MONASH University

FACULTY OF INFORMATION TECHNOLOGY

FIT1008/FIT2085 - Assignment 3

A3 - The Potion Seller

Purpose	*To further facilitate your skills of using and manipulating readily available implementations of Abstract Data Types, implementing your own and applying them in practical problem solving. To solidify your knowledge of best- and worst-case complexity analysis of algorithms and apply it in the context of recursive sorting algorithms.*
Your task	*In this assessment, you will be given several tasks. In each of them, based on the problem description provided, you will need to implement a Python program that solves the problem. You will also need to test and document your code and analyse its complexity. Dealing with the tasks will involve your practical interaction with Stack, List, Queue, and Hash ADTs covered in the lessons, (to be) implemented in Python using object-oriented programming. Additionally, some of the tasks will require you to implement and use recursive sorting algorithms such QuickSort and MergeSort. Finally, you will practice manipulating Binary Search Trees (BSTs) and see how self-balancing can be achieved and used in practice. Note that some parts of the code functionality may be provided to you, in which case your task will be to complete it.*
Value	**40**% of your total marks for the unit
Word Limit	N/A
Due Date	**11:55 pm AEST, 29 May 2022 (Week 12)**
Submission	• Via Moodle Assignment Submission. • Turnitin will be used for similarity checking of all submissions.

Assessment Criteria	*Your solution will be assessed using the following criteria:*
	1. *Correctness of your solution:*
	  a. *The code does what it is supposed to*
	  b. *The code follows good design practices*
	  c. *The fraction of our tests passed by your code*
	2. *How well is your code documented?*
	  a. *The fraction of methods accompanied by a detailed doc-string*
	  b. *Quality of documentation (preconditions, postconditions, invariants indicated?)*
	3. *Complexity analysis provided for student's solution*
	  a. *The fraction of your methods accompanied by best-case complexity correctly identified*
	  b. *The fraction of your methods accompanied by worst-case complexity correctly identified*

	4. *Group work:* a. *Division of work overall (based on the description you provided as a group)* b. *Your personal contributions made (based on your peers' opinions)* c. *Ideally, we want each person in a group contribute equally* 5. *Interview:* a. *Your ability to explain design choices made, during the interview* b. *Your ability to discuss issues identified by your TA in your solution (if any)* *c. We want to see if you understand the issues and can suggest possible ways to resolve them in real time*
Late Penalties	● 10% deduction per calendar day or part thereof for up to one week ● Submissions more than 7 calendar days after the due date will receive a mark of zero (0) and no assessment feedback will be provided.
Support Resources	See Moodle Assessment page
Feedback	Feedback will be provided on student work via: general cohort performance specific student feedback ten working days post submission

INSTRUCTIONS

Advice for Groups

This assignment does not have tasks that are separated or ordered in terms of difficulty, or in terms of workload. While at the end of the day only one person can code a particular function, it is in your team's best interest that before writing any code your entire team goes through the tasks and discusses the best possible solutions, and splits up the work based on this assessment of difficulty/workload, not “Person 1 does Task 1, Person 2 does …”.

Throughout this assessment, you’ll need to decide whether to use certain algorithms / data structures. If you are feeling lost, try:

1. To let the complexity requirements of the question guide your judgement, and
2. Failing that, just list off techniques covered in the second half of the unit and see if one of them might be appropriate.

While there are some requirements given for each task (such as *add these attributes*, *implement this method*, etc), it may be in your best interest to add additional attributes / methods to previous parts of your code, to solve certain problems *more efficiently*.

While there are no requirements for testing for this Assignment, in contrast to Assignment 2, you are playing a very dangerous game if you do not add any tests. So we encourage you to implement and use them in order to make your final solution bulletproof.

The final task might require you to implement an ADT/Algorithm not present in the scaffold given. If you are still feeling stuck, try playing the game with a few examples! You might make some discoveries on the best solution, and failing that, attend consultations / optional applied sessions.

Documentation Requirements

Python documentation akin to that available in the example_documentation.py file on Moodle is expected of all files submitted for the assignment. All methods mentioned in the tasks below require docstrings which (1) go over the approach taken and (2) the worst-case time complexity. There are also particular documentation requirements for the final few tasks, detailed below where appropriate.

Problem Background - Potion Seller

You are constructing a new bartering game “Potion Seller”.
The game is centred around the titular Potions, and contains 3 groups of people:

- Vendors working for the company PotionCorp
- Adventurers, looking to buy Potions
- You, the player, attempting to be the man-in-the-middle of this transaction.

The aim of this game is to buy potions from the vendors, and sell these to the adventurers for big profits.

On any given day, your aim is to find discrepancies between the adventurers buying price, and the vendors selling price. For example, you might buy 5 Litres of a Potion of Instant Health from the vendors for $25, and sell this to the adventurers for $125, for a profit of $100.

A few things are assumed about these groups:

- The adventurers have access to infinite amounts of money - They will always buy a potion if you offer it to them, at the rate they specify.
- The vendors have a shared inventory of Potions, supplied to them by PotionCorp. A vendor can only sell a single potion at a time, but may choose any potion from the inventory that no other vendor is selling.

Each potion has the following attributes:

- A category - The type of potion sold (Healing, Damage, Buff, etc)
- A name - Specifying the exact effects (“Potion of Minor Regeneration”, “Deadly Poison”, “Potion of Undying Strength”, etc)
- A price per litre - The amount paid per litre, to purchase this potion from a vendor ($10/L, $23.5/L, etc). Note that one can buy fractional quantities of each potion (I can buy 250ml of the first potion for $2.5)

The selling price to adventurers changes day by day, and so will be covered later on. A potion is uniquely identified by its name. All buy prices of potions are unique.

The vendors, however, do not offer all items at all times. Each day, each vendor will only offer 1 item from this inventory.

The game is played over multiple days, and each day progresses as follows:

1. Certain potions/quantities are added to the inventory of the vendors by PotionCorp
2. Each vendor picks a single potion from the inventory to offer (Which is different from that of every other vendor)
3. The player can purchase potions from the vendors
4. The player can sell potions to the adventurers for profits

Task Descriptions

Problem Setup

Before simulating this bartering game, we need to do ***a bit :-)*** of setup, so that we can make our code as efficient as possible. In particular, we need to work on implementations of:

- Hash Tables
- Binary Search Trees
- AVL Trees

While implementing all the bits of preliminary implementation, you will also deal with efficient sorting algorithms, recursion, and generators. Once we have all the aforementioned implementations, and some special methods, on hand, playing the game should be a lot easier.

Prime Number Generation

To assist with later tasks, you should first implement the function `largest_prime(k: int) -> int` (located in primes.py), which returns the largest prime number strictly less than `k`. You may want to take inspiration from the Sieve of Eratosthenes.

You can guarantee that k will always be larger than 2 (1 is not a prime number!), and at most 100,000.

Hashing Potions

In order to keep track of Potions in the game, we want to create an object for them. In addition to this, since these potions will eventually be stored in a hash table, we also need a way to hash them. To analyse the efficacy of your hash function, you will implement two hash functions, one which you think is good, and another which you think will cause issues for your hash table.

Your task (in potion.py) is to create a Potion class, with the following attributes:

- `potion_type: str`
- `name: str`
- `buy_price: float`
- `quantity: float`

And must implement the following methods:

- Instance Method `__init__(self, potion_type: str, name: str, buy_price: float, quantity: float) -> None`
- Class Method `create_empty(cls, potion_type: str, name: str, buy_price: float) -> Potion`, which creates a potion with 0 quantity.
- Class Method `good_hash(cls, name: str, tablesize: int) -> int`, which hashes a potion name, given a fixed table size.
- Class Method `bad_hash(cls, name: str, tablesize: int) -> int`, which hashes a potion name, given a fixed table size.

Hash Table Analysis

Now that we have potion objects and hashing functions, change the file `hash_table.py` and add the following functionality:

- Instance Method `__init__(self, max_potions: int, good_hash: bool=True, tablesize_override: int=-1) -> None`, which initialises the hash table. `max_potions` is the maximum number of elements that will be added to the hash table. If `good_hash` is true, use `Potion.good_hash` to hash keys, otherwise use `Potion.bad_hash`. If `tablesize_override` is -1, then your class should select a reasonable choice for tablesize, given the value of `max_potions`. Otherwise, your class should set the tablesize to the exact value of `tablesize_override`.
- Instance Method `hash(self, potion_name: str) -> int`, Which hashes a potion name.
- Instance Method `statistics(self) -> tuple`, which returns a tuple of 3 values:
    - the total number of conflicts (conflict_count)
    - the total distance probed throughout the execution of the code (probe_total)
    - the length of the longest probe chain throughout the execution of the code (probe_max)

Additionally, you need to provide a short paragraph on what you expect the output of the statistics method to be for your good hash function, and bad hash function, and why this is. (Aroundabout 50-200 words). You should produce sufficient fake data to validate or invalidate your prediction and present these results. **All analysis should be included in a pdf file named analysis.pdf**, and you should make it apparent where fake data has been generated and stored in analysis.pdf.

Some instance variables have already been initialised in the __init__ method to give you a headstart for `statistics`. To get the statistics function working, you will need to modify other methods of the Hash Table.

