TREEANT *the <u>not so tiny</u> image editor*

Alejandro López Martínez - 09069151K
Guillermo Ramos Santo - 08365401W

Index

1. Analysis

1. Data flow from user or external files to the system, internally in the system between every relevant component or storage, and from system to user or external files.
2. ADT specifications:
    1. Explanation of the adaptations made to the standard specification of the ADT to fit requirements. If any ADT is used more than once, its description must be repeated for every case or stated to be the same as the previous.
    2. Definition of operations of the ADT
        1. Name, arguments, and return values

2. Design

1. Design decisions / Assumptions made.
2. Diagram of the representation of the ADT in the memory of the computer (box diagrams) and explanation of every operation. Those explanations must be detailed only in case of operations that require it.
3. Use case diagrams (UML notation)
4. Class diagram
5. Explanation of classes
    1. Explanation of significant methods, including those created for operations and any other needed.

6. Explanation of the behavior of the program

3. Implementation

1. Diagram with source files and their relations
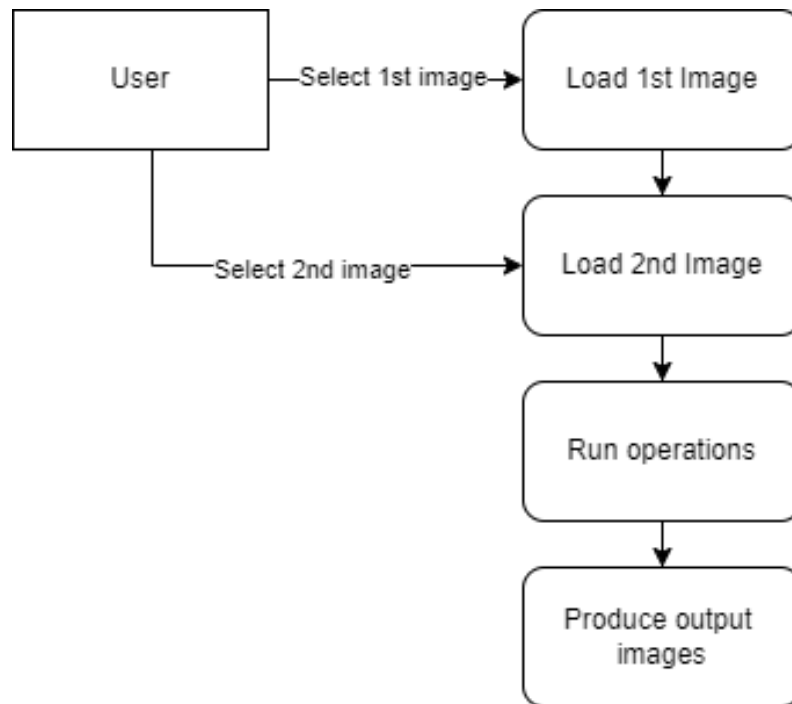2. Explanation of every difficult section of the program

4. Review

1. Running time of the solution (using big-Oh notation)
2. Possible enhancements for the solution (in terms of efficiency and others)
3. Reasoning on convenience or need of use of Dynamic DS in every case, comparing with Static DS characteristics.

5. References

1. Analysis

1. Data flow:

2. ADT specifications:

QueuePixel

1. Adaptations made to standard specification of the ADT:
Standard specification:

    SPEC QUEUE[ITEM]
            GENRES queue, item
            OPERATIONS:
                    enqueue: queue item -> queue
                    dequeue: queue -> item
                    front: queue -> item
                    makenull: queue -> queue
                    empty: queue -> boolean

    ENDSPEC

Adapted specification:

    SPEC QUEUE[ITEM]
            GENRES queue, item
            OPERATIONS:
                    enqueue: queue item -> queue
                    dequeue: queue -> item
                    empty: queue -> boolean
                    copyQueue: queue -> queue

    ENDSPEC

Adaptations:

- Item holds pixel data represented as RGBPixelXY, so the queue can efficiently manage pixel elements throughout various operations related to image processing.
- Dequeue returns an ElemPixel that contains the pixel.
- Enqueue and empty have been preserved from the standar specification.
- Front and makenull aren't used.
- Added a new operation copyQueue.

2. Operations:

1. Constructor: QueuePixel()

- Name: Constructor QueuePixel
- Arguments: None
- Return Value: None
- Description: Initializes an empty queue by setting the front and rear pointers to nullptr, indicating that there are no elements in the queue.

2. Destructor: ~QueuePixel()

- Name: Destructor ~QueuePixel
- Arguments: None
- Return Value: None
- Description: Cleans up the memory used by the queue by repeatedly dequeuing and deleting elements until the queue is empty. This prevents memory leaks by ensuring all dynamically allocated elements are removed.

3. isEmpty()

- Name: isEmpty
- Arguments: None
- Return Value: bool (returns true if the queue is empty, false otherwise)
- Description: Checks whether the queue is empty by verifying if front is nullptr. If front is nullptr, it returns true; otherwise, it returns false.

4. dequeue()

- Name: dequeue
- Arguments: None
- Return Value: ElemPixel* (pointer to the element removed from the front of the queue)
- Description: Retrieves and removes the element at the front of the queue.

5. enqueue(RGBPixelXY* ppix)

- Name: enqueue

- Arguments:
  - RGBPixelXY* ppix: A pointer to an RGBPixelXY object, representing a pixel to be added to the end of the queue.
- Return Value: None
- Description: Adds a new element to the rear of the queue.

6. copyQueue()
- Name: copyQueue
- Arguments: None
- Return Value: QueuePixel* (a pointer to the newly created copy of the queue)
- Description: Creates a deep copy of the entire queue.

SinglyLinkedList

1. Adaptations made to standard specification of the ADT:
Standard specification:

```
SPEC LIST[ITEM]
        GENRES list, item, position
        OPERATIONS:
                insert: item position list -> list
                delete: position list -> list
                locate: item list -> position
                retrieve: position list -> item
                next: position list -> item
                previous: position list -> item
                makenull: list -> list
                empty: list -> boolean
ENDSPEC
```

Adapted specification:

```
SPEC LIST[ITEM]
        GENRES list, item
        OPERATIONS:
                insert: item list -> list
                retrieve: list -> item
                empty: list -> boolean
                getHeader: list -> item
                copyList: list -> list
ENDSPEC
```

Adaptations:
- Genre position not used.
- Item holds pixel data represented as RGBPixelXY, so the list can efficiently manage pixel elements throughout various operations related to image processing.

- Insert has been modified so the pixel is inserted at the end of the list (this way you don't need position).
- Retrieve has been modified so the pixel is retrieved from the start of the list (this way you don't need position), it returns an ElemPixel which contains the pixel.
- Empty has been preserved from the standar specification.
- Delete, locate, next, previous and makenull weren't used.
- Added new operations: copyList and getHeader.

2.Operations:

1. Constructor: SinglyLinkedList()

- Name: Constructor SinglyLinkedList
- Arguments: None
- Return Value: None
- Description: Initializes an empty singly linked list by setting the header and last pointers to nullptr. These pointers indicate the start and end of the list.

2. Destructor: ~SinglyLinkedList()

- Name: Destructor ~SinglyLinkedList
- Arguments: None
- Return Value: None
- Description: Deallocates all memory used by the list to prevent memory leaks. Repeatedly retrieves and deletes elements from the list until it is empty.

3. insert(RGBPixelXY* ppix)

- Name: insert
- Arguments:
  - RGBPixelXY* ppix: A pointer to an RGBPixelXY object, representing the pixel to be inserted into the list.
- Return Value: None
- Description: Adds a new pixel element to the end of the list.

4. retrieve()

- Name: retrieve
- Arguments: None
- Return Value: ElemPixel* (pointer to the element removed from the front of the list)
- Description: Retrieves and removes the first element of the list.

5. isEmpty()

- Name: isEmpty
- Arguments: None
- Return Value: bool (true if the list is empty, false otherwise)
- Description: Checks whether the list is empty by verifying if the header is nullptr.

6. copyList()

- Name: copyList
- Arguments: None
- Return Value: SinglyLinkedList* (a pointer to the newly created copy of the list)
- Description: Creates a deep copy of the entire list.

7. getHeader()

- Name: getHeader
- Arguments: None
- Return Value: ElemPixel* (pointer to the header element of the list)
- Description: This method returns the pointer to the header element, which represents the first element of the list.

SumRGBList

1. Adaptations made to standard specification of the ADT:
Standard specification:

```
SPEC LIST[ITEM]
        GENRES list, item, position
        OPERATIONS:
                insert: item position list -> list
                delete: position list -> list
                locate: item list -> position
                retrieve: position list -> item
                next: position list -> item
                previous: position list -> item
                makenull: list -> list
                empty: list -> bool
ENDSPEC
```

Adapted specification:

```
SPEC LIST[ITEM]
        GENRES list, item
        OPERATIONS:
                insert: item list -> list
                retrieve: list -> item
                empty: list -> boolean
                getHeader: list -> item
ENDSPEC
```

Adaptations:
- Genre position not used.
- Item holds an integer represented as a sumRGB (sum or R, G and B components) value.
- Insert has been modified, so the integer is inserted at the end of the list (this way you don't need position).

- Retrieve has been modified, so the integer is retrieved from the end of the list (this way you don't need position), it returns an SumRGBListNode which contains the pixel.
- Empty has been preserved from the standar specification.
- Delete, locate, next, previous and makenull weren't used.

2.Operations:

1. Constructor: SumRGBList()

- Name: Constructor SumRGBList
- Arguments: None
- Return Value: None
- Description: Initializes an empty singly linked list by setting the header and last pointers to nullptr. These pointers indicate the start and end of the list.

2. Destructor: ~SumRGBList()

- Name: Destructor ~SumRGBList
- Arguments: None
- Return Value: None
- Description: Deallocates all memory used by the list to prevent memory leaks. Repeatedly retrieves and deletes elements from the list until it is empty.

3. insert(int value)

- Name: insert
- Arguments:
    - int value: The integer value to be inserted into the list. This value represents the sumRGB value of a pixel.
- Return Value: None
- Description: Adds a new node with the given value to the end of the list.

4. retrieve()

- Name: retrieve
- Arguments: None
- Return Value: SumRGBListNode* (pointer to the node removed from the front of the list)
- Description: Retrieves and removes the first node of the list.

5. isEmpty()

- Name: isEmpty
- Arguments: None
- Return Value: bool (true if the list is empty, false otherwise)
- Description: Checks whether the list is empty by verifying if the header is nullptr.

- Name: getHeader
- Arguments: None
- Return Value: SumRGBListNode* (pointer to the header node of the list)
- Description: This method returns the pointer to the header node, which represents the first element of the list.

BinarySearchTree

1. Adaptations made to standard specification of the ADT:
Standard specification:

```
SPEC BST[NODE]
        GENRES: node bst label
        OPERATIONS:
                search: node bst label -> boolean
                insert: node bst label -> bst
                delete: node bst label -> bst
ENDSPEC
```

Adapted specification:

```
SPEC BST[NODE]
        USES: singlylinkedlist sumrgblist pixel
        GENRES: node bst
        OPERATIONS:
                nodeHeight: node bst -> int
                biggestNode: node node int -> bst
                insertImg1Pixels: pixel bst -> bst
                insertImg2Pixels: pixel bst -> bst
                isEmpty: bst -> boolean
                makenull: node bst -> bst
                obtainSumRGBInNodes: node sumrgblist -> bst
                getRoot: bst -> node
                pixelsFromNodesWithMoreOF1:
                                        node singlylinkedlist -> bst
                pixelsFoundInNodes:
                                        node singlylinkedlist -> bst
        ENDSPEC
```

Adaptations:
- Genre label not used.
- Node holds a singly linked list of pixels with the same sumRGB value.
- The insert operation has been adapted into two methods:
  - insertImg1Pixels: This method inserts pixels into the tree based on the sumRGB of each pixel. If a node with the same sumRGB already exists, the pixel is inserted into the list of pixels in that node. If no such node exists, a new node is created. This process respects the binary search tree property.

○ insertImg2Pixels: Similar to the first method, but it only inserts pixels whose sumRGB already exists in a node. If no node for a given sumRGB exists, the pixel is discarded.
● New operations added: biggestNode, obtainSumRGBInNodes, pixelsFoundInNodes, nodeHeight, pixelsFromNodesWithMoreOF1, getRoot, isEmpty and makenull.

2. Operations:

1. Constructor: BinarySearchTree()
● Name: Constructor BinarySearchTree
● Arguments: None
● Return Value: None
● Description: Initializes a binary search tree with its root pointer set to nullptr.

2. Destructor: ~BinarySearchTree()
● Name: Destructor ~BinarySearchTree
● Arguments: None
● Return Value: None
● Description: Deallocates memory by recursively deleting all nodes in the tree using the makenull method.

3. makenull(TreeNode* nnode)
● Name: makenull
● Arguments: TreeNode* nnode:
    ○ Pointer to the node to delete.
● Return Value: None
● Description: Recursively deletes all nodes in the tree using postorder traversal.

