Hash tables

Hash tables

- Hash tables are very commonly used data structures.
- They provide very fast access (if things are done right ☺…more on that soon)

- Essentially this is **content addressable** data structure.
- E.g.:
    - A regular array (not a hash table) is indexed by an integer which represents its position in the array.
    - int [] myArray;
    - int x = myArray[1];

    - Hash table:
    - Assume we store some integer ID in the hash table. This is how we would access it:
        - int johnID = myHashTable[“john”]; ← indexing by a key (in this case, a string key)
        - int doeID = myHashTable[“doe”];

- How do we make this work?
    - As in, how do we make it such that a string is an index, not an integer.

- The key used to access the hash table is converted to an integer index into the hash table.
- Think of the hash table as just a regular array, and once we convert the key to an index (integer), we access the contents at that index in the array.

Imagine a function:

```
GetElement ( string key )
        indexIntoTheHashTableArray = ComputeIndex( key )

        return hashTableArray [ indexIntoTheHashTableArray ]
```

- Converting the key to an integer index is known as *hashing the key*, or getting its hash value.
- For this, we use a hash function.

- For example:
- Lets look at a very simple hash function… Note this is shown only for example purposes here, a hash function used in a hash table would have to be *better* than this:
  - hashFunction(“john”) could do something as simple as following:
    - Add the ascii values of the key and then mod that with the hash table size.
    - So, above call would essentially do the following:
      - (‘j’ + ‘o’ + ‘h’ + ‘n’) % hashTableSize $\leftarrow$ This gives u the index into the array (hash table)
        - Result of this would be between 0 to (hashTableSize – 1)
      - Question: Why do we do a mod with hash table size?
- hashFunction(“nhoj”)
- hashFunction(“jnoh”)
- hashFunction(“kmoh”)

- (‘j’ + ‘o’ + ‘h’ + ‘n’) % hashTableSize

106 + 111 + 104 + 110 = 431

431 % 7 = 4

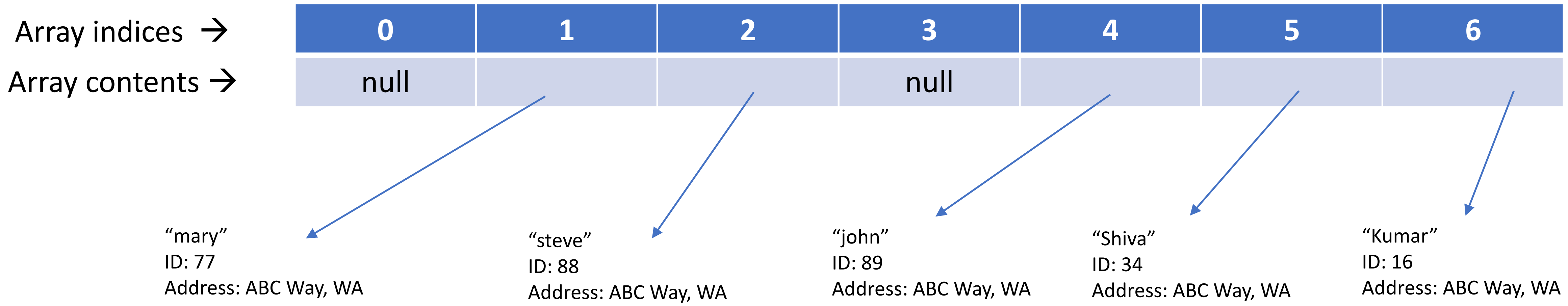
- So, the entry for “john” would be at index 4 in the hash table

- The table size should ideally be a prime number.

Here's a hash table of size 7

The numbers in blue show the indices of the hash table (the array).

The key used in this hash table will need to hash to one of the indices.
Then we access that index in the hash table, and get the value there.



nulll indicates an empty slot in the array.
So, indices 0 and 3 do not have anything in it.

- So, this is what we need to do in order to access an element in the hash table:
  - Hash the key to get an integer (hash value, which is an integer)
  - (Hash value % hash table size) is the index in the hash table (array)
  - If there is a value in that position, **and** its key is the same as what you are looking for, then you found it.

Lets look at these steps:

1. Hash the key:
   - Mathematical operation.
   - This is **constant** time. So, O(1)

2. Index into the hash table with index as (Hash value % hash table size)
   - Since we have an array, this is also **constant** time.

3. Check the key at index (if not null), to make sure we have the key we are looking for.
   - We will consider this as **constant** time as well, with respect to number of entries in the table.

4. So, the complexity of accessing something in the hash table is O(1).

5. If we had to write and entry into the hash table, it would be the same complexity.

Collisions

- What happens when there is a collision in the hash table?

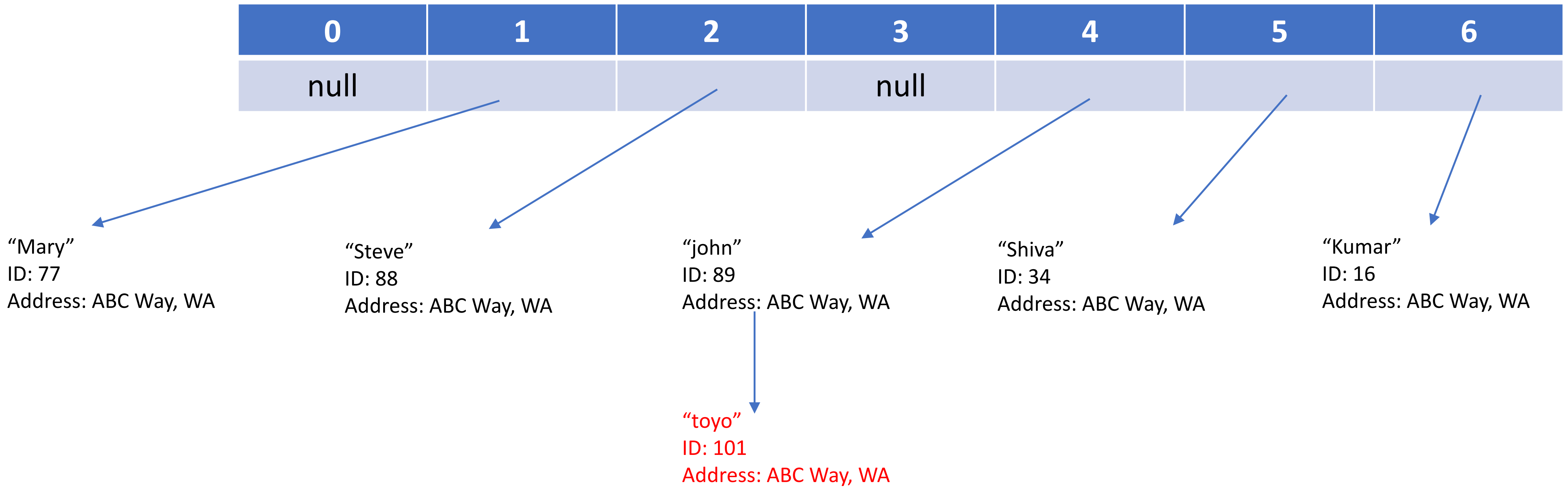
- Since we are computing an integer value from a key, it is certainly possible that more than one keys hash to the same integer value. This is a **collision**.

  - Chances of this happening are higher if:
    - Hash function is not great.
    - Table is not big enough compared to number of entries we want to store (load factor).
      - We will talk about load factor in a few slides after we look at collision resolution.

  - There are different ways to handle a collision.
    - Chaining.
    - Linear probing.

Chaining

- In chaining, you maintain a list at **each index** in the array, and entries with the same hash value are added to the list at that index.
- Lets take the key to be “toyo”
- (‘t’ + ‘o’ + ‘y’ + ‘o’) % hashTableSize
- 116 + 111 + 121 + 111 = 459 % 7 = 4



Chaining

- When you have chaining in a hash table, the look up time for entries that are being chained will increase.

- Worst case:
    - Lets say we have a really bad hash function that returns the same hash value for all keys.
        - Or
    - a hash table of size 1
        - (Both of these are highly unlikely in the real world).
    - Then all entries would hash to the same hash value(index), and you would essentially have a list at that index.
    - This would then have a search complexity of O(N) … linear…Oops

Linear probing

- Insertion:
  - When inserting a key k at index i:
    - If the table contains a **different** key at that index, then we look at the next index (i+1) or (i+2), etc. until we see an empty slot.

- Searching:
  - During a search for key k at index i:
    - If index i contains a **different** key, then we look at indices i+1, i+2, and so on, until we either find the key, or an empty slot is seen.

Load factor

- Load factor α is defined as
  - α = N / hashTableSize
    - If N = 75, hashTableSize = 100, LoadFactor = 0.75
  - Typical range of α is from 0 (empty) to 1 (full), so
    - 0 <= α <= 1
    - but α can be > 1
    - When using chaining, α can be > 1
      - although ideally we don't want it to be > 1 for O(1) searches
    - When using linear probing, max value of α is 1.
      - What I mean here is that if α is 1, then no more entries can be stored in the hash table using linear probing, since none of the slots in the table are empty.

Load factor

- As α approaches 1, the number of empty spaces in the table decrease, and if using linear probing, then collision probability increases.
- If α == 1, it means the table is full.

- So, size of the table is an important factor in hash table performance, because if it is too small and load factor is high, then collision probability increases.

- Does this mean we should aim for a very low load factor, say, 0.25 or so?
  - If yes, why?
  - If no, why?

  - [redacted]
  - [redacted]

- Now, when α increases, we may need to increase the size of the hash table.
- This means allocating a bigger hash table.
- However, we cannot just allocate a bigger table and copy elements from the original hash table.
  - Can someone guess why?

Rehashing

- For populating the new bigger hash table, we need to:
  - For each element in the original hash table
    - Generate the element’s key’s hash value. (Why do we need to generate this again?)
      - The hash value doesn’t change due to resizing if the table, but we generate again because we didn’t store the hash value anywhere.
    - Determine newIndex = hash value % newTableSize
    - Insert at newIndex in the new table.

Hash function

- Hash function is an important part of hash tables.
  - It should give a “good” distribution of keys in the table.
  - It should be deterministic, i.e., passing it the same key should return the same hash value.
- In Java, all objects contain a hashing function:
  - public int hashCode();
- In your class, you can write your own hash code function.

Hash function

Then we would have something like:

```
int MyHashFunction( Employee  c )
{
        return c.hashCode();                    // assumes c is not-null.
        OR
        return hashFunction( c.GetKey() );
}
```

- **<u>Semantically</u>**, we do the following for the various actions in a hash table:
  - *Adding an element: hashTable[“john”] =*
    - hashTable[MyHashFunction(e) % TABLE_SIZE] = e;
      - First check for collisions.
  - *Searching*
    - hashTable[MyHashFunction(e) % TABLE_SIZE].equals(e);
      - First check if hashTable[MyHashFunction(e)] is non-null
  - *Removal*
    - hashTable[MyHashFunction(e) % TABLE_SIZE] = null;
      - First check if hashTable[MyHashFunction(e)] .equals(e)

Open addressing

- Open addressing is a way to handle collisions

- Linear probing that we looked at is a form of open addressing.

- Other techniques that fall under open addressing
    - Quadratic probing
    - Double hashing

LAB

- Write a function hash(key) that returns the hash value (as an integer) of the string key.
- Write a function index(hashValue, size) that converts hashValue into an index into an array of size size.
- Implement an array based Map using the hash and index functions. Use a fixed size array.