CSE221 Data Structures (Fall 2021)
Instructor: Prof. Young-ri Choi
TAs: Gyeongchan Yun, Jin Yang and Ju Young Bang
Due date: Nov 21, 2021, 11:59 pm.

Notes:

- Be aware of plagiarism. You are allowed to use Piazza for QnA, but do **NOT** discuss with your classmates **directly**.
- Check correctness of your code on **uni server** before final submission.
- Please carefully **follow the grading strategy** on the last part. We will not tolerate any issues caused by not following the strategy.
- Do **NOT** change the declaration of **given functions in the skeleton code**
- You can add your own global/static variable, function or classes, but do **NOT** use STL or other libraries except provided.
- **Grading**: 100 test cases (1 point per each case)
- For grading, we only consider 3 header files in **assignment3/** directory: '**`FlatHash.h`**', '**`HierarchyHash.h`**', '**`NRKFlat.h`**'
- Maximum 3 late days are allowed with a 10% reduction per day.

Preparation Method

To start your assignment, you have to update your local git repository. Enter the below command.

```
$ cd uni+student_id
$ git pull origin master
```

If successfully updated, you will be able to see the assignment3/ directory.

Assignment 3: Hash Tables

The objective of this assignment is mainly two-fold: (1) to implement various hash tables and key overflow handling mechanisms correctly, and (2) to learn about the impact that various hash table factors have on space usage and time cost in diverse key insert/remove/search scenarios.

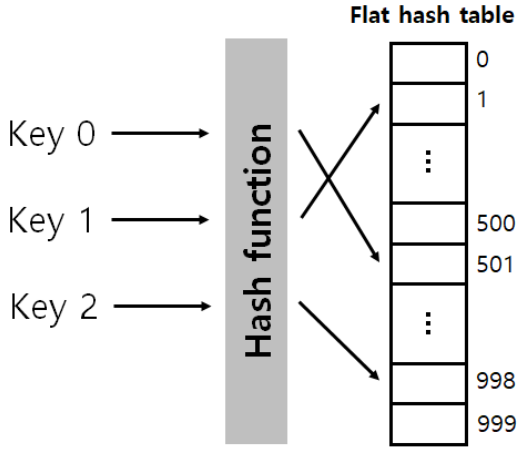
For each hash table implementation, we will **provide configurations and rules to *strictly* follow.** It is *extremely* critical to implement them correctly; otherwise, you will have a hash table which is in an **inconsistent state** where keys are present in different locations within the hash table after a series of inserts and deletes, compared to the correct implementation that strictly

follows configurations and rules. This indicates that your program fails in passing our tests.

All basic contents on hash tables can be found in slides at lecture 10. Starting from flat hash table (slide 6), we will ask you to implement two variations of hash table in this assignment. We will consider open addressing only (not chaining).

1. Flat hash table (50 pts)

Task 1 is regarding a flat hash table which is a single array-based hash table stored contiguously in memory. This represents the static hash table in slide 5. The following figure shows how the flat hash table initialized with 1000 buckets looks like: index ranges from 0 to 999.



This hash table must be pre-allocated with 1000 buckets placed in contiguous memory space when it is initialized. The number of buckets remains unchanged until the hash table is resized.

Here are some configurations and rules to follow:

(1) Hash table is initialized with 1000 buckets. There is a single slot for each bucket, thus 1000 slots in the hash table in total.

(2) Keys to insert and delete are all positive integers falling in the range of **[1, 1000000]**. In each slot, the hash table stores only the key itself. If a slot does not contain a key yet, it has **zero** in value.

(3) Since there is only one slot in each bucket, if a bucket is already occupied by a previously inserted key, your program **has to perform overflow handling**.

(4) Implement two overflow handling techniques: (i) linear probing (slide 23) and (ii) quadratic probing (slide 45). When performing quadratic probing (i.e., $key+j^2 \% b$ for $j=0, 1, 2, \ldots, b-1$), not all buckets can be examined (slide 49). Therefore, if quadratic probing cannot find an empty bucket, it then should perform linear probing (i.e., $key+j \% b$ for $j=0, 1, 2, \ldots, b-1$), which guarantees successful insertion of a key as long as there is at least one empty bucket.
(5) Upon deletion of a key in linear probing approach, perform shifting. For key deletion in quadratic probing approach, use tombstones.
(6) Hash function is simply **division** where the divisor equals to the number of buckets (slide 17). Use the given function in `FlatHash.h` named `HashFunction(const unsigned int key)` to hash.
(7) Increase the size of the hash table if the loading factor is >= α (slide 7). When resizing for the current hash table with ***B*** buckets, create a new hash table with ***2B*** buckets and rehash currently resident keys into the new hash table. Note that resizing occurs only during key insertion: thus, **we never decrease hash table size even when we delete.**
(8) Rehashing works as follows: you scan the old hash table from the first bucket to the last bucket in order, see if the key is resident, and insert each encountered resident key into the new hash.

Time cost will be used to evaluate hashing performance. To briefly explain, time cost reports **the number of buckets probed for the given operation**. Especially in task 1 and 2, **the number of buckets probed** should **always be greater than zero** because you need to probe (or access) at least one bucket for insert, remove and search. In cases where a negative time cost must be returned, multiply -1 to the number of probed buckets. Please read the detailed description below carefully for more information.

Note that time cost does not include the following:
(1) Resizing cost including memory reallocation, and rehashing resident keys.
(2) Shifting cost that occurs after key deletion.

`FlatHash.h` includes a list of functions you have to implement:

- `FlatHash(overflow_handle _flow, float _alpha)`
  Constructor. You need to configure either using `LINEAR_PROBING` or `QUADRATIC_PROBING` here. In addition, you should also configure `alpha` (α) which is the loading factor used to determine resizing. When `_flow` is set as `LINEAR_PROBING`, you consider approach (i) in (4); do linear probing and perform shifting upon key deletion. If you set

`_flow` as `QUADRATIC_PROBING`, you consider approach (ii) in (4); you first start quadratic probing phase and if failed in finding an empty slot, switch to linear probing phase. In `QUADRATIC_PROBING`, no matter which phase you are in, you must use tombstones only, without performing any shifting. In each slot, **zero value means an empty slot** and **any value within [1, 1000000] is a valid (or resident) key**. Thus, to mark as tombstone, you can pick any value **greater than 1000000 (> 1000000)** which is out of the valid key range.

- `~FlatHash()`
  Destructor.
- `int insert(const unsigned int key)`
  Insert `key` into hash table. This function has to deal with resizing if the loading factor becomes greater or equal to α (loading factor >= α) **after insertion**. Return value is the time cost. `insert` can meet a tombstone while probing slots. Since that slot is not occupied by a valid key, you can do insertion. In case the given `key` is an already-inserted key, you must NOT insert the duplicate but must return the time cost it took as a negative number, until the duplicate is found by the given overflow handling method. For example, assume that the key of 1 is already inserted and the time cost taken to find the duplicate is 10. Then, insert(1) must return -10.
- `int remove(const unsigned int key)`
  Delete `key` from hash table if exists. Return value is the time cost. `remove` can meet a tombstone while looking for a key to delete. Since you did not do shifting, next slots that are going to be probed can contain the key to delete. Thus, you have to keep moving unless the following two situations: 1) The key is found 2) An empty slot (zero-value) is encountered. In case the given `key` does NOT exist in your hash table, you should return the time cost it took as a negative number, until it turns out that the hash table does not have the given `key` by the given overflow handling method.
- `int search(const unsigned int key)`
  Search `key` from hash table. Return value is the time cost. `search` can meet a tombstone while searching for a key. Similar to `remove`, if you run into a tombstone, keep moving on unless the following two situations: 1) The key is found 2) An empty slot (zero-value) is encountered. In case the given `key` does NOT exist in your hash table, you should return the time cost it took as a negative number, until it turns out that the hash table does not have the given `key` by the given overflow handling method.
- `void print()`

List out currently resident **valid** keys. Our skeleton code includes comments on the printing format of `print()`, which you should follow. In short, all valid keys in the hash table are printed out and they are in "(bucket_index$_1$:key$_1$,bucket_index$_2$:key$_2$,…,bucket_index$_n$:key$_n$)" format where bucket_indexes are in **numerically increasing order**. The outermost bracket must be printed out even when the hash table is empty (See your skeleton codes). There must be no space between each bucket, key pairs and the output must end with a newline character (\n).

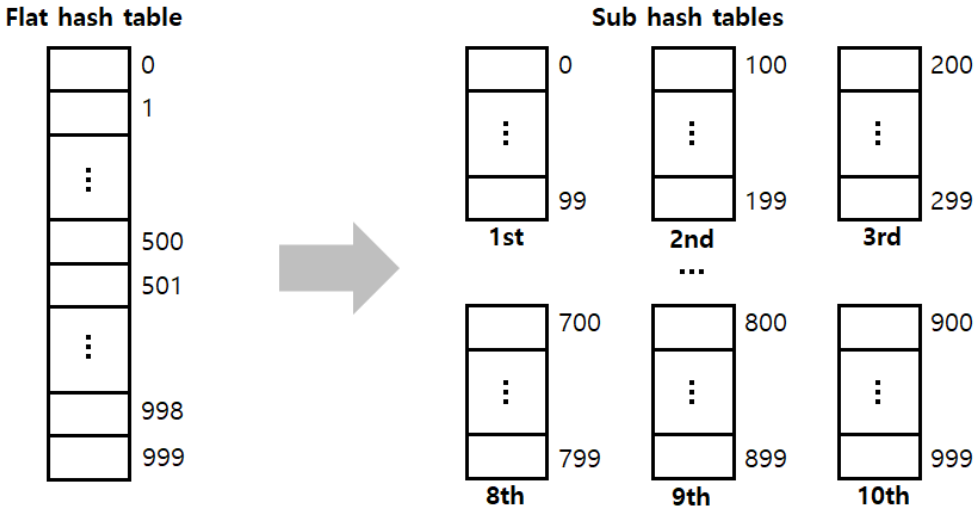
- `void clearTombstones()`
  Delete all tombstones from the current hash table to keep **valid** keys only. It works meaningfully on `QUADRATIC_PROBING` mode, while it causes no change on `LINEAR_PROBING` mode. It requires **rehashing**. The function `clearTombstones()` is going to be explicitly called by us to clear out all tombstones from the current hash table. Once it is called, the function scans the hash table from the first bucket to the last bucket in order and rehashes only valid keys (i.e., within a range of [1, 1000000]) to the hash table of the same size. Note that **tombstones are not categorized as valid keys**. You can reuse the existing hash table or create a new one with the same size. **You can leave the existing hash table without modifications when there are no tombstones at all**.

2. Hierarchical hash table (20 pts)

Memory consumption in a flat hash table is not related to loading factor. That is, the table uses the same amount of memory irrelevant of how many buckets are occupied by non-zero values. This is because all buckets are pre-allocated and physically located in memory. In the figure at task 1, the size of the hash table should be consistently 1000 buckets until resizing happens. In task 2, you will take an approach of saving space. If you are currently not storing many keys in hash table buckets, you attempt to allocate less memory.

To realize this, we propose a concept called hierarchical hash table, where you need to partition a flat hash table into a number of **sub hash tables**. A sub hash table is not initially pre-allocated, rather allocated on-demand when it actually has keys or tombstones to store. A sub hash table covers a partial range of full table buckets. Sub hash tables can be ordered by the range they are in charge of. The following figure shows how a flat hash table with 1000 buckets can be organized in 10 sub hash tables, where each sub hash table stores 100

continuous yet disjoint buckets.



Let's see how using sub hash tables instead of a flat hash table saves memory. For example, if you have only one key to store and it ought to be placed at bucket 0, you need to create only the first sub hash table which covers from bucket 0 to bucket 99. This way, you can defer creation of other sub hash tables until they have keys to store.

In order to keep tracking whether each sub hash table is created, and if created how to access such a sub hash table, you need to maintain pointers to sub hash tables: 10 pointers in our example above, where N^{th} pointer corresponds to N^{th} sub hash table. For example, if the first sub hash table is being created, the first pointer becomes valid and should point to the address of the first sub hash table; other entries are all still **null** since their sub hash tables do not exist yet.

Task 1's configurations and rules are applied identically to this task. The additions are:

(1) Each sub hash table is created on-demand when it has keys to store.
(2) Sub hash tables are all equal sized as 100 buckets, fixed.
(3) Like task 1 [See 1.(7)], don't forget to resize your hash table in `insert` method when loading factor gets >= α. In other words, you have to double the number of sub hash tables during resizing (not doubling the number of buckets for each sub hash table).
(4) For a sub hash table that was already created, if its buckets become all empty since all previously resident keys or tombstones have been deleted, your program has to **free memory** used by the sub hash table

to save memory. In this case, your program has to ***nullify the corresponding*** hash table pointer.

(5) Use the given function in `HierarchyHash.h` named `HashFunction(const unsigned int key)` to hash.

The only difference between the implementation of task 1 and task 2 is how to structure hash table space. So when we measure the time costs, both should report the same value. On the other hand, the primary goal of task 2 is about saving memory consumption. In measuring memory consumption, you need to consider space used by valid sub hash tables only. You can ignore space used for sub hash table pointers.

Along with all functions in task 1, there is an additional function to implement on top of task 1:

- `int getAllocatedSize()`
  Return the number of **allocated** hash table buckets (not the number of **occupied** buckets by valid keys).

Our skeleton code `HierarchyHash.h` includes some comments on the printing format of `print()` for hierarchical hash table, which is slightly different from the one for flat hash tables. It must include the ID of sub hash tables. In short, all keys present in the sub hash table are printed out in "sub_hash_table_ID:(bucket_index$_1$:key$_1$,bucket_index$_2$:key$_2$,…,bucket_index$_n$:key$_n$)" format where sub_hash_table_ID and bucket_indexes are both in numerically **increasing order**. Note that the sub hash table ID starts from 0, not 1. You must separate the printing of each sub table using `endl`. In addition, there should be no empty spaces between the output string. For example, when there are resident keys in sub hash table 0, 1, and 9, the print-out may look like follows:

```
0:(3:3,7:7,98:98,99:99)
1:(100:1099,101:1098,103:103)
9:(903:903)
```

In overflow handling, you should implement both linear probing and quadratic probing. Try to think about which one would overall consume less memory in use of hierarchical hash table and why.

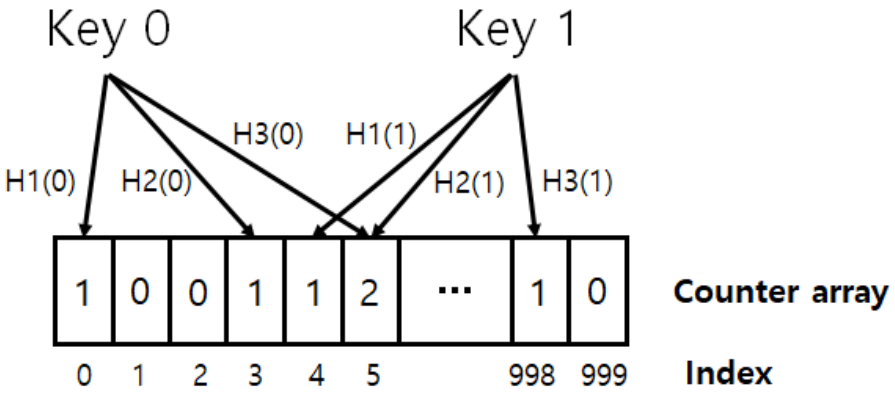
3. Non-resident key filter (NRK filter) (30 pts)

So far, you have attempted to save spatial costs from the Flat Hash baseline. In this task, meanwhile, you will have another approach to save time costs from

the same baseline.

Non-residence key filter, namely *NRK filter,* is intended to filter out keys that are currently not resident in the hash table before you are actually searching on the hash table. That is, by using the NRK filter, your program can avoid searching for the keys that are not currently resident in the hash table, effectively reducing time costs. While effective, by nature, NRK filter cannot filter out all non-resident keys. "Some" non-resident keys may not be filtered out and you may need to search for those keys in the hash table. However, for a number of non-resident keys, NRK filter can detect their non-residency and save delete/search costs.

Let's see how to build NRK filter and how non-resident keys can opportunistically be filtered out. The fundamental building block of NRK filter is three hash functions (H1, H2, H3) and an array of counters which are all initially zero. Before you insert a key, you are executing those hash functions and get three locations in the counter array, in which you increment each counter by one. Next when you later search for the same key which is certainly resident in the hash table, your program simply runs the hash functions to get locations in the same way, and check if each counter value has been every incremented. If all three encountered counters have been incremented, the key can possibly exist in the hash table. The following figure shows the snapshot of NRK filter with the number of counters as 1000 when the hash table is inserted with two keys (0 and 1) as an example.

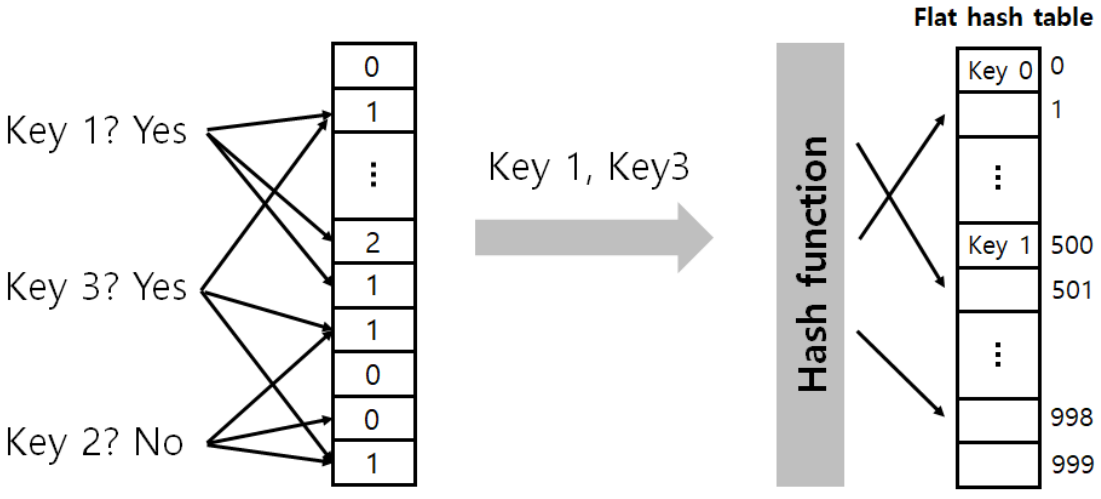


Note that when a key is being deleted, what NRK filter has to do is almost identical to insertion case in that it must find out the counters in the counter array using the same hash functions. The only difference is that in deletion case, NRK filter needs to **decrement** each counter by one. It is possible that

two or more distinct keys hit on the same counter. However, since we use a counter, we can factor in how many resident keys hit in a certain counter. For example, in the figure above, NRK filter has key 0 and key 1 hit on the same counter at index 5. Thus, the counter value is two.

When searching, a non-resident key is likely to hit on three counters where at least one counter has highly probabilistically value zero. However, it could be possible that **a non-resident key hit three counters that have been incremented by other resident keys**. This could probabilistically happen, since each key generates three locations and the number of counters is fixed and limited. Therefore, by nature, we cannot fully prevent location collisions from happening in NRK filter.

As a result, NRK filter returns either "possibly in the hash table" or "definitely not in the hash table". In detail, NRK filter tells us true positive for resident keys (for truly resident keys, yes! There exist). However, it may tell us false positive for some non-resident keys (huh! You sometimes say yes for non-resident keys). False positive answers are typically infrequent. In many cases, you will see true negative for non-resident keys (woot! All filtered out keys are certainly non-resident keys!), for which you save costs during search/delete operations. In this case, since there is **no false negative**, you can show the non-existence of the key even **without probing the table once**. In other words, there can possibly exist cases where the time cost is 0. The figure below shows the case where key 1 is true positive, key 2 is true negative, and key 3 is false positive.



`NRKFlat.h` includes skeleton code and inherits from `FlatHash.h`. The following is additional configurations and rules for NRK filter:

(1) We provide a function to calculate three hashing outputs once you give a key.
(2) When you calculate a location in the counter array using a hashing output, use **modulo** where the divisor equals the number of counters. Use the given function in `NRKFlat.h` named `HashFunction(const unsigned int key)` to hash.
(3) Filter size, which indicates the number of counters in an array, will be given as an additional argument to the constructor. Check the skeleton file and you will be able to see the following constructor definition: `NRKFlat(overflow_handle _flow, float _alpha, unsigned int _filter_size)`. Filter size is doubled when the hash table is resized.
(4) When your program performs hash table resizing, you should perform resizing for the counter array. As a result of resizing, all resident keys must be re-populated from the old hash table to the new hash table. Similarly, your program has to re-populate NRK filter's counters from old one to new one.

Grading strategy

During testing, we will issue a set of `insert` / `remove` / `search` operations to populate your hash table. If you follow the specifications of each assignment task correctly, your hash table would have **consistent internal status** (e.g., all resident keys and their locations in hash table buckets are consistent). During and after those operations, we will evaluate the results in following ways:

(1) We will verify your time & space cost reports. Those three operations must report time cost in **the number of buckets probed** and some other operations report space cost in **the number of buckets that are currently allocated in memory**. Once you implement each assignment task correctly, you will be reporting the correct **time & space cost** for each test operation we issue. Otherwise, you are reporting incorrect cost: your program fails our tests.

(2) To measure time / space cost, we will compare your outputs and the desired output generated by us under certain overflow handling configured in `_flow`.

(3) We will compare only the printed outputs. For example, you can get credit for a certain test case only when your program gives a perfect output for that case. So please **take care of your printing snippets, especially any trace of debugging stages.** Again, you should not include any printing code except for the `print()` method.

(4) In evaluation time, you will receive a **ZERO score** for the specific test case (among lots of test cases) where your program shows any kind of compile error, runtime error, or infinite loop over timeout (10 secs). Again, please do a final check on **uni-server**, not on your arbitrary environment because we will conduct the evaluation on uni-server.

Compile/Execution Method

We will compile your code by **ONLY** the Makefile we provided, which explicitly allows C++11 grammar. We emphasize that **you are not allowed to modify the Makefile**. The grading will only be done with the original Makefile. Any compile error occurred due to using different compilation will not be tolerated.

To compile the given files, enter the following command.

```
$ make
```

When the compilation is done, the following command will execute your code.

```
$ ./assign3
```

Submission Method

Note that main.cpp is an example of testcase which can be modified by yourself to check various cases. We only consider 3 header files in **assignment3/** directory for grading:
'**`FlatHash.h`**', '**`HierarchyHash.h`**', '**`NRKFlat.h`**'

This assignment must be submitted through GitLab. You will be able to push your source code through the following commands.

```
$ cd uni+student_id/assignment3
$ git add FlatHash.h  HierarchyHash.h  NRKFlat.h
$ git commit -m "Final commit" (commit message is not important for grading)
```

```
$ git push origin master
```

To double check whether your submitted code has any problem (e.g, git conflict, compile error, etc) or not, please follow the below commands.

```
$ git pull origin master
$ make
```