4. nodeHeight(TreeNode* node)
● Name: nodeHeight
● Arguments:
    ○ TreeNode* node: The node to calculate the height for.
● Return Value: int (The height of the given node.)
● Description: Calculates the height of the node using postorder traversal.

5. insertImg1Pixels(RGBPixelXY* ppix)
● Name: insertImg1Pixels
● Arguments: RGBPixelXY* ppix:
    ○ Pointer to the pixel from the first image to insert.
● Return Value: None
● Description: Iteratively inserts a pixel into the tree. If a node with the same sumRGB exists, the pixel is added to its list. Otherwise, a new node is created.

6. insertImg2Pixels(RGBPixelXY* ppix)
- Name: insertImg2Pixels
- Arguments:
  - RGBPixelXY* ppix: Pointer to the pixel from the second image to insert.
- Return Value: None
- Description: Iteratively inserts a pixel into the tree if its sumRGB matches a node. Discards it otherwise.

7. isEmpty()
- Name: isEmpty
- Arguments: None
- Return Value: bool (true if the tree is empty, otherwise false.)
- Description: Checks if the tree is empty by verifying if the root is nullptr.

8. obtainSumRGBInNodes(TreeNode* nnode, SumRGBList* slist)
- Name: obtainSumRGBInNodes
- Arguments: TreeNode* nnode:
  - The current node for traversal.
  - SumRGBList* slist: Auxiliary list to store sumRGB values.
- Return Value: None
- Description: Collects sumRGB values from nodes in the tree and inserts them into a list in sorted order using inorder traversal.

9. biggestNode(TreeNode* nnode, TreeNode* &largestNode, int &largestSize)
- Name: biggestNode
- Arguments:
  - TreeNode* nnode: The current node for traversal.
  - TreeNode* &largestNode: Reference to the node with the most pixels.
  - int &largestSize: Reference to the largest pixel count.
- Return Value: None
- Description: Finds the node with the most pixels in the tree using postorder traversal and updates the references.

10. getRoot()
- Name: getRoot
- Arguments: None
- Return Value: TreeNode* (The root of the tree.)
- Description: Returns the root of the binary search tree.

11.   pixelsFromNodesWithMoreOF1(TreeNode* nnode, SinglyLinkedList* llist)
- Name: pixelsFromNodesWithMoreOF1
- Arguments:
  - TreeNode* nnode: The current node for traversal.

- ○ SinglyLinkedList* llist: Auxiliary list to collect pixels.
- Return Value: None
- Description: Collects pixels from nodes where there are more pixels with originFile 1 than originFile 2. Inserts pixels from other nodes that don't meet the criteria as black pixels.

12. pixelsFoundInNodes(TreeNode* nnode, SinglyLinkedList* llist)
- Name: pixelsFoundInNodes
- Arguments:
  - ○ TreeNode* nnode: The current node for traversal.
  - ○ SinglyLinkedList* llist: Auxiliary list to collect pixels.
- Return Value: None
- Description: Collects all pixels from nodes in the tree and inserts them into a list using inorder traversal.

BinaryBalancedTree

1. Adaptations made to standard specification of the ADT:
Standard specification:

```
SPEC BST[NODE]
        GENRES: node bst label
        OPERATIONS:
                search: node bst label -> boolean
                insert: node bst label -> bst
                delete: node bst label -> bst
ENDSPEC
```

Adapted specification:

```
SPEC BST[NODE]
        USES: singlylinkedlist pixel
        GENRES: node bst
        OPERATIONS:
                nodeHeight: node bst -> int
                biggestNode: node node int -> bst
                insertImg2Pixels: pixel bst -> bst
                isEmpty: bst -> boolean
                makenull: node bst -> bst
                getRoot: bst -> node
                pixelsFoundInNodes: node singlylinkedlist -> bst
                balance: node node -> bst
                howBalanced: node bst -> int
                leftRotate: node node -> bst
                rightRotate: node node -> bst
                cloneTree: node bst -> node
                create_balance: node bst -> bst
ENDSPEC
```

Adaptations:
- Genre label not used.
- Node holds a singly linked list of pixels with the same sumRGB value.
- The insert operation has been modified into insertImg2Pixels, which only inserts pixels whose sumRGB already exist in a node. If no node with the corresponding sumRGB exists, the pixel is discarded. This method maintains the binary search tree property.
- New operations added: biggestNode, nodeHeight, isEmpty, makenull, getRoot, pixelsFoundInNodes, balance, howBalanced, leftRotate, rightRotate, cloneTree, and create_balance.

2. Operations:

1. Constructor: BinaryBalancedTree()
- Name: Constructor BinaryBalancedTree
- Arguments: None
- Return Value: None
- Description: Initializes a binary search tree (that will be balanced) with its root pointer set to nullptr.

2. Destructor: ~BinaryBalancedTree()
- Name: Destructor ~BinaryBalancedTree
- Arguments:None
- Return Value: None
- Description: Deallocates memory by recursively deleting all nodes in the tree using the makenull method.

3. makenull(TreeNode* nnode)
- Name: makenull
- Arguments:
  - TreeNode* nnode: Pointer to the node to delete.
- Return Value: None
- Description: Recursively deletes all nodes in the tree using postorder traversal.

4. howBalanced(TreeNode* node)
- Name: howBalanced
- Arguments:
  - TreeNode* node: The node to calculate the balance factor for.
- Return Value: int (The balance factor of the node.)
- Description: Calculates the difference between the heights of the left and right subtrees of the given node.

5.balance(TreeNode* node, TreeNode* parent)
- Name: balance
- Arguments:
  - TreeNode* node: The node to start balancing from.
  - TreeNode* parent: The parent of the current node.
- Return Value: None

- Description: Balances the tree starting from the given node by performing rotations based on balance factors.

6. rightRotate(TreeNode* node, TreeNode* parent)
   - Name: rightRotate
   - Arguments:
     - TreeNode* node: The node to perform a right rotation around.
     - TreeNode* parent: The parent of the current node.
   - Return Value: None
   - Description: Performs a right rotation around the given node to rebalance the tree.

7. leftRotate(TreeNode* node, TreeNode* parent)
   - Name: leftRotate
   - Arguments:
     - TreeNode* node: The node to perform a left rotation around.
     - TreeNode* parent: The parent of the current node.
   - Return Value: None
   - Description: Performs a left rotation around the given node to rebalance the tree.

8. cloneTree(TreeNode* node)
   - Name: cloneTree
   - Arguments:
     - TreeNode* node: The root node of the tree to clone.
   - Return Value: TreeNode* (The root of the cloned tree.)
   - Description: Recursively creates a copy of the tree starting from the given node, preorder traversal is used.

9. create_balance(TreeNode* rroot)
   - Name: create_balance
   - Arguments:
     - TreeNode* rroot: The root of the tree to balance and order.
   - Return Value: None
   - Description: Clones the input tree and balances it maintaining its order.

10. nodeHeight(TreeNode* node)
    - Name: nodeHeight
    - Arguments:
      - TreeNode* node: The node to calculate the height for.
    - Return Value: int (The height of the given node.)
    - Description: Calculates the height of the node using postorder traversal.

11. isEmpty()
    - Name: isEmpty
    - Arguments: None
    - Return Value: bool (true if the tree is empty, otherwise false)

● Description: Checks if the tree is empty by verifying if the root is nullptr.

12. biggestNode(TreeNode* nnode, TreeNode* &largestNode, int &largestSize)
    ● Name: biggestNode
    ● Arguments:
        ○ TreeNode* nnode: The current node for traversal.
        ○ TreeNode* &largestNode: Reference to the node with the most pixels.
        ○ int &largestSize: Reference to the largest pixel count.
    ● Return Value: None
    ● Description: Finds the node with the most pixels in the tree using postorder traversal and updates the references.

13. getRoot()
    ● Name: getRoot
    ● Arguments: None
    ● Return Value: TreeNode* (The root of the tree.)
    ● Description: Returns the root of the binary balanced tree.

14. insertImg2Pixels(RGBPixelXY* ppix)

    ● Name: insertImg2Pixels
    ● Arguments:
        ○ RGBPixelXY* ppix: Pointer to the pixel to insert.
    ● Return Value: None
    ● Description: Iteratively inserts a pixel into the tree if its sumRGB matches a node. Discards it otherwise.

15. pixelsFoundInNodes(TreeNode* nnode, SinglyLinkedLis*t llist)

    ● Name: pixelsFoundInNodes
    ● Arguments:
        ○ TreeNode* nnode: The current node for traversal.
        ○ SinglyLinkedList* llist: Auxiliary list to collect pixels.
    ● Return Value: None
    ● Description: Collects all pixels from the nodes in the tree into a list using inorder traversal.

2. Design

1. Design decisions / Assumptions made:

- We decided to implement the insertion of pixels into the **Binary Search Tree** (T1) and the **Binary Balanced Tree** (T2) iteratively instead of recursively. This approach allowed us to better understand and control the flow of the program, making it easier to debug and observe the behavior of the insertion process.

- In the **biggestNode** method, we chose to pass some parameters by references. This decision ensures that changes made within the method are reflected outside of it. This approach also allowed us to effectively implement recursion, as it provides a clear way to propagate changes through the recursive calls.

- We have chosen to calculate the maximum depth of the tree by applying the nodeHeight method to the tree's root, which computes its total height. This value directly represents the maximum tree depth, corresponding to the total number of distinct levels in the tree.

- We decided to calculate the node with the most pixels by traversing all the pixels in the lists of all the tree nodes, instead of storing an attribute in the lists to track the number of pixels each list contains. This approach allowed us to observe how this decision affected the running time.

- Although many of our methods are void, we decided to include a return statement at the end of these methods. This choice was made to improve code readability and make the recursive logic easier to understand, even though the return statement is not strictly necessary for void methods.

- The time taken to deallocate the used memory is only considered when displaying the total execution time of phase 2. This way it doesn't impact the time measurements of other program tasks/groups.

- The time spent creating an auxiliary copy of **queuePix2** (to ensure that original data is not modified) has been included in the total time measurement for group 2. However, this time is not counted as part of the insertion process for adding all pixels from **queuePix2** into each tree.

- To handle the lists related to the **sumRGB** values in the **Binary Search Tree** (T1), we created an auxiliary class called **SumRGBList**, along with its corresponding nodes **SumRGBListNode**. This class represents a list of integers where we store the **sumRGB** values present in the nodes of the tree. Additionally, the list of **sumRGB** values that are NOT present in any node is derived through calculations based on the list containing the values that are present in the nodes.

- For **Output 1**, we consider all pixels present in the nodes where there are more pixels with the **originFile** attribute set to 1 than those with **originFile**

set to 2. This means we include pixels from both image 1 and image 2. Similarly, for the nodes that do not meet this criterion, we also include all the pixels from the node; however, these pixels will be placed as black pixels.

- For **Output 2** and **Output 3**, we also consider all the pixels from the nodes present in the tree, meaning we include pixels from both image 1 and image 2.

- For the outputs, we created a class called **SinglyLinkedList**, which is essentially a list of pixels. Depending on the output we want to generate, this list stores the pixels that will be included in the final image.

2. ADT diagrams and operation explanations:

BinarySearchTree:
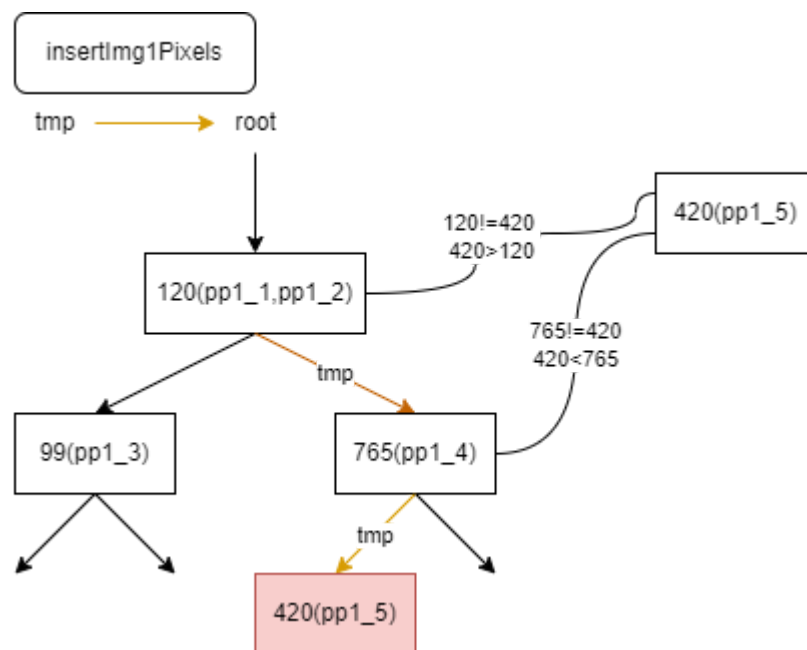
insertImg1Pixels: Inserts a pixel in a node if they share SumRGB value, if not, it traverses to the correct next node. If no node with the pixel's SumRGB value currently exists, then a new one with a new list is created, and the pixel is inserted into the node's list.
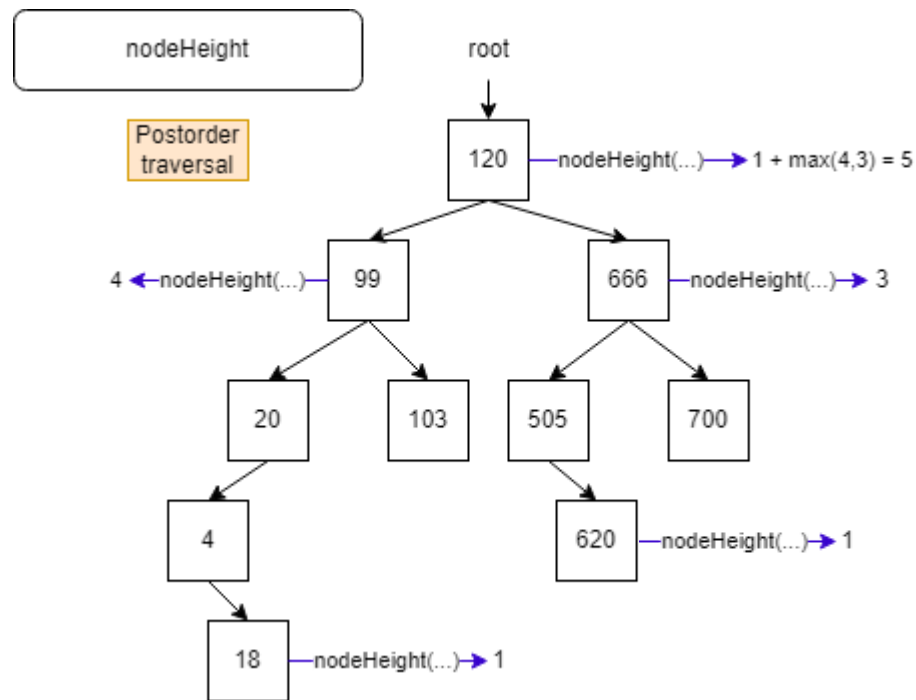
nodeHeight: This method calculates the height of the given node. Postorder traversal is used to find the height of both subtrees, and the greater of the two heights is returned. Then, 1 is added to account for the current level of the node.
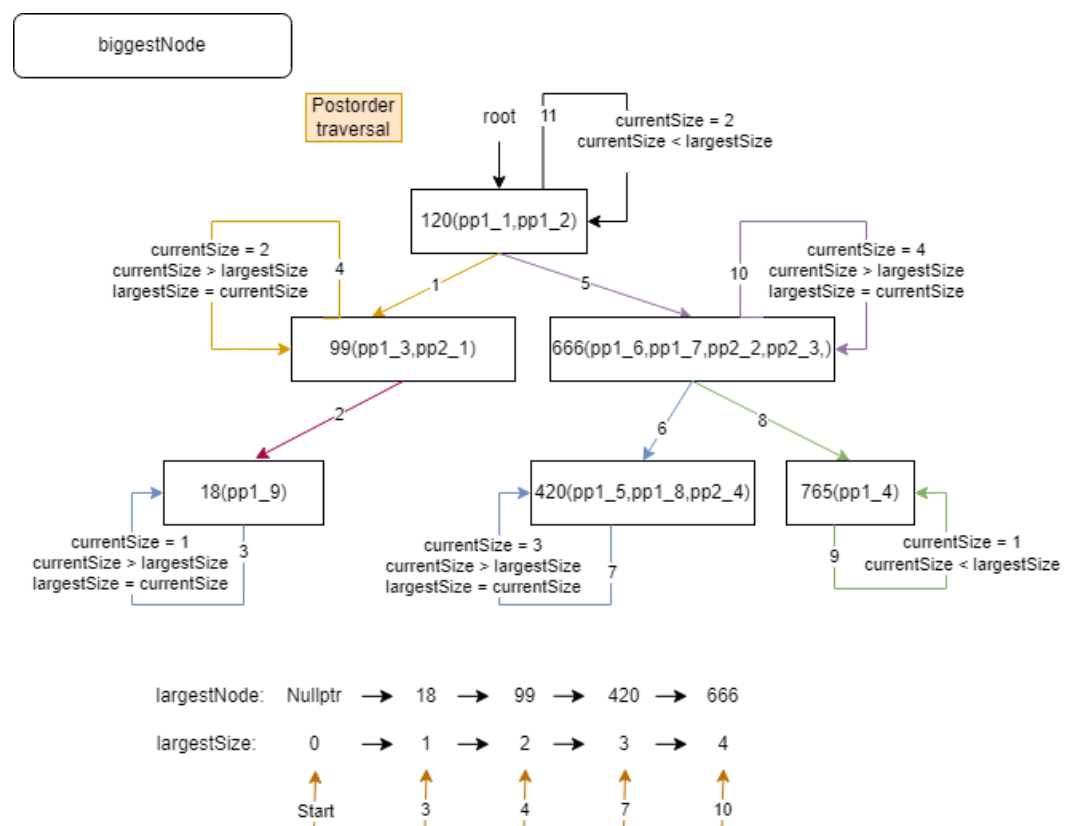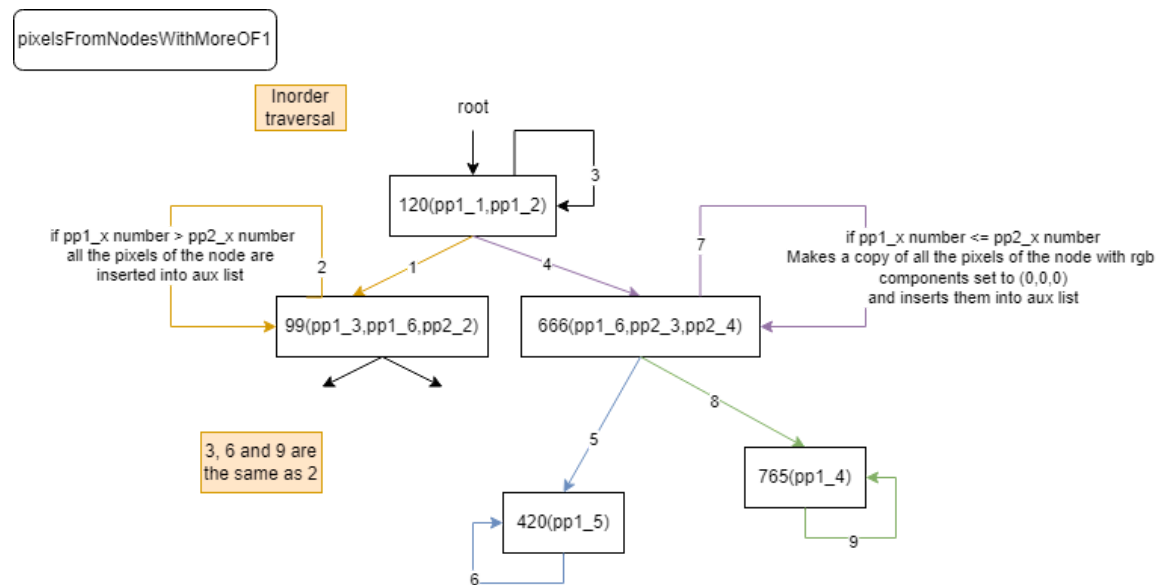

biggestNode: finds the node of the tree with the most pixels by using a postorder traversal. It compares the size of each node's list and updates the largestNode and largestSize references used when a larger node is found.
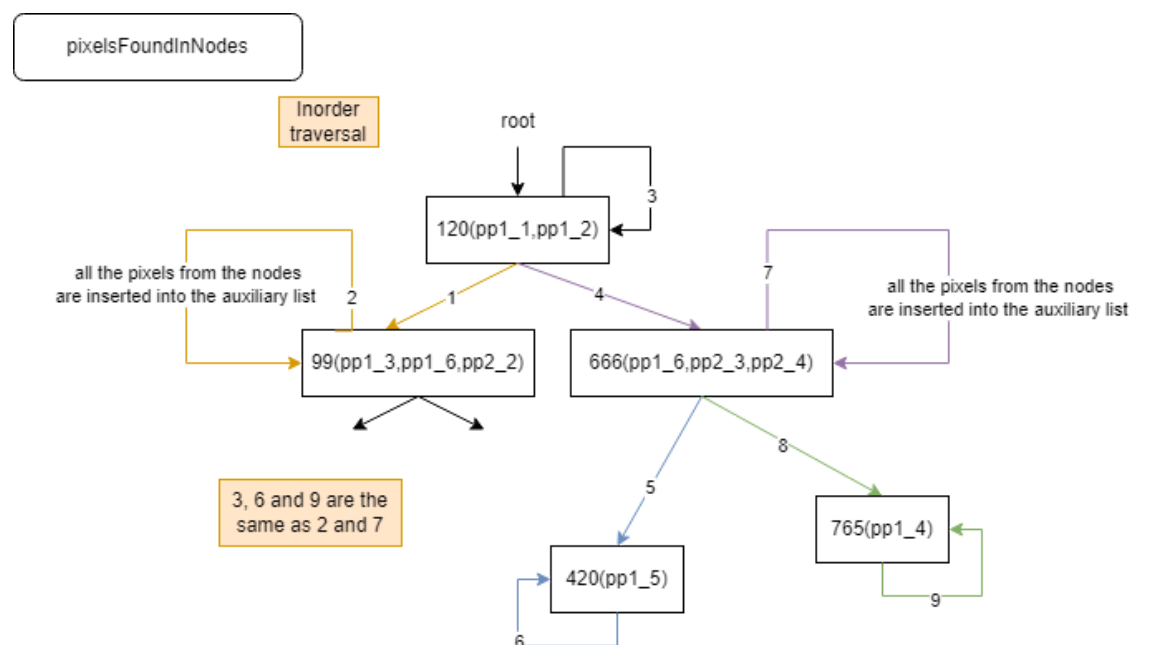
obtainSumRGBInNodes: This method collects the sumRGB values of all nodes in the tree using the inorder traversal. The values are inserted into a list (SumRGBList) in sorted order due to the properties of the binary search tree.

insertImg2Pixels: Inserts a pixel in a node if they share SumRGB value, if not, it traverses to the correct next node, when that happens in a leaf, the pixel is discarded. If no node with the pixel's SumRGB value currently exists, then the pixel is discarded.

Inorder traversal

root

120(pp1_1,pp1_2)

3

if pp1_x number > pp2_x number
all the pixels of the node are
inserted into aux list

2

1

4

7

if pp1_x number <= pp2_x number
Makes a copy of all the pixels of the node with rgb
components set to (0,0,0)
and inserts them into aux list

99(pp1_3,pp1_6,pp2_2)

666(pp1_6,pp2_3,pp2_4)

3, 6 and 9 are
the same as 2

8

5

765(pp1_4)

420(pp1_5)

9

6

pixelsFromNodesWithMoreOF1: Traverses the tree in an inorder way, for each node it counts the number of pixels from the second image and from the first image. Pixels from nodes where there are more pixels from the first image than pixels from the second image are inserted into an auxiliary list of pixels. If this condition is not satisfied, then instead of adding them unchanged to an auxiliary list they will first be copied and the copied pixels RGB values will be set to (0, 0, 0), and then added to the list.

pixelsFoundInNodes

Inorder traversal

root

120(pp1_1,pp1_2)

3

all the pixels from the nodes
are inserted into the auxiliary list

2

1

4

7

all the pixels from the nodes
are inserted into the auxiliary list

99(pp1_3,pp1_6,pp2_2)

666(pp1_6,pp2_3,pp2_4)

3, 6 and 9 are the
same as 2 and 7

8

5
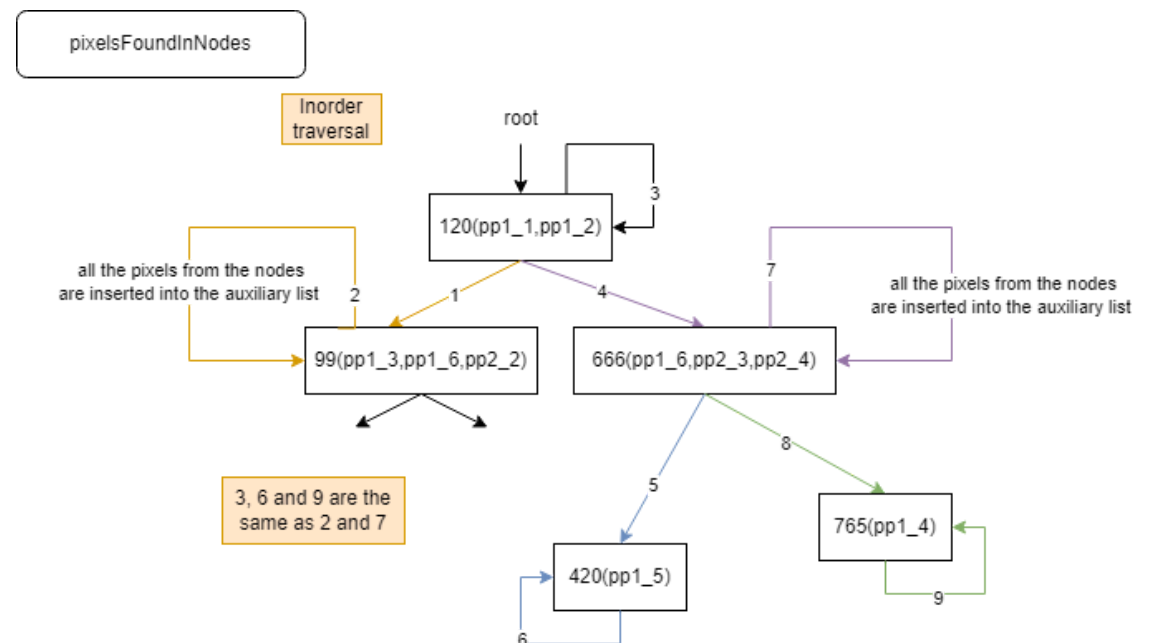
420(pp1_5)

765(pp1_4)

9

6

pixelsFoundInNodes: This method collects all the pixels from all nodes in the tree using inorder traversal and inserts them into a provided pixel list (SinglyLinkedList).
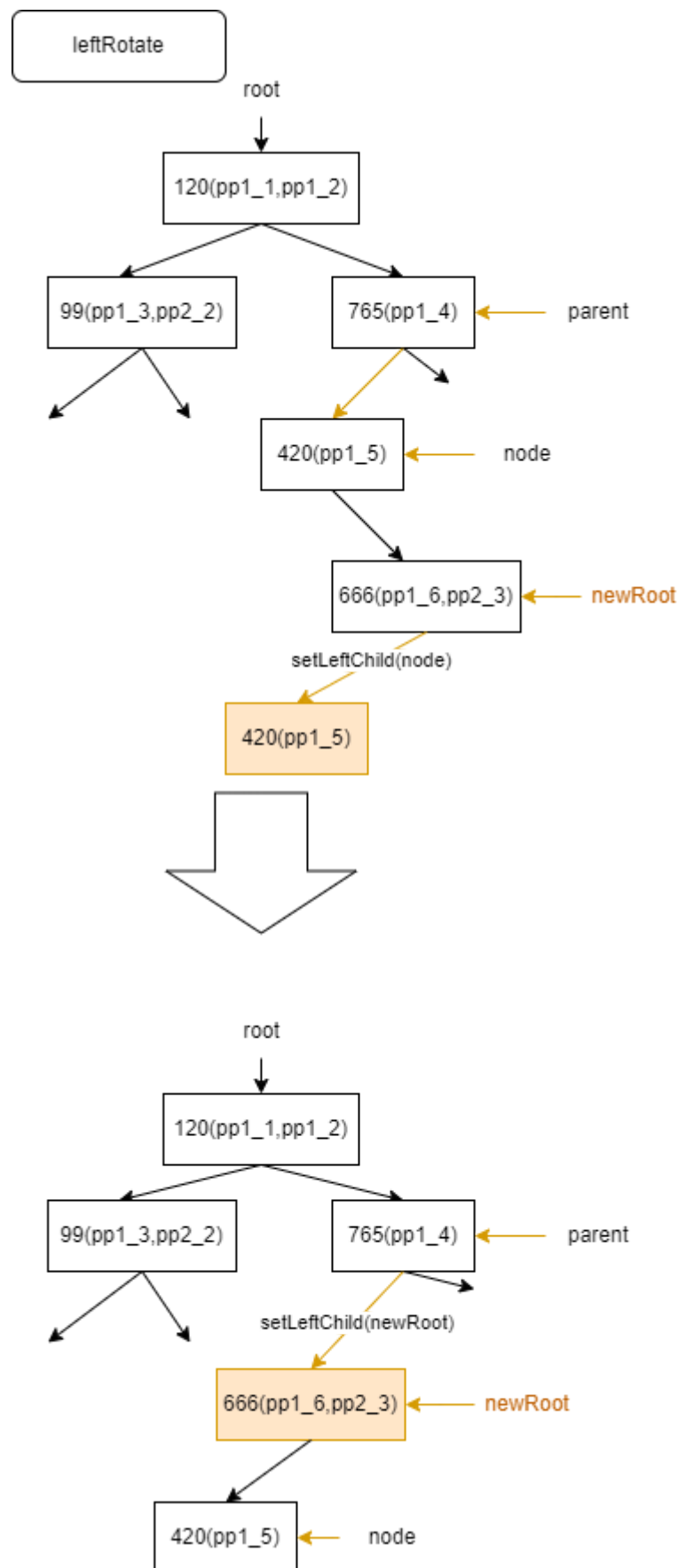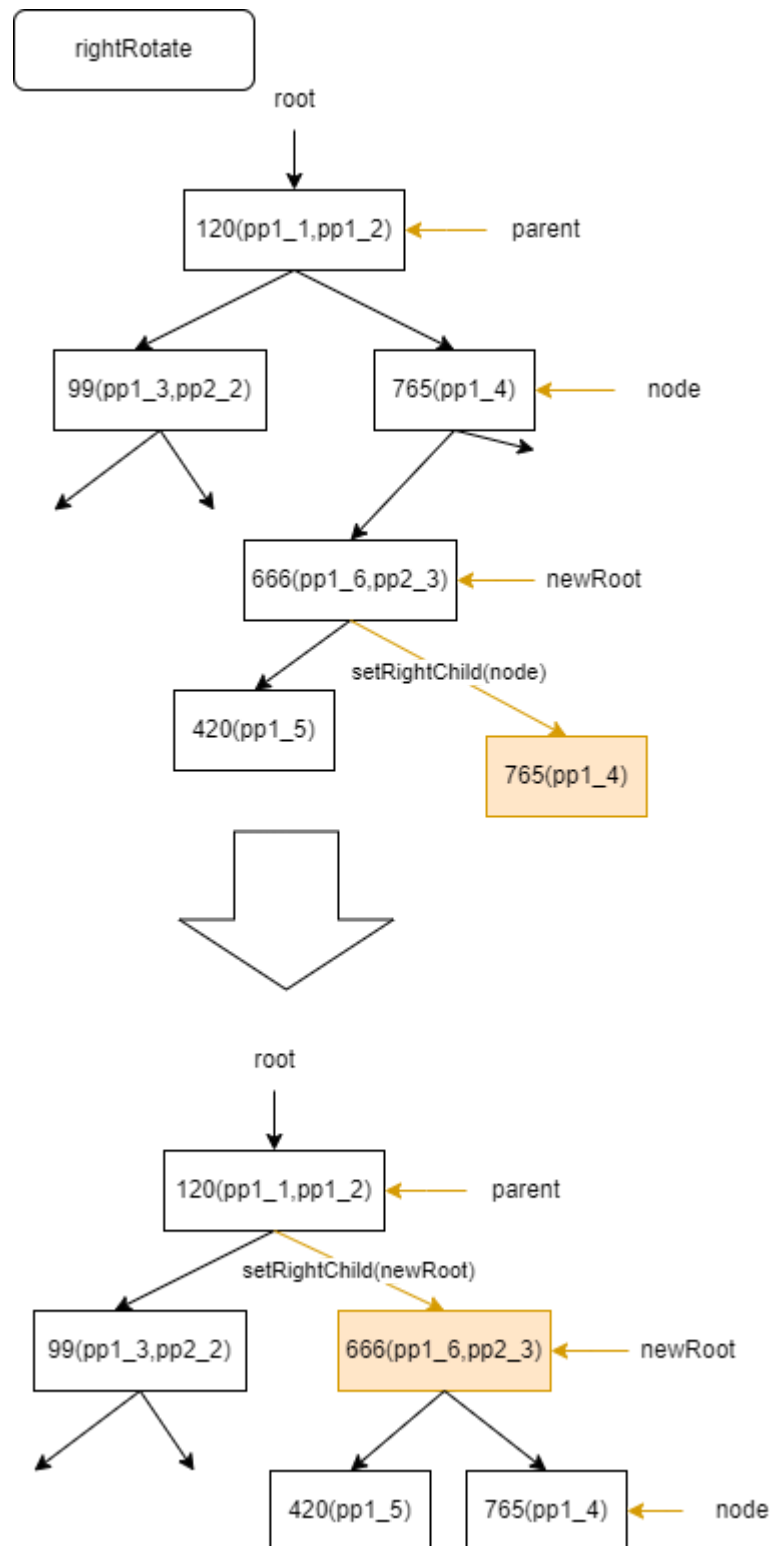
BinaryBalancedTree:
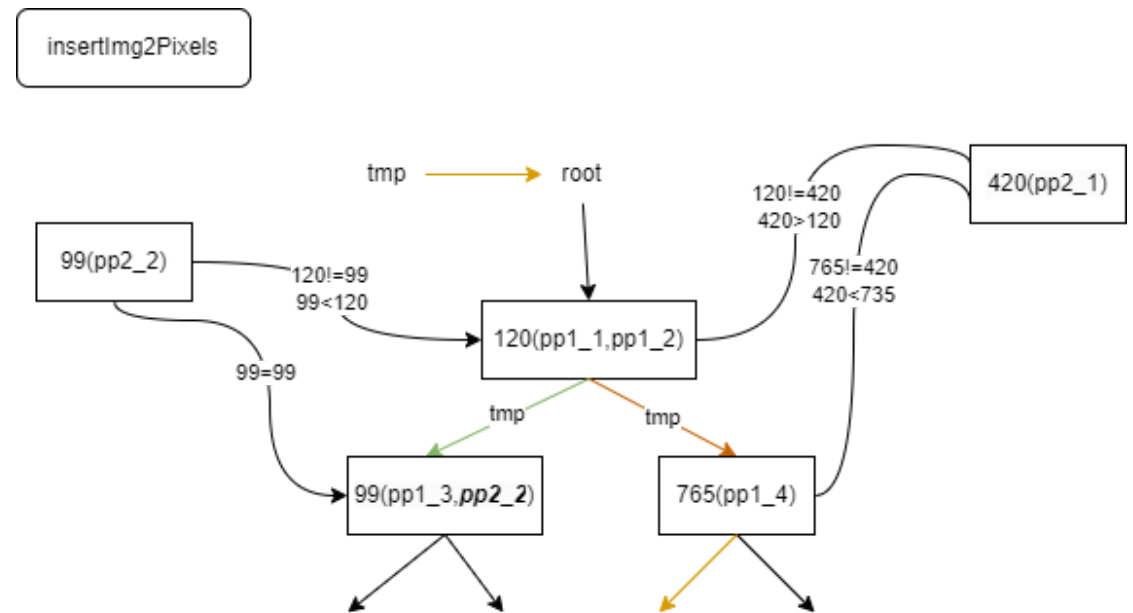
insertImg2Pixels

insertImg2Pixels: Inserts a pixel in a node if they share SumRGB value, if not, it traverses to the correct next node, when that happens in a leaf, the pixel is discarded. If no node with the pixel's SumRGB value currently exists, then the pixel is discarded.

