CSE 5311: Design and Analysis of Algorithms

Binary Search Trees

Alex Dillhoff

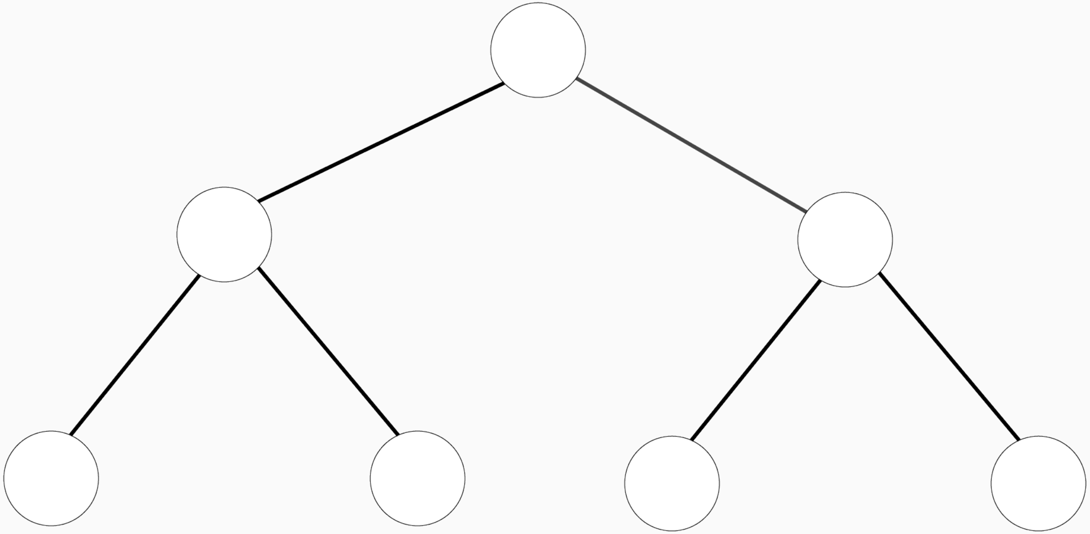
University of Texas at Arlington

Binary Trees

Binary trees are a graph-based data structure.

They are defined as a hierarchical tree of nodes in which each node has at most two sub-nodes.

Binary Trees



Binary Trees

Tree-based data structures share a few common properties and definitions:

- The **size** of a tree T is determined by the total number of nodes in T.
- The **root** of a tree T is the starting point of T.
- A **leaf node** in a tree T is a node that has no sub-nodes.
- The **height** of a tree T is determined by the *length* of the shortest path between the root of T and the lowest leaf node of T.

Binary Trees

Each node in a binary tree contains the following information:

- A reference to the **parent** node.
- A reference to the **left** sub-node.
- A reference to the **right** sub-node.
- A **key** representing some identifier.
- A **value** containing the data.

Binary Trees

```
class Node:
def __init__(self, key, value):
    self.key = key
    self.value = value
    self.left = None
    self.right = None
    self.parent = None
```

Binary Trees

Binary trees are especially useful for sorting and searching data efficiently.

The data can be as simple as a single scalar value or as complex as a multi-membered `class` object as long as a suitable **key** can be chosen to represent each node.

Constructing a Binary Tree

In general, a binary tree can be constructed in any fashion as long as each node has **at most** two sub-nodes.

Although the usefulness of a binary tree comes from utilizing their hierarchical nature, there is no requirement that the data need to be sorted in any particular way.

