CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 7-8
Submit via Gradescope

Name: Shri Datta Madhira (NUID: 001557772)
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:

STACKS: Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is pre-specified. In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy. The INSERT operation on a stack is often called PUSH, and the DELETE operation is often called POP. S.top indexes the most recently inserted element. When S.top = 0, it means that the Stack is empty. If we attempt to pop an empty stack, we say the stack under-flows, which is normally an error. If S.top exceeds n, the stack overflows.

QUEUE: In a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy. We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE. The queue has a head and a tail. When an element is enqueued,it takes its place at the tail of the queue. The element dequeued is always the one at the head of the queue. In a queue, we "wrap around" in the sense that location 1 immediately follows location n in a circular order. If we attempt to dequeue an element from an empty queue, the queue underflows. When Q.head = Q.tail + 1, the queue is full, and if we attempt to enqueue an element, then the queue overflows.

LINKED LISTS: A linked list is a data structure in which the objects are arranged in a linear order. The order in a linked list is determined by a pointer in each object. each element of a doubly linked list L is an object with an attribute key and two other pointer attributes: next and prev. If x.prev = NIL, the element x has no predecessor and is therefore the first element, or head, of the list. If x.next = NIL, the element x has no successor and is therefore the last element, or tail, of the list. An attribute L.head points to the first element of the list. If L.head = NIL, the list is empty. A sentinel is a dummy object that allows us to simplify boundary conditions while deleting an element from the Linked Lists.

IMPLEMENTING POINTERS and OBJECTS: We can represent a collection of objects that have the same attributes by using an array for each attribute. For a given array index x, the array entries key[x], next[x], and prev[x] represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the key, next, and prev arrays. A variable L holds the index of the head of the list.
In some systems, a garbage collector is responsible for determining which objects are unused.

ROOTED TREES: We use the attributes p, left, and right to store pointers to the parent, left child, and right child of each node in a binary tree T. The root of the entire tree T is pointed to by the attribute T.root. If T.root = NIL, then the tree is empty. The left-child, right-sibling representation. Instead of having a pointer to each of its children, however, each node x has only two pointers:
1. x.left-child points to the leftmost child of node x, and
2. x.right-sibling points to the sibling of x immediately to its right.

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

Solution:

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in Θlgn worst-case time. If the tree is a linear chain of n nodes, however, the same operations take Θn worst-case time.
What is a binary search tree? The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property:

```
Let x be a node in a binary search tree.
If y is a node in the left subtree of x, then y.key <= x.key.
If y is a node in the right subtree of x, then y.key >= x.key.
```

SEARCHING: Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

MINIMUM: We can always find an element in a binary search tree whose key is a minimum by following left child pointers from the root until we encounter a NIL.
MAXIMUM: The same goes with the maximum element but we follow the right child pointers from the root until we encounter a NIL.

SUCCESSOR: If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key.
PREDECESSOR: If all keys are distinct, the predecessor of a node x is the node with the greatest key smaller than x.key.

INSERTION: Begins at the root of the tree and the pointer x traces a simple path downward looking for a NIL to replace with the input item z.
DELETION: The overall strategy for deleting a node ´ from a binary search tree T has three basic cases,

```
CASE 1: If z has no children, then we simply remove it by modifying its
parent to replace z with NIL.

CASE 2: If z has just one child, then we elevate that child to take z's
position in the tree by modifying z's parent to replace z by z's child.

CASE 3: If z has two children, then we find z's successor y - which
must be in z's right subtree | and have y take z's position in the tree.
The rest of z's original right subtree becomes y's new right subtree,
and z's left subtree becomes y's new left subtree. This case is a tricky
one because, as we shall see, it matters whether y is z's right child.
```

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

Solution:

PUSH(S, 4): S.top = 1

1	2	3	4	5	6
4					

PUSH(S, 1): S.top = 2

1	2	3	4	5	6
4	1				

PUSH(S, 3): S.top = 3

1	2	3	4	5	6
4	1	3			

POP(S): S.top = 2

1	2	3	4	5	6
4	1				

PUSH(S, 8): S.top = 3

1	2	3	4	5	6
4	1	8			

POP(S): S.top = 2

1	2	3	4	5	6
4	1				

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

Solution:

Algorithm 1: Algorithm to ENQUEUE and DEQUEUE

```
Function ENQUEUE(Q,x):
    If Q.head == Q.tail+1 :
        Return "OVERFLOW"
    Q[Q.tail] = x
    If Q.tail == Q.length :
        Q.tail = 1
    Else
        Q.tail = Q.tail + 1
    Return 1
==========================================================
Function DEQUEUE(Q):
    If Q.head == Q.tail :
        Return "UNDERFLOW"
    x = Q[Q.head]
    If Q.head == Q.length :
        Q.head = 1
    Else
        Q.head = Q.head + 1
    Return x
```

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

Solution:

Algorithm 4: DEQUEUE

```
Function QUEUE-2-STACK(Q):
    S1[1...n], S2[1...n]
    ENQUEUE(x,S1,S2)
    DEQUEUE(S1,S2)
==========================================================
Function QUEUE-2-STACK-ENQUEUE(x,S1,S2):
    S2.top = S2.top + 1
    S2[S2.top] = x
==========================================================
Function QUEUE-2-STACK-DeQUEUE(S1,S2):
    If STACK-EMPTY(S1) and STACK-EMPTY(S2) :
        Return "UNDERFLOW"
    If STACK-EMPTY(S1) :
        While !STACK-EMPTY(S2):
            S1.top = S1.top + 1
            S2.top = S2.top - 1
            S1[S1.top] = S2[S2.top+1]
        Return S1[S1.top]
```

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

Solution:

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

Solution:

Algorithm 5: Algorithm to implement Stack using LinkedList

```
Function STACK-LINKED-LIST():
    PUSH(S, x)
    POP(S)
================================================================
Function PUSH(S, val):
    temp = Node()
    temp.data = val
    temp.next = head
    head = temp
================================================================
Function POP(S):
    temp = head
    head = temp.next
    data = temp.data
    Free-Memory(temp)
    Return data
```

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

Solution: To perform the UNION operation in $O(1)$ time, we need to use a **Circular Double Linked List**. By using the circular double linked list, we will have the link from tail node to head node and also from head node to tail node. By having these links, we can access the tail node of S1 in O(1) time and append the head of S2 to the tail of S1 in O(1) time. An algorithm for this is shown below.

Algorithm 6: Algorithm to support UNION in O(1) time using a suitable list data structure.

```
Function UNION-O(1)-LIST(S1, S2):
    COMMENT: head1 is head of S1 list.
    COMMENT: head2 is head of S2 list.
    COMMENT: Storing tail1 in temp.
    temp = head1.prev
    COMMENT: Storing tail2 in temp2
    temp2 = head2.prev
    COMMENT: Making the tail1.next as head2
    temp.next = head2
    COMMENT: Making head2.prev as tail1.
    head2.prev = temp
    COMMENT: Making temp2 our new tail and making new connections.
    head1.prev = temp2
    temp2.next = head1
```

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

Solution:

Algorithm 7: Algorithm to print out the keys of each node in the tree.

```
Function TREE-PRINT(T):
    If T == NULL :
        Return
    print(T.data)
    TREE-PRINT(T.right)
    TREE-PRINT(T.left)
```

10. ***(Extra Credit)*** *Exercise 10.1.*

Solution:

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

Solution:

Let us assume the given node is x. It has 2 children, $child_1$ and $child_2$.

Successor(x) is the left most element in the right subtree, and Predecessor(x) is the right most element in the left subtree.

If the Successor(x) has an element to the left of it, say 'a', that means, that element 'a' is smaller than Successor(x) but bigger than x, according to the binary search tree property. Mathematically, $\boldsymbol{x < a < Successor(x)}$. This contradicts our initial result. Therefore, for any node 'x', it's successor will not have any left child, Successor(x).left = NIL.

The same argument goes for the predecessor. If the Predecessor(x) has an element to the right of it, say 'b', that means that element 'b' is bigger than Predecessor(x) but smaller than x, according to the binary search tree property. Mathematically, $\boldsymbol{x > b > Predecessor(x)}$. This contradicts out initial result. Therefore, for any node 'x', it's predecessor will not have any right child, Predecessor(x).right = NIL

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

Solution: The time taken by TREE-MINIMUM(Root) to find the minimum of a n-node binary search tree is $O(h)$.

So, we still have n-1 nodes to be called and printed. By using TREE-SUCCESSOR(x), we can observe that, for each node 'x', we are traversing at most 2 edges of the tree.

So, the time taken to run this for n-1 other nodes is $O(n)$. So, the total time complexity will be $O(h) + O(n)$ which asymptotically equals $O(n)$.

