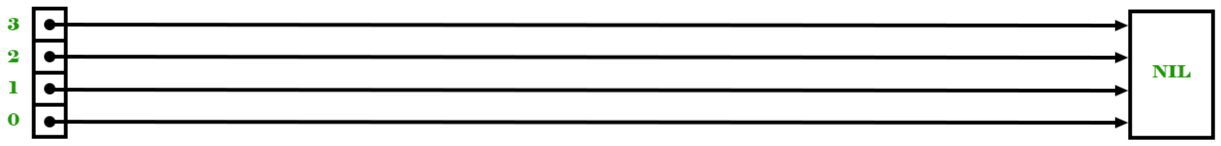
Homework 4

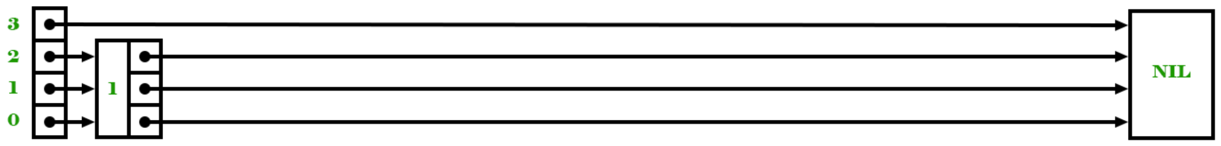
Exercise 1

Let us insert 1, 5, 7, 30, 2, 4, 18, 32, 34, 36 into a skip-list with max_level = 3 and p = 0.5 (tossing an actual coin).
For all insertions the general idea is the same: inserting the node into its position in each sorted list whose level $\leq$ level of the node.

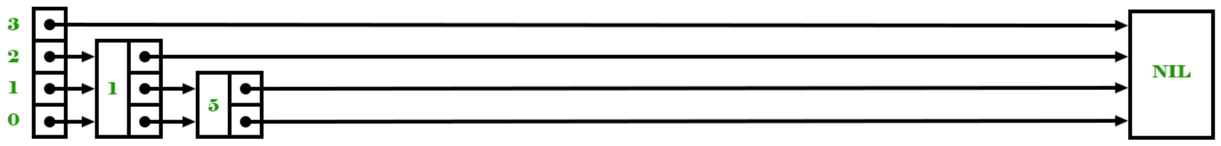
0. Empty list.



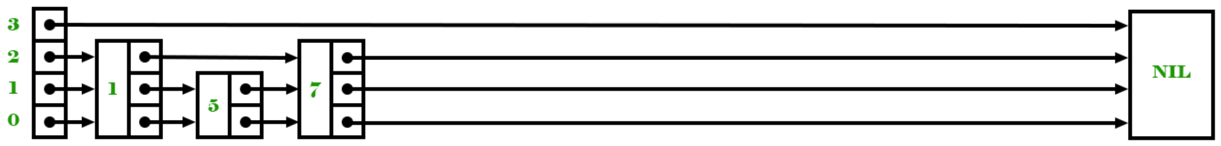
1. Inserting 1, level 2.



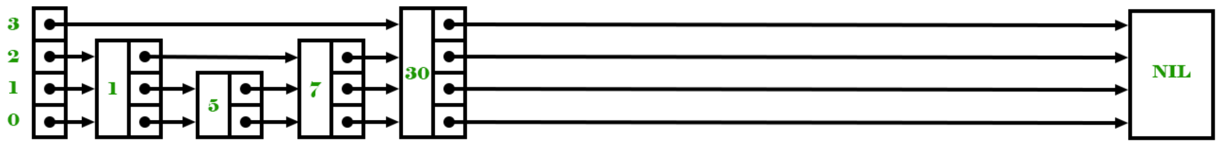
2. Inserting 5, level 1.



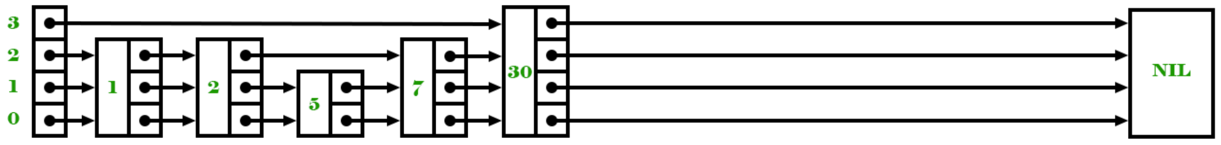
3. Inserting 7, level 2.



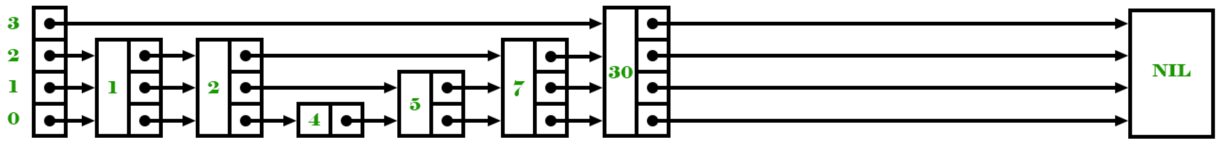
4. Inserting 30, level 3.



5. Inserting 2, level 2.

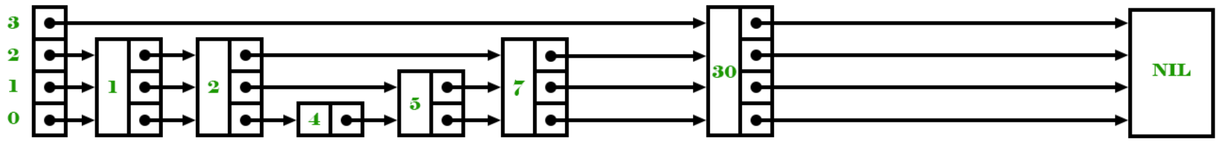


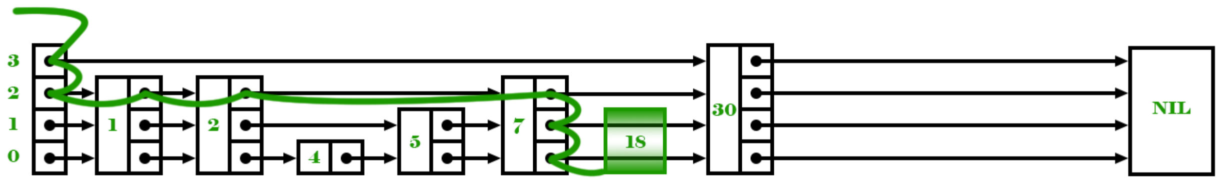
6. Inserting 4, level 0.

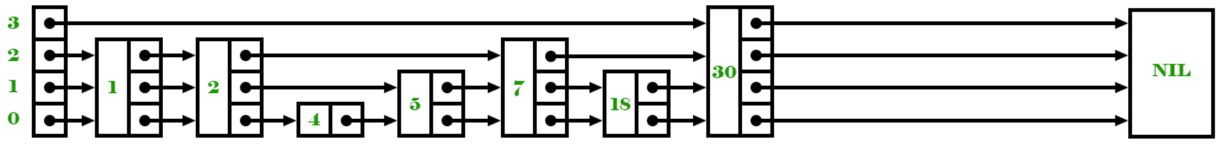


Now, let us see how 18 is added to the list and what comparisons are done.

