CS 3310 Data and File Structures

Instructor Dr. Ajay K. Gupta

Western Michigan University

Grad TA: Rajani Pingili

Anfaal FaisaL

SOFTWARE LIFE CYCLE REPORT – FOR ASSIGNMENT

PHASE 1: SPECIFICATION

1. The main goal of this assignment was to empirically and analytically compare different data hashing algorithms and the time complexity.
2. Implement a several hashing algorithms
3. Implement three probing algorithms (linear, pseudo-random, and double-hashing)
4. Practice good coding conventions

The application had to do the following:

1. Read and convert data from text file
2. Create n required bags with different hashing and probing algorithms
3. Fill each bag with 125 random elements from main data holder
4. Find random element in each bag
5. Write average time for each bag and each type of bag
6. Print internal data of bag and time values - raw data and average

PHASE 2: DESIGN

1. Java Project
    a. App
        i. **public** App**(Item[]** itemsArray**, int** n) Construct application class and internal bag
        ii. **public static void** main(**String[]** args) Entry point of program
        iii. **private static Item[]** readFile() Raed data from file an return it as array
        iv. **private void** search(**Item[]** itemsArray) Search random value on bags and save time of each search
    b. Bag
        i. **public** Bag**(Item[]** items**, boolean** isOpenHashing**, int** hashingType**, int** probingType**, int** numberBag) Constructor. Create bag with selected type of hashing and probing
        ii. **private int** getHash(**Item** item**, int** hashingType) Calculate hash of Item object with selected type of hashing
        iii. **public String** print(**boolean** fullType) Made short or long text representation of bag
        iv. **public int** getNumber() Getter of bug number
        v. **public LinkedList<Integer>** find(**Item** item) Search Item in bug, using hash
        vi. **public LinkedList<Integer>** getStrengths(**LinkedList<Integer>** res, **Item** item) Find and return list of strengths of selected items.
    c. Item
        vii. **private** Item() Private empty constructor
        viii. public Item(String line) Constructor, which create item, using string data
        ix. **public String** getName()
        v. **public int** getMinimumStrenght()
        vi. **void** resetPointer() public String getRarity()
        vii. **public** int getCurrentStrenght()
        viii. **public void** setCurrentStrenght(**int** currentStrenght)
        ix. **public Item** clone() throws CloneNotSupportedException Clone item with exact vales
        x. **public String** toString()
        xi. **public boolean** equalsName(Item item) Check, if item has the same name

This app provide testing and comparison of three different types of probing, using different algorithms to obtain hash value.

First method of probing is Open hashing, which use array of lists as hash table. Lists (mainly LinkedList, because that type of list make more advantage and speed in program conditions) help to avoid hash conflicts, when different items pretend to get the same array place in hash table.

Simplest method of resolving hashing conflicts is linear probing, which select next free cell of array, if needed cell is also occupied by another Item. More complex method is pseudo random

probing, in which transition to next free place of array is not linear, but pseudorandom (controllable) which, in theory, help to better filling of hashtable.

There is exist double hashing probing, which use second hash with hashing conflict. Total hash recalculate until free space not find

Space complexity analysis

Analysis of space complexity for the code is summarized in the table below:

Method	Input space	Private fields space
App	2	3 helper methods and n*12 created bags
main	1	4
readFile	0	4+750 (total items created)
search	1	4+n*15 (StringBuilder objects and lists of find items)
Bag	5	8-12 (depending on type of object)
getHash	2	1-2 (depending of hashing type)
print	1	1-2 (depending of type listing)
find	1	2
getStrengths	2	1
Item	0	0
Item	1	5
getName	0	0
getMinimumStrenght	0	0
getCurrentStrenght	0	0
setCurrentStrenght	1	0
clone	0	1
toString	0	1
equalsName	1	1

