Data Structures, Part 2

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Goals for today

- Review of arrays and lists
- Hash tables
- Trees and searches

REVIEW:
ARRAYS AND LISTS

Data structures

- Programs need to store data
- Best way to store data *depends on **how** data*:
  - Is **written** to the data structure
  - Is **read** from the data structure
  - Is **modified** in the data structure
- Consider needs of a program when choosing the most appropriate data structure

Abstract vs. concrete

- **Abstract** data type:
  - *Define characteristics and operations* for the data structure
  - May not guarantee any performance requirements
- **Concrete** data type:
  - The data structure is *defined by its implementation*
  - Has specific performance measurements
- An *abstract data type* may be implemented using more than one *concrete data types*

Pointers and memory

```
>>> a = 3
>>> b = a
>>> c = 3
```

- Variables point to a *location* in **memory** where an object is stored
- Different *types* of objects require different **space** in memory
- E.g., a **double** *float* is typically 64-bit (i.e., requires 8 bytes in memory)

Memory

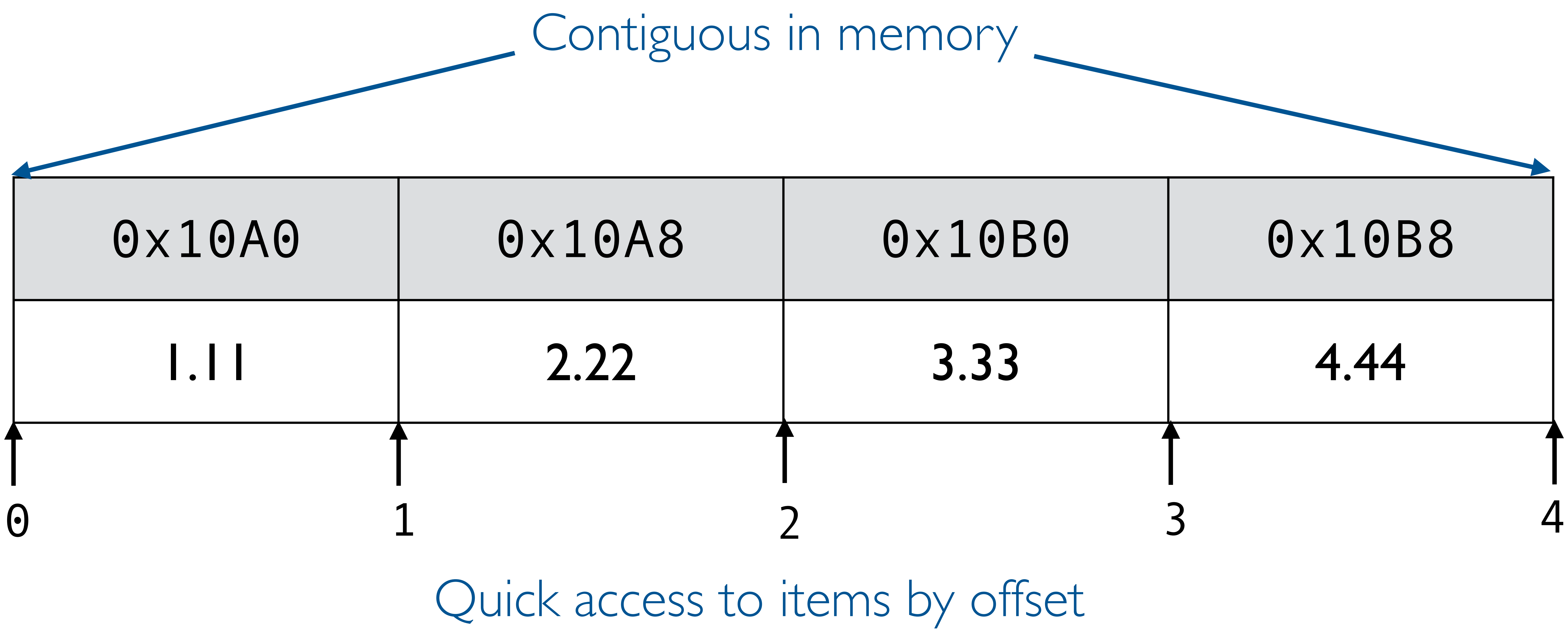
	Address	**Value**
a	0x1000	3
	0x1001	
b	0x1002	
	0x1003	
c	0x1004	3

Arrays

- **Ordered sequence** data type
- Stored in a single *block of memory*
- Items are stored ***contiguously*** in memory
- All items must be the ***same data type***

Arrays in Python

```python
import array as arr

x = arr.array("d", [1.11, 2.22, 3.33, 4.44])
```



Performance of arrays

- Very fast random read/write of existing items
- Very fast traversal of items (contiguous in memory)
- Somewhat slow searching for specific items
- Very slow insertion/deletion of new items

Appending to an array in Python

```
x = arr.array("d", [1.11, 2.22, 3.33, 4.44])

x.append(5)
```

Need to allocate a new block of memory

Old

0x10A0	0x10A8	0x10B0	0x10B8
1.11	2.22	3.33	4.44

Copy
old elements
to new array

New

0x20A0	0x20A8	0x20B0	0x20B8	0x20C0
1.11	2.22	3.33	4.44	5.0

Considerations for data structures

- What performance characteristics are needed?
  - **Read/write** (of existing items)
  - **Insertion/deletion** (of new items)
  - **Traversal** (iteration over all items)
  - **Searching** (find a specific item)
- Memory space requirements

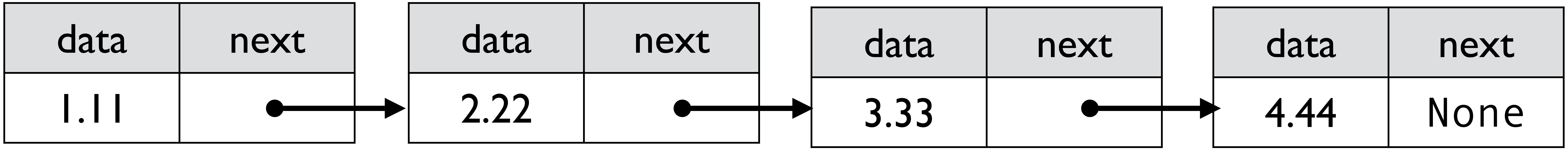
Linked lists

- **Ordered sequence** data type
- Items stored in *linked nodes*
- Nodes stored ***non-contiguously*** in memory
- May be heterogenous (different data types)

Singly-linked lists

- Linked lists are a chain of nodes
- Each node stores data and points to next node

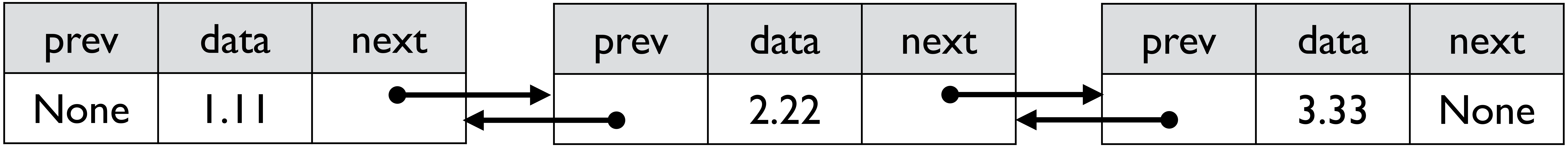
`(1.11 · (2.22 · (3.33 · (4.44 · None))))`



Nodes are typically *not* contiguous in memory

Doubly-linked lists

- Nodes also point to the *previous* node
- Traverse list in either direction
- Link first and last node to make list *circular*



Uses additional memory for increased flexibility

Linked lists in Python

```
class LList:

    def __init__(self):
        self.head = None

    def append(self, value):
        newcell = CONS(value, None)
        if self.head is None:
            self.head = newcell
        else:
            tail = self.head
            while tail.getrest() is not None:
                tail = tail.getrest()
            tail.setrest(newcell)
```

Linked lists in Python

```
class LList:

    def __init__(self):
        self.head = None

    def append(self, value):
        newcell = CONS(value, None)
        if self.head is None:
            self.head = newcell
        else:
            tail = self.head
            while tail.getrest() is not None:
                tail = tail.getrest()
            tail.setrest(newcell)
```

Nodes are cons cells

Traverse list to append item at end

Performance of linked lists

- Very slow random read/write of existing items
- Fast traversal of items (non-contiguous in memory)
- Somewhat slow searching for specific items
- Fast insertion/deletion of new items
    - Depends on location in the list

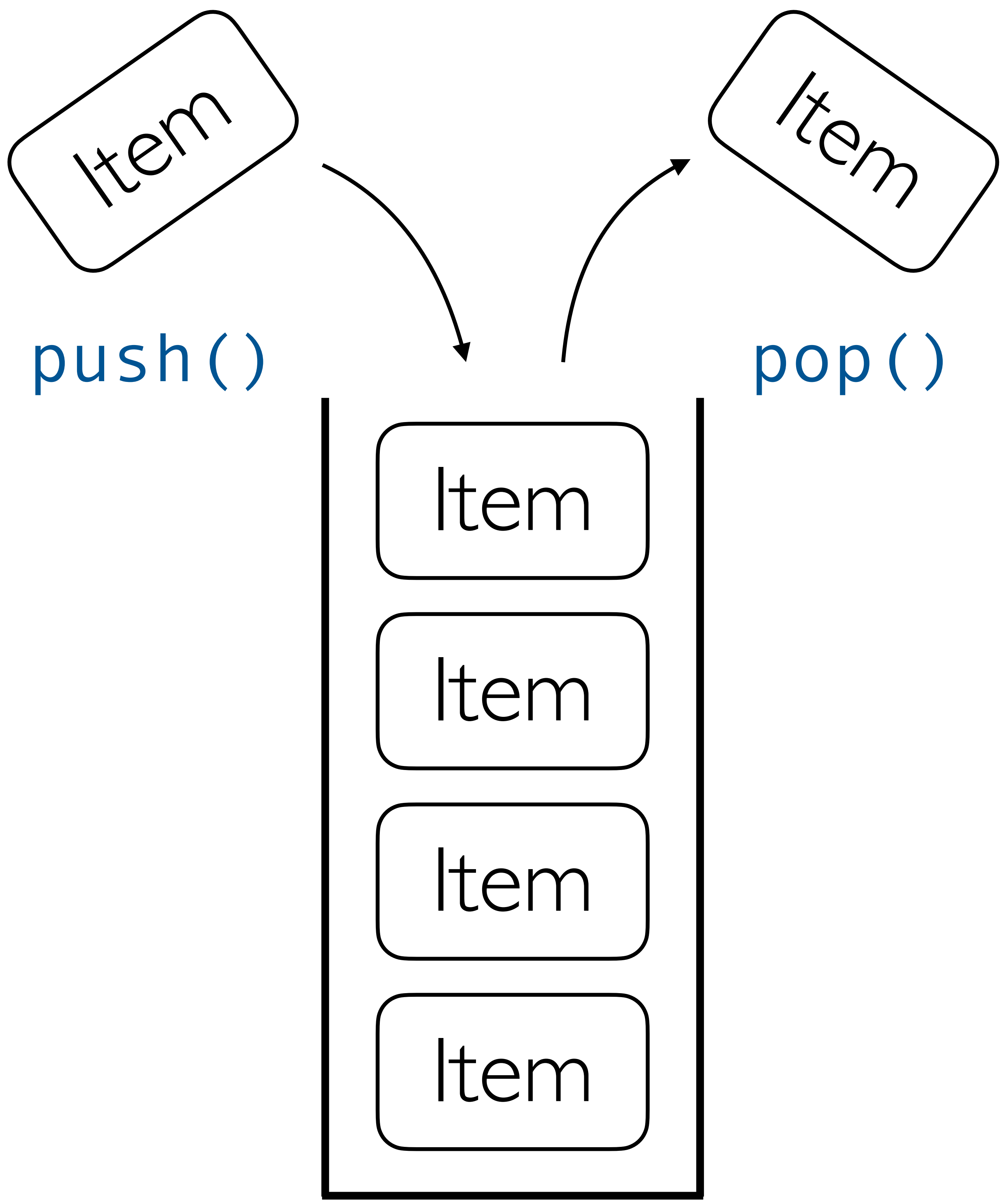
Linked list vs. array

Array	Linked List
Contiguous in memory	Non-contiguous in memory
Homogenous data types	Heterogenous data types
Fast random access	Slow random access
Slow append/insert/delete	Fast append/insert/delete

Stacks

- *Abstract* **ordered sequence** data type
- Must add/remove items in order
- Last-in, first-out (LIFO)

Stack characteristics

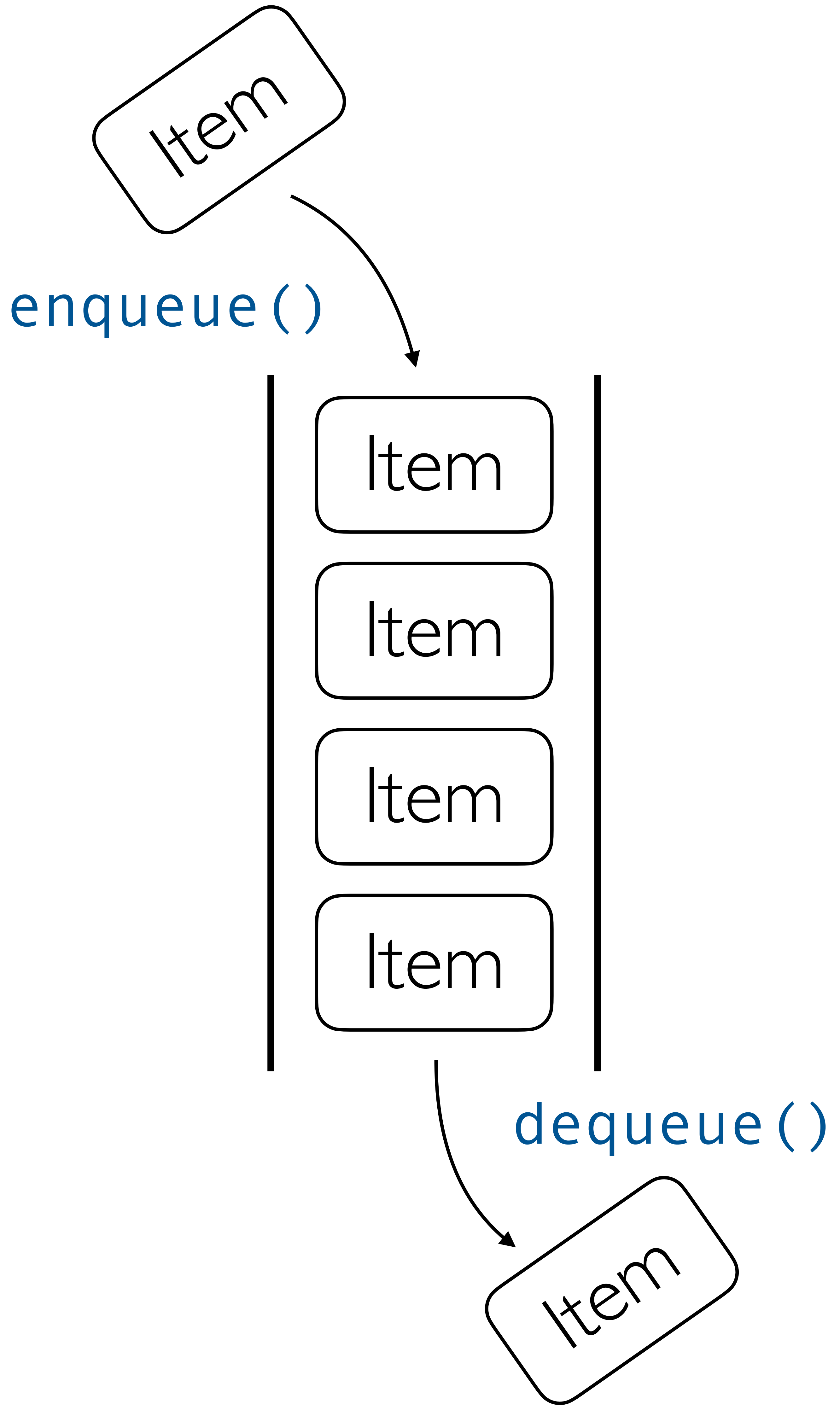


- Last-in, first-out (LIFO)
- Two primary operations:
  - **Push**: add item to top of stack
  - **Pop**: remove *and* return the top-most item of the stack
- Cannot access middle elements

Queues

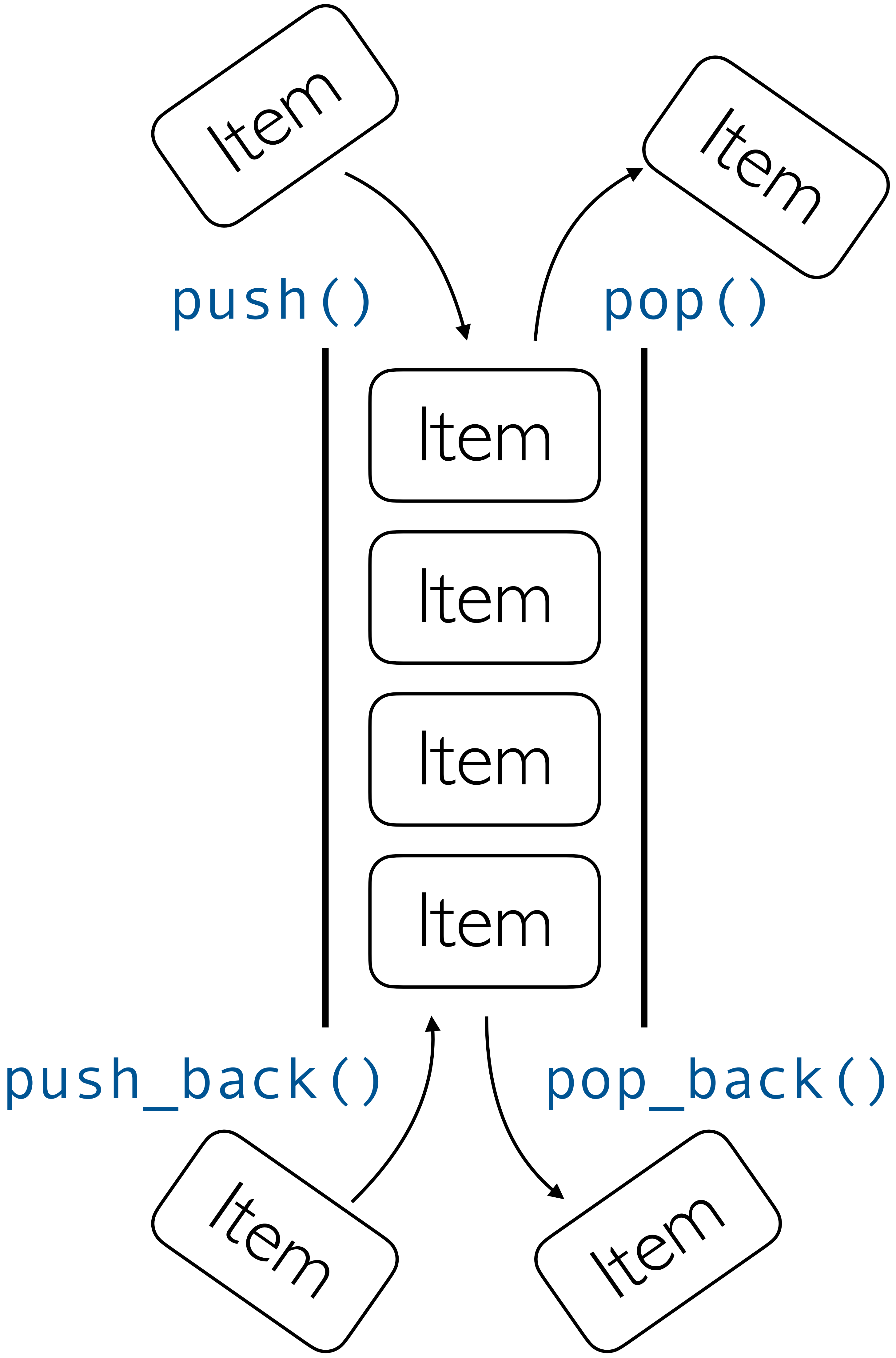
- *Abstract* **ordered sequence** data type
- Must add/remove items in order
- First-in, first-out (FIFO)

Queue characteristics



- First-in, first-out (FIFO)
- Two primary operations:
  - **Enqueue**: add item to end of queue
  - **Dequeue**: remove *and* return item from the front of queue
- Cannot access middle elements

Deque characteristics



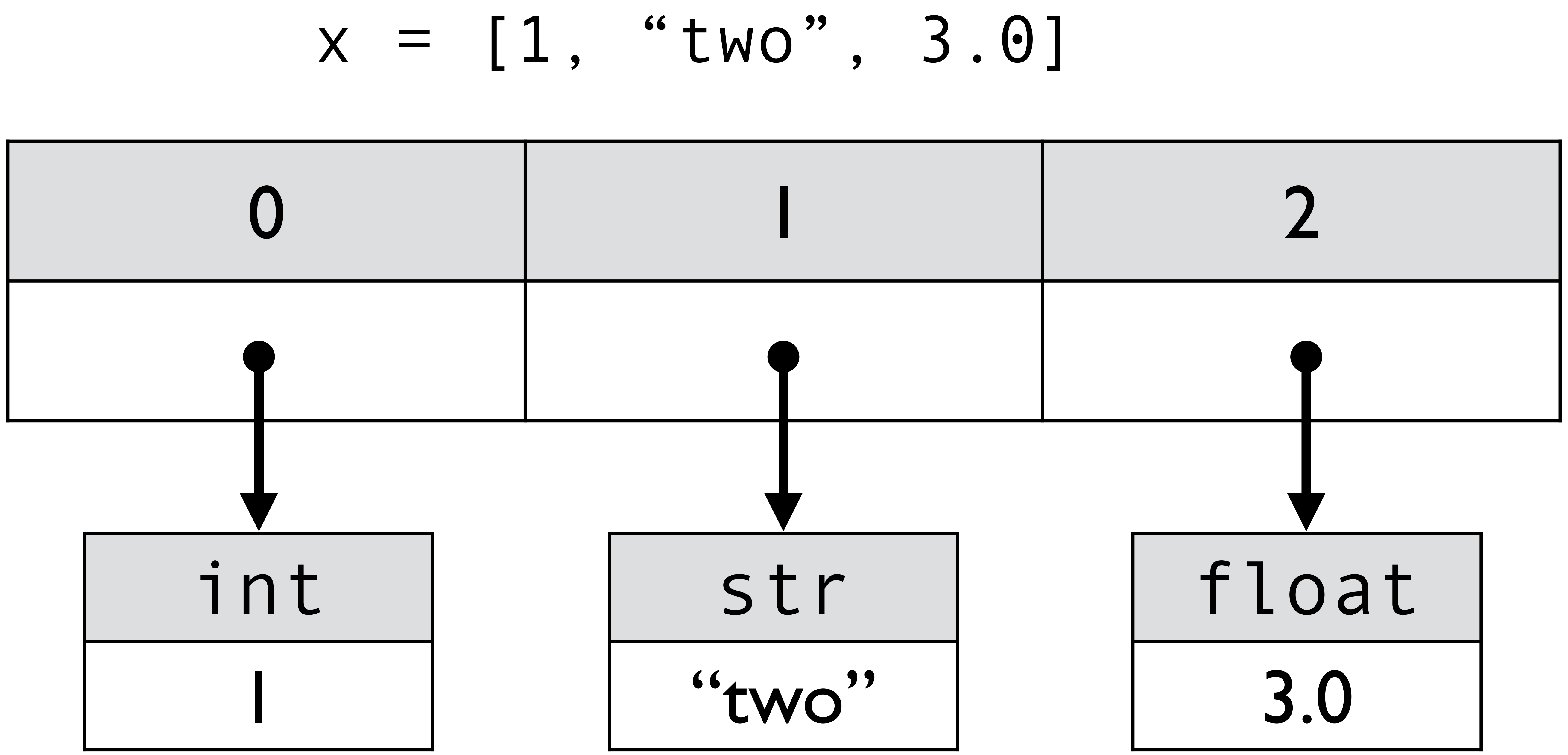
- *Double-ended* queue
- Four primary operations:
  - **Push & push_back**: <u>add</u> item to front/end of the deque
  - **Pop & pop_back**: <u>remove</u> *and* <u>return</u> front/end of deque
- Cannot access middle elements

Stacks and queues

- Many practical applications in computer science
- Stacks
  - Function calls go on the *call stack*
  - Parsing of language expressions
  - Memory management (allocating + freeing)
- Queues
  - CPU and I/O scheduling
  - Data traffic over a network
  - Algorithms such as breadth-first search (BFS)

Lists in Python

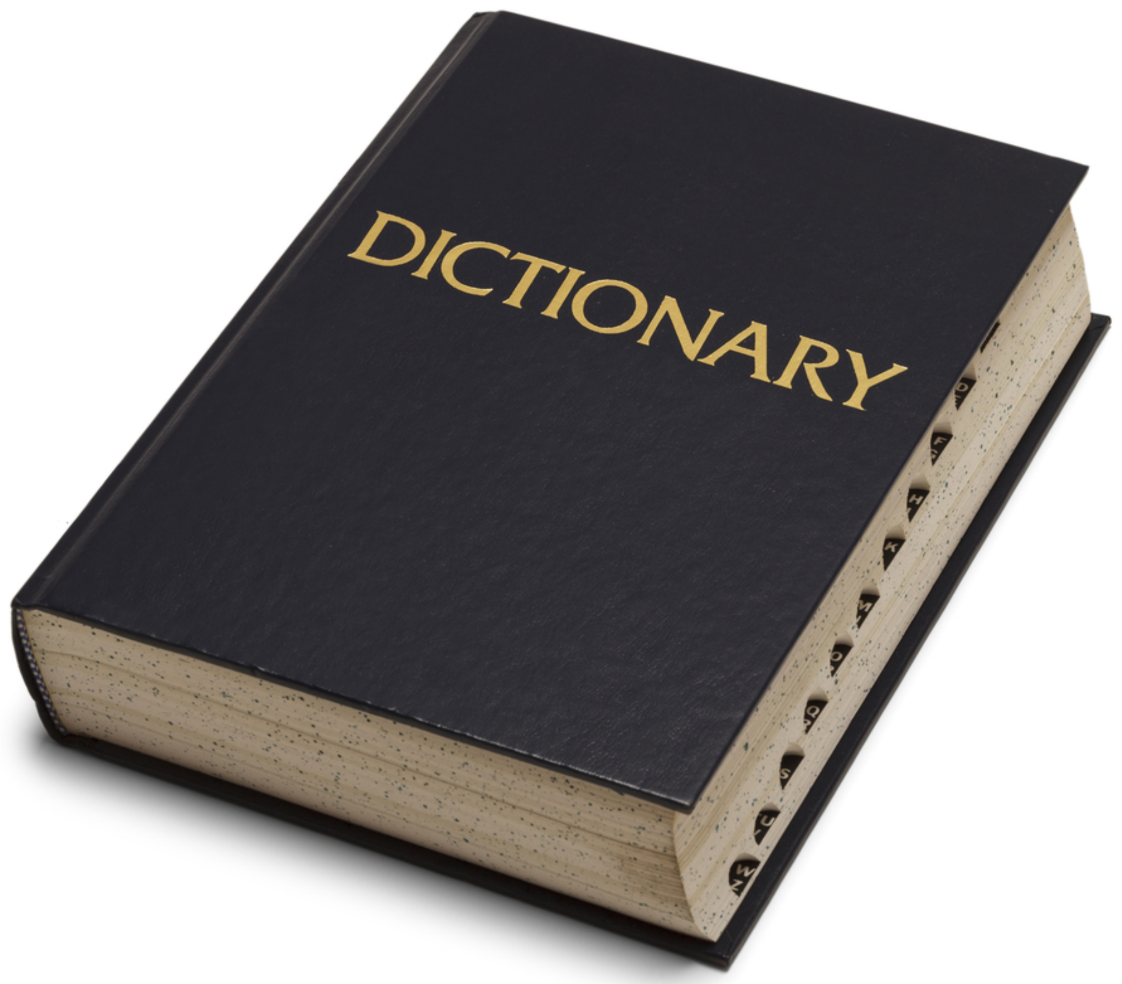
- Built-in lists in Python are *arrays of pointers*
- Good compromise of performance vs. flexibility



HASH TABLES

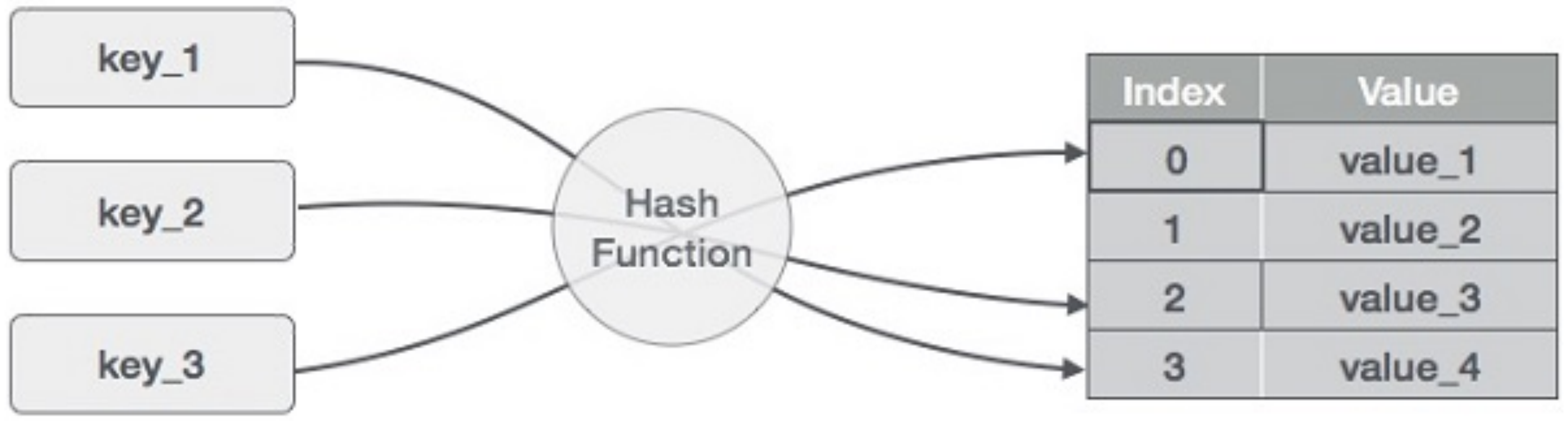
Associative arrays

- *Abstract* **unordered collection** data type
- Store collection of **key-value** pairs
- Access item by *key* rather than *index*
- Also called *dictionary*, *map*, etc.

