InvertedIndexReport

January 17, 2017

1 INVERTED INDEX SEARCH ENGIN

source code at my github repo

2 1. Report

2.1 Early Plan

first, I was trying to write the whole project in python3.5 and with the aid of the Wiley book, Algorithm and > Data structure in python, and I read the book up to end of chapter 8: trees, and I have built my code on that book basic, many expertise of python programming like annotation and decorator which supports in python3.5.2 had been used in this period and code developed practical, clean, designed, but snails, which takes about 2 weeks to complete, and are stored in my github repo: DataStructure_Python, in implementing TST and TrieST I have used the another famous book, Algorithm.4ed published by Addison-Wesley, which was written java friendly, I used its algorithm to developed my code.

2.2 Middle Plan

in middle of the project I decided to write the whole project in Jupyter notebook, it was challenging and also fun. but if you (like me) don't have a relible memory for remembering all of the methods for a class or variables, interacting with Jupyter gets hard.

2.3 Final Product

after the backend of the project completed, I should begin to implement the Graphical user interface (GUI), first I started to learn basic of Tkinter layout manager, but after a day or two, I gave up and search for something like swing in python, so I found Jython (an extinct project !!!!!) unfortunately Jython just support python2.7, but that was not a bad news because python2.7 was my first programming language that I learned. I translate all of my code in python2.7 and start coding in Jython with Jython compiler. after a while a faced with some frightening exception like "instance error" or "null pointer exception" or "java.lang.Runnable exception". But fortunately I overcame them finally and the whole project survive.

2.3.1 dependencies:

to run the project you need jython compiler, you can install this via this command:

`sudo apt-get install Jython`

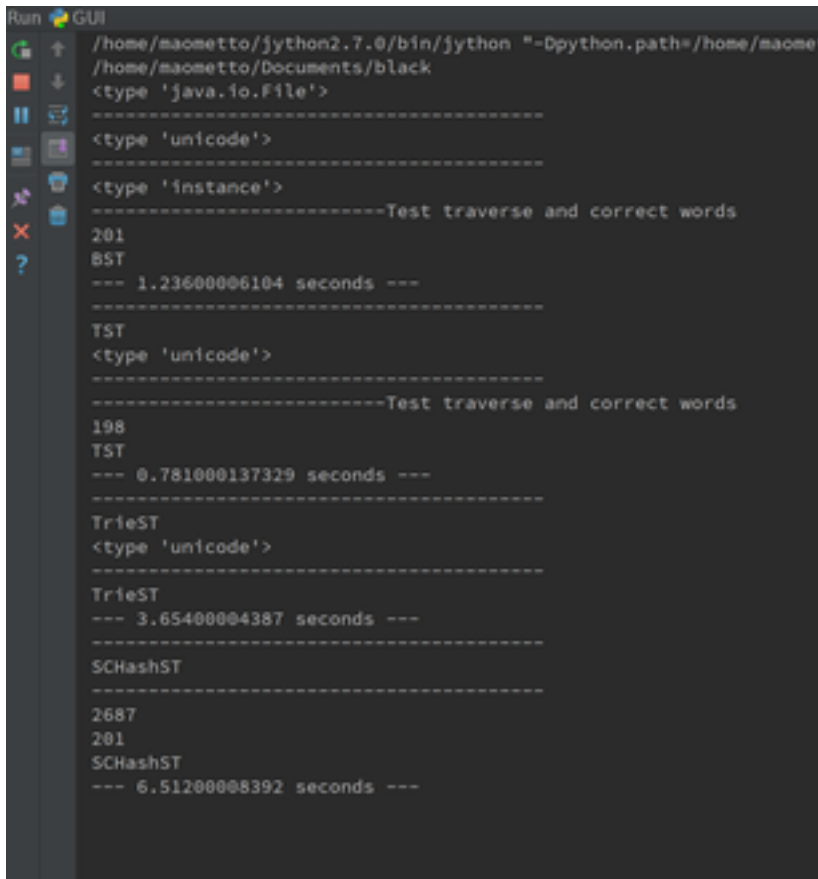to open jupyter notebook you should donwload this packages:

`sudo apt-get install anaconda3`

`sudo apt-get install ipython-notebook`

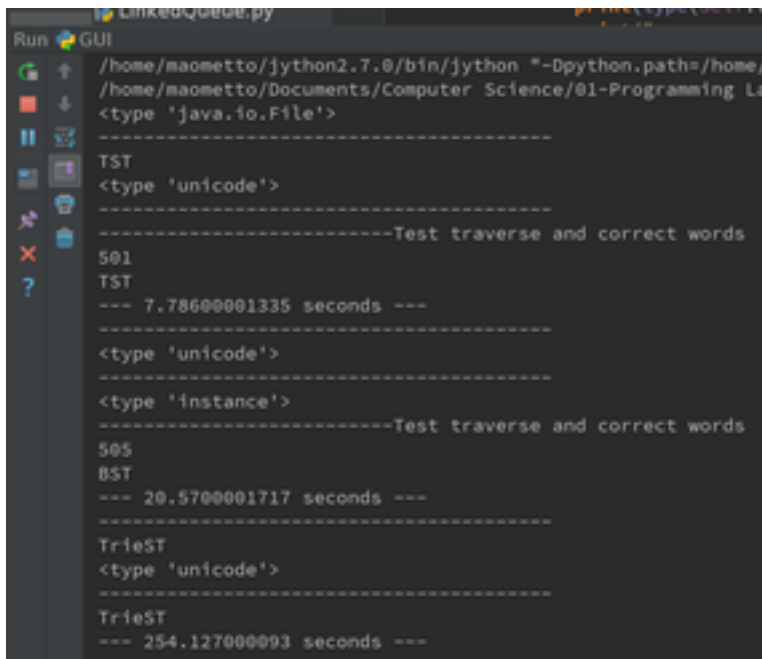`sudo apt-get install jupyter`

2.3.2   Observation

in the picture below, we can see the time spend for building TST was less than other, but it differ each time, actually when I test the code with relatively fewer documents, BST time sometimes was better than TST time.

```
Run  GUI
/home/maometto/jython2.7.0/bin/jython "-Dpython.path=/home/maome
/home/maometto/Documents/black
<type 'java.io.File'>
-----------------------------------------
<type 'unicode'>
-----------------------------------------
<type 'instance'>
---------------------------Test traverse and correct words
201
BST
--- 1.23600006104 seconds ---
-----------------------------------------
TST
<type 'unicode'>
-----------------------------------------
---------------------------Test traverse and correct words
198
TST
--- 0.781000137329 seconds ---
-----------------------------------------
TrieST
<type 'unicode'>
-----------------------------------------
TrieST
--- 3.65400004387 seconds ---
-----------------------------------------
SCHashST
-----------------------------------------
2687
201
SCHashST
--- 6.51200008392 seconds ---
```

