Comp 251: Assignment 1

Answers must be returned online by October 14th (11:59pm), 2021.

Extended to October 18th.

No late submissions will be accepted

We will not answer questions after the 15th

HW2 will still be released on the 15th.

General instructions (Read carefully!)

- **Important:** All of the work you submit must be done by only you, and your work must not be submitted by someone else. Plagiarism is academic fraud and is taken very seriously. For Comp251, we will use software that compares programs for evidence of similar code. This software is very effective and it is able to identify similarities in the code even if you change the name of your variables and the position of your functions. The time that you would spend modifying your code to avoid being caught would be better invested in creating an original solution.

  Please don't copy. We want you to succeed and are here to help. Here are a couple of general guidelines to help you avoid plagiarism:

  Never look at another assignment solution, whether it is on paper or on the computer screen. Never share your assignment solution with another student. This applies to all drafts of a solution and to incomplete solutions. If you find code on the web, or get code from a private tutor, that solves part or all of an assignment, do not use or submit any part of it! A large percentage of the academic offenses in CS involve students who have never met, and who just happened to find the same solution online, or work with the same tutor. If you find a solution, someone else will too. The easiest way to avoid plagiarism is to only discuss a piece of work with the Comp251 TAs, the CS Help Centre TAs, or the COMP 251 instructors.

- Your solution must be submitted electronically on codePost. Here is a short **tutorial** to help you understand how the platform works. You will receive an email inviting you to join the class there in early October.

- To some extent, collaborations are allowed. These collaborations should not go as far as sharing code or giving away the answer. **You must indicate on your assignments (i.e. as a comment at the beginning of your java source file) the names of the people with whom you collaborated or discussed your assignments (including members of the course staff). If you did not collaborate with anyone, you write "No collaborators". If asked, you should be able to orally explain your solution to a member of the course staff.**

- This assignment is due on October 14^{th} at 11h59:59 pm. It is your responsibility to guarantee that your assignment is submitted on time. We do not cover technical issues or unexpected difficulties you may encounter. Last minute submissions are at your own risk.

- Multiple submissions are allowed before the deadline. We will only grade the last submitted file. Therefore, we encourage you to submit as early as possible a preliminary version of your solution to avoid any last minute issue.

- Late submissions can be submitted for 24 hours after the deadline, and will receive a flat penalty of 20%. We will not accept any submission more than 24 hours after the deadline. The submission site will be closed, and there will be no exceptions, except medical.

- In exceptional circumstances, we can grant a small extension of the deadline (e.g. 24h) for medical reasons only. However, such request must be submitted before the deadline, and justified by a medical note from a doctor, which must also be submitted to the McGill administration.

- Violation of any of the rules above may result in penalties or even absence of grading. If anything is unclear, it is up to you to clarify it by asking either directly the course staff during office hours, by email at (`cs251@cs.mcgill.ca`) or on the discussion board on Ed (recommended). Please, note that we reserve the right to make specific/targeted announcements affecting/extending these rules in class and/or on the website. It is your responsibility to monitor Ed for announcements.

- The course staff will answer questions about the assignment during office hours or in the online forum. We urge you to ask your questions as early as possible. We cannot guarantee that questions asked less than 24h before the submission deadline will be answered in time. In particular, we will not answer individual emails about the assignment that are sent sent the day of the deadline.

Technical considerations

- You are provided some starter code that you should fill in as requested. Add your code only where you are instructed to do so. You can add some helper methods. Do not modify the code in any other way and in particular, do not change the methods or constructors that are already given to you, do not import extra code and do not touch the method headers. The format that you see on the provided code is the only format accepted for programming questions. **Any failure to comply with these rules will result in an automatic 0.**
- Public tests cases are available on codePost. You can run them on your code at any time. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will grade your code with a more challenging, private set of test cases. We therefore highly encourage you to modify that tester class, expand it and share it with other students on the discussion board. Do not include it in your submission.
- Your code should be properly commented and indented.
- **Do not change or alter the name of the files you must submit, or the method headers in these files**. Files with the wrong name will not be graded. Make sure you are not changing file names by duplicating them. For example, main (2).java will not be graded. Do not add packages or change the import statements.

- Submit your files individually on codePost (no zips)
- **You will automatically get 0 if the files you submitted on codePost do not compile, since you can ensure yourself that they do.**
- **Note that public test cases do not cover every situation and your code may encounter runtime errors when tested on private tests. This is why you need to add your own test cases and compile and run your code from command line on a linux system (see the relevant tutorial on MyCourses).**

Homework 1: Building a Hash table

Exercise 1 (20 points). ***Building and inserting keys into two hash tables*** We want to compare the performance of hash tables implemented using chaining and open addressing. In this assignment, we will consider hash tables implemented using the multiplication and linear probing methods. We will (respectively) call the hash functions h and g and describe them below. Note that we are using the hash function h to define g.

$$\begin{aligned} \text{Collisions solved by chaining (multiplication method):}\quad & h(k) = ((A \cdot k) \mod 2^w) >> (w - r) \\ \text{Open addressing (linear probing):}\quad & g(k, i) = (h(k) + i) \mod 2^r \end{aligned}$$

In the formula above, r and w are two integers such that $w > r$, and A is a random number such that $2^{w-1} < A < 2^w$. In addition, let n be the number of keys inserted, and m the number of slots in the hash tables. Here, we set $m = 2^r$ and $r = \lceil w/2 \rceil$. The *load factor* α is equal to $\frac{n}{m}$.

We want to estimate the number of collisions when inserting keys with respect to keys and the choice of values for A. Note that once you build a hash table, A should remain constant for that table until you change w.

