Dynamic Access Binary Search Trees

Splay Trees* are self-adjusting binary search trees in which the shape of the tree is changed based upon the accesses performed upon the elements.

When an element of a splay tree is accessed the node with the element is percolated, (i.e. splayed), up to the root by applying AVL rotations. The more frequently accessed items will conglomerate near the top of the tree. This results in excellent performance when a subset of the tree elements are accessed much more frequently then other elements.

If all elements of the tree have equal access probability than a balanced binary search tree should be used.

Splay trees do NOT store any balancing information in the nodes.

*D. D. Sleator and R. E. Tarjan, 1985.

Amortized Analysis

Amortized analysis: the complexity is analyzed based upon a sequence of operations instead of single individual operations.

Amortized analysis takes into account the impact of operations upon following operations. Infrequently occurring costly individual operations may not have much effect on the overall performance if they are far outweighed by the less inexpensive frequently executed operations.

Formally, if "a sequence of M operations has total worst-case running time O(M *f*(N)), we say that the amortized running time is O(*f*(N))" [Weiss].

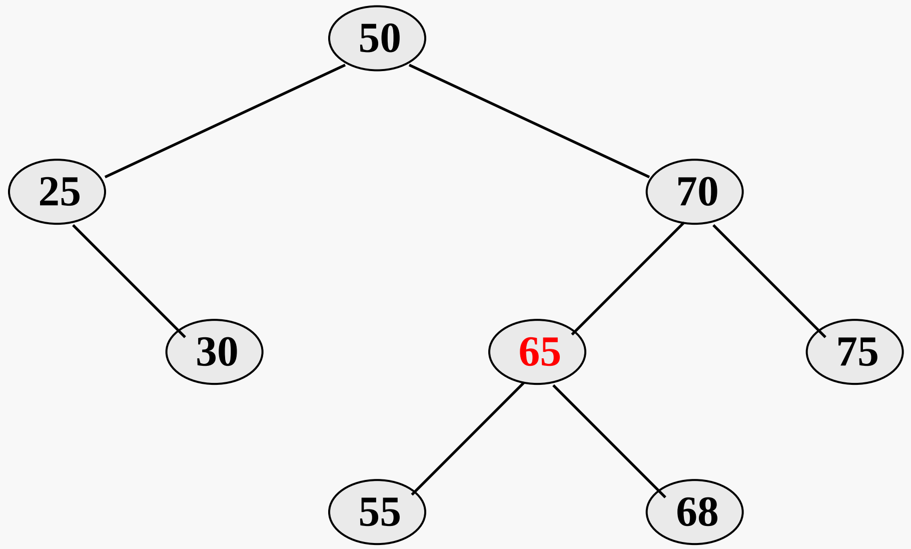
A series of splay tree M operations takes at most O(M log N). Thus a splay tree has an amortized cost of O(log N).

A single operation may take Θ(N) time, but be outweighed by other operations in a series. The amortized bound of O(log N) is not as strong of a bound as a worst-case O(log N) per operation bound.

*G. M. Adelson-Velskii and E. M. Landis, 1962.

Zig-Zag

Consider accessing the value 65 in the BST tree:



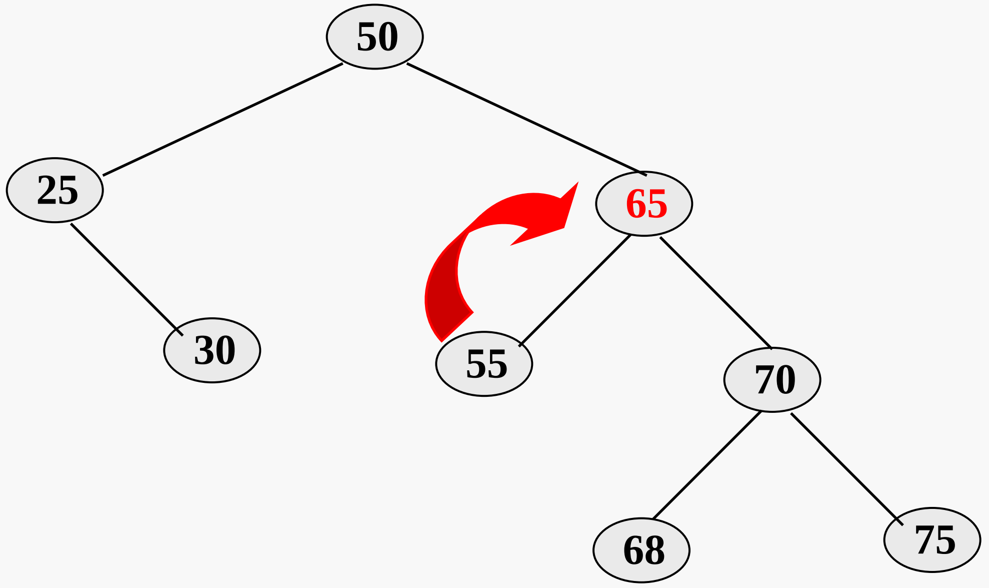
Zig-Zag:
left child of parent
right child of grandparent
→ right rotation(child, parent)
→ left rotation(child, grandparent)

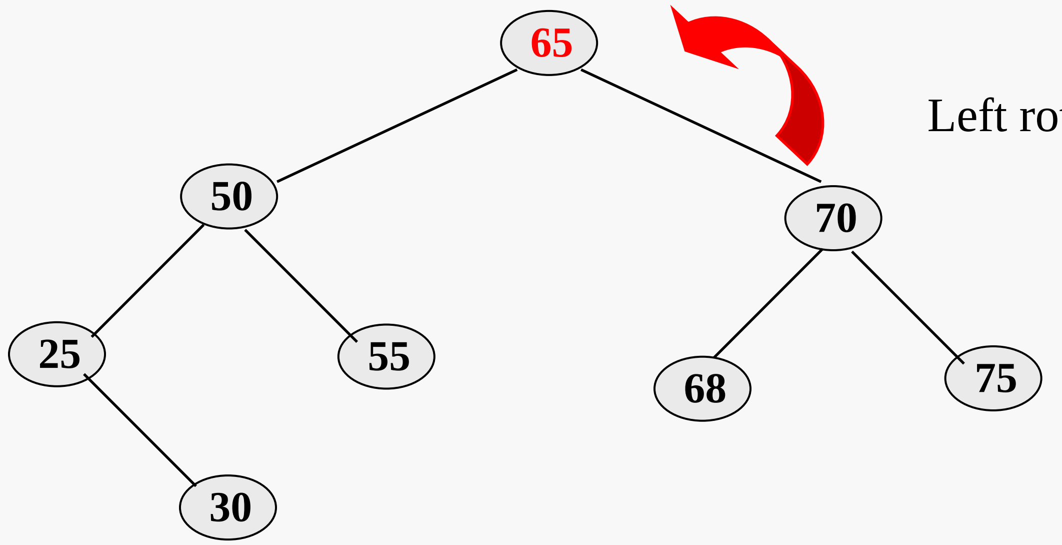
The **zig-zag case** occurs when the accessed item is the left child of its parent and the parent is a right child. The item is rotated with its parent, and then its grandparent. In this example the rotations are accomplished by a right rotation followed by a left rotation.

