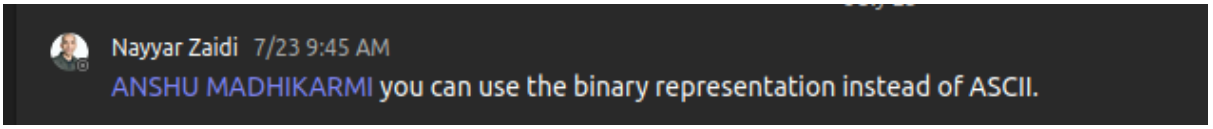
Report

1- Given a bucket of size 3, hash following values with extendible hashing: 16, 22, 26,20, 3, 1, 12, 11, 13, 19, 38, 47, 46. Use ASCII to represent as the hash code.

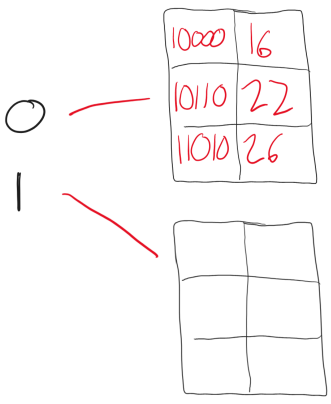
To clarify some things for this task. I had decided to do this part without the code since I wasn't sure if it was required. So I have included a drawn version below. I had also seen that in the teams we were allowed to use a binary representation and included the screenshot below.



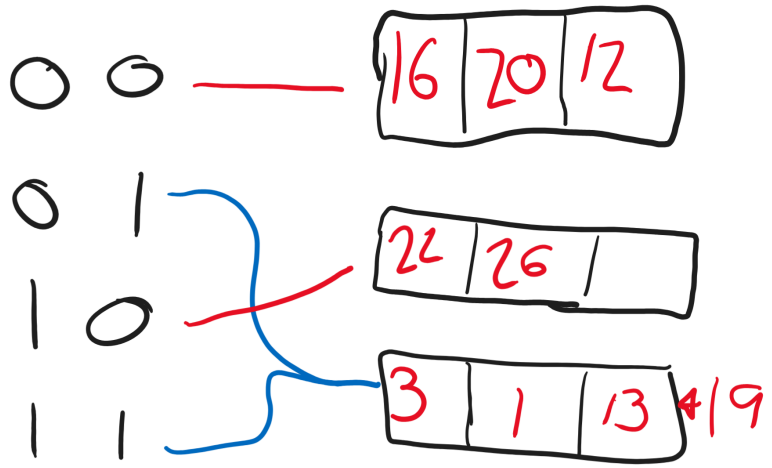
So I did:

ID	Num	Binary
1	16	10000
2	22	10110
3	26	11010
4	20	10100
5	3	11
6	1	1
7	12	1100
8	13	1101
9	19	10011
10	38	100110
11	47	101111
12	46	101110

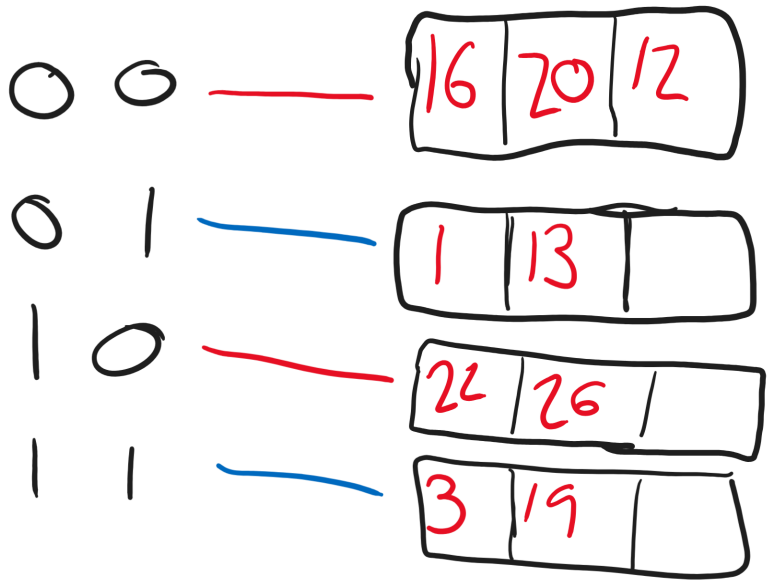
I then added each number by its ID to insert them in order the results were:
Added the first 3 until the first bucket overflow:



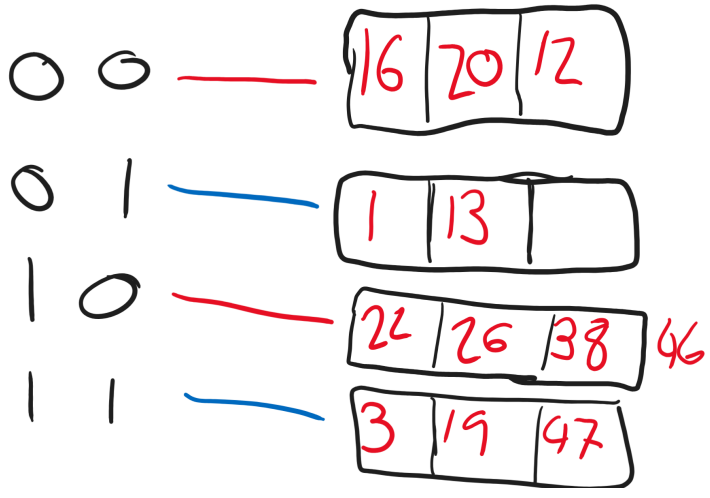
So I had broken up the bucket and continued the process until the next overflow:



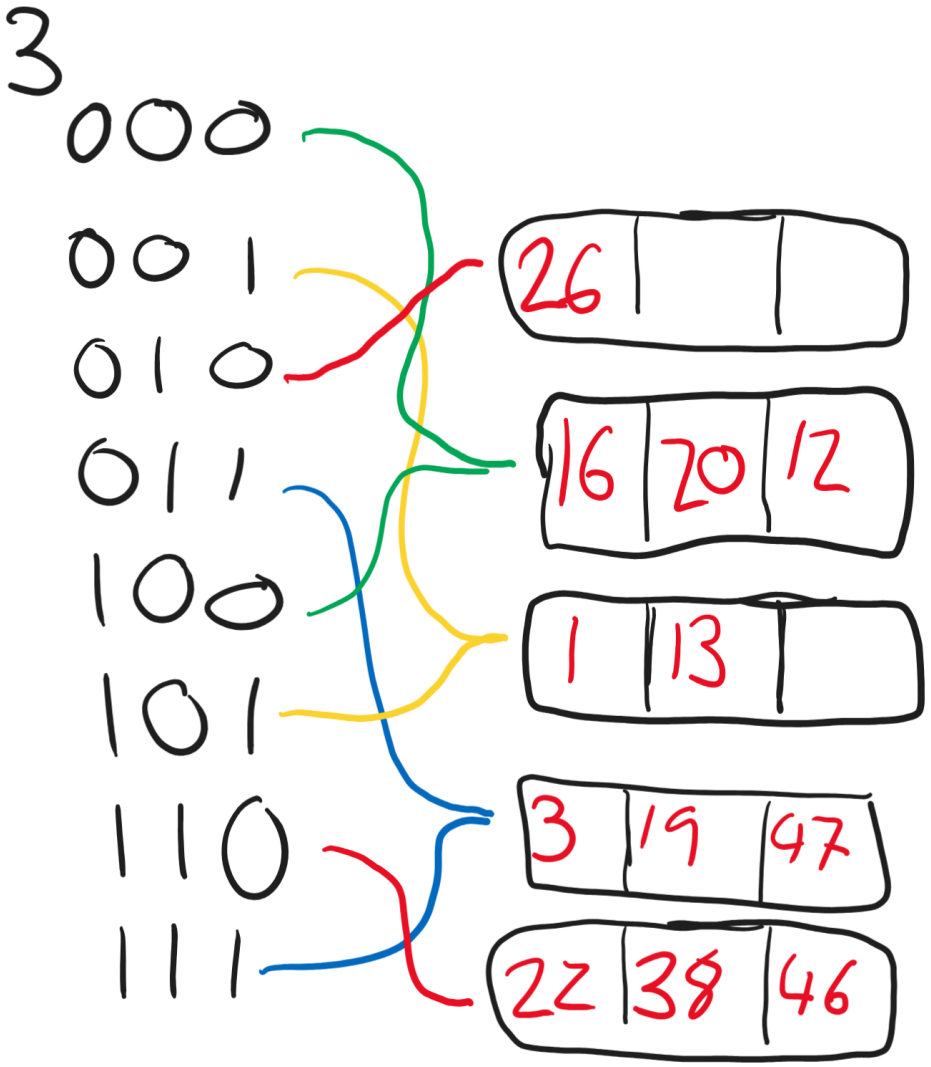
This was after the next overflow, just added a new bucket.



Overflow.



Final product after the overflow and added the last value of 46.



Since 46 was the last number to add, the image above is the completed product.

2 - We have provided you with the code for ‘extendible hashing’. Run the code with various use-cases so you can demonstrate the ownership of the code. Provide code reflections in following forms:

a. Evidence of running the code with various use cases:

So I had started to run the code once I had a foundational understanding of how it works. Although I soon found out that the provided code was broken. I had added in multiple values as a means to test when a bucket became full.
So I added these items:

```
t_id = 0
t_amount = 100
u_name = 'David'
```

```
t_id = 1
t_amount = 101
u_name = 'David2'
```

```
t_id = 2
t_amount = 102
u_name = 'David2'
```

This had added 2 different listings to the first bucket (ID 1). Now this bucket has a local depth of 1, since python uses 0 indexing this shows the bucket is at maximum capacity.

```
directory.directory_records[0].value.index
```

✓ 0.0s

```
[[0, 100, 'David'], [2, 102, 'David2']]
```

So I added :

```
t_id = 4
t_amount = 102
u_name = 'David2'
```

Which triggered this response:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[30], line 4
      2 t_amount = 102
      3 u_name = 'David2'
----> 4 insert([t_id, t_amount, u_name])

Cell In[22], line 69
     65 directory= new_directory
     67 for i in range(len(tempopary_memory)):
---> 69     insert(tempopary_memory[i],lock)

NameError: name 'lock' is not defined
```

So I realised I needed to look back in the code to find lock:

```
            for i in range(len(tempopary_memory)):


                insert(tempopary_memory[i],lock)
```

Lock here is not defined at all… removing it fixes it, maybe?

```
directory.directory_records
```

[62] ✓ 0.0s

```
[<__main__.DirectoryRecord at 0x722e9c1e4ca0>,
 <__main__.DirectoryRecord at 0x722e9c1e4a30>,
 <__main__.DirectoryRecord at 0x722e9c1e4e50>,
 <__main__.DirectoryRecord at 0x722e9c1e62f0>]
```

Well we have increased to global depth 2 since we can see 4 records, that's good!
At this point I had a strong understanding of how it operates and so I ran some more inserts with the intention to cause the buckets to split:

```
t_id = 0
t_amount = 100
u_name = 'David'
```

```
t_id = 1
t_amount = 101
u_name = 'David2'
```

```
t_id = 2
t_amount = 102
u_name = 'David2'
```

```
t_id = 4
```

```
t_amount = 102
u_name = 'David2'
insert([t_id, t_amount, u_name])
```

```
t_id = 8
t_amount = 102
u_name = 'David2'
insert([t_id, t_amount, u_name])
```

Thus outputting:

```
Hash Prefix:  0000000000000000
Bucket ID:  1
Local Depth:  2
Stored Values:  [[0, 100, 'David'], [8, 102, 'David2']]
Hash Prefix:  0000000000000001
Bucket ID:  2
Local Depth:  1
Stored Values:  [[1, 101, 'David2']]
Hash Prefix:  0000000000000010
Bucket ID:  5
Local Depth:  1
Stored Values:  [[2, 102, 'David2']]
Hash Prefix:  0000000000000011
Bucket ID:  6
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000000100
Bucket ID:  11
Local Depth:  1
Stored Values:  [[4, 102, 'David2']]
Hash Prefix:  0000000000000101
Bucket ID:  12
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000000110
Bucket ID:  13
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000000111
Bucket ID:  14
Local Depth:  1
Stored Values:  []
```

I then wanted to go further and try creating a .CSV of random numbers then import that CSV and insert it into the algorithm:

```
#13 random numbers like the task before (part 1)
sampleNumbers = random.sample(range(1, 30), 13)
print(sampleNumbers)
```

✓ 0.0s

```
[9, 8, 19, 1, 10, 2, 24, 28, 17, 16, 11, 23, 26]
```

randomNumber.csv

```
# Numbers
9
8
19
1
10
2
24
28
17
16
11
23
26
```

And finally returning:

```
Global Depth:  (4,)
Count of directory records:  16
Hash Prefix:  0000000000000000
Bucket ID:  1
Local Depth:  3
Stored Values:  [['16']]
Hash Prefix:  0000000000000001
Bucket ID:  2
Local Depth:  3
Stored Values:  [['1'], ['17']]
Hash Prefix:  0000000000000010
Bucket ID:  17
Local Depth:  1
Stored Values:  [['10'], ['2']]
Hash Prefix:  0000000000000011
Bucket ID:  18
Local Depth:  1
Stored Values:  [['19']]
Hash Prefix:  0000000000000100
Bucket ID:  23
Local Depth:  1
Stored Values:  [['28']]
Hash Prefix:  0000000000000101
Bucket ID:  24
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000000110
Bucket ID:  25
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000000111
Bucket ID:  26
Local Depth:  1
Stored Values:  [['23']]
Hash Prefix:  0000000000001000
Bucket ID:  35
Local Depth:  1
Stored Values:  [['8'], ['24']]
Hash Prefix:  0000000000001001
Bucket ID:  36
Local Depth:  1
Stored Values:  [['9']]
Hash Prefix:  0000000000001010
Bucket ID:  37
Local Depth:  1
Stored Values:  [['26']]
Hash Prefix:  0000000000001011
Bucket ID:  38
Local Depth:  1
Stored Values:  [['11']]
Hash Prefix:  0000000000001100
Bucket ID:  39
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000001101
Bucket ID:  40
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000001110
Bucket ID:  41
Local Depth:  1
Stored Values:  []
Hash Prefix:  0000000000001111
Bucket ID:  42
Local Depth:  1
Stored Values:  []
```

b. Modifications

The first one is the removal of the ‘lock’ from line 69 of the insertion algorithm. I feel like most people had done this. So I wanted to do something that would help visualise the returned results so I made a simple printDirectory() function:

```
def printDirectory(index):
    print("Global Depth: ", index.global_depth)
    print("Count of directory records: ", len(index.directory_records))
    for i in range(len(index.directory_records)):
        print("Hash Prefix: ",hash_funtion(index.directory_records[i].hash_prefix))
        print("Bucket ID: ",index.directory_records[i].value.id)
        print("Local Depth: ",index.directory_records[i].value.local_depth)
        print("Stored Values: ",index.directory_records[i].value.index)
```
✓ 0.0s

Which allowed me to get the results seen above. I would have liked to have done more but since I was catching up after being sick I had to move on.

c. Detail Comments of various lines of the code

What I will do is attach my .IPYNB to the end of this document so the comments can be seen there.

3.

This task has been reworked as a part of the discussion phase of this assignment. The sorting algorithm generates a series of values that is programmable and is stored as a list to simulate space on a disk. It is in part 3 of the IPYNB submission.