Hash tables

Hash tables naively

A hash table implements a set or map

The plan: take an array of some size k

Define a *hash function* that maps values to indices in the range {0,...,k-1}

- Example: if the values are integers, hash function might be $h(n) = n \bmod k$

To find, insert or remove a value x, put it in index $h(x)$ of the array

- Avoid searching through the whole array!

Hash tables naively, example

Implementing a set of integers, suppose we take a hash table of size 5 and a hash function $h(n) = n \bmod 5$

0	1	2	3	4
5		17	8	

This hash table contains {5, 8, 17}

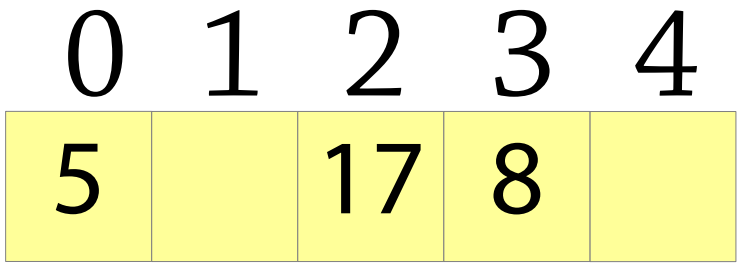
Inserting 14 gives:

0	1	2	3	4
5		17	8	**14**

Similarly, if we wanted to find 8, we would look it up in index 3

A problem

This idea doesn't work.
What if we want to insert 12 into the set?



We should store 12 at index 2, but there's already something there!

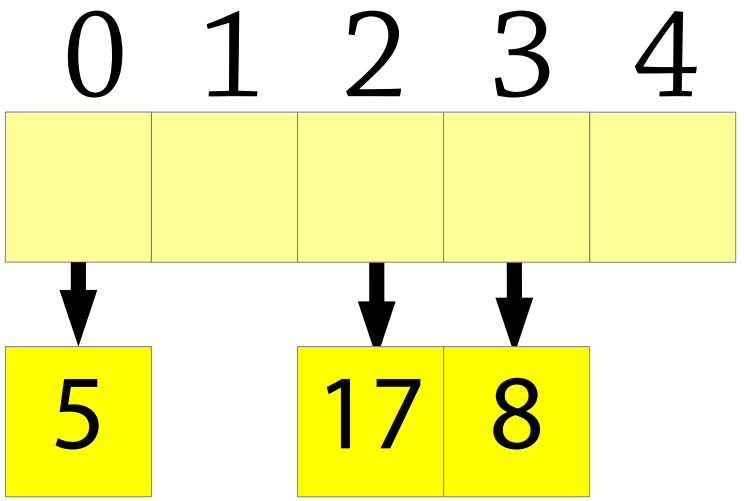
This is called a *collision*

Real hash tables are naive hash tables plus tricks for dealing with and avoiding collisions!

Handling collisions: chaining

Instead of an array of elements, have an array of *linked lists* (chains)

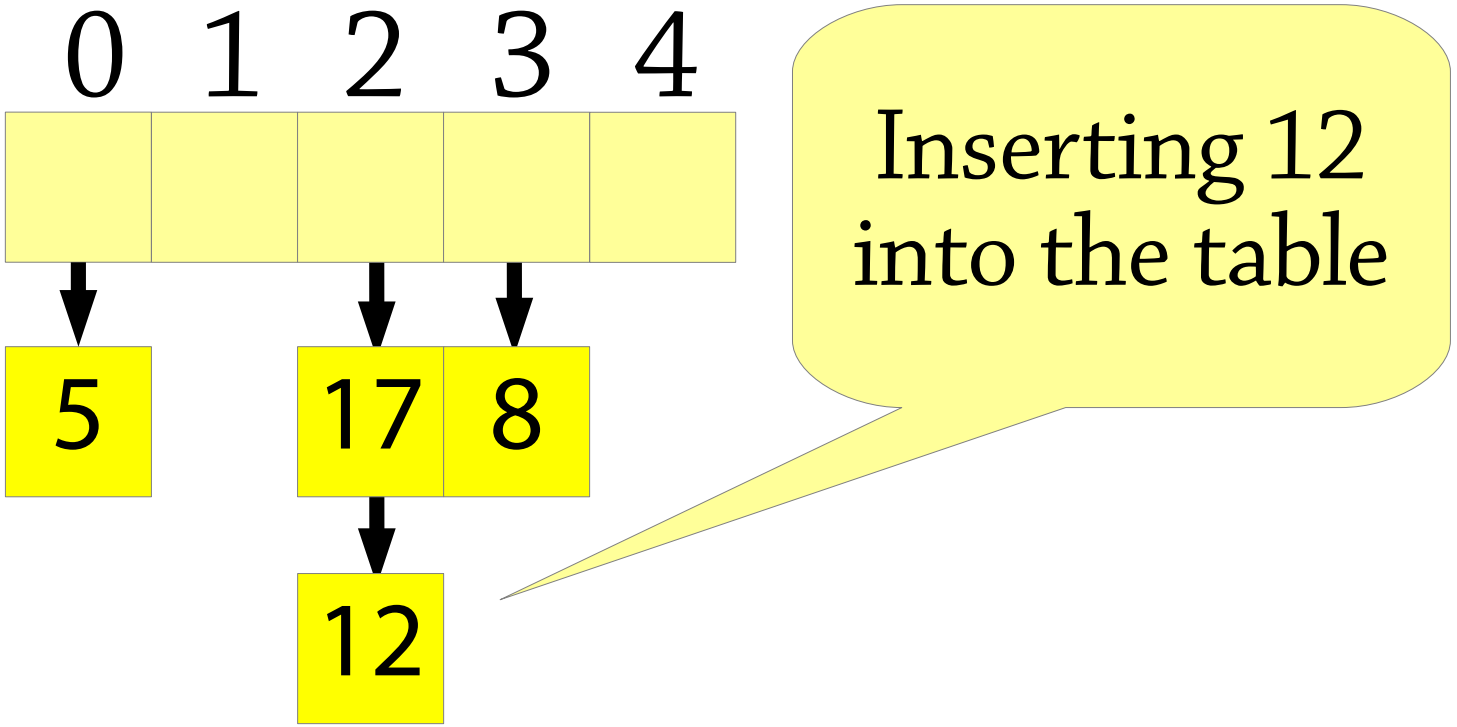
To add an element, calculate its hash and insert it into the list at that index



Handling collisions: chaining

Instead of an array of elements, have an array of *linked lists* (chains)

To add an element, calculate its hash and insert it into the list at that index



Performance of chained hash tables

Chained hash tables are fast if the chains are small

- If the size is bounded, operations are O(1) time

But if the chains get big, everything gets slow

- Can degrade to O(n) in the worst case

There are two cases when this can happen! We have to avoid both of them.

Performance of chained hash tables

Case one: the hash table is too full

- If we try to store 1,000,000 values in an array of size 5, some chains will be 200,000 long

Solution: expand the hash table

- If the hash table gets too full (a high *load factor*), allocate a new array about twice as big (*rehashing*)
- load factor = number of elements / size of array

Problem: h(x) is specific to a particular size of array

- Allow the hash function to return an arbitrary integer (the *hash code* of *x*) and then take it modulo the array size:
  h(x) = x.hashCode() mod array.size
- Hash function of an integer will just be the integer itself

Performance of chained hash tables

Case two: the hash function is lousy

- Worst case: h(x) is a constant function, e.g. h(x) = 0
- Then all elements will end up in the same chain!

The hash function must distribute values evenly

- Each hash bucket has an equal chance of being chosen
- There are no observable patterns, e.g., easy ways to construct two values which always have the same hash

In other words, it should look like the hash function returns a *random* bucket

Chained hash tables – the theory

We need:

- to resize the hash table when it gets too full
- a hash function which appears to be random (no patterns, equal distribution)

If we do that, the average chain size will be constant and we get *expected* O(1) performance for insert/lookup/delete!

- Complexity analysis uses probability theory

When should we resize the hash table?

- If the load factor is 3 (number of elements = array size × 3), each operation needs on average ~2.5 comparisons
- Pick some constant load factor, resize when it reaches that

A slightly awkward problem

In reality, the hash function does not return a random hash code!

- Common hash functions can have patterns

This breaks the nice theory we have. Here is one problem:

- If we double the size of the array when resizing, the array size will always be even
- If we then insert only even numbers into the hash table, only the even buckets will be used

To fix this, we make the array size always be a *prime number* (while roughly doubling it each time) – this masks patterns in the hash function

Chained hash tables – summary

Start with a naive hash table

Add chaining

Double the size of the array when the load factor is too high...

- ...but make sure the array size is always prime

Now you have a chained hash table!

- O(1) expected complexity for all operations

But how should we design hash functions?

Designing hash functions

A good hash function should distribute values evenly

- h(x) has a roughly equal chance of being any particular number
- That way, all chains will be roughly the same length!
- We want to avoid having a huge number of collisions

Defining good hash functions is a black art!

- Weird heuristics that are semi-backed-up by theory

We'll settle for: unlikely to insert many elements with the same hash

Defining a good hash function

A general property of good hash functions is: it should be hard to accidentally construct hash collisions

- In particular, similar values should not have the same hash

What is bad about the following hash function on strings?

> *Add together the character code of each character in the string*
> (character code of a = 97, b = 98, c = 99 etc.)

Maps e.g. *bass* and *bart* to the same hash code! (s + s = r + t)

Any anagrams will have the same hash code

A hash function on lists of digits

Let's start with a simpler problem, defining a hash function for a sequence of digits from 0-9:

- [0,0,9,3,4,2,1] etc.

Idea: write out the digits as a single number with a leading 1:

- hash([0,0,9,3,4,2,1]) = 10093421
- Without the leading 1 we would get the same hash for e.g. [0,1] and [1]

Formally, the hash would be:

$$10^n + d_0 \cdot 10^{n-1} + d_1 \cdot 10^{n-2} + \ldots + d_{n-1}$$

where d_i is the *i*th digit of the number

This hash function makes sure that different lists have different hashes – good!

Let's take this and apply it to strings...

A hash function on strings

An idea: map strings to integers as follows:

$$128^n + s_0 \cdot 128^{n-1} + s_1 \cdot 128^{n-2} + \ldots + s_{n-1}$$

where s_i is the code of the character at index i

If all characters are ASCII (character code 0 – 127), each string is mapped to a different integer!

The problem

For performance, we will calculate the hash using machine integers so the calculation

$$128^n + s_0 \cdot 128^{n-1} + s_1 \cdot 128^{n-2} + \ldots + s_{n-1},$$

will happen modulo 2^{32} or 2^{64} (*integer overflow*)

So the hash will only use the last few characters!

Solution: replace 128 with another number, e.g. 33

$$33^n + s_0 \cdot 33^{n-1} + s_1 \cdot 33^{n-2} + \ldots + s_{n-1}$$

This is (almost) what Java uses for strings

Hashing composite values

How can we hash an object with multiple fields?

```
class C { A a; B b; }
```

Use the same approach as for strings!

$$33^2 + 33 \times \text{hash}(a) + \text{hash}(b)$$

This comes out quite nicely in code too:

```
int hash = 1;
hash = hash*33 + a.hashCode();
hash = hash*33 + b.hashCode();
```

It's also available in `java.util`:

```
int hash = Objects.hash(a, b);
```

Hash functions

This is called *Bernstein hashing*, it's only one way of defining hash functions

- Bernstein discovered that using 33 as the constant gives good distribution
- Why? Nobody knows!

Many hashing techniques use similar tricks that are used in random number generators and cryptography

- The output of a good hash function just needs to look random
- The output of an encryption algorithm should also look random
- ...as should the output of a random number generator :)

Best to pick an existing technique and just use it; leave the design of new hashing techniques up to the cryptography wizards!

Linear probing

Another way of dealing with collisions is *linear probing*

Uses an array of values, like in the naive hash table

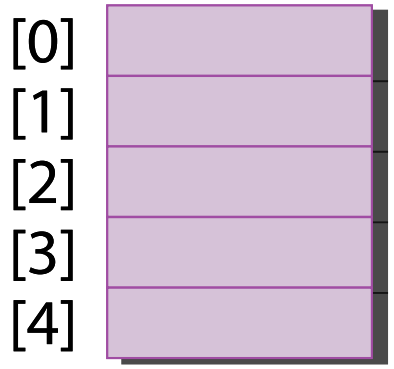
If you want to store a value at index *i* but it's full, store it in index *i+1* instead!

If that's full, try *i+2*, and so on

...if you get to the end of the array, wrap around to 0

Example of linear probing

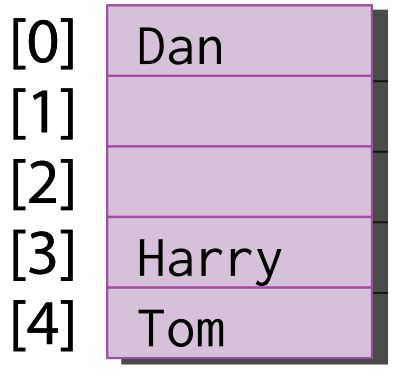
```
Tom  Dan  Harry Pete
```



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

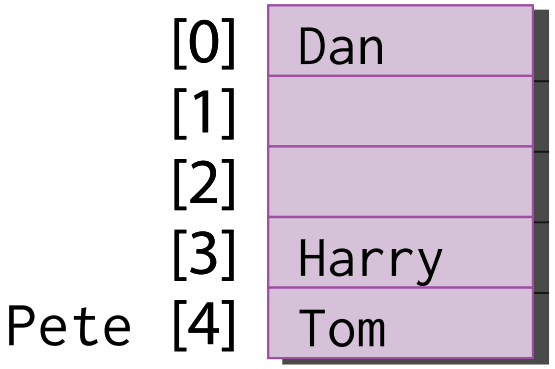
Example of linear probing

Pete



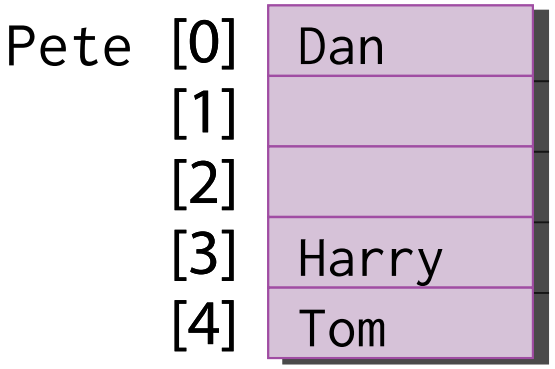
Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing



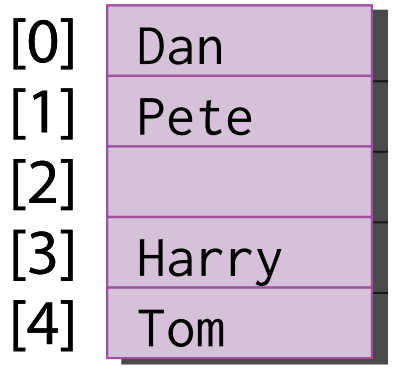
Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

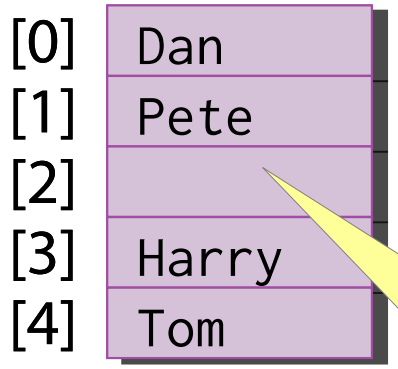
Example of linear probing



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing

Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



To find "Pete" (hash 3), you must start at index 3 and work your way all the way around to index 1

Searching with linear probing

To find an element under linear probing:

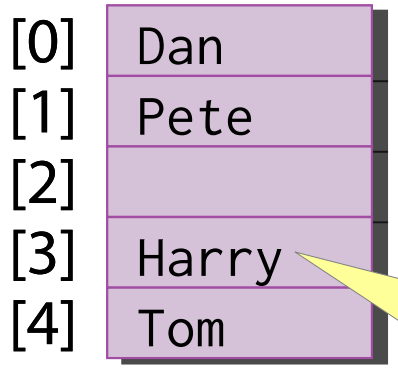
- Calculate the hash of the element, *i*
- Look at *array[i]*
- If it's the right element, return it!
- If there's no element there, fail
- If there's a *different* element there, search again at index *(i+1) % array.size*

We call a group of adjacent non-empty indices a *cluster*

Deleting with linear probing

Can't just remove an element...

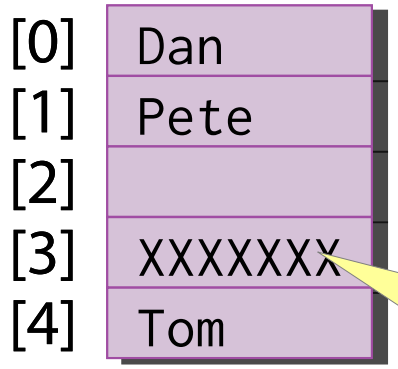
Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



If we remove Harry,
Pete will be in the wrong cluster
and we won't be able to find him

Deleting with linear probing

Instead, mark it as deleted (*lazy deletion*)



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

The search algorithm should skip over XXXXXXX

Deleting with linear probing

It's useful to think of the invariant here:

- Linear *chaining*: each element is found at the index given by its hash code
- Linear *probing*: each element is found at the index given by its hash code, *or a later index in the same cluster*

Naive deletion will split a cluster in two, which may break the invariant

Hence the need for an empty value that does not mark the end of a cluster

Linear probing performance

To insert or find an element under linear probing, you might have to look through a whole cluster of elements

Performance depends on the size of these clusters:

- Small clusters – expected O(1) performance
- Almost-full array – O(n) performance
- If the array is full, you can't insert anything!

Thus you need:

- to expand the array and rehash when it starts getting full
- a hash function that distributes elements evenly

Same situation as with linear chaining!

Linear probing vs linear chaining

In linear chaining, if you insert many values with the same hash, values with that hash become slower to access but other hashes are unaffected

In linear probing, you get a cluster and values with *nearby* hashes become slower to access too!

- Many different hash values end up part of the same cluster

As the array gets close to 100% full, you get very long clusters in the hash table and performance becomes dreadful

Linear probing needs a bigger array than linear chaining for the same performance

But: as you don't need to also create list nodes, you can create a bigger array in the same amount of memory

Probing vs chaining

25% of space wasted (three full buckets for each empty bucket)

load factor (#elements / array size)	#comparisons (linear probing)	#comparisons (linear chaining)
0 %	1.00	1.00
25 %	1.17	1.13
50 %	1.50	1.25
75 %	**2.50**	1.38
85 %	3.83	1.43
90 %	5.50	1.45
95 %	10.50	1.48
100 %	—	1.50
200 %	—	2.00
300 %	—	**2.50**

At least 50% space wasted (each list node contains a ”next” pointer)

Summary of hash table design

Several details to consider:

- *Rehashing*: resize the array when the load factor is too high
  - Keep the size a prime to mask patterns in the hash function
- *A good hash function*: need an even distribution
  - Consider using Bernstein hashing or java.util.Objects.hash in your own code, rather than rolling your own
- *Collisions*: either chaining or probing
  - Other alternatives to linear probing, e.g. quadratic probing, Robin Hood hashing
  - Both options are popular in hash table libraries

In return:

- *Expected* (average) O(1) performance if the hash function is random (there are no patterns)
- Better performance in practice than BSTs
- Disadvantage: hash tables are *unordered* so you can't get the elements in increasing order

Theoretical foundations of hash *functions* are a bit uncertain, but heuristics work well in practice

Bloom filters
(not on exam)

Bloom filters

Suppose we want a data structure for a set of values, but we don't have enough memory to store all the values.

Sounds hopeless doesn't it?

With a *Bloom filter* we can get a set which supports:

- Insertion (not deletion)
- Membership testing with *false positives*:
  if it says yes, it might not be in the set,
  but if it says no it's definitely not in the set

By increasing the amount of memory used, we can get the false positive rate arbitrarily low

- 1% false positive rate using 10 bits per element

Why?

One example:

- A spellchecker
- Dictionary is too big to fit in memory
- Use a Bloom filter, accept occasional misspellings

A pre-filter for an on-disk map:

- where we expect many searches for values that are not in the map
- Check the Bloom filter first, if it says no, the value is definitely not in the map
- Otherwise, check the on-disk map

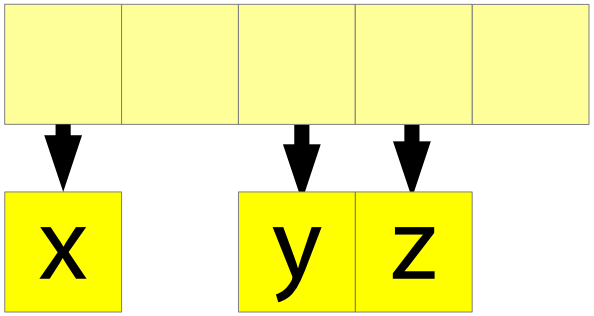
Special-purpose but pretty cool

Naive Bloom filters

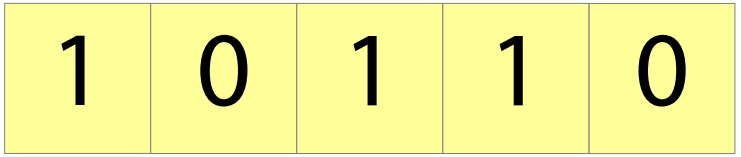
Take a hash table – but instead of storing a chain of values, each bucket stores a single *bit*

- 1 means: there is a value in this bucket

A chained hash table:



The corresponding naive Bloom filter:



Naive Bloom filters

A naive Bloom filter is an array of m bits

- The size of the array is chosen when the Bloom filter is created, and remains fixed

Initially, all bits are set to 0

To insert *x* into the Bloom filter:

- Calculate *h(x) mod m*, as in a hash table
- Set that bit in the array to 1

To check if *x* is in the Bloom filter:

- Calculate *h(x) mod m*
- Check if that bit in the array is 1

Just the same as chained hash tables, but with a single bit per bucket instead of a chain!

Naive Bloom filters – false positives

Suppose that (e.g.) half of the bits in the array are set to 1

Then looking up a value which is not in the Bloom filter, there is a chance of 50% it returns true anyway

Not so good!

If we want a 1% false positive rate, we'd need only 1% of the bits in the array set to 1

- So we need about 100 bits of memory per item inserted

We can do better!

Bloom filters

Have *several* hash functions $h_1 \ldots h_k$

As before, we have an array of size m

To insert *x* into the Bloom filter:

- Calculate $h_1(x) \bmod m$, $h_2(x) \bmod m$, ..., $h_k(x) \bmod m$
- Set all those bits to 1!

To search for *x* in the Bloom filter:

- Calculate $h_1(x) \bmod m$, $h_2(x) \bmod m$, ..., $h_k(x) \bmod m$
- Return true if all those bits are 1!

Bloom filters – false positives

Suppose that half of the bits in the array are set to 1, and we have k hash functions

Now suppose we look up a value *x* which is not in the Bloom filter

What is the chance of a false positive?

- There is a 50% chance that $h_1(x) \bmod m = 1$
- and a 50% chance that $h_2(x) \bmod m = 1$
- …
- and a 50% chance that $h_k(x) \bmod m = 1$

The chance that they all return 1 is one in 2^k!

Bloom filters – performance in practice

To get a Bloom filter with a false positive rate of p:

- Choose the number of hash functions, k, so that $1/2^k < p$.
- Choose the size of the array so that at most half of the bits are 1.
  If n is the number of items to be stored in the Bloom filter, then there are at most kn bits, so 2kn bits are enough (in fact a bit less because of hash collisions).

Note that the required number of bits grows:

- linearly in the number of items stored in the Bloom filter
- logarithmically in the false positive rate!

Some calculations:

- 1% false positive rate: 7 hash functions, 10 bits per item inserted
- 0.01% false positive rate: 13 hash functions, 20 bits per item inserted

So this is very space-efficient!