CPSC 319

ASSIGNMENT 3

John Ezekiel Juliano

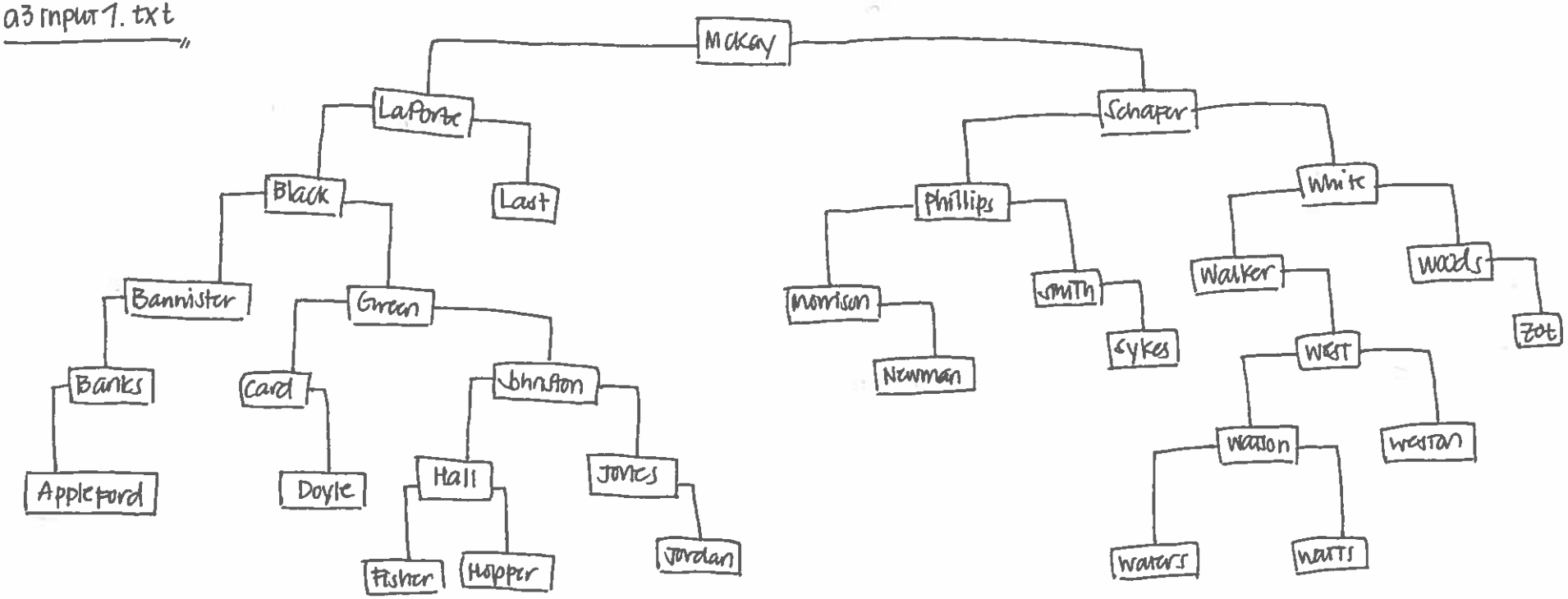
30000523

T05

Rashid Mamunur

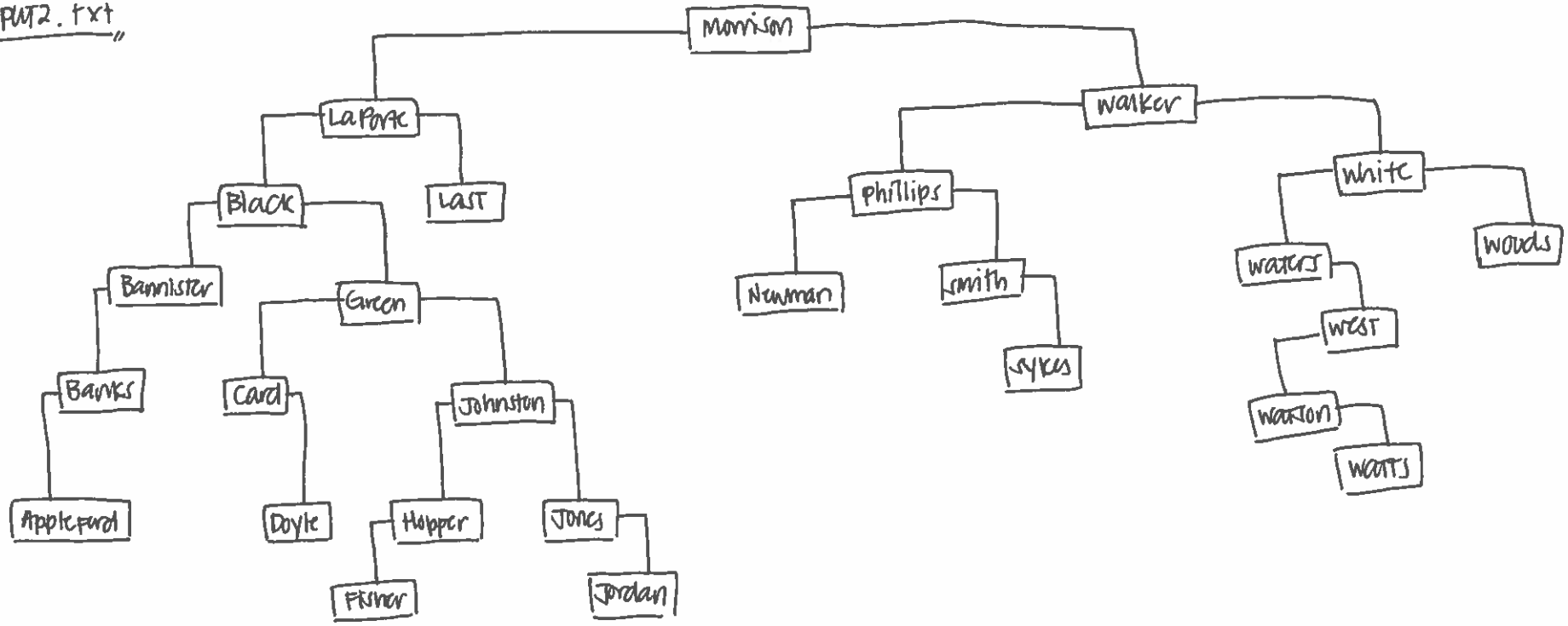
<Readme>
javac Assign3.java BST.java Student.java
java Assign3 [inputfilename] [output1_filename] [output2_filename]

a3input7.txt



height of Tree = 6 (7 levels)

a3input2.txt



height of Tree = 6 (7 levels)

COMPLEXITY ANALYSIS

(1) If the input is in random order, the height of the tree would be approximately $O(\lg n)$. This is similar to the bestcase scenario when the tree is balanced & the height is minimized. The worstcase on the other hand, when the first or last record (ie: a lastname w/ A or Z as the first letter), is inserted first, the tree degenerates to a linked list producing a height of $O(n)$.

(2) The worst case height is $O(n)$. This happens when a last name with a first letter of A or Z is inserted first. A linked list results from this instead of a binary search tree.

(3) Depth first traversal

```
public void DFT ( BSTnode root, PrintWriter cursor) {
    if (root != null) {
        DFT ( root.getLeft(), cursor );
        cursor.println ( root.getStudent() );
        DFT ( root.getRight(), cursor );
    }
}
```

The worst case scenario would be when the smallest data is inserted first & all successive data is larger than the other. This also means that there are no left subtrees, just right subtrees, and it resembles a linked list.

Taking into consideration that left children/subtrees are checked first, if there are n - inputs there would also be n - times checking for any left childrens.

This results into $O(n^2)$ assumming that one method call is equivalent to one unit of memory. The same thing also happens when the biggest data is inserted first & only a left subtree is present.

```
public void BFT ( PrintWriter cursor ){ // prints the tree level by level
    int level, height = treeHeight(root);
    for ( level =1 ; level <= height ; level ++)
        printLevel ( root , level, cursor );
}
```

↳ the worst case for this function would be "height" times calls to printLevel. This would mean a $O(n)$ for a binary search tree that degenerated to a linked list.

```
privat int treeHeight (BSTnode root) { // computes the height of the tree
    if (root==null)
        return 0;
    else {
        int leftHeight = treeHeight (root.left);
        int rightHeight = treeHeight (root.right);
        if( leftHeight > rightHeight)
            return (leftHeight +1);
        else
            return (rightHeight +1);
    }
}
```

↳ the worst case scenario is if the the is one sided (ie: left-heavy or right-heavy). This makes a space complexity of $O(n^2)$ because it checks one side then, the other. After that, a comparism is made.

```
private void printLevel (BSTnode root, int level, PrintWriter cursor){ // prints the nodes per level
    if (root == null)
        return;
    if (level == 1)
        cursor.println( root.getStudent());
    else if (level > 1) {
        printLevel(root.left, level-1, cursor);
        printLevel (root.right, level-1, cursor);
    }
}
```

↳ the worst case for this function is if every node had a left & right child (except for the last nodes). This would require occupying n-times the unit of memory $\Rightarrow O(n)$.

