BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS-F211: Data Structures and Algorithms

Lab 8: Hashing and Bloom Filters

Introduction

Welcome to Week 8 of Data Structures and Algorithms! Today we shall be familiarizing ourselves with one of the most important data structures out there, namely hash tables. Hash tables are used to look up records based on a key *quickly*. This is another instance of the space-time tradeoff that we have been looking at in the past few weeks. They are widely used in a variety of applications ranging from cache lookups to case matching in switch cases. After hash tables, we would be implementing bloom filters which is a probabilistic data structure that aims to optimise the time taken to check the membership of an element in a set.

Topics to be covered in this labsheet

- The Dictionary ADT
- Hashing
    - Some common families of hash functions
    - Collision resolution techniques
- Bloom Filters
    - Vanilla Bloom Filters
    - Counting Bloom Filters

The Dictionary ADT

A dictionary stores key-value pairs of data. A data item might be in the form of an object or record having many fields. Data items have to be looked up based on a particular field called the **key** field. You can think of a filing cabinet or a library management system where the records are looked up based on a single value. In the case of a library, this value is the call number of the book. Although the book might have a lot of auxiliary information such as the title, name of the author, checked-out status etc, we only look at the call number while retrieving a particular book.

Formally, the Dictionary ADT is described as follows:

Behaviour (Interface)

The following methods are included in the Dictionary ADT:

- `int size(Dictionary D)`: Returns the number of elements currently stored in the Dictionary.
- `Boolean isEmpty(Dictionary D)`: Returns true when D is empty
- `LIST elements(Dictionary D)`: Returns a list of all Elements stored in the Dictionary.
- `Element *findElem(Dictionary D, key k)`: Returns an occurrence of an Element with key k stored in the dictionary.
- `LIST findAllElem(Dictionary D, key k)`: Returns a list of all occurrences of Elements with key k stored in the Dictionary.
- `insertItem(Dictionary D, key k, Element e)`: Inserts the Element e with key k.
- `removeElem(Dictionary D, key k)`: Removes an Element with key k from the Dictionary.
- `removeAllElem(Dictionary D, key k)`: Removes all Elements with key k from the Dictionary.

Note that we assume that our ADT has elements of a custom user-defined data type called Element which is associated with a key k. The main operation supported by a dictionary is searching by the key.

Now Dictionaries can be implemented in a variety of ways such as a linked list, an array, a sorted array, etc. Try to think about the time and space complexities of the above operations in these implementations. How would you achieve better than `O(n)` time complexity for `findElem()` on the average case? What is the drawback of this approach?

Hashing

Hash Tables are an effective way of implementing the Dictionary ADT. A hash table is basically an array where we define a mapping from the key space to the index space. The key space is the range of values that the key can take. For example, if the key is a 10-digit mobile number, the key space is of size 10^{10}. Similarly, if the key is a character array in C of length 5, the keyspace is 128^5 (range of ASCII = 128).

Now, the hash table is basically an array. So the index space is `0` to `size - 1`, where size is the size of the array. The mapping between the key space to the index space is called a hash function.

In this section, we assume collisions are resolved by **chaining**. In chaining, each element of the hashtable is a linked list and we place elements that hash to the same slot in the same linked list.

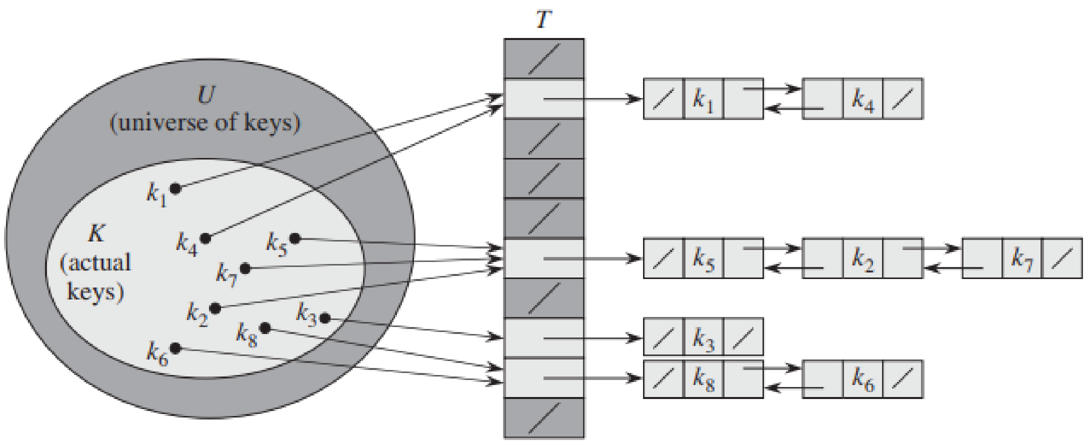

Figure 1: Collision resolution by Chaining

Figure 1 shows collision resolution by chaining. Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j. For example, h(k1) = h(k4) and h(k5) = h(k7) = h(k2).

Let us look at some common hash functions:

Some common families of hash functions

Direct Addressing

Let us consider the case that the range of keys is `0` to `m - 1` and the keys are distinct. Here, we can set up the hash table as an array `T[0 ... m-1]` in which

- `T[i] = x if x ∈ T and key[x] = i`
- `T[i] = NULL otherwise`

This is called a direct address table.

In general, in direct addressing, there is a one-to-one mapping between the key and the index. This is known as a perfect hashing function. The size of the hash table is equal to the key range. It takes `O(1)` time complexity to retrieve the elements since each index at the hash table is mapped to from a single key. Collision (two keys mapping to the same value) would occur only when both keys are equal.

However, the space complexity here is `O(r)` where r is the size of the range of values that can be taken. Each location at the hash table acts as a "bucket" or a "bin". In practice, most of the time the number of entries to be hashed (n) is often times much much smaller than the range of the keys(r). In these cases, direct addressing leads to a huge amount of wasted space which requires us to look for more optimal hash functions.

Usually, hash functions map the range into a much smaller range, ideally proportional to the number of entries to be stored in the table. However, here we lose the direct addressing capability.

Division Method

In this method, a key k is mapped into one of the m slots by taking the remainder of k divided by m.

$$h(k) \ = \ k \, mod \, m$$

The advantage of this function is that it is extremely fast as it requires only one division operation. However, we must ensure that we don't pick certain values such as m = 2^p. When m is a power of 2, we end up taking only some of the least significant bits of the number and the other more significant bits to end up being dropped.

For example, suppose that your set of keys is 1, 3, 51, 80.

If we take m = 8, then the hashed values become:

$$\begin{aligned} h(1) &= 1\,mod\,8 = 8 \\ h(3) &= 3\,mod\,8 = 3 \\ h(51) &= 51\,mod\,8 = 3 \\ h(80) &= 80\,mod\,8 = 0 \end{aligned}$$

Here, the hashed values are just the least significant 3 bits of the key.

This leads to many clashes as the hash won't depend on all bits of k leading to more possible clashes.

Thus decent choices for m are usually primes, not too close to a power of 2 (or 10).

Multiplication Method

The multiplication method works in two steps. First, the key k is multiplied by a constant A in the range 0 < A < 1. Then the fractional part of the product kA is extracted as kA mod 1. Then, we multiply this value by m and take the floor of the result. Thus, the hash function is:

$$h(k) = \lfloor m\,(kA\,mod\,1) \rfloor$$

The advantage of the multiplication method is that, unlike the division method, there is no restriction on the value of m here. m can freely be taken as a power of 2 and thus the function can be efficiently implemented as shown in figure 2:

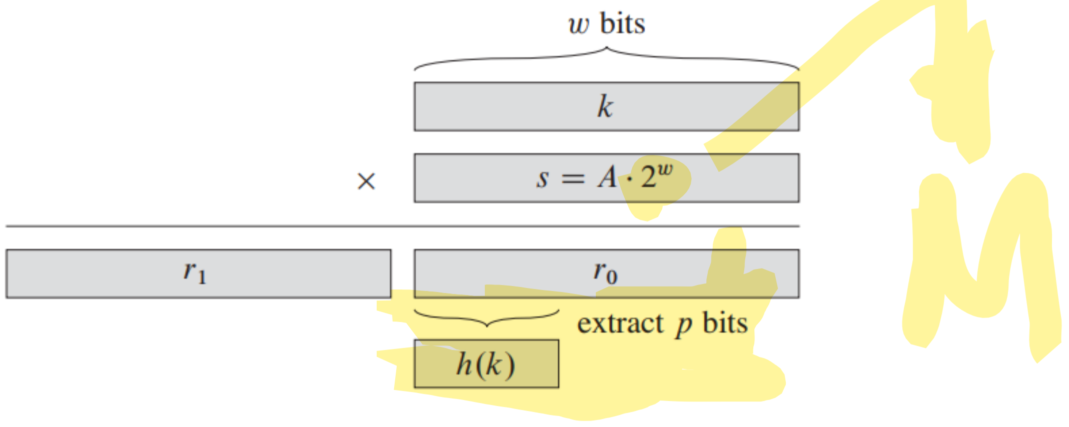

Figure 2: Illustration of Multiplicative Hashing

The w-bit representation of the key k is multiplied by the w-bit value s = A . 2^w . The p highest-order bits of the lower w-bit half of the product form the desired hash value h(k).

Although this method works with any value of the constant **A**, it works better with some values than with others. The optimal choice depends on the characteristics of the data being hashed. Knuth suggested using the value of **A** as the fractional part of the Golden Ratio:

$$A = (\sqrt{5} - 1) / 2 = 0.6180339887....$$

The basic idea is to have a irrational number that when multiplied with the key leads to a seemingly random fractional part. We take the p most significant bits of this fractional part and use it as our hashed value.

Multiplicative hashing can be implemented in C as follows:

```
int mulHash(int key, int size)
{
    double A = 0.6180339887;
    double frac = key * A - (int)(key * A);
    return (int)(size * frac);
}
```

Task 1: Create a struct that contains a long long int and a char array of size 10.

```
struct student
{
    long long int ID;
    char name[30];
};
```

Now, write a program that reads a text file that contains data in the following format (the file name is *t1_data.txt*):

```
Number_of_Students(N)
ID1,First_Name1 Last_Name1
ID2,First_Name2 Last_Name2
...
IDN First_NameN Last_NameN
```

The program should read the data from the file and store it in a hash table. Collisions are resolved by chaining.

The hash table should be of size 2*Number_of_Students.

While storing the data in the hash table, you should check for collisions and count the total number of collisions encountered.

Try the following hash functions and compare the number of collisions:
```
1. ID % size
2. ID % power_of_2_just_less_than_size
3. ID % a_prime_number_just_less_than_size
4. Multiplicative hash with A = 0.6180339887, m = size
5. ID % prime_less_than_0.9_times_size
6. (ID * a_prime_number_just_less_than_size) % size
```

The program should print the number of collisions for each hash function.

Interpreting keys as natural numbers

Most hash functions assume that the universe of keys is the set $N = \{0, 1, 2, \ldots\}$ of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers. For example, we can interpret a character string as an integer expressed in suitable radix notation. Thus, we might interpret the identifier "pt" as the pair of decimal integers (112, 116), since p = 112 and t = 116 in the ASCII character set; then, expressed as a radix-128 integer, pt becomes $128^1 * 112 + 128^0 * 116 = 14452$. In the context of a given application, we can usually devise some such method for interpreting each key as a (possibly large) natural number.

Home Exercise 1: Now for the data given in *t1_data.csv* in Task 1, perform the same hash table construction and comparison as described in Task 1, this time taking the name field to be the key. Apply the method described in the above section to convert the name to a natural number and then apply the 5 hash functions described in Task 1 on this new key.

Collision Resolution Techniques

Open Addressing

Open addressing is an alternative to chaining for handling collisions. Here all the keys are stored directly in the hash table. There are no linked lists present in the slots. Thus, in open addressing, the hash table can "fill up" which is not possible in chaining. We systematically store the colliding keys in other locations in a deterministic manner that makes it possible to examine the slots in the same order during lookup. This order is called the **probe sequence** for that element. Instead of having a fixed probe sequence for all elements which would lead to the same slots being examined always for lookup (linear search in an array), the sequence depends on the key to be inserted. Thus the probe sequence is a function of the key k.

Let the original hash function being used be h'(k) for the key k. This hash function can lead to collisions. So we define a new hash function h(k,i) that is a function of both the key k and the slot to be returned at the i^{th} probe. In this context, we call h'(k) as the auxiliary hash function and the hash function h depends on h'.

The look-up operation is also implemented differently in open addressing. In chaining, we could simply traverse up to the end of the linked list and see whether the element is present or not. However, in open addressing, we have to check the occupied elements of the probe sequence until we find either the required element, an empty slot or reach the end of the probe sequence (table completely filled, but element not found).

We face a minor issue with implementing deletion from an open address hash table. When we delete an entry, we cannot simply mark that slot as empty. This would interfere with our lookup logic where we were stopping when we reach an empty slot. If we did we would be unable to retrieve any key for which we had probed this location and found it occupied during insertion. We solve this problem by marking the slot as DELETED instead of NIL/EMPTY. Now we can continue to keep probing until an empty slot during look-up, however, we can stop at the DELETED slot when we are trying to find an unoccupied slot for insertion.

There are different ways of constructing h and consequently also the probe sequence. They are described below:

Linear Probing

The method of linear probing uses the following hash function:

$$h(k, i) = (h'(k) + i) \, mod \, m \text{ for i = 0, 1, 2, …, m-1}$$

Thus, initial probe i = 0, is the same as the auxiliary hash function h'(k). In case the slot is occupied, we successively probe the subsequent slots until we get an empty slot. Linear probing is easy to implement but suffers from a problem called **primary clustering**. There are long runs (clusters) of filled slots in the table leading to a large average search time.

Quadratic Probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 . i + c_2 . i^2) \, mod \, m$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and i = 0; 1. …, m-1. Note that only some values of c_1 and c_2 may work to be able to access all slots for a given m. Also, two elements having the same initial hash value have the same probe sequence. This issue is called **secondary clustering**.

Double Hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + i . h_2(k)) \, mod \, m$$

where both h_1 and h_2 are auxiliary hash functions.

The initial probe still goes to $T[h_1(k)]$. However, now the successive probes are offset by the amount $h_2(k)$ mod m. The function $h_2(k)$ must be designed such that it is always relatively prime to the hash table for the entire table to be searched. One way is taking m to be a power 2 and h_2 being an odd number.

Another way is taking m as a prime number and restricting h_2 to be less than m. For example, if m is prime we can take

$$h_1(k) = k \, mod \, m$$

$$h_2(k) = (k \, mod \, (m - 1)) + 1$$

Task 2: Now for the data given in *t1_data.csv* in Task 1, implement open addressing. Use a hash table of size 1024.

Create a `struct hash_table_element` that can store an instance of struct student and a field that signifies whether the slot is filed, empty or deleted (you can use an enum for this purpose).

Now create 4 programs where you initialise a hash table (array of hash_table_element) of the specified size and populate it with the data using the following different techniques respectively.

1) Linear Probing with auxiliary hash function h'(k) as the multiplicative hash function
2) Quadratic Probing with auxiliary hash function h'(k) as the multiplicative hash function
3) Double hashing with h_1 as the multiplicative hash function and $h_2(k) = 1 + (k \bmod 1023)$
4) Double hashing with $h_1(k) = (k \bmod 1024)$ and $h_2(k) = 1 + (k \bmod 1023)$

For each of the four programs, write the following methods:

```
#define TABLE_SIZE 1024
int hash(int ID, int m, int i);
void insert(struct hash_table_element *table, struct student *stud, int
m);
void delete(struct hash_table_element *table, int ID, int m);
struct student *search(struct hash_table_element *table, int ID, int m);
```

Now, you have been provided with a file called *queries.txt* that contains 100 different operations to be performed on the hash table.

The file has the following format:

```
op1 input1
op2 input2
…
op100 input100
```

Here, op is the operation to be performed and input is the input to the operation.
An operation is a number that specifies an operation (1 for Insert, 2 for delete, and 3 for search).

For insert operation, the input is of the following format: `ID,First_Name Last_Name`
For the delete operation, the input is the ID of the student to be deleted.
For the search operation, the input is the ID of the student to be searched.

The program should perform the operations in the order they are given in the file and at the end, print the total number of collisions for all the operations.

The program should also print the average number of collisions per operation.

Home Exercise 2: The first stage of a compiler is a lexical analyser (or "lexer" in short). It takes the source code and breaks it into tokens. The tokens are then passed to the parser, which builds a tree of the program. The tree is then passed to the code generator, which generates the machine code. The machine code is then passed to the linker, which links the code with any libraries that are needed. The linker then generates an executable file. The executable file is then passed to the operating system, which loads it into memory and executes it. This is a brief overflow of the compilation process.

Now, in the lexer, the tokens for symbols such as +, -, *, /, <, {, }, &, etc. can be generated in a straightforward manner while reading through the source code one character at a time. However, when we encounter words such as int, float, char, etc., we need to check if they are **keywords** or **identifiers** (because they follow the same lexical pattern). An identifier, or a name, refers to the class of strings that are used to identify some entity in a language. A reserved word is a special word of a programming language that cannot be used as a name. The keywords in C are reserved.

For example, in C, the words like `int`, `if`, `break`, etc. are keywords and cannot be used as identifiers. On the other hand, the words like `foo`, `bar`, `main`, `sum`, `i`, `j`, `printf`, `scanf`, etc. are allowed as identifiers.

So, in the lexer, we need to check if the word is a keyword or an identifier. This can be done by using a hash table. The hash table will have the keywords as keys. Then, when we encounter a word, we will check if it is a keyword or an identifier by looking it up in the hash table.

Given in the file *keywords.txt* are the keywords specific to the C programming language. You need to create a hash table of these keywords and then write a program that takes a word as input and checks if it is a keyword or an identifier.

Your hash table should have minimal collisions and the table size should be less than twice the number of keywords. Make use of open addressing to resolve collisions. Try different hash functions and probing techniques and minimize the number of collisions for each of them.

Bloom Filters

As we have seen, while hashing is a very powerful tool when our key space is much larger than our table size, we often get worse than O(1) look-up due to collisions in the table. We have to keep looking until we find the required element or know for sure that the element is not present. This strict requirement leads to the additional time required for lookup.

Consider the case where we want to create a set. If we create a hash set using the data, we would have a complexity of look-up proportional to the number of collisions for that element. However, if we relax the requirement of always getting the correct answer, allowing a small probability of error, we can get significantly better performance. This leads us to Bloom filters.

Standard Bloom Filters

A bloom filter is a probabilistic data structure that allows you to **efficiently**:

1. Insert an element into a set
2. Allow you to query whether an element is in a set

When the output for the second question is no, we know for sure that the element is not in the set. However, this data structure cannot answer "yes" to the query 'confidently'. The "yes" answer can be wrong. That is, the structure can have false positives, but it would never have false negatives. The false positivity rate is a function of the number of elements in the set, the size of the bloom filter and a parameter k, which is "the number of hash functions".

Let's look at how it works now. A Bloom filter is an array of m bits representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements. In the beginning, all m bits are zero. We have k hash functions, each of which has a range of (0, m-1). When we need to insert an element x_1 to the set, we hash it with each of the k hash functions and set the corresponding bits of the bloom filter to 1.

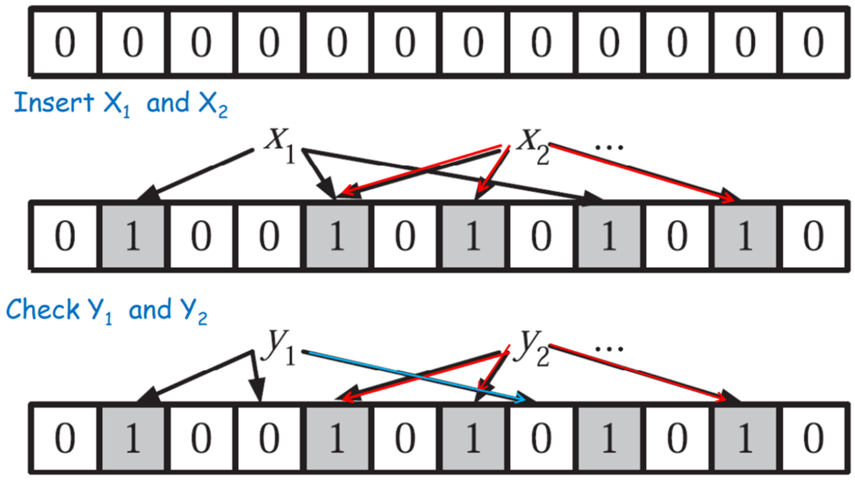

Figure 3. Insertion and look-up in a Bloom Filter

In Figure 3, we first insert elements x_1 and x_2 by setting the corresponding bits of the 3 hash functions to 1. Then we look up y_1 and y_2. For y_1, one of the bits is 0 and hence, we know that it is not present (true negative). Now y_2 is the same as x_2 and thus all the bits are 1, hence we get a "yes", y_2 is present. Now imagine we had a y_3 looked up with hash values 1, 4, and 6. Now while an element with these hash bits was not inserted, those bits are set to 1 independently due to other elements x_1 and x_2. Thus we get a false positive that the element is present when it actually is not.

Now consider we have a bloom filter of size 10. Now assume we have the three hash functions namely $h_1(k)$ = k mod 3, $h_2(k)$ = k mod 5, and $h_3(k)$ = k mod 7. Suppose we need to insert element 50 into this data structure.

So, $h_1(50) = 2$, $h_2(50) = 0$ and $h_3(50) = 1$

So we set the 0th, 1st and 2nd indices of the bloom filter.

Creating a bit array in C

We might be tempted to create an array of int in C, and set it to 0 or 1 based on whether we want the bit to be 0 or 1. However, this is extremely wasteful. An int in most systems is 32 bits and we are using 32 times the space we actually need to store a single bit. We need something better.

We can store more than a single element in one int. We can actually treat the single int as an array of 32 bits and create the bloom filter of n bits by just creating an integer array of $\lceil n/32 \rceil$ ints. (In general, $\lceil n/(8 * \text{sizeof(int)}) \rceil$ ints). We can set bits by using bitwise operations on the corresponding integers.

For example, consider the following line of code:

```
filter[j] = filter[j] | (1 << k);
```

This will set the k^{th} significant bit of the j^{th} integer.

We will use the following convention to map the bit indices to the corresponding int and bit indices:

- The first sizeof(int)*8 bits will be represented by the first int in the array.
- The next sizeof(int)*8 bits will be represented by the second int in the array.
- And so on.

Now in the first int, the least significant bit will be the first bit and the most significant bit will be the sizeof(int)*8th bit. Similarly, in the second int, the least significant bit will be the sizeof(int)*8+1th bit and the most significant bit will be the sizeof(int)*16th bit.

Thus, the bit at index i can be represented by the following:

```
int j = i / (sizeof(int) * 8); // This is the index of the int containing
the bit we want to set
int k = i % (sizeof(int) * 8); // This is the index of the bit we want to
set in the int
```

Now we can perform different operations on this bit in the following manner:

```
filter[j] = filter[j] | (1 << k); // Set the bit
filter[j] = filter[j] & ~(1 << k); // Clear the bit
int bit = (filter[j] >> k) & 1; // Get the bit
```

```
int *createBloomFilter(int size)
{
      int numIntsRequired = size % (sizeof(int) * 8) == 0 ? size /
(sizeof(int) * 8) : size / (sizeof(int) * 8) + 1;
    int *filter = (int *)malloc(numIntsRequired * sizeof(int));
    return filter;

}
```

```
void setBit(int *filter, int index)
{
     int i = index / (sizeof(int) * 8); // This is the index of the int
containing the bit we want to set
     int j = index % (sizeof(int) * 8); // This is the index of the bit we
want to set in the int
    filter[i] = filter[i] | (1 << j); // Set the bit
}

int checkBit(int *filter, int index) // Returns 1 if the bit is set, 0
otherwise
{
     int i = index / (sizeof(int) * 8); // This is the index of the int
containing the bit we want to set
     int j = index % (sizeof(int) * 8); // This is the index of the bit we
want to set in the int
    int bit = (filter[i] >> j) & 1; // Get the bit
    return bit;
}
```

Task 3: Create a bloom filter in C of size 256 bits. For this implement the insert and lookup methods that can set the corresponding bits while insertion and check them during look-up. You can use the setBit() and checkBit() functions defined above as subroutines in these functions.

Use the following 5 hash functions for the bloom filter:

h_1(k) = k % 256
h_2(k) = Multiplicative hash with size = 256, A = 0.6180339887
h_3(k) = k % 43
h_4(k) = (k ^ (k >> 3)) % 256
h_5(k) = 255 - (k % 197)

You have been given a file *bloom_numbers.txt* containing 50 keys to be inserted in the set. You have also been given the file *bloom_queries.txt* giving 100 lookups to be made to the set. Now you must find out the number of positive and negative outputs given by the bloom filter. In the queries file, 48 of the 100 lookups are actually present in the data file. Compare this with the results you obtain and find the false positivity rate of your implementation.

Counting Bloom Filters

With standard bloom filters, once inserted, deletion of elements is impossible because we cannot reset the bit as it might also be responsible for some other key. This leads us to counting bloom filters. In counting bloom filters. we expand each array element from a one-bit bucket to a multi-bit bucket. Thus, instead of the bit representing a binary 0 or 1, it is now a counter specifying how many hashes of present elements map to this location.

The insert operation is extended to increment the value of the buckets, and the lookup operation checks that each of the required buckets is non-zero. The delete operation then consists of decrementing the value of each of the respective buckets.

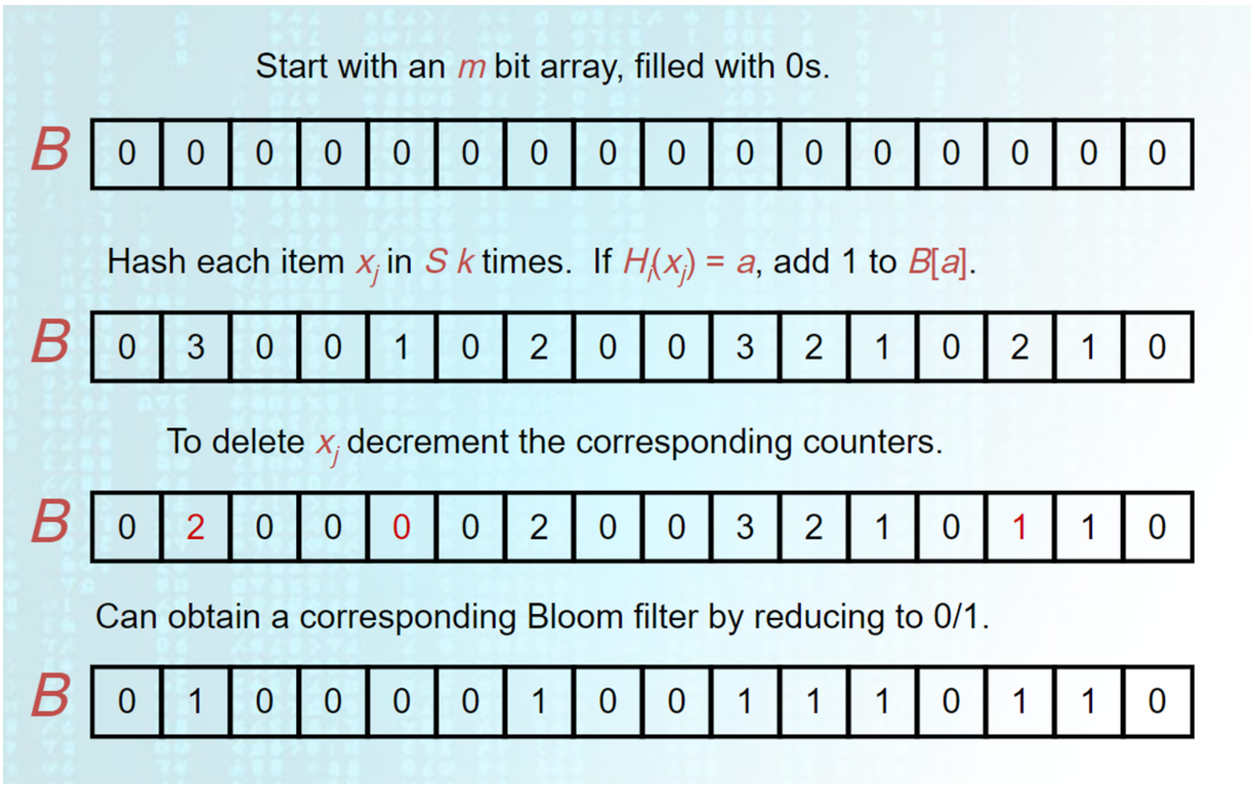
Start with an m bit array, filled with 0s.
B 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Hash each item x_j in S k times. If $H_i(x_j) = a$, add 1 to $B[a]$.
B 0 3 0 0 1 0 2 0 0 3 2 1 0 2 1 0
To delete x_j decrement the corresponding counters.
B 0 2 0 0 0 0 2 0 0 3 2 1 0 1 1 0
Can obtain a corresponding Bloom filter by reducing to 0/1.
B 0 1 0 0 0 0 1 0 0 1 1 1 0 1 1 0

Figure 4. Illustrating the working of Counting Bloom Filters

Arithmetic overflow of the buckets is a problem and the buckets should be sufficiently large to make this case rare. If it does occur it should revert to a standard bloom filter in this case, if any bucket reaches its maximum value, it cannot be incremented or decremented now.

Task 4: Generalise the bit-array implementation given above to create a 4-bit array. Implement the following functions to achieve the desired functionality:

1) `int *create4bitarray(int size)`: We want to use an array of integers to store an array of 4-bit counters. Thus, we would break up our integer in a similar way as we did in creating a bit array (createBloomFilter()). However, now we must do so such that blocks of 4 bits together represent each counter. So we would have ⌈4*n/(8 * sizeof(int))⌉ ints.

2) `int getCount(int *array, int index)`: This function would first extract the integer where the counter is present from the array. Then it would extract the counter (4 bits). [Hint: For extracting a 4-bit counter, think **bitwise and** with **15** ie $(1111)_2$. Also, you would need to left/right-shift to extract the correct counter as there are multiple counters present in a single integer]

3) `void incrementIndex(int *array, int index)`: It gets the 4 bits as in getCount(). Then it increments it if it is less than 15, flushes the old value of the count and sets the new value in the specific bits of the array.

4) `void decrementIndex(int *array, int index)`: Similar to incrementIndex(). Decrement the count if it is more than 0.

You can use the following main() to test out your implementation.

```
int main()
{
    int *array = create4bitarray(10);
    // Print the array
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", getCount(array, i));
    }
    printf("\n");
    incrementIndex(array, 0);
    incrementIndex(array, 5);
    incrementIndex(array, 5);
    incrementIndex(array, 0);
    incrementIndex(array, 9);
    // Print the array
    for (int i = 0; i < 10; i++)
    {
```

```c
        printf("%d ", getCount(array, i));
    }
    printf("\n");
    decrementIndex(array, 0);
    decrementIndex(array, 5);
    decrementIndex(array, 9);
    // Print the array
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", getCount(array, i));
    }
    printf("\n");
    incrementIndex(array, 3);
    incrementIndex(array, 3);
    incrementIndex(array, 0);
    incrementIndex(array, 0);
    incrementIndex(array, 0);
    incrementIndex(array, 0);
    incrementIndex(array, 0);
    incrementIndex(array, 1);
    incrementIndex(array, 2);
    for (int i = 0; i < 20; i++)
    {
        incrementIndex(array, 8); // Should saturate at 15
    }
    for (int i = 0; i < 20; i++)
    {
        decrementIndex(array, 1); // Should saturate at 0
    }
    // Print the array
    for (int i = 0; i < 10; i++)
    {
        printf("%d ", getCount(array, i));
    }
    printf("\n");
    return 0;
}
```

The expected output is:

```
0  0  0  0  0  0  0  0  0  0
2  0  0  0  0  2  0  0  0  1
1  0  0  0  0  1  0  0  0  0
6  0  1  2  0  1  0  0  15  0
```

Task 5: Now use the array of 4 bit counters implemented in the previous task to implement a counting bloom filter for the same setting as in Task 3. Use a counting bloom filter with 256 counters. For this implement the insert, delete and lookup methods. You can use the incrementIndex() and decrementIndex() functions defined above as subroutines in these functions.

Use the following 5 hash functions for the bloom filter:

h_1(k) = k % 256
h_2(k) = Multiplicative hash with size = 256, A = 0.6180339887
h_3(k) = k % 43
h_4(k) = (k ^ (k >> 3)) % 256
h_5(k) = 255 - (k % 197)

Use the file *bloom_numbers.txt* as in Task 3 to populate the bloom filter initially. Now, perform the queries specified in *count_bloom_queries.txt.* The format of count_bloom_queries.txt is as follows:

```
op1,num1
op2,num2
…
op100,num100
```

where op can be 1 (insert), 2 (delete) or 3 (query) and num is the key the operation is being performed on. Perform the queries on your bloom filter in the order specified in the file.

The given file has 35 lookup queries out of which 20 are actually not present in the set. Count the number of positive and negative responses output by your bloom filter and find its false positivity rate.