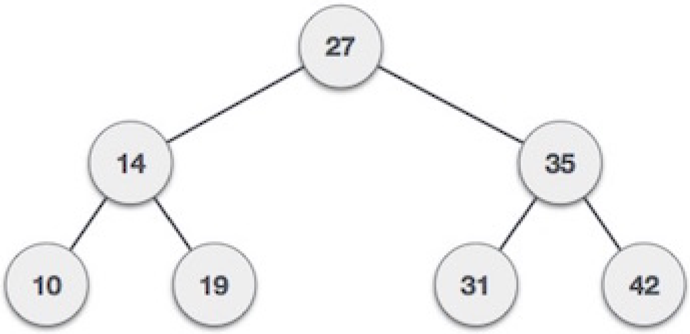
Binary Search Tree

1. Introduction

Binary Search Tree is a type of binary tree with following special properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.



https://www.tutorialspoint.com/data_structures_algorithms/images/binary_search_tree.jpg

Duplicate Values?

There are a few variation of Binary Search Tree definition. By most definitions, BST only allow distinct values, and duplicates are not allowed.

This is because allowing duplicate values will bring much more complexity than convenience. For example, duplicate value 27 may be inserted at different levels in the tree.

```
      27
    /    \
  14      35
    \
     19
       \
        27
```

Extend from Binary Tree

Node Class

Since BST is a type of Binary Tree, they share the same type of nodes in the tree. We will reuse the `Node` class.

In [1]:

```
class Node:

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

    def __str__(self):
        return '{}({},{})'.format(self.data,
                                  self.left.data if self.left else '',
                                  self.right.data if self.right else '')

    def __repr__(self):
        return '{}({},{})'.format(self.data,
                                  self.left.data if self.left else '',
                                  self.right.data if self.right else '')
```

Test:

In [30]:

```
n1 = Node(10, Node(5), Node(15))
print(n1)
```

```
10(5,15)
```

BinaryTree Class

We will use the `BinaryTree` class as the base class for our `BinarySearchTree` class.

In [3]:

```
class BinaryTree:

    def __init__(self, root=None):
        self.root = root

    def print_tree(self):
        self._print_tree([self.root])

    def _print_tree(self, node_list):
        # Convert node_list to a list if it is not
        if not isinstance(node_list, list):
            node_list = [node_list]
        # Stop recursion if the list is empty
        if not node_list:
            return
        # define a list to collect nodes in next layer
        next_layer = []
        while node_list:
            node = node_list.pop()
            print(node, end=' ')
            if node.left:
                next_layer.insert(0, node.left)
            if node.right:
                next_layer.insert(0, node.right)
        print()
        self._print_tree(next_layer)
```

BinarySearchTree Class

Defines a `BinarySearchTree` class which inherits from `BinaryTree`.

- No need to implement any additional attribute.

In [4]:

```
class BinarySearchTree(BinaryTree):
    pass
```

2. Insert a Node

The operation to insert a value to is a **recursive process** at each node of the tree.

Assume current node is not `None`,

- if the incoming value `val` is less than current node's value,
  - if left child is `None`, create a new node with the value and assign to it,
  - else recurse into left subtree.
- if the incoming value is greater than or equals to current node's value,
  - if right child is `None`, create a new node with the value and assign to it,
  - else recurse into right subtree.

Following recursive function `_add(node, val)` adds `val` to the tree where `node` is the current node.

In [5]:

```
def _add(node, val):
    if node is None: # for precaution
        return
    if val < node.data:
        if node.left is None:
            node.left = Node(val)
        else:
            _add(node.left, val)
    if val > node.data:
        if node.right is None:
            node.right = Node(val)
        else:
            _add(node.right, val)
```

Test:

- Construct following tree

```
       10
      /  \
     8    12
    /
   6
```

In [6]:

```
root = Node(10)
_add(root, 8)
_add(root, 12)
_add(root, 6)
print(root)
print(root.left, root.right)
```

```
10(8,12)
8(6,) 12(,)
```

Exercise

Implement a class `BinarySearchTree` inheriting from `BinaryTree` .

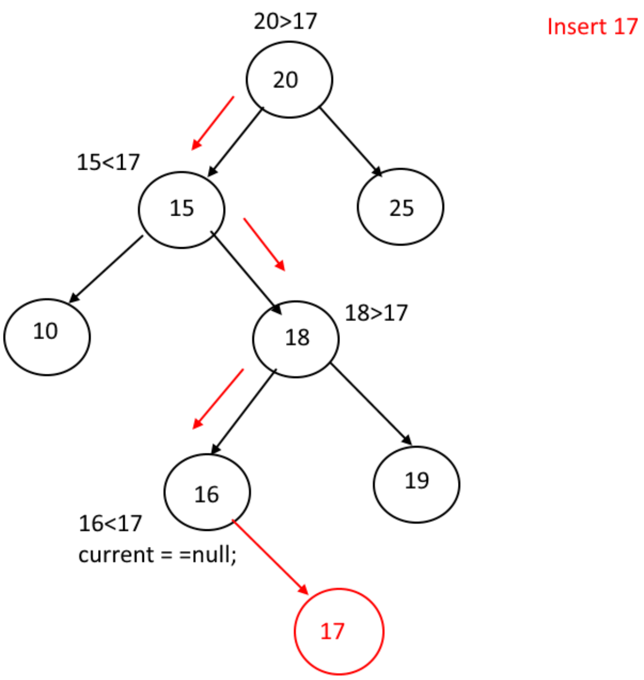
- It has a `root` attribute pointing to its root node.
- Implement its `add()` operation which adds a value `val` to the tree.
  - Use the above recursive function `_add()` .

In [7]:

```
class BinarySearchTree(BinaryTree):

    def add(self, val):
        if self.root is None:
            self.root = Node(val)
        else:
            self._add(self.root, val)

    def _add(self, node, val):
        if node is None: # for precaution
            return
        if val < node.data:
            if node.left is None:
                node.left = Node(val)
            else:
                self._add(node.left, val)
        if val > node.data:
            if node.right is None:
                node.right = Node(val)
            else:
                self._add(node.right, val)
```

test:

- Construct following tree
- Insert a value 17 to the tree



https://algorithms.tutorialhorizon.com/binary-search-tree-complete-implementation/

In [8]:

```
t = BinarySearchTree()
t.add(20)
t.add(15)
t.add(25)
t.add(10)
t.add(18)
t.add(16)
t.add(19)
# t.print_tree()
t.add(17)
t.print_tree()
```

```
20(15,25)
15(10,18) 25(,)
10(,) 18(16,19)
16(,17) 19(,)
17(,)
```

3. Find a Node

To search a given node in Binary Search Tree,

- If the value matches current node's data, return the node.
- If the value is greater than current node, recur into the right subtree of root node.
- Otherwise we recur into the left subtree.

Following recursive function `_find(node, val)` find the `val` in the tree where `node` is the root.

In [9]:

```
def _find(node, val):
    if node is None:
        return None
    print(node)  # print current node to show traversal

    if val == node.data:
        return node
    elif val < node.data:
        return _find(node.left, val)
    else:
        return _find(node.right, val)
```

Test:

- Find node with value 10 in the tree `t`.

In [10]:

```
result = _find(t.root, 10)
print(result)
```

```
20(15,25)
15(10,18)
10(,)
10(,)
```

Exercise

Implement a class `BinarySearchTree1` which inherits from `BinarySearchTree` .

- Implement its `find()` method which adds a value `val` to the tree. Use the above recursive function `_find()` .

In [11]:

```
class BinarySearchTree1(BinarySearchTree):

    def find(self, val):
        if self.root:
            return self._find(self.root, val)
        else:
            return None

    def _find(self, node, val):
        if node is None:
            return None
        print(node)  # print current node to show traversal

        if val == node.data:
            return node
        elif val < node.data:
            return self._find(node.left, val)
        else:
            return self._find(node.right, val)

```

Test:

- Construct a tree with values [10, 7, 8, 9, 6, 11, 4, 13, 2, 15]
- Find value `15` in the constructed tree

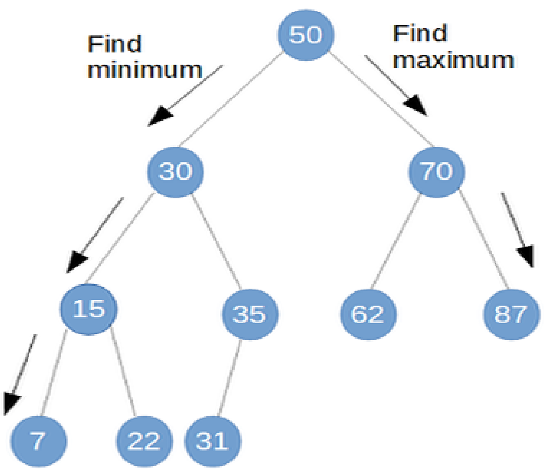
In [12]:

```
t = BinarySearchTree1()

s = [10, 7, 8, 9, 6, 11, 4, 13, 2, 15]
for i in s:
    t.add(i)

t.print_tree()
print('-'*10)

print(t.find(15))
```

```
10(7,11)
7(6,8) 11(,13)
6(4,) 8(,9) 13(,15)
4(2,) 9(,) 15(,)
2(,)
----------
10(7,11)
11(,13)
13(,15)
15(,)
15(,)
```

4. Find the Min/Max Node

With the structure of Binary Search Tree, finding the node with minimum value or maximum value is a simple operation.



