Welcome to A2 - Mountain Climber!

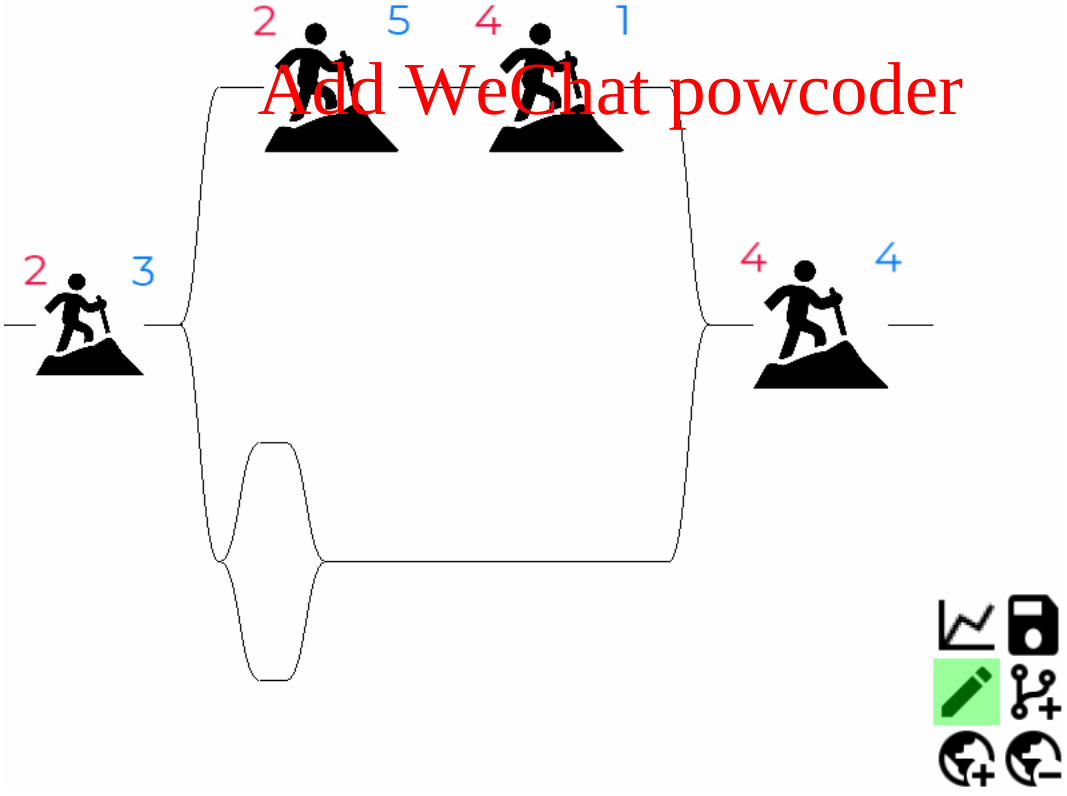
Now that you've acquainted yourself with Stacks, Queues, Lists and the like, it's time to explore more complex data structures, and use them to solve more complicated tasks!

In this assignment, you'll work on the following features:

- A Mountain Trail representation
- A plot view that shows the longest trails suitable for a given hiker
- A trip organiser that keeps track of the relative ranking of trips

In doing this, you'll need to demonstrate knowledge on the following topics:

- Linked Structures
- Hash Tables and their various methods / approaches
- Use of recursive sorting algorithms and recursive methods



Please Note: With the advent of A2 the ban on python inbuilts is mostly lifted with the exception of the `sorted` function and `dict` inbuilt type! Feel free to use `list` and all other python functionality to its full extent.

The only exception on this exception is that if you are essentially using a list as a stack then please use the provided `LinkedStack` instead.

Additionally, using with the GUI (Graphical User Interface) for this assignment is **ENTIRELY OPTIONAL**, and is not guaranteed to work on all OSes. You shouldn't need to look at `main.py` at all.

Please make sure to read the next slide for some important information before diving into the rest of the assignment.

Please read all instructions carefully. It is useful to have a general understanding of the app and of trails before you begin; however, you do not need a deep understanding to complete most of the tasks.

To reiterate what is said in the next slide, you do **not** need to read through or change the contents of `main.py`, `draw_trails.py`, or `utils.py`. Each task will specify what files need to be edited.

The template git repository can be found here. Follow the instructions in the "Getting Started with Git" document to copy across the repository (KEEPING IT PRIVATE) and get coding.

What is the app?

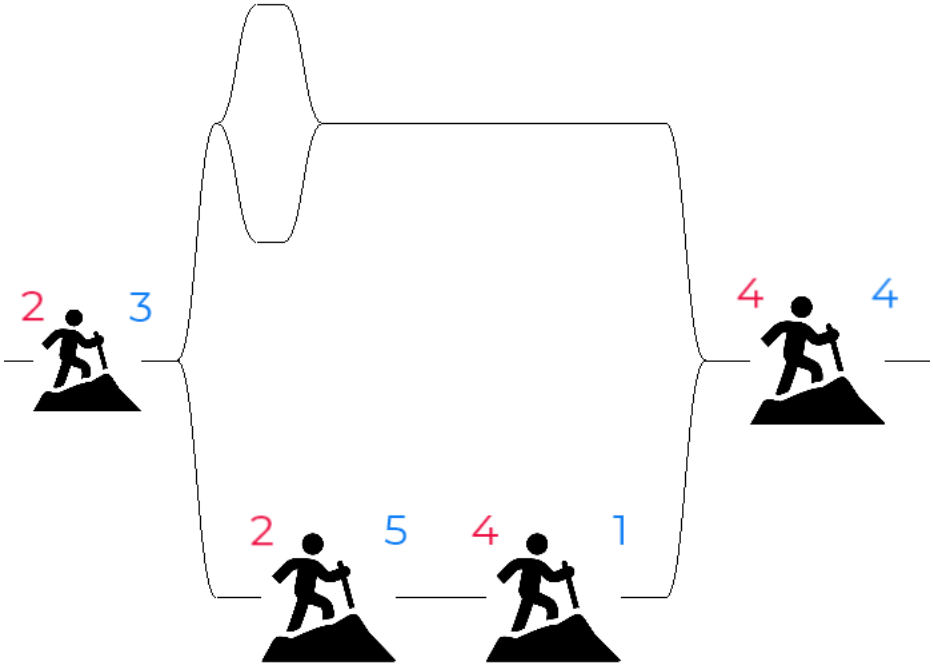
NOTE
The adding/removing mountains/branches feature won't work until you have implement the first task slide "Trail creation & Edit methods"
The graph feature won't work until you have implemented nearly everything (More Trail Methods, Mountain Manager, Mountain Organiser, Double Key Table)

This app gives information about a certain mountain range, which has various interconnected mountains.