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

Solution:

The in-order walk always takes $O(n)$ time regardless of the tree since we have to visit all the nodes of the tree. The insertions of the tree here are the real problem.

The worst case happens when the given numbers are already sorted in ascending order. Then the tree gets skewed to one side. The next element to be inserted will have to travel through all the elements that came before it. Thus making the running time $\boldsymbol{\theta(n^2)}$.

The best case happens when the given numbers form a perfect (balanced) binary search tree. At each insertion, the numbers only need to travel for $\theta(\lg n)$ distance before finding their place. This gives us a time complexity of $\boldsymbol{O(n \lg n)}$.

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

Solution:

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

Solution:

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

Solution:

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 data stuctures that can index by strings (like hash tables). Use a language that easily implements linked lists, like C/C++.You can test your code on "Alice in Wonderland" by Lewis Carroll,at

`http://www.ccs.neu.edu/home/vip/teach/Algorithms/7_hash_RBtree_simpleDS/hw_hash_RBtree/alice_in_wonderland.txt`

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, built a linked list with positions in the given text. Output this list together with the count.

Solution:

```
import math


class Node:
    def __init__(self, datakey=None, dataval=None):
        self.datakey = datakey
        self.dataval = dataval
        self.nextval = None


class SLL:
    def __init__(self):
        self.headval = None

    def insert_at_start(self, newkey, newval):
        NewNode = Node(newkey, newval)
        NewNode.nextval = self.headval
        self.headval = NewNode

    def delete(self, Removekey):
        HeadVal = self.headval
        if HeadVal is not None:
            if HeadVal.datakey == Removekey:
                self.headval = HeadVal.nextval
                HeadVal = None
                return
        while HeadVal is not None:
```

```
            if HeadVal.datakey == Removekey:
                break
            prev = HeadVal
            HeadVal = HeadVal.nextval
        if HeadVal is None:
            return
        prev.nextval = HeadVal.nextval
        HeadVal = None

    def print(self, index):
        printval = self.headval
        if printval is None:
            print(index, "---> None")
            return
        print(index, end=" ")
        while printval is not None:
            print("-->", end=" ")
            print("[", printval.datakey, ", ", printval.dataval, "]",
               end=" ")
            printval = printval.nextval
        print()

    def search(self, key):
        is_there = 0
        temp_head = self.headval
        while temp_head is not None:
            if temp_head.datakey == key:
                is_there = 1
                break
            else:
                temp_head = temp_head.nextval
        if is_there == 1:
            return temp_head
        else:
            return -1


def display_hash(hashTable):
    for j in range(len(hashTable)):
        hashTable[j].print(j)


# Hashing Function to return key for every value.
def Hashing(key):
    hash_val = 5381
```

```
    asc = 0
    ord_list = [ord(c) for c in key]
    ord_list.reverse()
    for i in range(len(ord_list)):
        asc += (ord_list[i] * (128 ** i)) % (len(HashTable))
    return (((hash_val << 5) + hash_val) + asc) % (len(HashTable))


# Insert Function to add values to the hash table
def insert(Hashtable, head, hash_val, key, value):
    head.insert_at_start(key, value)
    Hashtable[hash_val] = head


f = open("./aliceinwonderland", 'r')
lines = f.read()
f.close()
length = len(lines.split())

# Creating Hashtable
HashTable = []
for index in range(math.ceil(length / 5)):
    HashTable.append(SLL())

for i in lines.split():
    flag = 0
    hash_value = Hashing(i.lower())
    hashes_head = HashTable[hash_value]
    if hashes_head.headval is None:
        insert(HashTable, hashes_head, hash_value, i.lower(), 1)
    else:
        flag = hashes_head.search(i.lower())
        if flag == -1:
            insert(HashTable, hashes_head, hash_value, i.lower(), 1)
        else:
            flag.dataval += 1

ch = True
while ch:
    print("1. Insert 2. Delete 3. Print all keys 4. Search 0. Exit")
    option = int(input("Enter any option: "))
    if option == 1:
        word = str(input("Enter the word you want to Insert: "))
        hash_value = Hashing(word.lower())
        head = HashTable[hash_value]
```

```
        flag = head.search(word.lower())
        if flag == -1:
            insert(HashTable, head, hash_value, word.lower(), 1)
        else:
            flag.dataval += 1
    elif option == 2:
        word = str(input("Enter the word you want to Delete: "))
        hash_value = Hashing(word.lower())
        head = HashTable[hash_value]
        head.delete(word.lower())
    elif option == 3:
        display_hash(HashTable)
    elif option == 4:
        word = str(input("Enter the word you want to Search: "))
        hash_value = Hashing(word.lower())
        head = HashTable[hash_value]
        flag = head.search(word.lower())
        if flag == -1:
            print(word, "not found")
        else:
            print("KEY:", flag.datakey)
            print("WORD COUNT:", flag.dataval)
    elif option == 0:
        ch = False
    else:
        print("Enter a valid option.")
```