The symmetric case is **zag-zig** in which the accessed item is the right child of its parent and the parent is a left child.

Zig-Zag

Right rotation with parent.



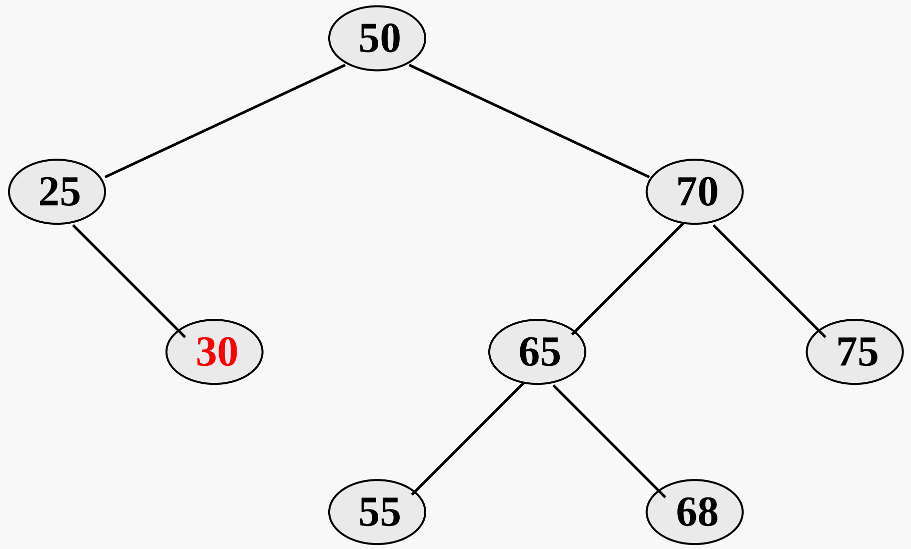


Left rotation with grandparent.

Splaying approximately halves the depth of other nodes on the access path.

Zag-Zig

Consider accessing the value 30 in the BST tree:

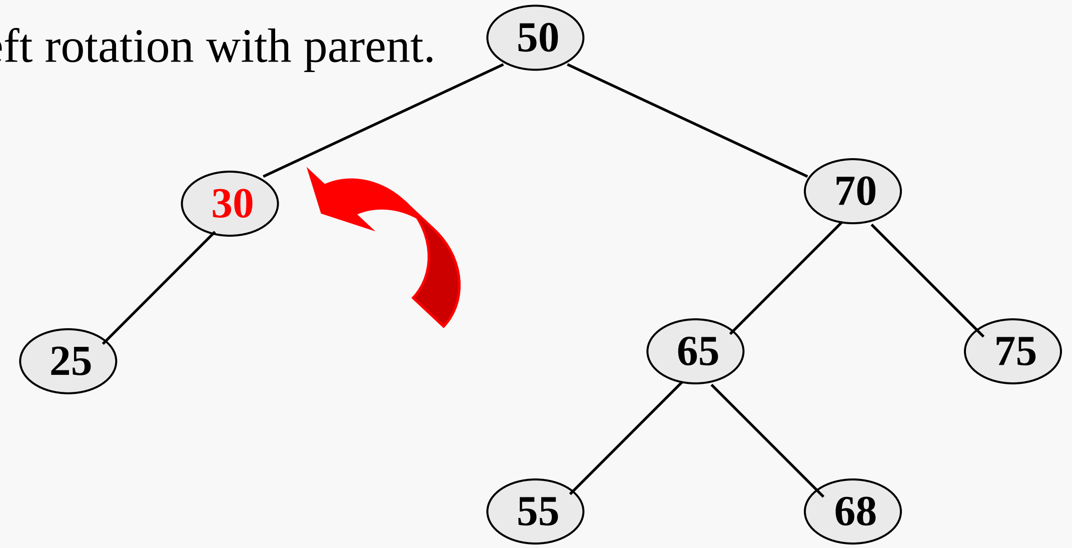


Zag-Zig:
right child of parent
left child of grandparent
→ left rotation(child, parent)
→ right rotation(child, grandparent)

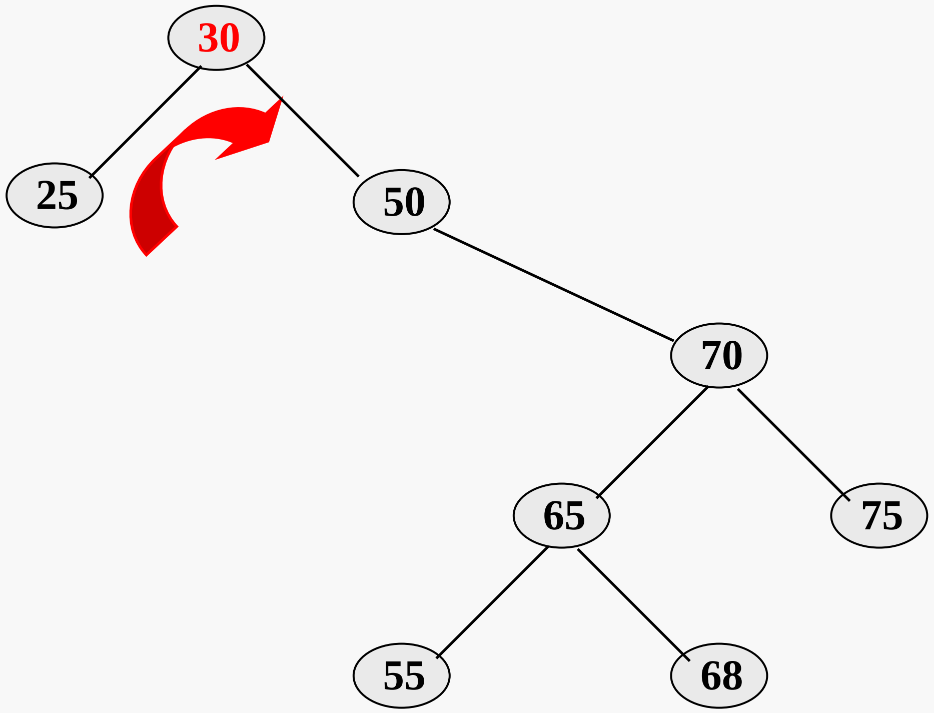
The **zag-zig case** occurs when the accessed item is the right child of its parent and the parent is a left child. The item is left rotated with its parent, and then right rotated with its grandparent.

Zag-Zig

Left rotation with parent.



