Trees

Binary Trees

- A binary tree is composed of zero or more nodes in which no node can have more than two children.

- Each node contains:
  ✓ A value (some sort of data item).
  ✓ A reference or pointer to a left child (may be null), and
  ✓ A reference or pointer to a right child (may be null)

- A binary tree may be empty (contain no nodes).

- If not empty, a binary tree has a root node.
  ✓ Every node in the binary tree is reachable from the root node by a unique path.

- A node with neither a left child nor a right child is called a leaf.

TreeNode Class

- Every node has a value, a pointer to the left subtree and a pointer to the right subtree.
- When a node is first created, the left and right pointers are set to None.

```
"""Tree Node Class"""


class TreeNode:
    """Initializes data members""“
    left, right, data = None, None, 0

    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
```

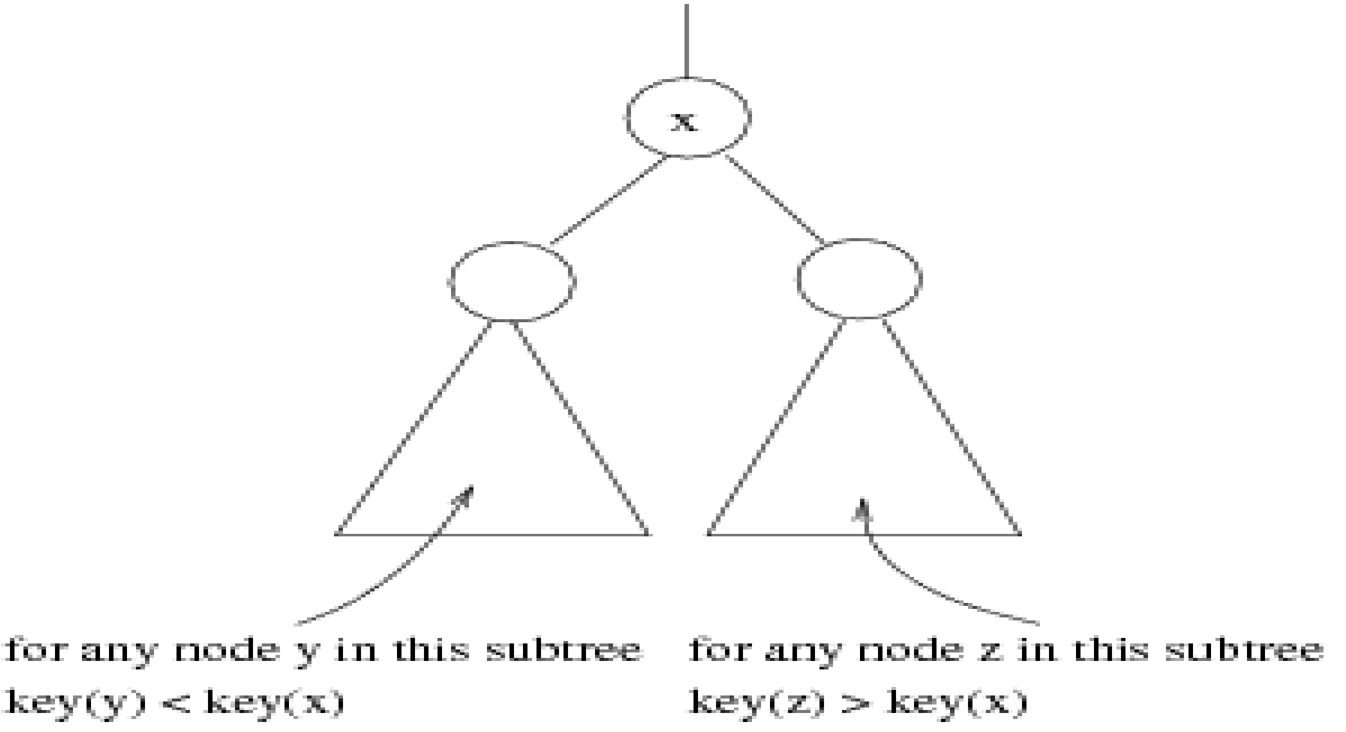
Tree Class

- This class represents the entire tree.
- Contains methods to create and manipulate the nodes, their data and links between them.

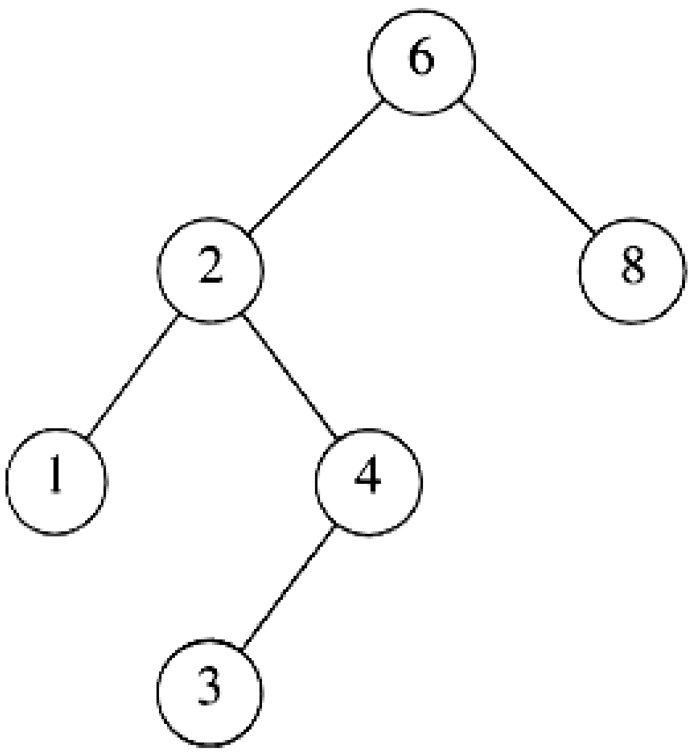
```
"""Tree Class"""


class Tree:
    """initializes the root member"""
    def __init__(self):
        self.root = None
```

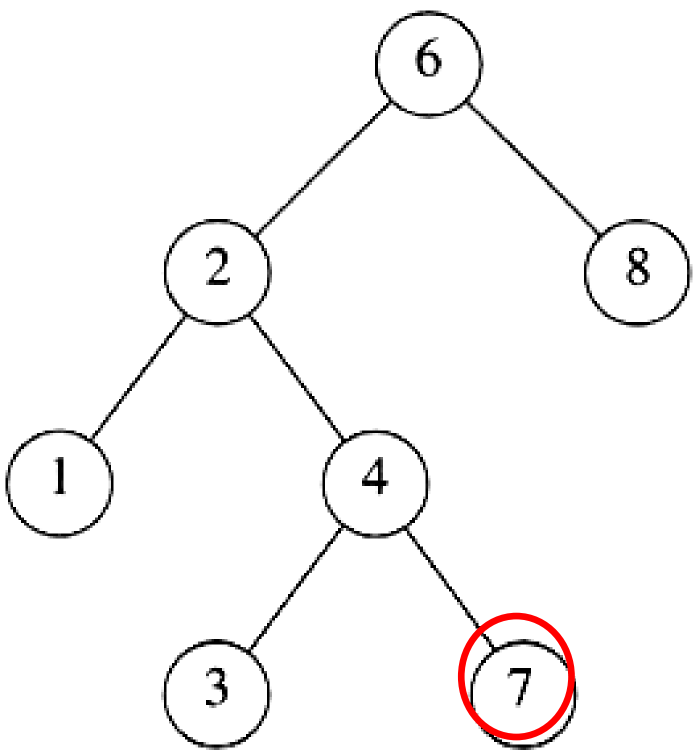
Binary Search Trees

- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.
- Binary search tree property
  - ✓For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X



Binary Search Trees





A binary search tree

Not a binary search tree

Binary Search Trees

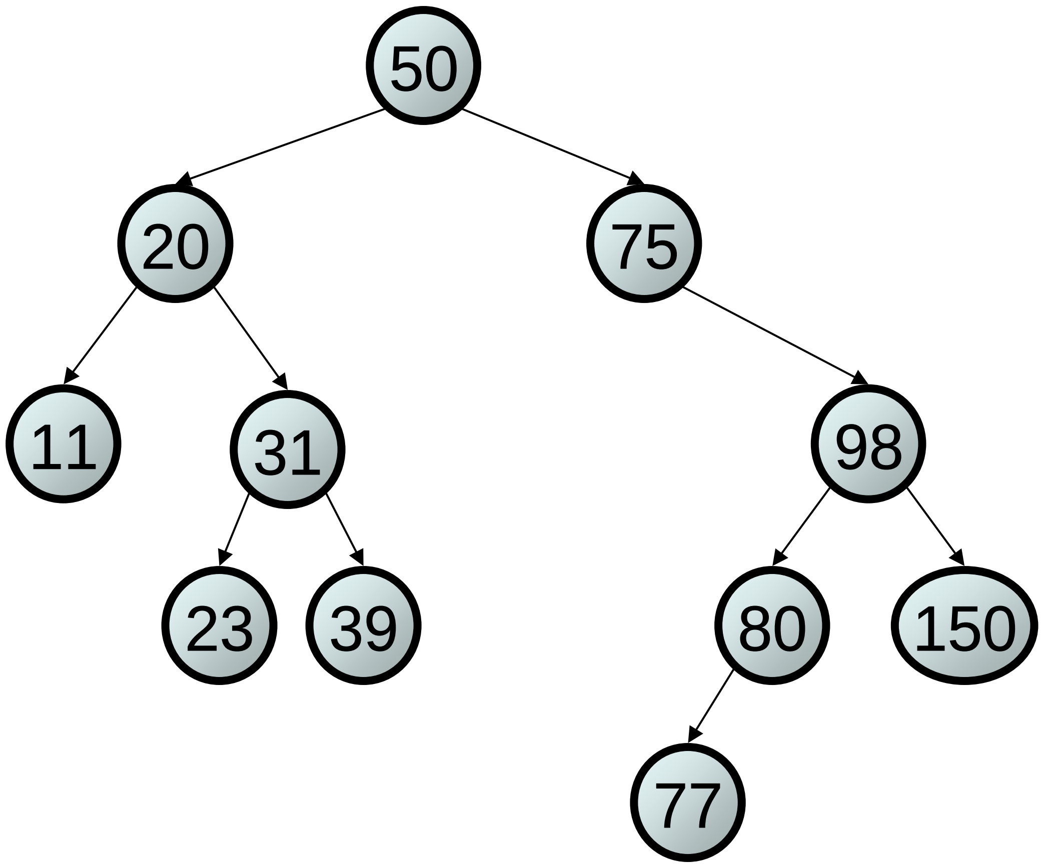
- Where is the smallest element in a binary search tree??

Ans: leftmost element

- Where is the largest element in a binary search tree??

