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.
-

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 length *n* simply by the following formula:

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

2.2. Polynomial Accumulation Function (PAF)

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

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

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 (33, 37, 39, and 41 are particularly good choices for working English words). When the *z* value is chosen as 33, the string "car" has the following hash value:

$$h(\text{car})=3*33^2+1*33+18*1=3318$$

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.

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

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

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

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 measure expended 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).

Load Factor	**Hash Function**	**Collision Handling**	**Collision Count**	**Indexing Time**	**Avg. Search Time**	**Min. Search Time**	**Max. Search Time**
α=50%	SSF	LP	389506	446,0376212sec	0,17738319sec	0,0152sec	1,9597sec
		DH	389749	437,8883125sec	0,18107920sec	0,0156sec	2,6633sec
	PAF	LP	187572	436,4981319sec	0,0781948sec	0,0171sec	3,0881sec
		DH	125163	429,9088289sec	0,05854239sec	0,0183sec	1,5236sec
α=80%	SSF	LP	409578	446,3789001sec	0,16541799sec	0,0211sec	2,6235sec
		DH	383274	429,3602417sec	0,16915910sec	0,0095sec	1,2965sec
	PAF	LP	161699	437,9107955sec	0,24888079sec	0,0161sec	4,3262sec
		DH	132665	411,287048sec	0,0525492sec	0,0118sec	1,2674sec

Table 1. *Performance matrix*