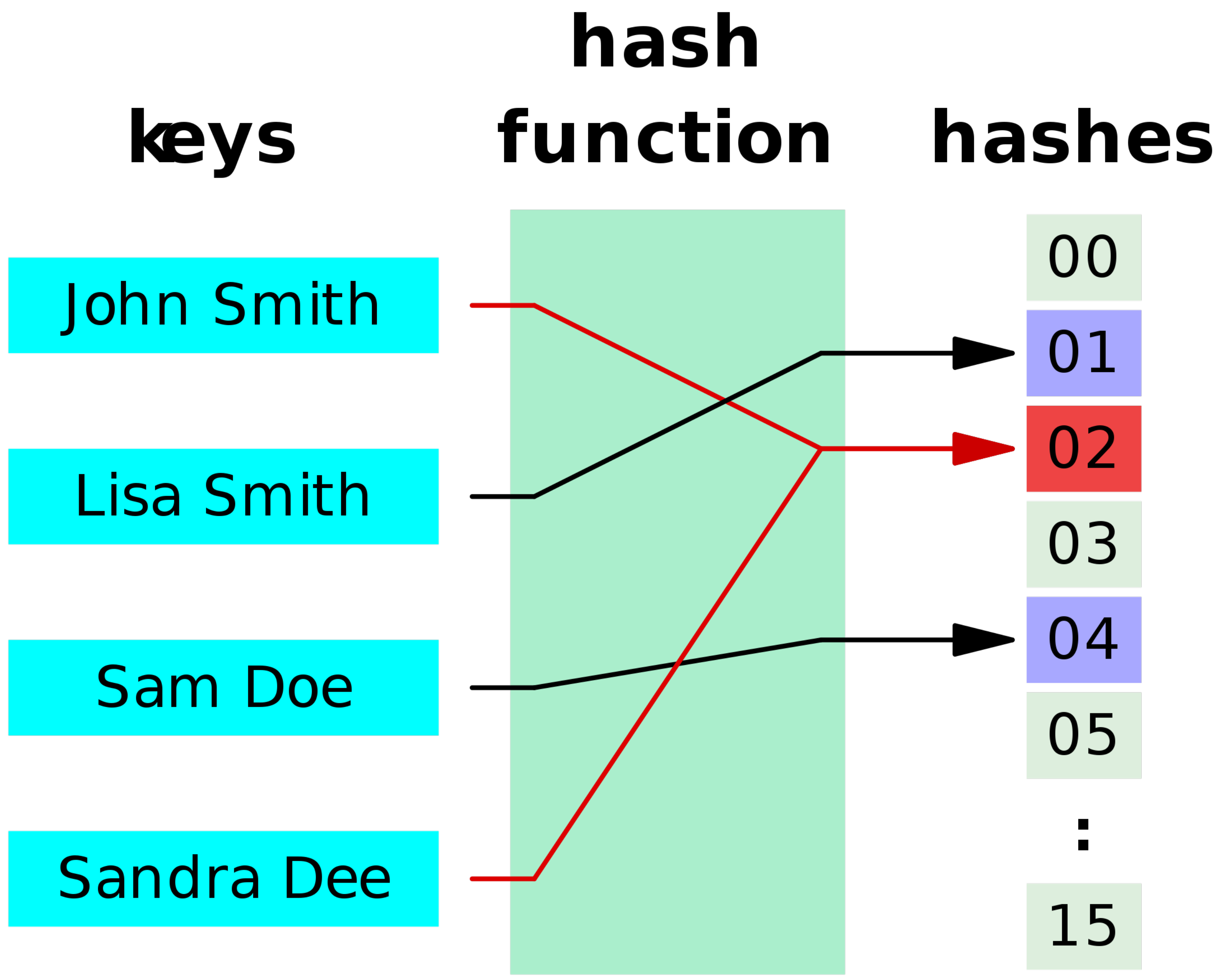


Hash tables / hash maps

- **Unordered associative** (key-value) data type
- Items stored in *buckets* by hashing key
- Buckets store items with same *hash code*
- May be heterogenous



Hash function



- Map **keys** to fixed range of **indices**
  - E.g., `hash(key) = key mod table_size`
- Equal keys have equal *hash codes*
- Equal hashes do *not* imply equal keys
  - Different keys with same hash is a **collision**
- Reduce space of keys to indexable size

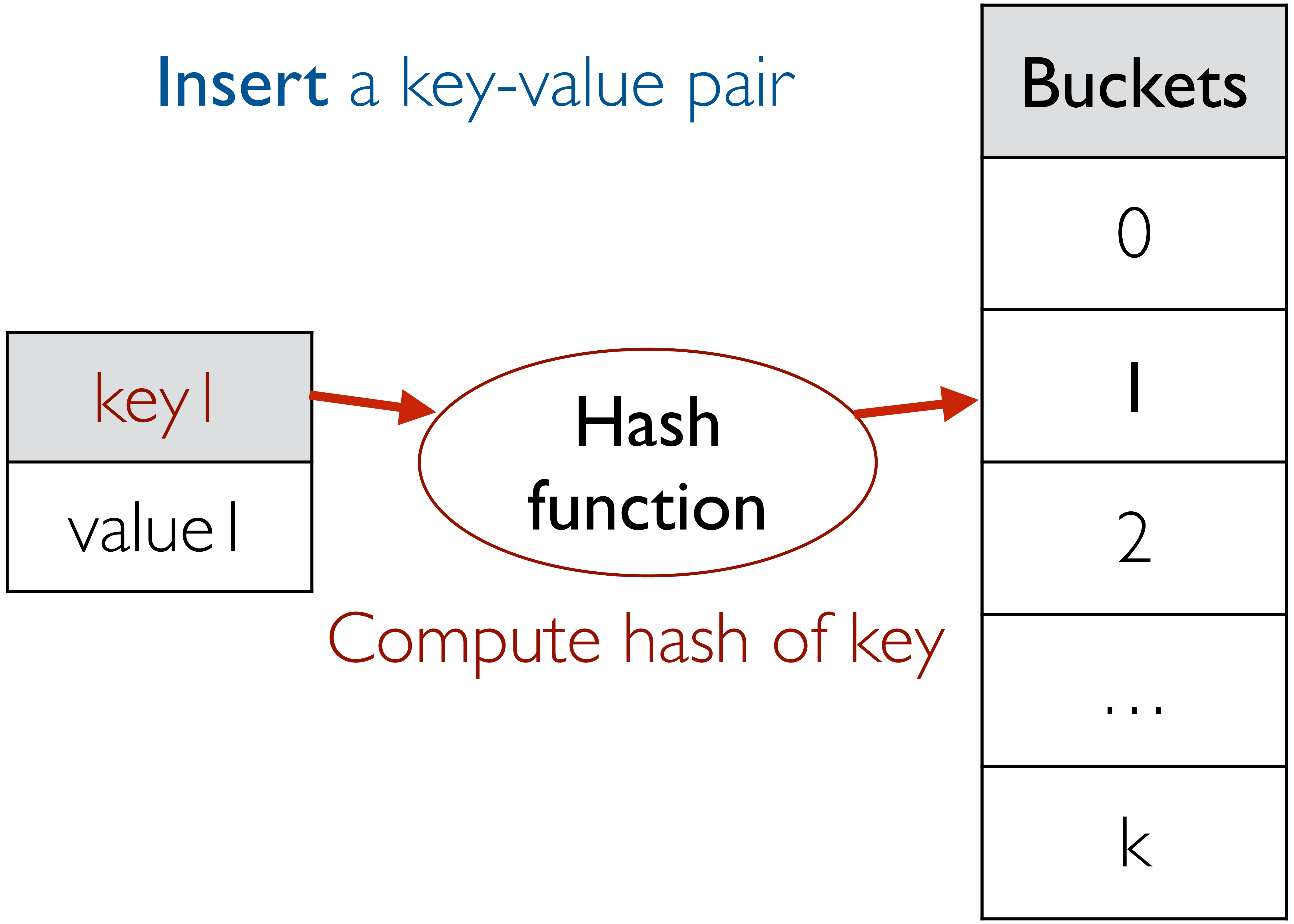
Hash table

Hash function

Buckets
0
1
2
…
k

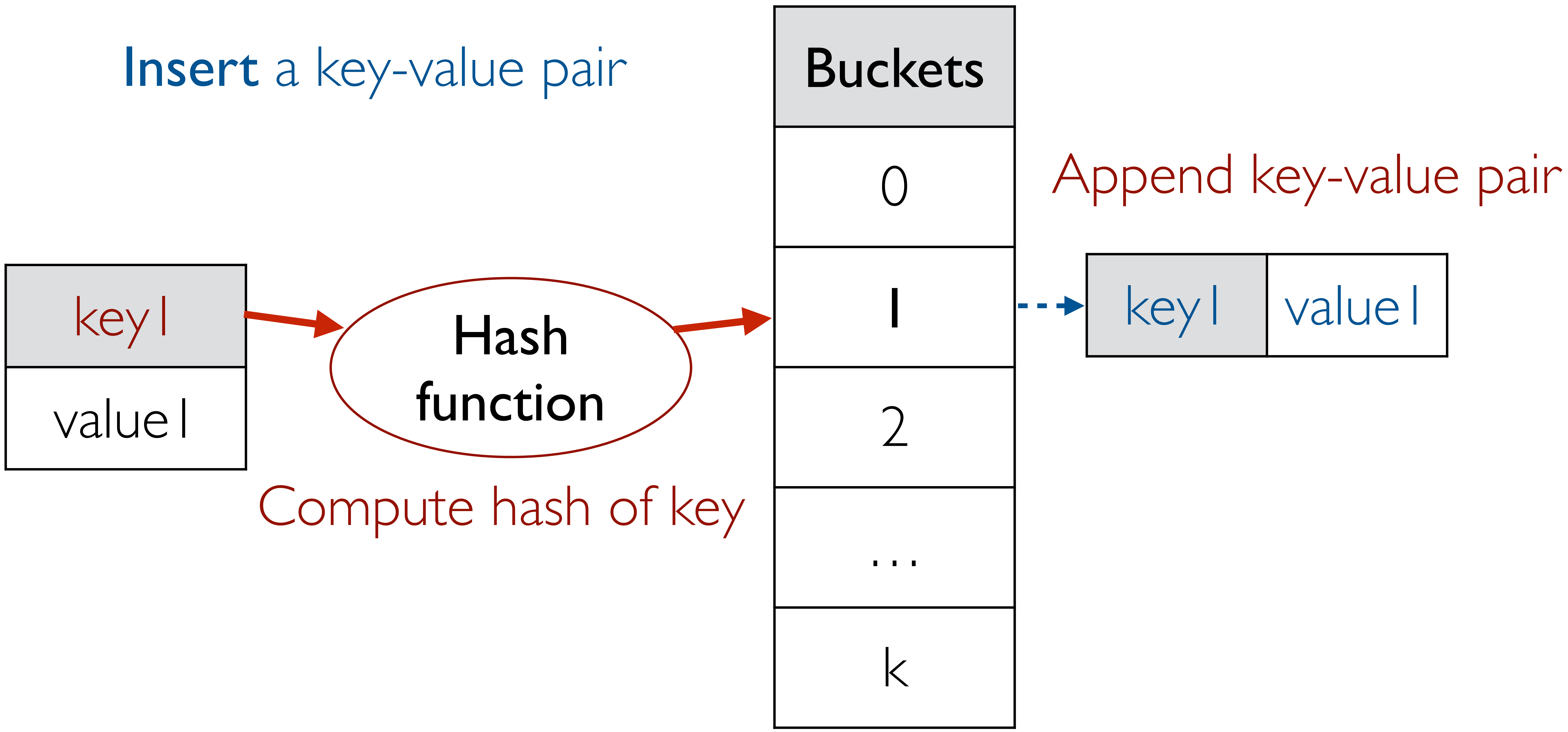
- Store array of *k* buckets
- *Hash* maps keys to buckets

Inserting into a hash table



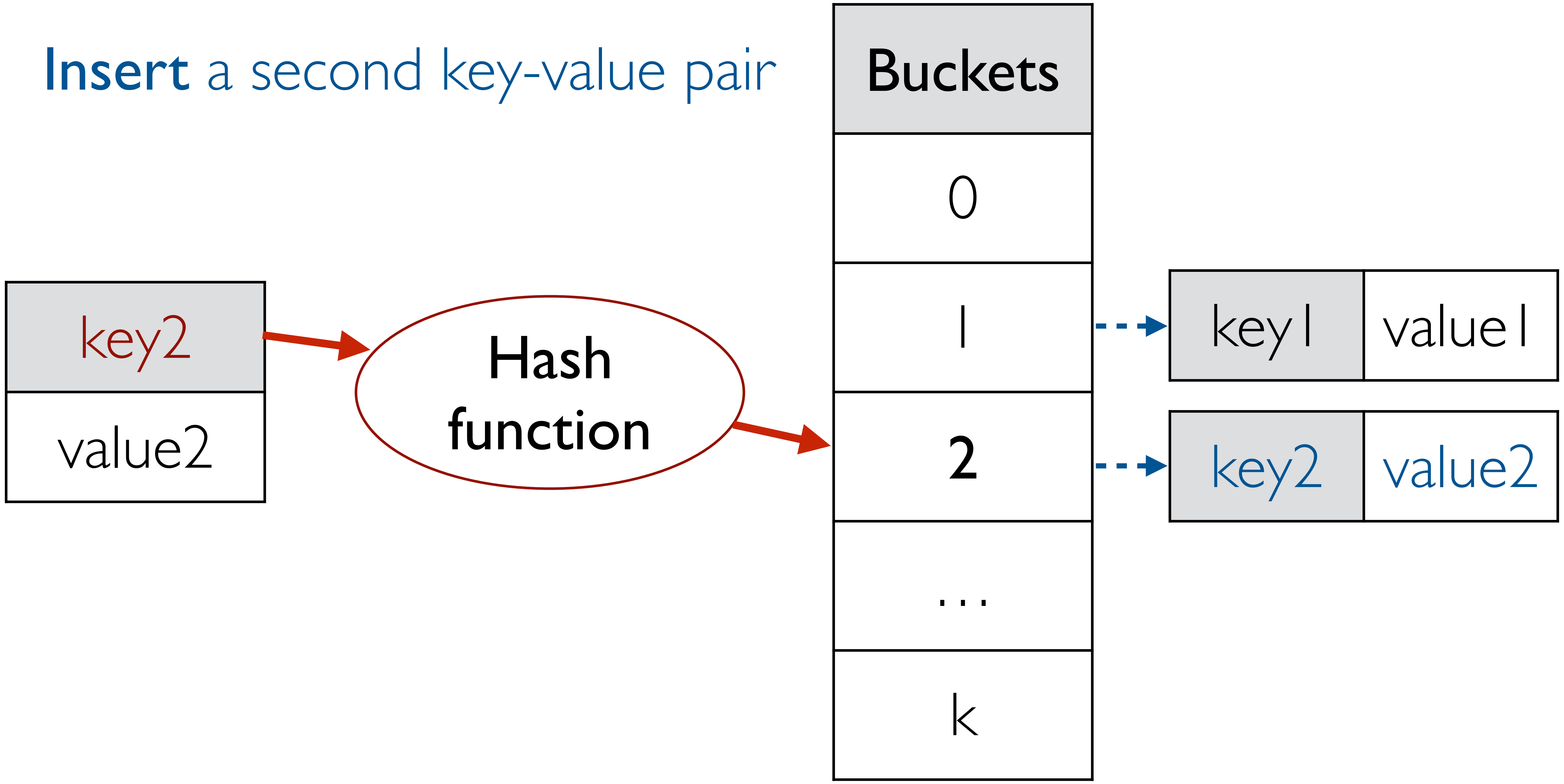
- Compute hash of key
- Find appropriate bucket
- Buckets contain *lists*

Inserting into a hash table



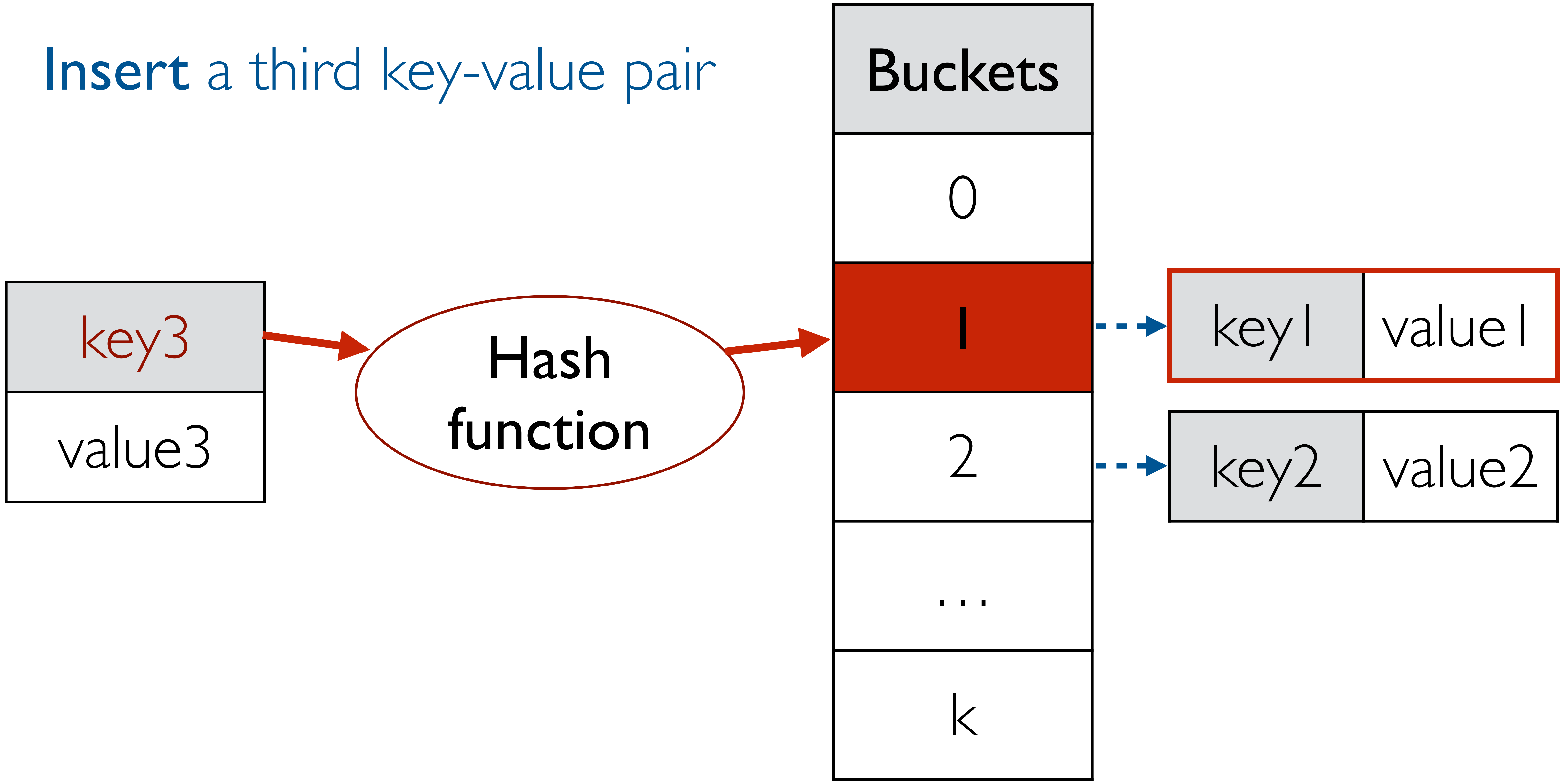
Inserting into a hash table

Insert a second key-value pair



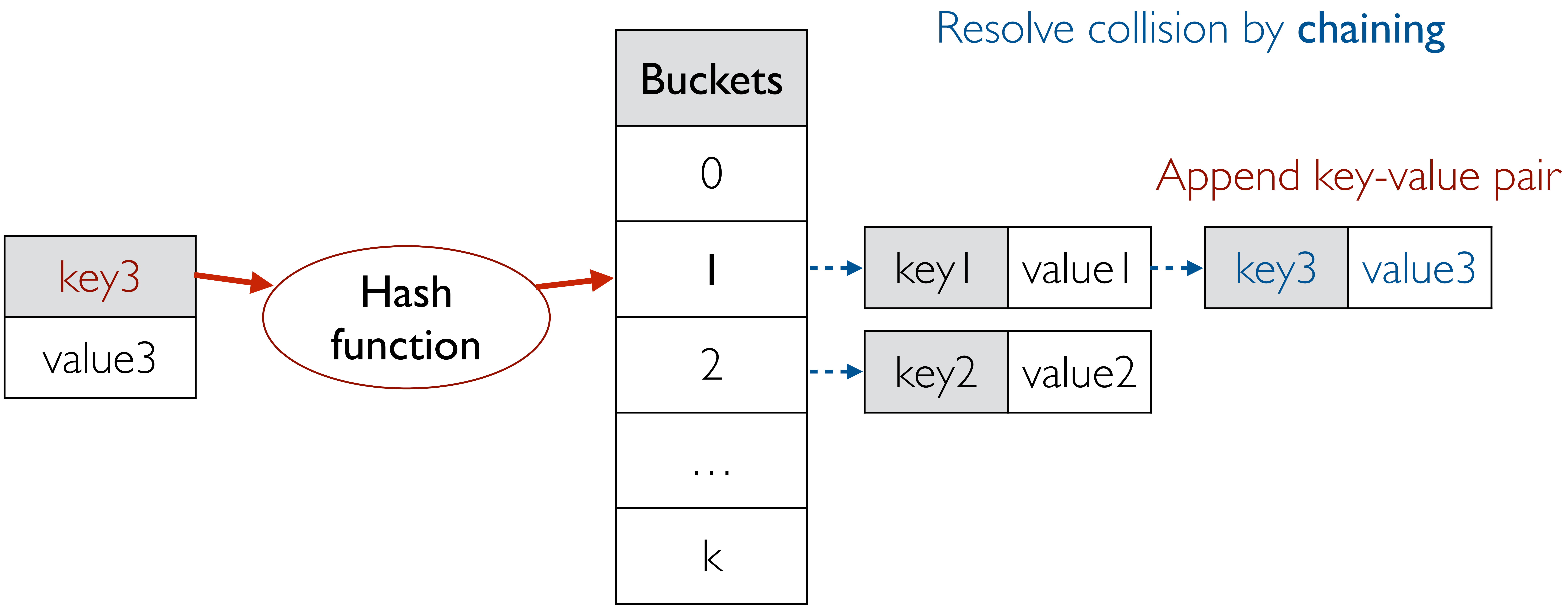
Handling collisions

Insert a third key-value pair

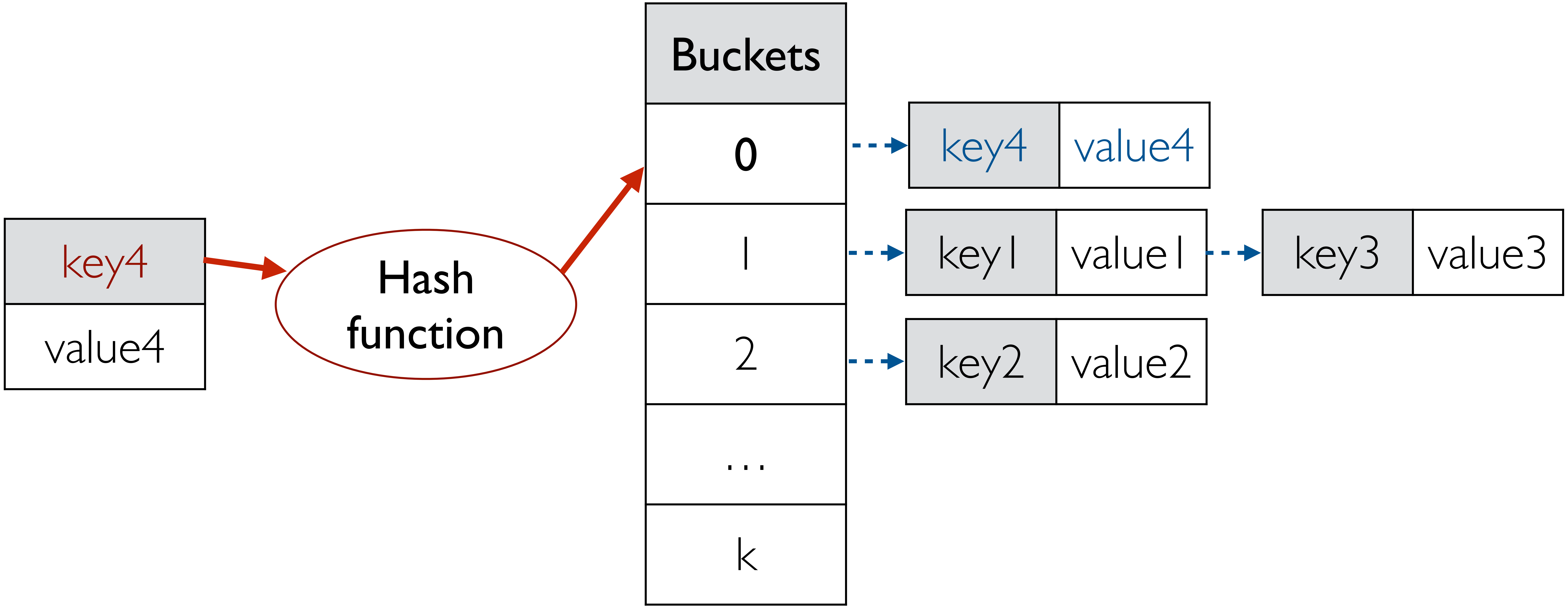


- Different keys may have same hash code
- **Chain** items with the same hash code

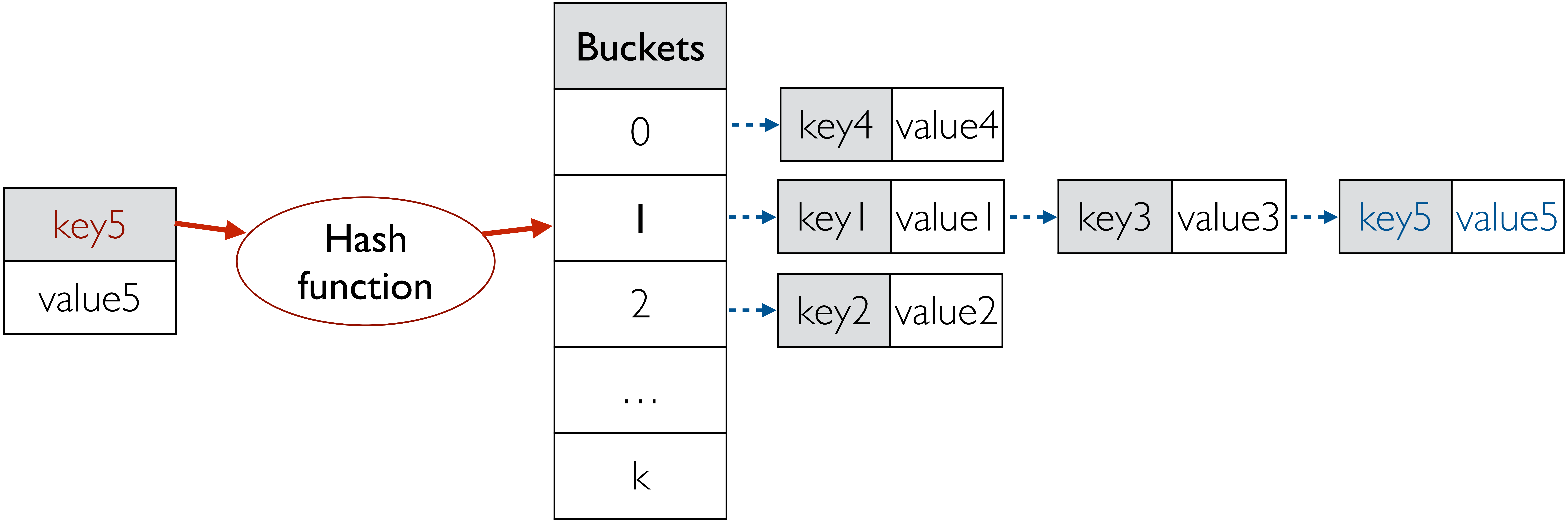
Handling collisions



More insertion

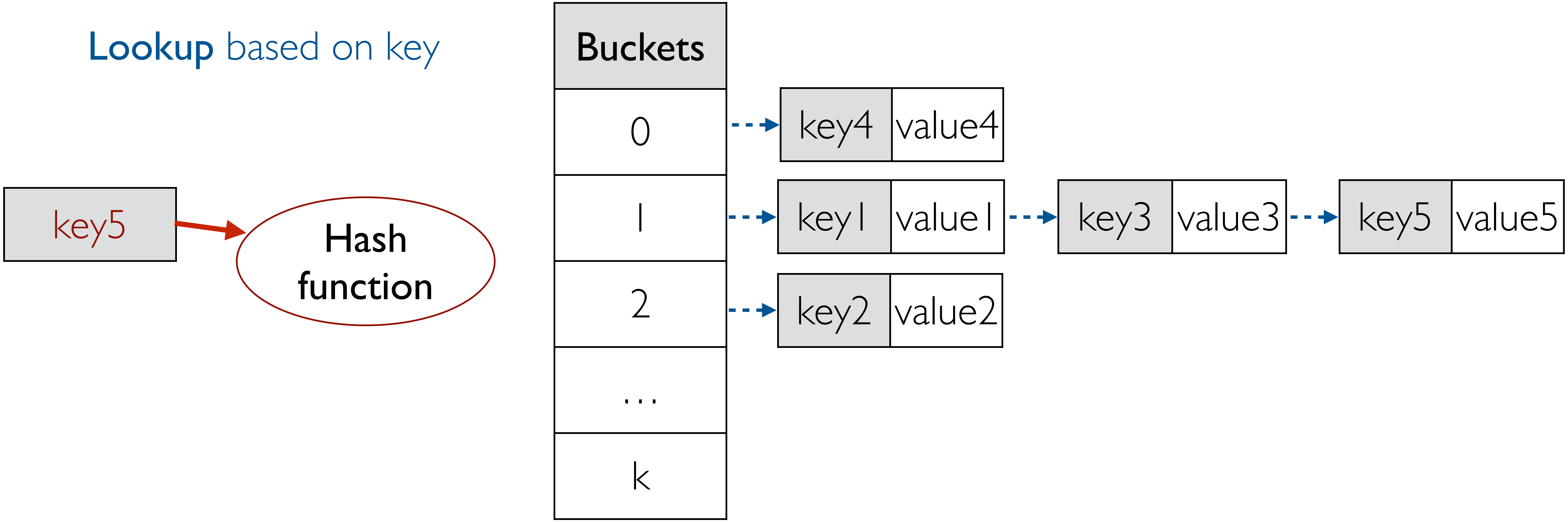


More chaining

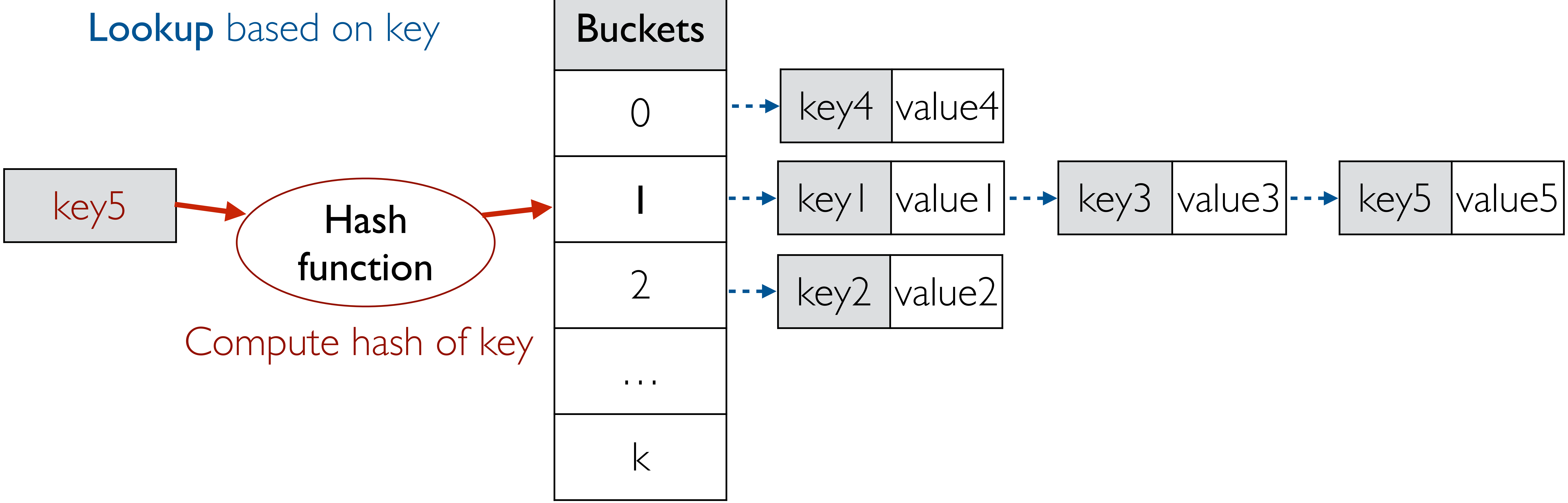


Looking up an item in a hash table

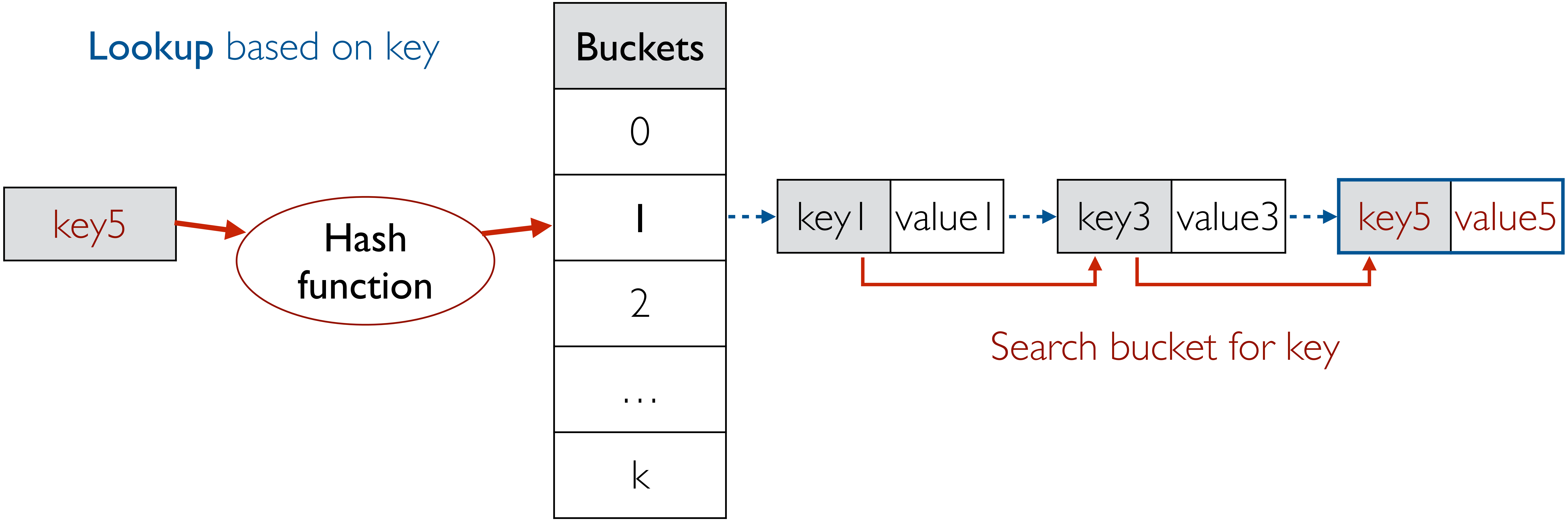
Lookup based on key



Look up an item in a hash table



Look up an item in a hash table



Hash tables in Python

```
class HTable:

    def __init__(self, buckets=1000):
        self.numbuckets = buckets
        self.size = 0
        self.keys = [[] for i in range(buckets)]
        self.values = [[] for i in range(buckets)]

    def set(self, key, value):
        bucket = self.hashkey(key)
        if key in self.keys[bucket]:
            i = self.keys[bucket].index(key)
            self.values[bucket][i] = value
        else:
            self.keys[bucket].append(key)
            self.values[bucket].append(value)
            self.size += 1
```