Ans: rightmost element

BST: Insert item



- Insert the following items to the binary search tree.
  50
  20
  75
  98
  80
  31
  150
  39
  23
  11
  77

BST: Insert item

- What is the size of the problem?

Ans. Number of nodes in the tree we are examining

- What is the base case(s)?

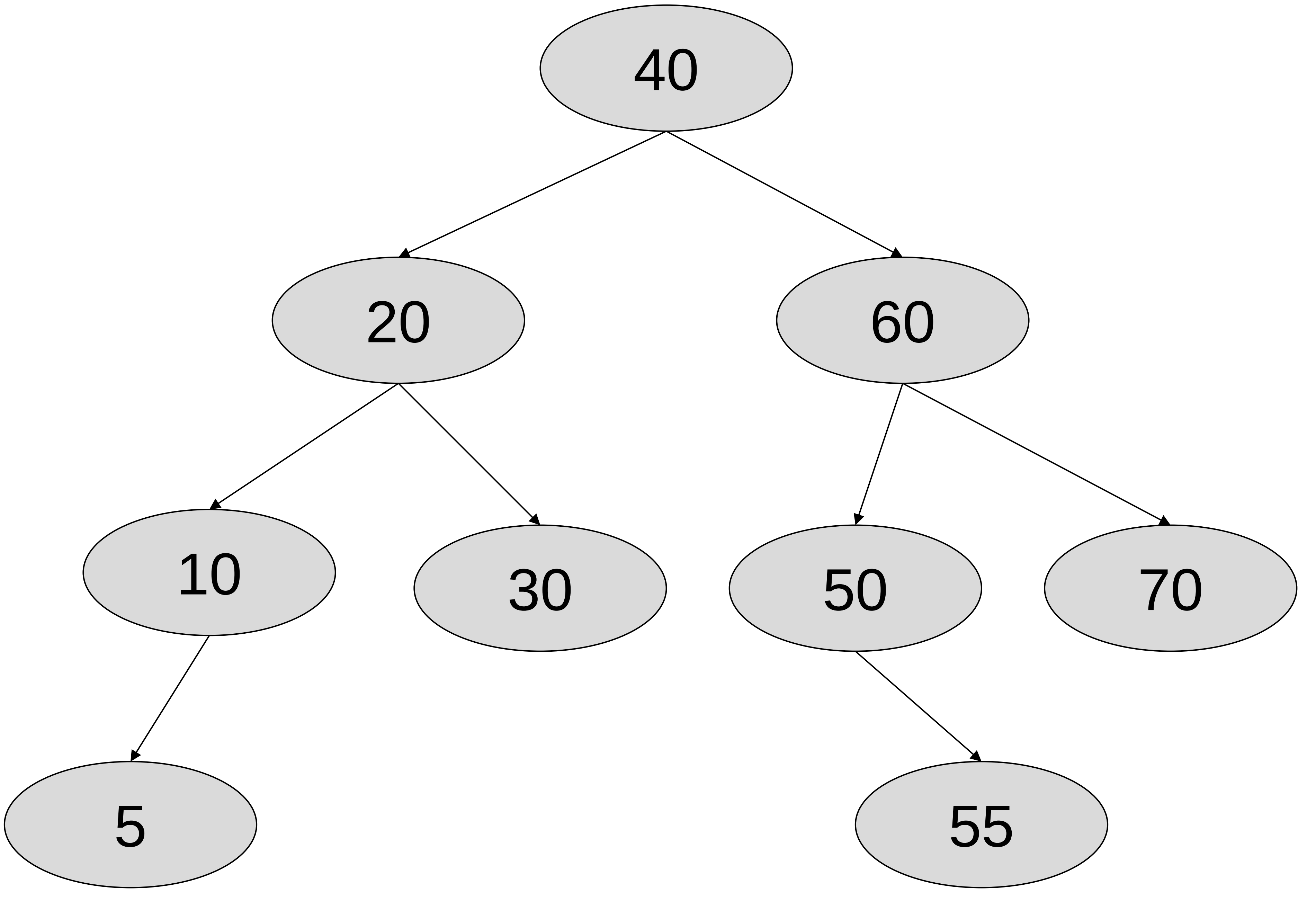
Ans. The tree is empty

- What is the general case?

Ans. Choose the left or right subtree

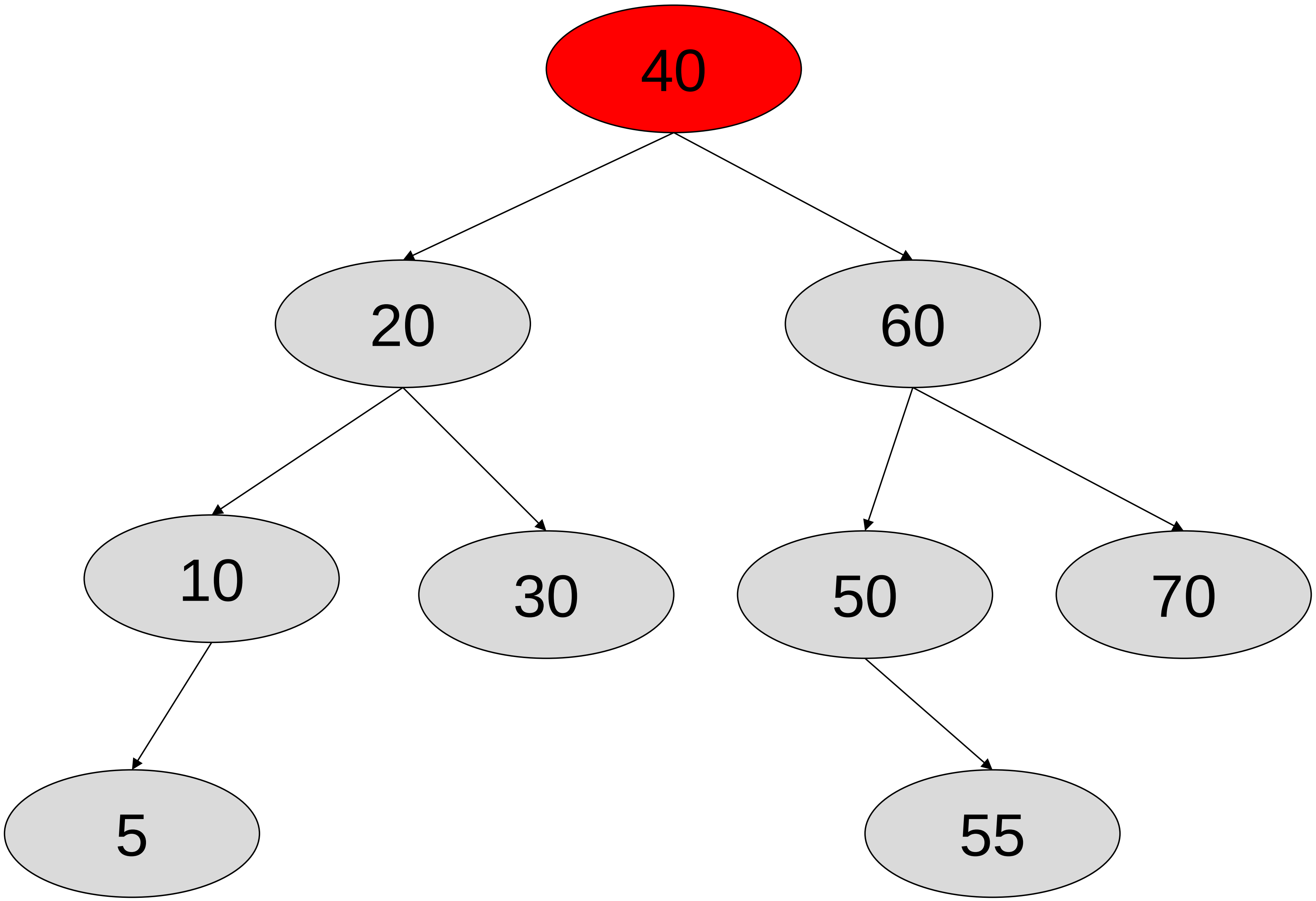
BST: Lookup

- Search for the element 55 in the below binary search tree.



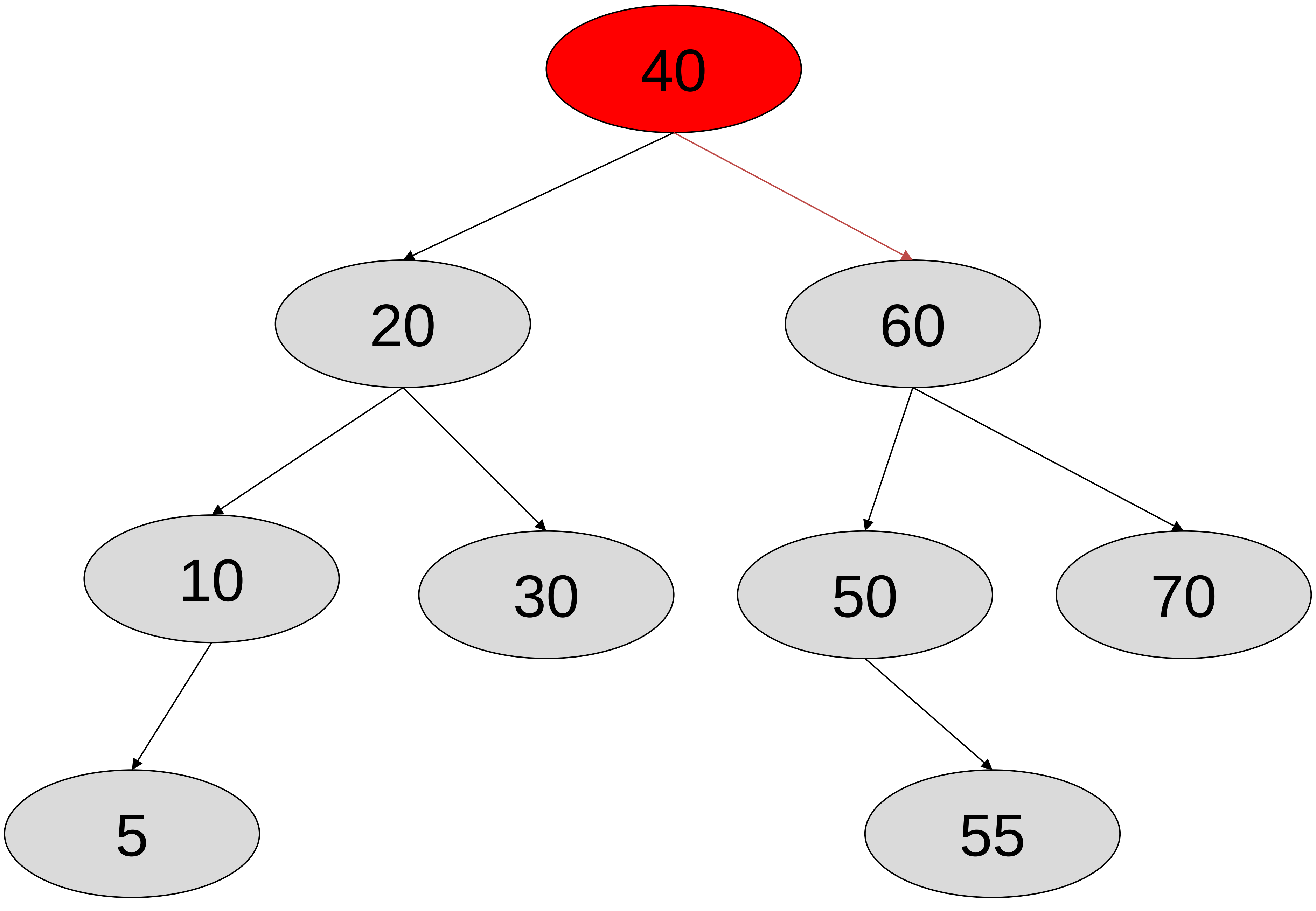
BST: Lookup

- Search for the element 55 in the below binary search tree.