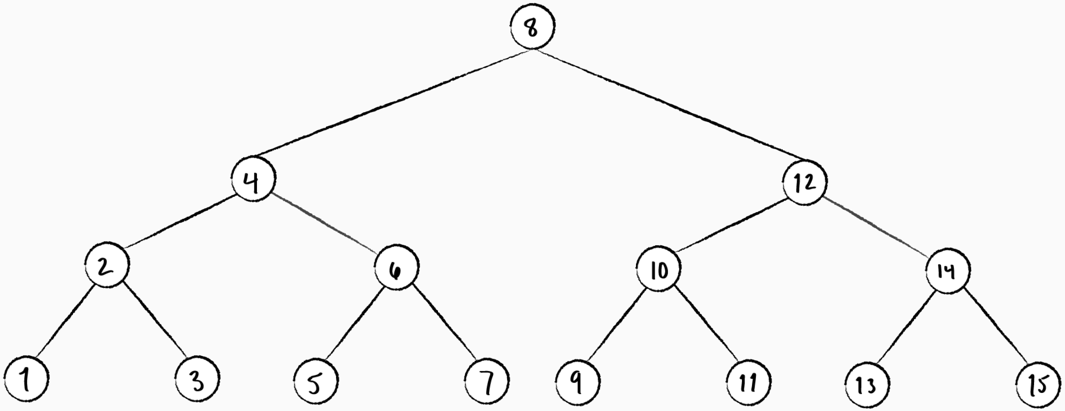
Binary Search Trees

To take advantage of the benefits of efficient sorting and searching, a Binary Search Tree must be used.

A Binary Search Tree is defined by the following property:

If x is a node in a binary search tree and y is a sub-node of x, then y is a *left sub-node* if $y.key \leq x.key$ and y is a *right sub-node* if $y.key \geq x.key$.

Binary Search Trees



Operations

There are several operations that can be performed on a Binary Search Tree.

1. Traversal
2. Search
3. Insertion
4. Deletion
5. Minimum/Maximum
6. Successor/Predecessor

Operations - Traversal

Given a binary tree T, there are several different ways to traverse the nodes of T.

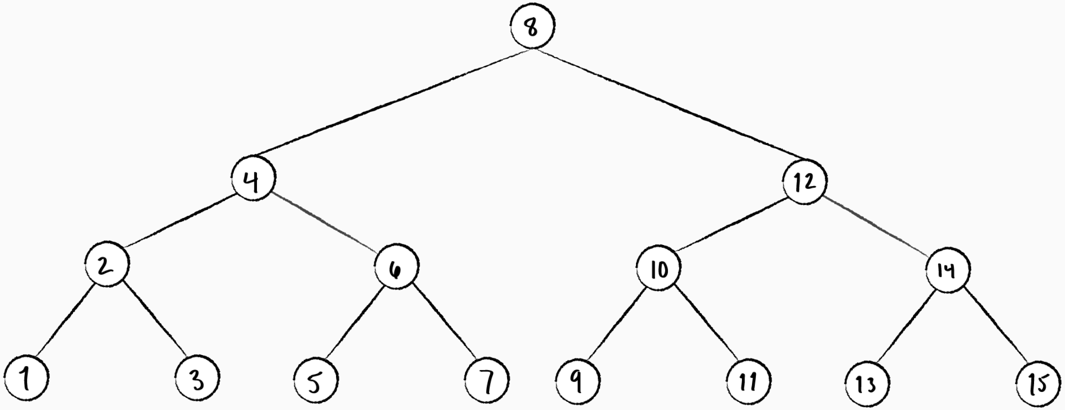
1. Depth First Search (Preorder)
2. Depth First Search (Inorder)
3. Depth First Search (Postorder)
4. Breadth First Search

Depth First Search (Preorder)

To perform a preorder DFS, the *key* of the current node is printed *before* moving to the sub-nodes.

```
def preorder_dfs(n):
    if n is not None:
        print(n.key)
        preorder_dfs(n.left)
        preorder_dfs(n.right)
```

