

# HW8

Prathamesh Pawar

March 2023

Sources: [stackoverflow](#), [GeeksforGeeks](#)

# 1 Problem 1: Hash Function

```
Users > prathameshpawar > Desktop > Algo_homework > HW8 > HW8.py > Node > __init__
1 import re
2 import string
3 import math
4 import statistics
5 from collections import Counter
6
7
8
9 with open('alice.txt', 'r') as file:
10     alice = file.read()
11
12 class Node:
13     def __init__(self, value, next=None, key=None):
14         self.value = value
15         self.next = next
16         self.key = key
17
18 class hash_node:
19     def __init__(self, key, value, next=None):
20         self.key = key
21         self.value = value
22         self.next = next
23
24 class linked_list:
25
26     def __init__(self, head=None):
27         self.head = head
28
29     def display(self):
30         node = self.head
31
32         while node.next != None:
33             print(node.value)
34             node = node.next
35
36     def insert(self, key, value, Maxhash):
37
38         node = self.head
39
40         map_key = get_hash_key(key, Maxhash)
41         print(key, map_key)
42
43
44
45         while(map_key > 0):
46             node = node.next
47             map_key -= 1
48
49         node1 = node.key
50         if node1.head == None:
51             node1.head = hash_node(key, value)
52         else:
53             x = hash_node(key, value)
54             x.next = node1.head
55             node1.head = x
56
57
58
59     def delete(self, key_to_be_removed, Maxhash):
60
61         node = self.head
62
63         map_key = get_hash_key(key, Maxhash)
64         while(map_key > 0):
65             node = node.next
66             map_key -= 1
67
68         node1 = node.key
69
70         while node1.next != None:
71             if node1.next.key == key_to_be_removed:
72                 node1.next = node1.next.next
73                 break
74             node1 = node1.next
75
76
77
78
79     def increase(self, key_to_increase, increment):
80
81         node = self.head
82
83         while node.next != None:
84
85             if node.key == key_to_increase:
86                 node.value += increment
87                 break
```

```

88         node = node.next
89
90
91
92     def find_key(self, key_to_find):
93
94         node = self.head
95
96         while node!=None:
97
98             node1 = node.key.head
99
100             while node1!= None:
101
102                 if node1.key == key_to_find:
103                     return node1
104                 node1 = node1.next
105             node = node.next
106
107     def list_all_keys(self):
108
109         keys_dict={}
110         node = self.head
111
112         while node.next!=None:
113             node1 = node.key.head
114
115             while node1.next != None:
116
117                 keys_dict[node1.key] = node1.value
118                 print(node1.key,node1.value)
119                 node1 = node1.next
120             node = node.next
121         return keys_dict
122
123     def get_collisions(self):
124         x=[]
125         node = self.head
126
127         while node != None:
128             c=0
129             node1 = node.key.head
130
131             while node1!=None:
132                 c+=1
133                 node1 = node1.next
134             node = node.next
135             x.append(c)
136
137         return x
138
139
140
141     def clean_text(s):
142
143         s = s.lower()
144         s = re.sub(r'[\n\t]', ' ', s)
145         s = re.sub(r'[-A-Za-z ]', ' ', s)
146         s = s.split()
147         s = [i.strip() for i in s if i.strip()]
148
149         return s
150
151     def get_ascii_addition(s):
152         x=0
153         for n,i in enumerate(s):
154             x+=(ord(i))*2*(3+n)*2-(int(n/35))*2
155
156         return x
157
158     def get_hash_key(s, Maxhash):
159         s = get_ascii_addition(s)
160         s= math.floor(Maxhash*((s+0.6180339887)*1))
161         return s
162
163     def create_hash_map(Maxhash):
164         temp = None
165         for i in reversed(list(range(Maxhash))):
166             temp = Node(1,temp,linked_list())
167
168         hash_map = linked_list(temp)
169
170         return hash_map
171
172
173     def main():
174         Maxhash = 30

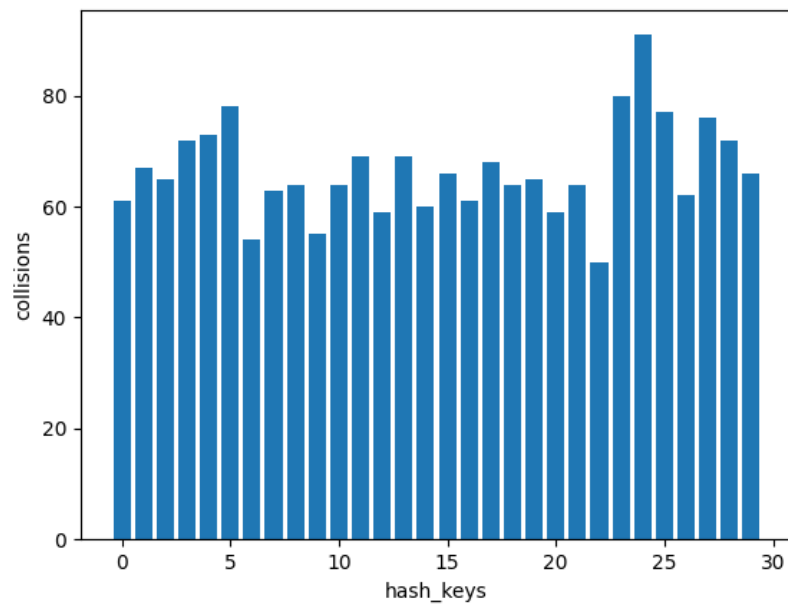
```

```

175 x = clean_text(alice)
176 x_dict = dict(Counter(x))
177 hash_map = create_hash_map(Maxhash)
178
179 for k, v in x_dict.items():
180     hash_map.insert(k, v, Maxhash)
181
182 print(statistics.variance(hash_map.get_collisions()))
183
184
185
186
187 if __name__ == "__main__":
188     main()

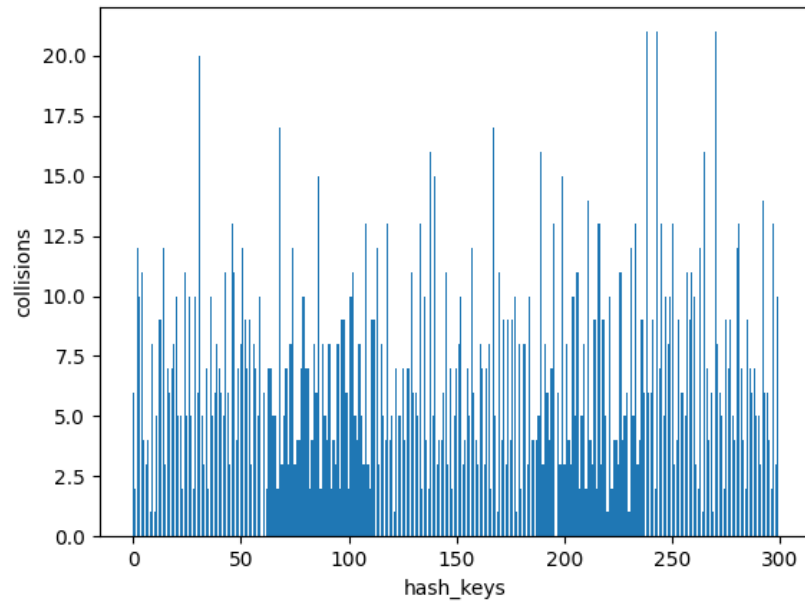
```

## 1.1 Results 1:



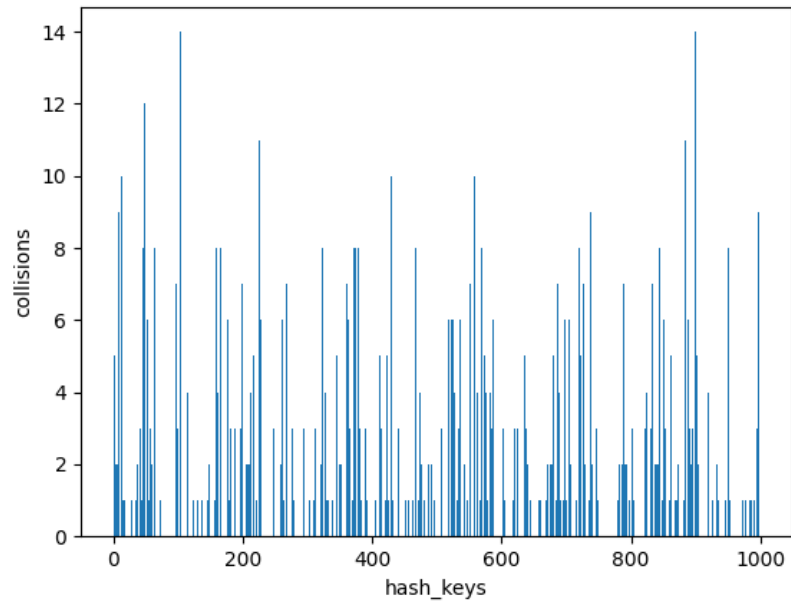
Variance = 71.4

## 1.2 Results 2:



Variance = 16.08

### 1.3 Results 3:



Variance = 5.70

## 2 Problem 2: RB Tree

```
1
2 import sys
3
4 class Node():
5     def __init__(self, item):
6         self.item = item
7         self.parent = None
8         self.left = None
9         self.right = None
10        self.color = 1
11
12
13 class RedBlackTree():
14     def __init__(self):
15         self.TNULL = Node(0)
16         self.TNULL.color = 0
17         self.TNULL.left = None
18         self.TNULL.right = None
19         self.root = self.TNULL
20
21     def pre_order_helper(self, node):
22         if node != TNULL:
23             sys.stdout.write(node.item + " ")
24             self.pre_order_helper(node.left)
25             self.pre_order_helper(node.right)
26
27     def in_order_helper(self, node):
28         if node != TNULL:
29             self.in_order_helper(node.left)
30             sys.stdout.write(node.item + " ")
31             self.in_order_helper(node.right)
32
33     def post_order_helper(self, node):
34         if node != TNULL:
35             self.post_order_helper(node.left)
36             self.post_order_helper(node.right)
37             sys.stdout.write(node.item + " ")
38
39     def search_tree_helper(self, node, key):
40         if node == TNULL or key == node.item:
41             return node
42
43         if key < node.item:
44             return self.search_tree_helper(node.left, key)
45         return self.search_tree_helper(node.right, key)
46
47     def __rb_transplant(self, u, v):
48         if u.parent == None:
49             self.root = v
50         elif u == u.parent.left:
51             u.parent.left = v
52         else:
53             u.parent.right = v
54         v.parent = u.parent
55
56     def delete_node_helper(self, node, key):
57         z = self.TNULL
58         while node != self.TNULL:
59             if node.item == key:
60                 z = node
61
62             if node.item <= key:
63                 node = node.right
64             else:
65                 node = node.left
66
67         if z == self.TNULL:
68             print("Cannot find key in the tree")
69             return
70
71         y = z
72         y_original_color = y.color
73         if z.left == self.TNULL:
74             x = z.right
75             self.__rb_transplant(z, z.right)
76         elif (z.right == self.TNULL):
77             x = z.left
78             self.__rb_transplant(z, z.left)
79         else:
80             y = self.minimum(z.right)
81             y_original_color = y.color
82             x = y.right
83             if y.parent == z:
84                 x.parent = y
85             else:
86                 self.__rb_transplant(y, y.right)
87                 y.right = x
88                 y.right.parent = y
89
90             self.__rb_transplant(z, y)
91             y.left = z.left
92             y.left.parent = y
93             y.color = z.color
94
95
```

```

95
96 def fix_insert(self, k):
97     while k.parent.color == 1:
98         if k.parent == k.parent.parent.right:
99             u = k.parent.parent.left
100             if u.color == 1:
101                 u.color = 0
102                 k.parent.color = 0
103                 k.parent.parent.color = 1
104                 k = k.parent.parent
105             else:
106                 if k == k.parent.left:
107                     k = k.parent
108                     self.right_rotate(k)
109                     k.parent.color = 0
110                     k.parent.parent.color = 1
111                     self.left_rotate(k.parent.parent)
112                 else:
113                     u = k.parent.parent.right
114                     if u.color == 1:
115                         u.color = 0
116                         k.parent.color = 0
117                         k.parent.parent.color = 1
118                         k = k.parent.parent
119                     else:
120                         if k == k.parent.right:
121                             k = k.parent
122                             self.left_rotate(k)
123                             k.parent.color = 0
124                             k.parent.parent.color = 1
125                             self.right_rotate(k.parent.parent)
126                         if k == self.root:
127                             break
128             self.root.color = 0
129
130
131 def __print_helper(self, node, indent, last):
132     if node != self.TNULL:
133         sys.stdout.write(indent)
134         if last:
135             sys.stdout.write("R----")
136             indent += "    "
137         else:
138             sys.stdout.write("L----")
139             indent += "    |    "
140
141         s_color = "RED" if node.color == 1 else "BLACK"
142         print(str((node.item) + "(" + s_color + ")"))
143         self.__print_helper(node.left, indent, False)
144         self.__print_helper(node.right, indent, True)
145
146 def preorder(self):
147     self.pre_order_helper(self.root)
148
149 def inorder(self):
150     self.in_order_helper(self.root)
151
152 def postorder(self):
153     self.post_order_helper(self.root)
154
155 def searchTree(self, k):
156     return self.search_tree_helper(self.root, k)
157
158 def minimum(self, node):
159     while node.left != self.TNULL:
160         node = node.left
161     return node
162
163 def maximum(self, node):
164     while node.right != self.TNULL:
165         node = node.right
166     return node
167

```



```

168     def successor(self, x):
169         if x.right != self.TNULL:
170             return self.minimum(x.right)
171
172         y = x.parent
173         while y != self.TNULL and x == y.right:
174             x = y
175             y = y.parent
176         return y
177
178     def predecessor(self, x):
179         if (x.left != self.TNULL):
180             return self.maximum(x.left)
181
182         y = x.parent
183         while y != self.TNULL and x == y.left:
184             x = y
185             y = y.parent
186         return y
187
188     def left_rotate(self, x):
189         y = x.right
190         x.right = y.left
191         if y.left != self.TNULL:
192             y.left.parent = x
193
194         y.parent = x.parent
195         if x.parent == None:
196             self.root = y
197         elif x == x.parent.left:
198             x.parent.left = y
199         else:
200             x.parent.right = y
201         y.left = x
202         x.parent = y
203
204     def right_rotate(self, x):
205         y = x.left
206         x.left = y.right
207         if y.right != self.TNULL:
208             y.right.parent = x
209
210         y.parent = x.parent
211         if x.parent == None:
212             self.root = y
213         elif x == x.parent.right:
214             x.parent.right = y
215         else:
216             x.parent.left = y
217         y.right = x
218         x.parent = y
219
220     def height(self, node):
221         if node == self.TNULL:
222             return 0
223         else:
224             left_height = self.height(node.left)
225             right_height = self.height(node.right)
226
227             if left_height > right_height:
228                 return left_height + 1
229             else:
230                 return right_height + 1
231
232

```

```

232
233     def insert(self, key):
234         node = Node(key)
235         node.parent = None
236         node.item = key
237         node.left = self.TNULL
238         node.right = self.TNULL
239         node.color = 1
240
241         y = None
242         x = self.root
243
244         while x != self.TNULL:
245             y = x
246             if node.item < x.item:
247                 x = x.left
248             else:
249                 x = x.right
250
251         node.parent = y
252         if y == None:
253             self.root = node
254         elif node.item < y.item:
255             y.left = node
256         else:
257             y.right = node
258
259         if node.parent == None:
260             node.color = 0
261             return
262
263         if node.parent.parent == None:
264             return
265
266         self.fix_insert(node)
267
268     def insert_array(self, array):
269         for key in array:
270             self.insert(key)
271
272
273
274     def get_root(self):
275         return self.root
276
277     def delete_node(self, item):
278         self.delete_node_helper(self.root, item)
279
280     def print_tree(self):
281         self.__print_helper(self.root, "", True)
282
283
284 if __name__ == "__main__":
285     bst = RedBlackTree()
286
287     bst.insert_array([5,4,6,0,5,17])
288     bst.insert(55)
289     bst.print_tree()
290
291     print("Result")
292     bst.delete_node(4)
293     bst.print_tree()
294

```

### 3 Problem 2: Skiplist

```
1 import random
2
3 class Node:
4     def __init__(self, key, level):
5         self.key = key
6         self.forward = [None]*(level+1)
7
8 class Skiplist:
9     def __init__(self, max_level, probability):
10        self.max_level = max_level
11        self.probability = probability
12        self.header = self.create_node(self.max_level, -1)
13        self.level = 0
14
15    def create_node(self, level, key):
16        node = Node(key, level)
17        return node
18
19    def random_level(self):
20        level = 0
21        while random.random() < self.probability and level < self.max_level:
22            level += 1
23        return level
24
25    def insert(self, key):
26        update = [None]*(self.max_level+1)
27        current = self.header
28        for i in range(self.level, -1, -1):
29            while current.forward[i] and current.forward[i].key < key:
30                current = current.forward[i]
31            update[i] = current
32            current = current.forward[0]
33        if current == None or current.key != key:
34            new_level = self.random_level()
35            if new_level > self.level:
36                for i in range(self.level+1, new_level+1):
37                    update[i] = self.header
38                self.level = new_level
39            node = self.create_node(new_level, key)
40            for i in range(new_level+1):
41                node.forward[i] = update[i].forward[i]
42                update[i].forward[i] = node
43
44    def delete(self, search_key):
45
46        update = [None]*(self.max_level+1)
47        node = self.header
48
49
50
51        for i in range(self.level, -1, -1):
52            while(node.forward[i] and node.forward[i].key < search_key):
53                node = node.forward[i]
54            update[i] = node
55
56
57        node = node.forward[0]
58
59
60        if node != None and node.key == search_key:
61
62            for i in range(self.level+1):
63
64                if update[i].forward[i] != node:
65                    break
66                update[i].forward[i] = node.forward[i]
67
68            while(self.level>0 and self.header.forward[self.level] == None):
69                self.level -= 1
70
71
72    def lookup(self, key):
73        node = self.header
74
75        for i in range(self.level, -1, -1):
76            while(node.forward[i] and
77                node.forward[i].key < key):
78                node = node.forward[i]
79
80        node = node.forward[0]
81
82        if node and node.key == key:
83            print("The Skiplist has the value ", key)
84
```

```

86     def display_list(self):
87         print("Skip List")
88         head = self.header
89         for level in range(self.level-1):
90             print("Level {}: ".format(level), end=" ")
91             node = head.forward[level]
92             while node != None:
93                 print(node.key, end=" ")
94                 node = node.forward[level]
95             print("")
96
97
98
99     def main():
100         skip_list = SkipList(3, 0.5)
101
102         skip_list.display_list()
103
104
105 if __name__ == "__main__":
106     main()

```