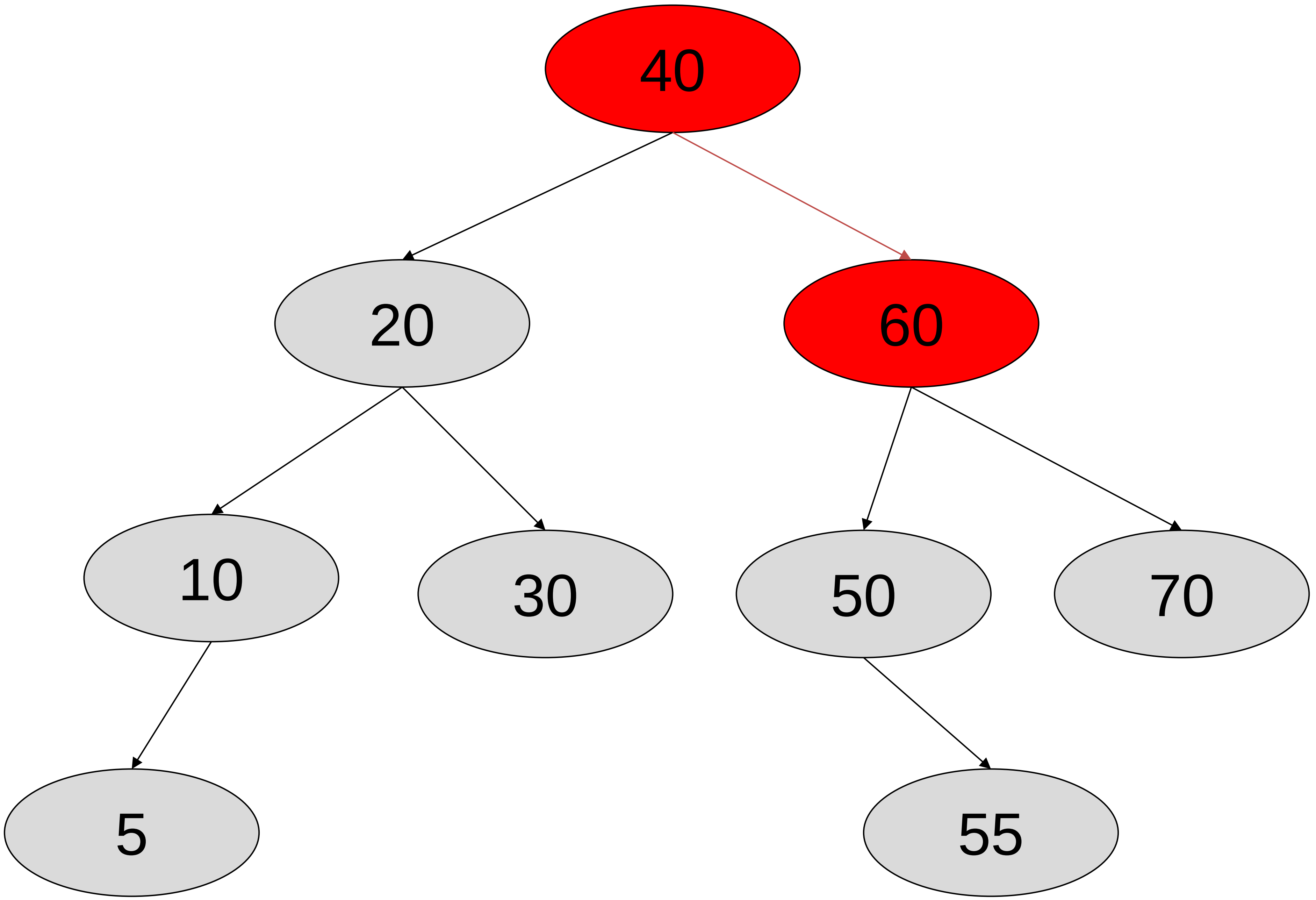
BST: Lookup

- Search for the element 55 in the below binary search tree.



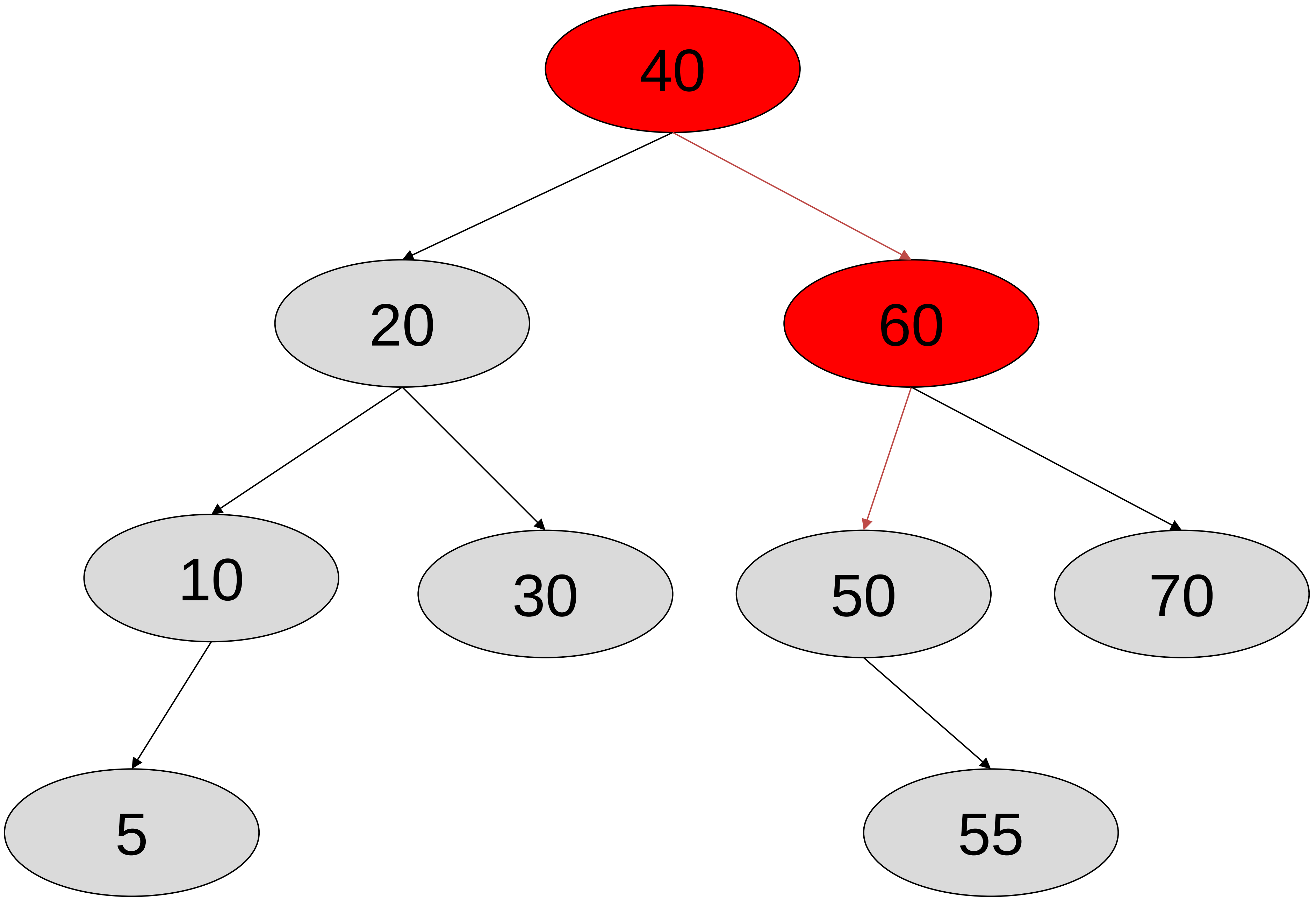
BST: Lookup

- Search for the element 55 in the below binary search tree.



