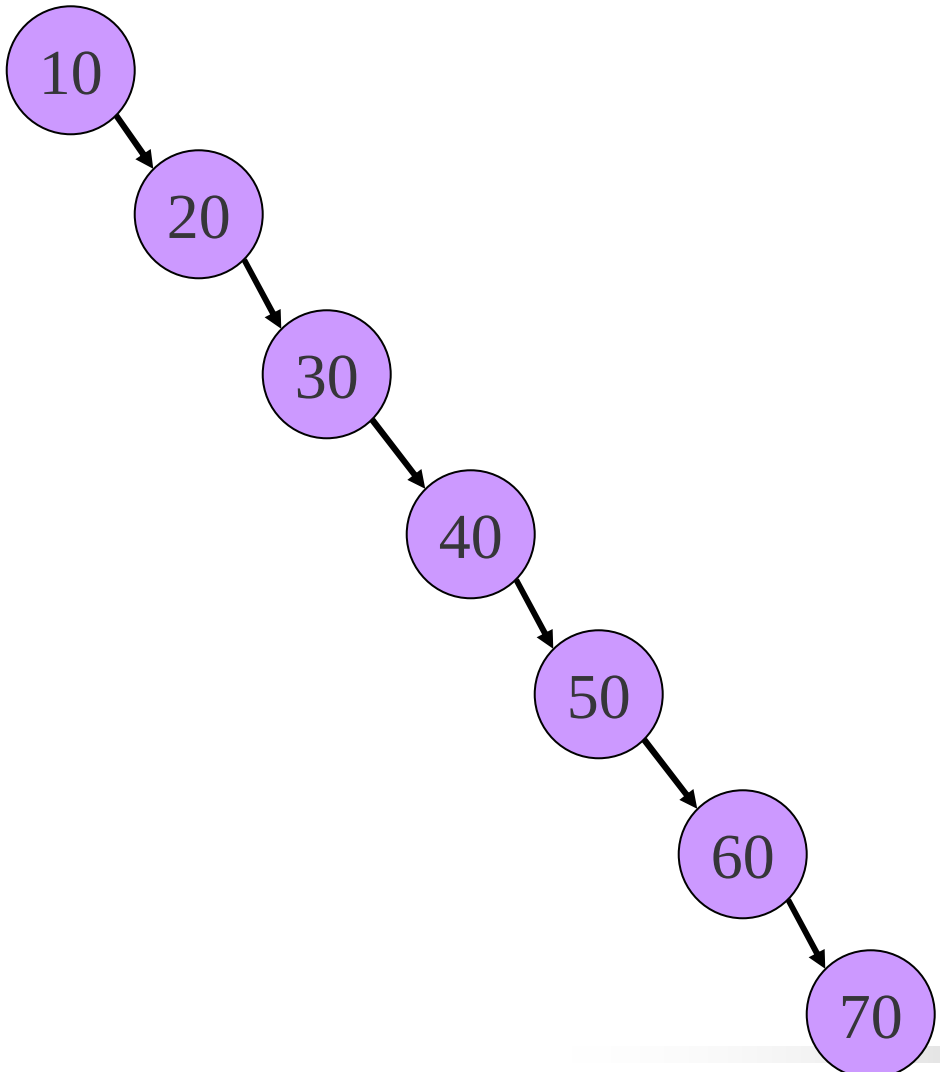
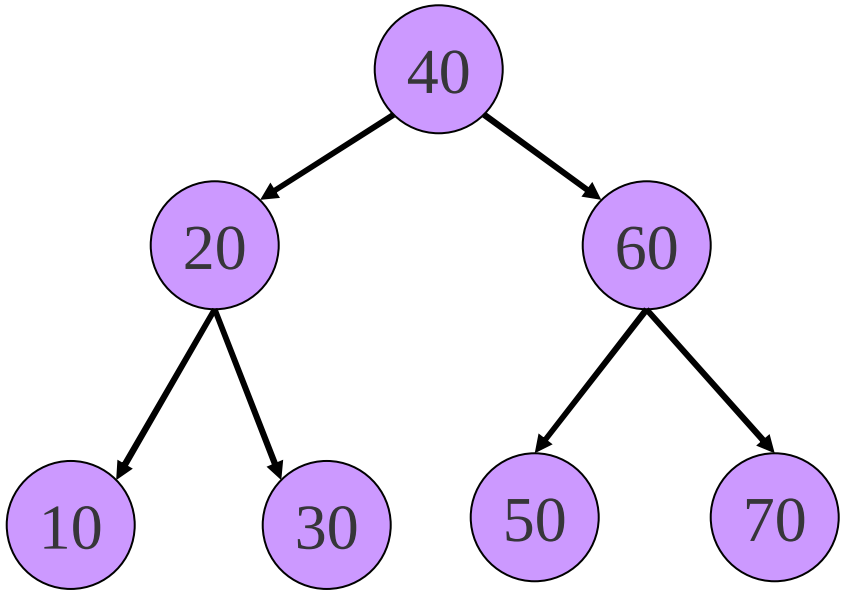
Advanced Implementations of Tables: Balanced Search Trees and Hashing

Balanced Search Trees

- Binary search tree operations such as insert, delete, retrieve, etc. depend on the length of the path to the desired node
- The path length can vary from $\log_2(n+1)$ to $O(n)$ depending on how balanced or unbalanced the tree is
- The shape of the tree is determined by the values of the items and the order in which they were inserted

Examples

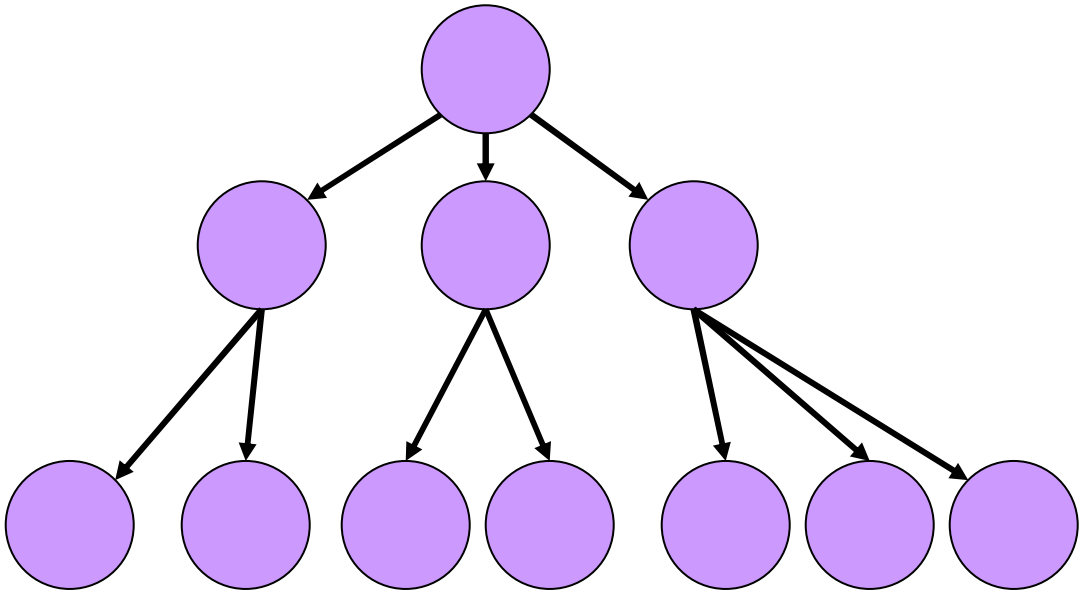




Can you get the same tree with different insertion orders?

2-3 Trees

- Each internal node has two or three children
- All leaves are at the same level
- 2 children = 2-node, 3 children = 3-node

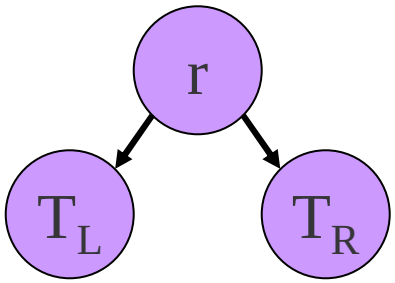


2-3 Trees (continued)

- 2-3 trees are not binary trees (duh)
- A 2-3 tree of height h always has at least 2^h-1 nodes
  - i.e. greater than or equal to a binary tree of height h
- A 2-3 tree with n nodes has height less than or equal to $\log_2(n+1)$
  - i.e less than or equal to the height of a full binary tree with n nodes

Definition of 2-3 Trees

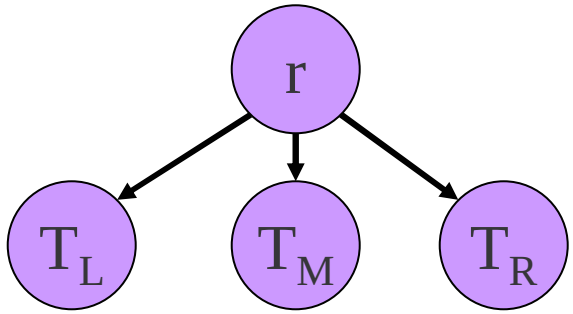
- T is a 2-3 tree of height h if
- T is empty (height 0), OR
- T is of the form



- Where r is a node that contains one data item and T_L and T_R are 2-3 trees, each of height h-1, and the search key of r is greater than any in T_L and less than any in T_R, OR

Definition of 2-3 Trees (continued)

- T is of the form



- Where *r* is a node that contains two data items and T_L, T_M, and T_R are 2-3 trees, each of height *h*-1, and the smaller search key of *r* is greater than any in T_L and less than any in T_M and the larger search key in r is greater than any in T_M and smaller than any in T_R.

Placing Data in a 2-3 tree

1. A 2-node must contain
   - a single data item whose value is
     - greater than any in its left subtree, and
     - smaller than any in its right subtree
2. A 3-mode must contain two data items,
   - the smaller of which is
     - greater than any in its left subtree, and
     - smaller than any in its middle subtree, and
   - the greater of which is
     - greater than any in its middle subtree, and
     - smaller than any in its right subtree
3. A leaf may contain either one or two data items

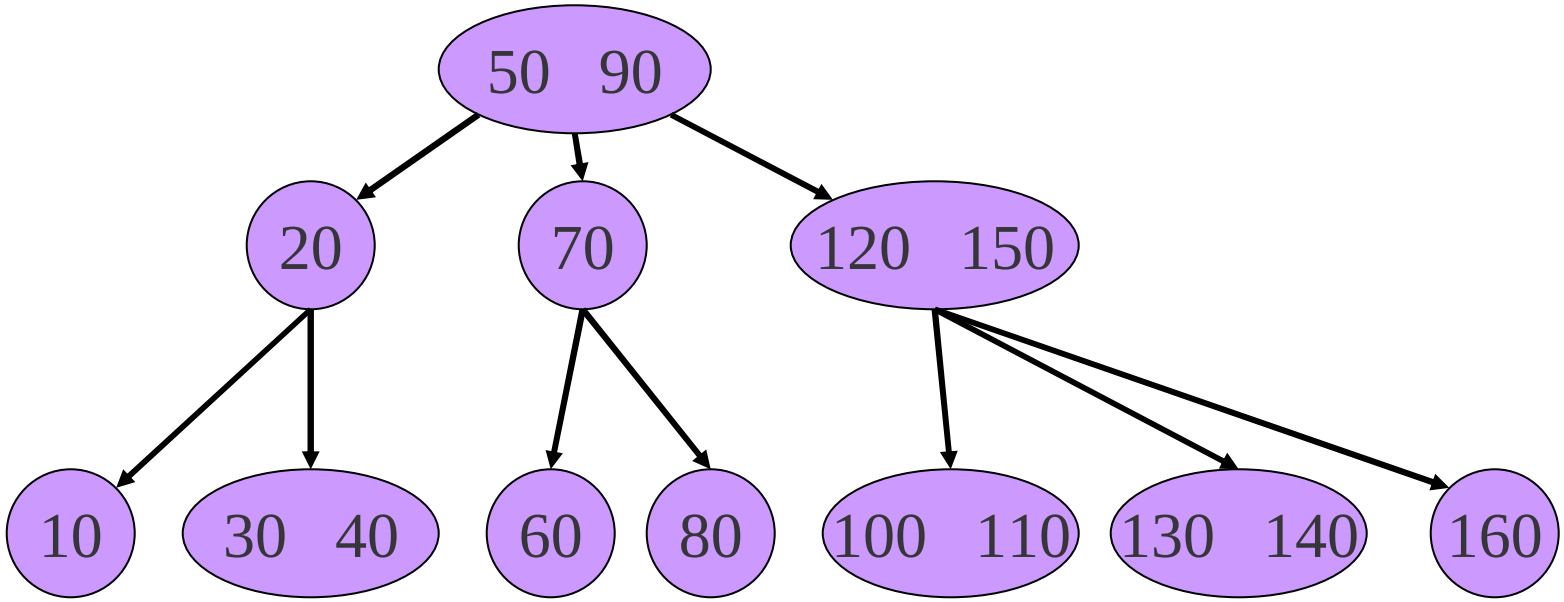
2-3 Node Code

```
import searchkeys.*;

public class Tree23Node {
    private KeyedItem smallitem;
    private KeyedItem largeItem;
    private Tree23Node leftChild;
    private Tree23Node middleChild;
    private Tree23Node rightChild;
}
```

