Due: Thursday, March 17th @ 10:00p ET

This is not a team project. Do not copy someone else's work.

Assignment Overview

Hashtables are a powerful data structure known for their support of insertion, deletion, and lookups in O(1) time. This O(1) performance makes them a go-to structure for programmers in all domains, from software engineering to scientfic computing to machine learning. Other data structures we have discussed, such as linked lists (O(n) lookup, insertion, and deletion), and AVL Trees (O(logn) lookup, insertion, and deletion) lack that O(1) ability across the board.

Believe it or not, you're already familiar with hashtables: Python dictionaries and C++ unordered maps are implemented as hashtables under the hood.

In this project, you'll be refining your hashtable expertise by implementing a hashtable from scratch in Python and applying it to an application problem involving the management of message posts.

The implementation, inspired by a 2012 post to the Python mailing list, will maintain insertion-ordering, supporting iteration over keys, values, and key-value pairs such that items are returned during iteration in the order they were inserted. It will also exploit locality-of-reference cache optimization by replacing a sparse table of (large) entries with a more space-efficient sparse table of (small) indices indexing into a dense table of (large) entries.

Quoting from the post,

```
The current memory layout for dictionaries is unnecessarily inefficient.
It has a sparse table of 24-byte entries containing the hash value, key pointer,
and value pointer.
Instead, the 24-byte entries should be stored in a dense table referenced by a
sparse table of indices.
For example, the dictionary:

d = {'timmy': 'red', 'barry': 'green', 'guido': 'blue'}

is currently stored as:

entries = [['--', '--', '--'],
           [-8522787127447073495, 'barry', 'green'],
           ['--', '--', '--'],
           ['--', '--', '--'],
           ['--', '--', '--'],
           [-9092791511155847987, 'timmy', 'red'],
           ['--', '--', '--'],
           [-6480567542315338377, 'guido', 'blue']]

Instead, the data should be organized as follows:
```

```
indices =  [None, 1, None, None, None, 0, None, 2]
entries =  [[-9092791511155847987, 'timmy', 'red'],
            [-8522787127447073495, 'barry', 'green'],
            [-6480567542315338377, 'guido', 'blue']]
```

In lecture, we discussed hashtable architectures similar to the former, entries-only setup mentioned above.

In this project, we deal with an architecture similar to the latter, indices-and-entries setup mentioned above. A graphical summary of the architecture used in this project is presented below.

Assignment Notes

1. Using any of the following will result in the **loss of all testcase and manual points** associated with the violating function.
    1. a Python dictionary or set,
    2. a Python container/collection other than a basic list or tuple, or
    3. the nonlocal keyword
2. Calling magic or protected methods improperly in this project will result in a **2 point deduction for each violation, up to a maximum of 20 points.**
    1. Calling a magic method like __len__() with the syntax `table.__len__()` instead of the proper syntax `len(table)` is a violation
    2. Calling a protected method like _insert() outside of its implementing class, e.g., in the application problem, is a violation (use __setitem__() instead)
    3. Read below to learn more about magic methods and protected methods
3. Changing function signatures will result in a **2 point deduction for each violation, up to a maximum of 20 points.**
    1. When working with libraries, open-source codebases, or industry codebases, you will rarely be able to change function signatures, so it's good practice to avoid it now!
    2. Changing function signatures will often cause testcases to fail, too.
4. In this project, you will be implementing and using many of Python's *magic methods*.
    1. Magic methods have two underscores on the front and the back, such as __len__().
    2. In this project, these magic methods won't be doing much, they will call the other protected methods that you write!
    3. In other words, these magic methods should not be more than a few lines.
    4. Refer to the syntax table below to see how the magic methods relate to other methods in this project.
5. In this project, you will also be implementing and using *protected methods*.
    1. Protected methods are methods prefaced with a single underscore, such as a function called _insert().
    2. Protected methods are meant only to be called inside other functions within the class.
    3. This is Python's way of implementing the C++ equivalent of private methods; protected methods meant to be treated as private. Protected methods should not be called outside of the implementing class, which means they should not be called in the application problem.