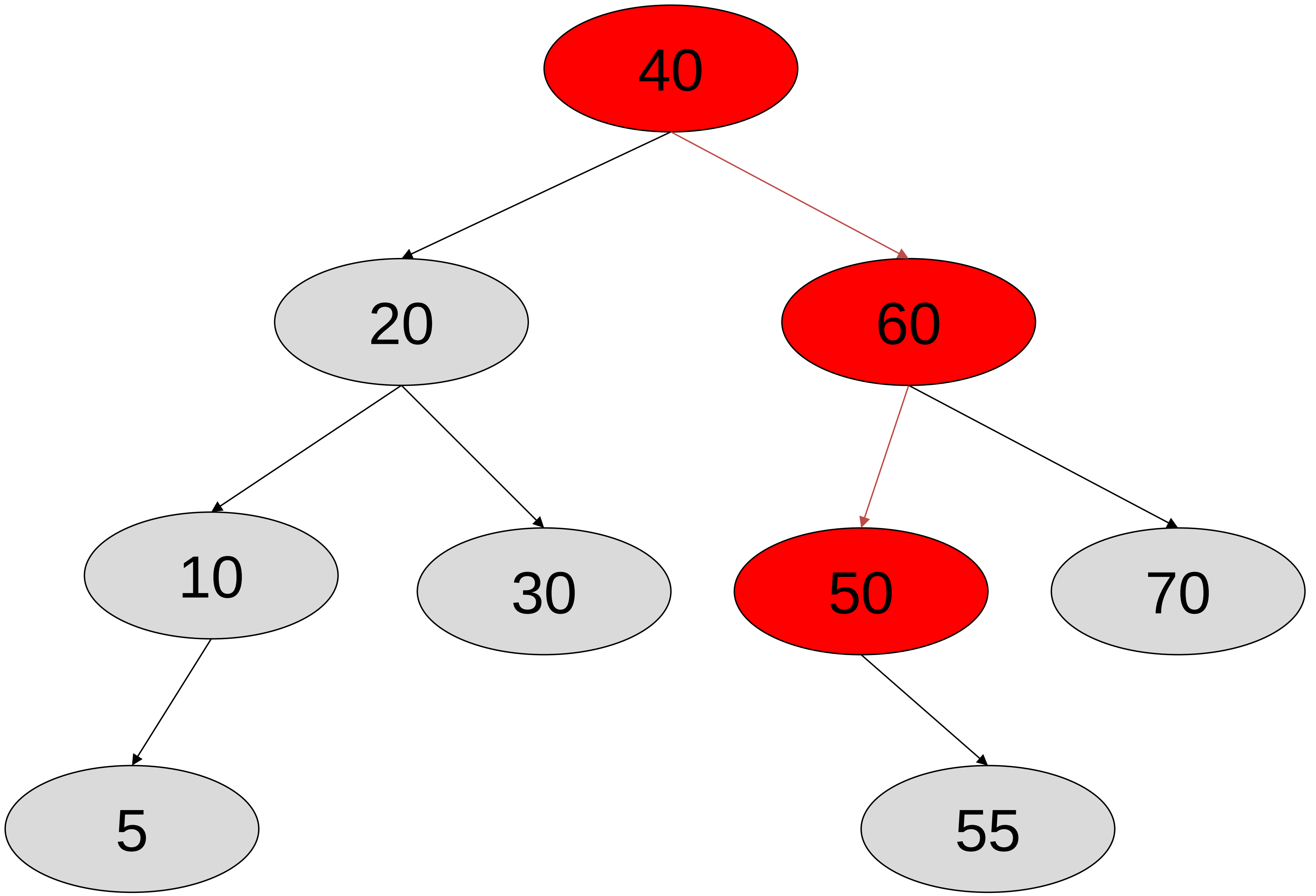
BST: Lookup

- Search for the element 55 in the below binary search tree.



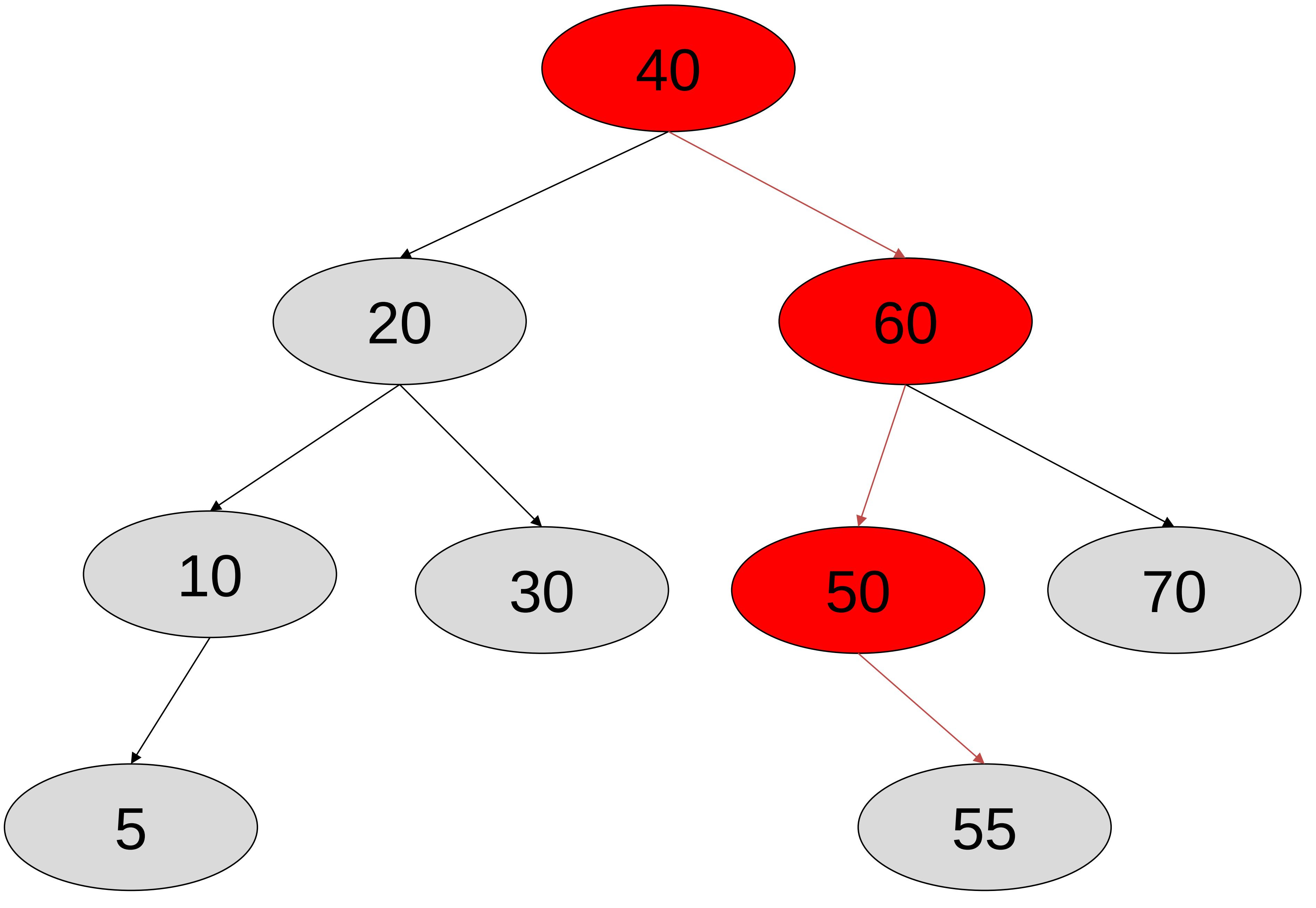
BST: Lookup

- Search for the element 55 in the below binary search tree.



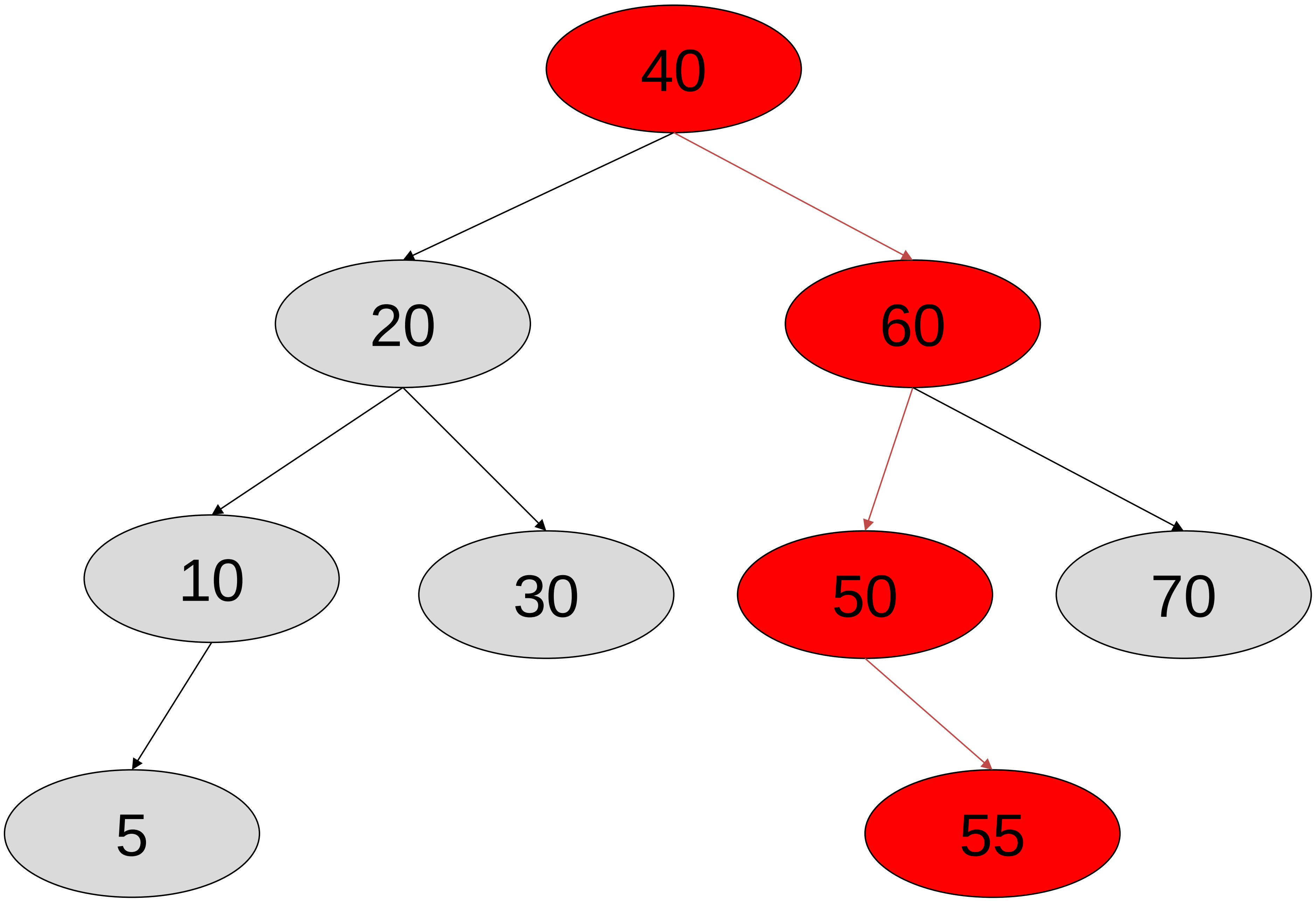
BST: Lookup

- Search for the element 55 in the below binary search tree.



BST: Lookup

- Search for the element 55 in the below binary search tree.



BST: Lookup

- What is the size of the problem?

Ans. Number of nodes in the tree we are examining

- What is the base case(s)?

Ans. 1. When the key is found.

2. The tree is empty (key was not found).

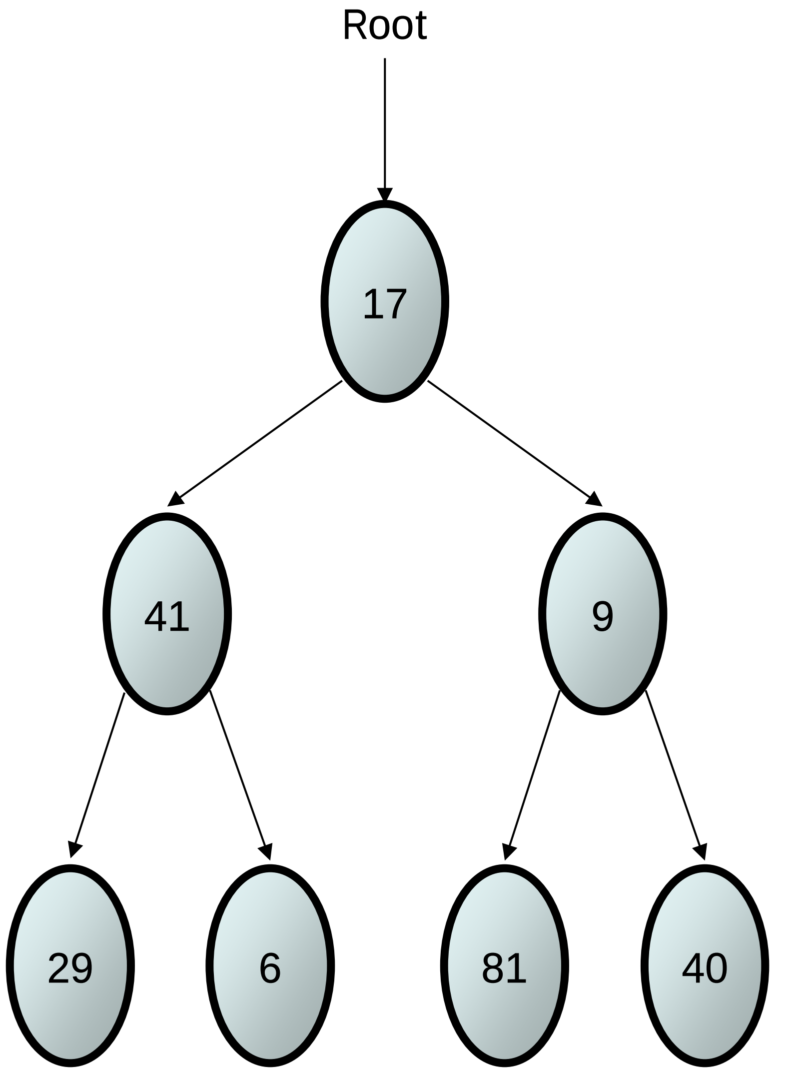
- What is the general case?

Ans. Search in the left or right subtrees.

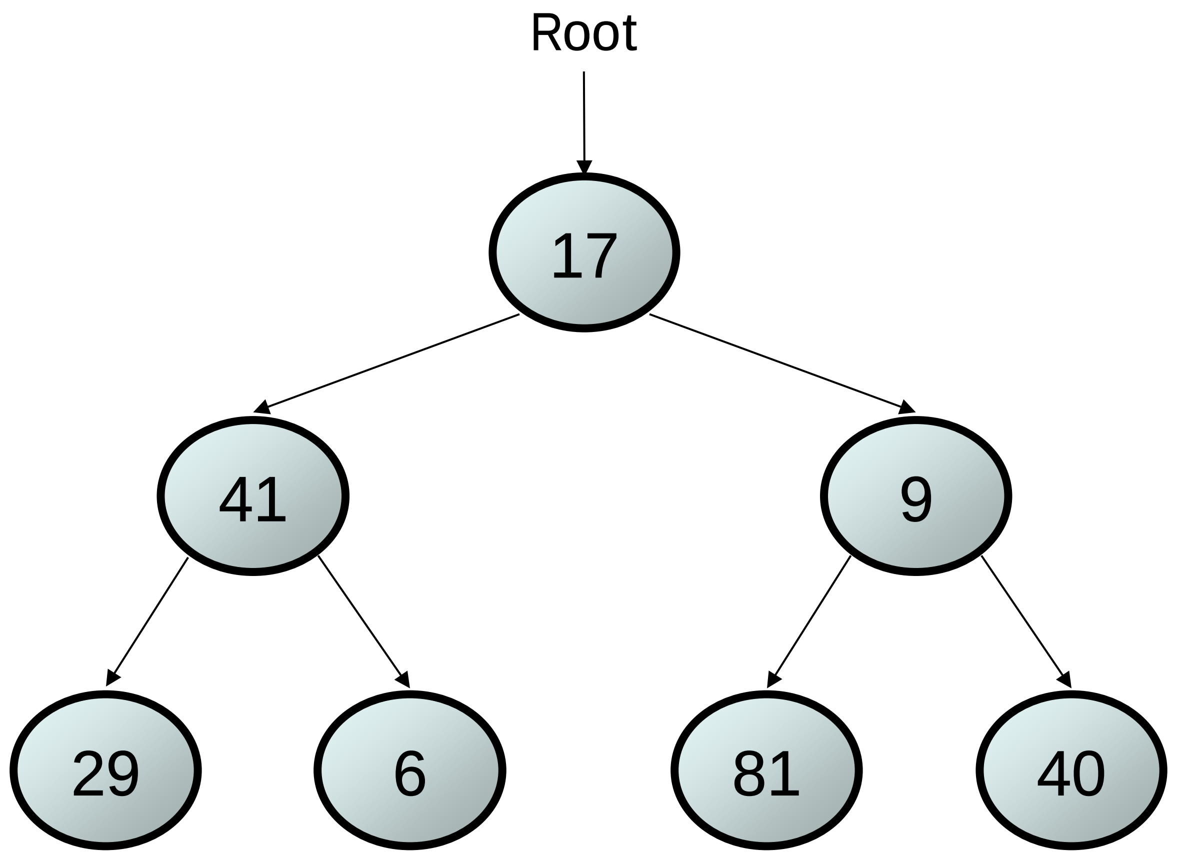
Traversals

- **traversal**: An examination of the elements of a tree.
  - A pattern used in many tree algorithms and methods

- Common orderings for traversals:
  - **pre-order**: process root node, then its left/right subtrees
  - **in-order**: process left subtree, then root node, then right
  - **post-order**: process left/right subtrees, then root node



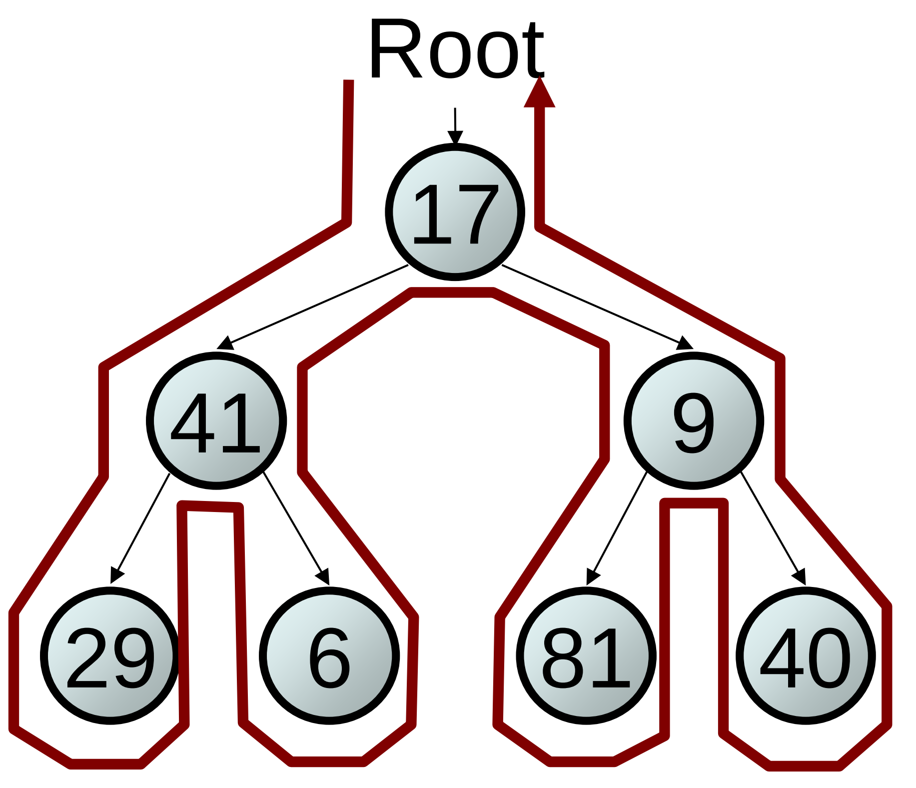
Traversal example



- pre-order: 17 41 29 6 9 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

Traversal trick

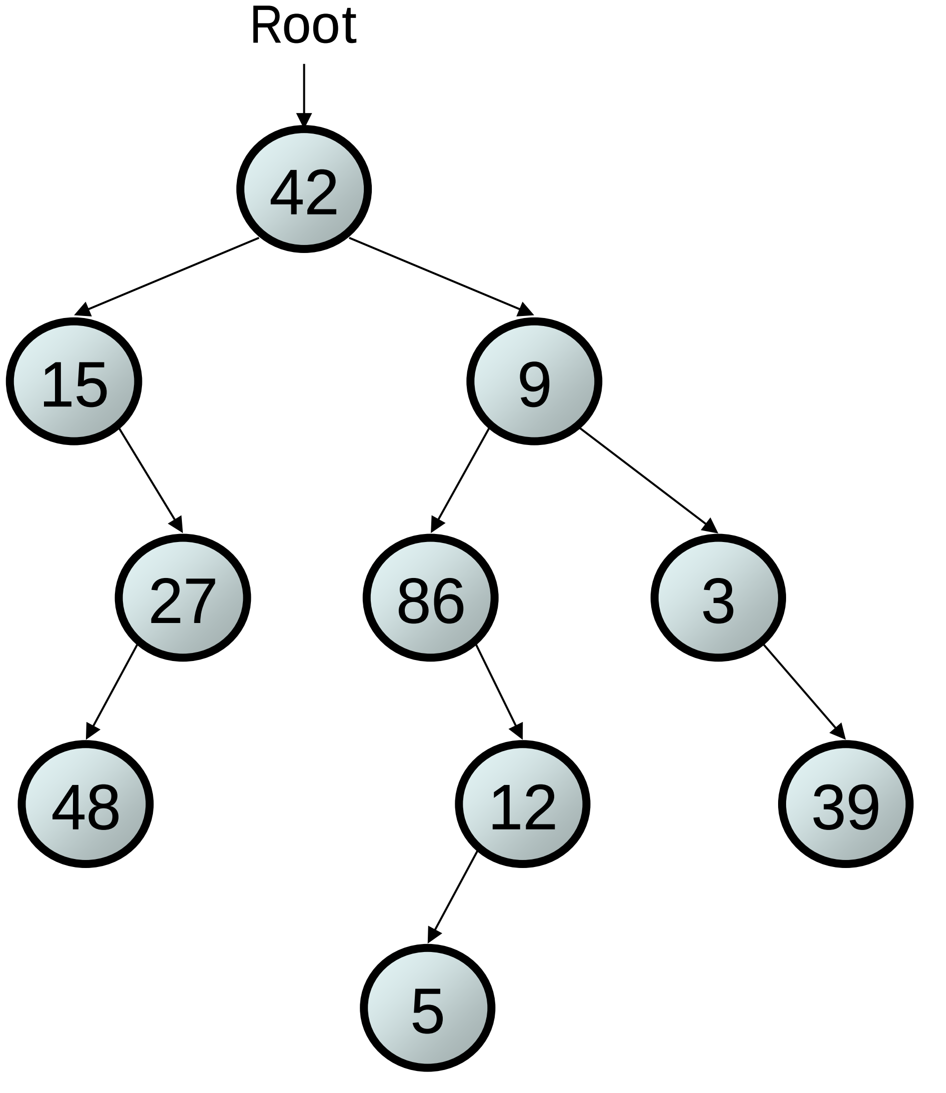
- To quickly generate a traversal:
  - Trace a path around the tree.
  - As you pass a node on the proper side, process it.
    - pre-order: left side
    - in-order: bottom
    - post-order: right side



