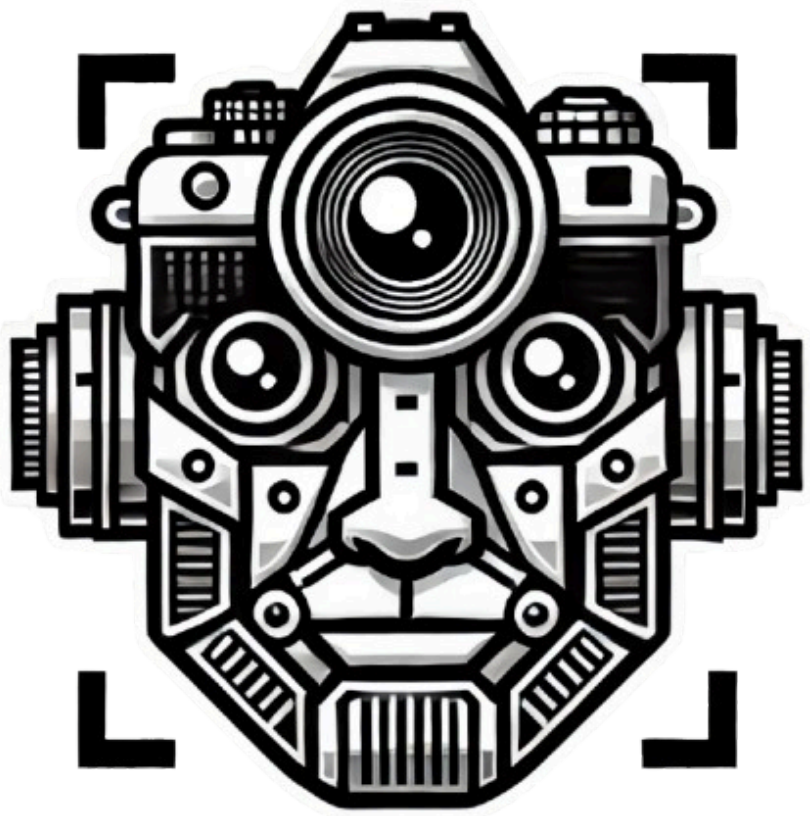
PHOTOFORMER *the tiny image editor*



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 previous.
    2. Definition of operations of the ADT
        1. Name, arguments, and return values

2. Design

1. 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.
2. Use case diagrams (UML notation)
3. Class diagram
4. Explanation of classes
    1. Explanation of significant methods, including those created for operations and any other needed.
5. Explanation of the behaviour of the program

3. Implementation

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

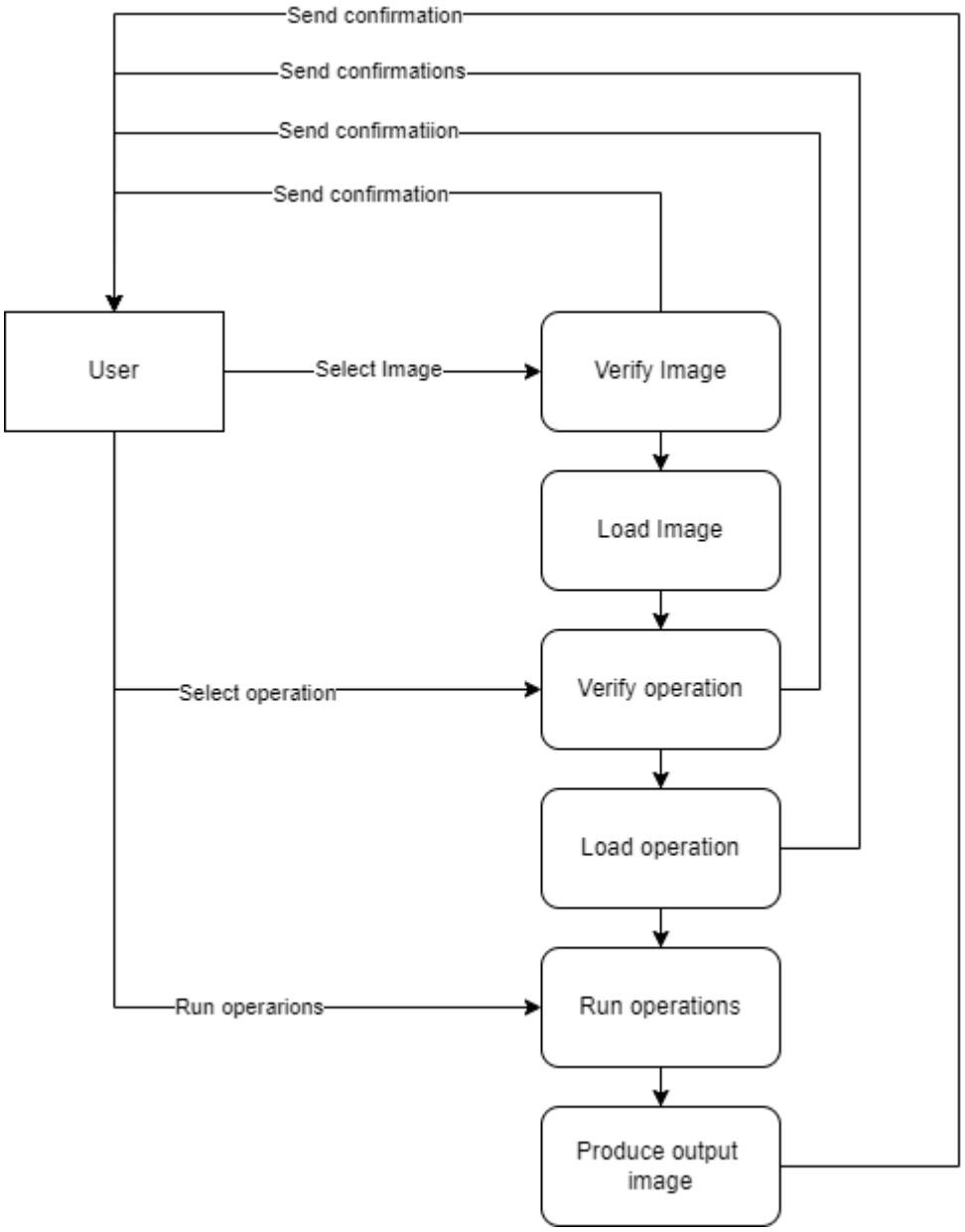
4. Review

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:



1. Input Handling:
    - Images are loaded from disk using the TinyImageJM class, which handles BMP file reading and stores image data in memory.
    - User inputs, such as the image file name and operations to perform, are collected via standard input (std::cin).
2. Internal Processing:
    - The Core class initializes the image processing by creating an instance of TinyImageJM and loading the image.
    - The image data is processed using various operations (like reverse, sort, flip, flop, and darken) through methods in the Operation class.
    - Data is stored and manipulated using data structures such as queues and stacks (QueuePixel, OpStack).

3. Output Handling:
    - Processed images are saved back to disk using the saveImageToDisk method in the TinyImageJM class.

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 weren'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 the element at the front of the queue. If the queue is not empty:
  - Stores the front element in a temporary pointer (tmp).
  - Updates front to the next element in the queue.
  - If the queue becomes empty (front is nullptr after the operation), it sets rear to nullptr.
- Notes: Returns nullptr if the queue is empty.

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. If the queue is empty:
  - Sets the new element as both the front and rear.
  - Otherwise, it sets the next pointer of the current rear to the new element and then updates rear to point to the new element.

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. It iterates through the original queue, starting from front, and for each element:
  - Retrieves the RGBPixelXY data from the current element.
  - Creates a new RGBPixelXY object that duplicates the pixel's properties.
  - Enqueues this copied pixel into the new queue.
- Notes: Returns a pointer to the new QueuePixel object, which is a copy of the original queue with independent elements.

QueueDS

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

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 weren't used.

2. Operations:

1. Constructor: QueueDS()

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

2. Destructor: ~QueueDS()

- Name: Destructor ~QueueDS
- Arguments: None
- Return Value: None
- Description: Deallocates the memory used by the queue by repeatedly dequeuing and deleting elements from the front until the queue is empty. This cleanup 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 determining if front is nullptr. If so, it returns true; otherwise, it returns false.

4. dequeue()

- Name: dequeue
- Arguments: None
- Return Value: ElemPixel* (a pointer to the element at the front of the queue)
- Description: Retrieves the front element of the queue. If the queue is not empty:
  - Stores the front element in a temporary pointer (tmp).
  - Updates front to point to the next element in the queue.
  - If the queue becomes empty after this operation (front becomes nullptr), it sets rear to nullptr as well.
- Notes: Returns nullptr if the queue is empty.

5. enqueue(RGBPixelXY* ppix)

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

- Description: Adds a new element to the rear of the queue. It creates a new ElemPixel object using the provided pixel. If the queue is empty:
  - Sets the new element as both front and rear.
  - Otherwise, it sets the next pointer of the current rear to the new element and then updates rear to point to the new element.

DoublyLinkedList

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:
                insertLast: item list -> list
                retrieveLast: list -> item
                empty: list -> bool
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 end 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.

2. Operations:

1. Constructor: DoublyLinkedList()

- Name: Constructor DoublyLinkedList
- Arguments: None
- Return Value: None
- Description: Initializes an empty doubly linked list by setting both the header and last pointers to nullptr, indicating that the list is currently empty.

2. Destructor: ~DoublyLinkedList()

- Name: Destructor ~DoublyLinkedList
- Arguments: None
- Return Value: None
- Description: Cleans up the memory used by the doubly linked list by repeatedly retrieving and deleting the last element until the list is empty. This ensures that all dynamically allocated elements are removed, preventing memory leaks.

3. insertLast(RGBPixelXY* ppix)

- Name: insertLast
- Arguments:
  - RGBPixelXY* ppix: A pointer to an RGBPixelXY object, representing a pixel element to be added to the end of the list.
- Return Value: None
- Description: Inserts a new element at the end of the list. It creates a new ElemPixel element using the provided pixel and adds it to the list as follows:
  - If the list is empty, the new element is set as both the header and last element.
  - If the list is not empty, the new element's previous pointer is set to the current last element, and the next pointer of the current last element is updated to point to the new element. The last pointer is then updated to the new element.

4. retrieveLast()

- Name: retrieveLast
- Arguments: None
- Return Value: ElemPixel* (pointer to the retrieved element)
- Description: Retrieves the last element from the list. If the list is not empty, it performs the following:
  - Stores the current last element in a temporary pointer (tmp).
  - Updates the last pointer to point to the previous element.
  - If the list becomes empty after removal (i.e., last becomes nullptr), it sets the header to nullptr.

- Returns a pointer to the removed element, allowing the calling code to manage the deleted element if necessary.

5. isEmpty()

- Name: isEmpty
- Arguments: None
- Return Value: bool (returns true if the list is empty, false otherwise)
- Description: Checks if the list is empty by determining whether the header pointer is nullptr. Returns true if the list is empty and false if it contains elements.

OpStack

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

```
SPEC STACK[ITEM]
        GENRES stack, item
        OPERATIONS:
                push: stack item -> stack
                pop: stack -> item
                top: stack -> item
                makenull: stack -> stack
                empty: stack -> bool
ENDSPEC
```

Adapted specification:

```
SPEC STACK[ITEM]
        GENRES stack, item
        OPERATIONS:
                push: stack item -> stack
                pop: stack -> item
                empty: stack -> bool
ENDSPEC
```

Adaptations:

- Item holds an operation represented as Operation.
- Push, empty and pop have been preserved from the standar specification.
- Makenull wasn't used.

2. Operations:

1. Constructor: OpStack()

- Name: Constructor OpStack
- Arguments: None
- Return Value: None

- Description: Initializes an empty stack by setting the top pointer to nullptr, indicating that the stack has no elements.

2. Destructor: ~OpStack()

- Name: Destructor ~OpStack
- Arguments: None
- Return Value: None
- Description: Deallocates the memory used by the stack by repeatedly popping and deleting elements from the top until the stack is empty. This prevents memory leaks by ensuring all dynamically allocated operations are removed.

3. pop()

- Name: pop
- Arguments: None
- Return Value: Operation* (a pointer to the operation that was on top of the stack)
- Description: Retrieves the top operation from the stack. If the stack is not empty:
  - Stores the top element in a temporary pointer (tmp).
  - Updates top to point to the next element in the stack.
  - Returns the tmp pointer, containing the popped operation.
- Notes: Returns nullptr if the stack is empty.

4. isEmpty()

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

5. push(std::string op)

- Name: push
- Arguments:
  - std::string op: A string representing an operation to be pushed onto the stack.
- Return Value: None
- Description: Creates a new Operation object for the provided operation string and places it on top of the stack. The new operation's next pointer is set to the current top of the stack, and top is updated to point to the new operation.

StackDS

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

```
SPEC STACK[ITEM]
        GENRES stack, item
        OPERATIONS:
                push: stack item -> stack
                pop: stack -> item
                top: stack -> item
                makenull: stack -> stack
                empty: stack -> bool
ENDSPEC
```

Adapted specification:

```
SPEC STACK[ITEM]
        GENRES stack, item
        OPERATIONS:
                push: stack item -> stack
                pop: stack -> item
                empty: stack -> bool
ENDSPEC
```

Adaptations:

- Item holds pixel data represented as RGBPixelXY so the stack can efficiently manage pixel elements throughout various operations related to image processing.
- Push and empty have been preserved from the standar specification.
- Pop returns an ElemPixel which contains the pixel.
- Makenull wasn't used.

2. Operations:

1. Constructor: StackDS()

- Name: Constructor StackDS
- Arguments: None
- Return Value: None
- Description: Initializes an empty stack by setting the top pointer to nullptr, indicating there are no elements in the stack.

2. Destructor: ~StackDS()

- Name: Destructor ~StackDS
- Arguments: None
- Return Value: None
- Description: Cleans up memory used by the stack by repeatedly popping and deleting elements from the top until the stack is empty. This ensures that all dynamically allocated elements are properly removed, avoiding memory leaks.

3. isEmpty()

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

4. pop()

- Name: pop
- Arguments: None
- Return Value: ElemPixel* (pointer to the element at the top of the stack, or nullptr if the stack is empty)
- Description: Retrieves the top element of the stack. If the stack is not empty:
  - Stores the top element in a temporary pointer (tmp).
  - Updates top to point to the next element in the stack.
  - Returns tmp, the retrieved element.
- Notes: Returns nullptr if the stack is empty.

5. push(RGBPixelXY* ppix)

- Name: push
- Arguments:
  - RGBPixelXY* ppix: A pointer to an RGBPixelXY object representing the pixel data to be pushed onto the stack.
- Return Value: None
- Description: Inserts a new element at the top of the stack:
  - Creates a new ElemPixel with the given RGBPixelXY pixel data.
  - Sets the new element's next pointer to the current top element.
  - Updates top to point to the new element, making it the new top of the stack.

SortedDoublyLList

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:

insertSorted: item list -> list

retrieveFirst: list -> item

empty: list -> bool

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 in order based on the sum of its RGB values and its original order as second criteria (this way you don't need position).
- Retrieve has been modified so the pixel retrieved is the first 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.

2. Operations:

1. Constructor: SortedDoublyLList()

- Name: Constructor SortedDoublyLList
- Arguments: None
- Return Value: None
- Description: Initializes an empty sorted doubly linked list by setting both header and last pointers to nullptr, indicating the list is empty.

2. Destructor: ~SortedDoublyLList()

- Name: Destructor ~SortedDoublyLList
- Arguments: None
- Return Value: None
- Description: Cleans up the memory used by the list by repeatedly retrieving and deleting elements from the front until the list is empty. This ensures that all dynamically allocated elements are removed.

3. isEmpty()

- Name: isEmpty
- Arguments: None

- Return Value: bool (returns true if the list is empty, false otherwise)
- Description: Checks whether the list is empty by verifying if header is nullptr. If header is nullptr, it returns true; otherwise, it returns false.

4. insertSorted(RGBPixelXY* ppix)

- Name: insertSorted
- Arguments:
  - RGBPixelXY* ppix: A pointer to an RGBPixelXY object, representing the pixel to be inserted into the list.
- Return Value: None
- Description: Inserts a new element into the list in sorted order based on the sum of the RGB (from lowest to highest) values of the pixel (R + G + B), with ties broken by the pixel's X and Y coordinates (ascending order):
  - If the list is empty, the new element becomes both the header and last.
  - If the list is not empty, it iterates through the list to find the appropriate position based on the pixel's RGB sum and coordinates.
  - Once found, it updates pointers to insert the new element in sorted order. If no position is found (new element has the largest RGB sum), it inserts it at the end of the list.
- Notes: This operation maintains the sorted order of the list.

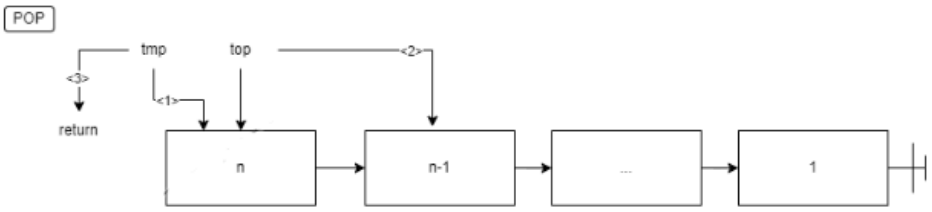
5. retrieveFirst()

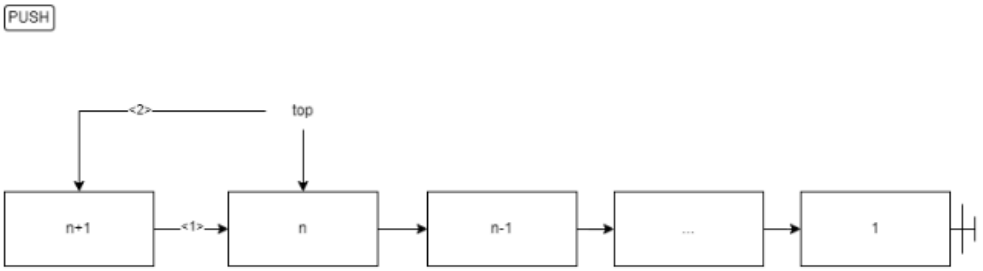
- Name: retrieveFirst
- Arguments: None
- Return Value: ElemPixel* (pointer to the element at the front of the list, or nullptr if the list is empty)
- Description: Retrieves the first element in the list. If the list is not empty:
  - Stores the header element in a temporary pointer (tmp).
  - Updates header to the next element in the list.
  - If the updated header is not nullptr, it sets header's previous pointer to nullptr. Otherwise, it sets last to nullptr, indicating the list is now empty.
- Notes: Returns nullptr if the list is empty.

2. Design

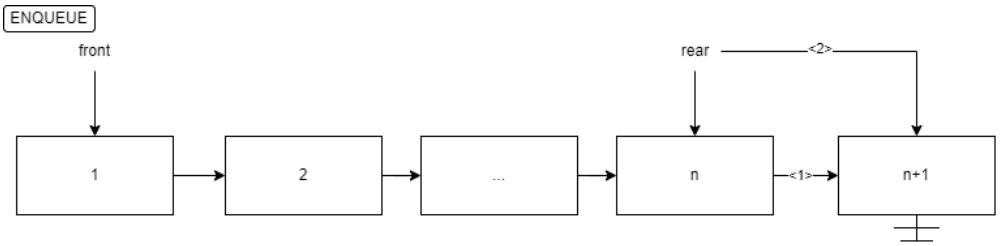
1. ADT diagrams and operation explanations:

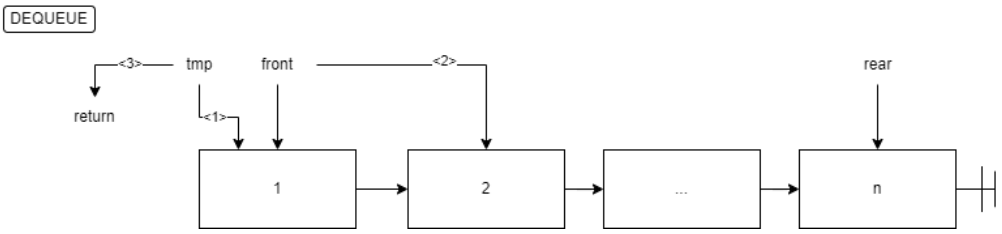
StackDS:



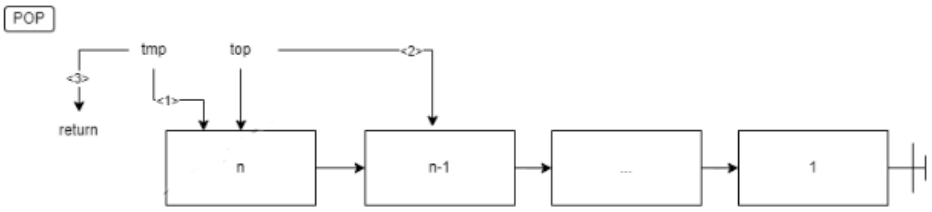


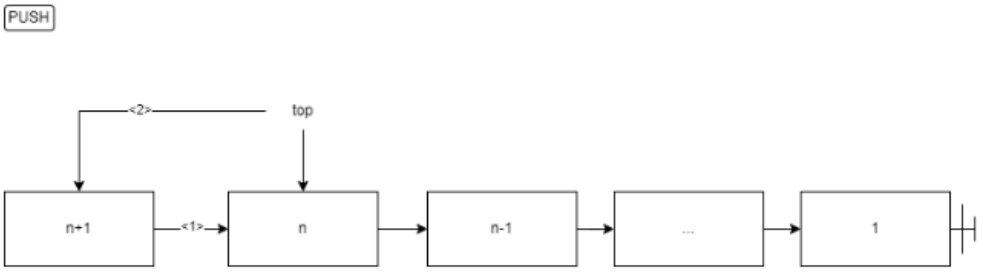
QueueDS:



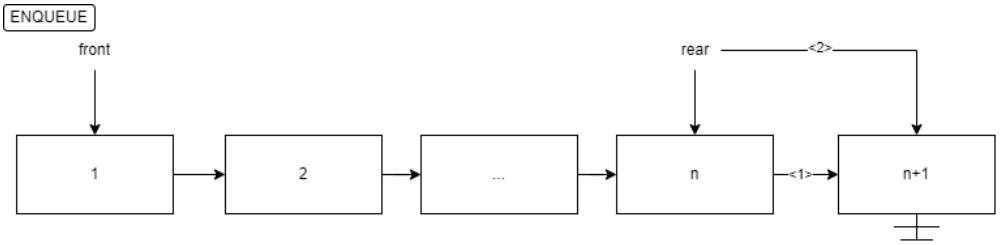


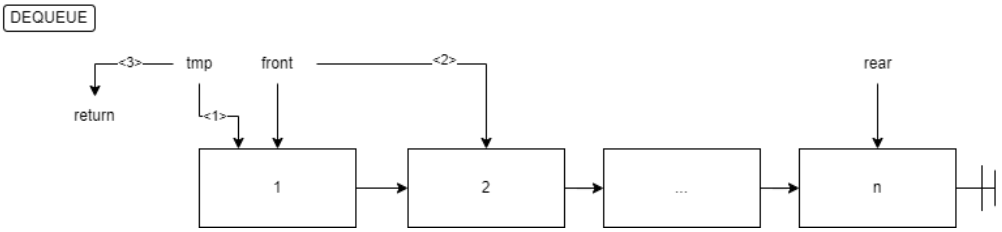
OpStack:

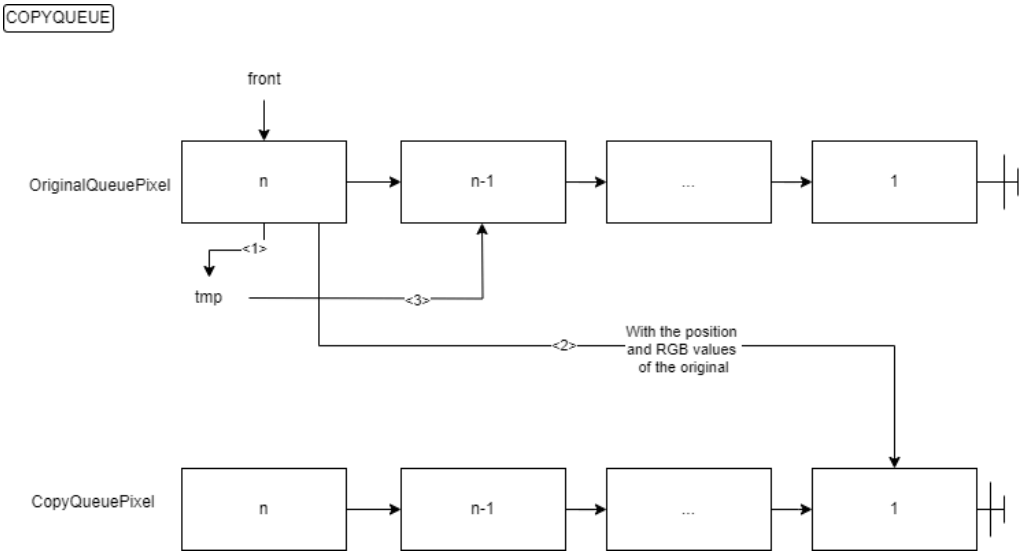




QueuePixel:

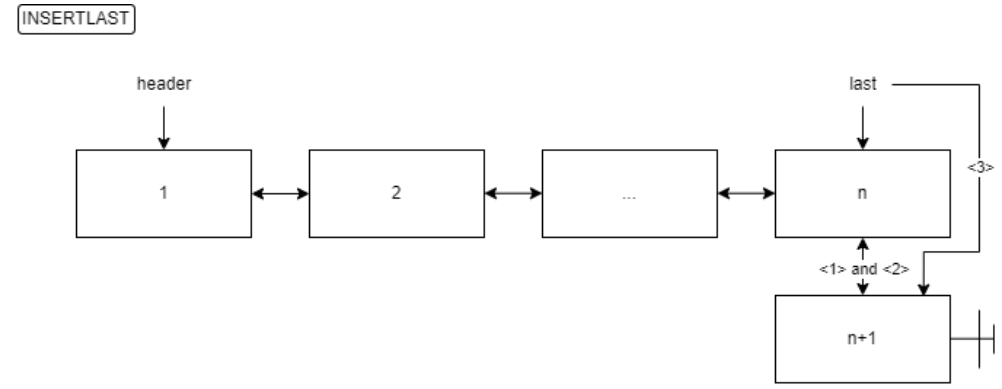


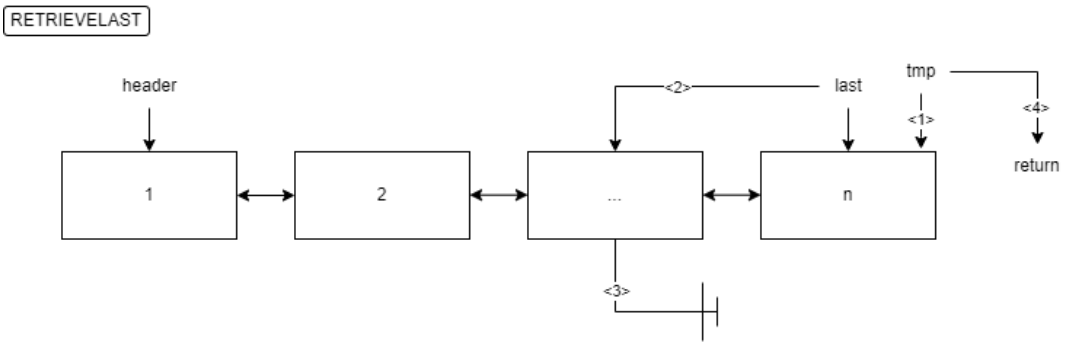




copyQueue: Creates a deep copy of the entire queue. Returns a pointer to the new QueuePixel object, which is a copy of the original queue with independent elements.

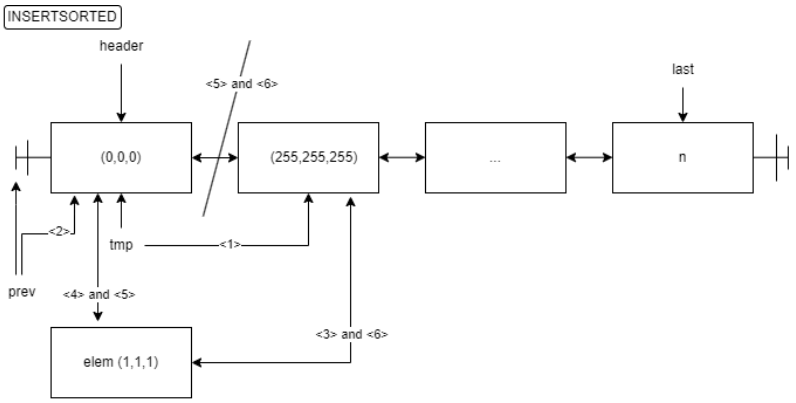
DoublyLinkedList:

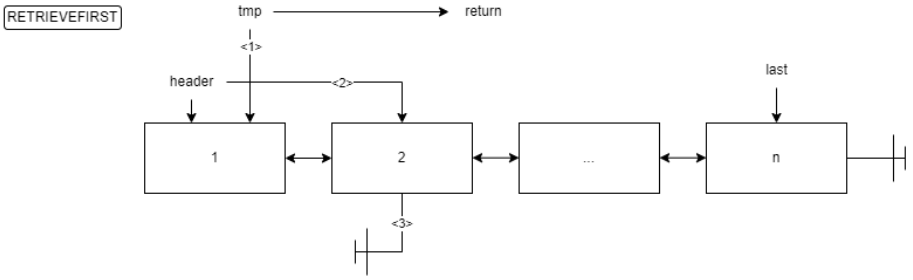




insertLast: Inserts a new element at the end of the list.
retrieveLast: Retrieves the last element from the list.

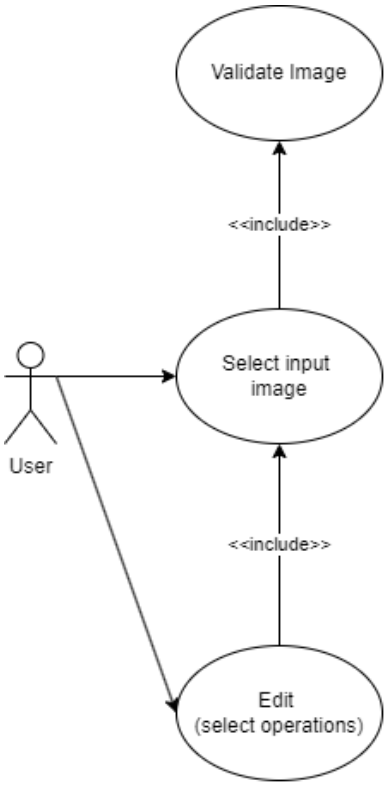
SortedDoublyLinkedList



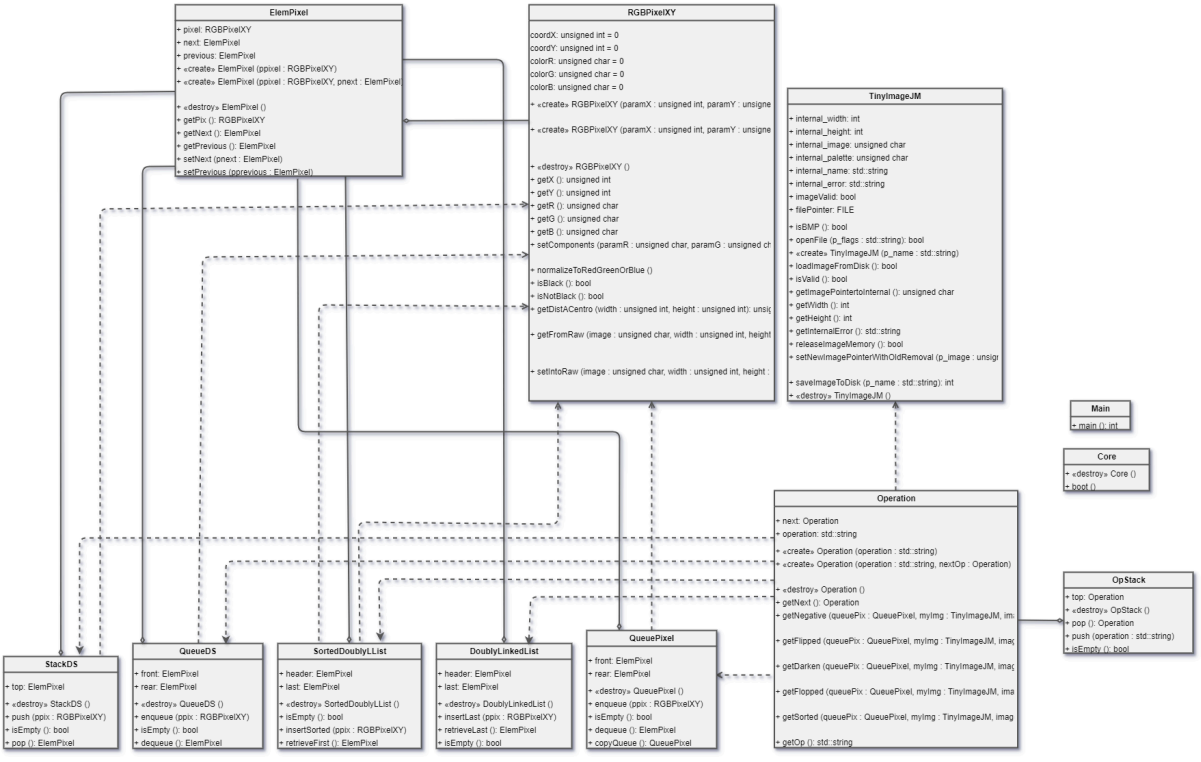


insertSorted: Inserts a new element into the list in sorted order based on the sum of the RGB (from lowest to highest) values of the pixel (R + G + B), with ties broken by the pixel's X and Y coordinates (ascending order).
retrieveFirst: Retrieves the first element in the list.

2. Use case diagram:



3. Class diagram:



4. 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 user input for operations to be performed on images and processes these operations.

    2. ElemPixel Class

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

    3. Operation Class

        The Operation class represents an operation applied to an image.

        - void getNegative(...): Applies a negative filter to the image by changing each pixel's RGB values with the negative value (255-R, G or B value).
        - void getFlipped(...): Flips the image. Using the reverse input/output nature of the stack structure.
        - void getDarken(...): Darkens the image, comparing the RGB mean ((R value+G value+B value)/3) to 128 we discover if the pixel is closer to black or white, we store them in a queue and then we generate an image with the pixel value mean between the original pixel and the black or white pixel.
        - void getFlopped(...): Flops the image using a doubly linked list.
        - void getSorted(...): Sorts the image pixels by RGB sum (lowest to highest) and original pixels order as second criteria.

    4. OpStack Class

        The OpStack class manages a stack for storing operations that will be applied to images.

    5. QueueDS Class

        The QueueDS class manages a queue DS for storing ElemPixel elements. It is used to perform the getNegative() and getDarken() operation

    6. QueuePixel Class

        The QueuePixel class manages a queue for pixel elements.

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

    7. StackDS Class

        The StackDS class manages a stack data structure for storing ElemPixel elements. It is used to perform the getFlipped() operation.

    8. DoublyLinkedList Class

        The DoublyLinkedList class manages a doubly linked list class. It is used in the getFlopped() operation.

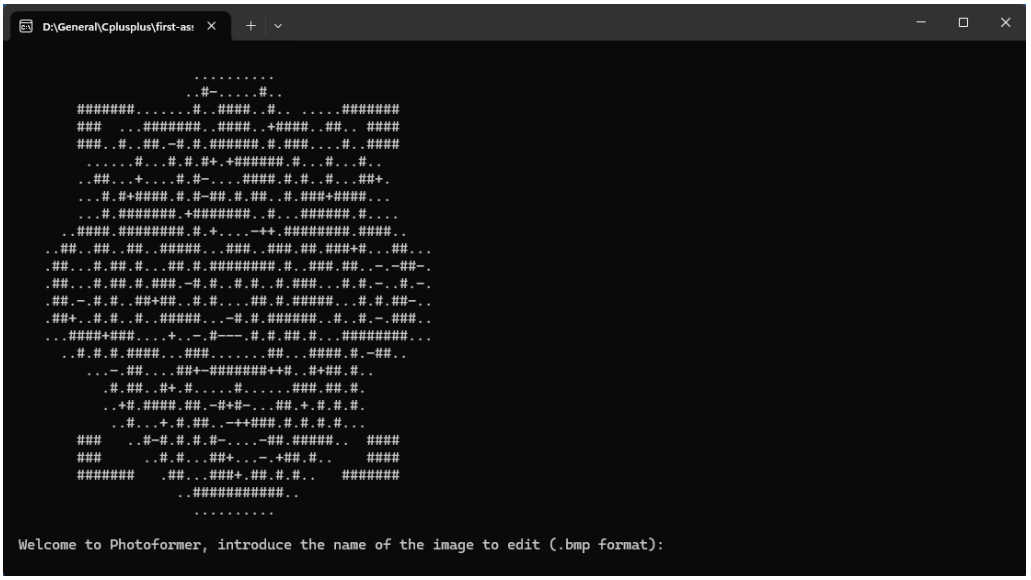
9. SortedDoublyLList Class

The SortedDoublyLList class manages a sorted doubly linked list used in the getSorted() operation.

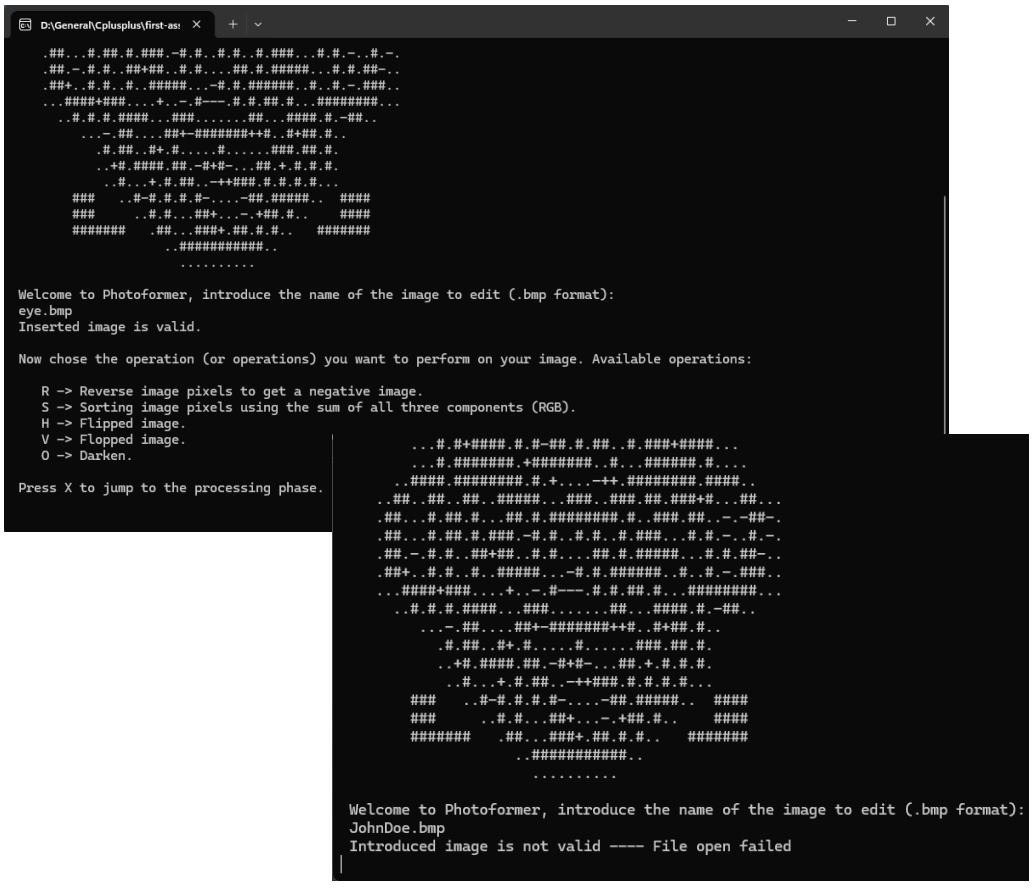
- void insertSorted(...). This method inserts a pixel element into the sorted doubly linked list in a sorted position based on the sum of the RGB values (lowest to highest) and the original order as second criteria.

5. Program behavior:

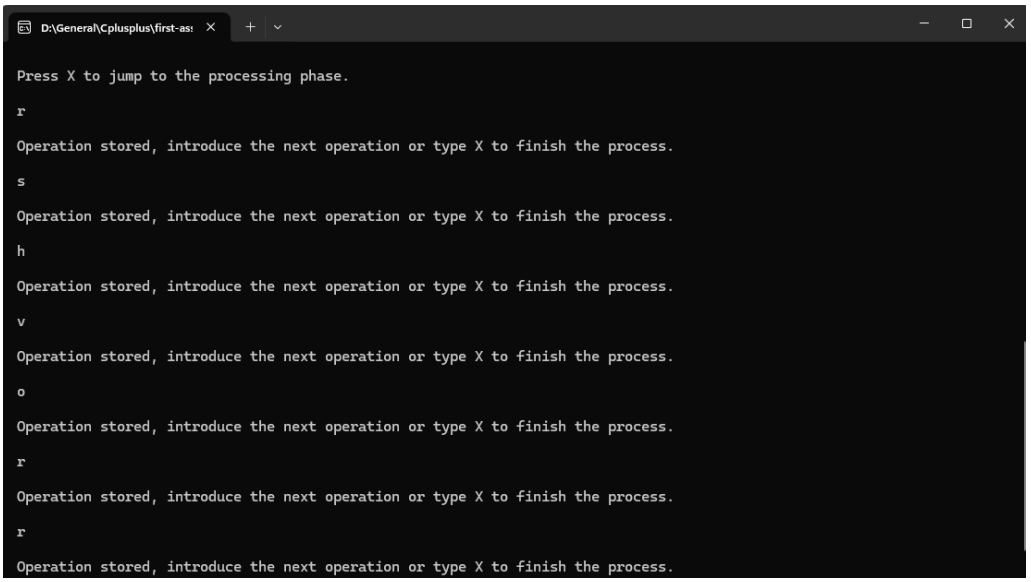
Having launched Photoformer the user is greeted and asked to enter the image name.



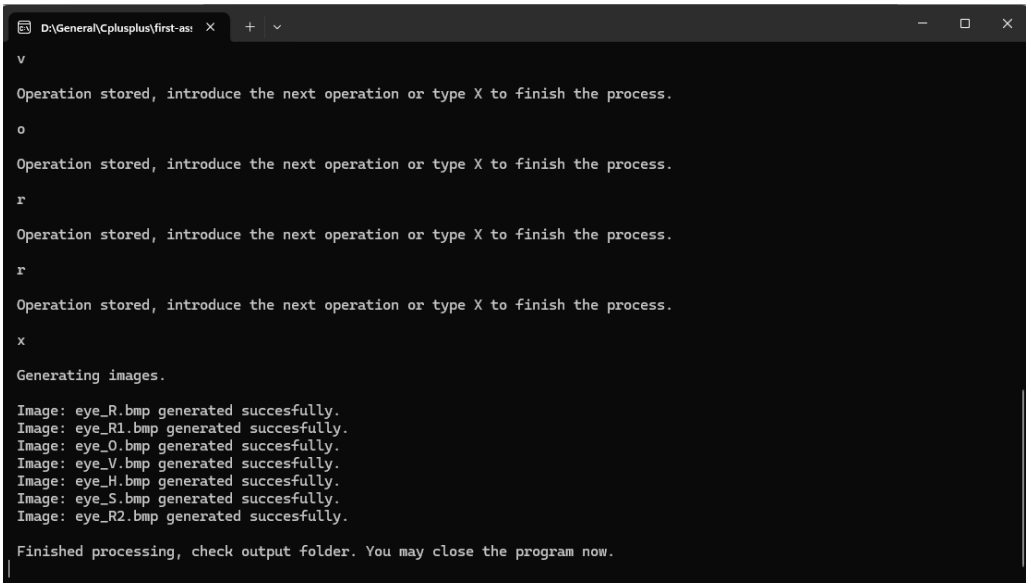
Provided that the image to edit is valid (.bmp included in the name) the program will list the available operations, if not the user will receive an error message and will be forced to re-run Photoformer.



If valid, the user will be able to select the operations to perform (the user can request Photoformer to perform the same operation several times, the output name will reflect this process). When finished the user will have to enter an “X” to end the input phase and enter the processing phase.

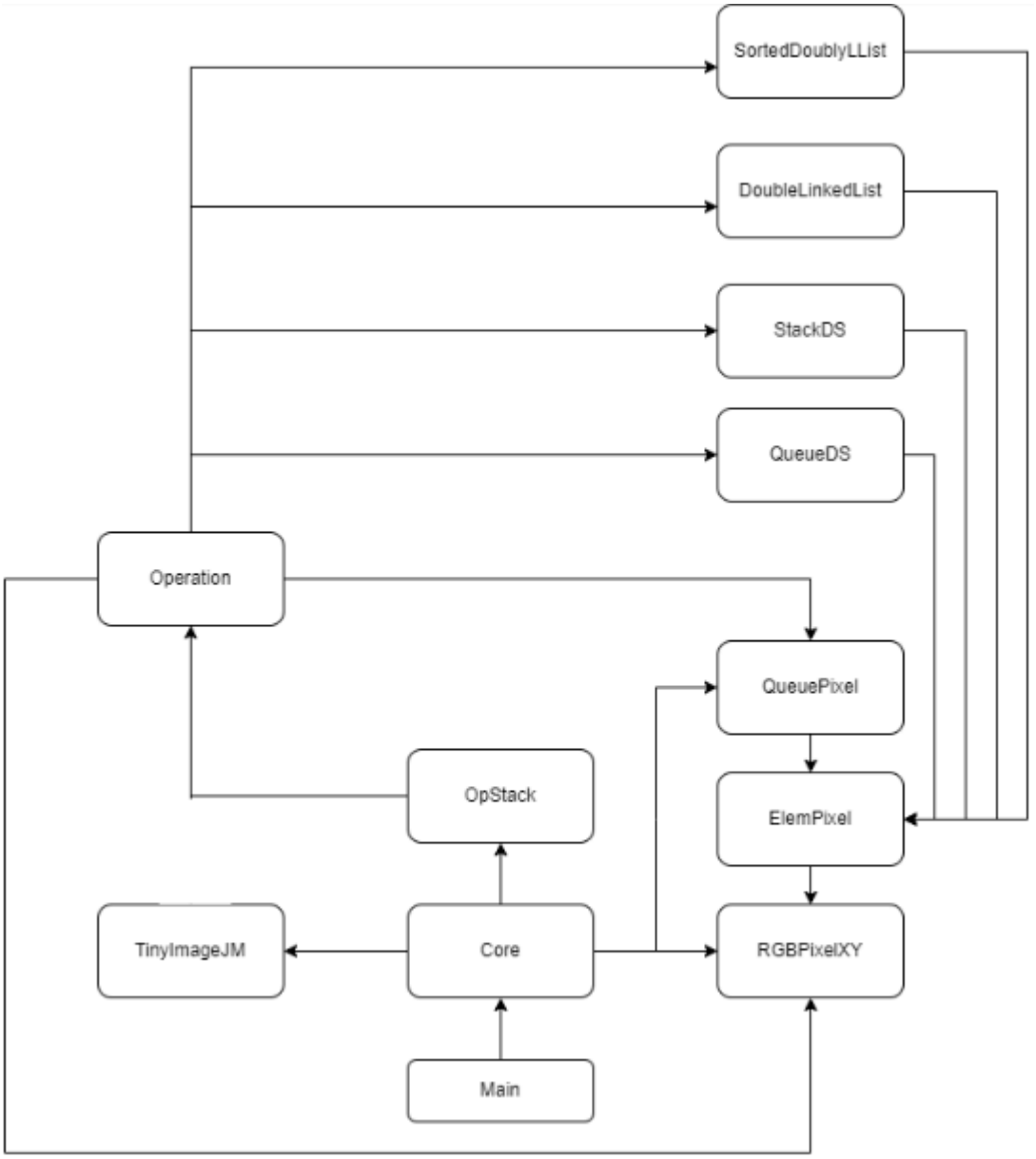


Finished the processing, the user will see a prompt asking him to check the output folder.



3. Implementation

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



2. Difficult parts explanations:

- The sorting algorithm used in getSorted():

  The insertion of pixels in a sorted manner based on their RGB values is very demanding in a computational perspective, especially with larger images.
  This method uses a sorted doubly linked list to maintain the order of pixels. When each pixel is inserted, the algorithm must traverse the list to find the correct position based on the sum of RGB values and the original order as second criterion.

- Memory management and cleanup:

  Properly deallocating memory to prevent leaks, especially in the data structures used (stacks, queues, lists…) that utilize dynamic memory allocation. After creating objects we take care of deleting those objects properly after using them.

4. Review

1. Running time:
    1. Loading Phase Efficiency

        Scope: Method Level

        Analyzed:

        - Pixel queue creation: q*ueuePixel* object → O(1)
        - Validation of the image format *isValid* check → 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 → O(N)

        Time Complexity: O(N)

    2. User Interaction Phase Efficiency

        Scope: Functionality Level

        Analyzed:

        - User inputs to select desired image transformations
        - Storage of operations in the operation stack (OpStack)

        Efficiency Metrics:

        - Time Complexity: O(1)

        As in this phase we only ask for input and use the insert method in a stack ADT.

    3. Data Processing Phase Efficiency

        Scope: Whole Program

        Analyzed:

        - Processing each operation stored in the operation stack based on user input.
        - Execution of operations:
        - R (Reverse pixels): Converts image pixels to negative.

            The getNegative function has an overall time complexity of O(N), where n is the total number of pixels in the image. Each major step: copying the pixel queue, allocating memory for the new image, transforming each pixel to its negative, and saving the result—processes all pixels linearly, resulting in O(N) complexity. Therefore, the function's runtime grows proportionally with the image size.

        - S (Sort pixels): Sorts pixels based on RGB component sum and original position.

            The getSorted function has a time complexity that can be analyzed through several key operations. First, copying the pixel queue takes O(N) time for a queue with n pixels. Next, while dequeuing each pixel is O(1), inserting these pixels into a sorted list can take O(N) for each of the n elements, leading to an overall time complexity of O(N^2).

Finally, retrieving pixels to place them into a new image takes O(N) time, as both retrieving and setting each pixel requires O(1). Therefore, the overall time complexity of the function is O(N^2).

- H (Flip horizontally): Flips image horizontally.
  The getFlipped function has a time complexity of O(N). Each step: copying queues, allocating memory, stacking pixels, flipping coordinates, and saving the image processes all pixels linearly. This makes the runtime grow proportionally with the image size.
- V (Flip vertically): Flips image vertically.
  The getFlopped function has a time complexity of O(N). Each main step: copying queues, allocating memory, building and retrieving from a list, swapping coordinates, and saving the image processes all pixels linearly, so the runtime scales proportionally with the image size.
- O (Darken): Darkens image pixels.
  The getDarken function has a time complexity of O(N). This complexity comes from several linear operations: copying the pixel queue, allocating memory for a new image, processing each pixel to determine whether it should be black or white, and mixing original and processed pixels to create the final image. Each of these steps processes all pixels, leading to a total runtime that scales directly with the size of the image.

Efficiency Metrics:

Operation	Time Complexity
R (Reverse)	O (N)
S (Sort)	O (N^2)
H (Flip H)	O (N)
V (Flip V)	O (N)
O (Darken)	O (N)

Where n is the total number of pixels.
Considerations:

- Loop Iterations: Each operation iterates over the pixel queue, affecting both time and memory.

2. Possible enhancements:

Avoid doing a second entire copy of the original queue (besides the first copy for not modifying the original queue data) when using some attribute of the pixels from the original queue for processing the final pixel that will be in the output image (as done in getFlipped(), getFlopped() and getDarken()). This would reduce both memory usage and processing time.

Optimize the getSorted() operation, particularly the insertSorted() method, which currently has high runtime when handling images that are not small. For images over 400 x 400 pixels, the processing time becomes significantly noticeable. Using other kinds of data structures may have a better running time result.

3. ADT reasoning:
    - getNegative - Queue:
      A queue was used because this method needs to process pixels in the same order they appear, going from the start to the end of the image. Since a queue is a FIFO (First-In, First-Out) structure it ensures that each pixel is processed in the order it appears in the image.

    - getFlipped - Stack:
      A stack was used for doing a horizontal flip because each row's pixel order needs to be inverted along the "y" axis. Since a stack is a LIFO (Last-In, First-Out) structure the last pixel pushed will be the first popped from the stack, being an effective way to store and retrieve pixels in reverse.

    - getFlopped - Doubly Linked List:
      A doubly linked list was used for doing a vertical flip because the order of pixels needs to be inverted along the "x" axis. A doubly linked list allows efficient traversal in both forward and reverse directions, making it easy to implement it in as a LIFO (Last-In, First-Out) structure; it was basically used as a stack. We chose to implement a doubly linked list for using different kinds of dynamic data structures.

    - getDarken - Queue:
      A queue was used because this method, as in getNegative, needs to process pixels in the same order they appear, going from the start to the end of the image. Since a queue is a FIFO (First-In, First-Out) structure it ensures that each pixel is processed in the order it appears in the image.

    - getSorted - Sorted Doubly Linked List:
      A sorted doubly linked list was used because sorting pixels by the sum of their RGB values requires inserting them in an ordered way. The sorted doubly linked list allows each pixel to be placed in order by RGB sum, preserving original order as second criteria. Even though using other kinds of data structures such as queues and then ordering it may have a better running time result, we have chosen to use a sorted doubly linked list to use different data structures implementations.

Dynamic data structures were chosen for their flexibility and efficiency. They allow for adaptable memory allocation, maintain necessary processing order (FIFO or LIFO) and support efficient insertions and deletions. Static data structures would have increased the waste of memory (we would need to know the exact number of pixels in advance to allocate memory accordingly) and lack the flexibility needed for the image processing tasks.

5. References

Interface:

https://en.cppreference.com/w/cpp/language/string_literal
https://www.asciiart.eu/image-to-ascii

Data flow diagram:

https://www.lucidchart.com/blog/data-flow-diagram-tutorial
https://www.canva.com/online-whiteboard/data-flow-diagrams/

Dynamic vs Static DS:

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

Implementations:

https://www.geeksforgeeks.org/create-doubly-linked-list-using-double-pointer-inserting-nodes-list-remains-ascending-order/
https://www.geeksforgeeks.org/doubly-linked-list-tutorial/
Stacks and Queues class slides.
Lists class slides.

Specifications:

Stacks and Queues class slides.
Lists class slides.

Running time:

Fundamentals of Data Structures class slides.