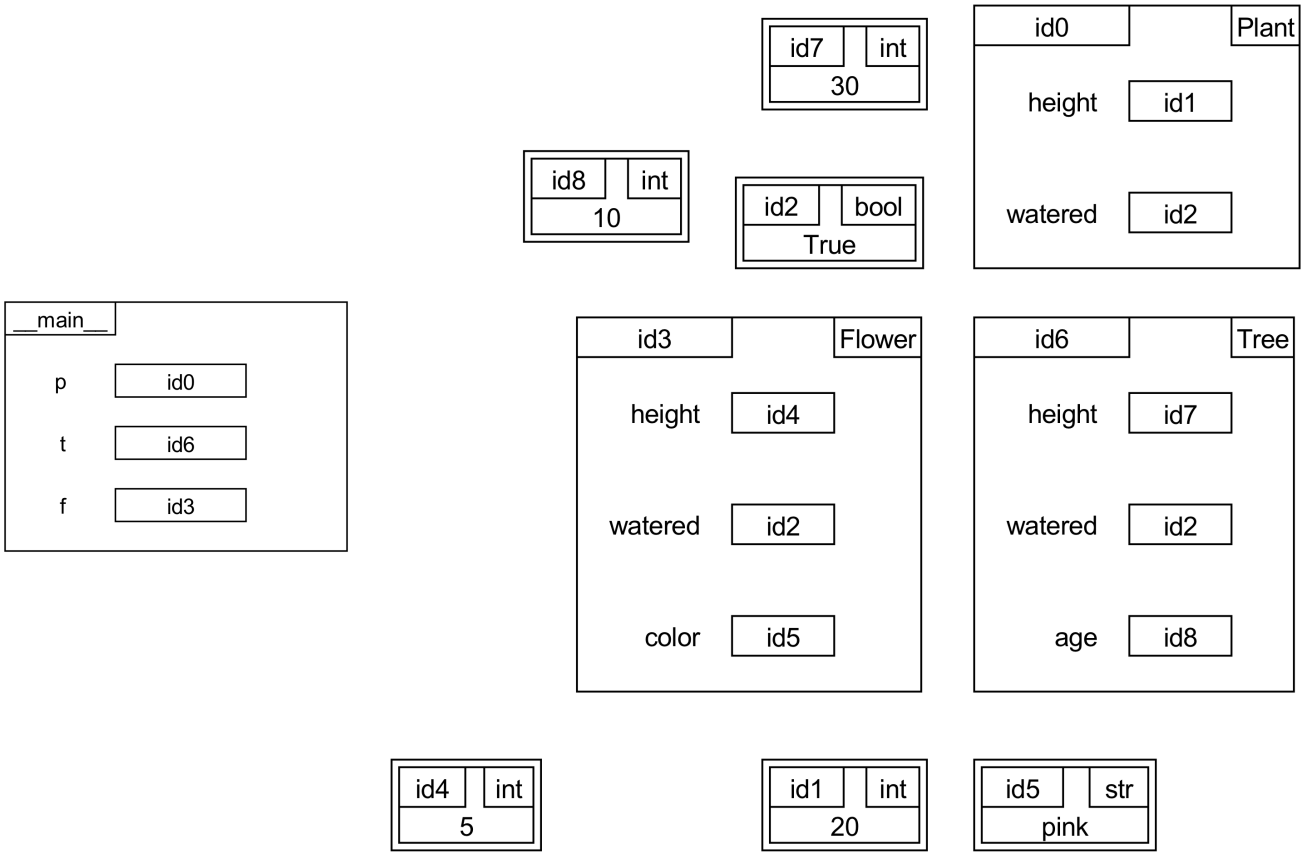
Question 1. [6 MARKS]

Consider the following client code which produces the memory model shown below.
On the next page, write code for the three classes required by the client code. Do not write any docstrings, but you must include type annotations. There must be an inheritance relationship in your code, and the parent class does not have to be abstract.

```
if __name__ == '__main__':

    p = Plant(20, True)

    t = Tree(30, 10)

    f = Flower(5, 'pink', True)
```



In the space below, write your classes. You do not need to include docstrings.