Hash tables in Python

```
class HTable:

    def __init__(self, buckets=1000):
        self.numbuckets = buckets
        self.size = 0
        self.keys = [[] for i in range(buckets)]
        self.values = [[] for i in range(buckets)]

    def set(self, key, value):
        bucket = self.hashkey(key)
        if key in self.keys[bucket]:
            i = self.keys[bucket].index(key)
            self.values[bucket][i] = value
        else:
            self.keys[bucket].append(key)
            self.values[bucket].append(value)
            self.size += 1
```

Initialize *k* buckets

Update existing key-value pair

Append new key-value pair

Performance of hash tables

- Very fast random read/write of existing items
  - Depends on *load factor* — can devolve to linked list if too high
- Somewhat slow traversal of items
- Very fast searching for specific items (by key)
- Slow searching for specific items (by value)
- Very fast insertion/deletion of new items

Load factor

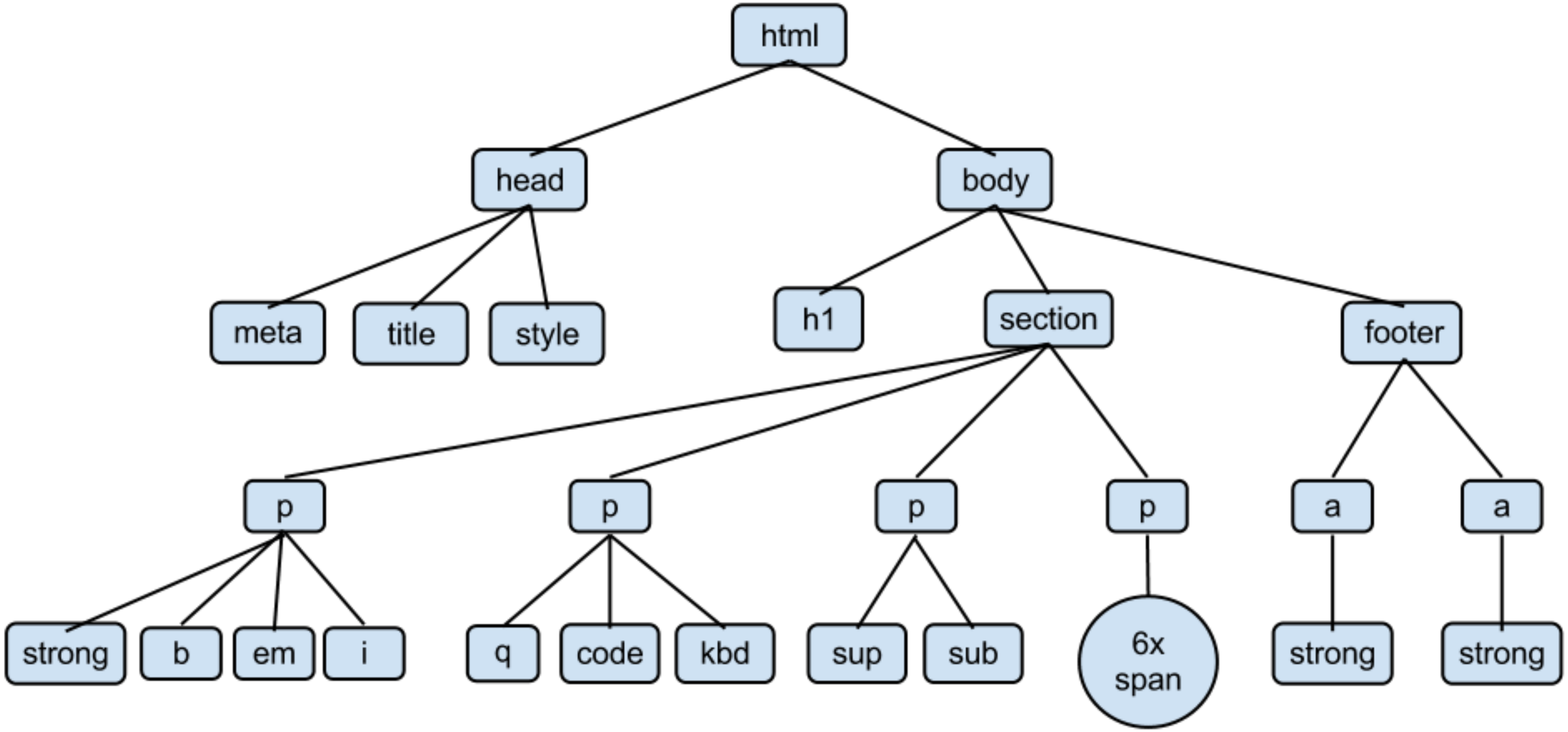
- Ideal hashing maps keys *evenly* across buckets
- Good *performance* relies on *small buckets*
- Measured by ***load factor = n / k***
  - ***n*** is the number of items in the table
  - ***k*** is the number of buckets
  - Average number of items in a bucket chain
- Very large load factors lead to *poor performance*
- Very small load factors lead to *poor memory use*

TREES AND SEARCHES

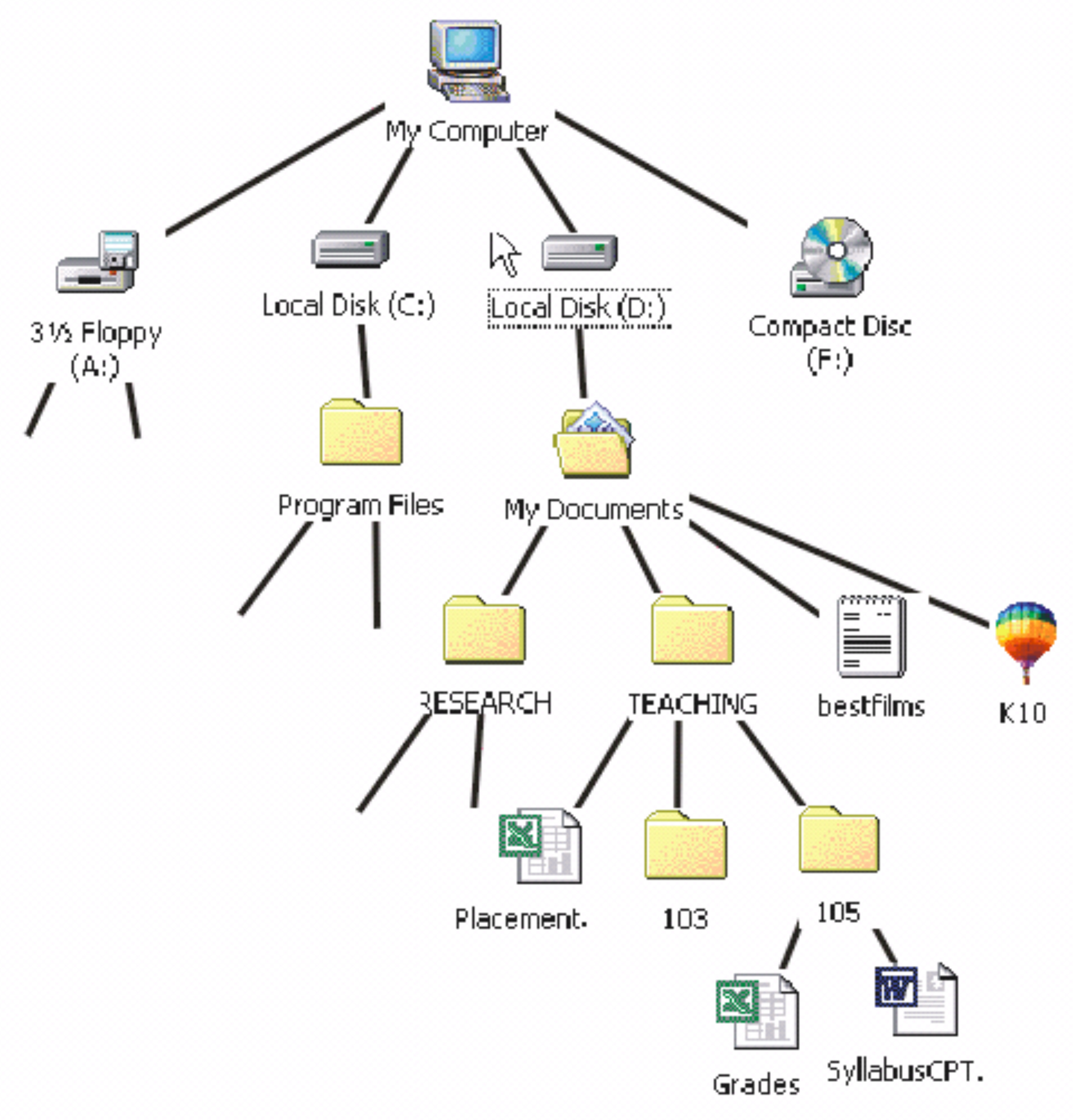
Trees

- *Abstract* **ordered collection** data type
- Hierarchical collection of **nodes** and **edges**
- Starts at a *root* node
- Nodes may have *children* nodes

Trees on the web

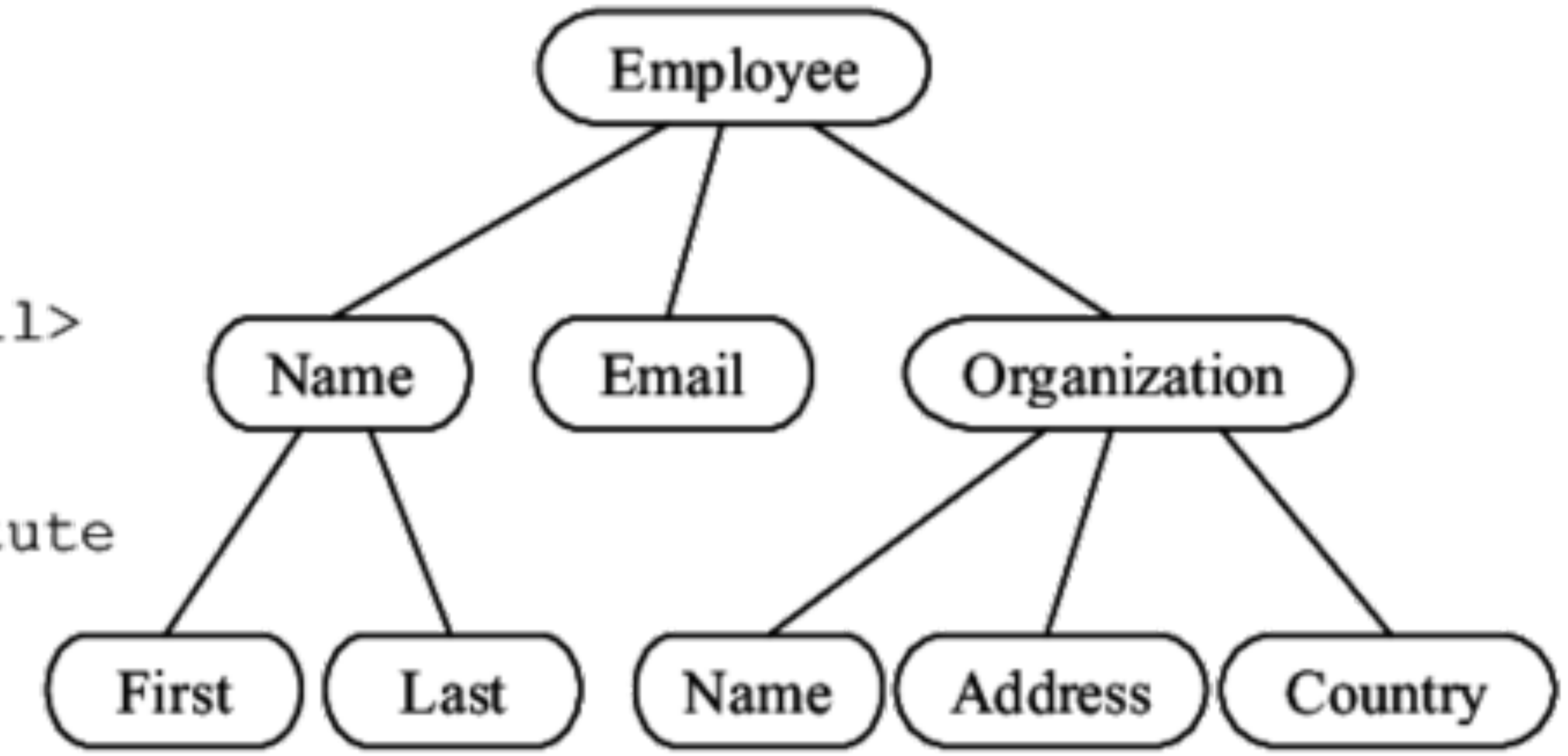


Trees locally



Trees as data

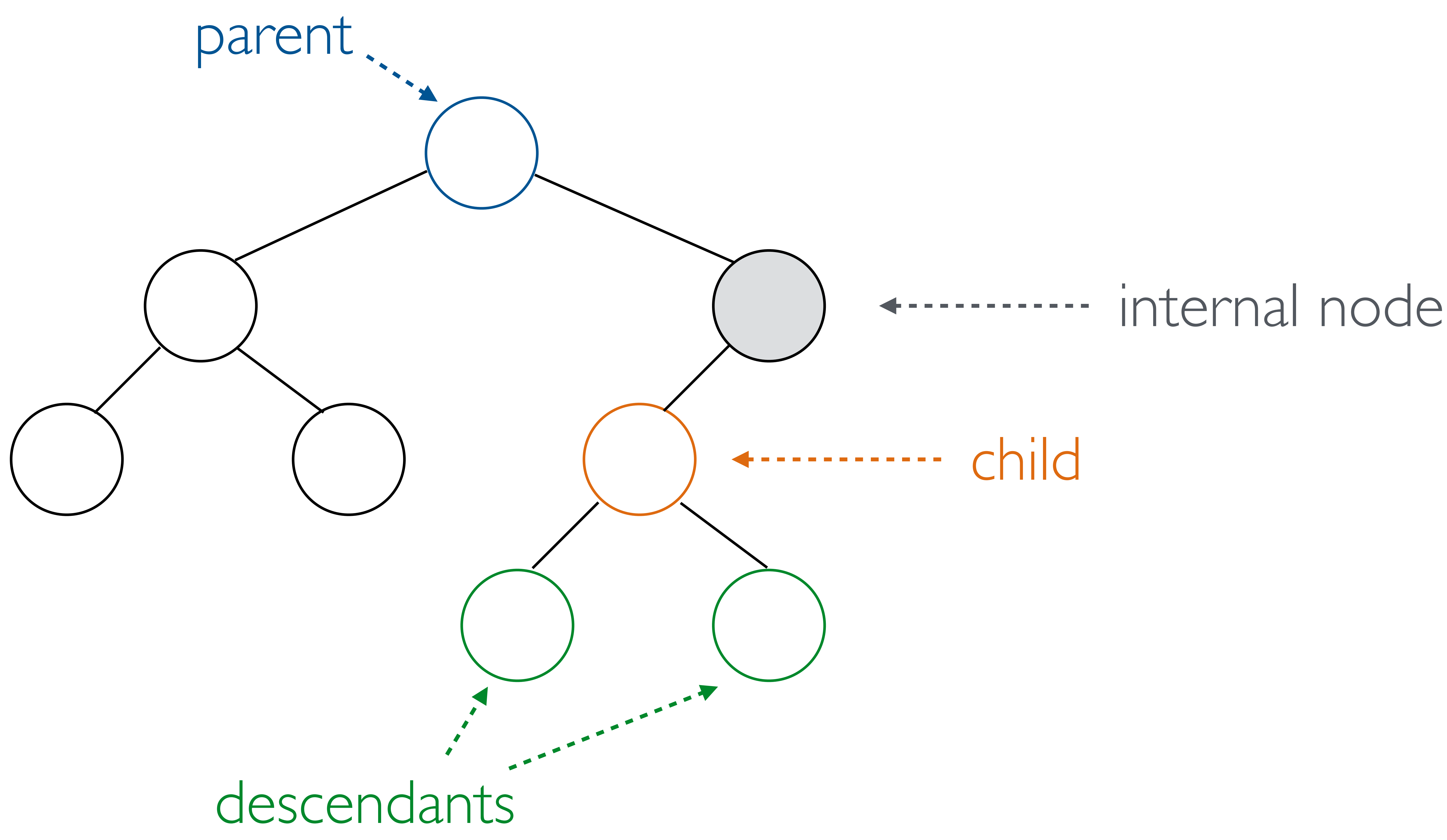
```
<Employee>
    <Name>
        <First>Lassi</First>
        <Last>Lehto</Last>
    </Name>
    <Email>Lassi.Lehto@fgi.fi</Email>
    <Organization>
        <Name>
            Finnish Geodetic Institute
        </Name>
        <Address>
            PO Box 15,
            FIN-02431 Masala
        </Address>
        <Country CountryCode="358">Finland</Country>
    </Organization>
</Employee>
```



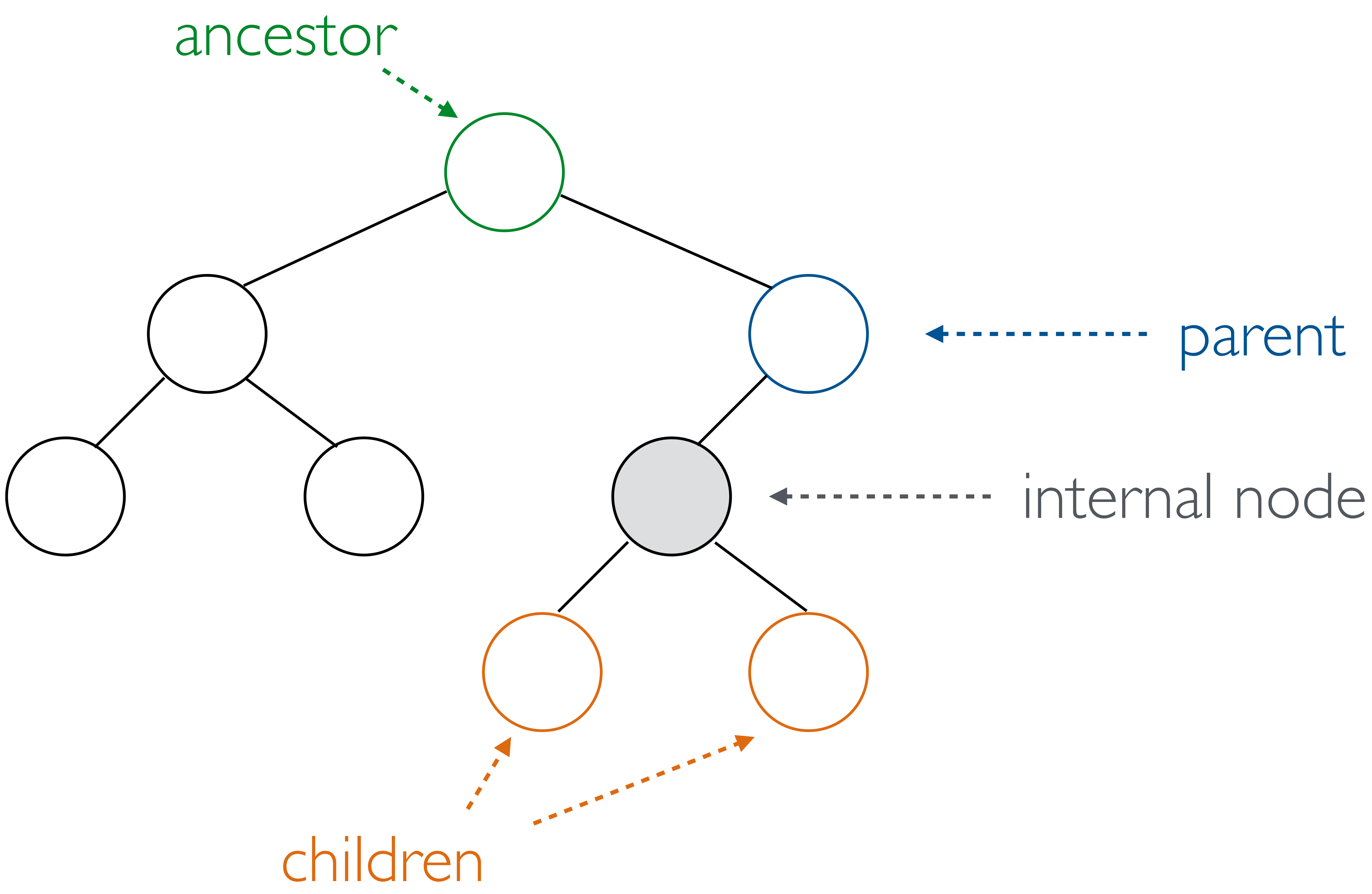
Tree vocabulary

Root	The top node of the tree, with no parent
Parent	Node connected immediately above
Child	Node connected immediately below
Sibling	Nodes that share the same parent
Ancestor	Node reachable by ascending tree from child to parent
Descendent	Node reachable by descending tree from parent to child
Neighbor	A parent or child node
Internal	A node with at least one child (possibly including root)

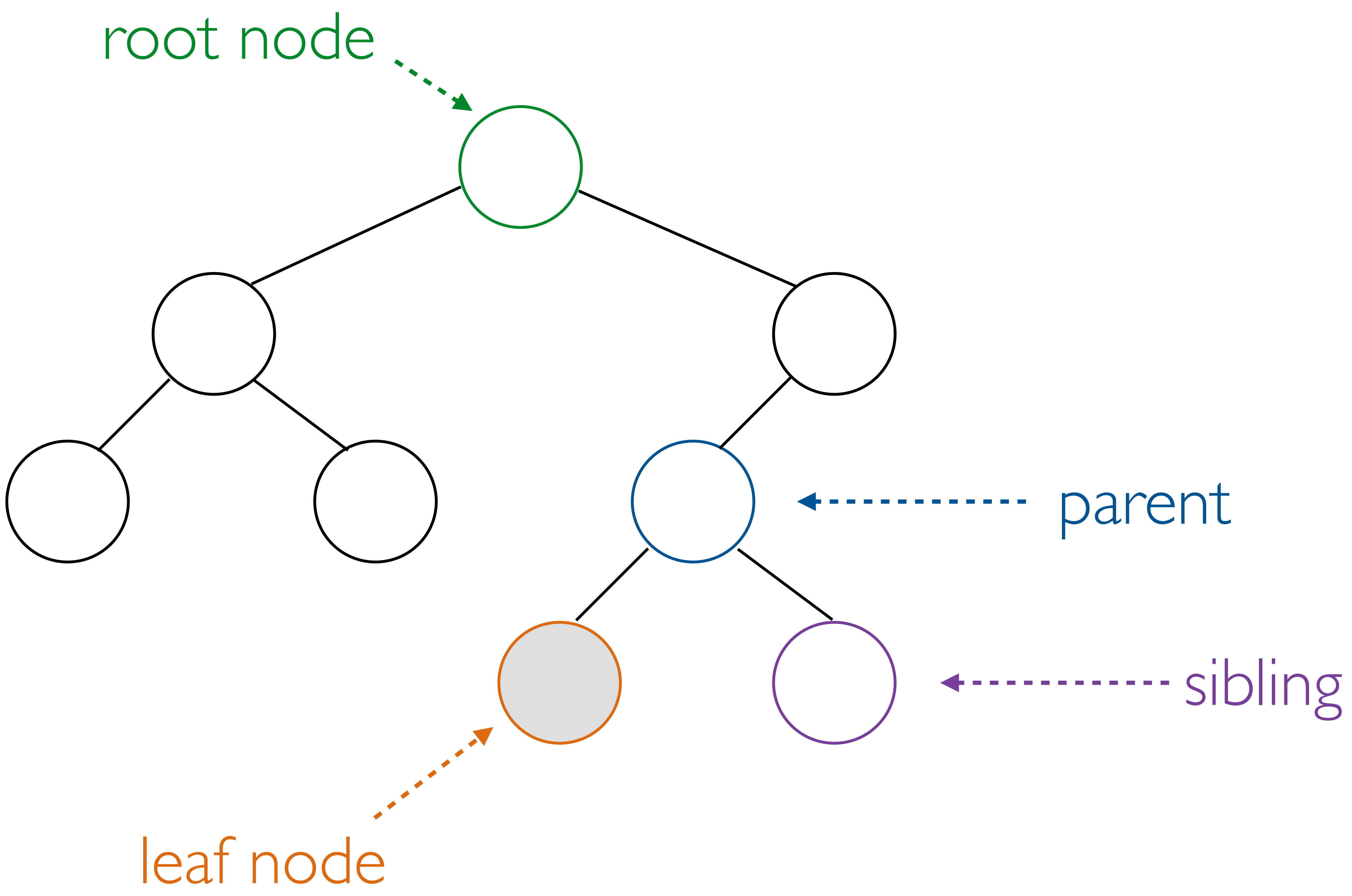
Tree vocabulary



Tree vocabulary



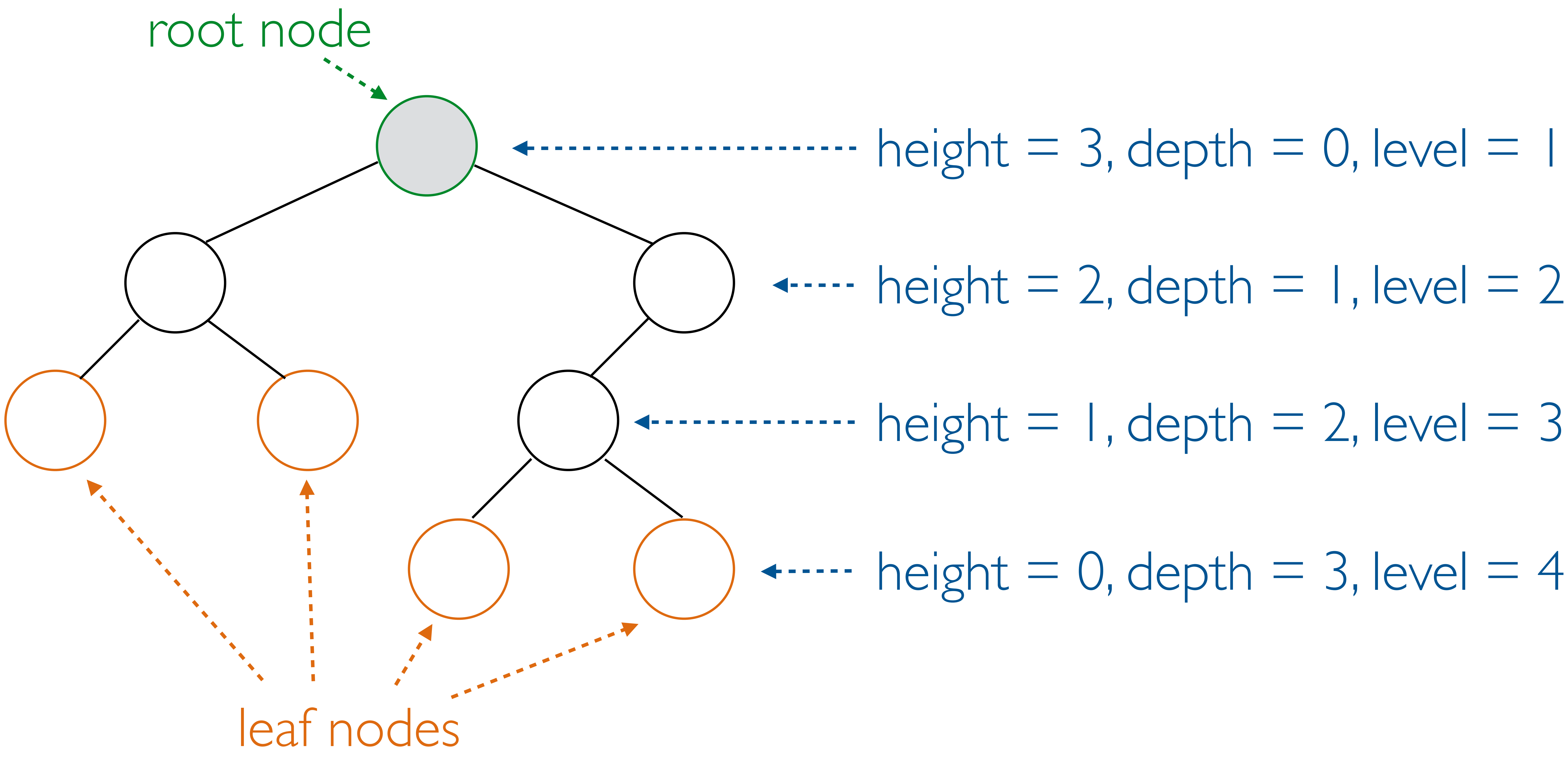
Tree vocabulary



Tree vocabulary

Degree	Number of children for a given node
Height	Number of edges from node down to descendant leaf
Depth	Number of edges from node up to root
Level	Depth + 1
Width	Number of nodes in a level
Breadth	Number of leaves in a tree
Path	Sequence of edges connecting two nodes

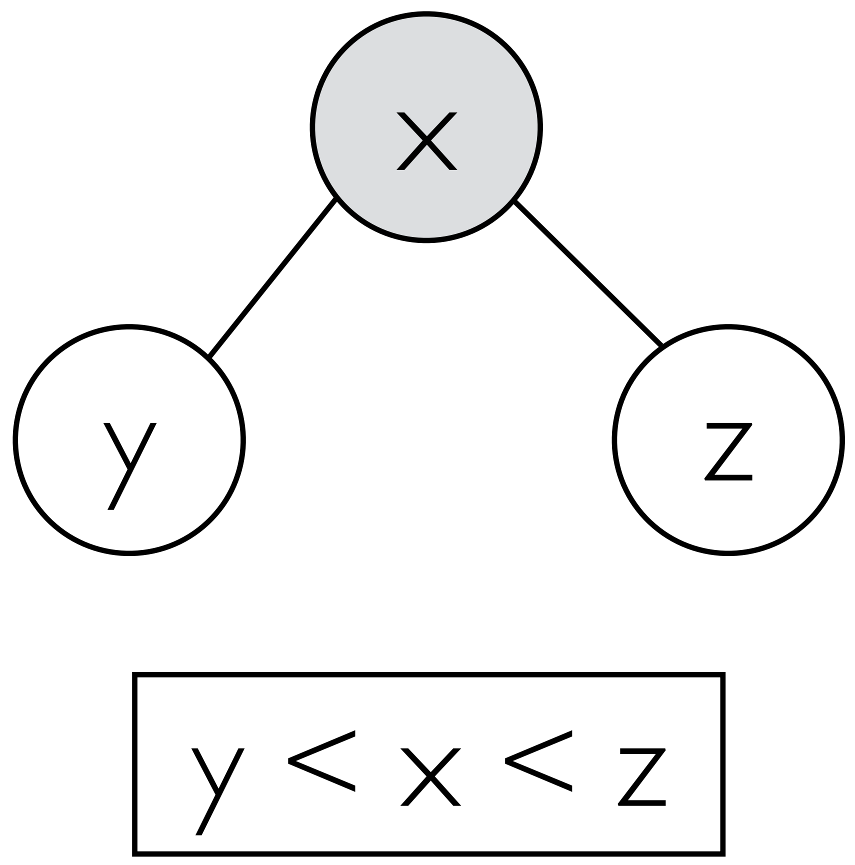
Tree vocabulary



Binary search trees (BST)

- **Ordered hierarchical** data type
- Items stored in *linked nodes*
  - Nodes form a tree structure
  - Each node can have up to **two children**
- *Organized* to facilitate fast searches
- May be **associative** (key-value)