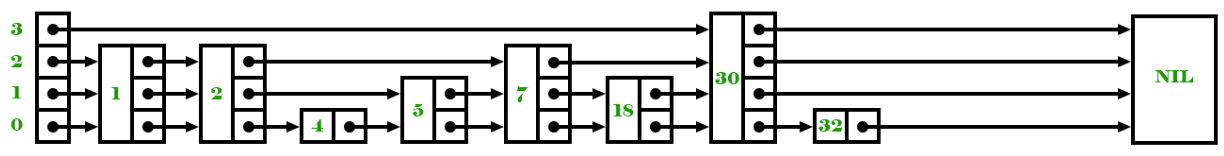
7. Inserting 18, level 1.



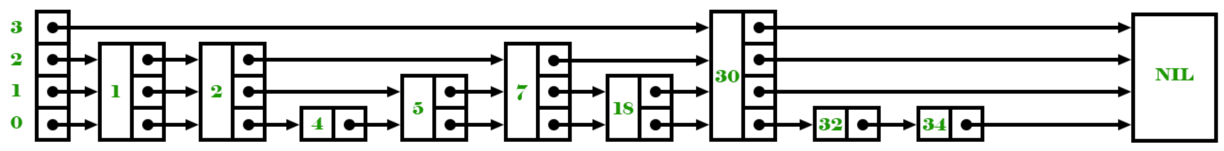




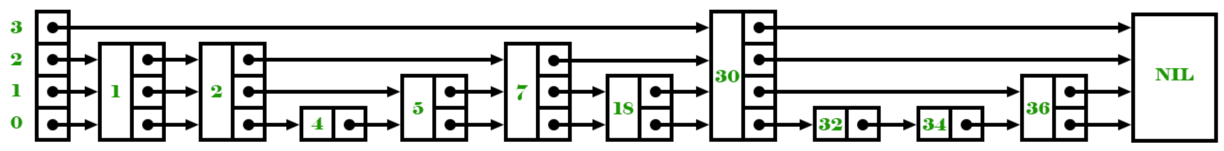
8. Inserting 32, level 0.



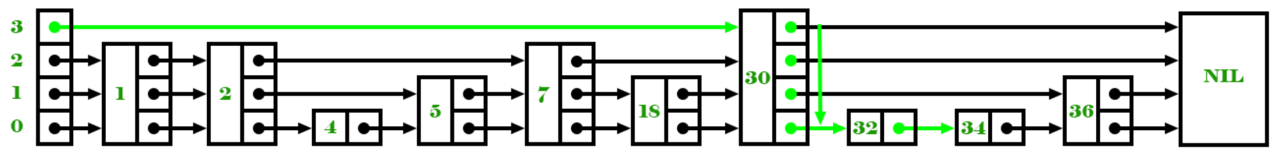
9. Inserting 34, level 0.



10. Inserting 36, level 1.



Fine! Now, let us simulate search for 34.



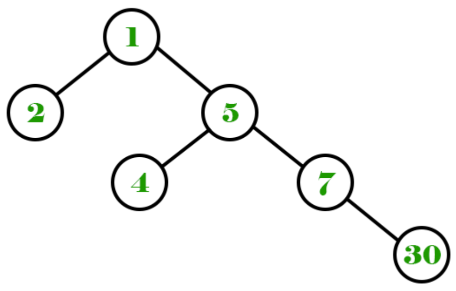
We see that there are 6 comparisons performed (in fact, comparisons happen each time the bullet (pointer) is lighted with green).

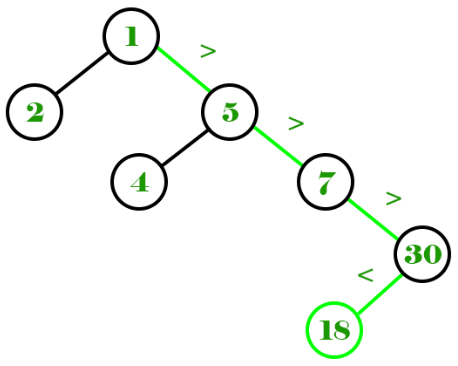
Exercise 2

Let us insert the same values into the Binary Search Tree.
We shall not provide all steps. For all insertions the general idea is the same: comparing the be-inserted value to the value in current node; if less - going to the left subtree, if greater - going to the right subtree, if equal - stop; repeating until the free (None) slot is found.

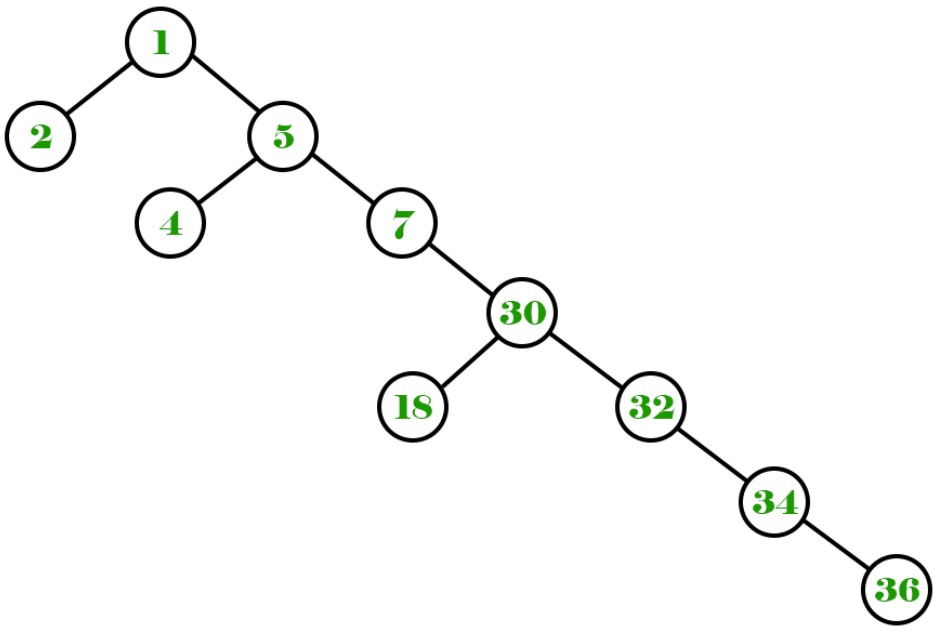
Let us insert 18 into the preliminary built tree.

