Practical 5: Linear Hashing

GAHAN SARAIYA, 18MCEC10

18mcec10@nirmauni.ac.in

I. INTRODUCTION

Aim of this practical is to implementing Linear Hashing.

- linear growth of bucket array
- dynamic decision to increase bucket size
- bucket split criteria - $average\ bucket\ occupancy > threshold$

II. LOGIC

1. Initialize
   - bucket array with 2 buckets initially
   - phase to 1
   - threshold to 70%
   - hash function $h(element, without\_splitting) = element \,\%\, 2^{phase\_number}$
   - hash function $h(element, during\_splitting) = element \,\%\, 2^{phase\_number+1}$
2. For each Insertion check for Average occupancy and decide whether split require or not.
3. can insert node directly into bucket if no splitting is require (when average occupancy < threshold) but might need chaining
4. otherwise splitting to be done from index 0 (first bucket)

III. IMPLEMENTATION

The code is implemented in Python as below

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""
Author: Gahan Saraiya
GiT: https://github.com/gahan9
StackOverflow: https://stackoverflow.com/users/story/7664524

Implementation of linear hashing
"""
from collections import OrderedDict
from math import log2, log
```

```python


class LinearHashing(object):
    def __init__(self, *args, **kwargs):
        self.threshold = kwargs.get('threshold', 0.7)
        self.data_capacity_per_bucket = kwargs.get('data_capacity_per_bucket', 2)
         ↪ # capacity to store data per bucket
        self.total_data = 0  # keep count of total inserted record
        self.buffer = {key: [] for key in range(2)}  # initial buffer
        self.index_counter = 0  # keep index counter from where we are supposed
         ↪ to start split in phase
        self.previous_phase = 1  # keeping phase number to reset index counter
        self.has_title = None

    @property
    def current_phase(self):
        return int(log2(len(self.buffer)))

    @property
    def buffer_capacity(self):
        return self.data_capacity_per_bucket * len(self.buffer)

    @property
    def threshold_outbound(self):
        return ((self.total_data + 1) / self.buffer_capacity) > self.threshold

    def hash_function(self, value, flag=0):
        """

        :param value: value on which hash function to be applied
        :param flag: set flag to 1 if splitting the bucket
        :return:
        """
        if not flag:
            # if no splitting require
            return value % (2 ** self.previous_phase)
        else:
            # if splitting require
            return value % (2 ** (self.current_phase + 1))

    def set_index_counter_if(self):
        """
        set index counter from where splitting to be done to 0 when phase changes
        :return: None
        """
        if self.current_phase != self.previous_phase:
            self.index_counter = 0
            self.previous_phase = self.current_phase
```

```python

    def insert(self, value, print_status=0):
        """

        :param value: value to be inserted
        :param print_status: set to 1 if
        :return:
        """
        self.set_index_counter_if()
        buffer_capacity_beefore_insert = self.buffer_capacity
        if self.threshold_outbound:
            # buffer to be extend
            self.buffer[len(self.buffer)] = []
            buffer_index = self.hash_function(value)
            self.buffer[buffer_index] = self.buffer.setdefault(buffer_index, []) + [value]
            # bucket to be split
            bucket_to_split = self.buffer[self.index_counter]
            self.buffer[self.index_counter] = []
            for data in bucket_to_split:
                buffer_idx = self.hash_function(data, flag=1)
                self.buffer[buffer_idx] = self.buffer.setdefault(buffer_idx, []) + [data]
            self.index_counter += 1
        else:
            buffer_index = self.hash_function(value)
            # self.buffer[buffer_index].append(value)
            self.buffer[buffer_index] = self.buffer.setdefault(buffer_index, []) + [value]
        self.total_data += 1

        if print_status:
            self.pretty_print(value, buffer_capacity_beefore_insert)
        return True

    def pretty_print(self, value, buffer_capacity_beefore_insert):
        data_dict = OrderedDict()
        data_dict["Sr No."] = self.total_data
        data_dict["Element"] = value
        data_dict["SplitIndex"] = self.index_counter
        data_dict["Phase"] = self.current_phase
        data_dict["Ratio"] = round(self.total_data / buffer_capacity_beefore_insert, 2)
        data_dict["Threshold"] = self.threshold
        # data_dict["Previous Phase"] = self.previous_phase
        if not self.has_title:
            print(" ".join(data_dict.keys()) + " " + "RESULT")
            self.has_title = True
```

```python
        print(" ".join("{:^{}s}".format(str(v), len(k)) for k, v in data_dict.items()), end=" ")
        print(self.buffer)

    def delete(self):
        return NotImplementedError

    def __repr__(self):
        return "\n".join(
            "{:>03d} -> {}".format(i, self.buffer[i]) if len(self.buffer[i]) <= self.data_capacity_per_bucket
            else "{:>03d} -> {} => {}".format(i, self.buffer[i][:self.data_capacity_per_bucket], self.buffer[i][self.data_capacity_per_bucket:])
            for i in sorted(self.buffer))

    def __str__(self):
        return "\n".join(
            "{:>03d} -> {}".format(i, self.buffer[i]) if len(self.buffer[i]) <= self.data_capacity_per_bucket
            else "{:>03d} -> {} => {}".format(i, self.buffer[i][:self.data_capacity_per_bucket], self.buffer[i][self.data_capacity_per_bucket:])
            for i in sorted(self.buffer))

def test(flag=None):
    """
    test fuction to test functionality of program
    """
    if not flag:
        capacity = 3
    elif len(flag) == 2:
        capacity = int(flag[1])
    elif len(flag) > 2:
        capacity, total_elements = map(int, flag[1:3])
        print("Capacity per bucket (without chaining): {}".format(capacity))
        hash_bucket = LinearHashing(data_capacity_per_bucket=capacity, threshold=0.7)

        import random
        input_lis = list(random.randint(0, 500) for i in range(total_elements))
        for i in input_lis:
            hash_bucket.insert(i, print_status=1)
    print("Capacity per bucket (without chaining): {}".format(capacity))
    hash_bucket = LinearHashing(data_capacity_per_bucket=capacity, threshold=0.7)
    input_lis = [3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15]
    for i in input_lis:
        hash_bucket.insert(i, print_status=1)
```

```


    print("Final Bucket Status")
    print(hash_bucket)


if __name__ == "__main__":
    import sys
    if len(sys.argv) > 1:
        test(sys.argv)
    else:
        test(0)
```

.1 Output

```
Capacity per bucket (without chaining): 2
Sr No. Element SplitIndex Phase Ratio Threshold RESULT
  1       3        0        1   0.25     0.7    {0: [], 1: [3]}
  2       2        0        1    0.5     0.7    {0: [2], 1: [3]}
  3       4        1        1   0.75     0.7    {0: [4], 1: [3], 2: [2]}
  4       1        1        1   0.67     0.7    {0: [4], 1: [3, 1], 2: [2]}
  5       8        2        2   0.83     0.7    {0: [4, 8], 1: [1], 2: [2], 3:
  ↪  [3]}
  6      14        1        2   0.75     0.7    {0: [8], 1: [1], 2: [2, 14], 3:
  ↪  [3], 4: [4]}
  7       5        1        2    0.7     0.7    {0: [8], 1: [1, 5], 2: [2, 14], 3:
  ↪  [3], 4: [4]}
  8      10        2        2    0.8     0.7    {0: [8], 1: [1], 2: [2, 14, 10],
  ↪  3: [3], 4: [4], 5: [5]}
  9       7        3        2   0.75     0.7    {0: [8], 1: [1], 2: [2, 10], 3:
  ↪  [3, 7], 4: [4], 5: [5], 6: [14]}
  10     24        4        3   0.71     0.7    {0: [8, 24], 1: [1], 2: [2, 10],
  ↪  3: [3], 4: [4], 5: [5], 6: [14], 7: [7]}
  11     17        0        3   0.69     0.7    {0: [8, 24], 1: [1, 17], 2: [2,
  ↪  10], 3: [3], 4: [4], 5: [5], 6: [14], 7: [7]}
  12     13        1        3   0.75     0.7    {0: [], 1: [1, 17], 2: [2, 10], 3:
  ↪  [3], 4: [4], 5: [5, 13], 6: [14], 7: [7], 8: [8, 24]}
  13     15        2        3   0.72     0.7    {0: [], 1: [1, 17], 2: [2, 10], 3:
  ↪  [3], 4: [4], 5: [5, 13], 6: [14], 7: [7, 15], 8: [8, 24], 9: []}
Final Bucket Status
000 -> []
001 -> [1, 17]
002 -> [2, 10]
003 -> [3]
004 -> [4]
005 -> [5, 13]
006 -> [14]
```

```
007 -> [7, 15]
008 -> [8, 24]
009 -> []
```

.2 Output

```
Capacity per bucket (without chaining): 3
Sr No. Element SplitIndex Phase Ratio Threshold RESULT
  1       3        0        1   0.17     0.7    {0: [], 1: [3]}
  2       2        0        1   0.33     0.7    {0: [2], 1: [3]}
  3       4        0        1    0.5     0.7    {0: [2, 4], 1: [3]}
  4       1        0        1   0.67     0.7    {0: [2, 4], 1: [3, 1]}
  5       8        1        1   0.83     0.7    {0: [4, 8], 1: [3, 1], 2: [2]}
  6      14        1        1   0.67     0.7    {0: [4, 8, 14], 1: [3, 1], 2:
  ↪  [2]}
  7       5        2        2   0.78     0.7    {0: [4, 8, 14], 1: [1], 2: [2], 3:
  ↪  [3], 5: [5]}
  8      10        0        2   0.53     0.7    {0: [4, 8, 14], 1: [1], 2: [2,
  ↪  10], 3: [3], 5: [5]}
  9       7        0        2    0.6     0.7    {0: [4, 8, 14], 1: [1], 2: [2,
  ↪  10], 3: [3, 7], 5: [5]}
  10     24        0        2   0.67     0.7    {0: [4, 8, 14, 24], 1: [1], 2: [2,
  ↪  10], 3: [3, 7], 5: [5]}
  11     17        1        2   0.73     0.7    {0: [8, 24], 1: [1, 17], 2: [2,
  ↪  10], 3: [3, 7], 5: [], 4: [4], 6: [14]}
  12     13        1        2   0.57     0.7    {0: [8, 24], 1: [1, 17, 13], 2:
  ↪  [2, 10], 3: [3, 7], 5: [], 4: [4], 6: [14]}
  13     15        1        2   0.62     0.7    {0: [8, 24], 1: [1, 17, 13], 2:
  ↪  [2, 10], 3: [3, 7, 15], 5: [], 4: [4], 6: [14]}
Final Bucket Status
000 -> [8, 24]
001 -> [1, 17, 13]
002 -> [2, 10]
003 -> [3, 7, 15]
004 -> [4]
005 -> []
006 -> [14]
```

.3 Output

```
Capacity per bucket (without chaining): 10
Sr No. Element SplitIndex Phase Ratio Threshold RESULT
  1       3        0        1   0.05     0.7    {0: [], 1: [3]}
  2       2        0        1    0.1     0.7    {0: [2], 1: [3]}
  3       4        0        1   0.15     0.7    {0: [2, 4], 1: [3]}
  4       1        0        1    0.2     0.7    {0: [2, 4], 1: [3, 1]}
```

```
  5      8        0        1   0.25     0.7    {0: [2, 4, 8], 1: [3, 1]}
  6     14        0        1    0.3     0.7    {0: [2, 4, 8, 14], 1: [3, 1]}
  7      5        0        1   0.35     0.7    {0: [2, 4, 8, 14], 1: [3, 1, 5]}
  8     10        0        1    0.4     0.7    {0: [2, 4, 8, 14, 10], 1: [3, 1,
  ↪  5]}
  9      7        0        1   0.45     0.7    {0: [2, 4, 8, 14, 10], 1: [3, 1,
  ↪  5, 7]}
  10    24        0        1    0.5     0.7    {0: [2, 4, 8, 14, 10, 24], 1: [3,
  ↪  1, 5, 7]}
  11    17        0        1   0.55     0.7    {0: [2, 4, 8, 14, 10, 24], 1: [3,
  ↪  1, 5, 7, 17]}
  12    13        0        1    0.6     0.7    {0: [2, 4, 8, 14, 10, 24], 1: [3,
  ↪  1, 5, 7, 17, 13]}
  13    15        0        1   0.65     0.7    {0: [2, 4, 8, 14, 10, 24], 1: [3,
  ↪  1, 5, 7, 17, 13, 15]}
Final Bucket Status
000 -> [2, 4, 8, 14, 10, 24]
001 -> [3, 1, 5, 7, 17, 13, 15]
```

IV. Summary

Comparison with Extendible Hashing

N	Number of records
B	Number of buckets
b	bucket capacity
s	Number of successful searches
u	Number of unsuccessful searches
b_s	1 + number of buckets accessed for successful search
b_u	1 + number of buckets accessed for unsuccessful search

Factor	Linear Hashing	Extendible Hashing
Storage utilization	$\frac{N}{B \times b}$	$\frac{N}{B \times b}$
Average unsuccessful search cost	$\frac{b_u}{u}$	$\frac{b_u}{u}$
Average unsuccessful search cost	$\frac{b_s}{s}$	$\frac{b_s}{s}$
Split(expansion) Cost	1 access to read primary buckets + k accesses to read k overflow buckets + 1 access to write old bucket + extra accesses to write the overflow buckets attached to old and new buckets	1 access to write old bucket + 1 access to write new bucket + extra access to write the overflow buckets attached to old and new buckets + accesses needed to update the directory pointers if the directory resides on the secondary storage
Insertion Cost	Unsuccessful search cost + Split cost	Unsuccessful search cost + Split cost

Conclusion

- instead of exponential growth as in extendible hashing it's directory structure grows linearly
- ✓ more efficient in terms of space utilization compared to extendible hashing
- ✓ hash function calculated dynamically
- ✗ for skew or non uniform data overflow chaining might be bottleneck in terms of search complexity as it may needs linear search for many records
- ✗ overhead of computing threshold and splitting node if require on each **insert** operation