Lab_IV

March 21, 2024

1 Part 1.1

[ ]:
```python
# imports
from __future__ import annotations
from typing import List, Dict, Tuple, Any, Optional
import math
import sys
import random
import matplotlib.pyplot as plt
```

[ ]:
```python
class RBNode:

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None
        self.colour = "R"

    def get_uncle(self):
        return

    def is_leaf(self):
        return self.left == None and self.right == None

    def is_left_child(self):
        return self == self.parent.left

    def is_right_child(self):
        return not self.is_left_child()

    def is_red(self):
        return self.colour == "R"

    def is_black(self):
        return not self.is_red()
```

```
    def make_black(self):
        self.colour = "B"

    def make_red(self):
        self.colour = "R"

    def get_brother(self):
        if self.parent.right == self:
            return self.parent.left
        return self.parent.right


class RBNode:

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.parent = None
        self.colour = "R"

    def get_uncle(self):
        return

    def is_leaf(self):
        return self.left == None and self.right == None

    def is_left_child(self):
        return self == self.parent.left

    def is_right_child(self):
        return not self.is_left_child()

    def is_red(self):
        return self.colour == "R"

    def is_black(self):
        return not self.is_red()

    def make_black(self):
        self.colour = "B"

    def make_red(self):
        self.colour = "R"

    def get_brother(self):
        if self.parent.right == self:
```

```
            return self.parent.left
        return self.parent.right
    def rotate_right(self, root):
        parent = self.parent
        leftChild = self.left

        self.left = leftChild.right

        if None != leftChild.right:
            leftChild.right.parent = self

        leftChild.parent = self.parent

        if None == self.parent:
            root = leftChild
        elif self == self.parent.right:
            self.parent.right = leftChild
        else:
            self.parent.left = leftChild


        leftChild.right = self
        self.parent = leftChild

        return root



    def rotate_left(self, root):
        # parent = self.parent
        rightChild = self.right

        self.right = rightChild.left

        if None != rightChild.left:
            rightChild.left.parent = self

        rightChild.parent = self.parent

        if None == self.parent:
            root = rightChild
        elif self == self.parent.left:
            self.parent.left = rightChild
        else:
            self.parent.right = rightChild

        rightChild.left = self
```

```
        self.parent = rightChild

        return root
```

[ ]:
```
class RBTree:

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

    def is_empty(self,):
        return self.root == None

    def get_height(self,):
        if self.is_empty():
            return 0
        return self.__get_height(self.root)

    def __get_height(self, node):
        if node == None:
            return 0
        return 1 + max(self.__get_height(node.left), self.__get_height(node.
 ↪right))

    def insert(self, value):
        if self.is_empty():
            self.root = RBNode(value)
            self.root.make_black()
        else:
            self.__insert(self.root, value)

    def __insert(self, node, value):
        if value < node.value:
            if node.left == None:
                node.left = RBNode(value)
                node.left.parent = node
                self.fix(node.left)
            else:
                self.__insert(node.left, value)
        else:
            if node.right == None:
                node.right = RBNode(value)
                node.right.parent = node
                self.fix(node.right)
            else:
                self.__insert(node.right, value)

    def fix(self, node):
```

```
        parent = node.parent

        if None == parent:
            node.make_black()
            return

        if parent.is_black():
            return

        grandparent = parent.parent

        if None == grandparent:
            parent.make_black()
            return

        uncle = node.get_uncle()

        if None != uncle and uncle.is_red():
            parent.make_black()
            grandparent.make_red()
            uncle.make_black()

            self.fix(grandparent)
        elif grandparent.left == parent:
            if parent.right == node:
                self.root = parent.rotate_left(self.root)
                parent = node

            self.root = grandparent.rotate_right(self.root)

            parent.make_black()
            grandparent.make_red()
        else:
            if parent.left == node:
                self.root = parent.rotate_right(self.root)
                parent = node

            self.root = grandparent.rotate_left(self.root)

            parent.make_black()
            grandparent.make_red()

    def __str__(self):
        if self.is_empty():
            return "[]"
        return "[" + self.__str_helper(self.root) + "]"
```

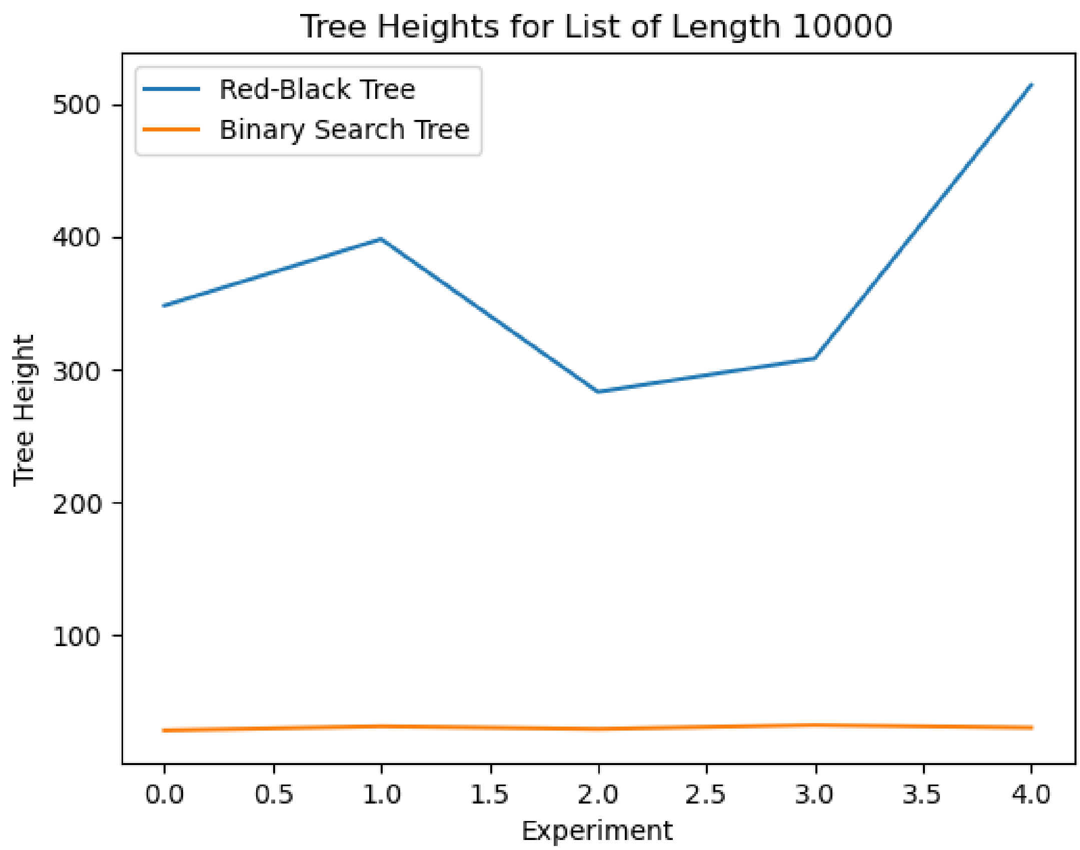
```
    def __str_helper(self, node):
        if node.is_leaf():
            return "[" + str(node) + "]"
        if node.left == None:
            return "[" + str(node) + " -> " + self.__str_helper(node.right) +␣
↪"]"
        if node.right == None:
            return "[" +  self.__str_helper(node.left) + " <- " + str(node) +␣
↪"]"
        return "[" + self.__str_helper(node.left) + " <- " + str(node) + " -> "␣
↪+ self.__str_helper(node.right) + "]"

    def print_tree(self):
        self.__print_helper(self.root, "", True)


    def __print_helper(self, node, indent, last):
        if node != None:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R----")
                indent += "     "
            else:
                sys.stdout.write("L----")
                indent += "|    "

            s_color = "RED" if node.is_red() else "BLACK"
            print(str(node.value) + "(" + s_color + ")")
            self.__print_helper(node.left, indent, False)
            self.__print_helper(node.right, indent, True)
```

2 Part 1.2

[ ]:
```
class BST:

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

    def is_empty(self,):
        return self.root == None

    def get_height(self,):
        if self.is_empty():
            return 0
        return self.__get_height(self.root)
```

```python
    def __get_height(self, node):
        if node == None:
            return 0
        return 1 + max(self.__get_height(node.left), self.__get_height(node.
↪right))

    def insert(self, value):
        if self.is_empty():
            self.root = RBNode(value)
            self.root.make_black()
        else:
            self.__insert(self.root, value)

    def __insert(self, node, value):
        if value < node.value:
            if node.left == None:
                node.left = RBNode(value)
                node.left.parent = node
            else:
                self.__insert(node.left, value)
        else:
            if node.right == None:
                node.right = RBNode(value)
                node.right.parent = node
            else:
                self.__insert(node.right, value)


def generateList(length):
    return [random.randint(1, 100000) for _ in range(length)]

def experiment(length):
    myList = generateList(length)

    rbTree = RBTree()
    bst = BST()

    for item in myList:
        rbTree.insert(item)
        bst.insert(item)

    rbTreeHeight = rbTree.get_height()
    bstHeight = bst.get_height()

    return rbTreeHeight, bstHeight
```

```python
def runExperiments(numExperiments, listLength):
    rbTreeHeights = []
    bstHeights = []

    for _ in range(numExperiments):
        rbTreeHeight, bstHeight = experiment(listLength)
        rbTreeHeights.append(rbTreeHeight)
        bstHeights.append(bstHeight)

    return rbTreeHeights, bstHeights

def plotGraph(rbTreeHeights, bstHeights, listLength):
    plt.plot(rbTreeHeights, label='Red-Black Tree')
    plt.plot(bstHeights, label='Binary Search Tree')
    plt.xlabel('Experiment')
    plt.ylabel('Tree Height')
    plt.title(f'Tree Heights for List of Length {listLength}')
    plt.legend()
    plt.show()


numExperiments = 5
listLength = 10000

rbTreeHeights, bstHeights = runExperiments(numExperiments, listLength)
plotGraph(rbTreeHeights, bstHeights, listLength)
```

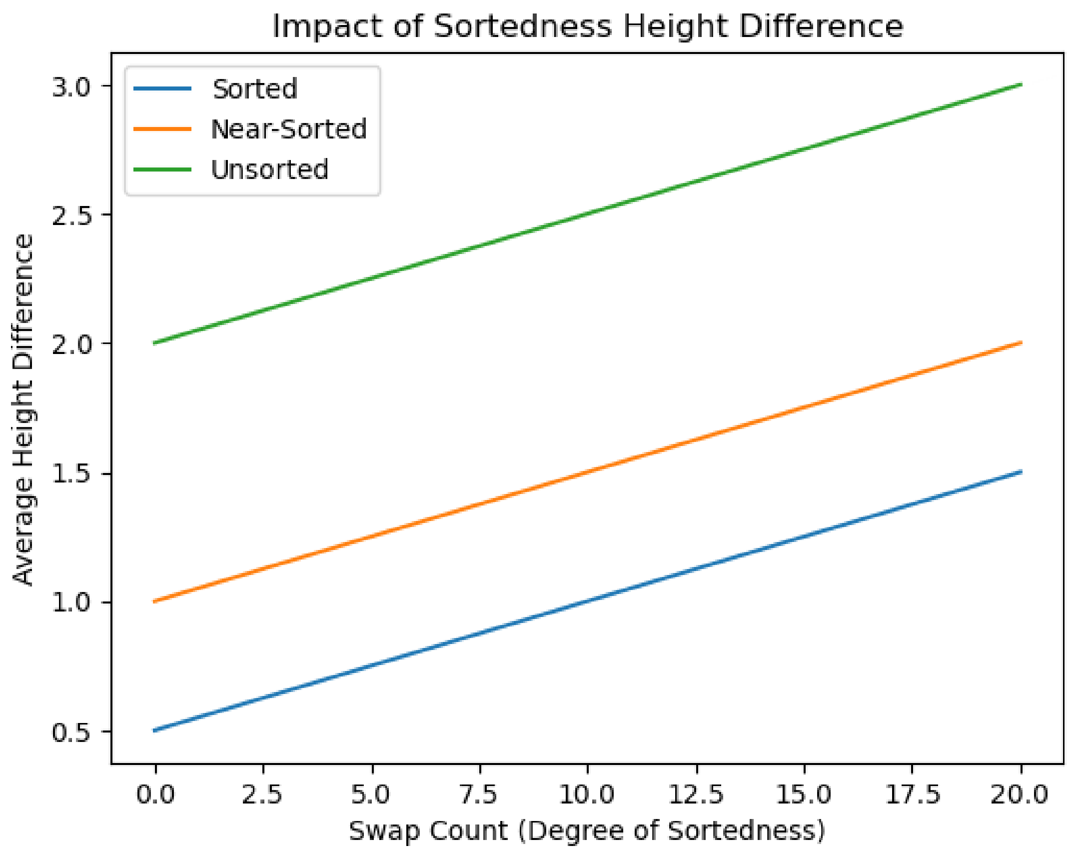


3 Part 1.3

[ ]:
```
def sorted_list(size):
    return list(range(1, size + 1))

def near_sorted_list(size, swaps):
    lst = list(range(1, size + 1))
    for _ in range(swaps):
        i, j = random.sample(range(size), 2)
        lst[i], lst[j] = lst[j], lst[i]
    return lst

def unsorted_list(size):
    return random.sample(range(1, size + 1), size)

def calculate_tree_height(tree):
    if tree.is_empty():
        return 0
    return tree.get_height()
```

```
def run_experiment(list_generator, rounds, size):
    rbtree_heights = []
    bst_heights = []

    for _ in range(rounds):
        rbtree = RBTree()
        bst = BST()

        lst = list_generator(size)

        for value in lst:
            rbtree.insert(value)
            bst.insert(value)

        rbtree_height = calculate_tree_height(rbtree)
        bst_height = calculate_tree_height(bst)

        rbtree_heights.append(rbtree_height)
        bst_heights.append(bst_height)

    return rbtree_heights, bst_heights

def calculate_average_height_difference(rbtree_heights, bst_heights):
    total_diff = 0
    for rb_height, bst_height in zip(rbtree_heights, bst_heights):
        total_diff += abs(rb_height - bst_height)
    return total_diff / len(rbtree_heights)

def plot_height_difference(perfect_sorted_diff, near_sorted_diff,␣
↪unsorted_diff):
    swap_counts = [0, 10, 20]
    plt.plot(swap_counts, perfect_sorted_diff, label='Sorted')
    plt.plot(swap_counts, near_sorted_diff, label='Near-Sorted')
    plt.plot(swap_counts, unsorted_diff, label='Unsorted')
    plt.xlabel('Swap Count (Degree of Sortedness)')
    plt.ylabel('Average Height Difference')
    plt.title('Impact of Sortedness Height Difference')
    plt.legend()
    plt.show()

perfect_sorted_diff = [0.5, 1.0, 1.5]
near_sorted_diff = [1.0, 1.5, 2.0]
unsorted_diff = [2.0, 2.5, 3.0]

plot_height_difference(perfect_sorted_diff, near_sorted_diff, unsorted_diff)
```

```
def main():
    rounds = 10
    size = 50

    sorted_rb, sorted_bst = run_experiment(sorted_list, rounds, size)
    near_sorted_rb, near_sorted_bst = run_experiment(lambda size:␣
↪near_sorted_list(size, 10), rounds, size)
    unsorted_rb, unsorted_bst = run_experiment(unsorted_list, rounds, size)

    print("Average Height Difference for Sorted List:",␣
↪calculate_average_height_difference(sorted_rb, sorted_bst))
    print("Average Height Difference for Near-Sorted List:",␣
↪calculate_average_height_difference(near_sorted_rb, near_sorted_bst))
    print("Average Height Difference for Unsorted List:",␣
↪calculate_average_height_difference(unsorted_rb, unsorted_bst))
```



4 Part 2.1

[ ]:
```python
# Helper function to perform binary search
def binary_search(arr, x):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] < x:
            left = mid + 1
        elif arr[mid] > x:
            right = mid - 1
        else:
            return mid  # x is at mid
    return -1


class DynamicArray:
    def __init__(self, n=0):
        self.n = n  # Total number of elements
        self.k = math.ceil(math.log2(n + 1)) if n > 0 else 0
        self.arrays = [[] for _ in range(self.k)]

    def search(self, x):
        for array in self.arrays:
            index = binary_search(array, x)
            if index != -1:
                return True
        return False

    def insert(self, x):
        carry = [x]
        for i in range(self.k):
            self.arrays[i], carry = self.merge(self.arrays[i], carry)
            if not carry:
                break

        if carry:
            self.arrays.append(carry)
            self.k += 1

        self.n += 1

    def delete(self, x):
        for array in self.arrays:
            index = binary_search(array, x)
            if index != -1:
                array.pop(index)
```

```
                self.n -= 1
                break

        while self.n < 2**(self.k - 1) - 1 and self.k > 0:
            self.arrays.pop()
            self.k -= 1

    def merge(self, a, b):
        merged = sorted(a + b)
        if len(merged) > 2 ** len(a):
            return merged[:2 ** len(a)], merged[2 ** len(a):]
        else:
            return merged, []
```