    4. Refer to the syntax table below to see how the protected methods relate to other methods in this project.
6. In this project, you'll be using Python generators to iterate over hashtable keys/values in a space-efficient manner
    1. Unlike an iteration returned in a list using *O(n)* space, a traversal returning a generator will use *O(1)* space by *yielding* each key/value in a sequential, on-demand manner
    2. See this link for a nice introduction to Python generators
7. Make sure you split the work between the magic and hidden methods appropriately.
    1. The majority of the testing will take place in the magic method testcases of __setitem__(), __getitem__() , and __delitem__() to simulate real-world use.
    2. As shown in the syntax table below, however, the magic methods __setitem__(), __getitem__(), and __delitem__() will call the protected methods _insert(), _get(), and _delete(), and these protected methods should handle all of the logic associated with insertion, lookup, and deletion.
    3. Use the small testcases for the _insert(), _get(), and _delete() functions to make sure you are dividing the work properly.
    4. Magic methods should not be more than a few lines.
8. If you inspect _hash_1 and _hash_2, you will see that they depend on the size of the string. For the purposes of this assignment, treat these as taking O(1) (constant) time.
9. A few guarantees:
    1. Capacity will not grow past ~1000
    2. All keys will be of type string

The following syntax table shows how protected methods, magic methods, and calls to magic methods relate to each other.

Protected	**Magic**	**How to Use Magic**
	*_len(self)_*	len(self)
_insert(self, key, value)	*_setitem_*(self, key, value)	self[key] = value
_get(self, key)	*_getitem_*(self, key)	self[key]
_delete(self, key)	*_delitem_*(self, key)	del self[key]
	*_contains_*(self, key)	key in self

Assignment Specifications

* Denotes *Expected Complexity*

class HashNode:

DO NOT MODIFY the following attributes/functions

- **Attributes**
    - **key: str:** The key of the hashnode (this is what is used in hashing)
    - **value: T:** Value being held in the hashnode. Note that this may be any type, such as a str, int, float, dict, or a more complex object
- **__init__(self, key: str, value: T) -> None**

    - Constructs a hashnode
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
    - **key: str:** The key of the hashnode
    - **value: T:** Value being held in the hashnode
    - **Returns:** None
- **__str__(self) -> str** and **__repr__(self) -> str**
    - Represents the hashnode as a string.
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
    - **Returns:** str representation of hashnode
- **__eq__(self, other: HashNode) -> bool**
    - Compares to see if two hashnodes are equal
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
    - **other: HashNode:** The hashnode we are comparing against
    - **Returns:** bool stating whether two hashnodes are equal

class HashTable:

DO NOT MODIFY the following attributes/functions

- **Class Constants**
    - **FREE: int:** Number representing a free space in the hashtable
    - **DELETED: int:** Number representing a deleted space in the hashtable
    - **PRIMES: int:** List of primes from 2-997, used for hashing
- **Attributes**
    - **capacity: int:** Capacity of the hash table
    - **size: int:** Current number of nodes in the hash table
    - **prime_index: int:** Current index of the prime numbers we are using in _hash_2()
    - **indices: List:** a sparse table of integer indices referencing where HashNodes are stored in entries
    - **entries: List:** a dense table of HashNode objects referenced by the integers in indices
    - You may edit the values of attributes in your functions, of course, just don't change the class structure
- **__init__(self, capacity: int = 8) -> None**
    - Construct an empty hashtable, with the capacity as specified in the input
    - **capacity: int**: initial capacity of the hashtable
    - **Returns:** None
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
- **__str__(self) -> str** and **__repr__(self) -> str**
    - Represents the hashtable as a string
    - **Returns:** str
    - *Time Complexity: O(N)*
    - *Space Complexity: O(N)*
- **__eq__(self, other: HashTable) -> bool**
    - Checks if two HashTables are equal

    - **other: HashTable:** the hashtable we are comparing against
    - **Returns**: bool stating whether hashtables they are equal
    - *Time Complexity: O(N)*
    - *Space Complexity: O(N)*
- **__len__(self) -> int**
    - Getter for the size (that, is, the number of elements) in the HashTable
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
    - **Returns:** int that is size of hash table
- **_hash_1(self, key: str) -> int**
    - The first of the two hash functions used to turn a key into a bin number
    - Assume this is O(1) time/space complexity
    - *Time Complexity: O(1)* (assume)
    - *Space Complexity: O(1)* (assume)
    - **key: str:** key we are hashing
    - **Returns:** int that is the bin number
- **_hash_2(self, key: str) -> int**
    - The second of the two hash functions used to turn a key into a bin number. This hash function acts as the tie breaker.
    - *Time Complexity: O(1)* (assume)
    - *Space Complexity: O(1)* (assume)
    - **key: str**: key we are hashing
    - **Returns:** int that is the bin number

IMPLEMENT the following functions

- **_hash(self, key: str, inserting: bool = False) -> int**
    - Given a key string, return an index i in self.indices
    - Should implement probing with double hashing
        - If the key exists in the hashtable, return the index i for which self.entries[self.indices[i]] is the existing HashNode
        - If the key does not exist in the hash table, return the index i of the next available empty position in self.indices
            - Collision resolution should implement double hashing with _hash_1() as the initial hash and _hash_2() as the step size
        - Recall that there are 2 possibilities when hashing for an index
            - When inserting, we want to insert into the next available slot i in self.indices, regardless of whether self.indices[i] is FREE or DELETED
            - When performing a lookup/deletion, i.e., not inserting, we want to
                - Continue until we either find the proper HashNode by checking the key of the HashNode at self.entries[self.indices[i]],
                - Or until we reach a slot self.indices[i] that has never held a value (FREE)
                - This is to preserve the collison resolution methodology
            - The inserting parameter should be used to differentiate between these two cases
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)*
    - **key: str:** The key being used in our hash function

    - **inserting: bool:** Whether we are doing an insertion. Important for the reasons described above.
    - **Returns:** int i at which self.indices[i] is FREE or DELETED for insertion, or refers to a HashNode at self.entries[self.indices[i]] we are looking up
- **_insert(self, key: str, value: T) -> None**
    - Use the key and value parameters to add an entry to the hashtable
        - The new HashNode should be appended to self.entries
        - self.indices[i] should be set such that self.entries[self.indices[i]] retrieves the new HashNode, where i is the integer index returned by _hash()
        - If the key exists, overwrite the existing value
            - This means you will need to call _hash() with inserting=False first to check if a key exists, then call _hash() a second time with inserting=True to find the location of insertion if the key does not exist
        - In the event that inserting causes the table to have a load factor of 0.5 or greater you must grow the table to double the existing capacity. This should use the _grow() method.
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)**
    - **key: str:** The key associated with the value we are storing
    - **value: T:** The associated value we are storing
    - **Returns:** None
- **_get(self, key: str) -> HashNode**
    - Find the HashNode with the given key in the hashtable.
        - Obtain i, the integer index of the key by calling _hash()
        - If self.indices[i] is not FREE or DELETED, then the entry at self.entries[self.indices[i]] will retrieve the desired HashNode
        - Return this desired HashNode if the key matches
        - If the element does not exist, return None
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)*
    - **key: str:** The key we looking up
    - **Returns:** HashNode with the key we looked up
- **_delete(self, key: str) -> None**
    - Removes the HashNode with the given key from the hashtable.
        - Obtain i, the integer index of the key by calling _hash()
        - If self.indices[i] is not FREE or DELETED, then the entry at self.entries[self.indices[i]] will retrieve the desired HashNode
        - Delete this desired HashNode if the key matches
        - If deletion occurs, set self.indices[i] to HashTable.DELETED and set self.entries[self.indices[i]] to None
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)*
    - **key: str:** The key of the Node we are looking to delete
    - **Returns:** None
- **_grow(self) -> None**
    - Double the capacity of the existing hash table.
        - Do **NOT** rehash deleted HashNodes
        - Must update self.prime_index

        - The value of self.prime_index should be the **index** of the largest prime **smaller** than self.capacity in the HashTable.primes tuple
        - See the provided implementation of __init__() for a hint
    - *Time Complexity: O(N)*
    - *Space Complexity: O(N)*
    - **Returns:** None
- **__setitem__(self, key: str, value: T) -> None**
    - Sets the value with an associated key in the HashTable
        - **This should be a short, ~1 line function**
        - The majority of the work should be done in the _insert() method
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)**
    - **key: str:** The key we are hashing
    - **value: T:** The associated value we are storing
    - **Returns:** None
- **__getitem__(self, key: str) -> T**
    - Looks up the value with an associated key in the HashTable
        - If the key does not exist in the table, raise a KeyError
        - **This should be a short, ~3 line function**
        - Tthe majority of the work should be done in the _get() method
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)*
    - **key: str:** The key we are searching
    - **Returns:** The value associated to the provided key
- **__delitem__(self, key: str) -> None**
    - Deletes the value with an associated key in the HashTable
        - If the key does not exist in the table, raise a KeyError
        - **This should be a short, ~3 line function**
        - The majority of the work should be done in the _get() and _delete() methods
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)*
    - **key: str:** The key we are deleting the associated value of
    - **Returns:** None
- **__contains__(self, key: str) -> bool**
    - Determines if a node with the key denoted by the parameter exists in the table
        - **This should be a short, ~3 line function**
        - The majority of the work should be done in the _get() method!
    - *Time Complexity: O(1)**
    - *Space Complexity: O(1)*
    - **key: str:** The key we are searching
    - **Returns:** True if key is in the hashtable, else False
- **update(self, pairs: List[Tuple[str, T]]) -> None**
    - Updates the hashtable using an iterable of key value pairs
        - If the value already exists, update it, otherwise enter it into the table
    - *Time Complexity: O(M)**, where *M* is length of pairs
    - *Space Complexity: O(M)*

