weight = weight;  
    }  
  
. . . // getters, toString, etc. as needed  
}
```

```
public class Edge {  
  
    private final Node source;  
    private final Node destination;  
    private final int weight;  
  
    public Edge(Node source, Node destination, int weight) {  
        this.source = source;  
        this.destination = destination;  
        this.weight = weight;  
    }  
  
    . . . // getters, toString, etc. as needed  
}
```

```
public class Edge {  
  
    private final Node source;  
    private final Node destination;  
    private final int weight;  
  
    public Edge(Node source, Node destination, int weight) {  
        this.source = source;  
        this.destination = destination;  
        this.weight = weight;  
    }  
  
. . . // getters, toString, etc. as needed  
  
}
```

```
public class Edge {  
  
    private final Node source;  
    private final Node destination;  
    private final int weight;  
  
    public Edge(Node source, Node destination, int weight) {  
        this.source = source;  
        this.destination = destination;  
        this.weight = weight;  
    }  
  
    . . . // getters, toString, etc. as needed  
  
}
```

```
public class Edge {  
  
    private final Node source;  
    private final Node destination;  
private final int weight;  
  
    public Edge(Node source, Node destination, int weight) {  
        this.source = source;  
        this.destination = destination;  
        this.weight = weight;  
    }  
  
. . . // getters, toString, etc. as needed  
  
}
```

```
public class Edge {  
  
    private final Node source;  
    private final Node destination;  
    private final int weight;  
  
    public Edge(Node source, Node destination, int weight) {  
        this.source = source;  
        this.destination = destination;  
        this.weight = weight;  
    }  
  
. . . // getters, toString, etc. as needed  
  
}
```

```
public class Edge {  
  
    @Override  
    public boolean equals(Edge otherEdge) {  
        Node otherSource = otherEdge.getSource();  
        Node otherDest = otherEdge.getDestination();  
        int otherWeight = otherEdge.getWeight();  
        return (otherSource.equals(source)  
                && otherDest.equals(destination)  
                && otherWeight == weight);  
    }  
  
    @Override  
    public int hashCode() {  
        return source.hashCode() + destination.hashCode() + weight;  
    }  
}
```

```
public abstract class Graph {  
  
    private Map<Node, Set<Edge>> adjacencySets;  
    private int numNodes;  
    private int numEdges;  
  
    public Graph() {  
        adjacencySets = new HashMap<Node, Set<Edge>>();  
        numNodes = 0;  
        numEdges = 0;  
    }  
  
    public int getNumNodes() { return numNodes; }  
    public int getNumEdges() { return numEdges; }  
  
    public boolean containsNode(Node node) {  
        return adjacencySets.containsKey(node);  
    }  
}
```

```
public abstract class Graph {  
  
    private Map<Node, Set<Edge>> adjacencySets;  
    private int numNodes;  
    private int numEdges;  
  
    public Graph() {  
        adjacencySets = new HashMap<Node, Set<Edge>>();  
        numNodes = 0;  
        numEdges = 0;  
    }  
  
    public int getNumNodes() { return numNodes; }  
    public int getNumEdges() { return numEdges; }  
  
    public boolean containsNode(Node node) {  
        return adjacencySets.containsKey(node);  
    }  
}
```

```
public abstract class Graph {  
  
    private Map<Node, Set<Edge>> adjacencySets;  
    private int numNodes;  
    private int numEdges;  
  
    public Graph() {  
        adjacencySets = new HashMap<Node, Set<Edge>>();  
        numNodes = 0;  
        numEdges = 0;  
    }  
  
    public int getNumNodes() { return numNodes; }  
    public int getNumEdges() { return numEdges; }  
  
    public boolean containsNode(Node node) {  
        return adjacencySets.containsKey(node);  
    }  
}
```

```
public abstract class Graph {  
  
    private Map<Node, Set<Edge>> adjacencySets;  
private int numNodes;  
    private int numEdges;  
  
    public Graph() {  
        adjacencySets = new HashMap<Node, Set<Edge>>();  
        numNodes = 0;  
        numEdges = 0;  
    }  
  
    public int getNumNodes() { return numNodes; }  
    public int getNumEdges() { return numEdges; }  
  
    public boolean containsNode(Node node) {  
        return adjacencySets.containsKey(node);  
    }  
}
```

```
public abstract class Graph {  
  
    private Map<Node, Set<Edge>> adjacencySets;  
    private int numNodes;  
private int numEdges;  
  
    public Graph() {  
        adjacencySets = new HashMap<Node, Set<Edge>>();  
        numNodes = 0;  
        numEdges = 0;  
    }  
  
    public int getNumNodes() { return numNodes; }  
    public int getNumEdges() { return numEdges; }  
  
    public boolean containsNode(Node node) {  
        return adjacencySets.containsKey(node);  
    }  
}
```

```
public abstract class Graph {  
  
    private Map<Node, Set<Edge>> adjacencySets;  
    private int numNodes;  
    private int numEdges;  
  
    public Graph() {  
        adjacencySets = new HashMap<Node, Set<Edge>>();  
        numNodes = 0;  
        numEdges = 0;  
    }  
  
    public int getNumNodes() { return numNodes; }  
    public int getNumEdges() { return numEdges; }  
  
public boolean containsNode(Node node) {  
    return adjacencySets.containsKey(node);  
}  
}
```

```
public abstract class Graph {  
    . . .  
    public boolean addNode(Node newNode) {  
        if (newNode == null || containsNode(newNode)) {  
            return false;  
        }  
  
        Set<Edge> newAdjacencySet = new HashSet<Edge>();  
        adjacencySets.put(newNode, newAdjacencySet);  
        numNodes++;  
        return true;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
public boolean addNode(Node newNode) {  
    if (newNode == null || containsNode(newNode)) {  
        return false;  
    }  
  
    Set<Edge> newAdjacencySet = new HashSet<Edge>();  
    adjacencySets.put(newNode, newAdjacencySet);  
    numNodes++;  
    return true;  
}  
}
```

```
public abstract class Graph {  
    . . .  
    public boolean addNode(Node newNode) {  
  
        if (newNode == null || containsNode(newNode)) {  
            return false;  
        }  
  
        Set<Edge> newAdjacencySet = new HashSet<Edge>();  
        adjacencySets.put(newNode, newAdjacencySet);  
        numNodes++;  
        return true;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
    public boolean addNode(Node newNode) {  
        if (newNode == null || containsNode(newNode)) {  
            return false;  
        }  
  
        Set<Edge> newAdjacencySet = new HashSet<Edge>();  
        adjacencySets.put(newNode, newAdjacencySet);  
        numNodes++;  
        return true;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
    public boolean addNode(Node newNode) {  
        if (newNode == null || containsNode(newNode)) {  
            return false;  
        }  
  
        Set<Edge> newAdjacencySet = new HashSet<Edge>();  
        adjacencySets.put(newNode, newAdjacencySet);  
        numNodes++;  
        return true;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
    public boolean addNode(Node newNode) {  
        if (newNode == null || containsNode(newNode)) {  
            return false;  
        }  
  
        Set<Edge> newAdjacencySet = new HashSet<Edge>();  
        adjacencySets.put(newNode, newAdjacencySet);  
        numNodes++;  
        return true;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
    public abstract boolean removeEdge(  
        Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
public Set<Node> getNodeNeighbors(Node node) {  
    if (!containsNode(node)) {  
        return null;  
    }  
    Set<Edge> nodeEdges = adjacencySets.get(node);  
    Set<Node> nodeNeighbors = new HashSet<Node>();  
    for (Edge e : nodeEdges) {  
        Node neighbor = e.getDestination(node);  
        nodeNeighbors.add(neighbor);  
    }  
    return nodeNeighbors;  
}  
  
public abstract boolean addEdge(  
    Node node1, Node node2, int weight);  
public abstract boolean removeEdge(  
    Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
    public abstract boolean removeEdge(  
        Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
    public abstract boolean removeEdge(  
        Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
    public abstract boolean removeEdge(  
        Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
    public abstract boolean removeEdge(  
        Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
    public abstract boolean removeEdge(  
        Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
}
```

```
public abstract boolean addEdge(  
    Node node1, Node node2, int weight);  
public abstract boolean removeEdge(  
    Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
  
    public Set<Node> getNodeNeighbors(Node node) {  
        if (!containsNode(node)) {  
            return null;  
        }  
        Set<Edge> nodeEdges = adjacencySets.get(node);  
        Set<Node> nodeNeighbors = new HashSet<Node>();  
        for (Edge e : nodeEdges) {  
            Node neighbor = e.getDestination(node);  
            nodeNeighbors.add(neighbor);  
        }  
        return nodeNeighbors;  
    }  
  
    public abstract boolean addEdge(  
        Node node1, Node node2, int weight);  
public abstract boolean removeEdge(  
    Node node1, Node node2, int weight);  
}
```

```
public abstract class Graph {  
    . . .  
    private boolean addEdgeFromTo(Node source,  
                                  Node destination, int weight) {  
  
        Edge newEdge = new Edge(source, destination, weight);  
        Set<Edge> sourceEdges = adjacencySets.get(source);  
  
        if (!sourceEdges.contains(newEdge)) {  
            sourceEdges.add(newEdge);  
            return true;  
        }  
        return false;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
  
private boolean addEdgeFromTo(Node source,  
                           Node destination, int weight) {  
  
    Edge newEdge = new Edge(source, destination, weight);  
    Set<Edge> sourceEdges = adjacencySets.get(source);  
  
    if (!sourceEdges.contains(newEdge)) {  
        sourceEdges.add(newEdge);  
        return true;  
    }  
    return false;  
}  
}
```

```
public abstract class Graph {  
    . . .  
    private boolean addEdgeFromTo(Node source,  
                                  Node destination, int weight) {  
    Edge newEdge = new Edge(source, destination, weight);  
    Set<Edge> sourceEdges = adjacencySets.get(source);  
  
    if (!sourceEdges.contains(newEdge)) {  
        sourceEdges.add(newEdge);  
        return true;  
    }  
    return false;  
}  
}
```

```
public abstract class Graph {  
    . . .  
    private boolean addEdgeFromTo(Node source,  
                                Node destination, int weight) {  
        Edge newEdge = new Edge(source, destination, weight);  
        Set<Edge> sourceEdges = adjacencySets.get(source);  
        if (!sourceEdges.contains(newEdge)) {  
            sourceEdges.add(newEdge);  
            return true;  
        }  
        return false;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
    private boolean addEdgeFromTo(Node source,  
                                  Node destination, int weight) {  
  
        Edge newEdge = new Edge(source, destination, weight);  
        Set<Edge> sourceEdges = adjacencySets.get(source);  
  
        if (!sourceEdges.contains(newEdge)) {  
            sourceEdges.add(newEdge);  
            return true;  
        }  
        return false;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
    private boolean addEdgeFromTo(Node source,  
                                Node destination, int weight) {  
  
        Edge newEdge = new Edge(source, destination, weight);  
        Set<Edge> sourceEdges = adjacencySets.get(source);  
  
        if (!sourceEdges.contains(newEdge)) {  
            sourceEdges.add(newEdge);  
            return true;  
        }  
        return false;  
    }  
}
```

```
public abstract class Graph {  
    . . .  
    private boolean addEdgeFromTo(Node source,  
                                Node destination, int weight) {  
  
        Edge newEdge = new Edge(source, destination, weight);  
        Set<Edge> sourceEdges = adjacencySets.get(source);  
  
        if (!sourceEdges.contains(newEdge)) {  
            sourceEdges.add(newEdge);  
            return true;  
        }  
        return false;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class UndirectedGraph extends Graph {  
  
    public UndirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node node1, Node node2, int weight) {  
  
        addNode(node1); // only adds if node not already in graph.  
        addNode(node2);  
  
        boolean addEdgeSuccess = (  
            addEdgeFromTo(node1, node2, weight)  
            && addEdgeFromTo(node2, node1, weight));  
  
        if (addEdgeSuccess) {  
            numEdges++;  
        }  
        return addEdgeSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

```
public class DirectedGraph extends Graph {  
  
    public DirectedGraph() {  
        super();  
    }  
  
    public boolean addEdge(Node source, Node dest, int weight) {  
  
        addNode(source); // only adds if node not already in graph.  
        addNode(dest);  
  
        boolean addSuccess = addEdgeFromTo(source, dest, weight);  
  
        if (addSuccess) {  
            numEdges++;  
        }  
        return addSuccess;  
    }  
}
```

# Recap: Graph Representation

---

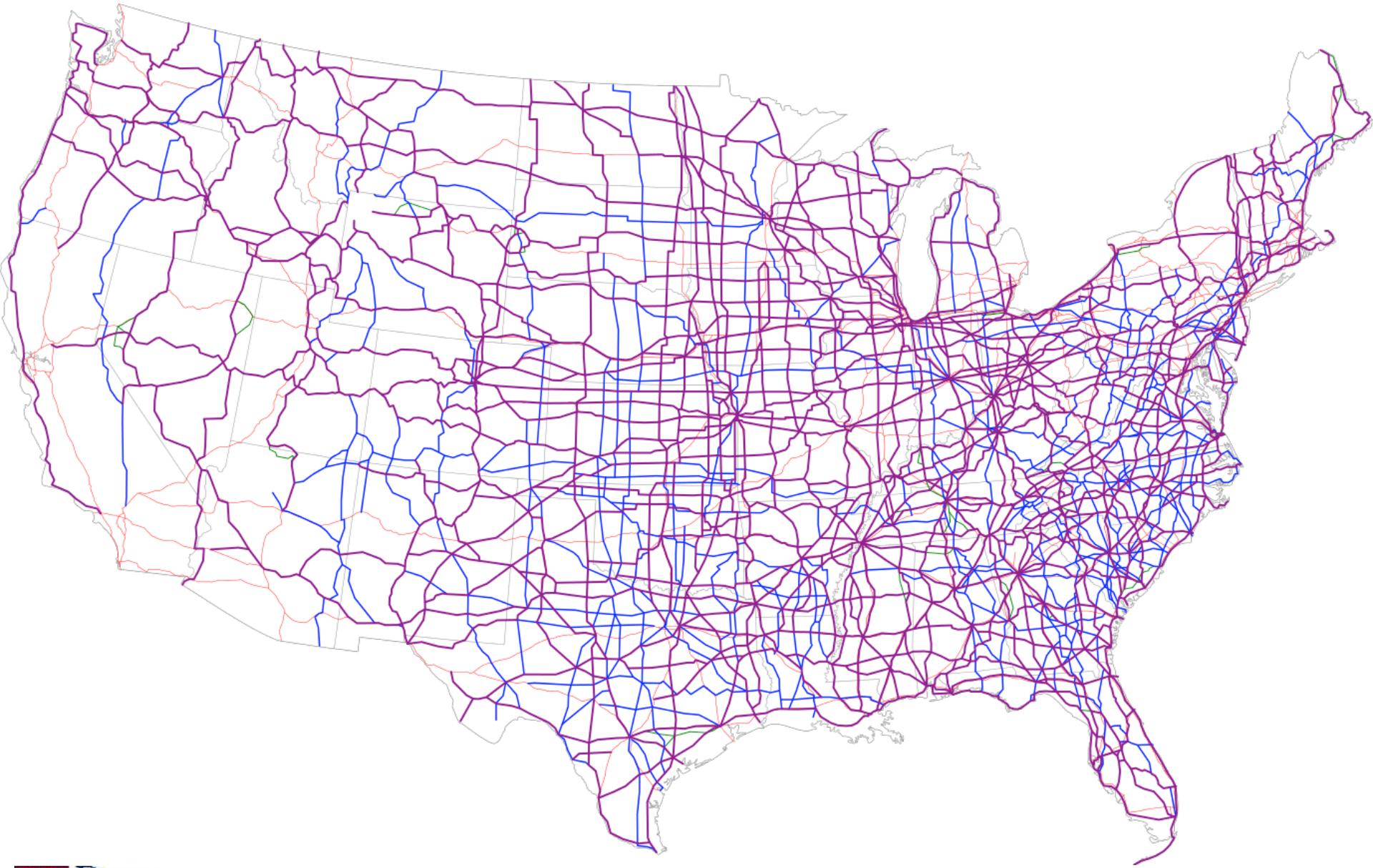
- **Adjacency Matrix**: 2-dimensional array in which values represent weights
- **Edge Set**: set of all edges
- **Adjacency Set**: mapping of nodes to edges

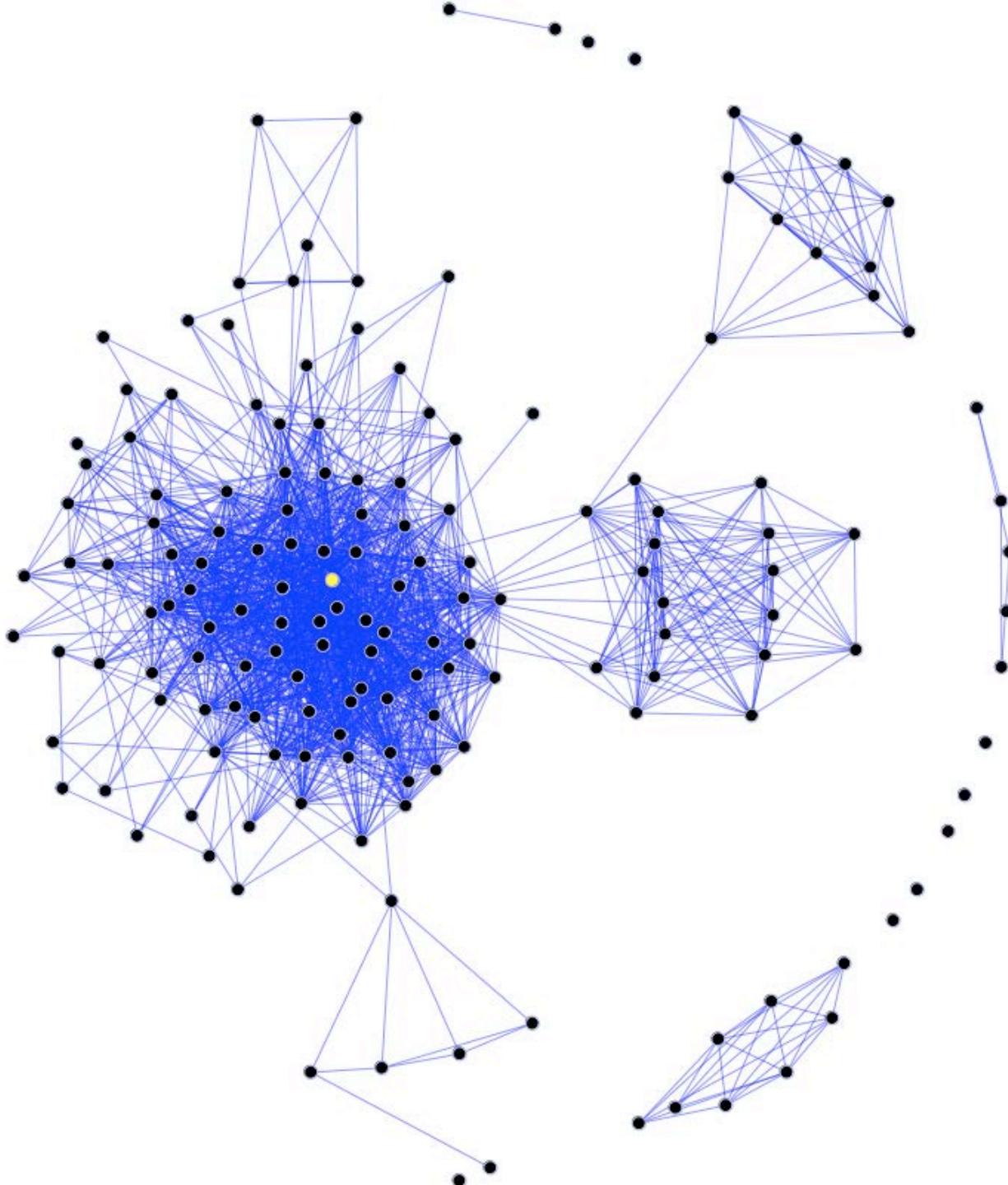
# **SD2x2.11**

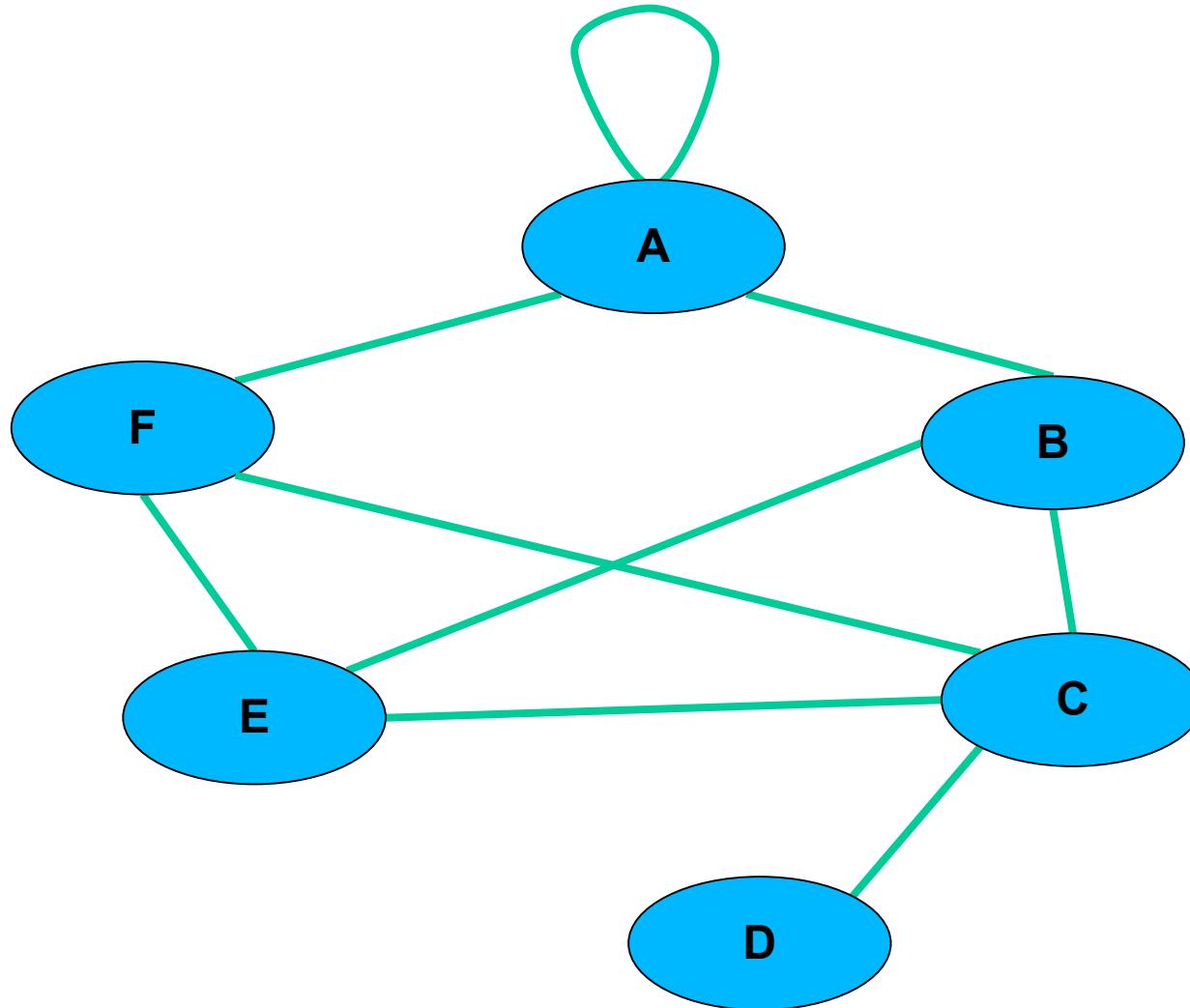
## **Graph BFS**

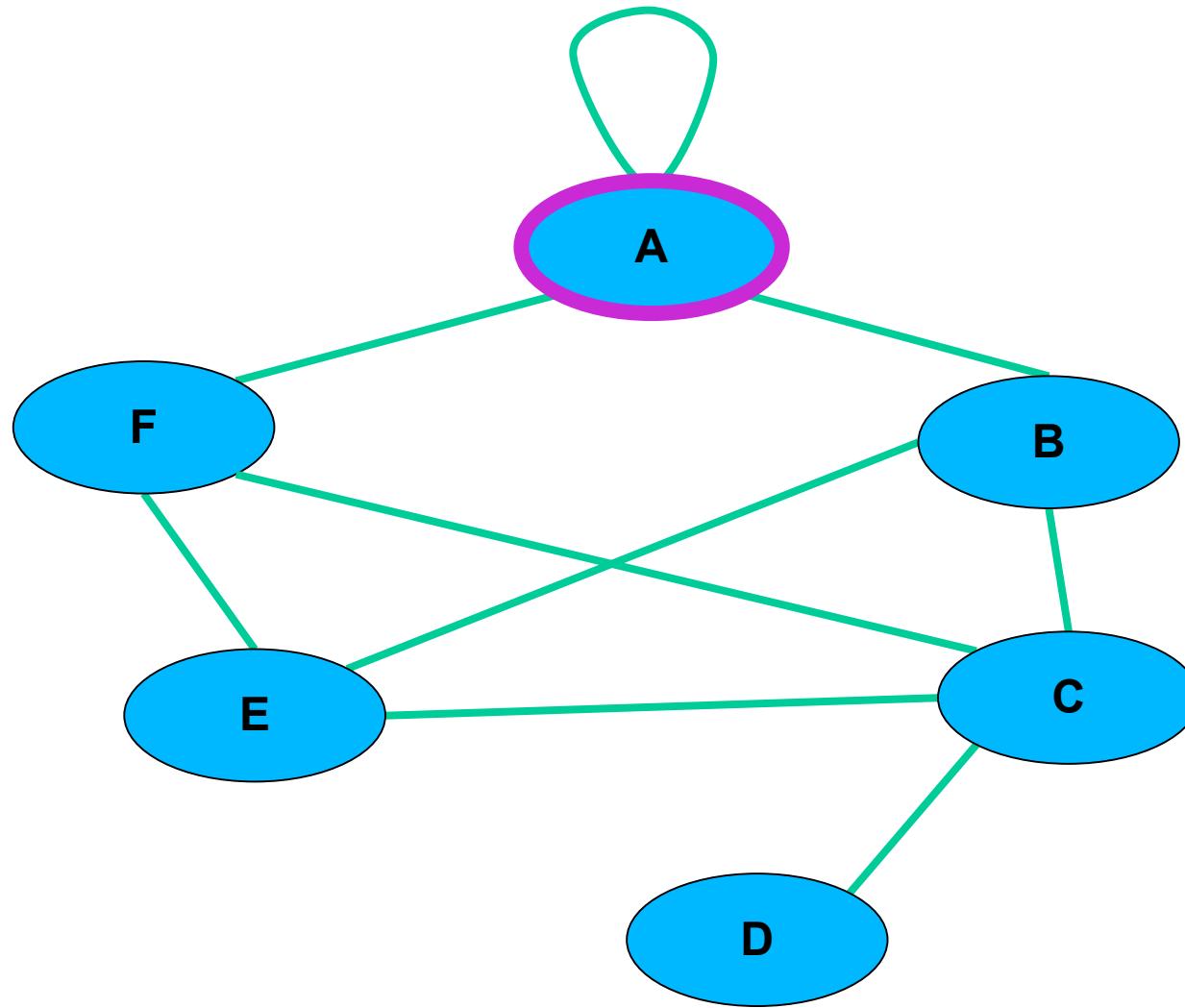
### **Chris**

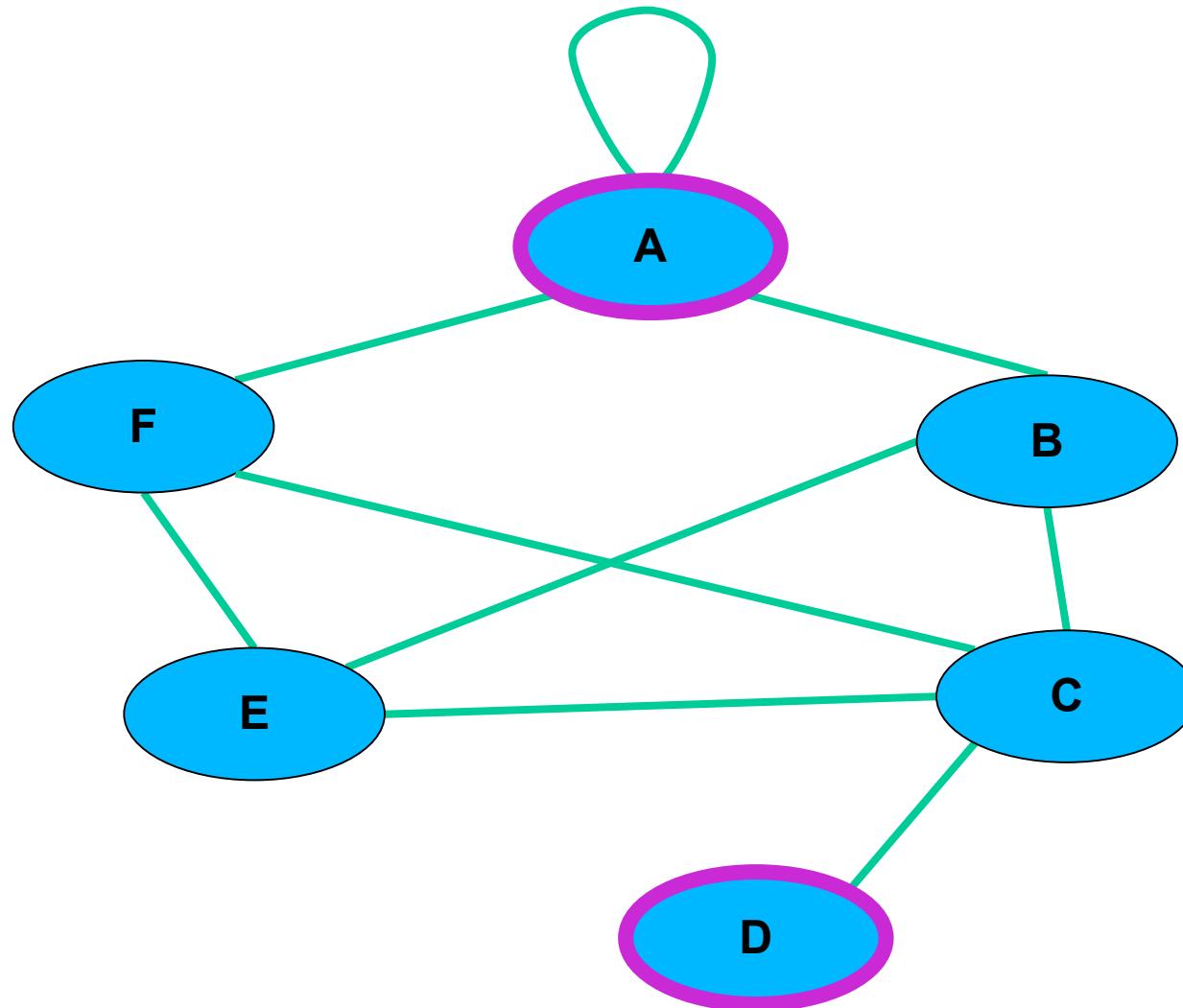
# **How do we find a path from one node to another?**