Traversing a 2-3 Tree

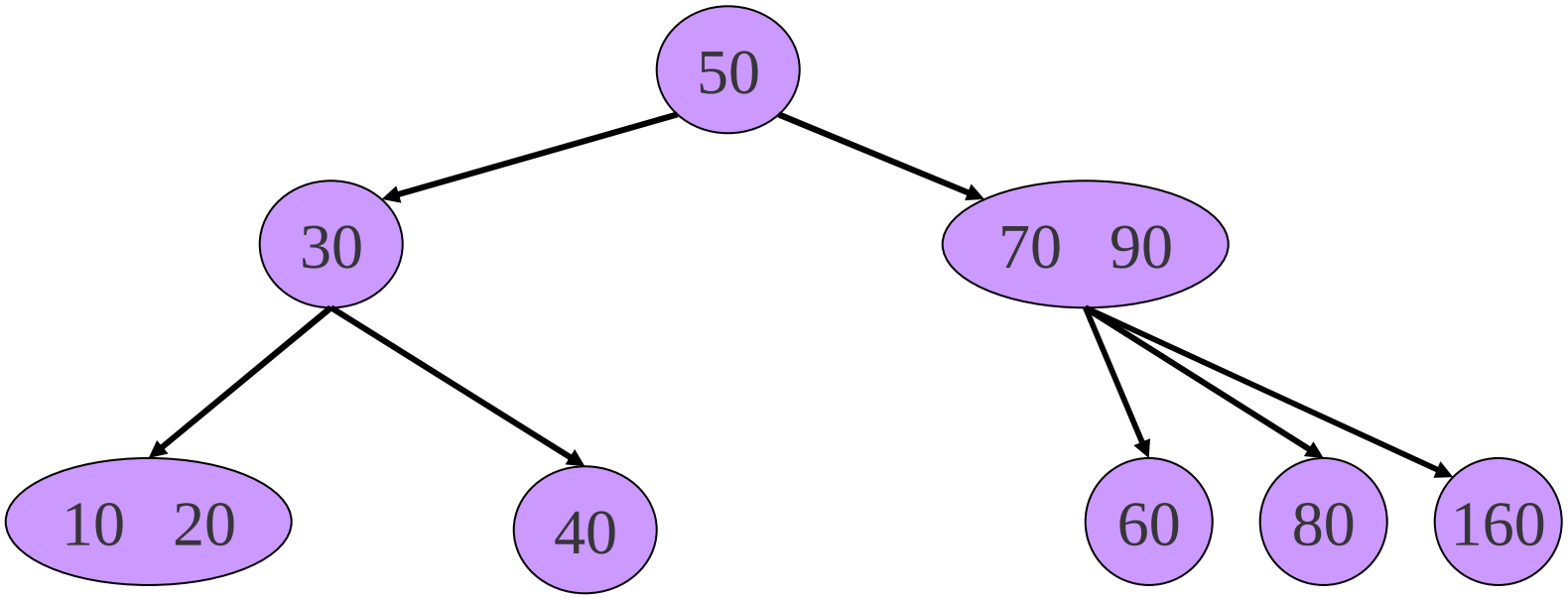
- Just like a binary tree: preorder, inorder, postorder



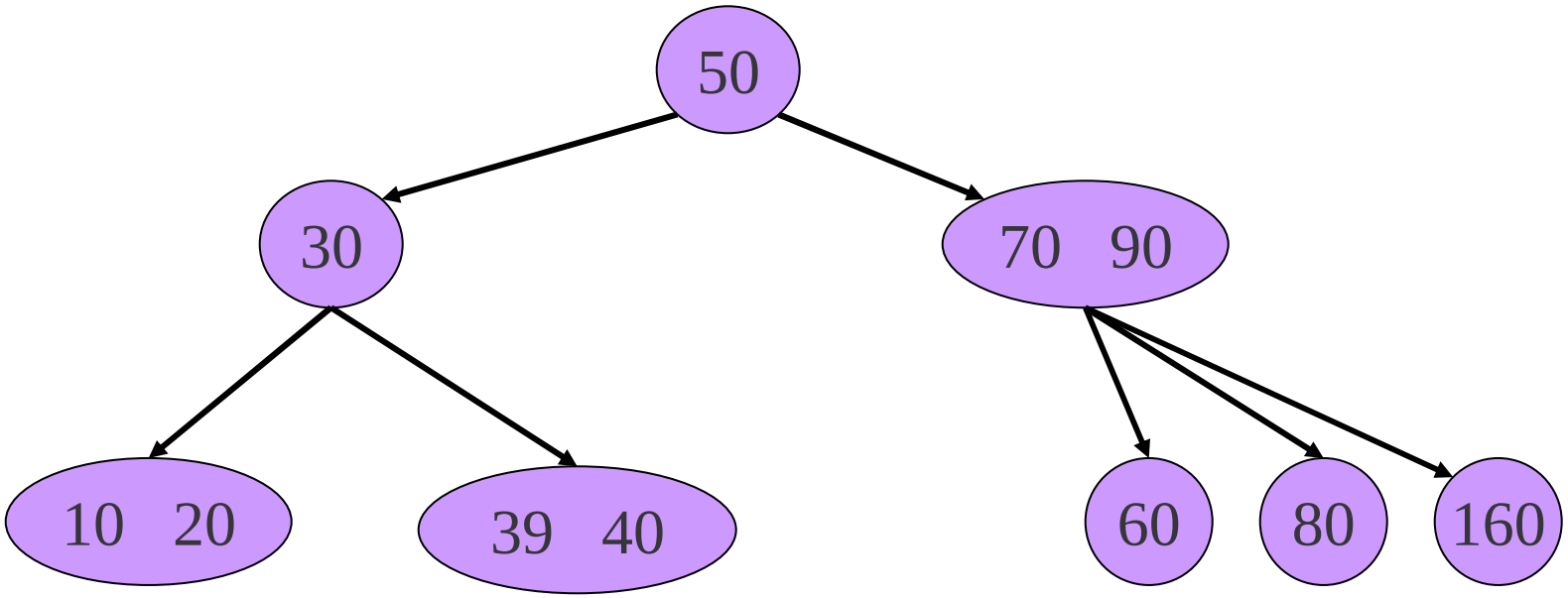
Searching a 2-3 tree

- Same efficiency as a balanced binary search tree: O(logn)
- BUT, easy to keep balanced, unlike binary search trees
- This means that as you insert elements, the balance is easily maintained, and worst case performance remains O(logn)

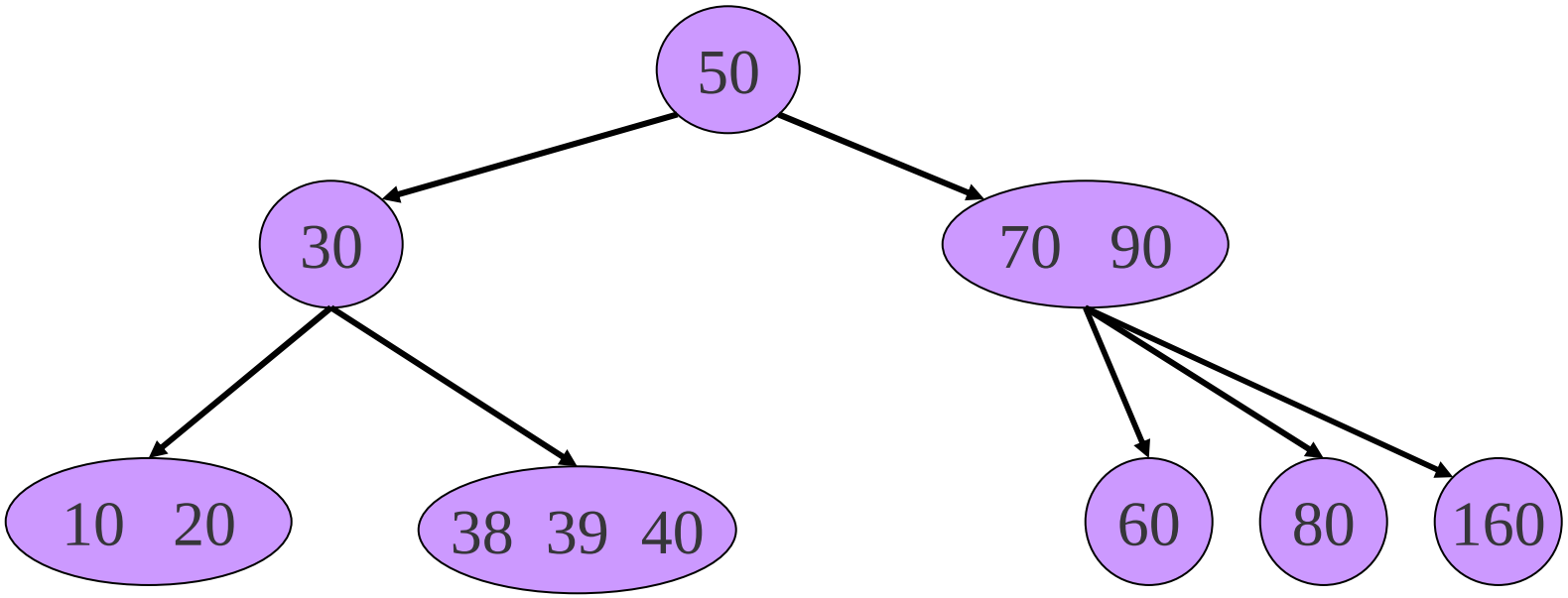
Inserting 39 Into A 2-3 Tree



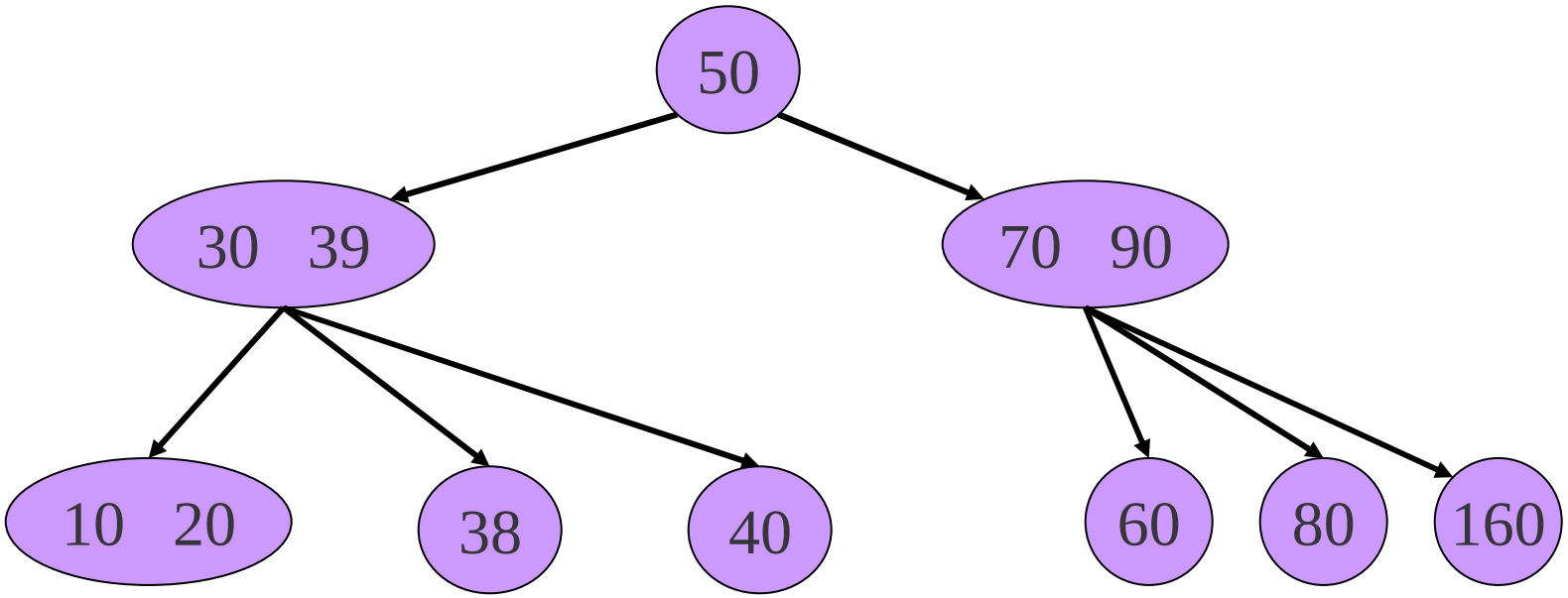
Inserting 38



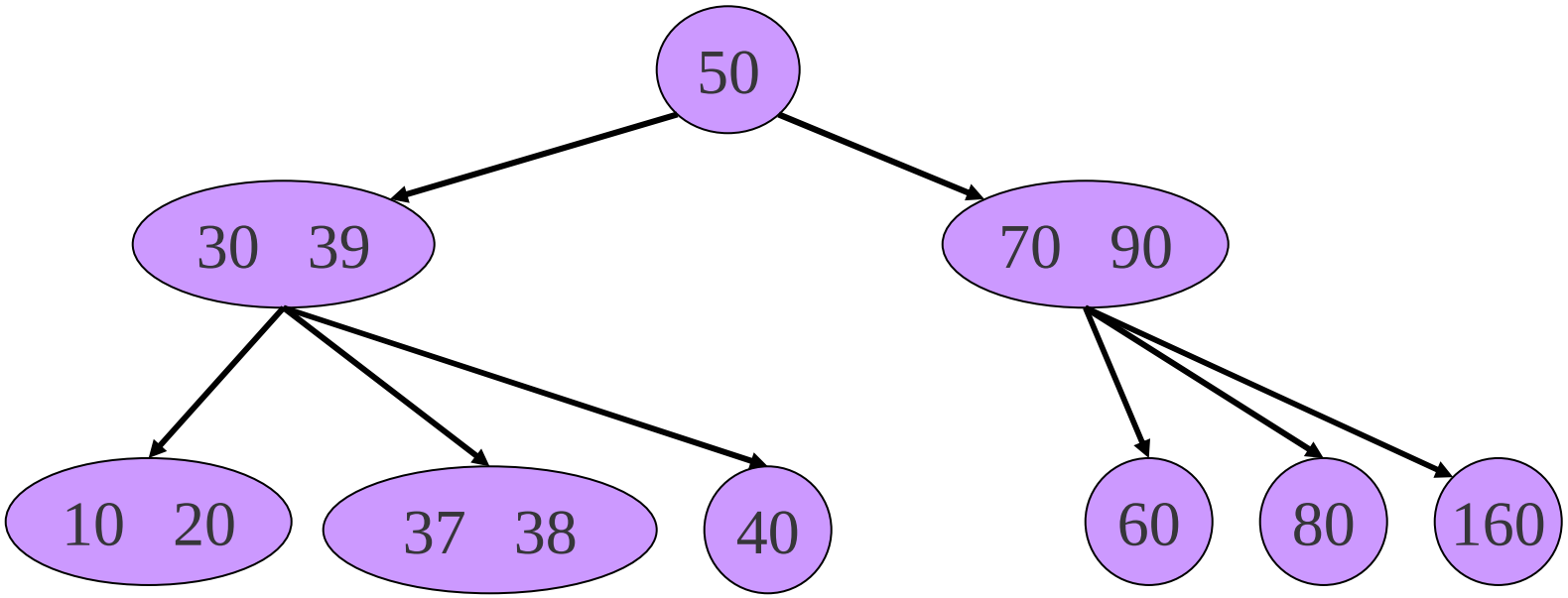
Inserting 38 (continued)