Each mountain has three associated pieces of information:

- The name of the mountain (text)
- The difficulty level of climbing this mountain (integer)
- The length across the mountain (integer)

This can be visualised nicely by `main.py`:



Each mountain is represented by the person walking up a hill icon, and the interconnected lines represent connections between mountains.

Here the red numbers on the left of each mountain represent the difficulty level, and the blue numbers on the right of each mountain represent the length.

As part of this assignment we'll be asking many queries about this collection of mountains.

On the right sidebar of the screen you'll see a few different buttons:

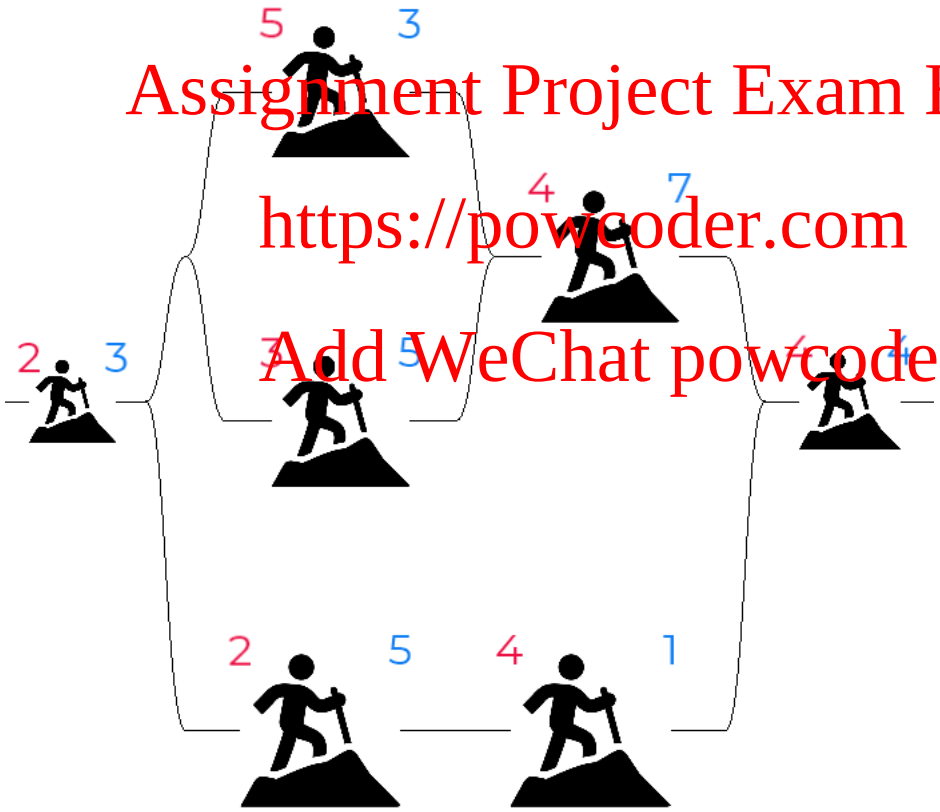
From top left to bottom right:

1. The plot icon shows a plot of mountains ranked by length as more difficulty levels are added. This is yet to be implemented, and is covered by Mountain Organiser and Manager.
2. The file icon allows you to save your current trail to a json file, which can be later loaded.

The final 4 icons are for selection modes, and once clicked change how you interact with the main view of the screen.

1. The pencil icon turns you into edit mode. Clicking on a mountain allows you to edit the difficulty, length, and name of the mountain.
2. The branch+ icon turns you into branch addition mode. Clicking on a straight horizontal line in the trail allows you to insert a branch.
3. The world+ icon turns you into mountain add mode. Clicking on a straight horizontal line in the trail allows you to insert a mountain.
4. The world- icon turns you into remove mode. Clicking on a branch or mountain in the trail allows you to remove that.

Finally, once you've made some changes and used the save icon to save your changes to a json file, you can load them back up next time using a second argument to `main.py`. For example calling `python main.py basic2.json` gives:



What is a trail?

Wiktionary defines a trail as: A route for travel over land, especially a narrow, unpaved pathway for use by hikers, horseback riders, etc.

In this app a trail is a route both *over* and *between* mountains.

Motivating the basic components of a trail

The basic trail

The most simple form of a trail is:

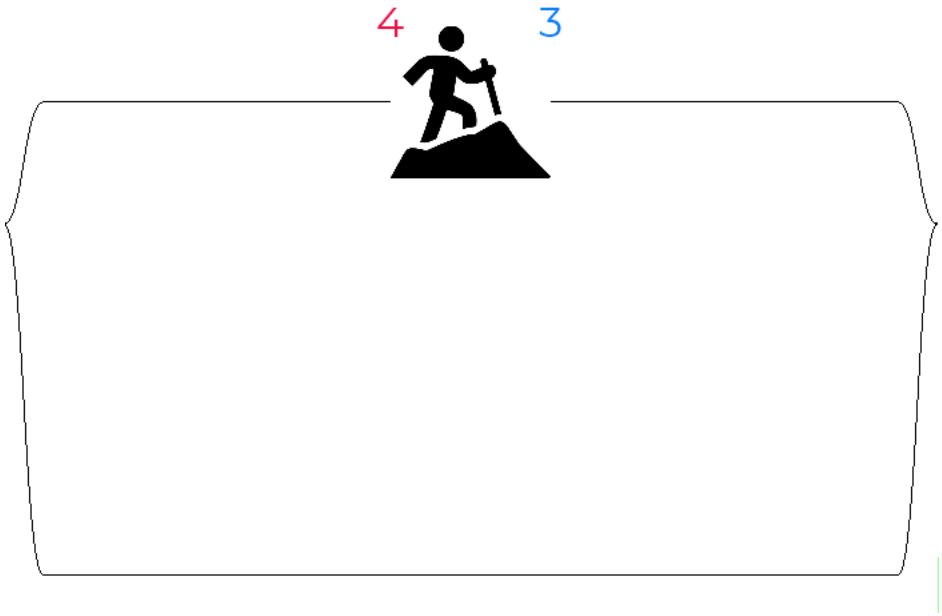
1. A line with no mountain on it; or

2. A line with a mountain on it.

Combining Trails

We can combine two trails to form a larger trail:

1. In parallel (Branching paths); or



2. In series (One after the other).

All possible trails can be generated by combining trails according to these rules.

Defining a Trail in Code

We can define a trail more formally using the following rules:

`Trail` is wrapper class* and takes exactly one argument. It can take one of the following objects: `None`, `TrailSeries`, or `TrailSplit`. These are typed as `TrailStore` in the code (this uses the typing Union, which means whenever we mention `TrailStore`, we are saying "this variable is either a `TrailSeries`, `TrailSplit` or `None`"). A Trail is represented by a `Trail` object containing some `TrailStore`, which might in turn contain more `Trail` objects in the hierarchy.

