CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 7 and 8
Submit via Gradescope

Name: Ketaki Nitin Kolhatkar
Collaborators: Ashwini Kumar, Pooja Ramakrishnan

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(50 points)***

Implement a hash for text. Given a string as input, construct a hash with words as keys, and word counts as values. Your implementation should include:

- a hash function that has good properties for text
- storage and collision management using linked lists
- operations: insert(key,value), delete(key), increase(key), find(key), list-all-keys

Output the list of words together with their counts on an output file. For this problem,you cannot use built-in-language datastuctures that can index by strings (like hashtables). Use a language that easily implements linked lists, like C/C++.
You can test your code on "Alice in Wonderland" by Lewis Carroll, at link
The test file used by TA will probably be shorter.

(Extra Credit) Find a way to record not only word counts, but also the positions in text. For each word, besides the count value, build a linked list with positions in the given text. Output this list together with the count.

```
    import re
CAP_IN = 50

class HashTable():
        def __init__(self):
            self.capacity = CAP_IN
            self.current_length = 0
            self.buckets = []
            for _ in range(self.capacity):
                self.buckets.append(None)

        def hash(self, key):
            key_length = len(key)
            hash_function_val = sum([(a + key_length) ** ord(b) for a,b in enumerate(key)]) % s
            return hash_function_val

        def insert(self, key, value):
            self.current_length += 1
            index = self.hash(key)
            node = self.buckets[index]
            if node is None:
                self.buckets[index] = Node(key, value)
                return

            prev = node
            while node is not None:
```

```
            if node.key == key:
                node.value += value
                return
            else:
                prev = node
                node = node.next
        prev.next = Node(key, value)

    def list(self):
        for i in range(self.capacity):
            node = self.buckets[i]
            while node is not None:
                print(node.key, node.value)
                node = node.next

    def find(self, key):
        index = self.hash(key)
        node = self.buckets[index]
        while node is not None and node.key != key:
            node = node.next

        if node is None:
            return None
        else:
            return node.value

    def remove(self, key):
        index = self.hash(key)
        node = self.buckets[index]
        prev = None

        while node is not None and node.key != key:
            prev = node
            node = node.next

        if node is None:
            return None
        else:
            self.current_length -= 1
            result = node.value
            if prev is not None:
                prev.next = prev.next.next
            return result

class Node:
```

```
        def __init__(self, key, value):
            self.key = key
            self.value = value
            self.next = None
        def __str__(self):
            return "<Node: (%s, %s), next: %s>" % (self.key, self.value, self.next != None)
        def __repr__(self):
            return str(self)

ht = HashTable()

with open('C:/Users/kolhatkar.k/Downloads/aiw.txt', 'r') as file:
    # reading each line
    for line in file:
        # reading each word
        for word in line.split():
            # displaying the words
            ht.insert(re.sub('[^A-Za-z]', '', word).lower(),1)
ht.insert("hello",1)
ht.insert("check",1)
ht.insert("hello",1)
ht.list()
ht.list()
```

2. ***(50 points)***

Implement a red-black tree, including binary-search-tree operations *sort, search, min, max, successor, predecessor* and specific red-black procedures *rotation, insert, delete.* The delete implementation is Extra Credit (but highly recommended).

Your code should take the input array of numbers from a file and build a red-black tree with this input by sequence of "inserts". Then interactively ask the user for an operational command like "insert x" or "sort" or "search x" etc, on each of which your code rearranges the tree and if needed produces an output. After each operation also print out the height of the tree.