Before insertion **After insertion**

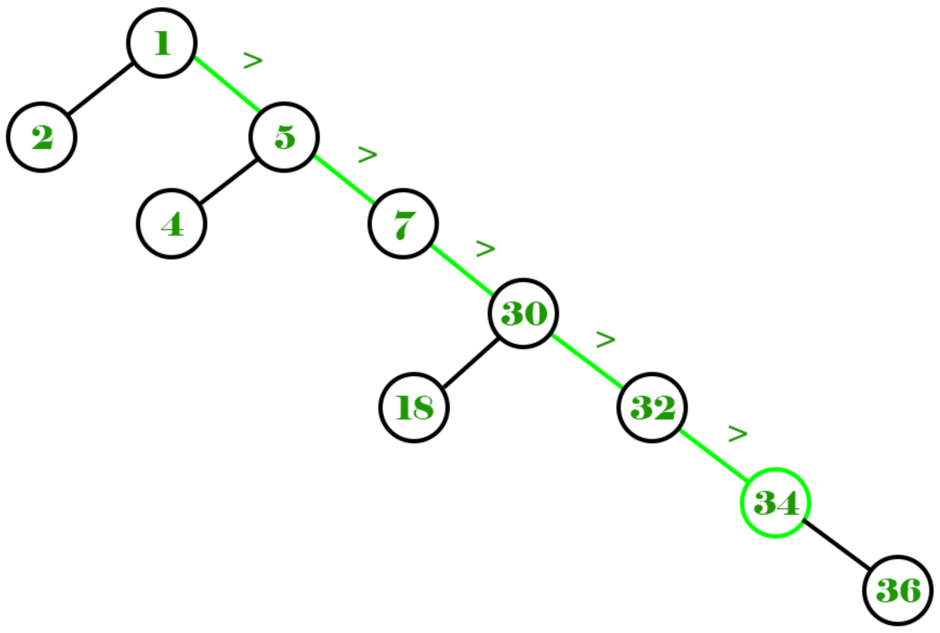




Let us see the final tree. It is very skewed. The biggest difference of depths of subtrees is 5.



Let us simulate search for 34. We see that there are 6 comparisons performed.

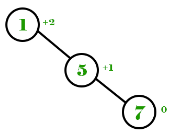


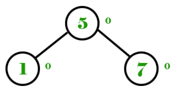
Exercise 3

Let us insert the same values into the AVL Tree.
We shall show only the steps when balancing is performed.

1. Inserting 7. The difference of heights in the root node is now +2, so re-balancing is needed. It is the Right-Right case, so we do Left Rotation aroung the grandfather.

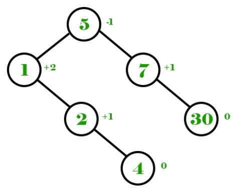
Before balancing **After balancing**

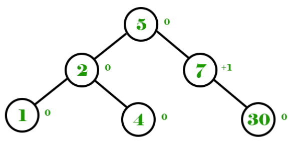




2. Inserting 4. The difference of heights in the node 1 is now +2, so re-balancing is needed. It is the Right-Right case, so we do Left Rotation aroung the grandfather.

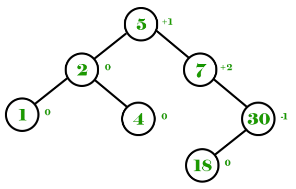
Before balancing **After balancing**



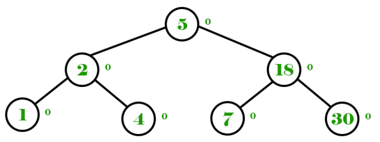


3. Inserting 18. The difference of heights in the node 7 is now +2, so re-balancing is needed. It is the Right-Left case, so at first we do Right Rotation aroung the father and then Left Rotation around the grandfather.

Before balancing

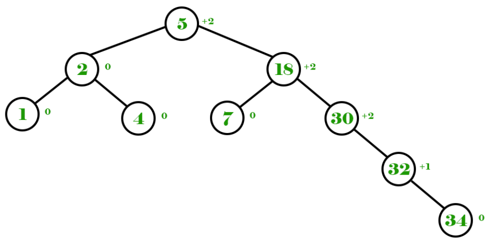


After balancing

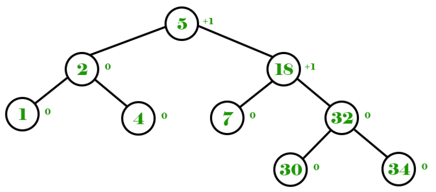


4. Inserting 34. The difference of heights in the node 30 is now +2, so re-balancing is needed. It is the Right-Right case, so we do Left Rotation aroung the grandfather.

Before balancing



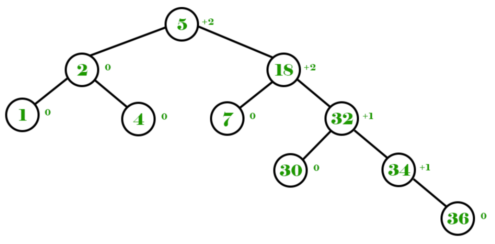
After balancing

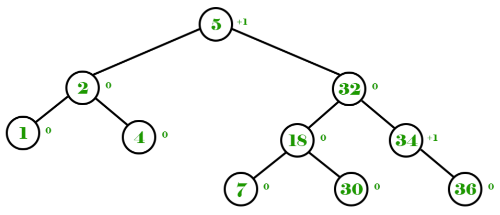


5. Inserting 36. The difference of heights in the node 18 is now +2, so re-balancing is needed. It is the Right-Right case, so we do Left Rotation of grandfather aroung the great-grandfather.

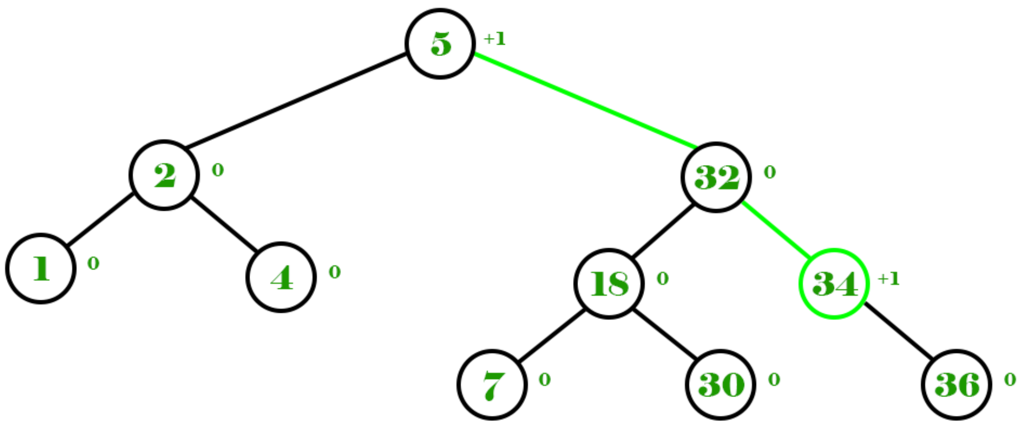
Before balancing

After balancing





Let us search for 34. We see that there are 3 comparisons performed.



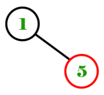
Exercise 4

Let us insert the same values into the Red-Black Tree.

1. Inserting 1 and 5. The root is always black. The new node is always red.

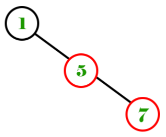
Inserting 1

Inserting 5

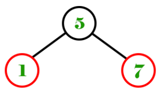


2. Inserting 7. There are two adjacent red nodes, so re-balancing is needed. The uncle is black and configuration is Right-Right, so we do Left Rotation around the grandfather and swap colors.

Before balancing

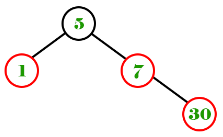


After balancing

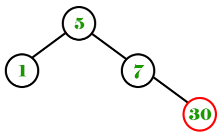


3. Inserting 30. There are two adjacent red nodes, so re-balancing is needed. The uncle is red, so we can just swap colors. Additionally, we re-color the root in black.

Before balancing

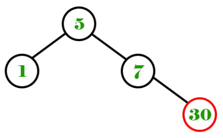


After balancing

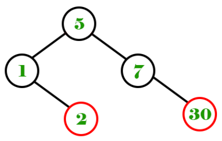


4. Inserting 2. No re-balance needed.

Before inserting 2



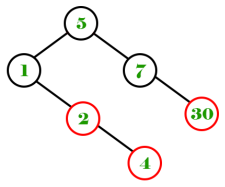
After inserting 2

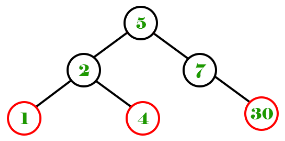


5. Inserting 4. There are two adjacent red nodes, so re-balancing is needed. The uncle is black and configuration is Right-Right, so we do Left Rotation around the grandfather and swap colors.

Before balancing

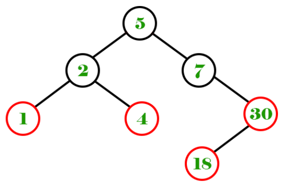
After balancing

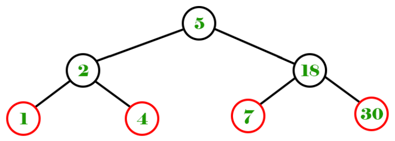




6. Inserting 18. There are two adjacent red nodes, so re-balancing is needed. The uncle is black and configuration is Right-Left, so at first we do Right Rotation around the father, then Left Rotation around the grandfather and then swap colors.

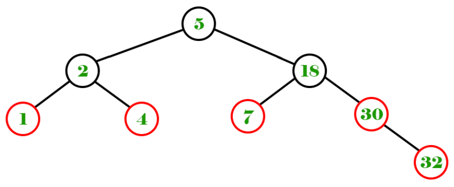
Before balancing **After balancing**

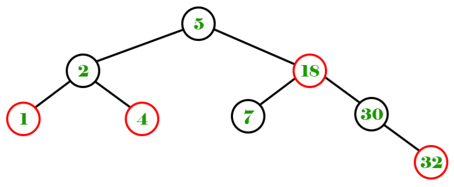




7. Inserting 32. There are two adjacent red nodes, so re-balancing is needed. The uncle is red, so we can just swap colors.

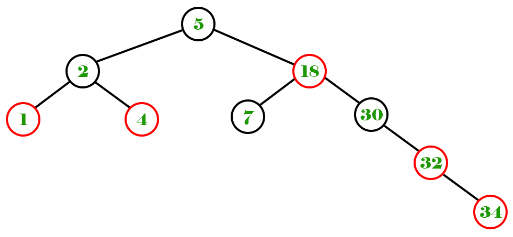
Before balancing **After balancing**

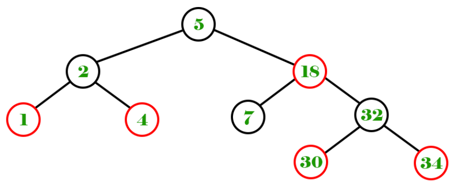




8. Inserting 34. There are two adjacent red nodes, so re-balancing is needed. The uncle is black and configuration is Right-Right, so we do Left Rotation around the grandfather and swap colors.

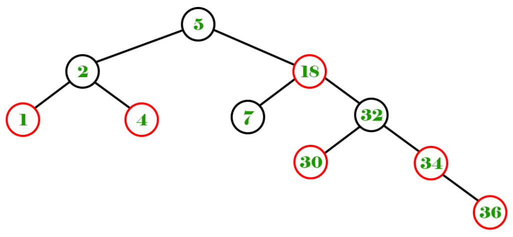
Before balancing **After balancing**



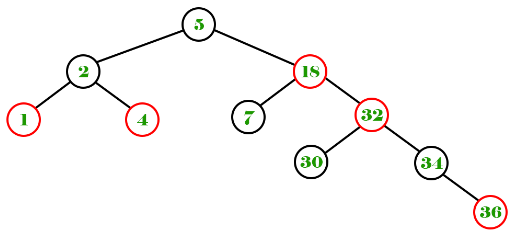


9. Inserting 36. There are two adjacent red nodes, so re-balancing is needed. The uncle is red, so we can just swap colors.

Before first balancing

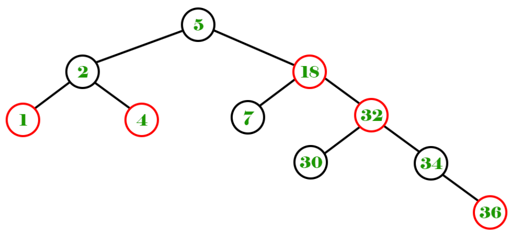


After first balancing

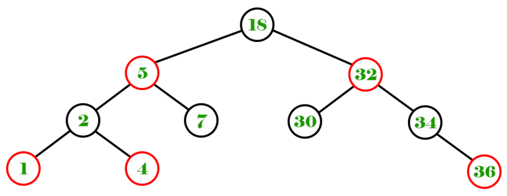


Now, let us consider the grandfather. There are two adjacent red nodes, so re-balancing is needed. The uncle is black and configuration is Right-Right, so we do Left Rotation around the grandfather and swap colors.

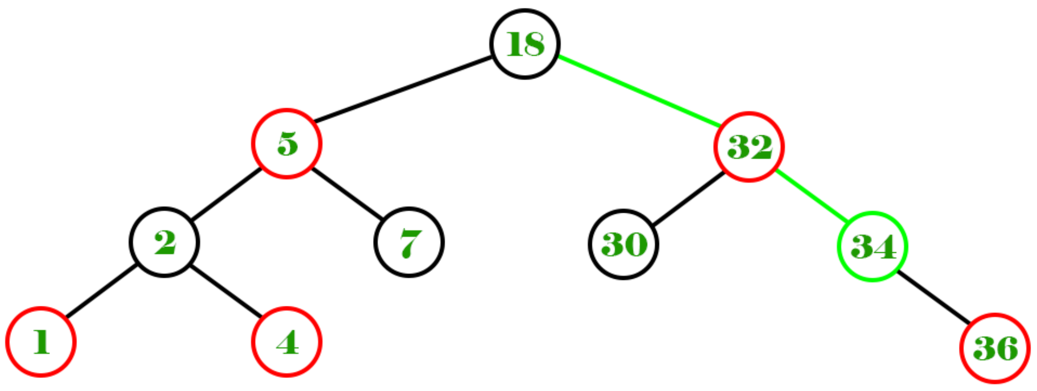
Before second balancing



After second balancing



Let us search for 34. We see that there are 3 comparisons performed.



Exercise 5

Let us write a program to build a K-d (2D) Tree, plot it and perform a search of points in an arbitrary interval.

In [167]:

```
import matplotlib.pyplot as plt
import numpy as np
import operator

from collections import deque
```

In [174]:

```python
class KdNode:

    def __init__(self, point, dim, left, right):
        self.point = point
        self.dim = dim
        self.left = left
        self.right = right

    def __str__(self, level=0, is_left=None) :
        string = "             " * level

        if is_left is not None:
            if is_left:
                string += "l: "
            else:
                string += "r: "
        else:
            string += "root: "

        string += f"({self.point[0]}, {self.point[1]})" + "\n"

        if self.left is not None:
            string += self.left.__str__(level + 1, True)
        if self.right is not None:
            string += self.right.__str__(level + 1, False)

        return string


class KdTree:

    def __init__(self, points, bounds):
        self.root = KdTree.build_recursive(0, points)
        self.bounds = bounds

    def build_recursive(dim, points):
            if len(points) == 0:
                return None

            points.sort(key=operator.itemgetter(dim))
            m = len(points) // 2
            median = points[m]
            while m + 1 < len(points) and points[m + 1][dim] == median[dim]:
                m += 1
            median = points[m]


            dim2 = (dim + 1) % 2
            return KdNode(median, dim, KdTree.build_recursive(dim2, points[:m]),
                                       KdTree.build_recursive(dim2, points[(m + 1):]))

    def seek_interval(self, xbounds, ybounds):
        result = []
        bounds = [xbounds, ybounds]
        unvisited = deque()
        unvisited.append(self.root)

        while len(unvisited) != 0:
            current = unvisited.popleft()
            dim = current.dim

            if current.point[dim] < bounds[dim][0]:
                if current.right is not None:
                    unvisited.append(current.right)
            elif current.point[dim] > bounds[dim][1]:
                if current.left is not None:
                    unvisited.append(current.left)
            else:
                dim = (dim + 1) % 2
                if bounds[dim][0] <= current.point[dim] <= bounds[dim][1]:
                    result.append(current.point)
                if current.left is not None:
                    unvisited.append(current.left)
                if current.right is not None:
                    unvisited.append(current.right)

        return result

    def plot(self):
        plt.rcParams["figure.figsize"] = (10, 10)
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title('2D Tree')

        plt.xlim(self.bounds)
        plt.ylim(self.bounds)

        if self.root is None:
            return

        KdTree.plot_recursive(self, self.root, self.bounds, self.bounds)
        plt.show()

    def plot_recursive(self, current, xbounds, ybounds):
        if current is None:
            return

        if current.dim == 0:
            KdTree.plot_recursive(self, current.left, [xbounds[0], current.point[0]], ybou
            KdTree.plot_recursive(self, current.right, [current.point[0], xbounds[1]], ybou
            plt.plot([current.point[0], current.point[0]], [ybounds[0], ybounds[1]], color=
        else:
            KdTree.plot_recursive(self, current.left, xbounds, [ybounds[0], current.point[1
            KdTree.plot_recursive(self, current.right, xbounds, [current.point[1], ybounds[
            plt.plot([xbounds[0], xbounds[1]], [current.point[1], current.point[1]], color=

    def __repr__(self):
        return str(self.root)
```

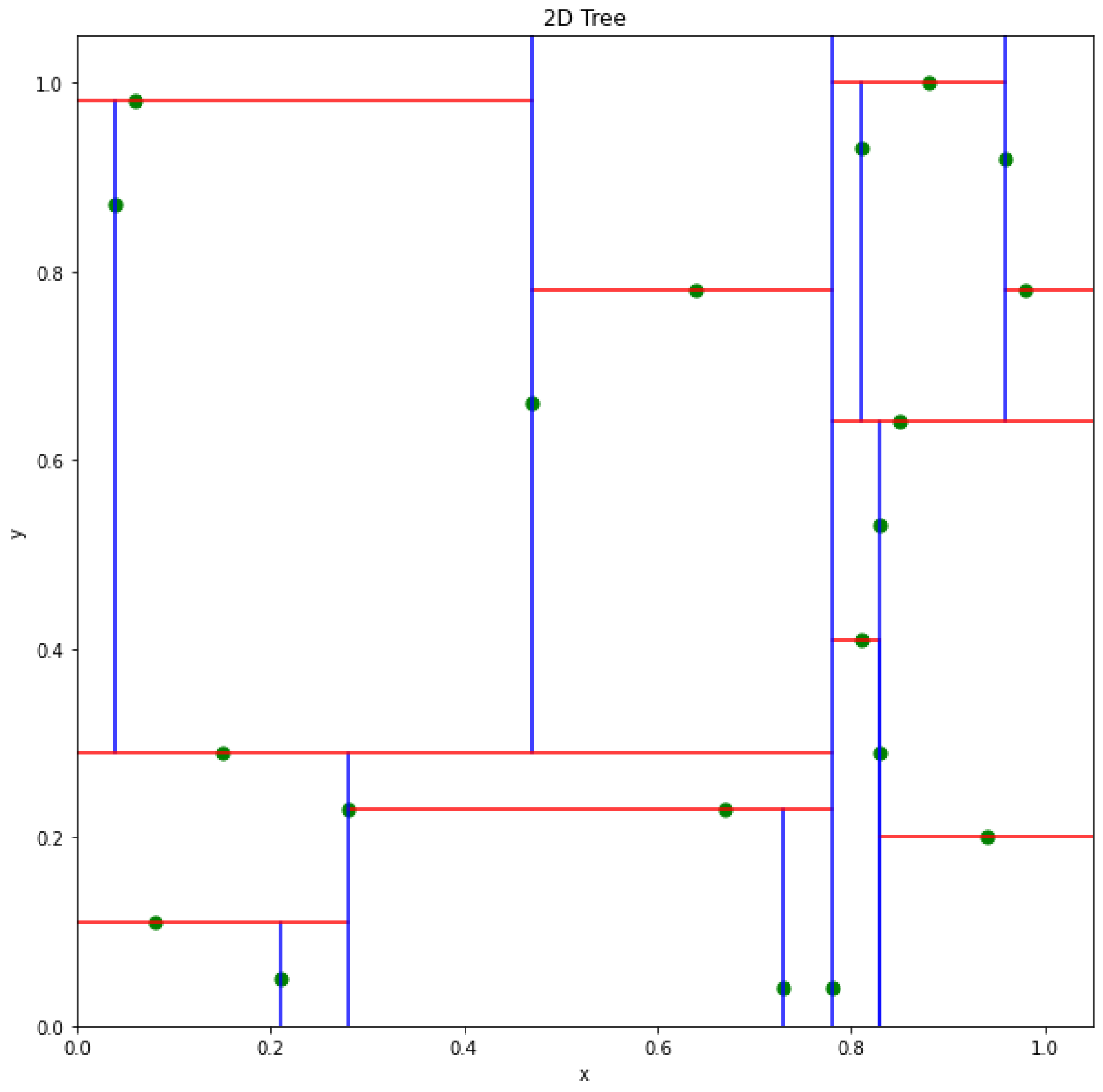
Let us plot the final tree.

In [176]:

```
points = [(0.67, 0.23), (0.06, 0.98), (0.28, 0.23), (0.83, 0.29), (0.83, 0.53), (0.78, 0.04
          (0.15, 0.29), (0.96, 0.92), (0.88, 1.0), (0.64, 0.78), (0.81, 0.41), (0.04, 0.87)
          (0.81, 0.93), (0.47, 0.66), (0.85, 0.64), (0.98, 0.78), (0.73, 0.04), (0.08, 0.11
```

In [177]:

```
tree = KdTree(points, (0.0, 1.05))
np_points = np.array(points)
plt.scatter(np_points[:, 0], np_points[:, 1], s=50, color='green')
tree.plot()
```



In [178]:

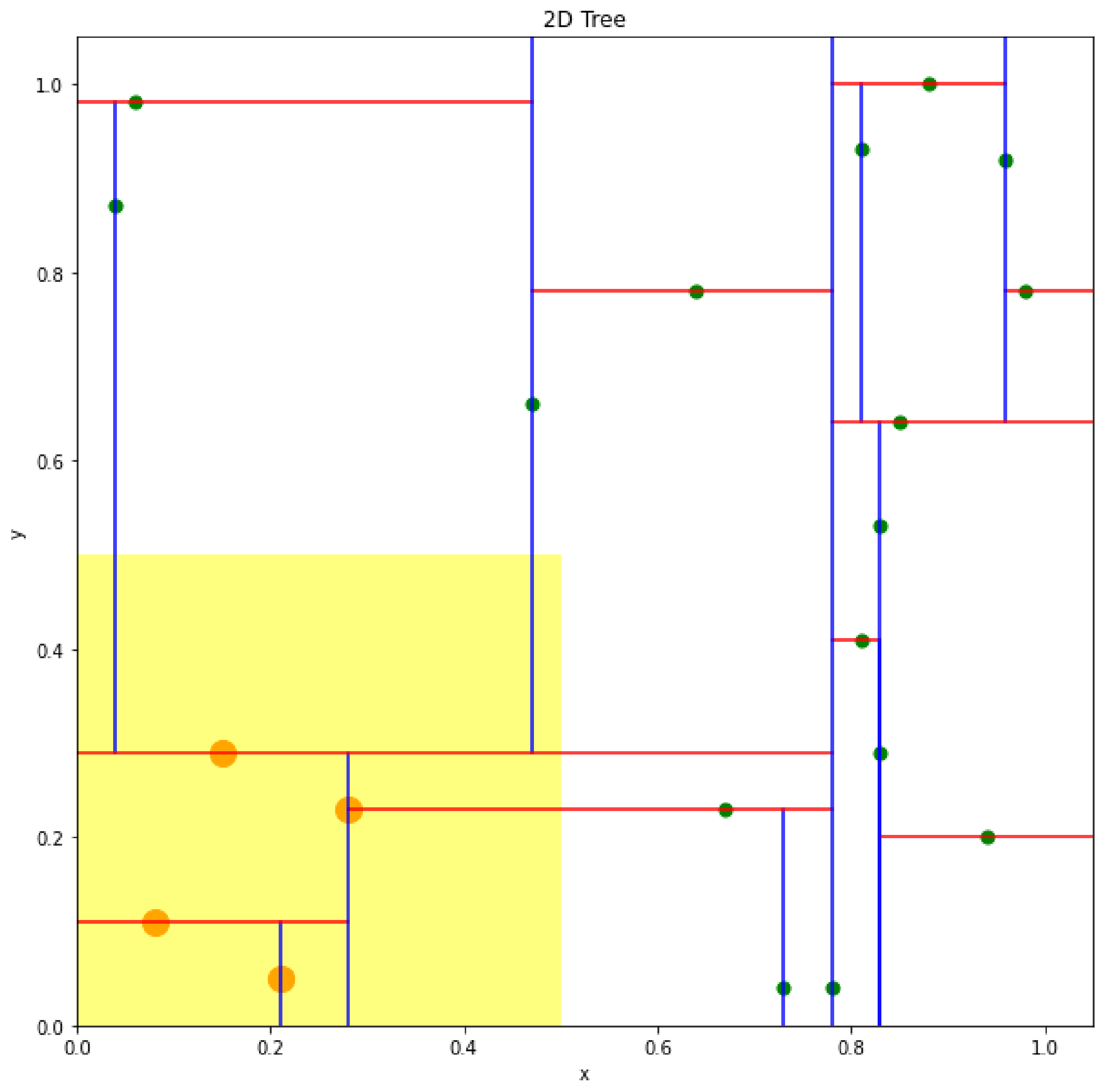
```
print(tree)
```

```
root: (0.78, 0.04)
            l: (0.15, 0.29)
                        l: (0.28, 0.23)
                                    l: (0.08, 0.11)
                                                l: (0.21, 0.05)
                                    r: (0.67, 0.23)
                                                l: (0.73, 0.04)
                        r: (0.47, 0.66)
                                    l: (0.06, 0.98)
                                                l: (0.04, 0.87)
                                    r: (0.64, 0.78)
            r: (0.85, 0.64)
                        l: (0.83, 0.53)
                                    l: (0.81, 0.41)
                                                l: (0.83, 0.29)
                                    r: (0.94, 0.2)
                        r: (0.96, 0.92)
                                    l: (0.88, 1.0)
                                                l: (0.81, 0.93)
                                    r: (0.98, 0.78)
```

Let us perform a search of points in the interval [[0, 0.5], [0, 0.5]].

In [190]:

```
found = np.array(tree.seek_interval([0, 0.5], [0, 0.5]))

plt.gca().add_patch(plt.Rectangle([0, 0], 0.5, 0.5, facecolor='yellow', alpha=0.5, fill=Tru
plt.scatter(np_points[:, 0], np_points[:, 1], s=50, color='green')
plt.scatter(found[:, 0], found[:, 1], s=200, color='orange')
tree.plot()
```



Exercise 6

Let us write a program to build the Binary Search Tree and perform insertion and search.

In [228]:

```
class Node:

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

    def __str__(self, level=0, is_left=None) :
        string = "   " * level

        if is_left is not None:
            if is_left:
                string += "l: "
            else:
                string += "r: "
        else:
            string += "root: "

        string += str(self.value) + "\n"

        if self.left is not None:
            string += self.left.__str__(level + 1, True)
        if self.right is not None:
            string += self.right.__str__(level + 1, False)

        return string



class BST:

    def __init__(self):
        self.root = None

    def append(self, value):
        if self.root is None:
            self.root = Node(value)
            return

        node = Node(value)
        current = self.root

        while True:
            if value < current.value:
                if current.left is None:
                    current.left = node
                    return True
                else:
                    current = current.left
            elif value > current.value:
                if current.right is None:
                    current.right = node
                    return True
                else:
                    current = current.right
            else:
                return False

    def contains(self, value):
        if self.root is None:
            return False

        current = self.root

        while current is not None:
            if value < current.value:
                    current = current.left
            elif value > current.value:
                    current = current.right
            else:
                return True

        return False

    def height(self):
        return BST.max_subtree_height(self.root)

    def max_subtree_height(root):
        if root is None:
            return 0
        return np.max([BST.max_subtree_height(root.left), BST.max_subtree_height(root.right

    def __repr__(self):
        return str(self.root)
```

Let us insert random values and check correctness of our implementation.

In [229]:

```
np.random.seed(201)
```

In [230]:

```
bst = BST()

values = np.arange(5)
np.random.shuffle(values)

for x in values:
    bst.append(x)

print('---BST---')
print(bst)
print(f"Height: {bst.height()}")
```

```
---BST---
root: 2
  l: 1
    l: 0
  r: 3
    r: 4

Height: 3
```

In [218]:

```
print(f"BST contanins 0: {bst.contains(0)}")
print(f"BST contains 200: {bst.contains(200)}")
```

```
BST contanins 0: True
BST contains 200: False
```

Let us find the average height of a tree containing 1000 nodes. We shall take an average result of 1000 experiments, where we insert randomly shuffled range of values of size 1000.