* For more information about wrappers, see this wiki article.

The **simplest trail** we can have is a ***line with no mountain***.

- This is the trail object `Trail(None)`.

We can combine **trails in sequence** by having ***a mountain and a trail***.

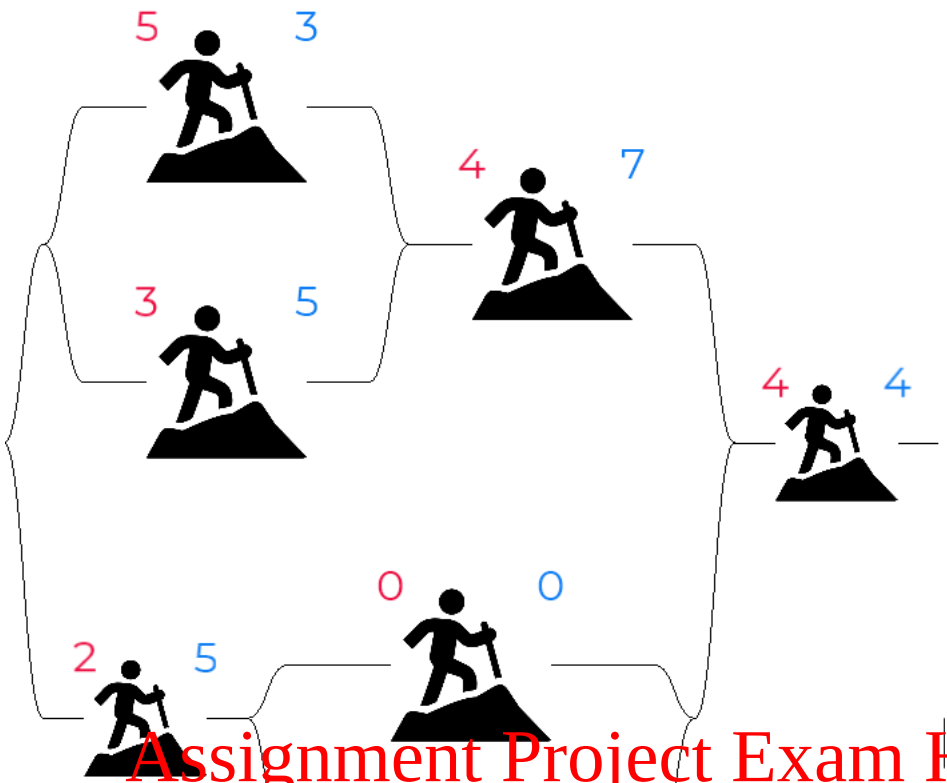
- This is the `TrailStore` object `TrailSeries(Mountain(name, difficulty, length), Trail(...))`.
- `TrailSeries` takes exactly two arguments.
- The first argument must be a mountain, and this is the *only* way we can add a mountain to a trail.
- The second argument is the remaining Trail that follows.

We can combine **trails in parallel** by having ***three trails***: two which are in parallel, and the trail that follows.

- This is the `TrailStore` object `TrailSplit(Trail(info), Trail(info), Trail(info))`.
- `TrailSplit` takes exactly three arguments.
- The first argument is the trail that splits upwards
- The second argument is the trail that splits downwards
- The third argument is the trail that follows once the previous two join together

Seeing it all in action

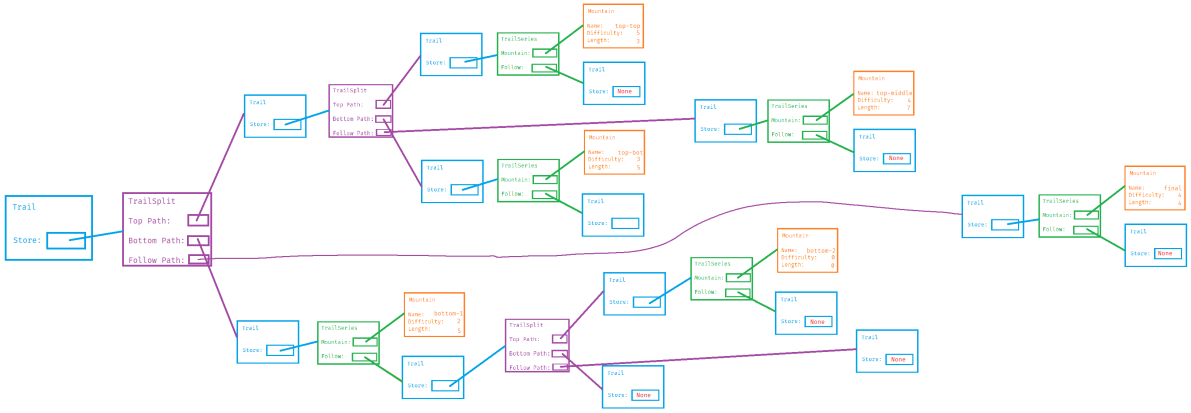
Note that every `Trail` object in code has a `store`, which contains one of the 3 objects above. So for example, we could model the below image with the following object:




```
trail = Trail(TrailSplit(
    Trail(TrailSplit(    # The top branch
        Trail(TrailSeries(
            Mountain("top-top", 5, 5),
            Trail(None),
        )),
        Trail(TrailSeries(
            Mountain("top-bot", 3, 5),
            Trail(None),
        )),
        Trail(TrailSeries(
            Mountain("top-middle", 4, 7),
            Trail(None),
        )),
    )),
    Trail(TrailSeries(   # The bottom branch
        Mountain("bottom-1", 2, 5),
        Trail(TrailSplit(
            Trail(TrailSeries(
                Mountain("bottom-2", 0, 0),
                Trail(None),
            )),
            Trail(None),
            Trail(None),
        )),
    )),
    Trail(TrailSeries(   # The following branch
        Mountain("final", 4, 4),
        Trail(None),
    )),
```

```
))
```

Which also looks like the following memory diagram (click the image to enlarge):



[TASK] Trail creation & Edit methods

It is highly recommend you fully read and understand the slide "What is a trail?" before continuing.

Complexity Analysis is not required for this task.

In this task, you'll be working on:

`trail.py`

The first task is to add a few helper methods to the Trail definitions in `trail.py` to support editing the Trails.

In `trail.py`, you'll find the following methods on a few different classes, already type hinted for you:

- `remove_branch`
- `remove_mountain`
- `add_mountain_before`
- `add_empty_branch_before`
- `add_mountain_after`
- `add_empty_branch_after`

These methods can be used to add and remove elements from the trail. They **should not modify** the existing Trail, but **create new Trail objects** representing *what the trail would look like* if edited.

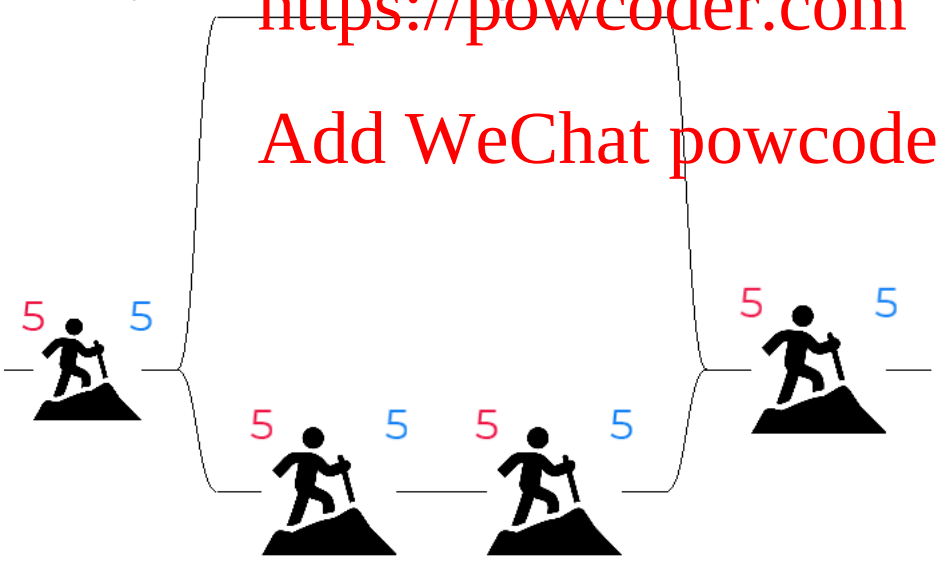
If this method is defined on the `Trail` class, you should return a new `Trail` instance, which has completed the action specified.

If this method is defined on a `TrailStore` class, you should return a new `TrailStore` instance, which has completed the action specified.

For example, suppose we have the following:

```
a, b, c, d = (Mountain(letter, 5, 5) for letter in "abcd")

empty = Trail(None)

series_b = TrailSeries(b, Trail(TrailSeries(d, Trail(None))))

split = TrailSplit(
    empty,
    Trail(series_b),
    Trail(TrailSeries(c, Trail(None)))
)

t = Trail(TrailSeries(
    a,
    Trail(split)
))
```


This is represented by the following diagram:



Calling `series_b.add_empty_branch_after()` returns the following object:

```
TrailSeries(
    mountain=b,
    following=Trail(store=TrailSplit(
        top=Trail(store=None),
        bottom=Trail(store=None),
        following=Trail(store=TrailSeries(
```

```
        mountain=d,
        following=Trail(store=None)
      ))
   ))
)
```

While calling `split.remove_branch` should return the following object:

```
TrailSeries(
   mountain=c,
   following=Trail(store=None)
)
```

And calling `empty.add_empty_branch_before()` should return the following object:

```
Trail(store=TrailSplit(
   top=Trail(store=None),
   bottom=Trail(store=None),
   following=Trail(store=None)
))
```


[TASK] Traversing Trails with Terrific Tricks

It is highly recommend you fully read and understand the slide "What is a trail?" before continuing.

In this task, you'll be working on:

`trail.py`

Now that we have defined our Trail objects, and can add extra objects on-top of them, we want to actually walk them!

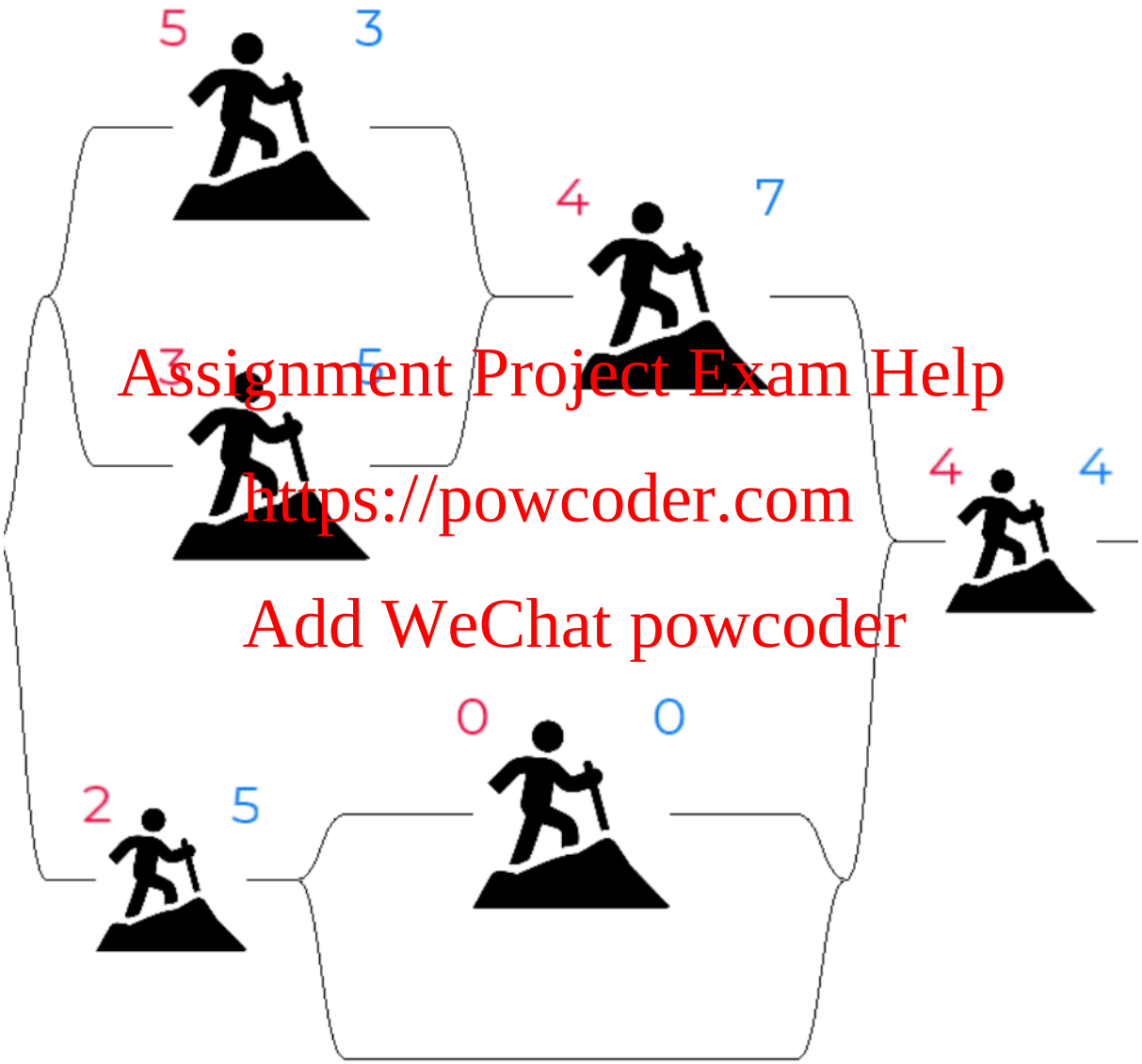
Now since we have splits in the road, we must decide which path to take. That's where Personalities come in!

A `WalkerPersonality` is a class that implements two methods:

- `add_mountain`, which allows a Walker to note that mountain they've just walked by
- `select_branch`, which given two Trails as input, decides which of them they will take by returning the Enum `PersonalityDecision`. If `STOP` is returned, then neither branch should be taken and the path ends here.

Your task is to, *without using recursion*, implement the Trail method `follow_path`, which takes in an argument `personality` of type `WalkerPersonality`, and on your trail, calls `personality.add_mountain` for every mountain on your trail this walker personality would pass by.

For example, if `personality` defined `select_branch` as `return PersonalityDecision.TOP` (Always choose the top (first) branch), then in the previous example:



We would call `personality.add_mountain` with Mountains named `top-top`, `top-middle` and `final` .

For (somewhat) more complicated personalities, see `personality.py`.

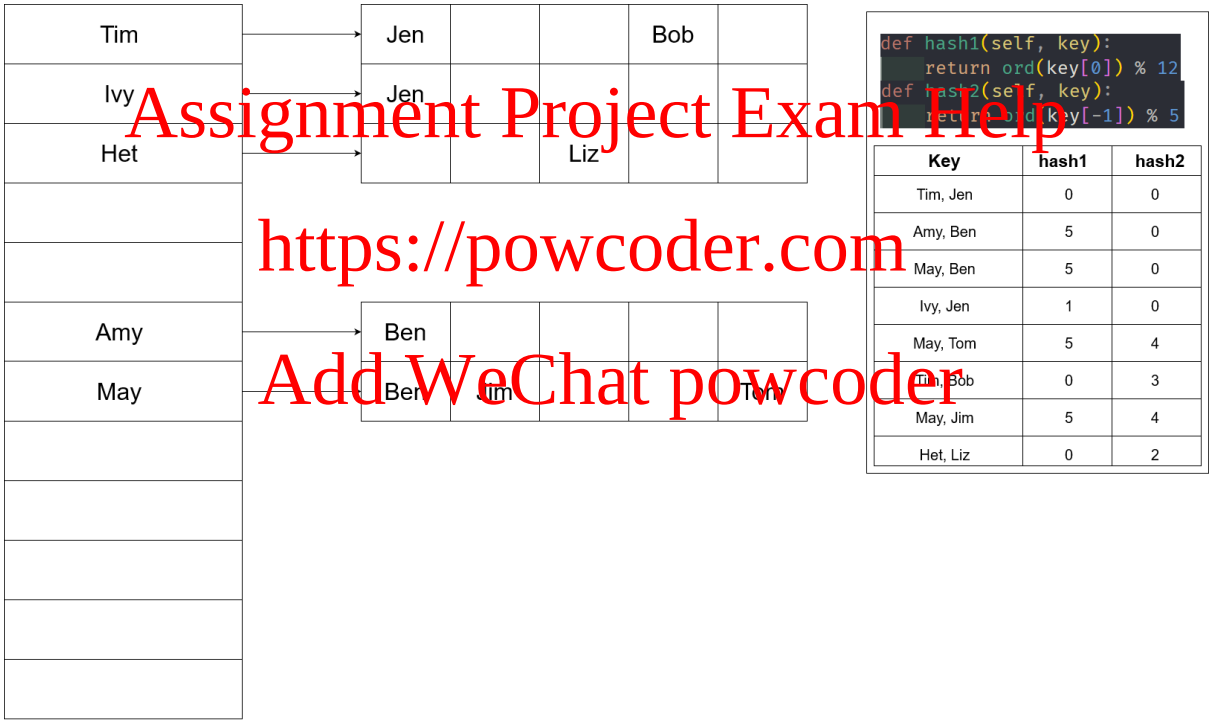
[TASK] Double Keyed Table

In this task and the next, we'll be working on some Data Structures that take inspiration from Hash Tables but make some important changes to the way probing and storage are done. You'll use these structures to solve other problems later in the spec sheet.

In this task, you'll be working on:

`double_key_table.py`

For this first data structure, we'll be working on a Hash Table that takes two keys rather than one. In terms of storage, this can be thought of as a hash table of hash tables, where a top-level key is used to determine the first position (which hashtable to insert into) and a bottom-level key is used to determine where in this selected hashtable to insert into:



Key	hash1	hash2
Tim, Jen	0	0
Amy, Ben	5	0
May, Ben	5	0
Ivy, Jen	1	0
May, Tom	5	4
Tim, Bob	0	3
May, Jim	5	4
Het, Liz	0	2

Here, both the top-level and lower level hash tables use Linear Probing to resolve collisions (Note that Het probes from 0->1->2 and "May, Jim" probes from 4->0->1). (The same example is used for `test_double_hash.py` in `test_example`.)

Your `DoubleKeyTable` should implement the following methods:

- `__init__(self, sizes=None, internal_sizes=None)`, create the underlying array. If `sizes` is not None, the provided array should replace the existing `TABLE_SIZES` to decide the size of the top-level hash table. If `internal_sizes`

is not None, the provided array should replace the existing `TABLE_SIZES` for the internal hash tables (See `hash_table.py` for an example)).

- `_linear_probe(self, key1: K1, key2: K2, is_insert: bool) -> tuple[int, int]`, return the:
    - Index to access in the top-level table, followed by
      Index to access in the low-level table
      In a tuple.
    -
    - Your linear probe method should create the internal hash table if `is_insert` is true and this is the first pair with `key1`.
- `keys(self, key:K1|None = None) -> list[K1|K2]`
    - If key = None, return all top-level keys in the hash table
    - If key != None, return all low-level keys in the sub-table of top-level `key`
- `values(self, key:K1|None = None) -> list[V]`
    - If key = None, return all values in all entries of the hash table
    - If key != None, restrict to all values in the sub-table of top-level `key`
- `iter_keys` and `iter_values`: The same functionality as above, but this should return an iterator that yields the keys/values one by one rather than searching the entire table at the start. You should NOT get all the keys/values at the start and just iterate through those, that won't be very efficient. Your iterator should only get the next item when it's needed.
- Have your code use `hash1` and `hash2` from the `DoubleKeyTable` that is already defined.
- Have both your top-level table and internal tables resize when the load factor of that table increases past 0.5 (See `hash_table.py` for an example of this logic) These resizes should occur independently (One internal table may be a different size to another, and the top level table should resize when the number of internal tables exceeds 0.5 irrespective of the resizing of the internal tables)
- `__getitem__`, `__setitem__`, and `__delitem__`. When deleting, if the key1,key2 pair was the only key1 element in the table, you should clear out the entirety of that internal table so that a new `key1` with the same hash can be inserted in that position. (See `test_delete` for more info.)
- `table_size(self)` . Returns the current size of our table.