- pre-order: 17 41 29 6 81 40
- in-order: 29 41 6 17 81 9 40
- post-order: 29 6 41 81 40 9 17

Exercise

- Give pre-, in-, and post-order traversals for the following tree:



  – pre: 42 15 27 48 9 86 12 5 3 39
  – in: 15 48 27 42 86 5 12 9 3 39
  – post: 48 27 15 5 12 86 39 3 42

Huffman Coding

Huffman Coding

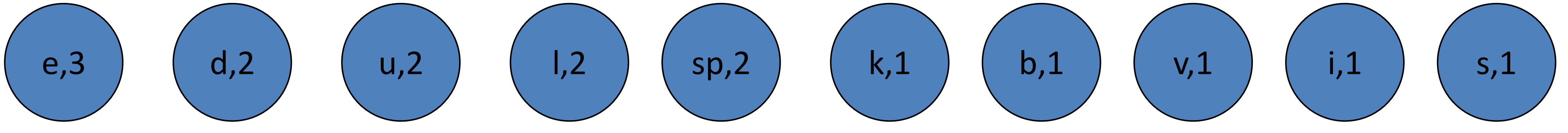
- Huffman codes can be used to compress information
  - Like WinZip – although WinZip doesn't use the Huffman algorithm
  - JPEGs do use Huffman as part of their compression process

- The basic idea is that instead of storing each character in a file as an 8-bit ASCII value, we will instead store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the filesize (usually ½)

Huffman Coding

- As an example, lets take the string:

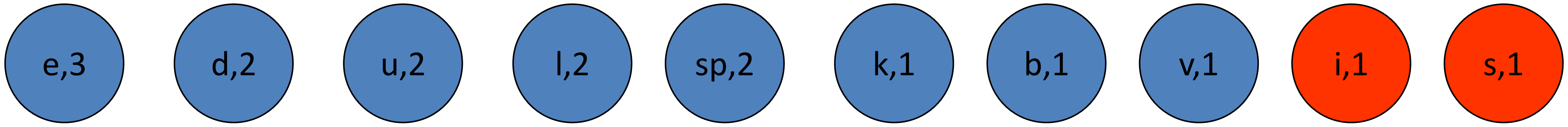
  "duke blue devils"
- We first to a frequency count of the characters:
  - e:3, d:2, u:2, l:2, space:2, k:1, b:1, v:1, i:1, s:1
- Next we use a Greedy algorithm to build up a Huffman Tree
  – We start with nodes for each character



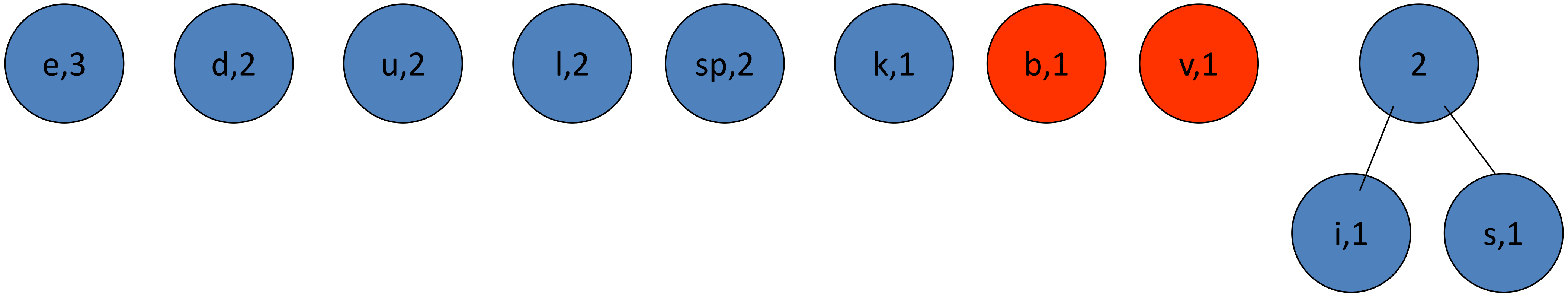
Huffman Coding

- We then pick the nodes with the smallest frequency and combine them together to form a new node
  - The selection of these nodes is the Greedy part
- The two selected nodes are removed from the set, but replace by the combined node
- This continues until we have only 1 node left in the set