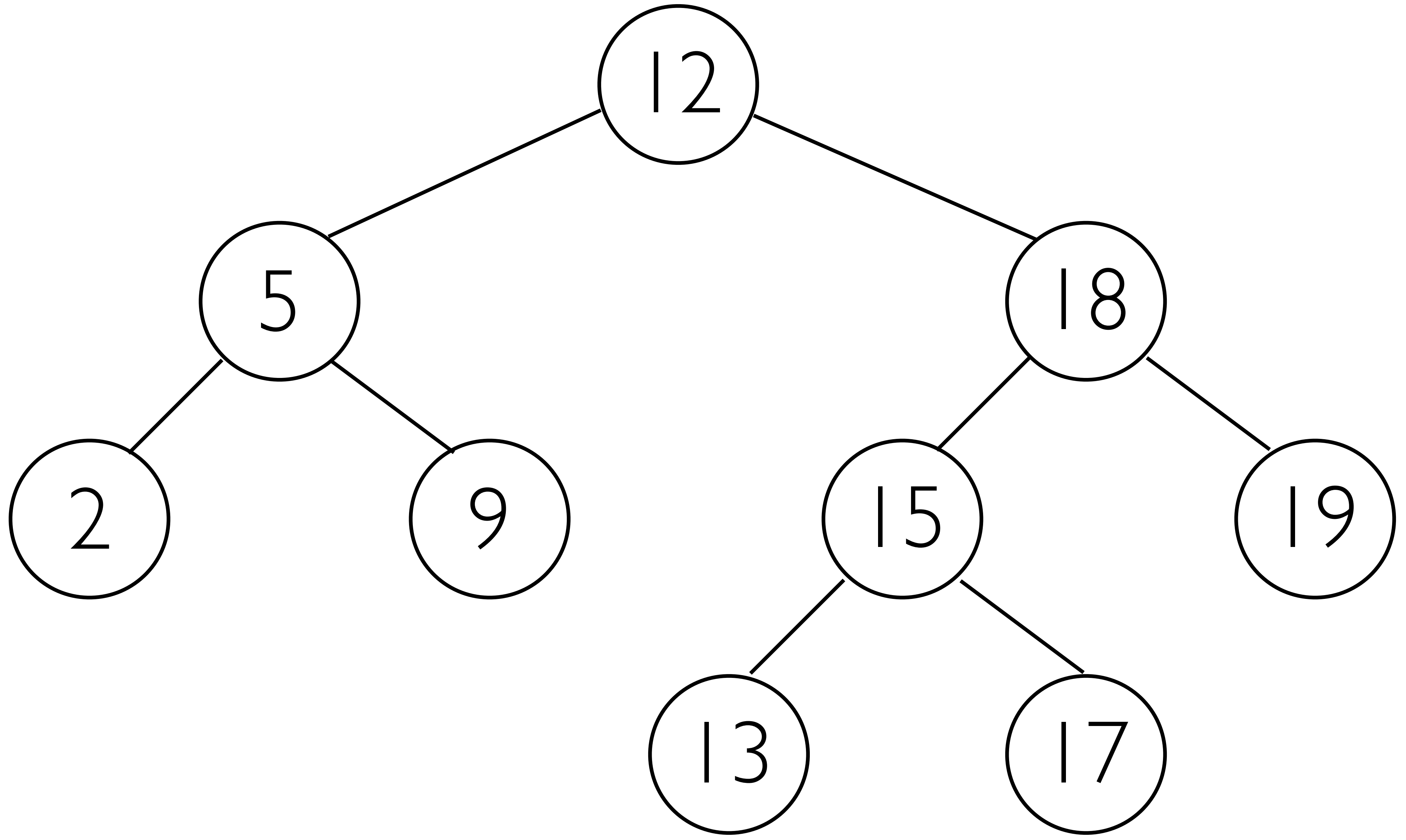
BST property



- Let **x** be a node in the BST
- If **y** is a node in the *left* subtree of **x**
  - Then **y.key < x.key**
- If **z** is a node in the *right* subtree of **x**
  - Then **z.key > x.key**
- Some definitions may allow *duplicates*

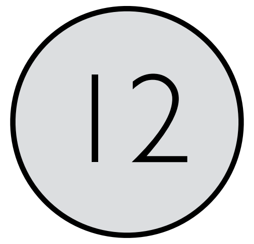
BST property

- Let **x** be a node in the BST
- For **y** in the *left* subtree of **x**
  - Then **y.key < x.key**
- For **z** in the *right* subtree of **x**
  - Then **z.key > x.key**



Building a BST

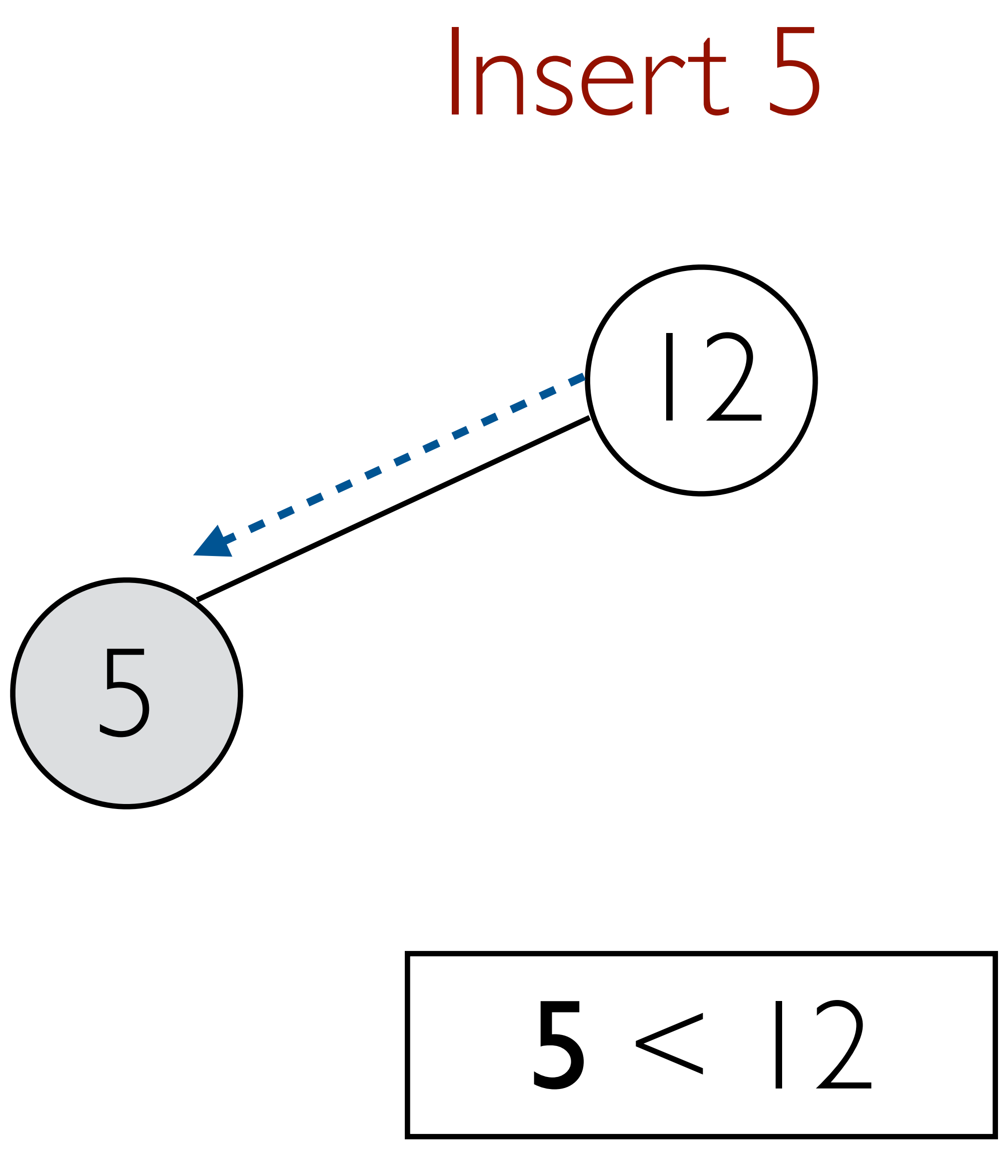
Insert 12



- Start with an empty tree (zero nodes)
- Insert a node
- The first (top) node is the root node

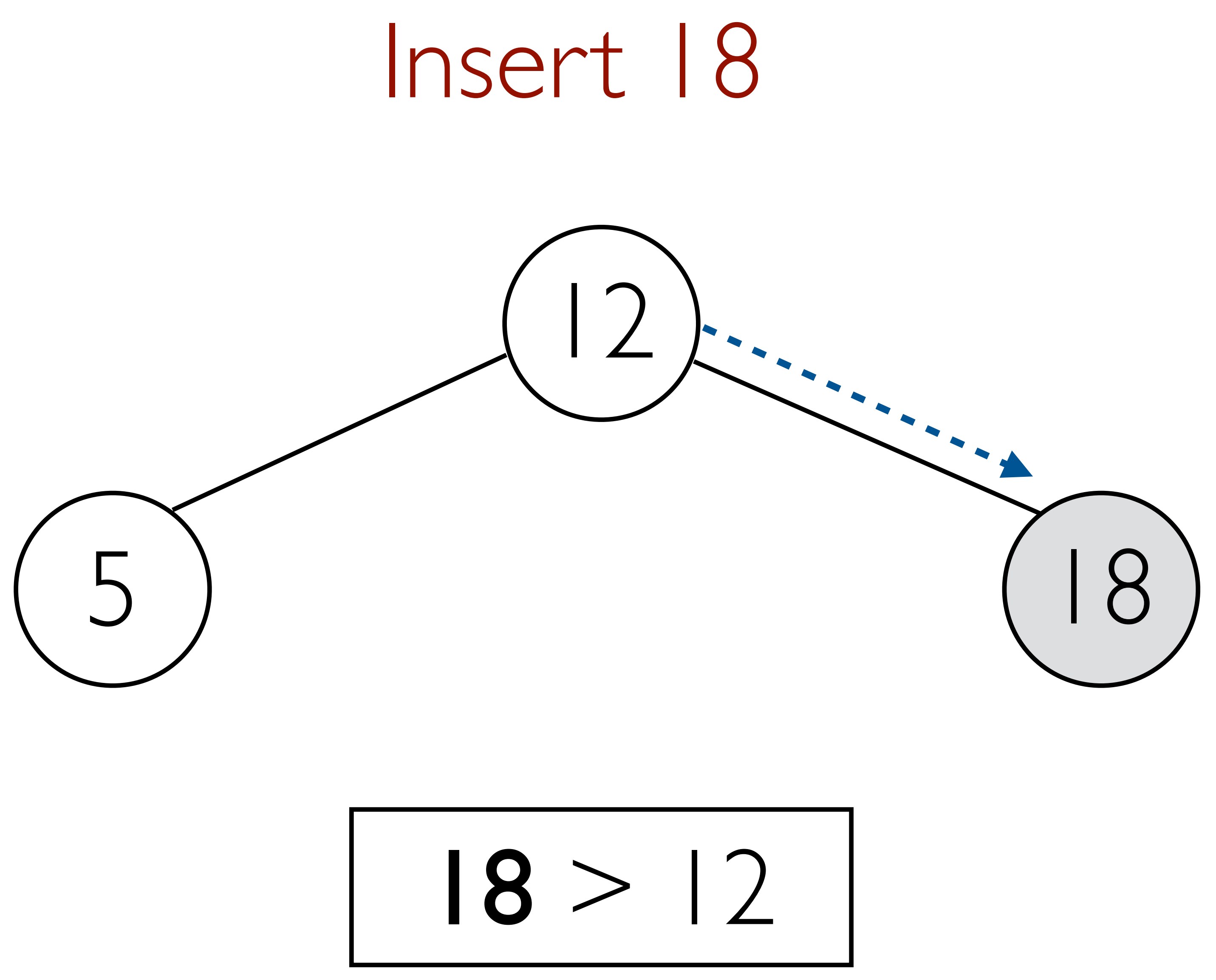
Building a BST

- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property



Building a BST

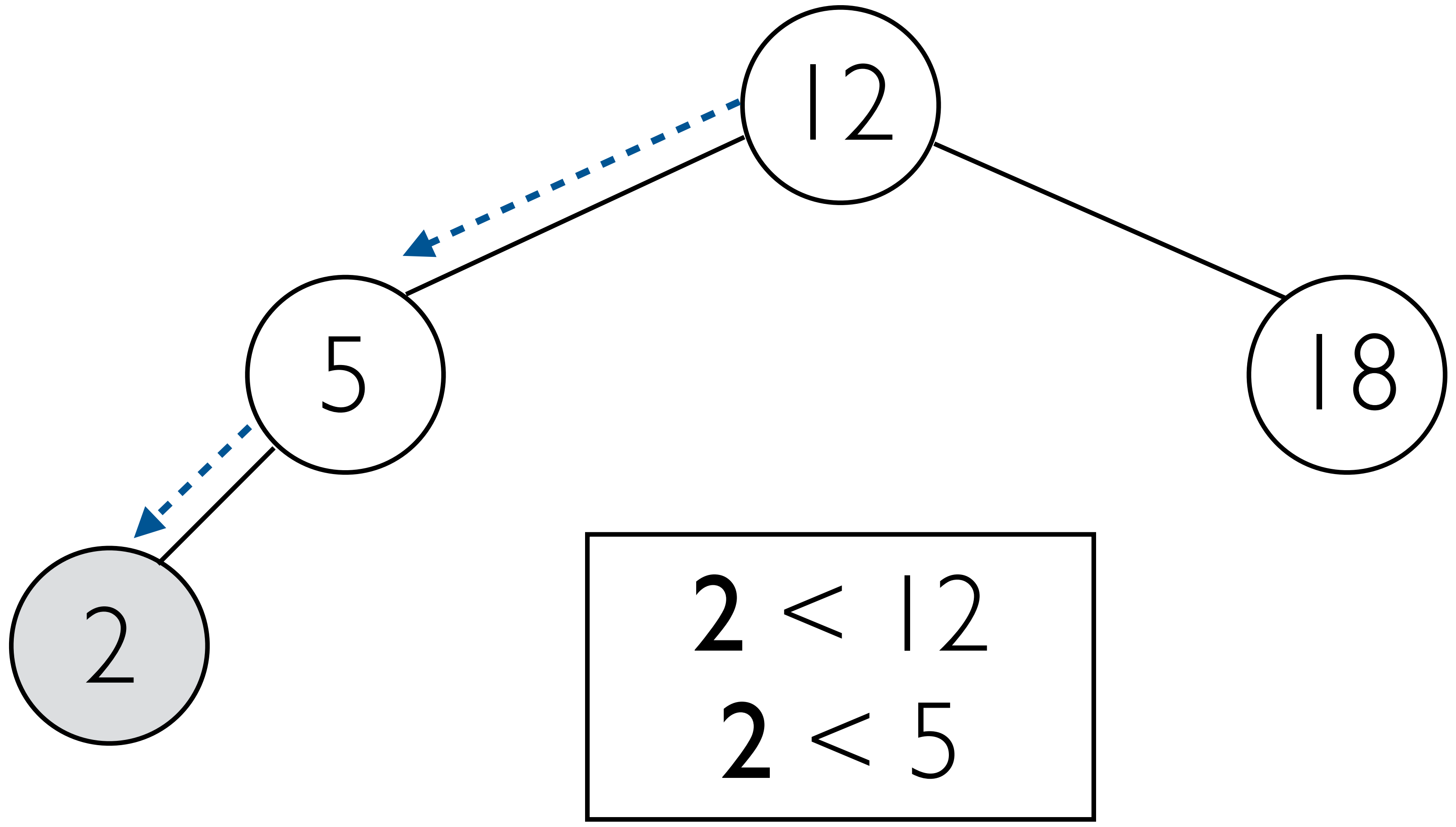
- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property



Building a BST

- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property

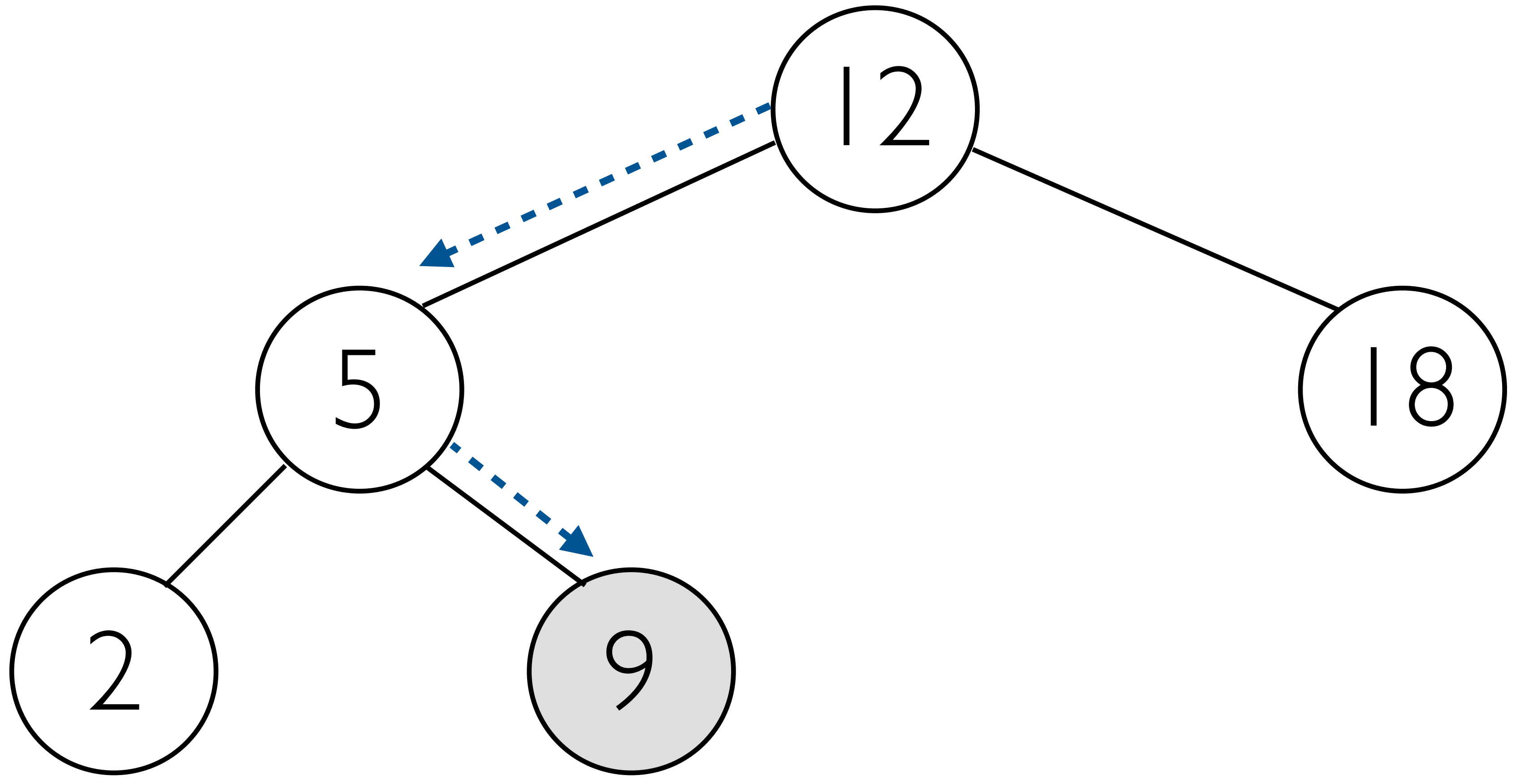
Insert 2



Building a BST

- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property

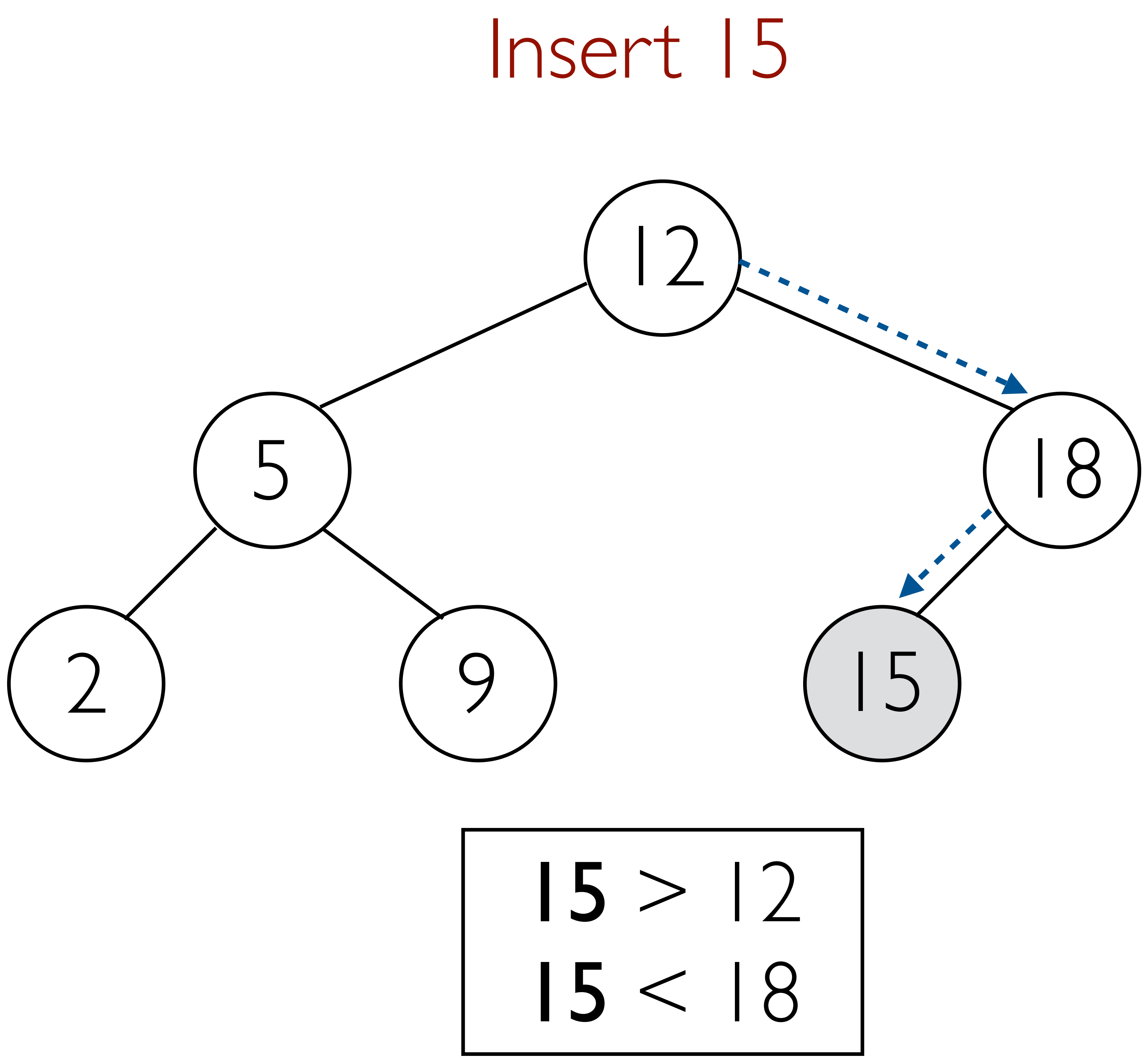
Insert 9



9 < 12
9 > 5

Building a BST

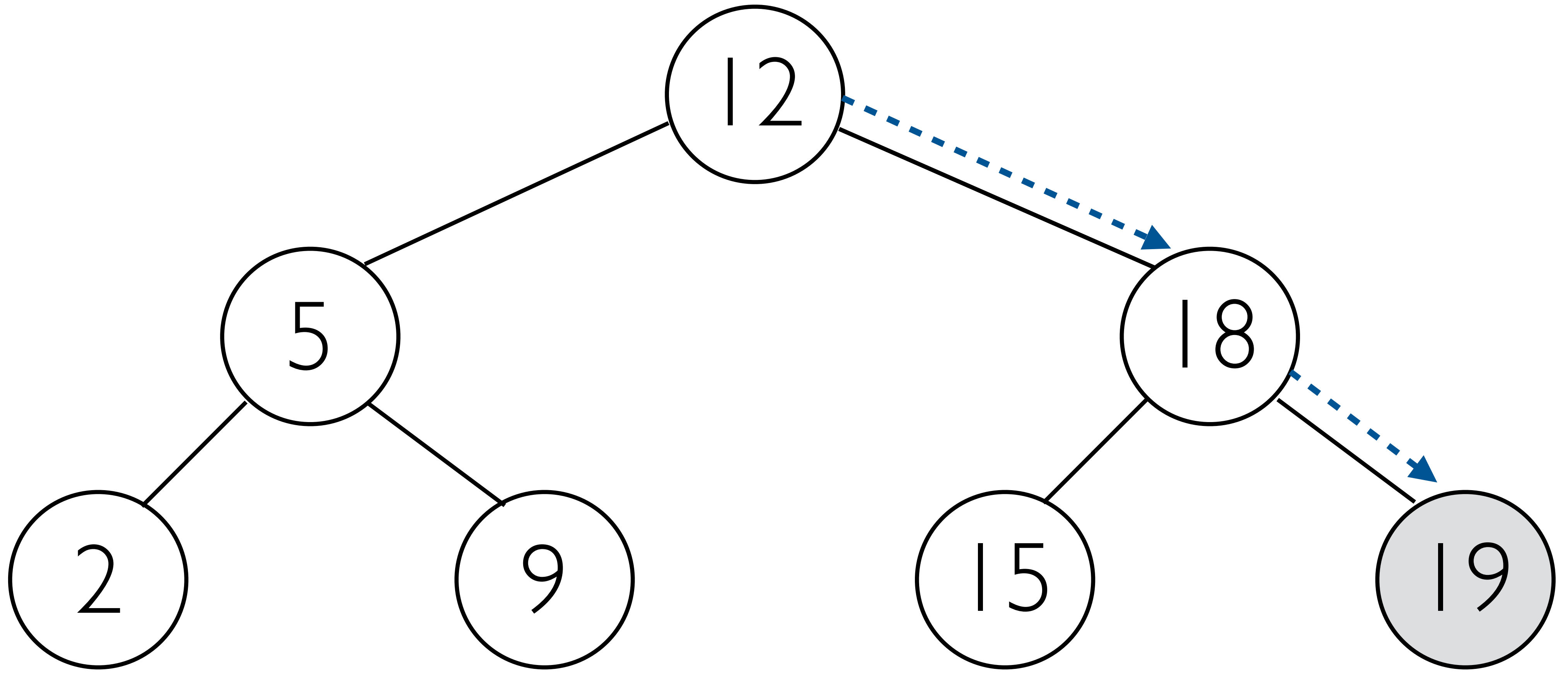
- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property



Building a BST

- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property

Insert 19

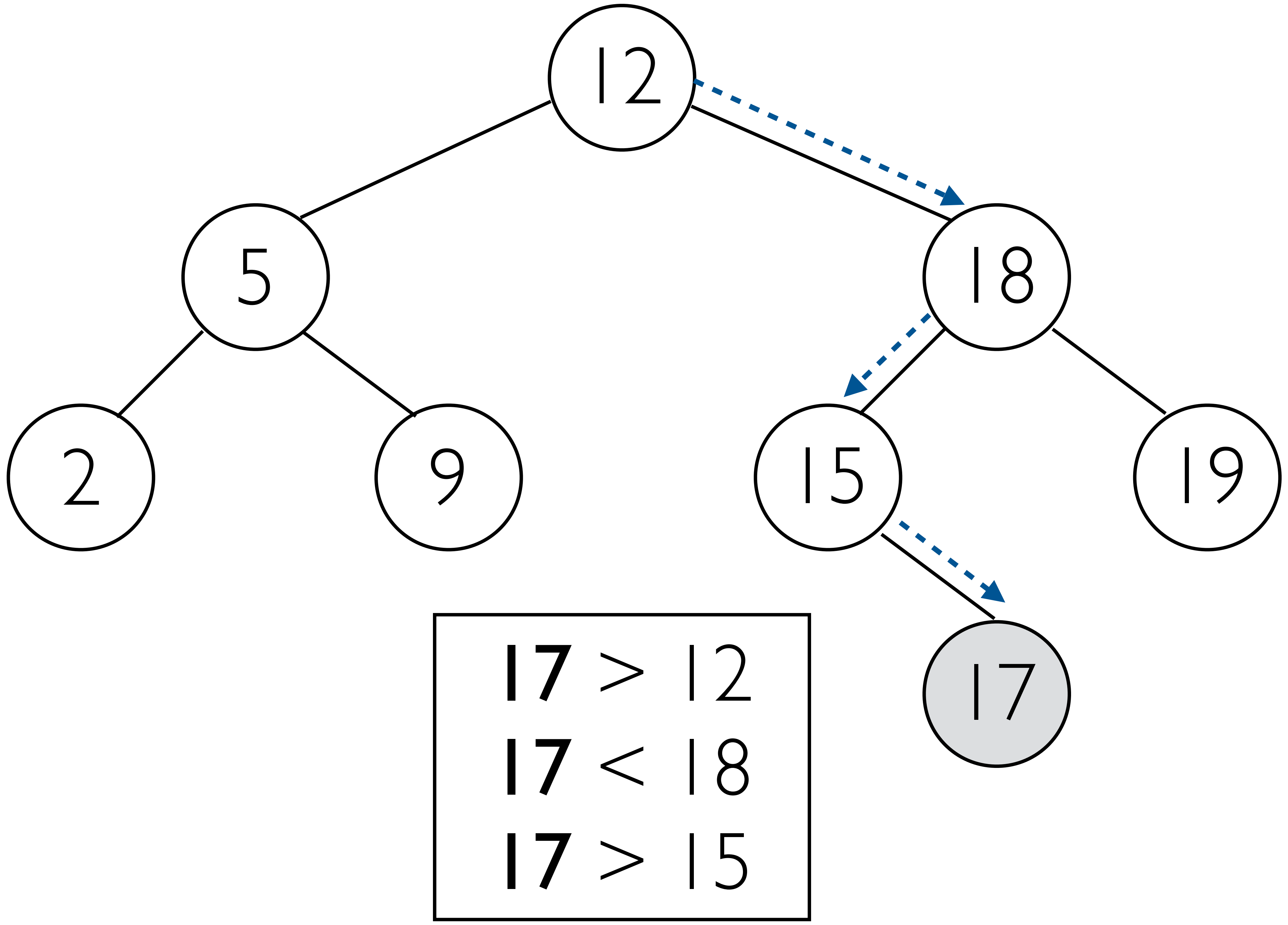


19 > 12
19 > 18

Building a BST

- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property

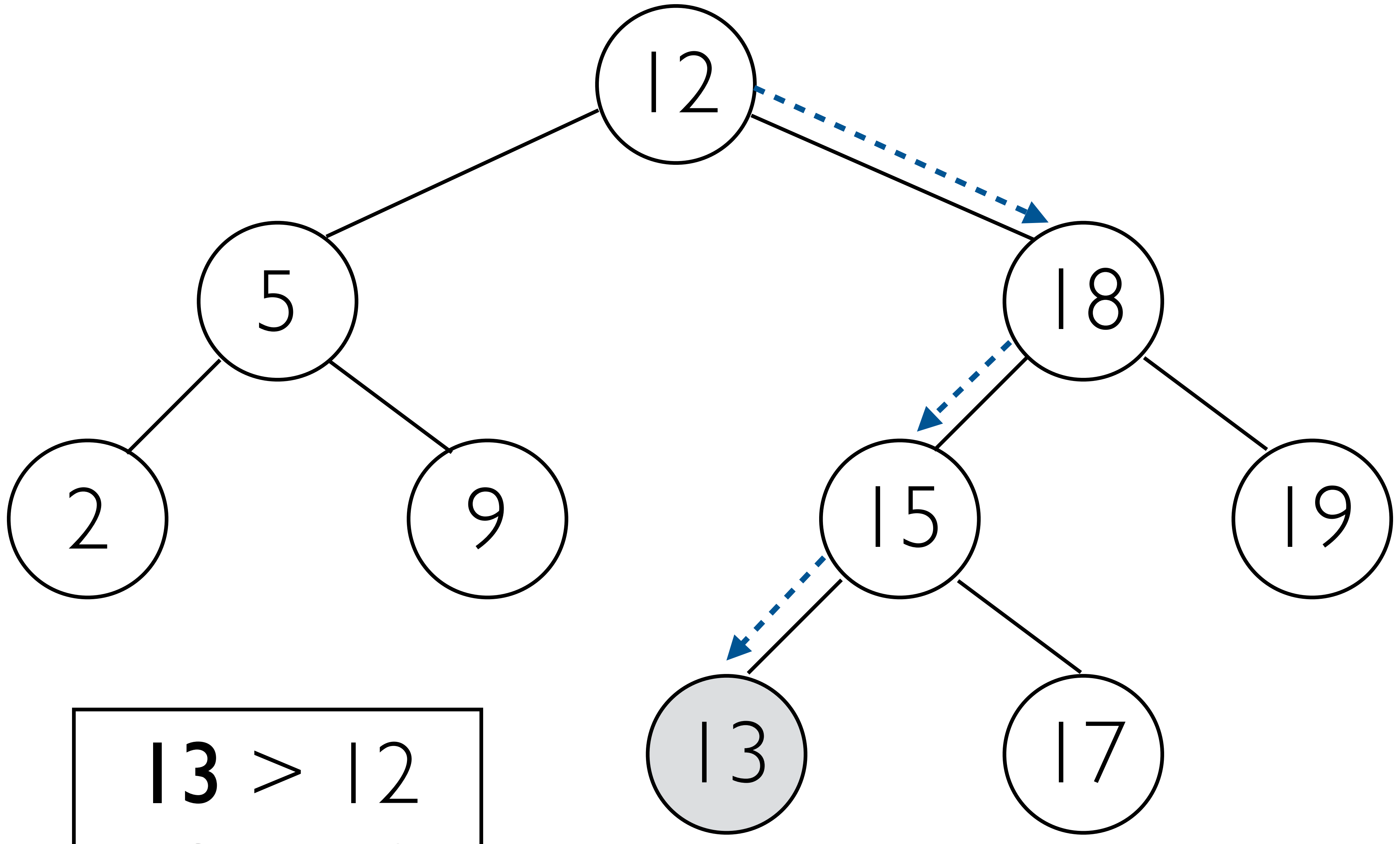
Insert 17



Building a BST

- Insert another node
- Traverse the tree
- Find a place to insert that satisfies BST property