Tip: When creating a new internal hash table, be sure to set `table.hash = lambda k: self.hash2(k, table)`. This ensures any internal table uses `hash2` for hashing keys.

[TASK] Infinite Depth Hash Table

This task is not part of the "Solo Standards" - so if for whatever reason your group was split, you don't have to submit this task.

In this task, you'll be working on:

`infinite_hash_table.py`

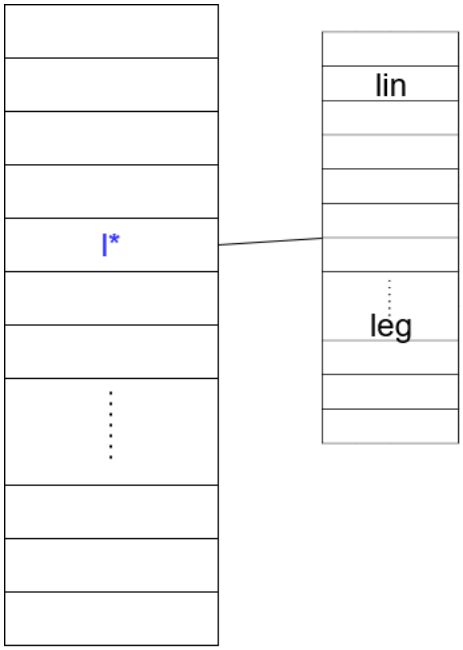
Time for another Hash Table implementation! This one is a bit different. We go back to having a single key, but rather than resolving collisions with probing, we resolve collisions by simply making more hash tables with new hash functions!

Your hash function will now use an instance variable called `level`. `key` is as it was before - the key being inserted into the table. `level` specifies what level of the hash table hierarchy we are hashing for. This will become more obvious with a worked example. You can assume that all keys that go into this table are strings of lowercase english letters.

Using the following hash function:

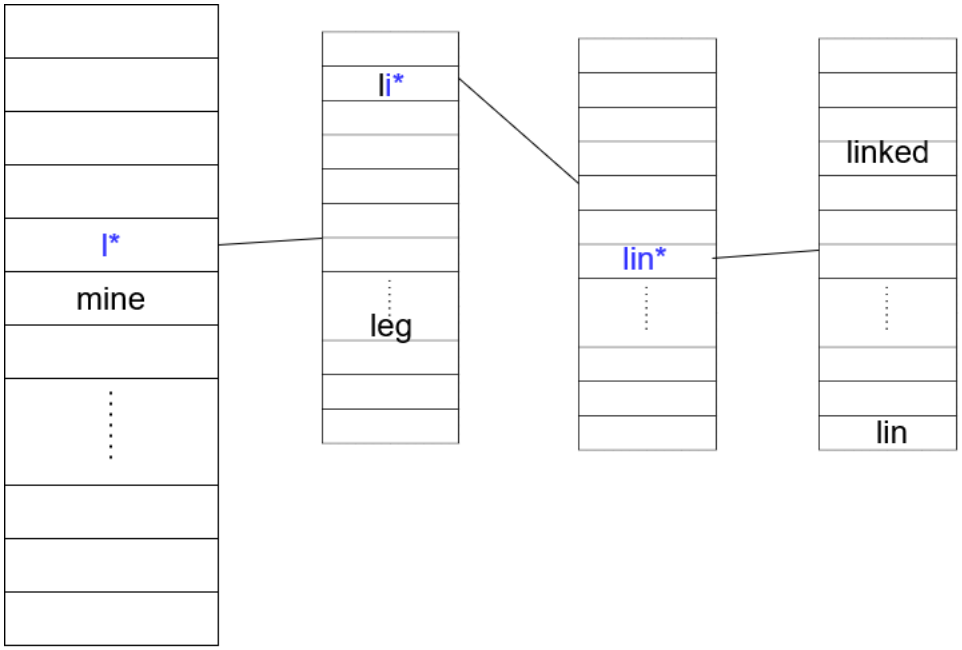
```
class InfiniteHashTable
    TABLE_SIZE = 97

    def hash(self, key: K) -> int:
        if self.level < len(key):
            return ord(key[self.level]) % (self.TABLE_SIZE-1)
        return self.TABLE_SIZE-1
```


Adding `lin` and `leg`, we'd make a new table at the position 4, resulting in the following diagram:



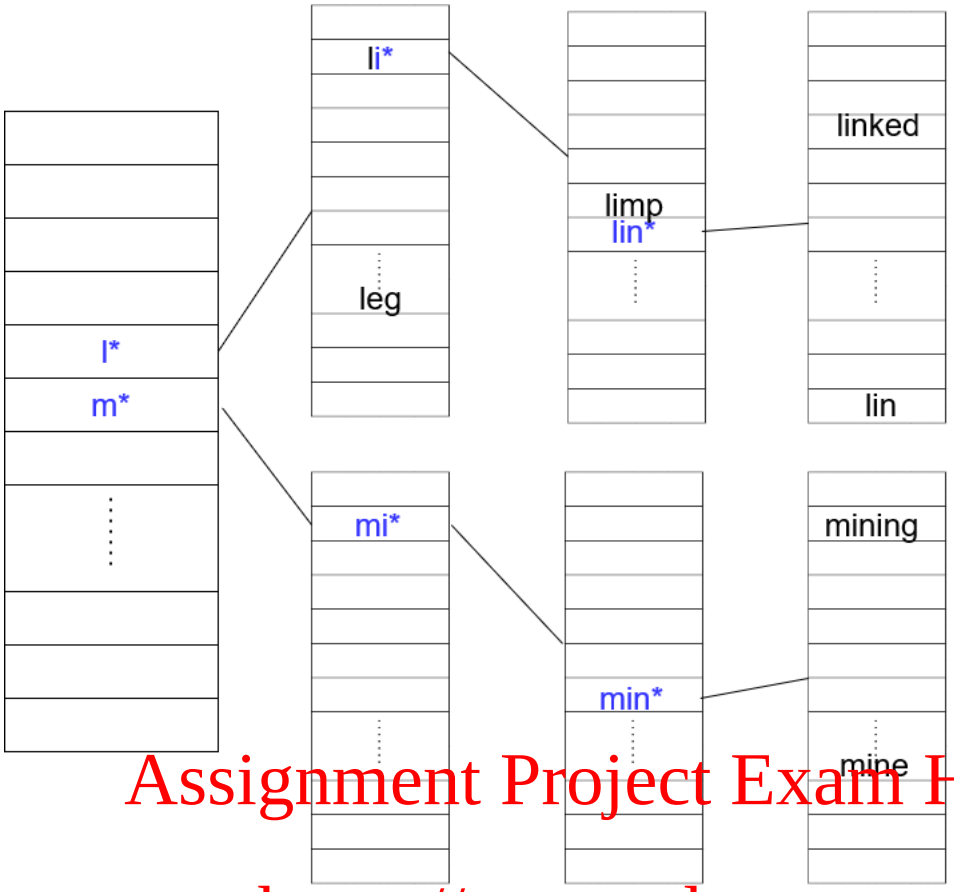
Key	Hash L0	Hash L1	Hash L2	Hash L3
lin	4	1	6	25
leg	4	23	25	26
mine	5	1	6	23
linked	4	1	6	3
limp	4	1	5	8
mining	5	1	6	1
jake	2	19	3	23
linger	4	1	6	25

