Advanced Database Systems Lab 3

Linear Hashing and Extendable Hashing

Darshankumar Kapadiya (21MCEC02)
Yagnesh M. Bhadiyadra (21MCEC11)

Code:

```
from flask import Flask
from flask_restful import Api, Resource, reqparse, request

app = Flask(__name__)
api = Api(app)

class Bucket_LH:
    def __init__(self):
        self.bucket = []
        self.next = None

    def print_bucket(self):
        # print(self.bucket)
        return self.bucket

    def insert(self, val: int):
        self.bucket.append(val)

    def size(self):
        return len(self.bucket)

    def has_next(self):
        if self.next is not None:
            return 1
        else:
            return 0


class LH:
    def __init__(self):
        self.next = 0
        self.bucket_size = 2
        self.load_factor = 0.70
        self.elements = 0
        self.hash_table_size = 4
        self.linear_hash_list = [Bucket_LH() for i in range(self.hash_table_size)]

    def insert(self, val: int):
        if self.next == self.hash_table_size:
            self.hash_table_size = self.hash_table_size * 2
            self.next = 0
```

```
        bucket_no = val % self.hash_table_size
        temp = self.linear_hash_list[bucket_no]
        while temp.has_next():
            temp = temp.next
        if len(temp.bucket) < self.bucket_size:
            temp.insert(val)
            self.elements = self.elements + 1
        else:
            temp.next = Bucket_LH()
            temp = temp.next
            temp.insert(val)
            self.elements = self.elements + 1

        if self.elements / (len(self.linear_hash_list) * self.bucket_size) >
self.load_factor:
            temp1 = Bucket_LH()
            temp2 = Bucket_LH()
            first = temp1
            second = temp2
            temp = self.linear_hash_list[self.next]
            if len(temp.bucket) == 0:
                self.linear_hash_list.append(second)

            for value in self.linear_hash_list[self.next].bucket:
                if value % (self.hash_table_size * 2) == self.next:
                    first.insert(value)
                else:
                    second.insert(value)

            current_bucket = self.linear_hash_list[self.next]
            while current_bucket.has_next():
                current_bucket = current_bucket.next
                for value in current_bucket.bucket:
                    if value % (self.hash_table_size * 2) == self.next:
                        if len(first.bucket) < self.bucket_size:
                            first.insert(value)
                        else:
                            first.next = Bucket_LH()
                            first = first.next
                            first.insert(value)
                    else:
                        if len(second.bucket) < self.bucket_size:
                            second.insert(value)
                        else:
                            second.next = Bucket_LH()
```

```
                            second = second.next
                            second.insert(value)
            self.linear_hash_list[self.next] = temp1
            self.linear_hash_list.append(temp2)
            self.next = self.next + 1

    def printLH(self):
        LH_list = []
        for bucket in self.linear_hash_list:
            LH_list.append(bucket.print_bucket())
            while bucket.has_next():
                bucket = bucket.next
                LH_list.append(" -> ")
                LH_list.append(bucket.print_bucket())
            LH_list.append(" || ")
        LH_list_string = ' '.join(map(str, LH_list))
        return LH_list_string

@app.route('/')
def home():
    return "To insert value use /LH(or EH)/insert/value, and to display linear(or
extendible) hash table /LH(or EH)"

table = LH()


@app.route('/LH')
def LH():
    return table.printLH()


@app.route('/LH/insert/<int:val>')
def insert_LH(val):
    table.insert(val)
    return f"{val} is inserted."

class bucket_eh (Resource):
        # values = []
        def __init__(self, local_depth=1):
            self.values = []
            self.local_depth = local_depth
        def reset(self):
            print ("deleted in descriptor object")
            self.values = []
        def insert_value(self, key):
```

```
            self.values.append(key)

class EH (Resource):
    def __init__(self):
        self.bucket_list = []
        self.directory = [None]*2
        self.global_depth = 1
    def insert_eh(self, key):
        bin_key = format(key, "09b")
        bit_compare = bin_key[(0-self.global_depth):]
        index_of_directory = int(bit_compare, 2) #for index of direcotry
        if(self.directory[index_of_directory] == None):
            b1 = bucket_eh()
            b1.insert_value(key)
            self.bucket_list.append(b1)
            self.directory[index_of_directory] = self.bucket_list[len(self.bucket_list)
- 1]

        else:
            b2 = self.directory[index_of_directory]
            if(b2.local_depth == self.global_depth):
                if(len(b2.values) == 2):
                    #increase global depth by 1
                    self.global_depth += 1
                    #first get all the keys of that bucket_eh and the key to be
inserted in a list
                    temp = []
                    for i in range(2):
                        temp.append(b2.values[i])
                    temp.append(key)

                    # distribute keys in the list according to their local depth
                    old_local_depth = b2.local_depth
                    b3_temp = bucket_eh(old_local_depth + 1)
                    b4_temp = bucket_eh(old_local_depth + 1)
                    for i1 in range(len(temp)):
                        bin_key_double = format(temp[i1], "09b")
                        bit_compare_double = bin_key_double[(0-self.global_depth):]
                        print(bit_compare_double)
                        index_of_directory_double = int(bit_compare_double, 2) #for
index of directory
                        only_bit_changed = bin_key_double[(0 - self.global_depth)]# :
(old_local_depth - self.global_depth)]
                        print(only_bit_changed)
                        if(only_bit_changed == '0'):
```

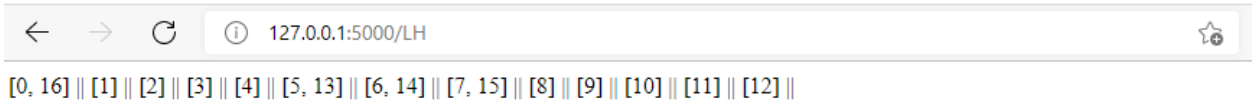
```
                    b3_temp.values.append(temp[i1])
                else:
                    b4_temp.values.append(temp[i1])

            print(b3_temp.values)
            print(b4_temp.values)
            if ((len(b4_temp.values)) > 2 or (len(b3_temp.values)) > 2):
                print('overflow may occur')
            #double the directory
            #make a directory with size equal to double the old directory
            directory_1 = [None]*(len(self.directory)*2)
            print('printing length of self.direcotry')
            print(len(self.directory))
            for i2 in range (len(self.directory)):
                if(self.directory[i2] !=  None):
                    temp = bucket_eh(self.directory[i2].local_depth)
                    for i3 in range (len(self.directory[i2].values)):
                        temp.insert_value(self.directory[i2].values[i3])
                    directory_1[i2] = temp
                    directory_1[i2 + len(self.directory)] = temp
                    print(directory_1[i2].values)
            print('done')
            if(len(b3_temp.values) >= 1):
                bin_key_temp_zero = format(b3_temp.values[0], "09b")
                bin_key_temp_zero = bin_key_temp_zero[(0-self.global_depth) :]
                directory_1[int(bin_key_temp_zero, 2)] = b3_temp
            if(len(b4_temp.values) >= 1):
                bin_key_temp_one = format(b4_temp.values[0], "09b")
                bin_key_temp_one = bin_key_temp_one[(0-self.global_depth) :]
                directory_1[int(bin_key_temp_one, 2)] = b4_temp
                # print(directory_1[0].values)
            # print(directory_1[1].values)
            # print(directory_1[2].values)
            # print(directory_1[3].values)
            self.directory = directory_1.copy()
        else:
            b2.values.append(key)
            self.directory[index_of_directory] = b2
    else:
        new_local_depth = b2.local_depth + 1
        if(len(b2.values) == 1):
            b2.values.append(key)
            self.directory[index_of_directory] = b2
        elif(len(b2.values) == 2):
            temp = []
```

```
                    for i in range(2):
                        temp.append(b2.values[i])
                    temp.append(key)

                    b3_temp = bucket_eh(new_local_depth)
                    b4_temp = bucket_eh(new_local_depth)

                    for i1 in range(len(temp)):
                        bin_key_double = format(temp[i1], "09b")
                        bit_compare_double = bin_key_double[(0-new_local_depth):]
                        print(bit_compare_double)
                        only_bit_changed = bin_key_double[(0 - new_local_depth)]# :
(old_local_depth - self.global_depth)]
                        print(only_bit_changed)
                        if(only_bit_changed == '0'):
                            b3_temp.values.append(temp[i1])
                        else:
                            b4_temp.values.append(temp[i1])
                    if(len(b3_temp.values) >= 1):
                        bin_key_temp_zero = format(b3_temp.values[0], "09b")
                        bin_key_temp_zero = bin_key_temp_zero[(0-new_local_depth) :]
                        self.directory[int(bin_key_temp_zero, 2)] = b3_temp
                    if(len(b3_temp.values) >= 1):
                        bin_key_temp_one = format(b4_temp.values[0], "09b")
                        bin_key_temp_one = bin_key_temp_one[(0-new_local_depth) :]
                        self.directory[int(bin_key_temp_one, 2)] = b4_temp




    def get_value(self):
        EH_list = []
        EH_list.append("(Directory index :: local depth - bucket values)")
        EH_list.append('\n')
        for i in range (len(self.directory)):
            if(self.directory[i] != None):
                EH_list.append(i)
                EH_list.append(" :: ")
                EH_list.append(self.directory[i].local_depth)
                EH_list.append(" - ")
                EH_list.append(self.directory[i].values)
                print(i, self.directory[i].local_depth, self.directory[i].values)
                if(i != len(self.directory) - 1):
                    EH_list.append("-->")
        EH_list_string = ' '.join(map(str, EH_list))
```

```
        return EH_list_string



hashing_1 = EH()
@app.route('/EH/insert/<int:key>')
def insert_eh(key):
    hashing_1.insert_eh(key)
    return f"{key} is inserted"

@app.route('/EH')
def EH():
    return hashing_1.get_value()


app.run(host ='0.0.0.0', port = 5001, debug=True)
```

Linear Hashing

Example 1:
Bucket size = 2
Load factor = 0.7
Initial Hash table size = 4
Values inserted: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
Hash table Output at the end of the insertion:



Example 2:
Bucket size = 2
Load factor = 0.7
Initial Hash table size = 4
Values Inserted: 0,8,16,24,32,40,48,56,64,72,80,88,96,104,112,120
Hash table Output at the end of the insertion:

Extendible Hashing

Bucket size = 2
Inserting values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11