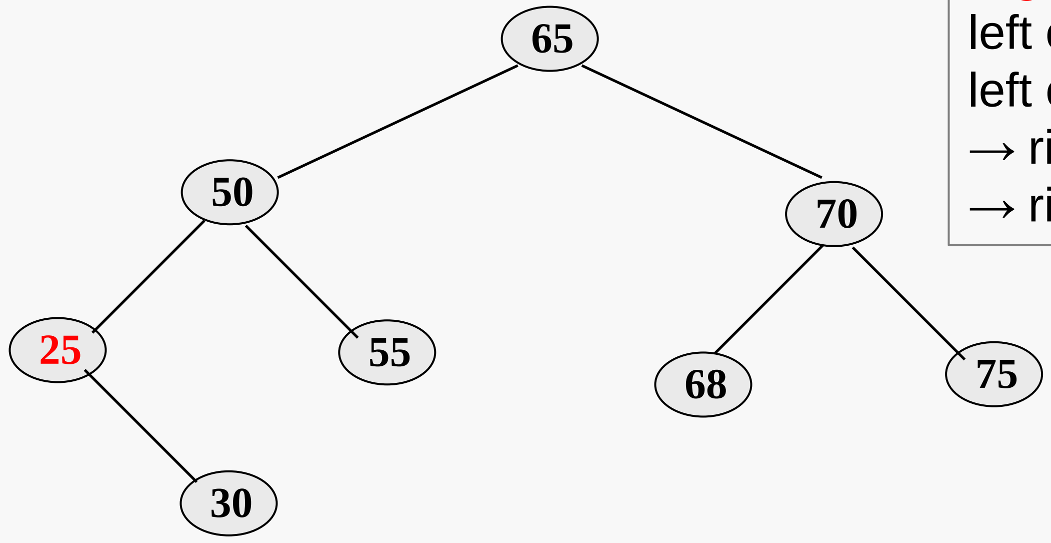
Right rotation with grandparent.



Splaying approximately halves the depth of other nodes on the access path.

Zig-Zig

Consider accessing the value 25 in the BST tree:



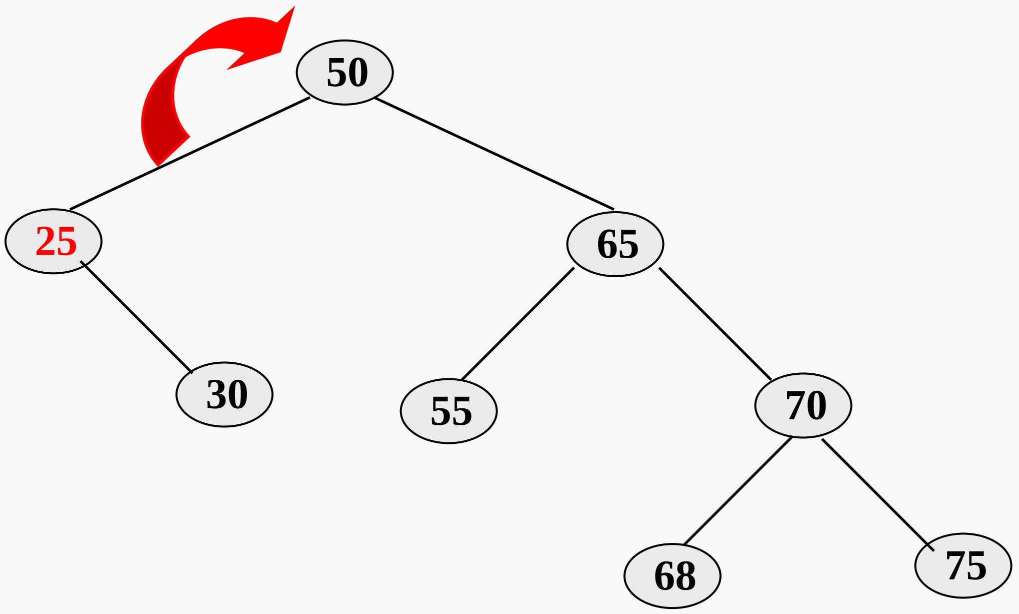
Zig-Zig:
left child of parent
left child of grandparent
→ right rotation(parent,grandparent)
→ right rotation(child, parent)

The **zig-zig case** occurs when the accessed item is the left child of its parent and the parent is also a left child. The parent is first rotated with the grandparent, and then the accessed items' node is rotated with its parent. In this example the rotations are accomplished by a right rotation followed by another right rotation.

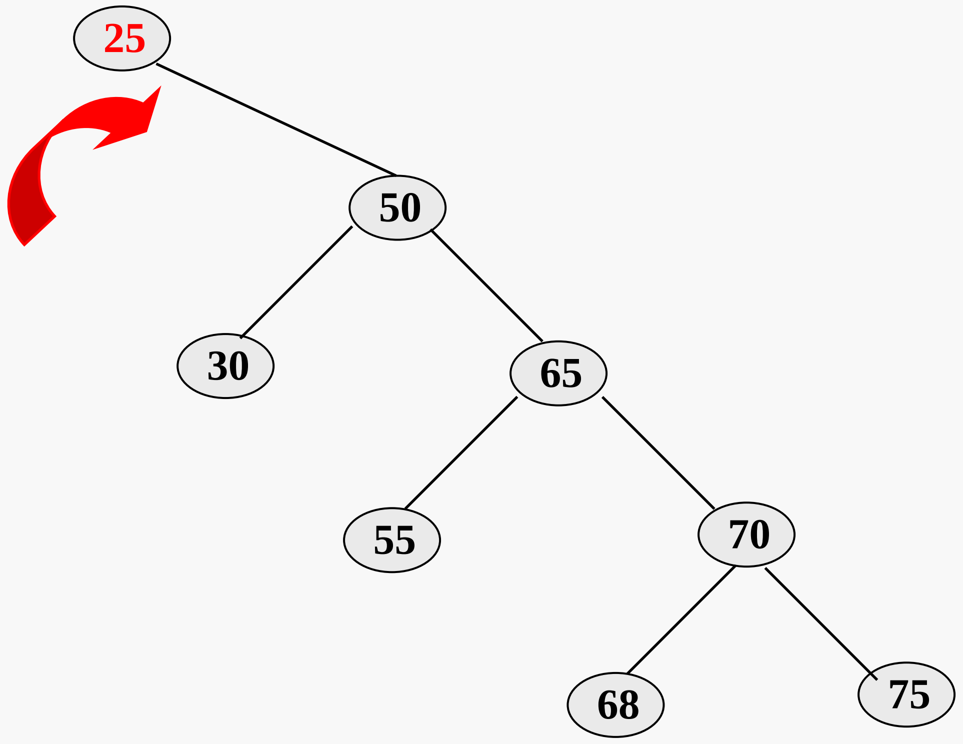
The symmetric case is **zag-zag** in the accessed item is the right child of its parent and the parent is also a right.

Zig-Zig

Right rotation between parent and grandparent.