Next, after inserting `mine` and `linked`, `mine` would be inserted at the top level, and what was previously the location for `lin` now needs to be separated into a new table:



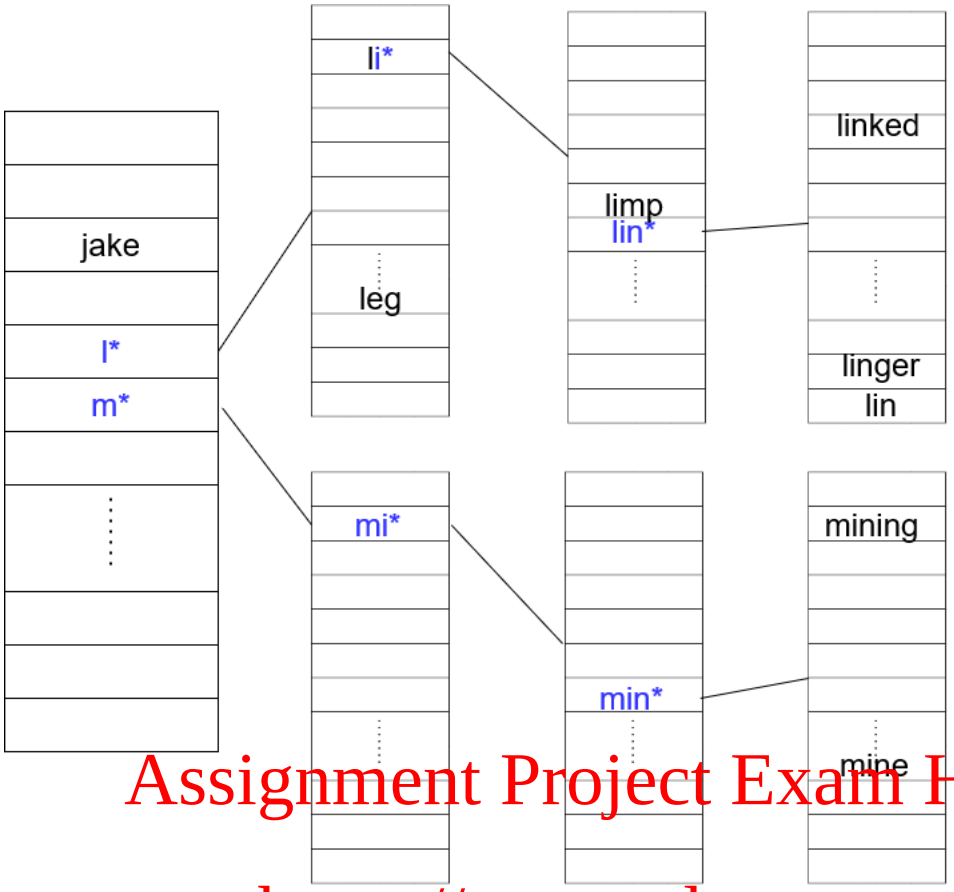
Key	Hash L0	Hash L1	Hash L2	Hash L3
lin	4	1	6	25
leg	4	23	25	26
mine	5	1	6	23
linked	4	1	6	3
limp	4	1	5	8
mining	5	1	6	1
jake	2	19	3	23
linger	4	1	6	25

Next, adding `limp` and `mining` will first add `limp` in the table corresponding to `li`, while `mining` will split the location formerly hosting `mine`:



Key	Hash L0	Hash L1	Hash L2	Hash L3
lin	4	1	6	26
leg	4	23	25	26
mine	5	1	6	23
linked	4	1	6	3
limp	4	1	5	8
mining	5	1	6	1
jake	2	19	3	23
linger	4	1	6	25

Finally, adding jake just sits at the top level (no collision), and adding linger navigates to the `lin*` table and inserts there:



Key	Hash L0	Hash L1	Hash L2	Hash L3
lin	4	1	6	26
leg	4	23	25	26
mine	5	1	6	23
linked	4	1	6	3
limp	4	1	5	8
mining	5	1	6	1
jake	2	19	3	23
linger	4	1	6	25

Your task is to define the following methods for `InfiniteHashTable`:

- `__init__`: Initialises the table. You may add optional arguments here if you like.
- `__getitem__` : Retrieves a value based on it's key.
- `__setitem__` : Sets a value based on the key.
- `__delitem__` : Remove a key/value pair. If this pair removed leaves only a single pair within the current table, then the current table should be

collapsed to a single entry in the parent table (This should continue upwards until there is a table with multiple pairs *or* we are at the top table). See `test_delete` for an example.

- `get_location` : Returns a list containing the indices required to retrieve a key. In the example above, `get_location(linger) == [4, 1, 6, 25]`

And finally, after you've implemented everything from before, there's one more function to implement: `sort_keys`. This function should return all keys that are currently in the table in lexicographically sorted order. Lexicographically sorted simply means that we compare a string letter by letter, and compare using the rule

$a<b<c<d<\cdots$

And prefixes of text are always smaller than the full text.

[TASK] Mountain Organiser

For this task, you may assume that all Double Key Table and all Infinite Hash Table methods are O(1), even if they obviously are not.

In this task, you'll be working on:

`mountain_organiser.py`

As you embark on your Mountain Climbing journey, you find that as time passes, the number of mountains you can climb increases. For the first week, maybe you can only do low-altitude, non-steep mountains, and as time goes on, you can climb more and more difficult mountains.

As you climb more and more mountains, you'd like to know the rank of each mountain, if all the mountains you've seen are ranked by their difficulty increasing. In cases where the difficulty is the same, you should order them by name lexicographically increasing (you can assume this is unique)

To achieve this, we'd like you to implement the `MountainOrganiser` class (in `mountain_organiser.py`), which requires 3 methods:

- `__init__`: Initialisation
- `add_mountains(self, mountains: list[Mountain]) -> None`: Adds a list of mountains to the organiser
- `cur_position(self, mountain: Mountain) -> int`: Finds the rank of the provided mountain given all mountains included so far. See below for an example. Raises `KeyError` if this mountain hasn't been added yet.