Binary Search Trees

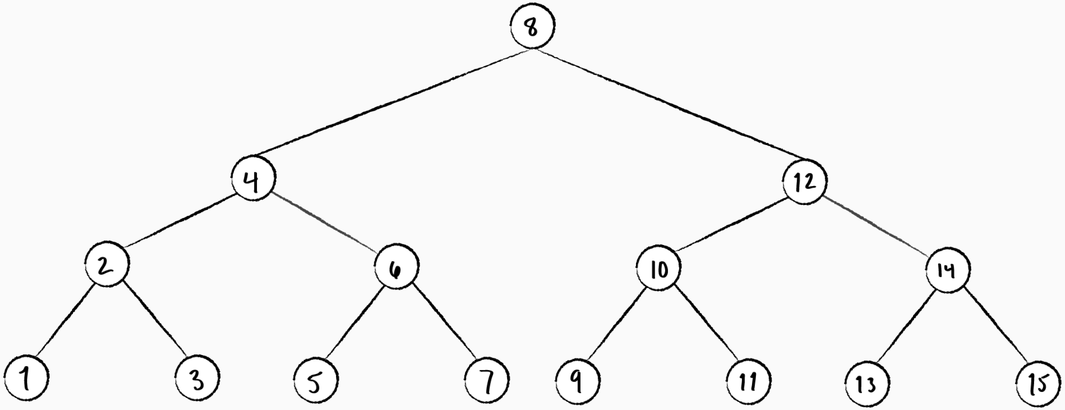


Depth First Search (Inorder)

To perform a inorder DFS, the *key* of the current node is printed *between* the sub-nodes.

```
def inorder_dfs(n):
    if n is not None:
        inorder_dfs(n.left)
        print(key)
        inorder_dfs(n.right)
```

Binary Search Trees



Depth First Search (Inorder)

For a Binary Search Tree, an inorder traversal will print the items out in order from least to greatest, according to they key.

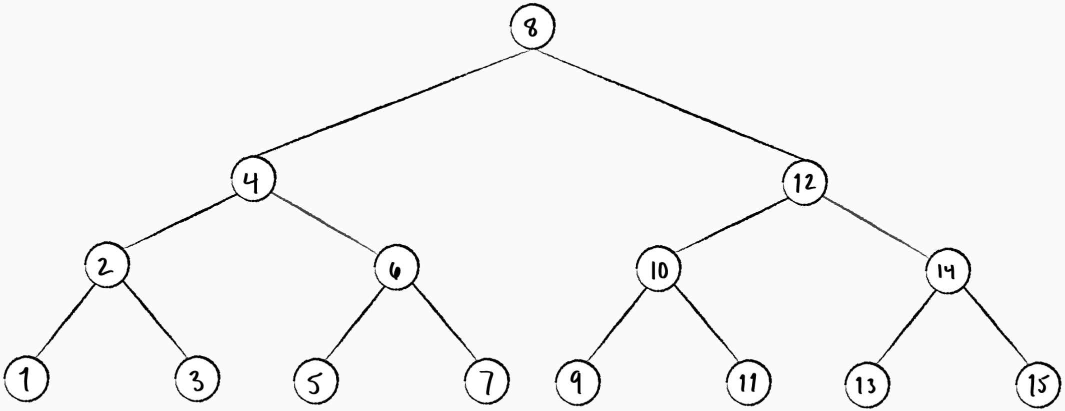
This traversal is the key to the efficiency of a Binary Search Tree.

Depth First Search (Postorder)

To perform a postorder DFS, the *key* of the current node is printed *after* moving to the sub-nodes.

```python
def postorder_dfs(n):
    if n is not None:
        postorder_dfs(n.left)
        postorder_dfs(n.right)
        print(n.key)
```

Binary Search Trees

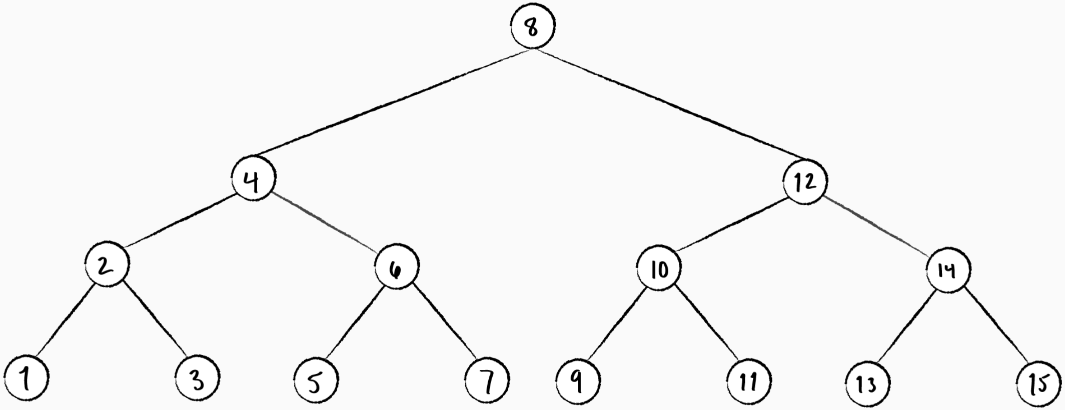


Breadth First Search

A Breadth First Search prints each level of the tree in order from top to bottom, left to right.

In this way, the breadth of the layer is explored before moving to the next level in the height of the tree.

Binary Search Trees



Operations - Search

Searching a Binary Search Tree involves looking at each node, starting with the root, until the desired *key* is found.

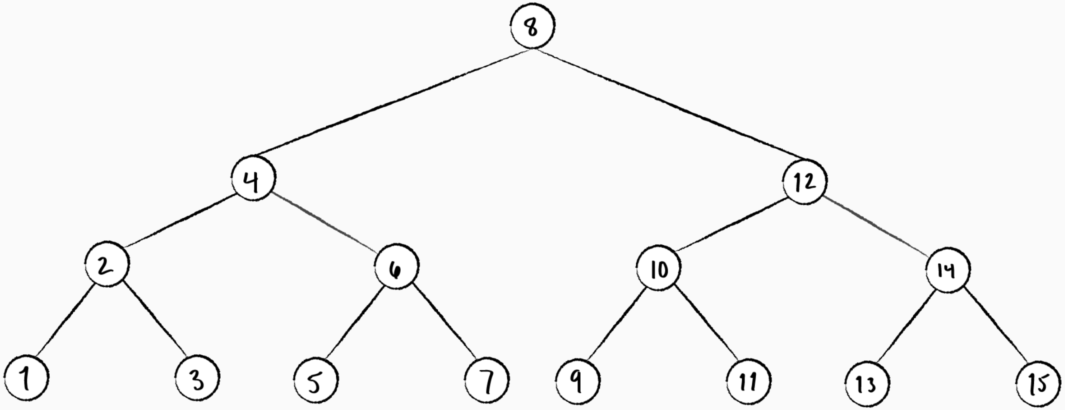
If the target value is less than the *key*, the left sub-node is traversed. Otherwise, the right sub-node is traversed.

This continues until a leaf node is reached.

Operations - Search

```
def tree_search(x, k):
    if x is None or k == x.key:
        return x
    if k < x.key:
        return tree_search(x.left, k)
    else:
        return tree_search(x.right, k)
```

