CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 11
Date: 24th Oct, 2019

Important Instructions

1. **List of Input Files:** input.txt
2. **Output Files to be submitted:** ROLLNO_A11_P1.c/.cpp
3. Files must be read using file handling APIs of C/C++. Reading through input redirection isn't allowed.
4. You are to **stick to the file input output formats strictly** as per the instructions.
5. Submission through **.zip files are not allowed**.
6. Write your name and roll number at the beginning of your program.
7. Do not use any global variable unless you are explicitly instructed so.
8. Use proper indentation in your code.
9. Please follow all the guidelines. Failing to do so will cause you to lose marks.
10. **There will be part marking.**

Hashing

Problem Statement- Neighbourhood hashing

Neighbourhood hashing is an improved version of reordering scheme which can be used for collision resolution in hash tables. It guarantees a small number of look-ups in finding entries. Moreover, these look-ups are in contiguous memory locations.

The main idea behind the hashing is that the hash table uses a single array of n buckets. Each bucket B has a neighbourhood of size H. The neighbourhood of a bucket B is defined as the bucket itself and the (H-1) buckets following the bucket B contiguously in memory (i.e, H buckets total). This also means that at any bucket, multiple neighbourhoods are overlapping.
Figure 1 is a schematic representation of neighbourhood of a bucket.

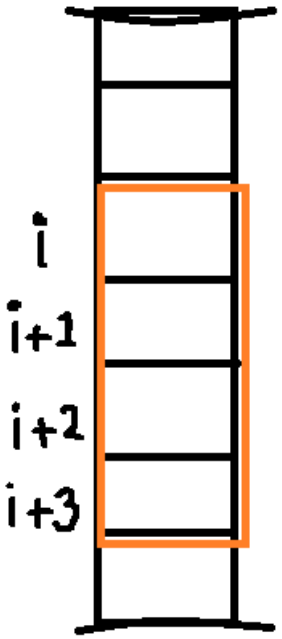


Figure 1: Neighbourhood of bucket i with $H = 4$.

Initial bucket is the bucket (or the index of the hash table) where the entry is actually hashed. Neighbourhood hashing guarantees that an entry will always be found in the neighbourhood of its initial bucket. When we are searching for a value, we will search it in its initial bucket and the neighbourhood of the bucket. On insert, we will keep the inserted value in the neighbourhood of its initial bucket through swapping.

- **Insertion:** To insert an item x in the hash map, where $hash(x)\%n = i$, we need:
  - From the bucket i, search for an empty bucket j through linear probing.
  - If the bucket j is in the neighborhood of i i.e. $(j - i < H)$, insert it there and terminate.
  - Otherwise, find in the interval $[j - H + 1, j)$, an item y where $hash(y)\%n >= j - H$. Swap the bucket for the item y with the empty bucket j and repeat until j is in the neighborhood of i.
  - An example of the insertion is shown in Figure 2.

  While following the algorithm of insertion, if you can not find any prospective place for swapping at any stage, then you can throw a message of insertion error.
- **Find:** To find an element in the hash table, we just compute its initial bucket. We then search in the initial bucket and its neighbours. If the key does not match any of the keys for the entries in the neighborhood of the initial bucket, then the entry is simply not in the table.
- **Deletion:** The deletion process is simple: replace the entry to delete by an empty entry, which can be null or whatever value was chosen for the implementation at hands.

A schematic diagram of the insertion is described below. Here the value of H is 4.

Part I: Build the hash table

Your task is to implement the above hashing technique. After initialization of the hash table, you have to do the following operations following the rule of neighbourhood hashing.

- insert entries into the hash table and print the hash table after a sequence of insertion operation.
- delete an entry from the hash table and print the hash table after deletion.
- insert entries into the hash table and print the table after the insertion operations.

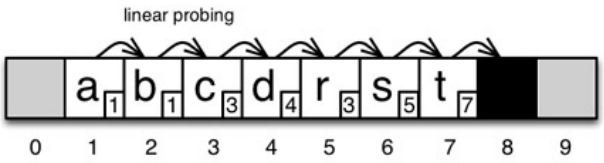
The table size / bucket array size (n), neighbourhood size of each bucket (H) and the entries are given in the input file.
Hash Function: You have to compute the hash key of the entries using the following hash function-
Your entries are two digit numbers (in between 0 and 99) and the hash function is the sum of the two digits. So the hash key will be the sum of two digits but you require another operation also. In order to convert the key into an index of the hash table, the modulo operation (%) is needed. The $sum\%bucket\_array\_size$ will be the final hash key. Lets take an example-
Suppose an entry is 77. So the hash key will be $7 + 7 = 14$ but in order to fit into an hash table (or bucket array) of size 10, one need to consider the modulo operation. So the final hash key will be- $14\%10 = 4$.

Sample Input File

FILE: *input.txt*

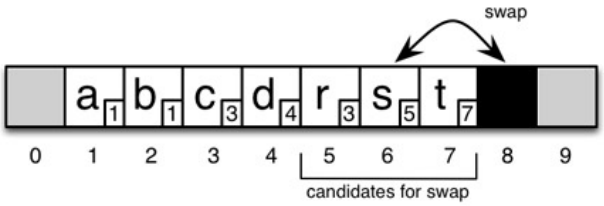
```
10
3
in 7
92
74
56
12
41
69
88
del 1
56
in 3
84
66
34
```

Step 1: Linear probing from the initial bucket to find an empty bucket



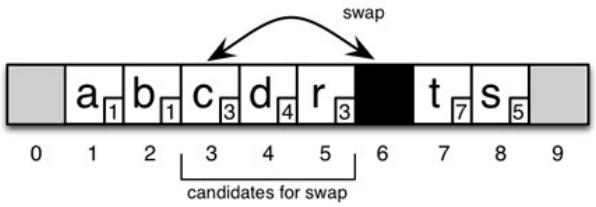
- The initial bucket is bucket 1, because mod(hash(x), size) = 1
- Starting from bucket 1, linear probing finds an empty bucket in position 8, but it is beyond the neighborhood of initial bucket in position 1

Step 2(a): The empty bucket is moved



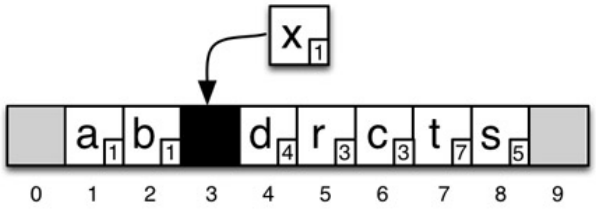
- The empty bucket needs to be moved
- Swap candidates are buckets 5, 6, and 7, because 8 - (H-1) = 5
- In bucket 5 there is r, but it cannot be swapped because its base bucket, which is bucket 3, is too far from bucket 8.
- Then bucket 6 is tried, and the base bucket of s being 5, it can be swapped with 8.

Step 2(b): The empty bucket is moved again



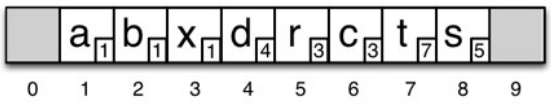
- The empty bucket is now in position 6
- It is still too far from bucket 1 and has to be moved again.
- Swap candidates are now bucket 3, 4 and 5 because 6 - (H-1) = 3
- Bucket 3 can be swapped because its base bucket is 3.
- The bucket will be moved again until it reaches a stable position, or until it cannot be moved, in which case the table needs to be resized.

Step 3: The new entry is inserted



- The empty bucket is now in position 3
- It is now in the neighborhood of the initial bucket, bucket 1, and thus does not need to be moved.
- The new entry can be inserted in bucket 3

Step 4: Final state after displacements and insertion



- The new entry x was inserted in the empty bucket, in bucket 3

Figure 2: Insertion Process with $H = 4$

First and second number of the input file specifies the bucket array size and neighbourhood size respectively. The third line containing the word "in" followed by the number 7 indicates that you have to insert 7 numbers into the hash table . Those 7 numbers are mentioned in the following 7 lines. Similarly the word "del" followed by the number 1 tells you that you have to delete a number from the hash table and the number is specified in the next line. You have to print the hash table after the sequence of insertion, deletion and insertion operation of the given entries. You are free to choose any data structure for the implementation.

Sample Output File

FILE: *output.txt*

```
92 74 56 12 41 69 88

92 74 12 41 69 88

92 74 84 66 12 69 41 88 34
```

Part II: Bitmap Representation

Neighbourhoods can be stored using bitmaps (bit arrays). With this solution, each bucket needs a bitmap in addition to all the data already needed for that bucket. Each bit in that bitmap indicates if the current bucket or one of its following (H-1) buckets are holding an entry which actually belongs to the current bucket. It is useful because from the least significant to the most significant, it tells us which neighbours contain a value belonging to the bucket, going from the current bucket to the most far away neighbour.
Figure 2 below shows the hash table presented in Figure 1, and along with the bitmaps for each of the buckets.

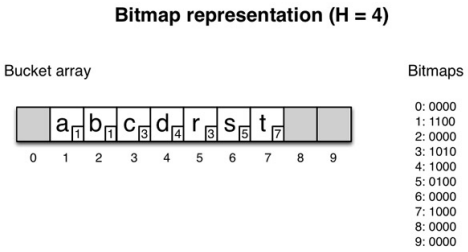


Figure 3: Bitmap Representation

Your task is to compute the Bitmap representation of the buckets of the final hash table that you have already built after the last insertion operation in Part I. You need to append the bitmaps in the file *output.txt* as shown below-

0000

1100

0000

1010

This above example is showing the bitmaps of a part of the hash table of Figure 3. Your output (bitmaps) will be different than the above one.

Hint: You have to incorporate some additional data structure for the bitmaps in your implementation.

You need not submit *"output.txt"*, but your code should write the output in the given format in a file named *"output.txt"*.