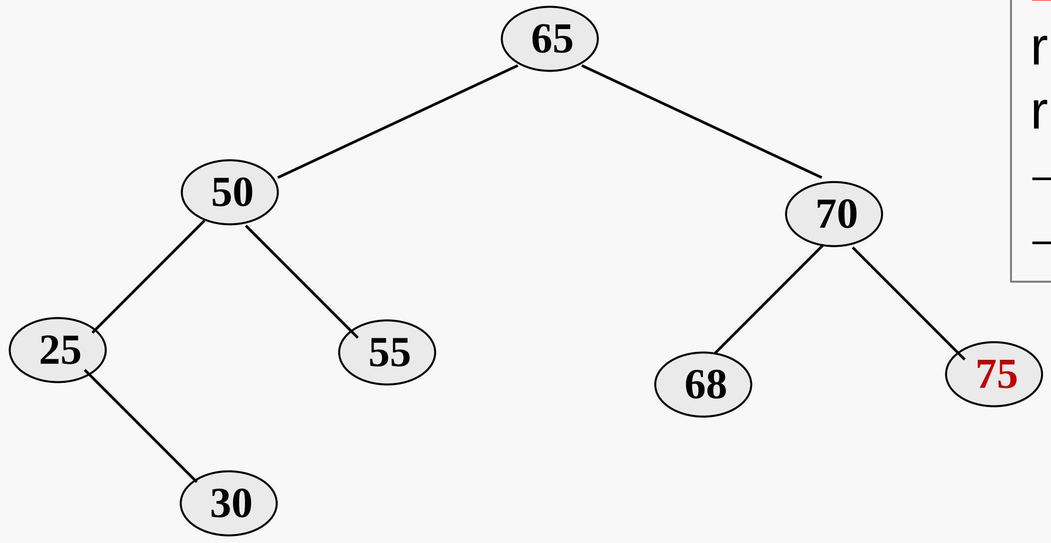
Right rotation with parent.



Splaying rotations continue until the root is reached. If the node is one level below the root only one rotation (zig/right or zag/left) is performed.

Zag-Zag

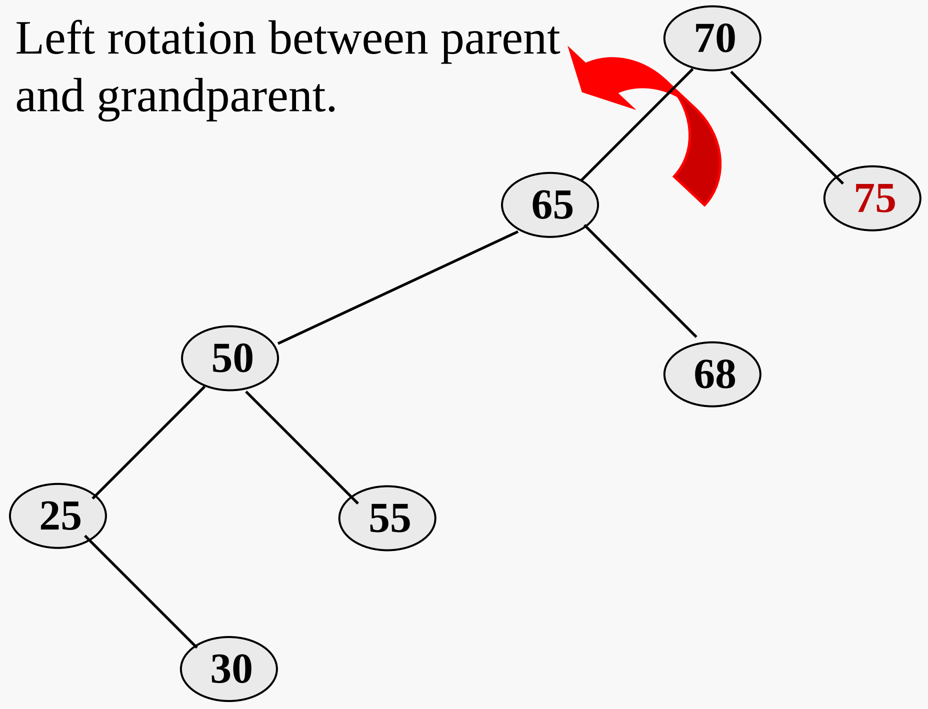
Consider accessing the value 75 in the BST tree:

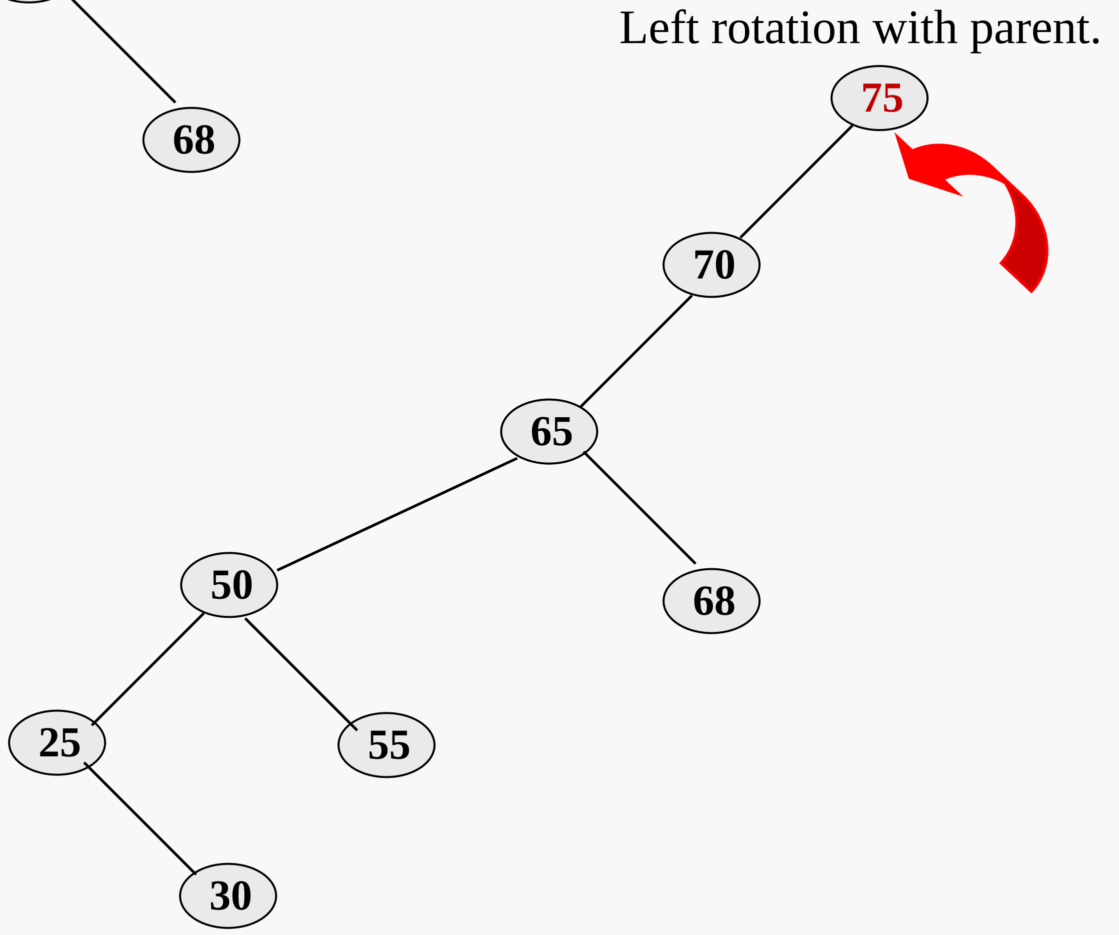


Zag-Zag:
right child of parent
right child of grandparent
→ left rotation(parent)
→ left rotation(grandparent)

The **zag-zag case** occurs when the accessed item is the right child of its parent and the parent is also a right child. The parent is first left rotated with the grandparent, and then the accessed items' node is left rotated with its parent.