Binary Search Trees



Operations - Minimum

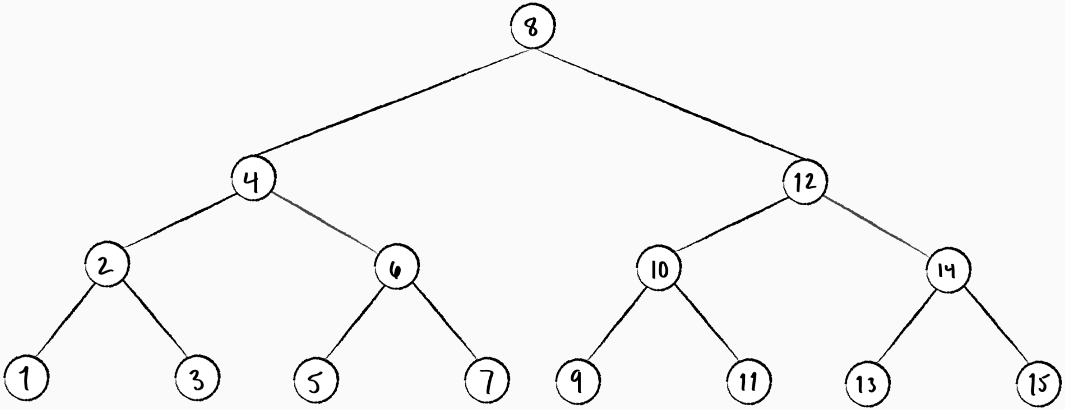
In a BST, the minimum value is the leftmost node.

Finding the minimum is as easy as traversing down the left branch until a leaf node is reached.

Operations - Minimum

```
def tree_minimum(x):
    while x.left is not None:
        x = x.left
    return x
```

Binary Search Trees



Operations - Maximum

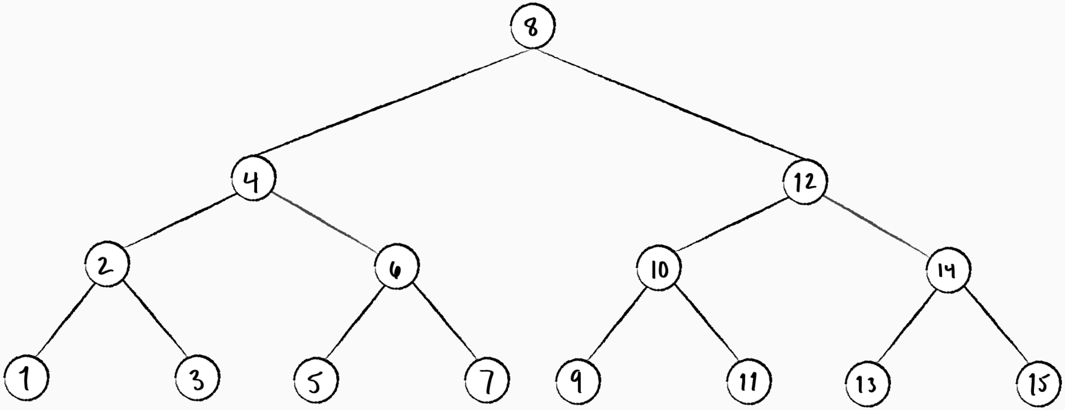
In a BST, the maximum value is the rightmost node.

Finding the maximum is as easy as traversing down the right branch until a leaf node is reached.

Operations - Maximum

```
def tree_maximum(x):
    while x.right is not None:
        x = x.right
    return x
```

Binary Search Trees



Operations - Successor

The successor and predecessor operations are useful for the delete operation.

The successor of a node x is the node with the smallest key greater than $x.key$.

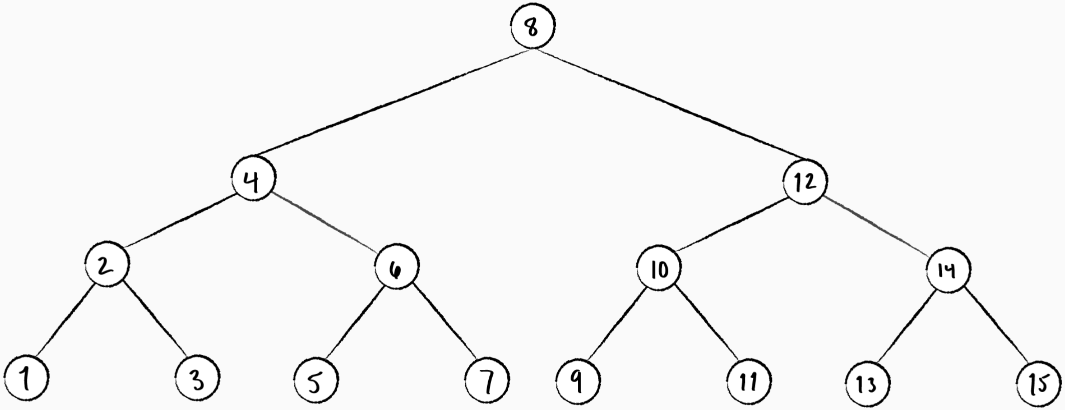
If x has a right subtree, then the successor of x is the minimum of the right subtree.

