In [27]:

```
import ctypes

class Array:
    def __init__(self, size):
        assert size > 0, "Array size must be > 0"
        self._size = size
        PyArrayType = ctypes.py_object * size
        self._elements = PyArrayType()
        self.clear(None)

    def __len__(self):
        return self._size

    def __getitem__(self, index):
        assert index >=0 and index < len(self), "Array subscript out of range!"
        return self._elements[index]

    def __setitem__(self, index, value):
        assert index >=0 and index < len(self), "Array subscript out of range!"
        self._elements[index] = value

    def clear(self, value):
        for i in range(len(self)):
            self._elements[i] = value

    def __iter__(self):
        return _ArrayIterator(self._elements)

class _ArrayIterator:
    def __init__(self, theArray):
        self._arrayRef = theArray
        self._curNdx = 0
    def __iter__(self):
        return self

    def __next__(self):
        if self._curNdx < len(self._arrayRef):
            entry = self._arrayRef[self._curNdx]
            self._curNdx += 1
            return entry
        else:
            raise StopIteration


# In[ ]:


class OrderedLinkedList:
    def __init__(self):
        self.head = None
        self.length = 0

    def append(self, element):
        currentNode = self.head
        previousNode = None
        while currentNode is not None:
            if currentNode.data > element:
                break
            previousNode = currentNode
            currentNode = currentNode.next

        newNode = _ListNode(element)
        if previousNode is None:
            newNode.next = self.head
            self.head = newNode
            self.length += 1
        else:
            newNode.next = currentNode
            previousNode.next = newNode
            self.length += 1


    def remove(self, element):
        currentNode = self.head
        previousNode = None
        if self.length > 0:
            if currentNode.data == element:
                self.head = currentNode.next
                currentNode = currentNode.next
                self.length -= 1
            while currentNode != None:
                if currentNode.data != element:
                    previousNode = currentNode
                    currentNode = currentNode.next
                elif currentNode.data == element:
                    previousNode.next = currentNode.next
                    currentNode = currentNode.next
                    self.length -= 1
        else:
            raise IndexError('Linked list is empty')
```

```
    def printList(self):
        for element in self:
            print(element)

    def __len__(self):
        return self.length

    def __iter__(self):
        return _OrderedLinkedListIterator(self.head)

    def __contains__(self, element):
        for nodes in self:
            if nodes == element:
                return True
        return False

    def extend(self, LLB):
        for elements in LLB:
            self.append(elements)

    def getHead(self):
        if self.head is not None:
            return self.head.data

# Task 1
class HashTable:
    def __init__(self, mode , constant, size = 5):
        self._hashArray = Array(size)
        self._loadFactor = 0.8
        self._occupiedSlots = 0
        self._size = size
        self._flag = "flag"
        self._mode = mode
        self._hashKeyConstant = constant

    def hashFunction(self, key, size):
        if key == self._flag:
            return 0
        return key % size

    def rehash(self, hashKey):
        if self._hashArray[hashKey] == self._flag:
            return hashKey
        return (hashKey + 1) % self._size

    def quadHash(self, hashKey, const):
        if self._hashArray[hashKey] == self._flag:
            return hashKey
        return (hashKey+(const**2)) % self._size

    def doubleHash(self, hashKey, const):
        if self._hashArray[hashKey] == self._flag:
            return hashKey
        hashKey1 = hashKey % self._size
        hashKey2 = 1+(hashKey % self._hashKeyConstant)
        return ((hashKey1 + (const*hashKey2)) % self._size)


    def add(self, element):
        if self._mode == "linear":
            self.addLinear(element)
        elif self._mode == "quad":
            self.addQuad(element)
        elif self._mode == "double":
            self.addDouble(element)
        elif self._mode == "chaining":
            self.addSepChain(element)
        else:
            raise ValueError("Invalid method")

    def remove(self, element):
        if self._mode == "linear":
            self.removeLinear(element)
        elif self._mode == "quad":
            self.removeQuad(element)
        elif self._mode == "double":
            self.removeDouble(element)
        elif self._mode == "chaining":
            self.removeSepChain(element)
        else:
            raise ValueError("Invalid method")

    def addLinear(self, element):
        hashKey = self.hashFunction(element, self._size)
        if self.isFree(hashKey):
            self._hashArray[hashKey] = element
            self._occupiedSlots += 1
            if self.load() >= self._loadFactor:
                self.increaseSize()
        elif self._occupiedSlots < self._size:
            hashKey2 = self.rehash(hashKey)
```

```
            if self.isFree(hashKey2):
                self._hashArray[hashKey2] = element
                self._occupiedSlots += 1
                if self.load() >= self._loadFactor:
                    self.increaseSize()
            else:
                while not self.isFree(hashKey2) and hashKey != hashKey2:
                    hashKey2 = self.rehash(hashKey2)

                if hashKey != hashKey2:
                    self._hashArray[hashKey2] = element
                    self._occupiedSlots += 1
                    if self.load() >= self._loadFactor:
                        self.increaseSize()

    def addQuad(self, element):
        constant = 1
        hashKey = self.hashFunction(element, self._size)
        if self.isFree(hashKey):
            self._hashArray[hashKey] = element
            self._occupiedSlots += 1
            if self.load() >= self._loadFactor:
                self.increaseSize()
        elif self._occupiedSlots < self._size:
            hashKey2 = self.quadHash(hashKey, constant)
            if self.isFree(hashKey2):
                self._hashArray[hashKey2] = element
                self._occupiedSlots += 1
                if self.load() >= self._loadFactor:
                    self.increaseSize()
            else:
                while not self.isFree(hashKey2) and hashKey != hashKey2:
                    constant += 1
                    hashKey2 = self.quadHash(hashKey2, constant)

                if hashKey != hashKey2:
                    self._hashArray[hashKey2] = element
                    self._occupiedSlots += 1
                    if self.load() >= self._loadFactor:
                        self.increaseSize()

    def addDouble(self, element):
        constant = 1
        hashKey = self.hashFunction(element, self._size)
        if self.isFree(hashKey):
            self._hashArray[hashKey] = element
            self._occupiedSlots += 1
            if self.load() >= self._loadFactor:
                self.increaseSize()
        elif self._occupiedSlots < self._size:
            hashKey2 = self.doubleHash(hashKey, constant)
            if self.isFree(hashKey2):
                self._hashArray[hashKey2] = element
                self._occupiedSlots += 1
                if self.load() >= self._loadFactor:
                    self.increaseSize()
            else:
                while not self.isFree(hashKey2) and hashKey != hashKey2:
                    constant += 1
                    hashKey2 = self.doubleHash(hashKey2, constant)

                if hashKey != hashKey2:
                    self._hashArray[hashKey2] = element
                    self._occupiedSlots += 1
                    if self.load() >= self._loadFactor:
                        self.increaseSize()

    def addSepChain(self, element):
        hashKey = self.hashFunction(element, self._size)
        if self._hashArray[hashKey] == None:
            self._hashArray[hashKey] = OrderedLinkedList()
            self._hashArray[hashKey].append(element)
            self._occupiedSlots += 1
        else:
            self._hashArray[hashKey].append(element)
            self._occupiedSlots += 1

    def removeLinear(self, element):
        hashKey = self.hashFunction(element, self._size)
        if self._hashArray[hashKey] == element:
            self._hashArray[hashKey] = self._flag
            self._occupiedSlots -= 1

        elif self._occupiedSlots > 0:
            hashKey2 = self.rehash(hashKey)
            if self._hashArray[hashKey2] == element:
                self._hashArray[hashKey2] = self._flag
                self._occupiedSlots -= 1

            else:
                while hashKey != hashKey2:
                    hashKey2 = self.rehash(hashKey2)
```

```
                if self._hashArray[hashKey2] == element:
                    self._hashArray[hashKey2] = self._flag
                    self._occupiedSlots -= 1
            print(f"{element} not found in the table")
            return False

    def removeQuad(self, element):
        constant = 1
        hashKey = self.hashFunction(element, self._size)
        if self._hashArray[hashKey] == element:
            self._hashArray[hashKey] = self._flag
            self._occupiedSlots -= 1

        elif self._occupiedSlots > 0:
            hashKey2 = self.quadHash(hashKey, constant)
            if self._hashArray[hashKey2] == element:
                self._hashArray[hashKey2] = self._flag
                self._occupiedSlots -= 1

            else:
                while hashKey != hashKey2:
                    constant += 1
                    hashKey2 = self.quadHash(hashKey2, constant)
                    if self._hashArray[hashKey2] == element:
                        self._hashArray[hashKey2] = self._flag
                        self._occupiedSlots -= 1
                print(f"{element} not found in the table")
                return False

    def removeDouble(self, element):
        constant = 1
        hashKey = self.hashFunction(element, self._size)
        if self._hashArray[hashKey] == element:
            self._hashArray[hashKey] = self._flag
            self._occupiedSlots -= 1

        elif self._occupiedSlots > 0:
            hashKey2 = self.doubleHash(hashKey, constant)
            if self._hashArray[hashKey2] == element:
                self._hashArray[hashKey2] = self._flag
                self._occupiedSlots -= 1

            else:
                while hashKey != hashKey2:
                    constant += 1
                    hashKey2 = self.doubleHash(hashKey2, constant)
                    if self._hashArray[hashKey2] == element:
                        self._hashArray[hashKey2] = self._flag
                        self._occupiedSlots -= 1
                print(f"{element} not found in the table")
                return False

    def removeSepChain(self, element):
        hashKey = self.hashFunction(element, self._size)
        if self._hashArray[hashKey] == None:
            print(f"{element} not found in the table")
        else:
            self._hashArray[hashKey].remove(element)
            self._occupiedSlots -= 1

    def __iter__(self):
        return self._hashArray.__iter__()

    def isFree(self, key):
        return self._hashArray[key] is None or self._hashArray[key] == self._flag

    def load(self):
        return self._occupiedSlots / self._size

    def increaseSize(self):
        if self._mode != "chaining":
            newSize = (self._size * 2) + 1
            tempArray = self._hashArray
            self._hashArray = Array(newSize)
            self._size = newSize
            self._occupiedSlots = 0
            for elements in tempArray:
                if elements:
                    self.add(elements)

    def printTable(self):
        table=[]
        if self._mode != "chaining":
            for item in self._hashArray:
                table.append(item)
        else:
            for item in self._hashArray:
                if item != None:
                    head = item.getHead()
                    table.append(head)
                    for i in item:
                        if i != head:
```

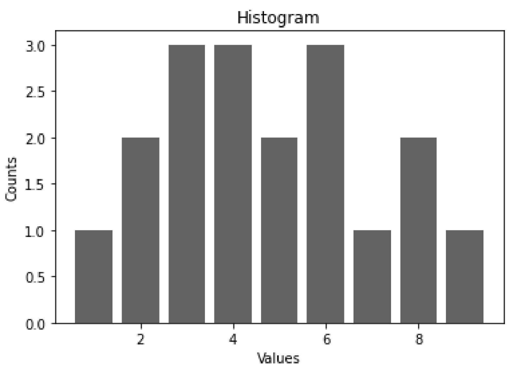
```
                            table.append(i)
                else:
                    table.append("None")
        return table

    def getOccupied(self):
        return self._occupiedSlots

    def getSize(self):
        return self._size

    def __len__(self):
        return self._occupiedSlots

def main(method):
    print("-"*30)
    print("Method: ", method)
    print("-"*30)
    h= HashTable(method, 10)
    data=[431,96,579,903,765,876,543,543,678,903,765]
    for i in data:
        h.add(i)
    print("=> Elements in the Hash Table:", len(h))
    print("=> Size of the Hash Table:", h.getSize())
    print("=> Current Load on the Hash Table:", round(h.load(), 2)*100,"%")
    h.remove(142)
    print("\nRemoved 142")
    print("Hash table after removal:")
    print(h.printTable())
    h.add(388)
    h.add(60)
    h.add(166)
    print("\nHash table after expansion:")
    print(h.printTable())
    print("=> Elements in the Hash Table:", len(h))
    print("=> Size of the Hash Table:", h.getSize())
    print("=> Current Load on the Hash Table:", round(h.load(), 2)*100,"%")
main("quad")
main("chaining")
```

```
------------------------------
Method:  quad
------------------------------
=> Elements in the Hash Table: 10
=> Size of the Hash Table: 23
=> Current Load on the Hash Table: 43.0 %
142 not found in the table

Removed 142
Hash table after removal:
[None, None, 876, None, 579, 96, 903, 765, None, None, None, 903, None, None, 543, 543, None, 431, None, None, 765, None, N
one]

Hash table after expansion:
[None, None, 876, None, 579, 96, 903, 765, None, None, 166, 903, None, None, 543, 543, None, 431, None, 60, 765, 388, None]
=> Elements in the Hash Table: 13
=> Size of the Hash Table: 23
=> Current Load on the Hash Table: 56.99999999999999 %
------------------------------
Method:  chaining
------------------------------
=> Elements in the Hash Table: 11
=> Size of the Hash Table: 5
=> Current Load on the Hash Table: 220.00000000000003 %
142 not found in the table

Removed 142
Hash table after removal:
[765, 96, 431, 876, 'None', 543, 678, 903, 903, 579]

Hash table after expansion:
[60, 765, 765, 96, 166, 431, 876, 'None', 388, 543, 543, 678, 903, 903, 579]
=> Elements in the Hash Table: 14
=> Size of the Hash Table: 5
=> Current Load on the Hash Table: 280.0 %
```

In [13]:

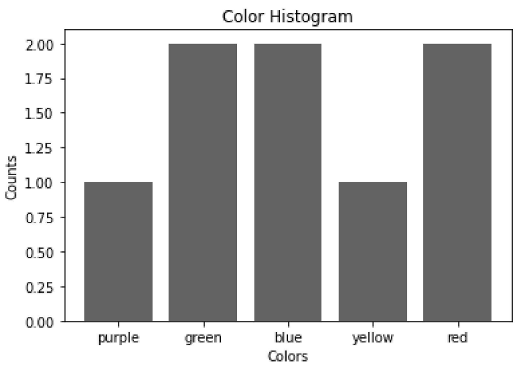
```
import matplotlib.pyplot as plt
class Histogram:
    def __init__(self):
        self.data = {}

    def incCount(self, item):
        if item in self.data:
            self.data[item] += 1
        else:
            self.data[item] = 1

    def remove(self, item):
        if item in self.data:
            self.data[item] -= 1
            if self.data[item] == 0:
                del self.data[item]

    def getCount(self, item):
        return self.data.get(item, 0)

    def items(self):
        return list(self.data.items())
    def totalCount(self):
        total=0
        for keys in self.data:
            total+=self.data[keys]
        return total
hist = Histogram()
data=[1,2,4,3,2,6,5,6,8,7,3,8,9,4,3,4,5,6]
for i in data:
    hist.incCount(i)

print(hist.getCount(3))
print("Total frequencies: ",hist.totalCount())
values = [item[0] for item in hist.items()]
counts = [item[1] for item in hist.items()]
plt.bar(values, counts)
plt.xlabel('Values')
plt.ylabel('Counts')
plt.title('Histogram')
plt.show()
# print(hist.get_data())
```

```
3
Total frequencies:  18
```



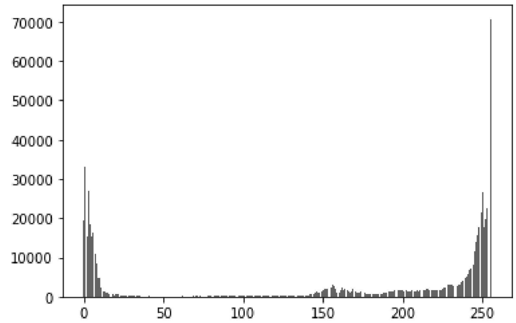
In [14]:

```
import matplotlib.pyplot as plt
class ColorHistogram:
    def __init__(self, bins):
        self.bins = bins
        self.data = [[] for _ in range(bins)]
    def add(self, value):
        """
        Add a value to the histogram
        """
        bin = hash(value) % self.bins
        for i, item in enumerate(self.data[bin]):
            if item[0] == value:
                self.data[bin][i] = (value, item[1] + 1)
                return
        self.data[bin].append((value, 1))
    def remove(self, value):
        """
        Remove a value from the histogram
        """
        bin = hash(value) % self.bins
        for i, item in enumerate(self.data[bin]):
            if item[0] == value:
                if item[1] > 1:
                    self.data[bin][i] = (value, item[1] - 1)
                else:
                    self.data[bin].pop(i)
                return
        raise ValueError(f"{value} not in histogram")
    def count(self, value):
        """
        Return the count of a value in the histogram
        """
        bin = hash(value) % self.bins
        for item in self.data[bin]:
            if item[0] == value:
                return item[1]
        return 0
    def items(self):
        """
        Return a list of tuples representing the values and their counts in the histogram
        """
        items = []
        for bin in self.data:
            items.extend(bin)
        return items

hist = ColorHistogram(10)
hist.add("red")
hist.add("blue")
hist.add("green")
hist.add("red")
hist.add("blue")
hist.add("green")
hist.add("yellow")
hist.add("purple")
colors = [item[0] for item in hist.items()]
counts = [item[1] for item in hist.items()]
plt.bar(colors, counts)
plt.xlabel('Colors')
plt.ylabel('Counts')
plt.title('Color Histogram')
plt.show()
```



In [20]:

```
class ColorHistogram:
    def __init__(self):
        self.histogram = [[] for _ in range(256)] # Initialize 2-D array of chains

    def addColor(self, color):
        """
        Add a color to the histogram.
        color: a tuple of 3 integers representing the RGB values (0-255) of the color
        """
        red, green, blue = color
        self.histogram[red].append(color)

    def getFrequency(self, color):
        """
        Get the frequency count of a color in the histogram.
        color: a tuple of 3 integers representing the RGB values (0-255) of the color
        """
        red, green, blue = color
        return len(self.histogram[red])

    def plotHistogram(self):
        """
        Plot the histogram using matplotlib.
        """
        import matplotlib.pyplot as plt

        # Get the frequencies of each color
        frequencies = [len(chain) for chain in self.histogram]

        # Plot the histogram
        plt.bar(range(256), frequencies)
        plt.show()
from PIL import Image

# Open an image
img = Image.open("sunset.jpg")

# Create a ColorHistogram object
histogram = ColorHistogram()

# Add the colors of the image to the histogram
for pixel in img.getdata():
    histogram.addColor(pixel)

# Plot the histogram
histogram.plotHistogram()
```



In [ ]: