CS335 Class Notes

Luis F. Uceta

2020-01-28

Contents

1 Lecture 6: Trees **3**
1.1 Preliminaries . . . 3
1.2 Implementation of trees . . . 4
1.3 Tree traversals . . . 5
1.3.1 Depth-first search . . . 6
1.4 Binary trees . . . 9
1.4.1 Implementation . . . 9
1.4.2 Expression trees . . . 10
1.4.3 Constructing an expression tree . . . 10
1.5 Binary search trees (BTSs) . . . 11
1.5.1 Operations on a BTS . . . 12
1.5.2 Average-case analysis . . . 16
1.6 Balanced trees . . . 17
1.7 AVL trees . . . 17
1.7.1 Insertion cases . . . 18

1 Lecture 6: Trees

1.1 Preliminaries

A **tree** is a widely used abstract data type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Trees (more specifically binary trees) are really important so a certain terminoly has developed for them:

- A **node** is a structure which may contain a value or condition, or represent a separate data structure.
- An **edge** is connection between one node and another.
- A **path** is a sequence of nodes and edges connecting a node with a descendant.
- An **ancestor** is a node reachable by repeated proceeding from child to parent. A **descendant** is a node reachable by repeated proceeding from parent to child. For example, if there's a path from node u to node v, then u is an ancestor of v and v is a descendant of u. If $u \neq v$, then u is a **proper ancestor** of v and v is a **proper descendant** of u.
- The **root** r of a tree is its top node known as the prime **ancestor**.
- The root of each subtree is said to be a **child** of the root r, and r is the **parent** of each subtree root.
- Two nodes u and v are **siblings** if they have the same parent.
- The **subtree** of a node, u, is the tree rooted at u and contains all of u's descendants.
- The **depth** of a node, u, is the length of the path from u to the root of the tree. This translates into the number of edges from node to the tree's root node. Thus, a root node will have a depth 0.
- The **height** of a node, u, is the length of the longest path from u to one of its descendants. It's defined as $h = max(height(u.left), height(u.right)) + 1$. Thus, the height of a tree is equal to the height of the root. Since leaves has no children, all leaves have height 0.
- A node, u, is a **leaf** if it has no children.

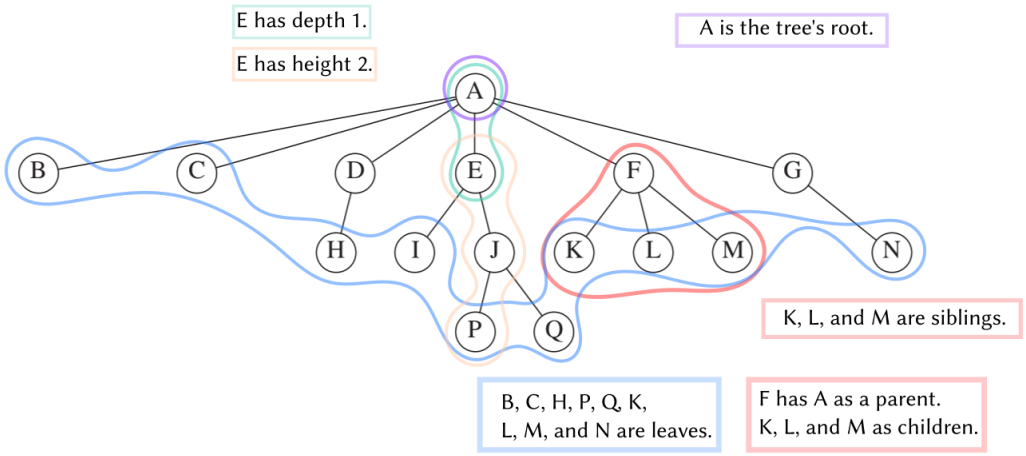


Figure 1: A tree

An m-ary tree is a rooted tree in which each node has no more than m children. When $m = 2$, the tree is known as a **binary tree** (i.e., each node has at most two children, the left child and the right child). When $m = 3$, the tree is a **ternary tree**.

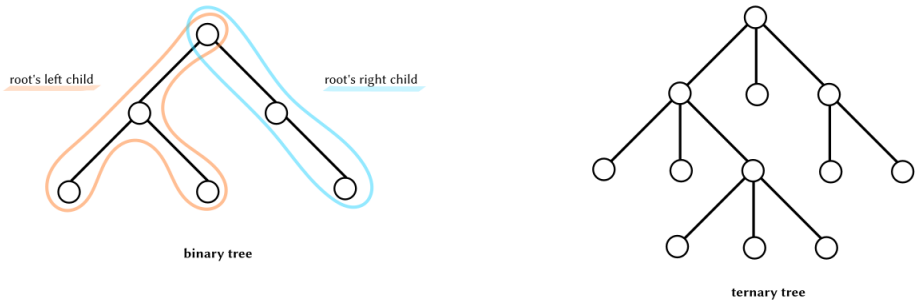
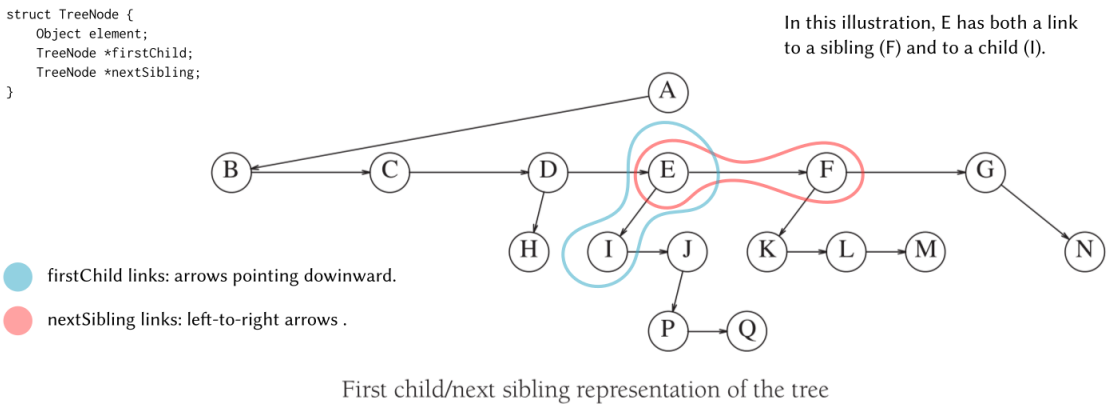


Figure 2: Binary and ternary tree

1.2 Implementation of trees

An obvious way to implement a tree would be to have in each node, besides its data, a link to each child of the node. However, the number of children per node can vary and is not known in advance which might translate into too much wasted space. The solution is simple: Keep the children of each in a linked list of nodes. In this way, each node can have as many children as the tree it composes allows.

```
template<typename Object>
struct TreeNode {
    Object element;
    TreeNode* firstChild;
    TreeNode* nextSibling;
}
```



First child/next sibling representation of the tree

Figure 3: First child/next-sibling representation

1.3 Tree traversals

A **tree traversal** refers to the process of visiting each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

Trees can be traversed either:

- by deepening the search as much as possible on each child before going to the next sibling. This is known as **depth-first search** (DFS).

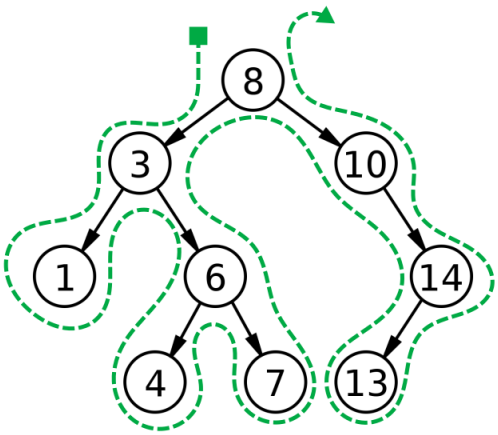


Figure 4: Depth-first search

- by visiting every node on a level before going to a lower level. This is known as **breadth-first search** (BFS).

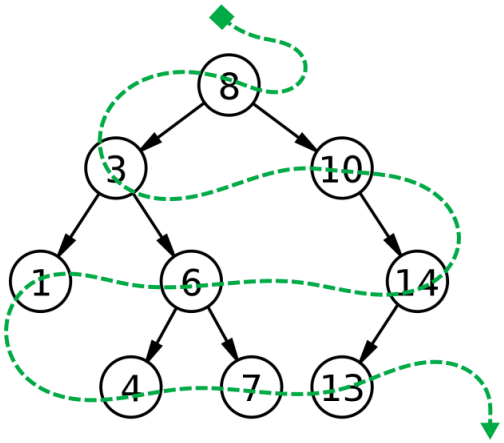


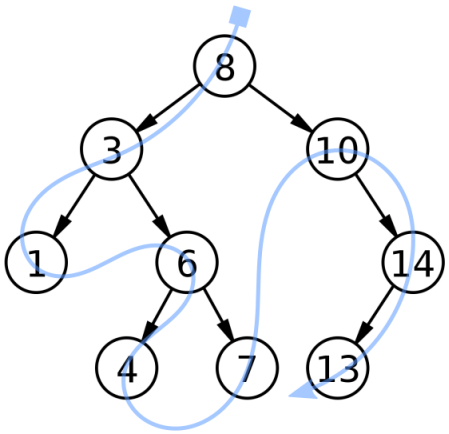
Figure 5: Breadth-first search

1.3.1 Depth-first search

Depending on when the work at a node is performed, a traversal of this type can be classified into:

- **pre-order traversal**. In a pre-order traversal, work at a node is performed before (*pre*) its children are processed.

> 1. Check if the current node is empty or null.
> 2. Display the data part of the root (or current node).
> 3. Traverse the left subtree by recursively calling the pre-order function.
> 4. Traverse the right subtree by recursively calling the pre-order function.



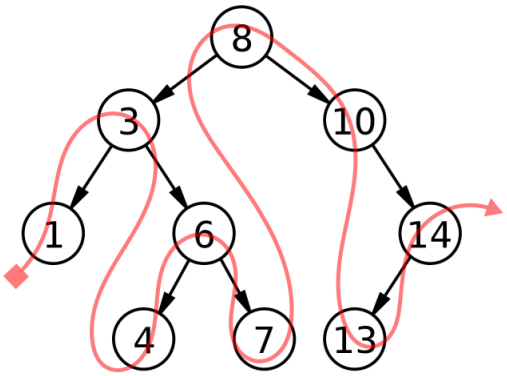

```
preorder(node)
    if (node == null)
        return
    visit(node)
    preorder(node.left)
    preorder(node.right)
```

8, 3, 1, 6, 4, 7, 10, 14, 13

Figure 6: Pre-order traversal

- **in-order traversal**. In an in-order traversal, work at a node is performed after its left child is visited, followed by the right child.

> 1. Check if the current node is empty or null.
> 2. Traverse the left subtree by recursively calling the in-order function.
> 3. Display the data part of the root (or current node).
> 4. Traverse the right subtree by recursively calling the in-order function.



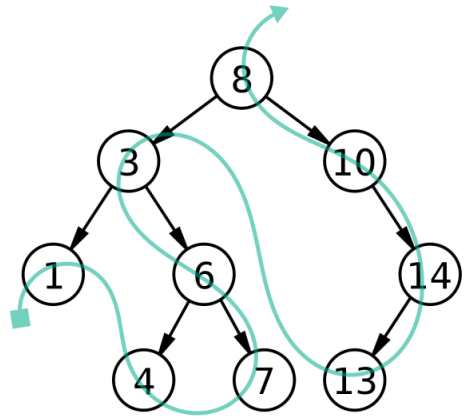

```
inorder(node)
    if (node == null)
        return
    inorder(node.left)
    visit(node)
    inorder(node.right)
```

1, 3, 4, 6, 7, 8, 10, 13, 14

Figure 7: In-order traversal

- **post-order traversal**. In a post-order traversal, work at a node is performed after (*post*) its children are evaluated.

> 1. Check if the current node is empty or null.
> 2. Traverse the left subtree by recursively calling the post-order function.
> 3. Traverse the right subtree by recursively calling the post-order function.
> 4. Display the data part of the root (or current node).




```
postorder(node)
    if (node == null)
        return
    postorder(node.left)
    postorder(node.right)
    visit(node)
```

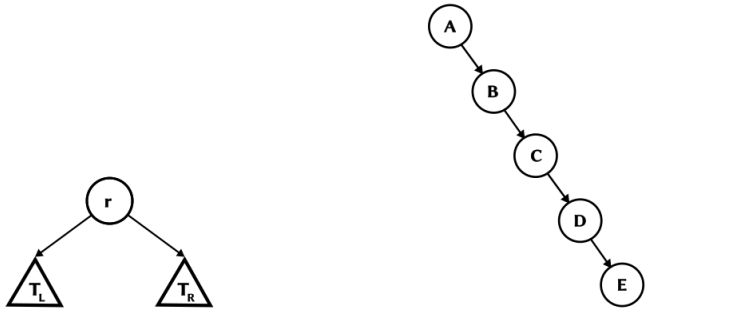
1, 4, 7, 6, 3, 13, 14, 10, 8

Figure 8: Post-order traversal

1.4 Binary trees

As stated earlier, a **binary tree** is a tree in which no node can have more than two children.

A property of a binary tree that is sometimes important is that the depth of an average binary tree is considerably smaller than N. An analysis shows that the average depth is $O(\sqrt{N})$, and that for a special type of binary tree, namely the **binary search tree**, the average value of the depth is $O(logN)$. Unfortunately, the depth can be as large as $N - 1$. In its worst case, we end up with a linked list (which is also a type of tree).



Generic binary tree

Worst-case binary tree (linked list)

Figure 9: Generic binary tree and linked list

1.4.1 Implementation

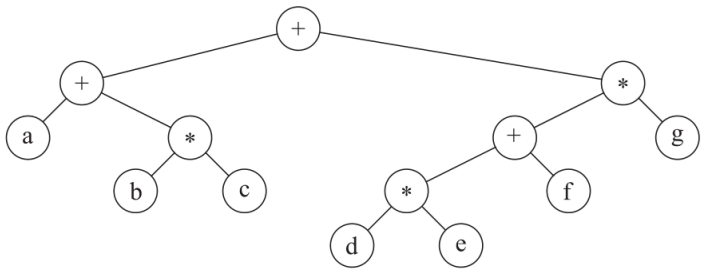
Given that a binary tree has at most two children per node, we can keep direct links to them. Thus, a binary node is just a structure consisting of the data plus two pointers to current node's left and right nodes. For example:

```
struct BinaryNode {
    Object element;    // data in the node
    BinaryNode* left;  // left child
    BinaryNode* right; // right children
}
```

Binary trees have many important uses not associated with searching. One of the principal uses of binary trees is in the area of compiler design (e.g., expression trees).

1.4.2 Expression trees

A **binary expression tree** is a specific kind of a binary tree used to represent expressions. The leaves of an expression are **operands**, such as constants or variable names, and the other nodes contain **operators**. We can evaluated an expression, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.



Expression tree for `(a + b * c) + ((d * e + f ) * g)`

Figure 10: Expression tree

- To produce an (overly parenthesized) *infix expression*, recursively produce parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This is an **in-order traversal**.
- To produce a *postfix expression*, recursively print out the left subtree, the right subtree, and then the operator. This is an **post-order traversal**.
- To produce a *prefix expression*, print out the operator first and then recursively print out the left and right subtrees. This is an **pre-order traversal**.

1.4.3 Constructing an expression tree

This algorithm describes the steps to convert a postfix expression into an expression tree:

```
ConvertPostfixToTree( EXPR ):
    stack ← []
    for EXPR → token:
        if token = operand:
            T ← Tree(root ← operand)
            stack.push(T)
        if token = operator:
            T1 ← stack.pop()
            T2 ← stack.pop()
            T ←  Tree(root ← operator, left ← T1, right ← T2)
            stack.push(T)
    return stack
```

Implementation: implementations/ExpressionTree

1.5 Binary search trees (BTSs)

A **binary search tree** is a special type of binary trees with the following property:

> For every node, X, in the tree, the values of all the items in its left subtree are smaller than the item in X, and the value of all the items in its right are larger than the item in X.

This implies that all the elements in the tree can be ordered in some consistent manner.

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that *each comparison allows the operations to skip about half of the tree*, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.

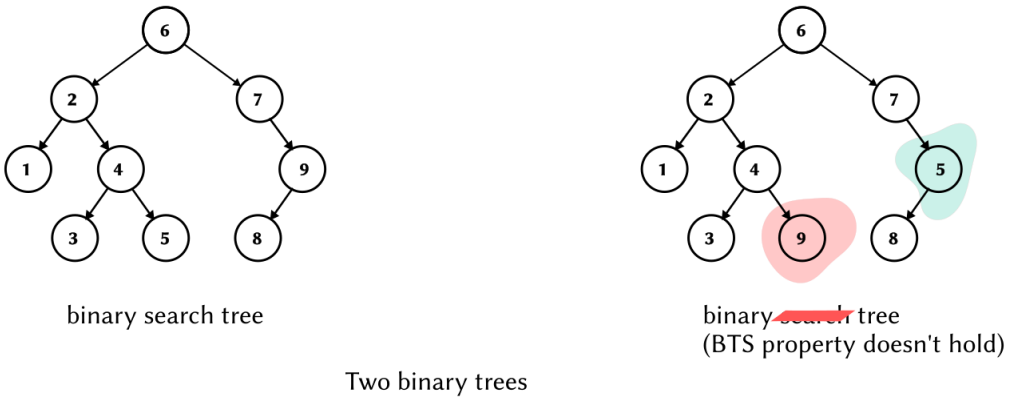


Figure 11: Binary search tree vs Non-binary search tree

Complete implementation: implementations/BinarySearchTree

1.5.1 Operations on a BTS

contains For this operation, we're required to return **true** if there is a node in the tree T that has item X. Otherwise, return **false**. If T is empty, then we can return **false** right away. Otherwise, if the item stored at T is X, we can return **true**. Otherwise, we can make a recursive call on a subtree of X, either left or right, depending on the relationship between X and the item stored in T. It's implied that the objects being compared have implemented the used relational operators appropriately.

```
Input: X is an item to search for and T is the node that roots
the subtree.
Output: Return true or false depending on whether the tree
contains the node storing the item.

contains( X, T ):
    if T = null:
        return false
    else if X < T.element():
        return contains X, T.left()
    else if X > T.element():
        return contains X, T.right()
    else:
        return true # Match
```

findMin and findMax These operations can either return the node (or its value) containing the smallest and largest element in the tree, respectively. To perform a `findMin`, start at the root and go

left as long as there's a left child. The stopping point is the smallest element. The `findMax` routine is similar, except that branching is to the right child.

Recursive algorithm of `findMin`:

```
Input: the root of the tree.
Output: the smallest element in the tree.


findMin( T ):
    if T = null:
        return T
    if T.left() = null:
        return T;
    return findMin T.left()
```

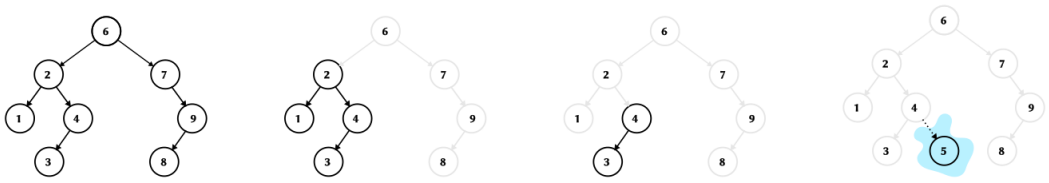
Iterative algorithm of `findMax`:

```
Input: the root of the tree.
Output: the largest element in the tree.

findMax( T ):
    if T ≠ null:
        until T.left() = null:
            T ← T.left()
    return T
```

insert To insert X into tree T, proceed down the tree as you would with the `contains` operation. If X is found, do nothing. Otherwise, insert X at the last spot on the path traversed.

Duplicates can be handled by keeping an extra field in the node object indicating the frequency of ocurrence for a particular element X. This adds some space extra space to the entire tree but is better than putting duplicates in the tree (which tends to make the tree very deep).

```
Input: X is an item to inserted in the tree T.
Output: void.

insert( X, T ):
    if T ← null:
        T ← BinaryNode(X, null, null)
    else if X < T.element()
        insert X, T.left()
    else if X > T.element()
        insert X, T.right()
    else:
        # duplicate; do nothing
```



Progression of inserting 5 into the tree

Figure 12: Progression of an insertion

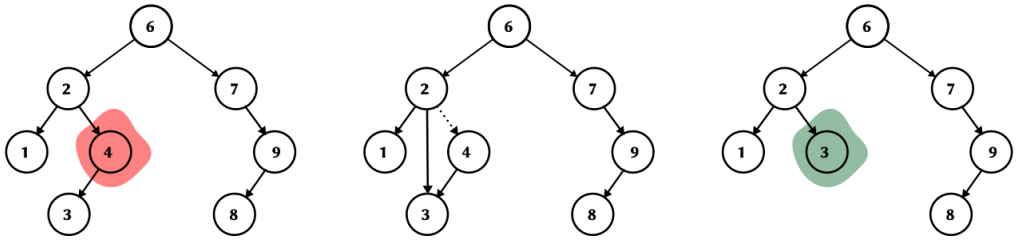
remove As is common with many data structures, the hardes operation is deletion. After finding the item to be deleted several possibilities must be considered:

- If the node is a leaf, it can be deleted immediately.

Removing a childless node (3) from the tree

Figure 13: Removing childless node from tree

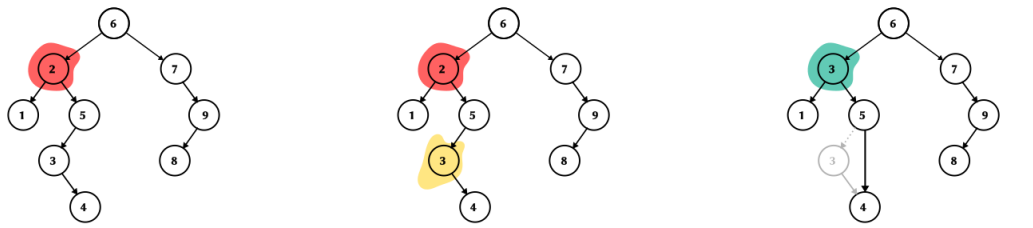
- If the node has one child, the node can be deleted after ist parent adjusts a link to bypass the node.



Progression of removing node (4) with one child

Figure 14: Removing single child node from tree

- If the node has two children, replace the data of this node with the smallest data of the right subtree and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second `remove` is an easy one.



Progression of removing node (2) with two children

Figure 15: Removing node with two children from tree

```
Input: X is an item to be removed and T is the tree.
Output: void

remove( X, T ):
    if T = null:
        return; # Item not found; do nothing
    if X < T.element():
        remove X, T.left()
    else if X > T.element():
        remove X, T.right()
    else if T.left() ≠ null AND T.right() ≠ null:
        min ← findMin T.right()
        T.element ← min.element()
        remove T.element(), T.right()
    else:
        old-node ← T
        T ← T.left ≠ null ? T.left() : T.right()
```

The algorithm above is inefficient because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It's easy to remove this inefficiency by writing a special `removeMin` method.

If the number of deletions is expected to be small, then a popular strategy to use is **lazy evaluation**: When an element is to be deleted, it's left in the tree and merely *marked* as being deleted.

Pros of lazy evaluation (in this context):

- Easy to handle if a deleted item is reinserted, do not need to allocate a new node.
- Do not need to handle finding a replacement node.

Cons of lazy evaluation (in this context):

- The depth of the tree will increase. However this increase is usually a small amount relative to the size of the tree.

1.5.2 Average-case analysis

The running time of all the operations (except `makeEmpty` and copying) is $O(d)$, where d is the depth of the node containing the accessed item.

The sum of the depths of all nodes in a tree is known as the **internal path length**.

Let $D(N)$ be the internal path length for some tree T of N nodes. An N-node tree consists of an i-node left subtree and an $(N-i-1)$-node right subtree, plus a root at depth zero for $0 \leq i \leq N$. $D(i)$ is the internal path length of the left subtree with respect to its root. The same holds for the right subtree. We get the following recurrence $D(N) = D(i) + D(N-i-1) + N - 1$.

In a BTS all subtree sizes are equally likely, so

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

By solving the above recurrence we show that $D(N) = O(NlogN)$. Thus the expected depth of any node is $O(logN)$.

Although it's tempting to say that this result implies the average running time of all the operatuons is $logN$, but this is not entirely true. The reason for this is that because of deletions, it's not clear that all BSTs are equally likely. In particular, the deletion algorithm favors making the left subtrees deeper than the right, because we're always replacing a deleted node with a node from the right subtree.

1.6 Balanced trees

If the input comes into a tree presorted, then a series of inserts will take *quadratic time* and give a very expensive implementation of a linked list, since the tree will consist only of nodes with no left children. One solution to the problem is to insist on an extra structural condition called **balance** where no node is allowed to get too deep.

There are quite a few general algorithms to implement **balanced trees**. Most are quite a bit more complicated than a standard binary search tree, and all take longer on average for updates. They do, however, provide protection against the embarrassingly simple cases. Below, we will sketch one of the oldest forms of balanced search trees, the AVL tree.

1.7 AVL trees

An AVL (Adelson-Velskii-Landis) tree is a a binary search tree that is *height-balanced*: At each node u, the height of the subtree rooted at `u.left` and the subtree rooted at `u.right` differ by at most 1. The balance condition must be easy to mantain, and it ensures that the depth of the tree $O(logN)$. The simplest idea is to require that the *left and right subtrees have the same height*.

It follows immediately that, if $F(h)$ is the minimum number of leaves in a tree of height h, then $F(h)$ obeys the Fibonacci recurrence $F(h) = F(h-1) + F(h-2)$ with base cases $F(0) = 1$ and $F(1) = 1$. This means $F(h)$ is approximately $\frac{\phi^h}{\sqrt{5}}$, where $\phi = (1+\sqrt{5})/2 \approx 1.61803399$ is the *golden ratio* (More precisely,$|\phi^h/\sqrt{5} - F(h)| \leq 1/2$.) This implies $h \leq log_\phi n \approx 1.44042008\log n$, although in practice is only slightly more than $\log N$.

An **AVL tree** is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. The height of an empty tree is defined to be -1.

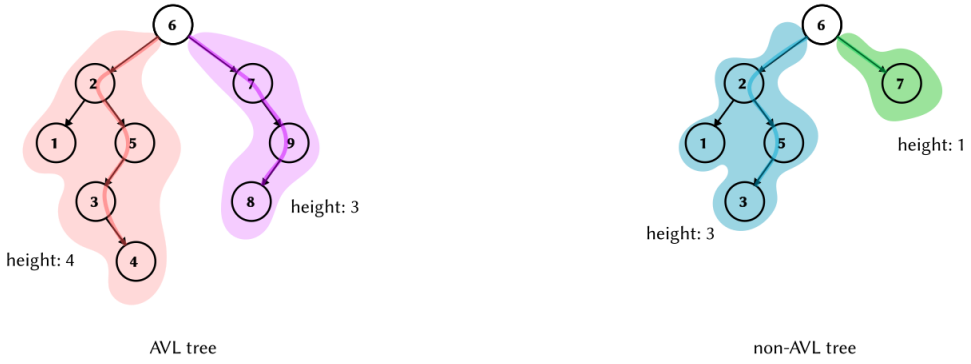


Figure 16: AVL tree and non-AVL tree

1.7.1 Insertion cases

After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we follow the path up to the root and update the *balancing information*, we may find a fnode whose new balance violates the AVL condition.

Suppose that α is the first node on the path that needs to be rebalanced. Since any node has at most two children, and a height imbalance requires that the two subtrees's heights of α differ by two, it's easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

Cases 1 and 4 are mirror image symmetries with respect to α, as are cases 2 and 4. Thus, as a matter of theory, there are only two basic cases. Although programmatically, there are still four cases.

A **single rotation** of the tree fixes case 1 (and 4) in which the insertion occurs on the *outside* (i.e., left-left or right-right). A **double rotation** (which is slightly more complex) fixes case 2 (and 3) in which the insertion occurs on the *inside* (i.e., left-right or right-left). The implementation of a double rotation simply involves two calls to the routine implementing a single rotation, although conceptually it may be easier to consider them two separate and different operations.

Single rotation

NOTE: $k_1 < k_2$

Single rotation to fix case 1:

In the "Single rotation to fix case 1" figure, subtree X has grown to an extra level, causing it to be exactly two levels deeper than Z. The subtree Y cannot be at the same level as the X because then k_2 would have been out of balance *before* the insertion, and Y cannot be at the same level as Z because then k_1 would be the first node on the path toward the root that was in violation of the AVL balancing property.

To ideally rebalance the tree, we would like to move X up a level and Z down a level. By grabbing child node k_1 and letting the other nodes hang, the result is that k_1 will be the new root. The binary search tree property tells us that in the original tree $k_2 > k_1$, so k_2 becomes the right child of k_1 in the new tree. X and Z remain as the left child of k_1 and right child of k_2, respectively. Subtree Y, which holds items that are between k_1 and k_2 in the original tree, can be placed as k_2's left child in the new tree and satisfy all the ordering requirements.

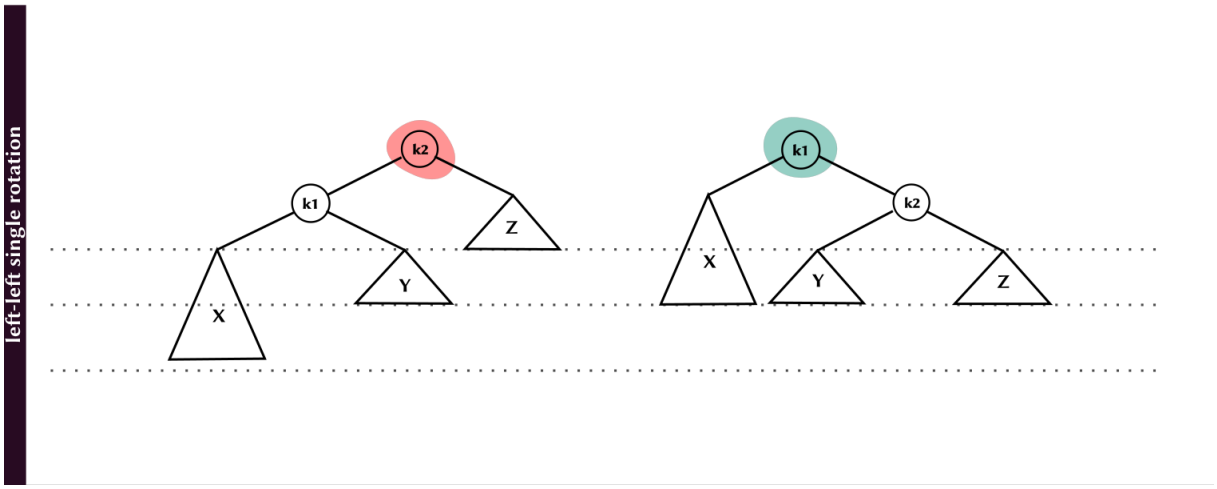


Figure 17: Single rotation to fix case 1

Single rotation to fix case 4:

Case 4 represents a symmetric case and the "Single rotation to fix case 4" figure shows how a single rotation is applied.

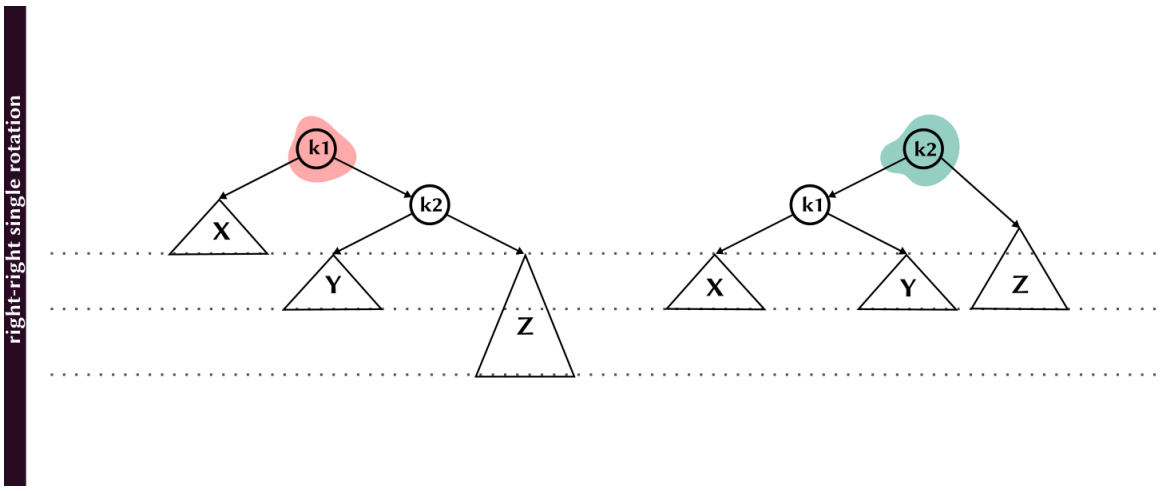


Figure 18: Single rotation to fix case 4

Example Let's suppose we start with an empty AVL tree and inserts the items $3, 2, 1$, and then 4 through 7 sequentially. As we insert certain items, we must do some rotations along the ways.