If x has no right subtree, then the successor of x is the lowest ancestor of x whose left child is also an ancestor of x.

Operations - Successor

```
def tree_successor(x):
    if x.right is not None:
        return tree_minimum(x.right)
    y = x.parent
    while y is not None and x == y.right:
        x = y
        y = y.parent
    return y
```

Binary Search Trees



Operations - Predecessor

The predecessor of a node x is the node with the largest key less than $x.key$.

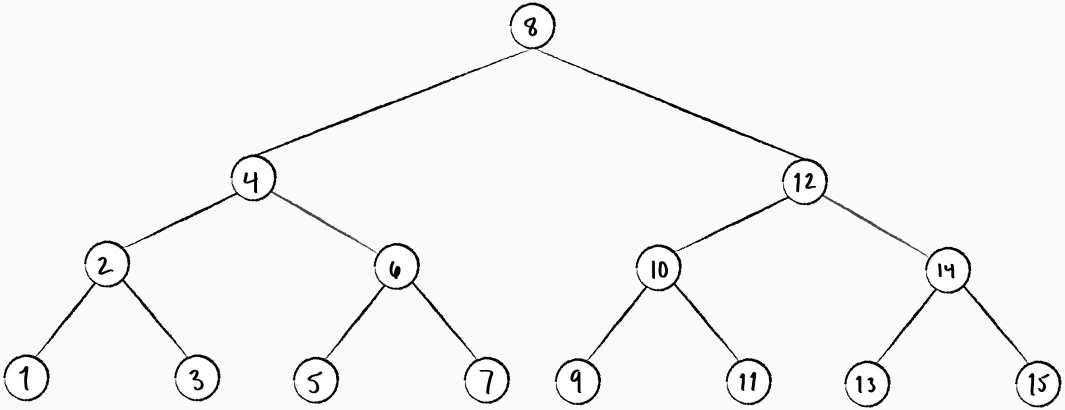
If x has a left subtree, then the predecessor of x is the maximum of the left subtree.

If x has no left subtree, then the predecessor of x is the lowest ancestor of x whose right child is also an ancestor of x.

Operations - Predecessor

```
def tree_predecessor(x):
    if x.left is not None:
        return tree_maximum(x.left)
    y = x.parent
    while y is not None and x == y.left:
        x = y
        y = y.parent
    return y
```

Binary Search Trees



Operations - Insertion

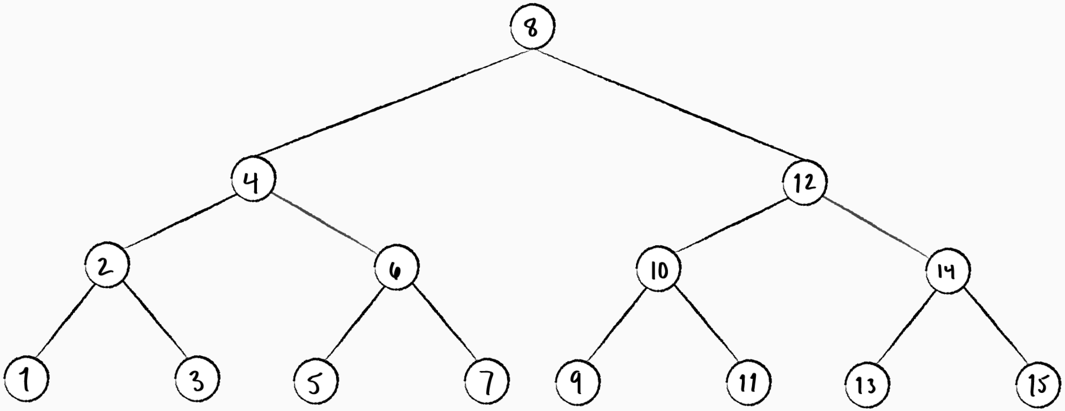
Besides searching, inserting a node into a Binary Search Tree is one of the greatest benefits of using them.

A new node is inserted depending on its *key* relative to the tree *T*.

Operations - Insertion

```python
def tree_insert(T, z):
    y = None
    x = T.root
    while x is not None:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if y is None:
        T.root = z
    elif z.key < y.key:
        y.left = z
    else:
        y.right = z
```

Binary Search Trees



Operations - Insertion

The **insert** operation is easy to implement and comes with the benefit that the tree *T* retains the property of a Binary Search Tree after the item is inserted.

Deleting a node is more complicated and may require reorganization of the tree.

Operations - Deletion

If the node is a leaf, the node can simply be set to NULL.

If the node has a single subnode, the subnode is then assigned to the parent of the current node.

If the node has two subnodes, the graph must be restructured depending on the data.

Operations - Delete

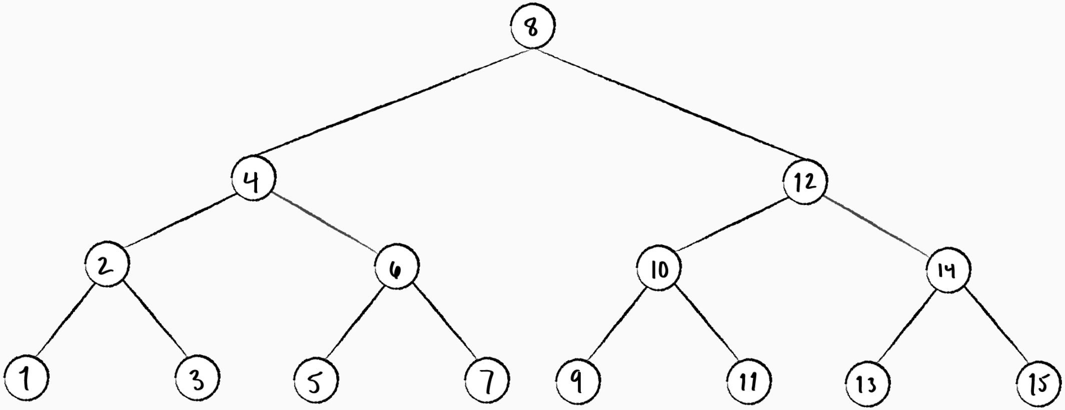
In the third case, node z has both a left and right subnode.

The first step is to find the successor of z: y.

Since z has 2 subnodes, its successor has no left subnode. Likewise, its predecessor has no right subnode.

If y is the right subnode of z, replace z by y.