Information on what Conflicts, Collisions, and Probes are, and how they differ, is provided below:

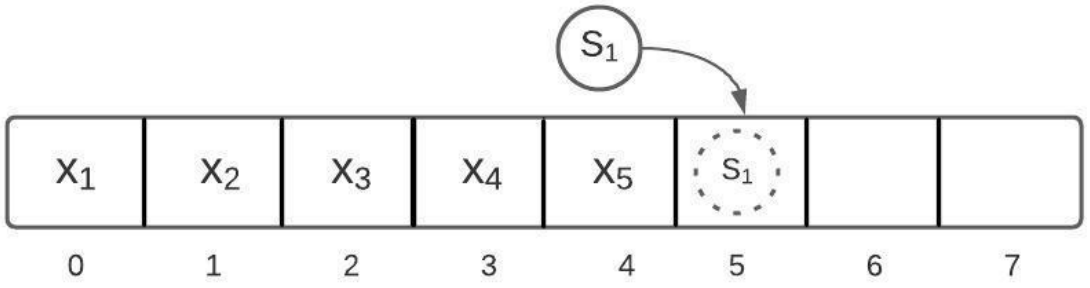
Conflict Handling

Recall that in practice hash functions may map two *distinct keys* s1 and s2 into the same hash value, i.e. h(s1)=h(s2) s.t. s1≠s2. Such situations are referred to as *collisions*.

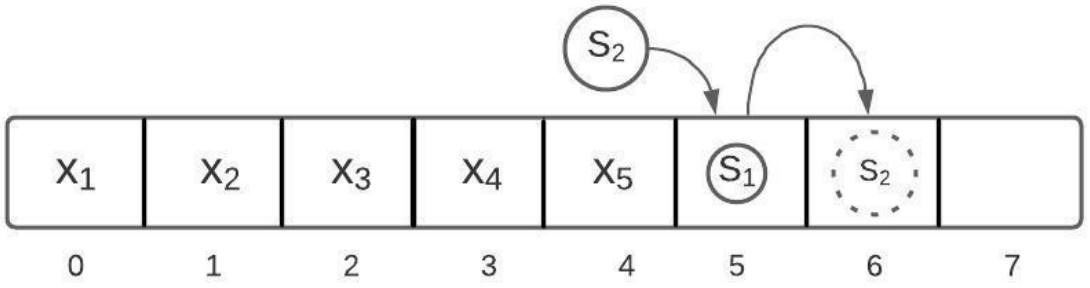
Also, when discussing open addressing, we introduced two more concepts: that of a *conflict*, which occurs when we attempt to insert a key into our hash table, but the location for that key is already occupied by a *different* key); and that of *probe chain*, which is the sequence of positions we inspect before finding an empty space suitable for our key.

Note that a conflict may be caused either by (a) a collision due to the bad choice of our hash function, or (b) some of the previous runs of linear probing. Also note that when adding a new key to the hash table, *at most one* conflict may happen due to the reasons above.

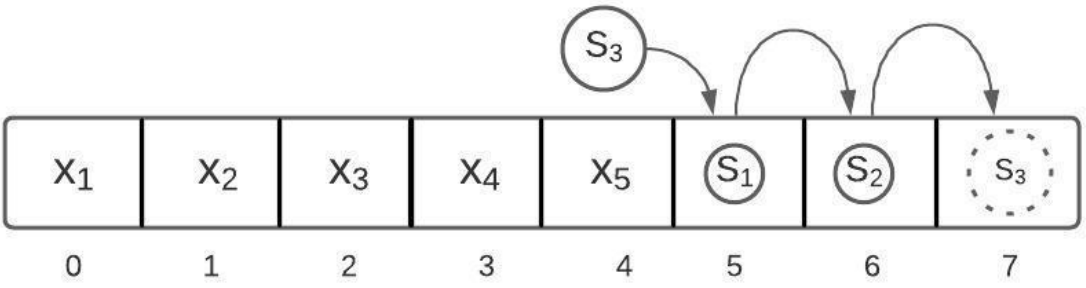
For example, consider inserting keys s1, s2 and s3 into an empty table with linear probing, assuming all three hash to location 5, i.e. all the three *collide* since h(s1)=h(s2)=h(s3)=5h(s_1)=h(s_2)=h(s_3)=5h(s1)=h(s2)=h(s3)=5. If we first insert s1, location 5 is empty and, thus, there is no conflict. If we then insert s2, location 5 is full but 6 is empty. This results in one conflict, with probe chain length 1. If we then insert s3, 5 and 6 are full, but 7 is free. This again results in one conflict, with probe chain length 2. (Both conflicts are caused by the aforementioned collisions h(s1)=h(s2)=h(s3)=5.) Please refer to the diagram below for reference:



Conflicts = 0
Probe Chain = [ ]
Probe length = 0



Conflicts = 1
Probe Chain = [5]
Probe length = 1



Conflicts = 1
Probe Chain = [5, 6]
Probe length = 2

Situation when there is no Collision but there is a Conflict

You might be wondering what the difference between a collision and conflict really is and you're not alone. To sum up from what we learned above:

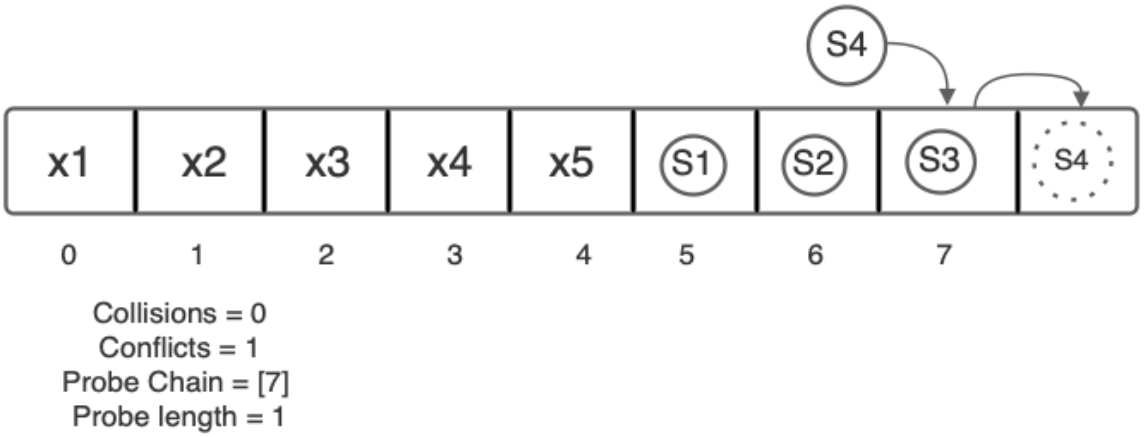
1. A collision happens when two values are hashed into the same location

2. A conflict happens when linear probing is required to find a position for the value being inserted.

So, there can be a situation where a key is present at a location that occurred due to a prior linear probing but wasn't originally hashed in that space. In this case, when we try to insert a new key to this position, there is no collision but there is a conflict. The reason for no collision is because that space would have been empty if the previous element was inserted after the current one.

For example -

Let's expand on the example that we used before and let's expand the hash table to have one extra spot. Now, let's consider an element s4 such that h(s4)=7. Now we know that s3 already exists in index 7, but that happened due to a linear probe as h(s3)=5. This means that we have a conflict, but no collision. The probe chain is still [7] and the probe length is 1.



BST Operations

As you already know, binary trees are *ubiquitous* in computer science and in programming. Same holds for binary search trees (BSTs), which enable us to apply them in a number of important practical settings.

Although being arguably one of the simplest and yet powerful data structures, binary search trees are susceptible to becoming ***unbalanced***, after a number of insertion and deletion operations. To overcome this issue, computer scientists proposed various modifications and extensions to BSTs that made them self-balancing. The basic idea is that whenever an insertion or deletion operation is performed, the balance of the tree is checked and rebalancing is done if needed. Examples of self-balancing BSTs include

- red-black trees
- B-trees and B+-trees
- **AVL trees**

Clearly, the advantage of self-balancing trees is that they guarantee that *insertion*, *deletion*, *lookup* operations can be performed in ***logarithmic time*** (on the number of nodes in the tree), which is in clear contrast with the simple binary search trees. Moreover, since they are valid BSTs, they provide us with a simple and efficient way to traverse our data *in order*.

In this set of tasks, we are going to implement *self-balancing AVL trees* by **extending the functionality** of class BinarySearchTree of file bst.py partially provided to you in the scaffold. Although the implementation of self-balancing is to be done in the next section of the assignment, here you need to prepare the foundation for that.

In the following tasks, you need to update the implementation of BSTs provided to you by adding the following missing functionality. Your concrete task is to update the class `BinarySearchTree` in file `bst.py` by adding implementations for the following methods:

- `get_successor(self, current: TreeNode) -> TreeNode`, which returns the successor of `current` (the smallest node *greater* than current) in the sub-tree. It should return `None` if there is no element greater in the subtree.
- `get_minimal(self, current: TreeNode) -> TreeNode`, which returns the smallest node in the current sub-tree.

Self-balancing AVL Trees

Invented in 1962 by Georgy Adelson-Velsky and Evgenii Landis, AVL trees serve as a simple extension of the standard binary search trees and represent one of the first examples of self-balancing BSTs.