The init methods need to be consistent with the client code calls:

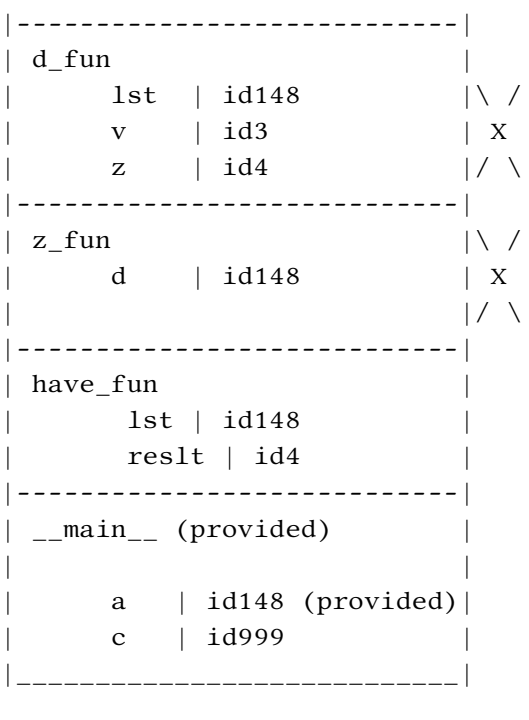
```
p = Plant(20, True)

t = Tree(30, 10)

f = Flower(5, 'pink', True)
```

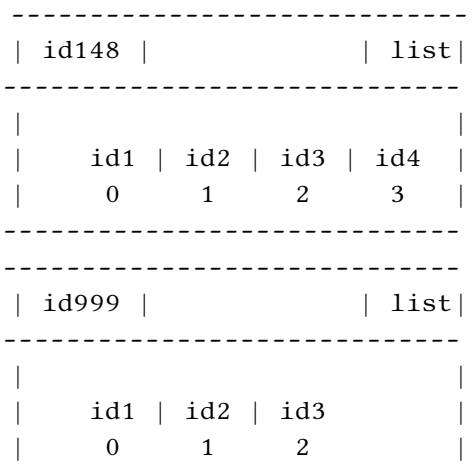
```
class Plant:
    height: int
    watered: bool

    def __init__(self, height: int, watered: bool) -> None:
        self.height = height
        self.watered = watered
        # could have default parameters if wanted


class Tree(Plant):
    age: int

    def __init__(self, height: int, age: int) -> None:
        Plant.__init__(height, True)
        self.age = age
        # could have default parameters if wanted


class Flower(Plant):
    colour: str

    def __init__(self, height: int, color: str, watered: bool) -> None:
        Plant.__init__(height, watered)
        self.colour = colour
        # could have default parameters if wanted
```

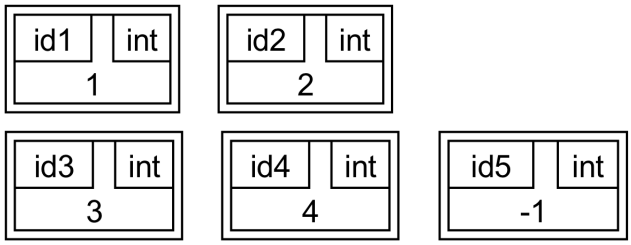
Question 2. [7 MARKS]

Part (a) [5 MARKS] Consider the following code. Draw the contents of the call stack **immediately before** `have_fun` returns. Draw any objects as needed in the space below to complete the memory model. If any stack frames were popped before `have_fun` returns, cross them out to indicate that they were popped. We have executed the first line of code in the stack frame for `__main__` to get you started, but you will have to draw the object with `id148`.

```
def d_fun(lst: list[int], v: int) -> None:
    z = v + 1
    lst.append(z)

def z_fun(d: list[int]) -> int:
    d_fun(d, 3)
    return d[-1]

def have_fun(lst: list[int]) -> int:
    rslt = z_fun(lst)
    return rslt

if __name__ == "__main__":
    a = [1, 2, 3]
    c = a[:]
    print(have_fun(a))
    print(c)
```







Part (b) [2 MARKS] In the space below, write what is printed when the code is executed.

```
4
[1, 2, 3]
```

Question 3. [6 MARKS]

Part (a) [1 MARK] Consider the `MyList` class below.
What is the name of the relationship between the `MyList` class and the `Queue` class? Composition

Part (b) [5 MARKS] Implement the `remove` method according to its docstring. Do not make any assumptions about the behaviour of the `Queue` class other than that it has the API specified on the aid sheet. You may not define any other methods or functions. You may not use any builtin python lists, sets, tuples, or dictionaries — use `Stack` or `Queue` objects instead. Comments are not required but are encouraged.