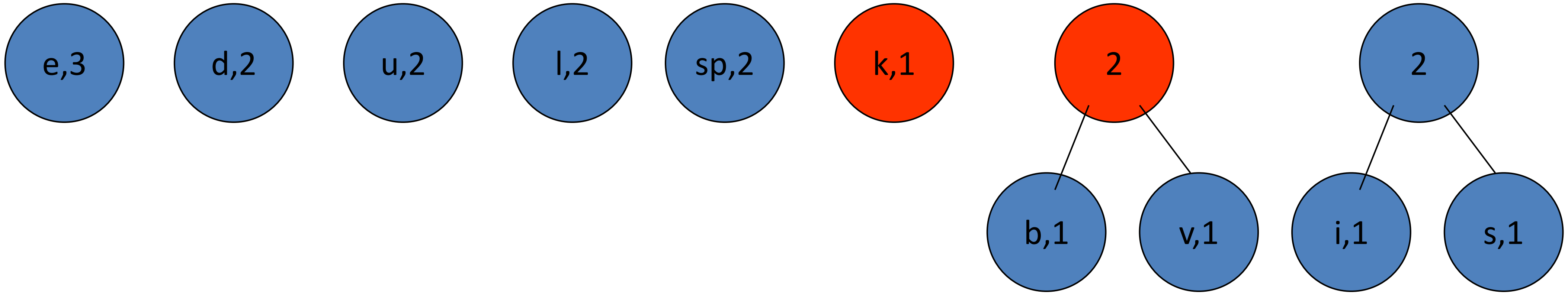
Huffman Coding



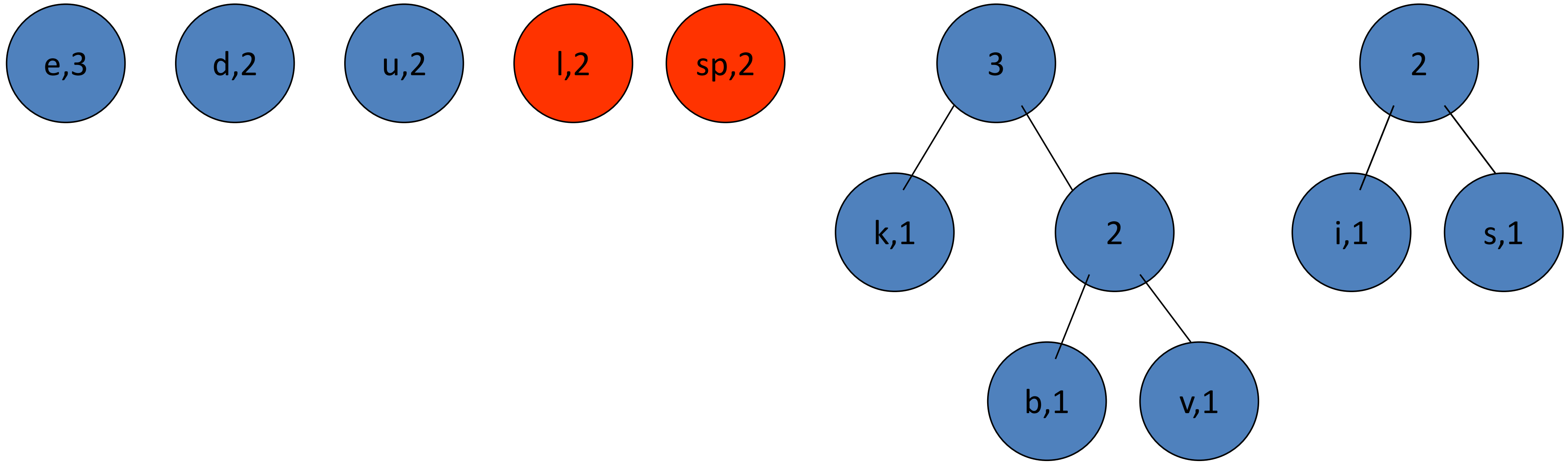
Huffman Coding



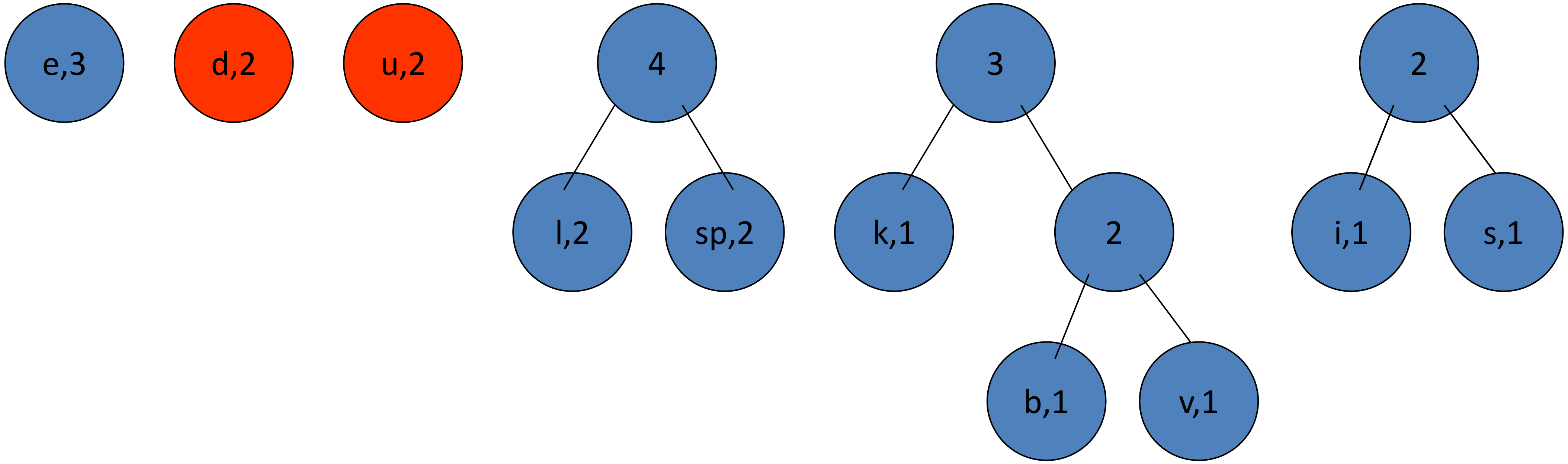
Huffman Coding



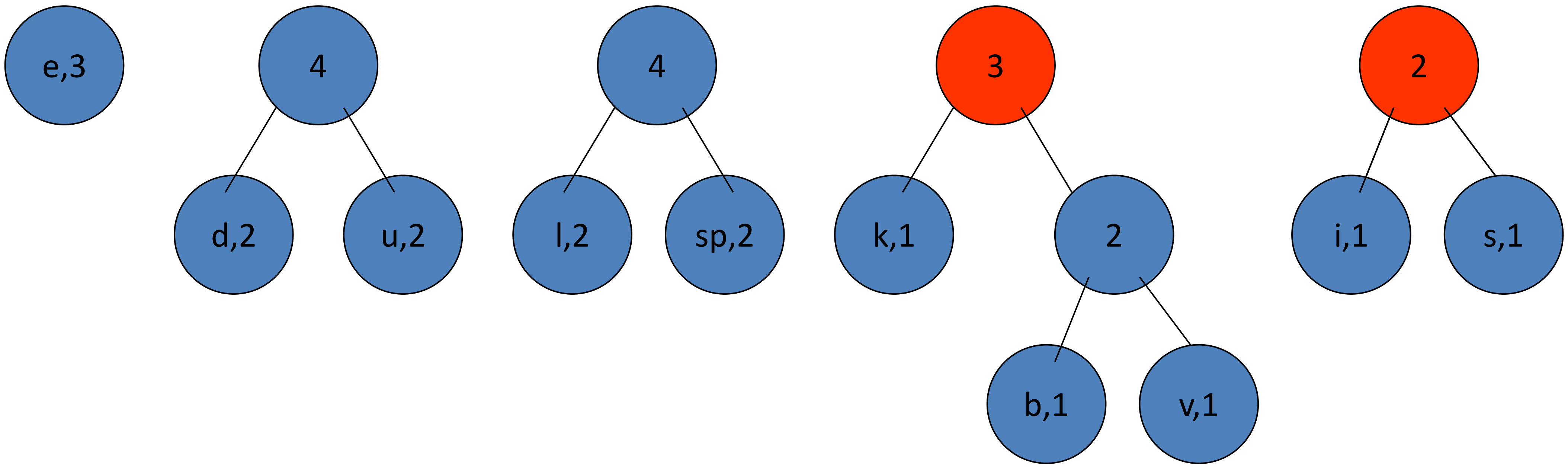
Huffman Coding



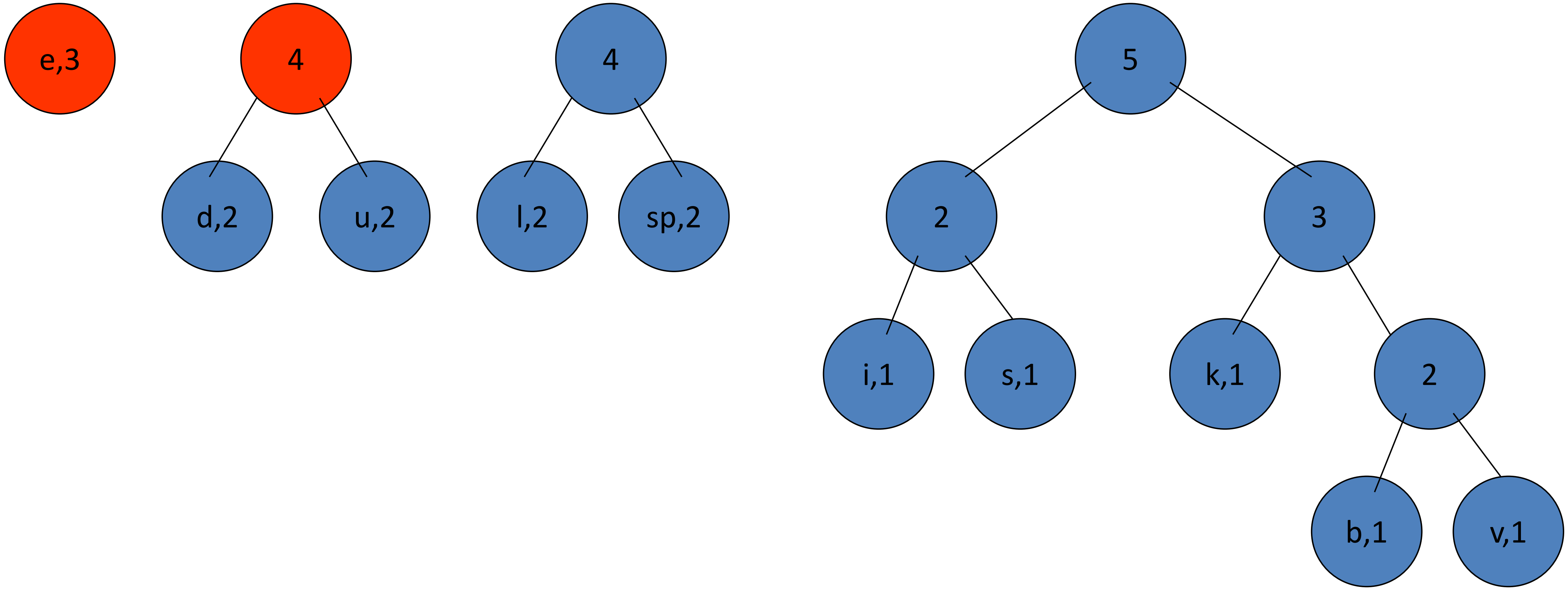
Huffman Coding



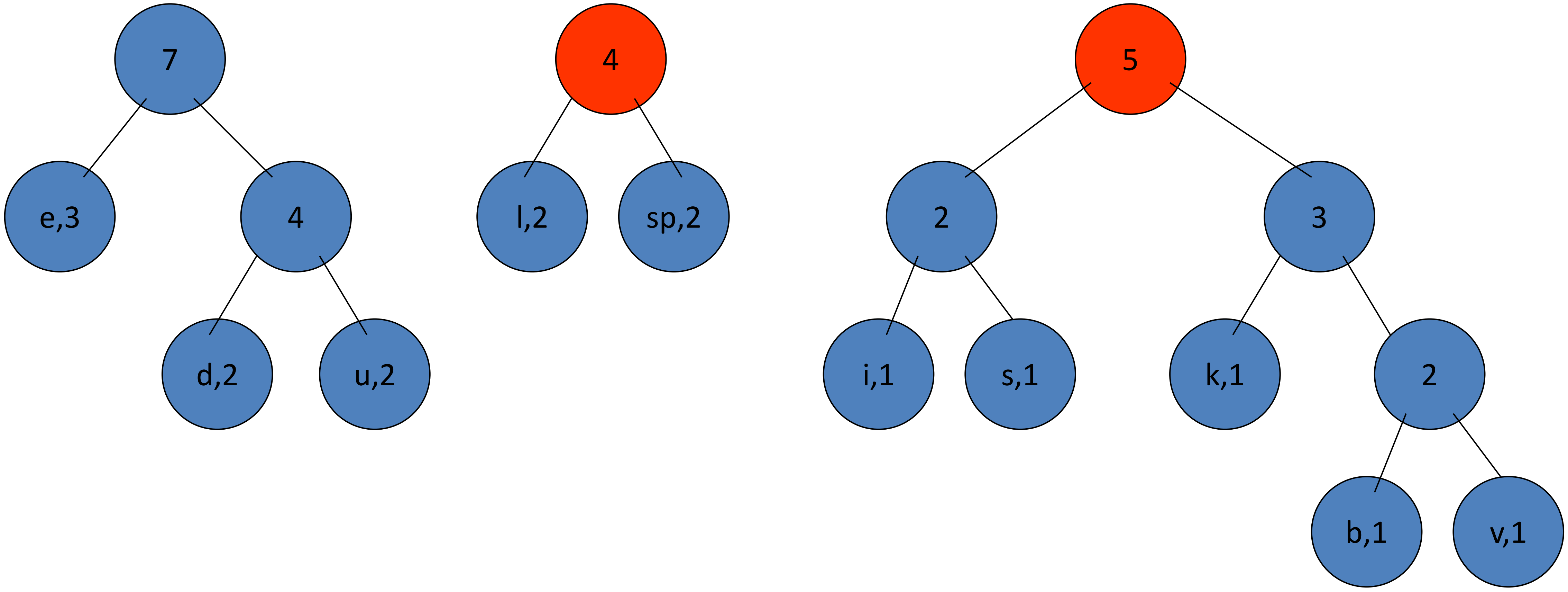
Huffman Coding



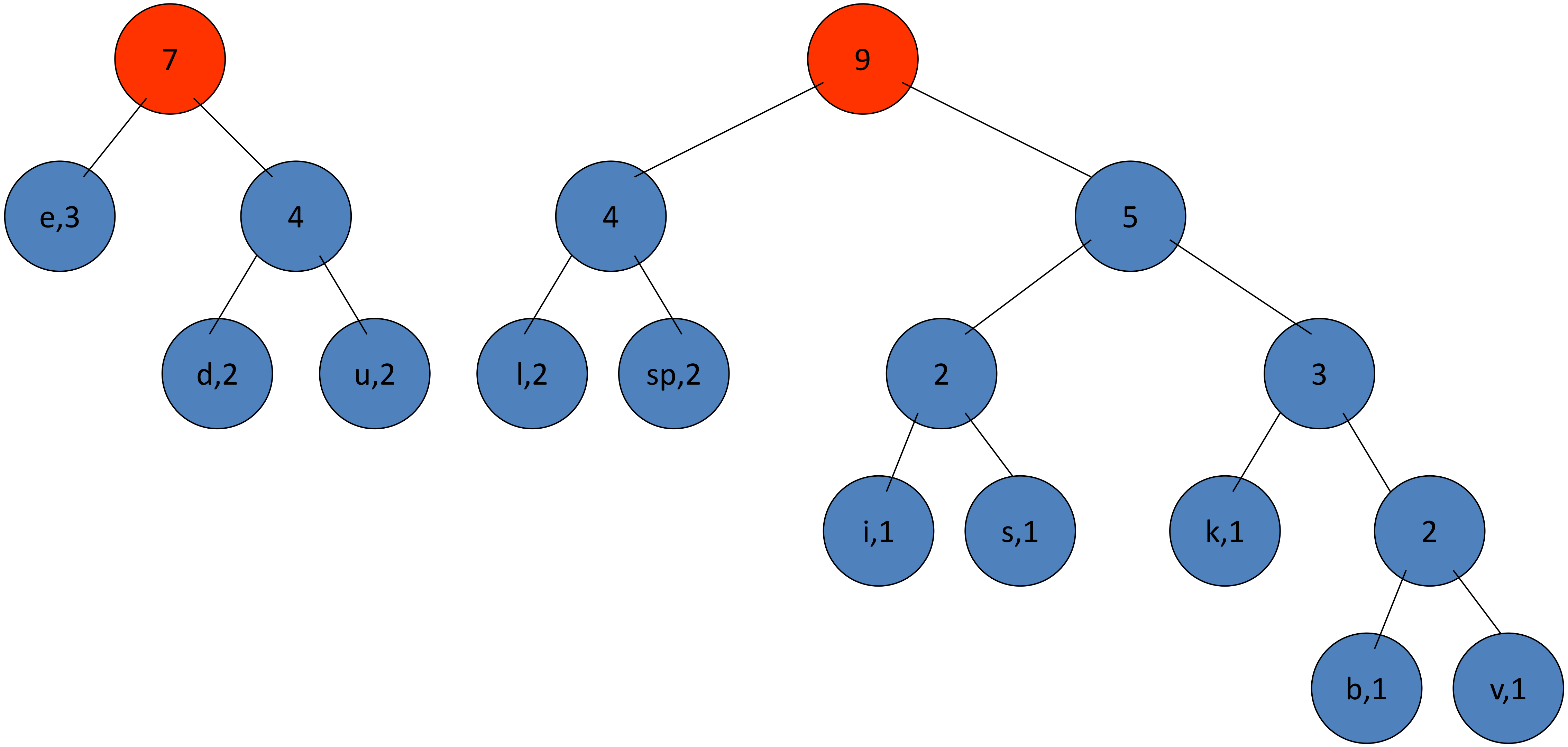
Huffman Coding



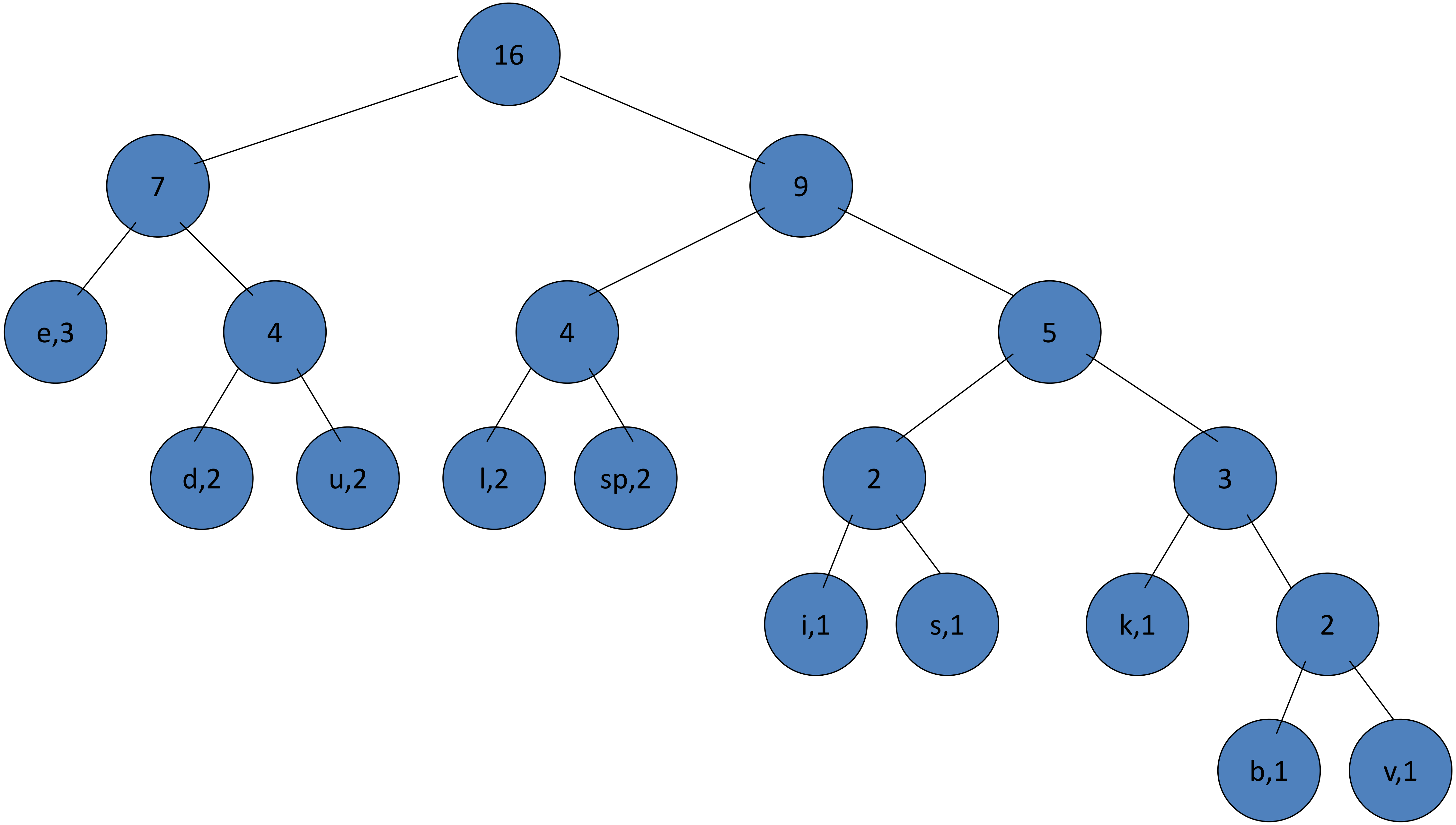
Huffman Coding



Huffman Coding



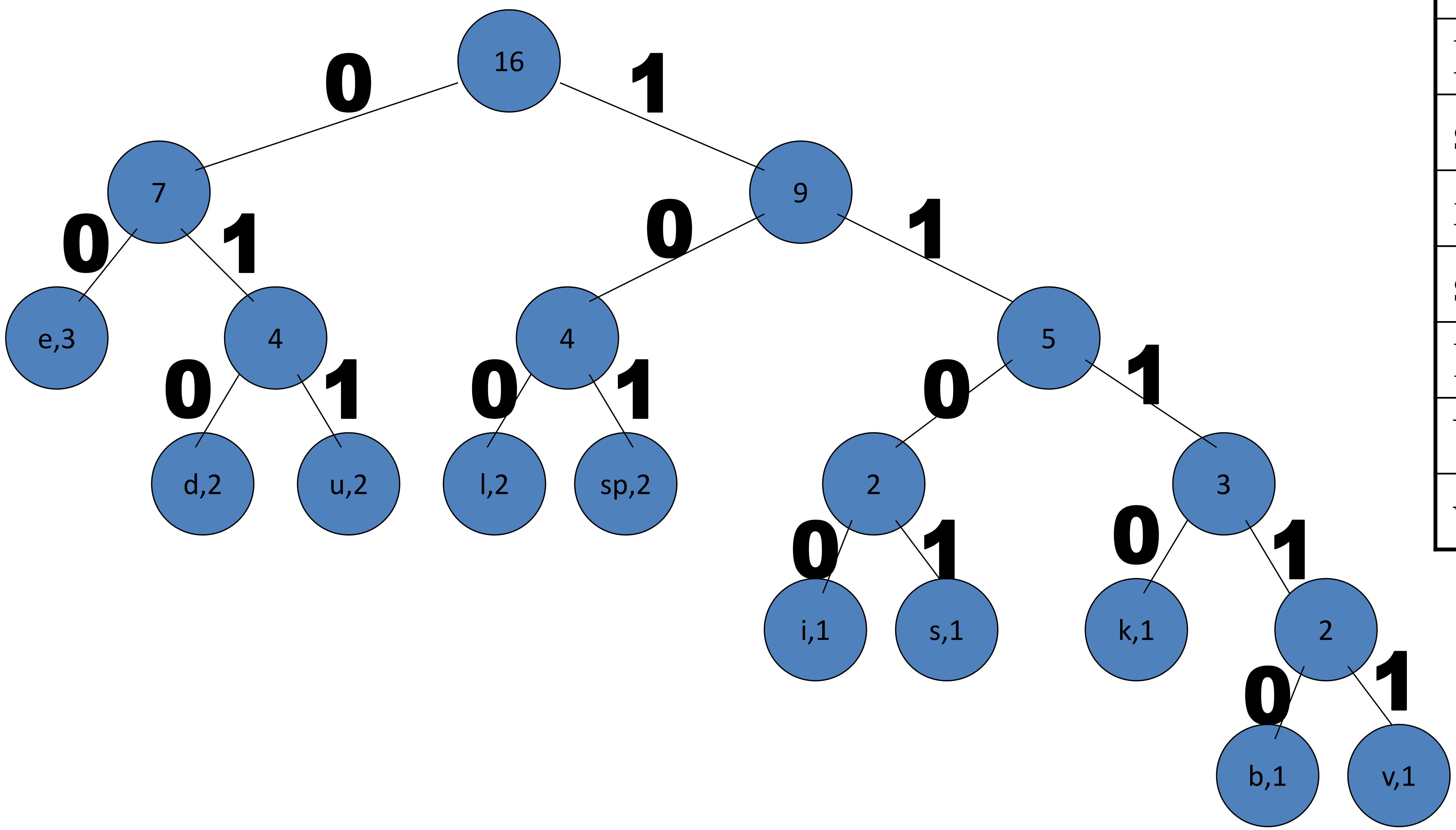
Huffman Coding



Huffman Coding

- Now we assign codes to the tree by placing a 0 on every left branch and a 1 on every right branch
- A traversal of the tree from root to leaf give the Huffman code for that particular leaf character
- Note that no code is the prefix of another code

Huffman Coding



e	00
d	010
u	011
l	100
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111

Huffman Coding

- These codes are then used to encode the string
- Thus, "duke blue devils" turns into:

010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101

- When grouped into 8-bit bytes:

01001111 10001011 11101000 11001010 10001111 11100100 1101xxxx

- Thus it takes 7 bytes of space compared to 16 characters * 1 byte/char = 16 bytes uncompressed

Huffman Coding

- Uncompressing works by reading in the file bit by bit
  - Start at the root of the tree
  - If a 0 is read, head left
  - If a 1 is read, head right
  - When a leaf is reached decode that character and start over again at the root of the tree
- Thus, we need to save Huffman table information as a header in the compressed file
  - Doesn’t add a significant amount of size to the file for large files (which are the ones you want to compress anyway)
  - Or we could use a fixed universal set of codes/freqencies

to be continued...