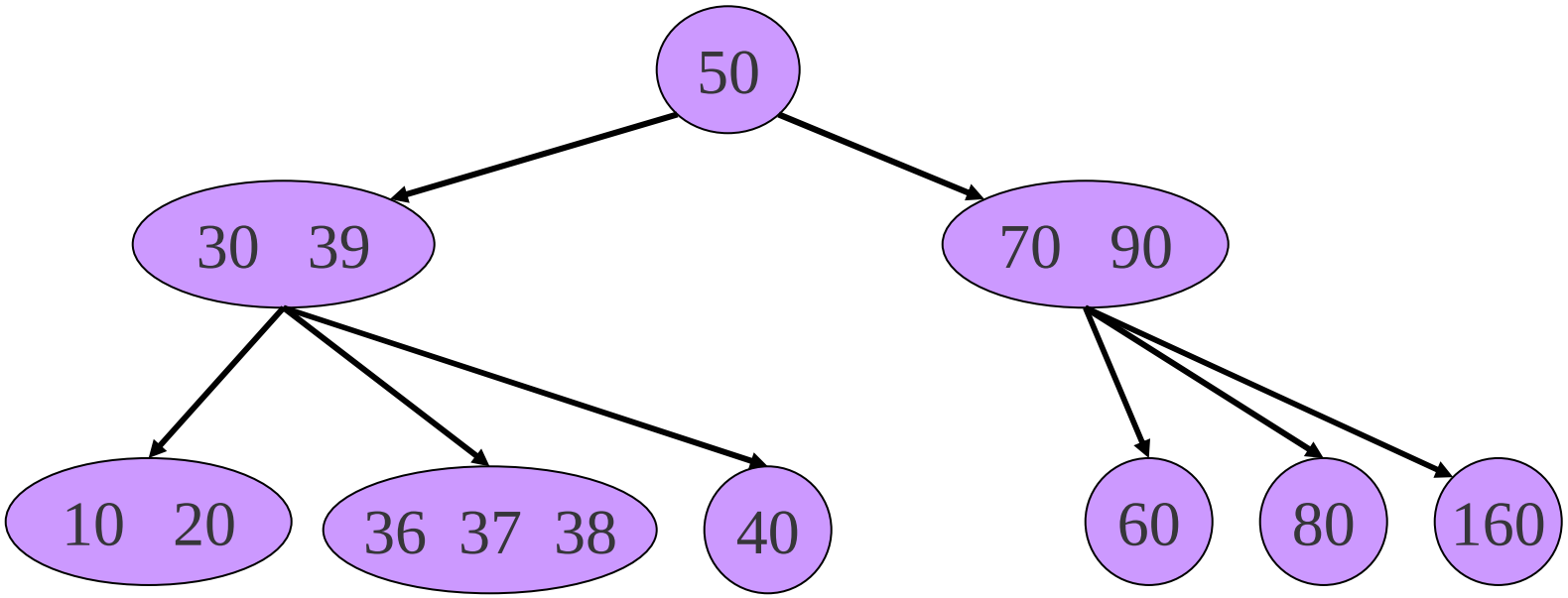
Inserting 37



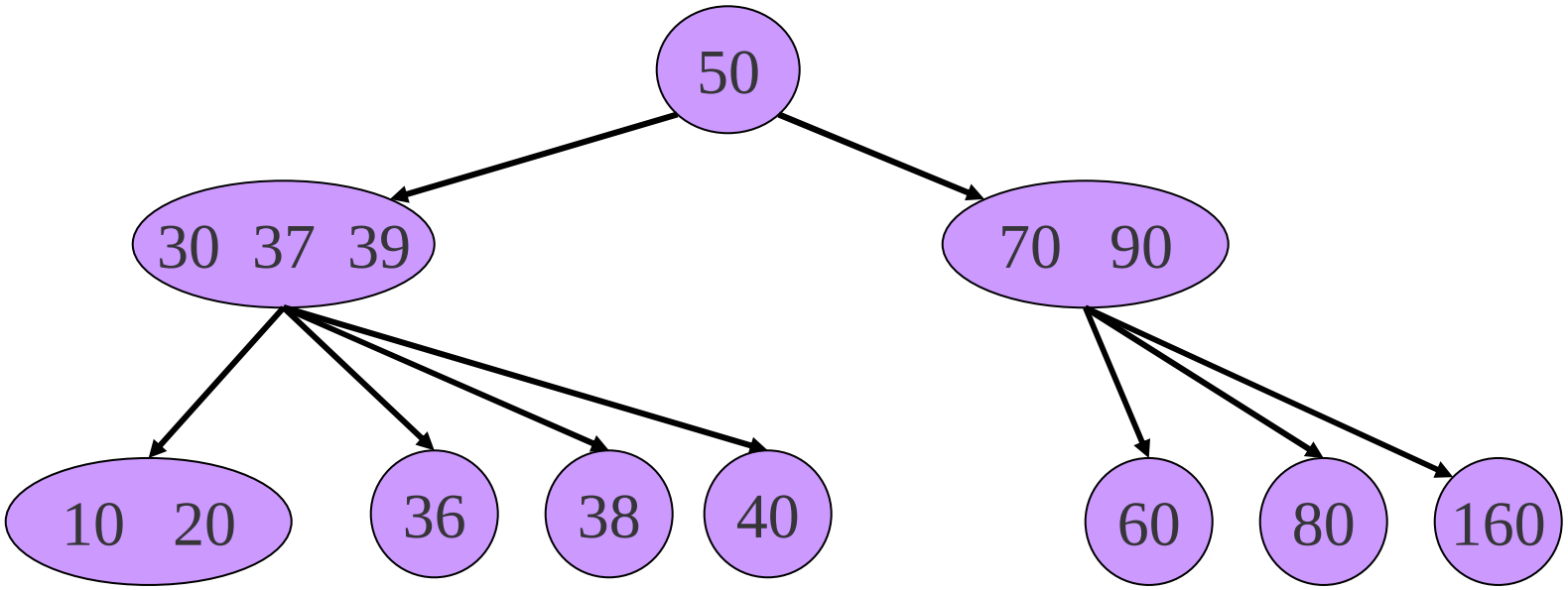
Inserting 36



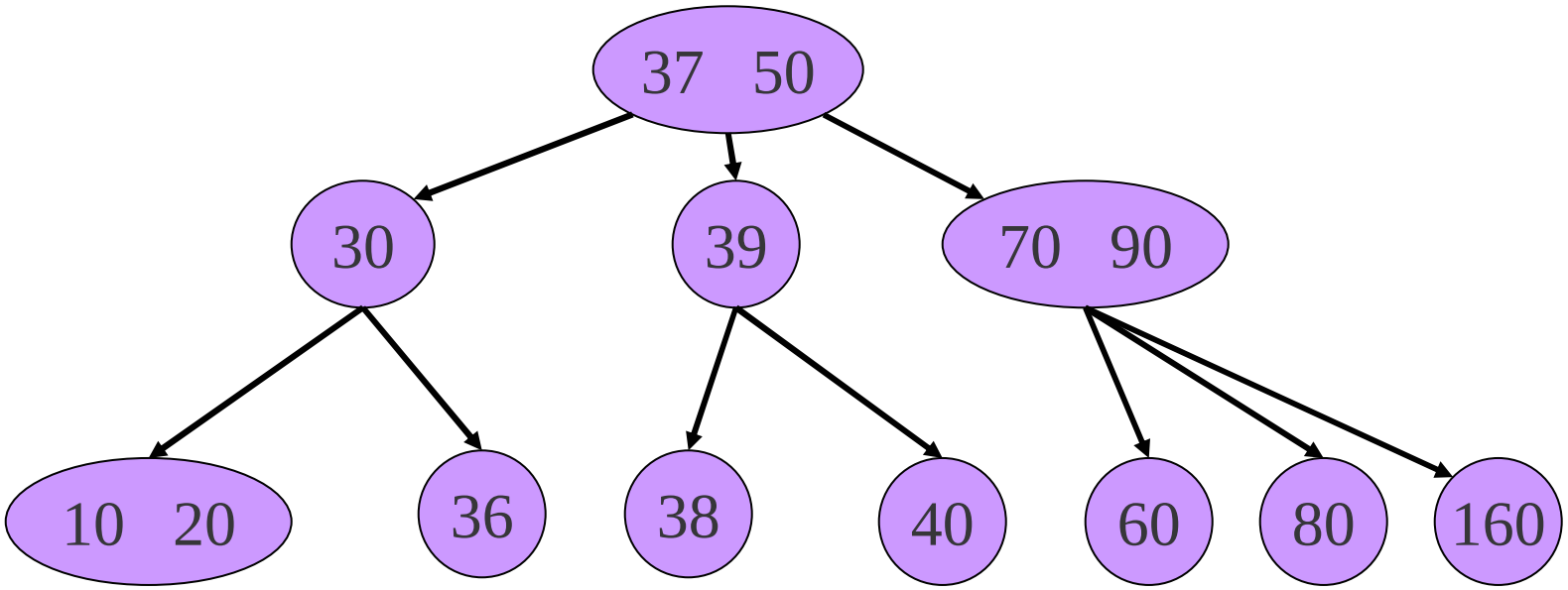
Inserting 36 (continued)



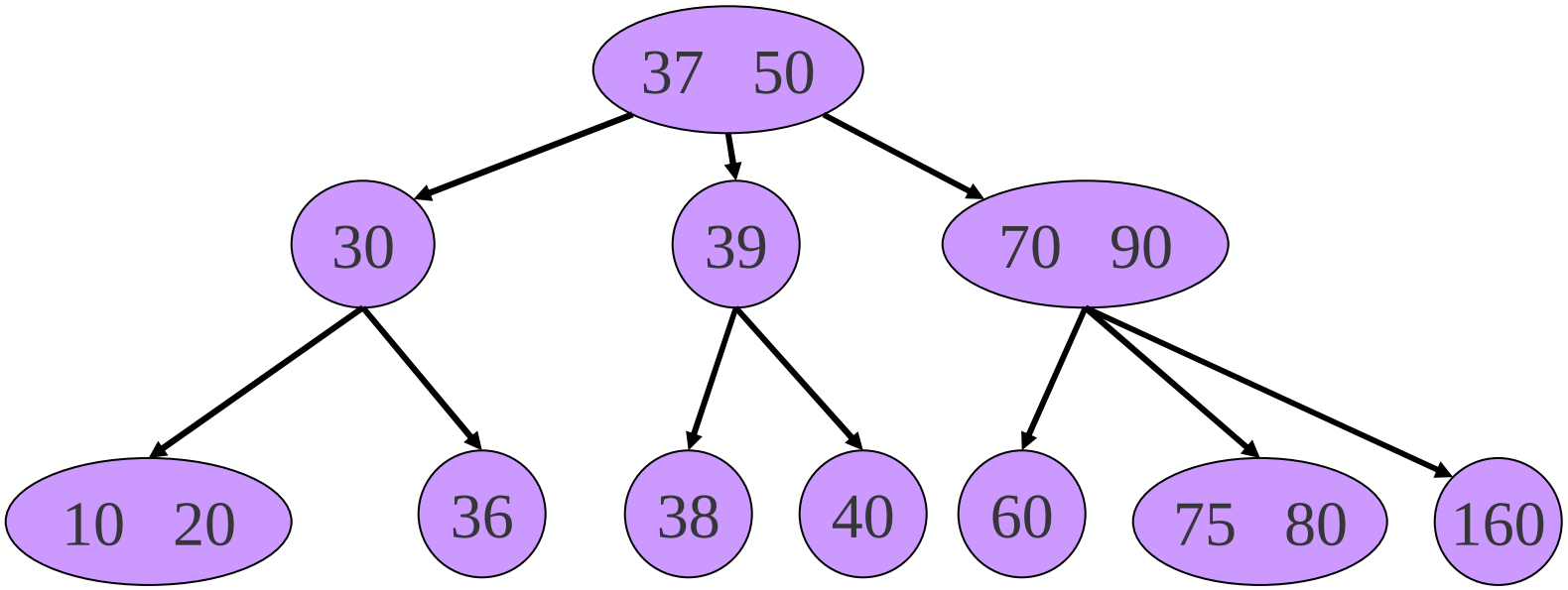
Inserting 36 (continued)