AVL trees build on the concept of ***balance factor*** of a *subtree* defined by the `root` of the subtree:

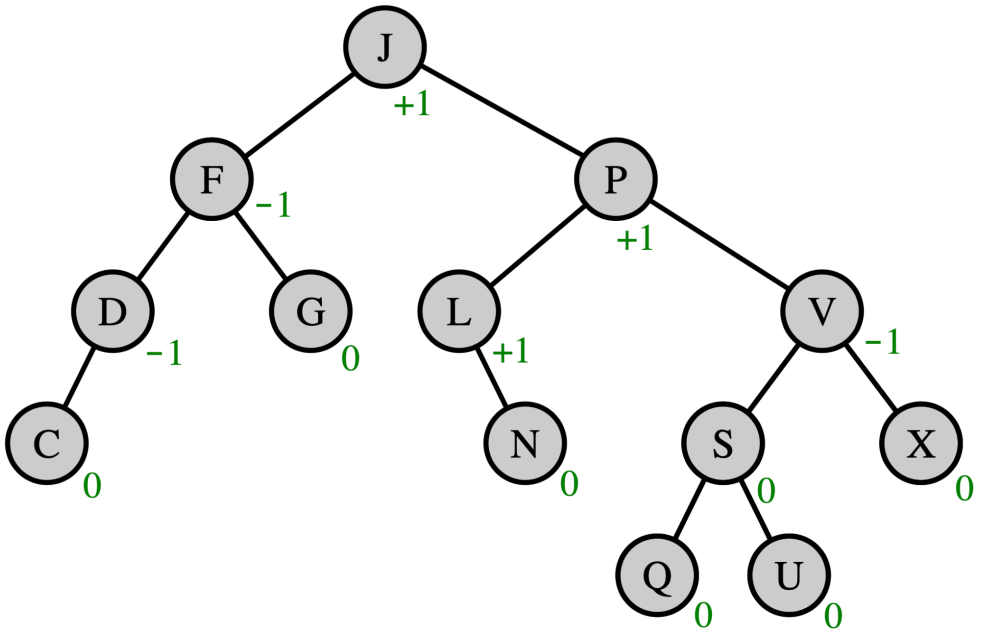
$$BalanceFactor(root) = Height(root.right) - Height(root.left)$$

A tree T is called ***balanced***, if the balance factor of every node in the tree T is in {-1, 0, 1}.

Hence, if for some of the nodes of T the balance factor is above these limits, the height of the corresponding left and right branches of the node-rooted subtree differ a lot thus breaking the worst-case logarithmic guarantees on the complexity of the main tree operations.

AVL trees only add a way to rebalance the tree if it is necessary. Besides that, the trees perform *exactly the same way* as the standard binary search trees.

Example of a balanced AVL tree with balance factors shown green:



How It Works – Basic Idea

Assume that after an insertion or deletion operation we end up having an unbalanced subtree with the root node having the balance factor from {-2, 2}. In this case, we can perform a ***rotation*** operation to return the balance factor of the subtree back to normal.

Note that the balance factor cannot get larger than 2 or less than −2 if each insertion and deletion operation is followed by rebalancing.

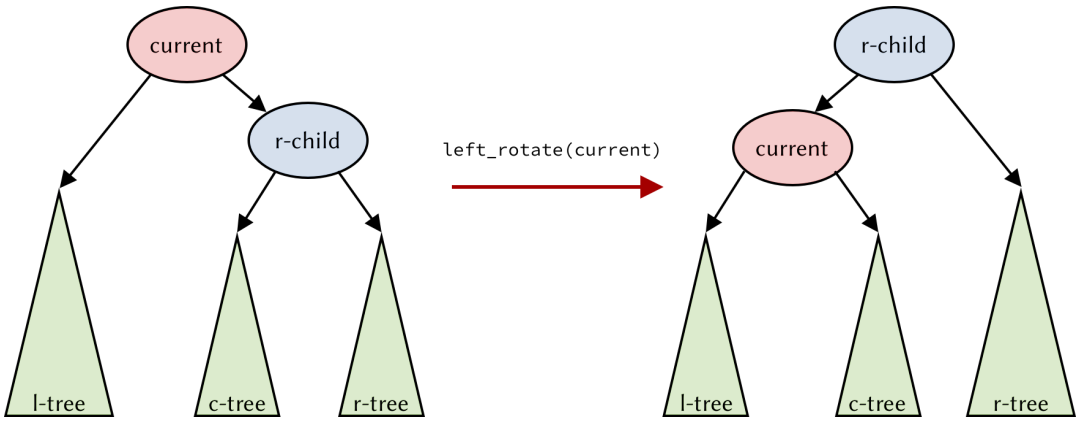
Below is an animated illustration of how the process works in practice (original GIF animation is taken from Wikipedia). Here the nodes in the tree are additionally labeled by the values of their ***balance factors***.