Exercise:

Construct a binary search tree with above structure.

In [13]:

```
t = BinarySearchTree1()
s = [50, 30, 70, 15, 35, 62, 87, 7, 22, 31]
for i in s:
    t.add(i)

t.print_tree()
```

```
50(30,70)
30(15,35) 70(62,87)
15(7,22) 35(31,) 62(,) 87(,)
7(,) 22(,) 31(,)
```

Minimum Value Node

Implement a function `_find_min()` to find a node with minimum value in the tree. The function takes a root node as input parameter.

- Starting from the root node, go to its **left child**.
- Keep traversing the left children of each node until a node with no left child. That node is a node with minimum value.

In [14]:

```
def _find_min(node):
    if node is None:
        return None
    if node.left is None:
        return node
    else:
        return _find_min(node.left)
```

Test:

- Find the node with minimum value in tree `t`.

In [15]:

```
mi = _find_min(t.root)
print(mi)
```

```
7(,)
```

Max Value Node

To find the node with max value:

- Starting from the root node go to its **right child**.
- Keep traversing the right children of each node until a node with no right child. That node is a node with max value.

In [16]:

```
def _find_max(node):
    if node is None:
        return None
    if node.right is None:
        return node
    else:
        return _find_max(node.right)
```

Test:

- Find the node with maximum value in tree `t`.

In [17]:

```
ma = _find_max(t.root)
print(ma)
```

```
87(,)
```

Enhance `BinarySearchTree`

Enhancement `BinarySearchTree1` by adding `find_max()` and `find_min()` function to the class.

- Name the new class `BinarySearchTree2`.

In [18]:

```
class BinarySearchTree2(BinarySearchTree1):

    def find_min(self):
        return self._find_min(self.root)

    def _find_min(self, node):
        if node is None:
            return None
        if node.left is None:
            return node
        else:
            return self._find_min(node.left)

    def find_max(self):
        return self._find_max(self.root)

    def _find_max(self, node):
        if node is None:
            return None
        if node.right is None:
            return node
        else:
            return self._find_max(node.right)
```

Test:

- Construct a tree with values [50, 30, 70, 15, 35, 62, 87, 7, 22, 31]

- Find nodes with minimum and maximum values

In [19]:

```python
t = BinarySearchTree2()
s = [50, 30, 70, 15, 35, 62, 87, 7, 22, 31]
for i in s:
    t.add(i)
# t.print_tree()

mi = t.find_min()
ma = t.find_max()

print(mi, ma)
```

```
7(,) 87(,)
```

5. Delete a Node (Optional)

Another **common operation** of a binary search tree is to delete a node from the tree.

This operation is more complicated because it may involve joining subtree if the deleted node is not a leaf node.

There are **3** scenarios to delete a node from a tree.

- Leaf Node, e.g. 7, 22, 31, 62, 87
- Node with 1 child, e.g. 35
- Node with 2 children, e.g. 15, 30, 50, 70

```
              50
           /      \
          30       70
        /   \     /   \
       15    35  62    87
     / \    /
    7   22 31
```

Find Parent Node

To join subtree, we need to know parent of the node to be delete.

In [20]:

```python
def _find_parent(parent, node, val):
    if node is None:
        return None
    if val == node.data:
        return parent
    elif val < node.data:
        return _find_parent(node, node.left, val)
    else:
        return _find_parent(node, node.right, val)
```

Test:

- Find parent node of the node with value `87`

In [21]:

```
# t.print_tree()
result = _find_parent(None, t.root, 87)
print(result)
```

```
70(62,87)
```

Skeleton Function to Delete Node

The `_delete()` function has the same skeleton as the `_find_parent()` function.

- Return `True` or `False` to indicate whether deletion is successful.
- Handle 3 cases when `val == node.data`.

In [22]:

```
def _delete(parent, node, val):
    if node is None:
        return False
    if val == node.data:
        if node.left and node.right:
            print('Node with 2 children')
            pass
        elif node.left or node.right:
            print('Node with single child')
            pass
        else:
            print('Leaf node')
            pass
        return True
    elif val < node.data:
        return _delete(node, node.left, val)
    else:
        return _delete(node, node.right, val)
```

Handle 3 Cases

Leaf Node

- Deleting the node alone is enough and no additional change is needed.
- To delete the node, set respective child attribute of parent node to `None`.

```
if parent.left == node:
    parent.left = None
else:
    parent.right = None
```

Node with Single Child

- Set the child of current node to be the child of parent node. No additional change is needed.

```
child = node.left if node.left else node.right
if parent.left == node:
    parent.left = child
else:
    parent.right = child
```

Node with 2 Children

- Find the smallest node `temp` in the right subtree of the current node
- Replace the value of current node with value of `temp`
- Delete the `temp` node

```
temp = _find_min(node.right)
node.data = temp.data
_delete(node, node.right, node.data)
```

<u>Exercise:</u>

Update the `_delete()` function with above code snippets.