In [233]:

```
n_values = 1000
n_tests = 1000
heights = [0] * n_tests

values = np.arange(n_values)

for i in range(n_tests):
    bst = BST()
    np.random.shuffle(values)

    for x in values:
        bst.append(x)

    heights[i] = bst.height()

print(f"Average BST height ({n_values} values) over {n_tests} random shuffles: {int(round(n
```

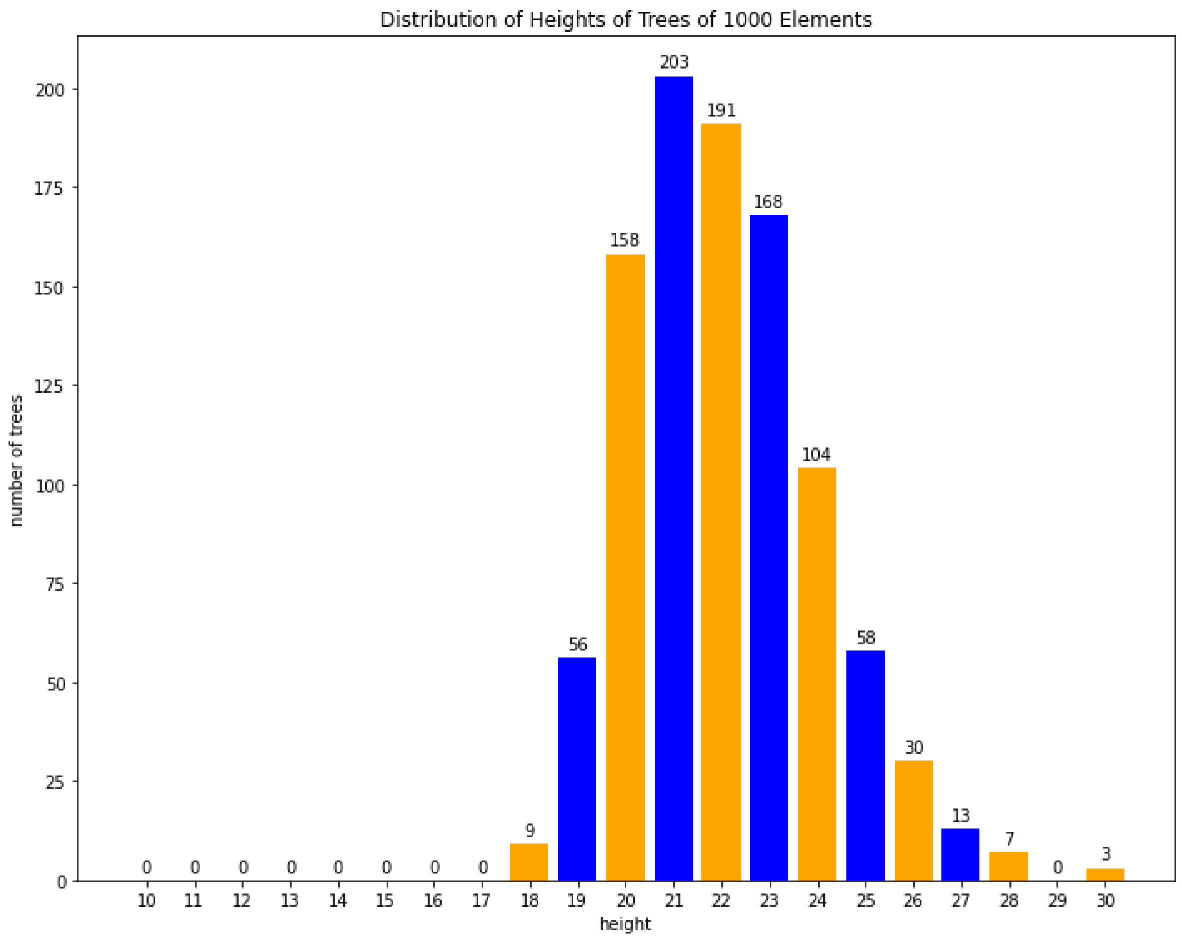
```
Average BST height (1000 values) over 1000 random shuffles: 22
```

So, the average height is 22. It is far more than the optimal (balanced) tree height $\log_2 1000 + 1 \approx 11$.

Let us see the distribution of heights in the experiment.

In [240]:

```
distribution = np.bincount(heights)

plt.rcParams["figure.figsize"] = (10, 8)
plt.xlabel('height')
plt.ylabel('number of trees')
plt.title(f"Distribution of Heights of Trees of {n_values} Elements")

labels = [str(i) for i in range(10, len(distribution))]
x = np.arange(10, len(distribution))
distribution = distribution[10:]

rects = plt.bar(x, distribution, color=['orange', 'blue'])
plt.xticks(x, labels)
plt.bar_label(rects, padding=3)

plt.tight_layout()
plt.show()
```



We see that there has been no configuration even close to the optimal one with height 11.

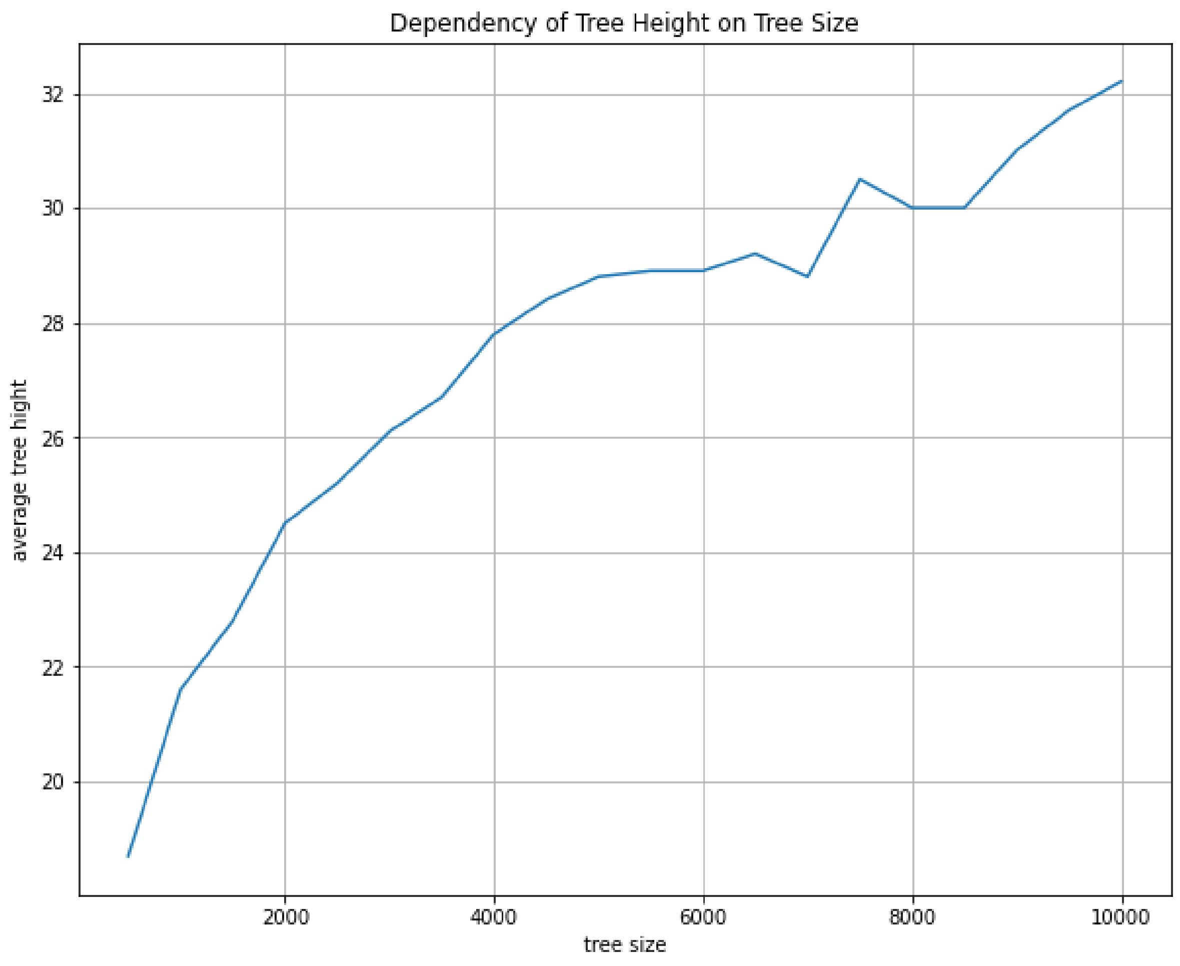
Let us plot the average tree height over the size of the tree.

In [241]:

```
n_tests = 20
n_values = [i * 500 for i in range(1, n_tests + 1)]
n_attempts = 10
heights = [0] * n_tests

for i in range(n_tests):
    values = np.arange(n_values[i])
    temp_heights = [0] * n_attempts

    for j in range(n_attempts):
        bst = BST()
        np.random.shuffle(values)

        for x in values:
            bst.append(x)

        temp_heights[j] = bst.height()

    heights[i] = np.mean(temp_heights)
```

In [244]:

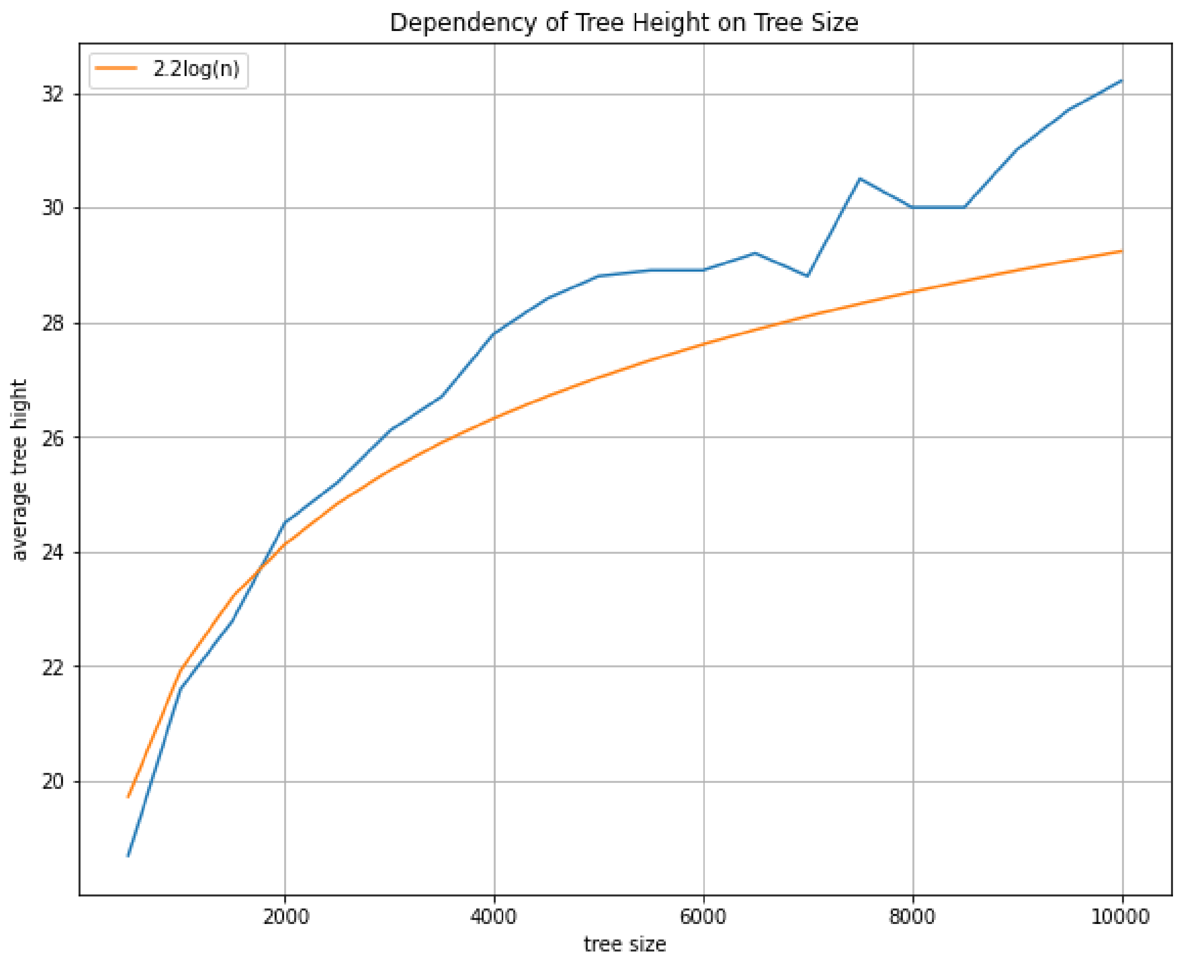
```
plt.plot(n_values, heights)
plt.xlabel('tree size')
plt.ylabel('average tree hight')
plt.title('Dependency of Tree Height on Tree Size')
plt.grid(True)

plt.show()
```



We see that the plot roughly resembles the log-function. Let us manually choose constants and compare the plots.

In [267]:

```
plt.plot(n_values, heights)
plt.plot(n_values, [2.2 * np.log2(x) for x in n_values], label='2.2log(n)')
plt.xlabel('tree size')
plt.ylabel('average tree hight')
plt.title('Dependency of Tree Height on Tree Size')
plt.grid(True)
plt.legend()

plt.show()
```



Exercise 7

Let us only describe the strategies of stress-testing as the task requires it.

General idea

Let us **enumerate the rungs** of the ladder from the lowest one as 0 to the highest one as $n-1$. If for arbitraty $0 \leq k < n$ the glass cup dropped from k-th rung breaks, we will consider the the cup is "less than" the k-th rung. Otherwise the cup will be considered as "greater than" the k-th rung. Now, we can compile the rungs (the indices) in an **ordered list** and perform a **search of the "closest left neighbour"** of the cup (we have a **comparison function** described previously). For instance, we can do it with **binary search** for $O(\log n)$ time.

The problem is that the number of cups is limited. It means that without at least $\log n + 1$ cups we risk to break all of them before finding the right rung. We should find the strategy that will allow us to **minimise the number of breaks**.

k = 2

The workaround is a **skip-list with 2 levels**. Firstly, we search the rung in the express lane (a subset of rungs) until we find the greater element (break the cup) or reach the end. Then we continue the search in the normal lane successively until the cup is broken or we reach the end of the lane which both mean that the right rung is found. For all this, as we see, two cups are enough. The time complexity of such search is $O(\log n)$, so we will drop the cups less than $A \log n$ times for some constant A.

k > 2

The workaround is a **skip-list with 2+ levels**. The strategy is the same: each cup is reserved for a certain lane.