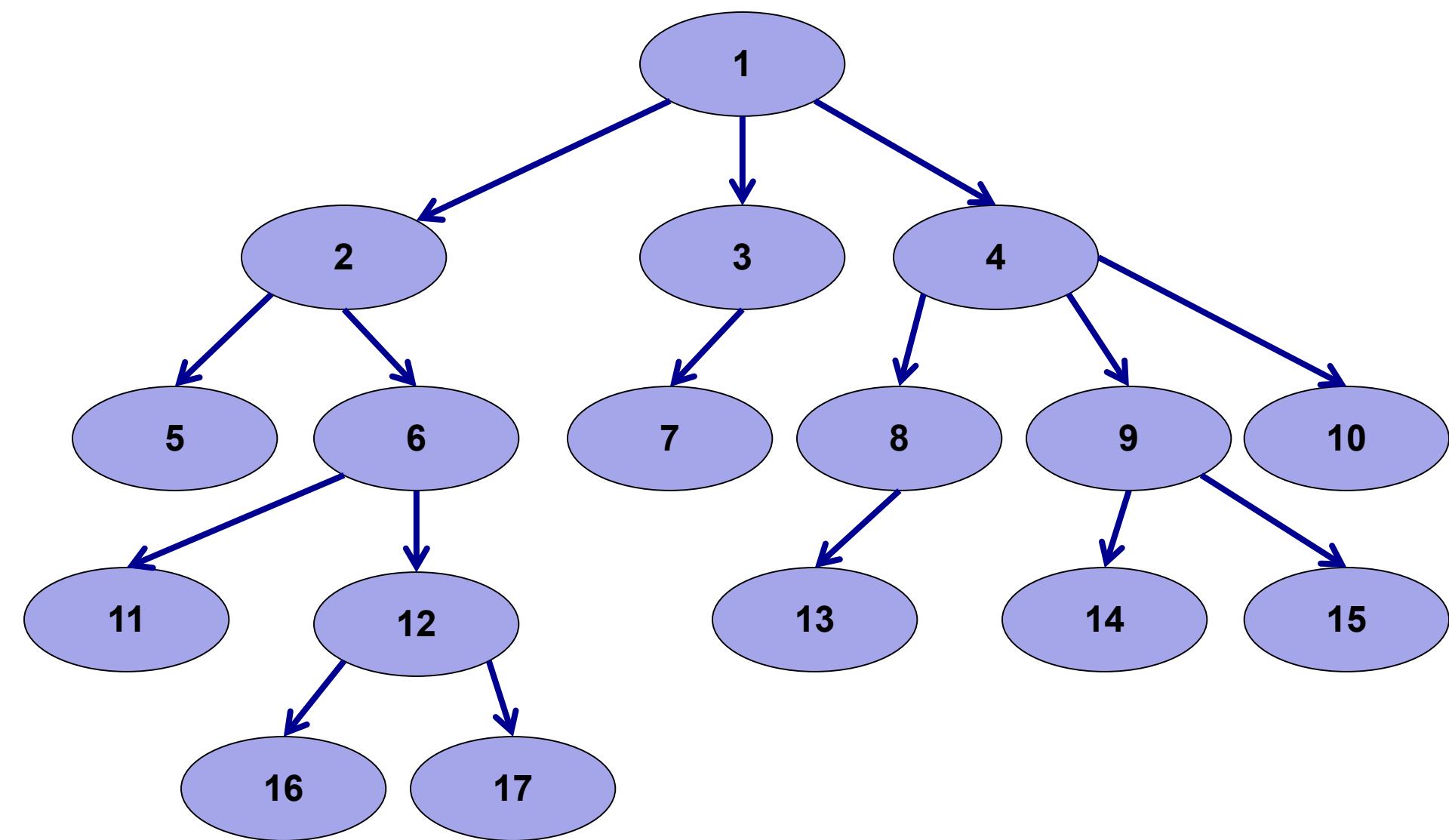


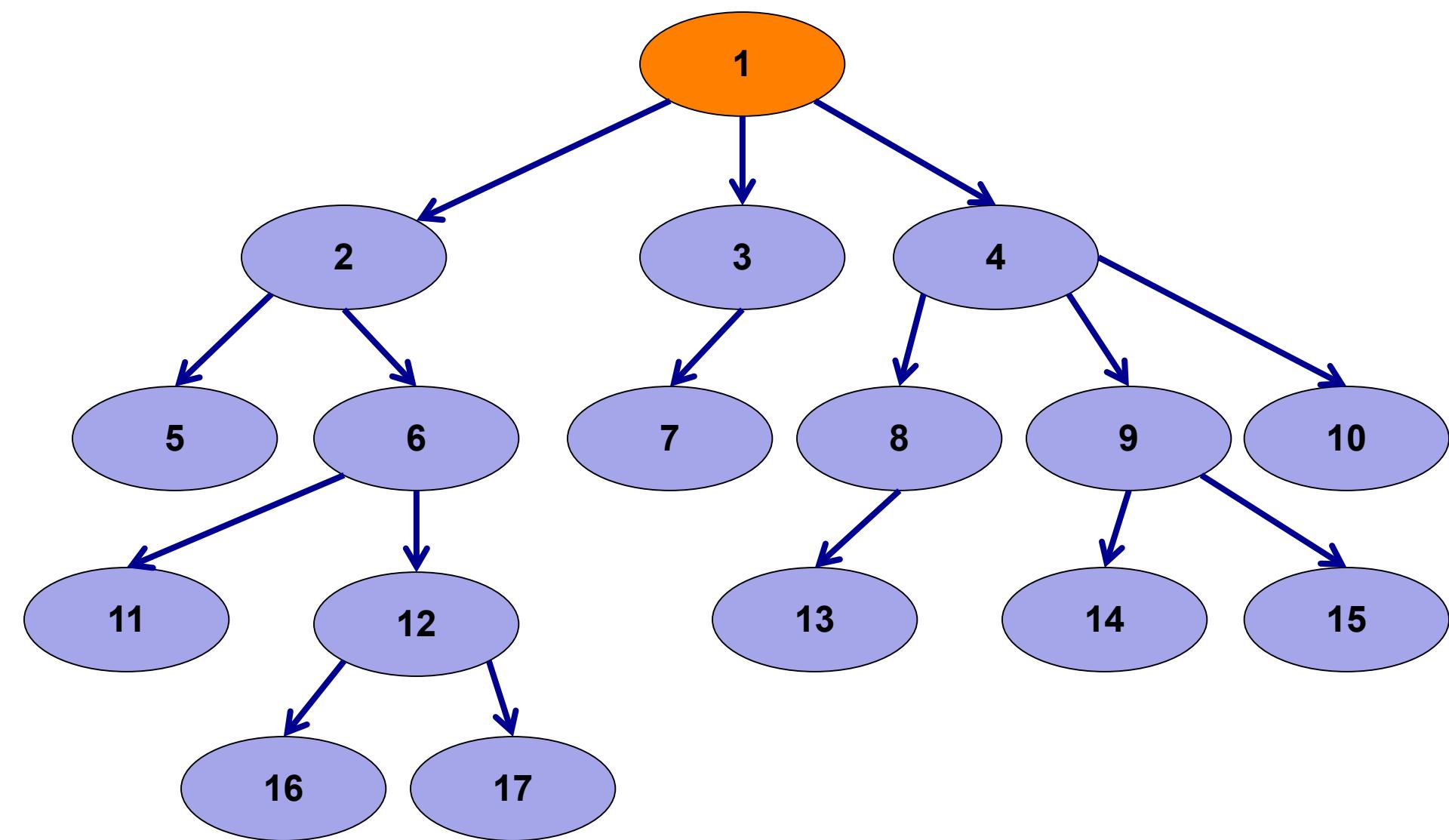


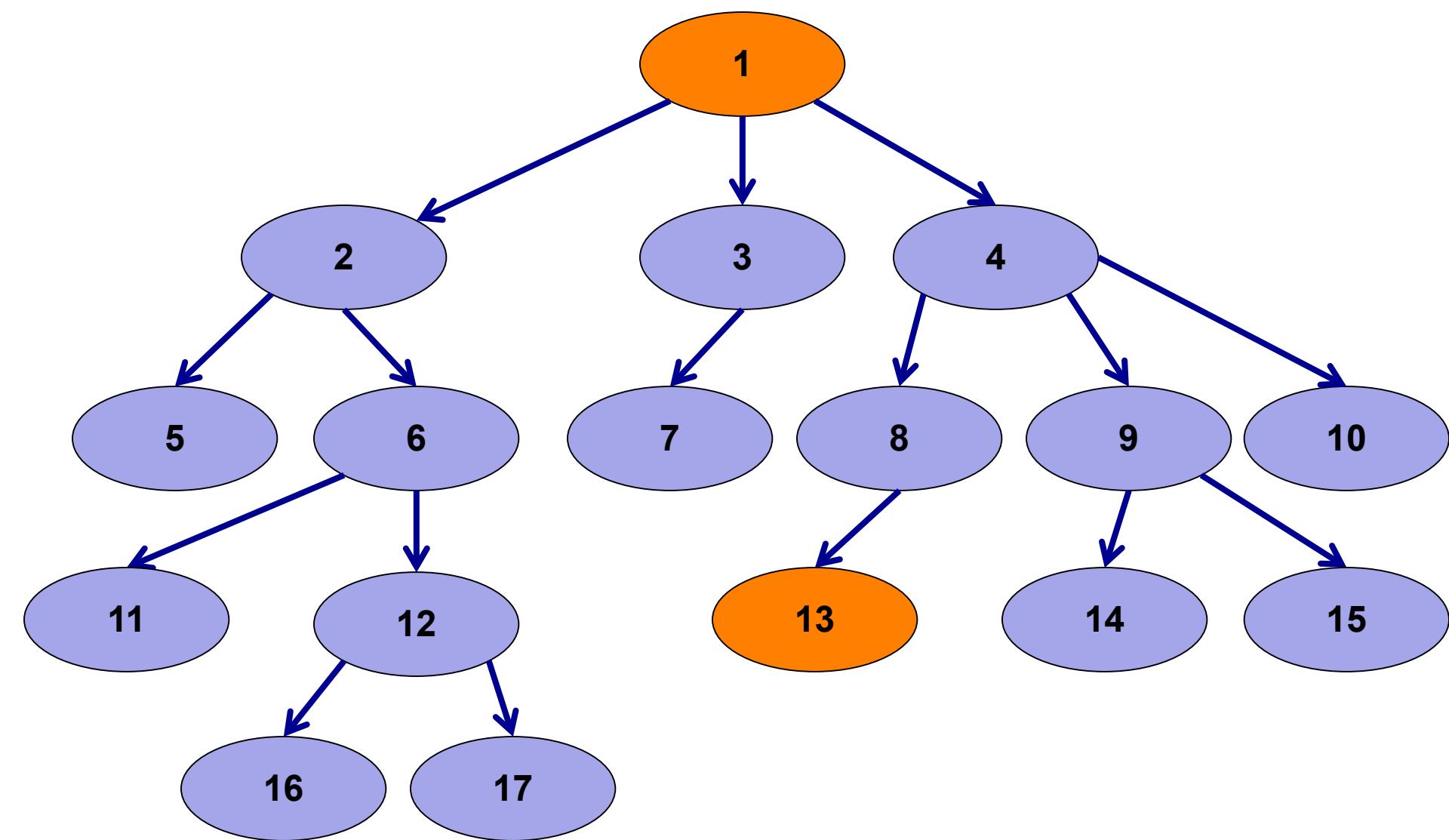


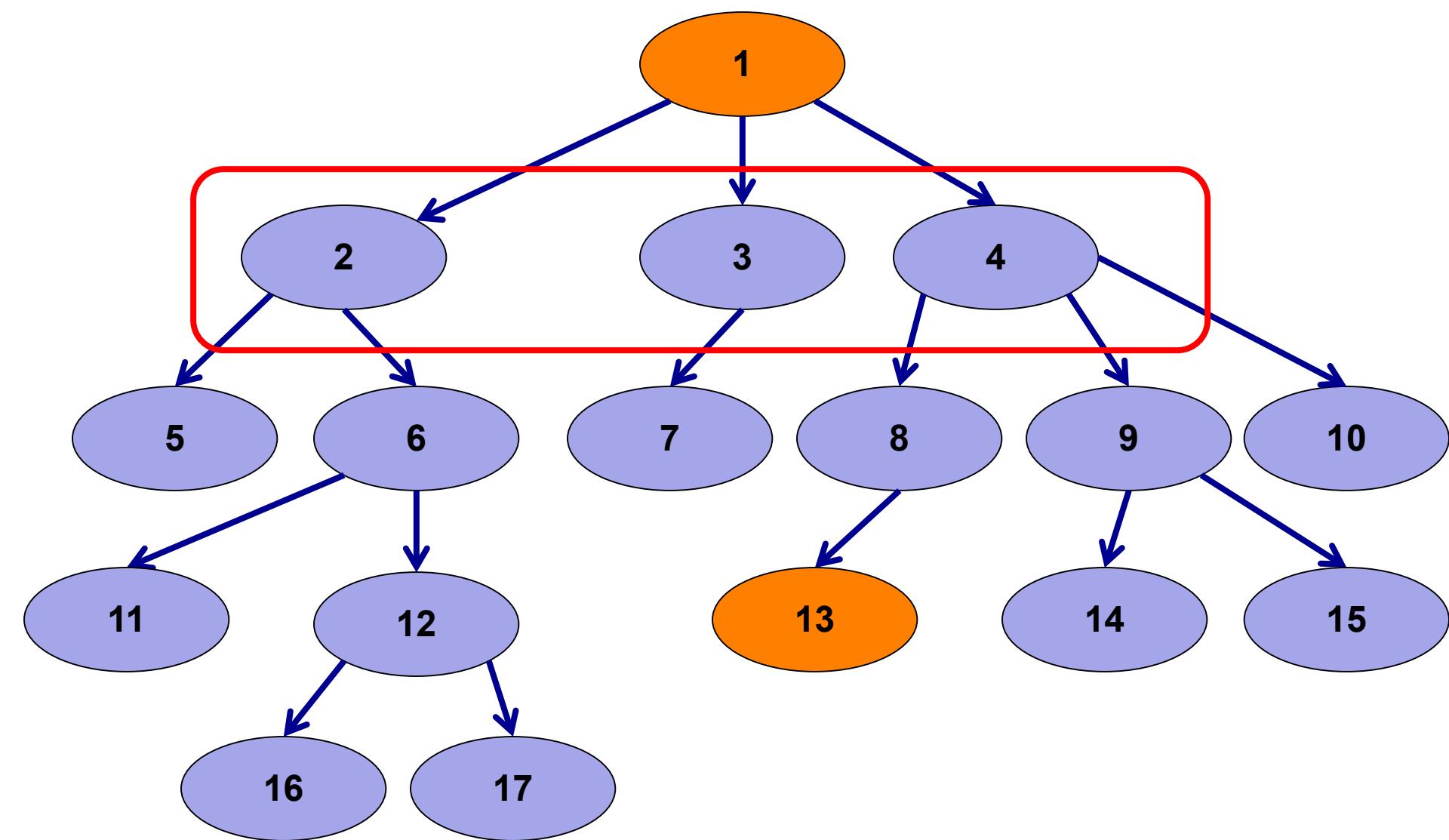


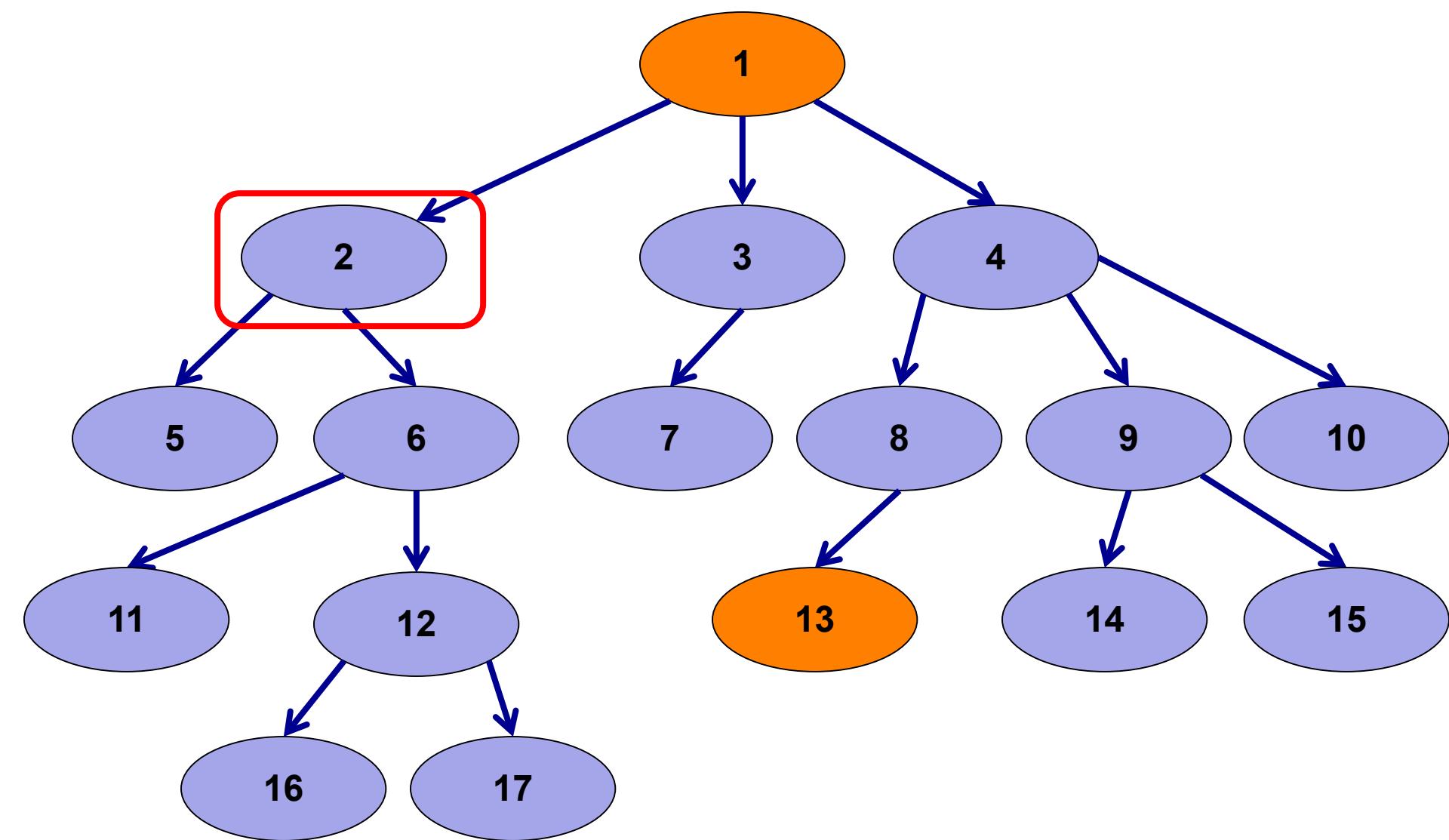


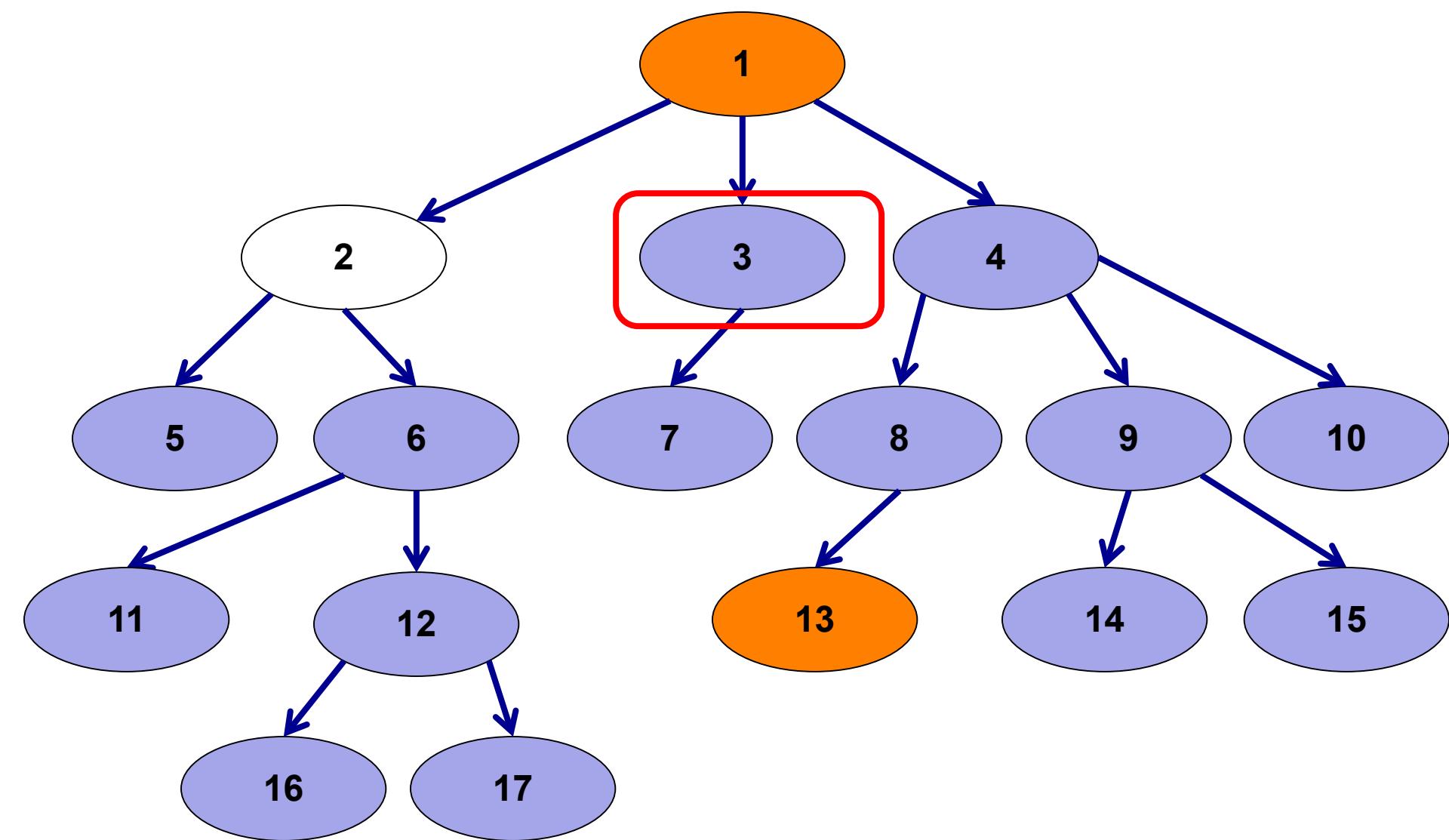


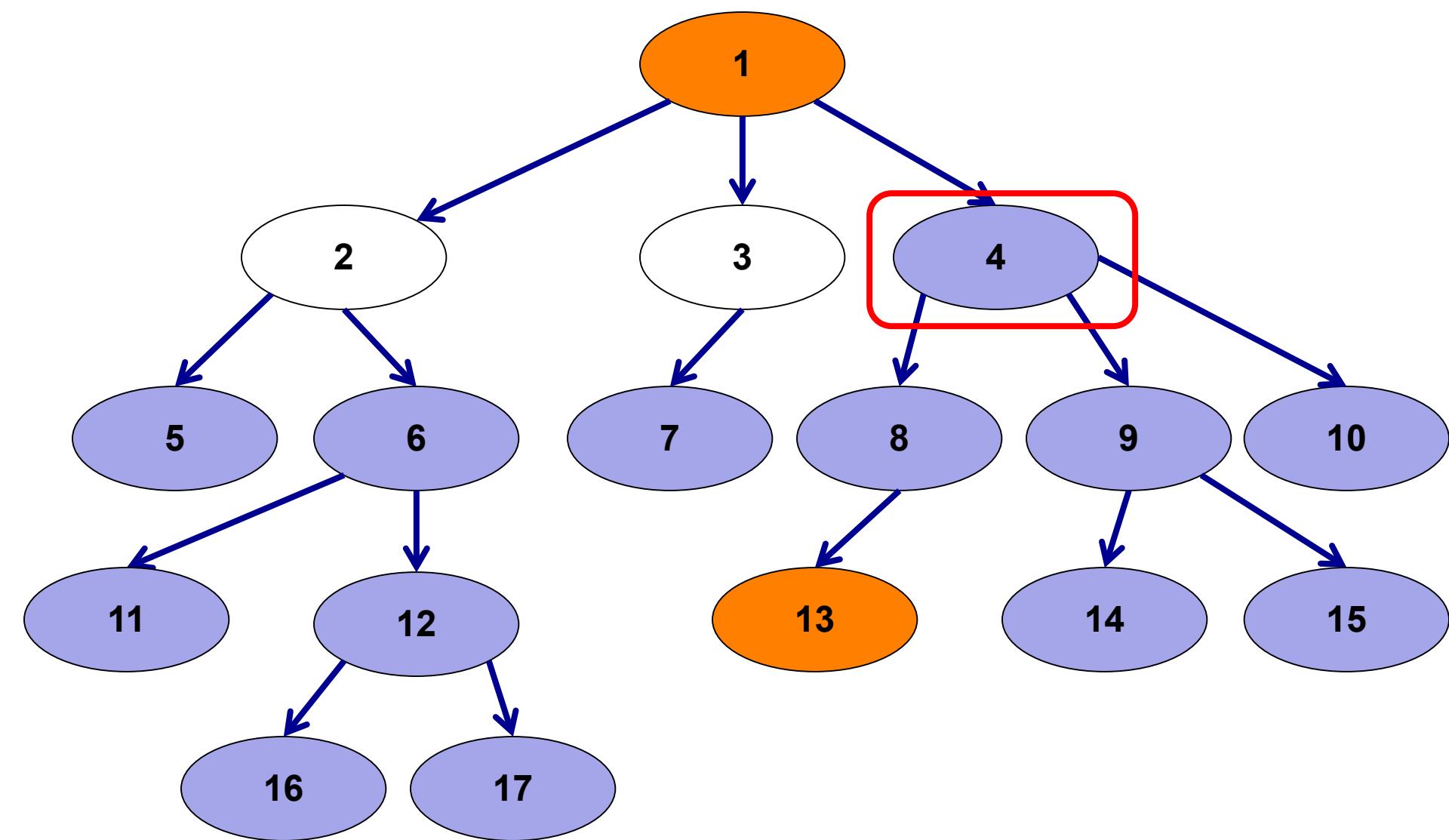


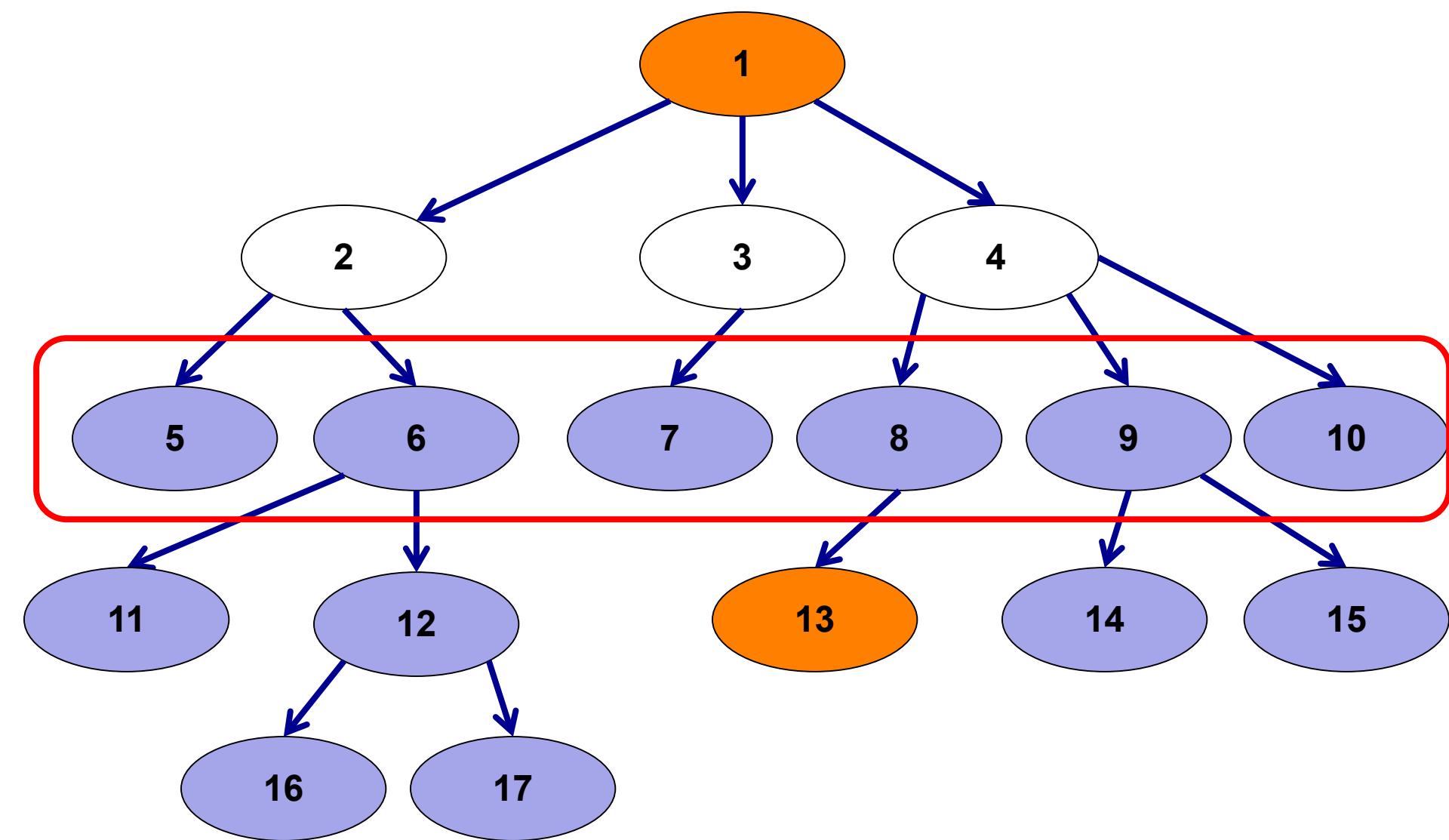


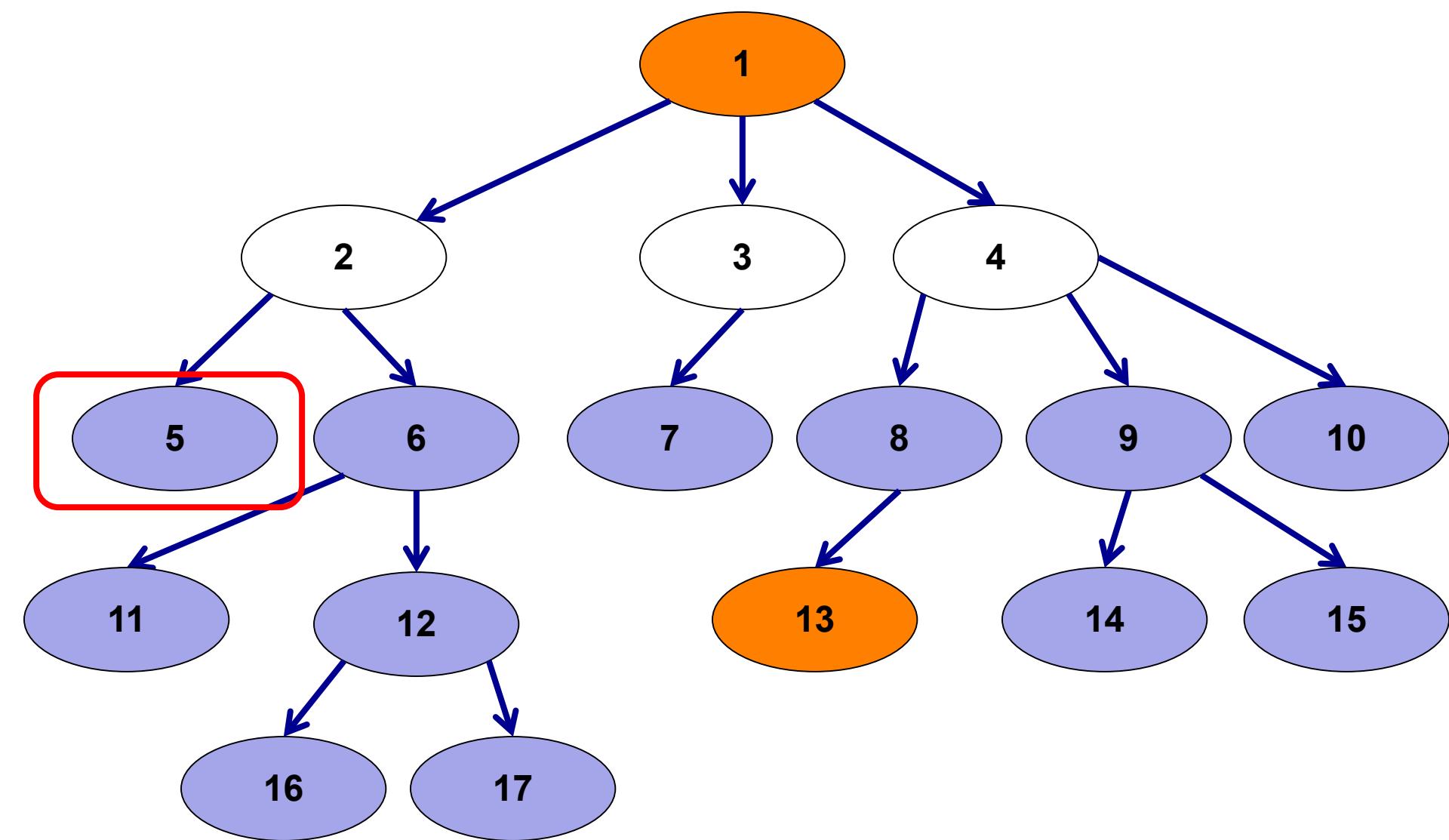


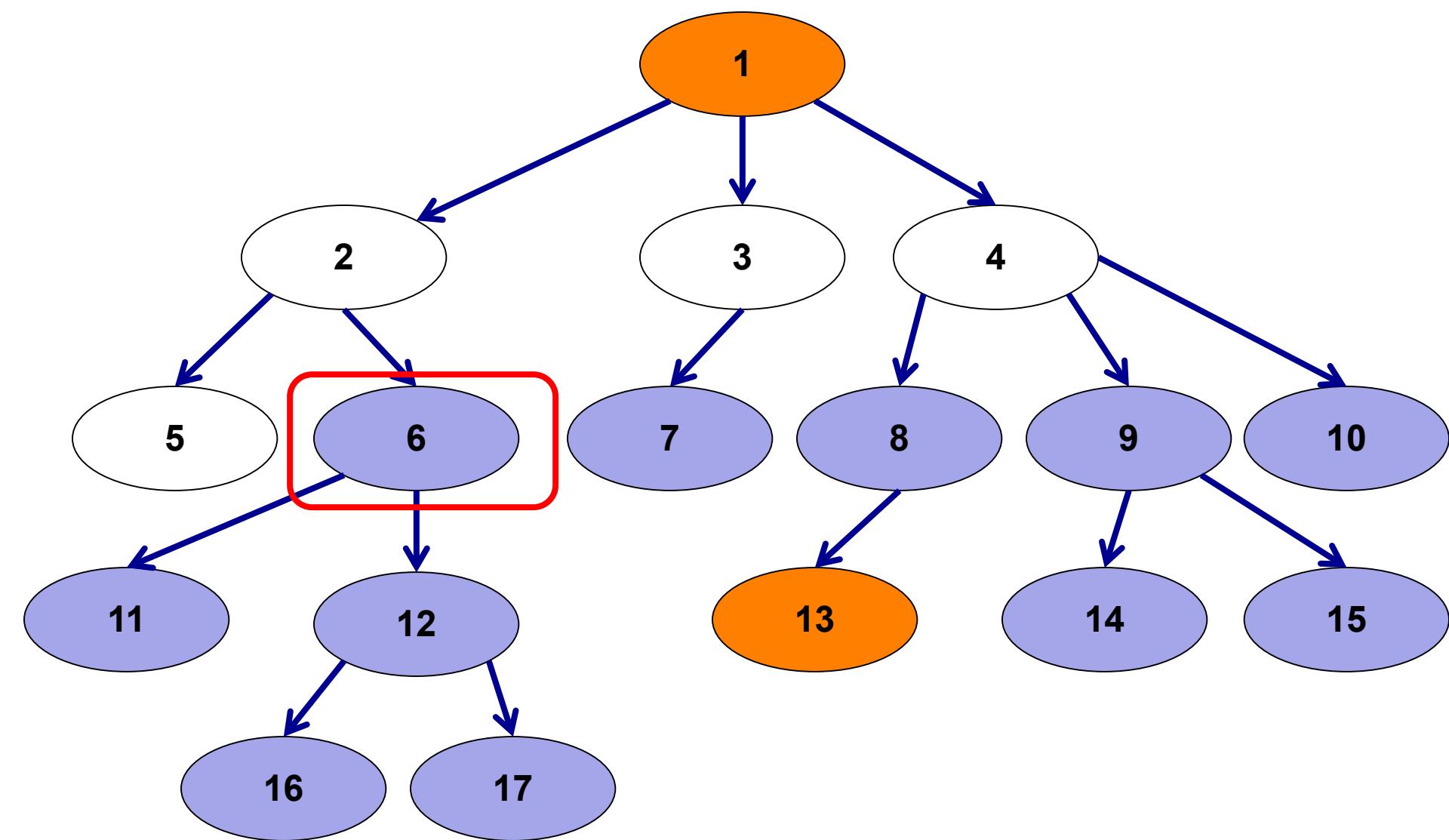


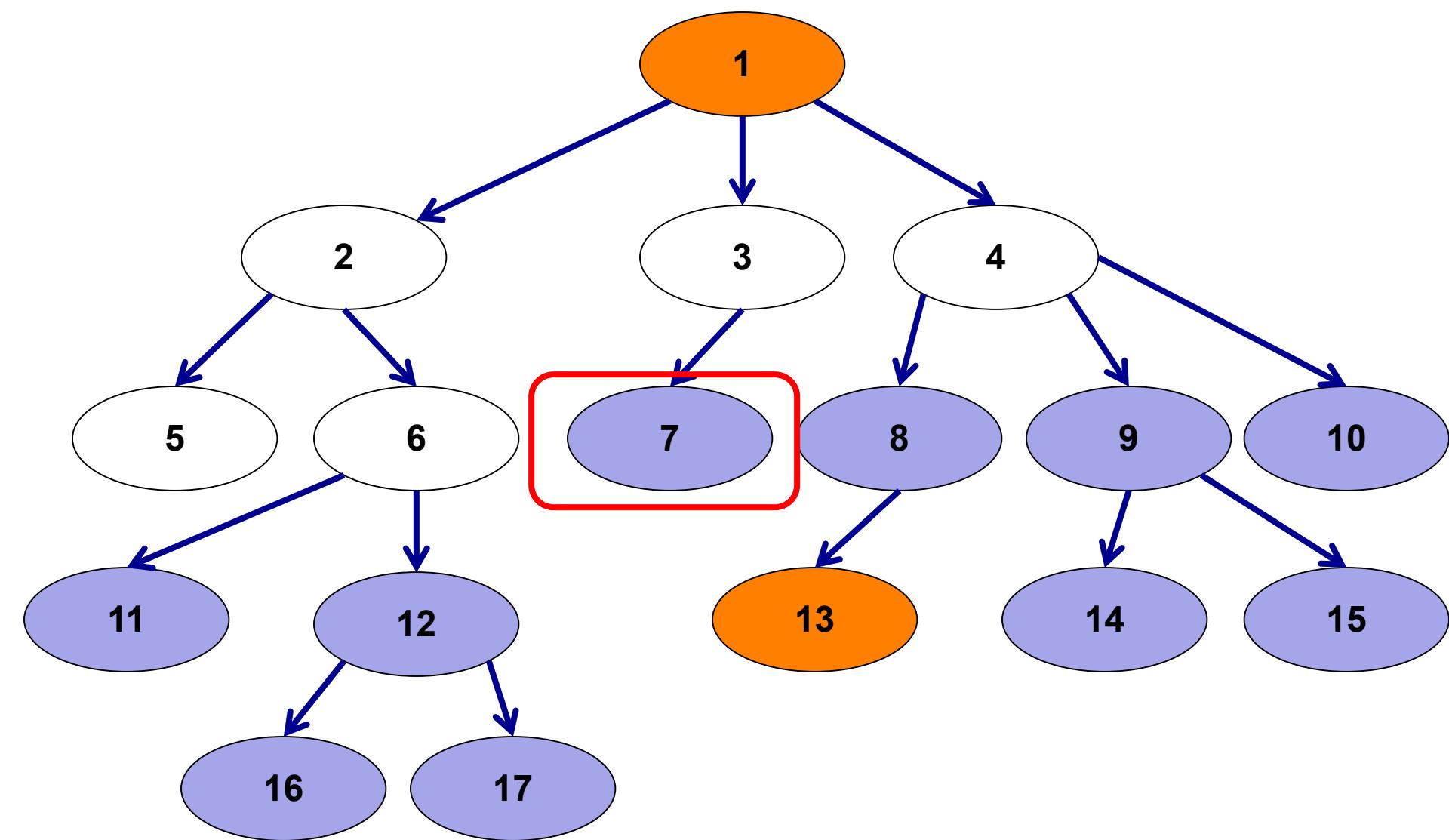


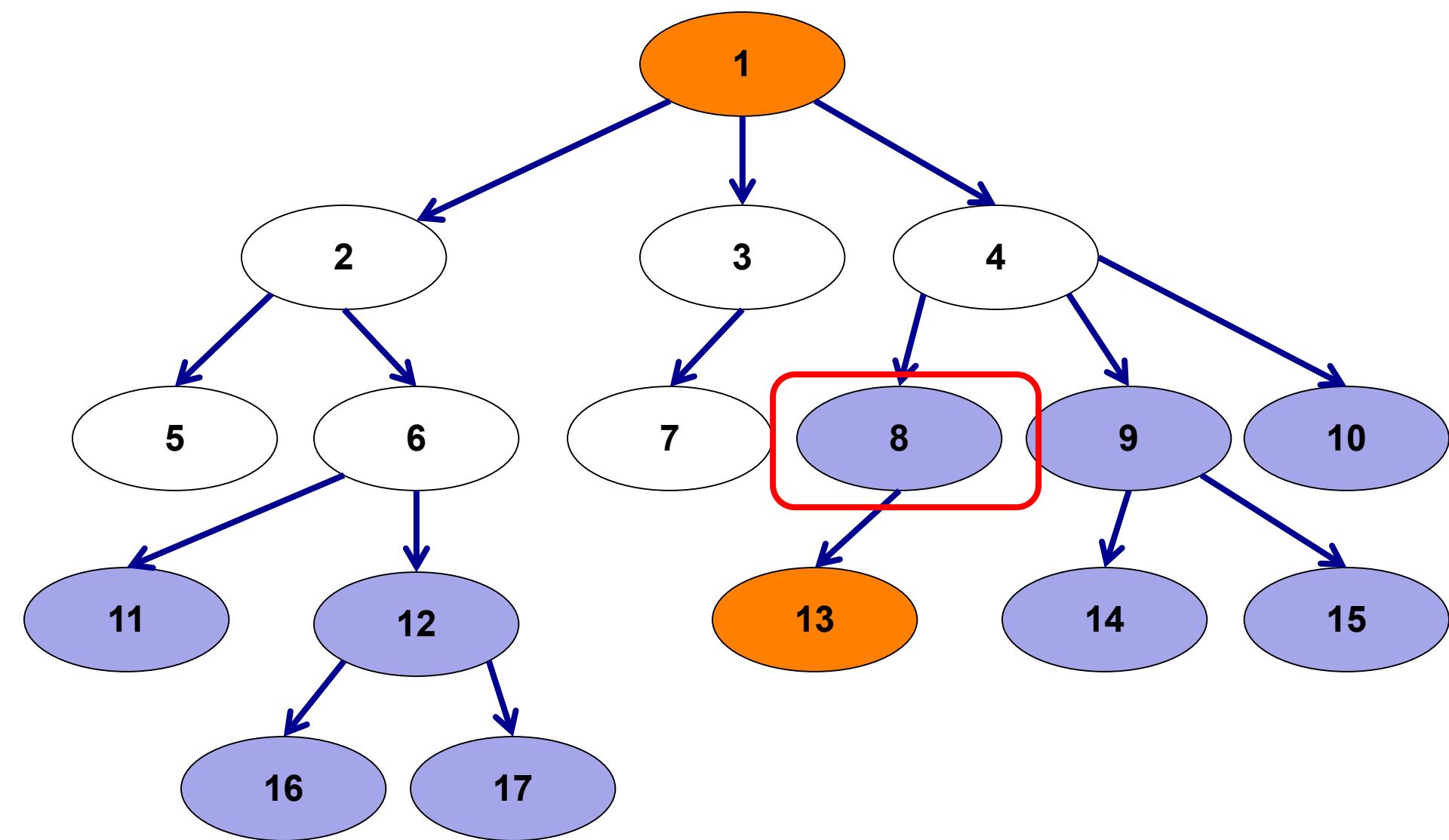


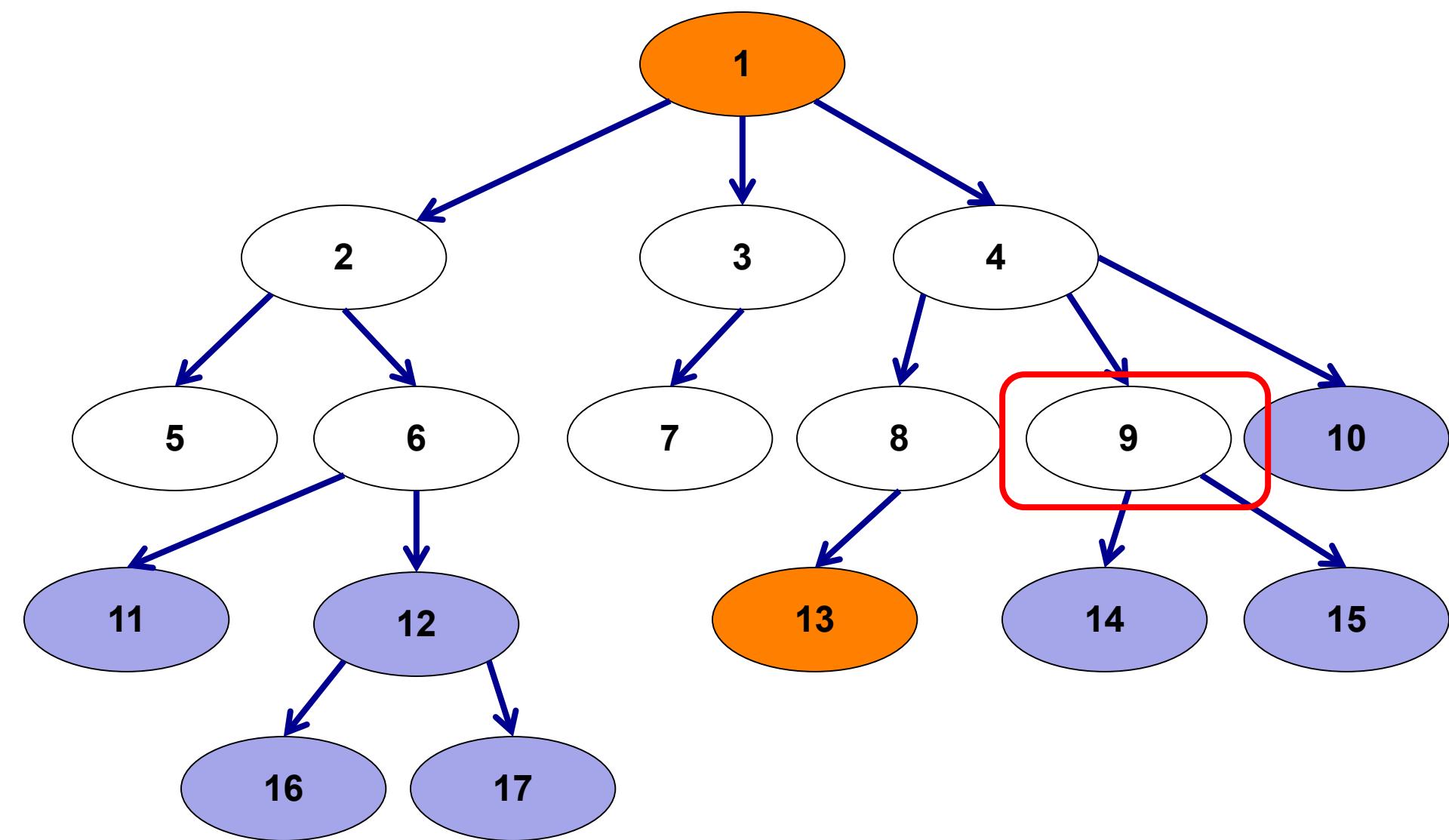


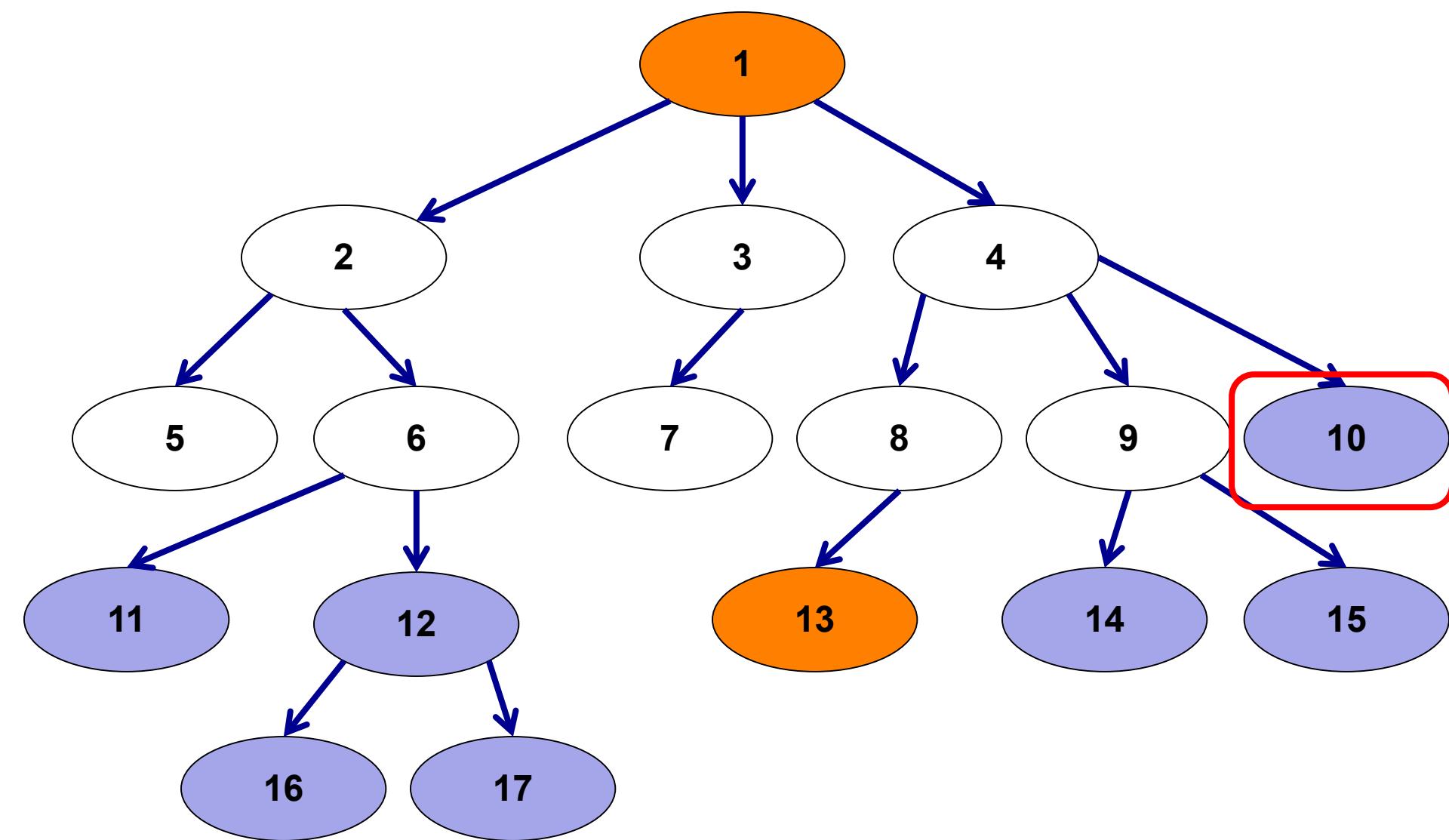


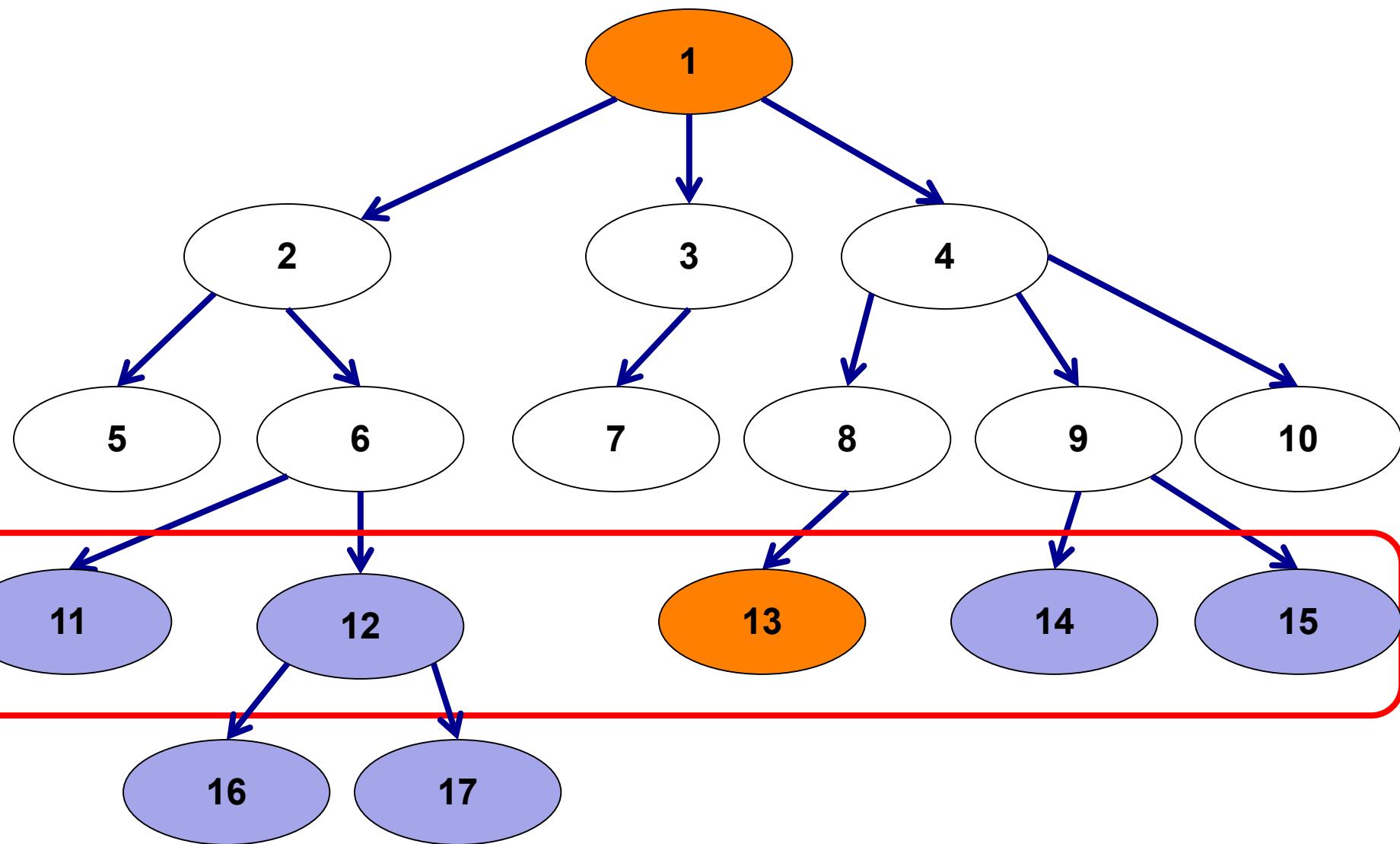


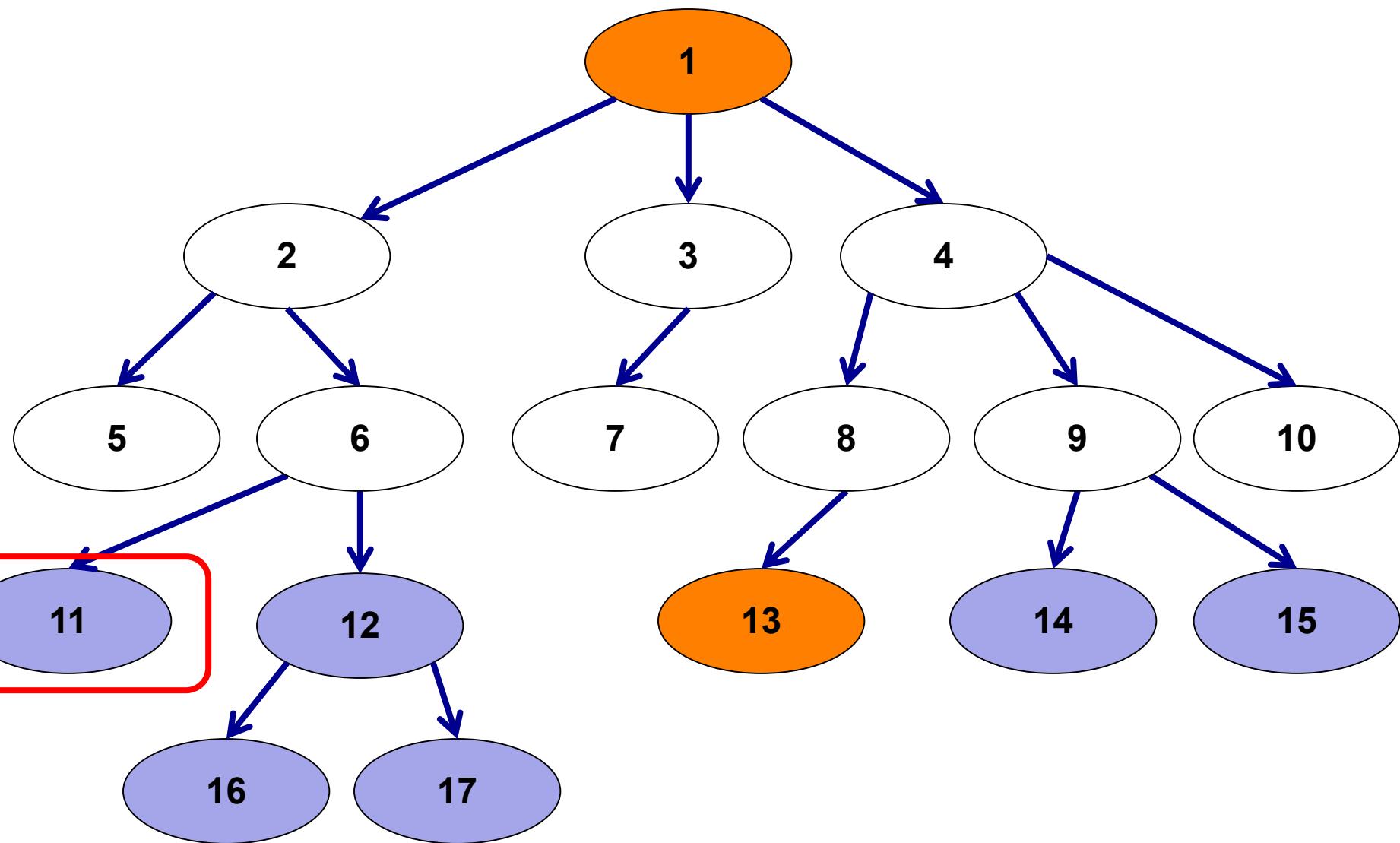


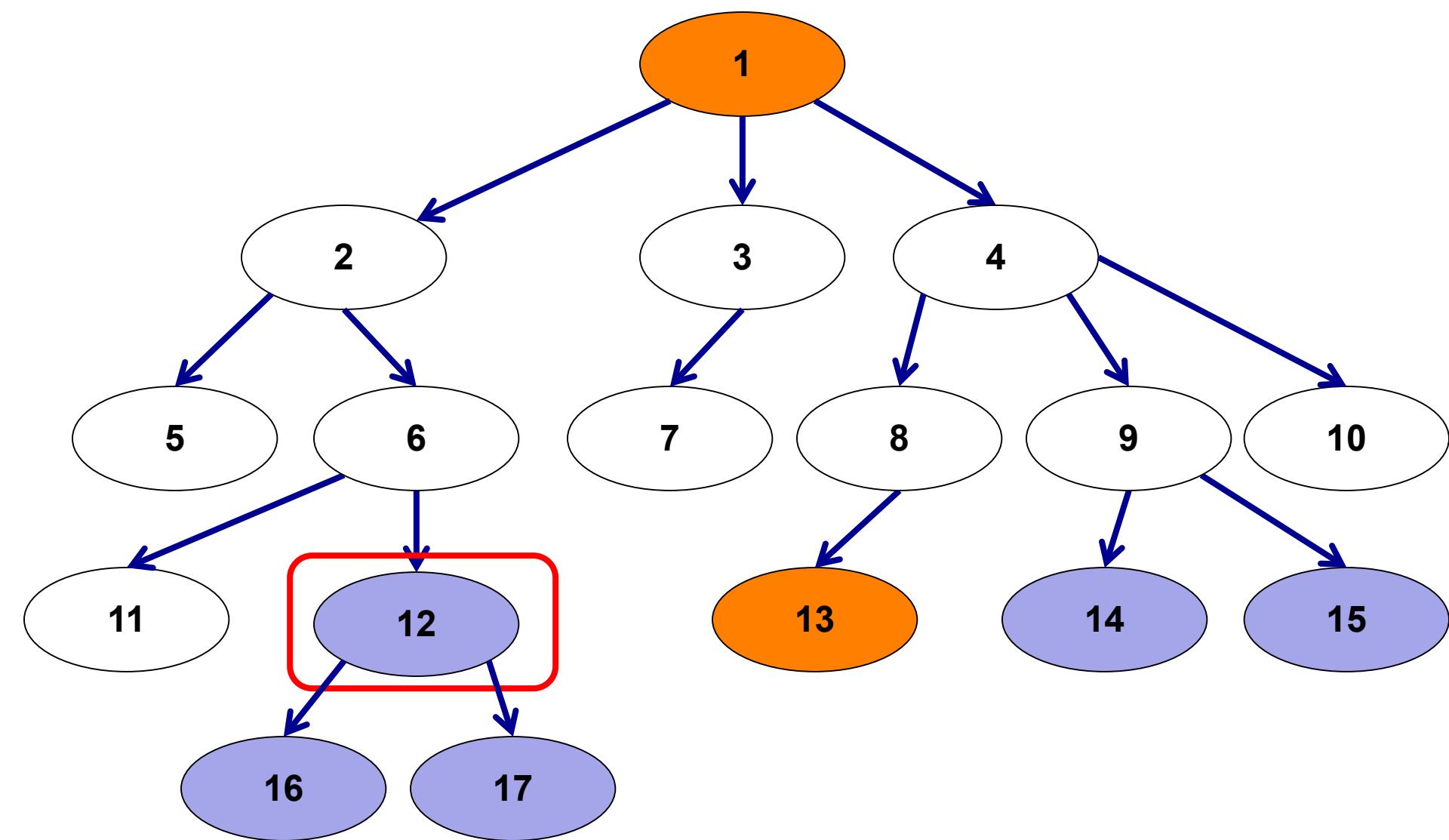


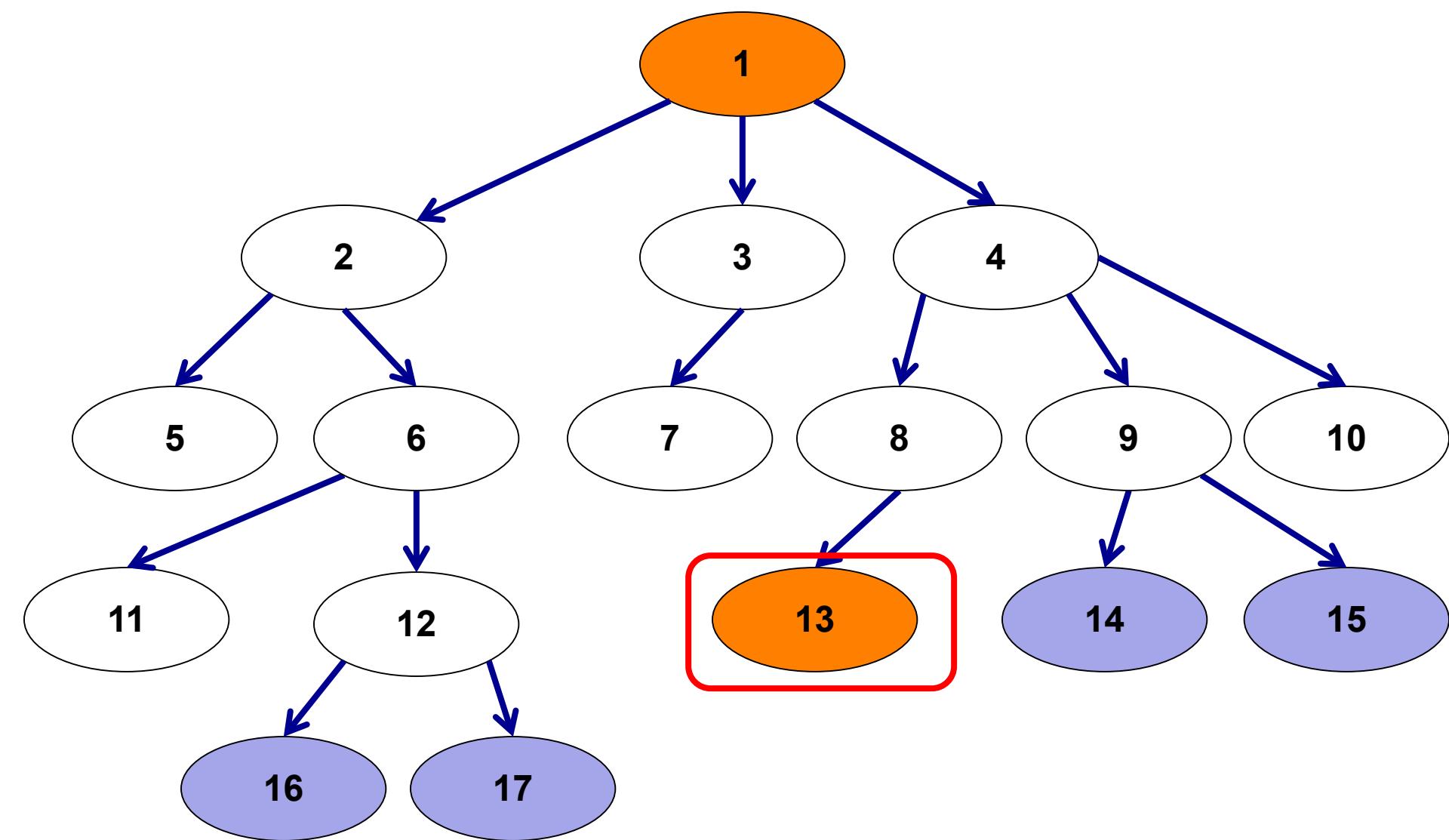


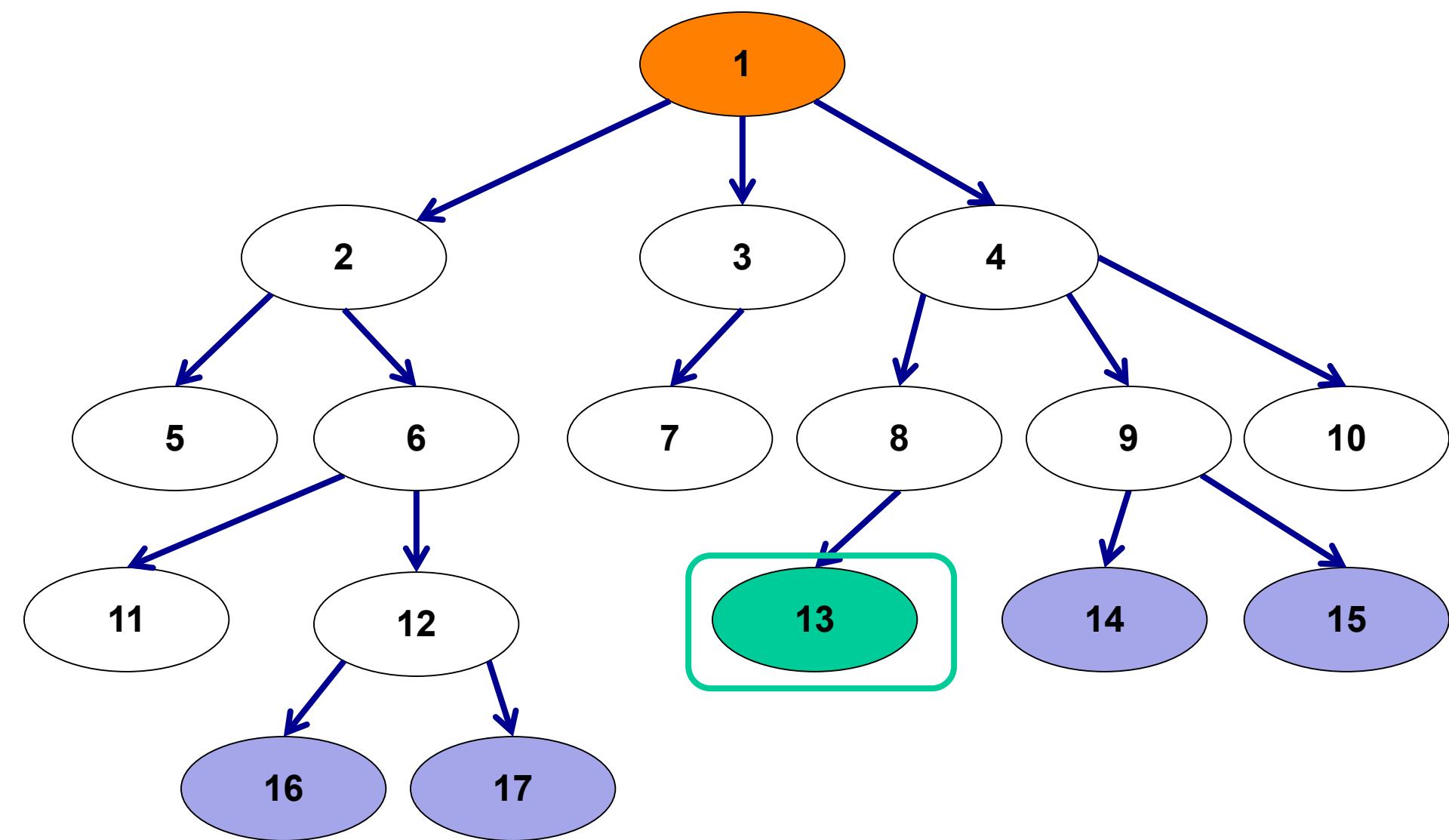








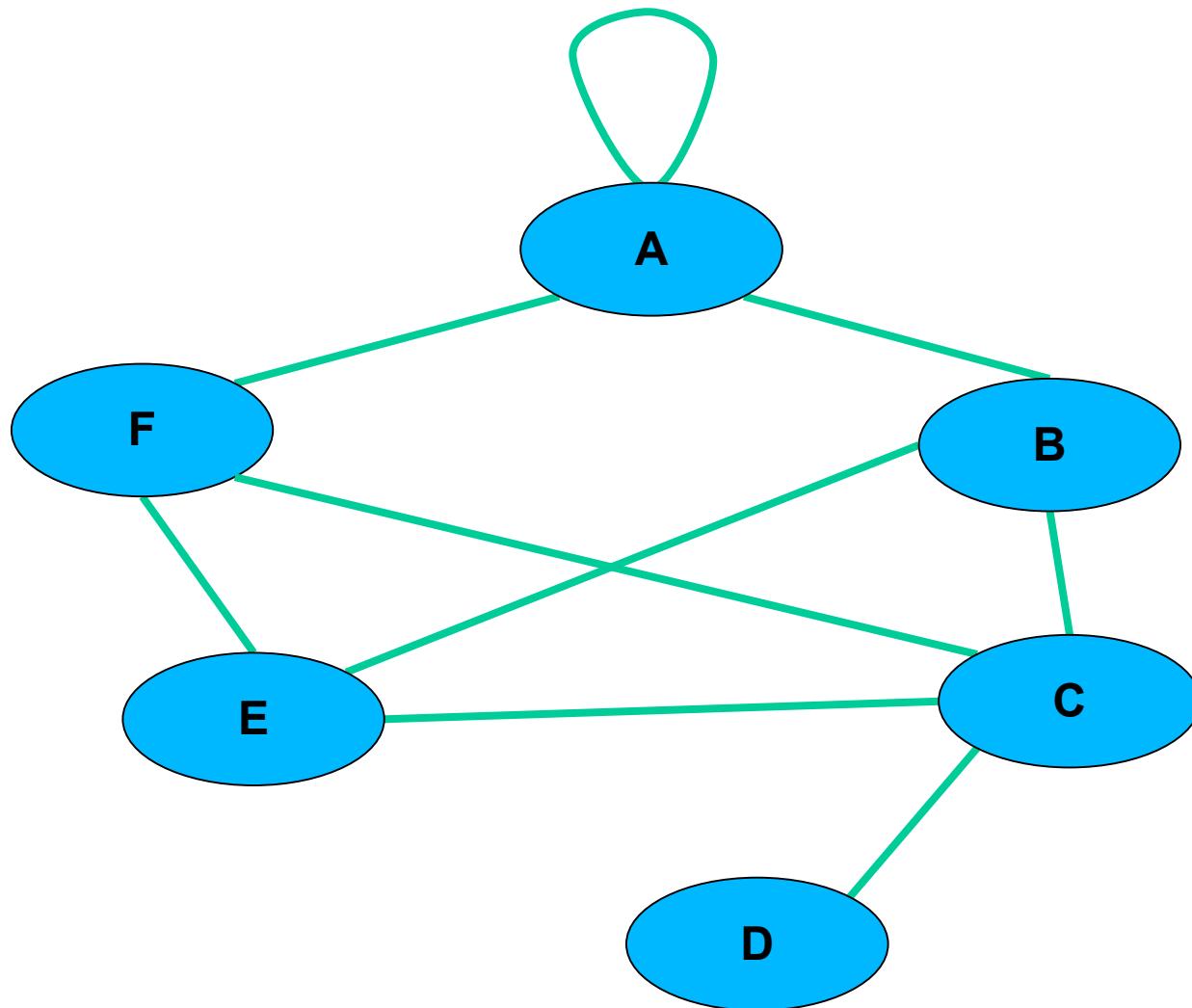




# Breadth-first search

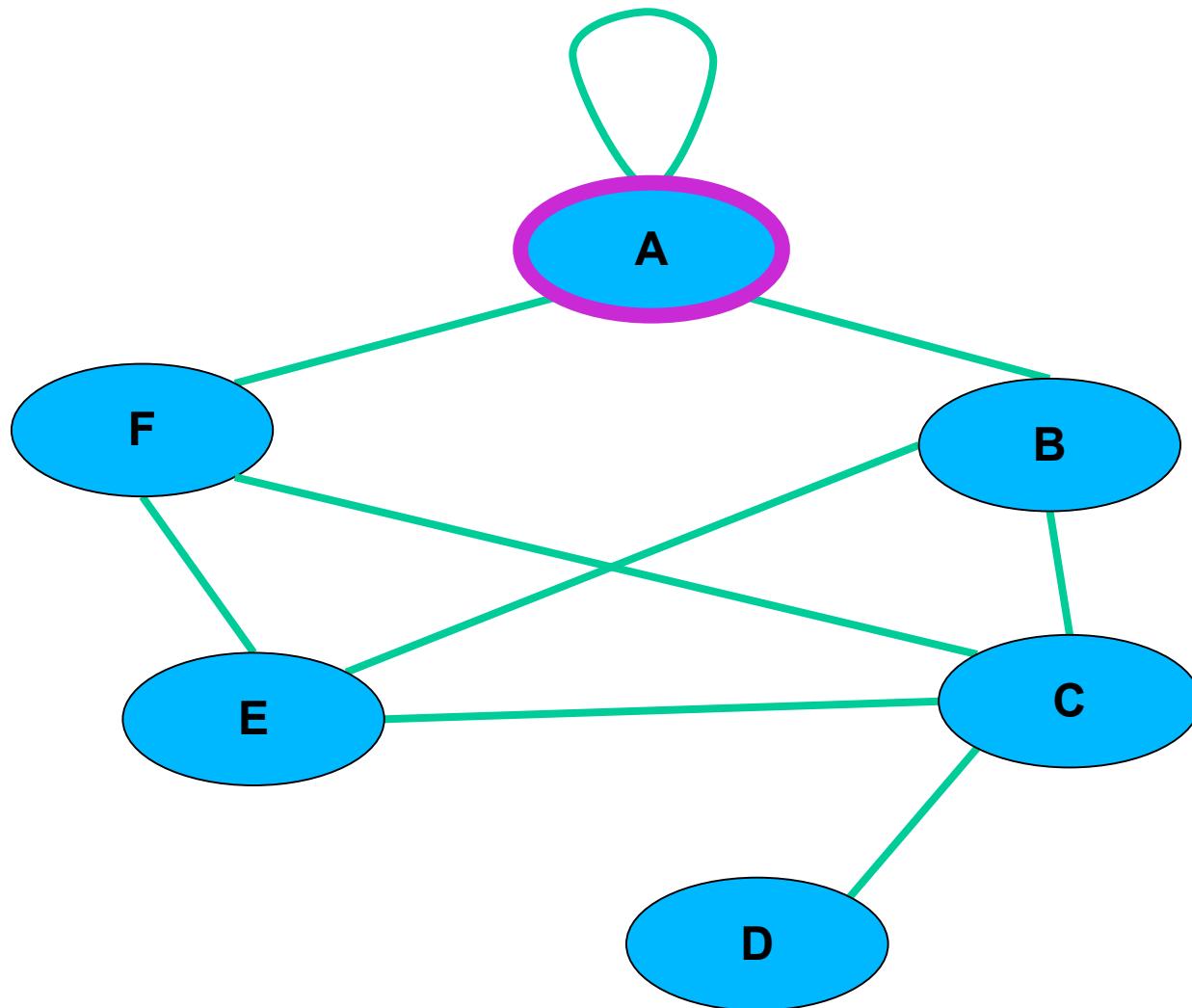
# Breadth-first search (BFS)

---



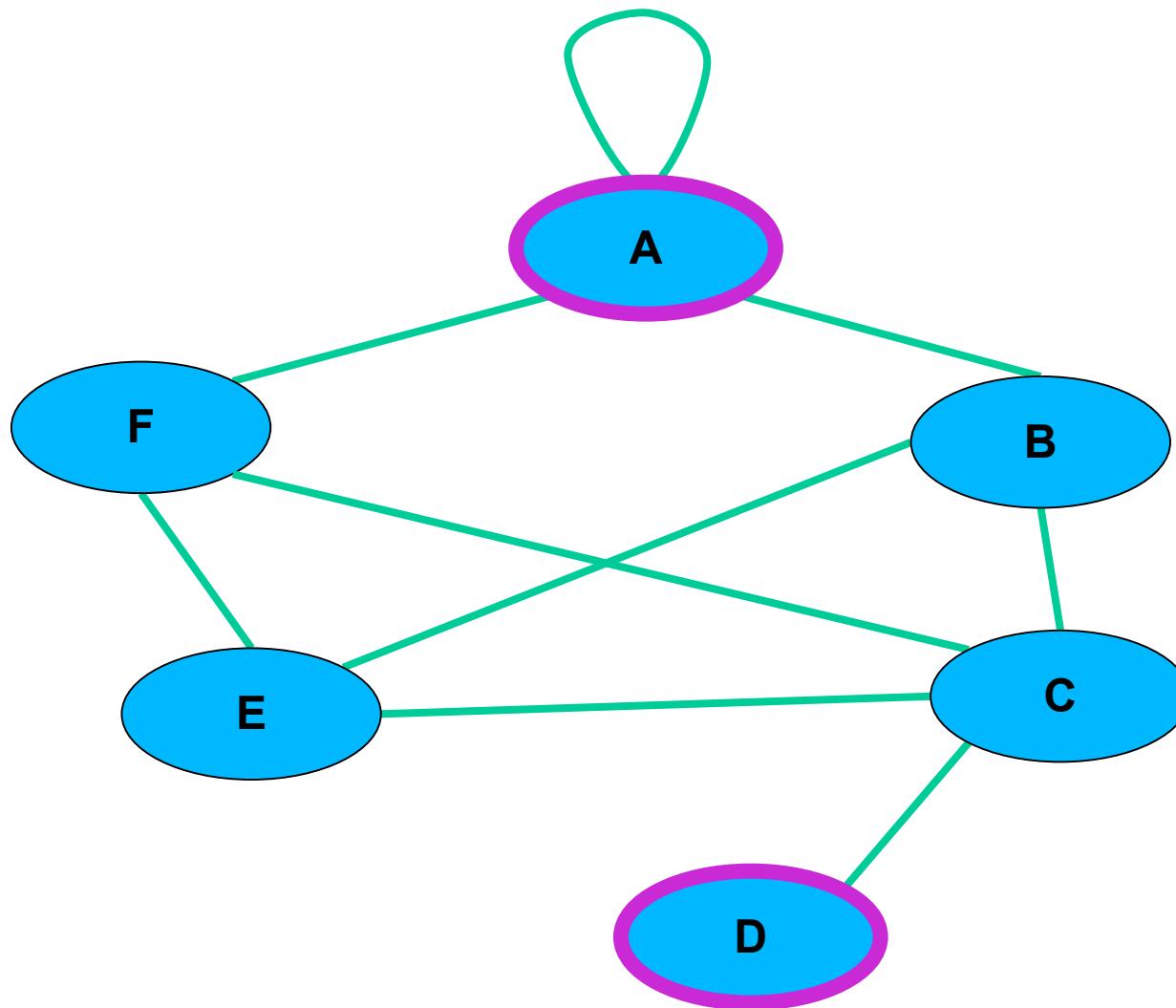
# Breadth-first search (BFS)

---



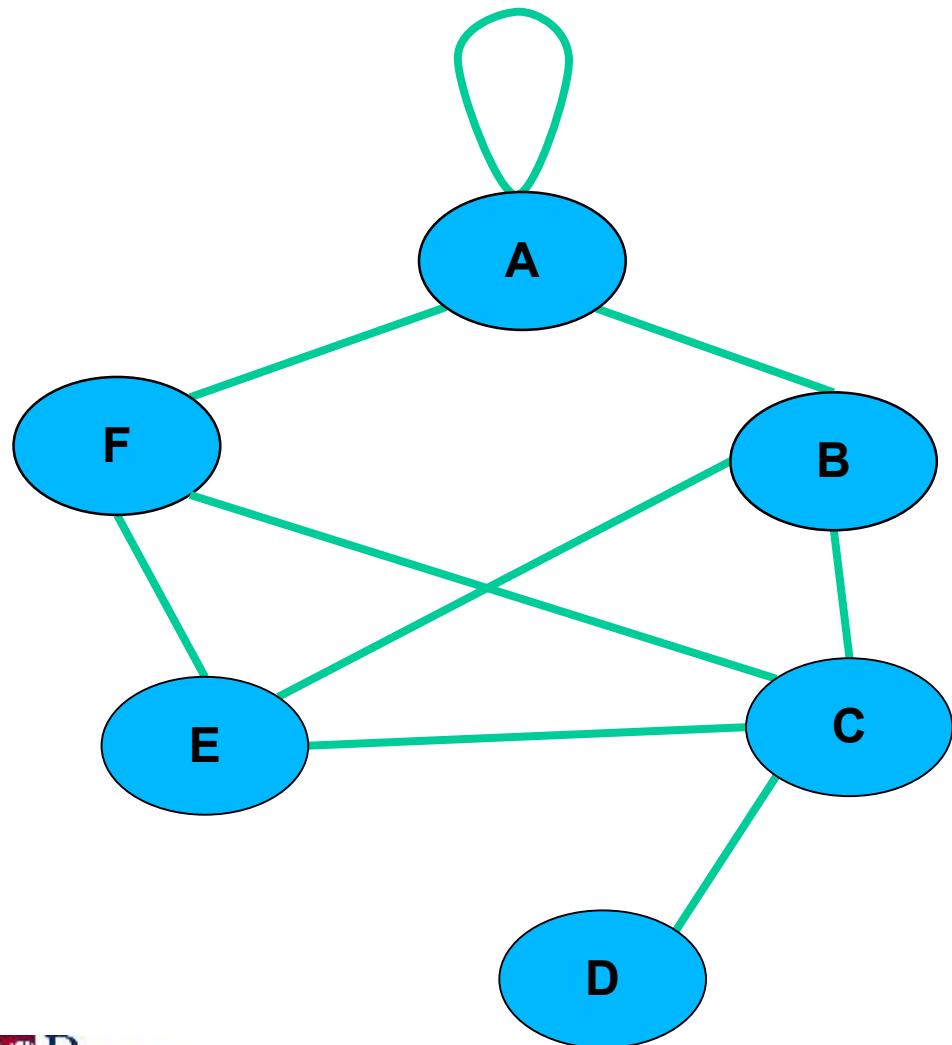
# Breadth-first search (BFS)

---



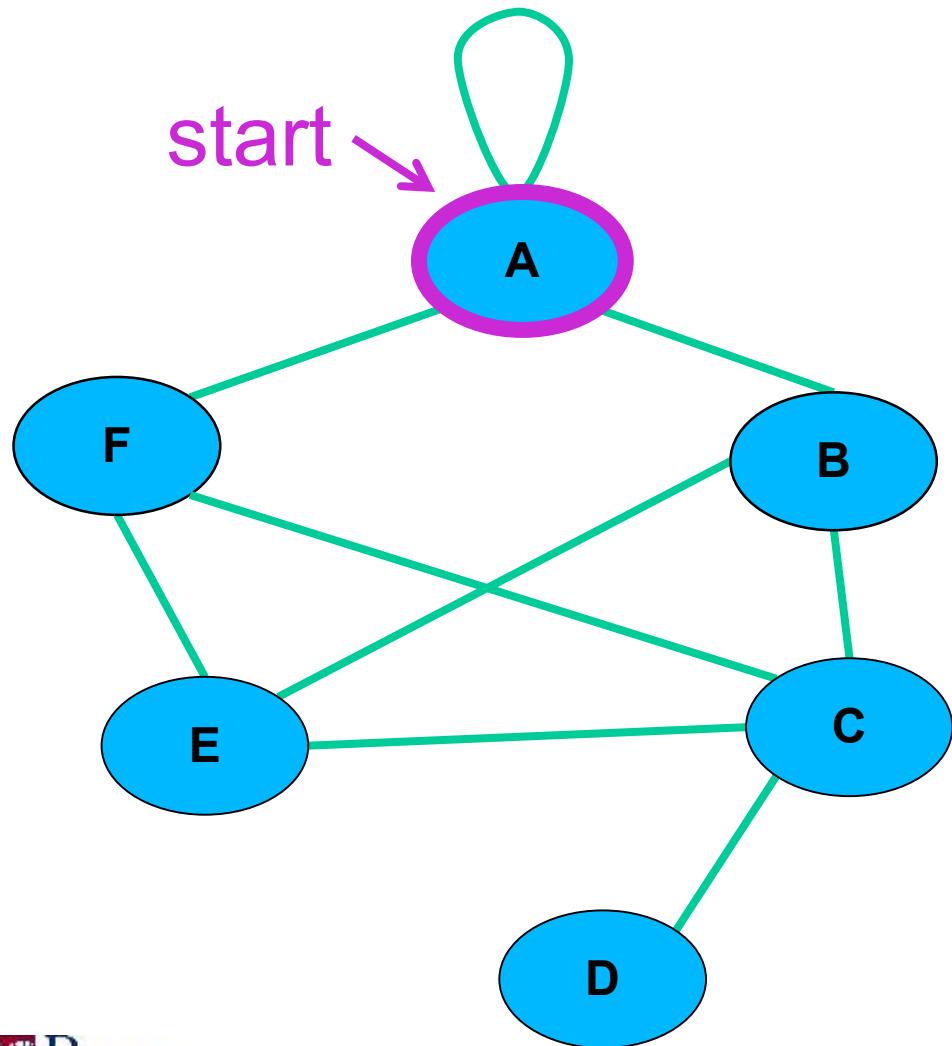
# Breadth-first search (BFS)

---

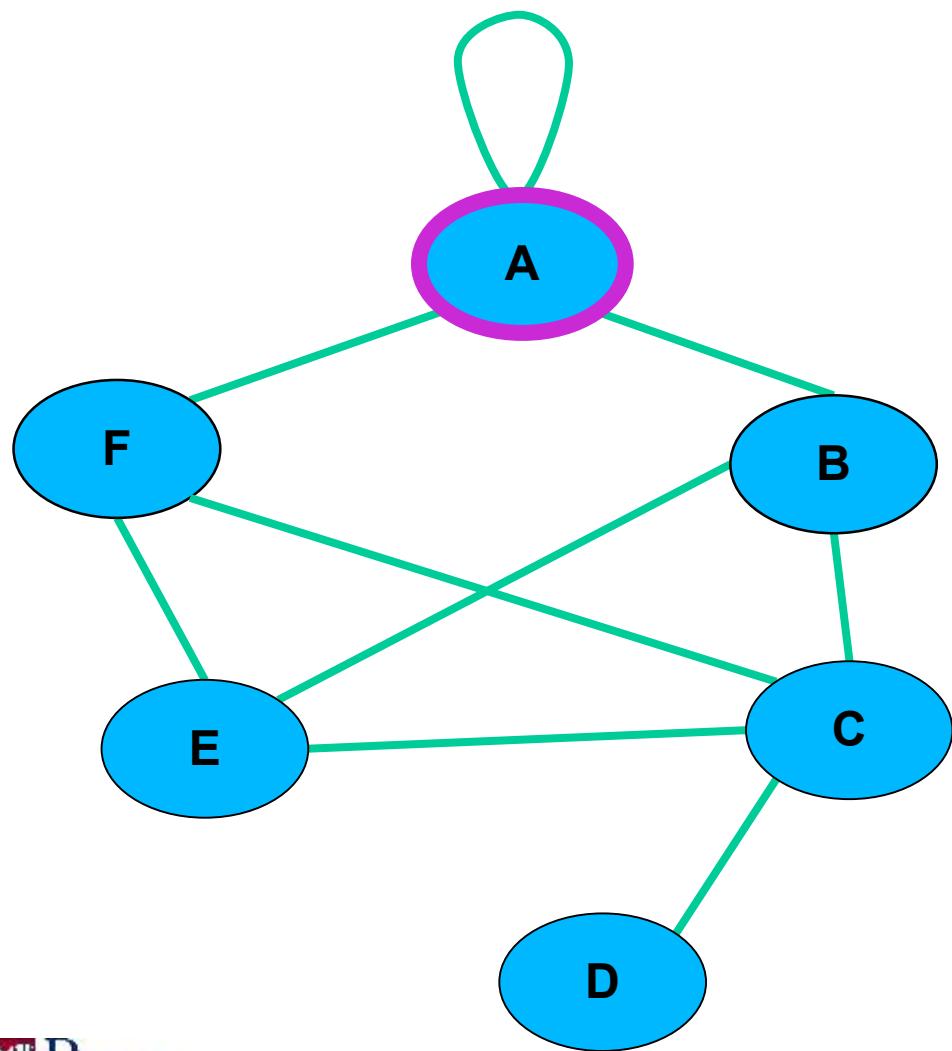


# Breadth-first search (BFS)

---



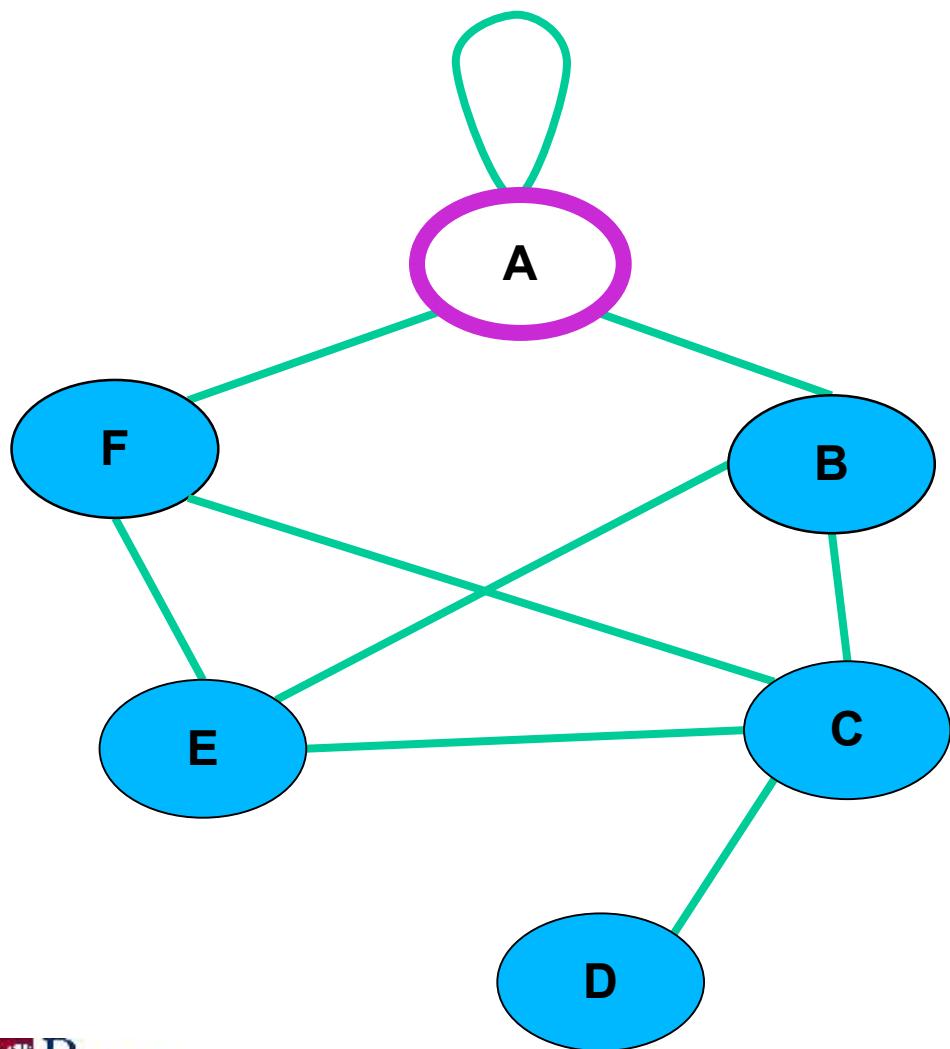
# Breadth-first search (BFS)



nodes marked:

A

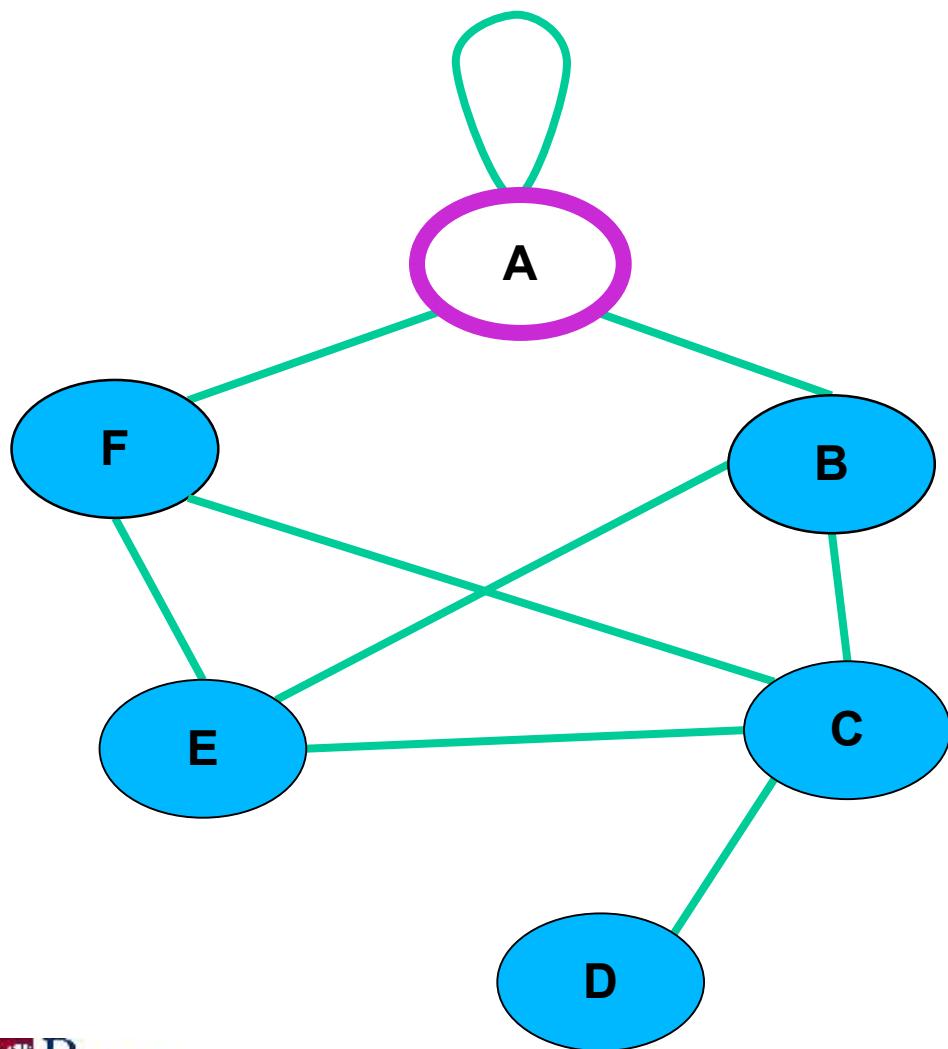
# Breadth-first search (BFS)



nodes marked:

A

# Breadth-first search (BFS)



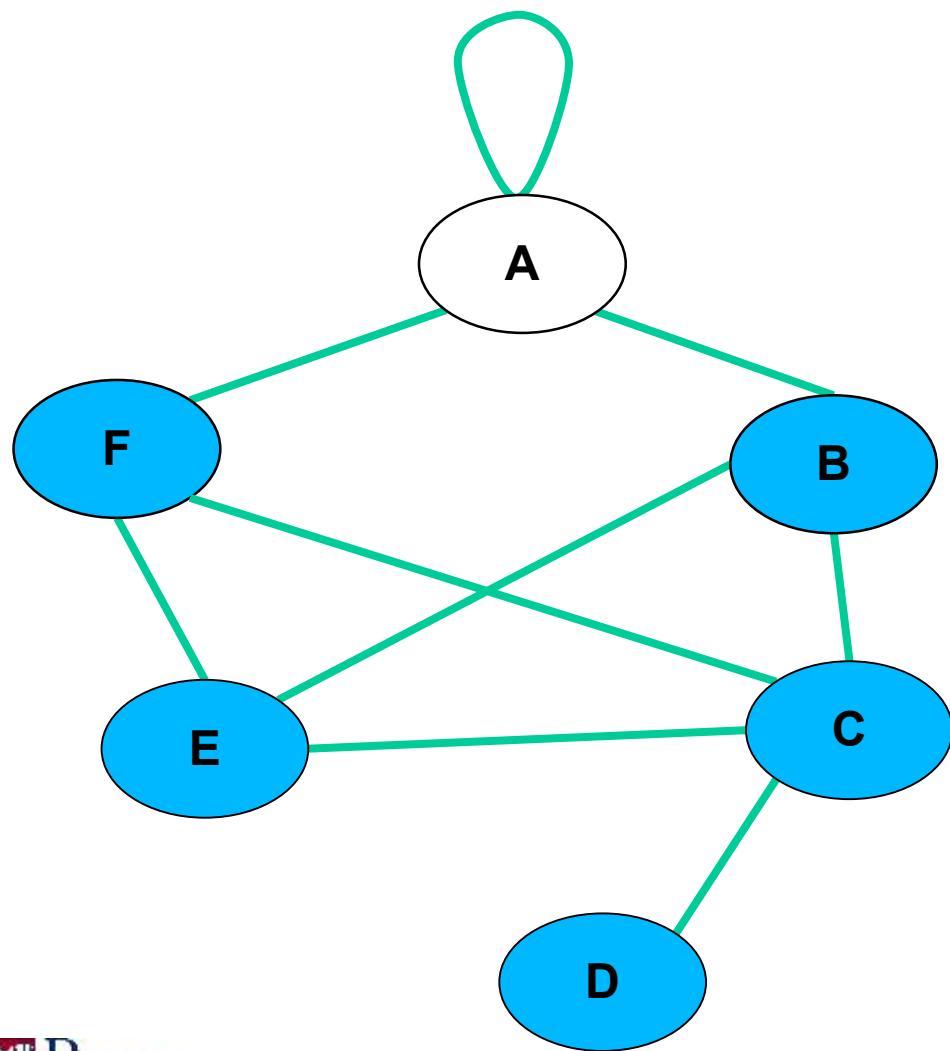
nodes marked:

A

queue of nodes to explore:

A

# Breadth-first search (BFS)



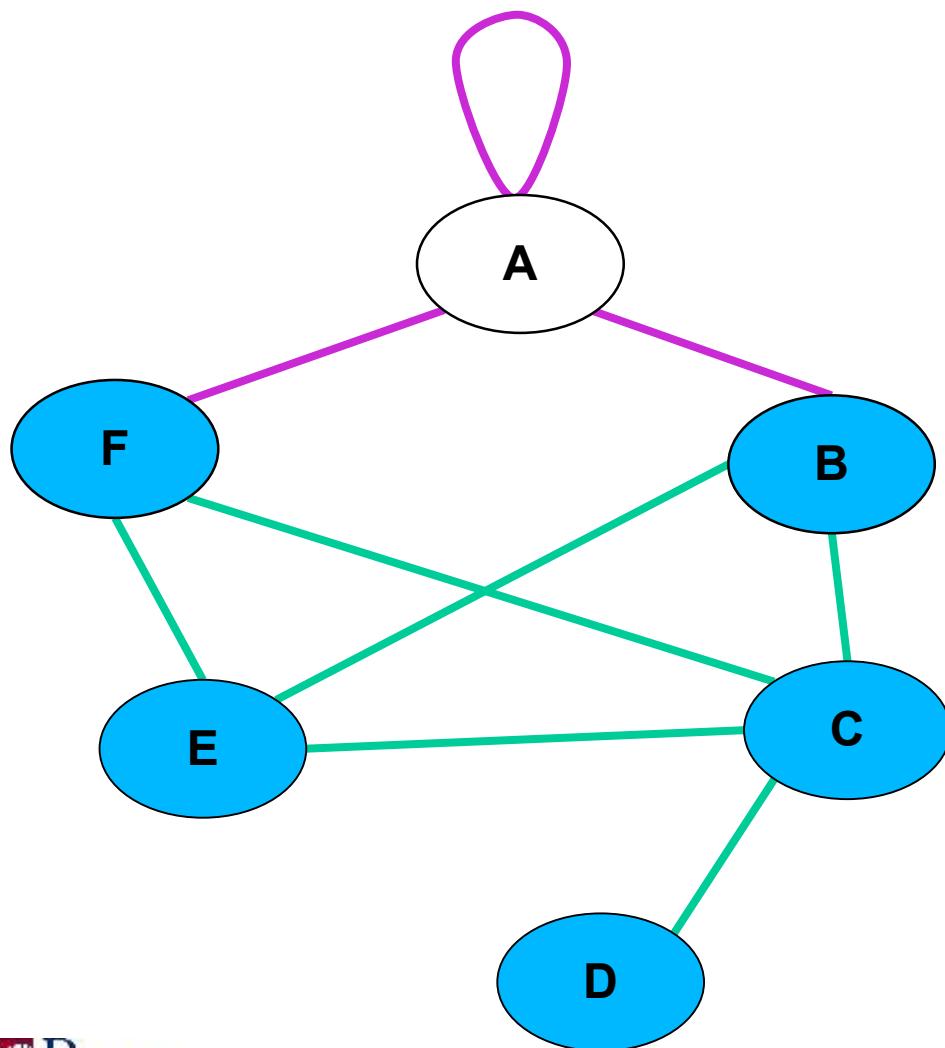
nodes marked:

A

queue of nodes to explore:

A

# Breadth-first search (BFS)



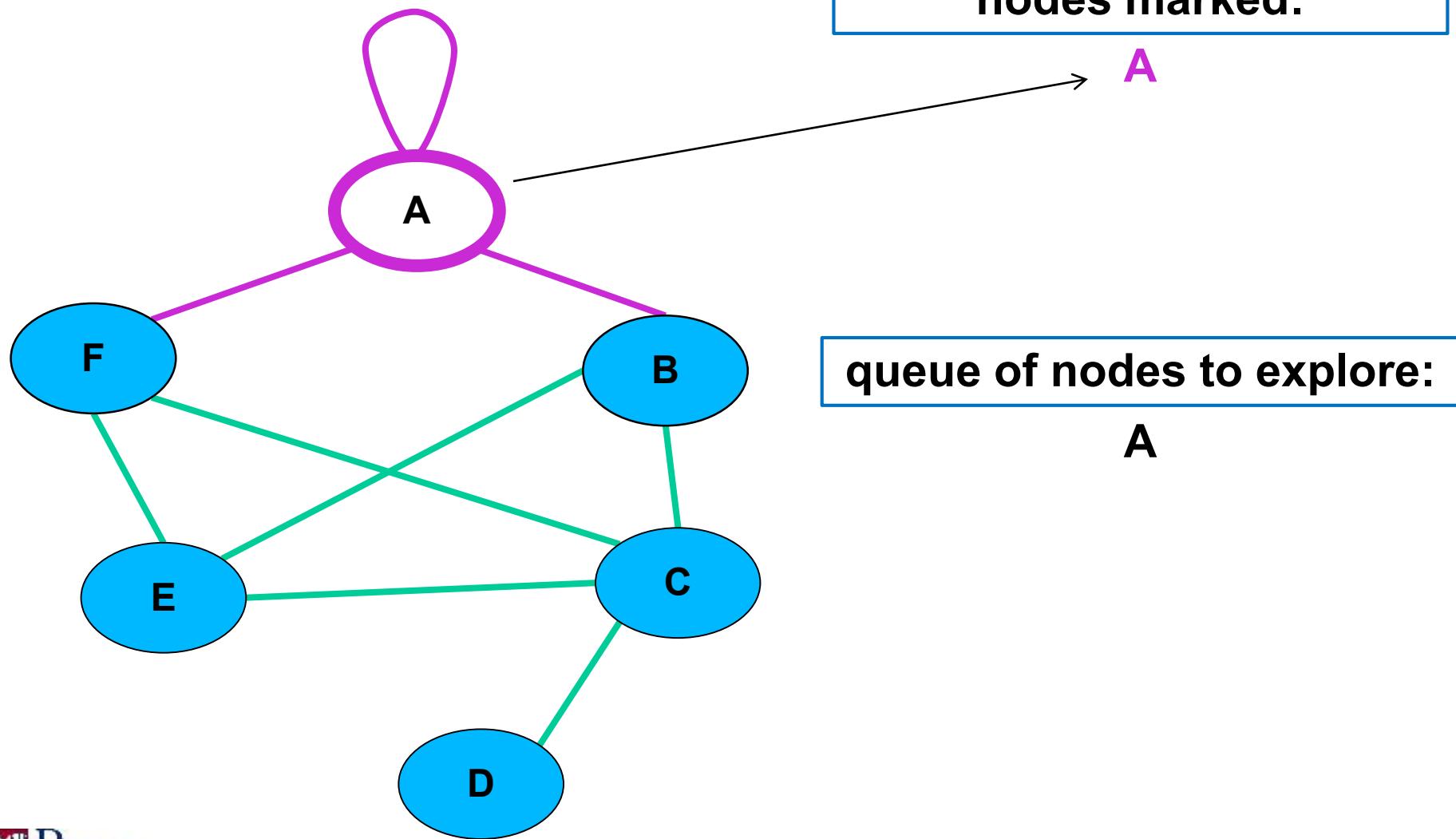
nodes marked:

A

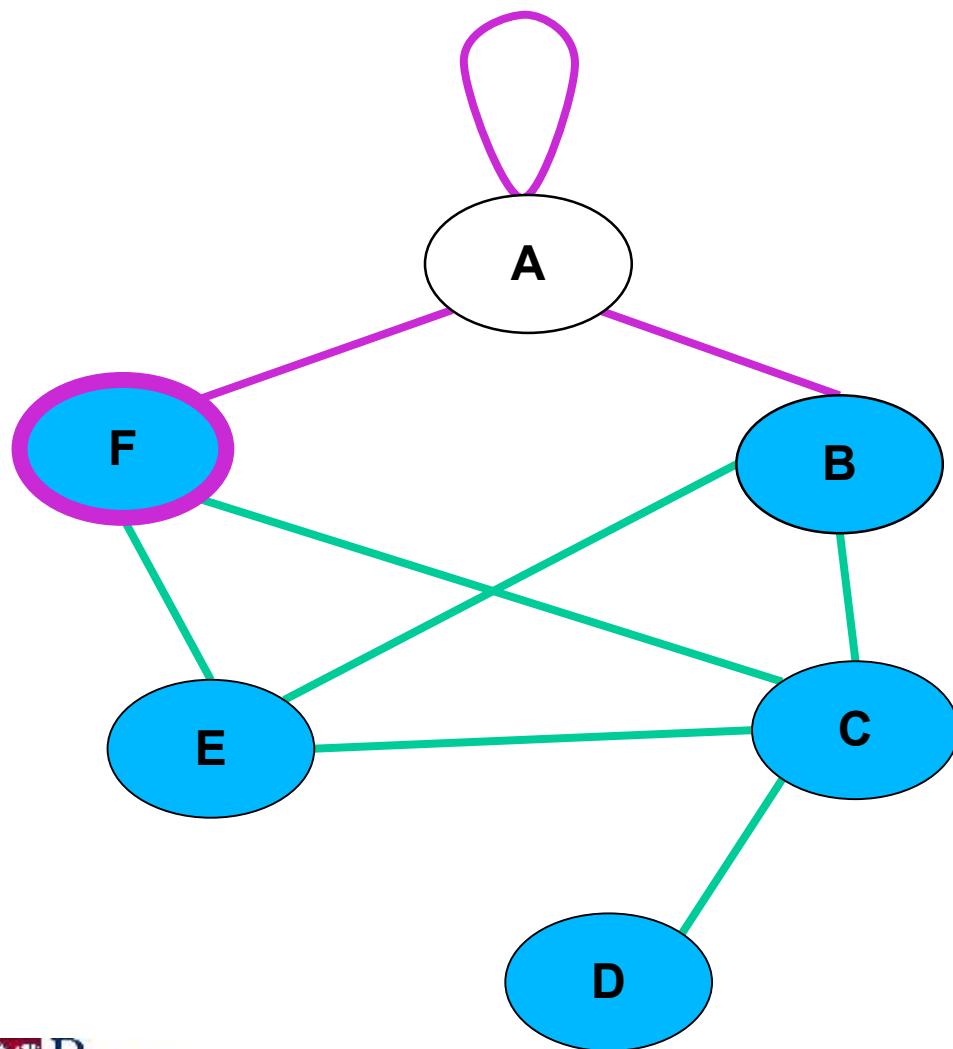
queue of nodes to explore:

A

# Breadth-first search (BFS)



# Breadth-first search (BFS)



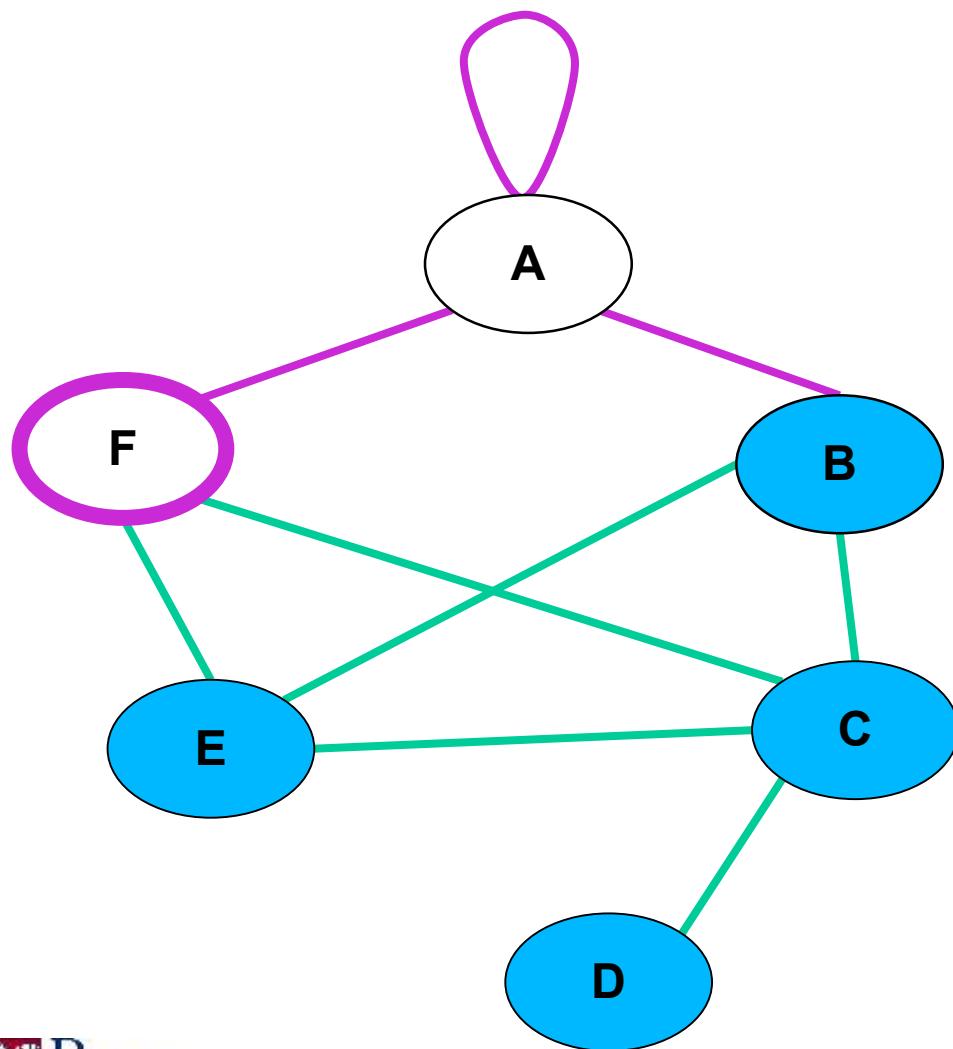
nodes marked:

A  
F

queue of nodes to explore:

A

# Breadth-first search (BFS)



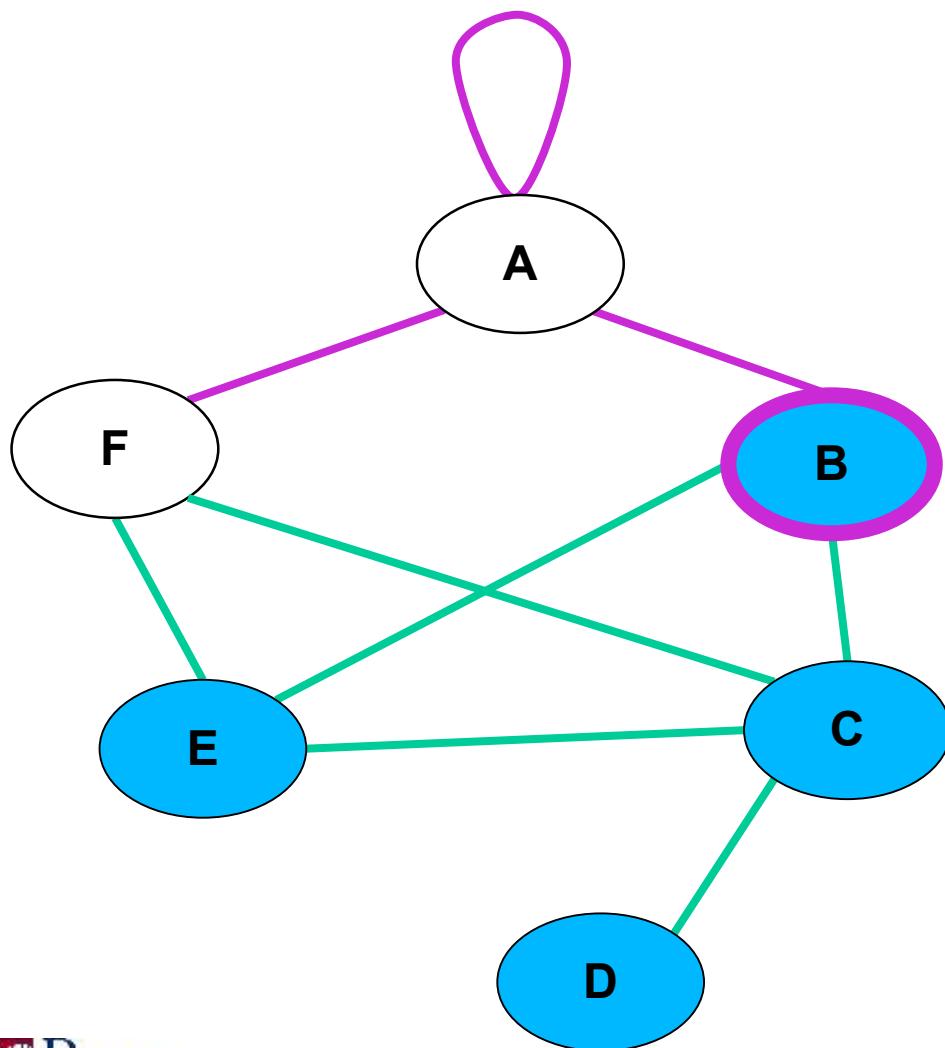
nodes marked:

A  
F

queue of nodes to explore:

A  
F

# Breadth-first search (BFS)



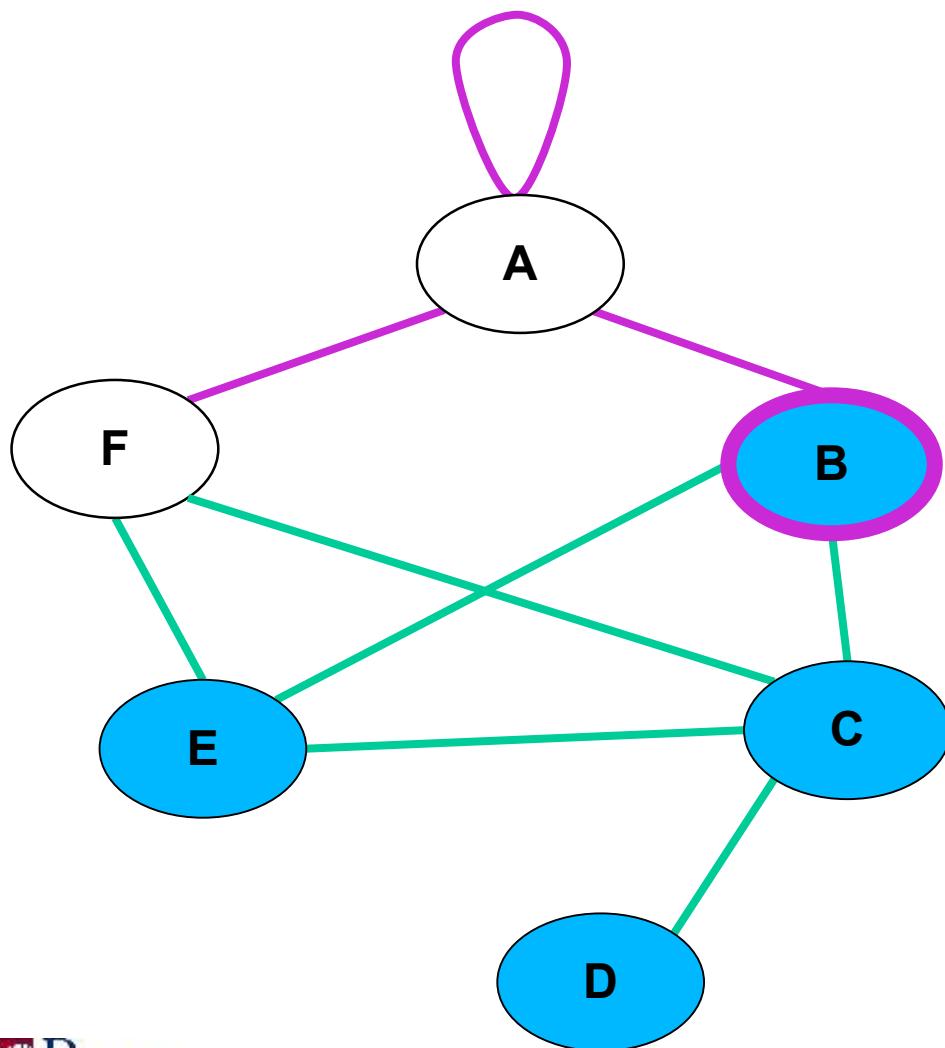
nodes marked:

A  
F

queue of nodes to explore:

A  
F

# Breadth-first search (BFS)



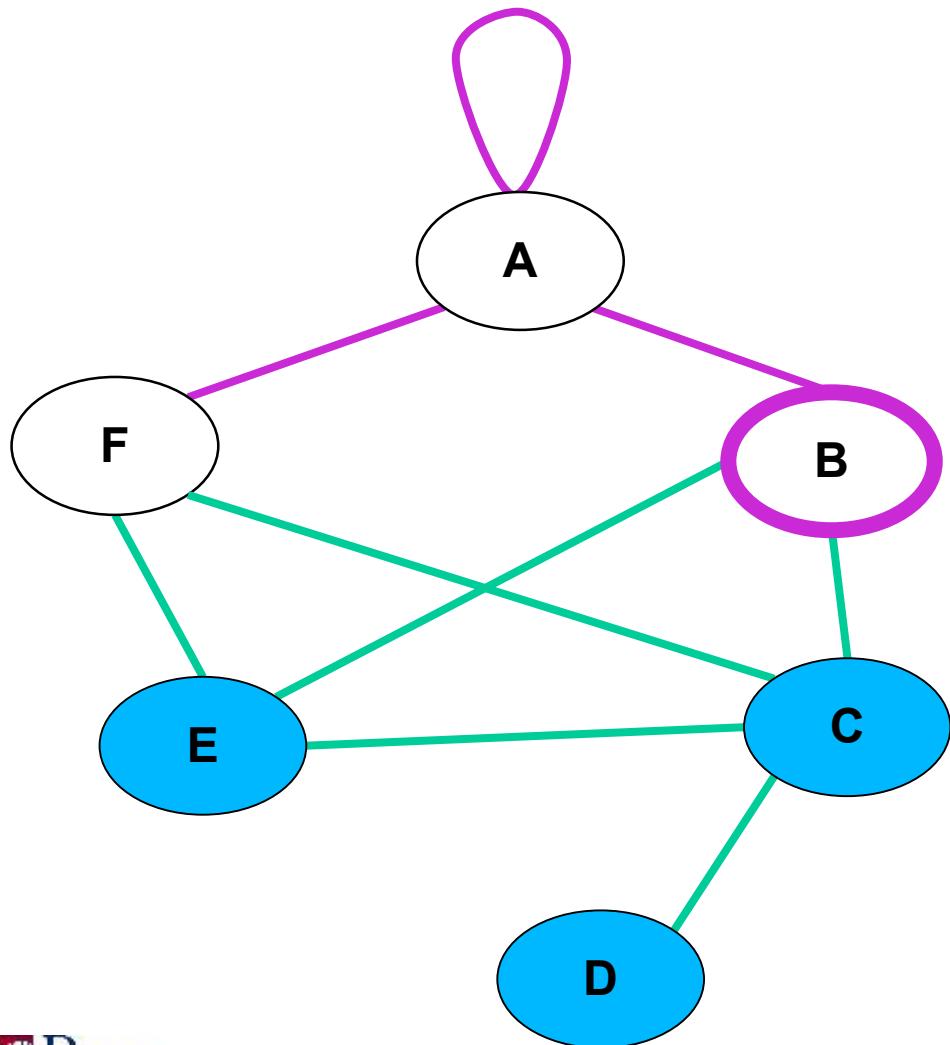
nodes marked:

A  
F  
B

queue of nodes to explore:

A  
F

# Breadth-first search (BFS)



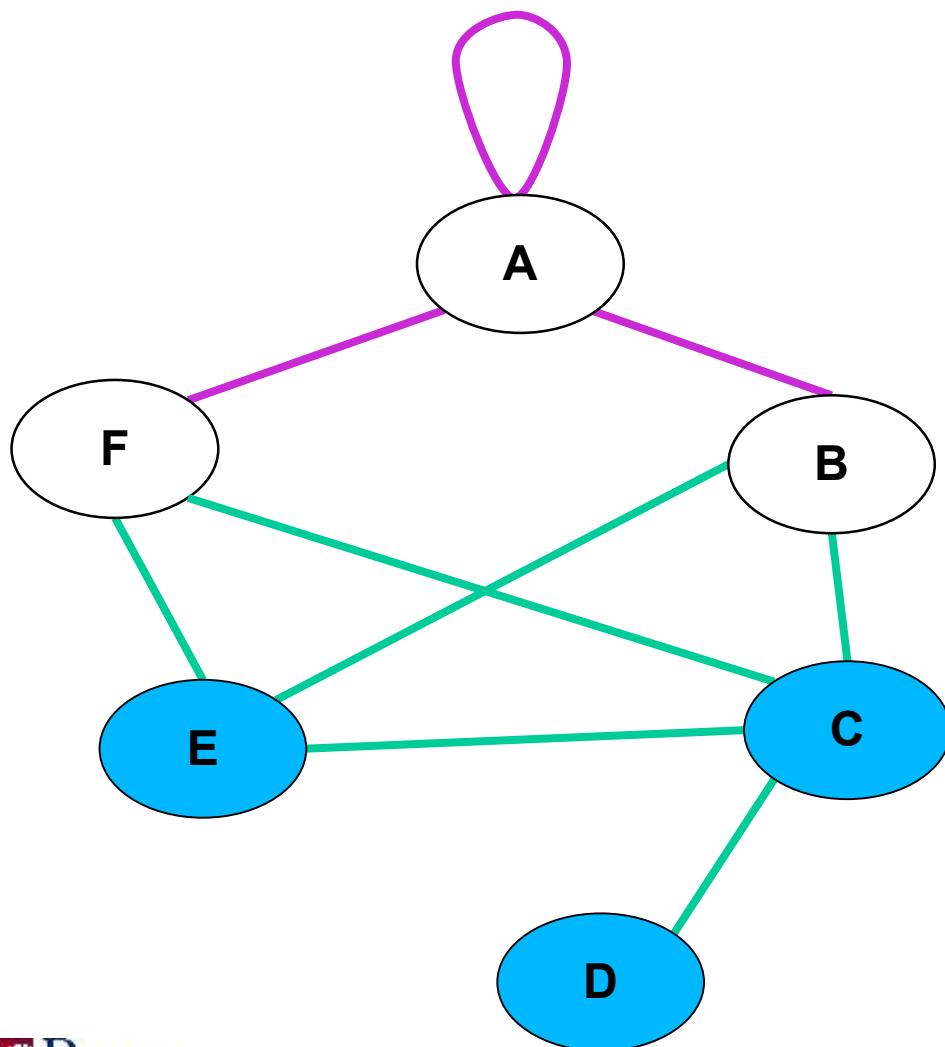
nodes marked:

A  
F  
B

queue of nodes to explore:

A  
F  
B

# Breadth-first search (BFS)



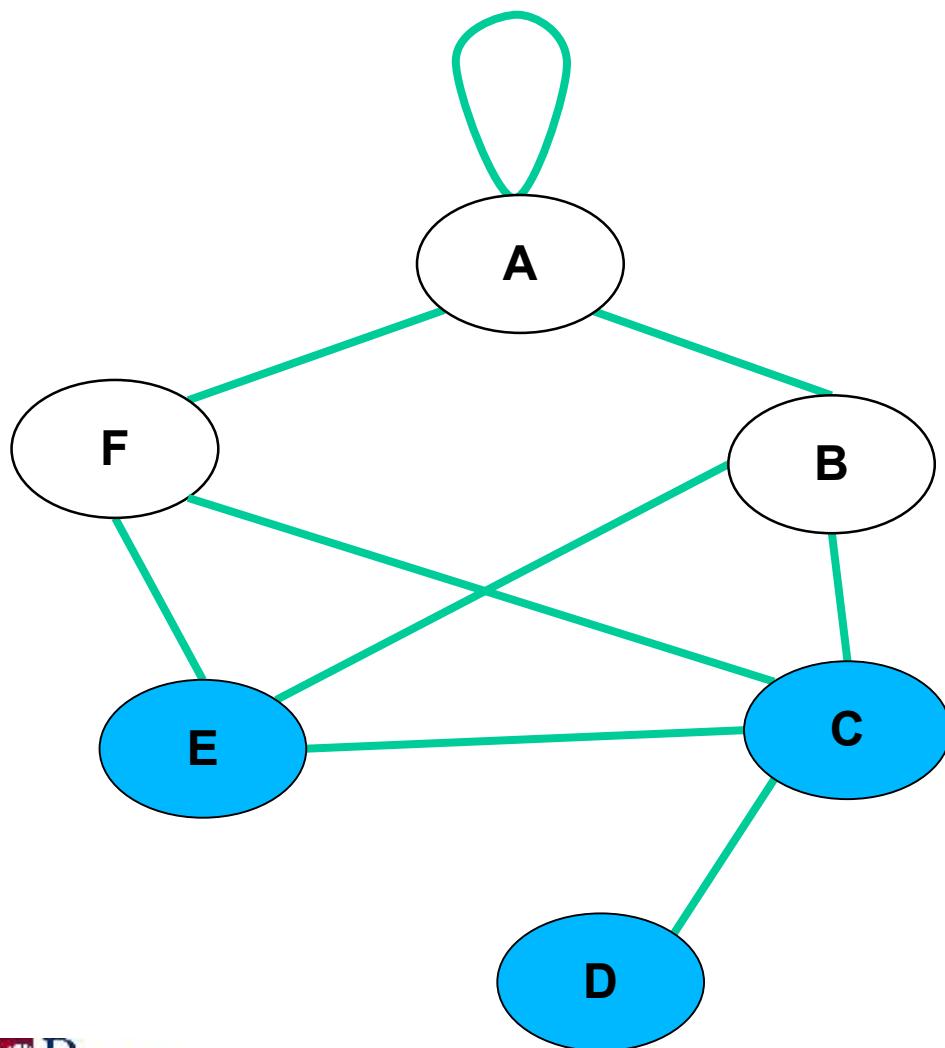
nodes marked:

A  
F  
B

queue of nodes to explore:

A  
F  
B

# Breadth-first search (BFS)



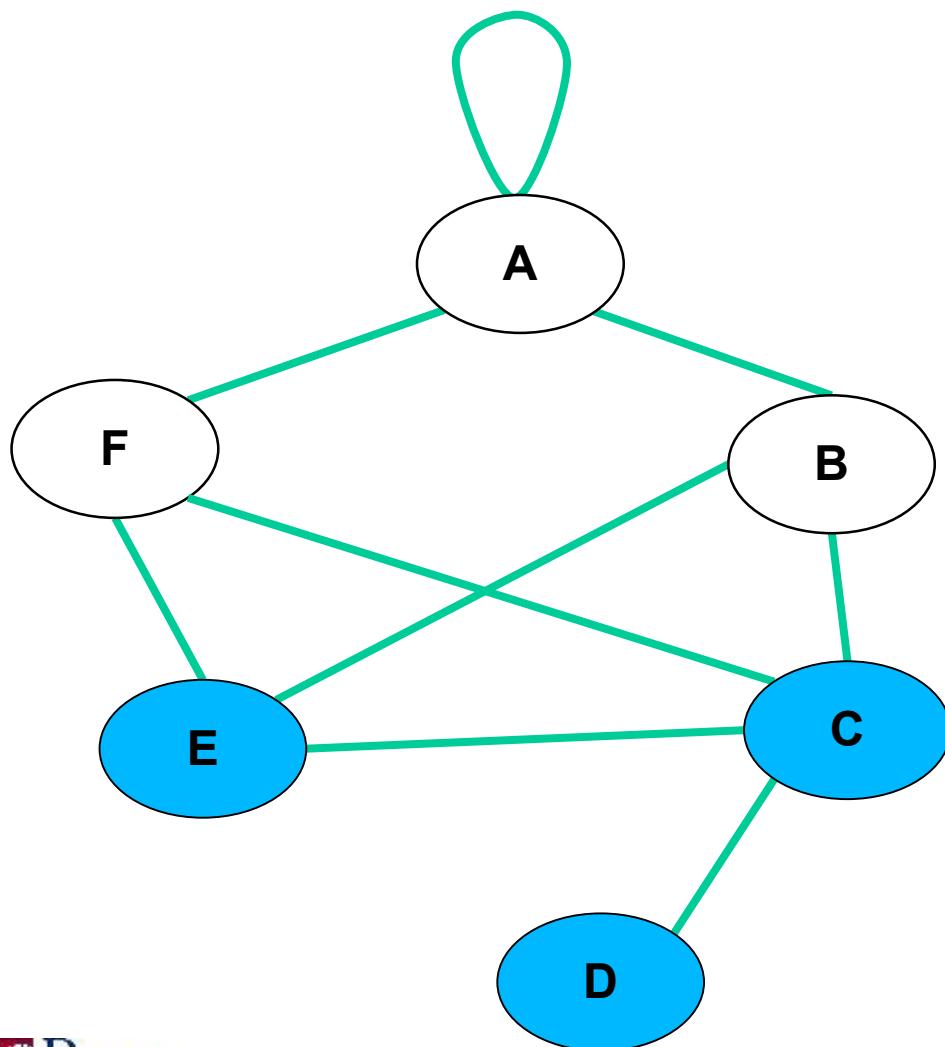
nodes marked:

A  
F  
B

queue of nodes to explore:

A  
F  
B

# Breadth-first search (BFS)



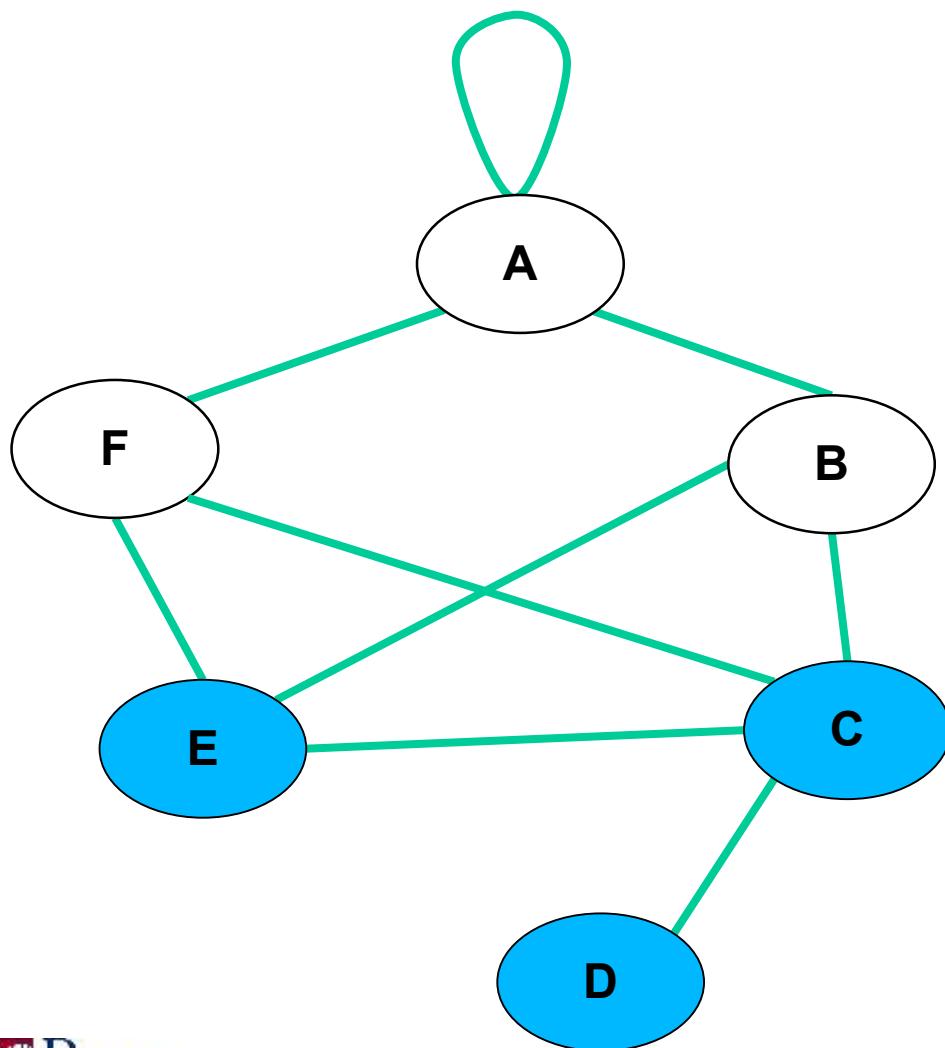
nodes marked:

A  
F  
B

queue of nodes to explore:

A  
F  
B

# Breadth-first search (BFS)



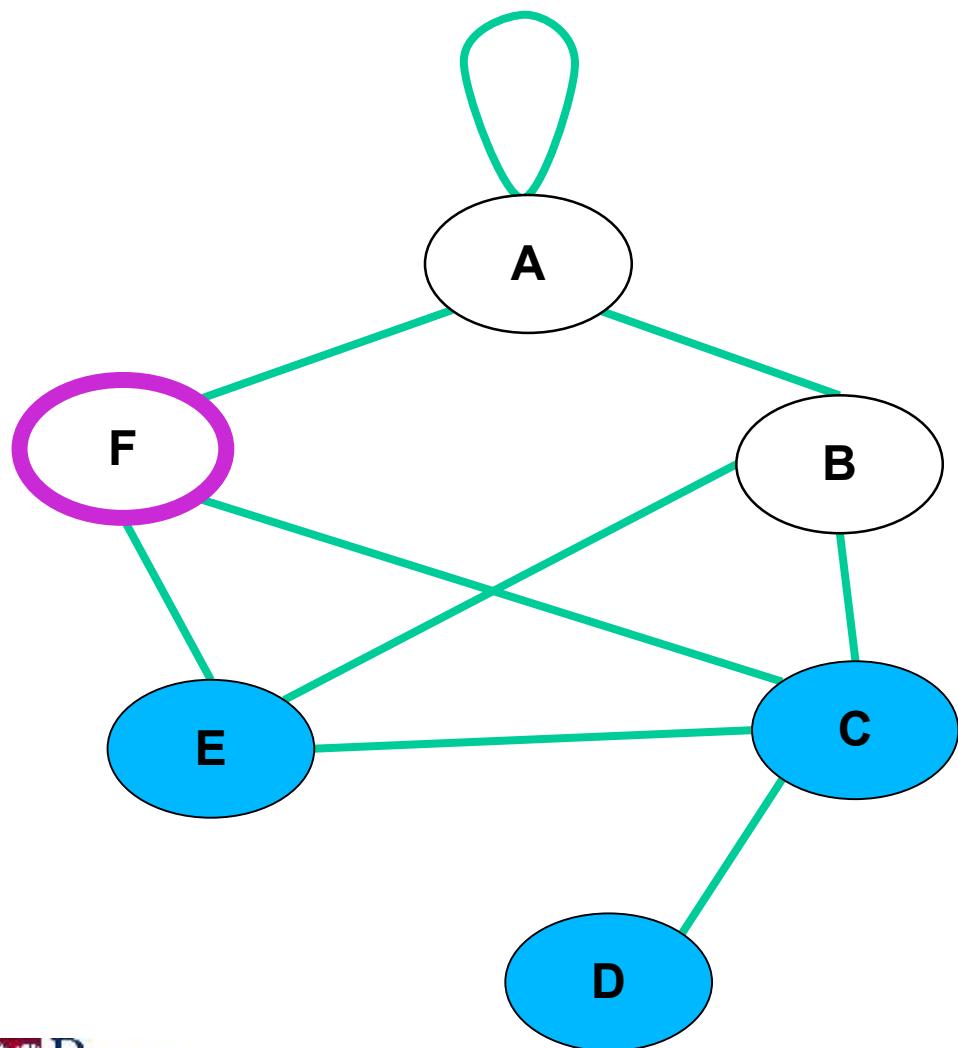
nodes marked:

A  
F  
B

queue of nodes to explore:

F  
B

# Breadth-first search (BFS)



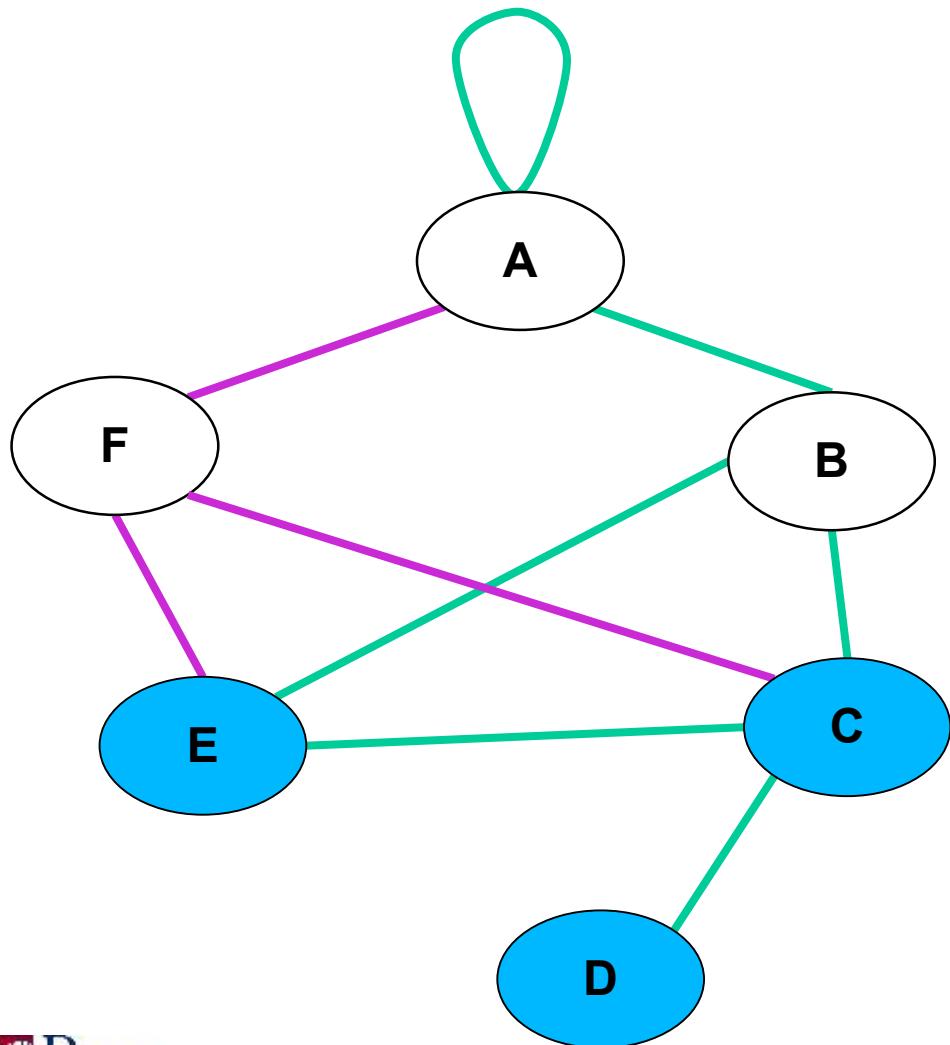
nodes marked:

A  
F  
B

queue of nodes to explore:

F  
B

# Breadth-first search (BFS)



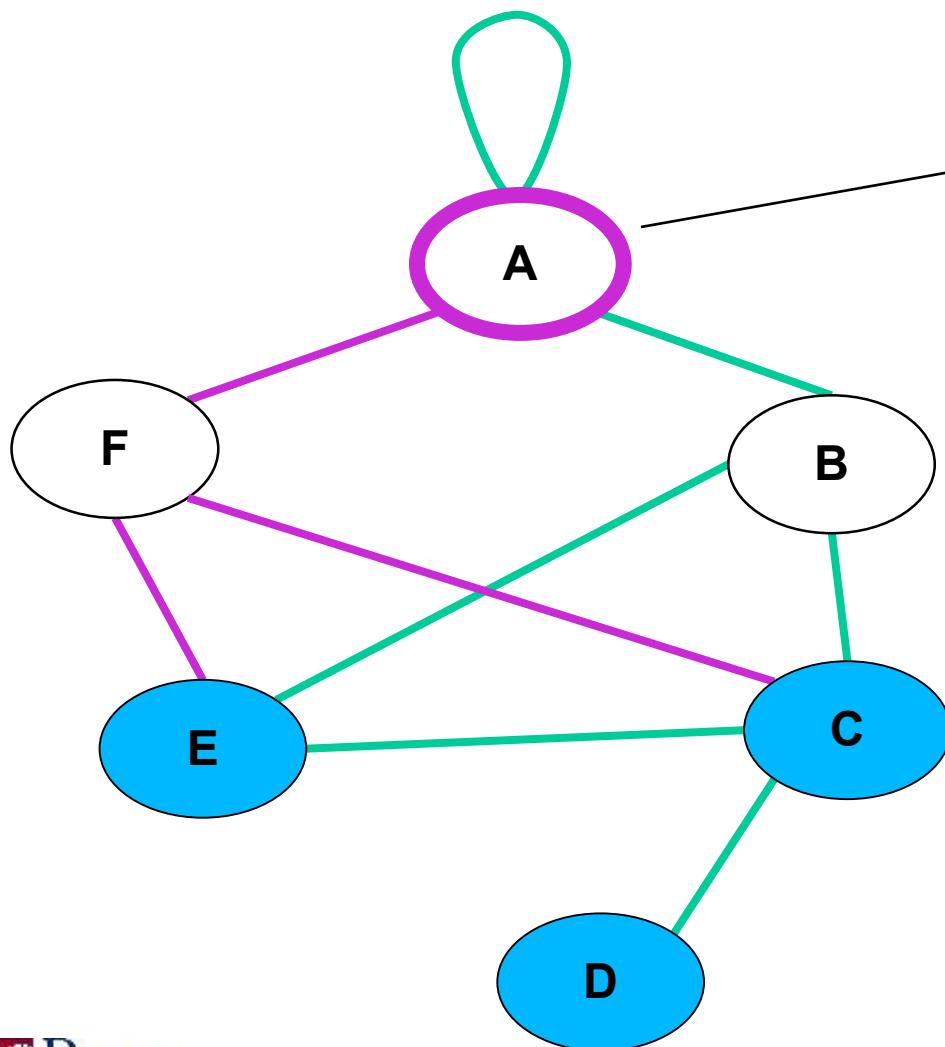
nodes marked:

A  
F  
B

queue of nodes to explore:

F  
B

# Breadth-first search (BFS)



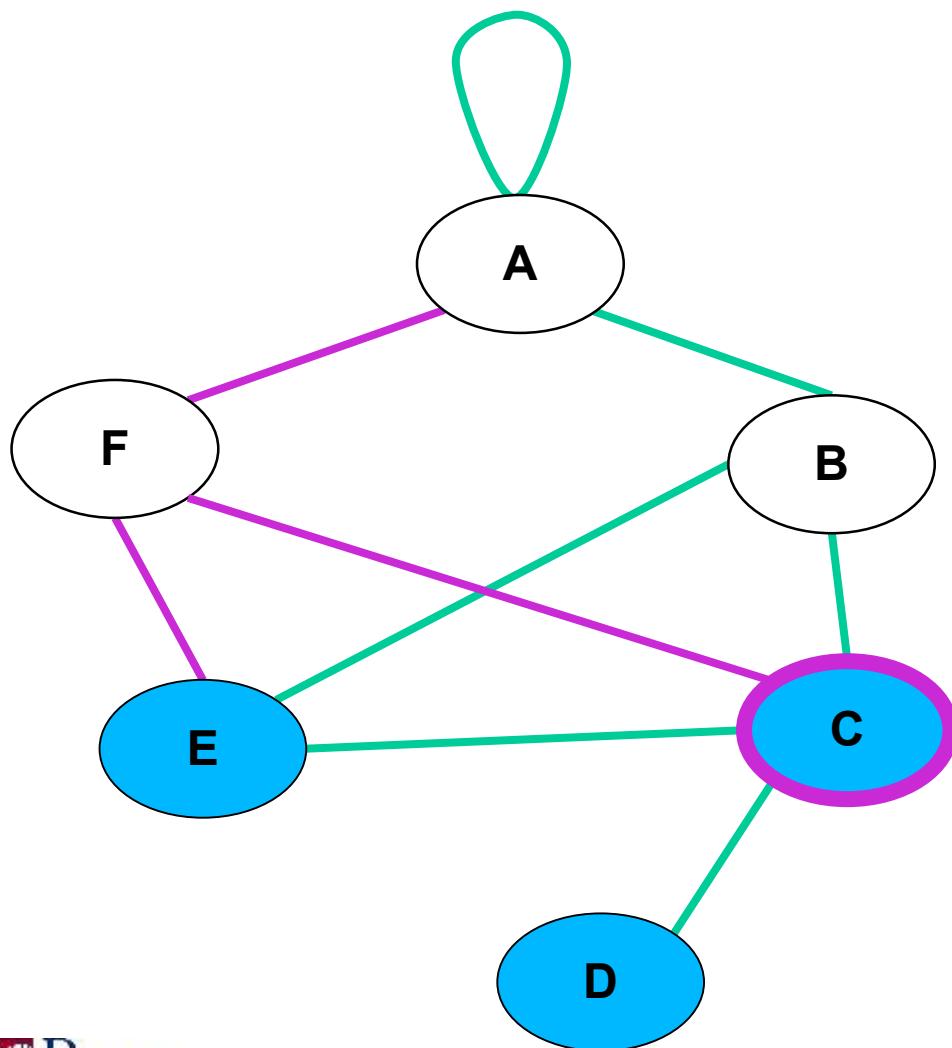
nodes marked:

A  
F  
B

queue of nodes to explore:

F  
B

# Breadth-first search (BFS)



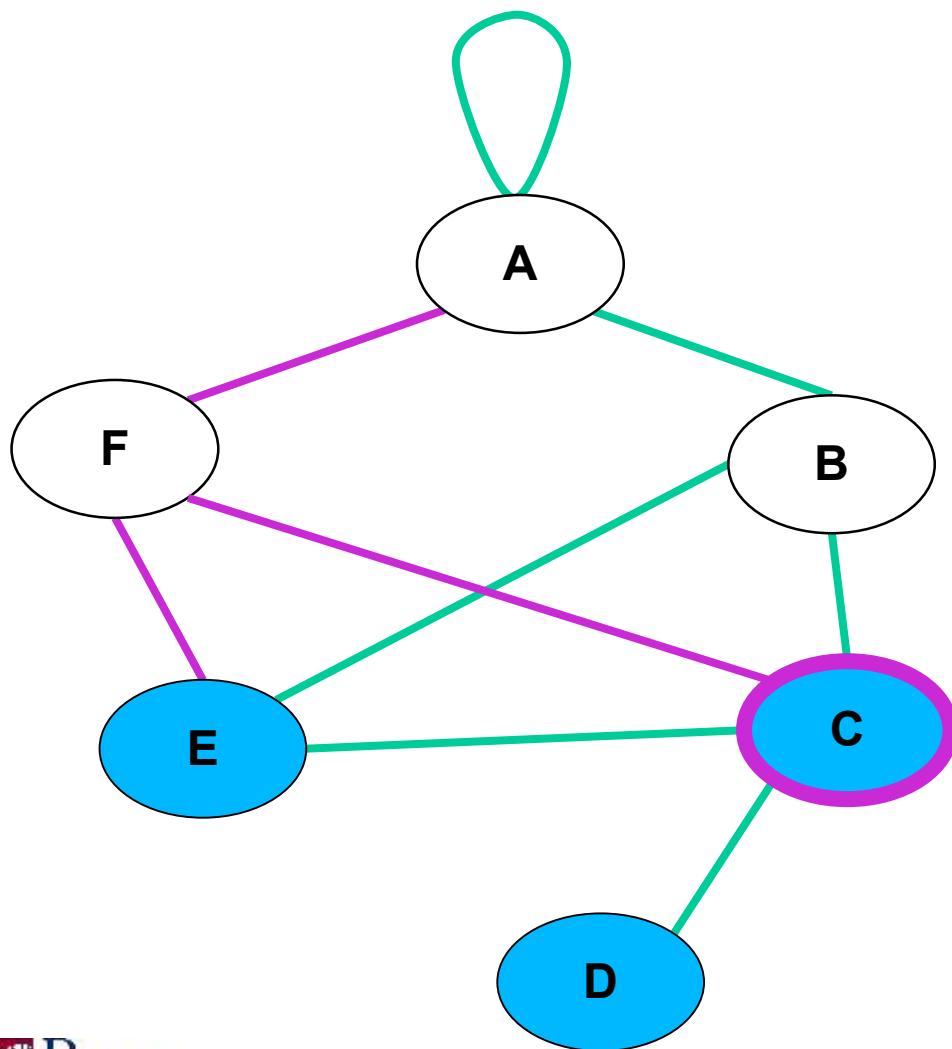
nodes marked:

A  
F  
B

queue of nodes to explore:

F  
B

# Breadth-first search (BFS)



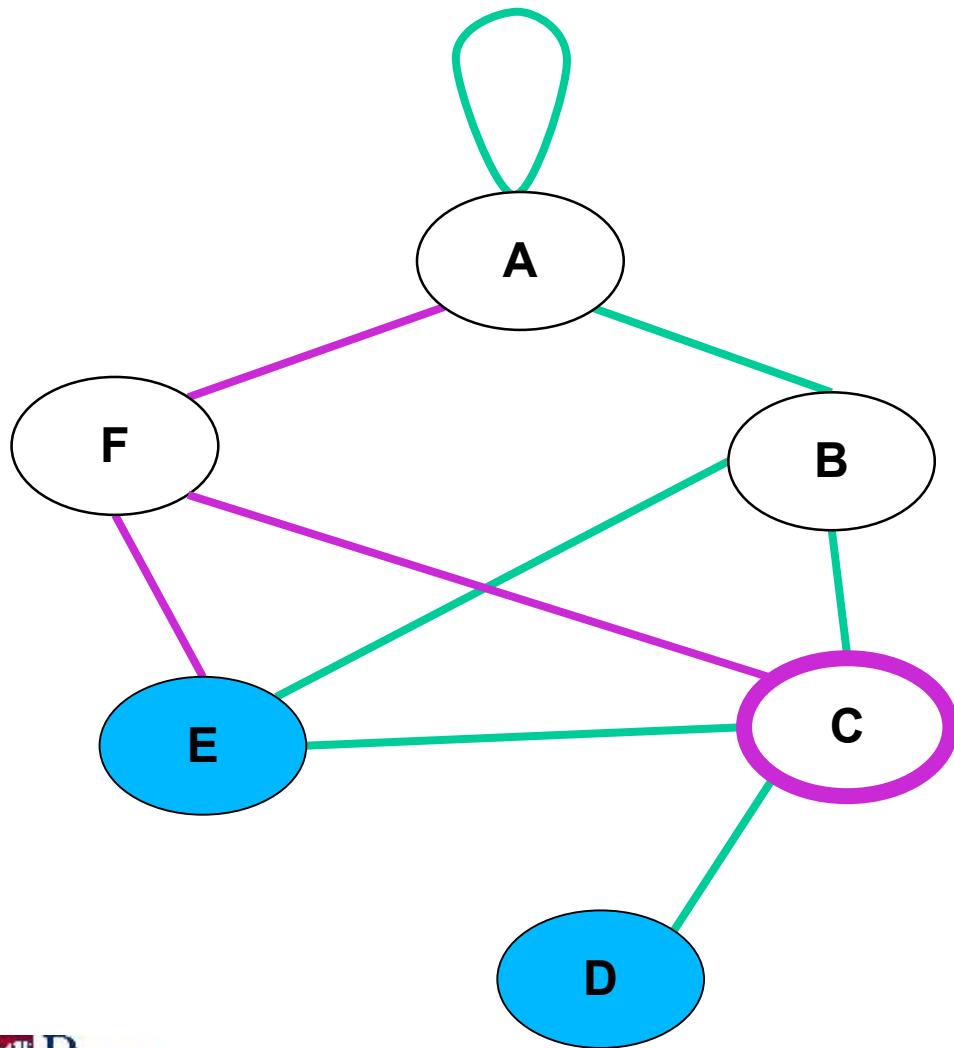
nodes marked:

A  
F  
B  
C

queue of nodes to explore:

F  
B

# Breadth-first search (BFS)



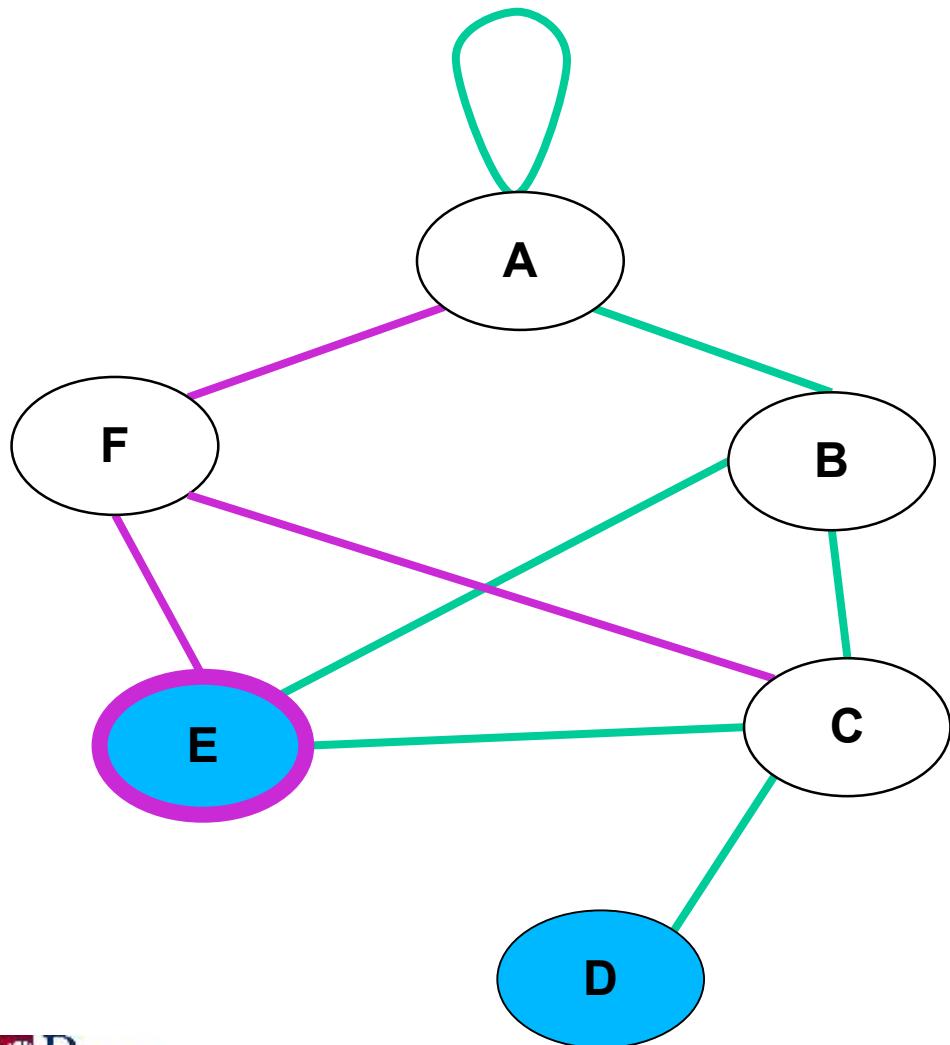
nodes marked:

A  
F  
B  
C

queue of nodes to explore:

F  
B  
C

# Breadth-first search (BFS)



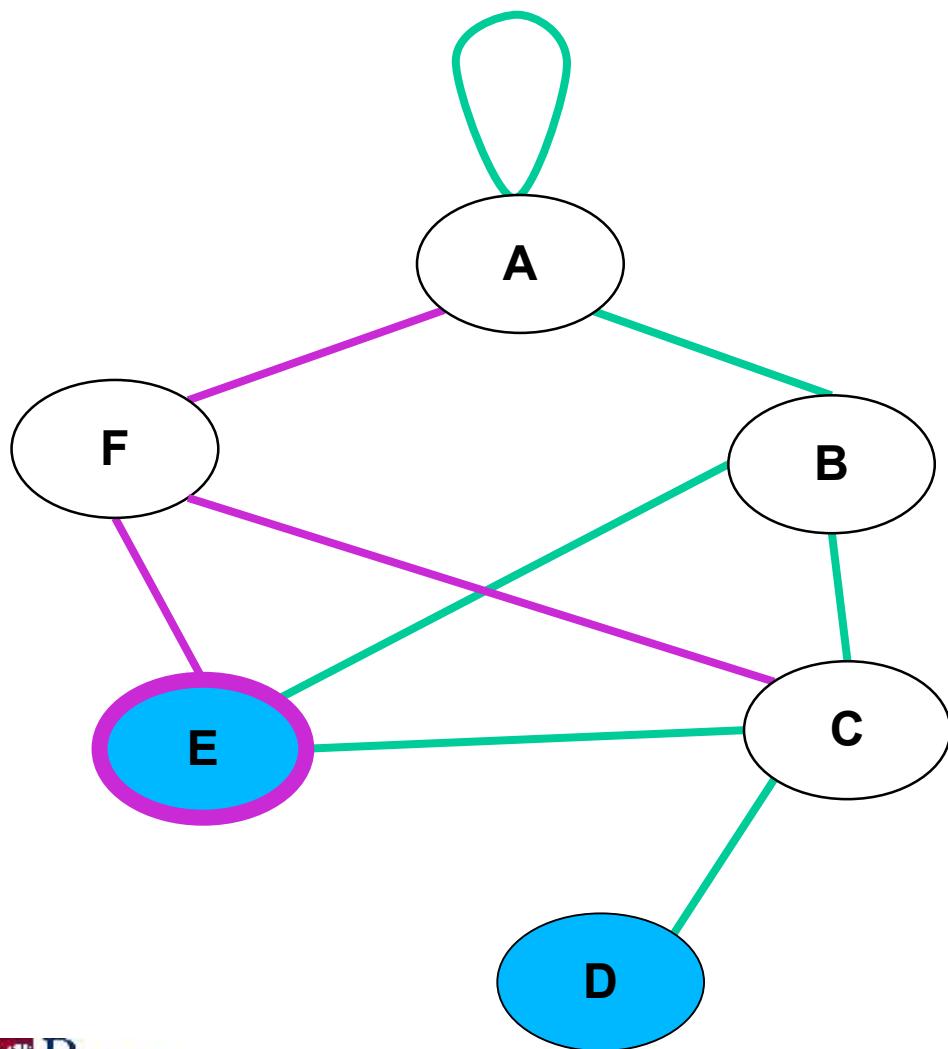
nodes marked:

A  
F  
B  
C

queue of nodes to explore:

F  
B  
C

# Breadth-first search (BFS)



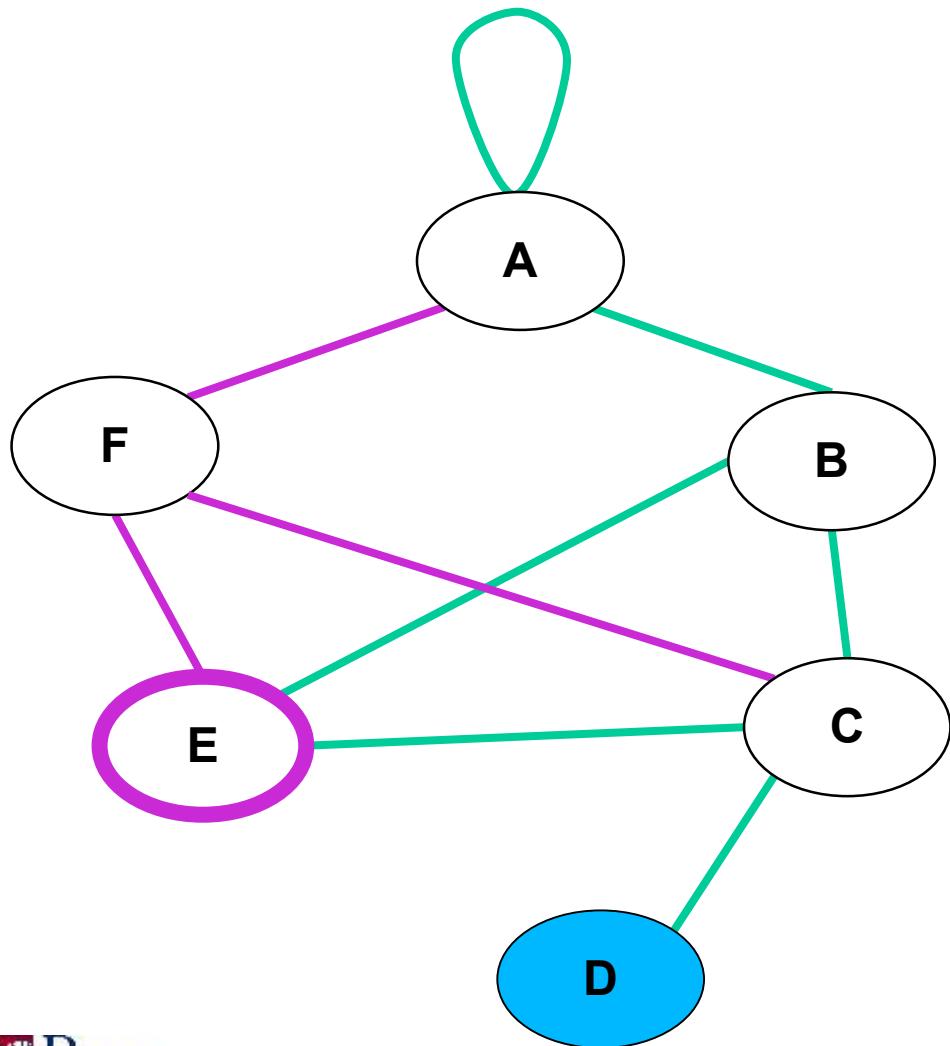
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

F  
B  
C

# Breadth-first search (BFS)



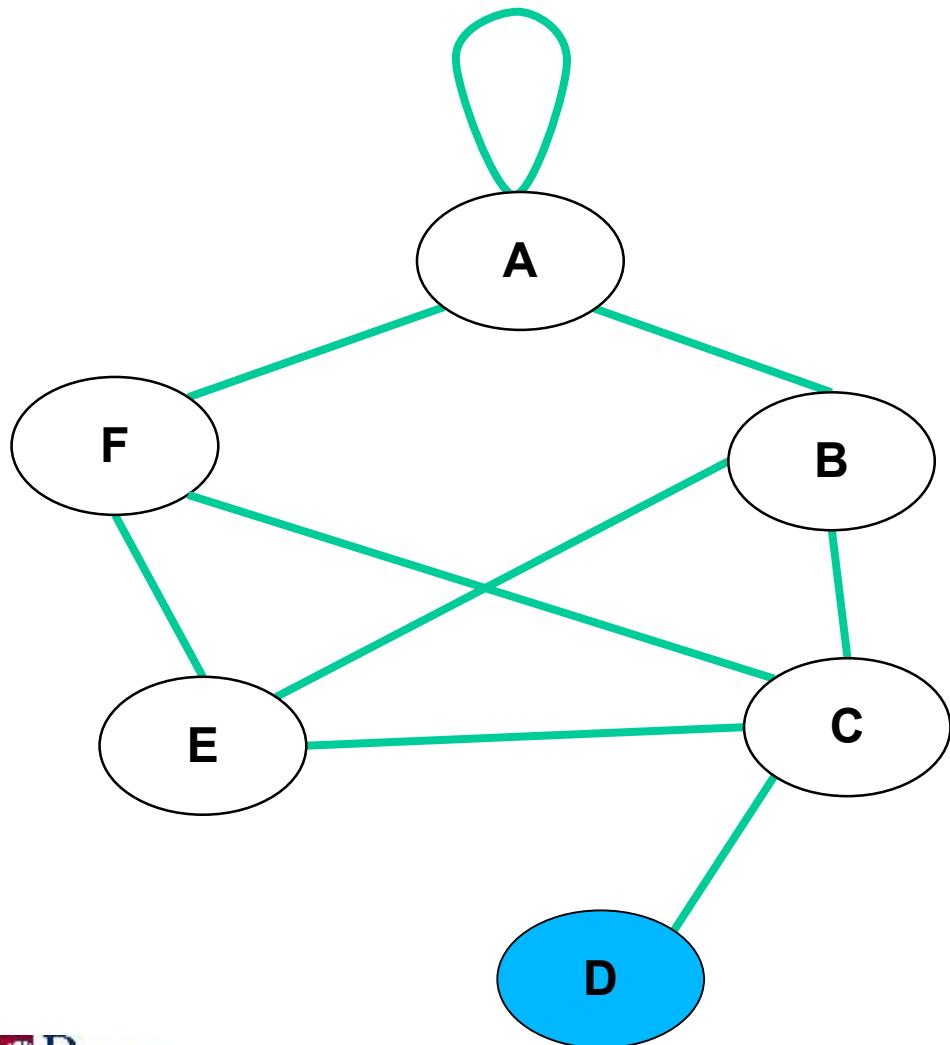
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

F  
B  
C  
E

# Breadth-first search (BFS)



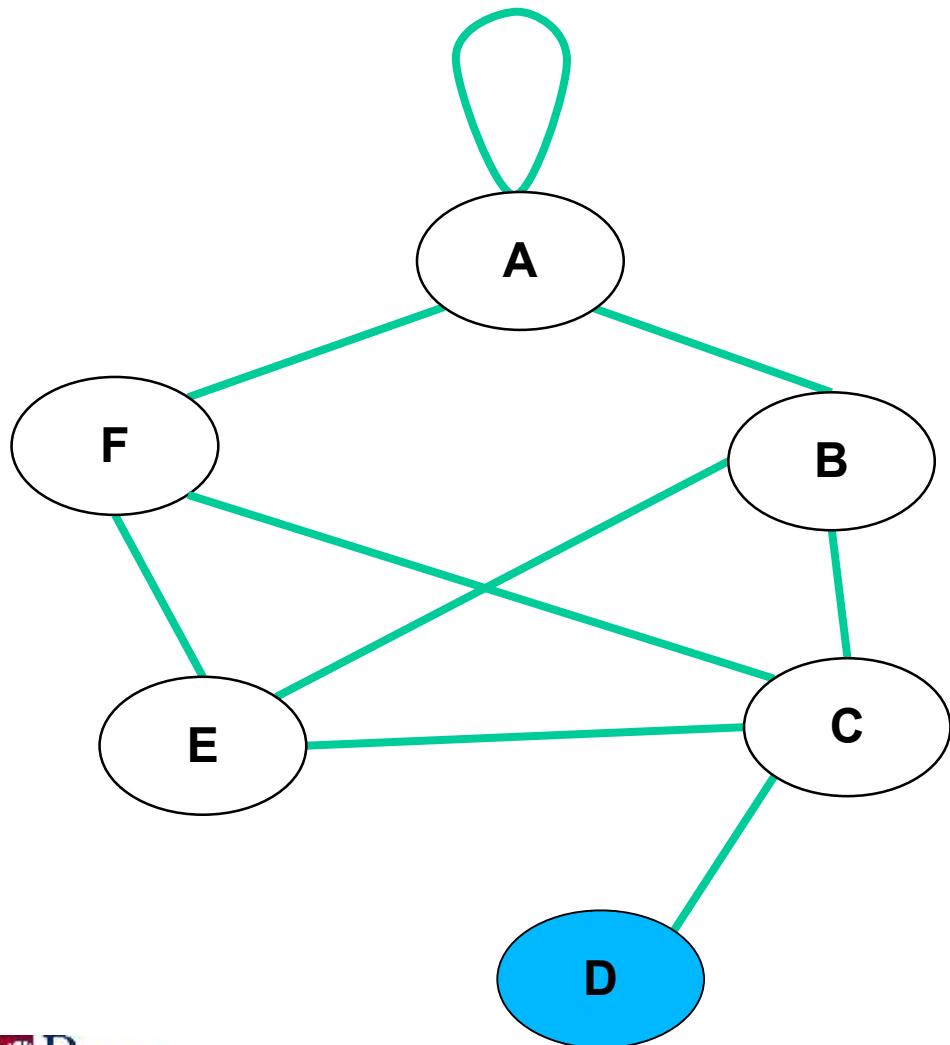
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

F  
B  
C  
E

# Breadth-first search (BFS)



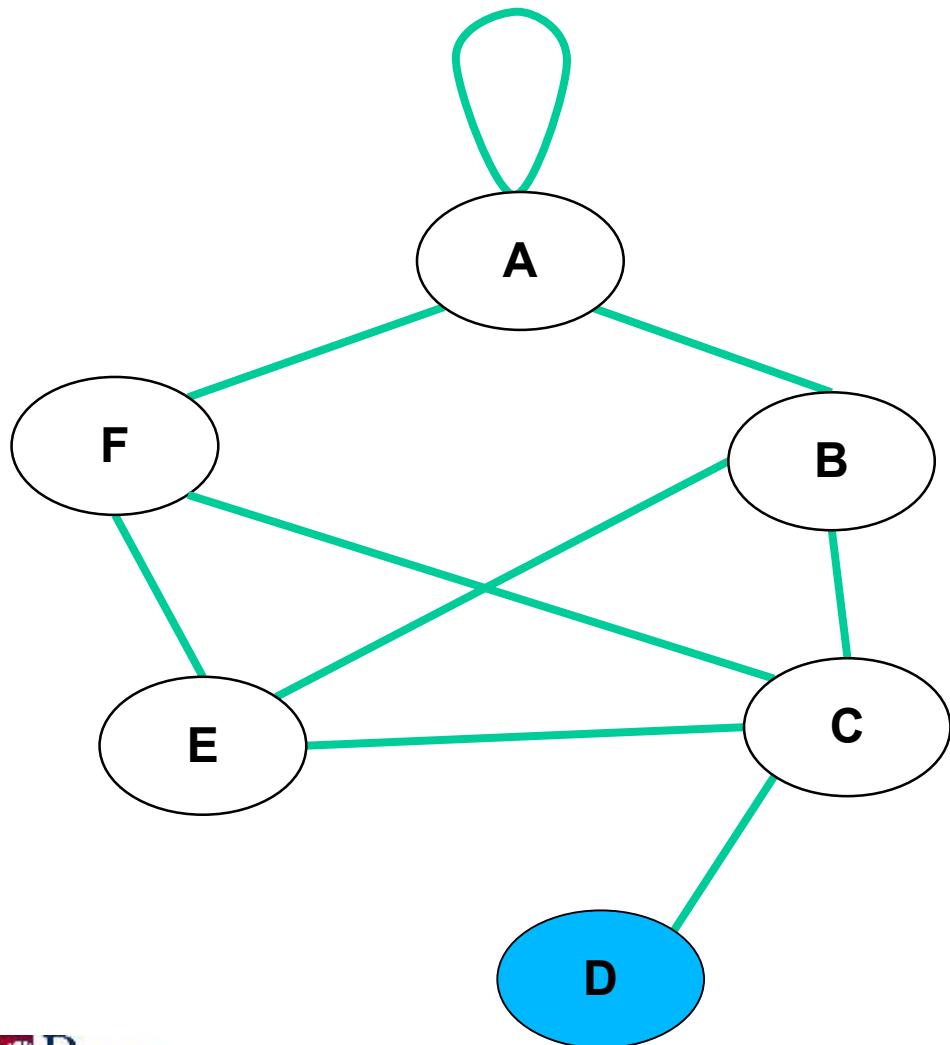
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

F  
B  
C  
E

# Breadth-first search (BFS)



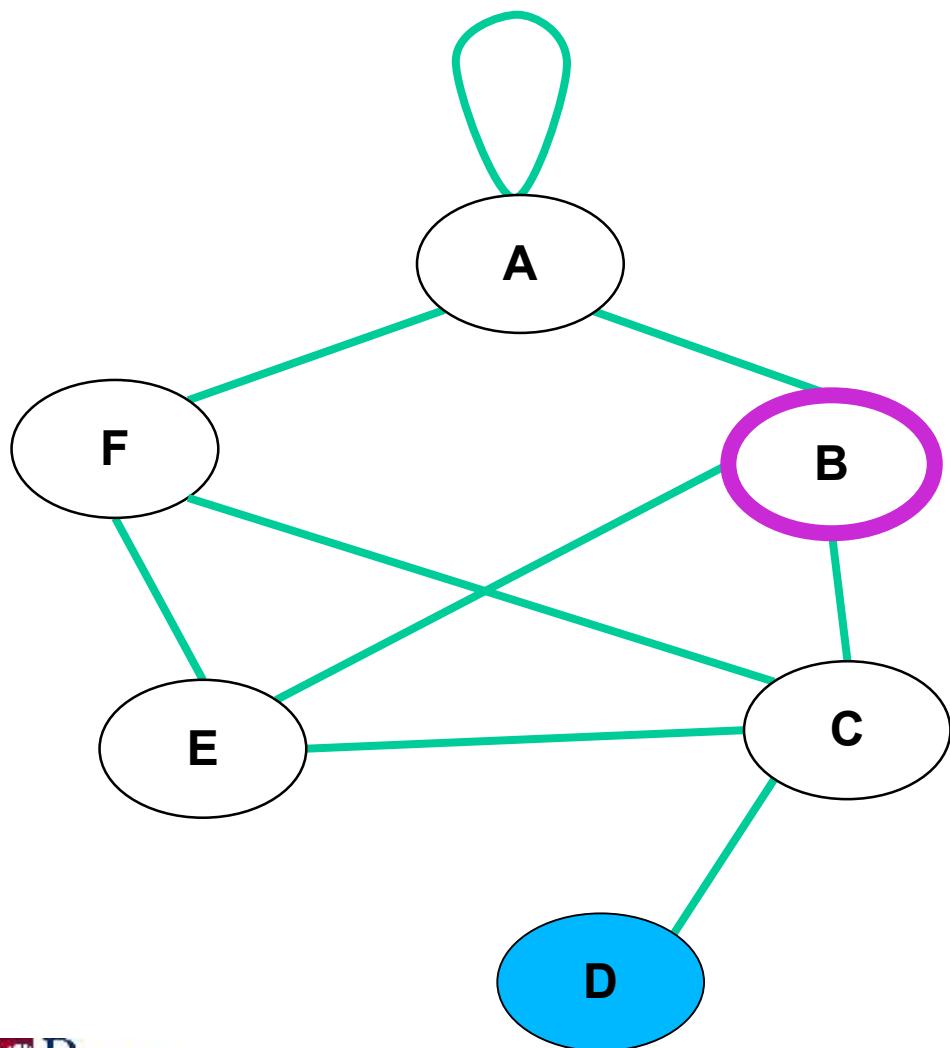
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

B  
C  
E

# Breadth-first search (BFS)



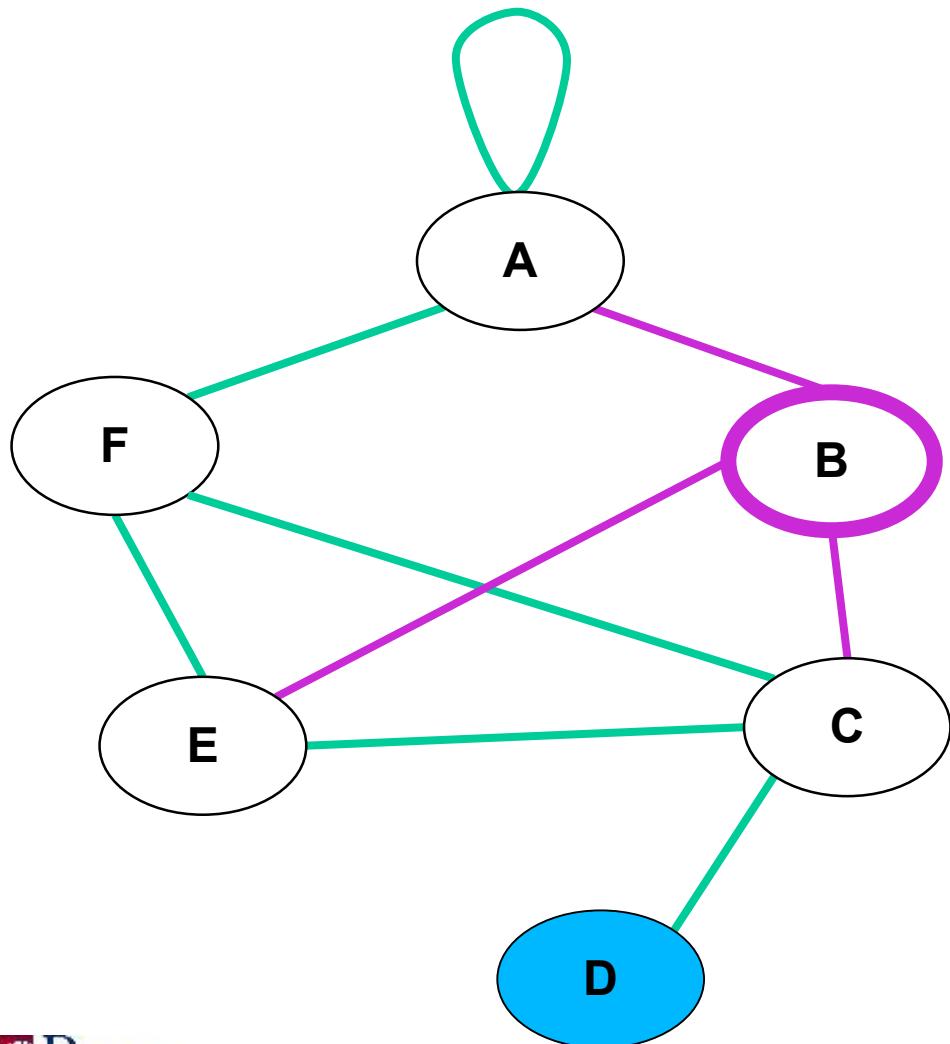
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

B  
C  
E

# Breadth-first search (BFS)



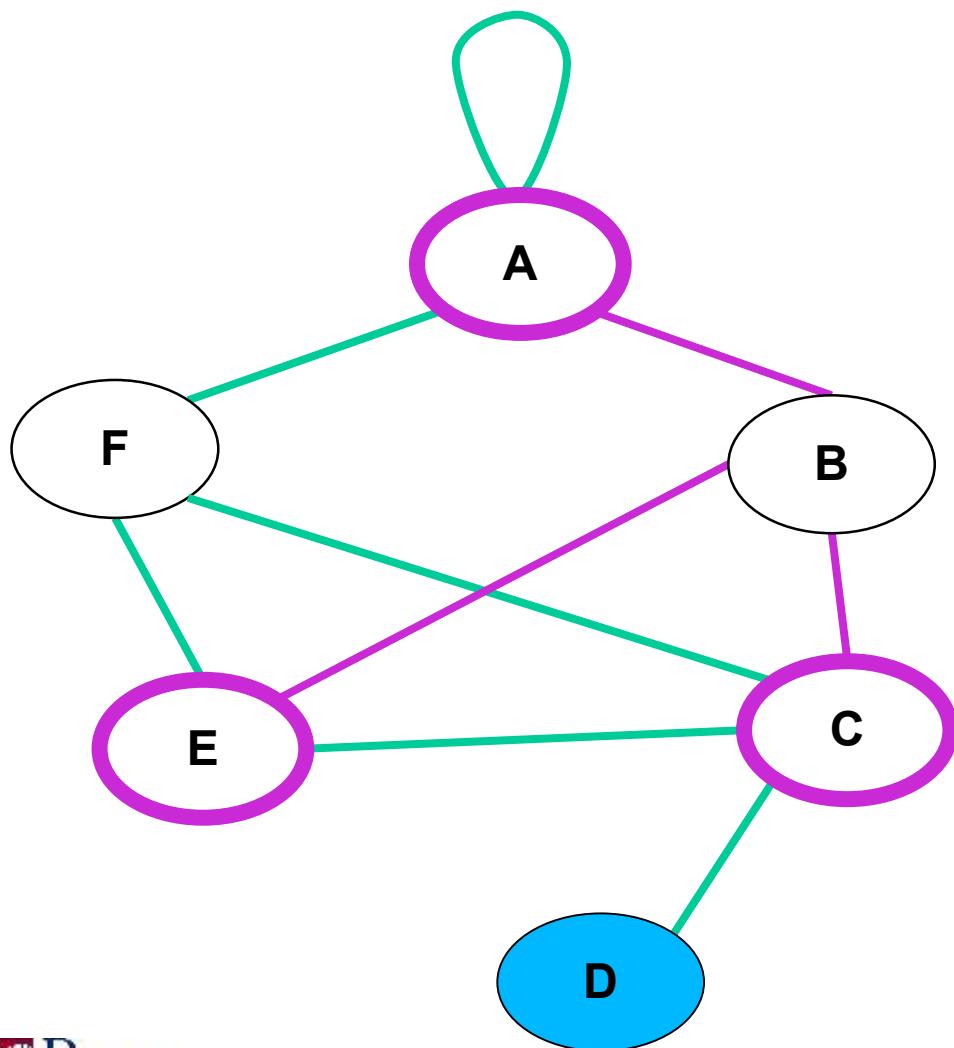
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

B  
C  
E

# Breadth-first search (BFS)



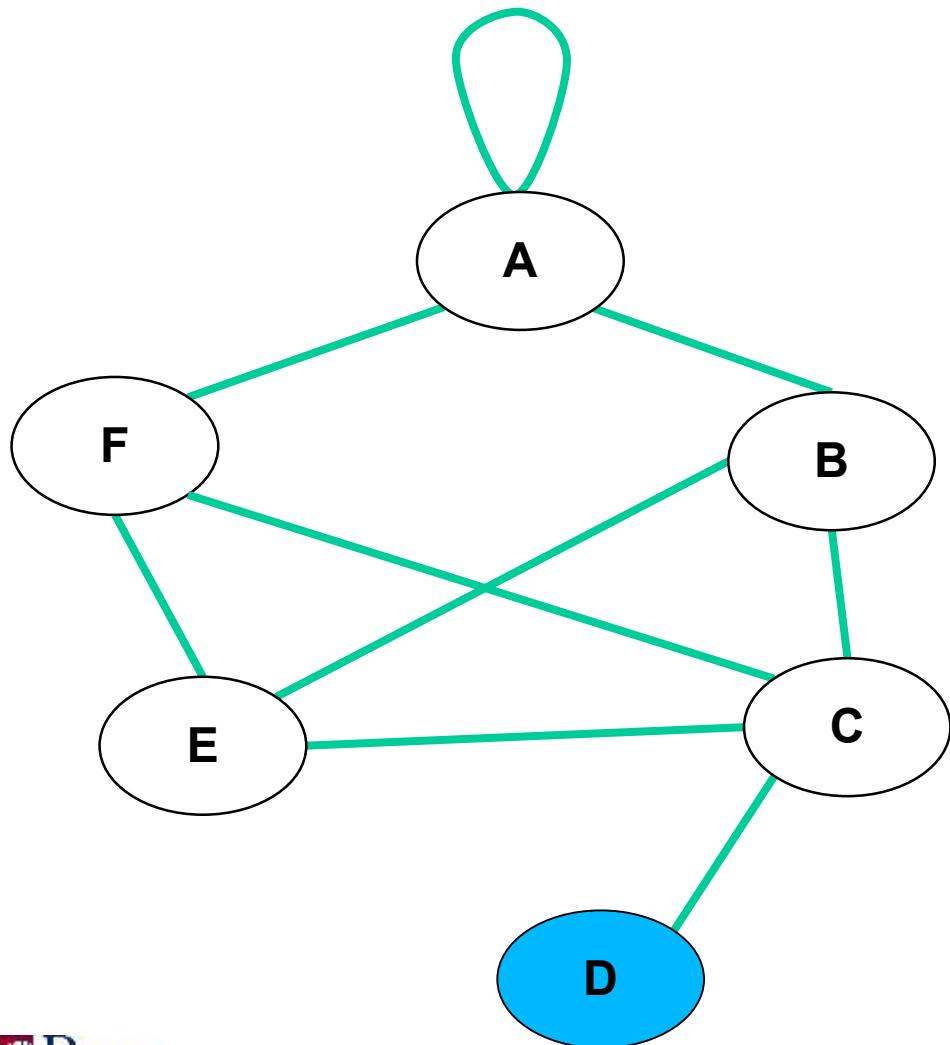
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

B  
C  
E

# Breadth-first search (BFS)



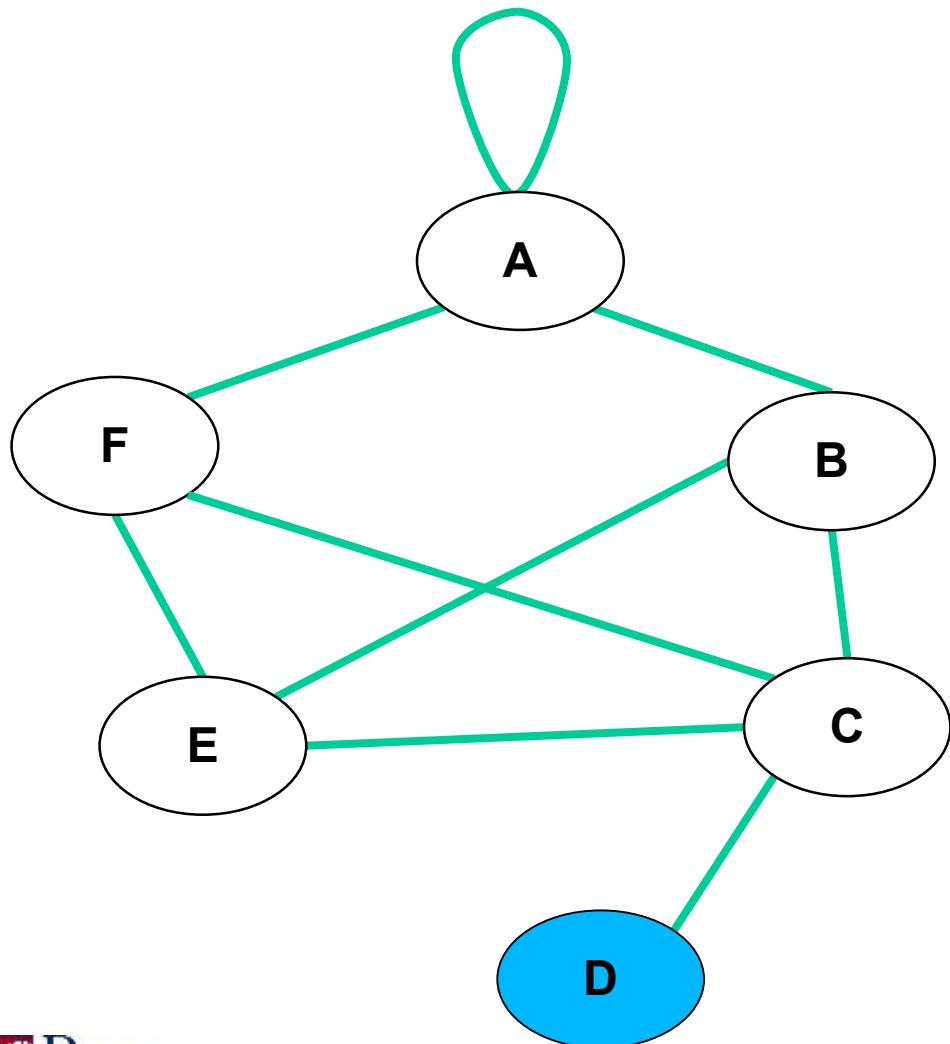
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

B  
C  
E

# Breadth-first search (BFS)



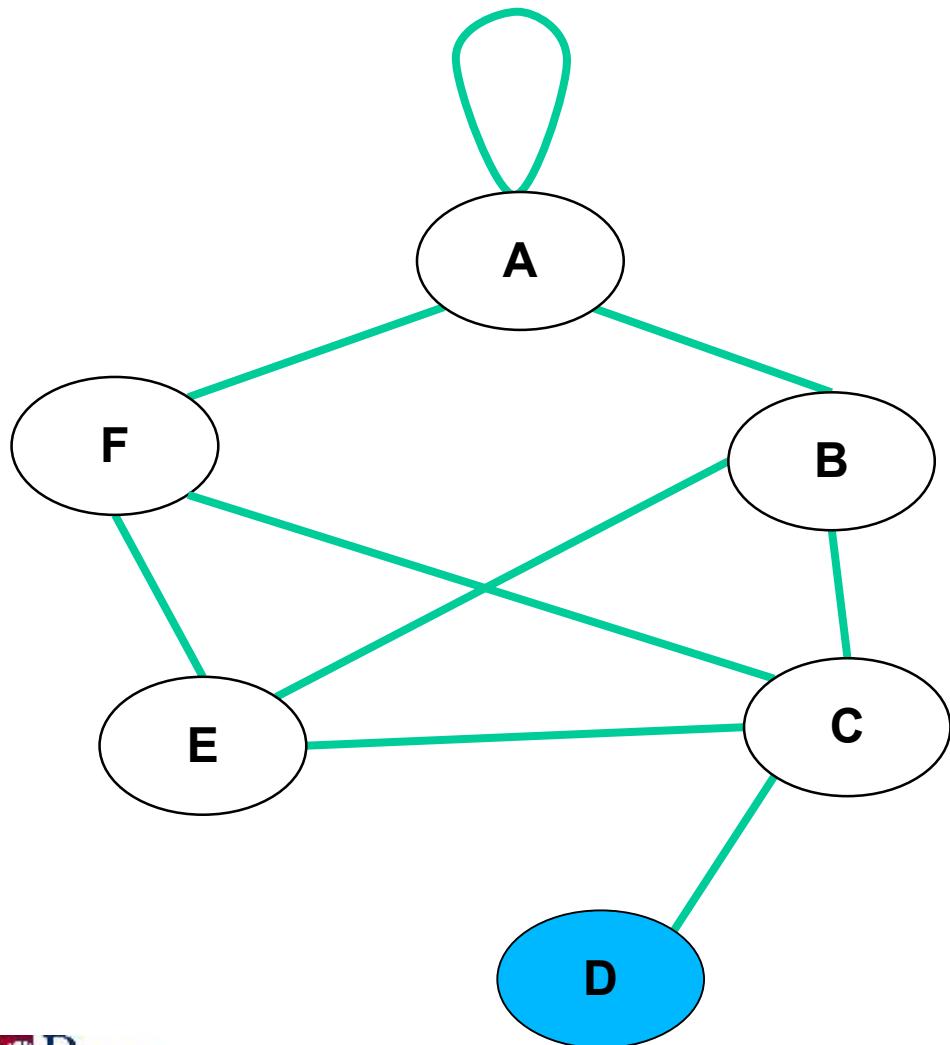
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

B  
C  
E

# Breadth-first search (BFS)



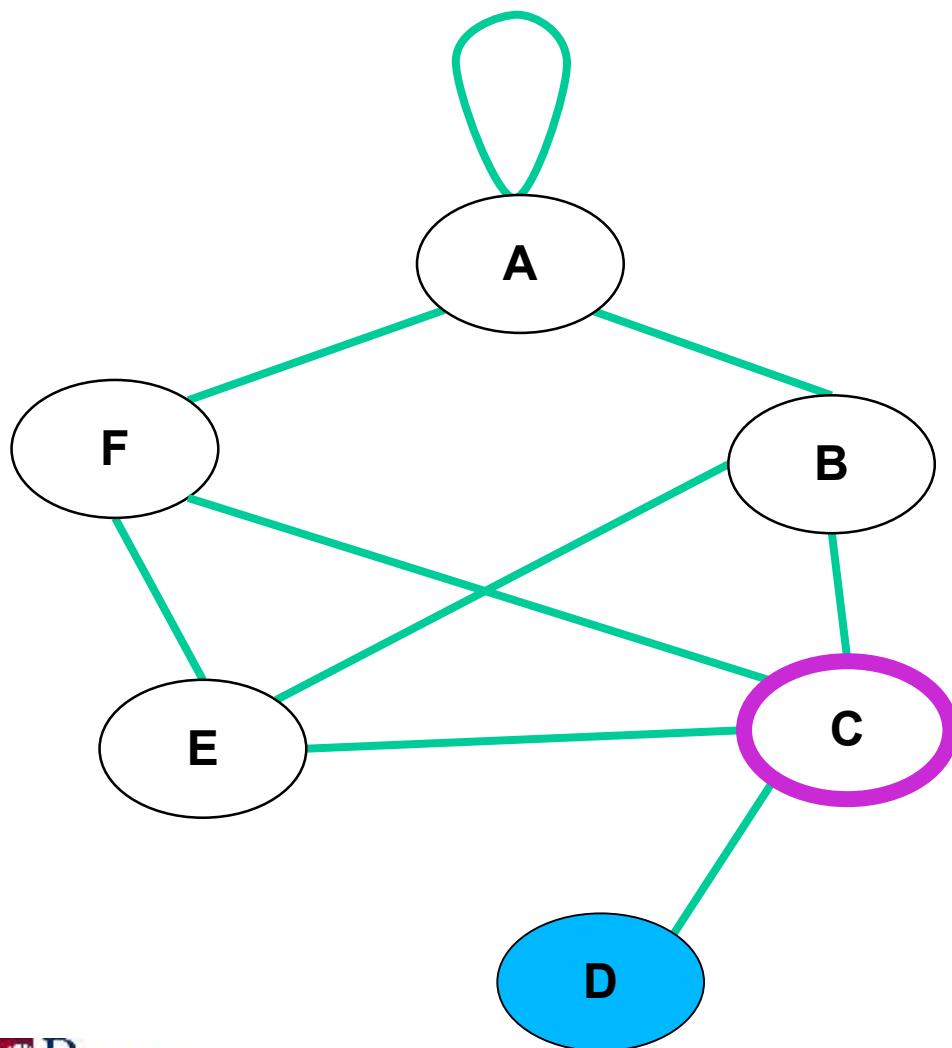
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

C  
E

# Breadth-first search (BFS)



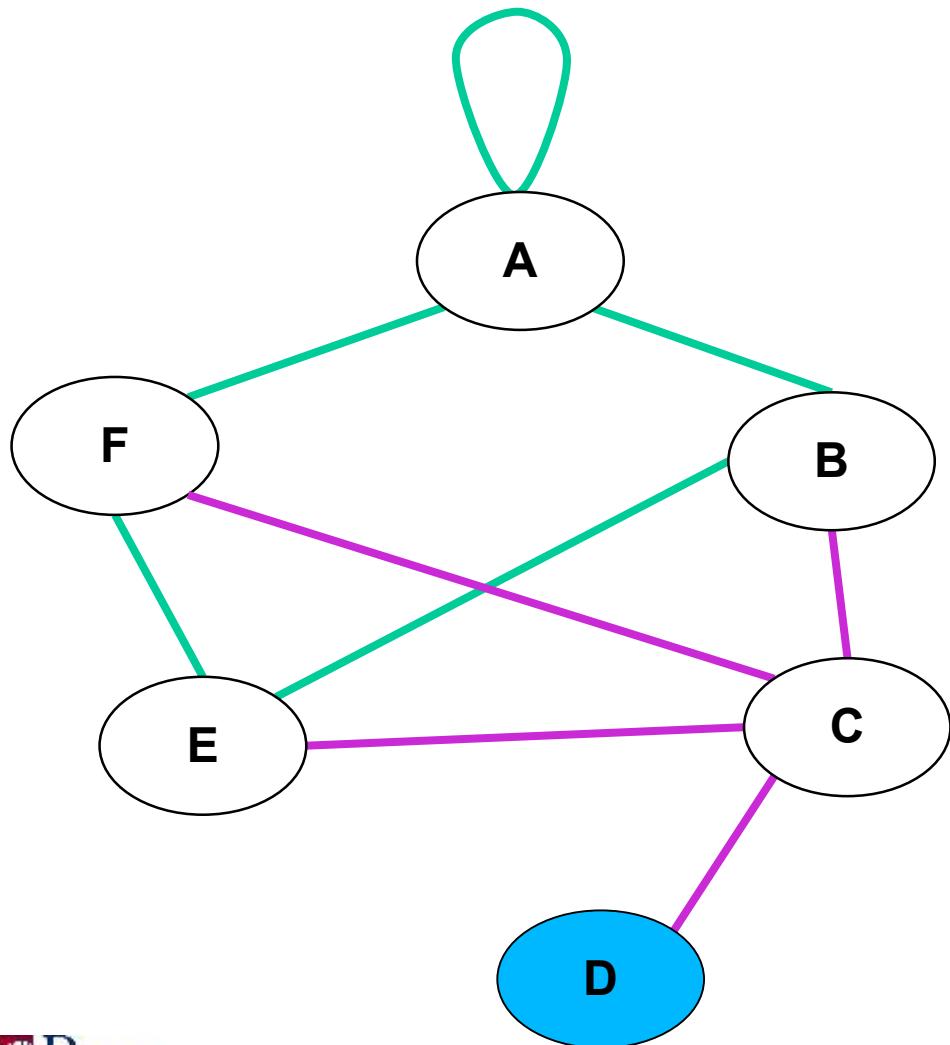
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

C  
E

# Breadth-first search (BFS)



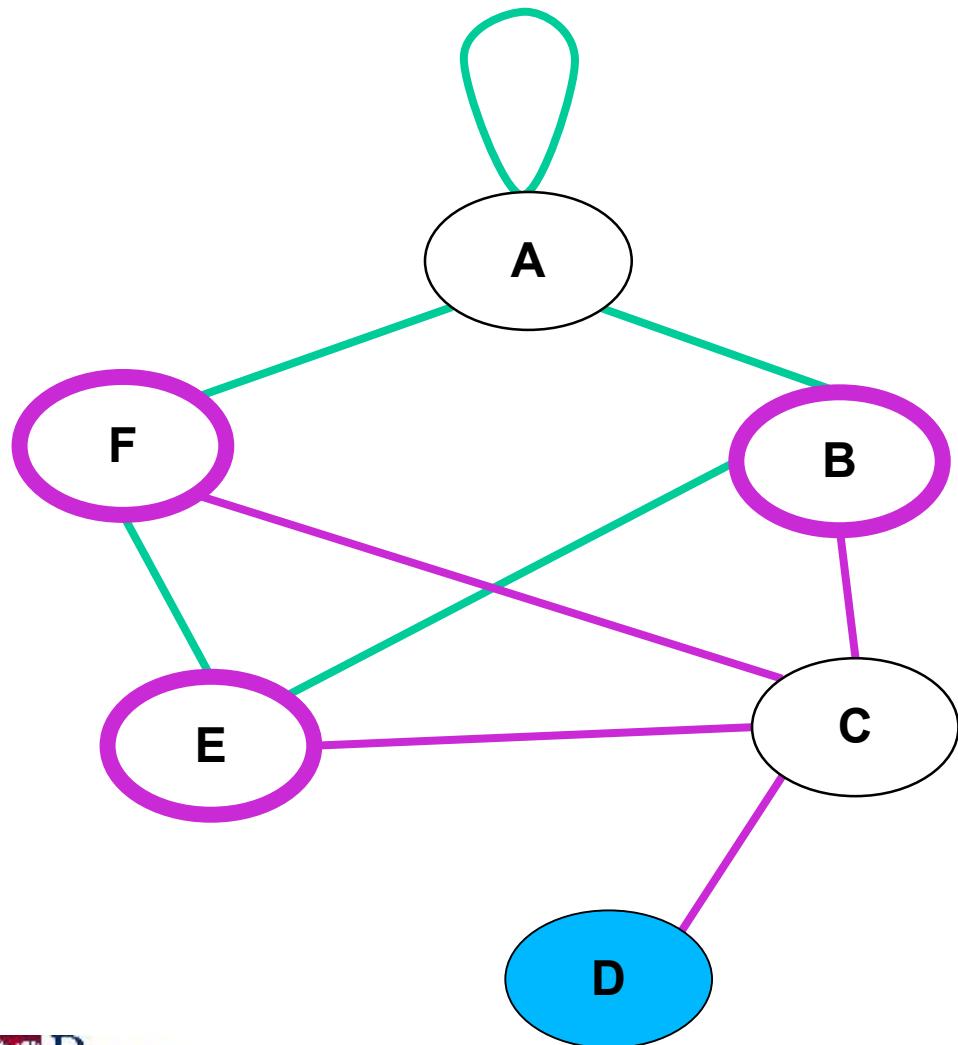
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

C  
E

# Breadth-first search (BFS)



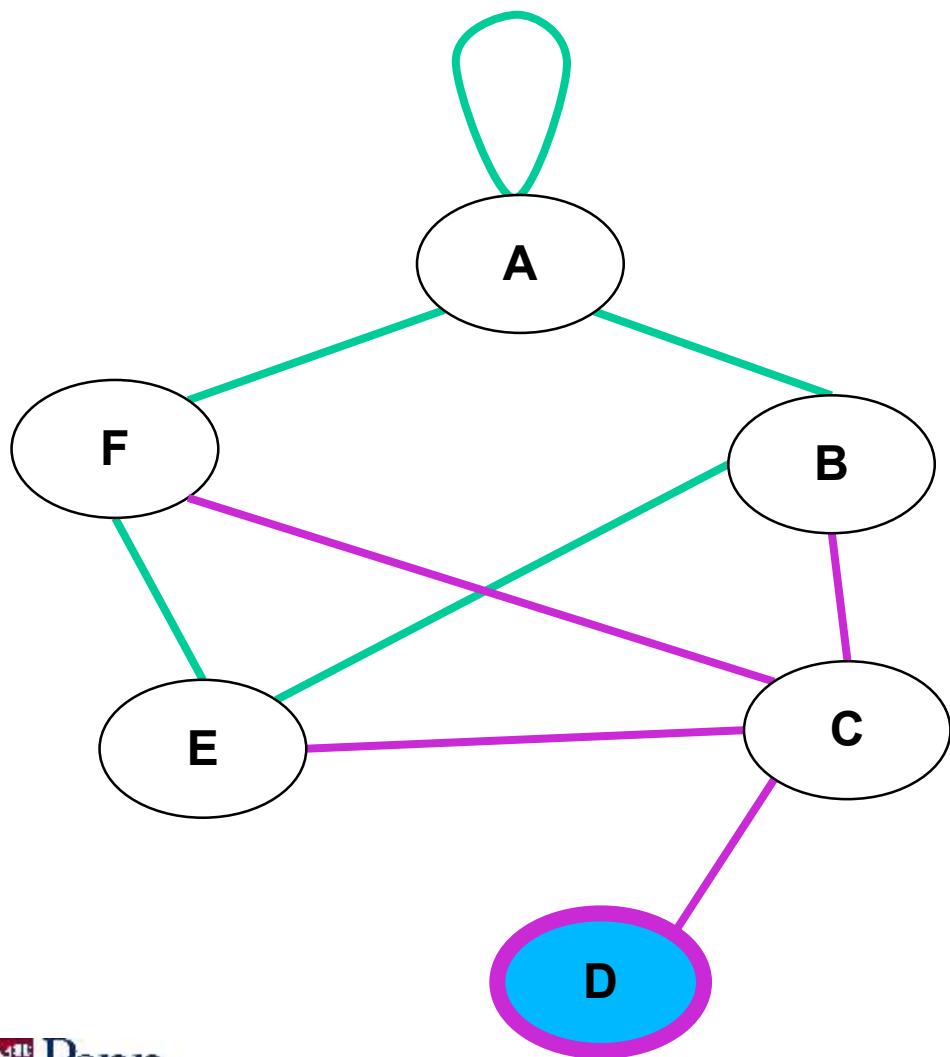
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

C  
E

# Breadth-first search (BFS)



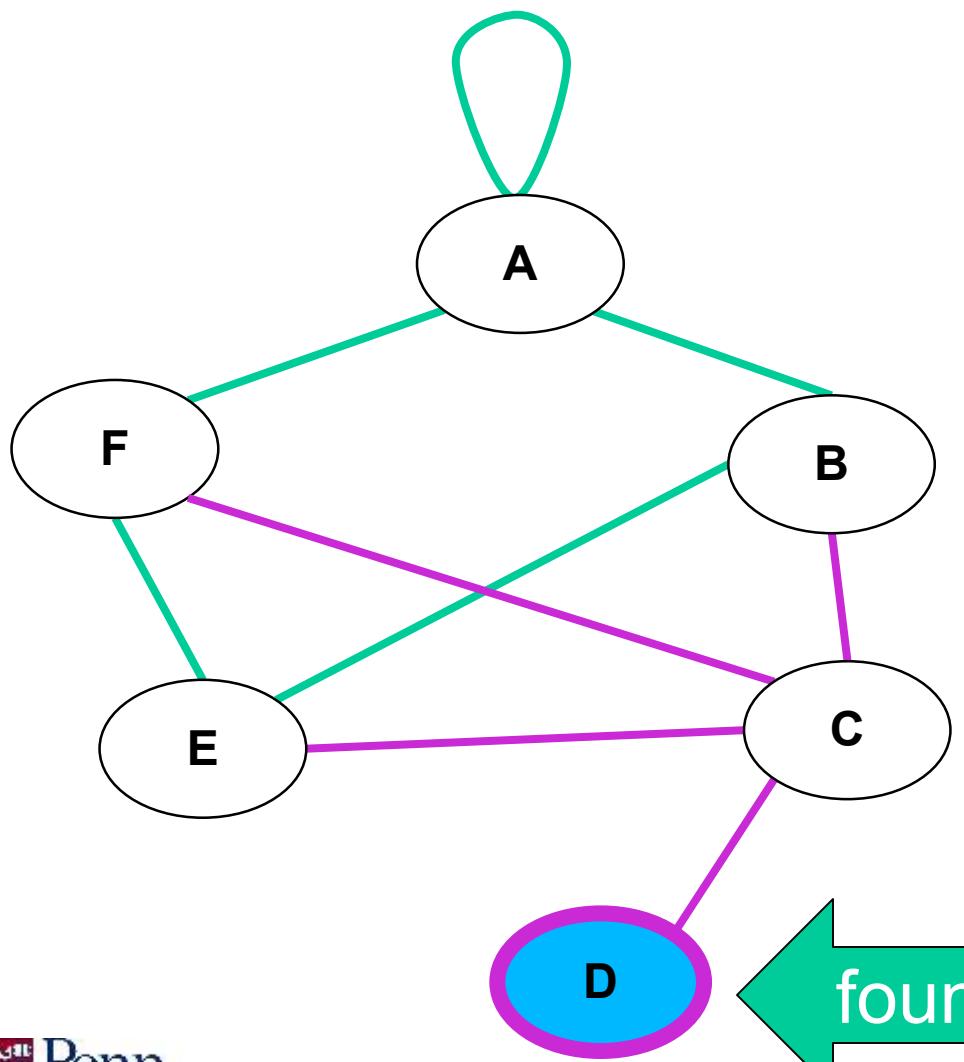
nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

C  
E

# Breadth-first search (BFS)



nodes marked:

A  
F  
B  
C  
E

queue of nodes to explore:

C  
E

found it!

```
public class BreadthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public BreadthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean bfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class BreadthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public BreadthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean bfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class BreadthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public BreadthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean bfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class BreadthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public BreadthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean bfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class BreadthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public BreadthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
public boolean bfs(String elementToFind, Node start) {  
    . . .  
}  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
if (start.getElement().equals(elementToFind)) {  
    return true;  
}  
Queue<Node> toExplore = new LinkedList<Node>();  
marked.add(start);  
toExplore.add(start);  
while (!toExplore.isEmpty()) {  
    Node current = toExplore.remove();  
    for (Node neighbor : graph.getNodeNeighbors(current)) {  
        if (!marked.contains(neighbor)) {  
            if (neighbor.getElement().equals(elementToFind)) {  
                return true;  
            }  
            marked.add(neighbor);  
            toExplore.add(neighbor);  
        }  
    }  
}  
return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
Queue<Node> toExplore = new LinkedList<Node>();  
marked.add(start);  
toExplore.add(start);  
while (!toExplore.isEmpty()) {  
    Node current = toExplore.remove();  
    for (Node neighbor : graph.getNodeNeighbors(current)) {  
        if (!marked.contains(neighbor)) {  
            if (neighbor.getElement().equals(elementToFind)) {  
                return true;  
            }  
            marked.add(neighbor);  
            toExplore.add(neighbor);  
        }  
    }  
}  
return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
    Queue<Node> toExplore = new LinkedList<Node>();  
    marked.add(start);  
    toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
                marked.add(neighbor);  
                toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

# Recap: Breadth-first search

---

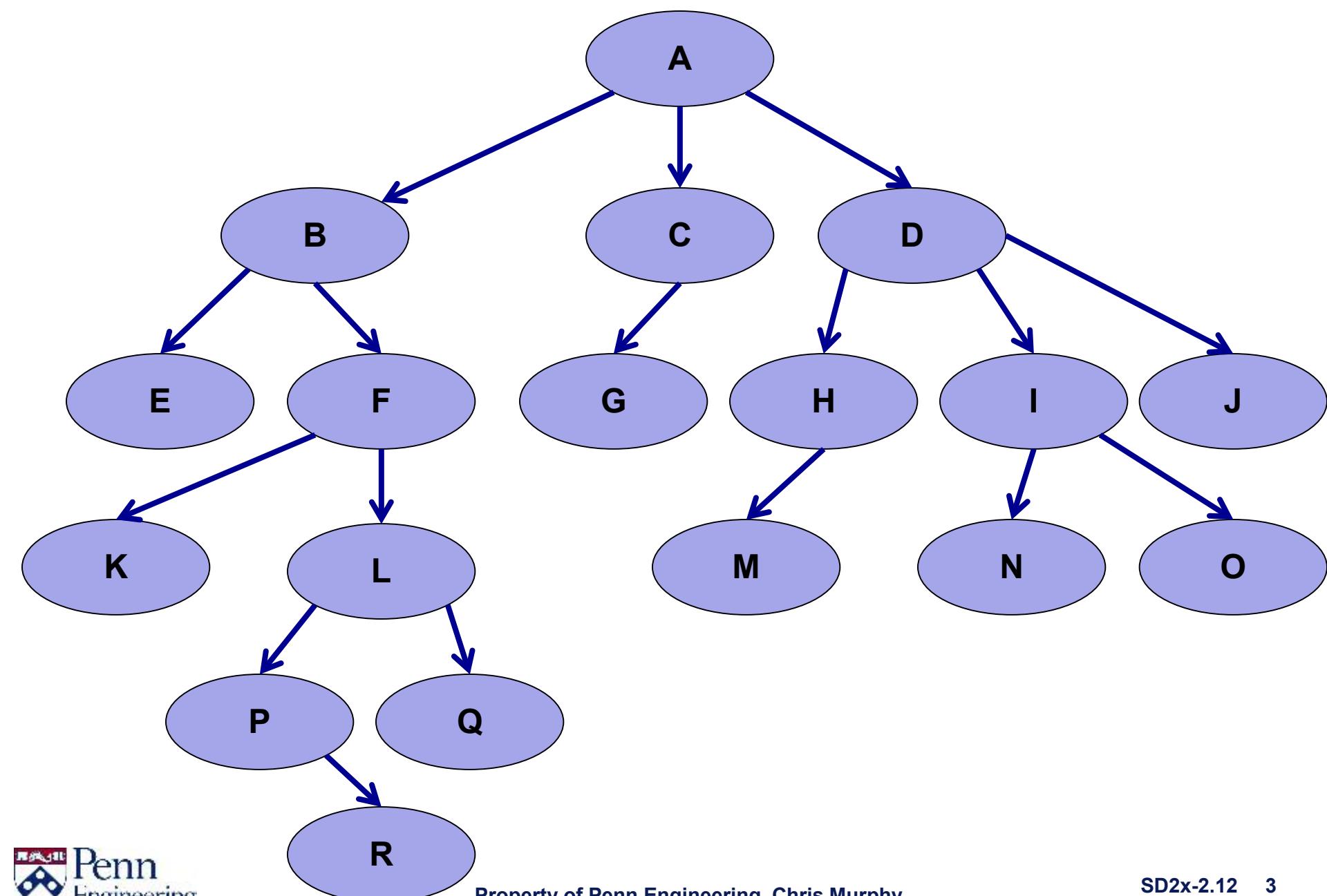
- Systematic way of traversing a graph
- Start with a source node...
- then visit all its neighbors...
- then visit all **their** neighbors...
- and so on, until destination is found.

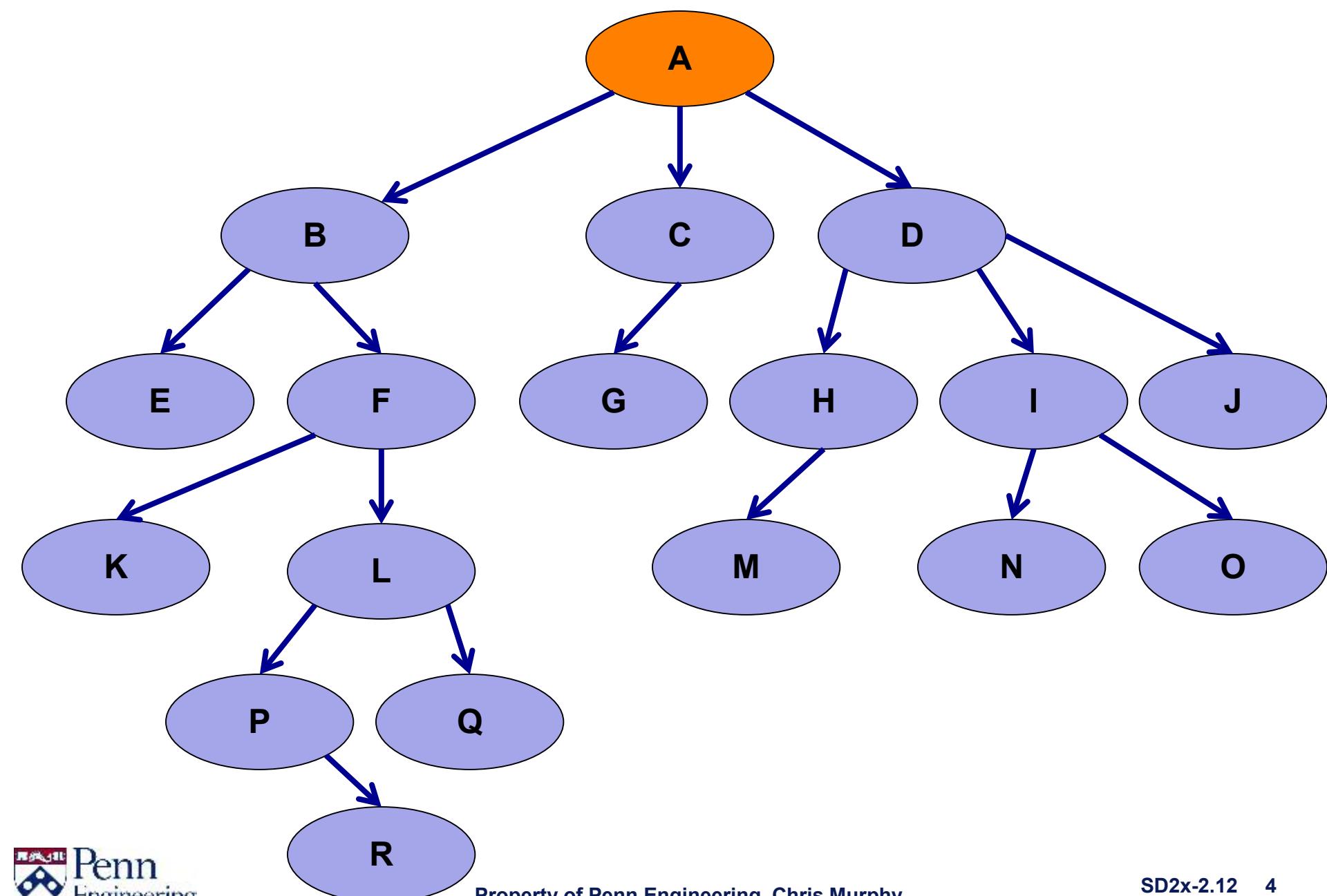
# **SD2x2.12**

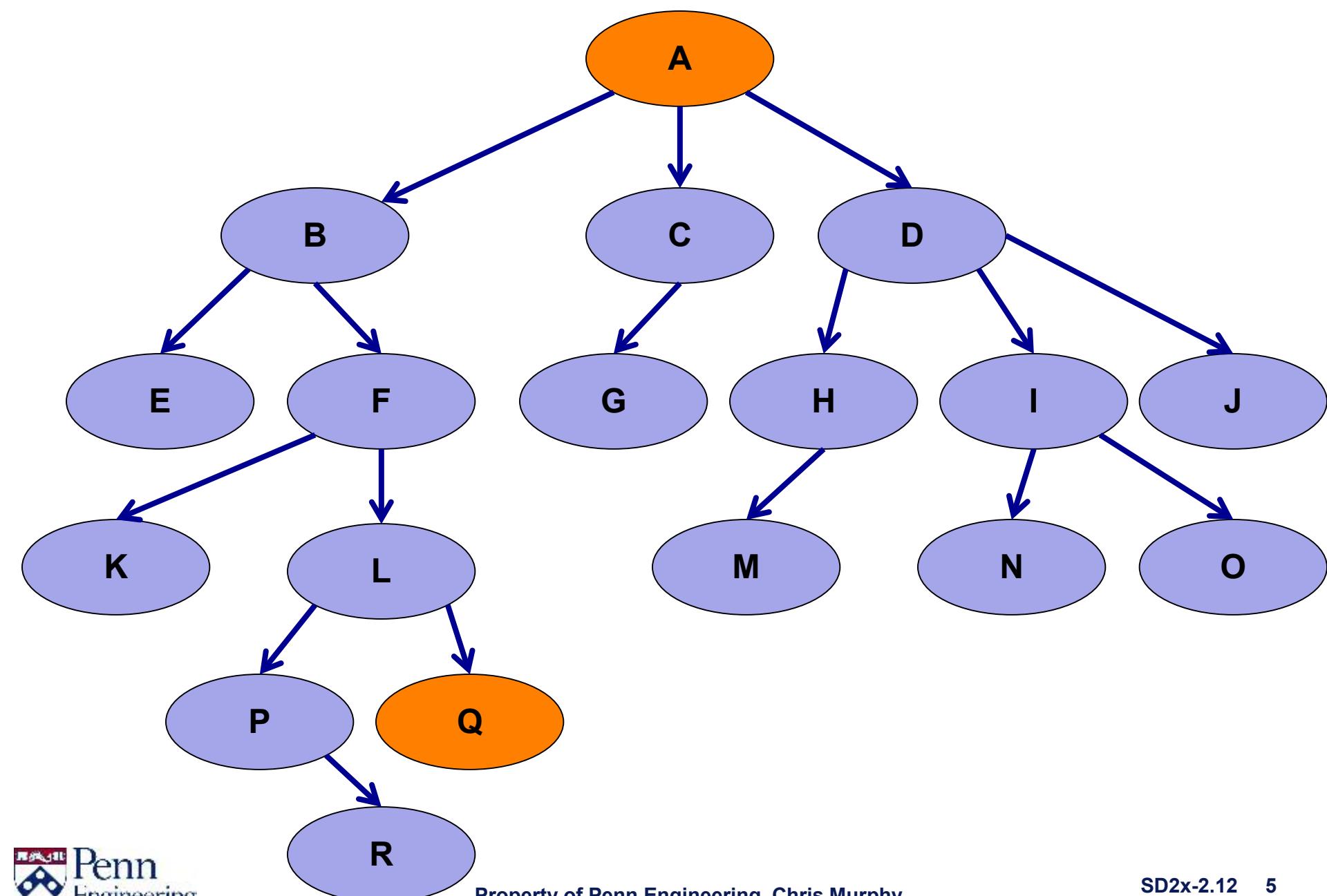
## **Graph DFS**

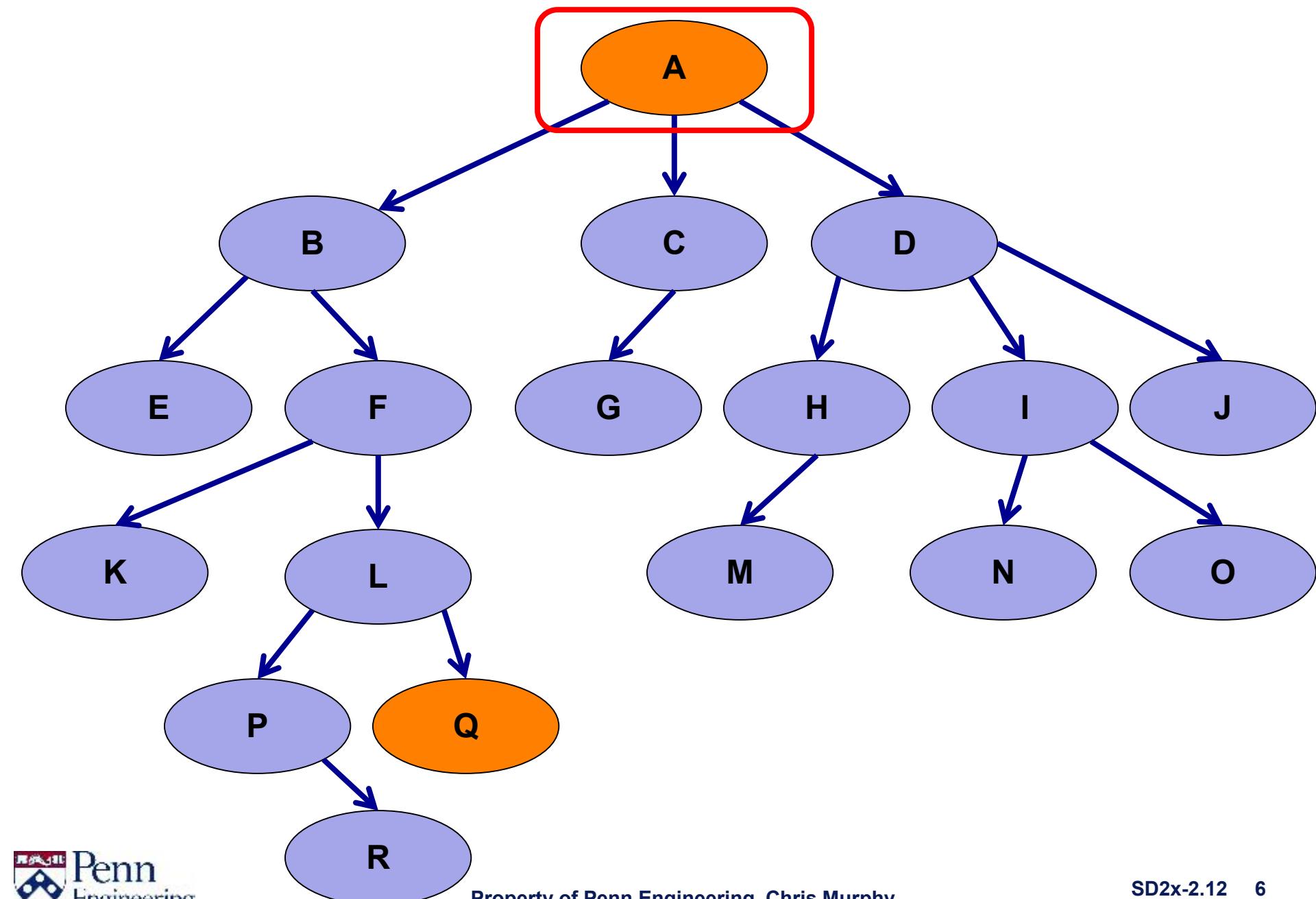
### **Chris**

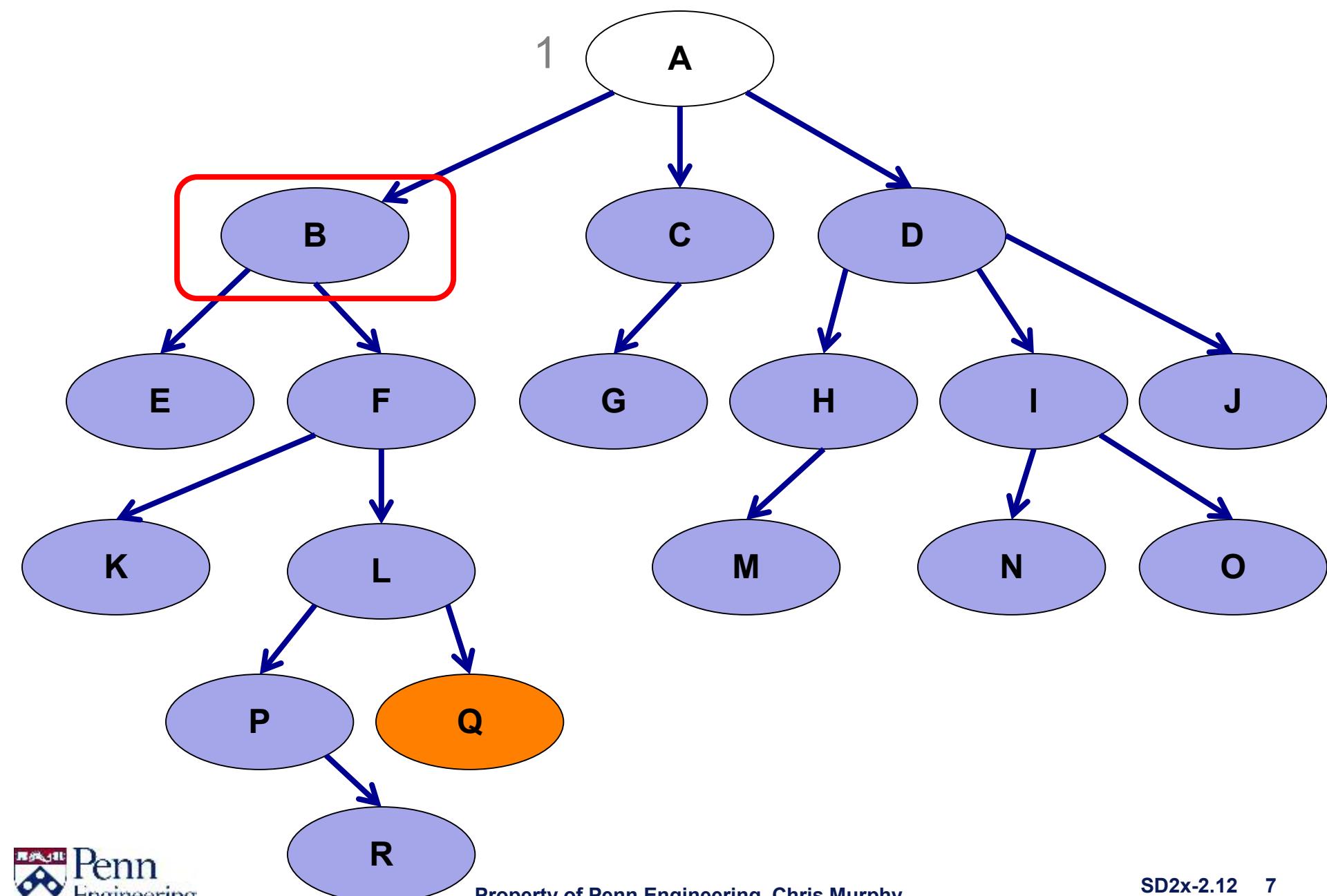
# **How do we find a path from one node to another?**

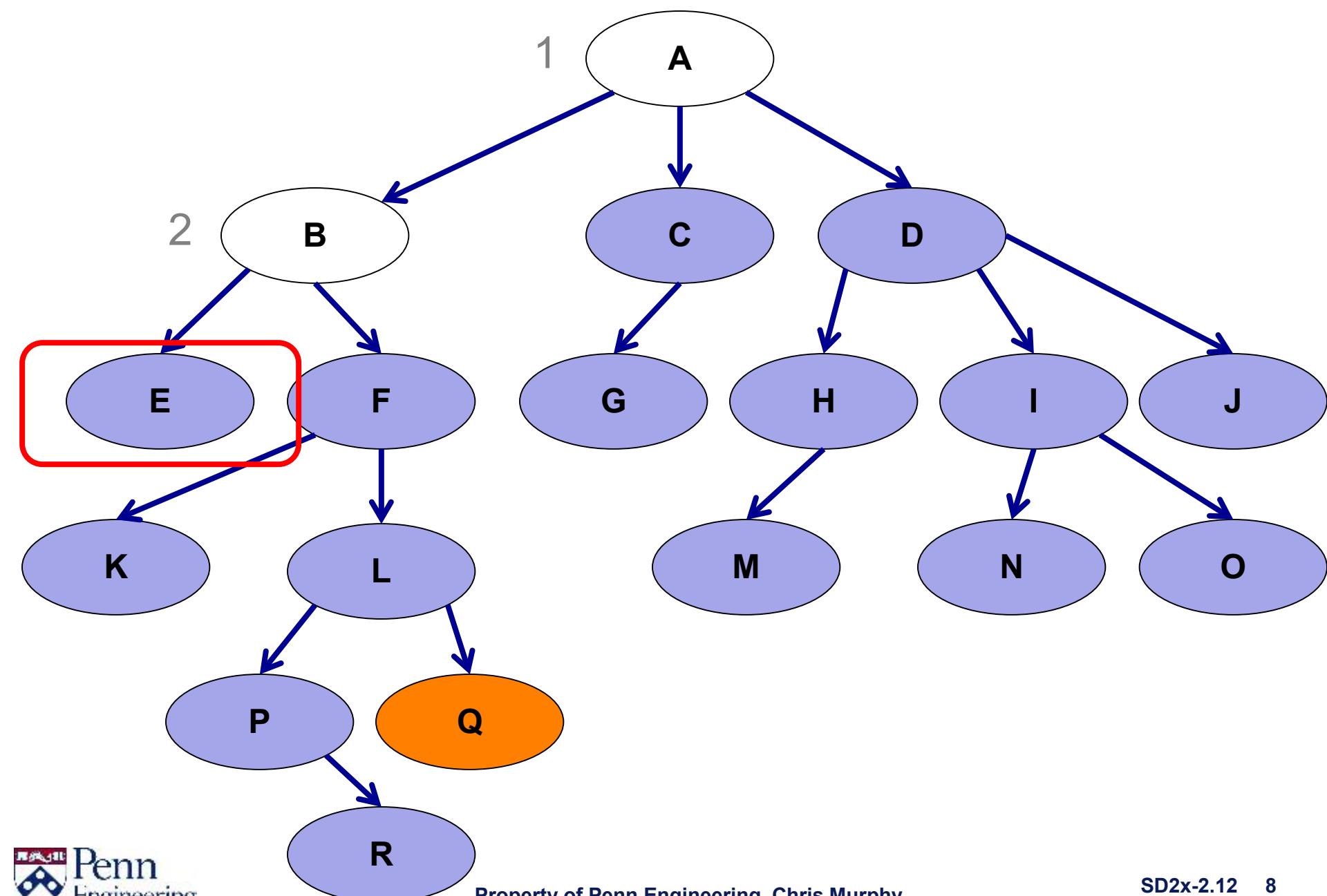


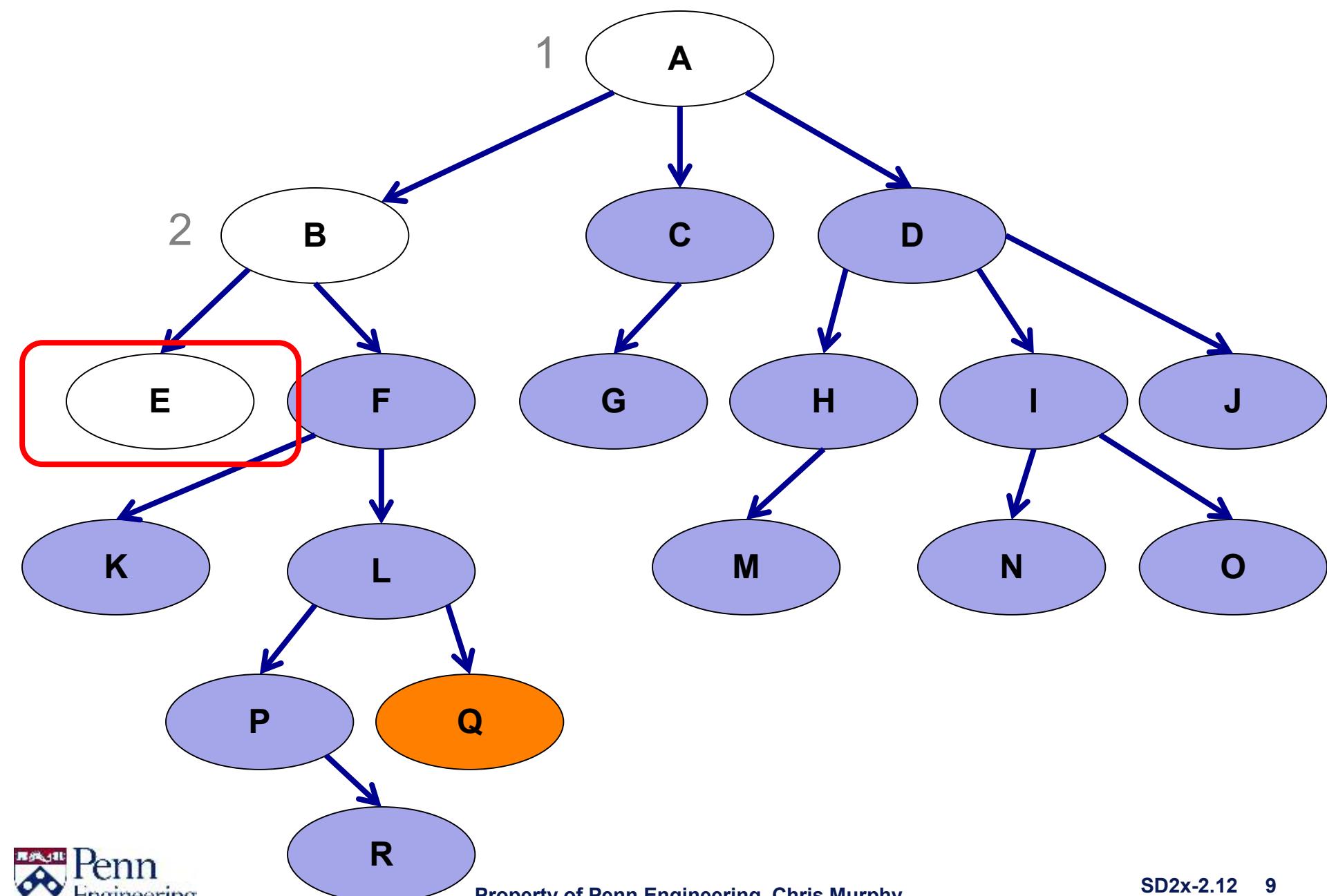


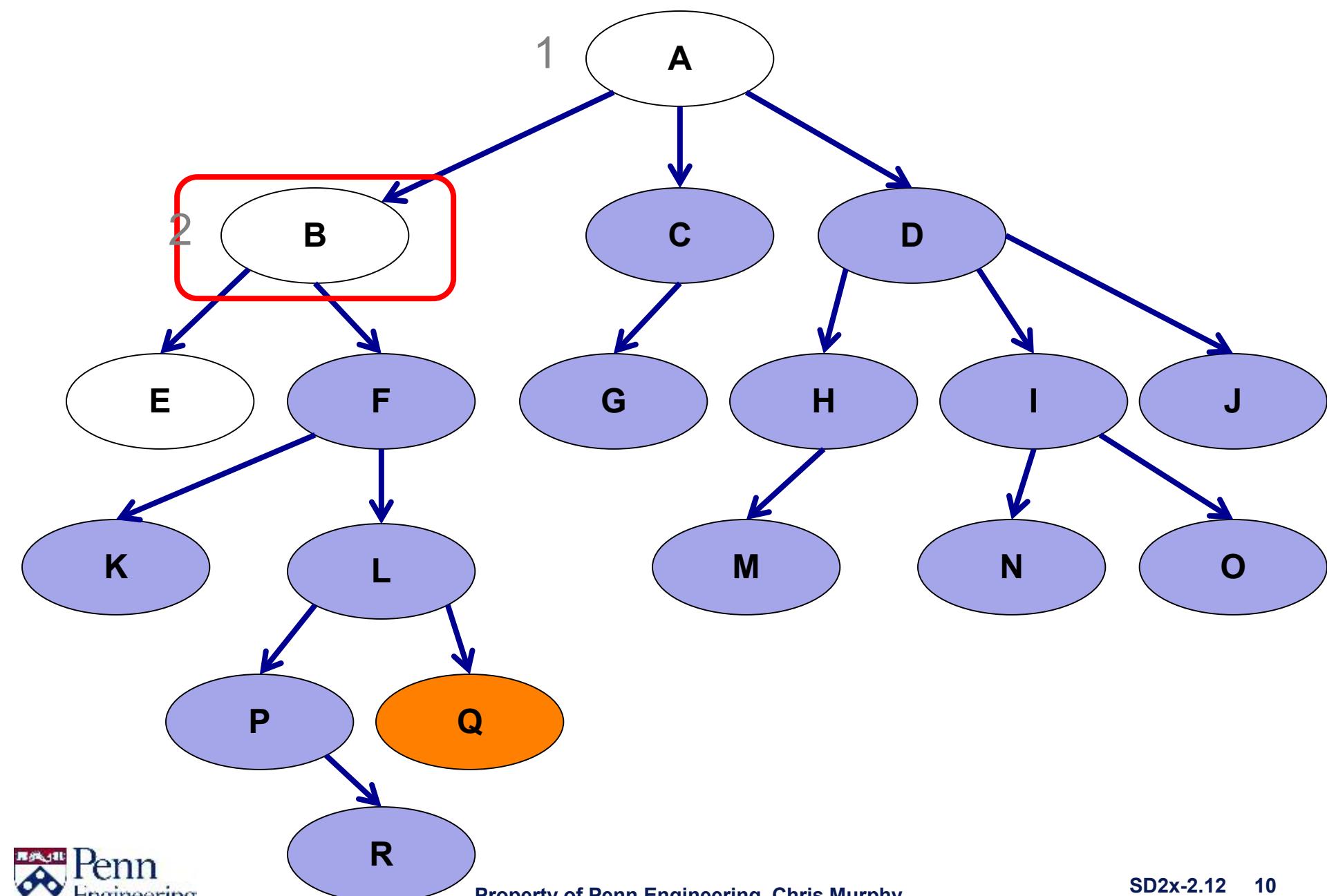


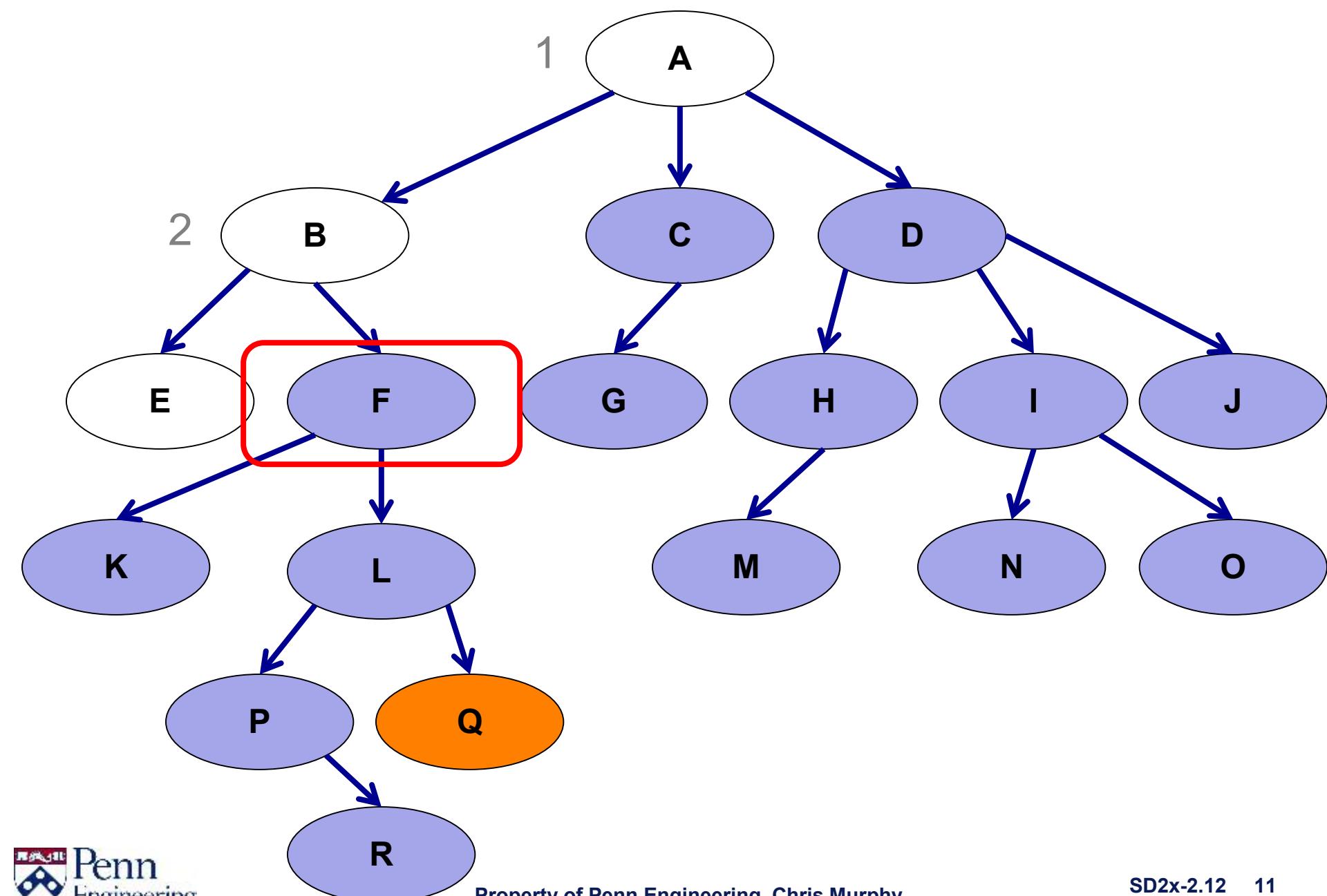


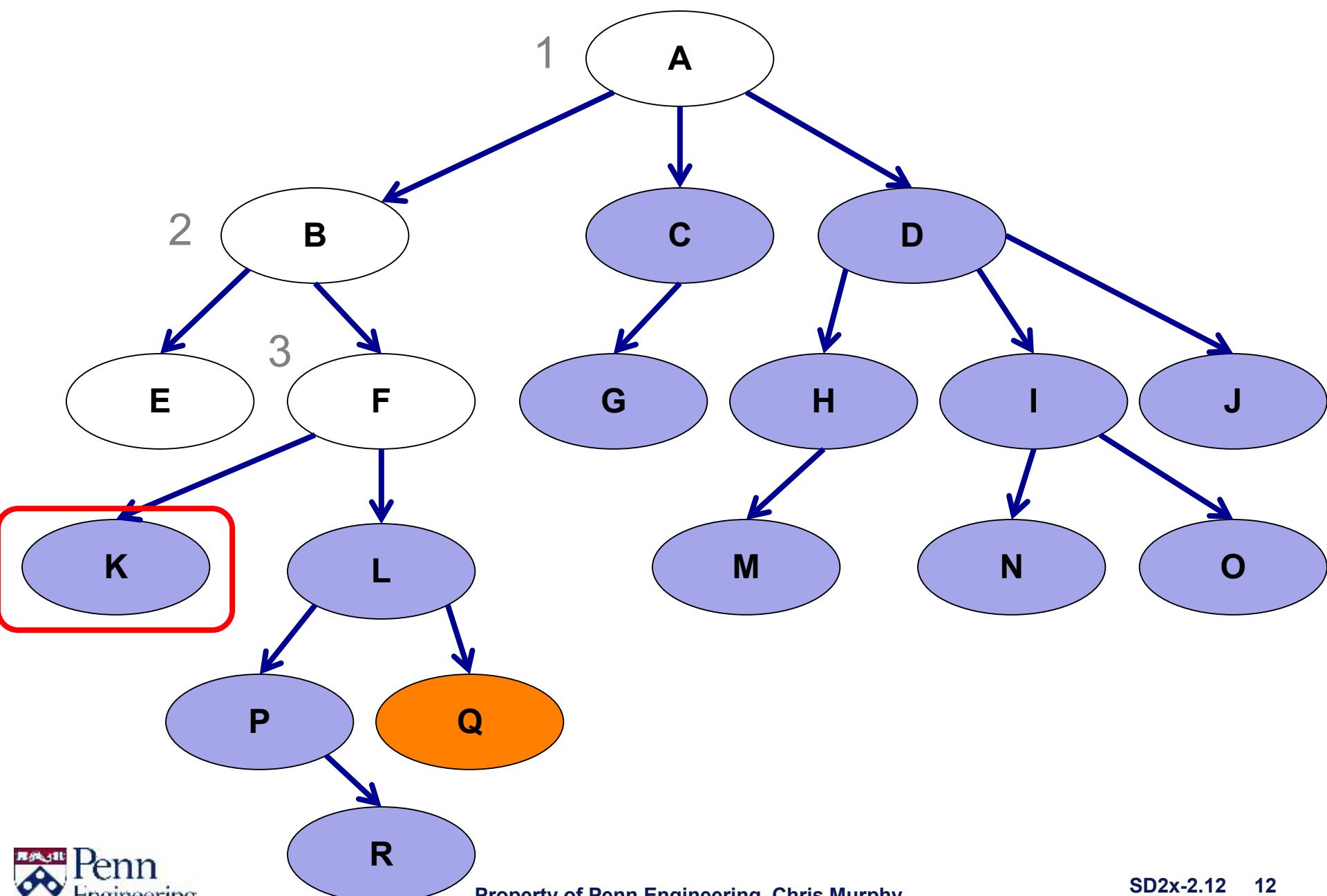


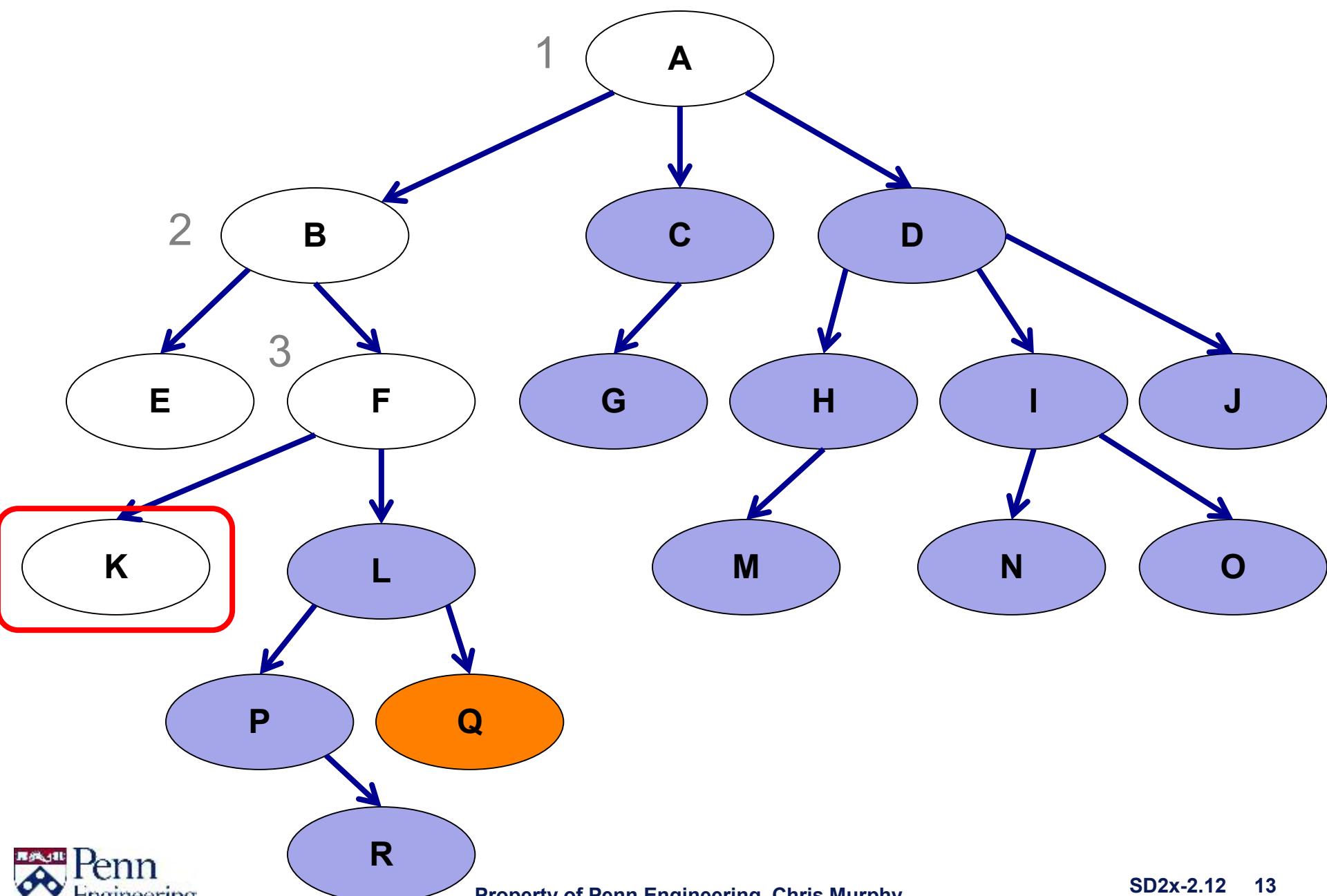


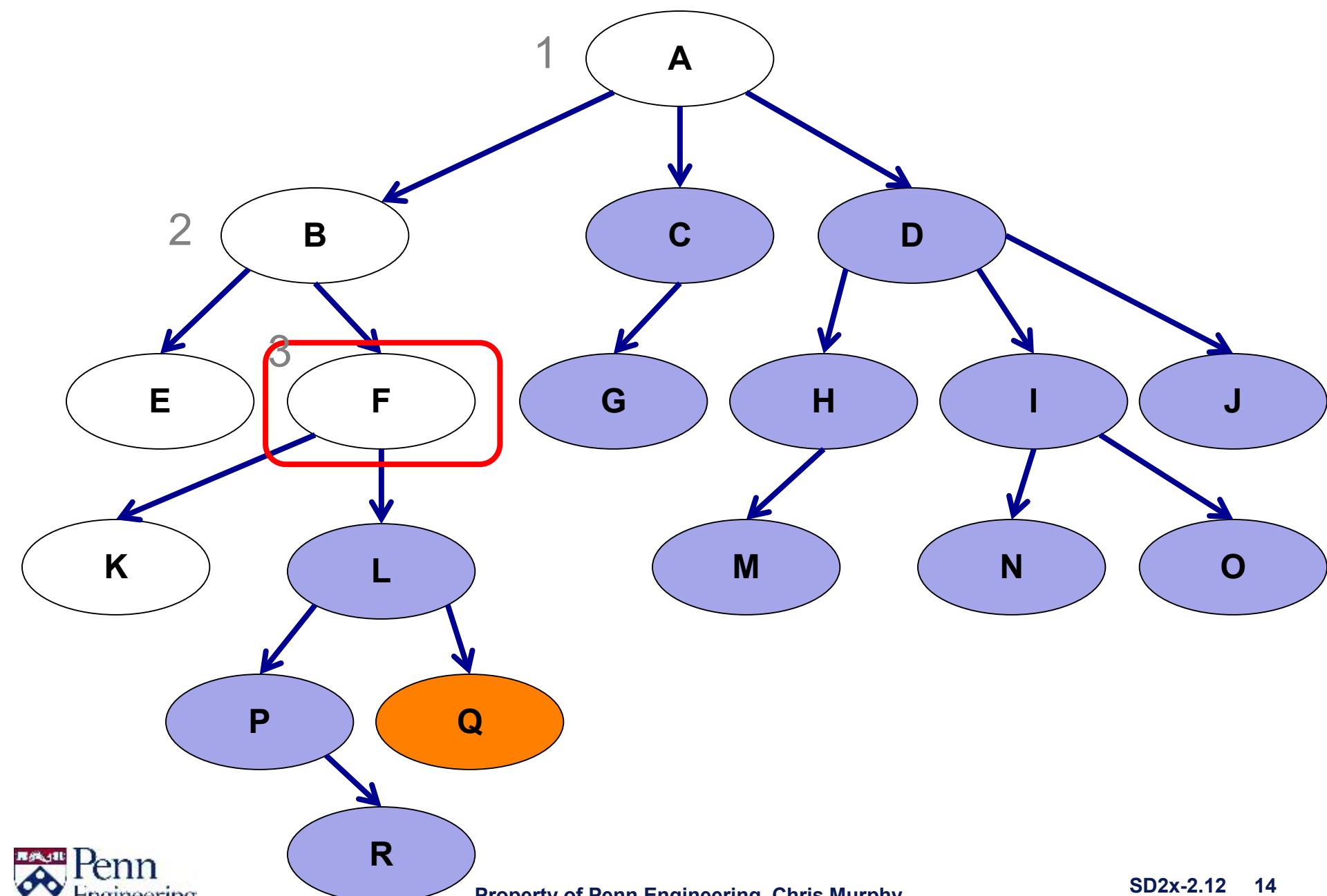


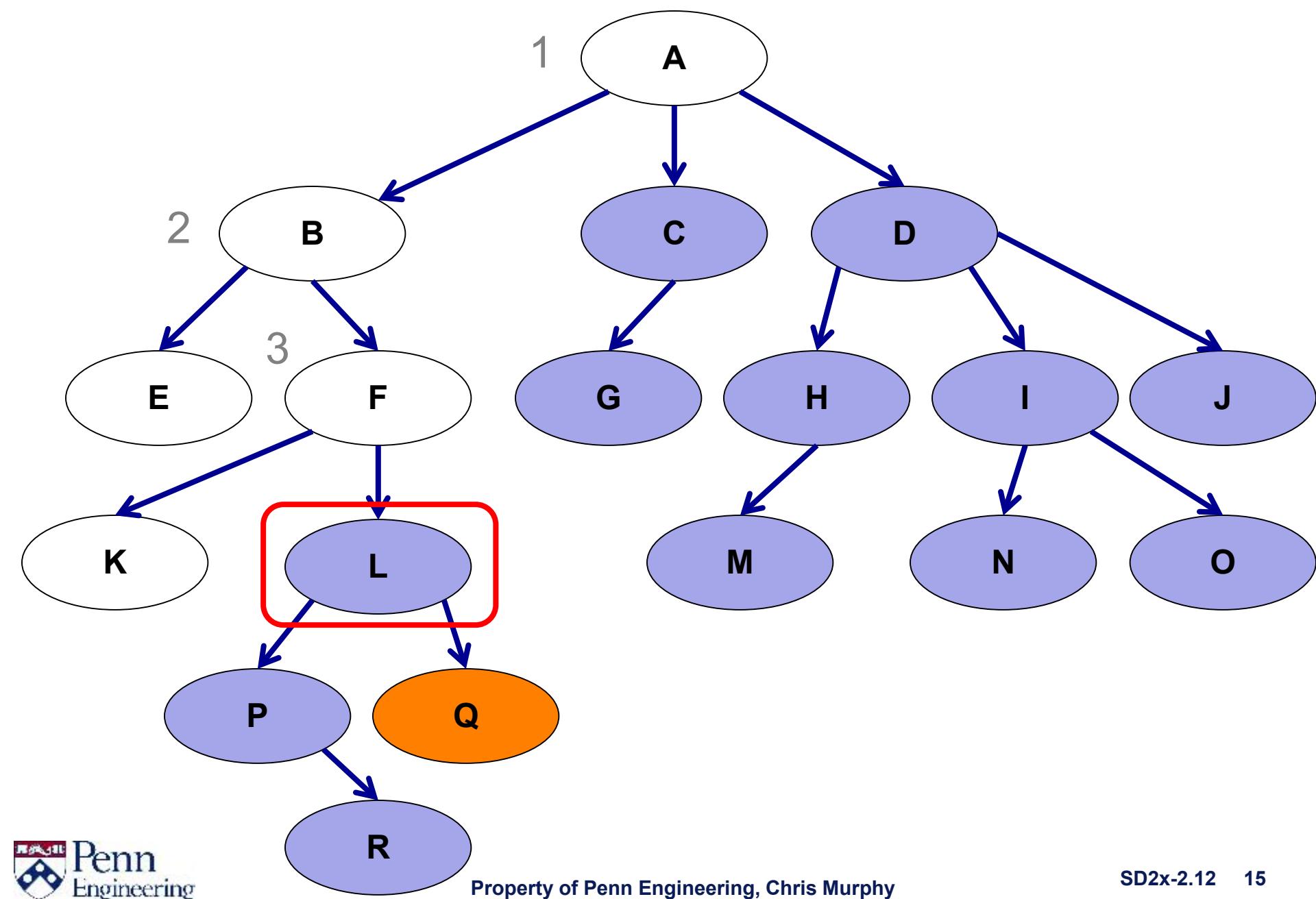


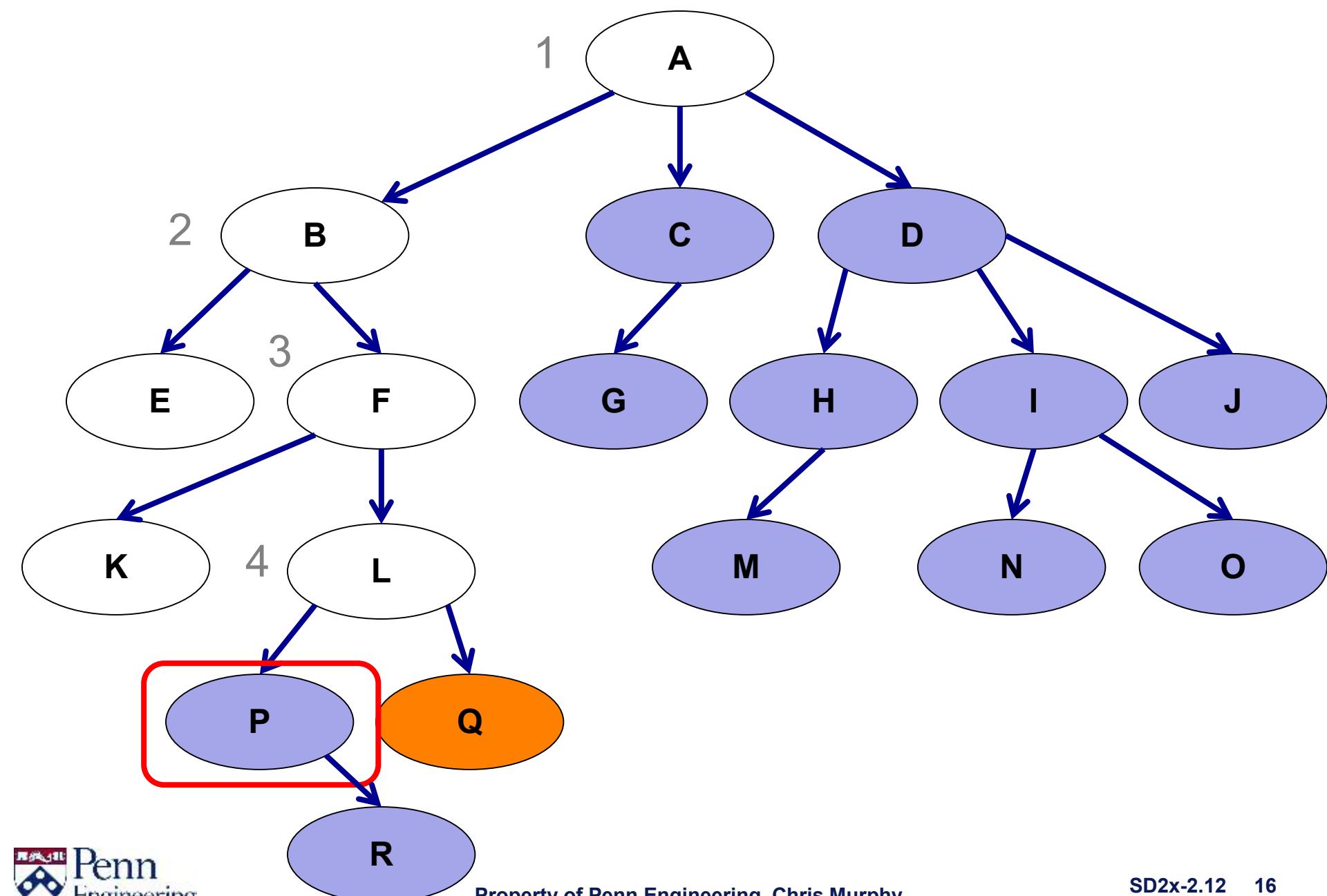


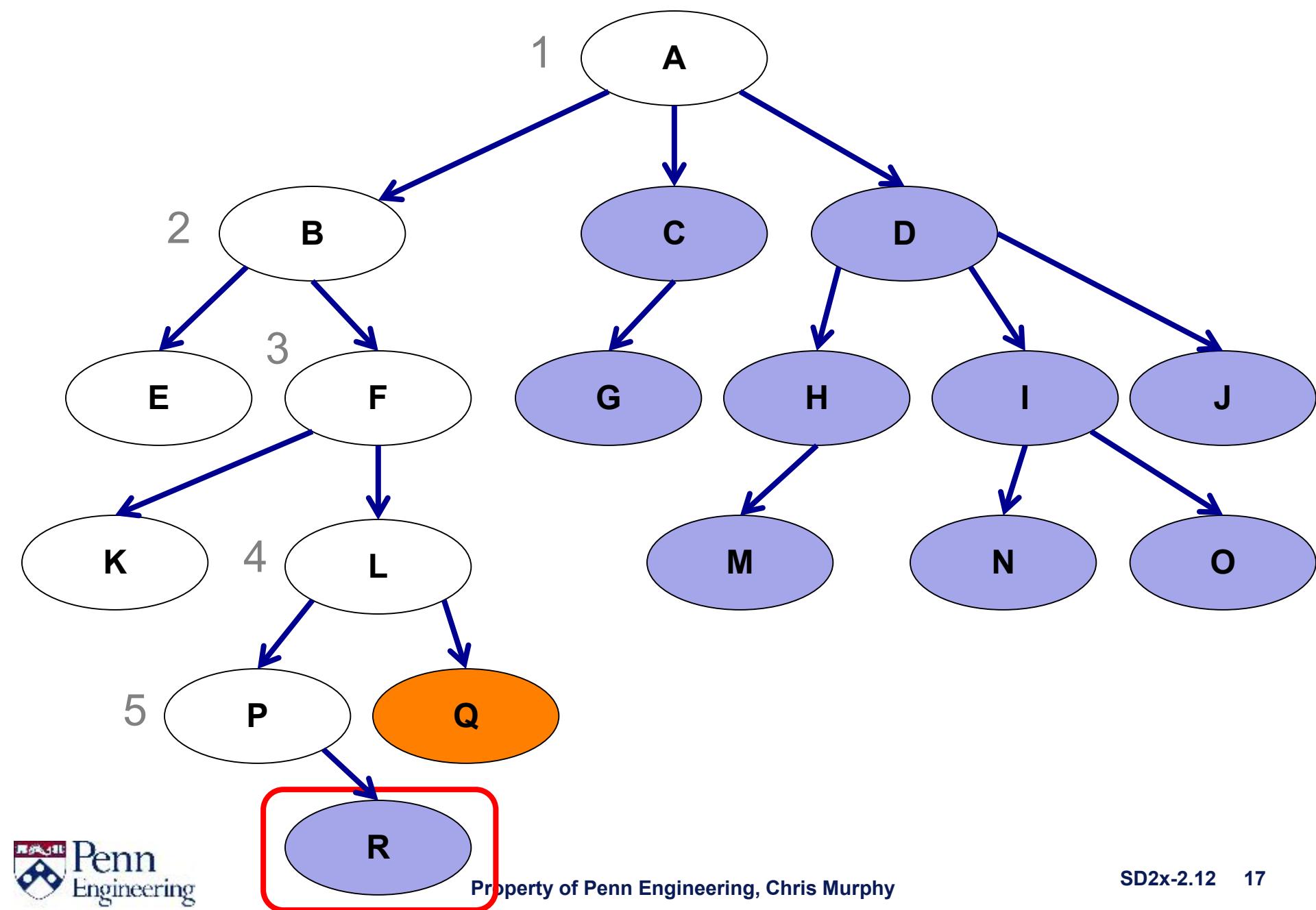


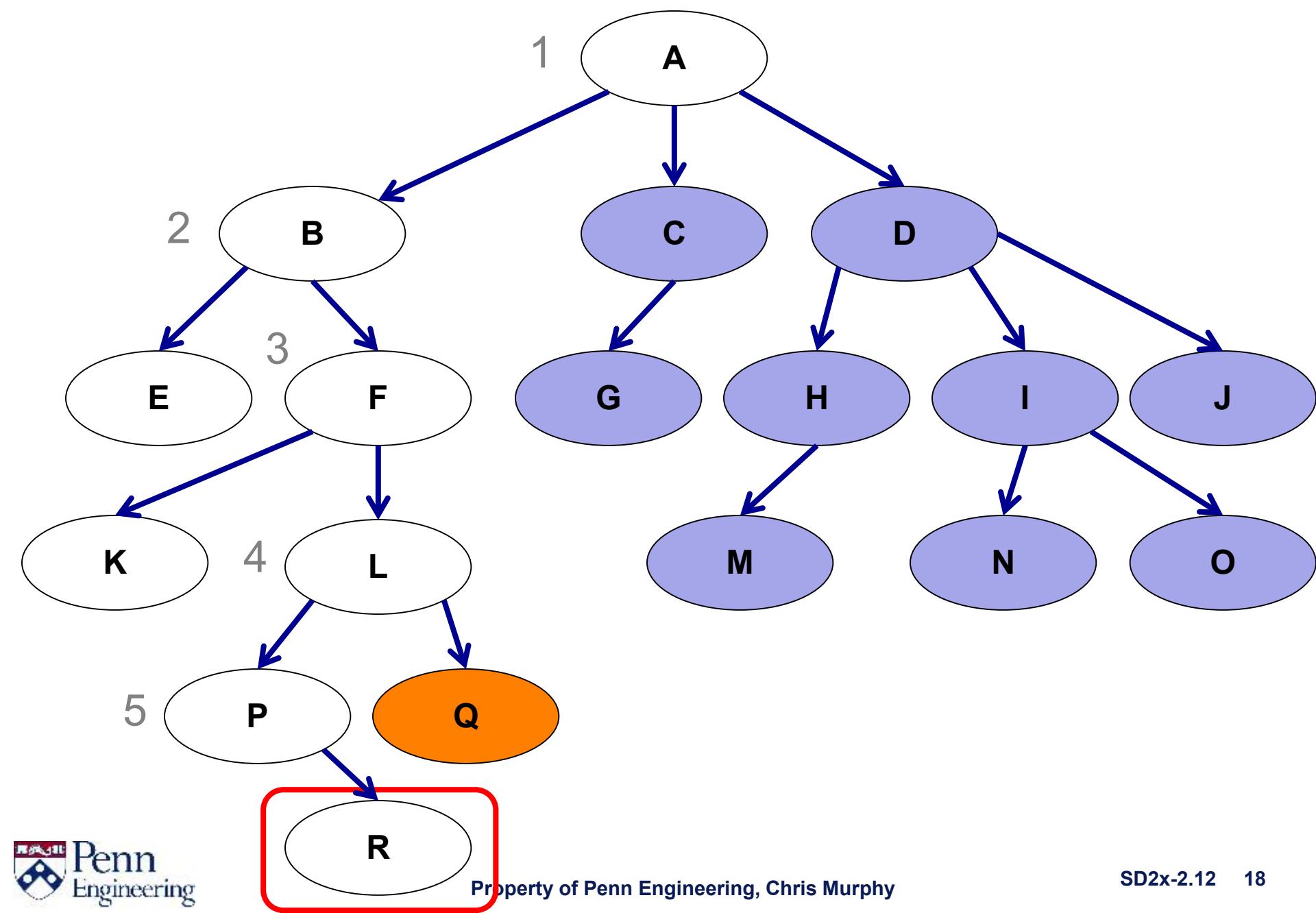


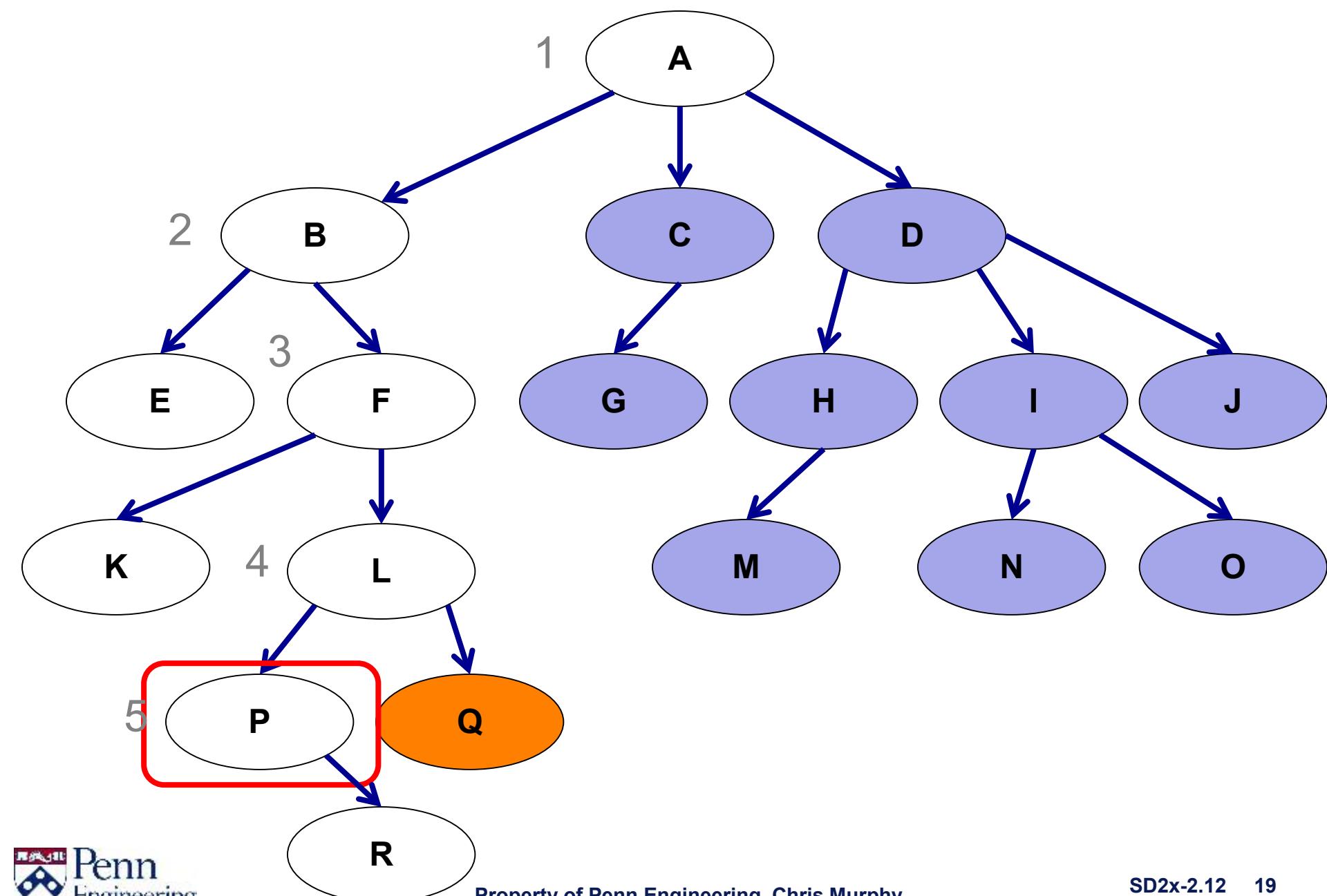


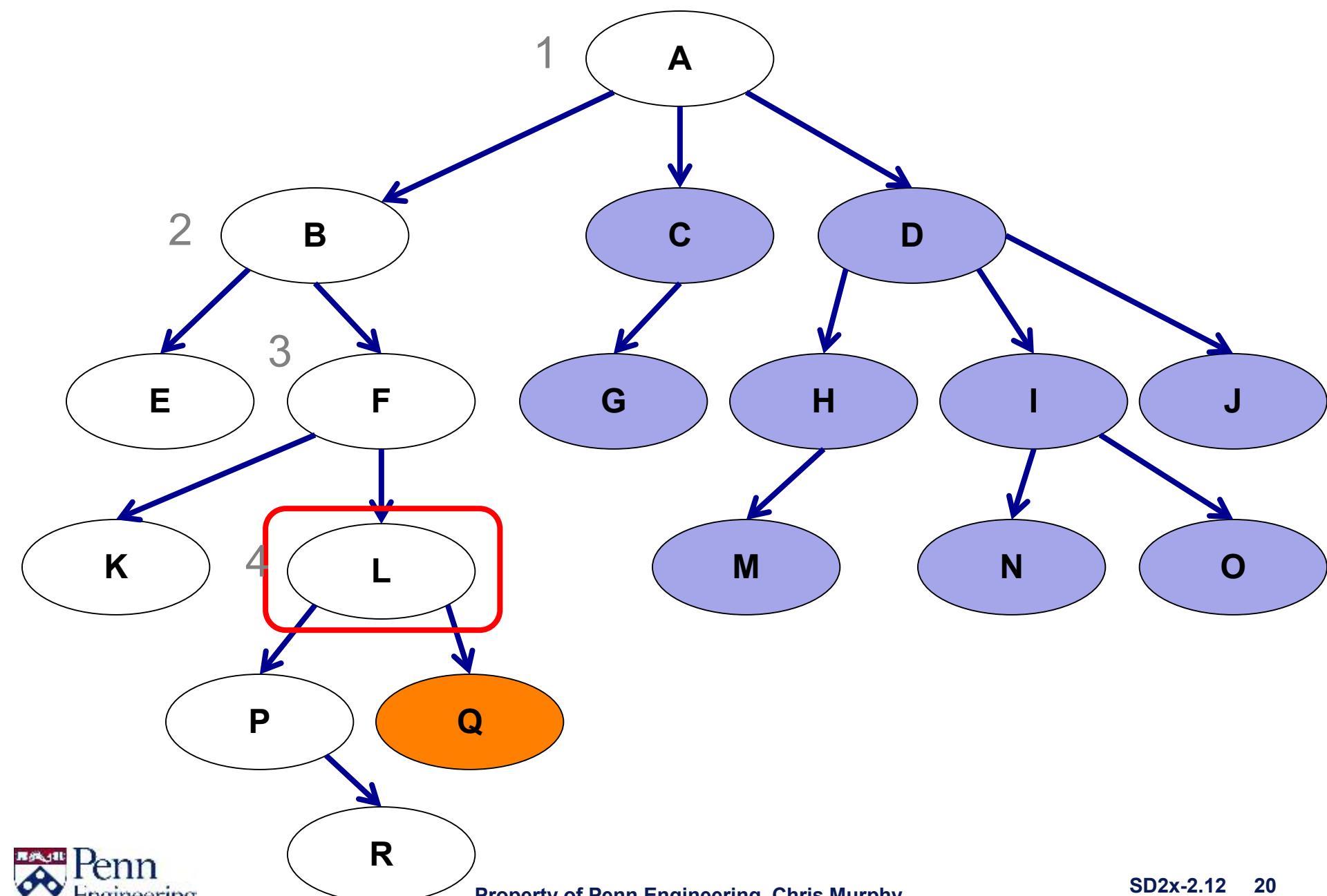


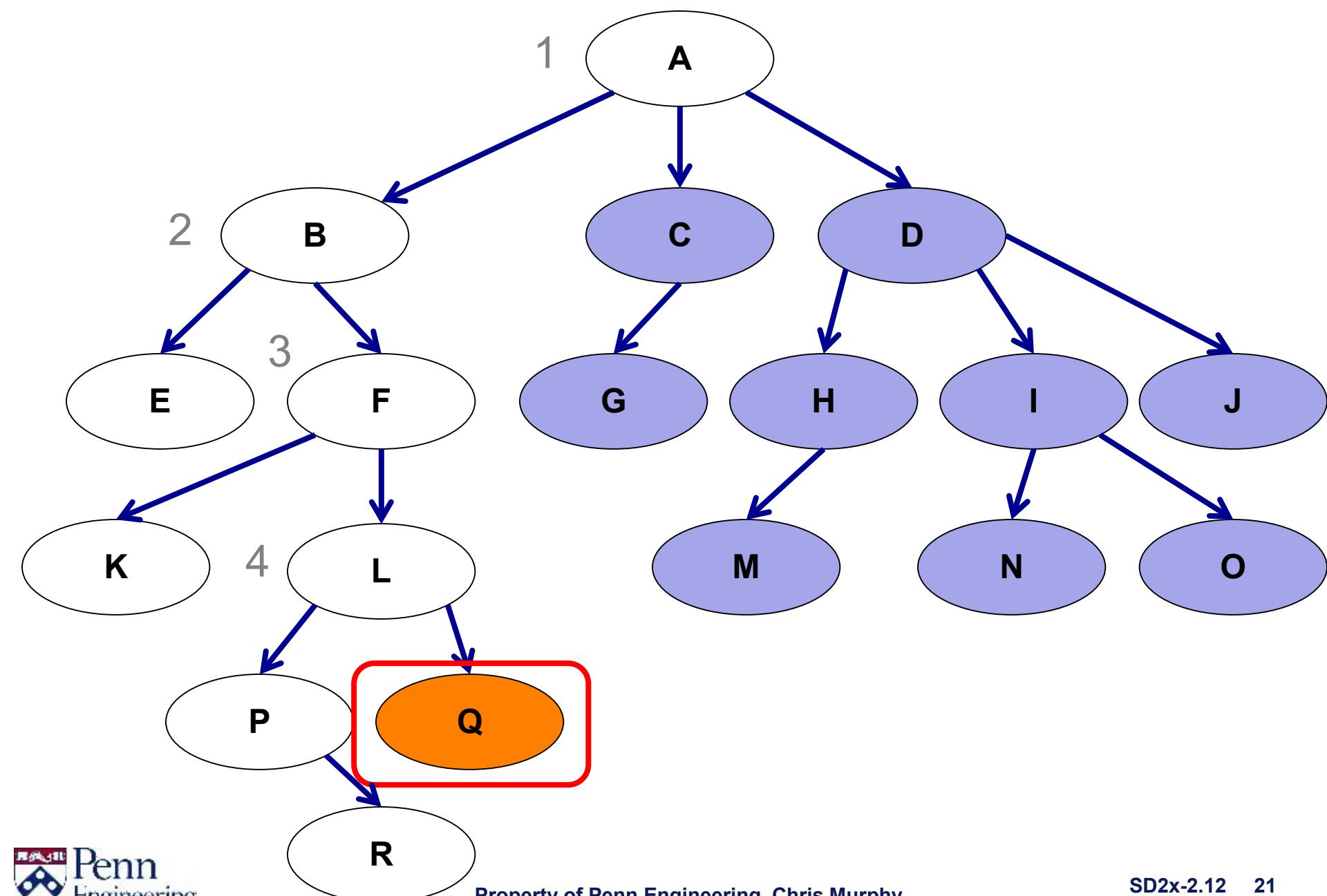


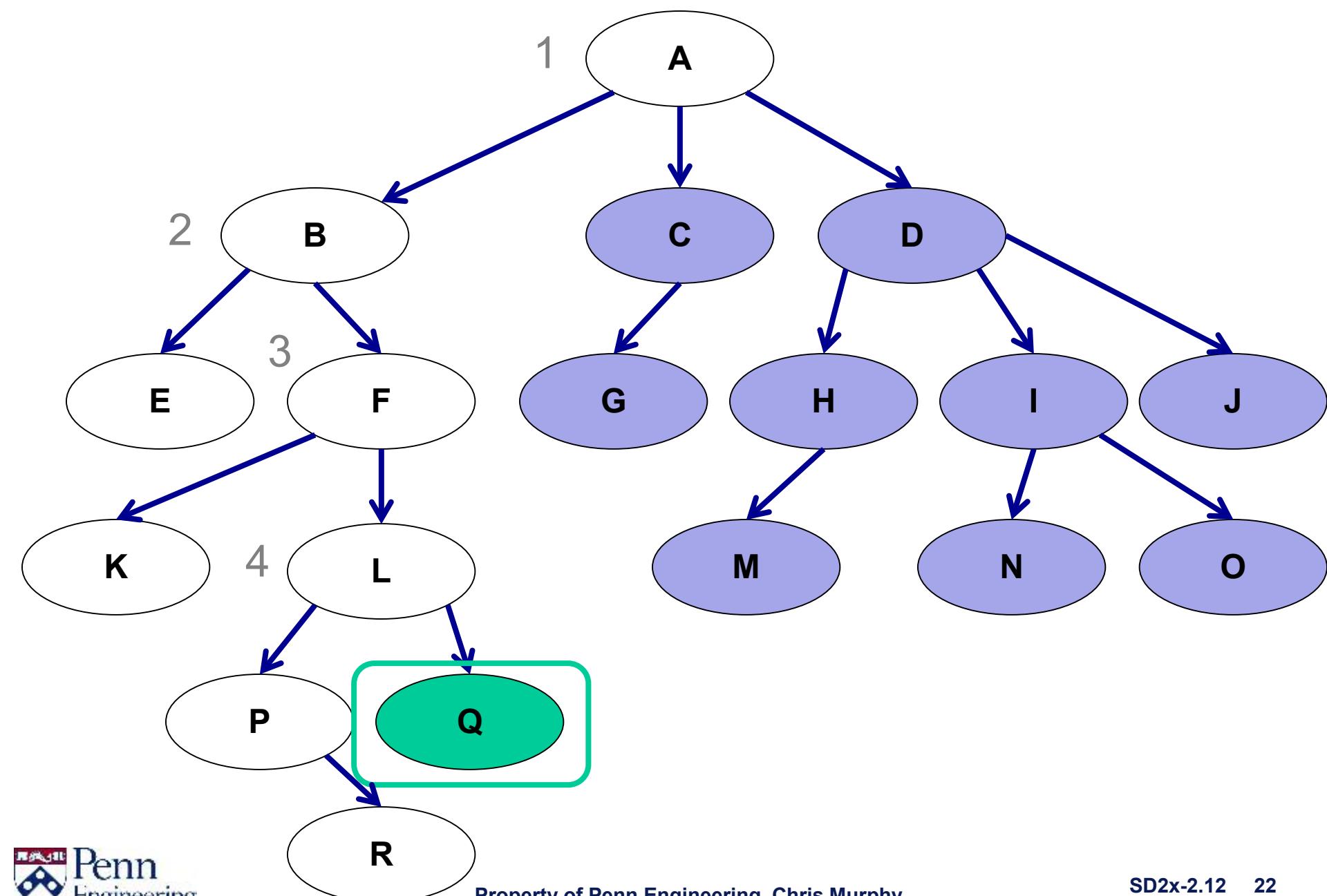












# Depth-first search

```
public class DepthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public DepthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean dfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class DepthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public DepthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean dfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class DepthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public DepthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean dfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class DepthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public DepthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
    public boolean dfs(String elementToFind, Node start) {  
        . . .  
    }  
}
```

```
public class DepthFirstSearch {  
    private Set<Node> marked;  
    private Graph graph;  
  
    public DepthFirstSearch(Graph graphToSearch) {  
        marked = new HashSet<Node>();  
        graph = graphToSearch;  
    }  
  
public boolean dfs(String elementToFind, Node start) {  
    . . .  
}  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

# Recap: Depth-first Search

---

- Recursive way of traversing a graph
- Start with a source node...
- then visit **one** of its neighbors...
- and so on, until destination is found

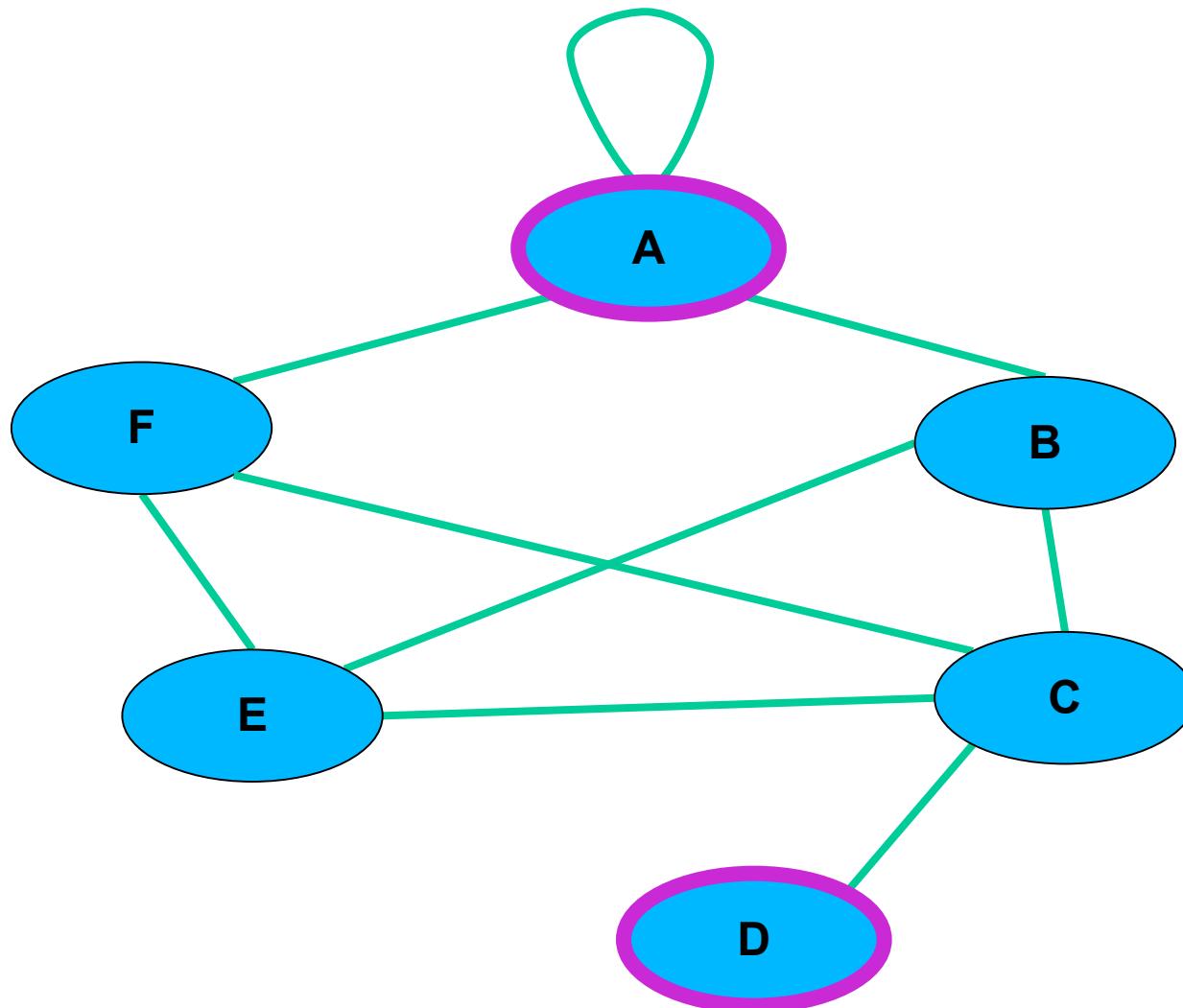
# DFS vs. BFS

```
public boolean dfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    } else {  
        marked.add(start);  
        for (Node neighbor : graph.getNodeNeighbors(start)) {  
            if (dfs(elementToFind, neighbor))  
                return true;  
        }  
        return false;  
    }  
}
```

```
public boolean bfs(String elementToFind, Node start) {  
    if (!graph.containsNode(start)) {  
        return false;  
    }  
    if (start.getElement().equals(elementToFind)) {  
        return true;  
    }  
Queue<Node> toExplore = new LinkedList<Node>();  
marked.add(start);  
toExplore.add(start);  
    while (!toExplore.isEmpty()) {  
        Node current = toExplore.remove();  
        for (Node neighbor : graph.getNodeNeighbors(current)) {  
            if (!marked.contains(neighbor)) {  
                if (neighbor.getElement().equals(elementToFind)) {  
                    return true;  
                }  
marked.add(neighbor);  
toExplore.add(neighbor);  
            }  
        }  
    }  
    return false;  
}
```

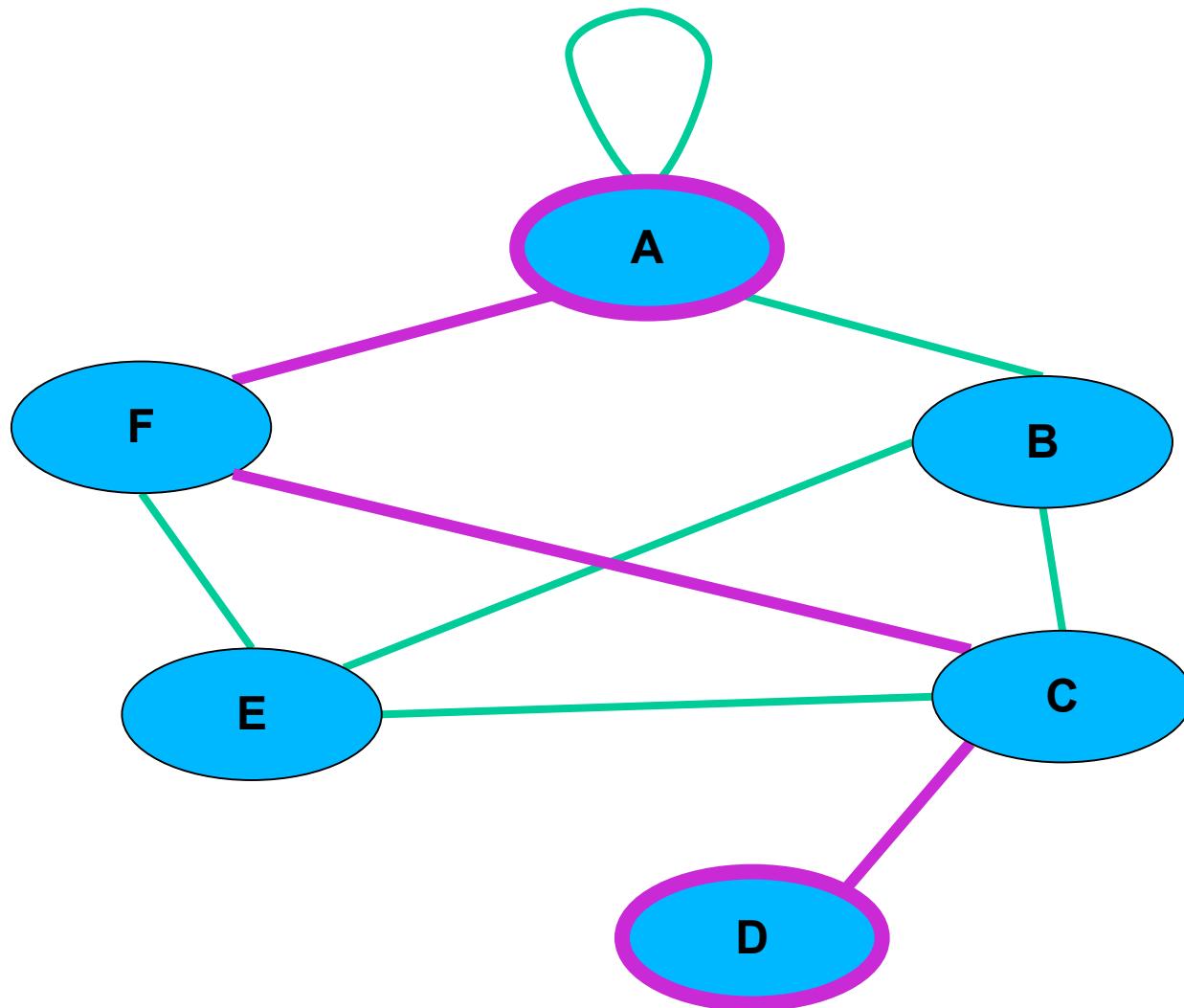
# Searching for D from A

---



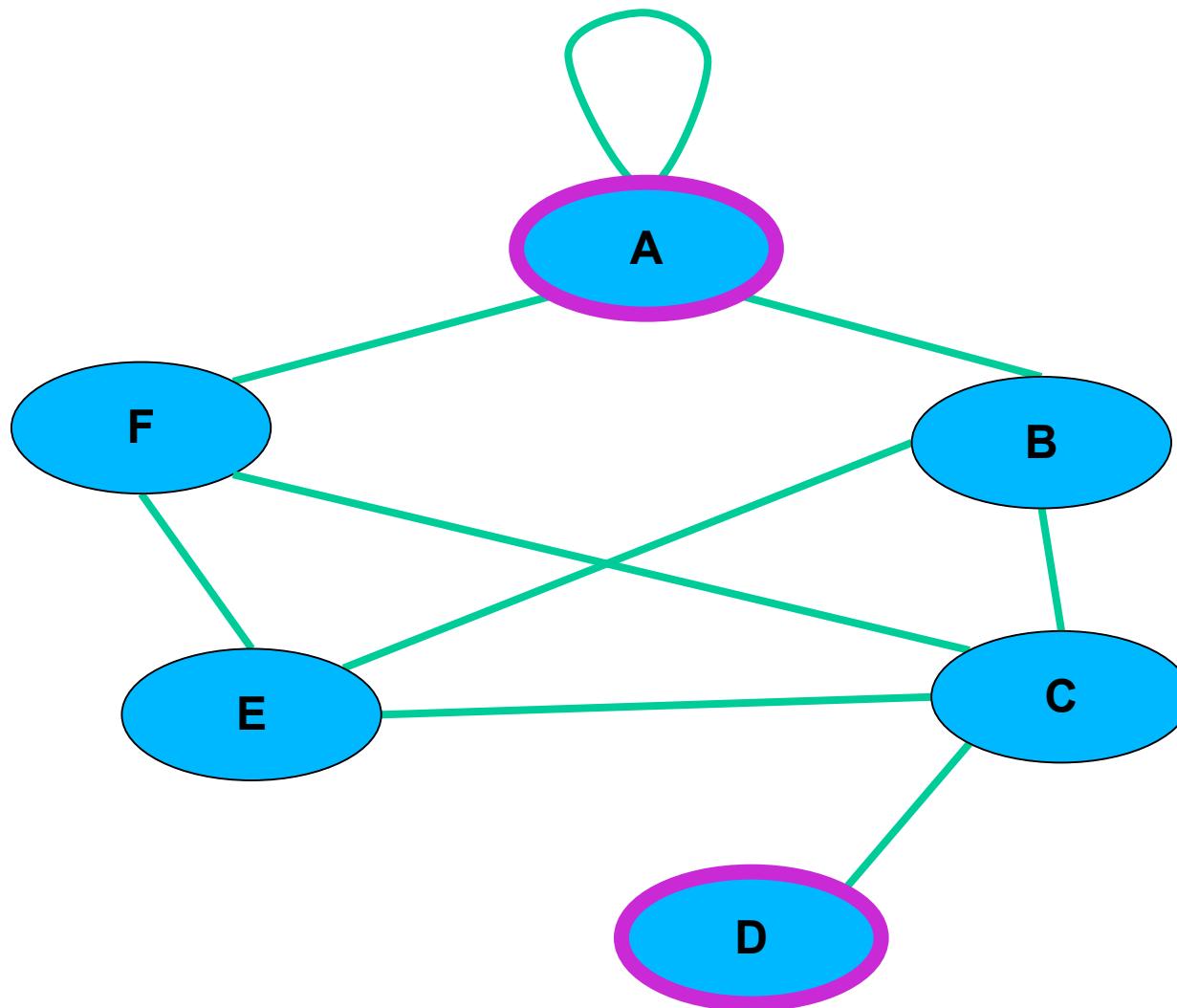
# Breadth-first search (BFS)

---



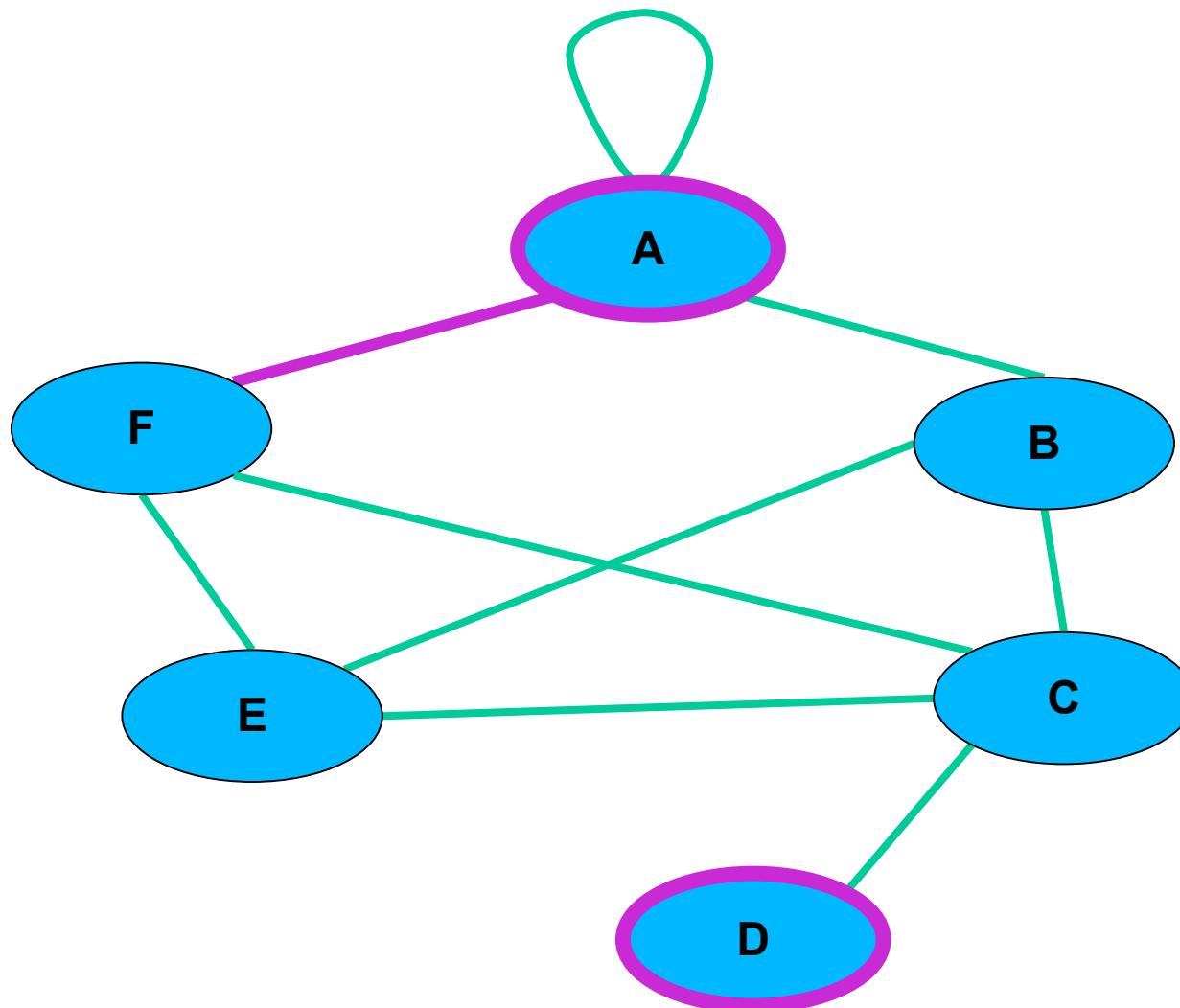
# Depth-first search (DFS)

---



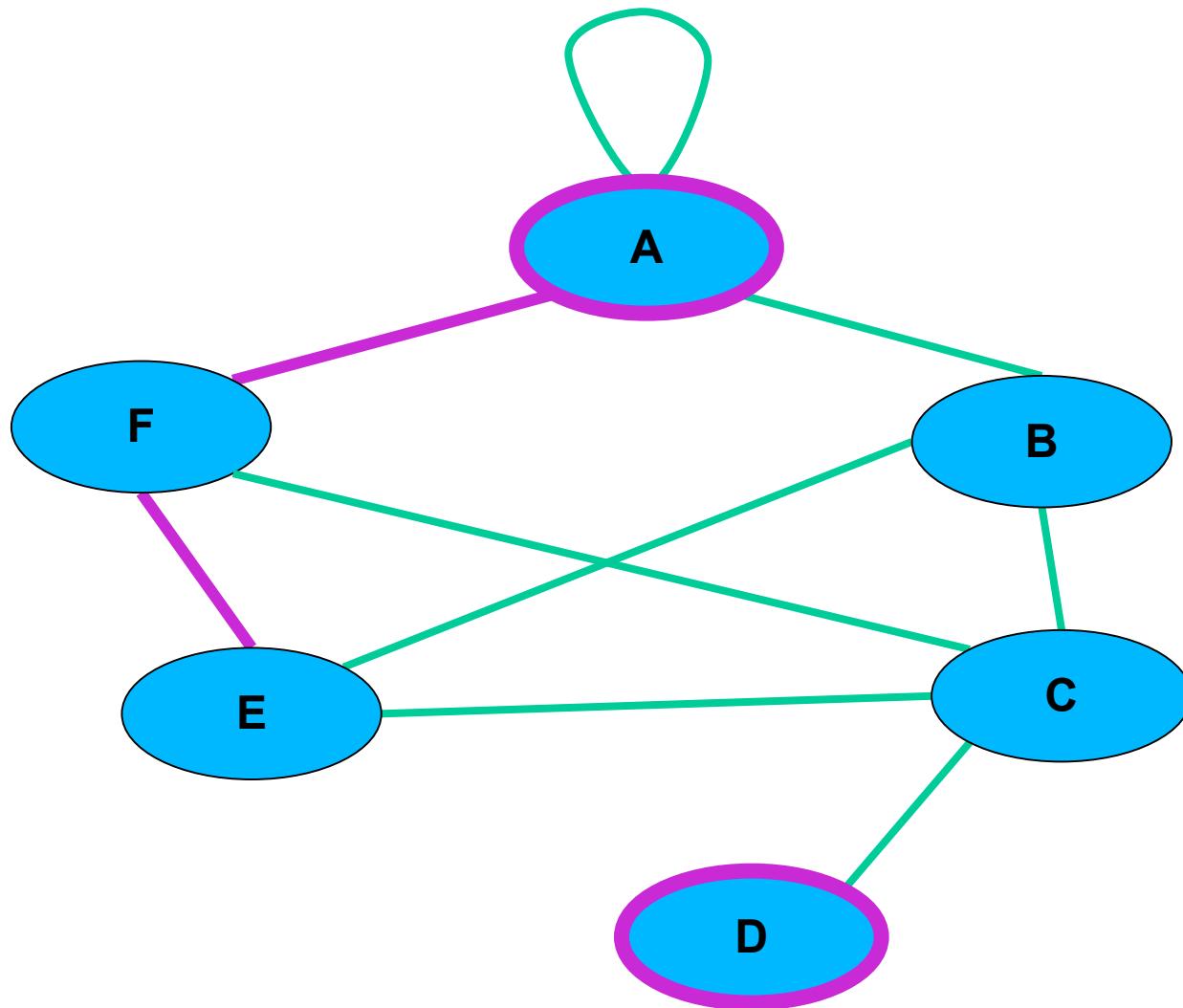
# Depth-first search (DFS)

---



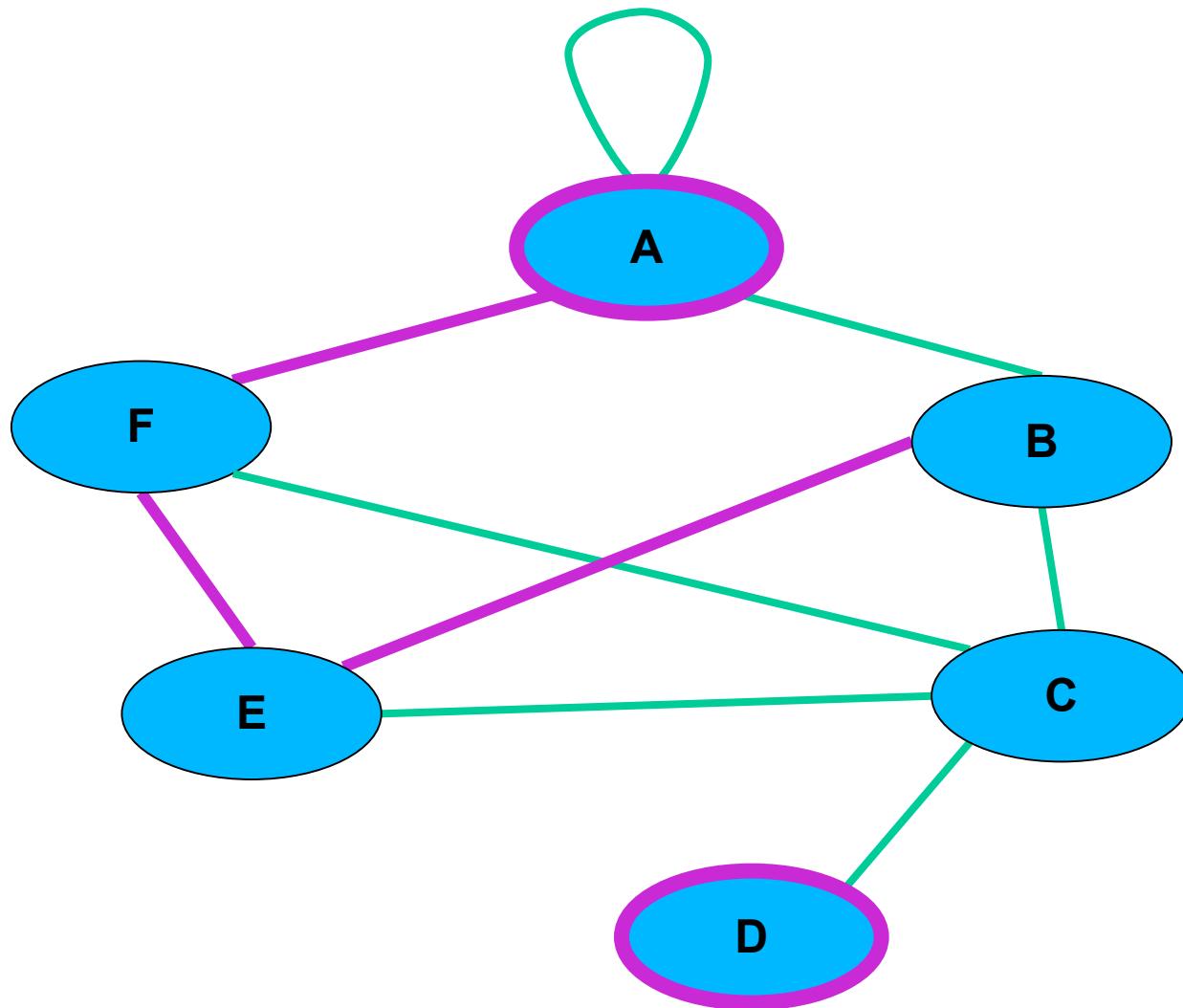
# Depth-first search (DFS)

---



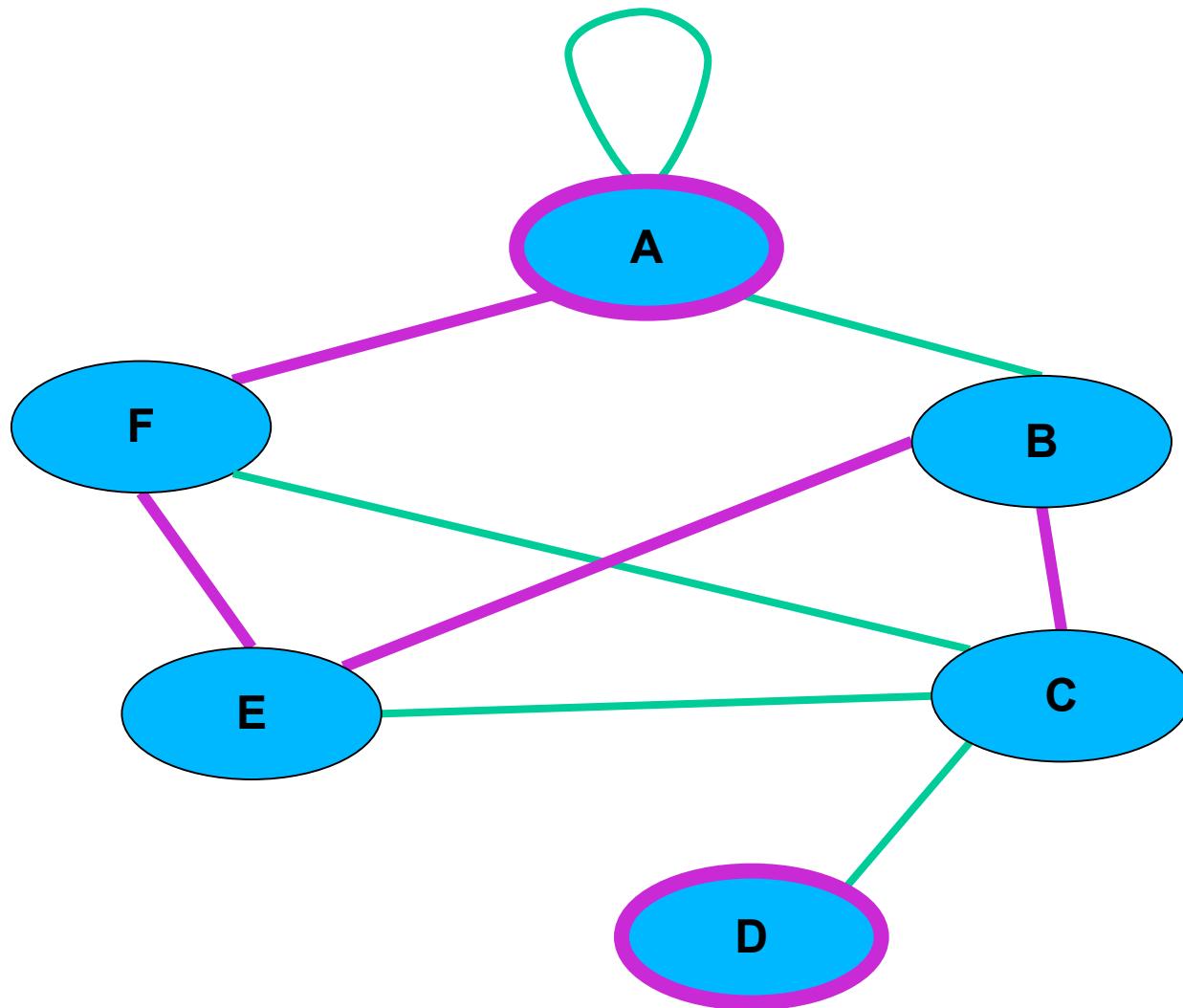
# Depth-first search (DFS)

---



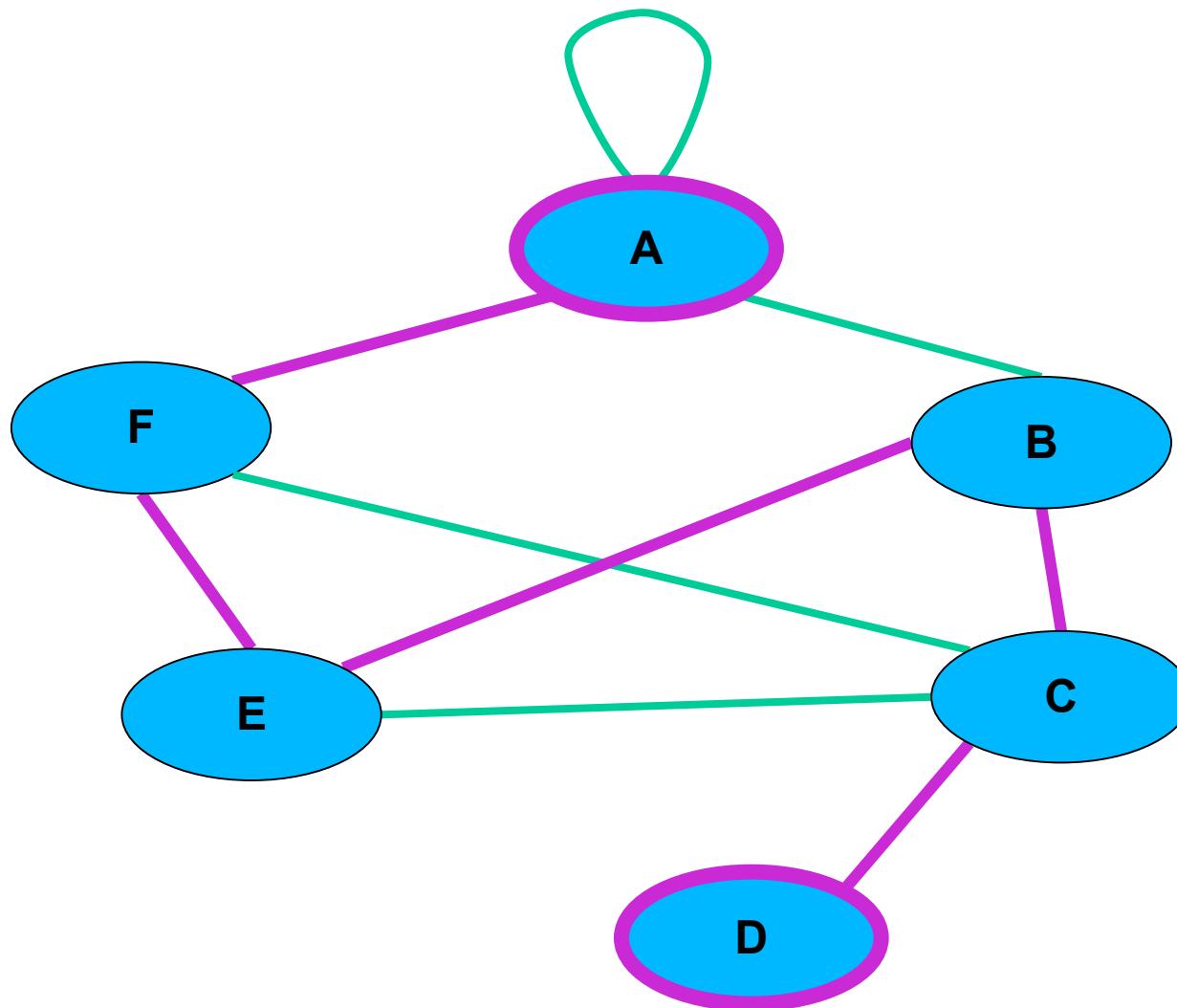
# Depth-first search (DFS)

---



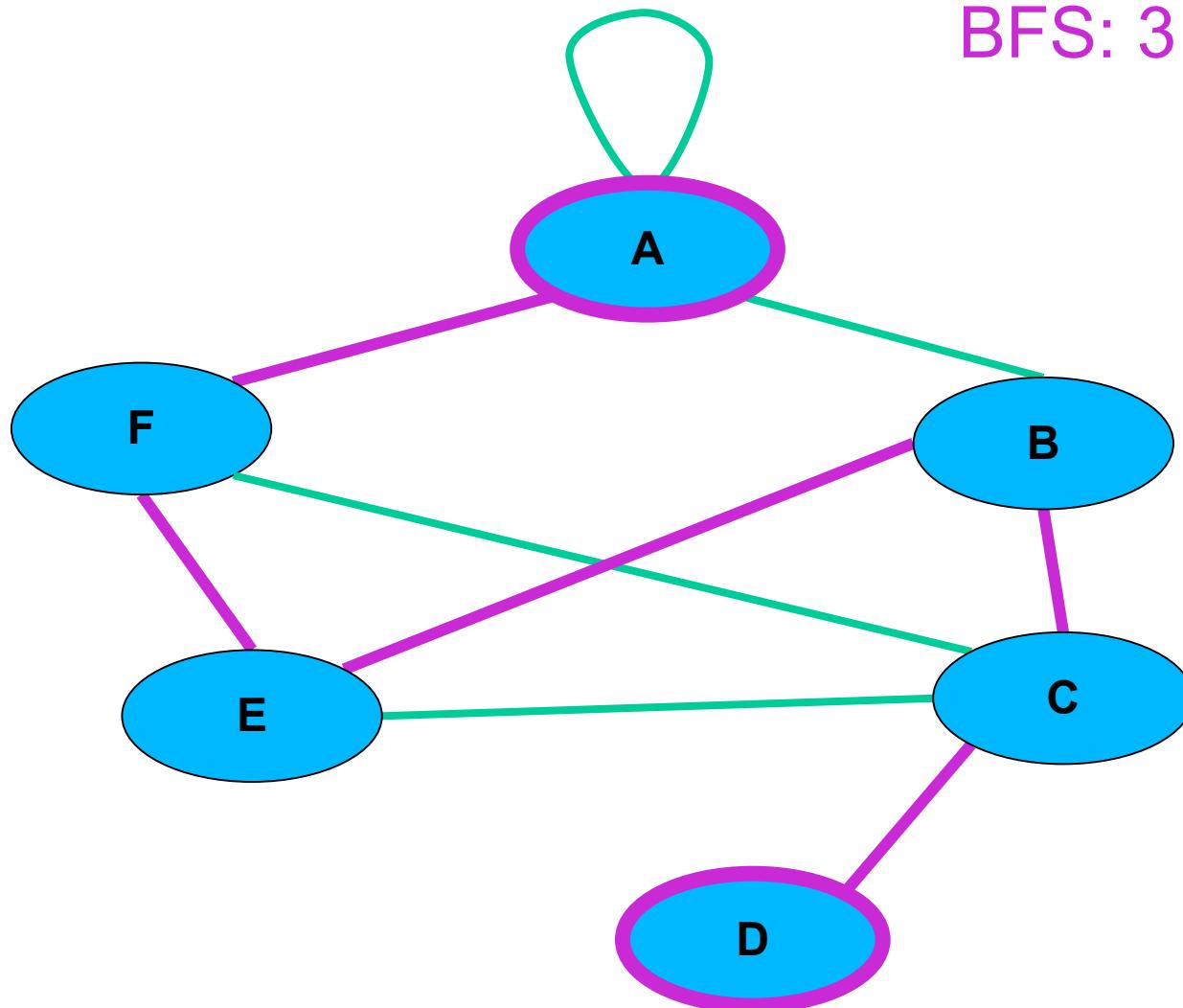
# Depth-first search (DFS)

---

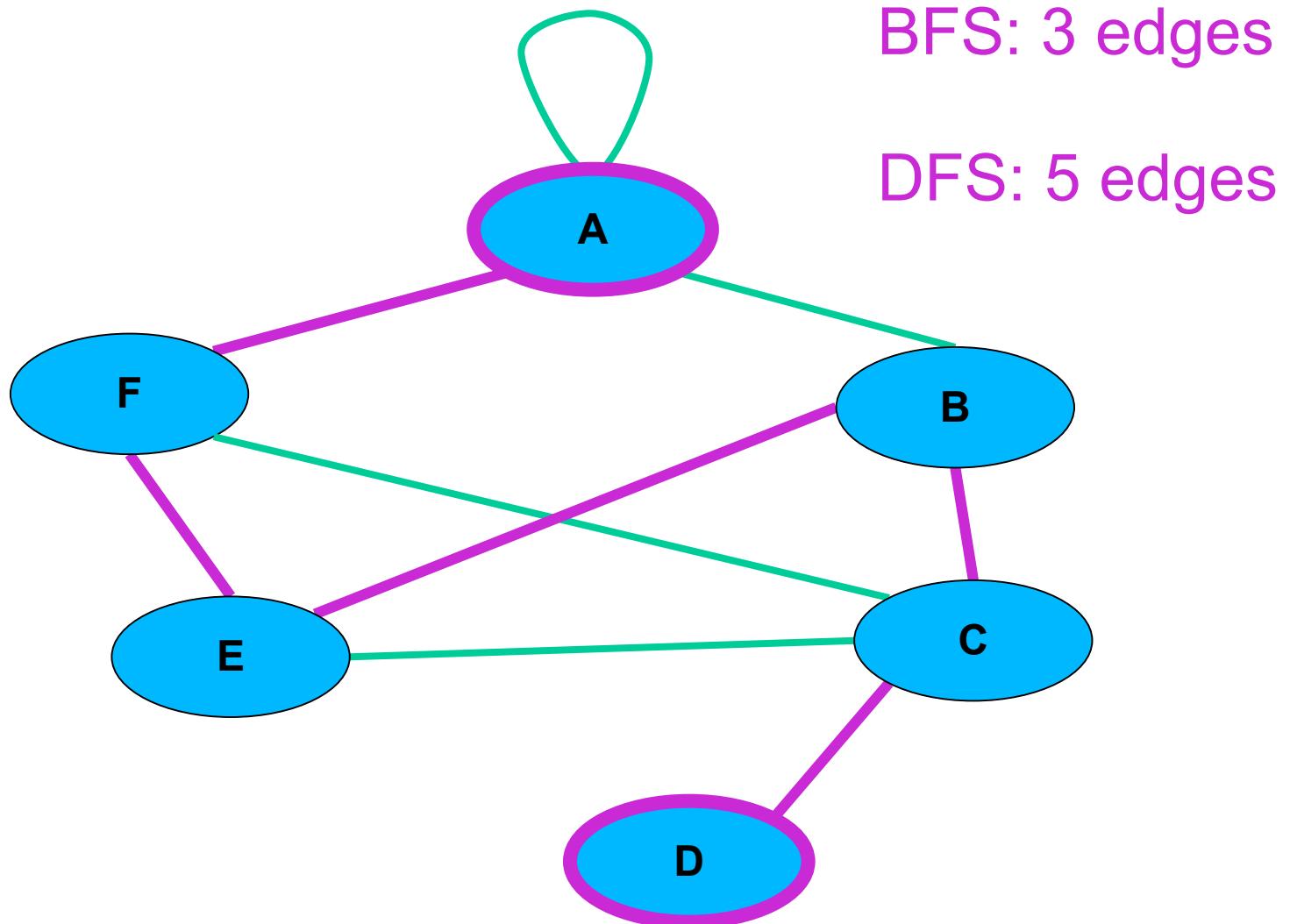


# Depth-first search (DFS)

BFS: 3 edges



# Depth-first search (DFS)



# Recap: BFS and DFS

---

- BFS and DFS are alternative mechanisms for traversing a graph
- DFS is likely to be more efficient
- BFS will find the minimum number of edges
- Other algorithms are used to solve other problems, e.g. shortest path