# cs 5800 - hw8

Yijing Xiao

November 2021

## 1 Implement a hash for text

```
import re
MAXHASH = 2500

class Node:
    def __init__(self, k, val):
        self.next = None
        self.key = k
        self.value = val

class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def add_front(self, key, value):
        newNode = Node(key, value)
        newNode.next = self.head
        self.head = newNode
        self.size += 1

    def remove(self, key):
        if self.head is None:
            return False
        if self.head.key == key:
```

```
            self.head = self.head.next
            self.size = self.size-1
            return True
        cur = self.head.next
        prev = self.head
        while cur is not None:
            if cur.key == key:
                prev.next = cur.next
                self.size = self.size-1
                return True
            prev = cur
            cur = cur.next
        return False

    def search(self, key):
        if self.head is not None:
            cur = self.head
            while cur is not None:
                if cur.key == key:
                    return cur
                cur = cur.next
        return None

    def iter(self):
        if not self.head:
            return
        cur = self.head
        print(cur.key, ":", cur.value)
        while cur.next:
            cur = cur.next
            print(cur.key, ":", cur.value)

def hashFunction(key):
    hash_num = 0
    for i in key:
        hash_num += ord(i)
    return hash_num
```

```
class HashMap:
    def __init__(self, capacity, function):
        self.buckets = []
        for i in range(capacity):
            self.buckets.append(LinkedList())
        #capacity = the total number of buckets to be crrated in the hash table
        self.maxhash = capacity
        #function = the hash function to use for hashing
        self.hash_function = function
        self.size = 0

    #empties the hash table
    def clear(self):
        self.buckets = []
        for i in range(self.capacity):
            self.buckets.append(LinkedList())
        self.size = 0

    #update the given key, value in the hash table
    def insert(self, key, value):
        hash_num = self.hash_function(key)
        index = hash_num % self.maxhash
        bucket = self.buckets[index]
        node = bucket.search(key) #check the bucket by key
        if node is not None: #already exist, then update
            node.value = value
            return
        else: #not exist, then add value
            self.buckets[index].add_front(key, value)
            self.size += 1

    #remove and free with given key
    def delete(self, key):
        hash_num = self.hash_function(key)
```

```
        index = hash_num % self.maxhash
        bucket = self.buckets[index]
        node = bucket.search(key) #check the bucket by key
        if node is not None: #already exist, then delete
            bucket.remove(key)
            self.size -= 1

    #search a key exists
    def find(self, key):
        hash_num = self.hash_function(key)
        index = hash_num % self.maxhash
        bucket = self.buckets[index]
        node = bucket.search(key) #check the bucket by key
        if node is not None:
            return True
        else:
            return False

    #return the value
    def get(self, key):
        hash_num = self.hash_function(key)
        index = hash_num % self.maxhash
        bucket = self.buckets[index]
        node = bucket.search(key) #check the bucket by key
        if node is not None:
            return node.value
        else:
            return False

    #get how many empty buckets in the table
    def empty_buckets(self):
        num = 0
        for buckets in self.buckets:
            if buckets.head is None:
```

```python
                num += 1
        return num

    def print_hash(self):
        for i in self.buckets:
            if i.size != 0:
                i.iter()

def test_file(file):
    hash_map = HashMap(MAXHASH, hashFunction)
    rgx = re.compile("(\w[\w']*\w|\w)")
    list_all_keys = set()
    with open(file) as f:
        for line in f:
            words = rgx.findall(line)
            for word in words:
                word = word.lower() #covert to lowercase
                list_all_keys.add(word)
                word_count = hash_map.get(word)
                if word_count is None:
                    hash_map.insert(word, 1)
                else:
                    hash_map.insert(word, word_count+1)
    f.close()

    with open('keys.txt', 'w') as f:
        for word in list_all_keys:
            value = hash_map.get(word)
            f.write(word)
            f.write(":")
            f.write(str(value))
            f.write('\n')
    f.close()

def test():
    hash_map = HashMap(MAXHASH, hashFunction)
    rgx = re.compile("(\w[\w']*\w|\w)")
```

```
    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?"'
    words = words = rgx.findall(line)
    #insert key
    for word in words:
        word = word.lower() #covert to lowercase
        #list_all_keys.add(word)
        word_count = hash_map.get(word)
        if word_count is None:
            hash_map.insert(word, 1)
        else:
            hash_map.insert(word, word_count+1)
    hash_map.print_hash()

    #delete key
    hash_map.delete('conversations')
    print("After delete:")
    hash_map.print_hash()

    #find key
    result = hash_map.find('on')
    print(result)

if __name__ == '__main__':
    test_file('alice_in_wonderland.txt')
    #test()
```

# 2 Implement a red-black tree

```python
class RBTree_node:
    def __init__(self, x):
        self.key = x
        self.left = None
        self.right = None
        self.parent = None
        self.color = 'black'

class RBTree:
    def __init__(self):
        self.nil = RBTree_node(0)
        self.root = self.nil

#class Function:
    def inorder_tree_walk(self, x):
        if x != None:
            self.inorder_tree_walk(x.left)
            if x.key != 0:
                print('key: ', x.key, 'parent: ', x.parent.key, 'color: ', x.color)
            self.inorder_tree_walk(x.right)

    def left_rotate(self, T, x):
        y = x.right           #set y
        x.right = y.left      #turn y's left subtree into x's right subtree
        if y.left != T.nil:
            y.left.parent = x
        y.parent = x.parent   #link x's parent to y
        if x.parent == T.nil:
            T.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
```

```python
        y.left = x              #put x on y's left
        x.parent = y

    def right_rotate(self, T, x):
        y = x.left
        x.left = y.right
        if y.right != T.nil:
            y.right.parent = x
        y.parent = x.parent
        if x.parent == T.nil:
            T.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def RBInsert(self, T, z):
        z.left = z.right = z.parent = T.nil

        y = T.nil
        x = T.root
        while x != T.nil:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y == T.nil:
            T.root = z
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z
        z.left = T.nil
        z.right = T.nil
```

```
        z.color = 'red'
        self.RBInsert_fixup(T, z)

    def RBInsert_fixup(self, T, z):
        while z.parent.color == 'red':
            if z.parent == z.parent.parent.left:
                y = z.parent.parent.right
                if y.color == 'red':
                    z.parent.color = 'black'                 #case 1
                    y.color = 'black'                        #case 1
                    z.parent.parent.color = 'red'            #case 1
                    z = z.parent.parent                      #case 1
                else:
                    if z == z.parent.right:
                        z = z.parent                         #case 2
                        self.left_rotate(T, z)               #case 2
                    z.parent.color = 'black'                 #case 3
                    z.parent.parent.color = 'red'            #case 3
                    self.right_rotate(T, z.parent.parent)    #case 3
            else: #same as then clause with 'right' and 'left' exchange
                y = z.parent.parent.left
                if y.color == 'red':
                    z.parent.color = 'black'
                    y.color = 'black'
                    z.parent.parent.color = 'red'
                    z = z.parent.parent
                else:
```

```
                    if z == z.parent.left:
                        z = z.parent
                        self.right_rotate(T, z)
                    z.parent.color = 'black'
                    z.parent.parent.color = 'red'
                    self.left_rotate(T, z.parent.parent)
        T.root.color = 'black'

    def transplant(self, T, u, v):
        if u.parent == T.nil:
            T.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        v.parent = u.parent

    def RBDelete_fixup(self, T, x):
        while x != T.root and x.color == 'black':
            if x == x.parent.left:
                w = x.parent.right
                if w.color == 'red':
                    w.color = 'black'                                    #case 1
                    x.parent.color = 'red'                               #case 1
                    self.left_rotate(T, x.parent)                        #case 1
                    w = x.parent.right                                   #case 1
                if w.left.color == 'black' and w.right.color == 'black':
                    w.color == 'red'                                     #case 2
                    x = x.parent                                         #case 2
                else:
```

```
                if w.right.color == 'black':
                    w.left.color == 'black'  #case 3
                    w.color = 'red'  #case 3
                    self.right_rotate(T, x)  #case 3
                w.color = x.parent.color  #case 4
                x.parent.color = 'black'  #case 4
                w.right.color = 'black'  #case 4
                self.left_rotate(T, x.parent)  #case 4
                x = T.root  #case 4
        else:  #same as then clause with 'right' and 'left' exchanged
            w = x.parent.left
            if w.color == 'red':
                w.color = 'black'
                x.parent.color = 'red'
                self.right_rotate(T, x.parent)
                w = x.parent.left
            if w.right.color == 'black' and w.left.color == 'black':
                w.color = 'red'
                x = x.parent
            else:
                if w.left.color == 'black':
                    w.right.color = 'black'
                    w.color = 'red'
                    self.left_rotate(T, w)
                    w = x.parent.left
                w.color = x.parent.color
                x.parent.color = 'black'
                w.left.color = 'black'
```

```
                    self.right_rotate(T, x.parent)
                    x = T.root
        x.color = 'black'

    def RBDelete(self, T, z):
        y = z
        y_original_color = y.color
        if z.left == T.nil:
            x = z.right
            self.transplant(T, z, z.right)
        elif z.right == T.nil:
            x = z.left
            self.transplant(T, z, z.left)
        else:
            y = self.tree_minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.transplant(T, y, y.right)
                y.right = z.right
                y.right.parent = y
            self.transplant(T, z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color
        if y_original_color == 'black':
            self.RBDelete_fixup(T, x)

    def tree_minimum(self, x):
        while x.left != self.nil:
            x = x.left
        return x

    def tree_maximum(self, x):
        while x.right != self.nil:
            x = x.right
```

```
        return x

    def tree_successor(self, x):
        if x.right != self.nil:
            return self.tree_minimum(x.right)
        y = x.parent
        while y != self.nil and x == y.right:
            x = y
            y = y.parent
        return y

    def tree_predecessor(self, x):
        if x.left != self.nil:
            return self.tree_maximum(x.left)
        y = x.parent
        while y != self.nil and x == y.left:
            x = y
            y = y.parent
        return y

    def iterative_tree_search(self, x, k):
        while x != self.nil:
            if k == x.key:
                return x
            elif k < x.key:
                x = x.left
            else:
                x = x.right
        return None

    def tree_depth(self, T) -> int:
        if T is None:
            return 0
        return max(self.tree_depth(T.left), self.tree_depth(T.right))+1

def test():
    nodes = [11,2,14,1,7,15,5,8,4]
```

```
    T = RBTree()
    #x = Function()
    for node in nodes:
        T.RBInsert(T, RBTree_node(node))

    T.inorder_tree_walk(T.root)
    h = T.tree_depth(T.root)
    print("The height of RB tree is:", h-1)

    #search node's key if it in the RB tree
    value = 7
    y = T.iterative_tree_search(T.root, value)
    if y != None:
        print(y.key, "is in the tree")
    else:
        print(value, "is not in the tree")
    #find node's predecessor
    if y != None:
        temp = T.tree_predecessor(y)
        print("The predecessor of", y.key, "is", temp.
            key)
        temp = T.tree_successor(y)
        print("The successor of", y.key, "is", temp.key
            )
    #insert new key to RB tree
    T.RBInsert(T, RBTree_node(10))
    print("After insert node:")
    T.inorder_tree_walk(T.root)
    #delete node
    delete = 14
    T.RBDelete(T, T.iterative_tree_search(T.root,
        delete))
    print("After delete node:")
    T.inorder_tree_walk(T.root)

if __name__ == '__main__':
    test()
```

## 3 Implement the skiplist data structure

```
import random
#random number has biggest limitation
maxLevel = 16
#random level, select random number between 1 to maxRand
randLevel = random.randint(1, maxLevel)

class SkipNode:
    def __init__(self, val):
        self.value = val
        self.right = None
        self.down = None

class SkipList:
    def __init__(self):
        #initialize header and sentinel to infinity
        header = [SkipNode(-float('inf')) for i in range(maxLevel)]
        sentinel = [SkipNode(float('inf')) for i in range(maxLevel)]

        #connect them together
        for i in range(maxLevel - 1):
            header[i].right = sentinel[i]
            header[i].down = header[i+1]
            sentinel[i].down = sentinel[i+1]
        #the last layer of header don't have "down" option
        header[-1].right = sentinel[-1]
        #skiplist initial pointer is header's first element
        self.head = header[0]

    #search begin with initial pointer
    def search(self, target:int) -> bool:
```

```
        node = self.head
        while node:
            if node.right.value > target:
                node = node.down
            elif node.right.value < target:
                node = node.right
            else:
                return True
        return False

    def add(self, num:int) -> None:
        #use prev array to store skpilist pointer before jump down
        prev = []
        node = self.head
        while node:
            if node.right.value >= num:
                prev.append(node)
                node = node.down
            else:
                node = node.right

        #arr is the array of pointer to be inserted, randomly in length
        arr = [SkipNode(num) for i in range(randLevel)]
        temp = SkipNode(None)
        for i,j in zip(prev[maxLevel - len(arr):], arr):
            j.right = i.right
            i.right = j
            temp.down = j
            temp = j

    def earse(self, num:int) -> bool:
        ans = False
        node = self.head
        while node:
            if node.right.value > num:
```

```python
                node = node.down
            elif node.right.value < num:
                node = node.right
            else:
                ans = True
                node.right = node.right.right
                node = node.down
        return ans

def test():
    sl = SkipList()
    sl.add(20)
    sl.add(40)
    sl.add(10)
    sl.add(20)
    sl.add(5)
    sl.add(80)
    sl.earse(20)
    sl.add(100)
    sl.add(20)
    sl.add(30)
    sl.earse(5)
    sl.add(50)
    print(sl.search(80))
    sl.earse(10)
    print(sl.search(10))

if __name__ == '__main__':
    test()
```