Homework 1: Lists

CS 201 Data Structures II
Habib University
Spring 2020

Due: 18h on Friday, 7 February

In this assignment, we will implement 2 different data structures to represent a list. We will use the list to implement an image and we will write operations on those images. We will then time the operations in order to plot and compare the relative performance of the backing data structures.

1 Image Operations

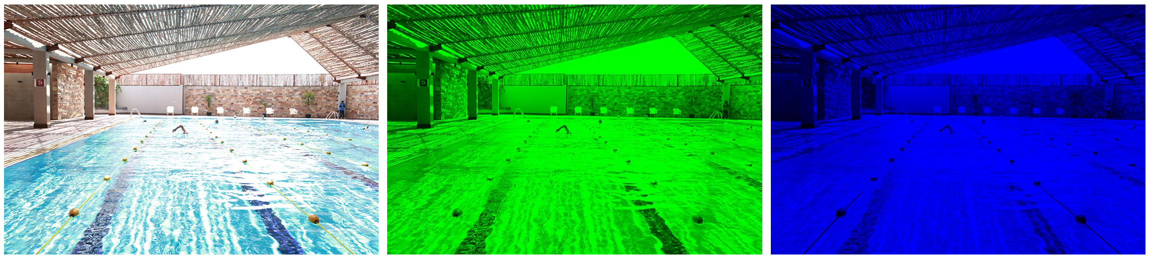
We will work with RGB images and perform 3 operations on them–channel suppression, rotations, and mask application. None of these operations is *destructive*. That is, the operations do not alter the original image, rather they return a new image containing the result of the operation.

1.1 Channel Suppression

An image is said to contain color values in different *channels*. In an RGB image, the channels are red, blue, and green. Each channel contains the intensities for that color for every pixel in the image. The values from all 3 channels at a pixel yields the RGB value at the pixel. The channel suppression operations switches off specifies channels. That is, all intensities in that channel will be turned to 0, or turned off. Figure 1 shows an original image and 2 modifications, once with the blue channel turned off, and then with only the blue channel turned on, i.e. the red and green channels turned off.

1.2 Rotation

Given a square image, i.e. one whose width is equal to its height, this operation generates a new image that contains rotations of the original image. Figure 2 shows an example of applying the operation. The resulting image has twice the dimensions of the original image, i.e. twice the width and twice the height.



(a) An RGB image of a swimming pool. (b) The image with the blue channel turned off. (c) The original image with only the blue channel turned on.

Figure 1: Example of channel suppression.

(a) A square image.

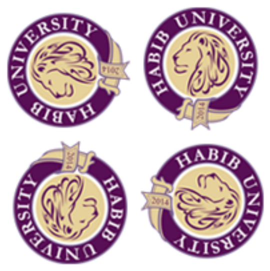

(b) The image obtained as a result of applying rotations to the original image.

Figure 2: Example of rotation.

1.3 Applying a Mask

A mask specifies certain *weights* and applying the mask to an image entails replacing the value of each pixel in the image with a *weighted average* of the values of its *neighbors*. The weights and the neighbors to consider for the average are specified by the mask.

The mask is an $n \times n$ array of non-negative integers representing weights. For our purposes, n must be odd. This means that the $n \times n$ array has a well defined center–the *origin*. The weights in the mask can be arbitrary integers–positive, negative, or zero.

For each pixel in the input image, think of the mask as being placed on top of the image so its origin is on the pixel we wish to examine. The intensity value of each pixel under the mask is multiplied by the corresponding value in the mask that covers it. These products are added together. Always use the original values for each pixel for each mask calculation, not the new values you compute as you process the image.

For example, refer to Figure 3a, which shows the 3×3 mask,

$$\begin{bmatrix} 1 & 3 & 1 \\ 3 & 5 & 3 \\ 1 & 3 & 1 \end{bmatrix}$$

and an image on which we want to perform the mask computation. Suppose we want to compute the result of the mask computation for pixel e. This result would be:

$$a + 3b + c + 3d + 5e + 3f + g + 3h + i$$

Some masks require a *weighted average* instead of a weighted sum. The weighted average in the case of Figure 3a for pixel e would be:

$$\frac{a + 3b + c + 3d + 5e + 3f + g + 3h + i}{1 + 3 + 1 + 3 + 5 + 3 + 1 + 3 + 1}$$

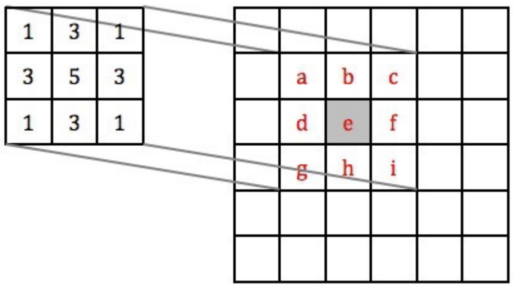
Instead of doing this calculation for each channel individually, use the average value of the red, green, and blue channels. For example, if the pixel is given by $(r, g, b) = (107, 9, 218)$, then use $(107 + 9 + 218)//3 = 111$ for each channel. This effectively converts the image to grayscale.

Note that sometimes when you center the mask over a pixel you want to operate on, the mask will hang over the edge of the image. In this case, compute the weighted sum of only those pixels the mask covers. For the example shown in Figure 3b, the result for the pixel e is given by:

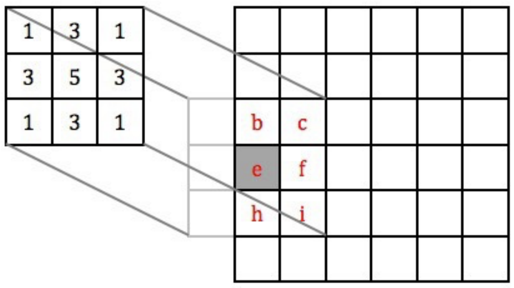
$$3b + c + 5e + 3f + 3h + i$$

If no average is required and the following if an average is required:

$$\frac{3b + c + 5e + 3f + 3h + i}{3 + 1 + 5 + 3 + 3 + 1}$$



(a) Overlay the 3×3 mask over the image so it is centered on pixel e to compute the new value for pixel e.



(b) If the mask hangs over the edge of the image, use only those mask values that cover the image in the weighted sum.

Figure 3: Applying a mask to an image.

1.3.1 Applications of Masks

Applying different masks leads to different properties. For example, applying the following mask above leads to blurring of the image. Figures 4a and 4b show the blurring effect of this mask. Note that color information is lost as mentioned above.

$$\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Another application we use in an implementation of *Canny Edge Detection* using *Sobel Operators*. Once the image has been blurred as above, two more *filters*, or masks, (the Sobel operators) are applied in succession to the blurred image. These filters determine the change in intensity, which approximates the horizontal and vertical derivatives.

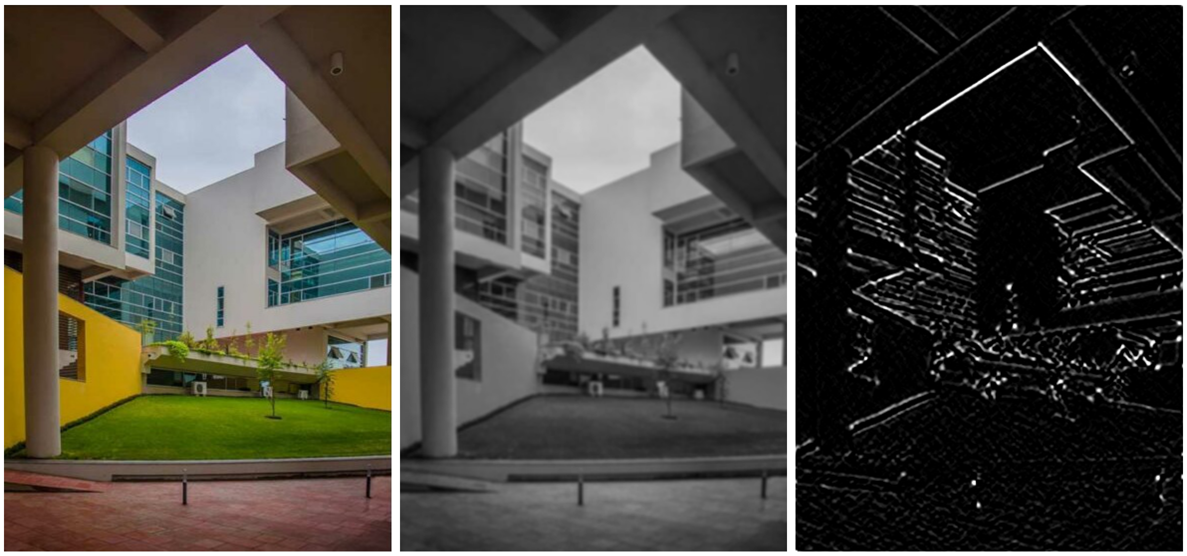
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

. After these operations are applied one after the other to the blurred image, the values obtained are used to search for edges based on the magnitude and direction of the change in intensity. An example of the final result is shown in Figure 4c.

2 Image

We treat an image as a grid of *pixels* where each pixel has an RGB value indicating the red, green, and blue intensities of the pixel. An image has *dimensions*, namely *width* and *height*, which determine the number of *rows* and *columns* in the image. Every pixel in the image is at a unique combination of row and column numbers which can therefore be used as coordinate system in the image. An image with width w and height h is said to be of size $w \times h$. Figure 5a shows the column and row numbers in a $w \times h$ image along with the resulting pixel coordinates. Note that the coordinate is just a means to locate a pixel in the image, it is not the value stored at the pixel. The value stored at a pixel a triplet denoting the red, green, and blue intensities respectively.

We will work with a *flattened*representation of an image. That is, we will store the pixel values in a 1-dimensional list structure as opposed to 2-dimensional structure. The list stores pixel values as they appear in the image from left to right and top to bottom. Figure 5b shows a 5×5 image with some supposed RGB



(a) An image with sharp details and several lines.

(b) Result of applying the blur mask to the original image.

(c) Result of applying the Sobel filters to the blurred image.

Figure 4: Blurring and detection of edges in an image using masks.

		Columns				
		0	1	2	$\dots$	$(w-1)$
Rows	0	$(0,0)$	$(0,1)$	$(0,2)$	$\dots$	$(0,w-1)$
	1	$(1,0)$	$(1,1)$	$(1,2)$	$\dots$	$(1,w-1)$
	2	$(2,0)$	$(2,1)$	$(2,2)$	$\dots$	$(2,w-1)$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	$(h-1)$	$(h-1,0)$	$(h-1,1)$	$(h-1,2)$	$\dots$	$(h-1,w-1)$

(a) Row and column numbers of an image with width w and height h. Pixel coordinates are also shown.

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y

(b) A 5×5 image with supposed pixel values.

Figure 5: Image dimensions and pixel coordinates.

values. Note that each value would be a triplet of integers, each integer between 0 and 255 inclusive. Using our representation, the image in Figure 5b will be represented as the list:

$$[a, b, c, d, e, f, g, h, i, j, k, l, , m, n, o, p, q, r, s, t, u, v, w, x, y]$$

3 Implementation Details and Tasks

3.1 Image Operations

You have to implement the image operations shown in Listing 1 on Page 6 and also included in the accompanying file `image_operations.py`. These correspond to those illustrated in Section 1. The operations manipulate an instance of our image type, called `MyImage`. Note that none of the operations is destructive. That is, the original image is not modified. The result of an operation is returned as a new `MyImage` instance.

3.2 Image

We implement our custom image type as `MyImage` as shown in Listing 2 on Page 7 and also included in the accompanying file `myimage.py`. Note that the implementation uses on `MyList` which is our implementation of the list. The given implementation is complete and you are not to change it except for the indicated line in `MyImage.__init__()`. You will change this line to use your implementation of `MyList`. This is the only change you are allowed to make in the implementation of `MyImage`.

3.3 List

We implement our custom list as `MyList` whose interface shown in Listing 3 on Page 9 and also included in the accompanying file `mylist.py`. Note that the implementation is mostly complete except for the segments marked as `pass`. The implementation includes a mechanism to allow iteration over the list and using some other native python syntax with it.

You have to copy the implementation to `ArrayList` and PointerList which will implement the list using python arrays and pointers respectively. See the implementation of BST in Listing 4 on Page 11 and also included in the accompanying file `bst.py` to understand pointers in python. You then have to modify the required line in `myimage`.py to use your type.

4 Timing

We want to perform a timing analysis on the execution of the image operations using our list implementations. Those details are forthcoming.

5 Bonus

You can earn bonus marks for implementing interesting image manipulation algorithms of your own. Please see the rubric for more details. Interesting submissions will be featured on the course's forthcoming web page.

6 Credits

This homework is adapted from Homework 3 of the Fall 2014 offering of 15-122: Principles of Imperative Computation at Carnegie Mellon University (CMU). Credit is due to Abdullah Zafar for identifying it.

A Image Operations

```python
def remove_channel(src: MyImage, red: bool = False, green: bool = False,
                   blue: bool = False) -> MyImage:
    """Returns a copy of src in which the indicated channels are suppressed.

    Suppresses the red channel if no channel is indicated. src is not modified.

    Args:
    - src: the image whose copy the indicated channels have to be suppressed.
    - red: suppress the red channel if this is True.
    - green: suppress the green channel if this is True.
    - blue: suppress the blue channel if this is True.

    Returns:
    a copy of src with the indicated channels suppressed.
    """
    pass


def rotations(src: MyImage) -> MyImage:
    """Returns an image containing the 4 rotations of src.

    The new image has twice the dimensions of src. src is not modified.

    Args:
    - src: the image whose rotations have to be stored and returned.

    Returns:
    an image twice the size of src and containing the 4 rotations of src.
    """
    pass


def apply_mask(src: MyImage, maskfile: str, average: bool = True) -> MyImage:
    """Returns an copy of src with the mask from maskfile applied to it.

    maskfile specifies a text file which contains an n by n mask. It has the
    following format:
    - the first line contains n
    - the next n^2 lines contain 1 element each of the flattened mask

    Args:
    - src: the image on which the mask is to be applied
    - maskfile: path to a file specifying the mask to be applied
    - average: if True, averaging should to be done when applying the mask

    Returns:
    an image which the result of applying the specified mask to src.
    """
    pass
```

Listing 1: Image Operations

B MyImage

```python
from PIL import Image


class MyImage:
    """ Holds a flattened RGB image and its dimensions.
    """

    def __init__(self, size: (int, int)) -> None:
        """Initializes a black image of the given size.

        Args:
        - size: (width, height) specifies the dimensions to create.

        Returns:
        none
        """
        # Save size, create a list of the desired size with black pixels.
        width, height = self.size = size
        self.pixels: MyList = MyList(width * height, value=(0, 0, 0))
        # ^ CHANGE this line to use your implementation of MyList.

    def _get_index(self, r: int, c: int) -> int:
        """Returns the list index for the given row, column coordinates.

        This is an internal function for use in class methods only. It should
        not be used or called from outside the class.

        Args:
        - r: the row coordinate
        - c: the column coordinate

        Returns:
        the list index corresponding to the given row and column coordinates
        """
        # Confirm bounds, compute and return list index.
        width, height = self.size
        assert 0 <= r < height and 0 <= c < width, "Bad image coordinates: "\
            f"(r, c): ({r}, {c}) for image of size: {self.size}"
        return r*width + c

    def open(path: str) -> MyImage:
        """Creates and returns an image containing from the information at file path.

        The image format is inferred from the file name. The read image is
        converted to RGB as our type only stores RGB.

        Args:
        - path: path to the file containing image information

        Returns:
        the image created using the information from file path.
        """
        # Use PIL to read the image information and store it in our instance.
        img: PIL.Image = Image.open(path)
        myimg: MyImage = MyImage(img.size)
        width, height = img.size
        # Covert image to RGB. https://stackoverflow.com/a/11064935/1382487
        img: PIL.Image = img.convert('RGB')
        # Get list of pixel values (https://stackoverflow.com/a/1109747/1382487),
        # copy to our instance and return it.
        for i, rgb in enumerate(list(img.getdata())):
            myimg.pixels.set(i, rgb)
        return myimg

    def save(self, path: str) -> None:
        """Saves the image to the given file path.

        The image format is inferred from the file name.

        Args:
        - path: the image has to be saved here.

        Returns:
        none
        """
        # Use PIL to write the image.
        img: PIL.Image = Image.new("RGB", self.size)
        img.putdata([rgb for rgb in self.pixels])
        img.save(path)

    def get(self, r: int, c: int) -> (int, int, int):
        """Returns the value of the pixel at the given row and column coordinates.

        Args:
        - r: the row coordinate
        - c: the column coordinate

        Returns:
        the stored RGB value of the pixel at the given row and column coordinates.
        """
        return self.pixels[self._get_index(r, c)]

    def set(self, r: int, c: int, rgb: (int, int, int)) -> None:
        """Write the rgb value at the pixel at the given row and column coordinates.

        Args:
        - r: the row coordinate
        - c: the column coordinate
        - rgb: the rgb value to write

        Returns:
        none
        """
        self.pixels[self._get_index(r, c)] = rgb

    def show(self) -> None:
        """Display the image in a GUI window.

        Args:

        Returns:
        none
        """
        # Use PIL to display the image.
        img: PIL.Image = Image.new("RGB", self.size)
        img.putdata([rgb for rgb in self.pixels])
        img.show()
```

Listing 2: Image Type

C MyList

```python
from PIL import Image


class MyImage:
    """ Holds a flattened RGB image and its dimensions.
    """

    def __init__(self, size: (int, int)) -> None:
        """Initializes a black image of the given size.

        Args:
        - size: (width, height) specifies the dimensions to create.

        Returns:
        none
        """
        # Save size, create a list of the desired size with black pixels.
        width, height = self.size = size
        self.pixels: MyList = MyList(width * height, value=(0, 0, 0))
        # ^ CHANGE this line to use your implementation of MyList.

    def _get_index(self, r: int, c: int) -> int:
        """Returns the list index for the given row, column coordinates.

        This is an internal function for use in class methods only. It should
        not be used or called from outside the class.

        Args:
        - r: the row coordinate
        - c: the column coordinate

        Returns:
        the list index corresponding to the given row and column coordinates
        """
        # Confirm bounds, compute and return list index.
        width, height = self.size
        assert 0 <= r < height and 0 <= c < width, "Bad image coordinates: "\
            f"(r, c): ({r}, {c}) for image of size: {self.size}"
        return r*width + c

    def open(path: str) -> MyImage:
        """Creates and returns an image containing from the information at file path.

        The image format is inferred from the file name. The read image is
        converted to RGB as our type only stores RGB.

        Args:
        - path: path to the file containing image information

        Returns:
        the image created using the information from file path.
        """
        # Use PIL to read the image information and store it in our instance.
        img: PIL.Image = Image.open(path)
        myimg: MyImage = MyImage(img.size)
        width, height = img.size
        # Covert image to RGB. https://stackoverflow.com/a/11064935/1382487
        img: PIL.Image = img.convert('RGB')
        # Get list of pixel values (https://stackoverflow.com/a/1109747/1382487),
        # copy to our instance and return it.
        for i, rgb in enumerate(list(img.getdata())):
            myimg.pixels.set(i, rgb)
        return myimg

    def save(self, path: str) -> None:
        """Saves the image to the given file path.

        The image format is inferred from the file name.

        Args:
        - path: the image has to be saved here.

        Returns:
        none
        """
        # Use PIL to write the image.
        img: PIL.Image = Image.new("RGB", self.size)
        img.putdata([rgb for rgb in self.pixels])
        img.save(path)

    def get(self, r: int, c: int) -> (int, int, int):
        """Returns the value of the pixel at the given row and column coordinates.

        Args:
        - r: the row coordinate
        - c: the column coordinate

        Returns:
        the stored RGB value of the pixel at the given row and column coordinates.
        """
        return self.pixels[self._get_index(r, c)]

    def set(self, r: int, c: int, rgb: (int, int, int)) -> None:
        """Write the rgb value at the pixel at the given row and column coordinates.

        Args:
        - r: the row coordinate
        - c: the column coordinate
        - rgb: the rgb value to write

        Returns:
        none
        """
        self.pixels[self._get_index(r, c)] = rgb

    def show(self) -> None:
        """Display the image in a GUI window.

        Args:

        Returns:
        none
        """
        # Use PIL to display the image.
        img: PIL.Image = Image.new("RGB", self.size)
        img.putdata([rgb for rgb in self.pixels])
        img.show()
```

Listing 3: List Type

D Pointers in python

```python
class Node:
    def __init__(self, data) -> None:
        self.data = data
        self.left = self.right = None


class BST:
    def __init__(self):
        self.root = None

    def _insert(self, root: Node, n: int) -> None:
        if n <= root.data:
            if root.left:
                self._insert(root.left, n)
            else:
                root.left = Node(n)
        else:
            if root.right:
                self._insert(root.right, n)
            else:
                root.right = Node(n)

    def insert(self, n: int):
        if not root:
            root = Node(n)
        else:
            self._insert(self.root, n)
```

Listing 4: BST in python