pixelsFoundInNodes

pixelsFoundInNodes: This method collects all the pixels from all nodes in the tree using inorder traversal and inserts them into a provided pixel list (SinglyLinkedList).
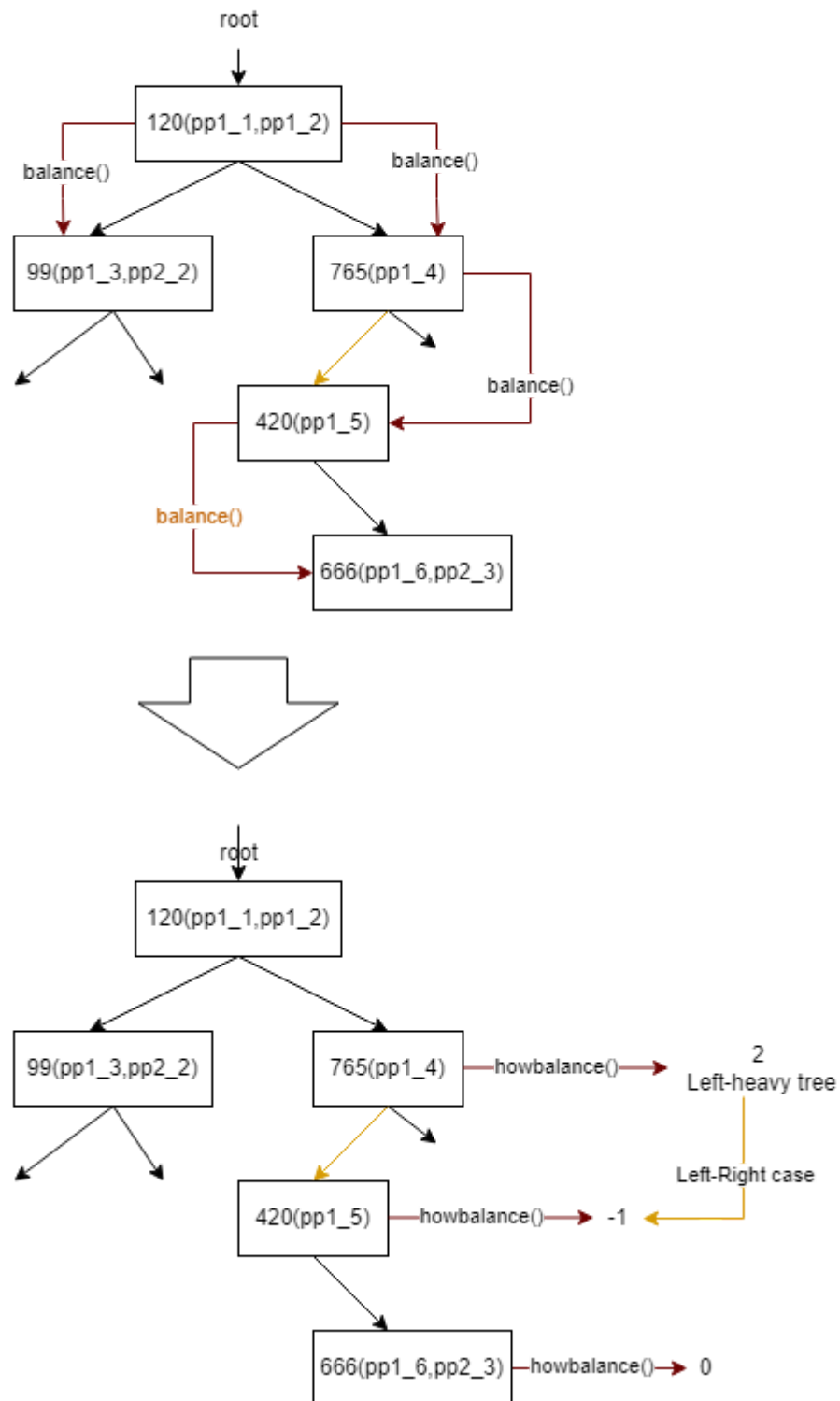
the balance diagram continues through the left and rightRotate one's for simplicity's sake

balance: It traverses the tree in a postorder way, checking the balance factor of the nodes, when a sufficient unbalanced node is found it performs the needed operations (right or leftRotate ones) in order to balance it.
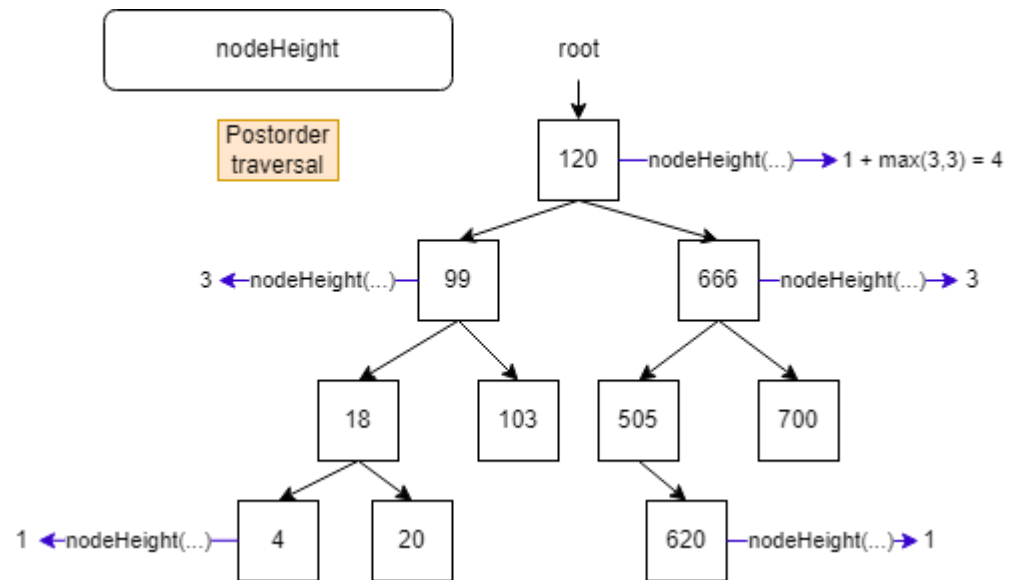
leftRotate

root

120(pp1_1,pp1_2)

99(pp1_3,pp2_2)   765(pp1_4) ← parent

420(pp1_5) ← node

666(pp1_6,pp2_3) ← newRoot

setLeftChild(node)

420(pp1_5)

root

120(pp1_1,pp1_2)

99(pp1_3,pp2_2)   765(pp1_4) ← parent

setLeftChild(newRoot)

666(pp1_6,pp2_3) ← newRoot

420(pp1_5) ← node

leftRotate: It is given a pointer to a node and its parent. Creates a pointer to the node which will be the new root of the subtree, and by using the operations [setLeftChild() and setRightChild()], balances the tree, or allows for it to be balanced in the next operation.

rightRotate

root

120(pp1_1,pp1_2) ← parent

99(pp1_3,pp2_2)   765(pp1_4) ← node

666(pp1_6,pp2_3) ← newRoot

setRightChild(node)

420(pp1_5)

765(pp1_4)

root

120(pp1_1,pp1_2) ← parent

setRightChild(newRoot)

99(pp1_3,pp2_2)   666(pp1_6,pp2_3) ← newRoot

420(pp1_5)   765(pp1_4) ← node

This would be the result from the balance method, a proper balanced binary search tree.

rightRotate: It is given a pointer to a node and its parent. Creates a pointer to the node which will be the new root of the subtree, and by using the operations [setLeftChild() and setRightChild()], balances the tree, or allows for it to be balanced in the next operation.

nodeHeight: This method calculates the height of the given node. Postorder traversal is used to find the height of both subtrees, and the greater of the two heights is returned. Then, 1 is added to account for the current level of the node.
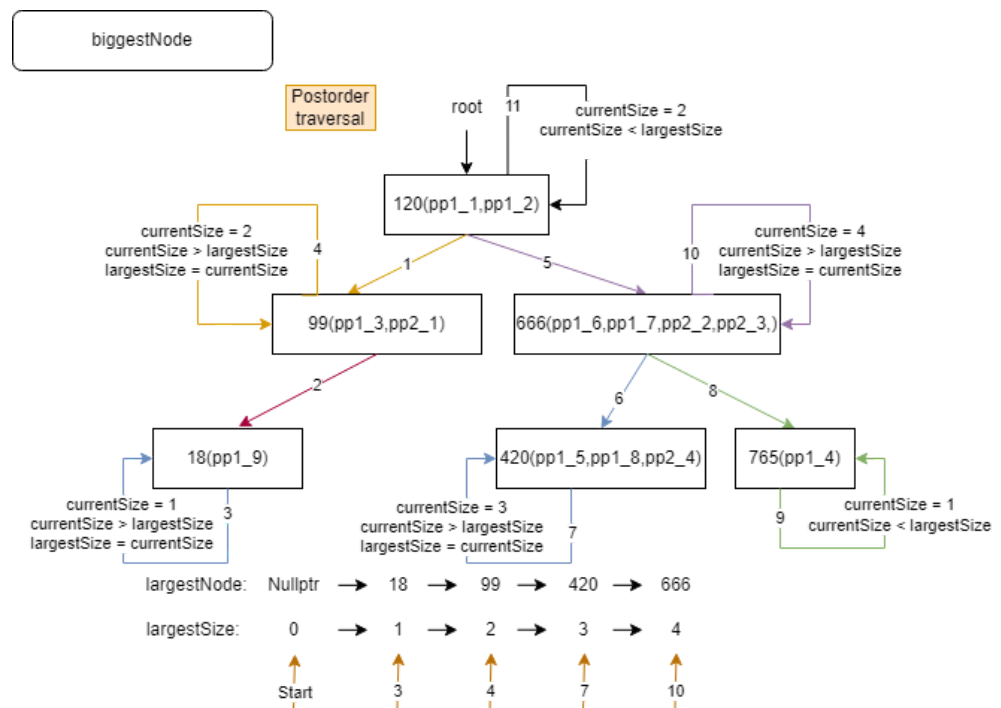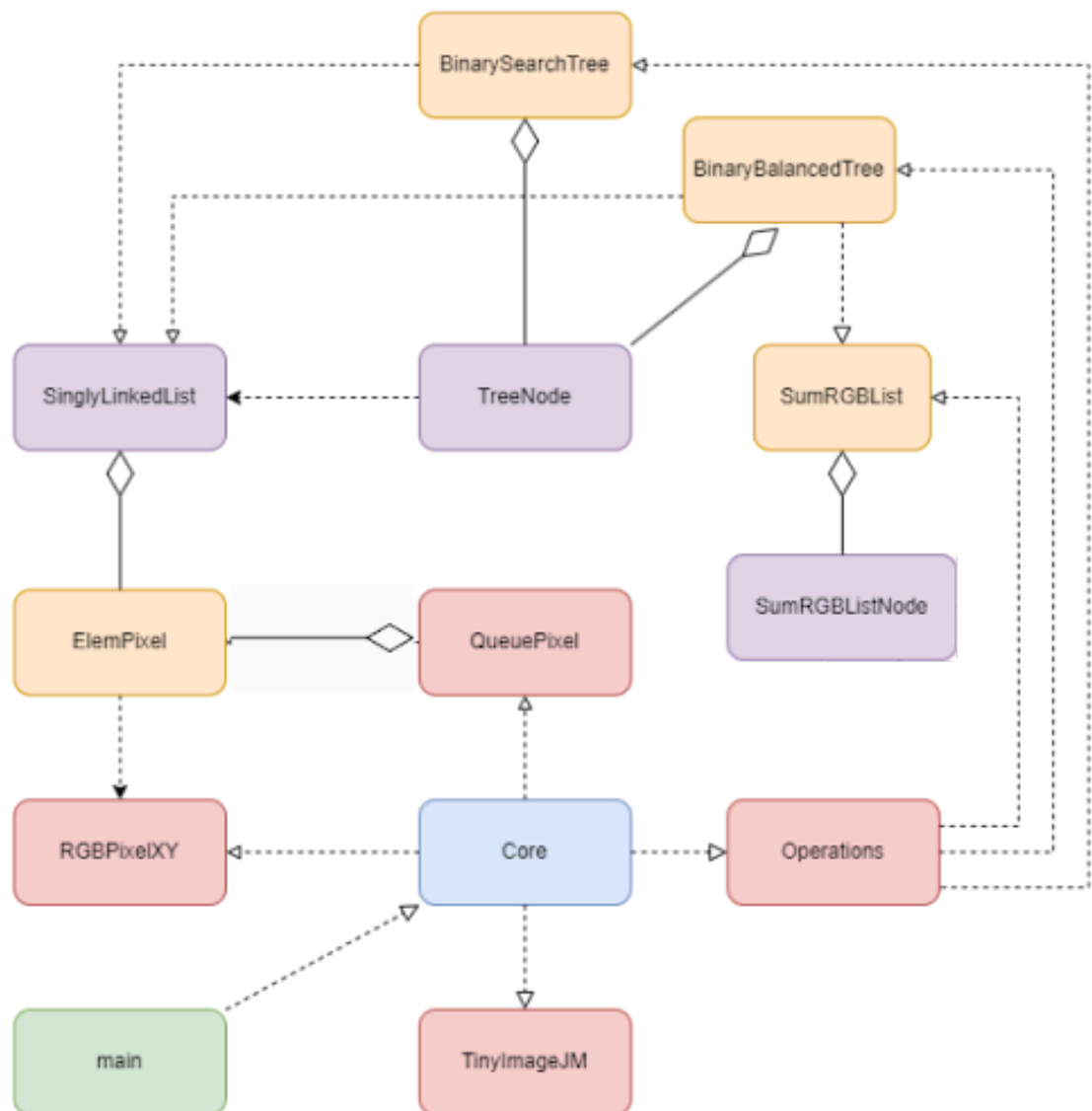

biggestNode: finds the node of the tree with the most pixels by using a postorder traversal. It compares the size of each node's list and updates the largestNode and largestSize references used when a larger node is found.

3. Use case diagram:
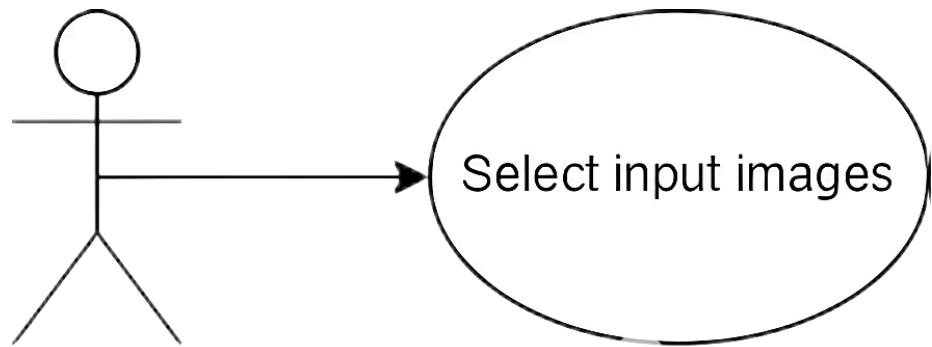
4. Class diagram:

colors added to make it easier to explain it during the presentation.

