CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 7 and 8
Submit via Gradescope

Name: Zerun Tian

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. ***(30 points, Mandatory)*** *Write up a max one-page summary of all concepts and techniques in CLRS Chapter 10 (Simple Data Structures)*

Solution:

I. Stacks and Queues

A stack is a last-in, first-out (LIFO) data structure. A stack of size n can be used to store at most n elements. It could be easily and efficiently implemented using an array. Both push and pop operations could be done in constant time.

On the other hand, a queue is a first-in, first-out (FIFO) data structure. A queue of size n can be used to store at most $n - 1$ elements using the book's implementation. It could be implemented using an array with two pointers where one is pointing to the head, and the other one is pointing to the tail in which a new element will be enqueued. Both enqueue and dequeue operations could be done in constant time by managing the two pointers correctly.

II. Linked list

A list could be singly linked or doubly linked. It could be sorted where the head contains the smallest element and the tail contains the largest element. It could be circular where the prev pointer of the head points to the tail, and the next pointer of the tail points to the head.

A circular, doubly linked list with a sentinel effectively allows us to manipulate pointers without worrying about the boundary conditions. This mainly adds clarity to our code while reducing some constant factors in terms of time complexity.

III. Implementing pointers and objects

It is feasible to implement a linked list in a language that doesn't natively support pointers and objects. The main idea is to use arrays for storage and indices for referencing to different data. For example, if objects have the same attributes, we could use multiple arrays, each of which stores a designated attribute for all objects, and the index to the array maps to a particular object one-to-one. Moreover, it is also feasible to store objects in a single array where an object's attributes are stored in contiguous subarray. It can flexibly fit objects of different attributes (heterogeneous objects), but there are overheads in managing them. Furthermore, allocating and freeing objects can be achieved by maintaining a separate free list.

IV. Representing rooted trees

A rooted tree can be represented using a linked data structure. A binary tree can be trivially represented using nodes and pointers. In case of a root tree with unbounded branching factor, the linked structure can also be used to efficiently manifest such a tree. Instead of having pointers to all children, a parent links to the left-most child and the immediately right sibling.

2. ***(30 points, Mandatory)*** *Write up a max one-page summary of all concepts and techniques in CLRS Chapter 12 (Binary Search Trees)*

Solution:

I. Definition of binary search tree

A search tree supports versatile operations, which allows us to use it as a priority queue or a dictionary in some sense. Most operations on a complete binary search tree of n nodes takes $\theta(\lg(n))$ time, proportional to its height. To be a binary search tree, a binary tree must maintain that a node's value must be greater than or equal to any value in its left subtree and must be smaller than or equal to any node in the right subtree. We can read the values in sorted order by doing an in-order traversal of the tree from the root.

II. Querying a binary search tree

There are some operations we can query from a binary search tree. These include finding a particular node, getting the minimum and maximum, and asking for a node's successor and predecessor. Note that the successor of a node N is different from the notion of children in that a successor has value greater than and the closest to the node's value. In addition, the successor might appear in levels above node N.

III. Insertion and deletion

It is relatively simple to insert a node into a BST because it is attached to one of the leafs. At every intermediate node, we decide the branch to go based on the binary-search-tree property. This operation takes time proportional to the height of the tree.

Deleting a node z from a binary search tree is a bit hectic. We tackle it case by case. First, if the node z does not have children, we simply remove it. Second, if the node z does have one child, the child will take over z's spot. Last but not the least, if z has two children, it is tricky to get it right. The successor of z needs to be found to take over z's position in the tree. The cases are vividly illustrated in the Figure 12.4 of the book. There are some subroutines such as TRANSPLANT and TREE-DELETE that facilitate shifting nodes around for the tricky case of delete.

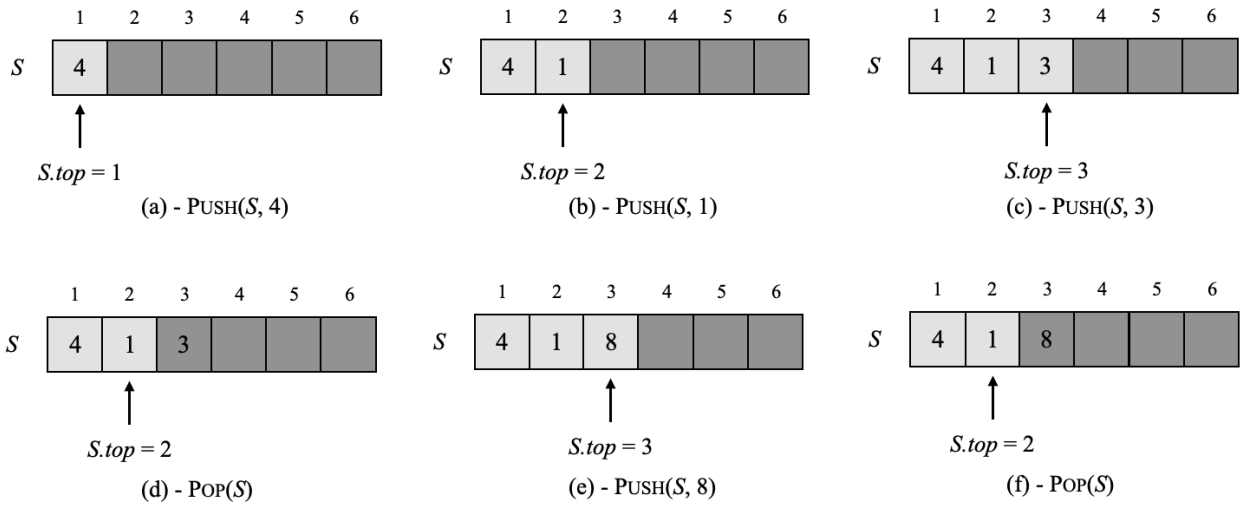
IV. Randomly built binary search trees

We explore the expected height of a randomly built binary search tree of n distinct keys. It was shown to be upper bounded by $O(\lg n)$ with the help of random variables, rules of expectations, as well as induction detailed in the proof of the theorem 12.4.

3. ***(10 points)*** *Exercise 10.1-1.*

Using Figure 10.1 as a model, illustrate the result of each operation in the sequence PUSH(S, 4), PUSH(S, 1), PUSH(S, 3), POP(S), PUSH(S, 8), and POP(S) on an initially empty stack S stored in array $S[1..6]$.

Solution:



4. ***(10 points)*** *Exercise 10.1-4.*

Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

Solution:

Because a queue of size n is designed to fit $n-1$ elements, it is full when Q.head == Q.tail + 1 or when Q.tail == Q.length if Q.head is still pointing to the first position.

```
1: function ENQUEUE(Q, x)
2:     if Q.head == Q.tail + 1 or (Q.head == 1 and Q.tail == Q.length) then
3:         error "overflow"                                  ▷ throws error and returns
4:     Q[Q.tail] = x
5:     if Q.tail == Q.length then
6:         Q.tail = 1
7:     else
8:         Q.tail = Q.tail + 1
```

A queue becomes empty when the head and tail pointer point to the same position.

```
1: function DEQUEUE(Q)
2:     if Q.head == Q.tail then
3:         error "underflow"                                 ▷ throws error and returns
4:     x = Q[Q.head]
5:     if Q.head == Q.length then
6:         Q.head = 1
7:     else
8:         Q.head = Q.head + 1
9:     return x
```

5. ***(10 points)*** *Exercise 10.1-6.*

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Solution:

Let's create two empty stacks A and B. The stack A is used to accept elements added to the queue. The stack B is mainly used when we want to remove elements from the queue.

The definition of ENQUEUE is trivial,

```
1: function ENQUEUE(A, B, x)
2:     PUSH(A, x)                          ▷ call the PUSH api of stack
```

We fill B by popping elements from A. Thus, B stores what have been in A in reversed order. This allows us to dequeue simply by popping from B. We refill B the same way when it becomes empty. The definition of DEQUEUE is as follows,

```
1: function DEQUEUE(A, B)
2:     if STACK-EMPTY(B) then
3:         while not STACK-EMPTY(A) do
4:             x = POP(A)
5:             PUSH(B, x)
6:     return POP(B)
```

ENQUEUE runs in $O(1)$-time because PUSH runs in $O(1)$-time. DEQUEUE runs in $O(n)$-time in the worst case (i.e. stack B is empty), otherwise it runs in $O(1)$-time by calling POP on B once.

Note that PUSH, POP, STACK-EMPTY are stack related functions defined in CLRS on page 233.

6. ***(Extra Credit)*** *Exercise 10.1-7.*

7. *(10 points)* *Exercise 10.2-2.*

Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$-time.

Solution:

To achieve "push", we simply link the next pointer of a new node to the current head of a list L.

```
1: function PUSH(L, value)
2:     init a new node x
3:     x.key = value
4:     x.next = L.head
5:     L.head = x
```

To achieve "pop", we want to remove the element lastly added, which is at the head of the list.

```
1: function POP(L)
2:     x = L.head
3:     if x ≠ NIL then
4:         key = x.key
5:         L.head = L.head.next
6:         return key
7:     else
8:         error "underflow"
```

Pointer manipulations and if condition checks are constant time operations in the above two functions. Thus, their runtimes are $O(1)$.

8. ***(10 points)*** *Exercise 10.2-6.*

The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

Solution:

Suppose the data structure we used to store S_1 and S_2 is a circular, doubly linked list with a sentinel. We are going to do some pointer manipulations to combine the two lists into one. Let's call the two lists A and B and try to link A's sentinel's prev ("tail") to the B's sentinel's next ("head").

```
1: function UNION(A, B)
2:     A.nil.prev.next = B.nil.next          ▷ link A's "tail" with B's "head"
3:     B.nil.next.prev = A.nil.prev
4:     B.nil.prev.next = A.nil               ▷ link A's sentinel with B's "tail"
5:     A.nil.prev = B.nil.prev
6:     delete B.nil                          ▷ the sentinel of B is no longer needed
```

The four pointer manipulations and the delete are constant-time operations. We thus manage to do UNION in $O(1)$ time.

9. ***(10 points)*** *Exercise 10.4-2.*

Write an $O(n)$ time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

Solution:

Suppose we want to do a pre-order traversal of the tree.

```
1: function PRINTTREE(T)
2:     if T.root ≠ NIL then
3:         print T.root.key
4:         PRINTTREE(T.root.left)
5:         PRINTTREE(T.root.right)
```

The print function has the recurrence $T(n) = 2T(n/2) + 1$. Using the master theorem, we have $\log_b^a = \log_2^2 = 1 < 0 = c$. Thus, the runtime is $O(n)$.

10. ***(Extra Credit)*** *Problem 10-1.*

11. ***(10 points)*** *Exercise 12.2-5.*

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Solution:

If a node has two children, it means that its predecessor and successor are under it. A successor exists in the right subtree, and it is the smallest element of the right subtree. We can find the successor at the end of a path going downward to the left. If it were to have a left child (i.e. a child that is even smaller), it shouldn't have been the minimum value for the right subtree: a contradiction. On the other hand, the predecessor exists in the left subtree, and it is the maximum element of the left subtree. We find the predecessor at the end of a path going downward to the right. If it were to have a right child (i.e. a child that is even greater), it shouldn't have been the maximum value of the left subtree: a contradiction.

12. ***(10 points)*** *Exercise 12.2-7.*

An alternative method of performing an in-order tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

Solution:

To help analyze the runtime of the algorithm, we keep track of how many times each node is visited while doing these procedures. TREE-MINIMUM visits nodes along a path downward to the left from the root. Those nodes are visited once. By calling TREE-SUCCESSOR on the smallest node (let's call it N_1), we find the successor to be its direct parent node N_2, which have been visited twice. Now, we aim to find the successor of N_2. We will either going upward when N_2 does not have a right child, or going downward when it does. If we decide to move upward, we will no longer visit nodes below N_2. If we go downward, we eventually will visit back to N_2 and up in order to find the successor of the largest value of N_2's right subtree. Regardless which direction we are going, we won't visit a node more than thrice.

The algorithm is upper bounded by $O(3n) = O(n)$ time. It is lower bounded by $\Omega(n)$ because there are n nodes in the tree to be visited at least once. Overall, the algorithm runs in $\Theta(n)$ time.

13. ***(10 points)*** *Exercise 12.3-3*

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an in-order tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Solution:

In the worst case, the numbers are already sorted when they are inserted. The BST will be extremely unbalanced. TREE-INSERT has to visit $n-1$ nodes in order to insert the last number into the right spot. It has to visit $n-2$ nodes to insert the second to last number. So on so forth, the cost is $(n-1)+(n-2)+(n-3)+...+2+1 = ((1+n-1)n)/2 = \Theta(n^2)$, which is the worst-case runtime.

In the best case, numbers are arranged in a way such that TREE-INSERT gradually forms a balanced BST while inserting the numbers one by one. We know that a balanced binary tree of n nodes have height $\Theta(\lg n)$. The algorithm is clearly upper bounded by $O(n \lg n)$ because at most the last number to be inserted visits $\lg n$ nodes (the height of the tree) to find its spot. We have shown that any comparison-based sorting algorithm is lower bounded by $\Omega(n \lg n)$. Thus, the runtime of this algorithm is $\Theta(n \lg n)$.

14. ***(Extra Credit)*** *Problem 12-3.*

15. ***(Extra Credit)*** *Problem 10-2.*

16. ***(Extra Credit)*** *Problem 15-6.*

17. ***(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.

Solution:

Please refer to code listed on the next page.

```
import re
import argparse
from pathlib import Path
from os import linesep

BASIC_TOKEN_REGEX = re.compile(r"[\w]*[a-zA-Z0-9_.]*[\w]")

def build_tokenizer(regex=BASIC_TOKEN_REGEX, exclude=None, lower_case=True):
    if exclude is None:
        exclude = set()

    def _tokenize(text):
        tokens = []
        for match in re.findall(regex, text):
            if lower_case:
                match = match.lower()
            if match in exclude:
                continue
            tokens.append(match)
        return tokens

    return _tokenize


def tokenize_text(file_path, encoding='iso-8859-1'):
    tokenize = build_tokenizer()
    with file_path.open("r", encoding=encoding) as fp:
        for line in fp:
            if len(line.strip()) > 0:
                for token in tokenize(line):
                    yield token


def write_to_file(file_path, content):
    with open(file_path, "w") as fp:
        fp.write(content)


class Node(object):
    def __init__(self, key, value, nxt=None):
        self.key = key
        self.value = value
        self.nxt = nxt


class HashTable(object):
    def __init__(self):
        self._size = 4999
        self._T = [None for _ in range(self._size)]
```

```
def _hash(self, text) -> int:
    h = 7
    for c in text:
        h = 31 * h + ord(c)
    return h % self._size

def insert(self, key, value):
    index = self._hash(key)
    if self._T[index] is None:
        self._T[index] = Node(key, value)
    else:
        self._T[index] = Node(key, value, self._T[index])

def delete(self, key):
    index = self._hash(key)
    prev = None
    cur = self._T[index]
    while cur is not None and cur.key != key:
        prev = cur
        cur = cur.child
    if cur is not None:
        nxt = None
        if cur.child is not None:
            nxt = cur.child
        if prev is None:
            self._T[index] = nxt
        else:
            prev.child = nxt
    else:
        print(f'key {key} does not exist')

def increase(self, key):
    node = self._find(key)
    if node is None:
        raise KeyError(f"The key {key} does not exist in the table")
    node.value += 1

def _find(self, key):
    index = self._hash(key)
    if self._T[index] is None:
        return None
    cur = self._T[index]
    while cur is not None and cur.key != key:
        cur = cur.child
    if cur is None:
        return None
    return cur
```

```
    def find(self, key):
        node = self._find(key)
        if node is None:
            return 0
        else:
            return node.value

    def list_all_keys(self):
        keys = []
        for t in self._T:
            if t is not None:
                cur = t
                while cur is not None:
                    keys.append(cur.key)
                    cur = cur.child
        return keys


def create_args_parser():
    parser = argparse.ArgumentParser('Hash table implementation')
    parser.add_argument('file_path', type=str, action='store',
                        help='the path of a text file to be indexed')
    return parser


def main():
    parser = create_args_parser()
    args = parser.parse_args()
    input_file = Path(args.file_path)
    if not input_file.is_file():
        raise FileNotFoundError(f'{input_file} is not found')

    table = HashTable()
    for word in tokenize_text(input_file):
        try:
            table.increase(word)
        except KeyError as e:
            table.insert(word, 1)

    output = linesep.join(f'{word} {table.find(word)}'
                          for word in table.list_all_keys())
    write_to_file(f'./{input_file.name}_table_dump', output)


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

18. ***(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:

Please refer to code listed on the next page.

```
import argparse
from os import linesep
from enum import Enum
from pathlib import Path

from print_tree import print_tree, PlaceHolder


def read_numbers_from(path):
    nums = []
    with path.open('r') as fp:
        for line in fp:
            sanitized = line.strip()
            try:
                sanitized = float(sanitized)
            except ValueError as e:
                print(e)
            else:
                nums.append(sanitized)
    return nums


class NodeColor(Enum):
    RED = 'R'
    BLACK = 'B'

    def __str__(self):
        return self.value


class Node(object):
    def __init__(self, key=None):
        self.color = None
        self.key = key
        self.left = None
        self.right = None
        self.p = None
        self.is_sentinel = False

    def __str__(self):
        if self.is_sentinel:
            return 'nil'
        else:
            return f'{self.color} {self.key}'


class print_red_black_tree(print_tree):
    def get_children(self, node: Node):
        children = []
```

```
        if node.right is not None and not node.right.is_sentinel:
            children.append(node.right)
        else:
            children.append(PlaceHolder)
        if node.left is not None and not node.left.is_sentinel:
            children.append(node.left)
        else:
            children.append(PlaceHolder)
        return children

    def get_node_str(self, node: Node):
        return f'[{node.color} {node.key}]'


class RedBlackTree(object):
    def __init__(self):
        self._sentinel = Node()
        self._sentinel.is_sentinel = True
        self._root = self._sentinel
        self._root.p = self._sentinel
        self._root.left = self._sentinel
        self._root.right = self._sentinel

    @property
    def root(self):
        return self._root

    @root.setter
    def root(self, value: Node):
        self._root = value

    @property
    def nil(self):
        return self._sentinel

    def print(self):
        return print_red_black_tree(self._root)


class RBTreeOps(object):
    @staticmethod
    def left_rotate(tree: RedBlackTree, x: Node):
        y = x.right
        x.right = y.left
        if y.left != tree.nil:
            y.left.p = x
        y.p = x.p
        if x.p == tree.nil:
            tree.root = y
```

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

    @staticmethod
    def right_rotate(tree: RedBlackTree, y: Node):
        x = y.left
        y.left = x.right
        if x.right != tree.nil:
            x.right.p = y
        x.p = y.p
        if y.p == tree.nil:
            tree.root = x
        elif y == y.p.right:
            y.p.right = x
        else:
            y.p.left = x
        x.right = y
        y.p = x

    @staticmethod
    def insert(tree: RedBlackTree, z: Node):
        y = tree.nil
        x = tree.root
        while x != tree.nil:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right
        z.p = y
        if y == tree.nil:
            tree.root = z
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z
        z.left = tree.nil
        z.right = tree.nil
        z.color = NodeColor.RED
        RBTreeOps.insert_fixup(tree, z)

    @staticmethod
    def insert_fixup(tree: RedBlackTree, z: Node):
        while z.p.color == NodeColor.RED:
            if z.p == z.p.p.left:  # z's parent is a left child
```

```
            y = z.p.p.right  # uncle of z
            if y.color == NodeColor.RED:  # case 1
                z.p.color = NodeColor.BLACK
                y.color = NodeColor.BLACK
                z.p.p.color = NodeColor.RED
                z = z.p.p
            else:  # case 2 or 3
                if z == z.p.right:  # transform from case 2 to 3
                    z = z.p
                    RBTreeOps.left_rotate(tree, z)
                z.p.color = NodeColor.BLACK  # case 3
                z.p.p.color = NodeColor.RED
                RBTreeOps.right_rotate(tree, z.p.p)
        else:
            y = z.p.p.left
            if y.color == NodeColor.RED:
                z.p.color = NodeColor.BLACK
                y.color = NodeColor.BLACK
                z.p.p.color = NodeColor.RED
                z = z.p.p
            else:
                if z == z.p.left:
                    z = z.p
                    RBTreeOps.right_rotate(tree, z)
                z.p.color = NodeColor.BLACK
                z.p.p.color = NodeColor.RED
                RBTreeOps.left_rotate(tree, z.p.p)
    tree.root.color = NodeColor.BLACK

@staticmethod
def sort(tree: RedBlackTree):
    def _sort(x):
        if x != tree.nil:
            left = _sort(x.left)
            right = _sort(x.right)
            return left + [x.key] + right
        else:
            return []

    return _sort(tree.root)

@staticmethod
def search(tree: RedBlackTree, key):
    def _search(x: Node, k):
        if x == tree.nil or k == x.key:
            return x
        if k < x.key:
            return _search(x.left, k)
        else:
```

```
            return _search(x.right, k)

    return _search(tree.root, key)

@staticmethod
def get_minimum(x: Node):
    while not x.left.is_sentinel:
        x = x.left
    return x

@staticmethod
def get_maximum(x: Node):
    while not x.right.is_sentinel:
        x = x.right
    return x

@staticmethod
def get_successor(tree: RedBlackTree, x: Node):
    if x.right != tree.nil:
        return RBTreeOps.get_minimum(x.right)
    y = x.p
    while y != tree.nil and x == y.right:
        x = y
        y = y.p
    return y

@staticmethod
def get_predecessor(tree: RedBlackTree, x: Node):
    if x.left != tree.nil:
        return RBTreeOps.get_maximum(x.left)
    y = x.p
    while y != tree.nil and x == y.left:
        x = y
        y = y.p
    return y

@staticmethod
def get_height(tree: RedBlackTree):

    def _get_height(x: Node):
        if x != tree.nil:
            return 1 + max(_get_height(x.left), _get_height(x.right))
        else:
            return 0

    return _get_height(tree.root)
```

```
def create_args_parser():
    parser = argparse.ArgumentParser('Red black tree implementation')
    parser.add_argument('file_path', type=str, action='store',
                        help='the path of a text file that contains numbers
                              line by line')
    return parser


def test():
    tree = RedBlackTree()
    n3 = Node(key=3)
    RBTreeOps.insert(tree, Node(key=4))
    RBTreeOps.insert(tree, Node(key=2))
    RBTreeOps.insert(tree, n3)
    RBTreeOps.insert(tree, Node(key=1))
    RBTreeOps.insert(tree, Node(key=0))
    tree.print()
    print(RBTreeOps.sort(tree))
    print(RBTreeOps.search(tree, 2))
    print(RBTreeOps.search(tree, 0))
    print(RBTreeOps.search(tree, 5))
    print(RBTreeOps.get_minimum(tree.root))
    print(RBTreeOps.get_maximum(tree.root))
    print(RBTreeOps.get_successor(tree, n3))
    print(RBTreeOps.get_predecessor(tree, n3))


def main():
    parser = create_args_parser()
    args = parser.parse_args()
    input_file = Path(args.file_path)
    if not input_file.is_file():
        raise FileNotFoundError(f'{input_file} is not found')

    tree = RedBlackTree()

    for num in read_numbers_from(input_file):
        RBTreeOps.insert(tree, Node(key=num))

    tree.print()

    while True:
        print("> Enter 'i', 's', 'f', or 'e' to select one of the
                 operations")
        print('- [i]insert, [s]sort, [f]find, [e]exit')
        option = input(': ').lower()
        if option == 'i':
            print('> Enter a number you want to insert')
            parsed_val = ''
```

```
            while type(parsed_val) is not float:
                value = input(': ')
                try:
                    parsed_val = float(value.strip())
                except ValueError:
                    print(f'> {value} is not a number, please retry')
            RBTreeOps.insert(tree, Node(key=parsed_val))
            tree.print()
        elif option == 's':
            print(f'| Sort result: {RBTreeOps.sort(tree)}')
        elif option == 'f':
            print('> Enter a number you want to search')
            parsed_val = ''
            while type(parsed_val) is not float:
                value = input(': ')
                try:
                    parsed_val = float(value.strip())
                except ValueError:
                    print(f'> {value} is not a number, please retry')
            print(f'| Search result: {RBTreeOps.search(tree, parsed_val)}')
        elif option == 'e':
            break
        else:
            print(f'{option} is invalid, please retry{linesep}')
            continue
        print(f'| Current height of the tree: {RBTreeOps.get_height(tree)}
              {linesep}')


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