DOKUZ EYLUL UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CME2201

DATA STRUCTURES

ASSIGNMENT 1

REPORT

by

Ecem Şevval Çınar

2018510078

Lecturers

Assistant Professor Zerrin Işık
Research Assistant Ali Cüvitoğlu
Research Assistant Ezgi Demir Karaman

IZMIR

14.12.2020

CHAPTER ONE

PROGRESS DESCRIPTION

In this assignment, words in a document names 'story.txt' got indexed. File has been read word by word. Then, words have been inserted into Hash Table data type.

Requirements

1. Usage of Java programming language and generic data types
2. Base functions/methods of a Hash Table
3. Object Oriented Programming
4. Exception handling

CHAPTER TWO

TASK SUMMARY

2.1 Completed Tasks

- Main functions such as "put, get and resize" have been implemented.
- Hash Function (h2(h1(key)))
    - Hash Code (h1: keys → integers)
        - Polynomial Accumulation Function (PAF) (given in assignment)
        - My Hash Code Function
    - Compression Function (h2: integers → index [0, N-1])
        - Division (given in assignment)
        - My Compression Function
- Collision Handling using "distances to the initial indexes (DIB)"
- Performance Matrix

CHAPTER THREE

EXPLANATION OF ALGORITHMS

3.1 Functions

1.Hash Function Given in Assignment:

```
public int hashCodeFunction(K key) { // h1(x) --> Polynomial Accumulation Function (PAF)
    String keyy= key.toString().toLowerCase(Locale.ENGLISH); // to lowercase

    int z = 37; // prime number i used
    int poly[] = new int[keyy.length()]; // for horner
    int n = keyy.length();
    for (int i = 0; i <n; i++) {
        poly[i] = (int)  Math.abs((int) keyy.charAt(i) - 96);
    }

    // i use horner's rule to handle overflows
    int polyNumber = poly[0];
    for (int i = 1; i < n; i++) {
        polyNumber = polyNumber*z + poly[i];
    }
    polyNumber = Math.abs(polyNumber); // math.abs() is used to prevent negative values
    return polyNumber;
}
```

- This is the first part of Hash Function that calculates the Hash Code. More precisely, the part where the keys are converted to integers.
- Horner's rule has been used to handle the overflows instead of ignoring.
- Alphabetical values of words have been found using their ASCII codes.
- Then, Math.abs() has been used to prevent negative values. (For example "51" has negative hash code. When it comes to calculating the index using "Division" compression function, if it is processed negatively, the index will be negative. But negative index does not exist!! Therefore, it will be give "Index Out of Bounds" exception. Therefore, it has been prevented with taking the absolute .)

```
int index= hashCodeFunction(key)%size;
return index;
```

- Then, hash code has been converted to the range [0,N-1] using this compression function. (Division function → using modulus operator)

2.My Hash Function

Hash code function for converting each word to an integer key:

```
public int myHashCodeFunction(K key) { // h1(x) --> implement by me:)
    String keyy = key.toString().toLowerCase(Locale.ENGLISH);
    int z =37; // prime number
    int hashCode=0;
    int w = keyy.length(); // daha fazla collison olmamasi icin sonradan bunu ekledim.
    // collison sayisi yari yariya azaldi fakat yine de PAF kadar etkili degil
    for (int i = 0; i < keyy.length(); i++) {
        if(i%2==0) {

            hashCode += z* (int)  Math.abs((int) keyy.charAt(i) - 96) *w ;
        }
        else if(i%2==1){
            hashCode -= z* (int)  Math.abs((int) keyy.charAt(i) - 96) *w;
        }
    }
    hashCode = Math.abs(hashCode); // negatif cikmamasi icin
    return hashCode;
}
```