Complexity Requirement

`add_mountains` should have complexity at most □(□log⁡(□)+□)$O(M\log(M)+N)$, where □M is the length of the input list, and □N is the total number of mountains included so far.

`cur_position` should have complexity at most □(log⁡(□))$O(\log(N))$, where □N is the total number of mountains included so far.

Consider the following example:

Name	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
Difficulty	2	9	6	1	6	3	7	8	6	4
Length	2	2	3	3	4	7	7	7	7	8

Add m1, m2

Name	m1	m2
Difficulty	2	9
Rank	0	1

Add m3, m4

Name	m4	m1	m3	m2
Difficulty	1	2	6	9
Rank	0	1	2	3

Add m5

Name	m4	m1	m3	m5	m2
Difficulty	1	2	6	6	9
Rank	0	1	2	3	4

Add m6, m7, m8, m9

Name	m4	m1	m6	m3	m5	m9	m7	m8	m2
Difficulty	1	2	3	6	6	6	7	8	9
Rank	0	1	2	3	4	5	6	7	8

See `test_mountain_organiser.py` for a code example of this diagram.

[TASK] Mountain Manager

For this task, you may assume that all Double Key Table and all Infinite Hash Table methods are O(1), even if they obviously are not.

This task is not part of the "Solo Standards" - so if for whatever reason your group was split, you don't have to submit this task.

In this task, you'll be working on:

`mountain_manager.py`

In order to make the graph view work, you'll need the Mountain Organiser implemented as well as this class: The Mountain Manager.

The mountain manager acts as a store which tracks all mountains in a trail (In other words, all mountains showing in `main.py`) and can be used to edit, add or remove mountains.

Mountains can also be filtered by difficulty, and a list of list containing all mountains, grouped by difficulty, in ascending order, can be generated.

On the file `mountain_manager.py` , implement the following methods:

- `__init__(self) -> None`: initialisation
- `add_mountain(self, mountain: Mountain) -> None` : Add a mountain to the manager
- `remove_mountain(self, mountain: Mountain) -> None`: Remove a mountain from the manager
- `edit_mountain(self, old_mountain: Mountain, new_mountain: Mountain) -> None`: Remove the old mountain and add the new mountain.
- `mountains_with_difficulty(self, diff: int) -> list[Mountain]`: Return a list of all mountains with this difficulty.
- `group_by_difficulty(self) -> list[list[Mountain]]`: Returns a list of lists of all mountains, grouped by and sorted by ascending difficulty.

See `tests/test_mountain_manager.py` for an example of this in action.

[TASK] More Trail methods

Complexity Analysis is not required for this task

In this task, you'll be working on:

`trail.py`

Now that we can do anything we want in the `main.py` GUI, there's a few more things we'd like to now about trails.

First off, we'd like to generate a list of all mountains contained within the Trail.

> In `trail.py`, implement the method `collect_all_mountains`, which returns a list of Mountains that are within this trail. This should run in O(N) time, where N is the total number of mountains and branches combined.

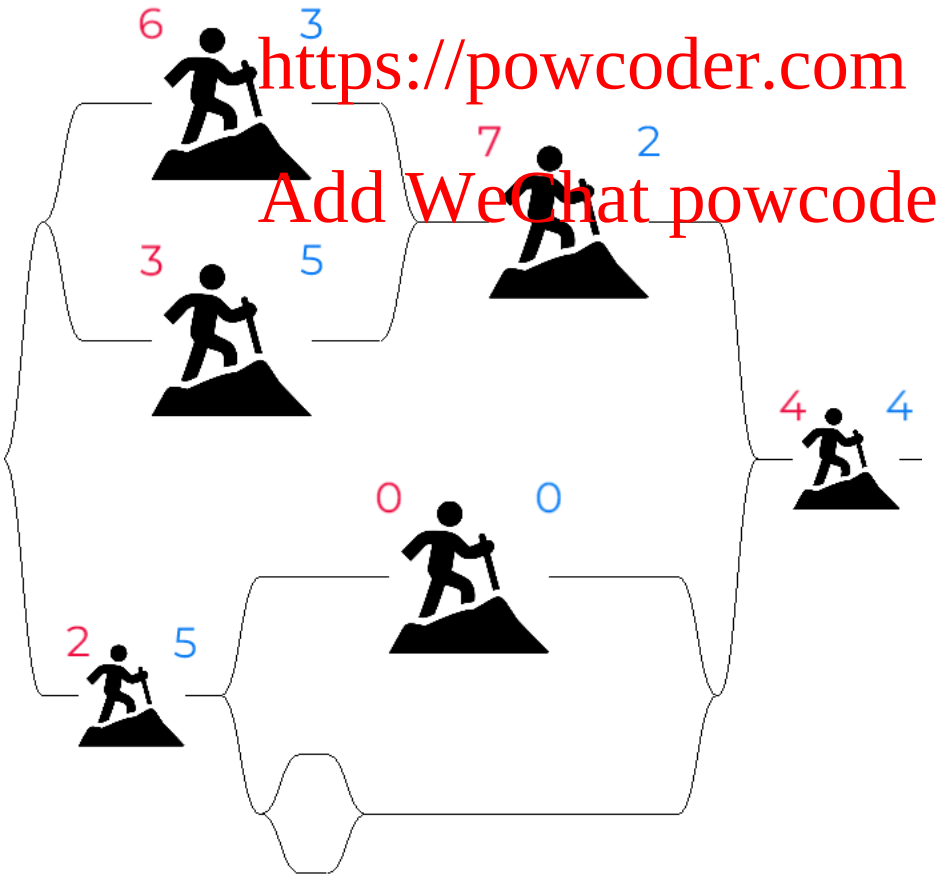
Once you've completed this, we'd also like to calculate all paths we could have taken through the trail, fitting some requirements.

> In `trail.py`, implement the method `difficulty_maximum_paths(self, diff: int) -> list[list[Mountain]]`. For this function, you can assume that no more than 5 branches are present in the trail used.

This method, when given an integer `diff`, should calculate all paths through the trail such that the maximum difficulty of all mountains in the path does not exceed `diff`. The return value of this should be a list containing lists, containing the Mountains on each path, in order taken in the path.

Paths are considered distinct if any branch decision chosen is different, even if the mountains traversed would be the same.

For example, for the following:



Calling `trail.difficulty_maximum_paths(5)` should return a list of size 3, containing the following (in any order):

- `[bot-one, bot-two, final]`
- `[bot-one, final]`
- `[bot-one, final]`

This is because any path taking the top branch must go through top-mid, with difficulty 7.

bot-one final is included twice because we can take either the top or bottom paths of the empty branch.