EXAMPLE OUTPUT:

```
1. Insert 2. Delete 3. Print all keys 4. Search 0. Exit
Enter any option: 4
Enter the word you want to Search: format
KEY: format
WORD COUNT: 4
1. Insert 2. Delete 3. Print all keys 4. Search 0. Exit
Enter any option: 1
Enter the word you want to Insert: format
1. Insert 2. Delete 3. Print all keys 4. Search 0. Exit
Enter any option: 5
Enter a valid option.
1. Insert 2. Delete 3. Print all keys 4. Search 0. Exit
Enter any option: 0

Process finished with exit code 0
```

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:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;

struct Node {
  int data;
  Node *parent;
  Node *left;
  Node *right;
  int color;
};

typedef Node *NodePtr;

class RBTree {
   private:
  NodePtr root;
  NodePtr TNULL;

  void initializeNULLNode(NodePtr node, NodePtr parent) {
    node->data = 0;
    node->parent = parent;
    node->left = NULL;
    node->right = NULL;
    node->color = 0;
  }
```

```
// Preorder
void preOrder(NodePtr node) {
  if (node != TNULL) {
    cout << node->data << " ";
    preOrderHelper(node->left);
    preOrderHelper(node->right);
  }
}

// Inorder
void inOrder(NodePtr node) {
  if (node != TNULL) {
    inOrderHelper(node->left);
    cout << node->data << " ";
    inOrderHelper(node->right);
  }
}

// Post order
void postOrder(NodePtr node) {
  if (node != TNULL) {
    postOrderHelper(node->left);
    postOrderHelper(node->right);
    cout << node->data << " ";
  }
}

NodePtr search(NodePtr node, int key) {
  if (node == TNULL || key == node->data) {
    return node;
  }

  if (key < node->data) {
    return searchTreeHelper(node->left, key);
  }
  return searchTreeHelper(node->right, key);
}
```

```
// For balancing the tree after deletion
void deleteFix(NodePtr x) {
  NodePtr s;
  while (x != root && x->color == 0) {
    if (x == x->parent->left) {
      s = x->parent->right;
      // Case 1: If sibling is RED.
      if (s->color == 1) {
        s->color = 0;
        x->parent->color = 1;
        leftRotate(x->parent);
        s = x->parent->right;
      }
      // Case 2: If sibling is BLACK, and its children are BLACK
      if (s->left->color == 0 && s->right->color == 0) {
        s->color = 1;
        x = x->parent;
      } else {
        if (s->right->color == 0) {
          s->left->color = 0;
          s->color = 1;
          rightRotate(s);
          s = x->parent->right;
        }

        s->color = x->parent->color;
        x->parent->color = 0;
        s->right->color = 0;
        leftRotate(x->parent);
        x = root;
      }
    } else {
      s = x->parent->left;
      if (s->color == 1) {
        s->color = 0;
        x->parent->color = 1;
        rightRotate(x->parent);
        s = x->parent->left;
      }

      if (s->right->color == 0 && s->right->color == 0) {
        s->color = 1;
        x = x->parent;
      } else {
        if (s->left->color == 0) {
```

```
          s->right->color = 0;
          s->color = 1;
          leftRotate(s);
          s = x->parent->left;
        }

        s->color = x->parent->color;
        x->parent->color = 0;
        s->left->color = 0;
        rightRotate(x->parent);
        x = root;
      }
    }
  }
  x->color = 0;
}

void rbTransplant(NodePtr u, NodePtr v) {
  if (u->parent == NULL) {
    root = v;
  } else if (u == u->parent->left) {
    u->parent->left = v;
  } else {
    u->parent->right = v;
  }
  v->parent = u->parent;
}

void delete(NodePtr node, int key) {
  NodePtr z = TNULL;
  NodePtr x, y;
  // Finding the appropriate node.
  while (node != TNULL) {
    if (node->data == key) {
      z = node;
      break;
    }

    if (node->data <= key) {
      node = node->right;
    } else {
      node = node->left;
    }
  }
  // Node not found.
```

```
  if (z == TNULL) {
    cout << "Key not found in the tree" << endl;
    return;
  }

  y = z;
  int y_original_color = y->color;
  // If the node being removed 'z' has only one child.
  if (z->left == TNULL) {
    x = z->right;
    rbTransplant(z, z->right);
  } else if (z->right == TNULL) {
    x = z->left;
    rbTransplant(z, z->left);
  }
  // If the node being removed 'z' has two children.
  else {
    y = minimum(z->right); // Then the node replacing 'z' should be z
       's successor.
    y_original_color = y->color;
    x = y->right;
    if (y->parent == z) {
      x->parent = y;
    } else {
      rbTransplant(y, y->right); // Assiging y->right to y's parent
      y->right = z->right;
      y->right->parent = y;
    }

    rbTransplant(z, y); // Assinging y to z's parent.
    y->left = z->left;
    y->left->parent = y;
    y->color = z->color;
  }
  delete z;
  // If it was a black node, it can induce some violations to RB
     properties.
  if (y_original_color == 0) {
    deleteFix(x);
  }
}

 public:
RBTree() {
  TNULL = new Node;
```

```
  TNULL->color = 0;
  TNULL->left = NULL;
  TNULL->right = NULL;
  root = TNULL;
}

void preorder() {
  preOrder(this->root);
}

void inorder() {
  inOrder(this->root);
}

void postorder() {
  postOrder(this->root);
}

NodePtr searchTree(int k) {
  return search(this->root, k);
}

NodePtr minimum(NodePtr node) {
  while (node->left != TNULL) {
    node = node->left;
  }
  return node;
}

NodePtr maximum(NodePtr node) {
  while (node->right != TNULL) {
    node = node->right;
  }
  return node;
}

NodePtr successor(NodePtr x) {
  if (x->right != TNULL) {
    return minimum(x->right);
  }

  NodePtr y = x->parent;
  while (y != TNULL && x == y->right) {
    x = y;
    y = y->parent;
```

```
  }
  return y;
}

NodePtr predecessor(NodePtr x) {
  if (x->left != TNULL) {
    return maximum(x->left);
  }

  NodePtr y = x->parent;
  while (y != TNULL && x == y->left) {
    x = y;
    y = y->parent;
  }

  return y;
}

void leftRotate(NodePtr x) {
  NodePtr y = x->right;
  x->right = y->left;
  if (y->left != TNULL) {
    y->left->parent = x;
  }
  y->parent = x->parent;
  if (x->parent == NULL) {
    this->root = y;
  } else if (x == x->parent->left) {
    x->parent->left = y;
  } else {
    x->parent->right = y;
  }
  y->left = x;
  x->parent = y;
}

void rightRotate(NodePtr x) {
  NodePtr y = x->left;
  x->left = y->right;
  if (y->right != TNULL) {
    y->right->parent = x;
  }
  y->parent = x->parent;
  if (x->parent == NULL) {
    this->root = y;
```

```
  } else if (x == x->parent->right) {
    x->parent->right = y;
  } else {
    x->parent->left = y;
  }
  y->right = x;
  x->parent = y;
}

// For balancing the tree after insertion
void insertFix(NodePtr k) {
  NodePtr u;
  while (k->parent->color == 1) {
    if (k->parent == k->parent->parent->right) {
      u = k->parent->parent->left;
      // Case 1: Uncle 'u' is RED.
      if (u->color == 1) {
        u->color = 0;
        k->parent->color = 0;
        k->parent->parent->color = 1;
        k = k->parent->parent;
      } else {
        // Case 2: Uncle is Black, k->parent is right and k is left
           child
        if (k == k->parent->left) {
          k = k->parent;
          rightRotate(k);
        }
        // Case 3: Uncle is black, k->parent is left and k is right
           child
        k->parent->color = 0;
        k->parent->parent->color = 1;
        leftRotate(k->parent->parent);
      }
    } else {
      u = k->parent->parent->right;
      // Case 1 : Uncle is RED
      if (u->color == 1) {
        u->color = 0;
        k->parent->color = 0;
        k->parent->parent->color = 1;
        k = k->parent->parent;
      } else {
        // Case 2: Uncle is Black, and k->parent is left and k is
           right child.
```

```
        if (k == k->parent->right) {
          k = k->parent;
          leftRotate(k);
        }
        // Case 3: Uncle is Black, and k->parent is left and k is
           left child.
        k->parent->color = 0;
        k->parent->parent->color = 1;
        rightRotate(k->parent->parent);
      }
    }
    if (k == root) {
      break;
    }
  }
  root->color = 0;
}