- Firstly, keys have been converted to lower case. Then, their alphabetical values have been found using ASCII codes. (For example, “a” is “97” in ASCII Table. → 97-96=1)
- Their alphabetic values have been multiplied by randomly selected prime number. Then, it is again multiplied by its length. Then, addition and subtaction have been done, respectively.
- Again, Math.abs() has been used to prevent negative values.

```
private int findPrimeforMAD(int size) {
    // table size degistiginde, MAD fonksiyonunda kullan
    // cunku asal sayi > table size olmali...
    prime = size+1; // bu sekilde asal sayimin table siz
    while(true) {
        int c=0;
        for (int i = 1; i <= prime; i++) {
            int k = prime%i;
                if(k==0)
                    c++;
            }
            if(c!=2)
                prime++;
            else if(c==2)
                break;
        }
    return this.prime=prime; // guncellendi
}
public int findHashIndex(K key) { // h2(h1(x)) --> compr
    // h2 (y) = ((ay + b) mod P) mod N
    // P is a prime number larger than N
    // a and b are nonnegative integers chosen randomly

    //kendi yazdigim compression function, MAD (Multiply
                                                //deneme

    int a = 13; // prime number i used for a
    int b=11; // prime number i used for b
    int y = myHashCodeFunction(key);
    int index = Math.abs(((a*y + b) % prime)) % size;


    //int index= hashCodeFunction(key)%size;  // THIS IS
    return index;
}
```

- For my compression function, “Multiply, Add and Divide (MAD)” has been implemented. “a” and “b” prime numbers has been selected randomly by me.
- With “findPrimeforMAD” function, prime number for modulus operation has been found. I cannot select this prime number myself. Because, after resizing process, the

size of the hash table changes. Therefore, the number must be also changed. So, it is necessary to update the prime number.

- Lastly, MAD function identifies the index.

3.Collision Handling Approach

```
public void put(K key, V value) {
    int currentCapacity = size();
    int tableLength = table.length;
    int loadFactor = (int)(currentCapacity*10/tableLength);

    // kelimeyi yerlestirmeden once resize yapmam gerekiyor mu diye kontrol ediyorum.... sonrasinda kelimeyi yerlestiriyorum.
    if(loadFactor>=5) { // 0.7'ye gore de resize yazilabilir, degistirmek icin 5 yerine 7 yazin
        resize();
        //System.out.println("Resized...");
    }

    int hashIndex = findHashIndex(key); // h2(h1(x)) process

    HashEntry<K,V> newHash = new HashEntry<K,V>(key,value);
    int DIB=0;
    while(true) {
        if(table[hashIndex]==null) { // if null, (key,word) can be inserted without any problem
            table[hashIndex] = new HashEntry<K,V>(newHash.getKey(),newHash.getValue());
            break; // break the loop, because insertion is succeed
        }
        else if(table[hashIndex]!=null && table[hashIndex].getKey().toString().equals(newHash.getKey())) { // if same word is added again, value(count) increases
            table[hashIndex].setValue((V) String.valueOf((Integer.parseInt(table[hashIndex].getValue().toString()) + 1)));
            break; // break the loop
        }
        int DIB2 = hashIndex - findHashIndex(table[hashIndex].getKey()); // bos olmayan indexteki datanin dib degeri
        if(DIB>DIB2) {
            collisionCount++;
            // yerlestirecegim yerdeki datayi kaybetmemek icin ilk olarak tempe atiyorum
            HashEntry<K,V> temp= new HashEntry<K,V>(table[hashIndex].getKey(),table[hashIndex].getValue());
            table[hashIndex]=null; // this is not necessary
            table[hashIndex] = newHash; // inserted
            newHash = new HashEntry<K,V>(temp.getKey(),temp.getValue()); // artik yerlestirmem gereken data, biraz once tempe attigim yani bulundugu yerden cikardigim entry
                                                                         // sirada onu yerlestirmek var
            DIB = hashIndex - findHashIndex(newHash.getKey()); // dib hesaplama
        }
        DIB++; // girmesi gereken indexten 1 uzaklasacagindan dib degerini de artiriyorum
        hashIndex++; // bir sonraki hashe bakiyorum
        hashIndex = hashIndex % table.length; // out of bounds vermemesi icin, mod aliyorum. eger hashIndex=table.length olmussa, bu islem sonucu hashIndex=0 oluyor
    }
}
```

- Firstly,it has been checked using load factor if “resize” process is required. Then, index calculated using Hash Functions.
- “HashEntry<K,V> newHash” variable has been created to keep the “key and value” which has to be inserted to Hash Table.
- With while loop, index position has been searched linearly. While searching, DIB value of “newHash” is kept.
- If empty bucket (index) is find, newHash can be inserted to the empty index directly. (Then break the loop, because insertion is succeed.)
- If index is not empty but the key in this index equals to the newHash’s key, that means its value (count of key in txt file) has to be increased by one. (Then break the loop, because insertion is succeed.)
- If it does not satisfy the above conditions, it does not enter into the “if” code. Then, DIB value is calculated for the entry that exists in index. If it has less DIB than the candidate entry (that means DIB(1)>DIB2), it swaps. But before that, I keep the entry in a temp entry not to lost it. After the swapping, the new candidate becomes temp. (newHash= temp line) The loop returns to the beginning and it countinues until the word is inserted the Hash Table.

4.Resize

```
private void resize() {
    //int oldSize = capacity;
    int newSize =  table.length*2;
    while(true) {
        int count=0;
        for (int i = 1; i <= newSize; i++) { // Checking whether newsize is prime or not
            int kalan = newSize%i;
            if(kalan==0) count++; //  zero means divided
        }
        if(count!=2) // it's not a prime number, I have to increase the number and check again
            newSize++;
        else if(count==2) //  prime number has 2 divisors. itself and 1. if found,
            break; // break the loop
    }

    // yeni size bulundu, sirada yeni hashtable olusturup eskisindekileri atmak var.

    this.size=newSize;      // size guncellendi, hash func ve hashtable olusturmak icin de guncellemis oldum
    findPrimeforMAD(size); // size guncellenince, prime number da guncellendi.... ( kendi hash fonksiyonum icin)
    tempTable = new HashEntry[size];
    for (int i = 0; i < table.length; i++) {
        if(table[i]!=null) {
            tempTable[i]= table[i]; // onceden yerlesenleri tempe attim
        }
    }
```

- Beginning of the put(key,value) method, it has been checked using load factor if “resize” process is required.
- If it is required, this function performs. Firstly, table size is multiplied by two to determine the new table size. Since length of table must be chosen as prime, it is checked whether it is prime, with the while loop. If it is not, with this while loop, it can be found.
- Before the updating of hash table, entries in there must be kept in a temp table.

```
table = new HashEntry[size]; // mevcut table'in size'i guncellendi
// simdi temptekileri, guncellenen size'li hash functiona sokup, table'a aticam
for (int i = 0; i < tempTable.length; i++) {
    if(tempTable[i]!=null) {
        int newIndex = findHashIndex(tempTable[i].getKey());
        HashEntry<K,V> newHash = new HashEntry<K,V>(tempTable[i].getKey(),tempTable[i].getValue());
        int DIB=0;
        while(true) {
            if(table[newIndex]==null) {
                table[newIndex] = new HashEntry<K,V>(newHash.getKey(),newHash.getValue());
                break;
            }
            collisionCount++;
            int DIB2= newIndex - findHashIndex(table[newIndex].getKey()); // mecvut olarak bulunanin dib hesap
            if(DIB>DIB2) { // dib yerlesir, dib2 olan tempe atilir
                HashEntry<K,V> temp = new HashEntry<K,V>(table[newIndex].getKey(),table[newIndex].getValue());
                table[newIndex]=null;
                table[newIndex] = newHash;
                newHash = temp;
                DIB = newIndex - findHashIndex(newHash.getKey());
            }
            DIB++;
            newIndex++;
            newIndex = newIndex%table.length;

        }
    }
}
```

- Then, table size has been updated. Entries that has been kept in temp table is inserted back to the main table. Here again, the entries have been placed according to their DIB value.

3.2 Description of Data Structure

Hash Table have been used to index the words. It is also desired to Hash Table data type has to be generic.

```
HashTable<String,Integer> myHashTable = new HashTable<String,Integer>();
```

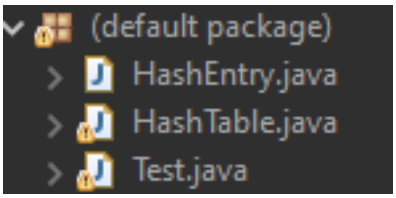
In main class, I've created the hastable as above. Generics are to allow type to be a parameter to methods and classes. So, with generics, it is possible to create classes that work with different data types. That is why all the methods and functions in the Hash Table class have been written this way. Thus, any non null object can be used as a key or a value.

For example: put(K key, V value) , getInfo(K key) …

In this assignment, it can be understood that, key → String and value → Integer.

Words in given file have been stored as string(key) type in Hash Table and their counts have been stored as integer(value).

3.1 Classes



In this assignment, I have only used 3 classes. HashEntry class is necessary for generic. HashTable class has essential methods and functions of project. Test class is the class in which the given word file is read and Project is tested.