Zag-Zag


Left rotation between parent and grandparent.
70
65
75
50
68
25
55
30


Left rotation with parent.
75
70
65
50
68
25
55
30

Modification

Deletion

- First access the node to force it to percolate to the root.
- Remove the root.
- Determine the largest item in the left subtree and rotate it to the root of the left subtree which results in the left subtree new root having no right child.
- The right subtree is then added as the right child of the new root.

Insertion

- Starts the same as a BST with the new node created at a leaf. The new leaf node is splayed until it becomes the root.

Splaying also occurs on unsuccessful searches. The last item on the access path is splayed in this case.

Pros & Cons

Pros:

- Simpler coding than AVL trees
- Average amortized cost is roughly the same as a balanced BST.
- No balance bookkeeping information need be stored.
- Rotations on long access paths tend to reduce search cost in the future.
- Most accesses occur near the root minimizing the number of rotations.

Cons:

- Height can degrade to be linear. (QTP: when would this occur?)
- Tree changes when only a search is performed. In a concurrent system multiple processes accessing the same splay tree can result in critical region problems.

Top-Down Splay Trees

Bottom-Up Splaying

The previous splay tree description is known as a "*bottom-up*" splay tree since the tree restructuring is performed back up the access towards the root.

Top-Down Splaying

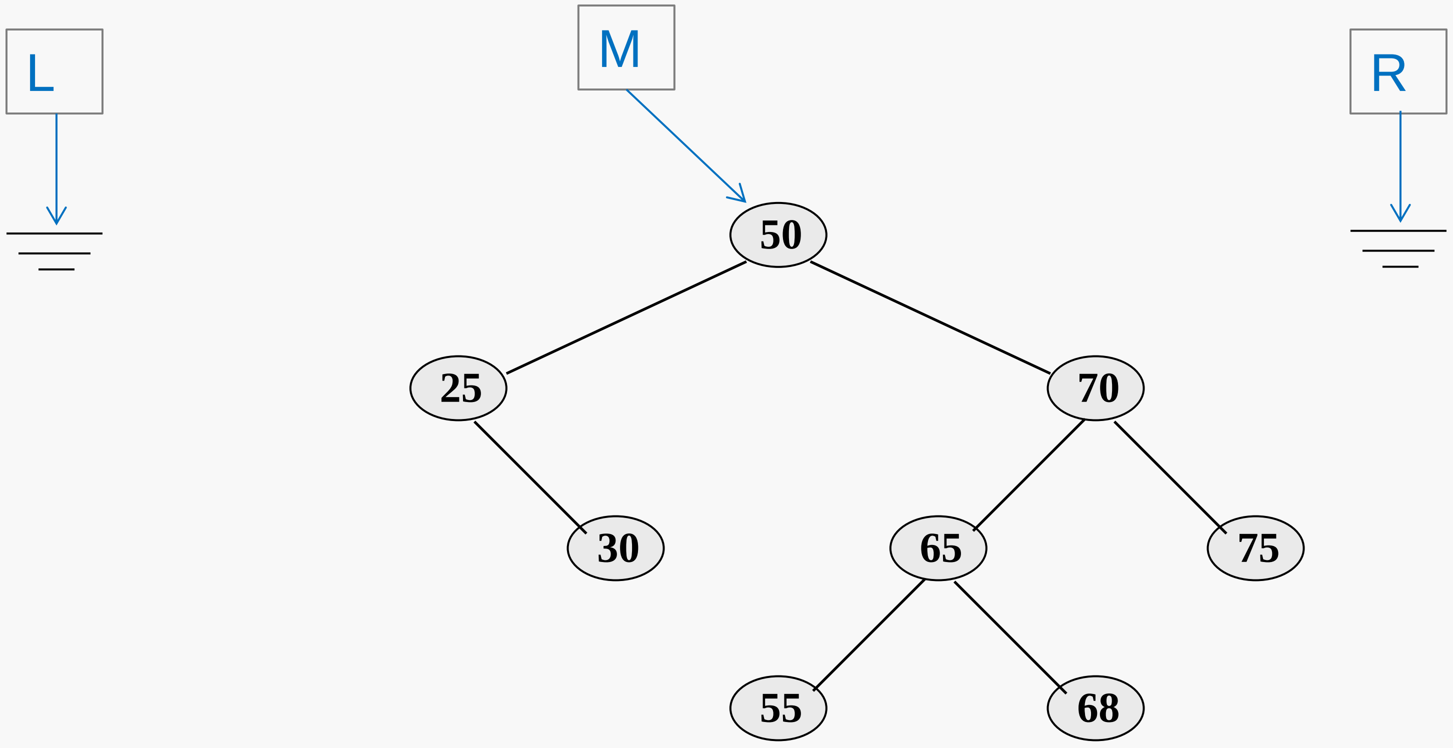
An alternate implementation involves performing the tree restructuring as one proceeds down the access path.

Alleviates the need to maintain the parent pointers using the recursive runtime stack.

As the search down the access path occurs three trees are maintained: left, middle and right. The left subtree holds nodes in the tree that are less than the target, but not in the middle tree. The right subtree holds nodes that are greater than the target, but not in the middle tree. The middle tree holds the current root of the access path which contains the target if it exists in the tree.

There are three cases for maintaining the left, right and middle trees: zig, zig-zig, zig-zag. (There are also three symmetric cases: zag, zag-zag, zag-zig.)

