CS 223 : Data Structure II

Perfect Hashing

Hisham Osama Ghazy (81)
Ahmed Moustafa El-Naggar (16)

Problem Statement

In this assignment, you're required to implement a perfect hashing data structure. We say a hash function is perfect for S if all lookups involve O(1) work. In section 2, background about universal hashing is provided. Sections 3 and 4 describe two methods for constructing perfect hash functions for a given set S. You're required to design, analyze and implement a perfect hash table as described in sections 3 and 4.

Implementation Details

Both versions of perfect hash tables ($O(n^2)$, $O(n)$) are implemented using the same class : PerfectHashTable. For choosing one of the two available versions, a builder class following the builder design pattern is used for that purpose with two static factory methods available:

1) buildSquareSpaceHashTable : returns a perfect hash table that uses $O(n^2)$ space.
2) buildLinearSpaceHashTable : returns a perfect hash table that uses O(n) space.

Hash functions used in hashing are universal hash functions.

PerfectHashTable Class:

```
4
5  public class PerfectHashTable {
6
7      private AbstractHashFunction hashFunction;
8      private PerfectHashTable[] hashTables;
9      private int[] values;
10
11     public PerfectHashTable() {
12
13     }
14
15     public void setHashFunction(AbstractHashFunction hashFunction) {
16         this.hashFunction = hashFunction;
17     }
18
19     public void setValues(int[] values) {
20         this.values = values;
21     }
22
23     public void setHashTables(PerfectHashTable[] hashTables) {
24         this.hashTables = hashTables;
25     }
26
27     public boolean search(int key) {
28         if(key <= 0){
29             return false;
30         }
31         int index = hashFunction.getHashIndex(key);
32         if (hashTables[index] == null) {
33             return values[index] == key;
34         }
35         return hashTables[index].search(key);
36     }
37
38 }
```

Attributes:

a) hashFunction : the hash function to be used.

b) hashTables : the second level of hash tables in case of collisions.

c) Values : array storing the keys in case of no collisions in some table cells.

Abstract Hash Function:

```
import java.util.ArrayList;

public abstract class AbstractHashFunction {

    protected ArrayList<Integer> parameters;
    protected static int FIXED_PARAMETERS_SIZE;

    public AbstractHashFunction(ArrayList<Integer> parameters) {
        this.parameters = parameters;
    }

    public abstract int getHashIndex(int key);
}
```

Attributes:

a) Parameters: list of the parameters needed for any hash function
b) FIXED_PARAMETERS_SIZE : size of the parameters list needed for any hash function

Matric Hash Function:

```
import java.util.ArrayList;
import java.util.Random;

import hashing.hash.function.utils.AbstractHashFunction;

public class MatrixHashFunction extends AbstractHashFunction {

    private int hashTableSize;
    private int keyLength;
    private int[][] matrix;

    public static void main(String[] args) {..

    public MatrixHashFunction(ArrayList<Integer> parameters) {..

    private void generateRandomMatrix() {..

    private int[] getKeyMatrix(int key) {..

    private int[] multiplyMatrices(int[][] matrixA, int[] matrixB) {..

    private int convertMatrixToNumber(int[] matrix) {..

    public int getHashIndex(int key) {..

}
```

$$hx = k$$

Attributes:

a) hashTableSize : the log of the size of the hash table.
b) keyLenght : the number of bits in the input keys.
c) Matrix : the matrix to be multiplied by the key matrix.

UniversalLinearHashFunction

```
import java.util.ArrayList;

public class UniversalLinearHashFunction extends AbstractHashFunction {

    private int aFactor;
    private int bFactor;
    private int hashTableSize;
    private int primeNunber;

    public UniversalLinearHashFunction(ArrayList<Integer> parameters) {

    @Override
    public int getHashIndex(int key) {
        return ((aFactor * key + bFactor) % primeNunber) % hashTableSize;
    }

    public static void main(String[] args) {

}
```

Equation = $((ak + b) \bmod p) \bmod m$

Attributes:

a) primeNumber : prime number greater that all input keys.
b) hashTableSize : the size of the hash table.

PerfectHashTableBuilder

```
public class PerfectHashTableBuilder {

    private final static int MAX_ITERATIONS = 10;
    private final static int MAX_HASH_TABLE_SIZE_FACTOR = 4;
    private final static int PRIME_LOWER_BOUND = 100000;

    private static ArrayList<Integer> getHashFunctionParameters(int m, int p) {
    public static PerfectHashTable buildSquareSpaceHashTable(int[] keys) {
        PerfectHashTable table = new PerfectHashTable();
        int squaredSpace = keys.length * keys.length;
        for (int i = 0; i < MAX_ITERATIONS; i++) {
            UniversalLinearHashFunction hashFunction = new UniversalLinearHashFunction(
                    getHashFunctionParameters(squaredSpace, PrimeNumberGenerator.getPrimeNumber(PRIME_LOWER_BOUND)));
            int[] hashedValues = new int[squaredSpace];
            boolean collisionOccured = false;
            for (int key : keys) {
                int hashedIndex = hashFunction.getHashIndex(key);
                if (hashedValues[hashedIndex] != 0 && hashedValues[hashedIndex] != key) {
                    collisionOccured = true;
                    break;
                }
                hashedValues[hashedIndex] = key;
            }
            if (!collisionOccured) {
                table.setHashFunction(hashFunction);
                table.setValues(hashedValues);
                PerfectHashTable[] hashTables = new PerfectHashTable[squaredSpace];
                table.setHashTables(hashTables);
                return table;
            }

        }
        return table;
    }

    public static PerfectHashTable buildLinearSpaceHashTable(int[] keys) {
```

Static Factory methods for Hash Tables creation.

Test Cases

In the Square-Sized Hashing:

In most of the test cases, **no collisions** have occurred in a table of $O(N^2)$ space.

In few cases, the no collision cases have been achieved after rebuilding the hash table 2 or 3 times.

Rebuilding the hash tables many times has occurred in the cases where the number of keys is relatively small ($\simeq 10 \sim 20$ keys)

In the Linear-Sized Hashing

By calculating $\Sigma (Ni)^2$ where Ni is the number of collided elements in the ith row of the $O(N)$ space table

File Name	$\Sigma (Ni)^2$
keys.txt	15
Keys1001000.txt	100
Keys10001000.txt	750
Keys100001000000.txt	10000

I.e.: This results are obtained after building the linear spaced table **10 times** and taking the average value of $\Sigma\,(Ni)^2$

Assumptions

- Duplicates keys are ignored and only **one value** of the key is taken into consideration.
- No negative keys are allowed.
- In $O(N)$ spaced hash table, several trials are made until $\Sigma\,(Ni)^2$ value is less than $4N$.