Insert 13



13 > 12
13 < 18
13 < 15

BST in Python

```
class BSTree:

    def __init__(self, root = None):
        self.root = root

    def insert(self, key):
        node, parent, current = Node(key), None, self.getroot()
        while current is not None:
            parent = current
            if node.getkey() < current.getkey():
                current = current.getleft()
            else:
                current = current.getright()
        if parent is None:
            self.setroot(node)
        elif node.getkey() < parent.getkey():
            parent.setleft(node)
        else:
            parent.setright(node)
```

BST in Python

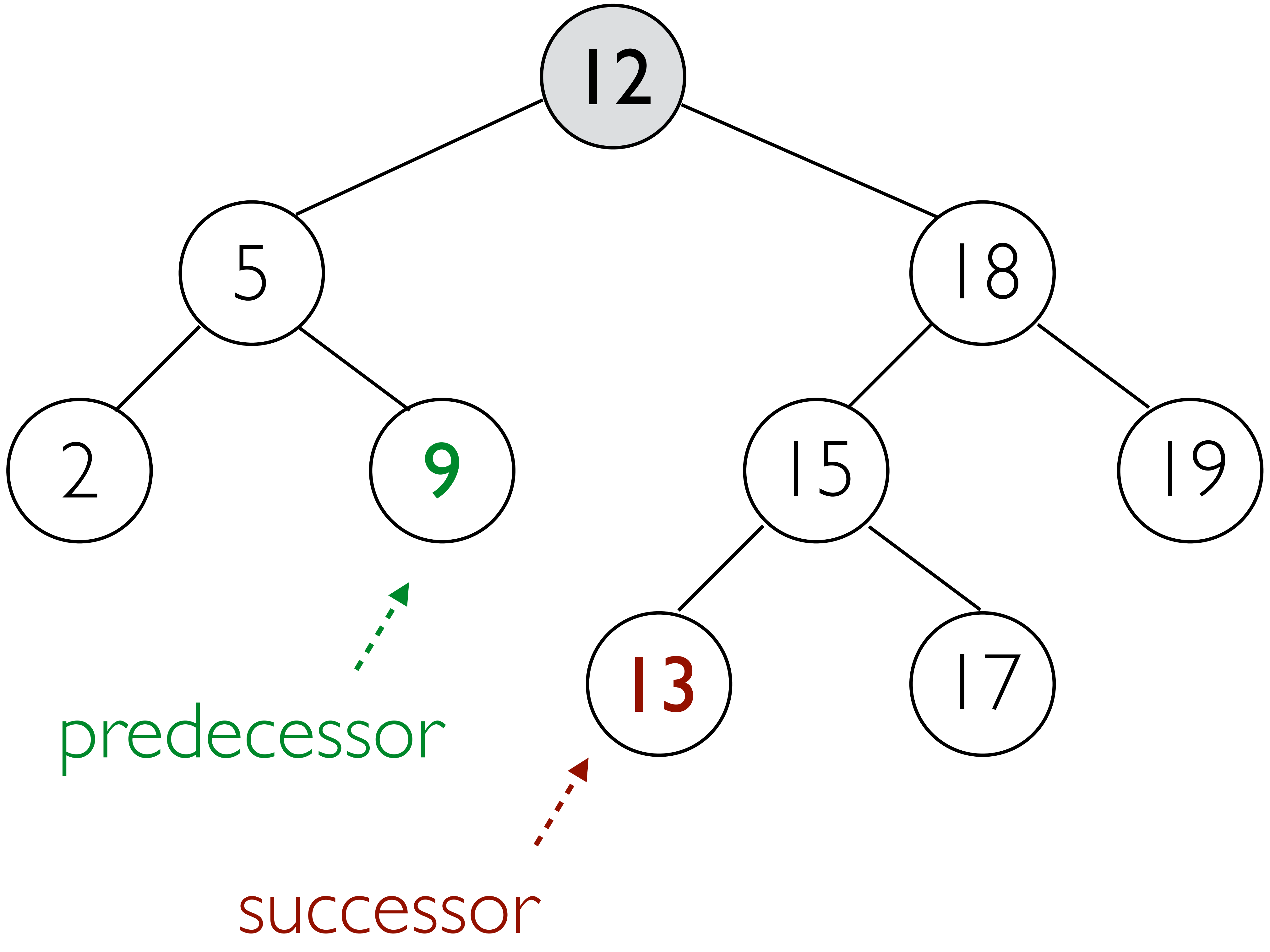
```
class BSTree:

    def __init__(self, root = None):
        self.root = root

    def insert(self, key):
        node, parent, current = Node(key), None, self.getroot()
        while current is not None:
            parent = current
            if node.getkey() < current.getkey():
                current = current.getleft()
            else:
                current = current.getright()
        if parent is None:
            self.setroot(node)
        elif node.getkey() < parent.getkey():
            parent.setleft(node)
        else:
            parent.setright(node)
```

Search for insertion position

Insert node at position

BST vocabulary

- The **successor** of a node **x** is:
  - The node with *<u>smallest</u>* key *<u>greater</u>* than **x.key**
- The **predecessor** of a node **x** is:
  - The node with *<u>largest</u>* key *<u>less</u>* than **x.key**



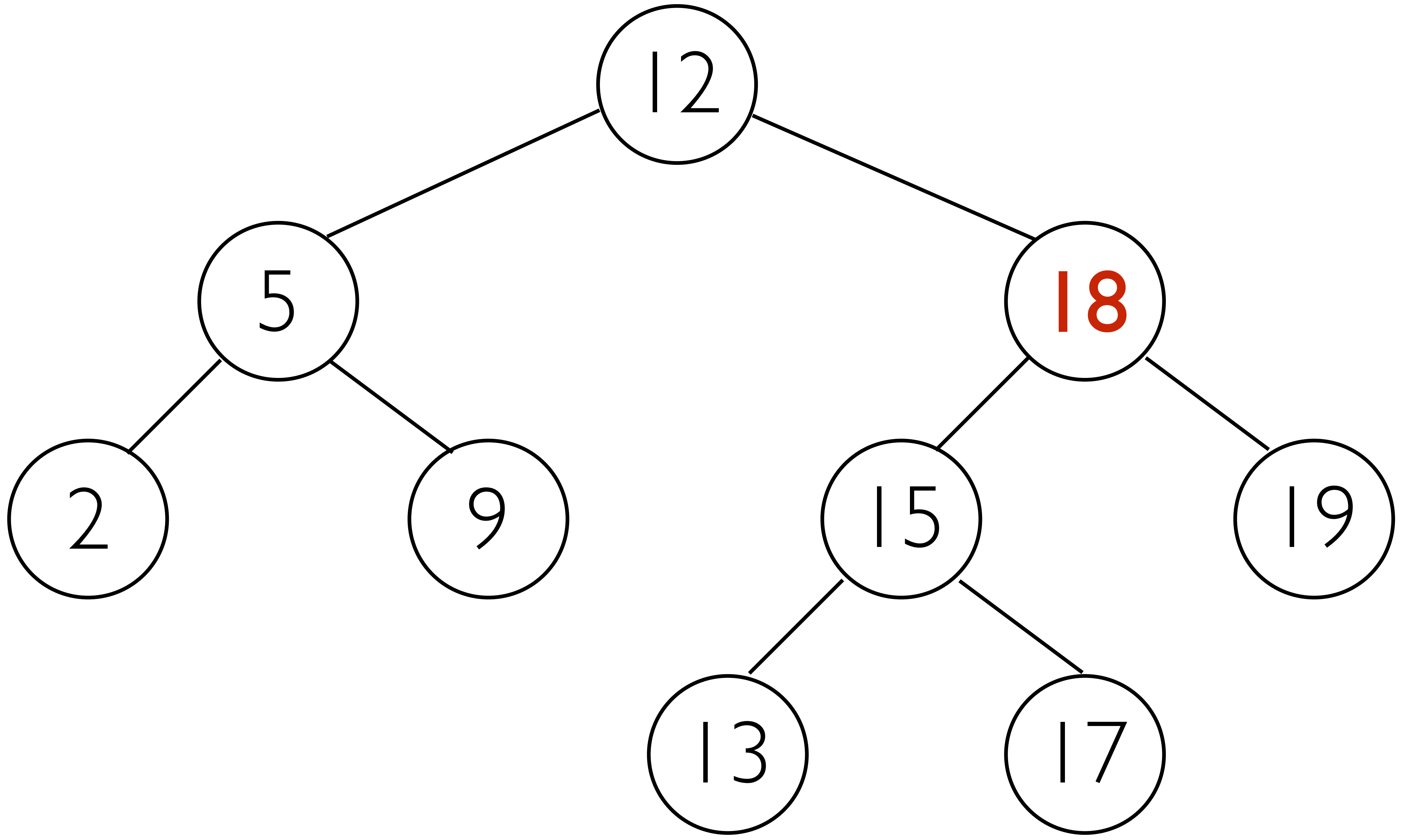
Deleting a node

- If it has *no children*, replace it with None
- If it has *one child*, replace it with its child
- It it has *two children*:
  - Replace it with its **successor**
  - The successor must inherit its subtrees

Deleting a node (simple)

- Delete a node
- The node has children
- Find its **successor**

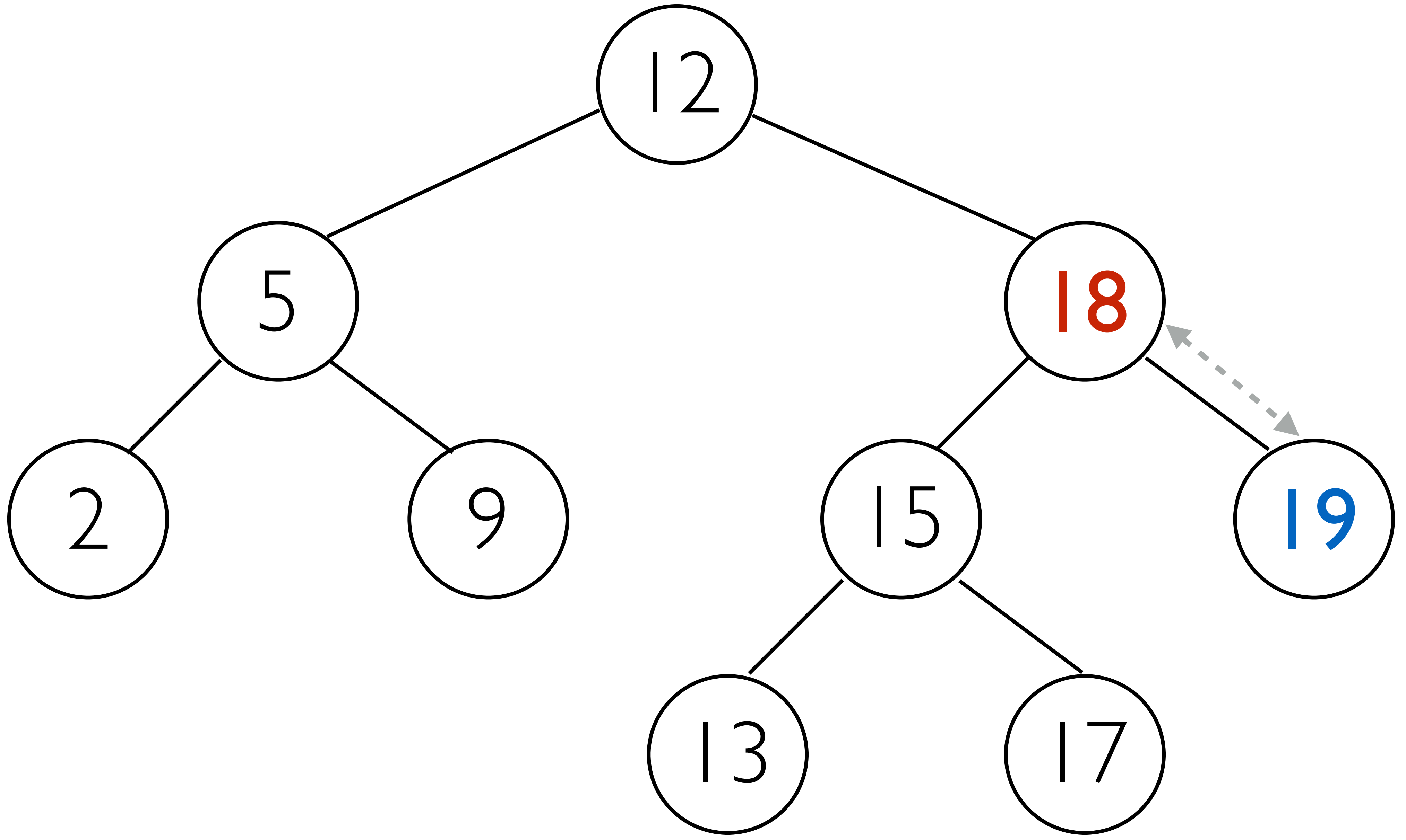
Delete 18



Deleting a node (simple)

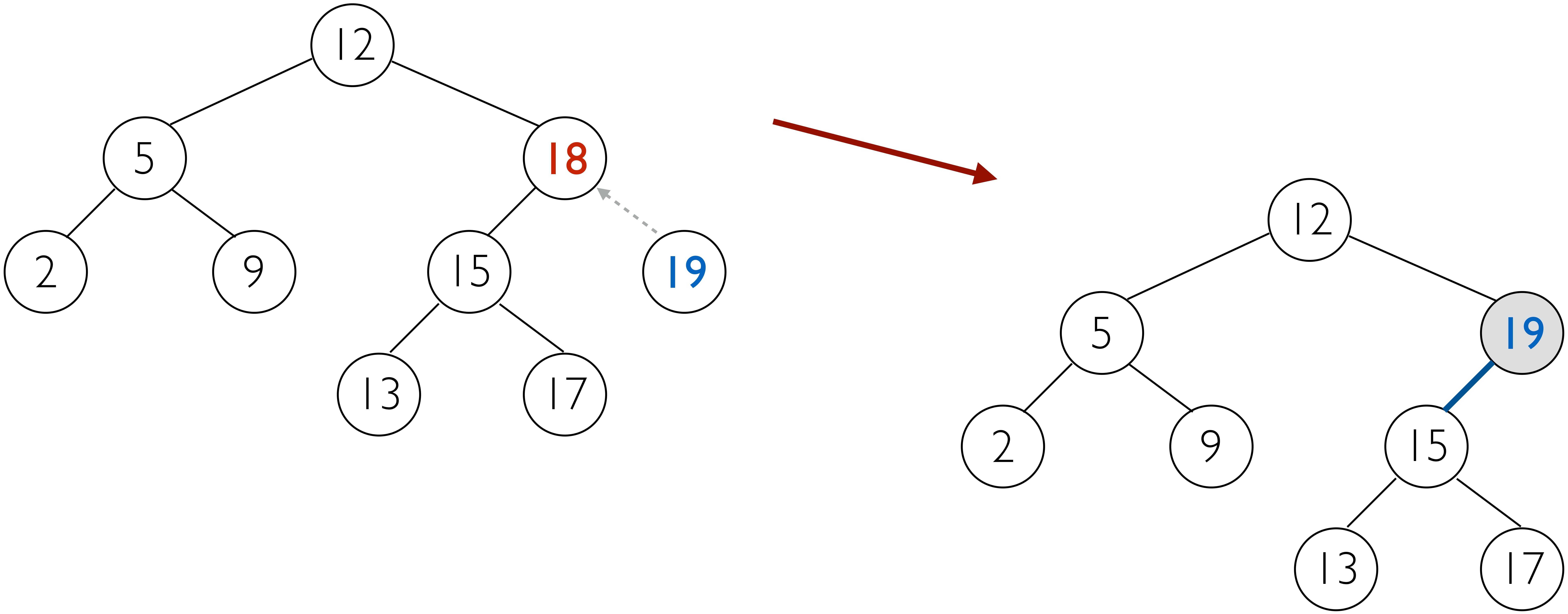
- Delete a node
- The node has children
- The **successor** is its <u>child</u>
  - That makes it easy!

Delete 18



Deleting a node (simple)

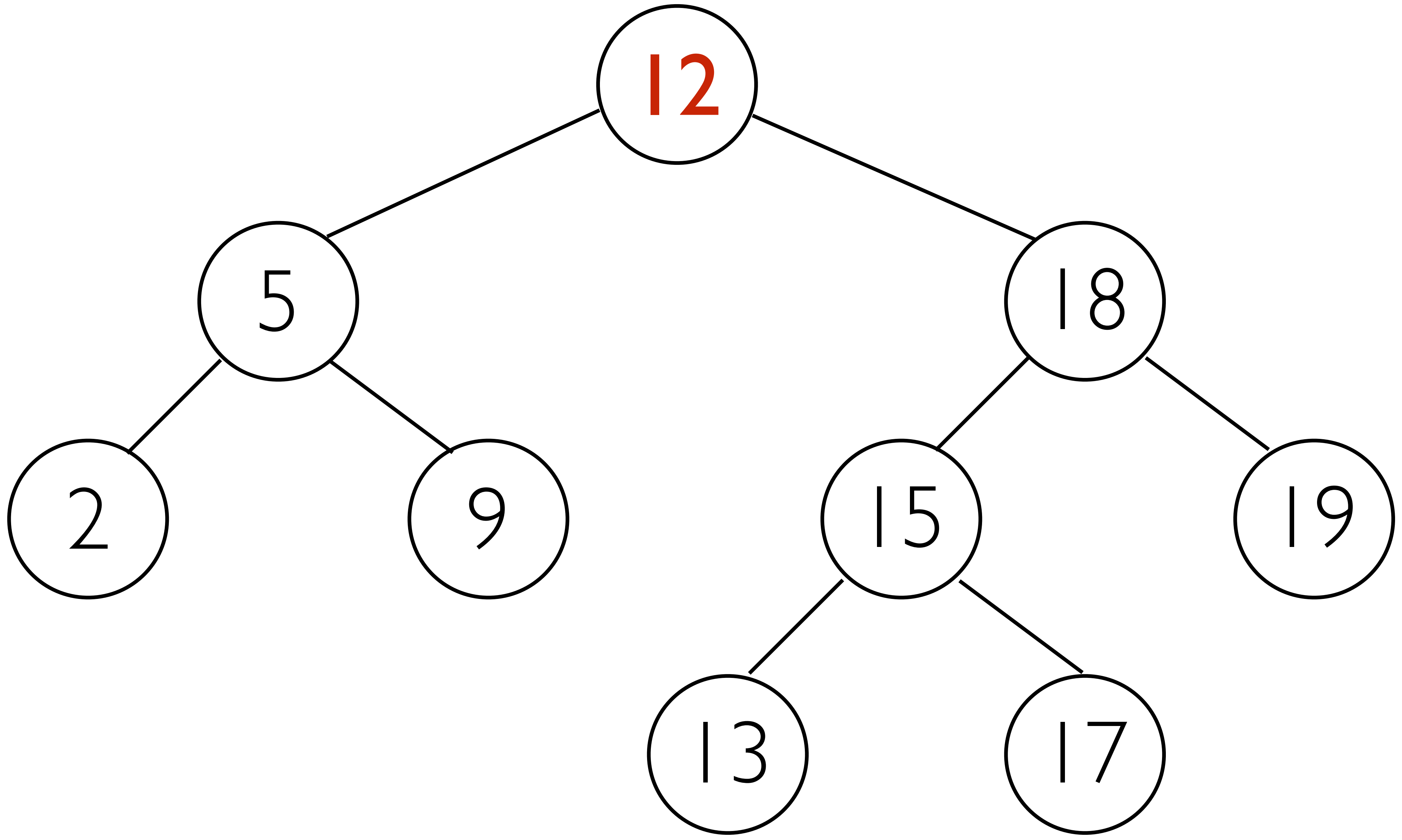
Delete 18



Deleting a node (complex)

- Delete a node
- The node has children
- Find its **successor**

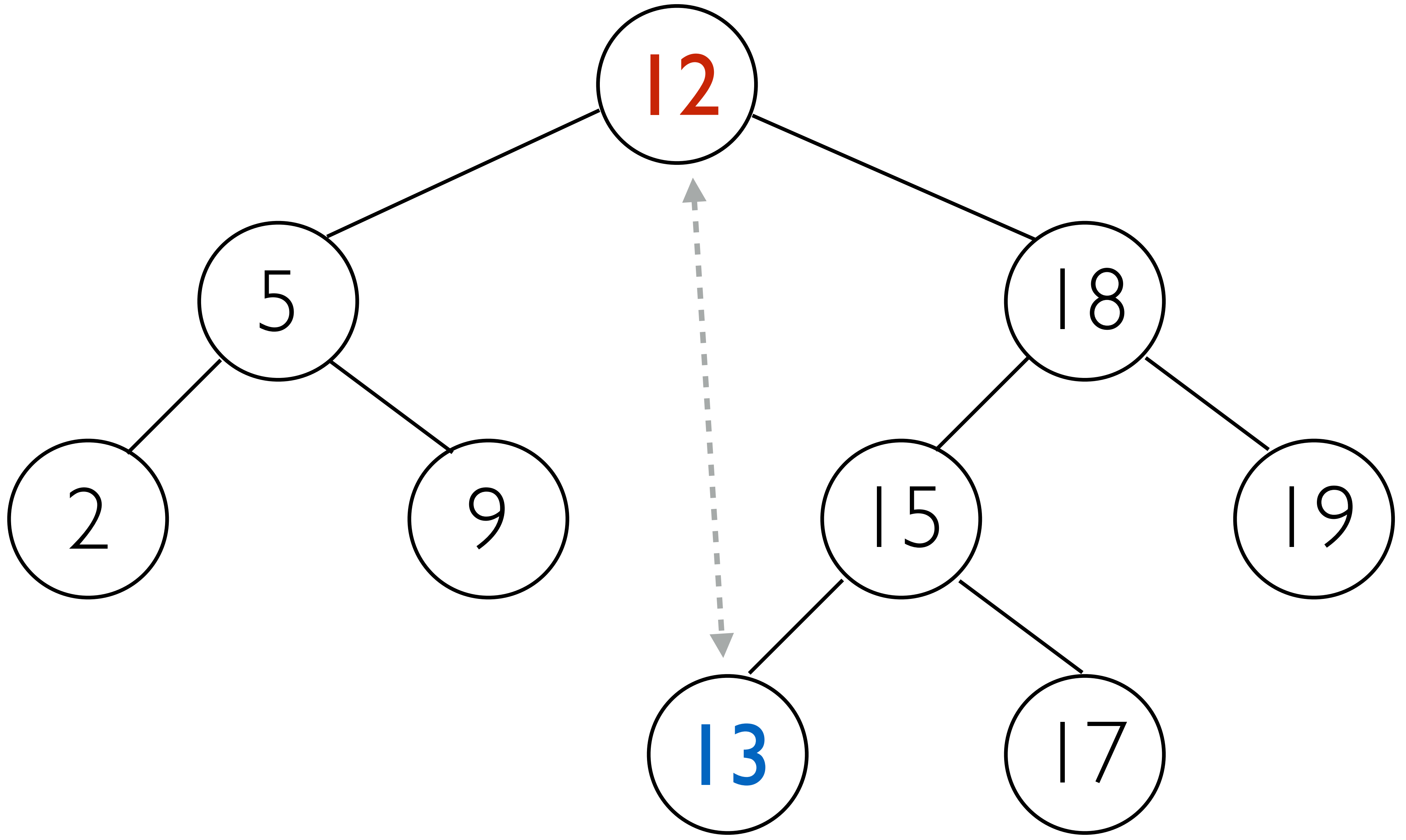
Delete 12



Deleting a node (complex)

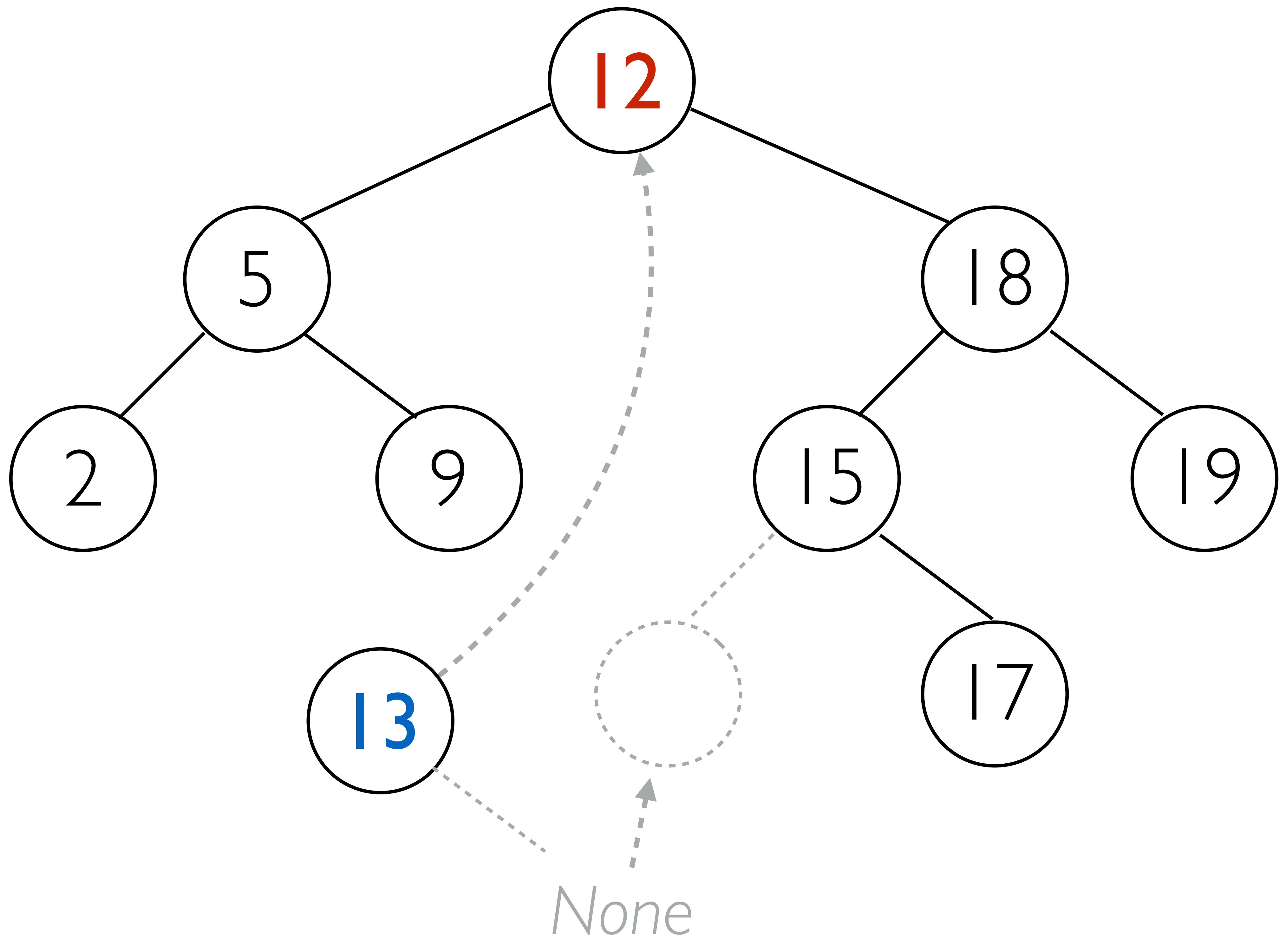
- Delete a node
- The node has children
- The **successor** is in a <u>subtree</u>
  - That makes it more complicated

Delete 12



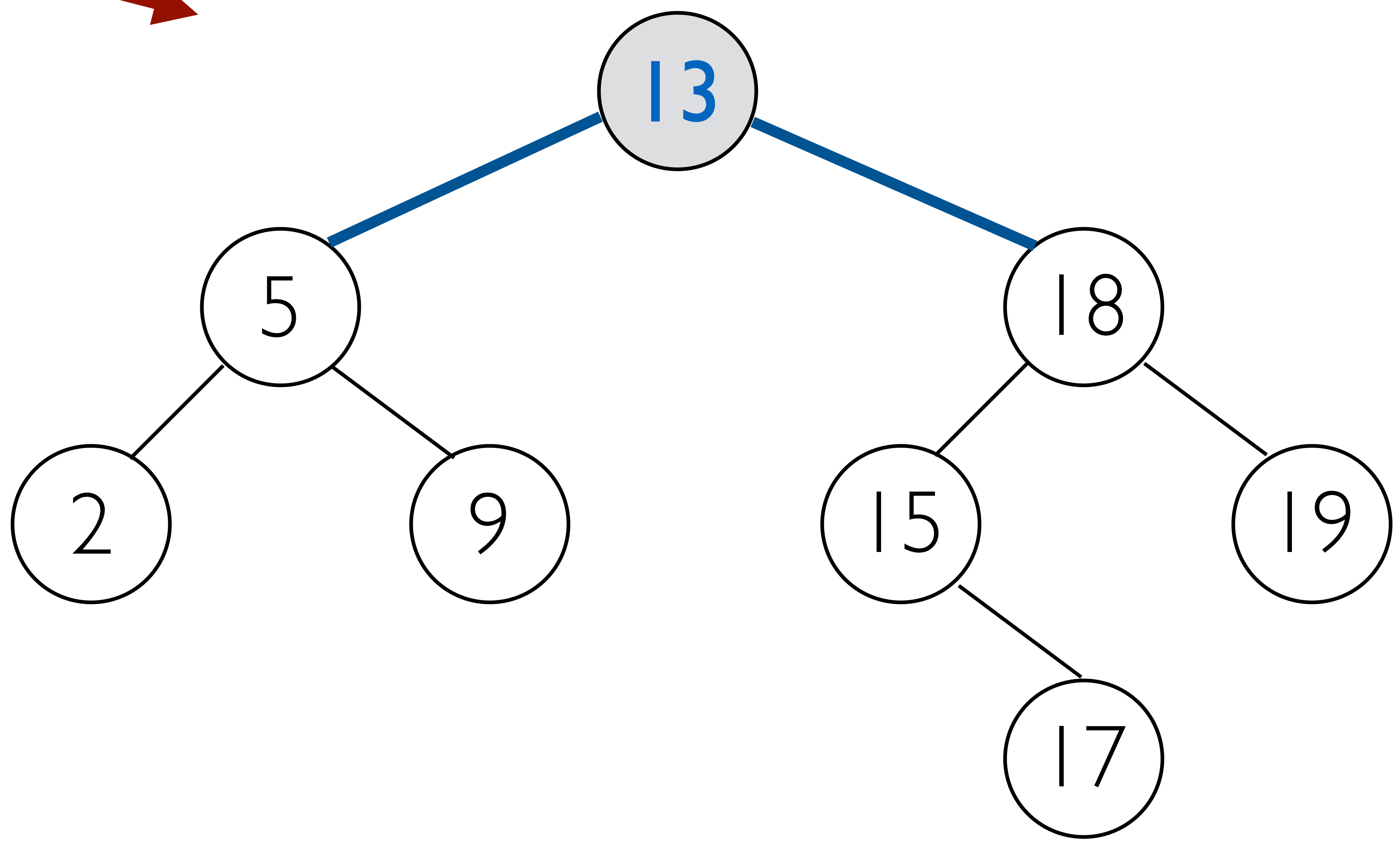
Deleting a node (complex)

Delete 12



Splice successor out of its position

Transplant to replace deleted node

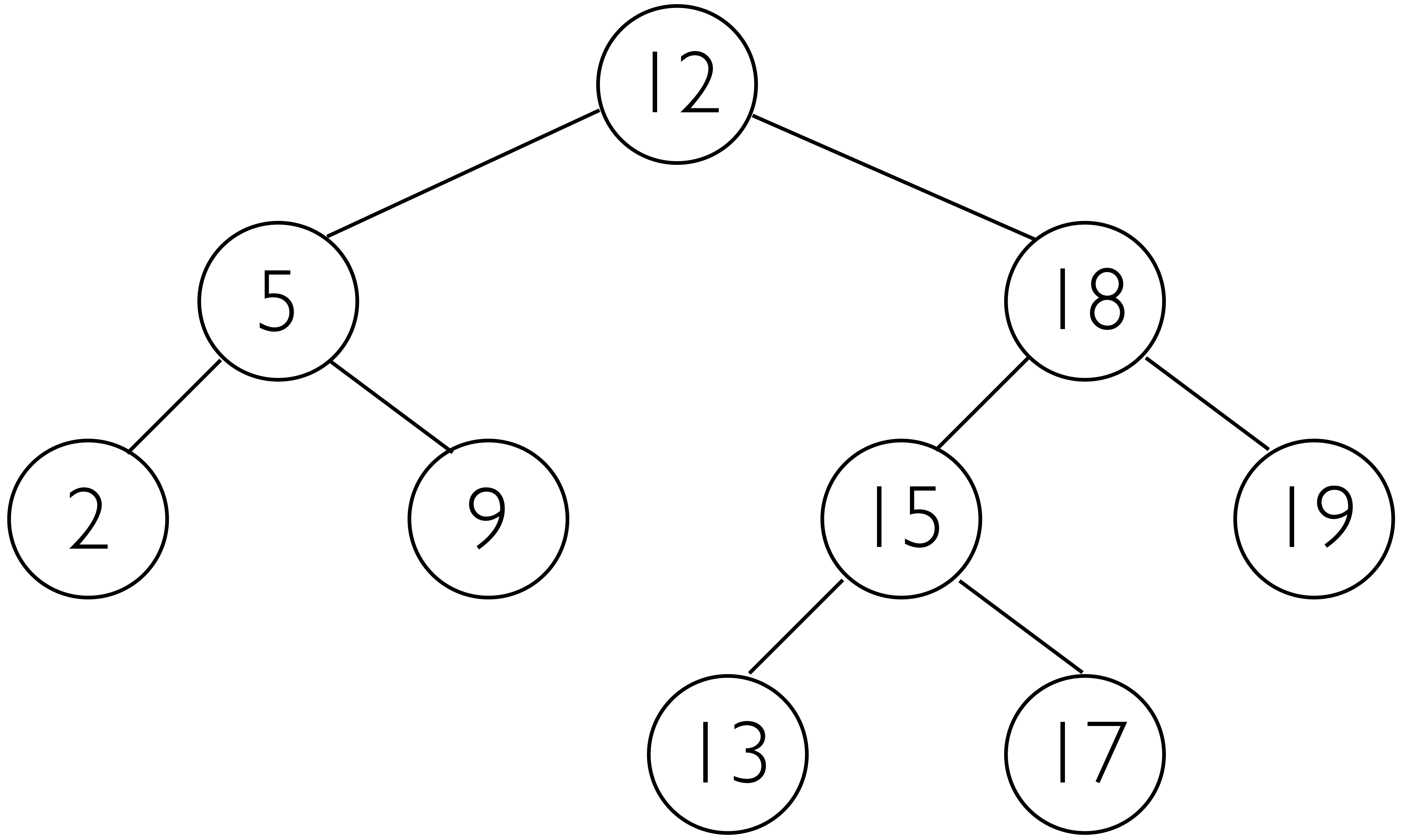


Traversing a tree

- A tree is a **nonlinear** data type
- There are multiple ways to traverse one
- **Depth-first-search** (DFS) explores as *deeply* as possible before backtracking
- **Breadth-first-search** (BFS) explores as *widely* as possible before backtracking

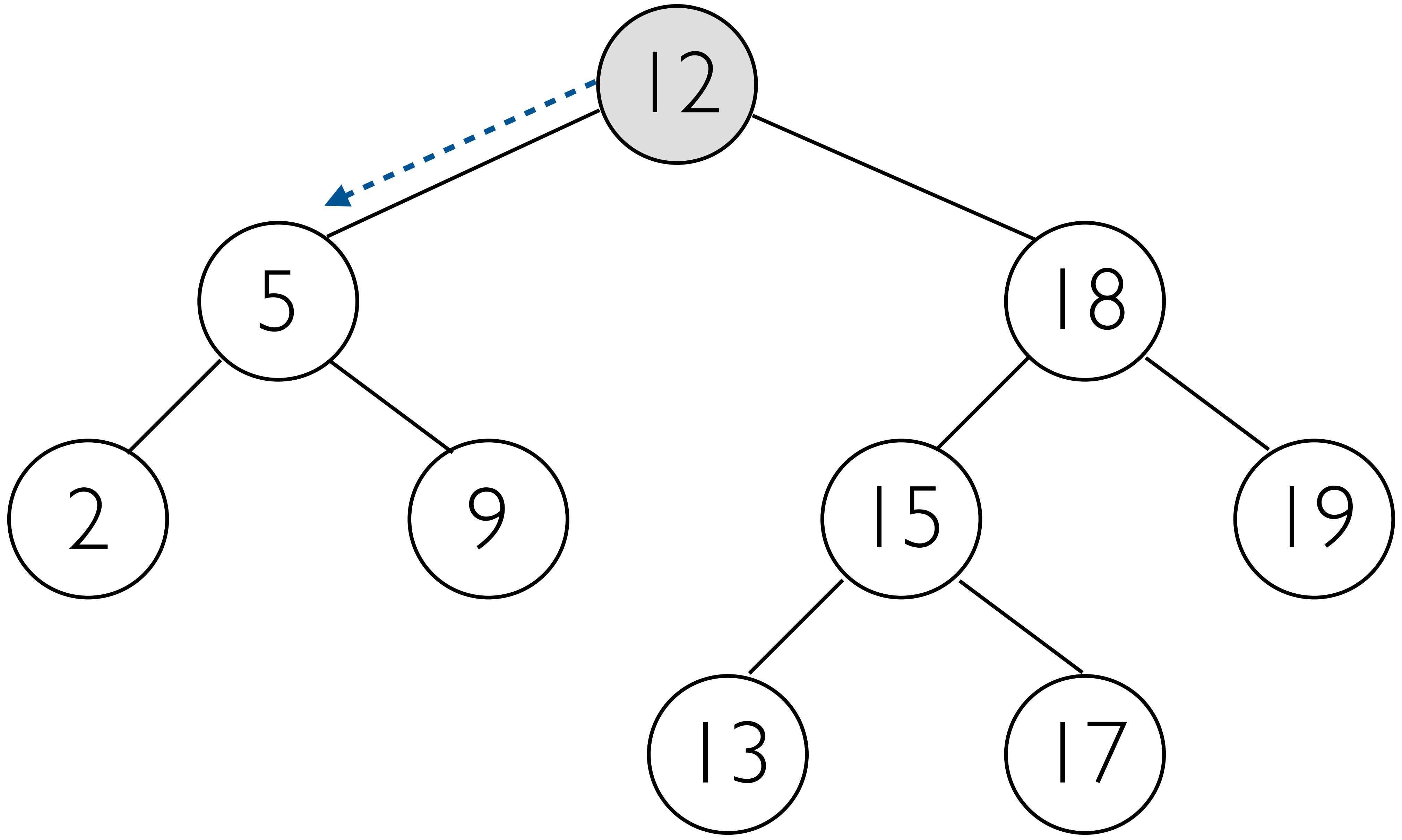
Depth-first traversal

- **Pre-order**: Report nodes as they are *visited*
- **In-order**: Report nodes as they are *backtracked*



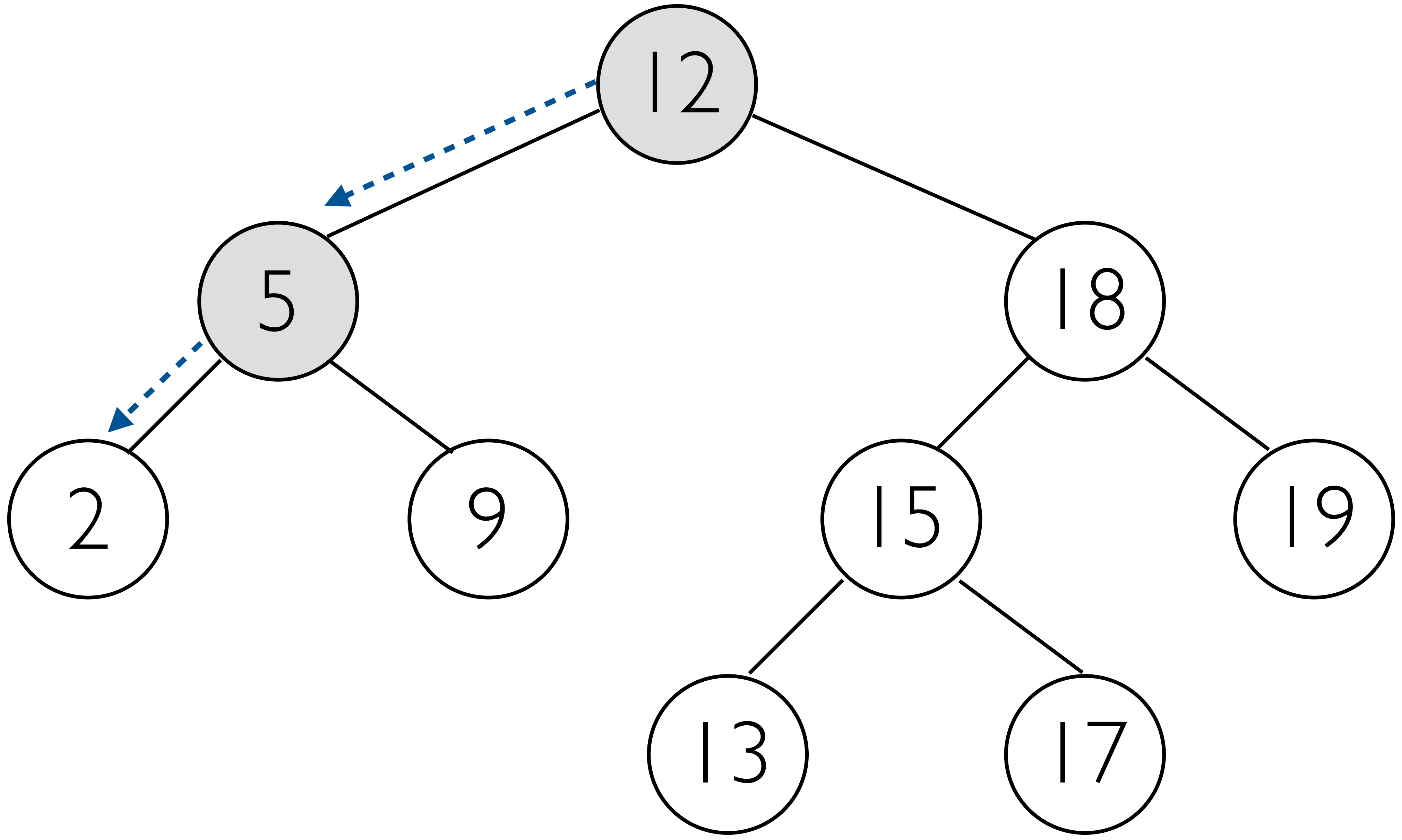
Depth-first traversal

- **Pre-order:**
  - 12
- **In-order:**
  - -



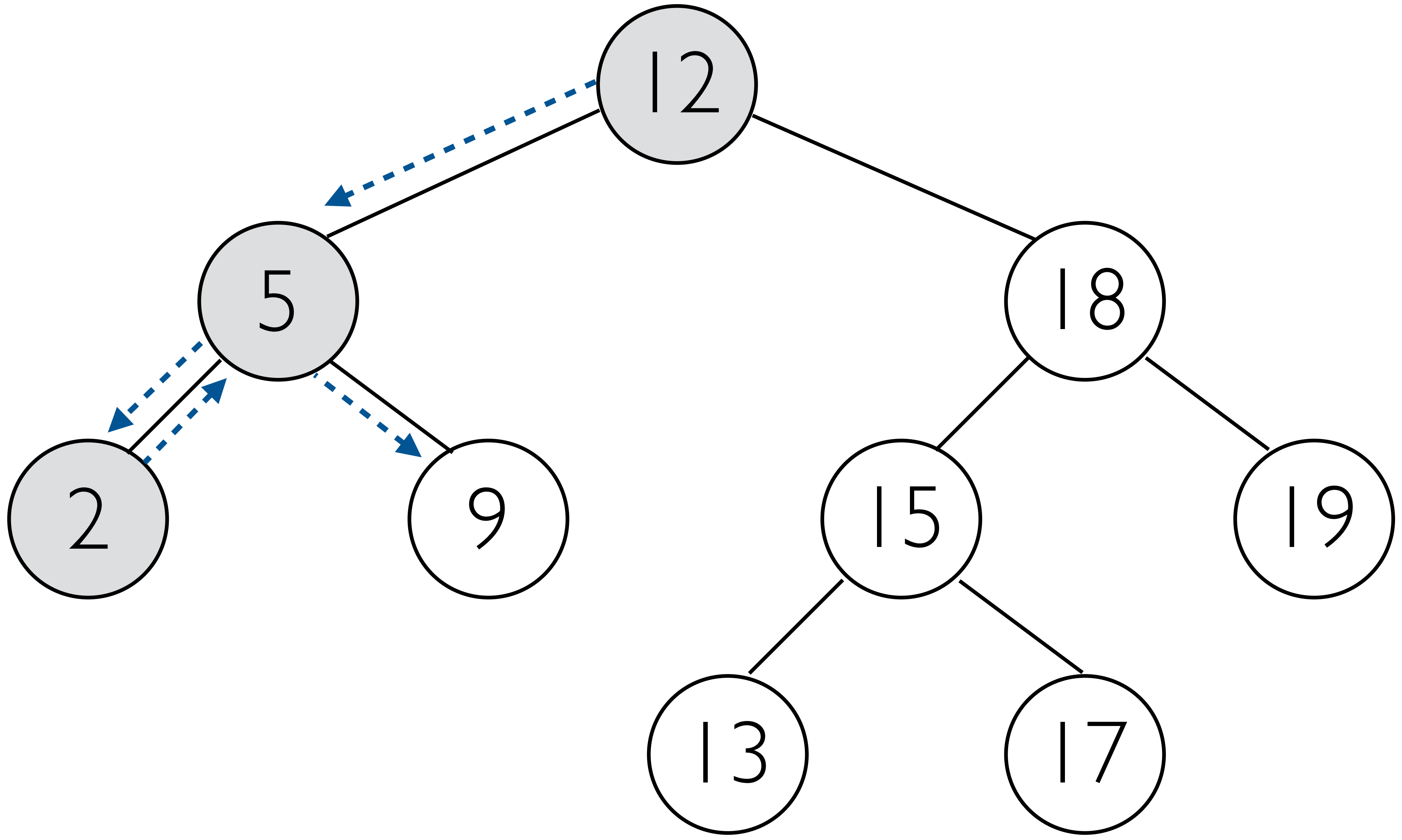
Depth-first traversal

- **Pre-order:**
  - 12→5
- **In-order:**
  - -



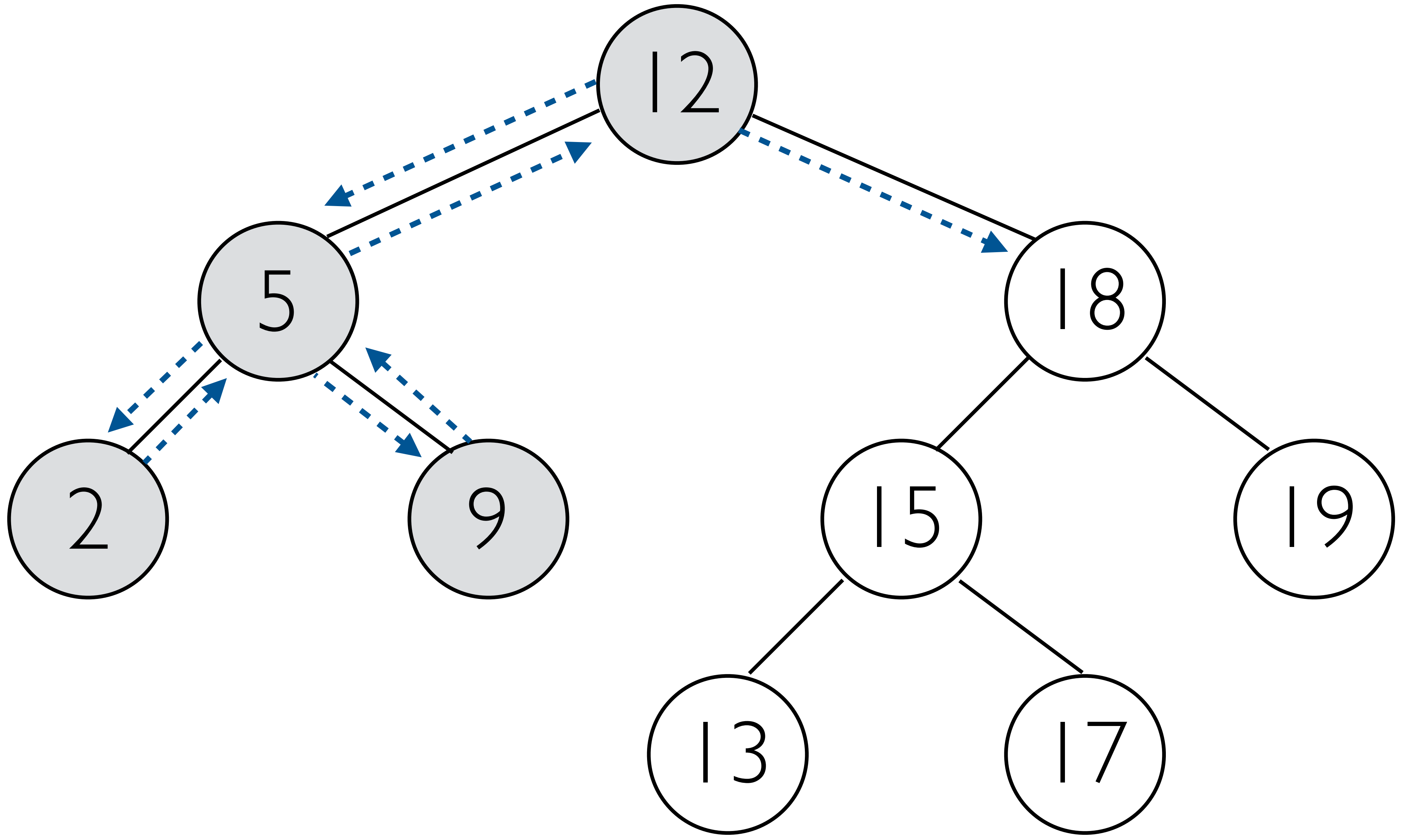
Depth-first traversal

- **Pre-order:**
  - 12→5→2
- **In-order:**
  - 2→5



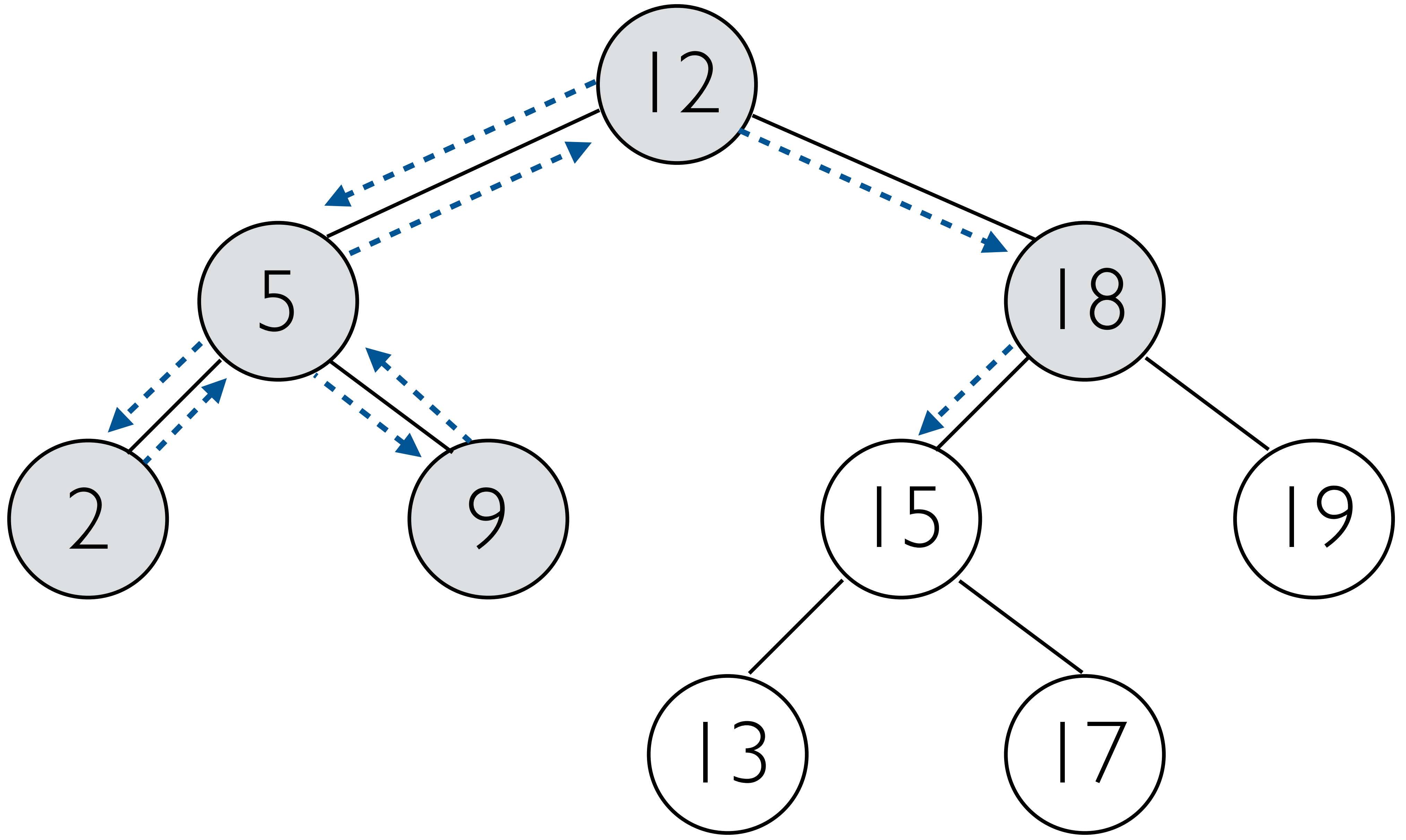
Depth-first traversal

- **Pre-order:**
  - 12→5→2→9
- **In-order:**
  - 2→5→9→12



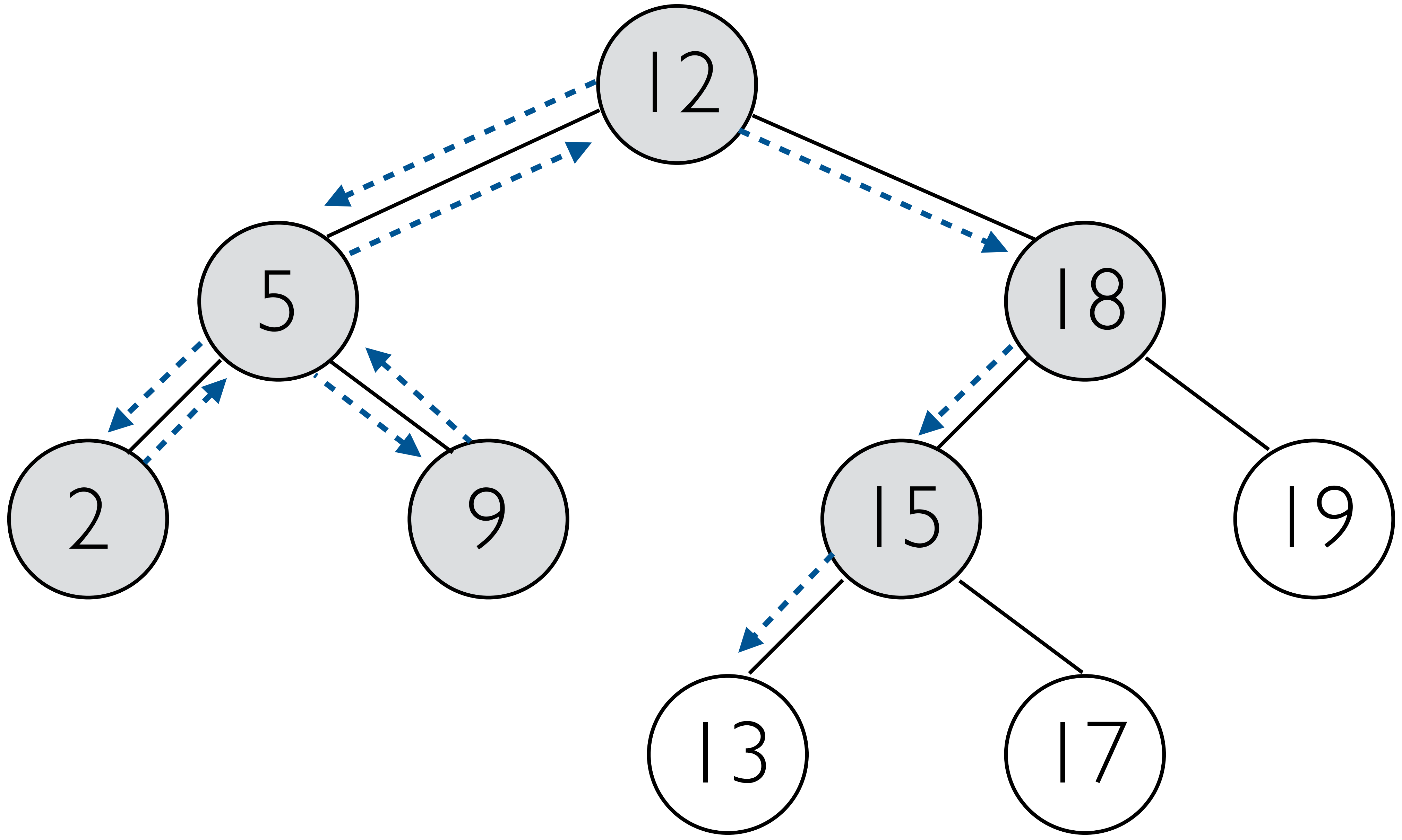
Depth-first traversal

- **Pre-order:**
  - 12→5→2→9→18
- **In-order:**
  - 2→5→9→12



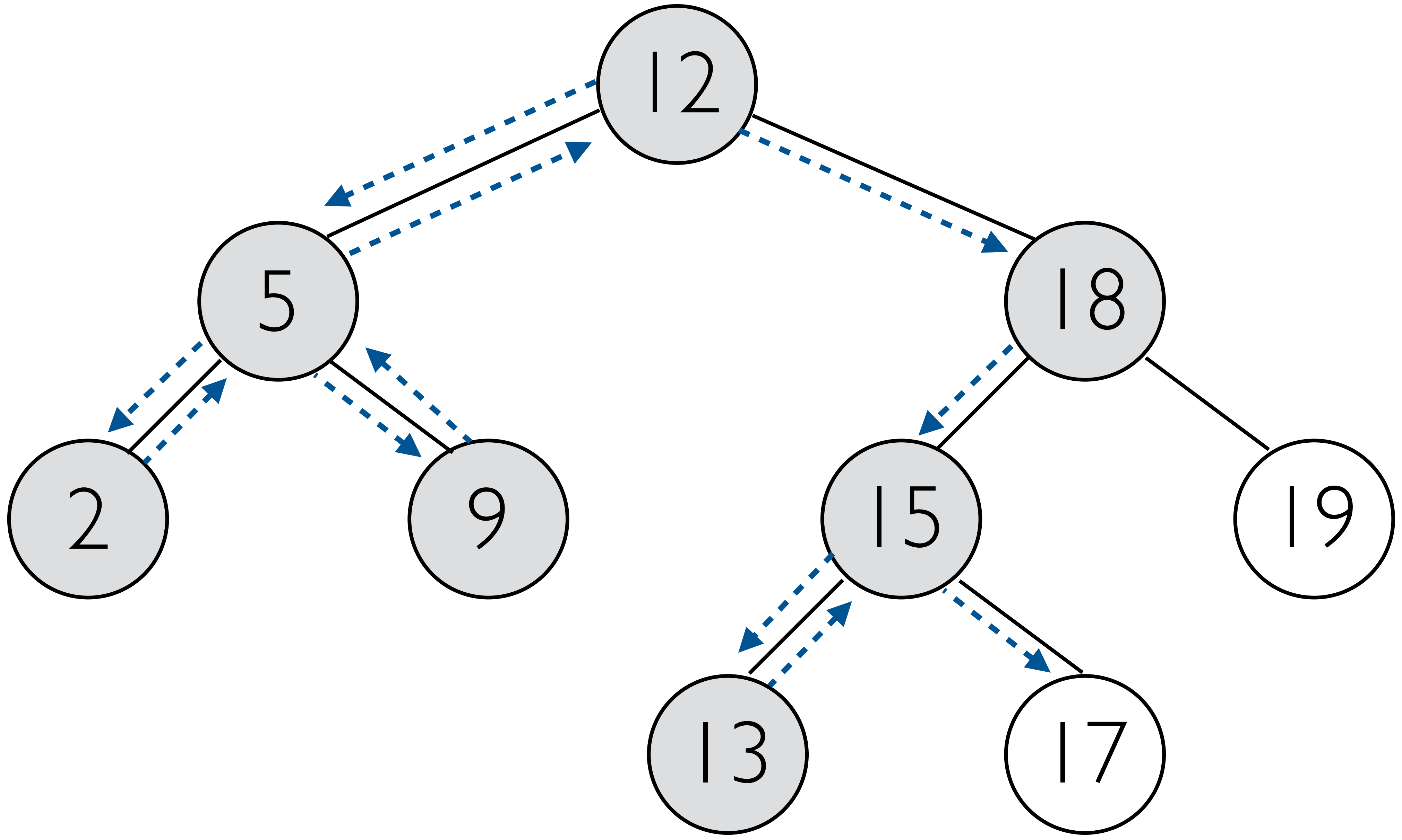
Depth-first traversal

- **Pre-order:**
  - 12→5→2→9→18→15
- **In-order:**
  - 2→5→9→12



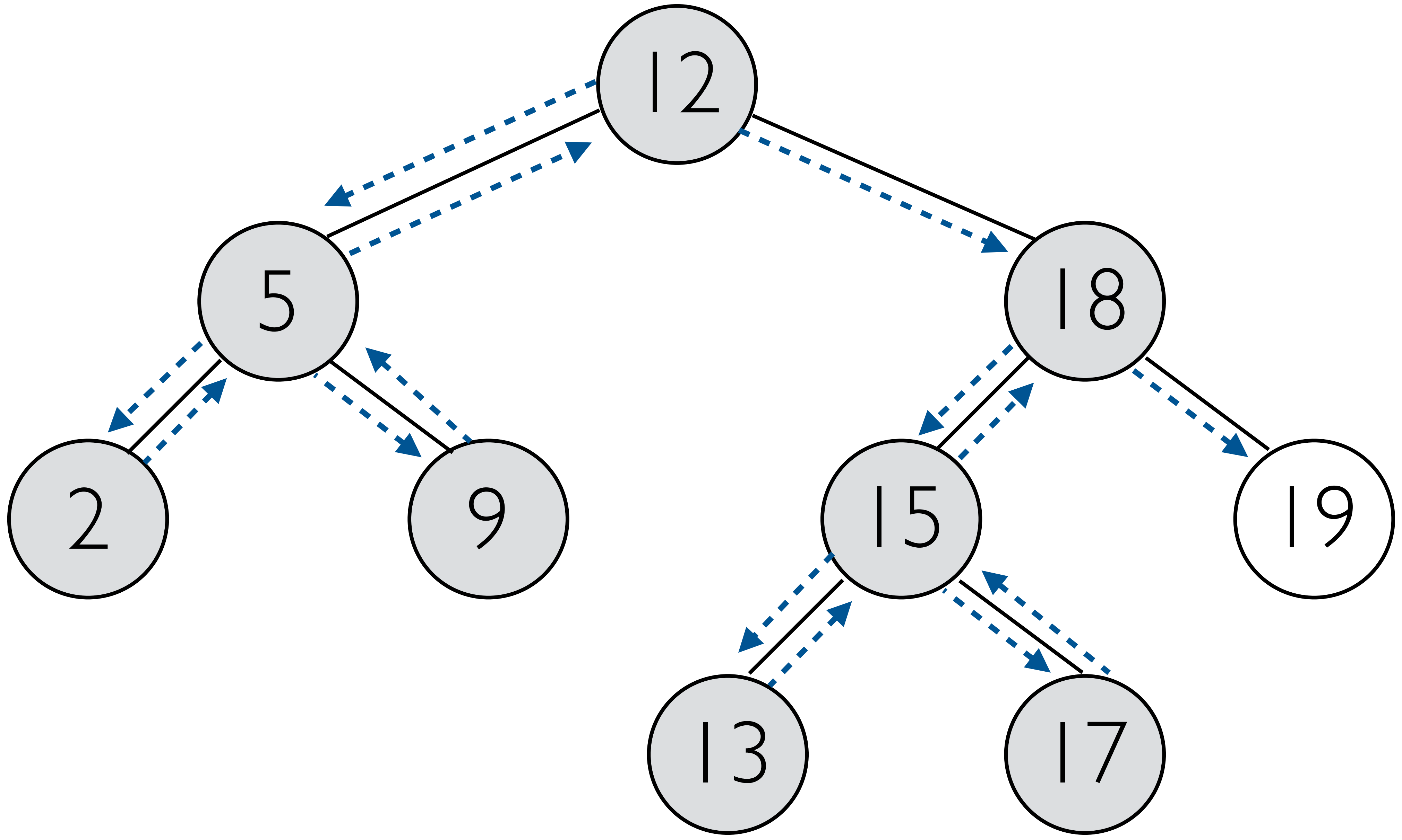
Depth-first traversal

- **Pre-order:**
  - 12→5→2→9→18→15→13
- **In-order:**
  - 2→5→9→12→13→15



Depth-first traversal

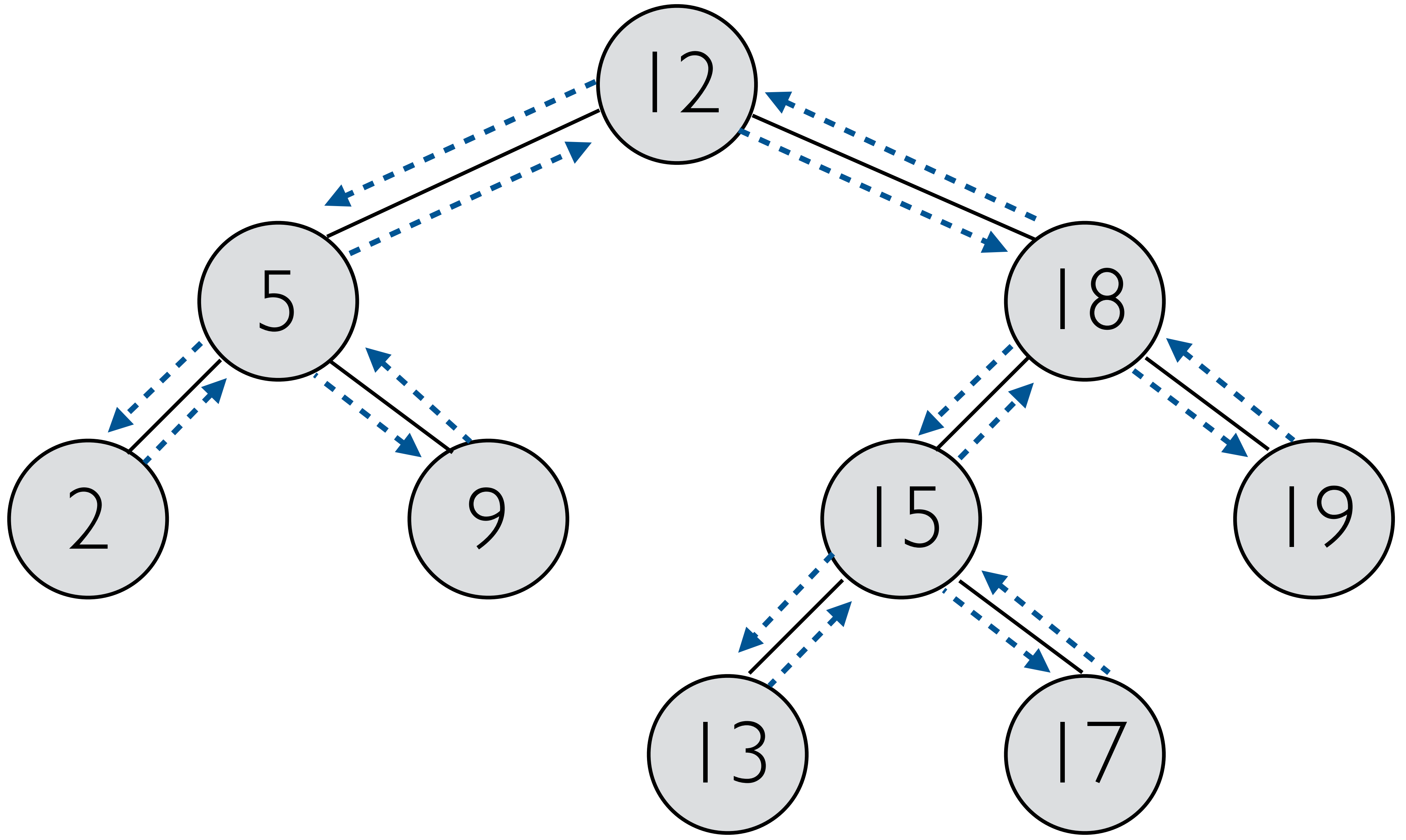
- **Pre-order:**
  - 12→5→2→9→18→15→13→17→19
- **In-order:**
  - 2→5→9→12→13→15→17→18