You can use any mechanism to implement the tree, for example with pointers and struct objects in C++, or with arrays of indices that represent links between parent and children. You cannot use any tree built-in structure in any language.

```
    import sys

class Node():
    def __init__(self, data):
        self.data = data
        self.parent = None
```

```
        self.left = None
        self.right = None
        self.color = 1

# class RedBlackTree implements the operations in Red Black Tree
class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL

    def search_tree_func(self, node, key):
        if node == self.TNULL or key == node.data:
            return node.data
        if key < node.data:
            return self.search_tree_func(node.left, key)
        return self.search_tree_func(node.right, key)

    # fix the red-black tree
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left  # uncle
                if u.color == 1:
                    # case 3.1
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        # case 3.2.2
                        k = k.parent
                        self.right_rotate(k)
                    # case 3.2.1
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right  # uncle

                if u.color == 1:
                    # mirror case 3.1
```

```
                u.color = 0
                k.parent.color = 0
                k.parent.parent.color = 1
                k = k.parent.parent
            else:
                if k == k.parent.right:
                    # mirror case 3.2.2
                    k = k.parent
                    self.left_rotate(k)
                # mirror case 3.2.1
                k.parent.color = 0
                k.parent.parent.color = 1
                self.right_rotate(k.parent.parent)
        if k == self.root:
            break
    self.root.color = 0

def print_tree(self,node, lines, level=0):
    if node.data != 0:
        self.print_tree(node.left, lines, level + 1)
        lines.append('-' * 4 * level + '> ' +
                     str(node.data) + ' ' + ('r' if node.color == 1 else 'b'))
        self.print_tree(node.right, lines, level + 1)


def searchTree(self, k):
    return self.search_tree_func(self.root, k)

def minimum(self, node):
    while node.left != self.TNULL:
        node = node.left
    return node.data

def maximum(self, node):
    while node.right != self.TNULL:
        node = node.right
    return node.data

def successor(self, x):
    if x.right != self.TNULL:
        return self.minimum(x.right)
    y = x.parent
    while y != self.TNULL and x == y.right:
        x = y
        y = y.parent
```

```
    return y

def predecessor(self, x):
    if (x.left != self.TNULL):
        return self.maximum(x.left)
    y = x.parent
    while y != self.TNULL and x == y.left:
        x = y
        y = y.parent
    return y

# rotate left at node x
def left_rotate(self, x):
    y = x.right
    x.right = y.left
    if y.left != self.TNULL:
        y.left.parent = x

    y.parent = x.parent
    if x.parent == None:
        self.root = y
    elif x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y
    y.left = x
    x.parent = y

# rotate right at node x
def right_rotate(self, x):
    y = x.left
    x.left = y.right
    if y.right != self.TNULL:
        y.right.parent = x

    y.parent = x.parent
    if x.parent == None:
        self.root = y
    elif x == x.parent.right:
        x.parent.right = y
    else:
        x.parent.left = y
    y.right = x
    x.parent = y
```

```
    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.data = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1
        y = None
        x = self.root
        while x != self.TNULL:
            y = x
            if node.data < x.data:
                x = x.left
            else:
                x = x.right

        # y is parent of x
        node.parent = y
        if y == None:
            self.root = node
        elif node.data < y.data:
            y.left = node
        else:
            y.right = node

        if node.parent == None:
            node.color = 0
            return

        # if the grandparent is None, simply return
        if node.parent.parent == None:
            return

        self.fix_insert(node)

    def get_root(self):
        return self.root

    def Treeprint(self):
        lines = []
        self.print_tree(self.root, lines)
        return '\n'.join(lines)

bst = RedBlackTree()
bst.insert(8)
```

```
bst.insert(18)
bst.insert(5)
bst.insert(15)
bst.insert(17)
bst.insert(25)
bst.insert(40)
bst.insert(80)
bst.insert(89)
bst.insert(453)

#prints the tree
print(bst.Treeprint())
print(bst.minimum(bst.root))
print(bst.maximum(bst.root))
print(bst.successor(bst.root))
```

3. ***(50 points)***

Implement Skiplists 50 points. Study the skiplist data structure and operations. They are used for sorting values, but in a datastructure more effficient than lists or arrays, and more guaranteed than binary search trees.

```
    from random import randint, seed


class SkipNode:
    def __init__(self, height=0, elem=None):
        self.elem = elem
        self.next = []
        for _ in range(height):
            self.next.append(None)

class SkipList:

    def __init__(self):
        self.head = SkipNode()
        self.length = 0
        self.maximumHeight = 0

    def __len__(self):
        return self.length

    def find(self, elem, update=None):
        if update == None:
            update = self.updateList(elem)
        if len(update) > 0:
```

```
            candidate = update[0].next[0]
            if candidate != None and candidate.elem == elem:
                return candidate
        return None

    def contains(self, elem, update=None):
        return self.find(elem, update) != None

    def randomHeight(self):
        height = 1
        while True:
            rand_int = randint(1,2)
            if rand_int == 1:
                break
            else:
                height += 1

        return height

    def updateList(self, elem):
        update = []
        for _ in range(self.maximumHeight):
            update.append(None)

        x = self.head
        for i in range(self.maximumHeight-1,-1,-1):
            while x.next[i] != None and x.next[i].elem < elem:
                x = x.next[i]
            update[i] = x
        return update

    def insert(self, elem):
        height = self.randomHeight()
        node = SkipNode(height, elem)

        self.maximumHeight = max(self.maximumHeight, len(node.next))
        while len(self.head.next) < len(node.next):
            self.head.next.append(None)

        update = self.updateList(elem)
        if self.find(elem, update) == None:
            for i in range(len(node.next)):
                node.next[i] = update[i].next[i]
                update[i].next[i] = node
            self.length += 1
```

```
        print(elem, height)

    def remove(self, elem):

        update = self.updateList(elem)
        x = self.find(elem, update)
        if x is not None:
            for i in range(len(x.next)-1,-1,-1):
                update[i].next[i] = x.next[i]
                if self.head.next[i] == None:
                    self.maximumHeight -= 1
            self.length -= 1

    def printList(self):
        for i in range(len(self.head.next) - 1, -1, -1):
            x = self.head
            while x.next[i] != None:
                print(x.next[i].elem, '--', end =" "),
                x = x.next[i]
            print('')


skp = SkipList()

skp.insert(10)
skp.insert(30)
skp.insert(41)
skp.insert(21)
skp.insert(66)
skp.printList()
skp.remove(30)
skp.printList()
if (skp.find(21)):
    print("element present")
else:
    print("Element not present")
print(skp.find(51))
```