In [23]:

```python
def _delete(parent, node, val):
    if node is None:
        return False
    if val == node.data:
        if node.left and node.right:
            print('Node with 2 children')
            temp = _find_min(node.right)
            node.data = temp.data
            _delete(node, node.right, node.data)
        elif node.left or node.right:
            print('Node with single child')
            child = node.left if node.left else node.right
            if parent.left == node:
                parent.left = child
            else:
                parent.right = child
        else:
            print('Leaf node')
            if parent.left == node:
                parent.left = None
            else:
                parent.right = None
        return True
    elif val < node.data:
        return _delete(node, node.left, val)
    else:
        return _delete(node, node.right, val)
```

<u>Test:</u>

- Construct a tree with values [50, 30, 70, 15, 35, 62, 87, 7, 22, 31]

In [24]:

```
t = BinarySearchTree2()
s = [50, 30, 70, 15, 35, 62, 87, 7, 22, 31]
for i in s:
    t.add(i)

t.print_tree()
```

```
50(30,70)
30(15,35) 70(62,87)
15(7,22) 35(31,) 62(,) 87(,)
7(,) 22(,) 31(,)
```

Leaf Node: Try to delete value 7, 22 and 31, one at a time.

In [25]:

```
import copy
x = copy.deepcopy(t)

_delete(None, x.root, 31)
x.print_tree()
```

```
Leaf node
50(30,70)
30(15,35) 70(62,87)
15(7,22) 35(,) 62(,) 87(,)
7(,) 22(,)
```

Node with Single Child: Try to delete value 35.

In [26]:

```
import copy
x = copy.deepcopy(t)

_delete(None, x.root, 35)
x.print_tree()
```

```
Node with single child
50(30,70)
30(15,31) 70(62,87)
15(7,22) 31(,) 62(,) 87(,)
7(,) 22(,)
```

Node with 2 Children: Try to delete value 15, 30, 50 and 70, one at a time.

```
              50
           /      \
          30       70
        /  \      /  \
       15   35  62   87
     / \    /
    7  22  31
```

In [27]:

```
import copy
x = copy.deepcopy(t)

_delete(None, x.root, 70)
x.print_tree()
```

```
Node with 2 children
Leaf node
50(30,87)
30(15,35) 87(62,)
15(7,22) 35(31,) 62(,)
7(,) 22(,) 31(,)
```

Final BST Class

Enhance `BinarySearchTree2` with `delete()` function to delete a node by value. Name the class `BinarySearchTree3` .

In [28]:

```
class BinarySearchTree3(BinarySearchTree2):

    def delete(self, val):
        return self._delete(None, self.root, val)

    def _delete(self, parent, node, val):
        if node is None:
            return False
        if val == node.data:
            if node.left and node.right:
                print('Node with 2 children')
                temp = self._find_min(node.right)
                node.data = temp.data
                self._delete(node, node.right, node.data)
            elif node.left or node.right:
                print('Node with single child')
                child = node.left if node.left else node.right
                if parent.left == node:
                    parent.left = child
                else:
                    parent.right = child
            else:
                print('Leaf node')
                if parent.left == node:
                    parent.left = None
                else:
                    parent.right = None
            return True
        elif val < node.data:
            return self._delete(node, node.left, val)
        else:
            return self._delete(node, node.right, val)
```

Test:

In [29]:

```
t = BinarySearchTree3()
s = [50, 30, 70, 15, 35, 62, 87, 7, 22, 31]
for i in s:
    t.add(i)

t.delete(70)
t.print_tree()
```

```
Node with 2 children
Leaf node
50(30,87)
30(15,35) 87(62,)
15(7,22) 35(31,) 62(,)
7(,) 22(,) 31(,)
```

6. BST with Duplicate Values (Optional)

What if I still need to be able to store duplicate values in the Binary Search Tree?

Possible Solution:

Add an attribute `count` to `Node` class. The count represent how many duplicate values (same as `data`) are in the tree.

- Insertion and deletion of duplicate values will increase or decrease the `count` value.
- Node will be removed from tree when its `count` value is 0.

Reference

Delete node in Binary Search Tree

- [https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/ (https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/)](https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/)
- [https://www.geeksforgeeks.org/binary-tree-data-structure/ (https://www.geeksforgeeks.org/binary-tree-data-structure/)](https://www.geeksforgeeks.org/binary-tree-data-structure/)