CHAPTER FOUR

PERFORMANCE MONITORING

Load Factor	Hash Function	Collision Count	Indexing Time	Avg. Search Time	Min. Search Time	Max. Search Time
α=50%	**PAF**	5951	76326800 nano sec. or 76 milli sec.	2050 nano seconds	1200 nano seconds	2800 nano seconds
	YHF	130920	98895600 nano sec. or 98 milli sec.	4271.43 nano seconds	1500 nano seconds	11400 nano seconds
α=70%	**PAF**	10287	85721500 nano sec. or 85 milli sec	2117.86 nano seconds	1100 nano seconds	3100 nano seconds
	YHF	131408	94188400 nano sec. or 94 milli sec.	4717.86 nano seconds	1200 nano seconds	14100 nano seconds

Table 1. Performance matrix

Load Factor Preference

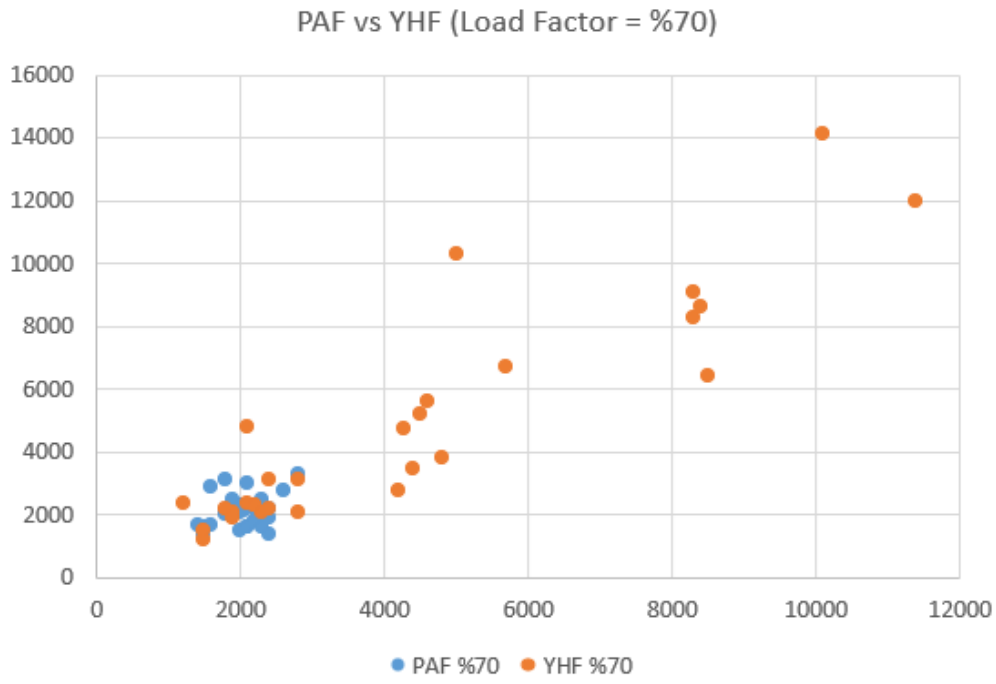
It is understood that load factor %50 causes less collisions. (When load factor is %50, resize function performs 3 times. In the end, length of table equals to 8009.) Since taking the load factor as %70 will cause more collisions, indexing time will also increase. (When load factor is %70, resize function performs 2 times. In the end, length of table equals to 4001.) Therefore, it is better to resize the Hash Table when it is half full.

Hash Function Preference

The main reason of collision count is Hash Functions design. Hash code function which I designed to convert keys to integer, has basic addition, subtraction and multiplication operations. However, Polynomial Accumulation Function (PAF) has more complicated process. So, the same hash codes are less likely to appear when PAF is used.

As compression function, MAD has been used in my hash function, but I do not think this is the reason why the indexing takes much time and collisions occurred more. (I have already calculated the time spent for indexing while using PAF for hash code calculations and MAD for converting

hash code to index. There are 4713 collisions occurred.) Hence, I can understand that PAF is more beneficial as a hash function.



Graph [1]

Search Time

The more keys with same hash codes, the more collisions there will be while inserting. (It is because same hash codes have same index values and same index values mean collision.) Likewise, the time spent, to find keys diverge from the index that should have been the result of the collision, will increase. (It can be understood from the graph [1].) Also, time spent while searching depends on the current performance of the computer. Therefore, it may be different each time while searching the time key word.