Binary Search Trees



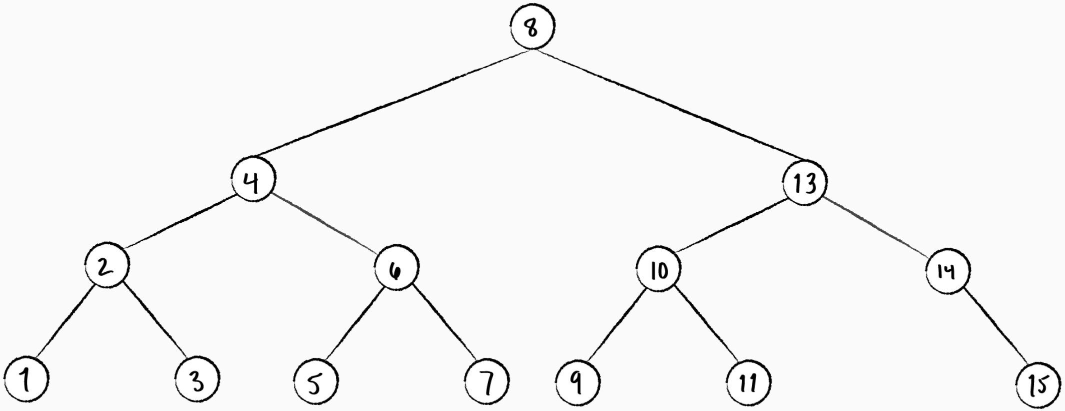
Operations - Delete

If *y* is not the right subnode of *z*, it is somewhere further down the right branch.

In this case, replace *y* by its right subnode before replacing *z* by *y*.

The figure below shows the removal of node 12 from the tree in the figure above.

Operations - Delete



Operations - Transplant

Even though only 1 node was moved (13 to 12’s old position), the process of deleting a node actually involves **transplanting** a subtree to a new position.

Operations - Transplant

```python
def transplant(T, u, v):
    if u.parent is None:
        T.root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    if v is not None:
        v.parent = u.parent
```

Operations - Tree Delete

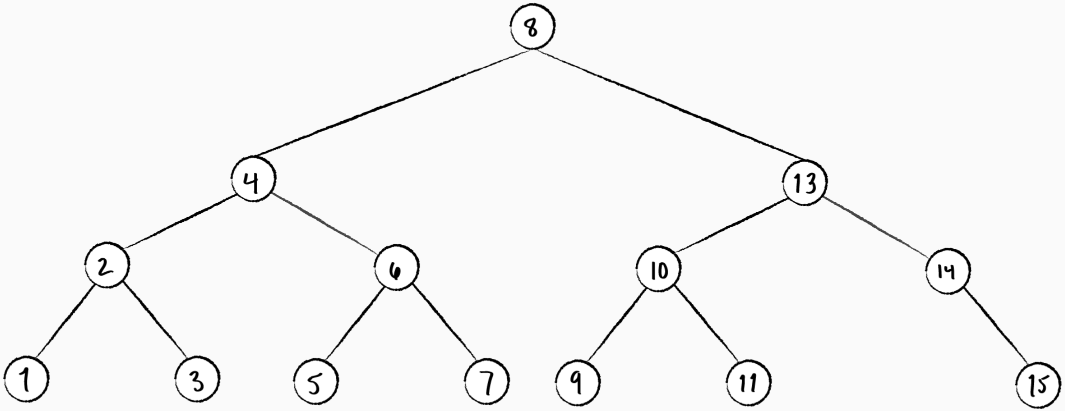
In the code above, `u` is the node to be replaced, and `v` is the node to replace it.

Updating `v`'s left and right subnodes are done in the calling function `tree_delete`.

Operations - Tree Delete

```
def tree_delete(T, z):
    if z.left is None:  # Case 1 and 2
        transplant(T, z, z.right)
    elif z.right is None: # Also case 1 and 2
        transplant(T, z, z.left)
    else: # Case 3
        y = tree_minimum(z.right) # get successor
        if y != z.right:
            transplant(T, y, y.right)
            y.right = z.right
            y.right.parent = y
        transplant(T, z, y)
        y.left = z.left
        y.left.parent = y
```

Operations - Delete



Analysis

Insert, delete, and search all run in $\Theta(h)$ time, where h is the height of the tree.

If the tree is balanced, $h = \Theta(\log n)$, and all operations run in $\Theta(\log n)$ time.

If the tree is not balanced, $h = \Theta(n)$, and all operations run in $\Theta(n)$ time.