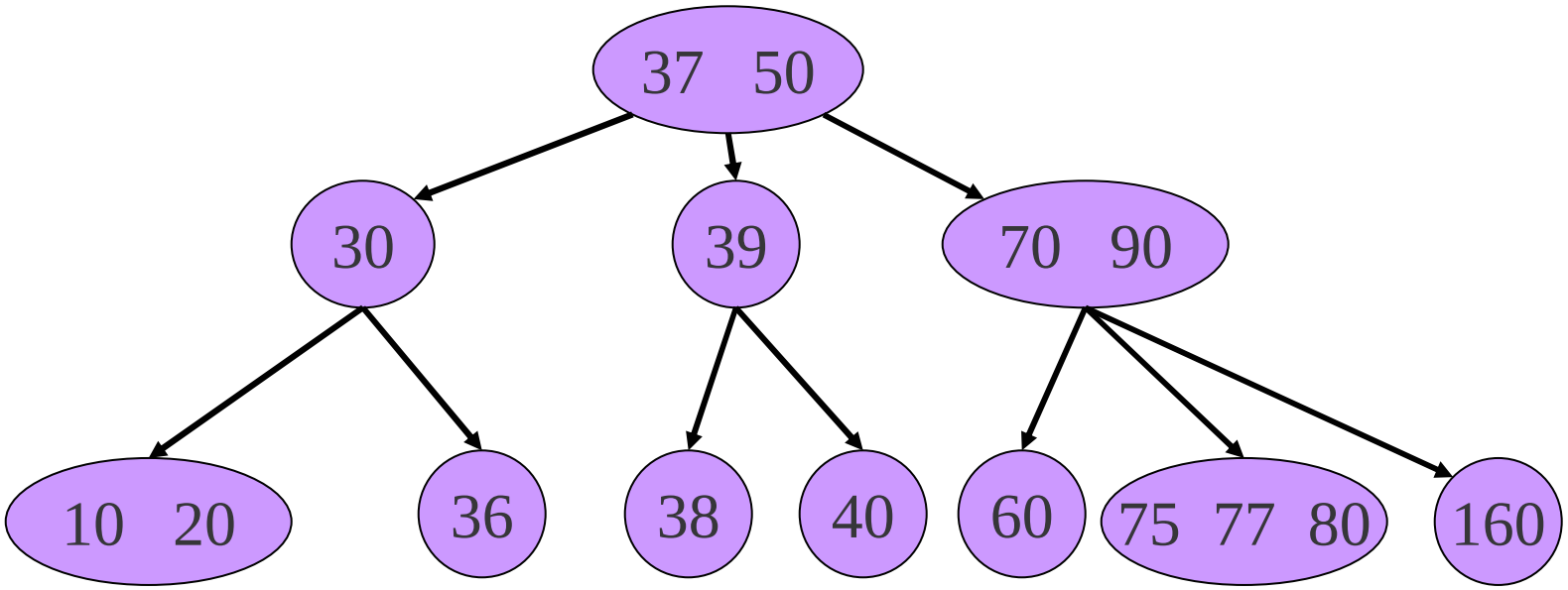
Inserting 75



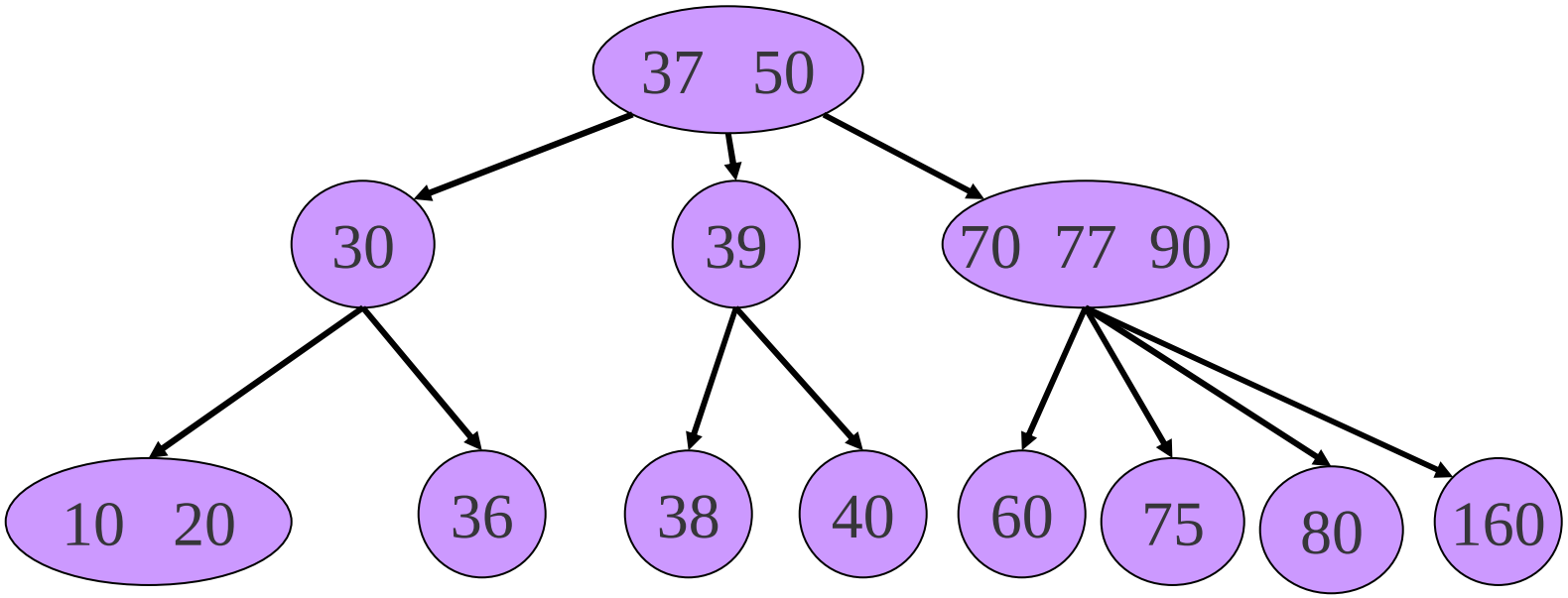
Inserting 77



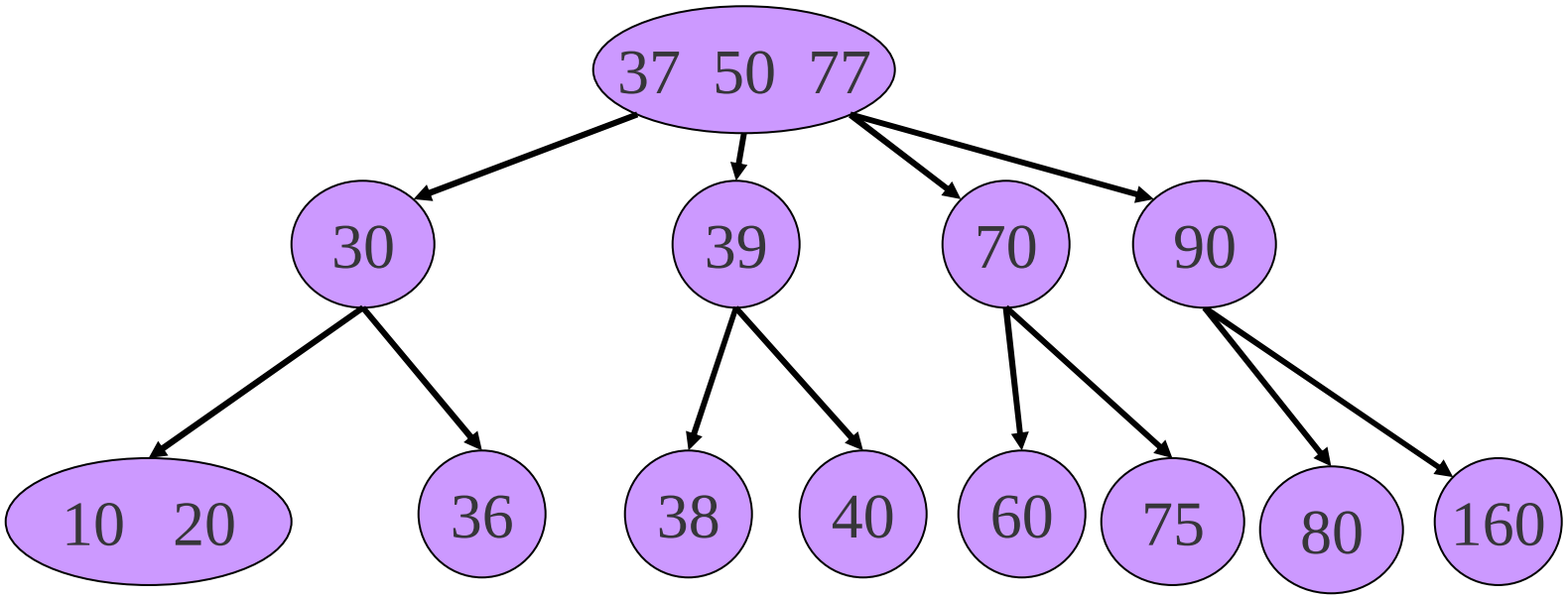
Inserting 77 (continued)



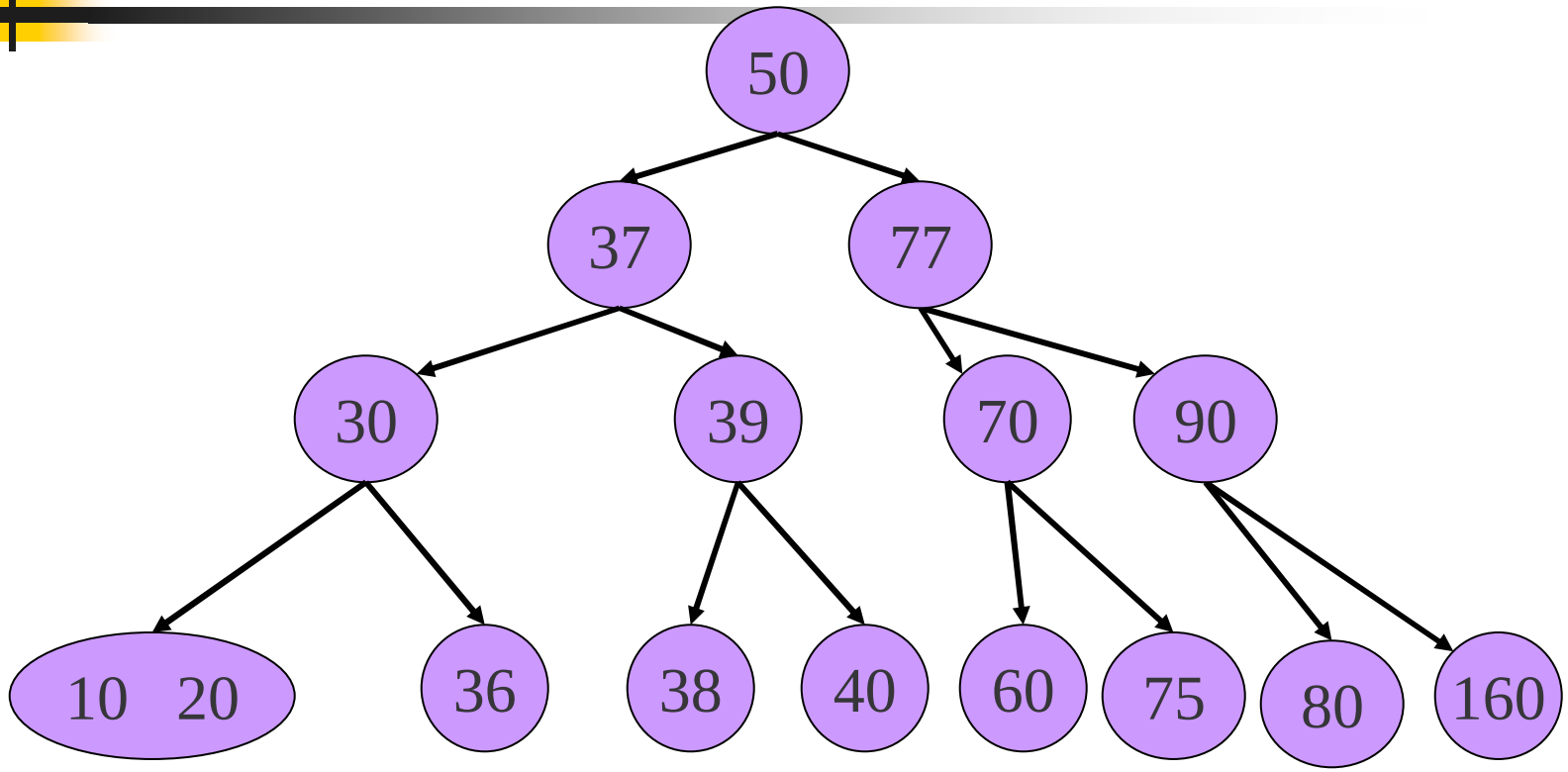
Inserting 77 (continued)



Inserting 77 (continued)



Inserting 77 (continued)



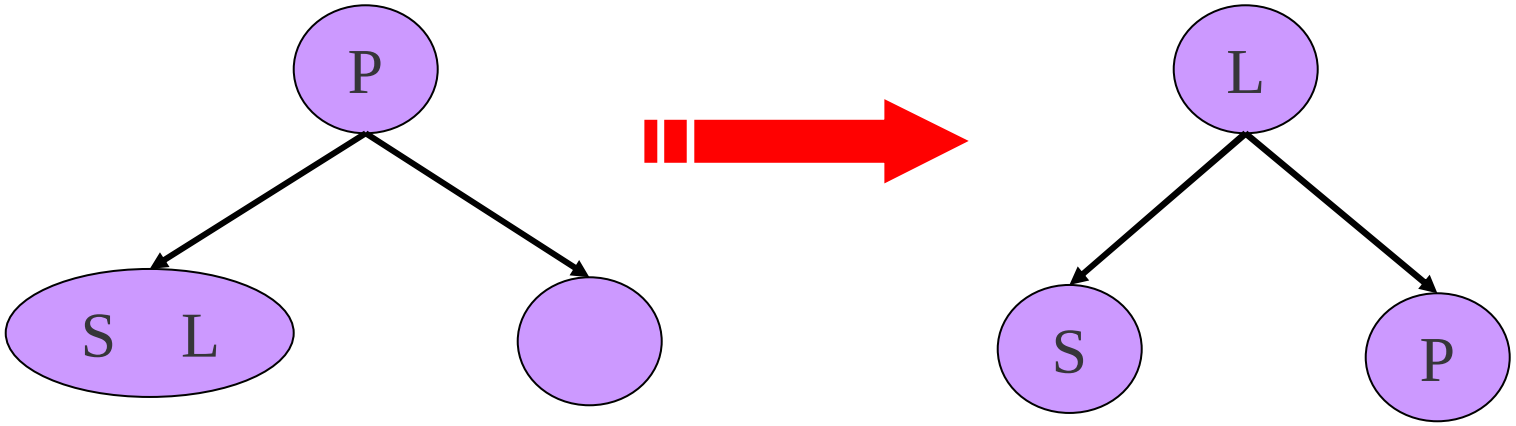
Inserting Into A 2-3 Tree

- Insert into the leaf node in which the search key belongs
- If the leaf has two values, stop
- If the leaf has three values, split the node into two nodes with the smallest and largest values, and
- Push the middle value into the parent node
- Continue with the parent node until either
  - you push a value into a node that had only one value, or
  - you create a new root node

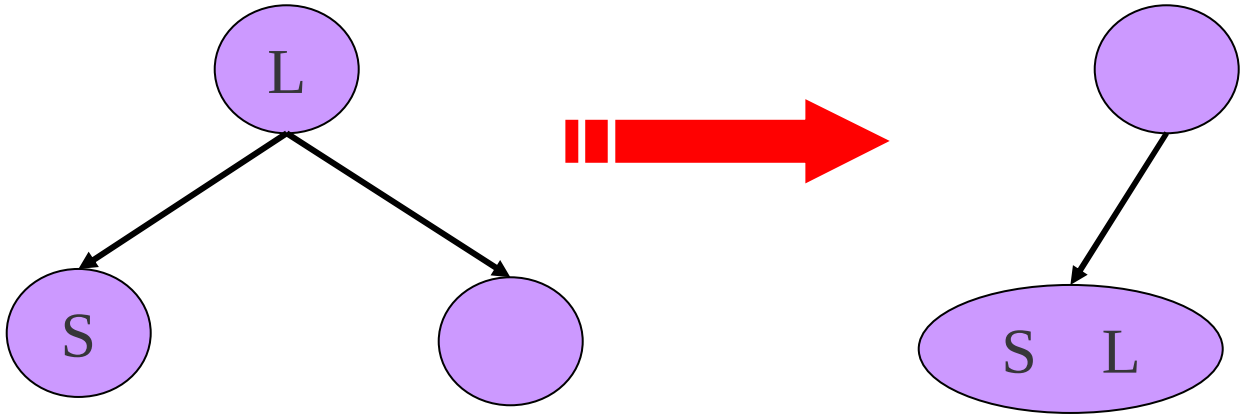
Deleting From A 2-3 Tree

- The inverse of inserting
- Delete the value (in a leaf), then
- Merge empty nodes
- If necessary, delete empty root

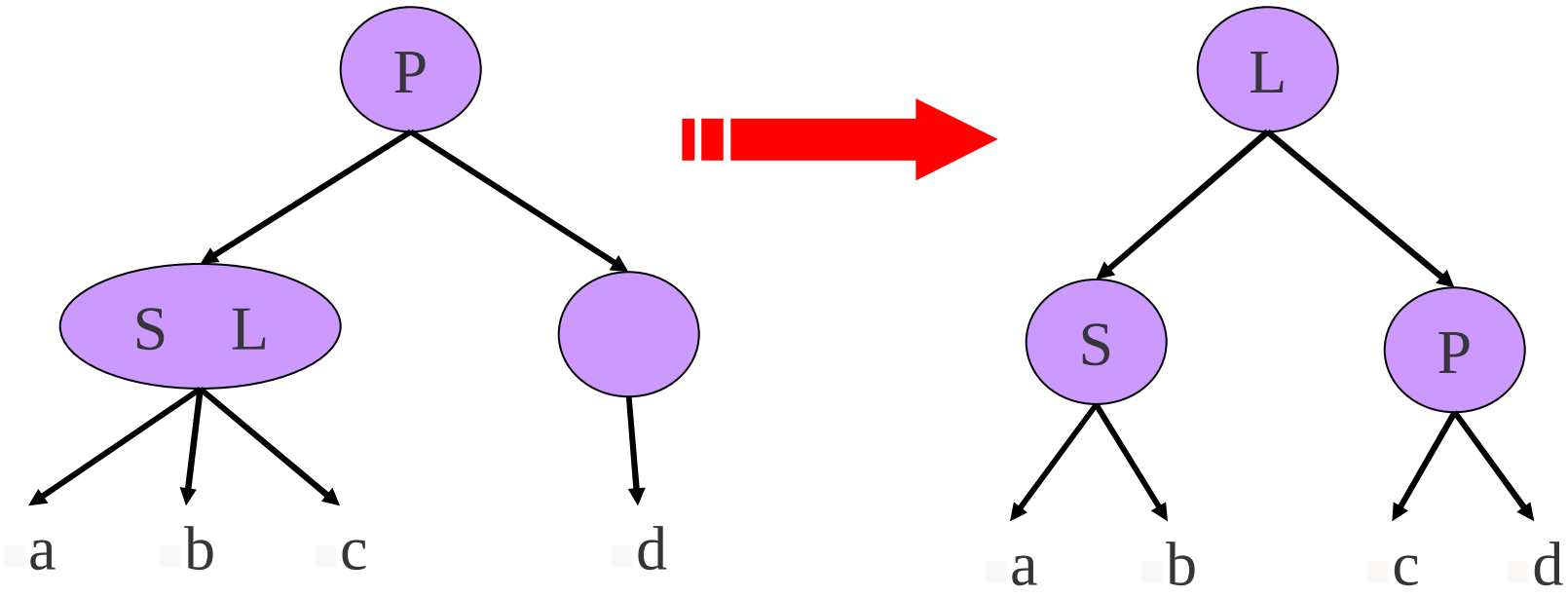
Redistribute I



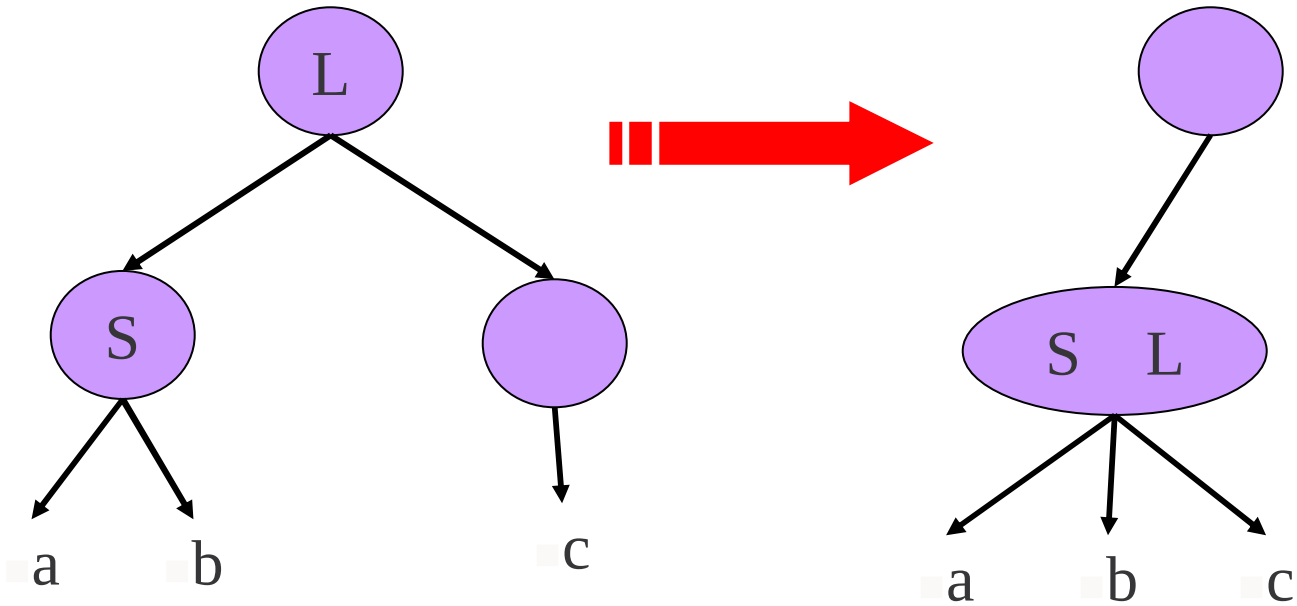
Merge I



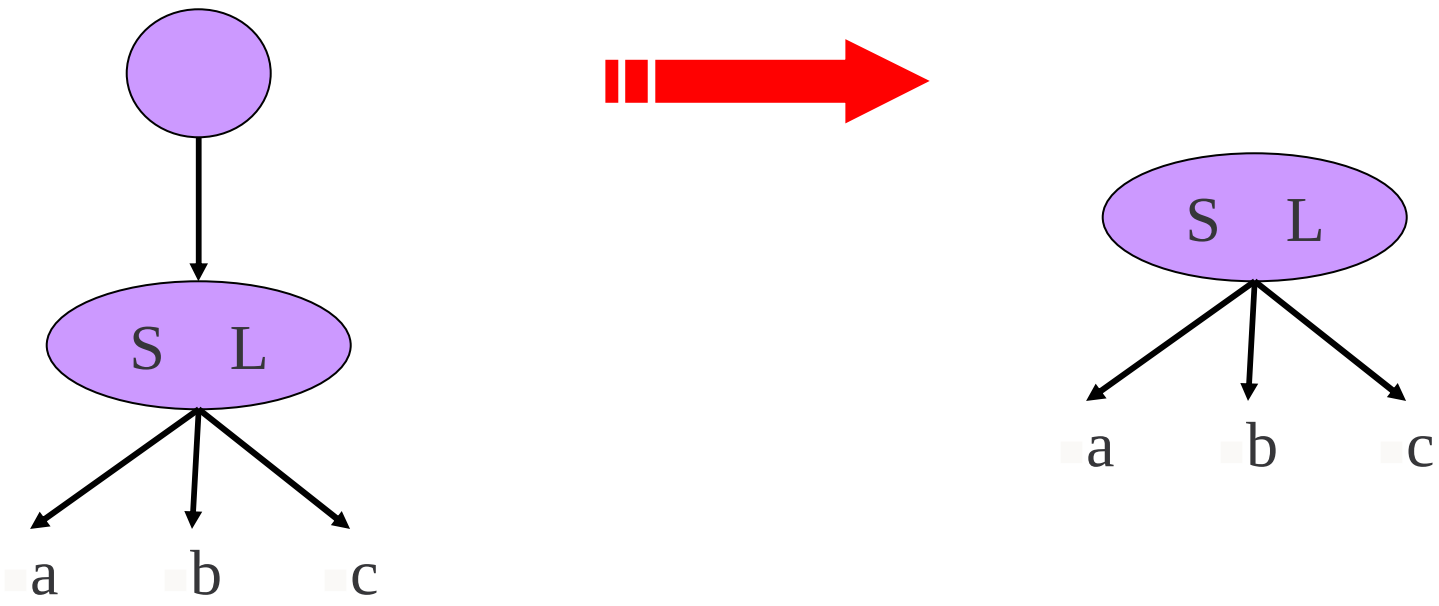
Redistribute II



Merge II



Delete



2-3 Trees: Results

- Slightly more complicated than binary search trees, BUT
- 2-3 Trees are always balanced
- Every operation takes O(logn)

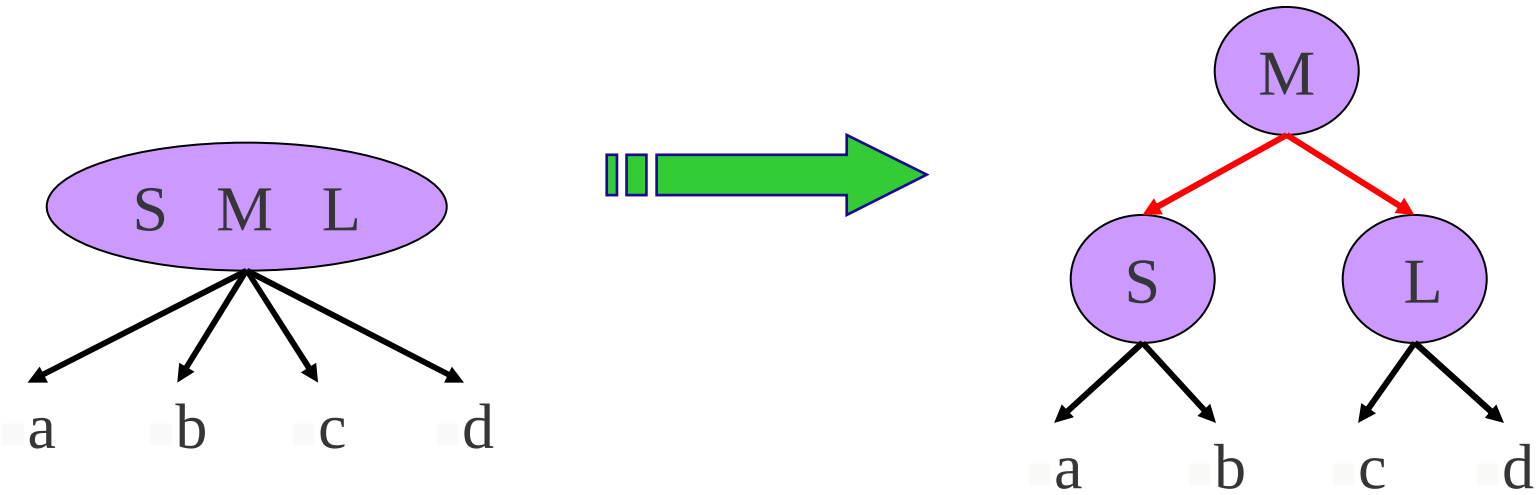
2-3-4 Trees

- Slightly less complicated than 2-3 Trees
- Each node can contain 1–3 values and have 1–4 children
- Inserting
  - Split 4-nodes on the way down
  - Insert into leaf
- Deleting:
  - Only delete from 3-node or 4-node

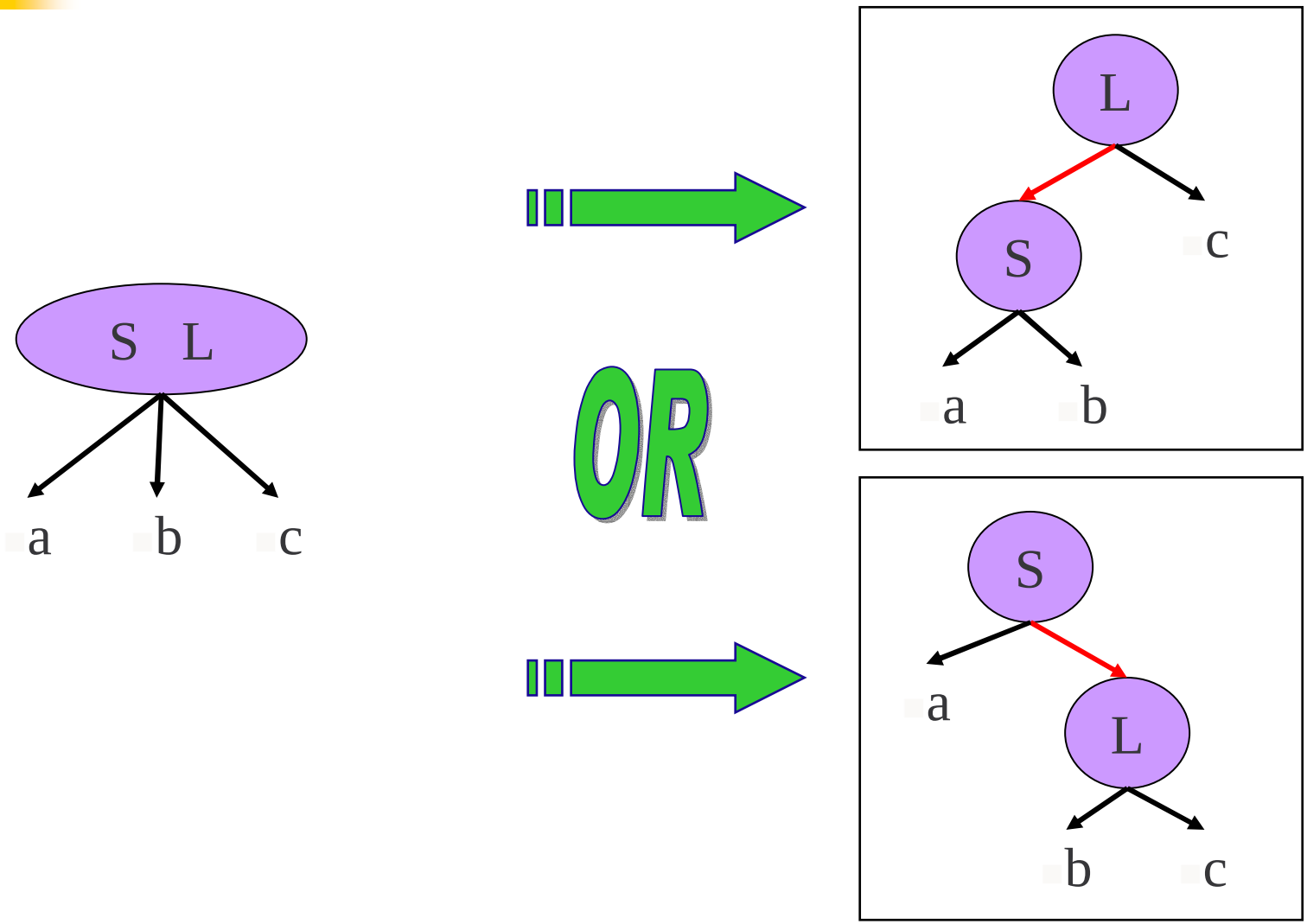
Red-Black Trees

- 2-3 Trees are always balanced
  - O(logn) time for all operations
- 2-3-4 Trees are always balanced
  - O(logn) time for all operations, and
  - Insertion and deletion can be done in a single pass from root to leaf
  - But, require slightly more storage per node
- Red-Black Trees have the advantage of 2-3-4 trees, without the overhead
  - Represent the 2-3-4 Tree as a binary tree with *colored* references (red or black)

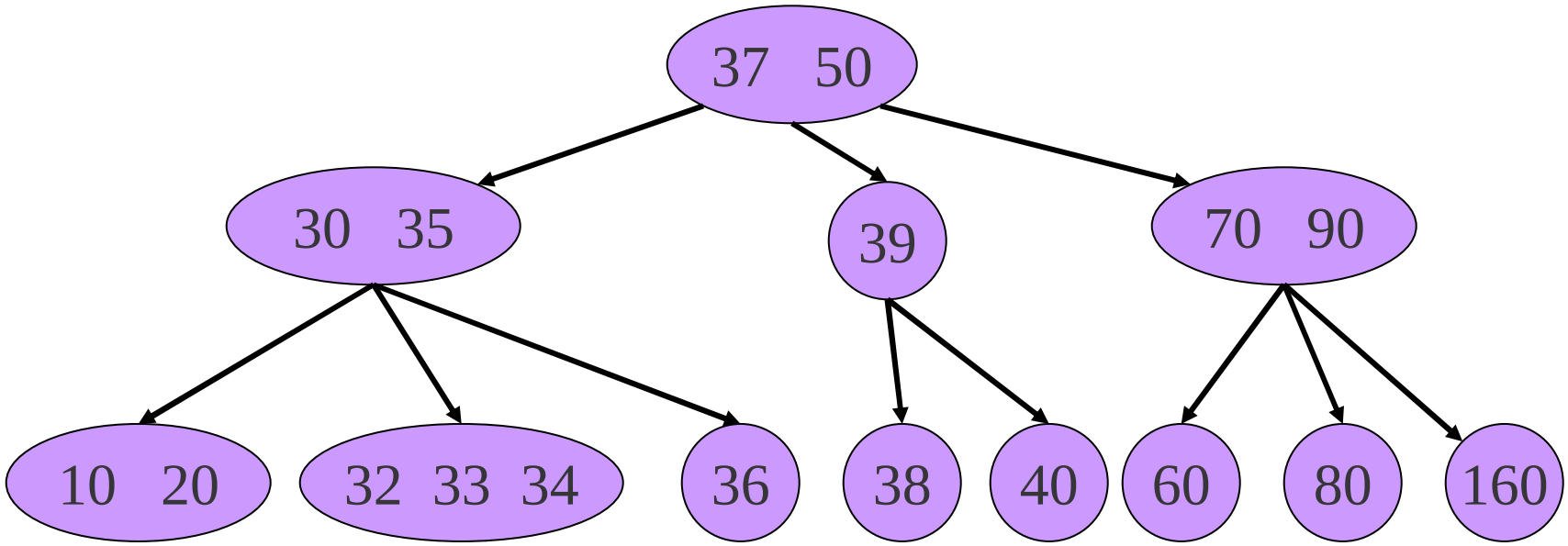
Red-Black Representation of a 4-Node



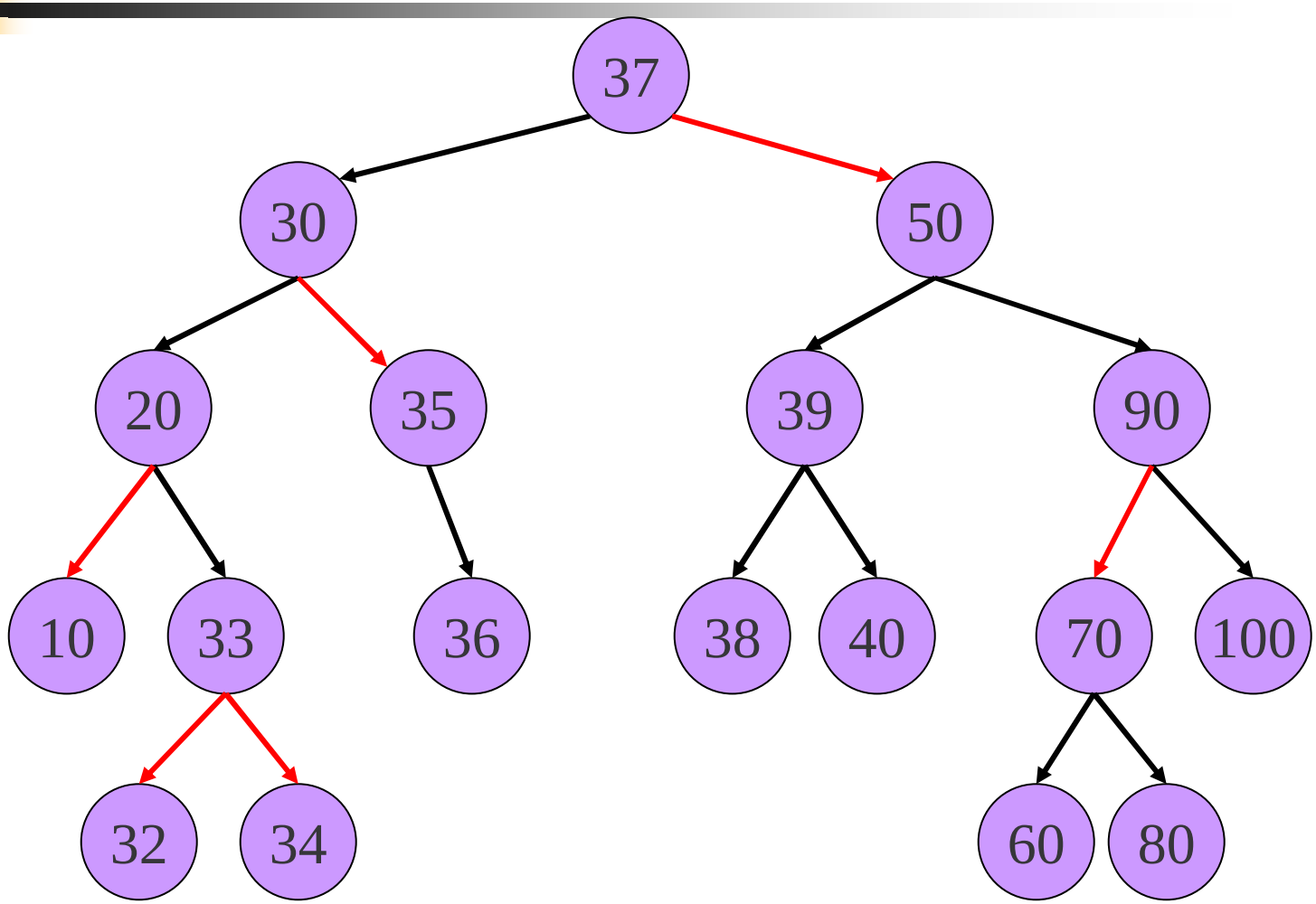
Red-Black Representation of a 3-Node



2-3-4 Tree



Equivalent Red-Black Tree



Red-Black Tree Node

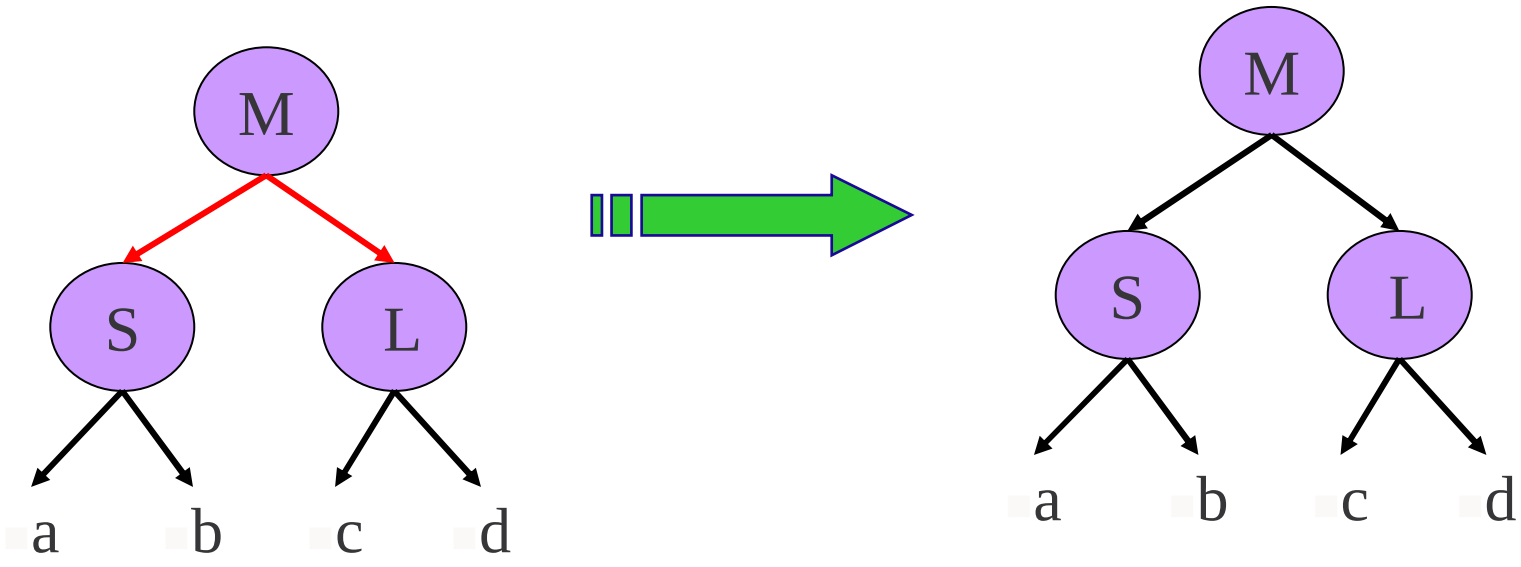
```
public class RBTreeNode {
    public static final int RED = 0;
    public static final int BLACK = 1;

    private KeyedItem item;
    private RBTreeNode leftChild;
    private RBTreeNode rightChild;
    private int leftColor;
    private int rightColor;
}
```