But in large scale, it was completely different, and TST win the game obviously, another thing that should be mention is that in large scale, trieST was silly, it took nearly 35 times of TST to be built!!!!!!!!! . it may caused by the memory allocation problem or my algorithm, but any way TrieST loose the game.

```
LinkedQueue.py
Run  GUI
   ↑  /home/maometto/jython2.7.0/bin/jython "-Dpython.path=/home/
   ↓  /home/maometto/Documents/Computer Science/01-Programming La
   ⚙  <type 'java.io.File'>
   ⚙  ----------------------------------------
   ⚙  TST
   ⚙  <type 'unicode'>
   ⚙  ----------------------------------------
   ✖  -------------------------Test traverse and correct words
   ?  501
      TST
      --- 7.78600001335 seconds ---
      ----------------------------------------
      <type 'unicode'>
      ----------------------------------------
      <type 'instance'>
      -------------------------Test traverse and correct words
      505
      BST
      --- 20.5700001717 seconds ---
      ----------------------------------------
      TrieST
      <type 'unicode'>
      ----------------------------------------
      TrieST
      --- 254.127000093 seconds ---
```

and about hashtable I use seprate chaining design, but unfortunatly i have no enough time to do my best, so it was so slow, which takes about 10 minuates to do the job. another thing that must be mentioned is that the BST can find 4 more word than other data structure except hashST, these words are : "yesterday", "men", "woman", and another word that I am not sure. I can't understand why exactly these words.

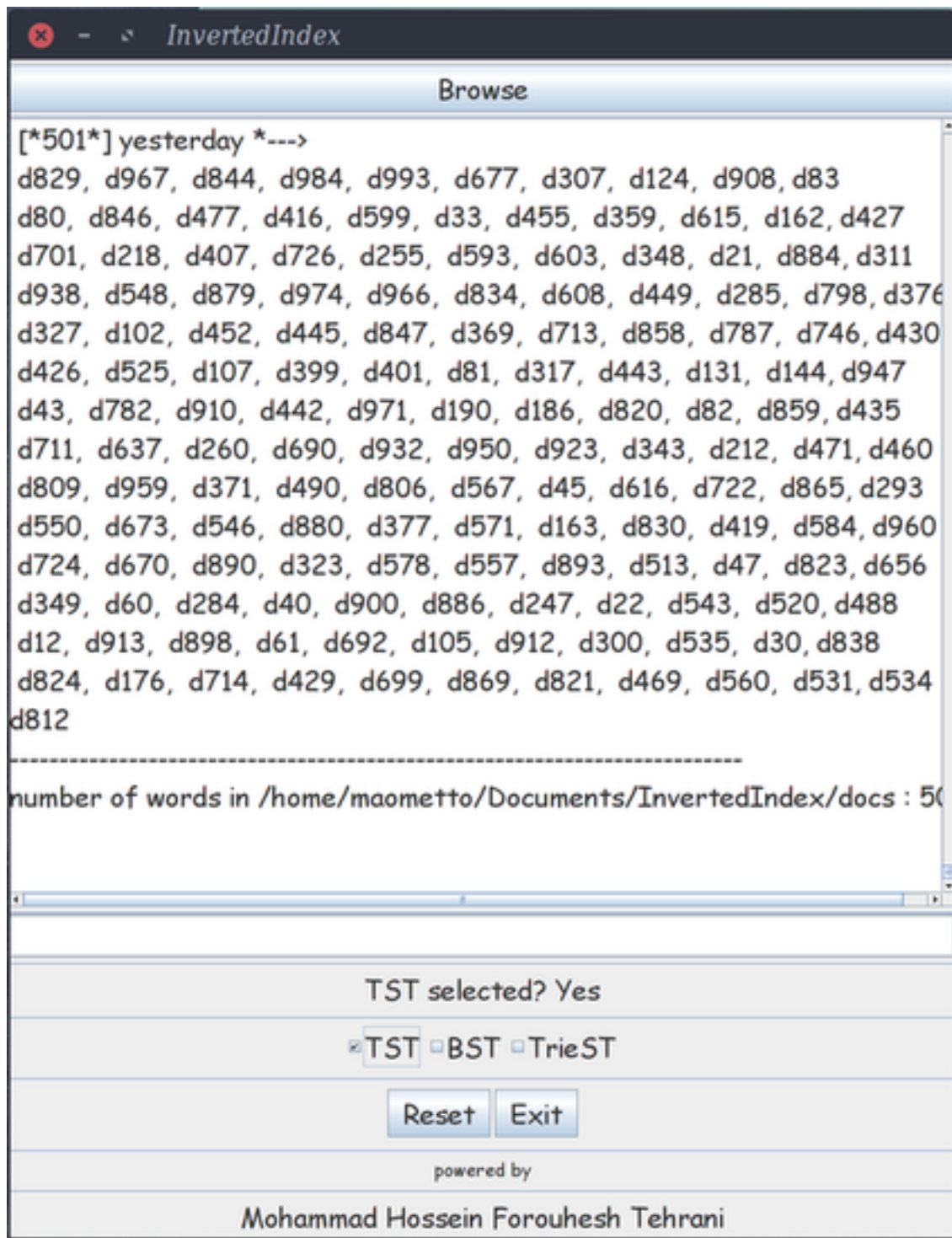## 2.4   here are some screenshoots of the project:

describtion:

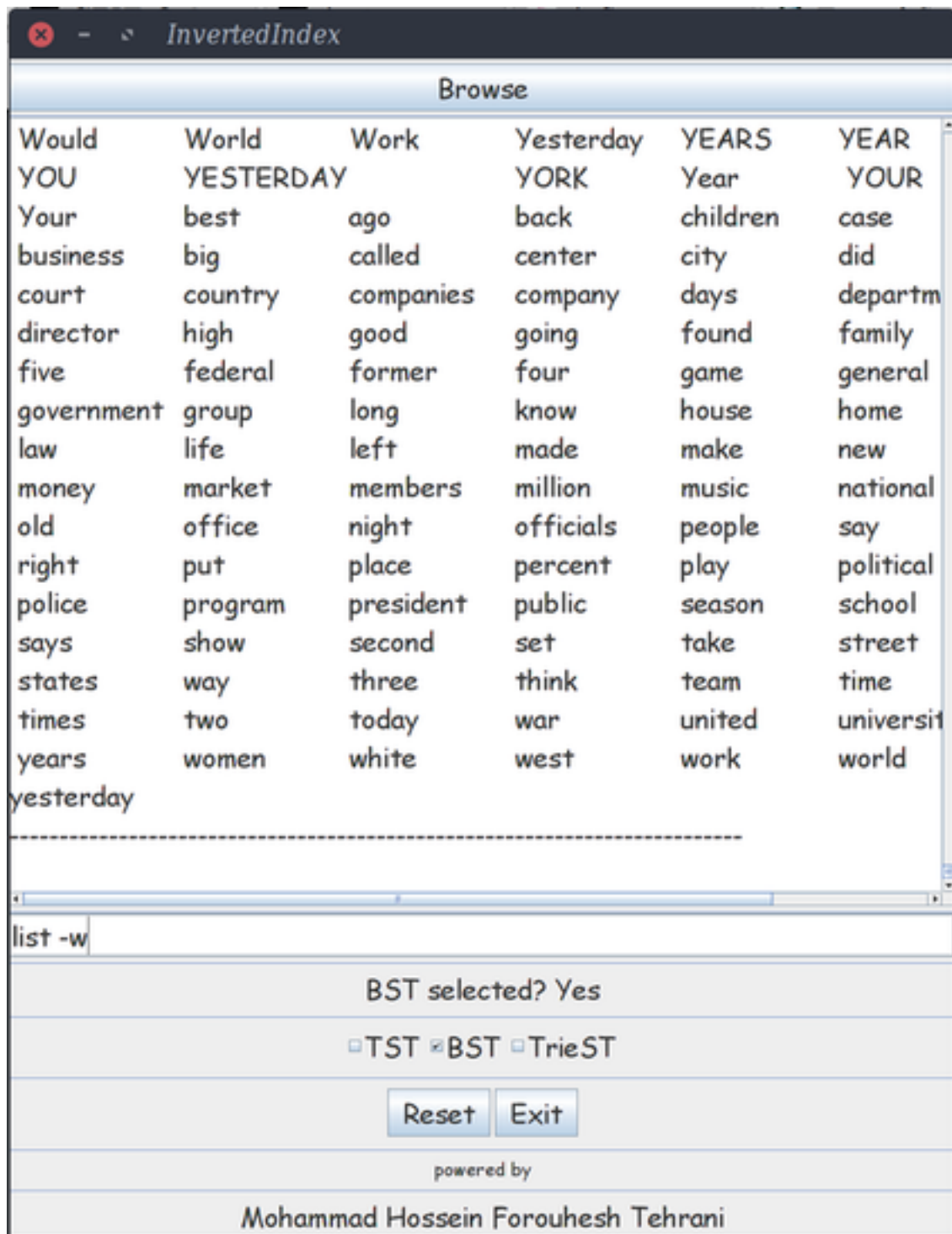this window will open when you run the program.

describtion:

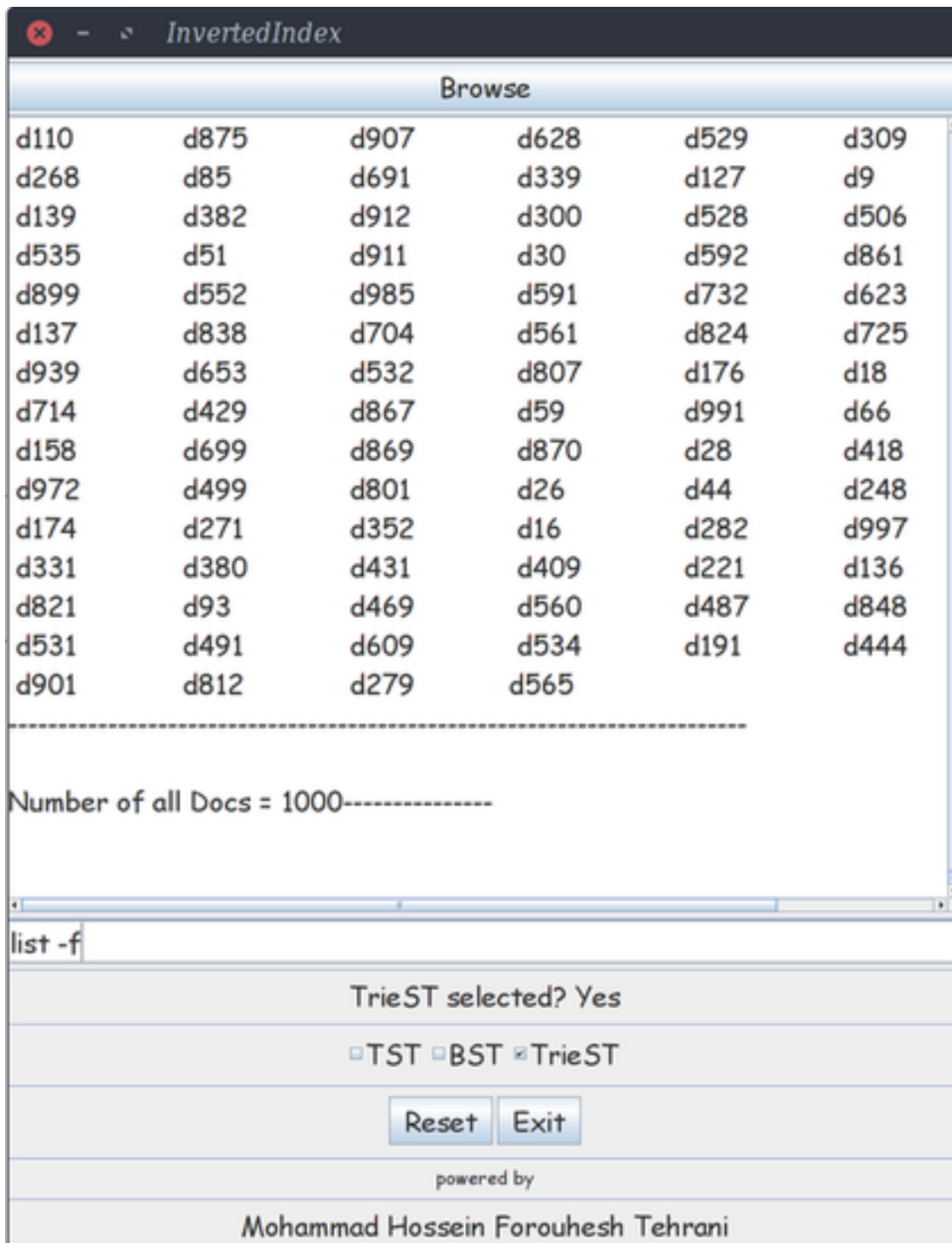when you hit the TST checkbox after a while, this will be shown to you.

describtion:

you may want just your words.

Browse

Would	World	Work	Yesterday	YEARS	YEAR
YOU	YESTERDAY		YORK	Year	YOUR
Your	best	ago	back	children	case
business	big	called	center	city	did
court	country	companies	company	days	departm
director	high	good	going	found	family
five	federal	former	four	game	general
government	group	long	know	house	home
law	life	left	made	make	new
money	market	members	million	music	national
old	office	night	officials	people	say
right	put	place	percent	play	political
police	program	president	public	season	school
says	show	second	set	take	street
states	way	three	think	team	time
times	two	today	war	united	universit
years	women	white	west	work	world
yesterday					

----

list -w

BST selected? Yes

□TST ▣BST □TrieST

Reset Exit

powered by

Mohammad Hossein Forouhesh Tehrani

describtion:

with this command you can see the files in the directory.

Browse

d110	d875	d907	d628	d529	d309
d268	d85	d691	d339	d127	d9
d139	d382	d912	d300	d528	d506
d535	d51	d911	d30	d592	d861
d899	d552	d985	d591	d732	d623
d137	d838	d704	d561	d824	d725
d939	d653	d532	d807	d176	d18
d714	d429	d867	d59	d991	d66
d158	d699	d869	d870	d28	d418
d972	d499	d801	d26	d44	d248
d174	d271	d352	d16	d282	d997
d331	d380	d431	d409	d221	d136
d821	d93	d469	d560	d487	d848
d531	d491	d609	d534	d191	d444
d901	d812	d279	d565		

Number of all Docs = 1000---------------

list -f

TrieST selected? Yes

☐TST ☐BST ☑TrieST

Reset Exit

powered by

Mohammad Hossein Forouhesh Tehrani

describtion:

you can search the words.

```
InvertedIndex
                            Browse
d997.txt    d331.txt    d331.txt    d331.txt    d331.txt
d331.txt    d331.txt    d380.txt    d380.txt    d380.txt
d431.txt    d431.txt    d431.txt    d409.txt    d409.txt
d409.txt    d221.txt    d221.txt    d221.txt    d221.txt
d221.txt    d136.txt    d136.txt    d136.txt    d136.txt
d136.txt    d136.txt    d136.txt    d136.txt    d821.txt
d821.txt    d93.txt     d93.txt     d93.txt     d93.txt
d469.txt    d469.txt    d469.txt    d560.txt    d560.txt
d560.txt    d560.txt    d487.txt    d487.txt    d487.txt
d487.txt    d487.txt    d848.txt    d848.txt    d848.txt
d531.txt    d531.txt    d531.txt    d531.txt    d531.txt
d491.txt    d491.txt    d491.txt    d491.txt    d609.txt
d609.txt    d609.txt    d534.txt    d534.txt    d534.txt
d534.txt    d534.txt    d534.txt    d534.txt    d534.txt
d534.txt    d191.txt    d191.txt    d191.txt    d444.txt
d444.txt    d901.txt    d901.txt    d901.txt    d812.txt
d812.txt    d812.txt    d279.txt    d279.txt    d279.txt
d565.txt    d565.txt    d565.txt    d565.txt    d565.txt
--------------------------------------------------------

search -w "work"

                      TST selected? Yes

                 ☑TST  ☐BST  ☐TrieST

                      Reset   Exit
                        powered by
            Mohammad Hossein Forouhesh Tehrani
```

3   2. Here is a short describtion of the data structure code:

3.1   DynamicArray

3.1.1   preview :

dynamic cpython array instead of python list

```python
In [1]: import ctypes

        class DynamicArray:
            """
            dynamic python array instead of python list
            """
            def __init__(self):
                """
                create an empty array
                """
                self._number_of_actual_elements = 0
                self._capacity = 1
                self._low_level_array = self._make_array(self._capacity)

            def __len__(self) -> int:
                """


                :return: the length of array
                """
                return self._number_of_actual_elements

            def __getitem__(self, index: int) -> object:
                """


                :type index: int
                :param index: the index of desired item in the array.
                :return: return the item.
                """
                if not 0 <= index < self._number_of_actual_elements:
                    raise IndexError("index out of range")
                return self._low_level_array[index]

            def is_empty(self):
                if self._number_of_actual_elements == 0:
                    return True
                return False

            def pop(self):
                if self.is_empty():
                    raise Exception("empty array")
                var = self._low_level_array[self._number_of_actual_elements - 1]
                self._number_of_actual_elements -= 1
                return var

            def append(self, item: object):
                """
                add object to the end of array
                :param item: desired object
                """
                if self._number_of_actual_elements == self._capacity:
                    self._resize(2 * self._capacity)
                self._low_level_array[self._number_of_actual_elements] = item
                self._number_of_actual_elements += 1
```

```python
def _resize(self, new_size: int):
    """
    resize the internal array to capacity new_size
    :param new_size: desired capacity
    """
    new_array = self._make_array(new_size)
    for k in range(self._number_of_actual_elements):
        new_array[k] = self._low_level_array[k]
    self._low_level_array = new_array
    self._capacity = new_size

def _make_array(self, capacity: int) -> object:
    """
    return new array with fixed capacity.
    :param capacity: the desired capacity
    :return: return ctypes array
    """
    return (capacity * ctypes.py_object)()
```

3.2   LinkedList

3.2.1   preview:

this is my linked list class which supports many feature like adding first and last and between, it

```python
In [2]: class LinkedList:
    class Node:
        """
        Lightweight, nonpublic class for storing a doubly linked node.
        """

        __slots__ = 'element', 'prev', 'after', 'upside_arrow', 'downside_arrow'  # streamline

        def __init__(self, element, prev, after):  # initialize node's fields
            self.element = element  # user's element
            self.prev = prev  # previous node reference
            self.after = after

    def __init__(self):
        """
        Create an empty list.
        """
        self.header = self.Node(None, None, None)
        self.trailer = self.Node(None, None, None)
        self.header.after = self.trailer  # trailer is after header
        self.trailer.prev = self.header  # header is before trailer
        self.size = 0

    def __len__(self):
        return self.size

    def __iter__(self):
        if self.is_empty():
            yield self.Node(None, None, None)
        current = self.header
```

```python
        while current is not None:
            yield current
            current = current.after

    def is_empty(self) -> bool:
        return self.size == 0

    def insert_between(self, element, predecessor: Node, successor: Node) -> Node:
        newest = self.Node(element, predecessor, successor)  # linked to neighbors
        predecessor.after = newest
        successor.prev = newest
        self.size += 1
        return newest

    def delete_node(self, node: Node) -> type(Node.element):
        predecessor = node.prev
        successor = node.after
        predecessor.after = successor
        successor.prev = predecessor
        self.size -= 1
        element = node.element  # record deleted element
        node.prev = node.after = node.element = None  # deprecate node
        return element

    # ----------------------------------------------------------------------------

    def add_first(self, element: type(Node.element)) -> Node:
        return self.insert_between(element, self.header, self.header.after)

    def add_last(self, element: type(Node.element)) -> Node:
        return self.insert_between(element, self.trailer.prev, self.trailer)

    def add_before(self, prevElement: type(Node.element), element: type(Node.element)) -> Node:
        original = self.search(prevElement)
        return self.insert_between(element, original.prev, original)

    def add_after(self, nextElement: type(Node.element), element: type(Node.element)) -> Node:
        original = self.search(nextElement)
        return self.insert_between(element, original, original.after)

    # ----------------------------------------------------------------------------

    def delete(self, undesireElement) -> type(Node.element):
        original = self.search(undesireElement)
        if original is None:
            return
        return self.delete_node(original)

    def search(self, desireElement: type(Node.element)) -> Node:
        head = self.header
        while head.after is not None:
            head = head.after
            if head.element == desireElement:
                return head
```

```
        return None
```

