# Data Structure
# Programming Project #4

郭建志

# Imagine

- You want to test whether an element is a member of a set
- In the set → return true
- Not in the set → return false

- You don't need exact results
- That is, some false positives are allowed

# Usage

- Ticket booking
- Matchmaking
- Dating services



tinder

Source:
https://tixcraft.com/activity/detail/20_MAYDAY
https://en.wikipedia.org/wiki/FIFA_21
https://en.wikipedia.org/wiki/Tinder_(app)

# Problem

- Given:
- Keys that are input in sequence

- Goal
- Check whether the key has been examined
- Bounded error is allowed

- Constraint:
- Limited storage and limited computation

# Simple Solutions

- Construct a set of elements
- Implement a binary search tree in C
- Use a balanced binary tree:
  `set` in C++
- Hash table:
  `unordered_map` in C++
- You may want to use libraries Guava or FastUtil in Java for convenience and better performance

# Problem

- The number of of distinct elements might be very large
- You have a limited memory space
- For real-time applications you need runtime guarantees
- Binary search tree requires $O(\log n)$ time, which is not constant

# Solution: The Idea of Bloom Filter

- The trick: don't store the distinct elements, but just a fixed number of bits
- Create a bit array of length x initially filled with false values (or 0s)
- Each incoming element gets mapped to a number (i.e., an index) between 0 and x
- The corresponding bit in the array is set to true
- To query an element's bit, simply return the bit value at it's position

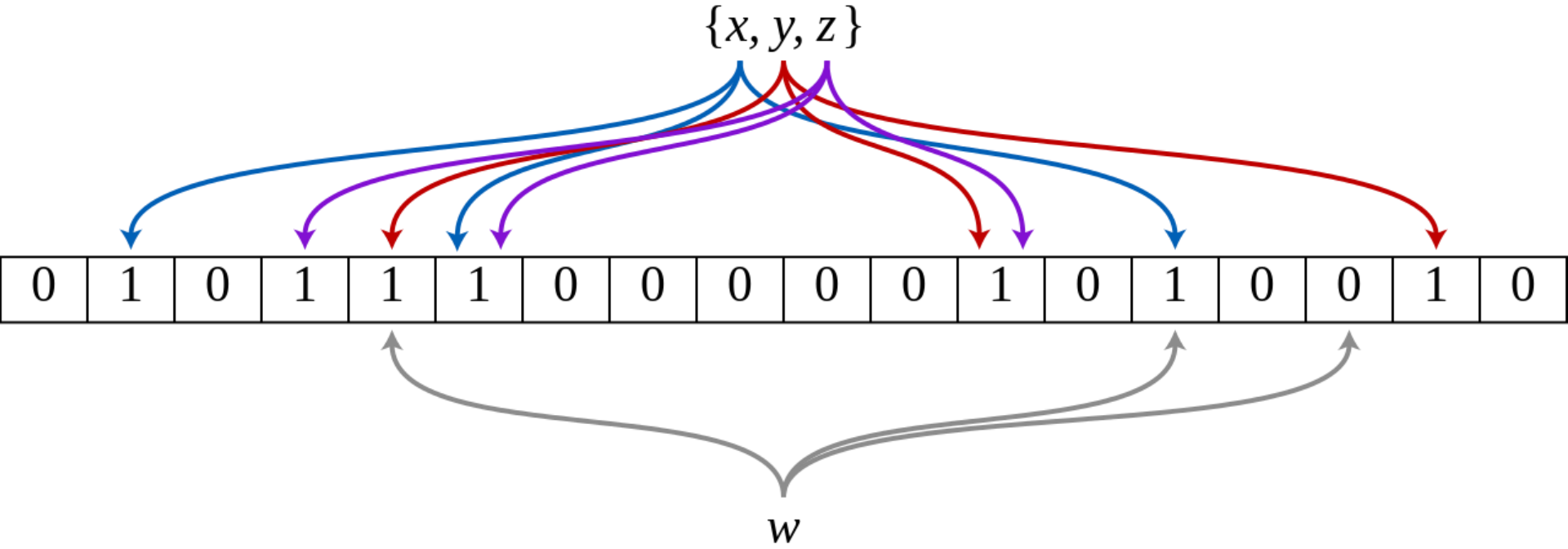
# Solution: The Idea of Bloom Filter

- The trick: don't store the distinct elements, but just a fixed number of bits
- Create a bit array of length x initially filled with false values (or 0s)
- Each incoming element gets mapped to a num
- The o true
- To query an element's bit, simply return the bit value at it's position

You are completely right:
There will be collisions!

# Solution: Bloom Filter

- Use multiple hash functions to compute the indices of multiple positions for each element
- When w is queried, return true if all the bits are true



# Solution: Bloom Filter

- Use multiple hash functions to compute the indices of multiple positions for each element
- When w is queried, return true if all the bits are true

You are completely right:
There will still be collisions!

# Solution: Bloom Filter

- Use multiple hash functions to compute the indices of multiple positions for each element
- When w is queried, return true if all the bits are true

You are completely right:
There will still be collisions!
… but less

# Some properties

- Only false positive, never false negative
- Has a constant memory and time consumption independent of the number of elements
- Has a lower false positive rate compared to the method with only one hash function

# You need to implement:

**void init(bool **bits, int m, int r, int **a, int **b, int p)**

1. Create an array with m bits for bits
2. Create an array with r elements uniformly chosen from [1, p-1] for pointer a using srand(1) (hint: use rand())
3. Create an array with r elements uniformly chosen from [1, p-1] for pointer b using srand(2) (note: a[i] and b[i] should be independent)

**int myhash(char *str, int count, int m, int r, int p, int *a, int *b)**

1. Use hash in <string> to covert str to an integer key

// You may use class string and class hash <string>

// note that 0 <= count <= r-1

2. Return (a[count] * key + b[count]) % p % m;

**void insert(bool *bits, int m, int r, int p, char *str, int *a, int *b)**

1. Find all the mapped bits in the following positions, bits[myhash(str, count, m, r, p, a, b)] for 0 ≤ count ≤ r-1
2. Set all of the mapped bits above to true

**bool query(bool *bits, int m, int p, int r, char *str, int *a, int *b)**

1. Examine whether all the mapped bits are true

bits[myhash(str, count, m, r, p, a, b)] for 0 ≤ count ≤ r-1

# Input Sample

```
50 10 10 3 1019
data
structures
serve
as
the
basis
for
abstract
data
type
type
data
object
```

```
#bits #hash_function
#words #tests prime
word1
word2
Word3
…
test1
test2
…
```

# Output Sample

```
type: true
data: true
object: false
```

```
test1: bool
test2: bool
test3: bool
…
```

Note: The bool values are **allowed** to have partial errors to some extent, since you are using a bloom filter instead of a binary search tree

# Note

- Superb deadline: 12/31 Thu
- Deadline: 1/7 Thu

- You are not allowed to use ”class” in STL to count the words
- You must implement a bit-array with the given size (i.e., #bits of input) to count the words

- **E-course**

- **C++ Source code**
  **(but only use C code unless hash < string>)**