DSAA CA1

Name: Law Wei Tin

Class: DAAA/FT/2A/02

Admin number: 2415761

Title bar and Selection Menu

```
(base) C:\Users\Law Wei Tin\OneDrive\Desktop\DSAA>python main.py

************************************************************************
* ST1507 DSAA: Welcome to:                                             *
*                                                                      *
*  ~ Haikumator - Haiku Generator Application~                         *
*----------------------------------------------------------------------*
*                                                                      *
*  - Done by: Law Wei Tin (2415761)                                    *
*  - Class DAAA/FT/2A/02                                               *
************************************************************************


Press Enter to continue...

Please select your choice: (1, 2, 3, 4, 5, 6, 7)
        1. Synonymize Haiku
        2. Zen-ize Haiku
        3. Antonymize Haiku
        4. Batch Processing
        5. Haiku Composer
        6. History Browser / Manage
        7. Exit

Enter Choice: 7

Bye, thanks for using ST1507 DSAA: Haikumator
```

Option 1: Synonymize Haiku

```
Please select your choice: (1, 2, 3, 4, 5, 6, 7)
       1. Synonymize Haiku
       2. Zen-ize Haiku
       3. Antonymize Haiku
       4. Batch Processing
       5. Haiku Composer
       6. History Browser / Manage
       7. Exit

Enter Choice: 1

-- Synonymize Haiku --

Select the Haiku you want to process
Enter path to haiku file: haiku_001.txt

Select a synonym thesaurus.
Enter path to synonym thesaurus file: haiku_001_synonyms.txt

Press Enter to continue...


The Haiku before preprocessing:
------------------------------
Last gust of autumn.
Red, yellow, brown unite.
Carpet under the trees.

The Synonymized Haiku after preprocessing:
------------------------------
Closing breath of december.
Reddish, tawny, auburn fuse.
Mat underneath the plants.

Press Enter to continue...
```

```
Do you want to save the text to a file? (y/n): y
Please enter new filename: haiku_001_synonymized.txt
The text has been saved in "haiku_001_synonymized.txt"

Press Enter to continue...

Do you want to give this another try? y/n: y

The Haiku before preprocessing:
------------------------------
Last gust of autumn.
Red, yellow, brown unite.
Carpet under the trees.

The Synonymized Haiku after preprocessing:
------------------------------
Finishing breath of equinox.
Reddish, ashen, swarthy merge.
Spread down the vegetation.

Press Enter to continue...

Do you want to save the text to a file? (y/n): n
File not saved.

Press Enter to continue...

Do you want to give this another try? y/n: n

Press Enter to continue...
```

This synonymize works as intended. Additional error handling included, such as:

- If the user attempts to enter a non-existent haiku file, the program will prompt them to either re-enter the file or return to the options menu. This allows users to exit the program if they realize they don't have the required haiku file.

- If the user tries to save a haiku to an existing file, the program will ask if they want to overwrite the file's contents.

Option 2: Zen-ize Haiku

```
Please select your choice: (1, 2, 3, 4, 5, 6, 7)
        1. Synonymize Haiku
        2. Zen-ize Haiku
        3. Antonymize Haiku
        4. Batch Processing
        5. Haiku Composer
        6. History Browser / Manage
        7. Exit

Enter Choice: 2

-- Zen-ize Haiku --

Select the Haiku you want to process
Please enter input file: haiku_002.txt

Select a synonym thesaurus.
Please enter input file: haiku_002_synonyms.txt

Press Enter to continue...


The Haiku before preprocessing:
------------------------------
Brilliance in the ocean.
Exclusively glitters for me.
My ship diverts, no more.

The Zen-ized Haiku after preprocessing:
------------------------------
Light in the deep.
Only gleams for me.
My ship turns, no more.

Press Enter to continue...

Do you want to save the text to a file? (y/n): y
Please enter new filename: haiku_002_zenized.txt
The text has been saved in "haiku_002_zenized.txt"

Press Enter to continue...
```

```
Do you want to give this another try? y/n: y

The Haiku before preprocessing:
------------------------------
Brilliance in the ocean.
Exclusively glitters for me.
My ship diverts, no more.

The Zen-ized Haiku after preprocessing:
------------------------------
Light in the deep.
Just gleams for me.
My ship turns, no more.

Press Enter to continue...

Do you want to save the text to a file? (y/n): y
Please enter new filename: haiku_002_zenized.txt
File “haiku_002_zenized.txt” exists. Override? (y/n): y
The text has been saved in "haiku_002_zenized.txt"

Press Enter to continue...

Do you want to give this another try? y/n: n

Press Enter to continue...
```

Additional Error handling includes the ones like our Option 1.

Option 3: Antonymize Haiku

```
Please select your choice: (1, 2, 3, 4, 5, 6, 7)
        1. Synonymize Haiku
        2. Zen-ize Haiku
        3. Antonymize Haiku
        4. Batch Processing
        5. Haiku Composer
        6. History Browser / Manage
        7. Exit

Enter Choice: 3

-- Antonymize Haiku --

Select the Haiku you want to process
Please enter input file: haiku_001_b.txt

Select a synonym thesaurus.
Please enter input file: haiku_001_synonyms.txt

Select an antonym thesaurus.
Please enter input file: haiku_001_antonyms.txt

Press Enter to continue...


The Haiku before processing:
------------------------------
Final breath of autumn.
Red, yellow, auburn fuse.
Carpet under the foliage.

The Antonymized Haiku after processing:
------------------------------
Newest stillness of march.
Khaki, blue, light scatter.
Ceiling overhead the fauna.

Press Enter to continue...
```

```
Do you want to save the text to a file? (y/n): y
Please enter new filename: haiku_001_antonymized.txt
The text has been saved in "haiku_001_antonymized.txt"

Press Enter to continue...

Do you want to give this another try? y/n: y

The Haiku before processing:
------------------------------
Final breath of autumn.
Red, yellow, auburn fuse.
Carpet under the foliage.

The Antonymized Haiku after processing:
------------------------------
Young serenity of march.
Emerald, blue, silver disperse.
Ceiling overhead the grass.

Press Enter to continue...

Do you want to save the text to a file? (y/n): y
Please enter new filename: haiku_001_antonymized.txt
File “haiku_001_antonymized.txt” exists. Override? (y/n): y
The text has been saved in "haiku_001_antonymized.txt"

Press Enter to continue...

Do you want to give this another try? y/n: n

Press Enter to continue...
```

Additional Error handling includes the ones like our Option 1.

Option 4: Batch Preprocessing

```
Please select your choice: (1, 2, 3, 4, 5, 6, 7)
        1. Synonymize Haiku
        2. Zen-ize Haiku
        3. Antonymize Haiku
        4. Batch Processing
        5. Haiku Composer
        6. History Browser / Manage
        7. Exit

Enter Choice: 4

-- Batch Processing --

Select the Haiku you want to process
Please enter input file: haiku_003.txt

Select a synonym thesaurus.
Please enter input file: haiku_003_synonyms.txt

Select an existing folder as to store the batch processed haikus
Please enter the folder name: HAIKU_003_SYNONYMIZED

Press Enter to continue...

Batch Processing started!
............
Batch Preprocessing completed with 12 permutations
All files saved to HAIKU_003_SYNONYMIZED


Press Enter to continue...
```

```
Enter Choice: 4

-- Batch Processing --

Select the Haiku you want to process
Please enter input file: haiku_003.txt

Select a synonym thesaurus.
Please enter input file: haiku_003_synonyms.txt

Select an existing folder as to store the batch processed haikus
Please enter the folder name: THIS_FOLDER_DOESNT_EXIST

Folder 'THIS_FOLDER_DOESNT_EXIST' doesn't exist.
+-----+---------------------------------------+
| Key | Action                                |
+-----+---------------------------------------+
|     | Retry entering folder name            |
| c   | Create folder and continue            |
| o   | Return to options menu                |
+-----+---------------------------------------+
Select an option [Enter/c/o]: c

Press Enter to continue...

Batch Processing started!
............
Batch Preprocessing completed with 12 permutations
All files saved to THIS_FOLDER_DOESNT_EXIST


Press Enter to continue...
```

Additional Error handling includes:

- When keying in a folder to store our batch pre-processed haikus, if we key in a non-existent folder, the program will prompt whether user wants to create this folder and continue, re-enter folder name or return to main menu.
- Furthermore, if user doesn't enter a folder but instead enters a file, the program will tell the user and allows user to re-enter.
- If user keys in an existing folder, program will ask to override the folder content.