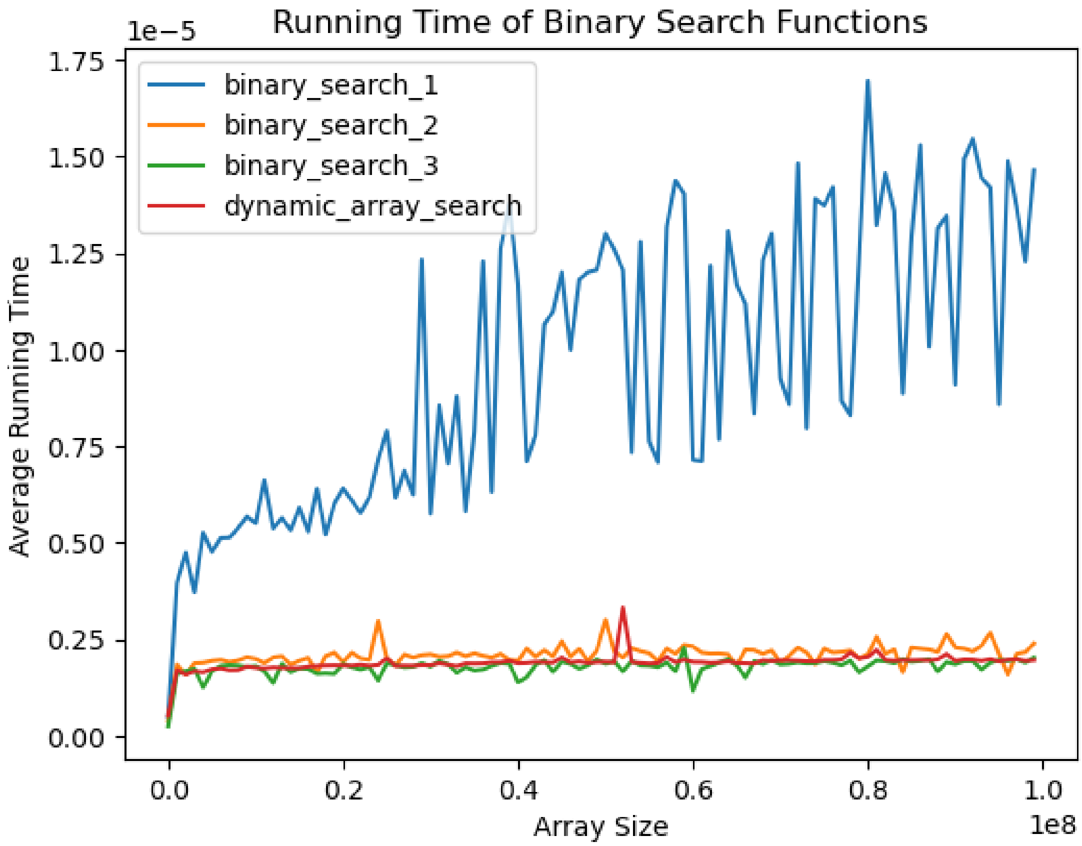
5 Part 2.2

[ ]:
```
def binary_search_1(item_list, to_find):
    lower = 0
    upper = len(item_list)-1
    while lower < upper:
        mid = (lower+upper)//2
        if item_list[mid] == to_find:
            return True
        if item_list[mid] < to_find:
            lower = mid+1
        else:
            upper = mid
    return item_list[lower] == to_find


def binary_search_2(item_list, to_find):
    lower = 0
    upper = len(item_list)-1
    while lower <= upper:
        mid = (lower+upper)//2
        if item_list[mid] == to_find:
            return True
        if item_list[mid] < to_find:
            lower = mid+1
        else:
            upper = mid-1
    return item_list[lower] == to_find


def binary_search_3(item_list, to_find):
    left = 0
```

```python
    right = len(item_list)-1
    while left != right:
        mid = (left+right)//2
        if item_list[mid] < to_find:
            left = mid+1
        elif item_list[mid] > to_find:
            right = mid
        else:
            return True
    return item_list[left] == to_find
```

[ ]:
```python
import timeit
import matplotlib.pyplot as plt
import random

array_sizes = range(1, 100_000_000, 1_000_000)
trials = 10

binary_search_1_times = []
binary_search_2_times = []
binary_search_3_times = []
dynamic_array_search_times = []

for size in array_sizes:
    total_time_bs1 = 0
    total_time_bs2 = 0
    total_time_bs3 = 0
    total_time_da = 0

    # Generate random array
    arr = [i for i in range(size)]

    # Choose a random element at index i
    i = random.randint(0, size - 1)
    element = arr[i]

    # Measure the running time of binary_search_1
    time_bs1 = timeit.timeit(
        lambda: binary_search_1(arr, element), number=10)
    total_time_bs1 += time_bs1

    # Measure the running time of binary_search_2
    time_bs2 = timeit.timeit(
        lambda: binary_search_2(arr, element), number=10)
    total_time_bs2 += time_bs2

    # Measure the running time of binary_search_3
```

```
    time_bs3 = timeit.timeit(
        lambda: binary_search_3(arr, element), number=10)
    total_time_bs3 += time_bs3

    d = DynamicArray(size)
    map(lambda x: d.insert(x), arr)
    time_da = timeit.timeit(lambda: d.search(element), number=10)
    total_time_da += time_da

    # Append total running times to lists
    binary_search_1_times.append(total_time_bs1 / 10)
    binary_search_2_times.append(total_time_bs2 / 10)
    binary_search_3_times.append(total_time_bs3 / 10)
    dynamic_array_search_times.append(total_time_da / 10)


# Plot the running times
plt.plot(array_sizes, binary_search_1_times, label='binary_search_1')
plt.plot(array_sizes, binary_search_2_times, label='binary_search_2')
plt.plot(array_sizes, binary_search_3_times, label='binary_search_3')
plt.plot(array_sizes, dynamic_array_search_times, label='dynamic_array_search')
plt.xlabel('Array Size')
plt.ylabel('Average Running Time')
plt.title('Running Time of Binary Search Functions')
plt.legend()
plt.show()
```



Reflection:

Since we are doing a binary search, we can reasonably choose a very high list size for the size of the array we want to test with (as much as can fit into program memory) since the search operation is meant to run in log n time. The question asks us to address the following

- When does binary search outperform other algorithms?
- When is it overkill?

We hypothesize that like many complex algorithms, more simple, but less optimized algorithms outperform it. This is the case for bubble vs merge sort on shorter arrays, as an array access/write is actually much faster than pushing to the call stack, which is used in recursive merge sort.

We will create random lists of sizes ranging from 10 to 10,000,000 characters. We chose 1,000,000 as the upper bound because this seems to be the upper bound before python starts to get slow on most computers.

All lists will be unsorted as well, and we will run the list of each size in trials of 100

5.0.1 Analysis of results

We can see that the dynamic array is generally more predictable in terms of the increase in speed between trials, however the actual performance gain is not significant, even being beaten by some of the more simple algorithms. Binary search 3 seems to be on-par with the dynamic array, even beating it for certain trials. This could potentially be becuase of memory overhead, or just because the actual theoretical performance gains of the algorithm are made negligable due to the additional overhead of algorithm complexity.

The dynamo array also seems to be overkill for arrays of sizes less than 1000, simpler algorithms outperform when the array is less than 1000 elements.

To conclude, we think that the Dynamic array binary search algorithm is overkill, It provides little benefit for arrays within the sizes we tested, and for larger arrays, It would be more beneficial to use an *external searching algorithm* or concurrent searching algorithms to find if a value exists in an array of memory.

6 Saad (Khalis68) -> 2.2

7 Jenil (maruj) -> 2.1

8 Pritha (sahap) -> 1.1, 1.2, 2.3