There are **two basic types** of rotation:

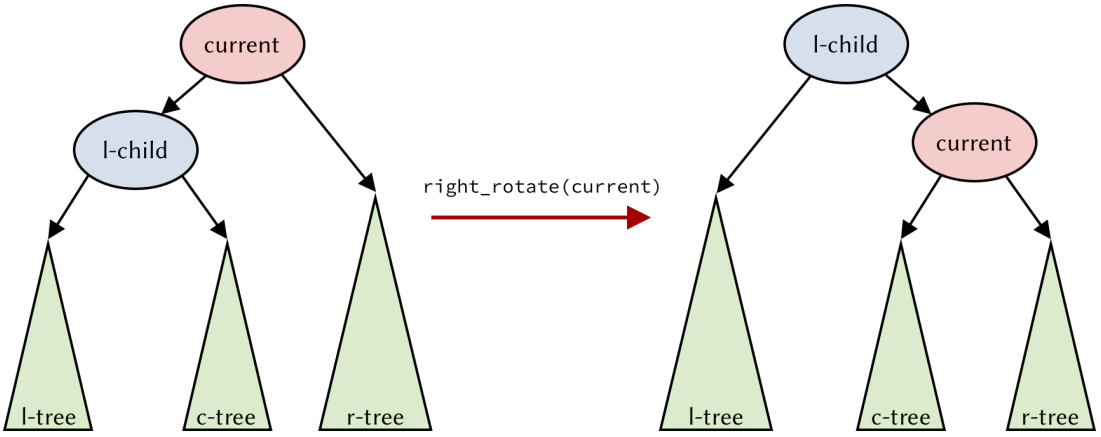
- *left* rotation
- *right* rotation

Your task is to update `avl.py` to add implementations for the following methods:

- `left_rotate(self, current: AVLTreeNode) -> AVLTreeNode`, which perform the left rotation of the current subtree, as shown below, and then updates the heights of *all the modified nodes*. Note that it **returns the new root (r-child in this case)**.



- `right_rotate(self, current: AVLTreeNode) -> AVLTreeNode`, which perform the right rotation of the current subtree, as shown below, and then updates the heights of *all the modified nodes*. Note that it **returns the new root (l-child in this case)**.



Putting it all together

Once the above functionality of AVL trees is prepared, you can start making them self-balancing. The code for rebalancing is given, and should complete the rotations correctly.

Now that you're done with subtree rebalancing, the final bit left is to apply it whenever the tree becomes unbalanced. When does this happen? Well, after you add or delete nodes. As a result, in class `AVLTree` redefine the following methods provided in the base class `BinarySearchTree` such that they apply subtree rebalancing after performing the corresponding operations:

- `insert_aux(self, current: AVLTreeNode, key: K, item: I) -> AVLTreeNode`
- `delete_aux(self, current: AVLTreeNode, key: K) -> AVLTreeNode`

Note that both methods should

- update the height of the current node,
- and rebalance the subtree rooted by the current node.

Kth Largest

Your task is to update avl.py, to add implementations for the following methods:

`kth_largest(self, k: int) -> AVLTreeNode`

Returns the kth largest node in the tree, using *recursion*. Here k=1 should give the largest node in the subtree. (Hint: You may need to add some extra code to other AVL operations, and create other attributes in order to achieve the intended complexity).

Complexity Requirements: $\mathbf{O}(\log(N))$

Where N is the total number of nodes in the tree.

A solution which does not achieve this complexity bound, but does achieve $\mathbf{O}(k)$, will achieve at most 75% of the marks for this task.

Playing the Game

With all of that out of the way, let's start working on the game!

Random Number Generation

For our game, we also need Random Number Generation. An easy way to implement this is with a Linear Congruential Generator. Some python code (copied from Wikipedia) is given below:

```
from typing import Generator

def lcg(modulus: int, a: int, c: int, seed: int) -> Generator[int,
None, None]:
    """Linear congruential generator."""
    while True:
        seed = (a * seed + c) % modulus
        yield seed
Random_gen = lcg(pow(2,32), 134775813, 1, 0)
```

But for our game, we need to use the Random generator in a particular way.

Your task is to implement a RandomGen class which implements two methods: `__init__(self, seed: int=0) -> None`, and `randint(self, k: int) -> int`, which generates a random number from 1 to k inclusive with the following approach:

- First, generate 5 random numbers using the lcg method above, dropping the 16 least significant bits of each number.
- Generate a new number, which is 16 bits long and has a 1 in each bit if at least 3 of the 5 generated numbers have a 1 in this bit.
- Return this new number, modulo k, plus 1.

This process is illustrated below, calling `randint(74)`:

1: Five Random Numbers

1341408904		01001111111101000100011010001000
3916732889		11101001011101001001100111011001
4161854668	Binary ⇨	11111000000100001101110011001100
11272702		00000000101011000000000111111110
483725054		00011100110101010000111011111110

2: Remove 16 Least Significant Bits

0100111111110100
1110100101110100
1111100000010000
0000000010101100
0001110011010101

2322421233341401 <- # 1s in each column

3: Generate new number
from # of 1s in each column

0100100011110100 = 18676

4: Modulo k

18676 % 74 = 28+1=29

Game Documentation

For all below tasks, the following documentation is required: ***THIS IS WORTH 5/50 MARKS OF YOUR GRADE***

- For the Game class, you must supply a docstring describing the ADTs used for your approach and *why* they were used (In the order for 30-50 words)
- For each of the functions with complexity requirements, you must also provide in the method docstring an *explanation* of why your code is the complexity it is. You will get marks here if you are honest about your algorithm's complexity, even if it is not within the requirements of the question
- For solve_game, you must give a text explanation of how your solution works, and provide adequate inline comments to aid this explanation. (In the order of 50 words)

Vendor Item Selection

WARNING: For the complexity requirements of this question, you may assume that all hash table operations from your previous tasks take constant time.

As explained in the problem background, each day vendors select an item from the current inventory.

Vendor potion selection should be implemented in the following manner:

- One at a time, each vendor (order doesn't matter) selects a random number from 1 to C (inclusive), where C is the total number of potions which are currently in stock (have a positive amount of litres in inventory). This random number generation should use your RandomGen class from earlier. Call this number p.
- The vendor then selects the p^{th} most expensive (highest price per litre) potion from the inventory, excluding all potions chosen by previous vendors.
- The next vendor continues this process (Now selecting from 1 to C-1) until all vendors have a potion to sell. It will be guaranteed that there will always be enough potions in stock for every vendor to sell a distinct potion.
- While the inventory doesn't actually change in this process, you can implement it easily as so: Every time a vendor selects a potion, that vendor removes the potion from the inventory (ensuring no other vendor can also sell this potion), and at the end of the process, all vendors add the potions back into inventory.

Your task is to implement three methods on the `Game` class to facilitate management of the potion inventory:

```
set_total_potion_data(self, potion_data: list) ->
None
```

`potion_data` is a list containing all potions that could possibly enter the vendor inventory over the course of the game. Each element of `potion_data` is a list containing 3 values, which can be passed to `Potion.create_empty` which you implemented earlier.
This method should also set the inventory of the vendors to contain 0 Litres of every potion listed.

```
add_potions_to_inventory(self, potion_name_amount_pairs: list[tuple[str, float]]) -> None:
```

Adds litres of particular potions into the current inventory of the vendor company. Each element of `potion_name_amount_pairs` is a tuple containing two elements: The name of the potion, and amount in litres to add to the inventory.

Complexity Requirements: $\mathbf{O}(C \times \log(N))$
Where C is the length of `potion_name_amount_pairs`, and N is the number of potions provided in `set_total_potion_data`.

```
choose_potions_for_vendors(self, num_vendors: int) -> list
```

Completes the vendor potion selection process as described above. `num_vendors` is a single integer > 0, specifying how many vendors will sell potions.
The function should return a list, specifying what the vendors will sell. Each element in the list should be a tuple containing a string and a float - the x^{th} index contains the name of the potion vendor x will sell, as well as the quantity of that potion that was in inventory (and has since been taken out).

Complexity Requirements: $\mathbf{O}(C \times \log(N))$
Where C is equal to num_vendors, and N is the number of potions provided in `set_total_potion_data`.

Example

Suppose we have 6 Potions:

Potion of Health Regeneration
- Category: Healing
- Price: $20/L

Potion of Extreme Speed
- Category: Buff
- Price: $10/L

Potion of Deadly Poison
- Category: Damage
- Price: $45/L

Potion of Instant Health
- Category: Healing
- Price: $5/L

Potion of Increased Stamina
- Category: Buff

- Price: $25/L

Potion of Untenable Odour

- Category: Damage
- Price: $1/L

and that we have 4 vendors, and at day 1, PotionCorp has the following inventory:

- Potion of Health Regeneration: 4 Litres
- Potion of Extreme Speed: 5 Litres
- Potion of Deadly Poison: 0 Litres
- Potion of Instant Health: 3 Litres
- Potion of Increased Stamina: 10 Litres
- Potion of Untenable Odour: 5 Litres