Option 5: Haiku Composer

```
Please select your choice: (1, 2, 3, 4, 5, 6, 7)
        1. Synonymize Haiku
        2. Zen-ize Haiku
        3. Antonymize Haiku
        4. Batch Processing
        5. Haiku Composer
        6. History Browser / Manage
        7. Exit

Enter Choice: 5

-- Template-Driven Haiku Generator --

Enter word-bank file: word_bank.txt

Press Enter to continue...


Generated Haiku:
------------------------------
Star wanders pale dawn.
Silently drifts stone dark light.
Warm moon sighs warmly.


Press Enter to continue...

Do you want to save this haiku to a file? (y/n): y
Enter filename: haiku_composed_001.txt
Saved haiku to "haiku_composed_001.txt"

Press Enter to continue...
```

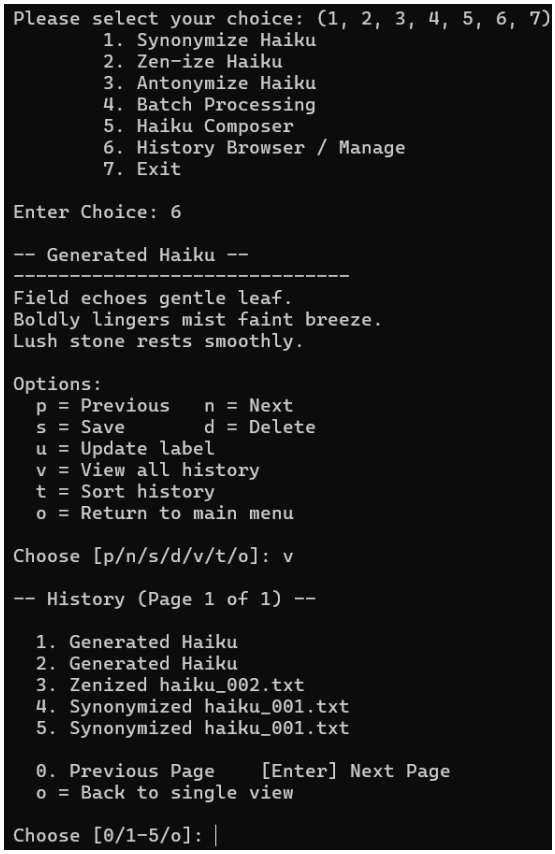
(description below)

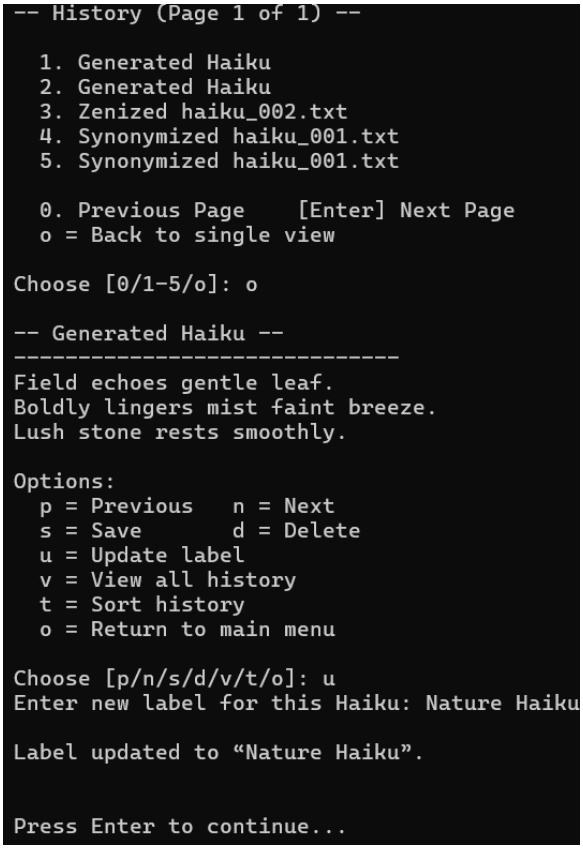
When program asks to enter word-bank file, enter word_bank.txt, provided in the DSAA folder. Additional words can be added to the file but must follow the format of the file.

This haiku composer will generate a new haiku using the word bank given, with consistent context in the haiku.

Additional Error handling includes the ones like our Option 1.

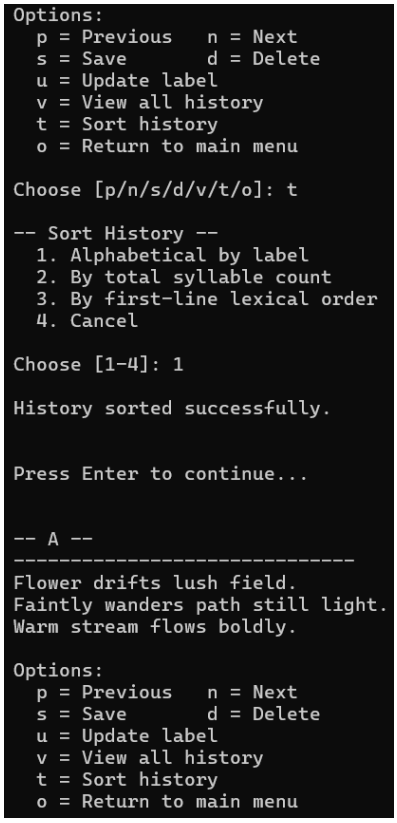
Option 6: History Browser/Manage

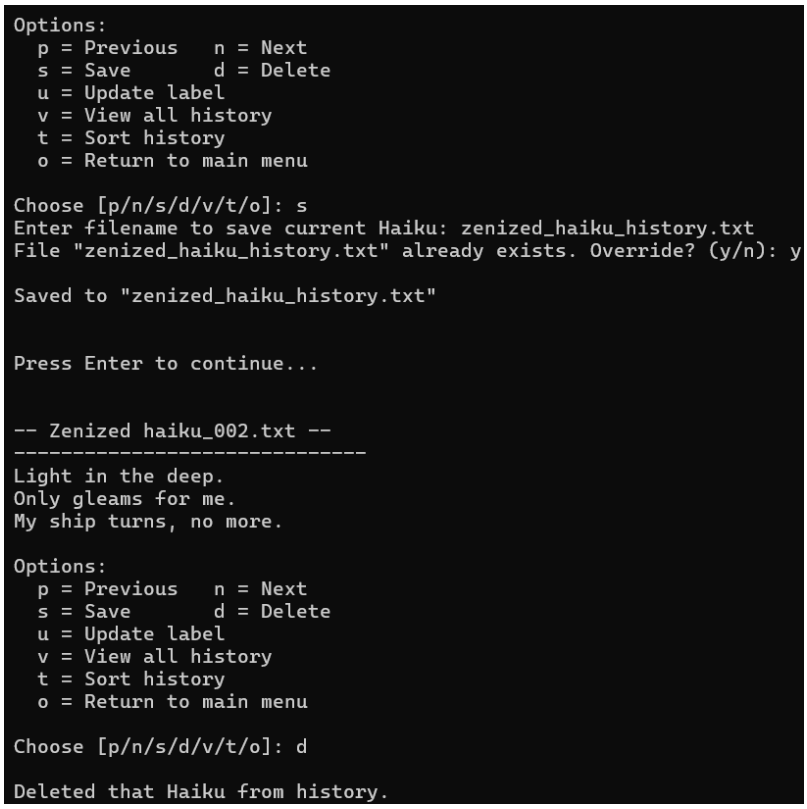




In option 6, we can see and manage the history of all our generated haikus in this session. Features are described below:

- Let’s say we synonymized or generated a haiku earlier. We can view it in our history, and the contents of the haiku can be seen as well. The history will show the most recently generated haiku, and user can scroll through the history to view the oldest generated haiku.
- If we want to label the haiku (for easy identification purposes), we can press “u”, to update the label.
- If we regret not saving our previously generated haikus, we can press “s” to save the current haiku that we are at.
- Press “d” to delete the current haiku we are at, if we want to clean history.
- We can press “v” to view the history in pages and easily zoom in (by selecting the haiku number) on which haiku that we want to view.





Side note: I updated the label of another haiku to “A” to demonstrate sorting.

- Press “t” to sort the history. For example, if we want to view the haiku labelled “A”, we can conveniently sort the haikus by their label. This allows user to not have to scroll endlessly if history size increases.
- Press “o” to go back to the options menu.

c) DESCRIPTION OF OBJECT-ORIENTED PROGRAMMING APPROACH

Encapsulation:

- Menu handles UI flow only. Data-structure classes (Stack, Queue, LinkedList, DoublyLinkedList, Trie) expose just the operations they need (push/pop, enqueue/dequeue, etc.).
- Keeps responsibilities single-purpose and lets you tweak internals (e.g., change the linked-list node layout) without touching the rest of the program.

Inheritance:

- HaikuTransformer parent class, children are: SynonymizeTransformer, ZenizeTransformer, AntonymizeTransformer.
- Eliminates duplicate code and centralises any future fixes (e.g., speeding up keyword lookup).

Polymorphism:

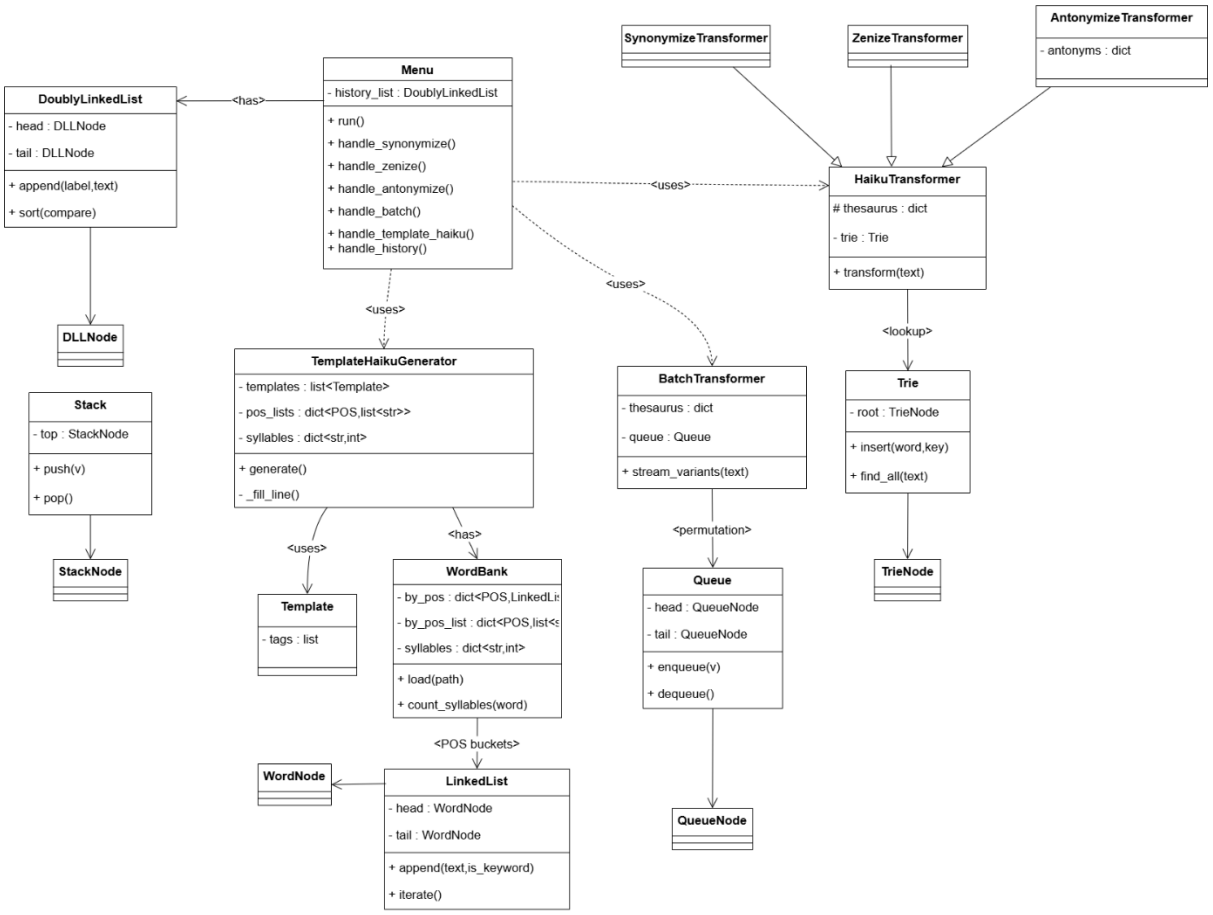
- All transformers expose the same transform(text) interface. The Menu logic can treat any subclass as a **HaikuTransformer**, swapping behaviour simply by constructing a different child class.

Hiding internal functions and variables:

- Internal functions and variables such as _fill_line, _merge_sort, _history_list are prefixed with "_" to signal privacy. This protects program from accidental misuse.

Method overriding:

- Each transformer overrides transform; merge-sort helpers override comparison by accepting a compare(a,b) callback.



Note: Node classes are behind-the-scenes details, so showing all their parts would make the diagram messy without helping us understand the overall design better.

d) DESCRIPTION OF DATA STRUCTURES AND ALGORITHMS

Trie used inside each transformer:

- The transformers must find **every keyword** in a haiku, even when words overlap. A Trie lets us scan the text once: each character follow-down is **O(1)**, so matching all words in a string of length L costs **O(L)**, outperforming repeated dictionary lookups or regexes for large vocabularies.

Queue used in BatchTransformer:

- Batch replacement explores every synonym permutation in breadth-first order. A FIFO queue naturally models this: **enqueue/dequeue are O(1)**, and breadth-first traversal avoids deep-recursion limits while guaranteeing deterministic variant numbering.

Singly Linked List inside WordBank:

- The word lists are built once (append-only) and then iterated many times when filling templates. The singly linked list gives **O(1)** append, minimal memory overhead, and sequential scans in **O(n)**, and since random access isn't required here, it is alright.

Doubly Linked List for History Browser:

- Users move **forward and backward** between haikus (prev/next) and may delete the "current" node. Two-way links give constant-time navigation and deletion (**O(1)**) without array shifting—ideal for an editable, bidirectional timeline.

Stack used for backtracking in Haiku Generator:

- LIFO order lets us *push* a choice and quickly *pop* it when backtracking while generating or transforming haikus. Both push/pop run in **O(1)** time and need no reallocation, keeping exploratory algorithms lightweight.

Merge sort on Doubly Linked List for History Browser:

- Splits the list, sorts each half, and merges. All pointer operations, no extra arrays.
- **O(n log n)** time, **O(log n)** stack space.
- Works on linked nodes without costly re-indexing; stable and predictable for the History browser's "sort by ..." option.

Backtracking for Haiku Generation:

- Recursively tries word choices to hit an exact syllable target, drops branches early when the running total overshoots.
- Needed to satisfy 5-7-5 + "no-repeat" rules. Brute force would explode; pruning keeps it fast enough.

Linear Time Trie scan:

- Walks the Trie once per character to spot every keyword/ synonym overlap in a single pass. **O(L)** where L = length of text.
- Faster than repeating in / regex searches for each word, especially when the vocabulary grows.

Queue-Based Synonym Expansion:

- Uses the queue to systematically replace one keyword at a time, generating every synonym permutation. Each enqueue/dequeue is **O(1)**.
- Guarantees deterministic ordering (v1.txt, v2.txt, ...) and avoids deep recursion.

Syllable counter:

- Regex finds vowel groups to estimate syllables. **O(k)** per word (*k* = word length).

Summary of Data Structures used

Data Structure	Custom/Built-in
DoublyLinkedList	Custom
LinkedList	Custom
Stack	Custom
Queue	Custom
Trie	Custom
Nodes (StackNode, etc.)	Custom
list	Built-in
dict	Built-in
set	Built-in
re	Built-in library

e) SUMMARY OF CHALLENGES AND LEARNING ACHIEVEMENTS

- I had trouble implementing my extra features, like my Haiku Generator. It was tough learning how to implement the backtracking algorithms to generate an optimal haiku. Also, making each line context-aware required several failed prototypes the current heuristic + fallback pipeline worked.
- The Haiku History was alright, but implementing the sorting algo was tough, and looked up online to learn more about it.
- Implementing the classes from scratch was tiring as well.

I learnt how to efficiently implement Data Structures and Algorithms in my project, together with understanding their Big-O better. Learning merge sort was an insightful task for me, and I am confident it will help me in future projects. I also learnt how to handle errors better, both in code and when the application runs.

APPENDIX

main.py:

```
from menu import Menu


if __name__ == "__main__":
    menu = Menu()
    menu.run()
```

menu.py:

```
"""
Name: Law Wei Tin
Class: DAAA/FT/2A/02
Admin No.: P2415761
"""

import os
import re

from haiku_utils import (
    SynonymizeTransformer,
    AntonymizeTransformer,
    ZenizeTransformer,
    BatchTransformer
)

from haiku_generator import Template, WordBank, TemplateHaikuGenerator
from doubly_linked_list import DoublyLinkedList
```

```
from general_utils import pause, prompt_retry_or_menu, load_thesaurus


class Menu:
    def __init__(self):
        # History of generated Haikus
        self._history_list = DoublyLinkedList()

    @staticmethod
    def count_syllables(word: str) -> int:
        w = word.lower().strip()
        if len(w) > 2 and w.endswith("e"):
            w = w[:-1]
        groups = re.findall(r"[aeiouy]+", w)
        return max(1, len(groups))

    def display_header(self):
        width = 72
        inner_width = width - 2
        border = "*" * width

        lines = [
            " ST1507 DSAA: Welcome to:",
            "",
            "  ~ Haikumator - Haiku Generator Application~",
            "-" * inner_width,
            "",
            "  - Done by: Law Wei Tin (2415761)",
```

```
        "  - Class DAAA/FT/2A/02"
    ]

    print("\n" + border)
    for txt in lines:
        print("*" + txt.ljust(inner_width) + "*")
    print(border + "\n")

def display_options(self):
    """Prints just the numbered menu options."""
    print("Please select your choice: (1, 2, 3, 4, 5, 6, 7)")
    print("\t1. Synonymize Haiku")
    print("\t2. Zen-ize Haiku")
    print("\t3. Antonymize Haiku")
    print("\t4. Batch Processing")
    print("\t5. Haiku Composer")
    print("\t6. History Browser / Manage")
    print("\t7. Exit\n")

def run(self):
    # Show header once
    self.display_header()
    pause()

    while True:
        self.display_options()
        choice = input("Enter Choice: ").strip()
```

```
        if choice == '1':
            self.handle_synonymize()
        elif choice == '2':
            self.handle_zenize()
        elif choice == '3':
            self.handle_antonymize()
        elif choice == '4':
            self.handle_batch()
        elif choice == '5':
            self.handle_template_haiku()
        elif choice == '6':
            self.handle_history()
        elif choice == '7':
            print("\nBye, thanks for using ST1507 DSAA: Haikumator")
            break
        else:
            print("Invalid choice, please try again.")
            pause()

    def handle_synonymize(self):
        print("\n-- Synonymize Haiku --\n")
        # Load Haiku file
        while True:
            path = input("Select the Haiku you want to process\nEnter path to haiku file:
").strip()
            try:
                with open(path, 'r', encoding='utf-8') as f:
                    original = f.read().strip()
```

```
            break
        except Exception as e:
            print(f"Failed to read haiku: {e}")
        if not prompt_retry_or_menu():
            return

    # Load thesaurus
    while True:
        tpath = input("\nSelect a synonym thesaurus.\nEnter path to synonym thesaurus file: ").strip()
        try:
            thes = load_thesaurus(tpath)
            break
        except Exception as e:
            print(f"Failed to load thesaurus: {e}")
        if not prompt_retry_or_menu():
            return

    transformer = SynonymizeTransformer(thes)
    pause()

    while True:
        result = transformer.transform(original)

        print("\nThe Haiku before preprocessing:")
        print("-" * 30)
        print(original, "\n")
        print("The Synonymized Haiku after preprocessing:")
```

```
print("-" * 30)
print(result)
pause()

if input("Do you want to save the text to a file? (y/n): ").strip().lower() == 'y':
    while True:
        fname = input("Please enter new filename: ").strip()
        if os.path.exists(fname):
            if input(f"File “{fname}” exists. Override? (y/n): ").strip().lower() != 'y':
                print("Okay, choose a different filename.")
                continue
        try:
            with open(fname, 'w', encoding='utf-8') as out:
                out.write(result)
            print(f"The text has been saved in \"{fname}\"")
        except Exception as e:
            print(f"Failed to save: {e}")
        pause()
        break
else:
    print("File not saved.")
    pause()

# Append to history
label = f"Synonymized {os.path.basename(path)}"
self._history_list.append(label=label, text=result)

if input("Do you want to give this another try? y/n: ").strip().lower() != 'y':
```

```
            pause()
            break


    def handle_zenize(self):
        print("\n-- Zen-ize Haiku --\n")
        # Load Haiku file
        while True:
            path = input("Select the Haiku you want to process\nPlease enter input file:
").strip()
            try:
                with open(path, 'r', encoding='utf-8') as f:
                    original = f.read().strip()
                break
            except Exception as e:
                print(f"Failed to read haiku: {e}")
            if not prompt_retry_or_menu():
                return

        # Load thesaurus
        while True:
            tpath = input("\nSelect a synonym thesaurus.\nPlease enter input file: ").strip()
            try:
                thes = load_thesaurus(tpath)
                break
            except Exception as e:
                print(f"Failed to load thesaurus: {e}")
            if not prompt_retry_or_menu():
                return
```

```
transformer = ZenizeTransformer(thes)
pause()

while True:
  result = transformer.transform(original)

  print("\nThe Haiku before preprocessing:")
  print("-" * 30)
  print(original, "\n")
  print("The Zen-ized Haiku after preprocessing:")
  print("-" * 30)
  print(result)
  pause()

  if input("Do you want to save the text to a file? (y/n): ").strip().lower() == 'y':
    while True:
      fname = input("Please enter new filename: ").strip()
      if os.path.exists(fname):
        if input(f"File “{fname}” exists. Override? (y/n): ").strip().lower() != 'y':
          print("Okay, choose a different filename.")
          continue
      try:
        with open(fname, 'w', encoding='utf-8') as out:
          out.write(result)
        print(f"The text has been saved in \"{fname}\"")
      except Exception as e:
        print(f"Failed to save: {e}")
```

```
            pause()
            break
        else:
            print("File not saved.")
            pause()

        # Append to history
        label = f"Zenized {os.path.basename(path)}"
        self._history_list.append(label=label, text=result)

        if input("Do you want to give this another try? y/n: ").strip().lower() != 'y':
            pause()
            break

    def handle_antonymize(self):
        print("\n-- Antonymize Haiku --\n")
        # Load Haiku file
        while True:
            path = input("Select the Haiku you want to process\nPlease enter input file:
").strip()
            try:
                with open(path, 'r', encoding='utf-8') as f:
                    original = f.read().strip()
                break
            except Exception as e:
                print(f"Failed to read haiku: {e}")
            if not prompt_retry_or_menu():
                return
```

```
# Load synonym thesaurus
while True:
    s_path = input("\nSelect a synonym thesaurus.\nPlease enter input file: ").strip()
    try:
        syn_thes = load_thesaurus(s_path)
        break
    except Exception as e:
        print(f"Failed to load synonym thesaurus: {e}")
    if not prompt_retry_or_menu():
        return

# Load antonym thesaurus
while True:
    a_path = input("\nSelect an antonym thesaurus.\nPlease enter input file: ").strip()
    try:
        ant_thes = load_thesaurus(a_path)
        break
    except Exception as e:
        print(f"Failed to load antonym thesaurus: {e}")
    if not prompt_retry_or_menu():
        return

transformer = AntonymizeTransformer(syn_thes, ant_thes)
pause()

while True:
    result = transformer.transform(original)
```

```
print("\nThe Haiku before processing:")
print("-" * 30)
print(original, "\n")
print("The Antonymized Haiku after processing:")
print("-" * 30)
print(result)
pause()

if input("Do you want to save the text to a file? (y/n): ").strip().lower() == 'y':
    while True:
        fname = input("Please enter new filename: ").strip()
        if os.path.exists(fname):
            if input(f"File “{fname}” exists. Override? (y/n): ").strip().lower() != 'y':
                print("Okay, choose a different filename.")
                continue
        try:
            with open(fname, 'w', encoding='utf-8') as out:
                out.write(result)
            print(f"The text has been saved in \"{fname}\"")
        except Exception as e:
            print(f"Failed to save: {e}")
        pause()
        break
else:
    print("File not saved.")
    pause()
```

```
        # Append to history
        label = f"Antonymized {os.path.basename(path)}"
        self._history_list.append(label=label, text=result)

        if input("Do you want to give this another try? y/n: ").strip().lower() != 'y':
            pause()
            break

def handle_batch(self):
    print("\n-- Batch Processing --\n")
    # Load Haiku file
    while True:
        print("Select the Haiku you want to process")
        path = input("Please enter input file: ").strip()
        try:
            with open(path, 'r', encoding='utf-8') as f:
                original = f.read().strip()
            break
        except Exception as e:
            print(f"\nFailed to read haiku: {e}")
            if not prompt_retry_or_menu():
                return

    # Load thesaurus
    while True:
        print("\nSelect a synonym thesaurus.")
        tpath = input("Please enter input file: ").strip()
        try:
```

```
            thes = load_thesaurus(tpath)
            break
        except Exception as e:
            print(f"\nFailed to load thesaurus: {e}")
            if not prompt_retry_or_menu():
                return

    # Prompt for output folder
    while True:
        print("\nSelect an existing folder as to store the batch processed haikus")
        outdir = input("Please enter the folder name: ").strip()

        if os.path.isdir(outdir):
            if os.listdir(outdir):
                if input(f"Folder “{outdir}” already exists and is not empty. Clear it? (y/n):
").strip().lower() == 'y':
                    for fname in os.listdir(outdir):
                        path = os.path.join(outdir, fname)
                        try:
                            # only files; leave subfolders alone
                            if os.path.isfile(path):
                                os.remove(path)
                        except Exception as e:
                            print(f"Failed to remove {path}: {e}")
                    print(f"Cleared existing files in “{outdir}”.")
                else:
                    print("Okay, choose a different folder.")
                    continue
```

```
    break

if os.path.exists(outdir):
    print(f"Path '{outdir}' exists but is not a directory.\n")
    continue

print(f"\nFolder '{outdir}' doesn't exist.")
print('+-----+--------------------------------------+')
print('| Key | Action                               |')
print('+-----+--------------------------------------+')
print('|     | Retry entering folder name           |')
print('| c   | Create folder and continue           |')
print('| o   | Return to options menu               |')
print('+-----+--------------------------------------+')
folderoption = input("Select an option [Enter/c/o]: ").strip().lower()

if folderoption == 'c':
    try:
        os.makedirs(outdir, exist_ok=True)
        break
    except Exception as e:
        print(f"\nFailed to create folder: {e}")
        pause()
        continue
elif folderoption == 'o':
    return
else:
    continue
```

```
    transformer = BatchTransformer(thes)
    pause()
    print("Batch Processing started!")

    last_idx = 0
    for idx, variant in transformer.stream_variants(original):
        print('.', end='', flush=True)
        fname = os.path.join(outdir, f"v{idx}.txt")
        with open(fname, 'w', encoding='utf-8') as f:
            f.write(variant)
        last_idx = idx

    print(f"\nBatch Preprocessing completed with {last_idx} permutations")
    print(f"All files saved to {outdir}\n")
    pause()

def handle_history(self):
    """
    Single-Haiku view + navigation + manage history,
    with a 't' option to sort history in-place.
    """
    if not self._history_list._head:
        print("\nNo Haikus in history yet.\n")
        pause()
        return

    current = self._history_list._tail
```

```
while True:
    print(f"\n-- {current.label} --")
    print("-" * 30)
    print(current.text + "\n")

    print("Options:")
    print("  p = Previous   n = Next")
    print("  s = Save      d = Delete")
    print("  u = Update label")
    print("  v = View all history")
    print("  t = Sort history")
    print("  o = Return to main menu\n")

    choice = input("Choose [p/n/s/d/v/t/o]: ").strip().lower()

    if choice == 'o':
        return

    if choice == 'p':
        if current._prev:
            current = current._prev
        else:
            print("\nAlready at first (oldest) Haiku.\n")
            pause()
        continue

    if choice == 'n':
```

```
    if current._next:
        current = current._next
    else:
        print("\nAlready at most recent Haiku.\n")
        pause()
    continue

if choice == 's':
    while True:
        fname = input("Enter filename to save current Haiku: ").strip()
        # Check for existing file
        if os.path.exists(fname):
            over = input(f"File \"{fname}\" already exists. Override? (y/n): ").strip().lower()
            if over == 'y':
                break        # proceed to save
            elif over == 'n':
                print("Okay, choose a different filename.")
                continue     # loop back and ask again
            else:
                print("Please enter 'y' or 'n'.")
                continue     # invalid answer, re-ask override
        else:
            break            # file doesn't exist, proceed to save

    try:
        with open(fname, 'w', encoding='utf-8') as f:
            f.write(current.text)
        print(f"\nSaved to \"{fname}\"\n")
```

```
    except Exception as e:
        print(f"\nFailed to save: {e}\n")
    pause()
    continue

if choice == 'd':
    to_delete = current
    if to_delete._prev:
        current = to_delete._prev
    elif to_delete._next:
        current = to_delete._next
    else:
        # That was the only node
        self._history_list._head = self._history_list._tail = None
        print("\nDeleted last Haiku; history is now empty.\n")
        pause()
        return

    # Splice out to_delete
    if to_delete._prev:
        to_delete._prev._next = to_delete._next
    else:
        self._history_list._head = to_delete._next
    if to_delete._next:
        to_delete._next._prev = to_delete._prev
    else:
        self._history_list._tail = to_delete._prev
```

```
        to_delete._prev = to_delete._next = None
        print("\nDeleted that Haiku from history.\n")
        pause()
        continue

    if choice == 'u':
        new_label = input("Enter new label for this Haiku: ").strip()
        if new_label:
            current.label = new_label
            print(f"\nLabel updated to “{new_label}”.\n")
        else:
            print("\nNo change made.\n")
        pause()
        continue

    if choice == 'v':
        # Paginate through entire history
        nodes = []
        node = self._history_list._tail
        while node:
            nodes.append(node)
            node = node._prev

        page = 0
        per_page = 5
        total_pages = (len(nodes) - 1) // per_page

        while True:
```

```
start = page * per_page
end = min(start + per_page, len(nodes))
print(f"\n-- History (Page {page + 1} of {total_pages + 1}) --\n")
for idx_in_page, nd in enumerate(nodes[start:end], start=1):
    print(f"  {idx_in_page}. {nd.label}")

print("\n  0. Previous Page    [Enter] Next Page")
print("  o = Back to single view\n")
subchoice = input("Choose [0/1-5/o]: ").strip().lower()

if subchoice == 'o':
    break
if subchoice == "":
    if page < total_pages:
        page += 1
    else:
        print("\nAlready at last page.\n")
        pause()
    continue
if subchoice == '0':
    if page > 0:
        page -= 1
    else:
        print("\nAlready at first page.\n")
        pause()
    continue
if subchoice.isdigit():
    n = int(subchoice)
```

```
            if 1 <= n <= (end - start):
                current = nodes[start + (n - 1)]
                break
            else:
                print("\nInvalid selection.\n")
                pause()
            continue

        print("\nInvalid choice. Please try again.\n")
        pause()

    continue

if choice == 't':
    # Sort history submenu
    print("\n-- Sort History --")
    print("  1. Alphabetical by label")
    print("  2. By total syllable count")
    print("  3. By first-line lexical order")
    print("  4. Cancel\n")

    sort_choice = input("Choose [1-4]: ").strip()
    if sort_choice == '1':
        # Compare whole label strings, case-insensitive
        def cmp_fn(a, b):
            la = a.label.lower()
            lb = b.label.lower()
            return (lb > la) - (lb < la)
```

```
        elif sort_choice == '2':
            def cmp_fn(a, b):
                sa = sum(Menu.count_syllables(w) for w in a.text.split())
                sb = sum(Menu.count_syllables(w) for w in b.text.split())
                return (sb - sa)

        elif sort_choice == '3':
            # Compare first-line text
            def cmp_fn(a, b):
                fa = a.text.split("\n", 1)[0].lower()
                fb = b.text.split("\n", 1)[0].lower()
                return (fb > fa) - (fb < fa)

        else:
            continue

        self._history_list.sort(cmp_fn)
        print("\nHistory sorted successfully.\n")
        # After sorting, reset current to tail
        current = self._history_list._tail
        pause()
        continue

    print("\nInvalid choice. Please try again.\n")
    pause()

def handle_template_haiku(self):
```

```
print("\n-- Template-Driven Haiku Generator --\n")

# 1) Define templates
templates = [
  Template(["Adj","Noun","Verb","Adv"]),
  Template(["Noun","Verb","Adj","Noun"]),
  Template(["Adv","Verb","Noun","Adj","Noun"])
]

# 2) Load word bank once
bank = WordBank()
while True:
  path = input("Enter word-bank file: ").strip()
  try:
    bank.load(path)
    break
  except Exception as e:
    print(f"Failed to load bank: {e}")
    if not prompt_retry_or_menu():
      return

pause()

# 3) Now loop generating until user quits
while True:
  gen = TemplateHaikuGenerator(templates, bank)
  haiku = gen.generate()
```

```
print("\nGenerated Haiku:")
print("-" * 30)
print(haiku + "\n")
pause()

# 4) auto–save into history
self._history_list.append(label="Generated Haiku", text=haiku)

# 5) ask user whether to save to a file
if input("Save this haiku to a file? (y/n): ").strip().lower() == 'y':
  while True:
    fname = input("Enter filename: ").strip()
    if os.path.exists(fname):
      over = input(f"File \"{fname}\" exists. Override? (y/n): ").strip().lower()
      if over != 'y':
        print("Okay, choose a different filename.")
        continue
    try:
      with open(fname, 'w', encoding='utf-8') as f:
        f.write(haiku)
      print(f"Saved haiku to \"{fname}\"")
    except Exception as e:
      print(f"Failed to save file: {e}")
    break

pause()
# 6) ask whether to generate another
if input("Generate another haiku? (y/n): ").strip().lower() != 'y':
```

```
            break
```

general_utils.py:

```
def pause(txt="\nPress Enter to continue...\n"):
    input(txt)


def prompt_retry_or_menu() -> bool:
    """
    Ask the user whether to retry the current action or return to the
    Options Menu. Returns True to retry, False to return.
    """
    while True:
        resp = input(
            "Press Enter to retry or 'o' to return to Options Menu:  "
        ).strip().lower()
        if resp == 'o':
            pause()
            return False
        else:
            print()
            return True


def load_thesaurus(path: str) -> dict:
    """Load synonyms from a thesaurus file into a dict: keyword -> list of synonyms."""
    synonyms = {}
    with open(path, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.strip()
```

```
        if not line or ':' not in line:
            continue
        key, vals = line.split(':', 1)
        words = [v.strip() for v in vals.split(',') if v.strip()]
        synonyms[key.strip().lower()] = words
    return synonyms


def stack_to_list(stack) -> list[str]:
    """Helper to convert a Stack of words into a list in order."""
    vals = []
    node = stack.top
    while node:
        vals.append(node.val)
        node = node.next
    return list(reversed(vals))
```

doubly_linked_list.py:

```
class DLLNode:
    """
    Node for a doubly-linked list. Carries:
     - `label` : e.g. "Zenized haiku_003"
     - `text` :  the actual 3-line haiku string
     - `prev` , `next` : pointers
    """
    def __init__(self, label: str, text: str):
        self.label = label
        self.text = text
        self._prev = None   # type: DLLNode
```

```
        self._next = None   # type: DLLNode


class DoublyLinkedList:
    """
    Maintains all generated-Haiku nodes in chronological order
    (head = oldest, tail = newest).
    """
    def __init__(self):
        self._head = None  # type: DLLNode
        self._tail = None  # type: DLLNode


    def append(self, label: str, text: str) -> DLLNode:
        """
        Create a new node with (label, text) and attach at the tail.
        Returns the new node, so caller can push it onto UndoStack.
        """
        node = DLLNode(label, text)
        if not self._head:
            self._head = self._tail = node
        else:
            self._tail._next = node
            node._prev = self._tail
            self._tail = node
        return node


    def remove_after(self, node: DLLNode):
        """
        If `node` is the new tail after an undo, remove everything
```

```
    that used to come after it. O(1) pointer splice.
    """
    curr = node._next
    while curr:
        nxt = curr._next
        curr._prev = curr._next = None
        curr = nxt
    node._next = None
    self._tail = node

# ── Merge-Sort Helpers ──────────────────────────────────────
def _split(self, head: DLLNode) -> DLLNode:
    slow, fast = head, head
    while fast._next and fast._next._next:
        slow = slow._next
        fast = fast._next._next
    mid = slow._next
    slow._next = None
    if mid:
        mid._prev = None
    return mid

def _merge(self, left: DLLNode, right: DLLNode, compare) -> DLLNode:
    dummy = DLLNode("", "")
    tail = dummy
    while left and right:
        if compare(left, right) <= 0:
            tail._next = left
```

```
            left._prev = tail
            left = left._next
        else:
            tail._next = right
            right._prev = tail
            right = right._next
        tail = tail._next

    tail._next = left if left else right
    if tail._next:
        tail._next._prev = tail

    head = dummy._next
    if head:
        head._prev = None
    return head

def _merge_sort(self, head: DLLNode, compare) -> DLLNode:
    if not head or not head._next:
        return head
    mid = self._split(head)
    left = self._merge_sort(head, compare)
    right = self._merge_sort(mid, compare)
    return self._merge(left, right, compare)

def sort(self, compare):
    """
    In-place merge sort of the entire list using `compare(a, b)`.
```

```
        Afterward, self._head and self._tail are updated.
        """
        if not self._head or not self._head._next:
            return
        new_head = self._merge_sort(self._head, compare)
        curr = new_head
        prev = None
        while curr:
            prev = curr
            curr = curr._next
        self._head = new_head
        self._tail = prev
```

queues.py:

```
class QueueNode:
    def __init__(self, value):
        self.value = value
        self._next = None


class Queue:
    """A simple FIFO queue using a linked list under the hood."""
    def __init__(self):
        self._head = None  # points to oldest node
        self._tail = None  # points to newest node

    def enqueue(self, value):
        node = QueueNode(value)
        if not self._tail:     # empty queue
```

```
            self._head = self._tail = node
        else:
            self._tail._next = node
            self._tail = node

    def dequeue(self):
        if not self._head:
            raise IndexError("dequeue from empty queue")
        node = self._head
        self._head = node._next
        if not self._head:     # now empty
            self._tail = None
        return node.value

    def is_empty(self):
        return self._head is None
```

stack.py:

```
class StackNode:
    def __init__(self, val):
        self.val = val
        self._next = None  # type: StackNode

class Stack:
    def __init__(self):
        self._top = None  # type: StackNode
```

```
    def push(self, v):
        node = StackNode(v)
        node._next = self._top
        self._top = node

    def pop(self):
        if not self._top:
            raise IndexError("pop from empty stack")
        v = self._top.val
        self._top = self._top._next
        return v

    def is_empty(self):
        return self._top is None
```

trie.py:

```
class TrieNode:
    def __init__(self):
        self._children = {}
        self._keyword = None   # set when a full keyword/synonym ends here

class Trie:
    def __init__(self):
        self._root = TrieNode()

    def insert(self, word, key):
        """
        Insert word into the trie, marking its end node with key.
```

```
        """
        node = self._root
        for ch in word:
            node = node._children.setdefault(ch, TrieNode())
        node._keyword = key

    def find_all(self, text):
        """
        Scan text once; yield (start_idx, end_idx, _keyword) for every match.
        """
        matches = []
        for i in range(len(text)):
            node = self._root
            j = i
            while j < len(text) and text[j] in node._children:
                node = node._children[text[j]]
                j += 1
                if node._keyword is not None:
                    matches.append((i, j, node._keyword))
            # next starting position
        return matches
```

haiku_utils.py:

```
import random
import re

from trie import Trie
from queues import Queue
```

```
class HaikuTransformer:
    def __init__(self, thesaurus):
        """
        `thesaurus`: key -> [synonyms or _antonyms...]
        """
        # Build an invert map: every lookup-word -> its canonical key
        self._invert = {}
        for key, lst in thesaurus.items():
            self._invert[key] = key
            for w in lst:
                self._invert[w] = key

        # Build and populate the trie
        self._trie = Trie()
        for word, key in self._invert.items():
            self._trie.insert(word.lower(), key)

        # Store the canonical thesaurus lists
        self._thesaurus = thesaurus

    def transform(self, text):
        raise NotImplementedError("Must implement in subclass")

# --- SYNONYMIZE ------------------------------------------------------------

class SynonymizeTransformer(HaikuTransformer):
    def transform(self, text):
```

```
matches = self._trie.find_all(text.lower())

# Sort matches by start index (and longest match first)
matches.sort(key=lambda x: (x[0], -(x[1]-x[0])))

# Rebuild output by walking through text + matches
out = []
last = 0
for start, end, key in matches:
  # skip overlapping matches
  if start < last:
    continue
  # append untouched slice
  out.append(text[last:start])
  orig = text[start:end]
  syns = self._thesaurus.get(key, [])
  if syns:
    choice = random.choice(syns)
    # preserve capitalization
    if orig[0].isupper():
      choice = choice.capitalize()
    out.append(choice)
  else:
    out.append(orig)
  last = end

out.append(text[last:])
return ''.join(out)
```

```
# --- ZENIZE ------------------------------------------------------------

class ZenizeTransformer(HaikuTransformer):
   def transform(self, text):
      matches = self._trie.find_all(text.lower())
      matches.sort(key=lambda x: (x[0], -(x[1]-x[0])))


      out = []
      last = 0
      for start, end, key in matches:
         if start < last:
            continue
         out.append(text[last:start])
         orig = text[start:end]
         syns = self._thesaurus.get(key, [])
         if syns:
            # find shortest length
            min_len = min(len(s) for s in syns)
            shortest = [s for s in syns if len(s) == min_len]
            choice = random.choice(shortest)
            if orig[0].isupper():
               choice = choice.capitalize()
            out.append(choice)
         else:
            out.append(orig)
         last = end
```

```
        out.append(text[last:])
        return ''.join(out)

# --- ANTONYMIZE ----------------------------------------------------------

class AntonymizeTransformer(HaikuTransformer):
    def __init__(self,
                 synonym_thes,
                 antonym_thes):
        # init base with synonym thesaurus to build invert map & trie
        super().__init__(synonym_thes)
        # store _antonyms
        self._antonyms = antonym_thes

    def transform(self, text):
        matches = self._trie.find_all(text.lower())
        matches.sort(key=lambda x: (x[0], -(x[1]-x[0])))

        out = []
        last = 0
        for start, end, key in matches:
            if start < last:
                continue
            out.append(text[last:start])
            orig = text[start:end]
            ants = self._antonyms.get(key, [])
            if ants:
                choice = random.choice(ants)
```

```
            if orig[0].isupper():
                choice = choice.capitalize()
            out.append(choice)
        else:
            out.append(orig)
        last = end

    out.append(text[last:])
    return ''.join(out)

# --- BATCH ----------------------------------------------------------------

class BatchTransformer:
  def __init__(self, thesaurus):
    self._thesaurus = thesaurus
    # keep the same keyword order for deterministic naming
    self.keywords= list(thesaurus.keys())
    # precompile one regex per keyword
    self.patterns = [
      re.compile(rf'\b{re.escape(k)}\b', re.IGNORECASE)
      for k in self.keywords
    ]

  def stream_variants(self, text) :
    """
    Yields (index, variant_text) for each permutation of `text`
    by replacing keywords in self.keywords order, using a FIFO Queue.
    """
```

```
    queue = Queue()
    # each item: (current_text, next_keyword_index)
    queue.enqueue((text, 0))

    idx = 1
    while not queue.is_empty():
      curr_text, kidx = queue.dequeue()

      # if all keywords replaced, yield final variant
      if kidx >= len(self.keywords):
        yield idx, curr_text
        idx += 1
        continue

      key = self.keywords[kidx]
      syns = self._thesaurus[key]
      pat = self.patterns[kidx]

      for syn in syns:
        # replace only the kidx-th keyword
        def _repl(m):
          return syn.capitalize() if m.group(0)[0].isupper() else syn
        new_text = pat.sub(_repl, curr_text)
        queue.enqueue((new_text, kidx + 1))
```

haiku_generator.py:

```
import re
```

```
import random
from linked_list import LinkedList
from stack import Stack

from general_utils import stack_to_list

# --- Haiku Generator (Extra Feature 2) ---------------------------------------

class Template:
    def __init__(self, tags):
        self.tags = tags  # e.g. ["Adj","Noun","Verb","Adv"]

class WordBank:
    def __init__(self):
        self.by_pos = {}       # POS -> LinkedList of words
        self.syllables = {}    # word -> syllable count
        self.by_pos_list = {}  # POS -> list of words

    def load(self, path):
        with open(path, encoding='utf-8') as f:
            for lineno, raw in enumerate(f, 1):
                line = raw.strip()
                if not line or line.startswith('#'):
                    continue
                parts = line.split(':', 1)
                if len(parts) != 2:
                    print(f"[WordBank] Skipping invalid line {lineno}: {line}")
                    continue
```

```
            word, pos = parts
            ll = self.by_pos.setdefault(pos, LinkedList())
            ll.append(word)

        # --- build syllable cache ----------
        for pos, ll in self.by_pos.items():
            words = [node.text for node,_ in ll.iterate()]
            for w in words:
                # only compute once
                self.syllables[w] = self.count_syllables(w)

        # --- flatten lists for fast iteration -----
        self.by_pos_list = {
            pos: [node.text for node,_ in ll.iterate()]
            for pos, ll in self.by_pos.items()
        }

    def count_syllables(self, word: str) -> int:
        w = word.lower().strip(".,!?")
        if len(w) > 2 and w.endswith("e"):
            w = w[:-1]
        groups = re.findall(r"[aeiouy]+", w)
        return max(1, len(groups))

class TemplateHaikuGenerator:
    """
    Generates a "golden" haiku (5-7-5 syllables, lines 1&3 <=4 words,
    line 2 strictly longer than line 1), with a soft penalty on reused words.
```

```
"""
TARGETS = [5, 7, 5]
MAX_LINE_ATTEMPTS = 5
MAX_TEMPLATE_FALLBACK_ATTEMPTS = 3
MAX_BACKTRACK = 1000

def __init__(self, templates, bank, penalty_weight = 0.1):
   self.templates = templates
   self.pos_lists = bank.by_pos_list    # POS → list[str]
   self.syllables = bank.syllables      # word → int
   self.penalty_weight = penalty_weight

   # Pre-shuffle once per POS to avoid repeated shuffle() calls
   self.randomized_pos_lists = {
      pos: random.sample(words, len(words))
      for pos, words in self.pos_lists.items()
   }

def generate(self):
   lines = []
   used_lines = set()
   global_count = {}  # word -> times used

   # Templates with <=4 slots, for lines 1 & 3
   short_tpls = [t for t in self.templates if len(t.tags) <= 4]

   for i, target in enumerate(self.TARGETS):
      kwargs = {}
```

```
    # golden-format constraints
    if i == 0:  # line 1
        kwargs['templates'] = short_tpls
        kwargs['max_words'] = 4
    elif i == 1:  # line 2
        sum1 = sum(self.syllables[w] for w in lines[0].split())
        kwargs['min_syll'] = sum1 + 1
    else:  # line 3
        kwargs['templates'] = short_tpls
        kwargs['max_words'] = 4
        sum2 = sum(self.syllables[w] for w in lines[1].split())
        kwargs['max_syll'] = sum2 - 1

    line = self._make_unique_line(
        existing_lines=lines,
        used_lines=used_lines,
        global_count=global_count,
        target=target,
        **kwargs
    )

    lines.append(line)
    used_lines.add(line)
    for w in line.split():
        global_count[w] = global_count.get(w, 0) + 1

# Capitalize & punctuate
```

```
    formatted = []
    for ln in lines:
        if not ln:
            formatted.append(ln)
        else:
            s = ln[0].upper() + ln[1:]
            if not s.endswith('.'):
                s += '.'
            formatted.append(s)

    return "\n".join(formatted)


def _make_unique_line(self, existing_lines,
                      used_lines, global_count,
                      *, target: int,
                      min_syll = None, max_syll = None,
                      templates = None, max_words = None):

    tpl_pool = templates if templates is not None else self.templates
    # 1) Guided backtracking attempts
    for _ in range(self.MAX_LINE_ATTEMPTS):
        tpl  = random.choice(tpl_pool)
        seq  = self._fill_line(tpl, target, min_syll, max_syll, global_count)
        cand = " ".join(seq)

        # enforce word-count limit & uniqueness
        if (cand
            and (max_words is None or len(seq) <= max_words)
```

```
            and cand not in used_lines
        ):
            return cand

    # 2) Random-template fallback
    for _ in range(self.MAX_TEMPLATE_FALLBACK_ATTEMPTS):
        tpl  = random.choice(tpl_pool)
        seq  = self._full_template_line(tpl)
        cand = " ".join(seq)

        if (cand
            and (max_words is None or len(seq) <= max_words)
            and cand not in used_lines
        ):
            return cand

    # 3) Single-word fallback
    for words in self.pos_lists.values():
        for w in words:
            if w not in used_lines:
                return w

    # 4) Give up
    return existing_lines[-1] if existing_lines else ""

def _fill_line(self, tmpl,
            target, min_syll,
            max_syll, global_count):
```

```
best_seq   = []
best_score = float('inf')
used_words = []
visited    = set()

def backtrack(idx, syll_sum, steps):
  nonlocal best_seq, best_score
  # early exit if perfect & zero repeats
  if best_score == 0 or steps > self.MAX_BACKTRACK:
    return

  state = (idx, syll_sum, tuple(used_words))
  if state in visited:
    return
  visited.add(state)

  # done?
  if idx == len(tmpl.tags):
    if (min_syll is not None and syll_sum < min_syll) or \
      (max_syll is not None and syll_sum > max_syll):
       return

    # base diff + penalty for repeats
    diff = abs(syll_sum - target)
    repeat_penalty = sum(global_count.get(w, 0) for w in used_words)
    score = diff + self.penalty_weight * repeat_penalty
```

```
            if score < best_score:
                best_score, best_seq = score, list(used_words)
            return

        pos = tmpl.tags[idx]
        for w in self.randomized_pos_lists.get(pos, []):
            if w in used_words:
                continue
            s = self.syllables[w]
            if syll_sum + s > target:
                continue
            if max_syll is not None and syll_sum + s > max_syll:
                continue

            used_words.append(w)
            backtrack(idx+1, syll_sum + s, steps+1)
            if best_score == 0:
                return
            used_words.pop()

    backtrack(0, 0, 0)
    return best_seq

def _full_template_line(self, tmpl):
    used = set()
    out  = []
    for pos in tmpl.tags:
        pool = self.randomized_pos_lists.get(pos, [])
```

```
            choices = [w for w in pool if w not in used] or pool
            w = random.choice(choices)
            out.append(w)
            used.add(w)
        return out
```

linked_list.py:

```
class WordNode:
    def __init__(self, text, is_keyword = False):
        self.text = text
        self.is_keyword = is_keyword
        self._next = None  # type: WordNode

class LinkedList:
    def __init__(self):
        self._head = None  # type: WordNode
        self._tail = None  # type: WordNode

    def append(self, text, is_keyword = False):
        node = WordNode(text, is_keyword)
        if not self._head:
            self._head = self._tail = node
        else:
            self._tail._next = node
            self._tail = node

    def iterate(self):
        """Yield (node, index) from head to tail."""
```

```
    curr, idx = self._head, 0
    while curr:
        yield curr, idx
        curr = curr._next
        idx += 1

def clone(self):
    """Deep-copy this list of WordNodes (preserving is_keyword)."""
    new = LinkedList()
    for node, _ in self.iterate():
        new.append(node.text, node.is_keyword)
    return new
```