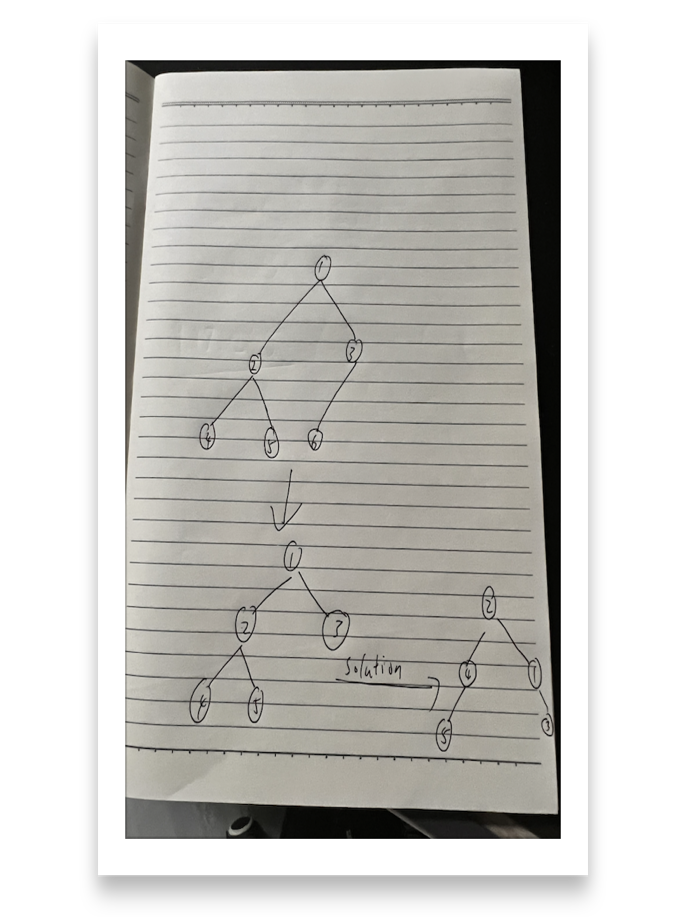
Problem 1



Solution for unbalanced problem:

Create a new node whose value is equal to the value of root. Insert it on the right-sub tree. Delete the root node. Create a new node(new\_root) whose value is equal to the left-most node of left-sub tree(name it as lm), and delete lm. And make new\_root the root node. In this situation the right-subtree increases its height by one.

Problem 2

See .java file

Problem 3

The advantage of taking p as prime in some cases is capable of avoiding collision problem.

For example Consider the values to be hashed K = {1,2,...,100}, and take p = 12 which is not a prime, since it is divisible by 3. Values 12k will be hashed to 0. Values 12k + 3 will go to 3. Values 12k + 6 will go to 6, and so on. If K is normally distributed then the choice of p is not critical. However, what if K is not? For example, says the numbers divisible by 3 have higher chance to occur, then buckets that are multiples of 3 have higher chance to have collision to happen. Therefore, in order to avoid this case of collision. One may let p be a prime.

Problem 4

(See hash.py for detail of algorithm)

Linear\_probing = [34, 0, 45, None, None, 10001, 6, 23, 7, None, None, 28, 12, 29, 11, 30, 33]

Double\_hashing = [34, None, 45, None, None, 10001, 6, 7, None, None, None, 28, 12, 30, None, None, 33]

I like double hashing better since with properly chosen function h’, it can be more efficient than linear probe to insert element.

Problem 5

For linear probing, first off performing linear probing search. Once we find the cell, mark it as deleted and then search forward until we hit another empty cell. This method of deletion can prevent us from incorrectly report that the key is not present. For double hashing, it is also suggested to mark the cell to be deleted deleted, and then we perform searching until we hit another empty cell.