Module 8 Homework - Resolving hash collisions

Overview

We'll resolve hash collisions using two techniques:

1) Separate chaining - when multiple items hash to the same bucket, we put them into that same bucket
    - In class and the textbook, we used **lists** as our buckets, resulting in a "list of lists"
    - Here, we'll use **linked lists** as our buckets, resulting in a "list of linked lists"
2) Open addressing - store only a single item in each "bucket". When a collision occurs, find the next open address by...
    - **linear probing** - scan ahead for the next empty bucket, 1 item at a time
    - other techniques for finding an open address include **quadratic probing** and **double hashing**, but we'll only do linear probing here.

Part 1: Separate chaining

We'll need linked list and node classes appropriate for hashing.

`class UniqueRecursiveNode`

We'll practice recursion by implementing recursive methods for a linked list of nodes with unique keys.

- Store keys and values when initialized
- `__eq__` and `__hash__` methods must be overloaded
    - two nodes are equal if they have the same key, even if their values or links are different.
    - two nodes with the same key should hash to the same number, even if their values or links are different.
    - Custom classes inherit `__eq__` and `__hash__` methods from the default `Object` class, but these methods do not work here because they are based on memory ID.
- Add recursive implementations for:
    - `add(key, value)`
        * `O(n)`
        * updates value of node with `key` or addes a new node with `key:value` pair, as appropriate
        * Returns *the number of nodes added* (either 0 or 1), so the `LinkedList` class can update `_len` appropriately
    - `get(key)`
        * `O(n)`
        * Returns value associated with key
        * Raises a `KeyError` if key not found
    - `remove(key)`

        * `O(n)`
        * removes node containing `key` and returns tuple of `new_link, value`
            - `new_link` - the node the preceding node should link to after this operation
            - `value` - the value associated with `key`
        * raise a `KeyError` if this key is not found
    - `contains(key)`
        * `O(n)`
        * returns True iff key is in the linked list starting at this node
- `__iter__()`
    - This is implemented for you. We'll discuss iterators more during Mod 9.

`class UniqueLinkedList`

You don't need to change anything here; it's implemented for you. Look over the methods to familiarize yourself with how a linked list uses recursive nodes - we may ask you to code something like this on an exam.

Note that we do not use a tail pointer. Our ideal use case is a large number of short linked lists, so the tail pointers add significant memory overhead without improving speed.

`SeparateChainingHashTable`

- Use the LinkedList class implemented above for your buckets
- Resolve hash collisions via separate chaining
- `__init__`
    - O(1)
    - Include optional parameters with the following names and default values:
        * MINBUCKETS: 2 - the minimum number of buckets in your hash map
        * MINLOADFACTOR: 0.5 - the minimum allowable load factor (unless removing buckets would move you below MINBUCKETS)
        * MAXLOADFACTOR: 1.5 - the maximum allowable load factor
    - Initialize with a list of `MINBUCKETS` `UniqueLinkedLists`
- `__setitem__(key, value)`
    - O(1) amortized
    - adds or updates (key, value) pair, as appropriate
- `__getitem__(key)`
    - O(1) amortized
    - returns value associated with key or raises a KeyError, as appropriate
- `__contains__(key)`

    - O(1) amortized
    - returns `True` or `False`, as appropriate

- `pop(key)`
    - O(1) amortized
    - removes key:value pair and returns `value` or raises a KeyError, as appropriate

- `get_loadfactor()`
    - O(1)
    - Returns the current load factor

- Maintain `O(n)` memory overhead by ensuring $MINLOADFACTOR \leq \alpha < MAXLOADFACTOR$, where α is the load factor (number of items / number of buckets)

Part 2 - Open Addressing

`LinearProbingHashTable`

Use the same public interface as `LinkedListHashTable`. However, resolve hash collisions with open addressing via linear probing:

- Initilalize the `HashTable` with a list of `None` objects.
- After hashing a key, if the desired bucket is empty, replace the appropriate `None` with a tuple of (`key`, `value`).
- If the bucket is occupied, iterate through the list to find the next empty spot.
- When removing, replace the tuple with `-1` instead of `None` to denote that something used to be there.
    - This is necessary for e.g. O(1) amortized `contains`, since you cannot return False until you find a spot this `key` could have been in but isn't. Here, that is equivalent to finding the `None` object (but *not* a `-1`) in a bucket.
    - If you find a `-1` while probing for an empty spot during e.g. `setitem`, you can overwrite it - do not keep scanning for a `None`.
- Your load factor must be `< 1` to avoid infinite loops while probing:
    - Use default values of `MINLOADFACTOR = 0.1` and `MAXLOADFACTOR=0.75`
    - If a user specifies a `MAXLOADFACTOR` ≥ 1, raise a `ValueError`

Testing

Tests for `UniqueRecursiveNode` and `UniqueLinkedList` are provided for you.

Remember to only test the *public interface* - do not test or access private variables or methods in your unittests.

Test Hash Tables

- `SeparateChaningHashTable` and `LinearProbingHashTable` have the same public interface, so use a test factory to factor out redundant unittests.

- "Test behavior, not implementation" is an important idiom here - there's not a great way to write unittests for amortized running times of O(1) or memory usage of O(n), but these are results of *implementations*, so we don't need to test them. Just test the input/output behavior for every method.
- The starter code includes skelteon code for tests you should write, including:
  - `test_put_get_sequential` - iteratively add items, and test that you get correct results from:
    - `__getitem__` (call w/ the public interface e.g. `my_hashtable[key]`)
    - `__len__`
    - `__contains__`
    - `get_loadfactor()`
    - pseudo code:

      ```
      # Create empty hash map
      # for i in range n:
      #     assert i not in n
      #     add i, value to hash map
      #     assert hm[i] gives correct value
      #     assert i in hm
      #     assert len is correct
      #     assert load factor is in correct range
      ```
  - `test_put_get_remove_sequential` - iteratively add items, then remove
    - build a hash map as above, then iteratively pop all keys, testing as you go
  - `LinearProbingHashTable` needs 1 additional test - ensure a `ValueError` is raised if a user specifies `MAXLOADFACTOR >= 1`.
  - Not required for this assignment, but if you really want to test that your code works:
    - Test how you handle duplicate key:value pairs (should update value but not length). See `TestUniqueLinkedList` for an example
    - Test how you handle random adds. It's helpful to use a python dictionary as a template for your "expected" value here:

      ```
      # Create empty hash map
      # Create empty dictionary
      # for i in range n:
      #     generate random key:value
      #     add to dictionary
      #     add to hash map
      #     test that dictionary and hash map are the same
      ```

Imports

No imports are allowed on the assignments, with the following exceptions:

- `unittest` and `random` - for testing.
- `typing` for type validation - not required, but feel free to use it if you'd like.

Submission

At a minimum, submit the following files with the specified classes:

- `UniqueLinkedList.py`
  - `class UniqueRecursiveNode`
  - `class UniqueLinkedList` **unmodified from the starter code**
- SeparateChainingHashTable.py
  - `class SeparateChainingHashTable`
- `LinearProbingHashTable.py`
  - `class LinearProbingHashTable`
- `TestHashTables.py`
  - `class TestHashTableFactory`
  - `class TestSeparateChainingHashTable.py`
  - `class TestLinearProbingHashTable.py`

This assignment is 100% manually graded. Students must submit **individually** by the Tuesday after Exam 2 (Tuesday 4/2/24) at 11:59 pm EST to receive credit.