The first problem occurs when it's time to insert the item 1 because the AVL tree property is violated at the root. Thus, we perform a single rotation between the root and its left child to fix the problem. The "First rotation after violation of AVL property" figure shows the before and after the rotation.

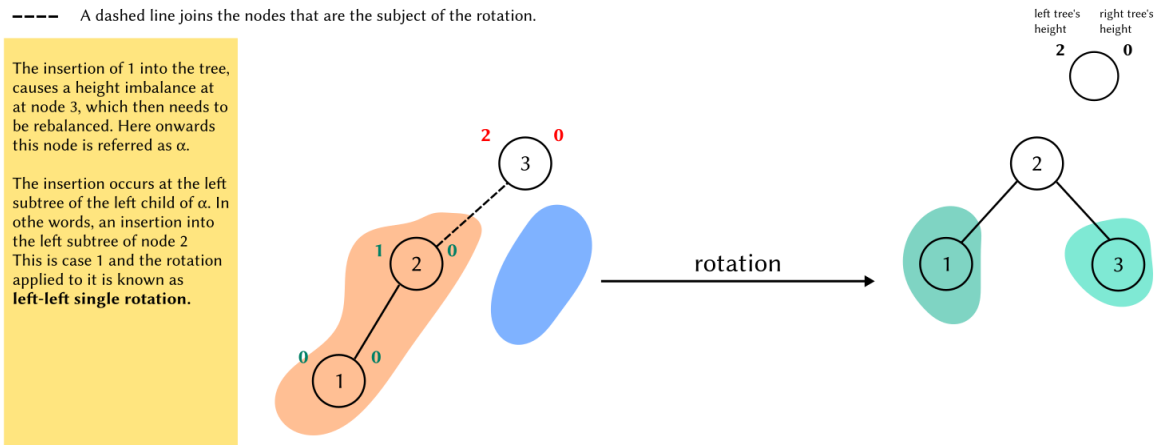


Figure 19: First rotation after violation of AVL property

Next we insert 4 but it causes no problems. However the insertion of 5 creates a violation at node 3 that is fixed by a single rotation. The rest of the tree has to be informed of this change so 2's right child must be reset to link to 4 instead of 3. The "Second rotation after violation of AVL property" figure shows the before and after the rotation.

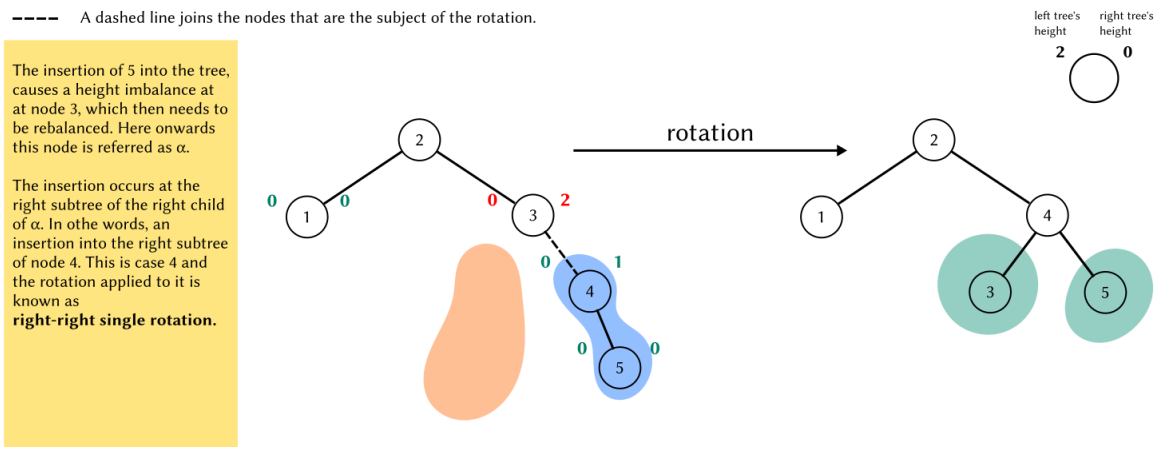


Figure 20: Second rotation after violation of AVL property

Next we insert 6 that causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be of height 2. Therefore, we perform a single rotation at the root between 2 and 4. The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2. Every item in this subtree must lie between 2 and 4, so this transformation makes sense. The "Third rotation after violation of AVL property" figure shows the before and after the rotation.

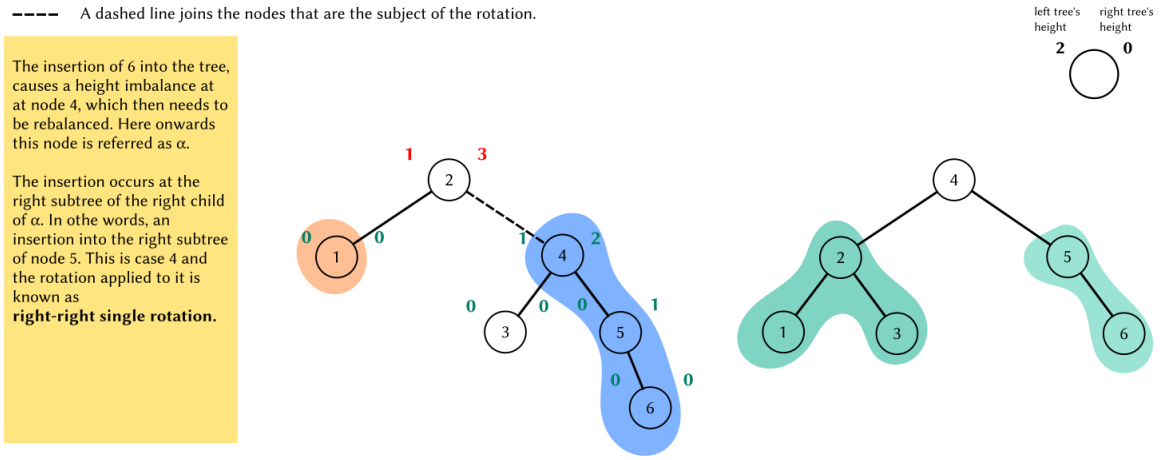


Figure 21: Third rotation after violation of AVL property

The next item we insert is 7, which causes another rotation. The "Fourth rotation after violation of AVL property" figure shows the before and after the rotation.

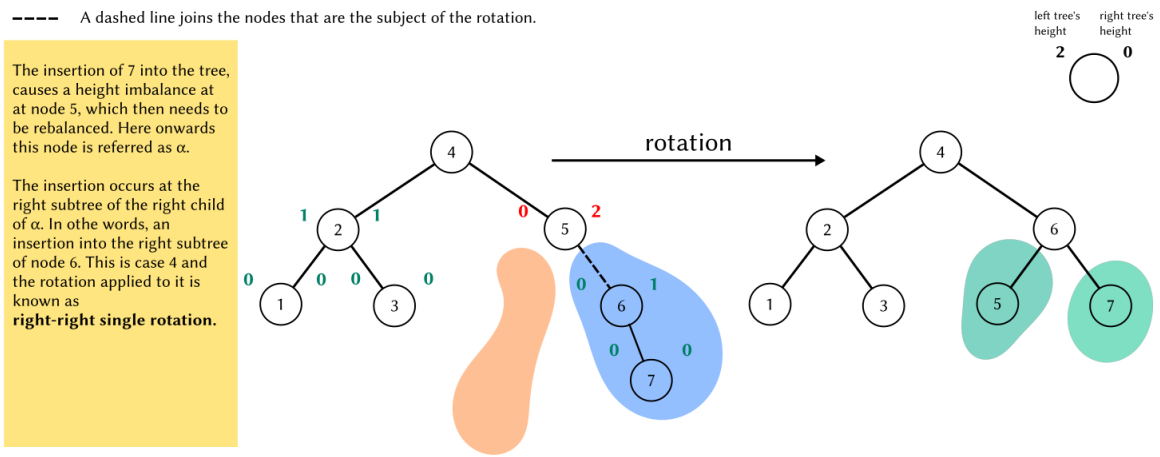


Figure 22: Fourth rotation after violation of AVL property

Double rotation The reason why single rotation doesn't work in cases 2 and 3 is because a subtree might be too deep, and a single rotation doesn't make it any less deep.

NOTE: $k_1 < k_2 < k_3$

Double rotation to fix case 2:

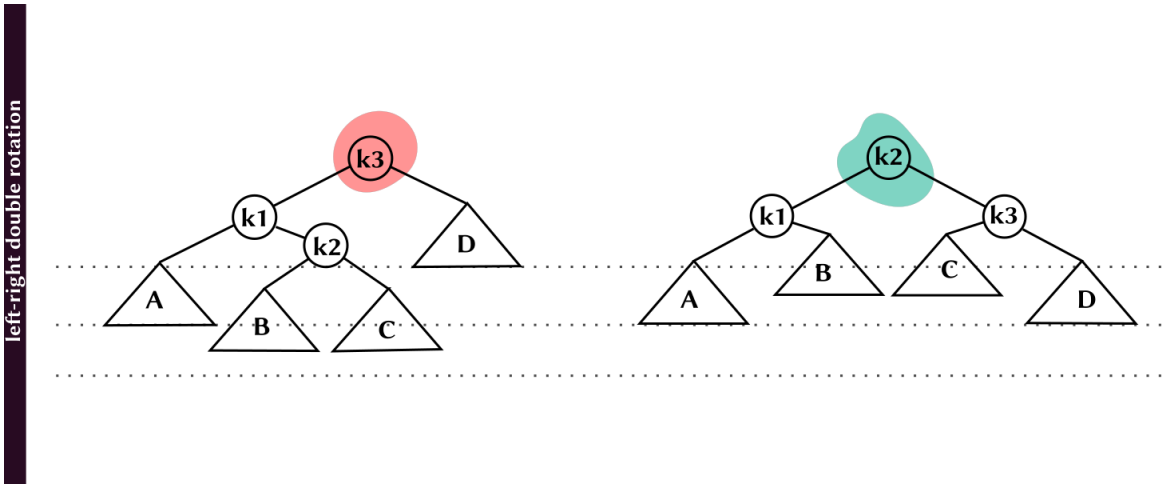


Figure 23: Left-right double rotation to fix case 2

Double rotation to fix case 3:

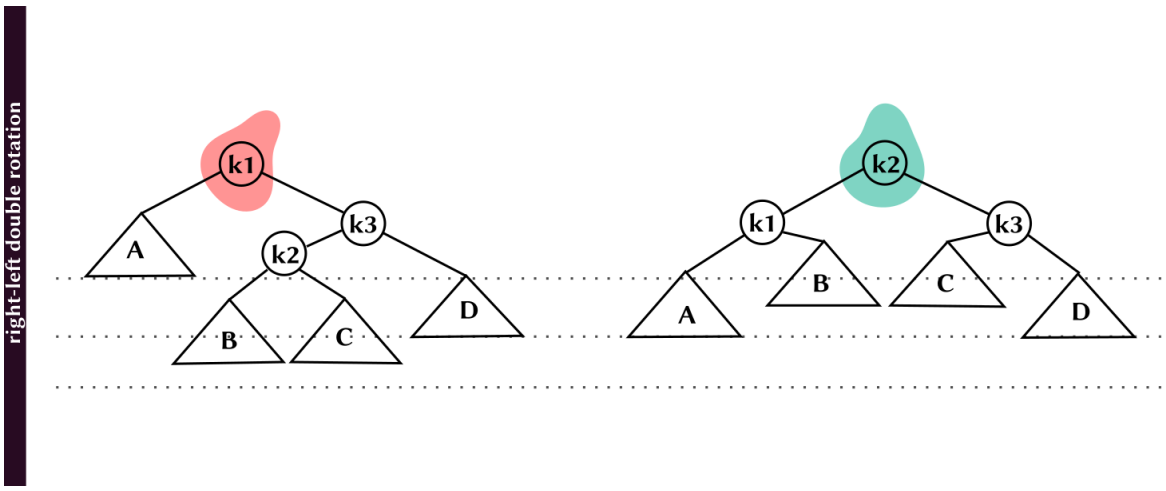


Figure 24: Right-left double rotation to fix case 3

Example For this example, we'll continue with the AVL tree depicted in the "Fourth rotation after violation of AVL property" figure from the "Single rotation" section. We'll start by inserting 10 through 16 (in reverse order), followed by 8 and then 9.

Inserting 16 is easy since it doesn't violate the AVL balance property. However, inserting 15 causes a height imbalance at node 7, which is case 3 and solved by a right-left double rotation.

