```
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()
```