

cs 5800 - hw8

Yijing Xiao

November 2021

1 Implement a hash for text

```
1 import re
2 MAXHASH = 2500
3
4 class Node:
5     def __init__(self, k, val):
6         self.next = None
7         self.key = k
8         self.value = val
9
10 class LinkedList:
11     def __init__(self):
12         self.head = None
13         self.size = 0
14
15     def add_front(self, key, value):
16         newNode = Node(key, value)
17         newNode.next = self.head
18         self.head = newNode
19         self.size += 1
20
21     def remove(self, key):
22         if self.head is None:
23             return False
24         if self.head.key == key:
```

```

25         self.head = self.head.next
26         self.size = self.size - 1
27         return True
28     cur = self.head.next
29     prev = self.head
30     while cur is not None:
31         if cur.key == key:
32             prev.next = cur.next
33             self.size = self.size - 1
34             return True
35         prev = cur
36         cur = cur.next
37     return False
38
39     def search(self, key):
40         if self.head is not None:
41             cur = self.head
42             while cur is not None:
43                 if cur.key == key:
44                     return cur
45                 cur = cur.next
46         return None
47
48     def iter(self):
49         if not self.head:
50             return
51         cur = self.head
52         print(cur.key, ":", cur.value)
53         while cur.next:
54             cur = cur.next
55             print(cur.key, ":", cur.value)
56
57     def hashFunction(key):
58         hash_num = 0
59         for i in key:
60             hash_num += ord(i)
61         return hash_num
62

```

```

63 class HashMap:
64     def __init__(self, capacity, function):
65         self.buckets = []
66         for i in range(capacity):
67             self.buckets.append(LinkedList())
68         #capacity = the total number of buckets to be
69             created in the hash table
70         self.maxhash = capacity
71         #function = the hash function to use for
72             hashing
73         self.hash_function = function
74         self.size = 0
75
76 #empties the hash table
77 def clear(self):
78     self.buckets = []
79     for i in range(self.capacity):
80         self.buckets.append(LinkedList())
81     self.size = 0
82
83 #update the given key, value in the hash table
84 def insert(self, key, value):
85     hash_num = self.hash_function(key)
86     index = hash_num % self.maxhash
87     bucket = self.buckets[index]
88     node = bucket.search(key) #check the bucket by
89         key
90     if node is not None: #already exist, then
91         update
92         node.value = value
93         return
94     else: #not exist, then add value
95         self.buckets[index].add_front(key, value)
96         self.size += 1
97
98 #remove and free with given key
99 def delete(self, key):
100     hash_num = self.hash_function(key)

```

```

97         index = hash_num % self.maxhash
98         bucket = self.buckets[index]
99         node = bucket.search(key) #check the bucket by
            key
100         if node is not None: #already exist , then
            delete
101             bucket.remove(key)
102             self.size -= 1
103
104     #search a key exists
105     def find(self , key):
106         hash_num = self.hash_function(key)
107         index = hash_num % self.maxhash
108         bucket = self.buckets[index]
109         node = bucket.search(key) #check the bucket by
            key
110         if node is not None:
111             return True
112         else:
113             return False
114
115     #return the value
116     def get(self , key):
117         hash_num = self.hash_function(key)
118         index = hash_num % self.maxhash
119         bucket = self.buckets[index]
120         node = bucket.search(key) #check the bucket by
            key
121         if node is not None:
122             return node.value
123         else:
124             return False
125
126     #get how many empty buckets in the table
127     def empty_buckets(self):
128         num = 0
129         for buckets in self.buckets:
130             if buckets.head is None:

```

```

131         num += 1
132     return num
133
134     def print_hash(self):
135         for i in self.buckets:
136             if i.size != 0:
137                 i.iter()
138
139 def test_file(file):
140     hash_map = HashMap(MAXHASH, hashFunction)
141     rgx = re.compile("(\w[\w']* \w \w)")
142     list_all_keys = set()
143     with open(file) as f:
144         for line in f:
145             words = rgx.findall(line)
146             for word in words:
147                 word = word.lower() #covert to
148                                     lowercase
149                 list_all_keys.add(word)
150                 word_count = hash_map.get(word)
151                 if word_count is None:
152                     hash_map.insert(word, 1)
153                 else:
154                     hash_map.insert(word, word_count+1)
155     f.close()
156
157     with open('keys.txt', 'w') as f:
158         for word in list_all_keys:
159             value = hash_map.get(word)
160             f.write(word)
161             f.write(":")
162             f.write(str(value))
163             f.write('\n')
164     f.close()
165
166 def test():
167     hash_map = HashMap(MAXHASH, hashFunction)
168     rgx = re.compile("(\w[\w']* \w \w)")

