hash functions

In [1]:
```
import os
import re
import string
from collections import Counter
import matplotlib.pyplot as plt
import pandas as pd
import time
import timeit
```

hash functions

Topic today: how different hash functions affect the computation time ?

last week we've talked about how hashtable can still work when collisions happened, But it still is something we want to avoid. and today we'll talk about how to avoid collisions, using different hash functions.

- First, we will set up the data we need
- secondly, we will hash some real life textual data, into our hashtable, and see the collisions distribution. Then, we will do the same thing, but with different hash functions, and see how it affects the collisions distribution.
- lastly, we will simulate the hash function computation time, and see how different hash functions affect the computation time.

STEP 1: data preparation

In [2]:
```
def filelist(root: str) -> list[str]:
    """Return a fully-qualified list of filenames under root directory
    traversing subdirectories recursively and return all the paths of files
    """
    return [os.path.join(root, f) for root, _, files in os.walk(root) for f


def get_text(fileName: str) -> str:
    f = open(fileName, encoding='latin-1')
    s = f.read()
    f.close()
    return s

def words(text: str) -> list[str]:
    """
    Given a string, return a list of words normalized as follows.
    Split the string to make words first by using regex compile() function
    and string.punctuation + '0-9\\r\\t\\n]' to replace all those
    char with a space character.
    Split on space to get word list.
    Ignore words < 3 char long.
    Lowercase all words
    """
```

```python
    regex = re.compile('[' + re.escape(string.punctuation) + '0-9\\r\\t\\n]'
    # delete stuff but leave at least a space to avoid clumping together
    nopunct = regex.sub(" ", text)
    words = nopunct.split(" ")
    words = [w for w in words if len(w) > 2]  # ignore a, an, to, at, be, ..
    words = [w.lower() for w in words]
    # print words
    return words

def text_data_set_up(path: str) -> list[str]:
    files = filelist(path)
    texts = [get_text(f) for f in files]
    word_list = [words(t) for t in texts]
    word_list = [w for wl in word_list for w in wl]

    return list(set(word_list))
```

In [3]:

```python
texts = text_data_set_up('./slate')
len(texts)
texts[:10]
```

Out[3]:

```
['siphons',
 'smacks',
 'gettler',
 'gabbing',
 'scooted',
 'comb',
 'upat',
 'corp',
 'prowled',
 'custom']
```

STEP 2 : comparing the hash functions in terms of collisions distribution

In [4]:

```python
def bad_hash(word: str, size: int = 26*26) -> int:
    a = ord(word[0]) - 97
    b = ord(word[1]) - 97

    return (a * 26 + b) % size


def good_hash(key, size: int = 26*26) -> int:
    key = str(key)
    h = 0
    for i in key:
        h += 31* h + ord(i)

    return  h % size
```

In [5]:

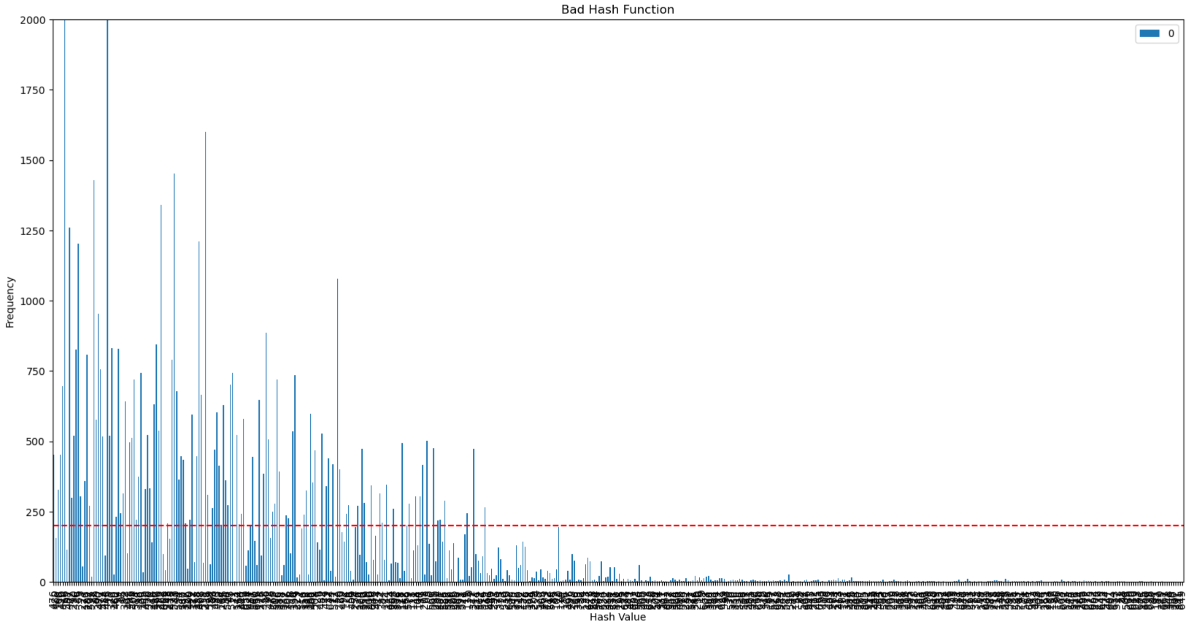
```python
good_hashes = [good_hash(w) for w in texts]
good_hash_counter = Counter(good_hashes)


hashes = [bad_hash(w) for w in texts]
bad_hash_counter = Counter(hashes)
```

In [6]:

```python
df = pd.DataFrame.from_dict(bad_hash_counter, orient='index')
fig, ax = plt.subplots()
df.plot(kind='bar', ax = ax, figsize=(20, 10), xlim=(0, 26*26), ylim=(0, 200
ax.set_xlabel('Hash Value')
```

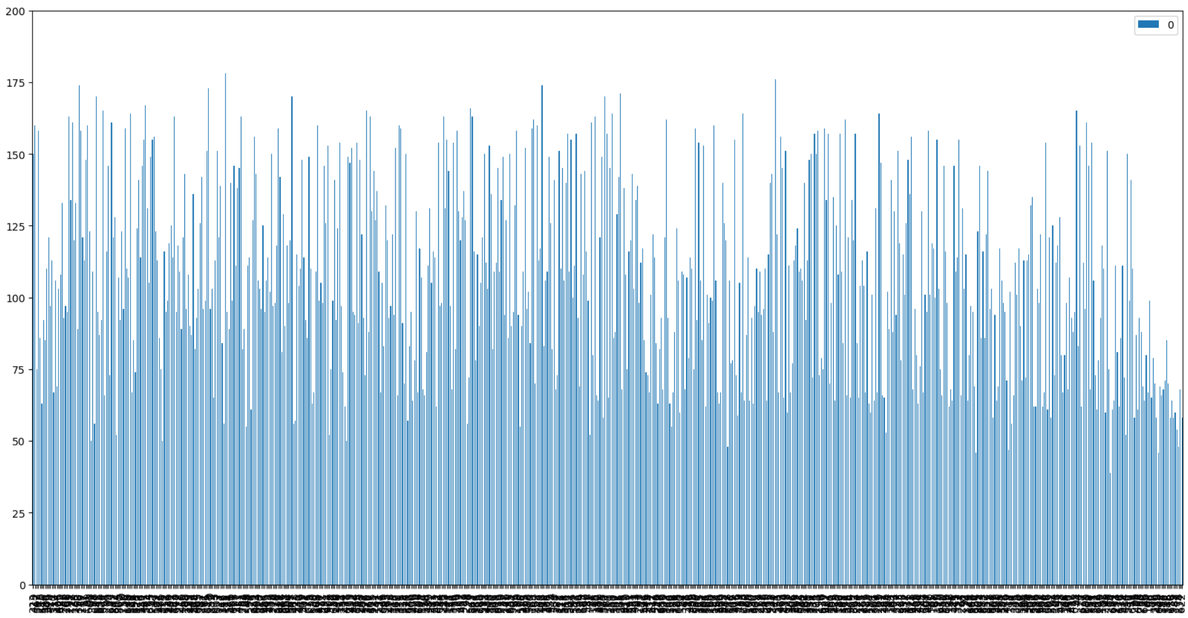
```
ax.set_ylabel('Frequency')
ax.set_title('Bad Hash Function')
ax.hlines(200, 0, 26*26, colors='r', linestyles='dashed')
```

Out[6]: <matplotlib.collections.LineCollection at 0x1437a1190>



In [7]:
```
df_g = pd.DataFrame.from_dict(good_hash_counter, orient='index')
df_g.sort_values(by=0, ascending=False).tail(10)

df_g.plot(kind='bar', figsize=(20, 10), xlim=(0, 26*26), ylim=(0, 200))
```

Out[7]: <AxesSubplot:>



STEP 3: comparing the hash functions in terms of computation time

3-1 Hash table class

In [8]:
```
from typing import Callable
```

```python
class Hashtable:
    def __init__(self, hash_func: Callable[[str,int],int], size: int) -> Non
        self.size = size
        self.table = [[] for _ in range(size)]
        self._hash = hash_func

    def __setitem__(self, key: str, value:str) -> None:
        index = self._hash(key, self.size)
        self.table[index].append((key, value))

    def __getitem__(self, key: str) -> str:
        index = self._hash(key, self.size)

        for k, v in self.table[index][::-1]:
            if k == key:
                return v

    def __delitem__(self, key: str) -> None:
        index = self._hash(key, self.size)

        for k, v in self.table[index]:
            if k == key:
                self.table[index].remove((k, v))
```

In [9]:

```python
bad_dict = Hashtable(bad_hash, 26*26)
good_dict = Hashtable(good_hash, 26*26)
```

3-2 timer function & testing tasks set up

In [10]:

```python
def timer(f):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = f(*args, **kwargs)
        end = time.time()
        return result, end - start
    return wrapper



@timer
def test_speed(my_dict: Hashtable, word_list: list[str]) -> None:
    for i, w in enumerate(word_list):
        my_dict[w] = i

    for w in word_list:
        x = my_dict[w]

    for w in word_list:
        del my_dict[w]
```

3-3 run test

In [11]:

```python
_ , good_time =test_speed(good_dict, texts)
_ , bad_time = test_speed(bad_dict, texts)
```

In [12]:

```python
print(
f'''{bad_time=} seconds

{good_time=} seconds''')
```

```
bad_time=2.6138880252838135 seconds

good_time=0.6940028667449951 seconds
```

In [ ]:

In [ ]: