```
import re
import argparse
from pathlib import Path
from os import linesep

BASIC_TOKEN_REGEX = re.compile(r"[\w]*[a-zA-Z0-9_.]*[\w]")

def build_tokenizer(regex=BASIC_TOKEN_REGEX, exclude=None, lower_case=True):
    if exclude is None:
        exclude = set()

    def _tokenize(text):
        tokens = []
        for match in re.findall(regex, text):
            if lower_case:
                match = match.lower()
            if match in exclude:
                continue
            tokens.append(match)
        return tokens

    return _tokenize


def tokenize_text(file_path, encoding='iso-8859-1'):
    tokenize = build_tokenizer()
    with file_path.open("r", encoding=encoding) as fp:
        for line in fp:
            if len(line.strip()) > 0:
                for token in tokenize(line):
                    yield token


def write_to_file(file_path, content):
    with open(file_path, "w") as fp:
        fp.write(content)


class Node(object):
    def __init__(self, key, value, nxt=None):
        self.key = key
        self.value = value
        self.nxt = nxt


class HashTable(object):
    def __init__(self):
        self._size = 4999
        self._T = [None for _ in range(self._size)]
```

```
def _hash(self, text) -> int:
    h = 7
    for c in text:
        h = 31 * h + ord(c)
    return h % self._size

def insert(self, key, value):
    index = self._hash(key)
    if self._T[index] is None:
        self._T[index] = Node(key, value)
    else:
        self._T[index] = Node(key, value, self._T[index])

def delete(self, key):
    index = self._hash(key)
    prev = None
    cur = self._T[index]
    while cur is not None and cur.key != key:
        prev = cur
        cur = cur.child
    if cur is not None:
        nxt = None
        if cur.child is not None:
            nxt = cur.child
        if prev is None:
            self._T[index] = nxt
        else:
            prev.child = nxt
    else:
        print(f'key {key} does not exist')

def increase(self, key):
    node = self._find(key)
    if node is None:
        raise KeyError(f"The key {key} does not exist in the table")
    node.value += 1

def _find(self, key):
    index = self._hash(key)
    if self._T[index] is None:
        return None
    cur = self._T[index]
    while cur is not None and cur.key != key:
        cur = cur.child
    if cur is None:
        return None
    return cur
```

```
    def find(self, key):
        node = self._find(key)
        if node is None:
            return 0
        else:
            return node.value

    def list_all_keys(self):
        keys = []
        for t in self._T:
            if t is not None:
                cur = t
                while cur is not None:
                    keys.append(cur.key)
                    cur = cur.child
        return keys


def create_args_parser():
    parser = argparse.ArgumentParser('Hash table implementation')
    parser.add_argument('file_path', type=str, action='store',
                        help='the path of a text file to be indexed')
    return parser


def main():
    parser = create_args_parser()
    args = parser.parse_args()
    input_file = Path(args.file_path)
    if not input_file.is_file():
        raise FileNotFoundError(f'{input_file} is not found')

    table = HashTable()
    for word in tokenize_text(input_file):
        try:
            table.increase(word)
        except KeyError as e:
            table.insert(word, 1)

    output = linesep.join(f'{word} {table.find(word)}'
                          for word in table.list_all_keys())
    write_to_file(f'./{input_file.name}_table_dump', output)


if __name__ == '__main__':
    main()
```