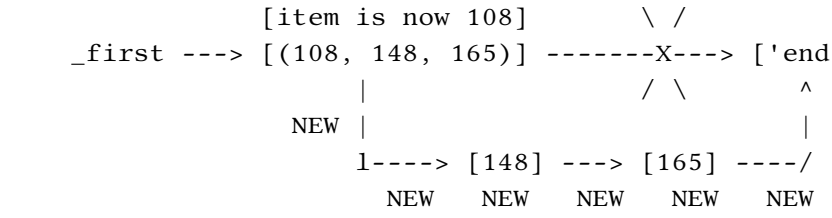
```
class MyList:
    """A list implemented using a Queue.
       The item at index 0 in this MyList is stored at the front of _queue"""
    _queue: Queue

    def __init__(self) -> None:
        """ Initialize <self> to represent an empty list. """
        self._queue = Queue()

    def remove(self, item: Any) -> None:
        """Remove the first occurence of <item> from this list.
        Raise a ValueError if <item> is not in this list."""



        # the code:

        temp_queue = Queue()
        cur_idx = 0
        found = False
        while not self._queue.is_empty():
            value = self._queue.dequeue()
            if found or value != item:
                temp_queue.enqueue(value)
            else:
                found = True  # don't put the item back on!

        # self._queue = temp_queue  # easy way
        while not temp_queue.is_empty():
            self._queue.enqueue(temp_queue.dequeue())

        if not found:
            raise ValueError
```

Question 4. [7 MARKS]

Consider the following docstring description of a new `LinkedList` method:

```
def expand_all_tuples(self) -> None:
    """Mutate <self> so that every tuple has been "expanded".
    Expanding a tuple of length n results in n-1 new nodes, each containing
    one item from the original tuple, in the original order. The original node
    contains the first item from the original tuple.

    Preconditions:
    - each tuple in self has length > 0

    >>> lst = LinkedList([('hi', 'there'), 42])
    >>> print(lst)  # the original list
    ('hi', 'there') -> 42
    >>> lst.expand_all_tuples()
    >>> print(lst)  # the mutated list after calling the method
    hi -> there -> 42
    """
```

Part (a) [2 MARKS] Modify the simplified `LinkedList` object diagram below to show how this `LinkedList` object will look after `expand_all_tuples` is executed. Hint: `self._first.item` will be mutated.



Note: the description says we modify the item of the original tuple-containing node!

Part (b) [5 MARKS] Implement this method by filling in the blanks in the partially complete code below. You must not call any other `LinkedList` methods. Do not add any additional lines of code or remove any.

```
def expand_all_tuples(self) -> None:
    """Mutate <self> so that every tuple has been "expanded".
    Expanding a tuple of length n results in n-1 new nodes, each containing
    one item from the original tuple, in the original order. The original node
    contains the first item from the original tuple.

    Preconditions:
    - each tuple in self has length > 0"""

    cur = ___ self._first ___


    while ___ cur is not None ___:  # traverse one node at a time


        if isinstance(___ cur.item ___, tuple):  # check if we are at a tuple


            part = cur.item


            cur.item = part[0]


            part = part[1:]


            while len(part) > 0:  # while the tuple still has items to expand


                node = ___ _Node(part[0]) ___  # create a new node object


                node.next, cur.next = ___ cur.next, node ___  # link new node in


                cur = ___ cur.next  # or cur = node ___  # advance cur


                part = ___ part[1:] ___  # the rest of the tuple to expand


        ___ cur = cur.next ___
```