```

```

168 line = 'Alice was beginning to get very tired of
        sitting by her sister on the bank, and of having
        nothing to do. Once or twice she had peeped
        into the book her sister was reading, but it had
        no pictures or conversations in it, "and what
        is the use of a book," thought Alice, "without
        pictures or conversations?" '
169 words = words = rgx.findall(line)
170 #insert key
171 for word in words:
172     word = word.lower() #covert to lowercase
173     #list_all_keys.add(word)
174     word_count = hash_map.get(word)
175     if word_count is None:
176         hash_map.insert(word, 1)
177     else:
178         hash_map.insert(word, word_count+1)
179 hash_map.print_hash()
180
181 #delete key
182 hash_map.delete('conversations')
183 print("After delete:")
184 hash_map.print_hash()
185
186 #find key
187 result = hash_map.find('on')
188 print(result)
189
190 if __name__ == '__main__':
191     test_file('alice_in_wonderland.txt')
192     #test()

```

2 Implement a red-black tree

```
1 class RBTree_node:
2     def __init__(self, x):
3         self.key = x
4         self.left = None
5         self.right = None
6         self.parent = None
7         self.color = 'black'
8
9 class RBTree:
10     def __init__(self):
11         self.nil = RBTree_node(0)
12         self.root = self.nil
13
14 #class Function:
15     def inorder_tree_walk(self, x):
16         if x != None:
17             self.inorder_tree_walk(x.left)
18             if x.key != 0:
19                 print('key:', x.key, 'parent:', x.
20                     parent.key, 'color:', x.color)
21             self.inorder_tree_walk(x.right)
22
23     def left_rotate(self, T, x):
24         y = x.right #set y
25         x.right = y.left #turn y's left subtree
26             into x's right subtree
27         if y.left != T.nil:
28             y.left.parent = x
29         y.parent = x.parent #link x's parent to y
30         if x.parent == T.nil:
31             T.root = y
32         elif x == x.parent.left:
33             x.parent.left = y
34         else:
35             x.parent.right = y
```

```

34         y.left = x                #put x on y's left
35         x.parent = y
36
37     def right_rotate(self, T, x):
38         y = x.left
39         x.left = y.right
40         if y.right != T.nil:
41             y.right.parent = x
42         y.parent = x.parent
43         if x.parent == T.nil:
44             T.root = y
45         elif x == x.parent.right:
46             x.parent.right = y
47         else:
48             x.parent.left = y
49         y.right = x
50         x.parent = y
51
52     def RBInsert(self, T, z):
53         z.left = z.right = z.parent = T.nil
54
55         y = T.nil
56         x = T.root
57         while x != T.nil:
58             y = x
59             if z.key < x.key:
60                 x = x.left
61             else:
62                 x = x.right
63         z.parent = y
64         if y == T.nil:
65             T.root = z
66         elif z.key < y.key:
67             y.left = z
68         else:
69             y.right = z
70         z.left = T.nil
71         z.right = T.nil

```



```

72         z.color = 'red'
73         self.RBInsert_fixup(T, z)
74
75     def RBInsert_fixup(self, T, z):
76         while z.parent.color == 'red':
77             if z.parent == z.parent.parent.left:
78                 y = z.parent.parent.right
79                 if y.color == 'red':
80                     z.parent.color = 'black'
81                                     #case 1
82                     y.color = 'black'
83                                     #case 1
84                     z.parent.parent.color = 'red'
85                                     #case 1
86                     z = z.parent.parent
87                                     #case 1
88             else:
89                 if z == z.parent.right:
90                     z = z.parent
91                                     #case
92                                     2
93                     self.left_rotate(T, z)
94                                     #case 2
95                     z.parent.color = 'black'
96                                     #case 3
97                     z.parent.parent.color = 'red'
98                                     #case 3
99                     self.right_rotate(T, z.parent.
100                                     parent) #case 3
101         else: #same as then clause with 'right' and
102             'left' exchange
103             y = z.parent.parent.left
104             if y.color == 'red':
105                 z.parent.color = 'black'
106                 y.color = 'black'
107                 z.parent.parent.color = 'red'
108                 z = z.parent.parent
109             else:

```

```

99         if z == z.parent.left:
100             z = z.parent
101             self.right_rotate(T, z)
102             z.parent.color = 'black'
103             z.parent.parent.color = 'red'
104             self.left_rotate(T, z.parent.parent
105                             )
106         T.root.color = 'black'
107     def transplant(self, T, u, v):
108         if u.parent == T.nil:
109             T.root = v
110         elif u == u.parent.left:
111             u.parent.left = v
112         else:
113             u.parent.right = v
114         v.parent = u.parent
115
116     def RBDelete_fixup(self, T, x):
117         while x != T.root and x.color == 'black':
118             if x == x.parent.left:
119                 w = x.parent.right
120                 if w.color == 'red':
121                     w.color = 'black'
122                                     #case 1
123                     x.parent.color = 'red'
124                                     #case 1
125                     self.left_rotate(T, x.parent)
126                                     #case 1
127                     w = x.parent.right
128                                     #case 1
129                 if w.left.color == 'black' and w.right.
130                     color == 'black':
131                     w.color = 'red'
132                                     #case 2
133                     x = x.parent
134                                     #case 2
135             else:

```

```

129         if w.right.color == 'black':
130             w.left.color == 'black'
131                 #case 3
132             w.color = 'red'
133                 #case 3
134             self.right_rotate(T, x)
135                 #case 3
136             w.color = x.parent.color
137                 #case 4
138             x.parent.color = 'black'
139                 #case 4
140             w.right.color = 'black'
141                 #case 4
142             self.left_rotate(T, x.parent)
143                 #case 4
144             x = T.root
145                 #case 4
146     else: #same as then clause with 'right'
147         and 'left' exchanged
148         w = x.parent.left
149         if w.color == 'red':
150             w.color = 'black'
151             x.parent.color = 'red'
152             self.right_rotate(T, x.parent)
153             w = x.parent.left
154         if w.right.color == 'black' and w.left.
155             color == 'black':
156             w.color = 'red'
157             x = x.parent
158         else:
159             if w.left.color == 'black':
160                 w.right.color = 'black'
161                 w.color = 'red'
162                 self.left_rotate(T, w)
163                 w = x.parent.left
164             w.color = x.parent.color
165             x.parent.color = 'black'
166             w.left.color = 'black'

```

```

157         self.right_rotate(T, x.parent)
158         x = T.root
159         x.color = 'black'
160
161     def RBDelete(self, T, z):
162         y = z
163         y_original_color = y.color
164         if z.left == T.nil:
165             x = z.right
166             self.transplant(T, z, z.right)
167         elif z.right == T.nil:
168             x = z.left
169             self.transplant(T, z, z.left)
170         else:
171             y = self.tree_minimum(z.right)
172             y_original_color = y.color
173             x = y.right
174             if y.parent == z:
175                 x.parent = y
176             else:
177                 self.transplant(T, y, y.right)
178                 y.right = z.right
179                 y.right.parent = y
180                 self.transplant(T, z, y)
181                 y.left = z.left
182                 y.left.parent = y
183                 y.color = z.color
184             if y_original_color == 'black':
185                 self.RBDelete_fixup(T, x)
186
187     def tree_minimum(self, x):
188         while x.left != self.nil:
189             x = x.left
190         return x
191
192     def tree_maximum(self, x):
193         while x.right != self.nil:
194             x = x.right

```

```

195         return x
196
197     def tree_successor(self, x):
198         if x.right != self.nil:
199             return self.tree_minimum(x.right)
200         y = x.parent
201         while y != self.nil and x == y.right:
202             x = y
203             y = y.parent
204         return y
205
206     def tree_predecessor(self, x):
207         if x.left != self.nil:
208             return self.tree_maximum(x.left)
209         y = x.parent
210         while y != self.nil and x == y.left:
211             x = y
212             y = y.parent
213         return y
214
215     def iterative_tree_search(self, x, k):
216         while x != self.nil:
217             if k == x.key:
218                 return x
219             elif k < x.key:
220                 x = x.left
221             else:
222                 x = x.right
223         return None
224
225     def tree_depth(self, T) -> int:
226         if T is None:
227             return 0
228         return max(self.tree_depth(T.left), self.
229                     tree_depth(T.right))+1
230
231     def test():
232         nodes = [11,2,14,1,7,15,5,8,4]

```

```

232 T = RBTree()
233 #x = Function()
234 for node in nodes:
235     T.RBInsert(T, RBTree_node(node))
236
237 T.inorder_tree_walk(T.root)
238 h = T.tree_depth(T.root)
239 print("The_height_of_RB_tree_is:", h-1)
240
241 #search node's key if it in the RB tree
242 value = 7
243 y = T.iterative_tree_search(T.root, value)
244 if y != None:
245     print(y.key, "is_in_the_tree")
246 else:
247     print(value, "is_not_in_the_tree")
248 #find node's predecessor
249 if y != None:
250     temp = T.tree_predecessor(y)
251     print("The_predecessor_of", y.key, "is", temp.
           key)
252     temp = T.tree_successor(y)
253     print("The_successor_of", y.key, "is", temp.key
           )
254 #insert new key to RB tree
255 T.RBInsert(T, RBTree_node(10))
256 print("After_insert_node:")
257 T.inorder_tree_walk(T.root)
258 #delete node
259 delete = 14
260 T.RBDelete(T, T.iterative_tree_search(T.root,
           delete))
261 print("After_delete_node:")
262 T.inorder_tree_walk(T.root)
263
264 if __name__ == '__main__':
265     test()

```

3 Implement the skiplist data structure

```
1 import random
2 #random number has biggest limitation
3 maxLevel = 16
4 #random level, select random number between 1 to
   maxRand
5 randLevel = random.randint(1, maxLevel)
6
7 class SkipNode:
8     def __init__(self, val):
9         self.value = val
10        self.right = None
11        self.down = None
12
13 class SkipList:
14     def __init__(self):
15         #initialize header and sentinel to infinity
16         header = [SkipNode(-float('inf')) for i in
17                    range(maxLevel)]
18         sentinel = [SkipNode(float('inf')) for i in
19                     range(maxLevel)]
20
21         #connect them together
22         for i in range(maxLevel - 1):
23             header[i].right = sentinel[i]
24             header[i].down = header[i+1]
25             sentinel[i].down = sentinel[i+1]
26         #the last layer of header don't have "down"
27         option
28         header[-1].right = sentinel[-1]
29         #skiplist initial pointer is header's first
30         element
31         self.head = header[0]
32
33         #search begin with initial pointer
34         def search(self, target:int) -> bool:
```

```

31         node = self.head
32     while node:
33         if node.right.value > target:
34             node = node.down
35         elif node.right.value < target:
36             node = node.right
37         else:
38             return True
39     return False
40
41     def add(self, num:int) -> None:
42         #use prev array to store skipilist pointer
43         before jump down
44         prev = []
45         node = self.head
46         while node:
47             if node.right.value >= num:
48                 prev.append(node)
49                 node = node.down
50             else:
51                 node = node.right
52
53         #arr is the array of pointer to be inserted,
54         randomly in length
55         arr = [SkipNode(num) for i in range(randLevel)]
56         temp = SkipNode(None)
57         for i,j in zip(prev[maxLevel - len(arr):], arr):
58             :
59             j.right = i.right
60             i.right = j
61             temp.down = j
62             temp = j
63
64     def earse(self, num:int) -> bool:
65         ans = False
66         node = self.head
67         while node:
68             if node.right.value > num:

```



```

66         node = node.down
67     elif node.right.value < num:
68         node = node.right
69     else:
70         ans = True
71         node.right = node.right.right
72         node = node.down
73     return ans
74
75 def test():
76     sl = SkipList()
77     sl.add(20)
78     sl.add(40)
79     sl.add(10)
80     sl.add(20)
81     sl.add(5)
82     sl.add(80)
83     sl.erase(20)
84     sl.add(100)
85     sl.add(20)
86     sl.add(30)
87     sl.erase(5)
88     sl.add(50)
89     print(sl.search(80))
90     sl.erase(10)
91     print(sl.search(10))
92
93 if __name__ == '__main__':
94     test()

```