Time complexity

Table of time complexity of methods

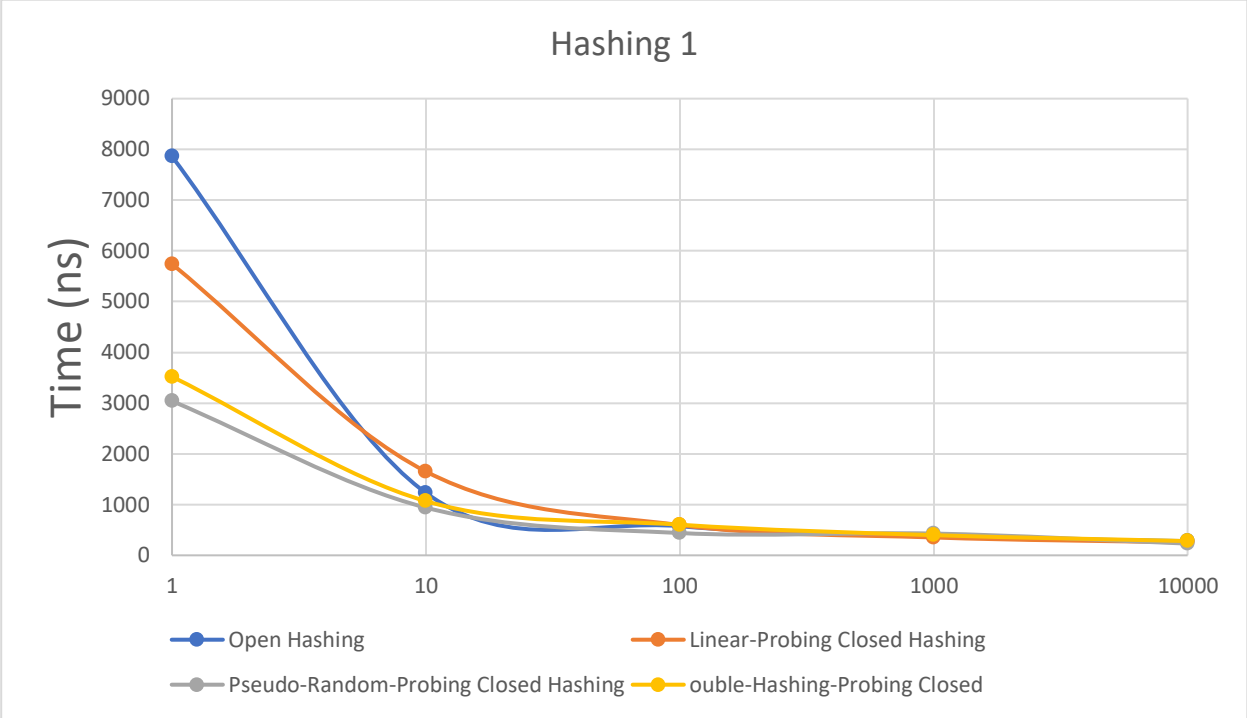
	Worst-Case	Average-Case	Best-Case	Space Complexity
Open hashing	O(n)	O(log n)	O(1)	O(n)
Linear probing	O(n)	O(log n)	O(1)	O(n)
Pseudo random probing	O(n)	O(log n)	O(1)	O(n)
Double hashing probing	O(n)	O(log n)	O(1)	O(n)

Results

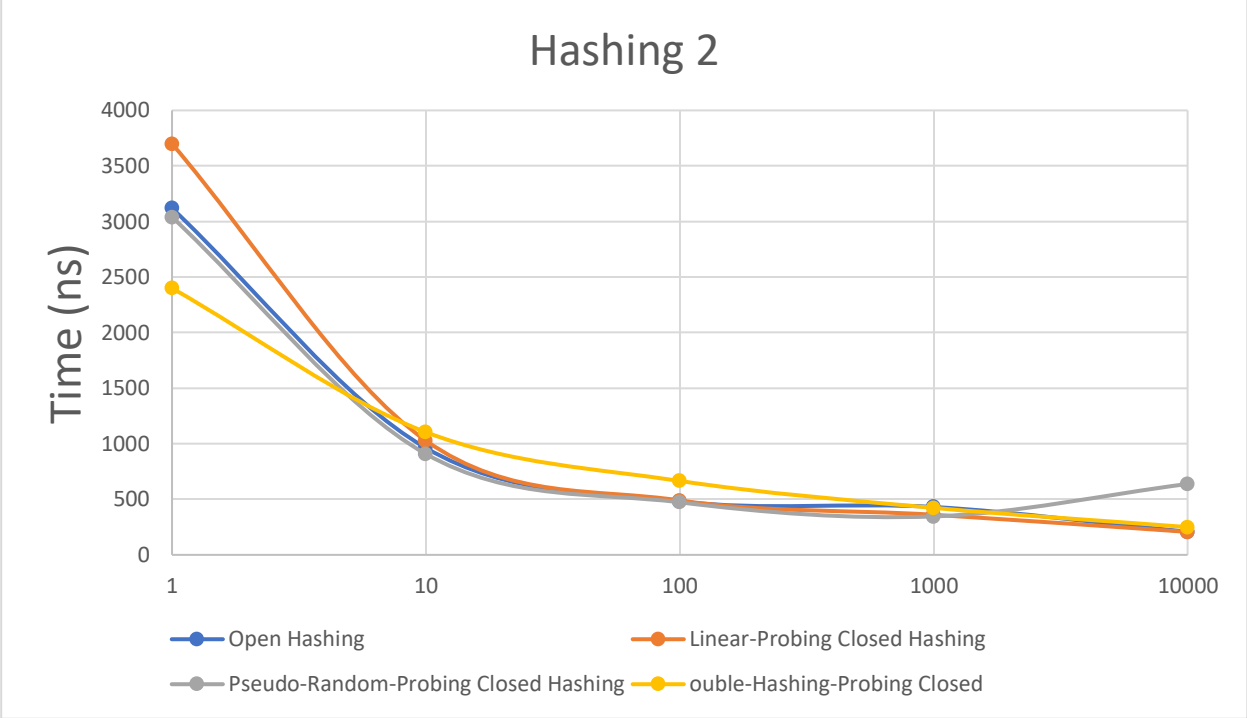
The empirical results agreed closely with the hand analysis as shown below. Each data point is the average of a 5 different instantiations of search the random item. The random numbers were from 0 to m included, where m is a maximal item array length..
System.nanotime() documentations shows that there may be some inaccuracies in measured time, so one non-logged circle of search was providing, during to some warming of Java Virtual Machine.

In all algorithms types, hash was calculated on base of sum of name and rarity item. due to most uniqueness of that parameters, and according to providing search on that data.

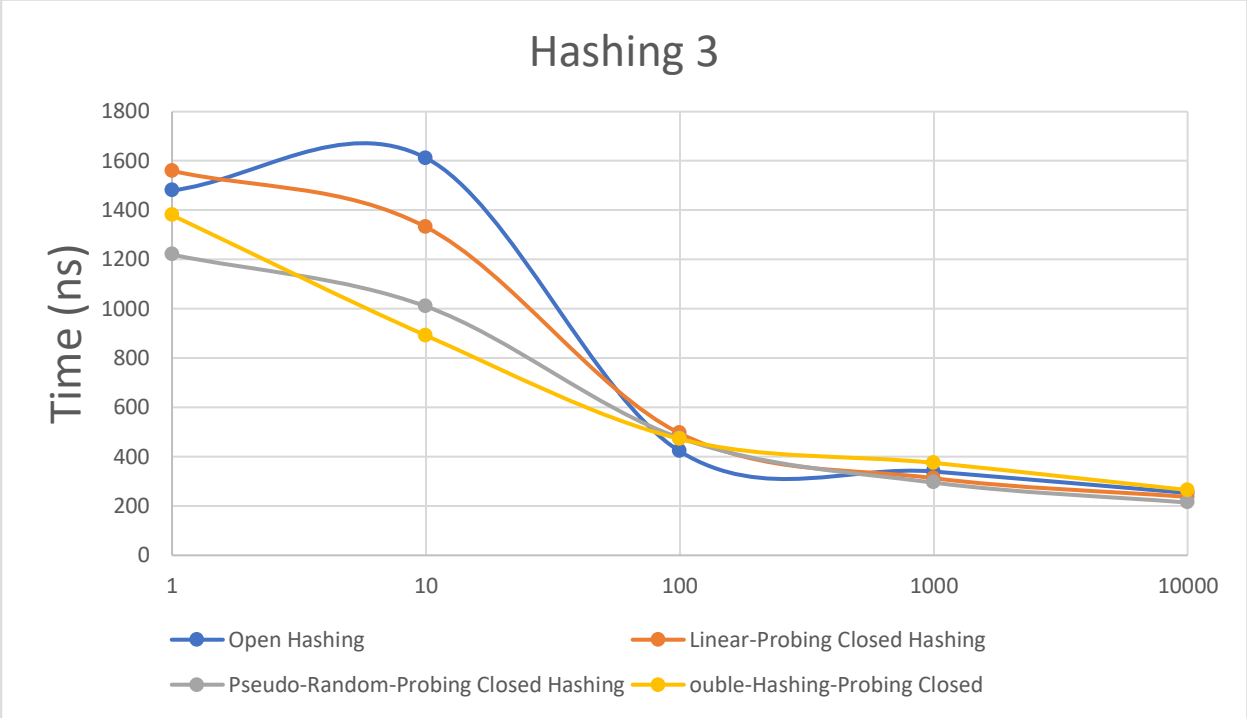
Each graph below show average time of search item, in using some number of bags, type of hashing and probing. Y-axis represented in $\log_{10}$, for better readability.



First, simple hashing algorithm, which sums and multiply values of bytes of hashed string. Most slow algorithm



Second algorithm ,which turn string into char type, and sum/multiply their values. Average time



Third algorithm which use built-in functional, of ahy object in Java – hash() function. Highest speed.

In all variations except when n=1 time complexity is similar to theorized, O(log n).

System.nanotime() was used to measure code execution time because System.militime() always returned a zero, since run time was less than a millisecond for small numbers of n.

PHASE 3: RISK ANALYSIS

There are no risks associated with this application. There is a possible risk of a file not found exception.

PHASE 4: VERIFICATION

The algorithms were tested multiple times, and the searched items was printed out multiple times to verify that each step of the process was computed correctly.

PHASE 5: CODING

The code of this program is include in the zip file. The code is explained by in line comments or in Javadoc.

PHASE 6: TESTING

Console text, which was used to testing behavior of program, showed in file output.txt. Due to very big amount of text data, all screen messages are turned into txt file.

PHASE 7: REFINING THE PROGRAM

No refinements are needed for this program.

PHASE 8: PRODUCTION

A zip file including the source files from eclipse and the output of the program have been included.

PHASE 9: MAINTENANCE

This application can be further improved once feedback from the grader is obtained.