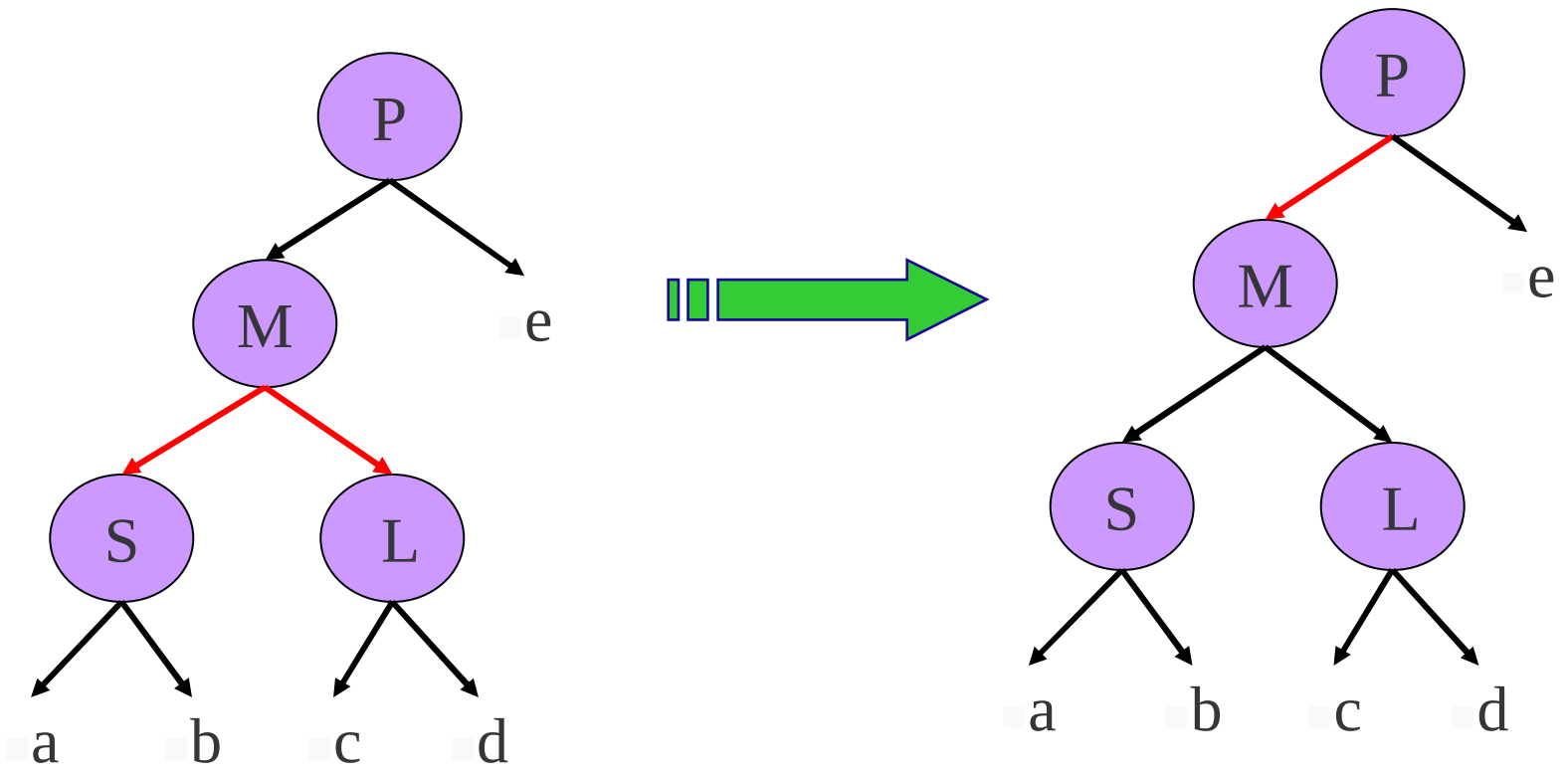
Searching and Traversing Red-Black Trees

- Red-Black trees are binary search trees
  - Just search them the same way you would any other binary search tree
- Inserting
  - Split 4-nodes on the way down by changing paired red child references to black
  - Insert into a leaf

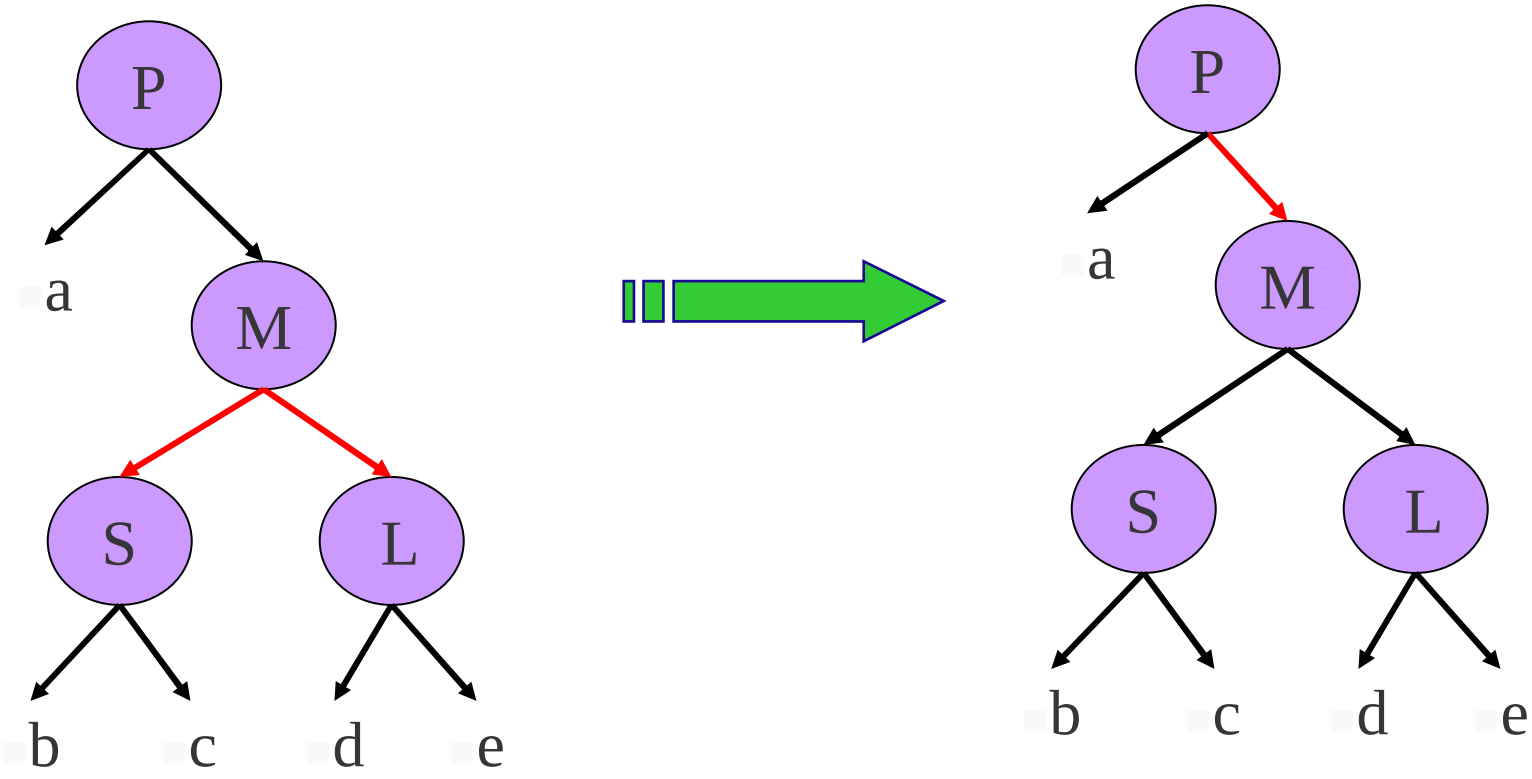
Splitting a 4-node root



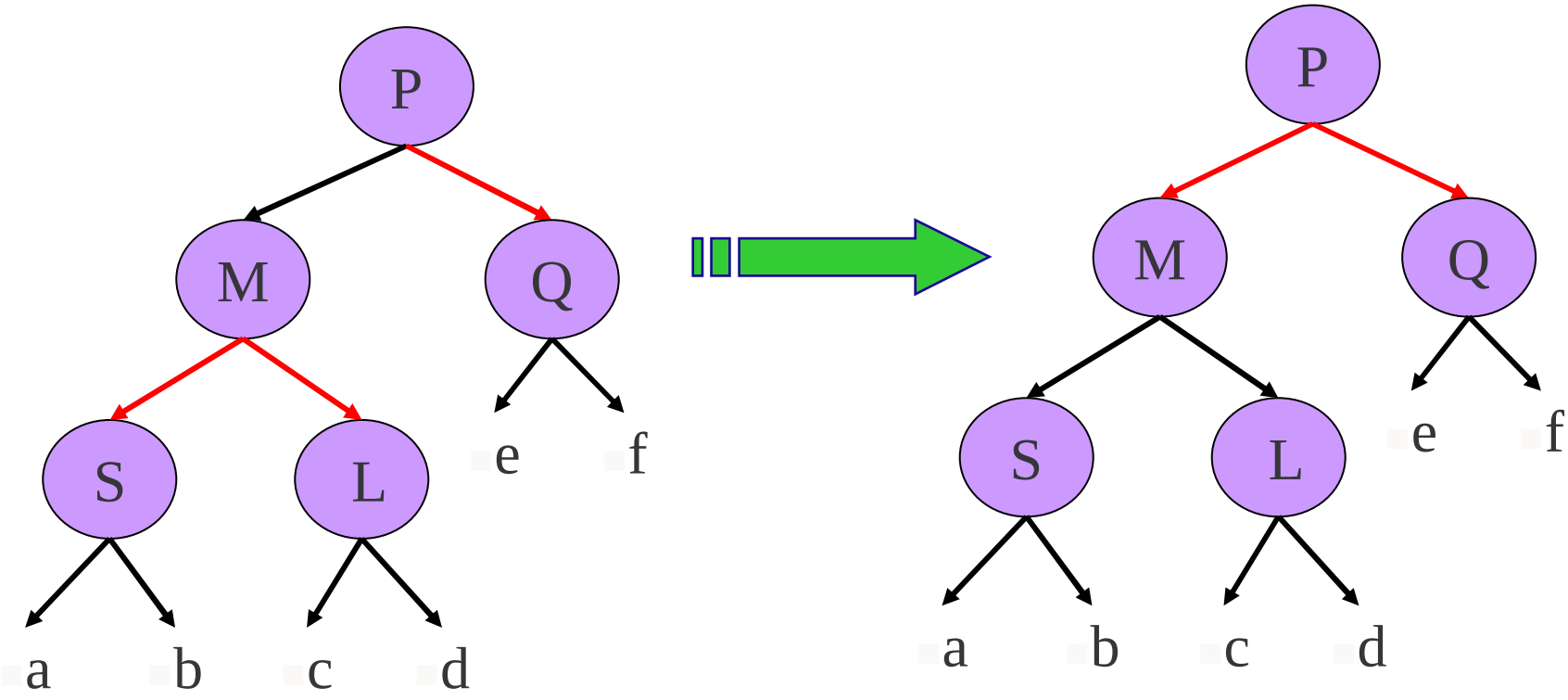
Splitting a 4-node



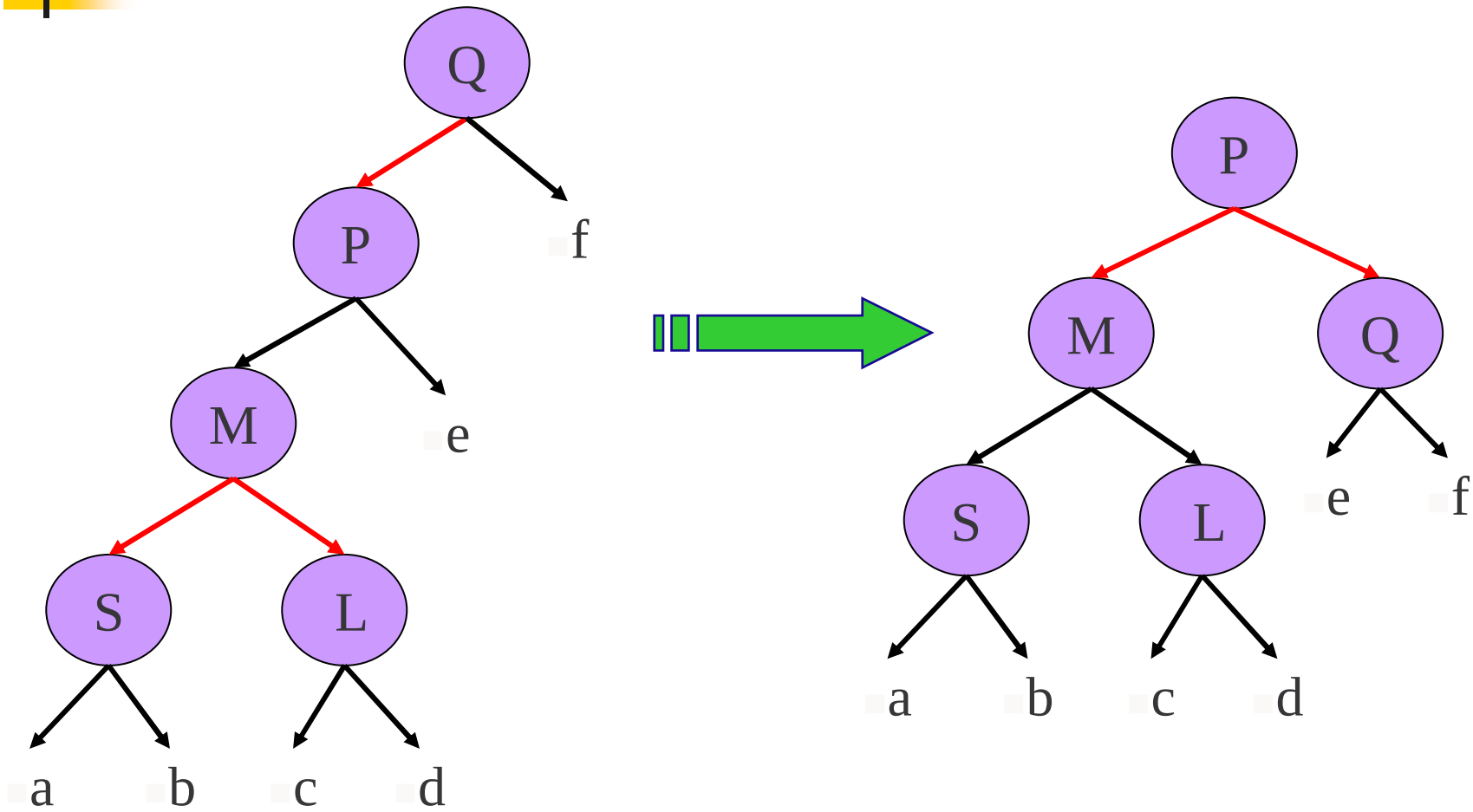
Splitting a 4-node



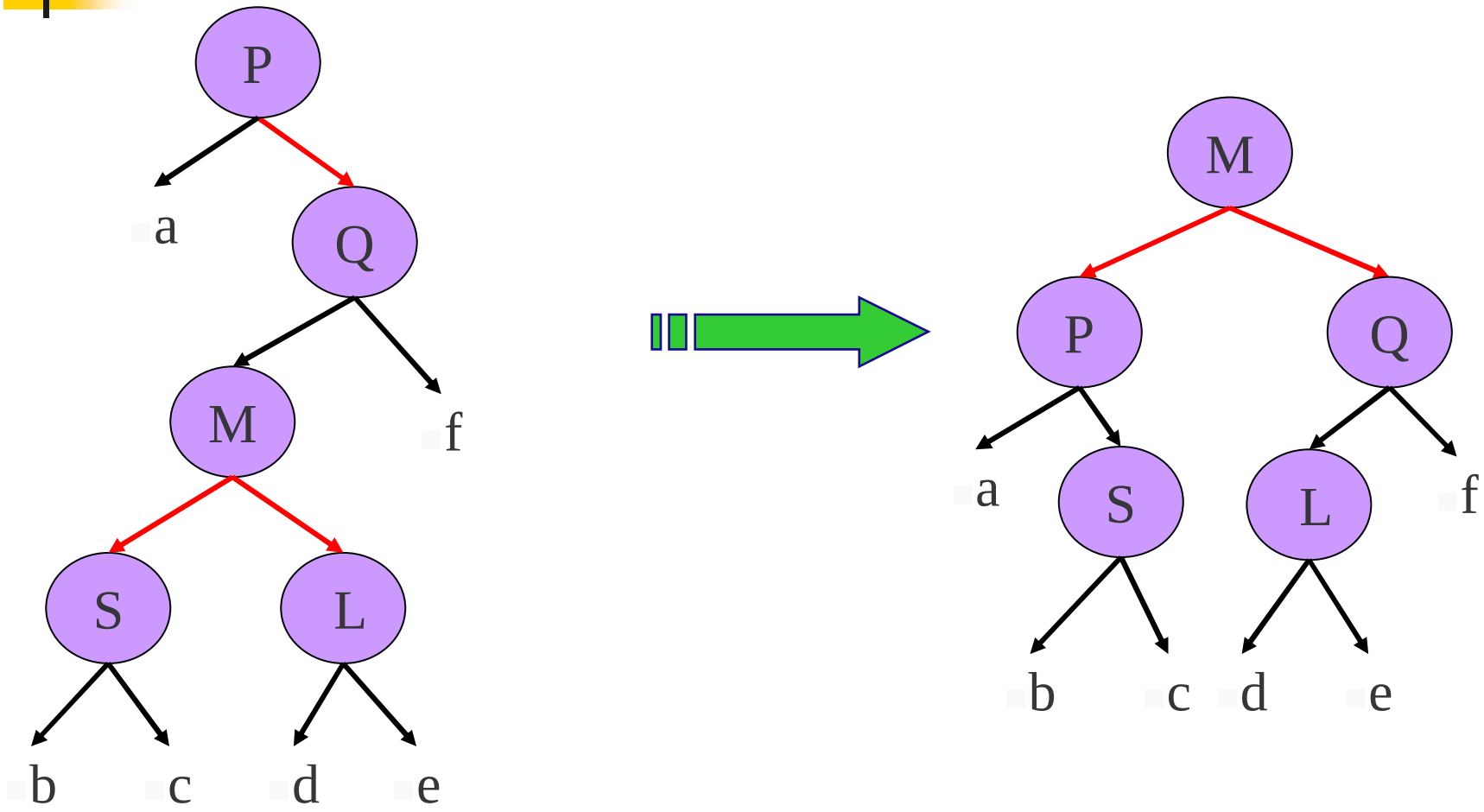
Splitting a 4-node



Splitting a 4-node



Splitting a 4-node

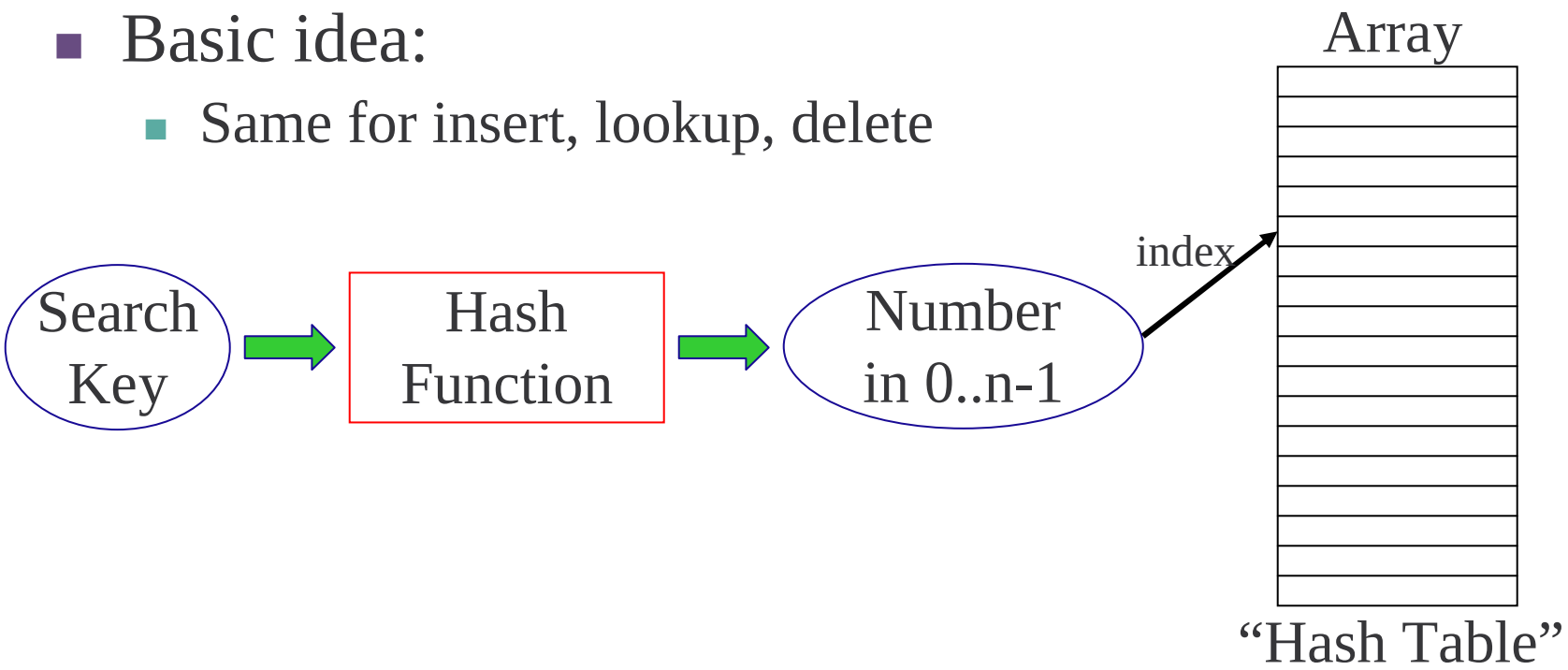


AVL Trees

- Insert in the appropriate spot
- Rotate as necessary to restore balance
  - Rotate your way back up the tree
- Every operation is O(logn)

Hashing

- Trees allow for efficient table operations, but
- What if you want O(1) behavior?
- What if time is much more critical than space?
- Basic idea:
  - Same for insert, lookup, delete



Issues

- How to create a hash function
  - Easy or difficult, depending upon the desired properties
- How large the array should be
  - Factors: how many items will be stored, how much memory you have, how fast you want the operations to be
- Static or dynamic
- Hash collisions: what if two input values produce the same hash value?

Hash Functions

- Goals
  - Fast, easy to compute
  - Distributes hash values evenly in the target range
- Possible functions for an *n*-entry hash table
  - $h(k)$ = random number in $0..n-1$
  - $h(k)$ = the sum of the digits in k
  - $h(k)$ = the first m digits of k, where $m = \log_{10}(n)$
  - $h(k) = k \times \mathrm{mod}(n)$
- Are these good or bad? Why?

Hash Functions on Strings

- Need to convert string to a number
- Possible solutions:
  - Add the binary representations of the characters
  - Concatenate the binary representations to get a very large number
    - Hello = $H \times 32^4 + e \times 32^3 + l \times 32^2 + l \times 32 + o$
  - Horner's rule: $(((H \times 32 + e) \times 32 + l) \times 32 + l) \times 32 + o$
  - This is a very big number, so
  - $(A \times B) \bmod n = (A \bmod n \times B \bmod n) \bmod n$
  - $(A + B) \bmod n = (A \bmod n + B \bmod n) \bmod n$
  - Apply the modulo operator early and often to keep the number small

Dealing with Hash Collisions

- Open addressing: Collision $\Rightarrow$ try to place the object in other locations in a predictable sequence
  - Linear probing: search starting from the current location
    - Issues: wrapping, slow, empty/deleted entries, clustering
  - Quadratic probing: search +1,4,9,16,25mod*n*
  - Double hashing: Use second hash function to determine the size of the steps:
    - $h_2(\mathrm{k}) \neq 0$
    - $h_2 \neq h_1$
    - Example: $h_1(k) = k \bmod 11$, $h_2(k) = 7 - (k \bmod 7)$
  - Note: size of steps and size of table must be relatively prime so that all entries get visited
    - Use prime size, step, etc.

Increase The Size of the Hash Table

- Increasing the actual size is infeasible
  - Everything must be rehashed and moved
- Chain-bucket hashing
  - Each entry in a hash table is a *bucket* into which multiple values may be added
  - Each bucket can be implemented as a *chain*, or linked list
- Now the size of the hash table is variable

Efficiency of Hashing

- Load factor

$$\alpha = \frac{Number\_of\_items}{Size\_of\_table}$$

- α is a measure of how full the table is
  - Small α ⇒ little chance of collision and low search time
  - Large α ⇒ high chance of collision and high search time

- Unsuccessful searches require more time than successful searches

Linear Probing

- Successful search

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

- Unsuccessful search

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

Quadratic Probing and Double Hashing

- Successful search

$$\frac{-\log_e(1-\alpha)}{\alpha}$$

- Unsuccessful search

$$\frac{1}{1-\alpha}$$

Chain-Bucket Hashing

- Successful search

$$1+\frac{\alpha}{2}$$

- Unsuccessful search

$$\alpha$$

How well does a hash function work?

- How fast is it to compute?
- How well does it scatter random data?
- How well does it scatter non-random data?
  - This can be very important
  - It is always possible to construct a worst case
- General principles:
  - The hash function should involve the whole search key
  - If a hash function involves a modulo operation, the base should be prime

Hashing vs. Binary Trees

- Hashing supports efficient insert and remove
- Hashing does not support efficient sorting
- Hashing does not support efficient range queries