CME 2201 - Assignment 1

In this assignment, you are expected to index words of a document named as 'story.txt'. You must read this file, split it word by word, and index each word to your hash table according to rules given below.

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.

1. Main Functionalities

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

Read the given input story.txt file, calculate the number of occurrences of each word as *count* value, insert this count data into the hash table accordingly.

- **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. When a word is searched in the hash table; key, count, and index of the word should be printed.

------ Output ------

Search: Ezgi
Key: 1243225
Count : 10
Index : 165

Search: Ali
Key: 68294842
Count : 3
Index : 132

Note: Results of the words of "Ezgi" and "Ali" were generated as an example. So, you will not obtain the same results by using the 'story.txt' file.

- **resize(int capacity)**

Make the hash table dynamically growable. The *put* method should double the current table size if the hash table reaches the maximum load factor. You should take the initial size of the table as 997 and call the *resize* method according to two different load factor values (50% and 70%).

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 polynomial accumulation function and your own hash function.

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

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

Note: Using of 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.

Your Own Hash Function (YHF)

Hash code for converting each word to an integer key must be implemented by yourself. The input value will be the word and the integer key will be returned by your hash code function. The hash (compression) function for converting a key to the index (address calculator) must be implemented by yourself.

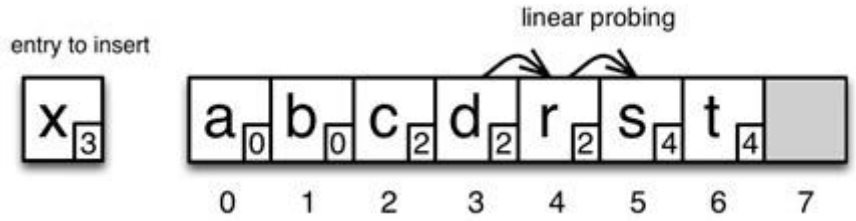
3. Collision Handling Approach

You are expected to implement a collision resolution technique based on open addressing. The insertion algorithm is as follows:

- Calculate the hash value and initial index of the entry to be inserted.
- Then search the position linearly.
- While searching, the distance from initial index is kept which is called DIB (Distance from Initial Bucket).
- If we can find the empty bucket, we can insert the new entry with the DIB value in here.
- If we encounter an entry which has less DIB than the candidate entry, swap them.

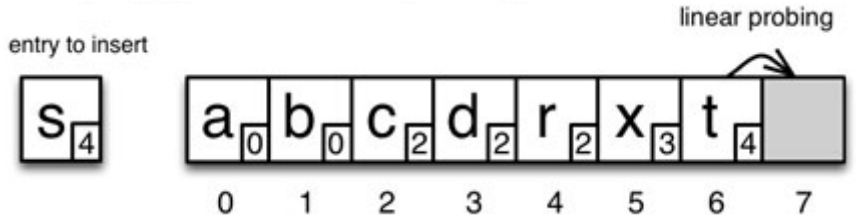
Example:

Step 1(a): Linear probing from the initial index to find a swap candidate



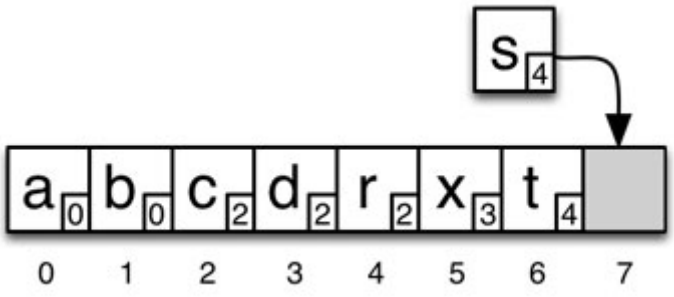
- The box at the bottom right shows the initial index value.
- The initial index is bucket 3, because mod(hash(x),size) = 3.
- The distances to the initial indexes, denoted as DIB(), are compared with a linear probing. Entries are swapped if and only if the distances are satisfying this condition:
  DIB(current entry) < DIB(entry to be inserted)
- At bucket 3, DIB(d) = 3-2=1 and DIB(x) = 3-3=0, thus the entry x cannot be inserted in here, and the linear probing continues.
- At bucket 4, DIB(r) = 4-2=2 and DIB(x) =4-3=1, the linear probing continues.
- At bucket 5, DIB(s) = 5-4=1 and DIB(x) =5-3=2, and since DIB(s) < DIB(x), the entry x is now inserted at the index of the entry s; and the entry s becomes a new entry to be inserted.

Step 1(b): The linear probing continues with the entry that was swapped out (entry s)



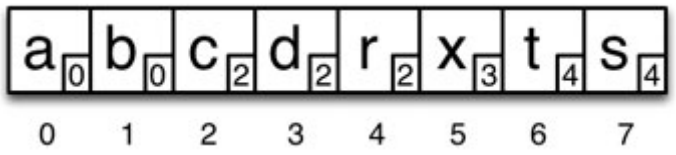
- The linear probing goes on, starting the bucket after the one where swap occurred, which means bucket 6.
- At bucket 6, DIB(t) = 6-4=2 and DIB(s) = 6-4= 2, thus the linear probing continues.
- At bucket 7, we find an empty bucket, thus we can insert entry s in here.

Step 2: The entry is inserted in an empty bucket



- An empty bucket was found, thus the current entry s is stored there, and the insertion process can stop.

Step 3: Final state after displacements and insertion



For entry retrieval, entries can be found using linear probing starting from their initial indexes, until they are encountered, or until an empty bucket is found, in which case it can be concluded that the entry is not in the table. The search can also be stopped if during the linear probing of a bucket is encountered for which the distance to the initial bucket is smaller than the DIB of the entry it contains.

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 70%) to decide resizing of hash table and two different hash functions (PAF and YHF).

You should count total number of collision occurrences and measure expended time while indexing words in the "story.txt" under each condition. In addition, you should calculate minimum, maximum and average search times by using the “search.txt” file that contains 100 words to search for (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 Count	Indexing Time	Avg. Search Time	Min. Search Time	Max. Search Time
α=50%	PAF					
	YHF					
α=70%	PAF					
	YHF					

Table 1. *Performance matrix*

Provided Resources

- Document to index: story.txt
- Word list to use in calculation of searching times: search.txt

Due date

December 16, 2020, 23:55

Submission

You must upload your all '.java' files as an archive file (.zip or .rar) to the Sakai platform. Your archived file should be named as 'studentnumber_name_surname.rar.zip', e.g., 2007510011_Ali_Yılmaz.rar.

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

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.

Grading Policy

Job	**Percentage**
Usage of Generic, OOP and Try-Catch	%20
Implementation of hash operations and collision handling approach	%60
Performance monitoring	%20