This information might be given to the Game class in the following fashion:

```
G = Game()
# There are these potions, with these stats, available over the course of the game.
G.set_total_potion_data([
       # Name, Category, Buying price from vendors.
       ["Potion of Health Regeneration", "Health", 20],
       ["Potion of Extreme Speed", "Buff", 10],
       ["Potion of Deadly Poison", "Damage", 45],
       ["Potion of Instant Health", "Health", 5],
       ["Potion of Increased Stamina", "Buff", 25],
       ["Potion of Untenable Odour", "Damage", 1],
])

# Start of Day 1
# Let's begin by adding to the inventory of PotionCorp:
G.add_potions_to_inventory([
       ("Potion of Health Regeneration", 4),
       ("Potion of Extreme Speed", 5),
       ("Potion of Instant Health", 3),
       ("Potion of Increased Stamina", 10),
       ("Potion of Untenable Odour", 5),
])

# Now for choosing vendors!
selling = G.choose_potions_for_vendors(4)
print(selling)
>>> [
       ("Potion of Extreme Speed", 5),
       ("Potion of Increased Stamina", 10),
       ("Potion of Health Regeneration", 4),
       ("Potion of Instant Health", 3),
        ]
```

- In this example, vendor 1 picked a random number from 1 to 5 and got 3, since Extreme Speed is the 3rd most expensive in inventory.
- vendor 2 picked a random number from 1 to 4 and got 1, since Increased Stamina is the 1st most expensive in the remaining inventory.

- vendor 3 picked a random number from 1 to 3 and got 1, since Health Regeneration is the 1st most expensive in the remaining inventory.
- vendor 4 picked a random number from 1 to 2 and got 1, since Instant Health is the 1st most expensive in the remaining inventory.

Game Mode

WARNING: For the complexity requirements of this question, you may assume that all hash table operations from your previous tasks take constant time.

Now that we can simulate the vendor process, we want to play the game optimally!
In this first game mode, you are given multiple attempts to buy and sell potions, the only difference being that you start with a different amount of funds.
In each attempt, you can buy as many potions from the vendors as their inventory allows, and at the end of the day, sell this back to the adventurers for profit. After each attempt, the day (and inventory of the vendors) resets and you can try with a new starting balance.
The player can purchase any potion in any floating point quantity, provided that this does not exceed the current inventory level of that potion.

There are some optimal ways to purchase potions from the vendors to maximise profits after reselling to the adventurers. We need your help to figure out the best way to purchase on this day, for each attempt!

For this task, you need to implement one method of the Game Class:

```
solve_game(self, potion_valuations: list[tuple[str, float]], starting_money: list[int]) -> list[float]
```

`potion_valuations` is a list of potions that each vendor is selling, paired with its valuation by the adventurers (How much the adventurers will pay per litre for the potion). How much of that potion is available from the vendors should be known in the Game class, even if that potion has been "removed" from inventory. You might need to store some additional information.

starting_money is a list containing, for each attempt, the starting allowance the player has.

The function should return a single floating point number for each element of `starting_money`; the maximum money that the player can have at the end of the day, given they start the day with the corresponding entry in `starting_money`. In particular, this function does not need to specify the potions used to achieve this result.
Any response within 10^{-6} absolute distance of the answer will be considered correct.

Complexity Requirements: $\mathbf{O}(N \times \log(N) + M \times N)$
Where N is the length of `potion_valuations`, and M is the length of `starting_money`.
For brownie points and nothing else, feel free to try solving the problem in $\mathbf{O}(N \times \log(N) + M \times \log(M))$.

Example

Continuing from the example given for the Vendor Item Selection task, suppose we play Game Mode with those vendor selections:

```
# Let’s suppose that the adventurers will pay 30, 15, 15, and 20 dollars per litre
for each of the potions listed below.
full_vendor_info = [
       (“Potion of Health Regeneration”, 30),
       (“Potion of Extreme Speed”, 15),
       (“Potion of Instant Health”, 15),
       (“Potion of Increased Stamina”, 20),
]

# Play the game with 3 attempts, at different starting money.
results = G.solve_game_1(full_vendor_info, [12.5, 45, 80])
print(results)
>>> [37.5, 90, 142.5]
```

This result is achieved for the following reasons:
Attempt 1 can make $37.5 by buying 2.5 litres of Instant Health for $12.5.
Attempt 2 can make $90 by buying 3 litres of Instant Health for $15, and 1 litre of Health Regeneration for $20, and 1 litre of Extreme Speed for $10.
Attempt 3 can make $142.5 by buying 3 litres of Instant Health for $15, 3 litres of Health Regeneration for $60, and 0.5 litres of Extreme Speed for $5.