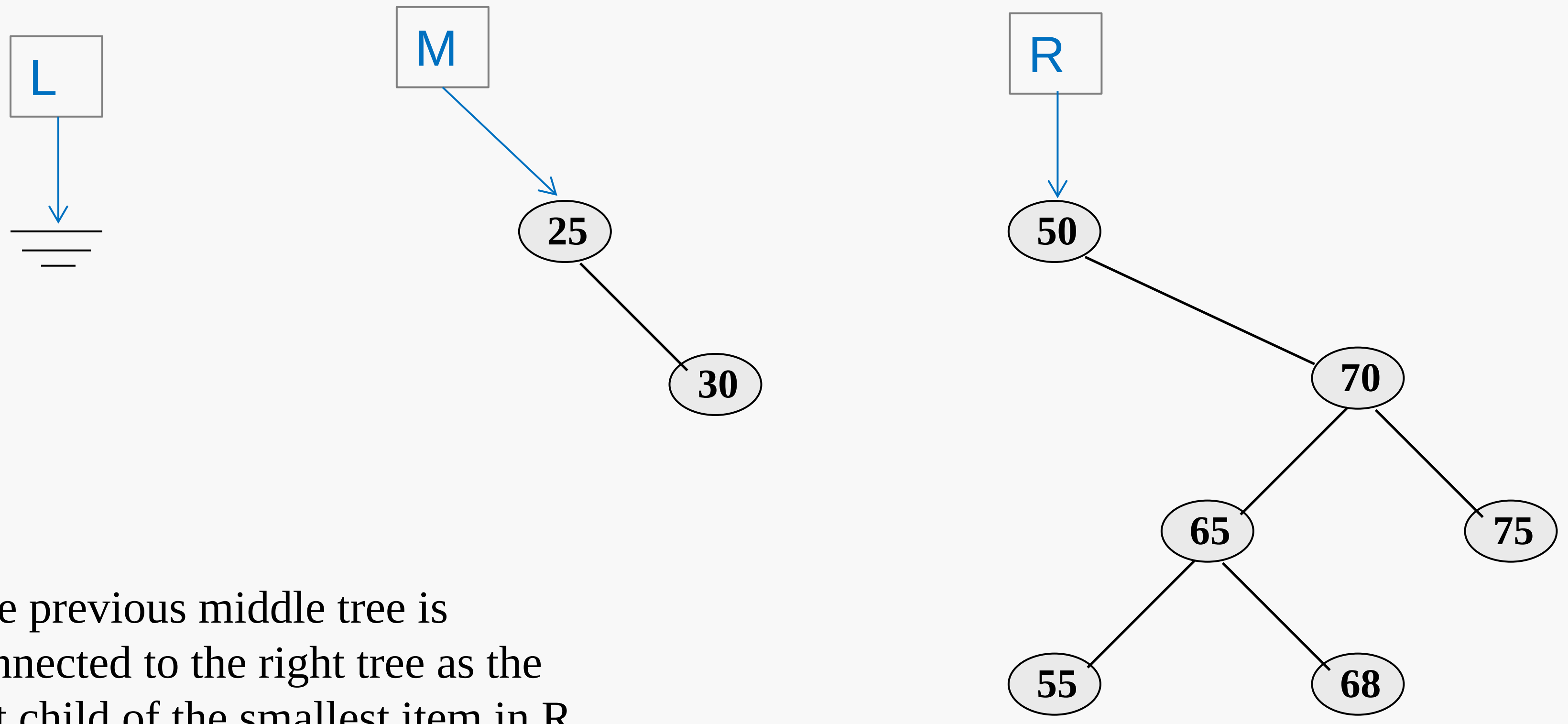
Top-Down Splaying: initialization



Consider accessing the value 20 in the BST tree:

Top-Down Splaying: zig

Zig case: when searching for an item smaller than the left subroot (in the middle tree) and the left subroot has no left child, but does have a right child.

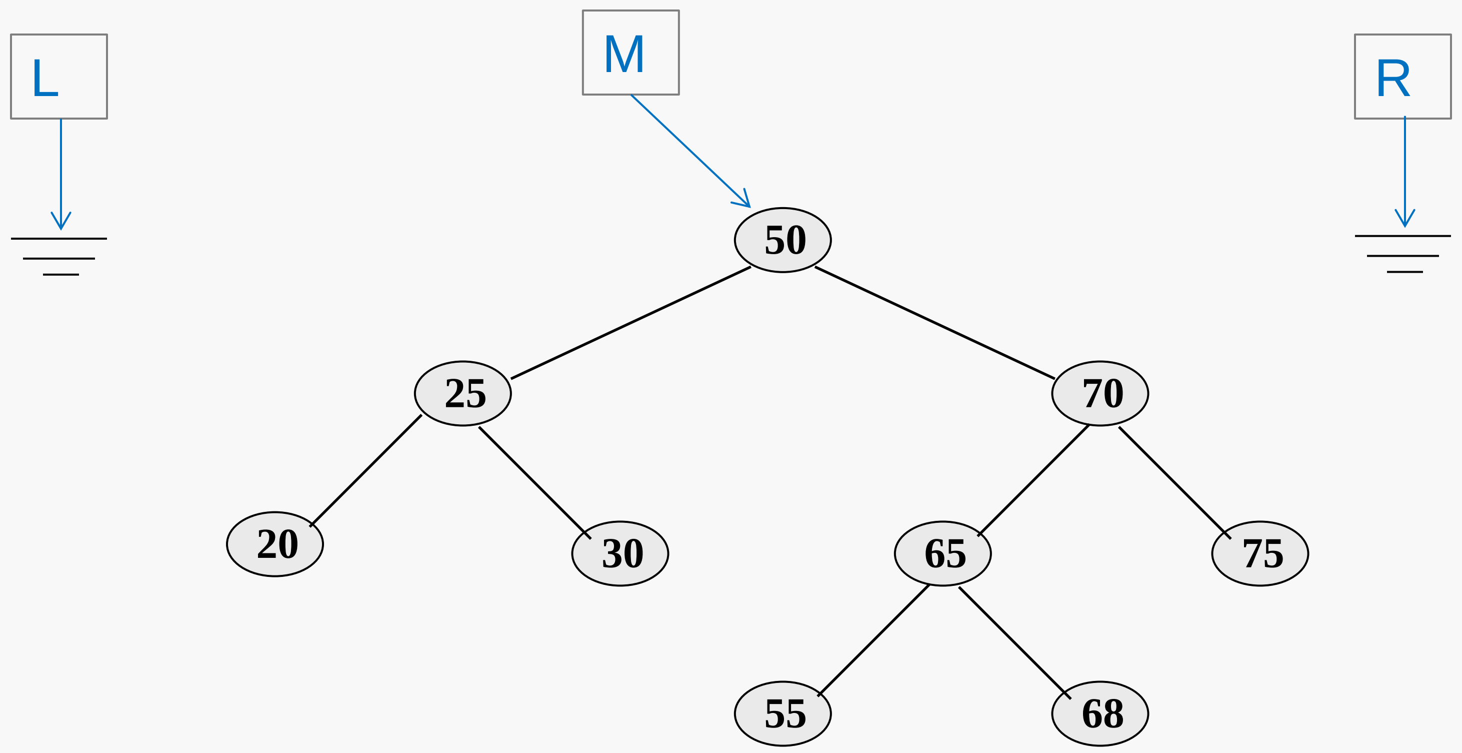


The previous middle tree is connected to the right tree as the left child of the smallest item in R, (or as the root if the right tree is empty as is the case here).

Top-Down Splaying: zig-zig

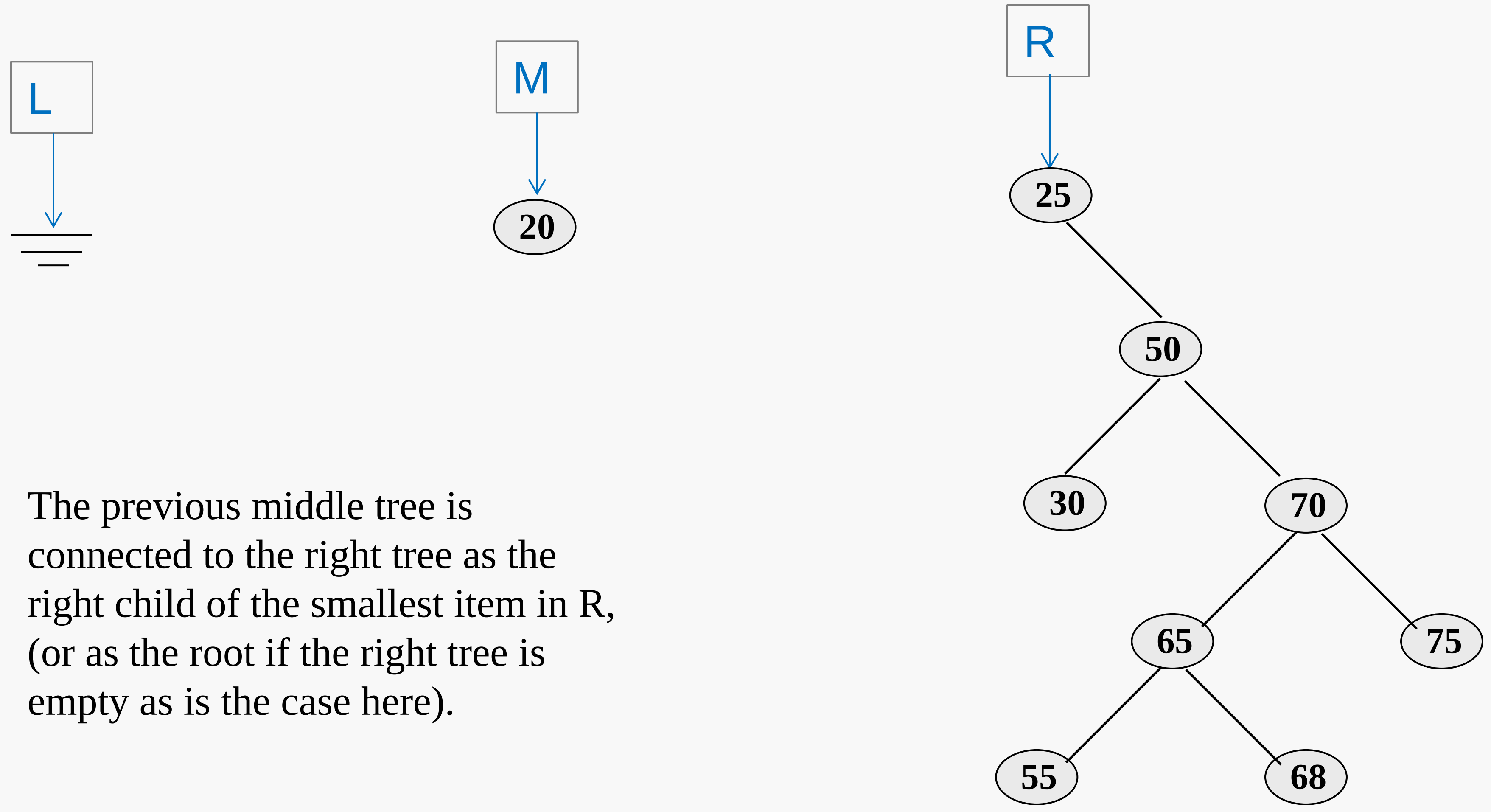
Zig-Zig case: when searching for an item smaller than the left subroot (in the middle tree) and the left subroot has a left child.

Consider an initial insertion of item 20 into the previous tree which yields the following the initial starting position:



Top-Down Splaying: zig-zig

Zig-Zig case: occurs when search determines target is smaller than the left subroot item in the middle tree and the middle tree left subroot is not null.

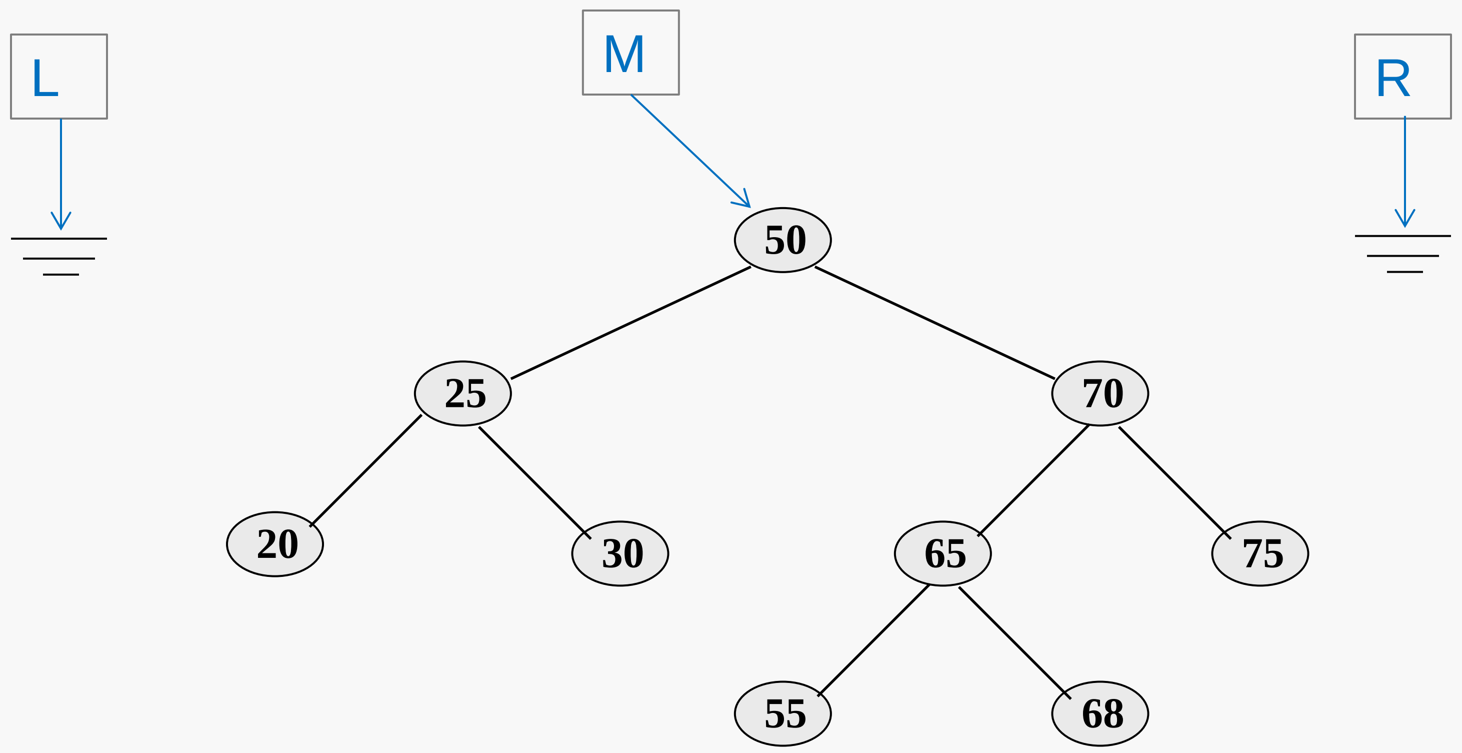


The previous middle tree is connected to the right tree as the right child of the smallest item in R, (or as the root if the right tree is empty as is the case here).

Top-Down Splaying: zig-zag

Zig-Zag case: when searching for an item greater than the left subroot (in the middle tree).

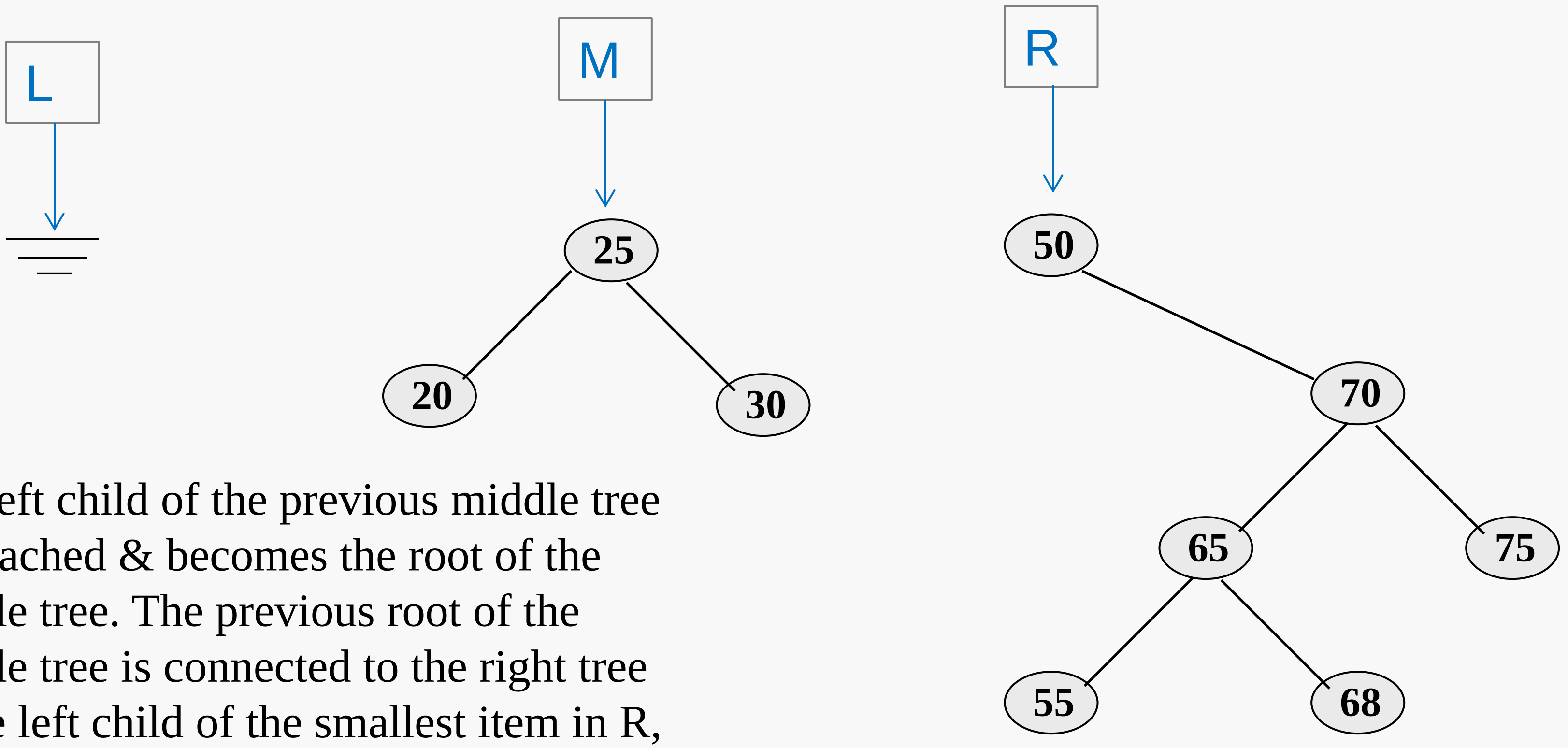
Consider a tree with the initial starting position:



Top-Down Splaying: zig-zag

Zig-Zag case: when searching for an item greater than the left subroot (in the middle tree).

Consider a tree with the initial starting position:

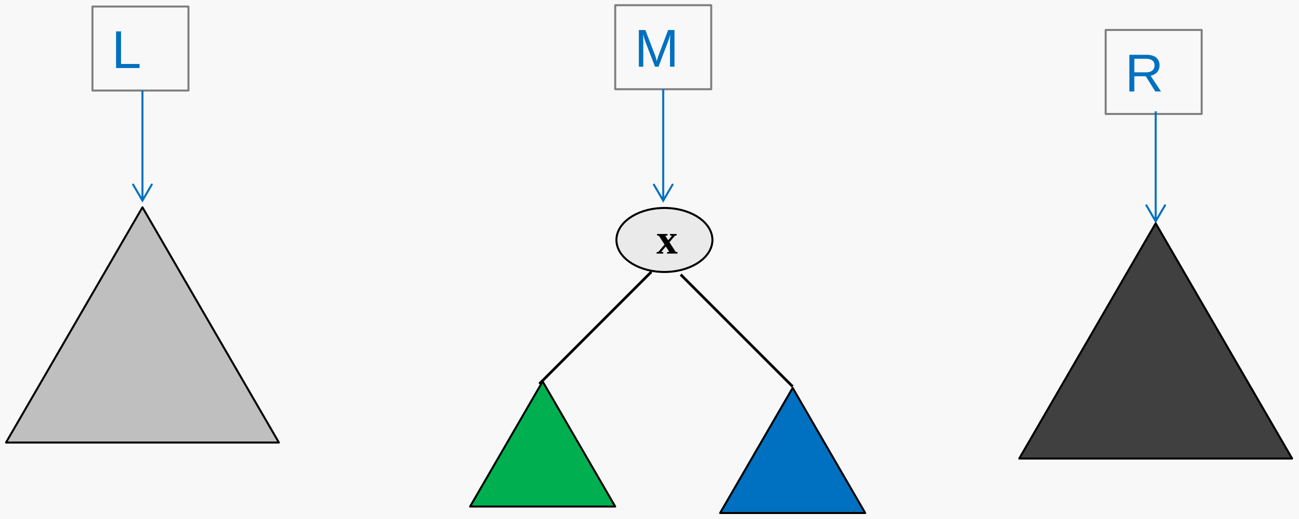


The left child of the previous middle tree is detached & becomes the root of the middle tree. The previous root of the middle tree is connected to the right tree as the left child of the smallest item in R, (or as the root if the right tree is empty as is the case here).

Top-Down Splaying: reconnection

Final Tree: once the last splaying dissection step has occurred the tree is reconnected.

Consider the possible state below that occurs from the last splaying step:



Top-Down Splaying: reconnection

Final Tree: reconnection results in the following state: