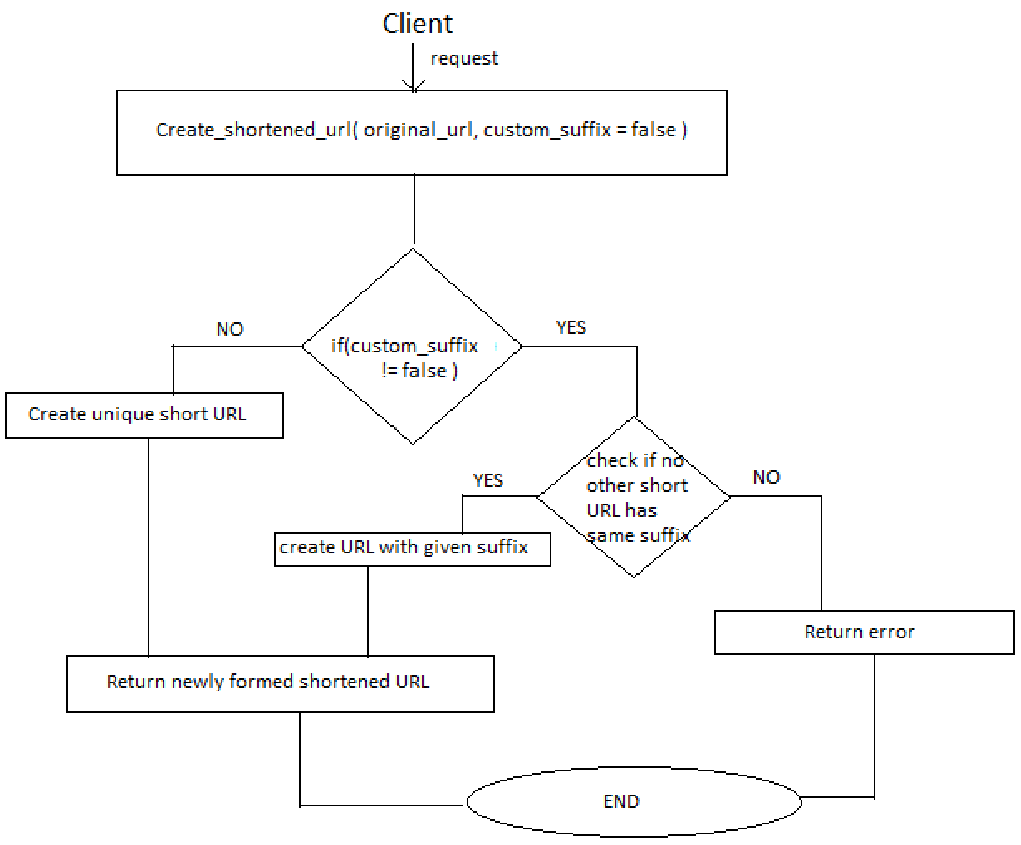
Design a TinyURL system

1. How does your system ensure that 2 URLs never map to the same shortened URL?

For creating a shortened url we will create a function Create_shortened_Url which have two params (original_url and custom_string (by default its value would be false).

Create_shortened_Url (original_url, custom_suffix = false)


Client
request
Create_shortened_url(original_url, custom_suffix = false)
NO
if(custom_suffix != false)
YES
Create unique short URL
check if no other short URL has same suffix
YES
NO
create URL with given suffix
Return error
Return newly formed shortened URL
END

- **Database design :**

  Database table will have two attributes:
  - ID [Primary key]
  - original_url

Each original_url will be mapped by a unique number which is ID. Using this ID we will generate unique sort url. For generating this unique id we will be using consistent hashing**.**

By considering that shortened url will only contain:

1. A-Z (Capital letters)
2. A-z (small letters)
3. 0-9 (digits)

Total of 62 characters

So, to convert unique id to a unique string **Base62** approach will be used.

Base62:

Convert number in base10 to base62

```
string Base62(long int n)
{
    string s =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    string shorturl;

    while (n)
    {
        shorturl.push_back(s[n%62]);
        n = n/62;
    }
    reverse(shorturl.begin(), shorturl.end());

    return shorturl;
}
```

To convert original url to sort url following methods can be used:

- **Generate a random string and check if already present in DB:**
  Problem is it takes more time if going in loop and continuously check if string already present in DB or not. It also fails with concurrent users.
- **Hashing (md5 approach):** Problem with this approach is chances of Collision, also it take a lot of memory and we cannot store that in main memory.
- **Range Based Partitioning:** In this scenario, the server will not be uniformly distributed.
- The best solution is **Consistent Hashing with multiple hash function.**

  Consistent Hashing: It is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed hash table by assigning them a position on an abstract circle or hash space. This allows servers and objects to scale without affecting the overall system. Using consistent hashing, we only need to calculate the nearest server in the hash space and thus we need not to save any hash map.

2. How will you ensure the system is very low latency?

We can use **cache** for URLs that are **frequently accessed**. System firstly check if the URL entered is available in cache before looking into DB servers. If found then return else check in DB.

If URL is very frequently accessed, in that case we will put the corresponding mapping in the cache and if cache is full replace it with the least frequently accessed url. In this way system will have low latency.

3. What will happen if the machine storing the URL mapping dies?

As we are employing more than one DB server (using Distributed Database). So we can ensure uniform distribution of load in case any of the DB server goes down. As system maintain sync between all the DB servers using crone Jobs so the data of that machine can be backed up.

4. How do you make sure your system is consistent?

As we are using consistent hashing with large hash space the chances of collision are negligible. And in database as we store unique Id mapped with original url. So if we have shortened url then it will be easily converted into that unique id and mapped with original url. So when we enter that shortened url it will definitely return the original url.

5. How do you make sure that your service never goes down?

- The service may go down due to any one of these reasons –
  - **Distributed Denial of Service –** In this attack a number of requests are hit to the server and ultimately eating up all your bandwidth.
    - The solution is using distributed server architecture as in DDoS only one region is generally taken down.
  - **Lack Of Bandwidth –** In case there are a lot of requests, there may be a bottleneck situation. So keep the bandwidth high as possible as per usage.
  - **Bad Upgradation –** If we do not use proper mechanism to upgrade our application, the server may remain down for a while.
    - Solution is to use **containerization,** and tools like Docker that ensures that we are up and running with our servers within few seconds/minutes.
- **By assuming that any of the machines die (not all):**

  As we are using Consistent hashing so if any of the machine will die then the machine next to it in hash space will take all the load of that machine.