Programming Task 2 (Subtask 2) (10 points)

This task is more suitable if you do have an implementation you can build on top of from assignment 1. Choose bigram index or permuterm index and **implement one of these approaches**. Define **4 queries which contain two search terms connected with AND**. At least one of the two search terms should contain a wildcard.

(1) One of the four queries should have the wildcard on the left (2) one should have the wildcard on the right (3) one should have a wildcard between other characters (4) one should have one wildcard on the left and one on the right.

```python
from typing import List, Tuple, Dict
import re # only used for normalization
import pandas as pd

class Index:
    def __init__(self):
        # stores size of postings list and pointer to list in a tuple
using the normalized term as a key
        self.dictionary = {}
        # separate data structure which uses the pointer values of the
dict as its keys to get the corresponding posting lists to an entry in
the dict
        self.postings_lists = {}
        # the id counter is used to create new posting lists ids (in
ascending order)
        self.postings_list_id_counter = 0
        self.dataset: pd.DataFrame = None # optional, used to retrieve
tweet by id
        self.permuterm_index = {} # used for wildcard queries

    def normalize_term(self, term: str) -> str:
        """
        Normalize the term by converting it to lowercase, removing any
non-alphanumeric characters, and stemming.
        """
        term = re.sub(r"\W+", "", term.lower())
        return term

    def get_tweet_texts(self, tweet_ids: List[str]) -> List[str]:
        '''
        Get the text content of tweets given their IDs.
        '''

        # filter the DataFrame to only include rows with tweet_id in
tweet_ids
        filtered_df =
self.dataset[self.dataset['tweet_id'].isin(tweet_ids)]

        # return the text column of the filtered DataFrame
        return filtered_df['text'].tolist()



    def index(self, filename: str, permuterm_index: bool = True):
        """
        Index the documents in the given file.
        """
        # use quoting = 3 to ignore separators in quotes
        self.dataset = pd.read_csv(filename, sep='\t', header=None,
names=['date', 'tweet_id', 'handle', 'name', 'text'], quoting=3)
        # drop content duplicates if everything except tweet id is
identical
        # why? compresses size and removes redundancy, if texts like
parols are written multiple times they will be included, as the date
will be different each time
        self.dataset = self.dataset.drop_duplicates(subset=['date',
'handle', 'name', 'text'])
        # sort lines ascending by tweet id, so the postings are
inserted in a sorted way automatically
        self.dataset =
self.dataset.sort_values(by='tweet_id').reset_index(drop=True)
        # one line per tweet
        for _, row in self.dataset.iterrows():
            tweet_id = int(row['tweet_id']) # extract tweet id
            tweet_text = str(row['text']) # extract tweet string
            terms = tweet_text.split() # split on any whitespace char
            unique_terms = set()
            for term in terms:
                # normalize for better query results and less
redundant terms
                normalized_term = self.normalize_term(term)
                if normalized_term and normalized_term not in
unique_terms:
                    unique_terms.add(normalized_term)
                    if normalized_term not in self.dictionary:
                        postings_list_id =
self.postings_list_id_counter
                        # create posting list entry for new term
                        self.postings_lists[postings_list_id] = []
                        # store pointer to posting list in dict
                        self.dictionary[normalized_term] = (0,
postings_list_id)
                        self.postings_list_id_counter += 1

                    # get posting list of normalized term
                    size, postings_list_id =
self.dictionary[normalized_term]
                    postings_list =
self.postings_lists[postings_list_id]
                    # if no postings in list or last posting list
entry does not match id, append the new tweet and let next point to
none
                    if not postings_list or postings_list[-1][0] !=
tweet_id:
                        postings_list.append((tweet_id, None))
                        # update postings list size for term
                        self.dictionary[normalized_term] = (
                            size + 1,
                            postings_list_id,
                        )
                        if len(postings_list) > 1:
                            # update old end-of-postings pointer from
None to new entry
                            postings_list[-2] = (
                                postings_list[-2][0],
                                len(postings_list) - 1,
                            )

        # add permuterm index if requested
        if permuterm_index:
            self.build_permuterm_index()

    def query_single_term(self, term: str) -> List[Tuple[int, int]]:
        """
        Query the index for a single term and return the postings
list.
        """
        # normalize query term before checking entries in dict
        normalized_term = self.normalize_term(term)
        if normalized_term in self.dictionary:
            size, postings_list_id = self.dictionary[normalized_term]
            return self.postings_lists[postings_list_id]
        return []

    def build_permuterm_index(self):
        """Build a permuterm index for a given set of terms."""
        for term in self.dictionary.keys():
            # add the special symbol to mark the end and then generate
permutations (as we normalize, we can assume that this symbol was not
in the term)
            rotated_term = term + '$'
            for i in range(len(rotated_term)):
                # rotate the term
                rotated_term = rotated_term[1:] + rotated_term[0]
                # insert into the index
                if rotated_term not in self.permuterm_index:
                    self.permuterm_index[rotated_term] = set()
                self.permuterm_index[rotated_term].add(term)

class Index(Index):
    def intersect_postings_lists(
        self,
        postings_list1: List[Tuple[int, int]],
        postings_list2: List[Tuple[int, int]],
    ) -> List[Tuple[int, int]]:
        """
        Intersect two postings lists and return the common document
IDs.
        """
        result = []
        iter1 = iter(postings_list1)
        iter2 = iter(postings_list2)
        posting1 = next(iter1, None)
        posting2 = next(iter2, None)
        # implementation of two lists as shown in the lecture,
precondition: postings must be sorted in ascending order (was ensured
in index method)
        while posting1 is not None and posting2 is not None:
            doc_id1, next_posting1 = posting1
            doc_id2, next_posting2 = posting2
            if doc_id1 == doc_id2:
                if len(result) > 0:
                    result[-1] = (result[-1][0], len(result))
                result.append((doc_id1, None))
                posting1 = next(iter1, None) if next_posting1 is not
None else None
                posting2 = next(iter2, None) if next_posting2 is not
None else None
            elif doc_id1 < doc_id2:
                posting1 = next(iter1, None) if next_posting1 is not
None else None
            else:
                posting2 = next(iter2, None) if next_posting2 is not
None else None
        return result


    def merge_postings_lists(
        self,
        postings_list1: List[Tuple[int, int]],
        postings_list2: List[Tuple[int, int]],
    ) -> List[Tuple[int, int]]:
        """
        Merge two postings lists and return the common document IDs.
(Used for OR queries of wildcard terms)
```

```
        """
        result = []
        iter1 = iter(postings_list1)
        iter2 = iter(postings_list2)
        posting1 = next(iter1, None)
        posting2 = next(iter2, None)

        while posting1 is not None and posting2 is not None:
            doc_id1, _ = posting1
            doc_id2, _ = posting2

            # append one if the same and advance both
            if doc_id1 == doc_id2:
                result.append(posting1)
                posting1 = next(iter1, None)
                posting2 = next(iter2, None)
            # append the smaller one and advance it
            elif doc_id1 < doc_id2:
                result.append(posting1)
                posting1 = next(iter1, None)
            # advance the other pointer if doc_id2 < doc_id1
            else:
                result.append(posting2)
                posting2 = next(iter2, None)

        # ad remaining items from either list if one list is exhausted
before the other
        while posting1 is not None:
            result.append(posting1)
            posting1 = next(iter1, None)

        while posting2 is not None:
            result.append(posting2)
            posting2 = next(iter2, None)

        return result


    def query(self, *terms: str) -> List[str]:
        """
        Query the index for any number of AND combined terms and
return the document IDs.
        Wildcard queries are supported.
        """
        if not terms:
            return []

        postings_lists_and = []

        for term in terms:
            or_terms = set()
            if '*' not in term:
                # No wildcard, direct search

postings_lists_and.append(self.query_single_term(term))
            else:
                # Process the wildcard query
                left_wildcard = term.startswith('*')
                right_wildcard = term.endswith('*')
                query_parts = term.strip('*').split('*')

                if left_wildcard and right_wildcard:
                    # Wildcard on both sides, take the middle part
without $ (p.4/87)
                    search_term = query_parts[0]
                elif left_wildcard:
                    # Wildcard on the left
                    search_term = query_parts[0] + '$'
                elif right_wildcard:
                    # Wildcard on the right
                    search_term = '$' + query_parts[0]
                else:
                    # Wildcard in the middle
                    search_term = query_parts[1] + '$' +
query_parts[0]

                for key in self.permuterm_index:
                    if key.startswith(search_term):
                        for t in self.permuterm_index[key]:
                            or_terms.add(t)

                if not or_terms: # no matching terms found, AND query
will be empty, so you can already ret empty list
                    return []

                or_terms = list(or_terms) # back to list for iterating

                # get posting list for all wildcard matches with OR
query
                postings_list = self.query_single_term(or_terms[0])
                for term in or_terms[1:]:
                    postings_list2 = self.query_single_term(term)
                    postings_list =
self.merge_postings_lists(postings_list, postings_list2)

                # append OR query result to AND query list, to
intersect results of wildcard term with other posting lists
                postings_lists_and.append(postings_list)

        postings_list = postings_lists_and[0]
        # AND query
        for l in postings_lists_and[1:]:
            postings_list =
self.intersect_postings_lists(postings_list, l)

        return [doc_id for doc_id, _ in postings_list]

# Create index for tweets file
index = Index()
index.index("tweets.csv", permuterm_index=True)

# OLD TASK (still works)
# “show me tweets of people who talk about the side effects of malaria
vaccines”
resp = index.query("malaria", "side", "effects")

print(resp)

index.get_tweet_texts(resp)


# COMMENT: seems to be what we searched for, but if we include vaccine
in the request we will not find these posts as they do not mention
vaccines directly
```

```
[968853898185314306, 968855540985204738]
```

```
['Steroid-based compounds against Malaria: highly effective,
synergistic to artemisinin, no resistance, no side effects, up-scaling
possible. #malaria  https://t.co/minlnwx1f7',
 'Steroid-based compounds against Malaria: highly effective,
synergistic to artemisinin, no resistance, no side effects, up-scaling
possible. #malaria  https://t.co/mm8ne1EGVS @jlugiessen @GICAfrica
@GSK https://t.co/UQoDej13Uu']
```

```
# SUBTASK 2 queries

# test wildcard queries (subtask 2)

# (1) One of the four queries should have the wildcard on the left
resp = index.query("*ple", "white")
print(resp)
print(index.get_tweet_texts(resp))
# (2) one should have the wildcard on the right
resp = index.query("pe*", "white")
print(resp)
print(index.get_tweet_texts(resp))
# (3) one should have a wildcard between other characters
resp = index.query("adam*bmw", "and")
print(resp)
print(index.get_tweet_texts(resp))
```

```
# (4) one should have one wildcard on the left and one on the right.
resp = index.query("*accin*", "canc*", "tumours")
print(resp)
print(index.get_tweet_texts(resp))


[958849246438010881]
['@adamwiththebmw @Suzbehnan @K_Janeski This!!! I hate it when people
say “White people”, because the difference between American and
European behavior is huge. Like, please don’t throw us under the same
bus as them, we have almost nothing in common, except for our skin
color.']
[958849246438010881]
['@adamwiththebmw @Suzbehnan @K_Janeski This!!! I hate it when people
say “White people”, because the difference between American and
European behavior is huge. Like, please don’t throw us under the same
bus as them, we have almost nothing in common, except for our skin
color.']
[958849246438010881]
['@adamwiththebmw @Suzbehnan @K_Janeski This!!! I hate it when people
say “White people”, because the difference between American and
European behavior is huge. Like, please don’t throw us under the same
bus as them, we have almost nothing in common, except for our skin
color.']
[959008534120759296]
['Cancer “vaccine” makes tumours vanish https://t.co/gzdlUaP2sT
#science via @CosmosMagazine']
```