We provide you a set of three template files that you will complete. This file contains three classes; a main class and one for each hash function. Those contain several helper functions, namely `generateRandom` that enables you to generate a random number within a specified range. Details on which functions are included, how to use them, and where to add in your code can be found as comments in the java files. Please read them with attention. Note that verb?main.java? is only included for you to test out the other classes. You do not have to use it, and you must not submit it.

Your first task is to complete the two java methods `Open_Addressing.probe` and `Chaining.chain`. These methods must implement the hash functions for (respectively) the linear probing and multiplication methods. They take as input a key k, as well as an integer $0 \leq i < m$ for the linear probing method, and return a hash value in $[0, m[$.

Next, you will implement the method `insertKey` in both classes, which inserts a key k into the hash table and returns the number of collisions encountered before insertion, or the number of collisions encountered before giving up on inserting, if the table is full. For chaining, we consider the slots as "buckets", so it is not important where you insert the keys, or whether the ArrayList is full.

Note that for this exercise as well as for the rest of the homework, we define the **number of collisions** in open addressing as the number of slots visited before inserting or removing a key, not counting the slot in which the insertion is successful. For chaining, we simply consider the number of other keys in the same bin at the time of insertion as the number of collisions. You can assume the **key is not negativ**e, and that all keys are **unique**.

Once this exercises is complete, we are done with the `Chaining` class. All further exercises will focus on the open addressing method.

Exercise 2 (15 points)**.** ***Searching and removing keys*** Then you will add a method `searchKey`, this one only in `Open_Addressing`. This method should take as input a key k, and should return an array of two integers. The first one will be the index of k in the table if k is found, otherwise it returns -1. The second element of the array will be the number of collisions that happened before finding the key. If the key is not in the hash table, the element should show the number of slots visited. The search phase should be optimized: you should visit as few slots as possible.

You will also implement a method `removeKey`, only in `Open_Addressing`. This method should take as input a key k, and remove it from the hash table while visiting the minimum number of slots possible. Like `insertKey`, it should output the number of collisions. If the key is not in the hash table, the method should simply not change the hash table, and output the number of slots visited. You can call `searchKey` in this method. You will notice from the code and comments that empty slots are given a value of -1. If applicable, you are allowed to use a different notation of your choice for slots containing a deleted element.

Note: we ask for the number of slots visited instead of the number of collisions in the case where the key is not in the hash table because the definition of collision makes no sense if there is no key to collide with.

Exercise 3 (15 points)**.** ***Automatic resize of the hash table***
At this point you have a working hash table. We will now focus on making it usable under real conditions. First of all, the hash table has a finite amount of slots m. As the load factor increases the efficiency gets lower. In practical terms, we don't want the load factor to ever reach values over 0.75. The common way around that is to double the size of your table when necessary. Implement the `insertKeyResize` method so that it inserts the key and resizes the hash table if the load factor goes (strictly) over 0.75. You can call `insertKey` in this method. Note that r, w and A will have to be updated and the keys already registered may have to be relocated. Return the number of collisions when inserting the key into the resized table.

Exercise 4 (20 points)**.** ***Optimization***
When you solved Exercise 2, you had the option to use a specific marker for slots in which a key was inserted and then removed. You may have noticed that, when a key is deleted after the insertion of other keys, it may cause collisions that are no longer necessary now that the responsible key has been removed. To fix this, another key in the table whose hash probing sequence encounters that slot could be relocated there. Implement the function `searchKeyOptimized`, which not only does the same job as the `searchKey` function, but can also relocate a key if that improves the performance of searching in the table by minimizing the number of collisions.

`searchKeyOptimized` takes as input a key, and has the same return type as `searchKey` : an array containing the index at which the key is found (or -1), and the number of collisions encountered before finding the key. The difference is that unlike `searchKey`, once `searchKeyOptimized` finds the key, it should, if possible, move it back to an unoccupied slot if this slot would minimize the number of slots visited of a future call to `searchKey`. Return the number of slots visited prior to moving the key, if it is found. Your function should not move any other key than the one passed as input.

Exercise 5 (10 points)**. *Concerns about denial of service attack***
The hash function we are using is pretty basic and does not offer a lot of protection against an attacker. If an attacker knows which hash function you are using, they could add to the table a large amount of keys that would collide every time. Put yourself in an attacker's shoes and implement the function `collidingKeys`, which produces keys that all have the same hash value through function h. This function takes as parameters a key k, an integer n indicating the number of colliding keys to be returned, and w (which we assume is known by the attacker). It returns an `int[]` containing the n keys that collide with k (you can include k in the returned array).This method should not call or use any of the other methods you wrote for this assignment, e.g. you cannot call `probe` from this method. Also, note that as an attacker you do not have access to A, only w. *Hint: you will have to find a mathematical expression for a set of keys that all share the same hash through function h. Second hint: remember that if you have to increment i, it means a collision occurred.*

Exercise 6 (20 points)**. *Universal hashing***
The best way to avoid the type of attack described in the previous exercise is to randomize the choice of the hash function. Since we have a resizing function, we can randomize the hash function each time we resize the table because we need to rehash all the keys anyway. In universal hashing, the randomization of hashing function occurs by choosing a function in a set (a family) of hashing functions. This set has mathematical properties that guarantees as few collisions as possible. For our needs we will use the following hash function family :

$$h(x) = ((ax + b) \mod p) \mod m$$

Here p is a prime number higher than the required size of the table m. a and b are random integers chosen between $[1, p-1]$ and $[0, p-1]$ respectively.

Fill in the new `Universal_Hashing` class so that it uses universal hashing with the same linear probing formula from above:

$$g(k, i) = (h(k) + i) \mod 2^r$$

Note that not all functions need to be overwritten, and feel free to reuse your previous code to get it done.

Submit your three Java classes (not main.java) via codePost. Test your code thoroughly!