In terms of the space occupied by these sets of function, the worst case would be $O(n^2)$.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;

public class Assign3 {

    BST recordTree;
    String inputFilename;
    String outputFilename1;
    String outputFilename2;

    public Assign3() {
        recordTree = new BST();
    }

    public void readTextFile() {
        try {
            BufferedReader in = new BufferedReader(new FileReader(inputFilename));
            Student data;
            String text = null, ln, dept, prog;
            char oc;
            int id = 0, y = 0;
            char[] charArr = new char[42];
            while((text = in.readLine()) != null) {
                charArr = text.toCharArray();
                oc = charArr[0];
                id =  Integer.parseInt(String.valueOf(charArr, 1, 7));
                ln = String.valueOf(charArr, 8, 25);
                dept = String.valueOf(charArr, 33, 4);
                prog = String.valueOf(charArr, 37, 4);
                y = Integer.parseInt(String.valueOf(charArr, 41, 1));
                data = new Student(ln,dept,prog,id,y);
                if(oc == 'I')
                    recordTree.insert(data);
                if(oc == 'D')
                    recordTree.delete(data);
                charArr = new char[42];
            }
            in.close();
        }
        catch(IOException e) {
            System.err.println(e.getMessage());
        }
        catch(NumberFormatException n) {
            System.err.println(n.getMessage());
        }
    }

    public static void main(String[] args) {
        Assign3 test = new Assign3();
        test.inputFilename = args[0].toString();
        test.outputFilename1 = args[1].toString();
        test.outputFilename2 = args[2].toString();
        test.readTextFile();
        PrintWriter cursor;
        try {
            cursor = new PrintWriter(test.outputFilename1);
            cursor.println("ID      LASTNAME                  DEPT PROG YEAR\n" +
                           "-----------------------------------------------");
            test.recordTree.DFT(test.recordTree.root, cursor);
            cursor.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
```

```
        try {
            cursor = new PrintWriter(test.outputFilename2);
            cursor.println("ID      LASTNAME                  DEPT PROG YEAR\n" +
                           "-----------------------------------------------");
            test.recordTree.BFT(cursor);
            cursor.close();
        }
        catch(FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

/*****************************************************************************************/

public class Student {

    String  lastName;
    String  department;
    String  program;
    int     ID;
    int     year;

    public Student(String l, String d, String p, int i, int y) {
        lastName = l;
        department = d;
        program = p;
        ID = i;
        year = y;
    }

    public String getLastname() {return lastName;}
    public String getDepartment() {return department;}
    public String getProgram() {return program;}
    public int getID() {return ID;}
    public int getYear() {return year;}

    @Override
    public String toString() {
        return String.format(ID + " " + lastName + " " + department + " " + program + " " + year
                             + "\n-----------------------------------------------");
    }

}

/*****************************************************************************************/

import java.io.PrintWriter;
/*
 * This class creates a binary search tree
 * that contains student data within each node
 */
public class BST {
    public class BSTnode{
        // student data
        Student student;
        // left and right child pointers
        BSTnode left;
        BSTnode right;
        public BSTnode(Student s) {
            student = s;
            left = null;
            right = null;
        }
        public Student getStudent() {return student;}
        public BSTnode getLeft() {return left;}
```

```
    public BSTnode getRight() {return right;}
}
// root node
public BSTnode root;
public BST() {
    root = null;
}
 // Search for a node that contains the key given in the argument of the method
public boolean search(String name) {
    BSTnode current = root;
    while(current!=null) {
        // if key is found return
        if(current.getStudent().getLastname().equalsIgnoreCase(name))
            return true;
        // if current key > key go to left subtree
        else if(current.getStudent().getLastname().compareTo(name) > 0)
            current = current.getLeft();
        // else go to right subtree
        else
            current = current.getRight();
    }
    return false;
}
// Deletes a node that contains the data specified by the method argument
public boolean delete(Student s) {
    BSTnode current = root, parent = root;
    // identifies if the node is a leftChild
    boolean leftChild = false;
    // find the node
    while(!(current.getStudent().getLastname().equalsIgnoreCase(s.getLastname()))) {
        // track the parent node
        parent = current;
        // if current key > key go left
        if(current.getStudent().getLastname().compareTo(s.getLastname()) > 0) {
            leftChild = true;
            current = current.getLeft();
        }
        // else go right
        else {
            leftChild = false;
            current = current.getRight();
        }
        // if current is null return
        if(current == null)
            return false;
    }
    // CASE 1: LEAF NODE
    if(current.getLeft() == null && current.getRight() == null) {
        // if empty
        if(current == root)
            root = null;
        // if the node to be deleted is a left child
        if(leftChild)
            parent.left = null;
        // if the node to be deleted is a right child
        else
            parent.right = null;
    }
    // CASE 2A: Node to delete has a left child
    else if(current.getRight() == null) {
        if(current == root)
            root = current.getLeft();
        if(leftChild)
            parent.left = current.getLeft();
        else
            parent.right = current.getLeft();
```

```
        }
        // CASE 2B: Node to delete has a right child
        else if(current.getLeft() == null) {
            if(current == root)
                root = current.getRight();
            if(leftChild)
                parent.left = current.getRight();
            else
                parent.right = current.getRight();
        }
        // CASE 3: Node to delete has 2 children
        else if(current.getLeft() != null && current.getRight() != null) {
            // find min value from the right subtree
            BSTnode min = getMin(current);
            // if empty
            if(current == root)
                root = min;
            // if current is a left child
            if(leftChild)
                parent.left = min;
            // if current is a right child
            else
                parent.right = min;
            // splice the node
            min.left = current.getLeft();
        }
        return true;
    }
    private BSTnode getMin(BSTnode node) {
        // minimum node
        BSTnode min = null;
        // minimum node's parent
        BSTnode minParent = null;
        // go to right subtree
        BSTnode current = node.getRight();
        while(current!=null) {
            minParent = min;
            min = current;
            // traverse left subtree of the right subtree
            current = current.getLeft();
        }
        // splice node
        if(min!=node.getRight()) {
            minParent.left = min.getRight();
            min.right = node.getRight();
        }
        return min;
    }
    public void insert(Student s) {
        BSTnode newNode = new BSTnode(s);
        // if empty
        if(root == null) {
            root = newNode;
            return;
        }
        BSTnode current = root;
        BSTnode parent = null;
        // find parent node then attach the new node
        while(true) {
            parent = current;
            // if current key > key go left
            if(current.getStudent().getLastname().compareTo(s.getLastname()) > 0) {
                current = current.getLeft();
                // if reached a null node attach to parent as left child
                if(current == null) {
                    parent.left = newNode;
```

```
                    return;
                }
            }
            // else go right
            else {
                current = current.getRight();
                // if reached a null node attach to parent as right child
                if(current == null) {
                    parent.right = newNode;
                    return;
                }
            }
        }
    }
    // DEPTH FIRST
    public void DFT(BSTnode root, PrintWriter cursor) {
        if(root!= null) {
            // go to node with the least key
            DFT(root.getLeft(), cursor);
            // process
            cursor.println(root.getStudent());
            // go to node with higher keys
            DFT(root.getRight(), cursor);
        }
    }
    // BREADTH FIRST
    public void BFT(PrintWriter cursor) {
        // find the height of the tree
        int level, height = treeHeight(root);
        // go down the tree level by level from L to R
        for(level = 1; level <= height; level++)
            printLevel(root,level,cursor);
    }
    private int treeHeight(BSTnode root) {
        if(root == null)
            return 0;
        else {
            int leftHeight = treeHeight(root.left);
            int rightHeight = treeHeight(root.right);

            if(leftHeight > rightHeight)
                return(leftHeight+1);
            else
                return(rightHeight+1);
        }
    }
    private void printLevel(BSTnode root, int level, PrintWriter cursor) {
        if(root == null)
            return;
        if(level == 1)
            cursor.println(root.getStudent());
        else if(level > 1) {
            // gets the first node from the left
            printLevel(root.left, level-1, cursor);
            // enables to print all other nodes on the same level as the previous one
            printLevel(root.right, level-1, cursor);
        }
    }
}
```