    - **pairs: List[Tuple[str, T]]:** list of tuples (key, value) being updated
    - **Returns:** None
- **keys(self, reverse: bool = False) -> Generator[T, None, None]**
    - Generates all keys in the table
        - If reverse == False, generates keys in order of insertion
        - If reverse == True, generates keys in reverse order of insertion
        - Note that order of insertion is naturally preserved by self.entries, though you will need to skip Nones in self.entries
    - Use the yield statement to return a Generator
    - *Time Complexity: O(N)**
    - *Space Complexity: O(N)*
    - **Returns:** Generator of keys
- **values(self, reverse: bool = False) -> Generator[T, None, None]**
    - Generates all values in the table
        - If reverse == False, generates values in order of insertion
        - If reverse == True, generates values in reverse order of insertion
        - Note that order of insertion is naturally preserved by self.entries, though you will need to skip Nones in self.entries
    - Use the yield statement to return a Generator
    - *Time Complexity: O(N)**
    - *Space Complexity: O(N)*
    - **Returns:** Generator of values
- **items(self, reverse: bool = False) -> Generator[Tuple[str, T], None, None]**
    - Generates tuples of all (key, value) pairs in the table
        - If reverse == False, generates pairs in order of insertion
        - If reverse == True, generates pairs in reverse order of insertion
        - Note that order of insertion is naturally preserved by self.entries, though you will need to skip Nones in self.entries
    - Use the yield statement to return a Generator
    - *Time Complexity: O(N)**
    - *Space Complexity: O(N)*
    - **Returns:** Generator of tuple (key, value) pairs
- **clear(self) -> None**
    - Clears table of HashNodes completely
        - In essence, a reset of the table, emptying self.indices and self.entries
        - Should not modify capacity
    - *Time Complexity: O(N)*
    - *Space Complexity: O(N)*
    - **Returns:** None

Application Problem: Discord Destroyer

Sometimes people get fed up with their favorite social media platforms and wish they could alter it to their will. With this problem we will be fighting the behemoth platform that is Discord. In order to make something

better eventually, we need to at least have the same basic functionality.

This comes in the form of starting with the basic functionality of a **channel**. A channel contains the fundamentals for functionality for a platform like this. Users are able to send a message, delete a message, get the most recent posts in order, and search through a user's messages.

To implement this functionality as efficiently as possible you will have to use two separate hashtables, a 1D HashTable to store the posts that are in a channel by 1D and 2D hashtable to store users' posts for quick iteration.

In order to circumvent the issue of users' future posts overwriting the past posts, each post must generate a unique post id and the key for storing will be a string in the form "user,post id".

*Note: No username will contain the character ','

Example 1

Suppose user "algorithmsfan2002" makes a post "I love Data Structures," and suppose this post creates the id "331" when generate_post_id is called with this user and message. Then the key for this post within the posts_by_id hashtable will be "algorithmsfan2002,331" and self.posts_by_id["algorithmsfan2002,331"] will hold "I love Data Structures".

This key "algorithmsfan2002,331" will also be the value stored within the users dictionary for fast lookup by user when we wish to get posts by user. In this example, the key "algorithmsfan2002,331" will be stored at self.ids_by_user["algorithmsfan2002"]["331"].

Example 2

- If these four posts are posted (from top to bottom oldest to newest)
    - "Hash(Table) Slinging Slasher" : "I love hashing algorithms!"
    - "Patrick A-Star" : "Remember your space complexity!"
    - "Hash(Table) Slinging Slasher" : "O(1) Lookups rule!"
    - "Patrick A-Star" : "NuDiscord rules!"

This ordering of posts will create the posts_by_id table that looks like this diagram:

This will also create an ids_by_users table that will look like this:

class DiscordDestroyer

DO NOT MODIFY the following attributes/functions

- **Attributes**
    - **posts_by_ids:** HashTable that will hold all the information about the posts in the channel
        - The Keys will be in the form of a string "user,post_id"
        - The Values will be a string containing the post information
    - **ids_by_users:** HashTable of HashTables that will store users' posts.

        - The keys for the outer HashTable will be in the form of "user"
        - The keys for the inner HashTable will be in the form of "post_id"
        - The value stored in the inner HashTable will be the key for accessing the post in the channel HashTable, i.e., "user,post_id"
    - **post_id_seed:** integer used for generating post_id
- **__init__(self) -> None**
    - Initializes the channel and creates the post HashTable
    - self.ids_by_users - Hashtable mapping user strings to list of Hashtable of posts from that user
    - self.posts_by_ids - Hashtable mapping id strings to post strings
    - self.post_id_seed - integer used for generating post_id
- **generate_post_id(self, user: str, message: str) -> int:**
    - Generates a unique post_id from the seed by hashing the seed. message
    - increments the seed by 1 after this hashing

You must implement the following functions

- **post(self, user: str, message: str) -> str**
    - Creates a post in the posts_by_id hashtable, updates the ids_by_user hashtable, and returns the id it is assigned
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
    - **user: str:** string representing the user who posted the post
    - **message: str:** string representing the message that was posted
    - **return:** post_id that was generated for the post
- **delete_post(self, post_id: str) -> bool**
    - Removes the post with provided post_id from posts_by_id and ids_by_user hashtables, and returns a bool indicating whether the post was actually deleted
    - *Time Complexity: O(1)*
    - *Space Complexity: O(1)*
    - **user: str:** string representing the user who posted the post
    - **post_id: int:** int representing the post_id of the post
    - **return:** bool representing if the post with the given post_id was deleted or not
- **get_most_recent_posts(self, v: int) -> Generator[Tuple[str, int, str], None, None]**
    - Returns a generator that will yield the V most recent posts
    - *Time Complexity: O(V)*
    - *Space Complexity: O(1)*
    - **v: int:** how many posts to yield
    - **return:** generator that will yield v posts
- **get_posts_by_user(self, user: str) -> Generator[Tuple[str, int, str], None, None]**
    - Returns a generator of all posts the user made
        - Be careful with time complexity on this one!
        - There's a reason we have the ids_by_user hashtable...
    - *Space Complexity: O(1)*
    - *Time Complexity: O(P)* where *P* is the number of posts the user has made
    - **user: str:** user to get the posts from
    - **return:** generator that will yield all the users posts

Submission

Deliverables

Be sure to upload the following deliverables in a .zip folder to Mimir by Thursday, March 17th @ 10:00p ET.

```
Project06.zip
    |— Project06/
        |— README.xml      (for project feedback)
        |— __init__.py     (for proper Mimir testcase loading)
        |— hashtable.py    (contains your solution source code)
```

Grading

- Tests (70)
  - Coding Standard: __/3
  - README.xml Validity Check: __/3
  - HashTable: __/50
    - _hash: __/10
    - _insert: __/1
    - _get: __/1
    - _delete: __/1
    - __setitem__: __/4
    - __getitem__: __/4
    - __delitem__: __/4
    - __contains__: __/4
    - update: __/3
    - keys/values/items: __/6
    - clear: __/2
    - comprehensive: __/10
  - DiscordDestroyer: __/14
    - application_post: __/3
    - application_delete: __/3
    - application_recent_posts: __/4
    - application_by_user: __/4
- Manual (30)
  - Time and space complexity points are **all-or-nothing** for each function. If you fail to meet time **or** space complexity in a given function, you do not receive manual points for that function.
  - Do not change the function signatures, (several point loss)
  - Loss of 1 point per missing docstring (max 3 point loss)
  - Loss of 2 points per improper method call (max 20 point loss)
    - Improper magic method call (e.g., using `table.__len__()` instead of `len(table)`)
    - Calling a protected method outside of its implementing class (e.g., calling `table._insert()` in application problem)
  - Loss of 2 points per changed function signature (max 20 point loss)
  - HashTable Time/Space: __/22

        - _hash: __/3
        - __setitem__: __/3
        - __getitem__: __/3
        - __delitem__: __/3
        - __contains__: __/2
        - _grow: __/2
        - update: __/2
        - keys/values/items: __/3
        - clear: __/1
    - DiscordDestroyer Time/Space: __/8
        - post: __/2
        - delete_post: __/2
        - get_most_recent_posts: __/2
        - get_posts_by_user: __/2

Appendix

Authors

Project developed by Aaron Jonckheere and Andrew McDonald.

Adapted from the work of Alex Woodring, Joseph Pallipadan, Zach Matson, Brandon Field, Yash Vesikar, Ian Barber, and Max Huang.