Depth-first traversal

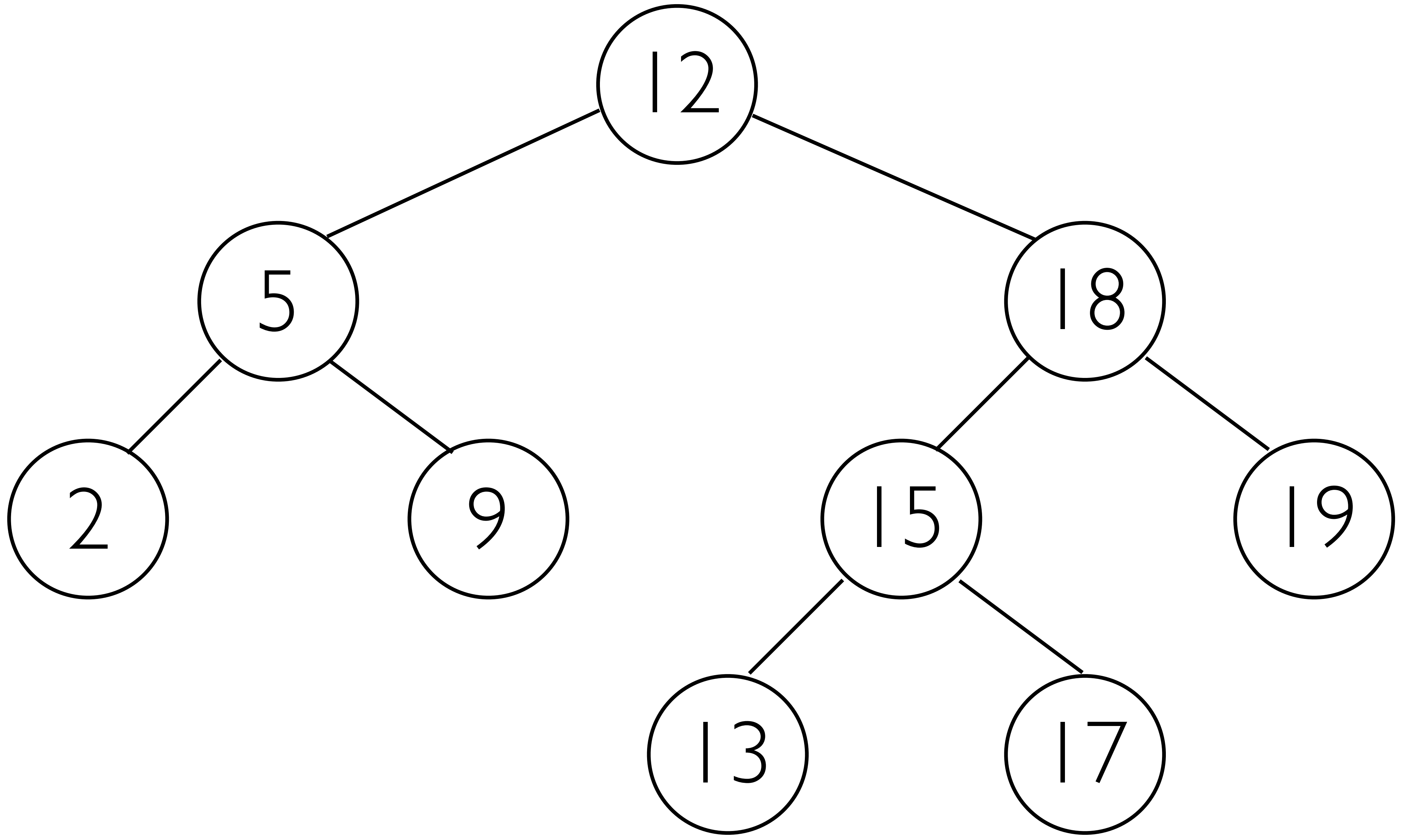
- **Pre-order:**
  - 12→5→2→9→18→15→13→17→19
- **In-order:**
  - 2→5→9→12→13→15→17→18→19

In-order traversal of a BST gives sorted keys!



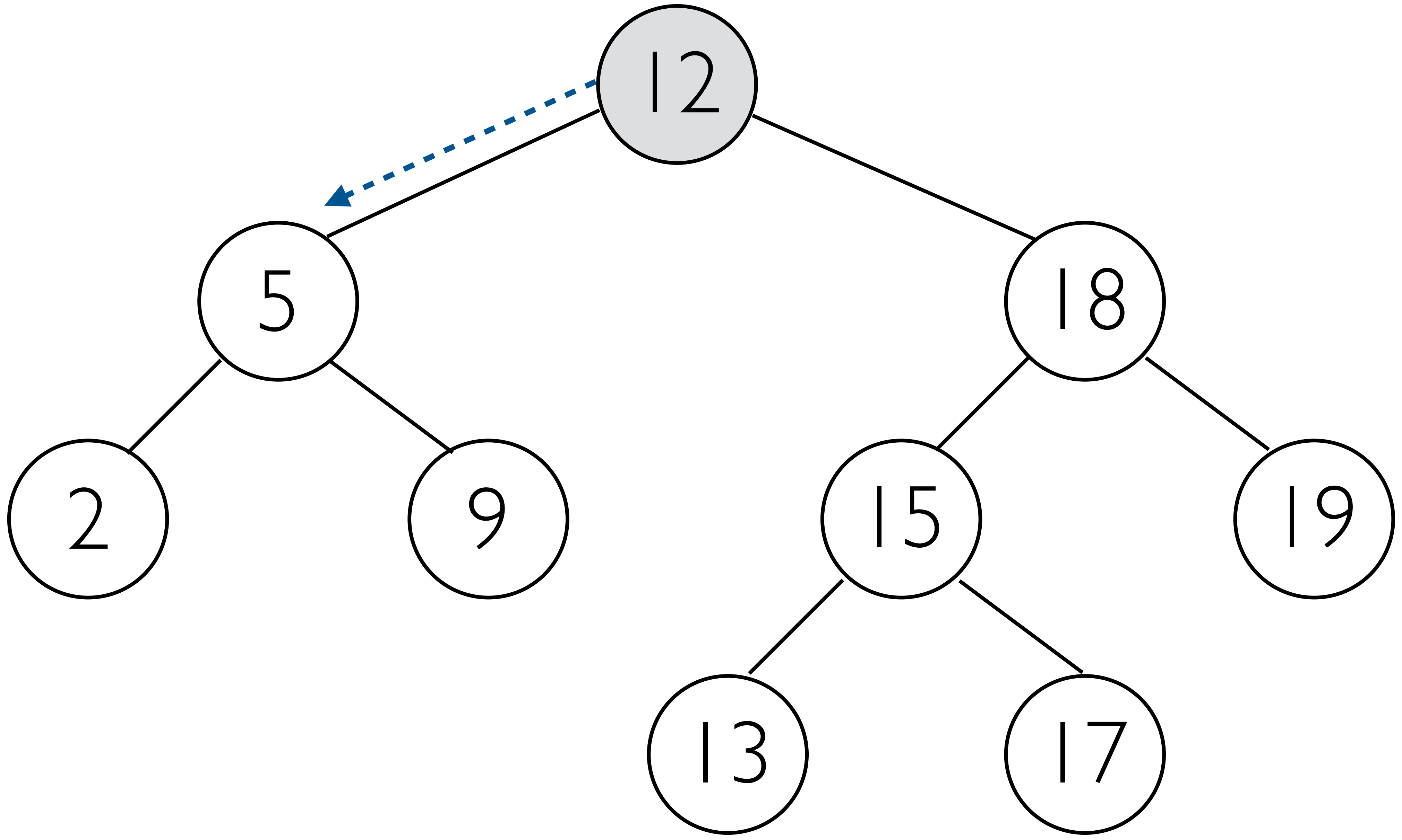
Breadth-first traversal

- Visit the root first
- Visit all current node's immediate children
- Explore all nodes on the current **level**
- Process to next level



Breadth-first traversal

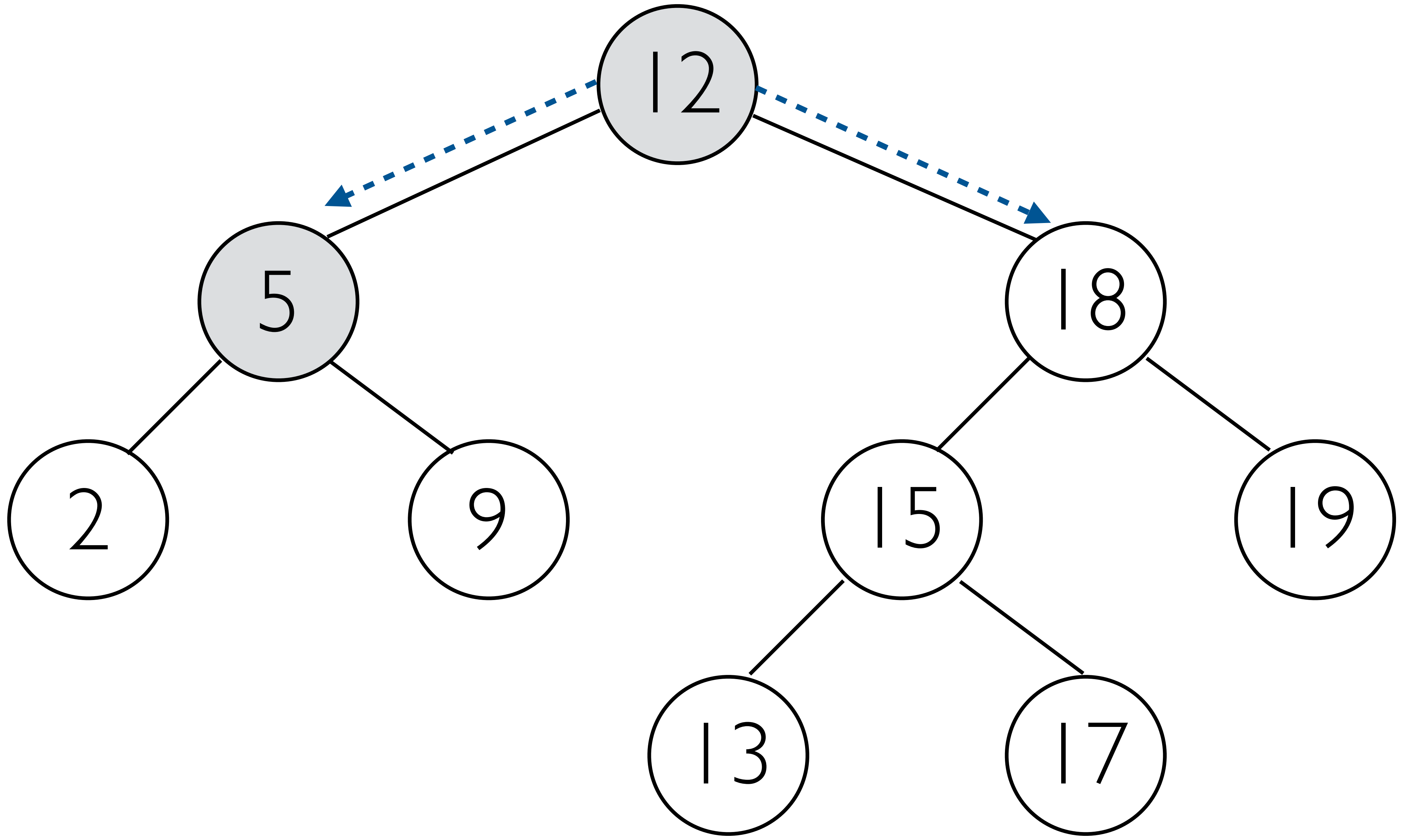
- **Level 1:**
  - 12



Level-order: 12

Breadth-first traversal

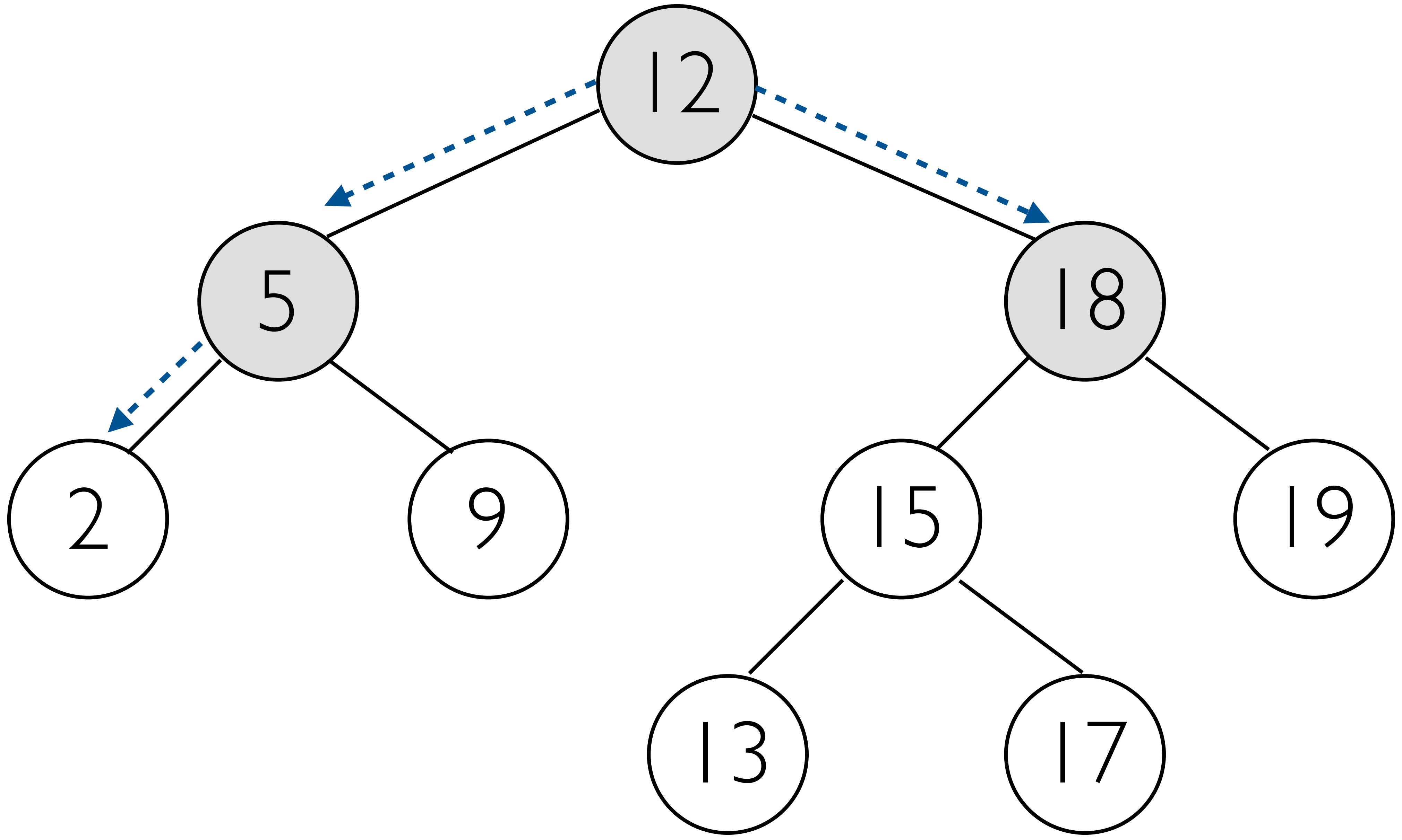
- **Level 1:**
  - 12
- **Level 2:**
  - 5



Level-order: 12→5

Breadth-first traversal

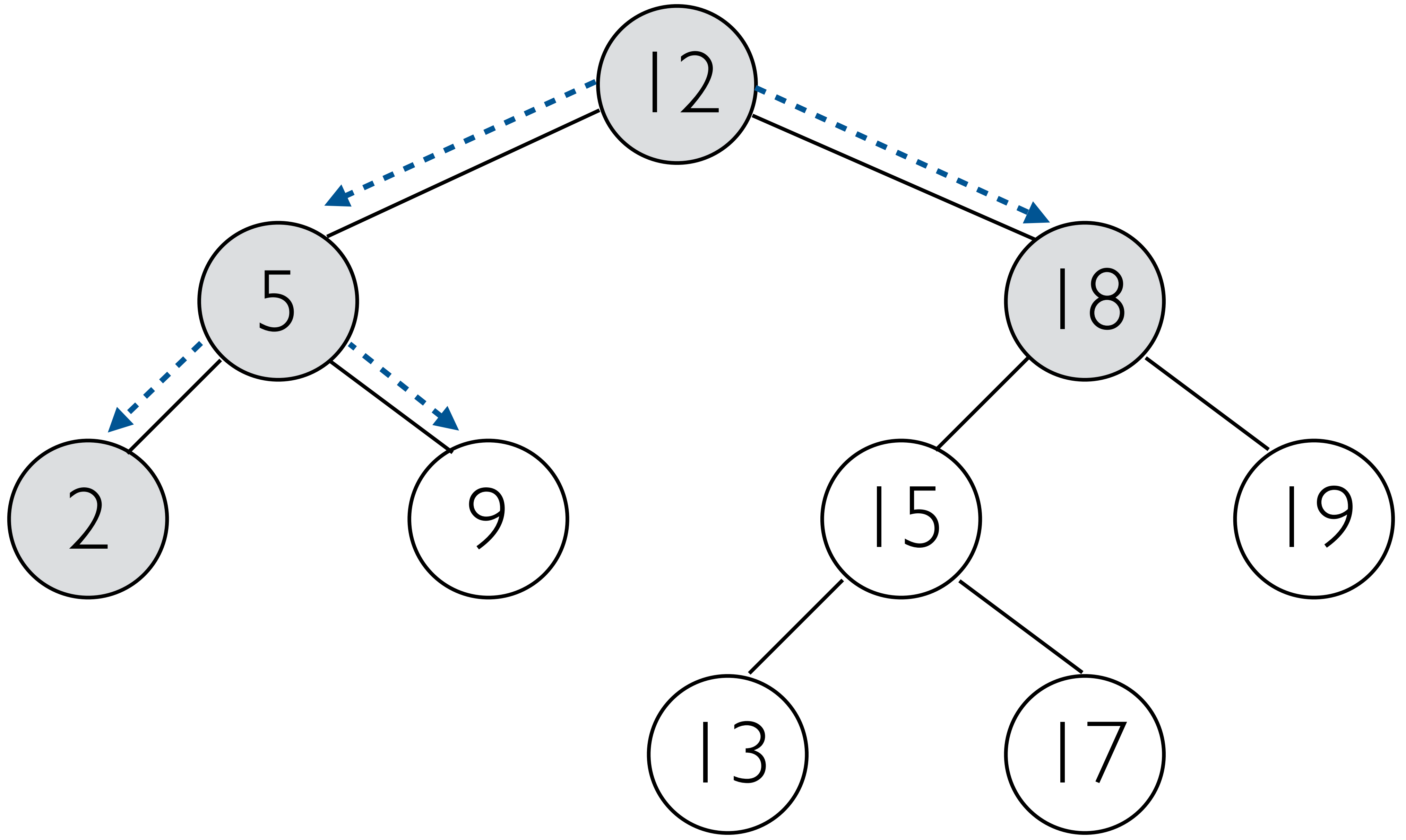
- **Level 1:**
  - 12
- **Level 2:**
  - 5→18



Level-order: 12→5→18

Breadth-first traversal

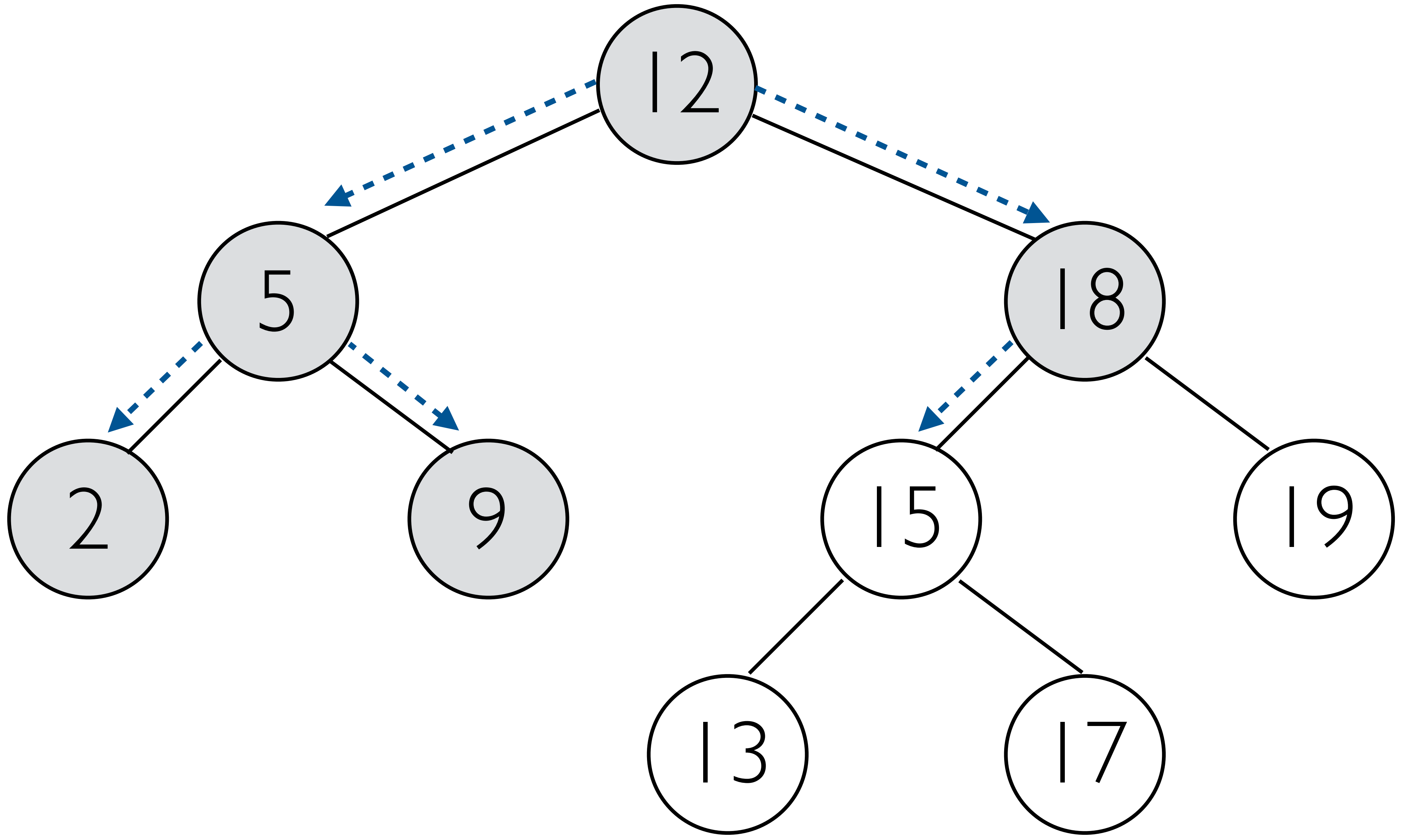
- **Level 1:**
  - 12
- **Level 2:**
  - 5→18
- **Level 3:**
  - 2



Level-order: 12→5→18→2

Breadth-first traversal

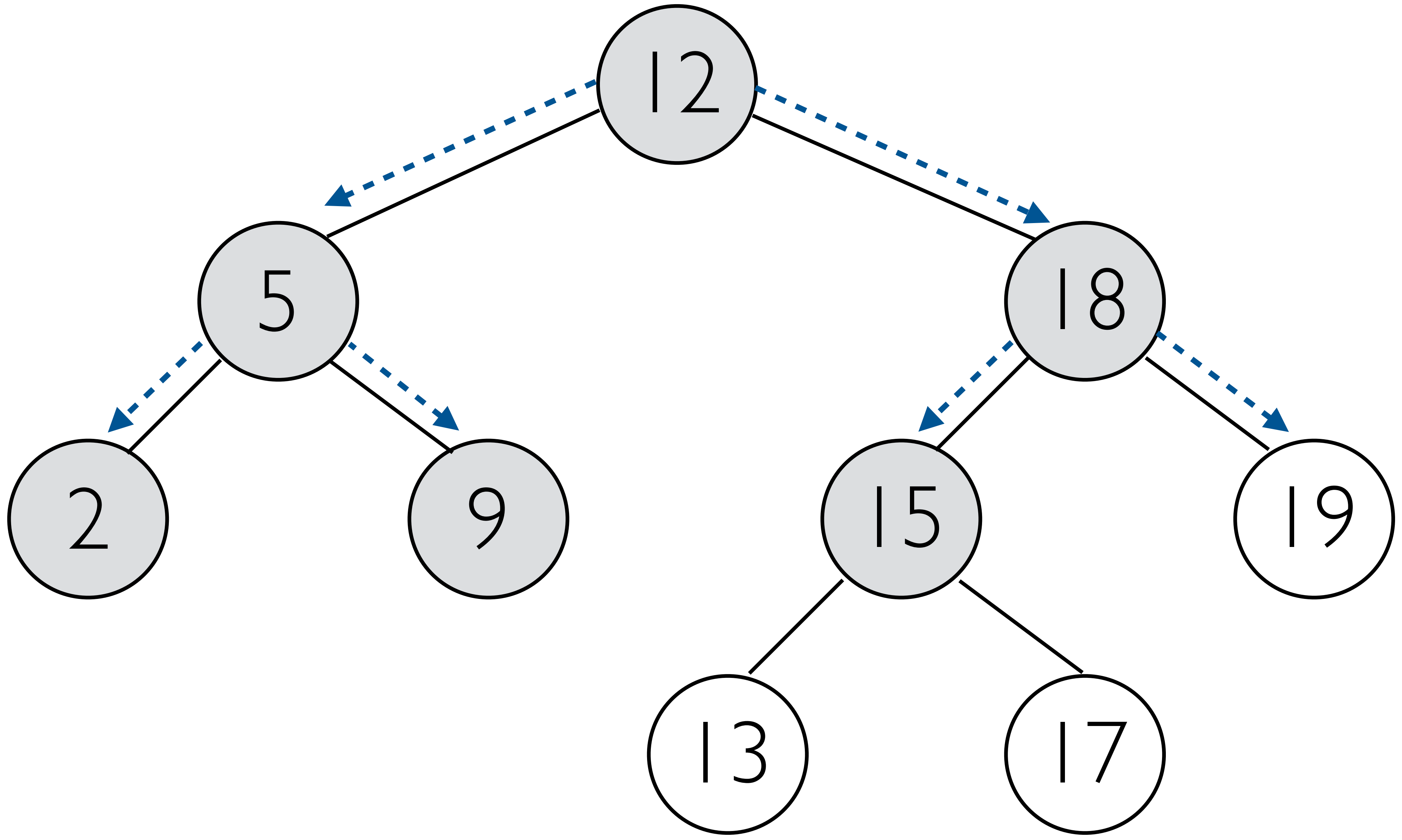
- **Level 1:**
  - 12
- **Level 2:**
  - 5→18
- **Level 3:**
  - 2→9



Level-order: 12→5→18→2→9

Breadth-first traversal

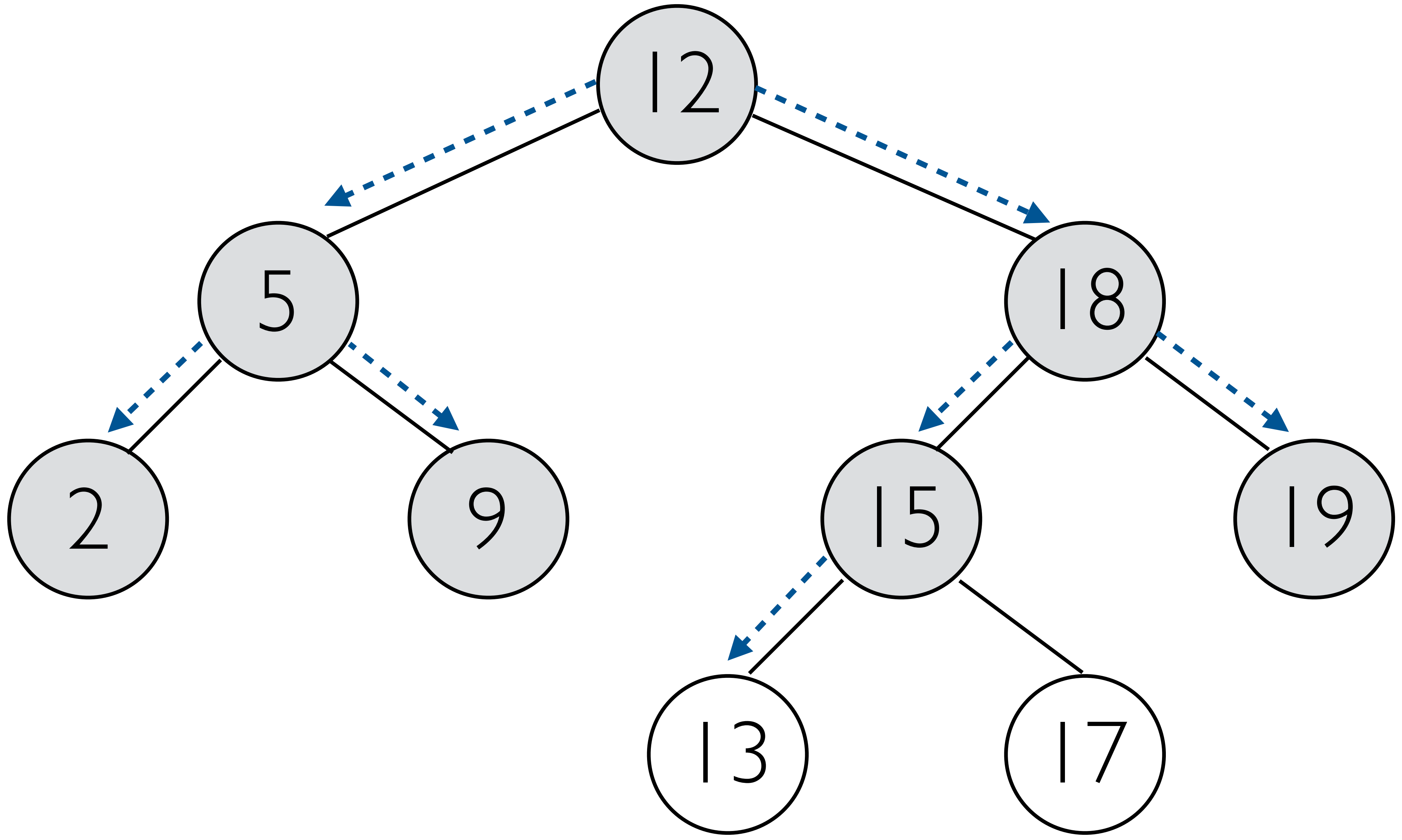
- **Level 1:**
  - 12
- **Level 2:**
  - 5→18
- **Level 3:**
  - 2→9→15



Level-order: 12→5→18→2→9→15

Breadth-first traversal

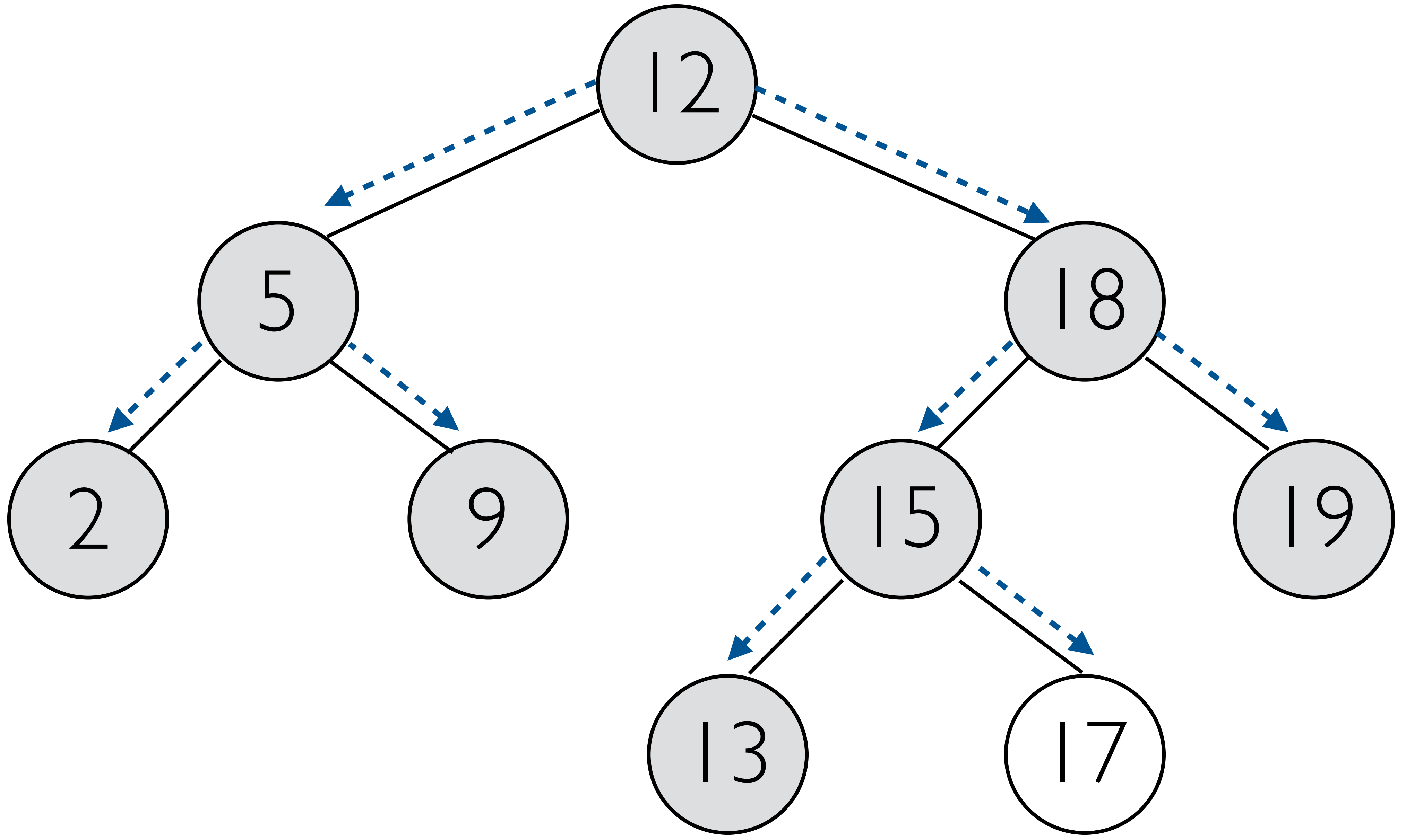
- **Level 1:**
  - 12
- **Level 2:**
  - 5→18
- **Level 3:**
  - 2→9→15→19



Level-order: 12→5→18→2→9→15→19

Breadth-first traversal

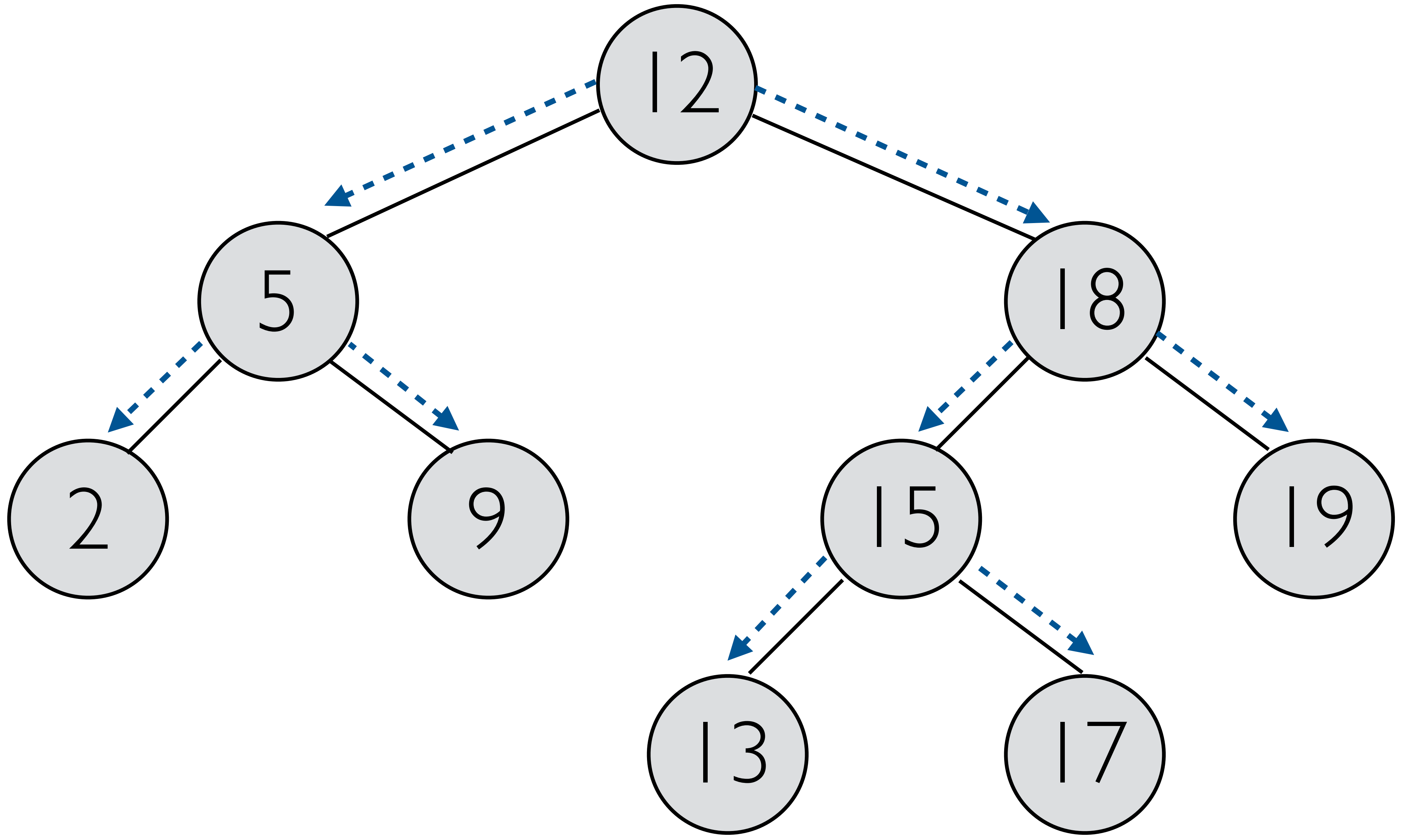
- **Level 1:**
  - 12
- **Level 2:**
  - 5→18
- **Level 3:**
  - 2→9→15→19
- **Level 4:**
  - 13



Level-order: 12→5→18→2→9→15→19→13

Breadth-first traversal

- **Level 1:**
  - 12
- **Level 2:**
  - 5→18
- **Level 3:**
  - 2→9→15→19
- **Level 4:**
  - 13→17



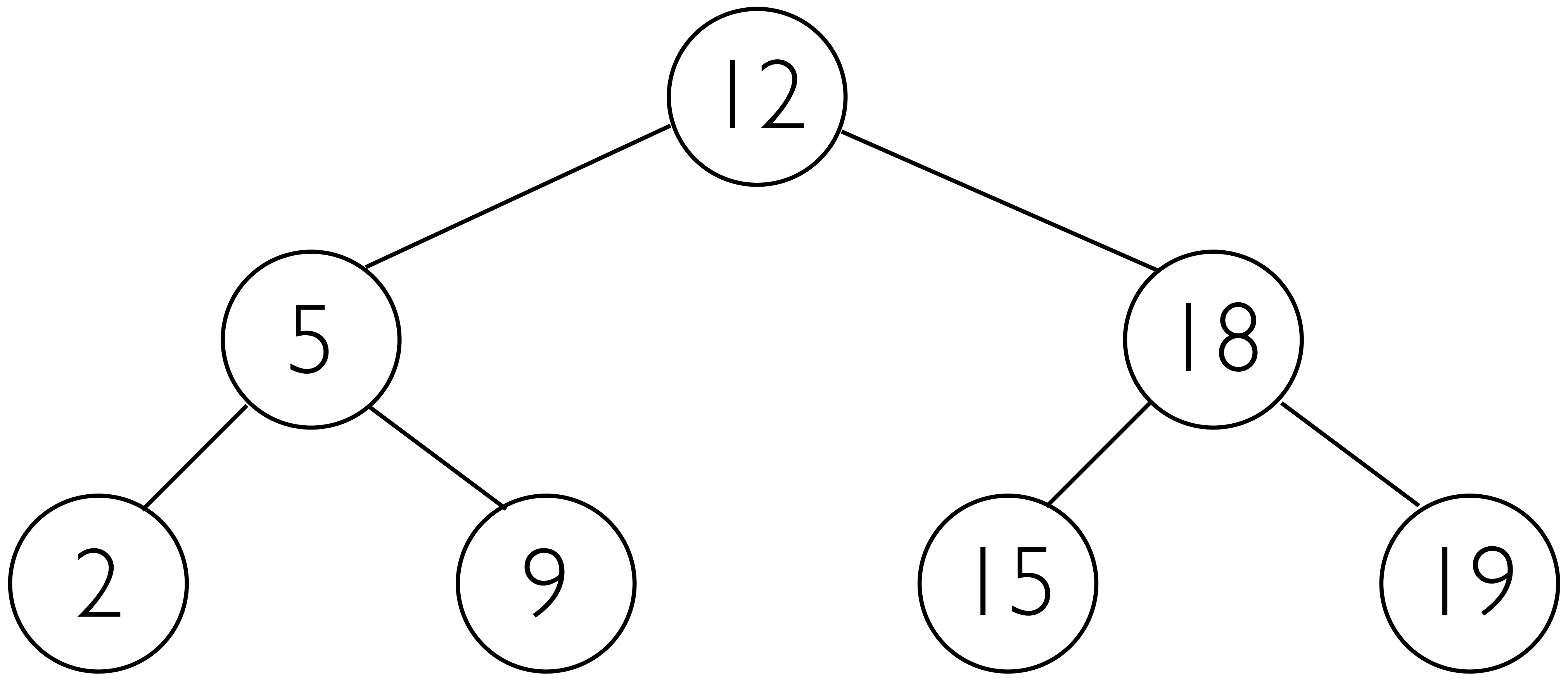
Level-order: 12→5→18→2→9→15→19→13→17

Performance of binary search trees

- Fast random read/write of existing items
- Somewhat slow traversal of items
- Very fast searching for specific items (by key)
- Slow searching for specific items (by value)
- Fast insertion/deletion of new items

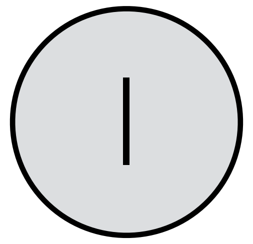
Balanced binary trees

- A tree **T** is **balanced** if:
  - The *difference in heights* of left and right subtrees is less than 1
  - The left subtree of **T** is balanced
  - The right subtree of **T** is balanced
- Keeps height small



Unbalanced BST

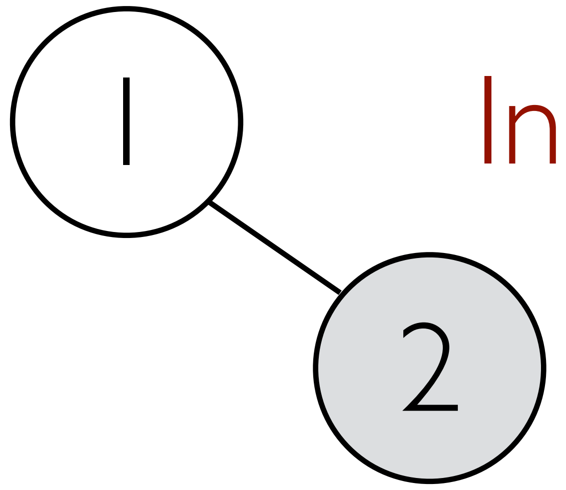
Insert 1



- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***

Unbalanced BST

- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***

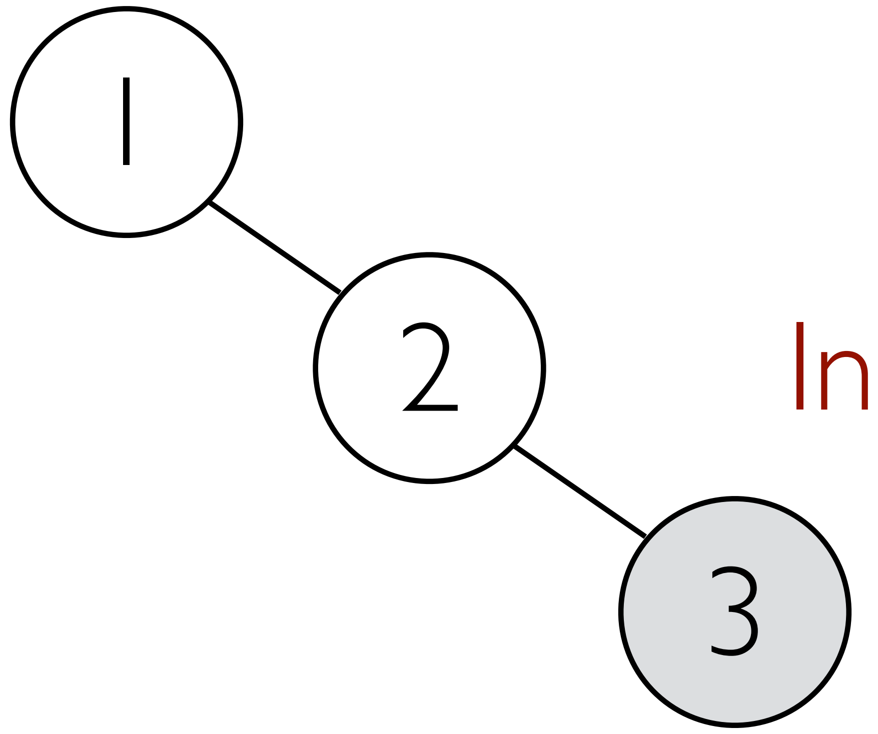


Insert 2

If you insert a sorted array into a BST…

Unbalanced BST

- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***

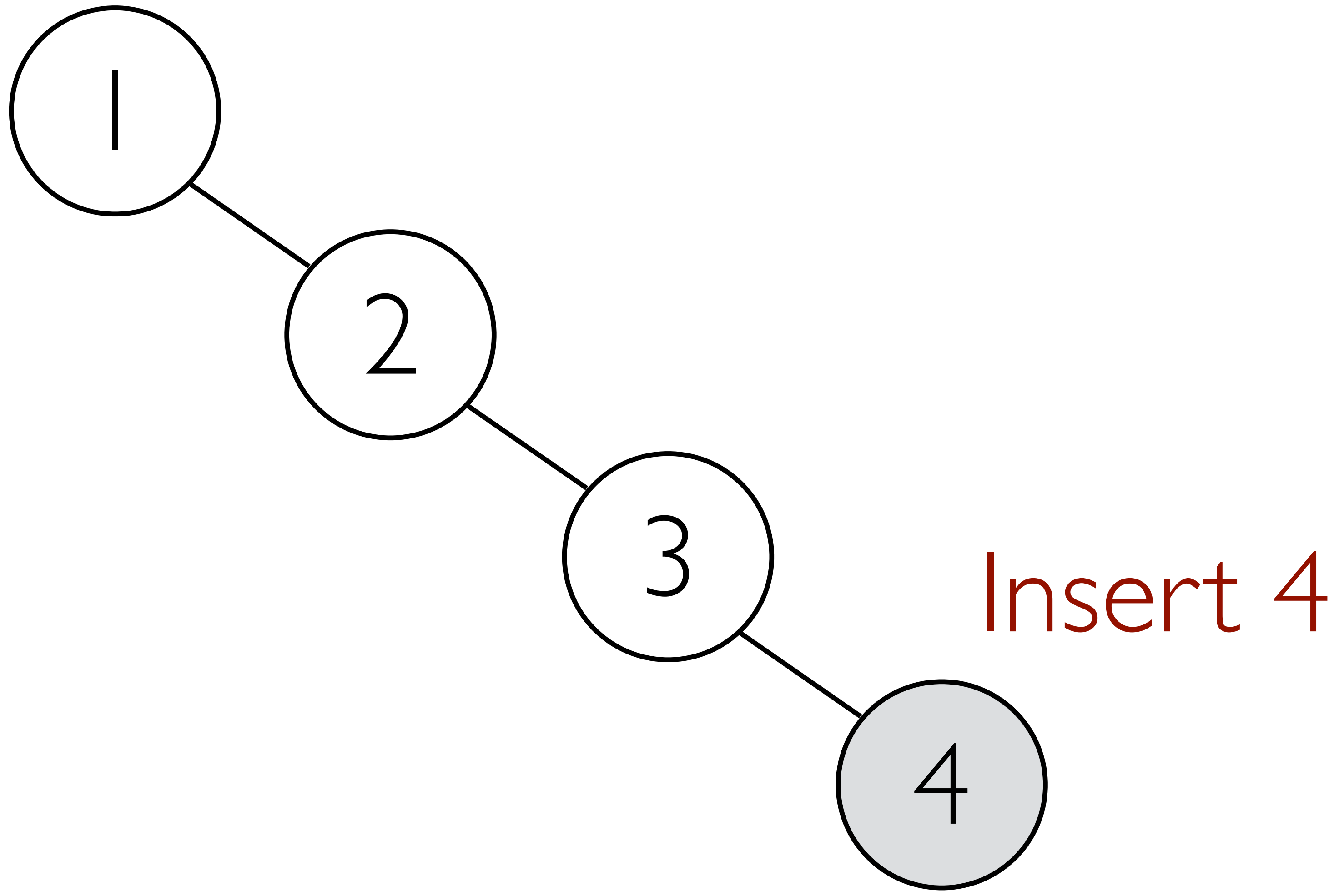


Insert 3

…the tree will be very unbalanced.

Unbalanced BST

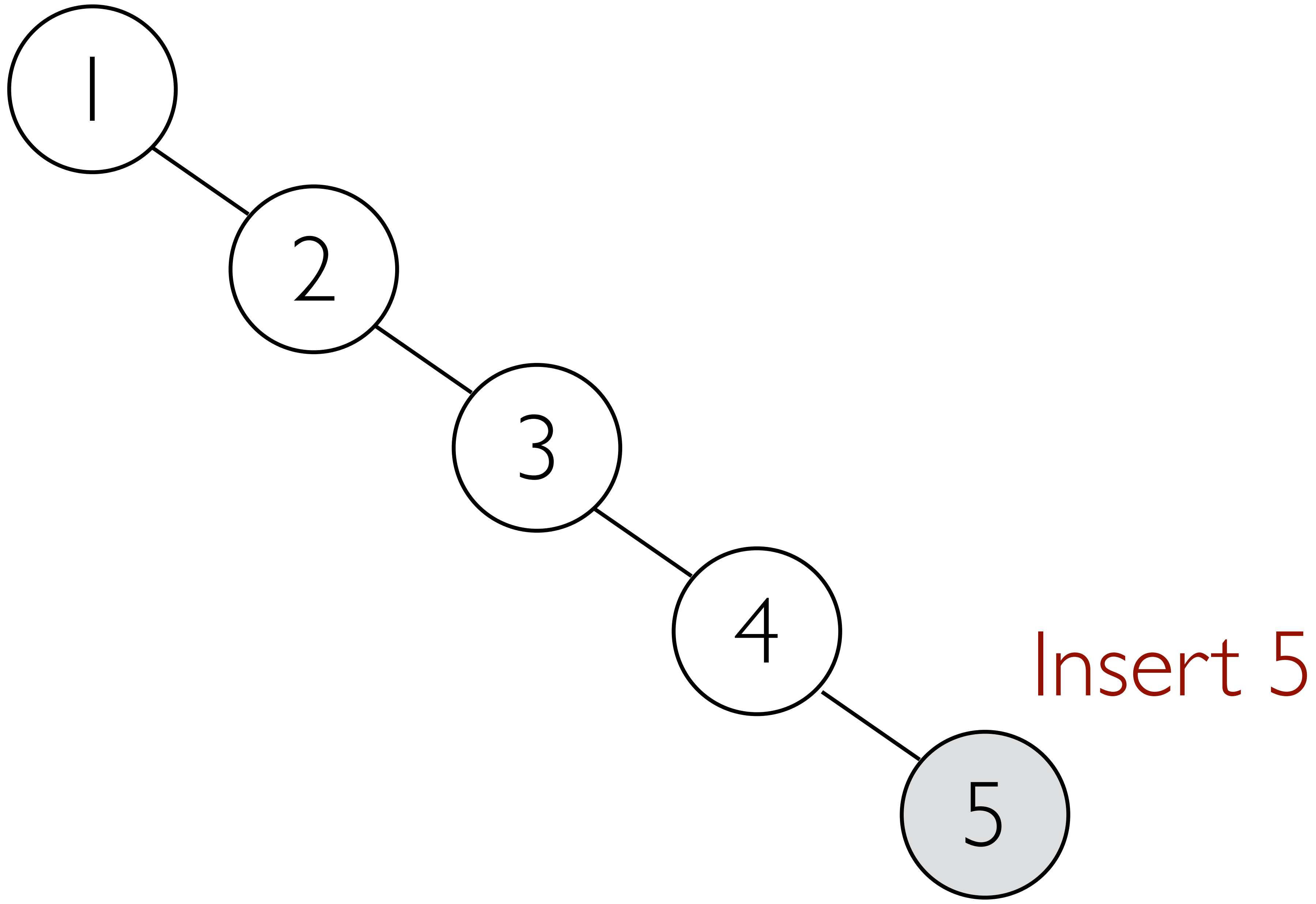
- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***



The problem gets worse and worse…

Unbalanced BST

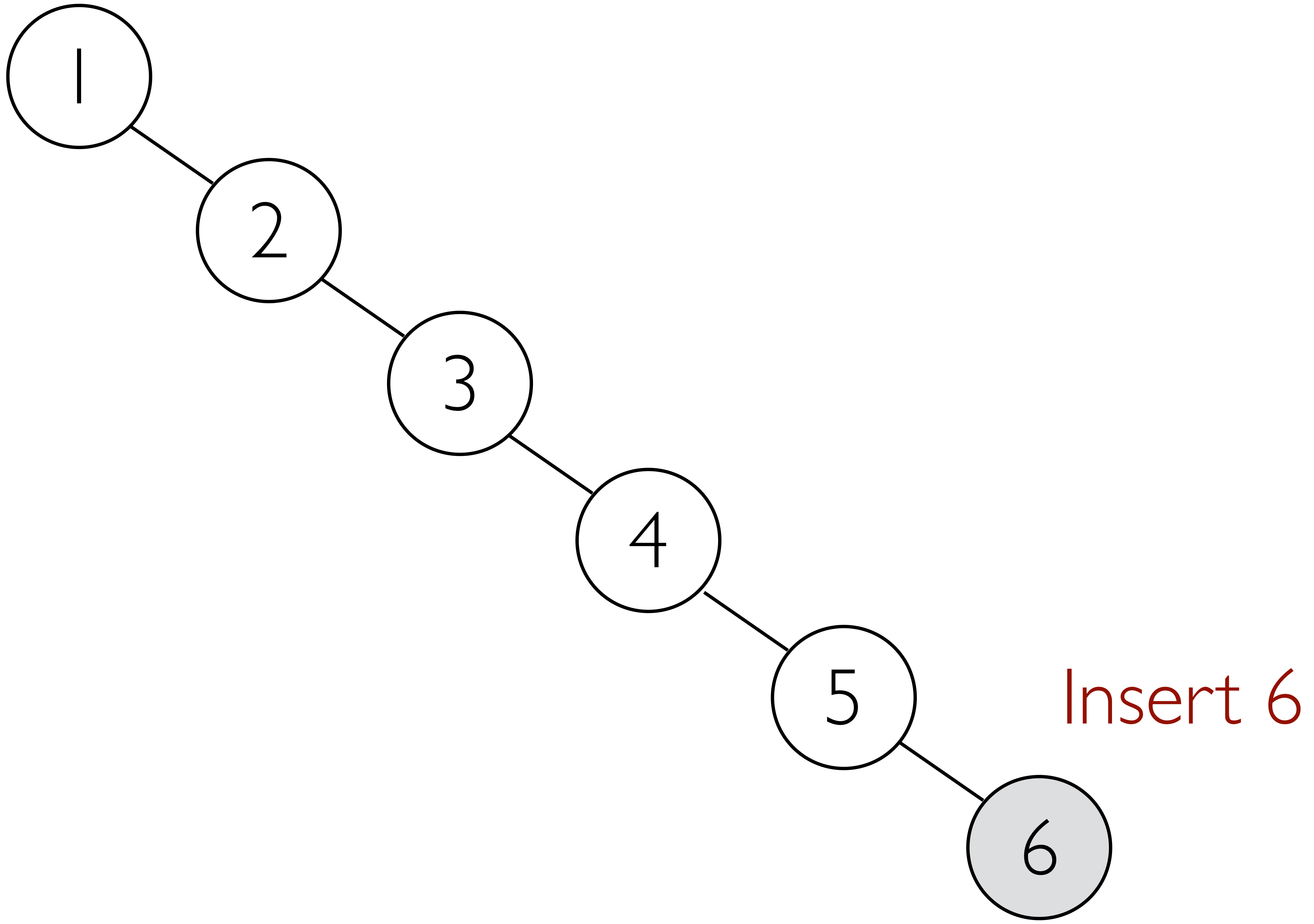
- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***



…until the BST devolves into a linked list.

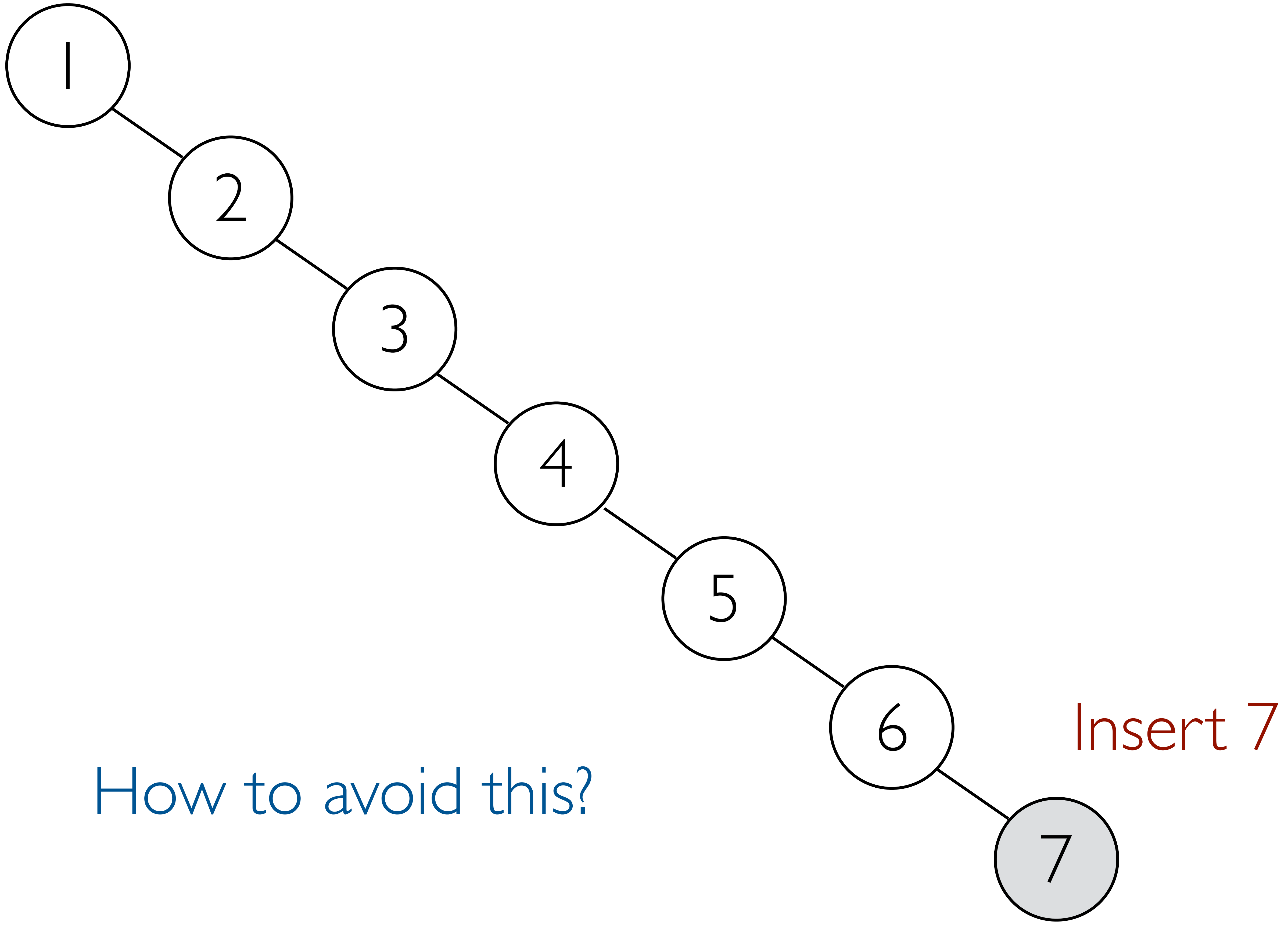
Unbalanced BST

- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***



Unbalanced BST

- BST performs best when **balanced**
- But tree structure depends on order of ***insertion*** and ***deletion***



How to avoid this?

Advanced trees

- Randomly built binary search trees
  - Randomized insertion order keeps height small on average
  - Difficult to guarantee unless building whole tree at once
- Self-balancing trees
  - Follow insertion/deletion rules that keep trees balanced
  - *Red-black tree* (self-balancing BST where nodes have "color")
  - *B-tree* (generalization of BST with more than two children)