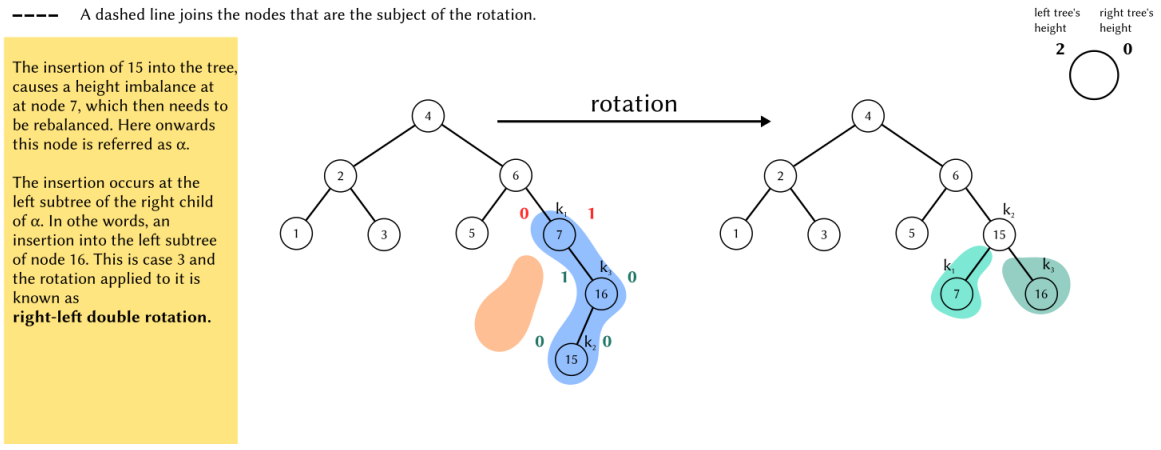


Figure 25: Fifth rotation after violation of AVL property

Next we insert 14, which also requires the same double rotation as before.

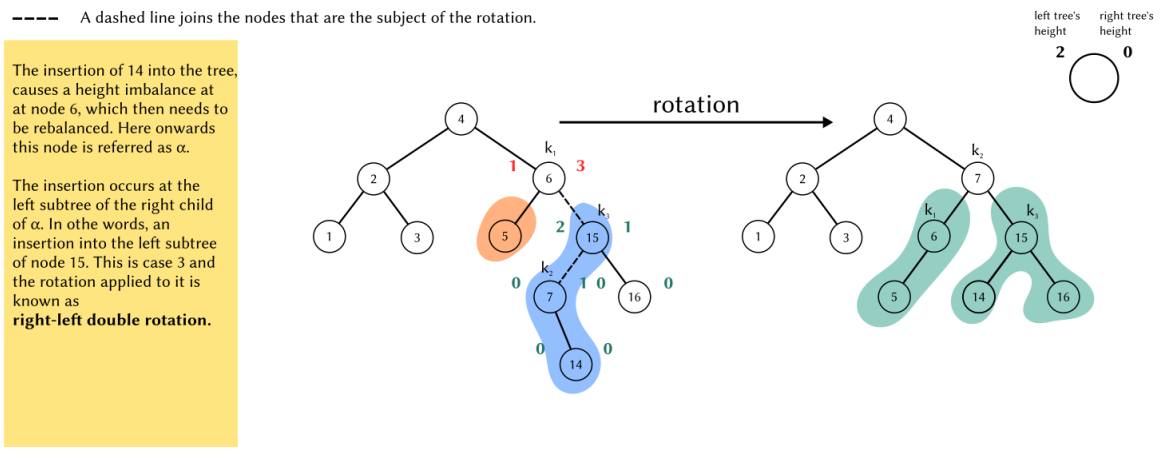


Figure 26: Sixth rotation after violation of AVL property

The insertions of 13, 12, 11, and 10 all cause height imbalance that are fixed using single rotations.

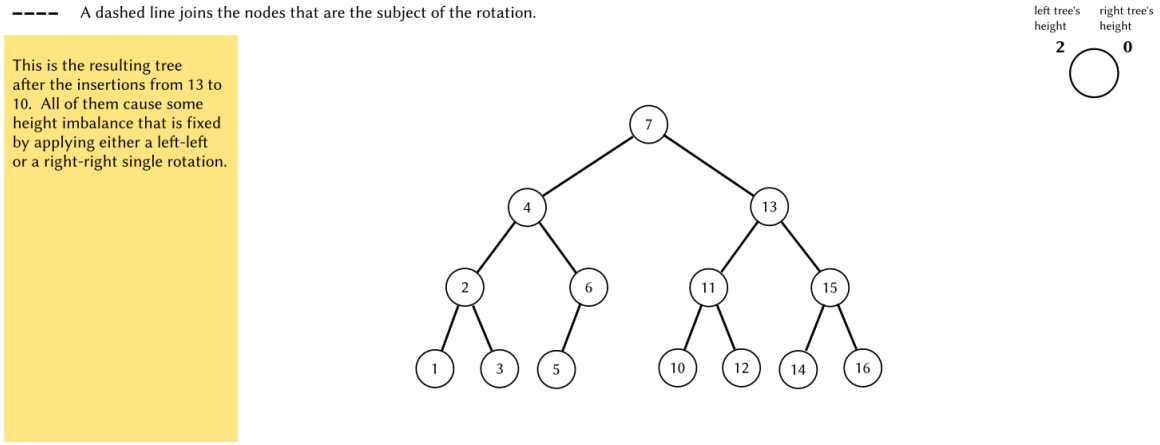


Figure 27: Tree after the seventh, eighth, ninth and tenth rotations after violation of AVL property

We can insert 8 without rotation, creating an almost perfectly balanced tree. The insertion of 9 causes a height imbalance at the node containing 10 which is fixed by a left-right double rotation.

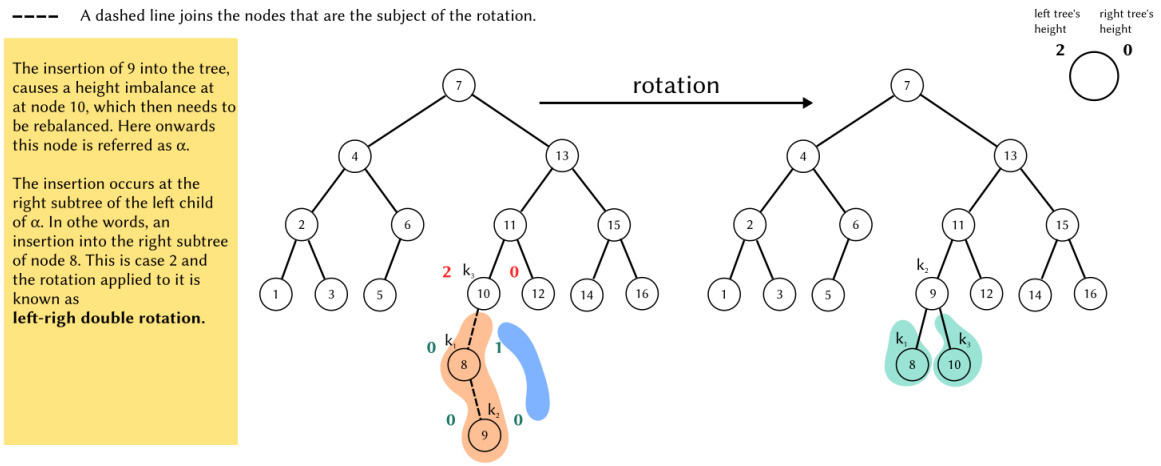


Figure 28: Tree after the twelfth rotation