CME 2201 - Assignment 1

INVERTED INDEX BY USING HASH TABLES

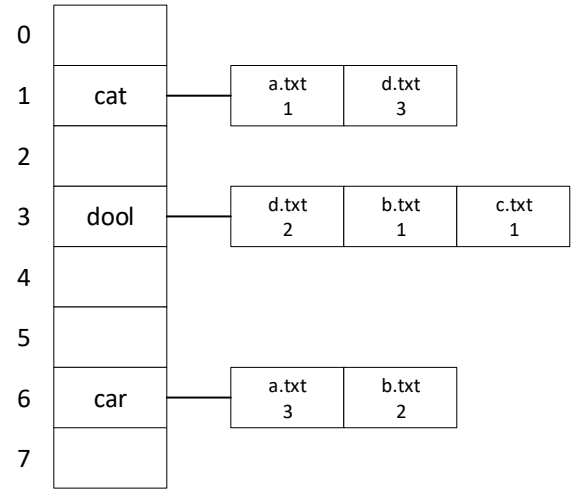
An **inverted index** is an index data structure, which is used to map all documents with their content. It keeps a word and all documents containing this word. There are two types of inverted indexes:

- **Record-level inverted index** contains a list of references to documents for each word.
- **Word-level inverted index** contains the positions of each word within a document.

Inverted index allows fast full text searches and is the most popular data structure used in document retrieval systems, used on a large scale for example in search engines. Its disadvantage is large storage overhead and high maintenance costs on update, delete, and insert.

In this assignment, you are expected to implement a record-level inverted index structure using your own hash table implementation in Java programming language. Your index structure will be used to find all documents that contain a particular word (e.g., return all documents in which "computer" occurs).

To build an inverted index, you should fetch all the documents, ignore any punctuation mark, remove stop words (stop words are the most frequent and useless words in documents, such as “I”, “the”, “we”, “is”, “an”), and then index each document with the remaining words. The structure of the inverted index, implemented with a hash table, should look like as below:



Note: You are free to choose the data structure to store file references.

1. Main Functionalities

- **put(Key k, Value v)**

If a word (k) is already present, then add a reference of the document (v) to the index; otherwise create a new entry. You should store the frequency of each word with the document identifier.

- **Value get(Key k)**

Search the given word (k) in the hash table. If the word is available in the table, then return an output as shown below, otherwise return a “not found” message to the user.

>Search: cat 2 documents found 3-d.txt 1-a.txt	> Search: dool 3 documents found 2-d.txt 1-b.txt 1-c.txt	> Search: ball Not found!

- **remove(Key k)**

Remove the given word (k) and the associated value from the inverted index.

- **resize(int capacity)**

Make the hash table dynamically growable. *Put* method should double the current table size if the hash table reach the maximum load factor.

2. Hash Function

To specify an index corresponding to given string key, firstly you should generate an integer hash code by using a special function. Then, resulting hash code has to be converted to the range 0 to N-1 using a compression function, such as modulus operator (N is the size of hash table).

You are expected to implement two different hash functions including simple summation function and polynomial accumulation function.

2.1. Simple Summation Function (SSF)

You can generate the hash code of a string *s* with the length *n* simply by the following formula:

$$\mathrm{h(s)} = \sum_{k=0}^{n-1} ch_k$$

2.2. Polynomial Accumulation Function (PAF)

The hash code of a string *s* can also be generated by using the following polynomial:

$$\mathrm{h(s)} = ch_0 * z^{n-1} + ch_1 * z^{n-2} + \ldots + ch_{n-2} * z^1 + ch_{n-1} * z^0$$

where ch_0 is the left most character of the string, characters are represented as numbers in 1-26 (case insensitive), and *n* is the length of the string. The constant *z* is usually a prime number (31, 33, 37, and 41 are particularly good choices for working English words). When the *z* value is chosen as 31, the string "car" has the following hash value:

$$h(\text{car}) = 3 * 31^2 + 1 * 31 + 18 * 1 = 2932$$

Note: Using this calculation on the long strings will result in numbers that will cause overflow. You should ignore overflows or use Horner's rule to perform the calculation and apply the modulus operator after computing each expression in Horner's rule.

3. Collision Handling

3.1. Linear Probing (LP)

Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell.

3.2. Double Hashing (DH)

Double hashing uses a secondary hash function *d(k)* and handles collisions by placing an item in the first available cell of the series.

$$d(k) = q - k \bmod q$$

$$h_2(k) = \big(h(k) + j\, d(k)\big) \bmod N$$

where $q < N$ (table size), q is a prime, and $j = 0, 1, \ldots, N-1$.

The secondary hash function *d(k)* cannot have zero values. The table size N must be a prime to allow probing of all the cells.

Example:

N = 13, k= 31, q= 7, h(k) = k mod 13 = 5, d(k) = 7 - k mod 7 = 4.	The 1st lookup index: 5 The 2nd lookup index: 5+ 1*4 = 9 % 13 = 9 The 3rd lookup index: 5+2*4 = 13 % 13 = 0 …

4. Performance Monitoring

You are expected to fill the performance matrix (Table 1) by running your code under different conditions including two different load factors (50% and 80%) to decide resizing of hash table, two different hash functions (SSF and PAF) and two different collision handling techniques (LP and DH).

You should count total number of collision occurrences and calculate spent time while loading documents into the inverted index structure under each condition. In addition, you should calculate min., max. and avg. search times by using the “search.txt” file that contains 1000 words to search (Search time means the time expended to find a particular key in the hash table. It does not include the time spent for outputs. To calculate avg. search time, divide the total expended time to the total number of searched keys). You can use System.nanoTime() or System.currentTimeMillis() for time operations.

Load Factor	Hash Function	Collision Handling	Collision Count	Indexing Time	Avg. Search Time	Min. Search Time	Max. Search Time
α=50%	SSF	LP					
		DH					
	PAF	LP					
		DH					
α=80%	SSF	LP					
		DH					
	PAF	LP					
		DH					

Table 1. *Performance matrix*

Provided Resources

- English stop-words lists (stop_words_en.txt)
- Delimiters to split document content (delimiters.txt)
- Documents to index: BBC news article datasets [1] (bbc.rar)
- Word list to use in calculation of searching times (search.txt)

Due date

05.12.2021 Sunday 23:59. Late submissions are not allowed.

Requirements

- Usage of Java programming language and generic data types are required.
- You need to implement base functions of a classical Hash Table by yourself (do not extend an available Java Hash Map class directly).
- Object Oriented Programming (OOP) principles must be applied.
- Exception handling must be used when it is needed.

Submission

You must upload your all '.java' files as an archive file (.zip or .rar). Your archived file should be named as 'studentnumber_name_surname.rar.zip', e.g., 2007510011_Ali_Yılmaz.rar and it should be uploaded in the SAKAI portal.

Prepare and upload a report with descriptions of your data structure, java code, and performance matrix.

You can ask your questions from the "FORUM -> Homework 1 - Questions" part of the SAKAI portal.

Plagiarism Control

The submissions will be checked for code similarity. Copy assignments will be graded as zero, and they will be announced in the SAKAI portal.

Grading Policy

Job	**Percentage**
Usage of Generic, OOP and Try-Catch	%30
Inverted index implementation	%50
Performance monitoring	%20

References

[1] D. Greene and P. Cunningham. "Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering", Proc. ICML 2006.