Loading Dependencies

Loading Data

In [1]:

```
from sklearn.datasets import fetch_20newsgroups
newsgroup = fetch_20newsgroups(subset='all')
```

Importing Libraries

In [2]:

```
import nltk
import time
import string
import numpy as np
import pandas as pd
from tqdm.auto import tqdm
from nltk.tokenize import word_tokenize
```

Preprocessing Dependencies

In [3]:

```
stopwords = nltk.corpus.stopwords.words('english')
stemmer = nltk.stem.SnowballStemmer('english')
lemmatizer = nltk.stem.WordNetLemmatizer()
```

In [4]:

```
def preprocess(article):
    article = word_tokenize(article.lower().strip())
    article = [
        lemmatizer.lemmatize(w)
        for w in article
            if w not in stopwords and w not in string.punctuation
    ]
    return article
```

In [5]:

```
def get_vocabulary(articles):
    vocabulary = set()
    for article in articles:
        for word in article:
            vocabulary.add(word)
    vocabulary = list(vocabulary)
    vocabulary_size = len(vocabulary)
    return (vocabulary, vocabulary_size)
```

BST Dependencies

In [6]:

```
class Node:
    def __init__(self, word, document_id):
        self.word = word
        self.postings = [document_id]
        self.document_frequency = 1

        self.left = None
        self.right = None

    def add_doc_to_word_posting(self, document_id):
        if document_id not in self.postings:
            self.postings.append(document_id)
            self.document_frequency += 1
```

In [7]:

```
class BST:
    def __init__(self):
        self.root = None

    def insert(self, word, document_id):
        if self.root is None:
            self.root = Node(word, document_id)
            return
        else:
            self.insert_word(self.root, word, document_id)

    def insert_word(self, node, word, document_id):
        if word < node.word:
            if node.left is None:
                node.left = Node(word, document_id)
            else:
                self.insert_word(node.left, word, document_id)
        elif word > node.word:
            if node.right is None:
                node.right = Node(word, document_id)
            else:
                self.insert_word(node.right, word, document_id)
        else:
            node.add_doc_to_word_posting(document_id)

    def search(self, word):
        if self.root:
            return self.search_word(self.root, word)
        else:
            return None

    def search_word(self, node, word):
        if word < node.word:
            if node.left is None:
                return None
            else:
                return self.search_word(node.left, word)
        elif word > node.word:
            if node.right is None:
                return None
            else:
                return self.search_word(node.right, word)
        else:
            return (node.postings, node.document_frequency)

#     def inorder(self):
#         if self.root:
#             self.inorder_traversal(self.root)

#     def inorder_traversal(self, node):
#         if node.left:
#             self.inorder_traversal(node.left)
#         print(f'{node.word} {node.postings} {node.document_frequency}')
#         if node.right:
#             self.inorder_traversal(node.right)
```

Creating Dictionaries

In [8]:

```
%%time
preprocessed_articles = list(map(preprocess, newsgroup['data']))
document_ids = list(newsgroup['target'])
```

```
Wall time: 3min 13s
```

Creating BST

In [9]:

```
def create_search_tree(articles, document_ids):
    search_tree = BST()
    for i in tqdm(range(len(articles))):
        article = articles[i]
        document_id = document_ids[i]
        for word in article:
            search_tree.insert(word, document_id)
    return search_tree
```

In [10]:

```
%%time
start_time = time.time()
search_tree = create_search_tree(preprocessed_articles, document_ids)
bst_creation_duration = time.time() - start_time
```

```
100%                                        18846/18846 [00:47<00:00, 393.15it/s]
```

```
Wall time: 46.9 s
```

Creating Hash Table

In [11]:

```
def create_hash_table(articles, document_ids):
    # vocabulary, vocabulary_size = get_vocabulary(articles)
    # hash_table_size = vocabulary_size
    hash_table_size = 100000
    hash_table = [list() for _ in range(hash_table_size)]
    for i in tqdm(range(len(articles))):
        article = articles[i]
        document_id = document_ids[i]
        for word in article:
            hash_idx = hash(word) % hash_table_size
            if not hash_table[hash_idx]: # if spot is empty
                hash_table[hash_idx].append([word, 1, [document_id]])


            else: # if spot is full
                word_present = False # check word is already there. If yes, increment doc f
                for j in range(len(hash_table[hash_idx])):
                    chain_word = hash_table[hash_idx][j][0]
                    if chain_word == word:
                        word_present = True
                        if document_id not in hash_table[hash_idx][j][2]:
                            hash_table[hash_idx][j][2].append(document_id)
                            hash_table[hash_idx][j][1] += 1
                        break

                if not word_present: # word not present in chain
                    hash_table[hash_idx].append([word, 1, [document_id]])

    return hash_table, hash_table_size
```

In [12]:

```
%%time
start_time = time.time()
hash_table, hash_table_size = create_hash_table(preprocessed_articles, document_ids)
hash_table_creation_duration = time.time() - start_time
```

```
100%                                        18846/18846 [00:09<00:00, 2018.24it/s]


Wall time: 10.3 s
```

Querying Dictionaries

In [13]:

```
vocabulary, vocabulary_size = get_vocabulary(preprocessed_articles)
```

Querying BST

In [14]:

```
search_result = search_tree.search("india")
search_result[0]
```

Out[14]:

```
[17, 19, 1, 13, 15, 14, 18, 0, 11]
```

Average Query Time

In [15]:

```
query_times = list()
for word in vocabulary:
    start_time = time.time()
    search_result = search_tree.search(word)
    query_duration = time.time() - start_time
    query_times.append(query_duration)
bst_query_duration = np.mean(query_times)
```

Querying Hash Table

In [16]:

```
def query_hash_table(hash_table, hash_table_size, word):
    hash_idx = hash(word) % hash_table_size
    for chain_word in hash_table[hash_idx]:
        if chain_word[0] == word:
            return chain_word[2]
```

In [17]:

```
search_result = query_hash_table(hash_table, hash_table_size, "india")
search_result
```

Out[17]:

```
[17, 19, 1, 13, 15, 14, 18, 0, 11]
```

In [18]:

```
query_times = list()
for word in vocabulary:
    start_time = time.time()
    search_result = query_hash_table(hash_table, hash_table_size, word)
    query_duration = time.time() - start_time
    query_times.append(query_duration)
hash_table_query_duration = np.mean(query_times)
```

Comparison between BST and Hash Table

In [19]:

```
comparison_df = pd.DataFrame()
comparison_df["Method"] = ['Hash Table', 'BST']
comparison_df.set_index("Method", inplace=True)
comparison_df["Creation Time (seconds)"] = [hash_table_creation_duration, bst_creation_dura
comparison_df["Query Time (μ seconds)"] = [hash_table_query_duration * 10 ** 6, bst_query_d
comparison_df["Memory Size (bytes)"] = [hash_table.__sizeof__(), search_tree.__sizeof__()]
comparison_df
```

Out[19]:

	Creation Time (seconds)	Query Time (μ seconds)	Memory Size (bytes)
Method			
Hash Table	10.310952	2.358792	824440
BST	46.896016	25.425855	32