5. Classes and operations explanations:
1. Core Class

    The Core class is responsible for initializing and booting the main application.

    - **void boot()**: This method starts the main application. It handles image loading, pixel insertion into trees, statistical analysis, image output generation and memory management.

2. ElemPixel Class

    The ElemPixel class represents an element containing a pixel and pointers to the next and previous elements.

3. QueuePixel Class

    The QueuePixel class manages a queue for pixel elements.

    - **QueuePixel* copyQueue( )** This method creates a deep copy of the entire queue.

4. SinglyLinkedList Class

    The SinglyLinkedList class manages a list for pixel elements.

    - **SinglyLinkedList* copyList( )** This method creates a deep copy of the entire list.

5. SumRGBListNode Class

    The SumRGBListNode class represents a node in the SumRGBList. Each node stores a sumRGB value and a pointer to the next node in the list.

6. SumRGBList Class

    The SumRGBList class manages a singly linked list that stores nodes, each containing an integer value.

7. TreeNode Class

    The TreeNode class manages a node in a binary tree that contains a list of pixels and an associated value used for sorting the tree (sumRGB), along with pointers to the node's left and right children.

8. BinarySearchTree Class

    The BinarySearchTree class manages a binary search tree with methods for inserting pixels, calculating statistical measures, and retrieving pixel data based on specific criteria.

    - **int nodeHeight(...)** This method calculates the height of the given node. Postorder traversal is used to find the height of both subtrees, and the greater of the two heights is returned. Then, 1 is added to account for the current level of the node. This method is used for the maximum tree depth calculation.

    - **void biggestNode(...)** This method finds the node of the tree with the most pixels by using a postorder traversal. It compares the size of each node's list and updates the largestNode and largestSize references used when a larger node is found.

    - **void insertImg1Pixels (...)** This method inserts pixels from the first image into the tree. If the tree is empty, a new root is created. Otherwise, the tree is traversed iteratively and when the appropriate node is found/created (a new one or an already existing node) the pixel is inserted based on the sumRGB value.

- **void insertImg2Pixels (...)** This method inserts pixels from the second image into the tree. It does it in a similar way to insertImg1Pixels(...) but in this case the pixel will be discarded if no node with the same sumRGB exists (no matching node for the sumRGB).
- **void obtainSumRGBInNodes(...)** This method collects the sumRGB values of all nodes in the tree using the inorder traversal. The values are inserted into a list (SumRGBList) in sorted order due to the properties of the binary search tree.
- **void pixelsFromNodesWithMoreOF1(...)** This method collects all pixels from nodes where there are more pixels with originFile 1 than with originFile 2 (pixel attribute) and inserts them in a provided pixel list (SinglyLinkedList), using the inorder traversal. Pixels from nodes that don't meet these criteria are inserted into the list as black pixels.
- **void pixelsFoundInNodes(...)** This method collects all the pixels from all nodes in the tree using inorder traversal and inserts them into a provided pixel list (SinglyLinkedList).

9. BinaryBalancedTree Class

The BinaryBalancedTree class manages a binary search tree with methods for balancing and maintaining the tree structure.
- **int nodeHeight(...)** This method calculates the height of the given node. Postorder traversal is used to find the height of both subtrees and return the greater. This method is used for the maximum tree depth calculation.
- **void biggestNode(...)** This method finds the node of the tree with the most pixels by using a postorder traversal. It compares the size of each node's list and updates the largestNode and largestSize references used when a larger node is found.
- **void insertImg2Pixels(...)** This method inserts pixels from the second image into the tree. If a pixel with a sumRGB that doesn't match any existing node sumRGB arrives, then the pixel is discarded.
- **void pixelsFoundInNodes(...)** This method collects all the pixels from all nodes in the tree using inorder traversal and inserts them into a provided pixel list (SinglyLinkedList).
- **void leftRotate(...)** This method performs a left rotation around the given node to balance the tree.
- **void rightRotate(...)** This method performs a right rotation around the given node to balance the tree.
- **int howBalanced(...)** This method calculates the balance factor of the node, which is the difference of the height between the left and right subtrees.
- **void balance(...)** This method balances the tree starting from the given node. It checks the balance factor of the node and performs rotations to restore balance.
- **TreeNode* cloneTree(...)** This method creates and returns a deep copy of the tree starting from the provided node. In this

way, the entire structure of the tree is duplicated without any references to the original tree.

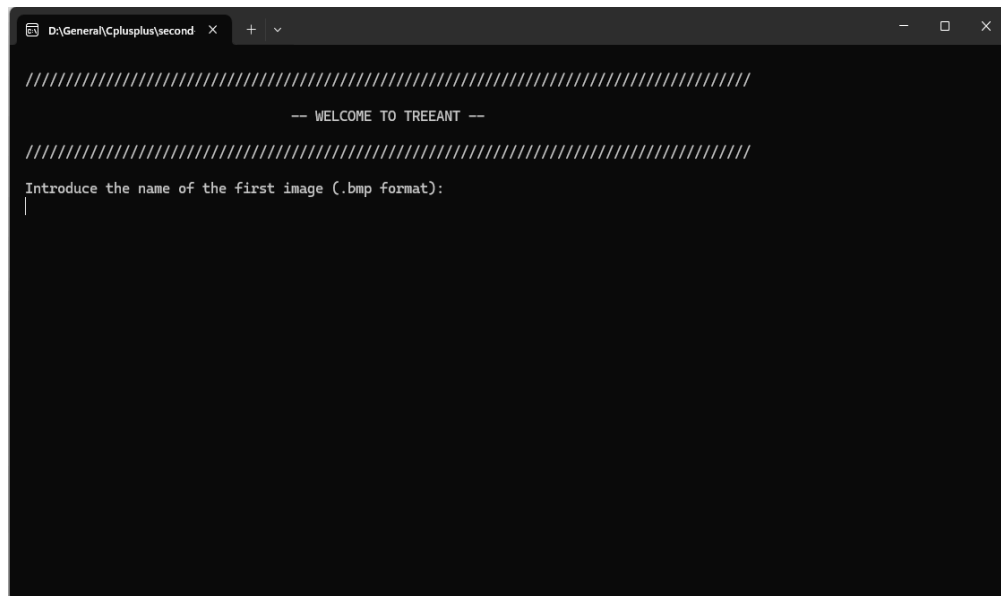- **void create_balance(...)** This method clones the input tree and balances it.

10. Operations Class

The Operations class provides methods for calculating statistical measures for trees T1 and T2, in different states. It also includes the methods for generating the output images.

- **Statistical Measures for Trees (T1 and T2)** The Operations Class includes methods for calculating statistical measures such as maximum tree depth and biggestNode for trees T1 and T2, based on various states of the trees. These states include different configurations, such as when the tree is formed using only the pixels from Image 1, or when it incorporates the pixels from Image 1 as well as those from Image 2.

- **void t1Img1Lists(...)** This method generates two lists: one for the sumRGB values (from 0 to 765) that are present in the nodes of the tree and another for those that are missing. The sumRGB values that are not present in the tree are calculated by checking gaps between consecutive values in the first list.

- **Output1(...)** This method generates the first output image based on tree T1 (BinarySearchTree), containing all pixels from nodes where there more pixels with originFile 1 than originFile 2, and containing all pixels from other nodes that doesn't meet this criteria but set to black color The image will have the same dimensions as the first image introduced in the program.

- **Output2(...)** This method generates the second output image based on tree T1 (BinarySearchTree), containing all pixels from its nodes. However, if a pixel is missing from both input images at a specific coordinate (no pixel in the tree corresponds to that coordinate), the output image will display a black pixel at that position. The image will have the same dimensions as the second image introduced in the program.
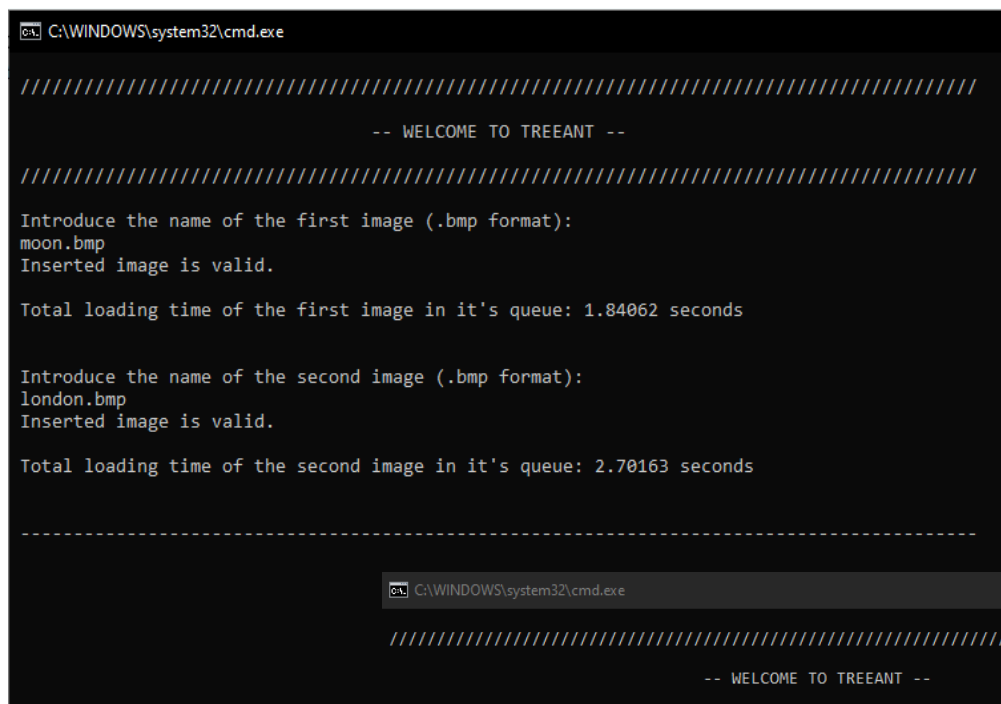
- **Output3(...)** This method generates the second output image based on tree T2 (BinaryBalancedTree), containing all pixels from its nodes. However, if a pixel is missing from both input images at a specific coordinate (no pixel in the tree corresponds to that coordinate), the output image will display a black pixel at that position. The image will have the same dimensions as the second image introduced in the program.

6. Program behavior:

Having launched Treeant the user is greeted and asked to enter the first image name.

Provided that we entered a valid image (.bmp included in the name), we will be shown the time that it took to load and asked to load the second one. If valid, Treeant will continue with its operation, if not the program will close.
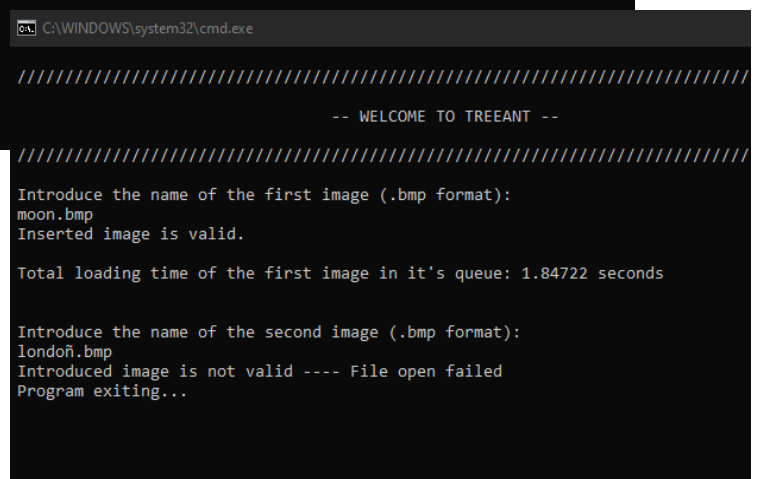
At this point, no further user interaction is required. After that, the program will display the time taken to load the first image into the first tree (T1), along with statistical measures such as the maximum tree depth and the node containing the largest number of pixels.

The program will then generate and display a list of values from 0 to 765 (sumRGB values) that correspond to the nodes created in the tree. It will also show the time taken to collect these values and generate the list.

```
C:\WINDOWS\system32\cmd.exe                                                    —    □    X

 Total loading time of the first image in the Binary Search Tree (T1): 3.74066 seconds


 ///////////////////////////////////////////////////////////////////////////////////////

                 TREE T1 WITH IMAGE 1 STATISTICAL MEASURES

 ///////////////////////////////////////////////////////////////////////////////////////


 Maximum tree depth (T1): 39

 Total time for obtaining the maximum tree depth: 0.0009973 seconds


 The node with the largest number of pixels is the one associated with the rgb sum: 681, with 97955 pixels
 .

 Total time for obtaining the node with the most pixels: 1.57879 seconds


 Values from 0 to 765 that have a node in tree (T1):
 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22
 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 |
 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |
 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 8
 6 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 1
 06 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 |
 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141
 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 1
 59 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 | 176 |
 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 | 192 | 193 | 194
 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 | 211 | 2
 12 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 |
 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247
 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 | 261 | 262 | 263 | 264 | 2
 65 | 266 | 267 | 268 | 269 | 270 | 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 | 281 | 282 |
 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 | 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300
 | 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 | 311 | 312 | 313 | 314 | 315 | 316 | 317 | 3
 18 | 319 | 320 | 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 | 331 | 332 | 333 | 334 | 335 |
 336 | 337 | 338 | 339 | 340 | 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 | 351 | 352 | 353
 | 354 | 355 | 356 | 357 | 358 | 359 | 360 | 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 | 3
 71 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 | 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 |
 389 | 390 | 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 | 401 | 402 | 403 | 404 | 405 | 406
 | 407 | 408 | 409 | 410 | 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 | 421 | 422 | 423 | 4
 24 | 425 | 426 | 427 | 428 | 429 | 430 | 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 | 441 |
 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 | 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459
 | 460 | 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 | 471 | 472 | 473 | 474 | 475 | 476 | 4
 77 | 478 | 479 | 480 | 481 | 482 | 483 | 484 | 485 | 486 | 487 | 488 | 489 | 490 | 491 | 492 | 493 | 494 |
 495 | 496 | 497 | 498 | 499 | 500 | 501 | 502 | 503 | 504 | 505 | 506 | 507 | 508 | 509 | 510 | 511 | 512
 | 513 | 514 | 515 | 516 | 517 | 518 | 519 | 520 | 521 | 522 | 523 | 524 | 525 | 526 | 527 | 528 | 529 | 5
 30 | 531 | 532 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 | 541 | 542 | 543 | 544 | 545 | 546 | 547 |
 548 | 549 | 550 | 551 | 552 | 553 | 554 | 555 | 556 | 557 | 558 | 559 | 560 | 561 | 562 | 563 | 564 | 565
 | 566 | 567 | 568 | 569 | 570 | 571 | 572 | 573 | 574 | 575 | 576 | 577 | 578 | 579 | 580 | 581 | 582 | 5
 83 | 584 | 585 | 586 | 587 | 588 | 589 | 590 | 591 | 592 | 593 | 594 | 595 | 596 | 597 | 598 | 599 | 600 |
 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 | 611 | 612 | 613 | 614 | 615 | 616 | 617 | 618
 | 619 | 620 | 621 | 622 | 623 | 624 | 625 | 626 | 627 | 628 | 629 | 630 | 631 | 632 | 633 | 634 | 635 | 6
 36 | 637 | 638 | 639 | 640 | 641 | 642 | 643 | 644 | 645 | 646 | 647 | 648 | 649 | 650 | 651 | 652 | 653 |
 654 | 655 | 656 | 657 | 658 | 659 | 660 | 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 | 669 | 670 | 671
 | 672 | 673 | 674 | 675 | 676 | 677 | 678 | 679 | 680 | 681 | 682 | 683 | 684 | 685 | 686 | 687 | 688 | 6
 89 | 690 | 691 | 692 | 693 | 694 | 695 | 696 | 697 | 698 | 699 | 700 | 701 | 702 | 703 | 704 | 705 | 706 |
 707 | 708 | 709 | 710 | 711 | 712 | 713 | 714 | 715 | 716 | 717 | 718 | 719 | 720 | 721 | 722 | 723 | 724
 | 725 | 726 | 727 | 728 | 729 | 730 | 731 | 732 | 733 | 734 | 735 | 736 | 737 | 738 | 739 | 740 | 741 | 7
 42 | 743 | 744 | 745 | 746 | 747 | 748 | 749 | 750 | 751 | 752 | 753 | 754 | 755 | 756 | 757 | 758 | 759 |
 760 | 761 | 762 | 763 | 764 | 765

 Total time for obtaining sumRGB values that have a node in the tree: 0.0718084 seconds
```

Followed by a list of values from 0 to 765 that do NOT have a node in the tree, and the time taken to generate this list.

The program will then show the time taken to convert tree T1 into the binary balanced tree (T2) using the pixels from the first image, accompanied by relevant statistical measures.

Following this, the second image will be loaded into tree T1, and the corresponding statistical measures will be displayed.

The next step involves loading the pixels from the second image (whose sumRGB values already had a node created in the tree) into the binary balanced tree (T2) displaying the time taken in the task.

```
Total time for obtaining sumRGB values that doesn't have a node in the tree: 0 seconds


--------------------------------------------------------------------------------

Total loading time of transforming T1 with the first image into a Binary Balanced Tree (T2): 4.68514 seco
nds


////////////////////////////////////////////////////////////////////////////////
                TREE T2 WITH IMAGE 1 STATISTICAL MEASURES
////////////////////////////////////////////////////////////////////////////////

Maximum tree depth (T2): 14

Total time for obtaining the maximum tree depth: 0 seconds

The node with the largest number of pixels is the one associated with the rgb sum: 681, with 97955 pixels
.

Total time for obtaining the node with the most pixels: 0.392994 seconds

Total time for performing group 1 operations: 10.4794 seconds


--------------------------------------------------------------------------------

Total loading time of the second image in the Binary Search Tree (T1): 5.30234 seconds


////////////////////////////////////////////////////////////////////////////////
                TREE T1 WITH IMAGES 1 AND 2 STATISTICAL MEASURES
////////////////////////////////////////////////////////////////////////////////

Maximum tree depth (T1): 39
Total time for obtaining the maximum tree depth: 0 seconds

The node with the largest number of pixels is the one associated with the rgb sum: 757, with 170825 pixel
s.
Total time for obtaining the node with the most pixels: 3.95573 seconds


--------------------------------------------------------------------------------

Total loading time of the second image in the Binary Balanced Tree (T2): 4.62719 seconds
```

After this, the program will display the statistical measures for tree T2.

Finally, the program will generate the three output images. Before finishing, it will display a message on the screen indicating that it is deallocating the memory used. Once this is complete, the program will display a final message to inform the user that the process has finished.

```
C:\WINDOWS\system32\cmd.exe                                              —    □    ×

/////////////////////////////////////////////////////////////////////////////////

            TREE T2 WITH IMAGES 1 AND 2 STATISTICAL MEASURES

/////////////////////////////////////////////////////////////////////////////////


Maximum tree depth (T2): 14

Total time for obtaining the maximum tree depth: 0 seconds


The node with the largest number of pixels is the one associated with the rgb sum: 757, with 170825 pixel
s.

Total time for obtaining the node with the most pixels: 2.68111 seconds


Total time for performing group 2 operations: 19.538 seconds


/////////////////////////////////////////////////////////////////////////////////

                        GENERATING OUTPUT IMAGES

/////////////////////////////////////////////////////////////////////////////////


Image: Output1.bmp generated succesfully.

Total time for obtaining first output image: 16.6062 seconds


Image: Output2.bmp generated succesfully.

Total time for obtaining second output image: 10.1615 seconds


Image: Output3.bmp generated succesfully.

Total time for obtaining third output image: 7.822 seconds


Total time for performing group 3 operations: 39.8863 seconds



Deallocating memory...


Total phase 2 time: 80.6927 seconds


/////////////////////////////////////////////////////////////////////////////////

Finished processing, check output folder. You may close the program now.
```
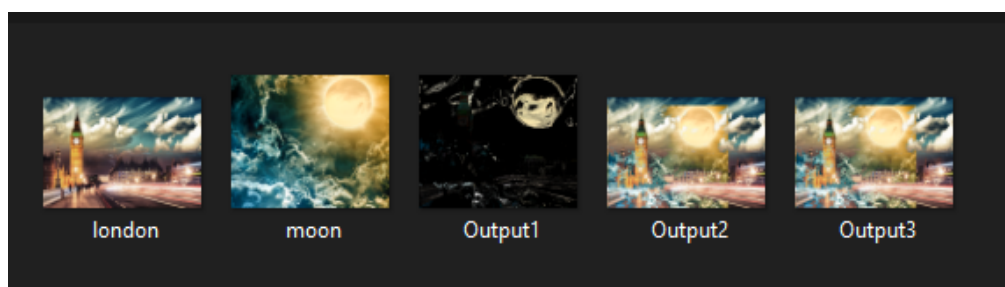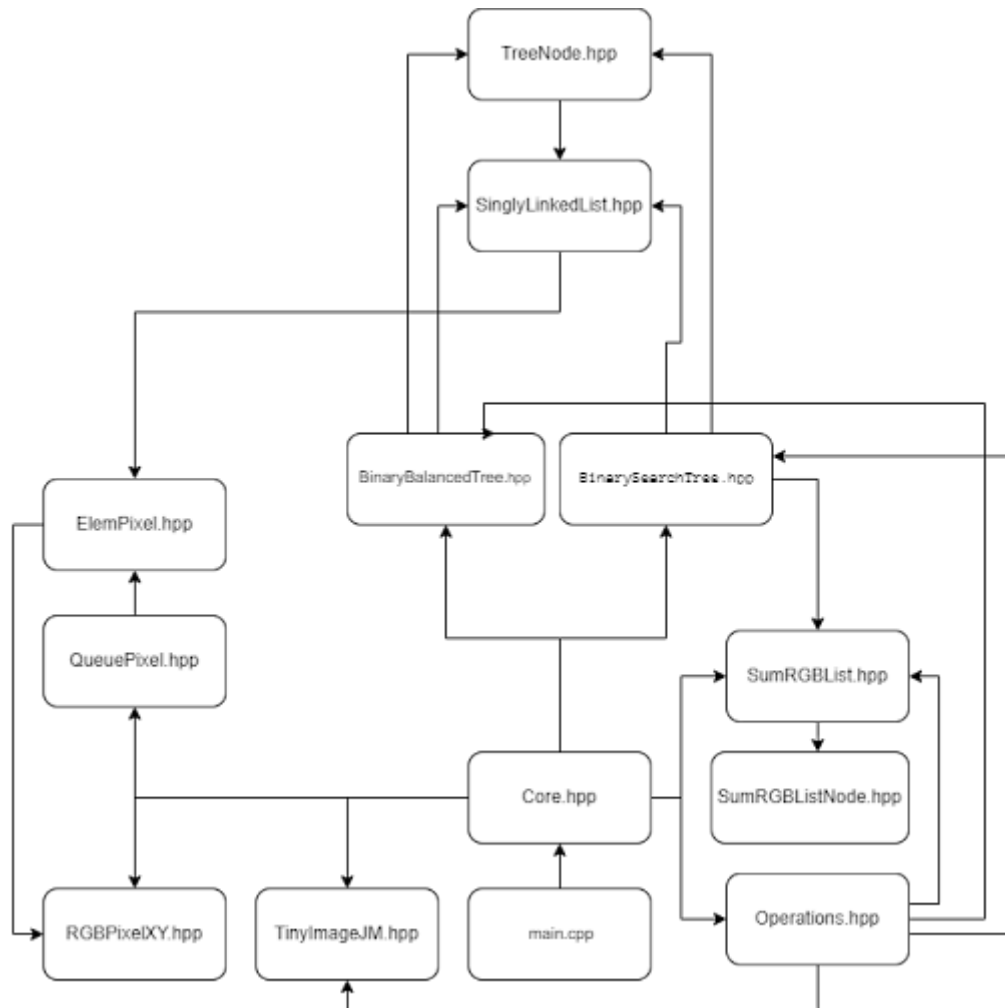
3. Implementation

1. Diagram with source files and their relations (smth —<<includes>> → smth):

every .hpp file has a .cpp file with the same name that includes it.

2. Difficult parts explanations:

One of the most challenging parts of the project was implementing the balancing for the Binary Balanced Tree.

Managing memory was a significant challenge, especially when dealing with large images. Both trees (Binary Search Tree and Binary Balanced Tree) need to store pixel data, and inserting large images with thousands of pixels led to high memory usage. We had to take special care to avoid unnecessary memory allocations and ensure that resources were deallocated properly (taking care to avoid double memory deallocation).

Generating the output images based on specific criteria for each of the images was complex. This required traversing the trees and filtering pixel data correctly (according to some criteria) before assembling the output image.

4. Review

1. Running time:

   1. Loading first image into queue Phase Efficiency
      Scope: Method Level
      Analyzed:

      - Pixel queue creation: $queuePixel$ object $\rightarrow$ O(1)

      - Validation of the image format $isValid$ check $\rightarrow$ O(1)

      - Pixel Enqueueing: enqueuing each pixel into $queuePixel$ takes $O(N_1)$ time, as it iterates over each pixel in the image to retrieve and enqueue it $\rightarrow O(N_1)$

      Running Time: $O(N_1)$ *N_1 equals the number of pixels in the first image.*

   2. Loading first image into T1 Efficiency
      Scope: Method level
      Analyzed:
      - BinarySearchTree creation: $T1$ object $\rightarrow$ O(1)

      - Pixel insertion: inserting each pixel into $T1$ implies iterating over every pixel $O(N_1)$ and for each pixel we will traverse the tree to insert it in the correct node, which in the worst case the tree will be completely unbalanced, and every pixel will have a different sumRGB value, so practically it will be a list O(D) D being depth which in the worst case will be the same as the number of nodes M. $\rightarrow O(N_1 * M)$

      Running Time: $O(N_1 * M)$ *N equals the number of pixels in image and M equals the number of nodes*

   3. Taking statistical measures of T1 Efficiency
      Scope: Method level
      Analyzed:
      -Calculating the maximum depth of $T1$: in order to perform this operation, in the worst case we iterate through every node so $\rightarrow$ O(M), being M the number of nodes.

      -Calculating the biggest node of $T1$: We need to traverse the tree and count every pixel. For each node in the tree, we traverse its list of pixels. Since the tree contains a total of N_1 pixels and the traversal of all nodes contributes O(M), where M is the number of nodes. The overall time complexity is $O(M + N_1)$ and because $M \leq N_1 \rightarrow O(N_1)$

      Running Time: $O(N_1)$ *N_1 the number of pixels in the first image.*

4. t1ImgList Efficiency

    Scope: Method level

    Analyzed:

        -Introducing every SumRGB present in T1 (via the method obtainSumRGBInNodes) $O(M)$, being M the number of nodes in T1. → $O(M)$

        - Iterating through the list in order to print the values. → $O(M)$

        - Iterate through the list in order to find "gaps" in the SumRGB values. →$O(M)$

    Running Time: $O(M + M + M)$ → $O(M)$ *Where M is the total number of nodes in T1*

5. Loading first image into T2 Efficiency

    Scope: Method level

    Analyzed:

        - BinaryBalancedTree creation → *T2*:

            The creation of *T2* implies:

                - Cloning *T1.*

                The process involves traversing the entire tree once. At each node, a copy of the pixel list is created. Since we are copying every pixel in the tree, the total number of pixels across all nodes is N_1. As a result, for each node, we perform an operation proportional to the number of pixels stored in that node. Since this operation is repeated for each node in the tree, the overall time complexity depends on both the number of nodes in the tree (M) and the total number of pixels (N_1) distributed across the nodes. →$O(M + N_1$), as we traverse the nodes once and copy all the pixels in the tree and because M ≤ N_1 → $O(N_1)$

                - Balancing.

                The balance method processes each node in the tree exactly once $O(M)$. For each node, it performs an $O(D)$ operation via the howBalance method, where D is the depth of the tree. In the worst case, for a completely unbalanced binary tree, the depth of the tree can be proportional to the number of nodes in the tree. This results in a time

complexity of: $O(M * D) \rightarrow O(M^2)$. This happens because howBalanced recalculates the height of each of the tree nodes.

Running Time: $O(N_1 + M^2)$ *M equals the number of Nodes in T1 and N_1 the number of pixels in the first image.*

6. Taking statistical measures of T2 Efficiency
Scope: Method level
Analyzed:

- Calculating the maximum depth of *T2*: in order to perform this operation, in the worst case we iterate through every node so $\rightarrow O(M)$

- Calculating the biggest node of *T2*: We need to traverse the tree and count every pixel. For each node in the tree, we traverse its list of pixels. Since the tree contains a total of N_1 pixels and the traversal of all nodes contributes $O(M)$, where M is the number of nodes. The overall time complexity is $O(M + N_1)$ and because $M \leq N_1 \rightarrow O(N_1)$

Running Time: $O(N_1)$ *N_1 the number of pixels in each node.*

7. Loading second image into queue Phase Efficiency
Scope: Method level
Analyzed:

- Pixel queue creation: q*ueuePixel* object $\rightarrow O(1)$

- Validation of the image format *isValid* check $\rightarrow O(1)$

- Pixel Enqueueing: enqueuing each pixel into q*ueuePixel* takes $O(N)$ time, as it iterates over each pixel in the image to retrieve and enqueue it $\rightarrow O(N_2)$

Running Time: $O(N_2)$ *N_2 equals the number of pixels in the second image.*

8. Loading second image into T1 Efficiency
Scope: Method level
Analyzed:

- Pixel insertion: inserting each pixel into nodes of the same SumRGB in *T1* implies iterating over every pixel $O(N_2)$ and for each pixel we will traverse the tree to insert it in the correct node, which in the worst case the tree will be completely unbalanced, so practically it will be a list $O(D)$ being D the depth of the tree which can be proportional to the number of nodes M. $\rightarrow O(N_2 * M)$

Running Time: $O(N_2 * M)$ *N_2 equals number of pixels in second image and M the number of nodes in T1*

9. Taking statistical measures of T1 Efficiency

Scope: Method level

Analyzed:

- Calculating the maximum depth of *T1*: in order to perform this operation, in the worst case we iterate through every node so → $O(M)$

- Calculating the biggest node of *T1*: We need to traverse the tree and count every pixel. For each node in the tree, we traverse its list of pixels. Since the tree contains a total of N_{1+2} pixels and the traversal of all nodes contributes $O(M)$, where M is the number of nodes. The overall time complexity is $O(M + N_{1+2})$ and because $M ≤ N_{1+2}$ → $O(N_{1+2})$

Running Time: $O(N_{1+2})$ *N_{1+2} equals the number of pixels in the first image plus the second image's pixels, which had an already existing equal SumRGB node when the insertion was done.*

10. Loading second image into T2 Efficiency

Scope: Method level

Analyzed:

- Pixel insertion: as the tree is balanced, the cost of traversing the tree will be $\log(M)$ (tree structure isn't modified after each insertion), being M the number of nodes. Also, we will run the insertion method the same number of times as pixels (N_2) in the second image exist, therefore →$O(N_2 \log(M))$

Running Time: $O(N_2 \log(M))$ *M equals the number of Nodes in T2 and N_2 the number of pixels in the second image.*

11. Taking statistical measures of T2 Efficiency

Scope: Method level

Analyzed:

- Calculating the maximum depth of *T2*: in order to perform this operation, in the worst case we iterate through every node so → $O(M)$

- Calculating the biggest node of *T2*: We need to traverse the tree and count every pixel. For each node in the tree, we traverse its list of pixels. Since the tree contains a total of N_{1+2} pixels and the traversal of all nodes contributes $O(M)$, where M is the number of nodes. The overall time complexity is $O(M + N_{1+2})$ and because $M ≤ N_{1+2}$ → $O(N_{1+2})$

Running Time: $O(N_{1+2})$ *N_{1+2} *equals the number of pixels in the first image plus the second image's pixels, which had an already existing equal SumRGB node when the insertion was done.*

12. Generating first output
Scope: Method level
Analyzed:

- Singly linked creation: SinglyLinkedList **object** → O(1)

- Call to t1 → pixelsFromNodesWithMoreOF1():
We need to traverse the entire binary tree and process each node. The function performs an inorder traversal, so it visits each node once, resulting in a time complexity of O(M), where M is the number of nodes. For each node, we first iterate through its pixel list to count the number of pixels with originFile 1 and 2, which takes $O(N_{1+2})$ for the total number of pixels. After counting, we traverse the list again to insert the pixels into the auxiliary list based on the criteria (either the original pixels or black pixels), which also takes $O(N_{1+2})$. Since each node's pixel list is traversed twice, the overall time complexity for each node is $O(N_{1+2} + N_{1+2})$. So, the final time complexity is $O(M + N_{1+2} + N_{1+2})$. Since $M \leq N_{1+2}$, the time complexity simplifies to $O(N_{1+2})$.

- The auxiliary list llist1 is traversed with a while loop:
This loop iterates over all elements in the list (N_{1+2}).
Each element in the list calls pp->setIntoRaw() to modify the new image, which executes in constant time O(1) per iteration.

Running Time: $O(N_{1+2})$ *N_{1+2} *equals the number of pixels in the first image plus the second image's pixels, which had an already existing equal SumRGB node when the insertion was done.*

13. Generating second output
Scope: Method level
Analyzed:

- Singly linked creation: SinglyLinkedList **object** → O(1)

- Call to t1->pixelsFoundinNodes():
We need to traverse the tree and collect every pixel. The function performs an inorder traversal, so it visits each node once, resulting in a time complexity of O(M), where M is the number of nodes. For each node in the tree, we traverse its list of pixels. Since the tree contains a total

of N_{1+2} pixels the overall time complexity is $O(M + N_{1+2})$
and because $M \leq N_{1+2} \rightarrow O(N_{1+2})$

- The auxiliary list llist2 is traversed with a while loop:
   This loop iterates over all elements in the list (N_{1+2}).

Running Time: $O(N_{1+2})$ *N_{1+2} equals the number of pixels in the first image plus the second image's pixels, which had an already existing equal SumRGB node when the insertion was done.*

14. Generating third output
   Scope: Method level
   Analyzed:
      - Singly linked creation: SinglyLinkedList object $\rightarrow O(1)$

      - Call to t2$\rightarrow$pixelsFoundinNodes():
         We need to traverse the tree and collect every pixel. The function performs an inorder traversal, so it visits each node once, resulting in a time complexity of $O(M)$, where M is the number of nodes. For each node in the tree, we traverse its list of pixels. Since the tree contains a total of N_{1+2} pixels the overall time complexity is $O(M + N_{1+2})$ and because $M \leq N_{1+2} \rightarrow O(N_{1+2})$

      - The auxiliary list llist3 is traversed with a while loop:
         This loop iterates over all elements in the list (N_{1+2}).

Running Time: $O(N_{1+2})$ *N_{1+2} equals the number of pixels in the first image plus the second image's pixels, which had an already existing equal SumRGB node when the insertion was done.*

Running time of the whole program

$\rightarrow O(N_1 + (N_1 * M) + N_1 + M + (N_1 + M^2) + N_1 + N_2 + (N_2 * M) + N_{1+2} + (N_2 * \log(M)) + N_{1+2} + N_{1+2} + N_{1+2} + N_{1+2})$

$\rightarrow O(4 * N_1 + (N_1 * M) + M^2 + N_2 + (N_2 * M) + (N_2 * \log(M)) + 5 * N_{1+2})$

$\rightarrow O((N_1 * M) + (N_2 * M) + M^2 + (N_2 * \log(M)))$ and because $N_2 * \log(M) \leq N_2 * M$

$\rightarrow O((N_1 * M) + (N_2 * M) + M^2)$ we will denote the number of pixels from the largest image as N_L

$\rightarrow O((N_L * M) + M^2)$ since $M^2 \leq (N_L * M)$

Finally $\rightarrow O(N_L * M)$ *N_L equals the number of pixels from the largest of both images and M equals the number of nodes in T1/T2 (both have the same number of nodes).*

2. Possible enhancements:

Currently, we have separate classes for the Binary Search Tree (T1) and the Binary Balanced Tree (T2). An improvement could be to create a base or generic class that can be used for both trees. This class could include common methods, such as pixel insertion, node's height calculation and finding the node with the most pixels, to avoid code duplication. This would not only reduce redundancy but also make the code easier to maintain and extend in the future.

One potential enhancement would be to optimize memory usage, especially when handling large images. Currently, when two large images are processed, the program consumes a significant amount of memory due to the way pixels are stored in the data structures.

3. ADT reasoning:

- Queue of Pixels:
  This queue was implemented to dynamically store the pixels of each image when they are loaded. Since the size of the image is not known beforehand, using static data structures such as an array would be inefficient and wasteful. This dynamic queue allocates memory as pixels are added, so there is no need to predefine a maximum size. As it is a queue, it ensures FIFO order, preserving the order in which the pixels appear in the image.

- List of Pixels / List of SumRGB values (int):
  Each node in the tree contains a dynamic list of pixels sharing the same sumRGB value. As in the previous case, the number of pixels with the same sumRGB value can't be predicted, for this reason dynamic lists grow as needed to store all relevant pixels efficiently. The same thing happens with the dynamic list used to store the sumRGB values of nodes in the Binary Search Tree. Using these dynamic lists, insertions and deletions are efficient, as no shifting is required (unlike a static array).

- Binary Search Tree and Balanced Binary Search Tree:
  The number of unique sumRGB values varies depending on the image, which means the structure must grow dynamically to introduce the new nodes. Using dynamic memory ensures that each node (with its list of pixels) can be allocated as needed, avoiding memory waste. A Binary Search Tree allows efficient insertion and search of nodes in a running time of O(log N) on average, if an array was used the insertion running time could take O(N).

To sum up, dynamic data structures provide the flexibility, efficiency and adaptability required to handle the changing input sizes and operations in this image processing program. Static data structures, on the other hand, would lack the scalability and memory efficiency needed for this assignment, making them unsuitable for tasks involving unpredictable or big amounts of data.

5. References

Implementations:

- https://www.geeksforgeeks.org/references-in-cpp/
- Provided content about references in the course
- Trees class slides

Specifications:

- Stacks and Queues class slides.
- Lists class slides.
- Trees class slides.

Dynamic vs Static DS:

- https://e2ehiring.com/blogs/static%20data%20structure%20vs%20dynamic%20data%20structure-413

Running time:

- Fundamentals of Data Structures class slides.

Diagrams:

- https://www.edrawsoft.com/es/article/class-diagram-relationships.html
- https://app.diagrams.net