

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

```
1  #!/usr/bin/python3
2  # -*- coding: utf-8 -*-
3  """
4  Author: Gahan Saraiya
5  GiT: https://github.com/gahan9
6  StackOverflow: https://stackoverflow.com/users/story/7664524
7
8  Implementation of linear hashing
9  """
10 from collections import OrderedDict
11 from math import log2, log
```

```
12
13
14 class LinearHashing(object):
15     def __init__(self, *args, **kwargs):
16         self.threshold = kwargs.get('threshold', 0.7)
17         self.data_capacity_per_bucket = kwargs.get('data_capacity_per_bucket', 2)
18         ↪ # capacity to store data per bucket
19         self.total_data = 0 # keep count of total inserted record
20         self.buffer = {key: [] for key in range(2)} # initial buffer
21         self.index_counter = 0 # keep index counter from where we are supposed
22         ↪ to start split in phase
23         self.previous_phase = 1 # keeping phase number to reset index counter
24         self.has_title = None
25
26     @property
27     def current_phase(self):
28         return int(log2(len(self.buffer)))
29
30     @property
31     def buffer_capacity(self):
32         return self.data_capacity_per_bucket * len(self.buffer)
33
34     @property
35     def threshold_outbound(self):
36         return ((self.total_data + 1) / self.buffer_capacity) > self.threshold
37
38     def hash_function(self, value, flag=0):
39         """
40         :param value: value on which hash function to be applied
41         :param flag: set flag to 1 if splitting the bucket
42         :return:
43         """
44         if not flag:
45             # if no splitting require
46             return value % (2 ** self.previous_phase)
47         else:
48             # if splitting require
49             return value % (2 ** (self.current_phase + 1))
50
51     def set_index_counter_if(self):
52         """
53         set index counter from where splitting to be done to 0 when phase changes
54         :return: None
55         """
56         if self.current_phase != self.previous_phase:
57             self.index_counter = 0
58             self.previous_phase = self.current_phase
```

```

58
59 def insert(self, value, print_status=0):
60     """
61
62     :param value: value to be inserted
63     :param print_status: set to 1 if
64     :return:
65     """
66     self.set_index_counter_if()
67     buffer_capacity_before_insert = self.buffer_capacity
68     if self.threshold_outbound:
69         # buffer to be extend
70         self.buffer[len(self.buffer)] = []
71         buffer_index = self.hash_function(value)
72         self.buffer[buffer_index] = self.buffer.setdefault(buffer_index, []) +
73             ↪ [value]
74         # bucket to be split
75         bucket_to_split = self.buffer[self.index_counter]
76         self.buffer[self.index_counter] = []
77         for data in bucket_to_split:
78             buffer_idx = self.hash_function(data, flag=1)
79             self.buffer[buffer_idx] = self.buffer.setdefault(buffer_idx, []) +
80                 ↪ [data]
81         self.index_counter += 1
82     else:
83         buffer_index = self.hash_function(value)
84         # self.buffer[buffer_index].append(value)
85         self.buffer[buffer_index] = self.buffer.setdefault(buffer_index, []) +
86             ↪ [value]
87     self.total_data += 1
88
89     if print_status:
90         self.pretty_print(value, buffer_capacity_before_insert)
91     return True
92
93 def pretty_print(self, value, buffer_capacity_beefore_insert):
94     data_dict = OrderedDict()
95     data_dict["Sr No."] = self.total_data
96     data_dict["Element"] = value
97     data_dict["SplitIndex"] = self.index_counter
98     data_dict["Phase"] = self.current_phase
99     data_dict["Ratio"] = round(self.total_data /
100         ↪ buffer_capacity_beefore_insert, 2)
101     data_dict["Threshold"] = self.threshold
102     # data_dict["Previous Phase"] = self.previous_phase
103     if not self.has_title:
104         print(" ".join(data_dict.keys()) + " " + "RESULT")
105         self.has_title = True

```

```

102     print(" ".join("{:~{}}s".format(str(v), len(k)) for k, v in
        ↪ data_dict.items()), end=" ")
103     print(self.buffer)
104
105     def delete(self):
106         return NotImplementedError
107
108     def __repr__(self):
109         return "\n".join(
110             "{:>03d} -> {}".format(i, self.buffer[i]) if len(self.buffer[i])
        ↪ <= self.data_capacity_per_bucket
111             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))
112
113     def __str__(self):
114         return "\n".join(
115             "{:>03d} -> {}".format(i, self.buffer[i]) if len(self.buffer[i])
        ↪ <= self.data_capacity_per_bucket
116             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))
117
118
119
120
121     def test(flag=None):
122         """
123         test fuction to test functionality of program
124         """
125         if not flag:
126             capacity = 3
127         elif len(flag) == 2:
128             capacity = int(flag[1])
129         elif len(flag) > 2:
130             capacity, total_elements = map(int, flag[1:3])
131             print("Capacity per bucket (without chaining): {}".format(capacity))
132             hash_bucket = LinearHashing(data_capacity_per_bucket=capacity,
        ↪ threshold=0.7)
133
134             import random
135             input_lis = list(random.randint(0, 500) for i in range(total_elements))
136             for i in input_lis:
137                 hash_bucket.insert(i, print_status=1)
138             print("Capacity per bucket (without chaining): {}".format(capacity))
139             hash_bucket = LinearHashing(data_capacity_per_bucket=capacity, threshold=0.7)
140             input_lis = [3, 2, 4, 1, 8, 14, 5, 10, 7, 24, 17, 13, 15]
141             for i in input_lis:

```

```

142         hash_bucket.insert(i, print_status=1)
143
144     print("Final Bucket Status")
145     print(hash_bucket)
146
147
148 if __name__ == "__main__":
149     import sys
150
151     if len(sys.argv) > 1:
152         test(sys.argv)
153     else:
154         test()

```

.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