// Inserting a node
void insert(int key) {
  NodePtr node = new Node;
  node->parent = NULL;
  node->data = key;
  node->left = TNULL;
  node->right = TNULL;
  node->color = 1;

  NodePtr y = NULL;
  NodePtr x = this->root;

  while (x != TNULL) {
    y = x;
    if (node->data < x->data) {
      x = x->left;
    } else {
      x = x->right;
    }
  }

  node->parent = y;
  if (y == NULL) {
    root = node;
  } else if (node->data < y->data) {
    y->left = node;
  } else {
```

```
      y->right = node;
    }

    if (node->parent == NULL) {
      node->color = 0;
      return;
    }

    if (node->parent->parent == NULL) {
      return;
    }

    insertFix(node);
  }

  NodePtr getRoot() {
    return this->root;
  }

  void deleteNode(int data) {
    deleteNodeHelper(this->root, data);
  }


  void printTree() {
    //   cout << "In printTree" << endl;
    //   cout << "Root: " << this->root << endl;
    if (root) {
        inorder(root);
    }
  }

    int height(Node *node){
        if(node == TNULL){
            return 0;
        }
        if(height(node->left) <= height(node->right)){
            return 1+height(node->right);
        }
        else{
            return 1+height(node->left);
        }
    }
};
```

```
int main() {
  RBTree bst;
  std::string line;
  int val;
    ifstream myfile;
    myfile.open("rbtree.txt");
    if(myfile.is_open()){
              while ( getline (myfile,line) )
              {
                  stringstream ss(line);
                  ss >> val;
                  bst.insert(val);
              }
              myfile.close();
    }
    else{
        cout << "Unable to open the file" << endl;
        exit(0);
    }
  int choice;
    cout << "Choices available:" << endl;
    cout << "1.insert 2.delete 3.print 4.min 5.max 6.successor 7.
       predecessor 8.search 9.height 0.exit" << endl;
    do{
        cout << "Enter any option above: ";
        cin >> choice;
        Node* root = bst.getRoot();
    switch (choice) {
        case 1:
        {
            cout << "Enter a value to insert:";
            cin >> val;
            bst.insert(val);
            break;
            // cout << endl
            //   << "After inserting" << endl;
            // bst.printTree();
        }
        case 2:
        {
            cout << "Enter a value to delete:";
            cin >> val;
            bst.deleteNode(val);
            break;
            // cout << endl
```

```
    //   << "After deleting" << endl;
    // bst.printTree();
}
case 3:
    bst.printTree();
    break;

case 4:
{
    Node* min = bst.minimum(root);
    cout << "Minimum of Tree:" << min->data << endl;
    break;
}
case 5:
{
    Node* max = bst.maximum(root);
    cout << "Maximum of Tree:" << max->data << endl;
    break;
}
case 6:
{
    cout << "Enter the element you want successor for:";
    cin >> val;
    Node* suc = bst.successor(bst.searchTree(val));
    cout << "Successor is:" << suc->data << endl;
    break;
}
case 7:
{
    cout << "Enter the element you want predecessor for:";
    cin >> val;
    Node* pred = bst.predecessor(bst.searchTree(val));
    cout << "Predecessor is:" << pred->data << endl;
    break;
}
case 8:
{
    cout << "Enter the element you want to search for:";
    cin >> val;
    Node* item = bst.searchTree(val);
    cout << "0 - BLACK 1-RED" << endl;
    cout << "The key is " << item->data << " Color:" << item->
       color << endl;
    break;
}
```

```
        case 9:
        {
            cout << "The height of the tree is:" << bst.height(root) <<
                endl;
            break;
        }
        case 0:
            exit(0);
        default:
            cout << "Please choose a relevant option" << endl;
            break;
    }
    }while(choice != 0);
}
```

EXAMPLE OUTPUT:

```
Choices available:
1.insert 2.delete 3.print 4.min 5.max 6.successor 7.predecessor 8.
   search 9.height 0.exit
Enter any option above: 4
Minimum of Tree:40
Enter any option above: 6
Enter the element you want successor for:40
Successor is:55
Enter any option above: 9
The height of the tree is:4
Enter any option above: 8
Enter the element you want to search for:75
0 - BLACK 1-RED
The key is 75 Color:0
Enter any option above: 0
Program ended with exit code: 0
```