CS5800: Algorithms — Virgil Pavlu

Homework 8

Name: Fengjen
Collaborators:

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.
Try these three values for $m = MAXHASH : 30, 300, 1000$. For each of these m values, produce a histogram over the lengths of collision lists. You can also calculate variance of these lengths.
If your hash is close to uniform in collisions, you should get variance close to zero, and almost all list-lengths around $\alpha = n/m$.
If your hash has long lists, we want to know how many and how long, for example print the lengths of the longest 10% of the lists. Variance of collision list lengths: 86.56 for m= 30
top 10% length of the lists for m=30: [97, 94, 92]
Variance of collision list lengths: 9.498933333333333 for m= 300
top 10% length of the lists for m=300: [21, 19, 18, 15, 15, 15, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 13, 13, 13, 13, 13, 13, 13, 13, 12, 12, 12, 12, 12, 12]
Variance of collision list lengths: 2.3395040000000003 for m= 1000
top 10% length of the lists for m=1000: [8, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4]

code:

(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.

Solution:

```
from LinkedList import Node, LinkedList
import re
import matplotlib.pyplot as plt
import numpy as np
class HashTable:
    def __init__(self, m):
        '''
```

```
    self.table is a list of linked list, each linked list stores keys that has the same has
    each key has a value that represent word count.
    :param m: max number of key can be stored
    '''
    self.m = m
    self.table = [LinkedList() for _ in range(m)]

def __hash(self, key):

    p = 31
    m = self.m
    hash_value = 0
    p_pow = 1
    for char in key:
        hash_value = (hash_value + (ord(char) - ord('a') + 1) * p_pow) % m
        p_pow = (p_pow * p) % m
    return hash_value

def insert(self, key, value=1): #default word count=1
    h = self.__hash(key)
    node = Node(key, value)
    if self.table[h].head is None:
      self.table[h].head = node

    else:
        current = self.table[h].head

        while current:
            if current.key == key:
                current.value += 1
                return
            else:
                current = current.next
        node.next = self.table[h].head # add new node in front of previous nodes
        self.table[h].head = node

def delete(self,key):
    h = self.__hash(key)
    if self.table[h].head is None:
        raise KeyError
    else:
        current = self.table[h].head
        prev = None
        while current:
            if current.key != key:
```

```
                prev = current
                current = current.next
            else:
                if prev:
                    prev.next = current.next #remove current
                else: #current is head
                    self.table[h].head = current.next
                return
        raise KeyError

def increase(self, key, value=1):
    h = self.__hash(key)
    current = self.table[h].head
    while current:
        if current.key != key:
            current = current.next
        else:
            current.value += value
            return
    raise KeyError

def find(self, key):
    h = self.__hash(key)
    if self.table[h].head is None:
        raise KeyError
    else:
        current = self.table[h].head
        while current:
            if current.key != key:
                current = current.next
            else:
                return f"count of {key}: {current.value}"
        raise KeyError

def list_all_keys(self):
    result = []
    for linked_list in self.table:
        if linked_list.head:
            current = linked_list.head
            while current:
                result.append(current.key)
                current = current.next
    return result

def plot_key_len(self):
```

```
        lengths = []
        for linked_list in self.table:
            current = linked_list.head
            count = 0
            while current:
                count += 1
                current = current.next
            lengths.append(count)

        # Plot histogram
        plt.hist(lengths, bins=range(max(lengths)+2), edgecolor='black', align='left')
        plt.xlabel('Length of Collision Lists')
        plt.ylabel('Frequency')
        plt.title('Histogram of Collision List Lengths')
        plt.show()

        # Calculate variance
        variance = np.var(lengths)
        print(f"Variance of collision list lengths: {variance} for m= {self.m}")
        top10 = int(len(lengths)/10)
        top10_len = sorted(lengths,reverse=True)[:top10]
        print(f"top 10% length of the lists for m={self.m}: {top10_len}")

def process_text(file_path, max_hash_size):
    # Initialize hashtable with given size
    hashtable = HashTable(max_hash_size)
    with open(file_path, "r", encoding="ISO-8859-1") as file:
        for line in file:
            words = re.findall(r'\b\w+\b', line)
            for word in words:
                hashtable.insert(word)
    hashtable.plot_key_len()
def main():

    for m in [30,300,1000]:
        process_text('alice_in_wonderland.txt',m)

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

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.

Solution:

```
import matplotlib.pyplot as plt
class Node:
    def __init__(self, key=None, color='B'):
        self.left = None
        self.right = None
        self.key = key
        self.color = color
        self.p = None #parent

class RedBlackTree:
    def __init__(self):
        self.NIL = Node() # sentinel node, black, no key
        self.root = self.NIL
        self.NIL.left = self.NIL.right =  self.NIL

    def sort(self,x):
        '''
        in order traverse
        '''
        if x != self.NIL:
            self.sort(x.left)
            print(x.key)
            self.sort(x.right)

    def search(self, key):
        '''
        recursively search a node with the given key starting from root
        :return: a node with given key
        '''
        return self._search(key, self.root)
    def _search(self, key, node):
        if node.key == key or node == self.NIL:
            return node
        # binary search
        elif node.key > key:
```

```
        return self._search(key, node.left)
    else:
        return self._search(key, node.right)

def min(self, x=None):
    '''
    Find min node of the subtree rooted at given node x
    '''
    if x is None:
        x = self.root
    while x.left != self.NIL:
        x = x.left
    return x

def max(self, x=None):
    if x is None:
        x = self.root

    while x.right != self.NIL:
        x = x.right
    return x

def successor(self, x):
    '''
    find the node that has next-greater value than x
    :param x: given key value
    :return: successor node
    '''
    node = self.search(x)
    if node.right != self.NIL:
        return self.min(node.right)
    else:
        y = node.p
        while y and node == y.right: #until node's ancestor is y.left or no y
            node = y
            y = y.p
        return y

def predecessor(self, x):
    '''
    find the node that has next-smaller value than x
    :param x: given key value
    :return: predecessor node
    '''
    node = self.search(x)
```

```
    if node.left != self.NIL:
        return self.max(node.left)
    else:
        y = node.p
        while y and node == y.left:
            node = y
            y = y.p
        return y

def _rotate_left(self, x):
    '''
    rotate the given node x in given tree

    case1: x == x.p.right
       \                        \
        x                        y
      /   \       ->           /
    w      y                  x
        /                    /    \
      z                    w      z

    case2: x== x.p.left
    '''
    y =  x.right
    x.right = y.left
    if y.left != self.NIL:
        y.left.p = x
    y.p = x.p
    if x.p == self.NIL: # if x was root
        self.root = y
    elif x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    y.left = x
    x.p = y

def _rotate_right(self, x):
    y = x.left
    x.left = y.right
    if y.right != self.NIL:
        y.right.p = x
    y.p = x.p
    if x.p == self.NIL:
        self.root = y
```

```
    elif x == x.p.right:
        x.p.right = y
    else:
        x.p.left = y
    y.right = x
    x.p = y


def insert(self, z):
    '''
    create a new node and link it to itsparent y. Then fit the structure with self._insert_
    :param z: key to be inserted
    '''
    x = self.root
    y = self.NIL
    new_node = Node(z)
    new_node.left = new_node.right = self.NIL

    while x != self.NIL:
        y = x
        if new_node.key < x.key:
            x = x.left
        else:
            x = x.right
    new_node.p = y
    if y == self.NIL:    # if tree is empty
        self.root = new_node
    elif new_node.key < y.key:
        y.left = new_node
    else:
        y.right = new_node
    new_node.color = "R"
    self._insert_fix(new_node)

def _insert_fix(self, z):
    '''
    fix the tree structure to meet RBTree's requirement after inserting a node
    :param z: new node to be inserted
    '''
    while z.p.color == "R": # initially z is red, there can't be 2 red in a row
        if z.p == z.p.p.left:
            y = z.p.p.right # y is z's uncle(z.p's sibling)
            if y.color == "R":
                z.p.color = "B"
                y.color = "B"
```

```
                z.p.p.color = "R"
                z = z.p.p # z and z.p layers fixed, check z.p.p and up
            else:
                if z==z.p.right:
                    z = z.p
                    self._rotate_left(z)
                z.p.color = "B"
                z.p.p.color = "R"
                self._rotate_right(z.p.p)
        else:
            y = z.p.p.left
            if y.color == "R":
                z.p.color = "B"
                y.color = "B"
                z.p.p.color = "R"
                z = z.p.p
            else:
                if z == z.p.left:
                    z = z.p
                    self._rotate_right( z)
                z.p.color = "B"
                z.p.p.color = "R"
                self._rotate_left( z.p.p)
    self.root.color = "B"

def delete(self):
    pass

def height(self, node=None):
    ''''Calculate the height of the tree.'''
    if node is None:
        node = self.root
    if node == self.NIL:
        return 0
    return 1 + max(self.height(node.left), self.height(node.right))

def plot_tree(self):
    if self.root == self.NIL:
        print("Tree is empty.")
        return

    fig, ax = plt.subplots(figsize=(10, 8))
    ax.set_aspect('equal')
    ax.axis('off')
```