3.3 LinkedStack

```
In [3]: class LinkedStack():

            # ---------------------------------- Nested Class ----------------------------------
            class Node:

                __slots__ = 'element', 'next'  # streamline memory usage

                def __init__(self, element, next):  # initialize node field
                    self.element = element  # reference to current element
                    self.next = next  # reference to the next node

            # ---------------------------------- Stack Methods ----------------------------------
            def __init__(self):
                self.head = self.Node(None, None)  # head is a kind of node
                self.size = 0

            def __len__(self) -> int:
                return self.size

            def __iter__(self):
                if self.is_empty():
                    yield
                current = self.head
                while current is not None:
                    yield current
                    current = current.next

            def is_empty(self) -> bool:
                return self.size == 0

            def push(self, element: type(Node.element)):
                self.head = self.Node(element, self.head)  # created and linked a new node
                self.size += 1  # size is incremented

            def top(self) -> type(Node.element):
                if self.is_empty():
                    raise Exception("stack is empty")
                return self.head.element

            def pop(self) -> Node:
                if self.is_empty():
                    raise Exception("stack is empty")
                answer = self.head.element
                self.head = self.head.next  # bypass the former node :)
                self.size -= 1  # size decremented
                return answer

            def querry(self):
                """
                for TTD and debuging.
```

```
                """
                for k in self:
                    print(k.element)
```

3.4 LinkedQueue

```python
In [4]: class LinkedQueue:
            """
            FIFO implementation of queue with using linked list as internal storage
            """

            # ------------------------------ Nested Class ------------------------------
            class Node:
                "light weight class for storing liked node"
                __slots__ = 'element', 'next'                      # streamline memory usage

                def __init__(self, element, next):                 # initialize node field
                    self.element = element                         # reference to current element
                    self.next = next                               # reference to the next node

            # ------------------------------ Stack Methods ------------------------------
            def __init__(self):
                self._head = self.Node(None, None)
                self._tail = self.Node(None, None)
                self._size = 0

            def __len__(self) -> int:
                """
                return the number of elements in the linked list
                :return: integer
                """
                return self._size

            def __iter__(self):
                """
                iterate thorough the linked list
                """
                if self.is_empty():
                    yield
                current = self._head
                while current is not None:
                    yield current
                    current = current.next

            def is_empty(self) -> bool:
                """

                :return: bool True if list is empty and False otherwise
                """
                return self._size == 0

            def first(self):
                """
                just Return the first element in the queue
```

```python
        """
        if self.is_empty():
            raise Exception("empty Error")
        return self._head.element

    def dequeue(self):
        """
        remove and return the first element
        """
        if self.is_empty():
            raise Exception("empty Error")
        var = self._head.element
        self._head = self._head.next
        self._size -= 1
        if self.is_empty():
            self._tail = None
        return var

    def enqueue(self, element):
        newest = self.Node(element, None)
        if self.is_empty():
            self._head = newest
        else:
            self._tail.next = newest
        self._tail = newest
        self._size += 1

    def querry(self):
        """
        for TTD and debuging.
        """
        for k in self:
            print(k.element)
```

3.5 Binary Search Tree

3.5.1 preview:

my BST class is implemented with aim of inner node class and an abstract meta class which provide t

```python
In [5]: from abc import ABCMeta, abstractmethod
        import os
        import re


        class ABCNode(metaclass=ABCMeta):
            @abstractmethod
            def insert(self, key, NodeType):
                pass

            @abstractmethod
            def find(self, key):
                pass

            @abstractmethod
```

```python
    def minimum(self):
        pass

    @abstractmethod
    def successor(self):
        pass

    @abstractmethod
    def delete(self):
        pass


class BST(object):
    class Node(ABCNode):
        __slots__ = "key", "parent", "left", "right", "size", "repetition"

        # --------------------------------------------------------------------------------

        def __init__(self, parent, key):
            """Create a new leaf with key t."""
            self.key = key
            self.parent = parent
            self.left = None
            self.right = None
            self.size = 1
            self.repetition = 1

        # --------------------------------------------------------------------------------

        def update_stats(self):
            """Updates this node's size based on its children's sizes."""
            self.size = (0 if self.left is None else self.left.size) \
                      + (0 if self.right is None else self.right.size)

        # --------------------------------------------------------------------------------

        def insert(self, key, NodeType) -> ABCNode:
            self.size += 1
            if key < self.key:
                if self.left is None:
                    self.left = NodeType(self, key)
                    return self.left
                else:
                    return self.left.insert(key, NodeType)
            elif key == self.key:
                self.repetition += 1
                return
            else:
                if self.right is None:
                    self.right = NodeType(self, key)
                    return self.right
                else:
                    return self.right.insert(key, NodeType)
```

```python
# --------------------------------------------------------------------------------

def find(self, key) -> ABCNode:
    """Return the node for key if it is in this tree, or None otherwise."""
    if key == self.key:
        return self
    elif key < self.key:
        if self.left is None:
            return None
        else:
            return self.left.find(key)
    else:
        if self.right is None:
            return None
        else:
            return self.right.find(key)


# --------------------------------------------------------------------------------

def rank(self, key) -> int:
    """Return the number of keys <= key in the subtree rooted at this node."""
    left_size = 0 if self.left is None else self.left.size
    if key == self.key:
        return left_size + 1
    elif key < self.key:
        if self.left is None:
            return 0
        else:
            return self.left.rank(key)
    else:
        if self.right is None:
            return left_size + 1
        else:
            return self.right.rank(key) + left_size + 1

def minimum(self) -> ABCNode:
    """Returns the node with the smallest key in the subtree rooted by this node."""
    current = self
    while current.left is not None:
        current = current.left
    return current

def successor(self) -> ABCNode:
    """
    Returns the node with the smallest key larger than this node's key,
    or None if this has the largest key in the tree.
    """
    if self.right is not None:
        return self.right.minimum()
    current = self
    while current.parent is not None and current.parent.right is current:
        current = current.parent
    return current.parent
```

```python
    # ----------------------------------------------------------------------------

    def delete(self) -> ABCNode:
        """Delete this node from the tree."""
        if self.left is None or self.right is None:
            if self is self.parent.left:
                self.parent.left = self.left or self.right
                if self.parent.left is not None:
                    self.parent.left.parent = self.parent
            else:
                self.parent.right = self.left or self.right
                if self.parent.right is not None:
                    self.parent.right.parent = self.parent
            current = self.parent
            while current.key is not None:
                current.update_stats()
                current = current.parent
            return self
        else:
            s = self.successor()
            self.key, s.key = s.key, self.key
            return s.delete()

    # ----------------------------------------check for error-----------------------

    def check(self, lower_key, higher_key):
        """
        Checks that the subtree rooted at key is a valid BST
        and all keys are between (lower_key, higher_key).
        """
        if lower_key is not None and self.key <= lower_key:
            raise Exception("BST RI violation")
        if higher_key is not None and self.key >= higher_key:
            raise Exception("BST RI violation")
        if self.left is not None:
            if self.left.parent is not self:
                raise Exception("BST RI violation")
            self.left.check(lower_key, self.key)
        if self.right is not None:
            if self.right.parent is not self:
                raise Exception("BST RI violation")
            self.right.check(self.key, higher_key)
        if self.size != 1 + (0 if self.left is None else self.left.size) + (
                0 if self.right is None else self.right.size):
            raise Exception("BST RI violation")

    def __repr__(self) -> str:
        return "<BST Node, key:" + str(self.key) + ">"

def __init__(self, NodeType=Node):
    self.root = None
    self.NodeType = NodeType
    self.psroot = self.NodeType(None, None)
```

```python
        self.content = DynamicArray()
        self.docs = DynamicArray()

    # --------------------------------------------------------------------------------
    def reroot(self):
        self.root = self.psroot.left

    def insert(self, key) -> Node:
        """Insert key into this BST, modifying it in-place."""
        if self.root is None:
            self.psroot.left = self.NodeType(self.psroot, key)
            self.reroot()
            self.root.update_stats()
            return self.root
        else:
            return self.root.insert(key, self.NodeType)

    def add_doc(self, doc_name: str):
        self.docs.append(doc_name)

    # --------------------------------------------------------------------------------
    def find(self, key) -> Node:
        """Return the node for key if is in the tree, or None otherwise."""
        if self.root is None:
            return None
        else:
            return self.root.find(key)

    def rank(self, key) -> int:
        """The number of keys <= key in the tree."""
        if self.root is None:
            return 0
        else:
            return self.root.rank(key)

    def traverse(self, node: Node = None) -> list:
        if node is None:
            node = self.root

        self.content.append(node)
        if node.left is not None:
            self.traverse(node=node.left)
        if node.right is not None:
            self.traverse(node=node.right)

    def delete(self, key) -> Node:
        node = self.find(key)
        if node is None:
            raise Exception("nadari in klid ro")
        deleted = node.delete()
        self.reroot()
        return deleted
```

```python
        # ----------------------------------------------------------------------

        def check(self):
            if self.root is not None:
                self.root.check(None, None)  # check in the Node class

        def __str__(self) -> str:
            if self.root is None:
                return '<empty tree>'

            def recurse(node):
                if node is None: return [], 0, 0
                label = str(node.key)
                left_lines, left_pos, left_width = recurse(node.left)
                right_lines, right_pos, right_width = recurse(node.right)
                middle = max(right_pos + left_width - left_pos + 1, len(label), 2)
                pos = left_pos + middle // 2
                width = left_pos + middle + right_width - right_pos
                while len(left_lines) < len(right_lines):
                    left_lines.append(' ' * left_width)
                while len(right_lines) < len(left_lines):
                    right_lines.append(' ' * right_width)
                if (middle - len(label)) % 2 == 1 and node.parent is not None and \
                              node is node.parent.left and len(label) < middle:
                    label += '.'
                label = label.center(middle, '.')
                if label[0] == '.': label = ' ' + label[1:]
                if label[-1] == '.': label = label[:-1] + ' '
                lines = [' ' * left_pos + label + ' ' * (right_width - right_pos),
                         ' ' * left_pos + '/' + ' ' * (middle - 2) +
                         '\\' + ' ' * (right_width - right_pos)] + \
                        [left_line + ' ' * (width - left_width - right_width) + right_line

                         for left_line, right_line in zip(left_lines, right_lines)]
                return lines, pos, width

            return '\n'.join(recurse(self.root)[0])
```

3.6 Ternary Search Tree

3.6.1 preview :

based on string searching method and functionality.

```python
In [6]: class TST:
            __slots__ = 'size', 'root', 'valid_words', 'docs'

            # --------------------------- inner class ---------------------------

            class Node:
                __slots__ = 'key_char', 'left', 'mid', 'right', 'value'

                def __init__(self, key_char: str):
                    self.key_char = key_char
                    self.left = None
```

19

```python
            self.mid = None
            self.right = None
            self.value = str()

    # ------------------- --------- end of inner class -----------------------

    def __init__(self):
        self.root = TST.Node(" ")
        self.size = 0
        self.valid_words = LinkedQueue()
        self.docs = DynamicArray()

    def __sizeof__(self) -> int:
        return self.size

    def __contains__(self, item: str) -> bool:
        if item is None:
            raise Exception("nothing to be contained!!!")
        return self.__getitem__(item) is not None

    def __getitem__(self, item: str) -> str:
        if item is None:
            raise Exception("call __getitem__ with None argument")
        if len(item) == 0:
            raise Exception("item must have length >= 1")
        x = self.get(self.root, item, 0)
        if x is None:
            return None
        return x.value

    def intable(self, stream: object) -> tuple:
        try:
            integer = int(stream)
            return True, integer
        except Exception as err:
            return False, err

    def get(self, x: Node, item: str, d: int) -> Node:
        """
        return sub-trie corresponding to given key
        """
        if x is None:
            return None
        if len(item) == 0:
            raise Exception("item must have length >= 1")
        char = item[d]
        if char < x.key_char:
            return self.get(x.left, item, d)
        elif char > x.key_char:
            return self.get(x.right, item, d)
        elif d < len(item) - 1:
            return self.get(x.mid, item, d + 1)
        else:
            return x
```

```python
def put(self, item: str, value: int) -> None:
    """
    Inserts the key-value pair into the symbol table
    """
    if item is None:
        raise Exception("call __setitem__ with None argument")
    else:
        self.size += 1
    self.root = self.set(self.root, item, value, 0)


def set(self, x: Node, item: str, value: int, d: int) -> Node:
    char = item[d]
    if x is None:
        x = TST.Node(char)
        # self.size += 1
    if char < x.key_char:
        x.left = self.set(x.left, item, value, d)
    elif char > x.key_char:
        x.right = self.set(x.right, item, value, d)
    elif d < len(item) - 1:
        x.mid = self.set(x.mid, item, value, d + 1)
    else:
        x.value = value
    return x

def longestPrefixOf(self, query: str) -> str:
    if query is None:
        raise Exception("call longestPrefixOf() with None argument")
    if len(query) == 0:
        return None
    length = int(0)
    x = self.root
    i = 0
    while x is not None and i < len(query):
        char = query[i]
        if char < x.key_char:
            x = x.left
        elif char > x.key_char:
            x = x.right
        else:
            i += 1
            if x.value is not None:
                length = i
            x = x.mid

    return query[0:length]          # testing required

def keys(self) -> LinkedQueue:
    queue = LinkedQueue()
    self.collect(self.root, str(), queue)
    return queue                        # queue is iterable?
```

```python
    def keysWithPrefix(self, prefix: str) -> LinkedQueue:

        if prefix is None:
            raise Exception("call keysWithPrefix() with None argument")
        queue = LinkedQueue()
        x = self.root
        x = self.get(x, prefix, 0)
        if x is None:
            return queue
        if x.value is not None:
            queue.enqueue(prefix)
        self.collect(x.mid, str(prefix), queue)
        return queue

    def collect(self, x: Node, prefix: str, queue: LinkedQueue) -> None:
        if x is None:
            return None
        self.collect(x.left, prefix, queue)
        if x.value is not None:
            queue.enqueue(str(prefix) + x.key_char)
        # self.collect(x.mid, str(prefix) + str(x.key_char), queue)
        prefix = prefix[:-1]
        self.collect(x.right, prefix, queue)

    def keysThatMatch(self, pattern: str) -> LinkedQueue:
        queue = LinkedQueue()
        self.patternMatching(self.root, str(), 0, pattern, queue)
        return queue

    def patternMatching(self, x: Node, prefix: str, i: int, pattern: str, queue: LinkedQueue):
        """
        some kind of collector
        """
        if x is None:
            return
        char = pattern[i]
        if char == '.' or char < x.key_char:
            self.patternMatching(x.left, prefix, i, pattern, queue)
        if char == '.' or char == x.key_char:
            if i == len(pattern) - 1 and x.value is not None:
                queue.enqueue(str(prefix) + str(x.key_char))
            if i < len(pattern) - 1:
                self.patternMatching(x.mid, str(prefix) + str(x.key_char), i + 1, pattern, queue)
                prefix = prefix[:-1]

        if char == '.' or char > x.key_char:
            self.patternMatching(x.right, prefix, i, pattern, queue)


    def add_doc(self, doc_name: str):
        self.docs.append(doc_name)

    def traverse(self):
        if self.size == 0:
```

```
                    raise Exception("empty tst can't be traversed")
                for q in self.keys():
                    if self[q.element] is not None and self[q.element] is not "":
                        yield q.element

            def validation(self):
                for v in self.traverse():

                    self.valid_words.enqueue(v)
```

3.7   TrieST

3.7.1   previwe :

this class is based on a light weight inner Node class, it can supports all 256 ascii character :))

```
In [7]: class TrieST:
            R = 256
            __slots__ = 'root', 'number_of_keys', 'valid_words', 'docs'

            # ------------------------------------------------------------------------------
            class Node:
                __slots__ = 'value', 'next'

                def __init__(self):
                    self.value = str()
                    self.next = [None] * TrieST.R

            def __init__(self):
                self.root = self.Node()
                self.number_of_keys = 0
                self.valid_words = LinkedQueue()
                self.docs = DynamicArray()


            def __sizeof__(self) -> int:
                return self.number_of_keys

            def __len__(self) -> int:
                return self.__sizeof__()

            def is_empty(self) -> bool:
                return self.__sizeof__() == 0

            def __getitem__(self, key: str) -> str:
                x = self.get(self.root, key, 0)
                if x is None:
                    return None
                return str(x.value)

            def __contains__(self, key: str) -> bool:
                return self.__getitem__(key) is not None

            def get(self, x: Node, key: str, d: int) -> Node:
                if x is None:
```

```python
            return None
        if d == len(key):
            return x
        char = key[d]
        return self.get(x.next[int(ord(char))], key, d + 1)

    def put(self, key: str, value: int):
        if value is None:
            del key
        else:
            self.root = self.set(self.root, key, value, 0)

    def set(self, x: Node, key: str, value: int, d: int) -> Node:
        if x is None:
            x = self.Node()
        if d == len(key):
            if x.value is None:
                self.number_of_keys += 1
            x.value = value
            return x
        char = key[d]
        x.next[int(ord(char))] = self.set(x.next[int(ord(char))], key, value, d + 1)
        return x

    def keys(self) -> LinkedQueue:
        return self.keysWithPrefix("")

    def keysWithPrefix(self, prefix: str) -> LinkedQueue:
        result = LinkedQueue()
        x = self.get(self.root, prefix, 0)
        self.collect(x, str(prefix), result)
        return result

    def collect(self, x: Node, prefix: str, result: LinkedQueue) -> None:
        if x is None:
            return
        if x.value is not None:
            result.enqueue(str(prefix))
        for i in range(self.R):
            prefix += chr(i)
            self.collect(x.next[i], prefix, result)
            prefix = prefix[:-1]

    def keysThatMatch(self, pattern: str) -> LinkedQueue:
        result = LinkedQueue()
        self.patternMatching(self.root, str(), pattern, result)
        return result

    def patternMatching(self, x: Node, prefix: str, pattern: str, result: LinkedQueue) -> None:
        if x is None:
            return
        d = len(prefix)
        if d == len(pattern) and x.value is not None:
            result.enqueue(str(prefix))
```

```python
        if d == len(pattern):
            return

        char = pattern[d]
        if char == '.':
            for i in range(self.R):
                prefix += str(i)
                self.patternMatching(x.next[i], prefix, pattern, result)
                prefix = prefix[:-1]
        else:
            prefix += str(char)
            self.patternMatching(x.next[int(ord(char))], prefix, pattern, result)
            prefix = prefix[:-1]

    def longestPrefix(self, query: str) -> str:
        length = self.longestPrefixOf(self.root, query, 0, -1)
        if length == -1:
            return None
        else:
            return query[:length]

    def longestPrefixOf(self, x: Node, query: str, d: int, length: int) -> int:
        if x is None:
            return length
        if x.value is not None:
            length = d
        if d == len(query):
            return length
        char = query[d]
        return self.longestPrefixOf(x.next[int(ord(char))], query, d + 1, length)

    def add_doc(self, doc_name: str):
        self.docs.append(doc_name)

    def traverse(self):
        query = self.keys()
        for q in query:
            if self[q.element] is not "":
                yield(q.element)

    def validation(self):
        for v in self.traverse():
            self.valid_words.enqueue(v)

    def delete(self, key: str, x: Node=None, d: int=None) -> Node:
        if x is None and d is None:
            self.root = self.delete(key, x=self.root, d=0)
        if x is None:
            return None
        if d == len(key):
            if x.value is not None:
                self.number_of_keys -= 1
            x.value = None
        else:
```

```
            char = key[d]
            x.next[int(ord(char))] = self.delete(x.next[int(ord(char))], key, d + 1)

        # remove subtrie rooted at x if it is completely empty
        if x.value is not None:
            return x
        for i in range(self.R):
            if x.next[i] is not None:
                return x
        return None
```

3.8   Unsorted Table Map

```
In [ ]: class UnsortedTableMap:
            class _Item:
                """store key-value pairs."""

                def __init__(self, k, v, doc_list_appendance=None):
                    self._key = k
                    self._value = v
                    self.doc_list = [doc_list_appendance]

                    # self.doc_list = list()
                    # self.doc_list = self.doc_string.strip(" ")

                def __eq__(self, other):
                    return self._key == other._key

                def __ne__(self, other):
                    return not (self == other)

                def __lt__(self, other):
                    return self._key < other._key

            def __init__(self):
                self._table = []

            def __contains__(self, val):
                for i in self:
                    if self[i] == val:
                        return True
                return False

            def __getitem__(self, k, get_doc=False, trav=False):
                try:
                    for item in self._table:
                        if k == item._key:
                            if get_doc:
                                return item.doc_list
                            elif trav:
                                return item
                            else:
                                return item._value
                except KeyError:
```

```python
            pass

    def __setitem__(self, k, v, set_doc=None):
        for item in self._table:
            if k == item._key:
                item._value = v
                return
            if v == item._value:
                item._key = k
                return

        self._table.append(self._Item(k, v, set_doc))

    def __delitem__(self, k):
        for j in range(len(self._table)):
            if k == self._table[j]._key:
                self._table.pop(j)
                return   # and quit
        raise KeyError('Key Error: ' + repr(k))

    def __len__(self):
        return len(self._table)

    def __iter__(self):
        for item in self._table:
            yield item._key                                  # yield the KEY
```

3.9 Seperate Chaining Hash Search Tree

```python
In [ ]: from random import randrange

        from UnsortedTable import UnsortedTableMap
        from LinkedQueue import LinkedQueue


        class SCHashST:
            def __init__(self, cap=11, p=109345121):
                self._table = cap * [None]
                self._n = 0
                self._prime = p
                self._scale = 1 + randrange(p - 1)
                self._shift = randrange(p)

            def _hash_function(self, key):
                return (hash(key) * self._scale + self._shift) % self._prime % len(self._table)

            def __len__(self):
                return self._n

            def __getitem__(self, key, get_doc=False, trav=False):
                j = self._hash_function(key)
                return self._bucket_getitem(j, key, get_doc=get_doc, trav=trav)

            def __contains__(self, val):
```

```python
        for i in self:
            if val == self[i]:
                return True
        return False

    def __setitem__(self, key, value):
        j = self._hash_function(key)
        self._bucket_setitem(j, key, value, None)
        if self._n > len(self._table) // 2:              # keep load factor <= 0.5
            self._resize(2 * len(self._table) - 1)        # number 2^x - 1 is often prime

    def setitem_(self, key, value, set_doc):
        j = self._hash_function(key)
        self._bucket_setitem(j, key, value, set_doc)
        if self._n > len(self._table) // 2:              # keep load factor <= 0.5
            self._resize(2 * len(self._table) - 1)        # number 2^x - 1 is often prime

    def __delitem__(self, key):
        j = self._hash_function(key)
        self._bucket_delitem(j, key)
        self._n -= 1

    def items(self):
        return [(key, self[key], self.__getitem__(key, get_doc=True)) for key in self]

    def _resize(self, c):
        old = list(self.items())
        self._table = c * [None]
        self._n = 0
        for (k, v, doc) in old:
            self.setitem_(k, v, set_doc=doc)

    def _bucket_getitem(self, j, key, get_doc=False, trav=False):
        bucket = self._table[j]
        # if bucket is None:
            # raise KeyError('Key Error: ' + repr(key))
        return bucket.__getitem__(key, get_doc=get_doc, trav=trav)

    def _bucket_setitem(self, j, key, value, set_doc):            # j = hash code
        if value in self:
            for (k, v, d) in self.items():
                if d != set_doc and value == v:
                    self.__getitem__(k, trav=True).doc_list.append(set_doc)
            return

        if self._table[j] is None:
            self._table[j] = UnsortedTableMap()
        oldsize = len(self._table[j])
        self._table[j].__setitem__(key, value, set_doc=set_doc)
        if len(self._table[j]) > oldsize:
            self._n += 1

    def _bucket_delitem(self, j, key):
        bucket = self._table[j]
```

28

```python
            if bucket is None:
                raise KeyError('Key Error: ' + repr(key))
            del bucket[key]

    def traverse(self):
        for i in self:
            yield self[i]

    def __iter__(self):
        for bucket in self._table:
            if bucket is not None:
                for key in bucket:                              # chaining
                    yield key


if __name__ == '__main__':
    import os, re
    stopwordsSCHashST = SCHashST()
    fileQueue = LinkedQueue()

    fp = open("StopWords.txt", '+r')
    for line in fp.readlines():
        key = (line.rstrip('\r\n'))
        fileQueue.enqueue(key)
    fp.close()

    i = 0
    for q in fileQueue:
        stopwordsSCHashST.setitem_(i, q.element, "ahmadPanah")
        i += 1

    print("---------------------------------------")
    print("SCHashST")
    print("---------------------------------------")
    # Trie Search
    words_tree = SCHashST()
    counter = 0
    for subdir, dirs, files in os.walk("/home/maometto/Documents/black"):
        for _file in files:
            if _file.endswith('.txt'):
                #self.files_list.append((str(_file)))
                fp = open(os.path.join(subdir, _file), 'r+')
                DATA = fp.read().replace('\n', ' ')
                for value in re.findall(r"[\w']+", DATA):
                    if value not in stopwordsSCHashST:
                        words_tree.setitem_(counter, value, _file)
                        counter += 1
                fp.close()

    print(counter)

    for i in words_tree:
        print words_tree.__getitem__(i, get_doc=True)
```

```python
print(len(words_tree))
```