```
        # Start recursive plotting from the root
        self._plot_tree_nodes(ax, self.root, 0, 0, 300, 50)
        plt.show()

    def _plot_tree_nodes(self, ax, node, x, y, dx, dy):
        if node != self.NIL:
            # Draw the current node
            color = 'red' if node.color == 'R' else 'black'
            ax.add_patch(plt.Circle((x, y), 15, color=color, ec='black', lw=2))
            ax.text(x, y, str(node.key), ha='center', va='center', color='white' if color == 'l
                    fontweight='bold')

            # Draw left child
            if node.left != self.NIL:
                ax.plot([x, x - dx], [y, y - dy], color='black', lw=1)  # edge
                self._plot_tree_nodes(ax, node.left, x - dx, y - dy, dx / 2, dy)

            # Draw right child
            if node.right != self.NIL:
                ax.plot([x, x + dx], [y, y - dy], color='black', lw=1)  # edge
                self._plot_tree_nodes(ax, node.right, x + dx, y - dy, dx / 2, dy)

def build_tree(file):
    tree = RedBlackTree()
    with open(file, 'r') as f:
        keys = f.read().split(',')
        for key in keys:
            tree.insert(int(key))
    return tree

def main():
    tree = build_tree('tree_input.txt')
    tree.plot_tree()
    print("Available commands: insert <key>, search <key>, successor <key>, predecessor <key>,
    while True:
        command = input("Enter command: ").strip().lower()

        if command.startswith("insert"):
            try:
                _, key = command.split()
                key = int(key)
                tree.insert(key)
                print(f"Inserted {key}")
            except ValueError:
                print("Invalid key. Please provide an integer.")
```

```
elif command.startswith("search"):
    try:
        _, key = command.split()
        key = int(key)
        result = tree.search(key)
        if result != tree.NIL:
            print(f"Found key {key}")
        else:
            print(f"Key {key} not found")
    except ValueError:
        print("Invalid key. Please provide an integer.")

elif command.startswith("min"):
    min = tree.min().key
    print(f"Minimum value in tree: {min}")

elif command.startswith("max"):
    max = tree.max().key
    print(f"Minimum value in tree: {max}")

elif command.startswith("successor"):
    try:
        _, key = command.split()
        key = int(key)
        successor = tree.successor(key).key
        print(f"successor of {key} is: {successor}")
    except ValueError:
        print("Invalid key. Please provide an integer.")

elif command.startswith("predecessor"):
    try:
        _, key = command.split()
        key = int(key)
        predecessor = tree.predecessor(key).key
        print(f"predecessor of {key} is: {predecessor}")
    except ValueError:
        print("Invalid key. Please provide an integer.")

elif command == "sort":
    print("Tree elements in sorted order:")
    tree.sort(tree.root)
    print()

elif command == "exit":
```

```
            print("Exiting...")
            break

        else:
            print("Unknown command. Please try again.")

        # Print the height of the tree after each operation
        print(f"Current tree height: {tree.height()}")
    tree.plot_tree()

def main2():
    tree = RedBlackTree()
    tree.insert(10)
    tree.insert(15)
    print(tree.successor(15).key)

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

3. (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. Review Slides skiplists.pdf and Visualizer. The demo will be a sequence of operations (asked by TA) such as for example insert 20, insert 40, insert 10, insert 20, insert 5, insert 80, delete 20, insert 100, insert 20, insert 30, delete 5, insert 50, lookup 80, etc

Solution:

```
import random

class SkipListNode:
    def __init__(self, key, level):
        '''
        node.next is a list storing pointers to the next node.
        If input level is 0, there will be 1 base layer(level 0) for storing all keys, therefo
        level is to mimic the randomized length of a number appearing in the skiplists
        :param key: key to store in this node
        :param level: number of a series of coin flip that is head, implemented by self.random_
        '''
        self.next = [None] * (level + 1)
        self.key = key

class SkipLists:
    def __init__(self):
        self.level = 0
```

```
    self.head = SkipListNode(float('-inf'), 0)

def random_level(self):
    level = 0
    while random.random() < 0.5:
        level += 1
    return level
def insert(self,key):
    '''insert a key into the skip list'''
    level = self.random_level()
    if level > self.level:
        self.head.next.extend([None] * (level - self.level))
        self.level = level
    new_node = SkipListNode(key, level)
    update = [None] * (self.level + 1)
    current = self.head

    for i in range(self.level, -1, -1): # start inserting from higher level to find the pos
        while current.next[i] and current.next[i].key < key:
            current = current.next[i]
        update[i] = current

    for i in range(level + 1): # insert new_node such that update->new_node->update.next fo
        new_node.next[i] = update[i].next[i]
        update[i].next[i] = new_node

def delete(self, key):
    current = self.head
    update = [None] * (self.level + 1)
    for i in range(self.level, -1, -1):
        while current.next[i] and current.next[i].key < key:
            current = current.next[i]
        update[i] = current
    current = current.next[0]
    if current and current.key == key:
        for i in range(len(current.next)):
            if update[i].next[i] == current:
                update[i].next[i] = current.next[i]
        while self.level > 0 and not self.head.next[self.level]:
            self.level -= 1

def search(self, key):
    trace = []
    current = self.head
    for i in range(self.level, 0, -1):
```

```
            while current.next[i] and current.next[i].key < key:
                current = current.next[i]
                trace.append(('go right', current.key))
            trace.append('go down')
        current = current.next[0]
        while current and current.key != key:
            trace.append(('go right',current.key))
            current = current.next[0]
        trace.append(('go right', current.key))
        print(trace)
        if current.key == key:
            return True
        else:
            return False

    def display(self):
        '''Display the skip list structure for each level.'''
        for i in range(self.level, -1, -1):
            current = self.head.next[i]
            level_nodes = []
            while current:
                level_nodes.append(current.key)
                current = current.next[i]
            print(f"Level {i}: {' -> '.join(map(str, level_nodes))}")

def main():
    sl = SkipLists()
    sl.insert(1)
    sl.insert(2)
    sl.insert(2)
    sl.insert(5)
    # sl.insert(10)
    # sl.display()
    # print()
    # sl.delete(2)
    # sl.display()
    # print()
    sl.insert(6)
    sl.insert(10)
    sl.display()
    print(sl.search(6))
if __name__ == '__main__':
    main()
```

4. (50 pts). Implement binomial heaps as described in class and in the book. You should use links (pointers) to implement the structure as shown in the figure 1. Your implementation should

include the operations: Make-heap, Insert, Minimum, Extract-Min, Union, Decrease-Key, Delete

Solution:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.child = None
        self.parent = None
        self.sibling = None
        self.degree = 0

class BinomialHeap:
    def __init__(self):
        '''
        self.roots is a linked list of roots of each heap
        heap1.root.sibling = heap2.root
        head2.root.sibling = heap3.root and so on
        '''
        self.roots = None

    def make_heap(self):
        return BinomialHeap()

    def insert(self, key):
        '''
        insert a node into self by a given key
        1. create a node
        2. create a BinomialHeap
        3. set new BinomialHeap's roots as new node
        4. union self with the new BinomialHeap
        '''
        new_node = Node(key)
        new_heap = BinomialHeap()
        new_heap.roots = new_node
        self.union(new_heap)

    def union(self, other_heap):
        '''
        union self with another BinomialHeap
        '''
        # Merge the root lists of the two heaps
        self.roots = self._link_roots(other_heap)

        if not self.roots:
            return
```

```
        prev = None
        current = self.roots
        next_root = current.sibling

        # Consolidate trees of the same degree
        while next_root:
            if current.degree != next_root.degree :  # move ahead if the degrees are different
                prev = current
                current = next_root
            else:   # if degrees are the same, merge the heaps as a bigger(higher degree) heap
                if current.key <= next_root.key:
                    # link 'next_root' under 'current' and keep 'current' as the root
                    current.sibling = next_root.sibling
                    self._merge(current, next_root)
                else: # next_root.key is smaller, so it will be the new root
                    if prev:
                        prev.sibling = next_root
                    else:
                        self.roots = next_root
                    self._merge(next_root, current) # next_root is current's parent
                    current = next_root
            next_root = current.sibling

    def _merge(self, root1, root2):
        '''
        combine 2 heaps of the same degree by making root2 a child of root1
        :param root1:  root with smaller key
        :param root2:  root with larger key
        '''
        root2.parent = root1
        root2.sibling = root1.child
        root1.child = root2
        root1.degree += 1

    def _link_roots(self, other):
        '''
        link the root lists of self and other in increasing order of degrees.
        '''
        new_roots = last = None
        r1 = self.roots
        r2 = other.roots

        while r1 or r2:
            # choose the lower degree root
```

```
        if (r1 and not r2) or (r1 and r1.degree <= r2.degree):
            chosen_root = r1
            r1 = r1.sibling
        else:
            chosen_root= r2
            r2 = r2.sibling
        # link the chosen node to the end of the linked list
        if not last:
            new_roots = chosen_root
        else:
            last.sibling = chosen_root
        last = chosen_root  #update the last node to chosen_root

    return new_roots

def minimum(self):
    '''
    Find the minimun node of the heaps, minimum key is at one of the roots
    Return: node which has min key
    '''
    if not self.roots:
        return None
    min_node = self.roots
    current = self.roots
    while current:
        if current.key < min_node.key:
            min_node = current
        current = current.sibling # next root
    return min_node

def extract_min(self):
    '''
    remove and return the minimum key
    '''
    min_node = self.minimum()
    if not min_node:
        return None

    if self.roots == min_node:
        self.roots = min_node.sibling
    else:
        current = self.roots
        while current.sibling != min_node:
            current = current.sibling
        current.sibling = min_node.sibling
```

```
        #create a new heap from the children of min_node
        child = min_node.child
        new_roots = None
        while child:
            new_child = child.sibling
            child.sibling = new_roots
            child.parent = None #previously min_node
            new_roots = child
            child = new_child

        new_heap = self.make_heap()
        new_heap.roots = new_roots
        self.union(new_heap) #union fixed heap(extracted min_node) back to heaps

        return min_node.key

    def find_node(self, root, key):
        if root is None:
            return None
        if root.key == key:
            return root
        # recursively search for the key in the child and sibling nodes
        child_result = self.find_node(root.child, key)
        if child_result:
            return child_result
        return self.find_node(root.sibling, key)

    def decrease_key(self, key, new_key):
        '''
        decrease the key of a given node to a new value
        '''
        node_to_decrease = self.find_node(self.roots, key)
        if not node_to_decrease:
            print(f"{key} not found")
            return

        if new_key >node_to_decrease.key:
            raise ValueError("New key is should be smaller than the current key.")

        node_to_decrease.key = new_key
        current = node_to_decrease
        parent = current.parent
        #flow up the node with smaller value
        while parent and current.key < parent.key:
```

```
            current.key, parent.key = parent.key, current.key
            current = parent
            parent = current.parent

    def delete(self, key):
        node_to_delete = self.find_node(self.roots, key)
        if not node_to_delete:
            print(f"{key} not in the heaps")
            return
        self.decrease_key(key, float('-inf'))
        self.extract_min()

    def print_heap(self, node, degree=0):
        '''Prints the binomial tree starting from the given node.'''
        if node is None:
            return
        print(" " * degree * 4 + f" {node.key}")
        # Traverse the child nodes
        child = node.child
        while child:
            self.print_heap(child, degree + 1)
            child = child.sibling

    def print_binomial_heaps(self):
        '''Prints each binomial tree in the heap starting from each root node.'''
        root = self.roots
        tree_number = 0
        while root:
            print(f"Binomial Tree {tree_number}:")
            self.print_heap(root)
            print()  # Newline between binomial trees
            root = root.sibling
            tree_number += 1

def main():
    keys = [1,2,5,32,6,4,7,11,21]
    binomial_heap = BinomialHeap()
    for k in keys:
        binomial_heap.insert(k)

    binomial_heap.print_binomial_heaps()

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

5.Exercise 16.3-3. Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in O(lg n) worst-case time. Give a potential function ϕ such that the amortized cost of INSERT is O(lg n) and the amortized cost of EXTRACT-MIN is O(1), and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

Let the state of the heap after i-th operation as T_i, c_i is the actual cost of the i-th operation.
Potential function: $\Phi(T_i)$
we set $\Phi(T_0) = 0$ and $\Phi(T_i) >= 0$ amortize cost:$\hat{c_i} = c_i + \Phi(T_i) - \Phi(T_{i-1})$
define the potential function for a min-heap with n elements as: $\Phi(n) = c \cdot n \log n$
actual cost of an insert operation is $c \cdot \log n$
change of potential: $\Phi(n+1) - \Phi(n) \approx c \cdot \log n$
amortize cost of insert= $c \cdot \log n + c \cdot \log n = O(logn)$

actual cost of EXTRACT-MIN: $c \cdot \log n$
change of potential: $\Phi(n) - \Phi(n-1) \approx -c \cdot \log n$
amortize cost of EXTRACT-MIN = $c \cdot \log n - c \cdot \log n = O(1)$