EEL 4851-001 Data Structures
Programming Assignments (Five)

Sudeep Sarkar

1. Introduction

1.1 Why is this document important to me?

The goal of this set of assignments is to provide you with hands-on experience with handling various kinds of data structures. In the lectures you will be introduced to the data structures at an *abstract* level and in the assignments you will write code to *implement* and *use* these data structures. The set of tasks chosen for your assignments all relate to *images*. You will design algorithms to perform simple image processing tasks using data structures such as arrays, stacks, queues, trees, and hash tables. These exercises are not only important from a grading standpoint but also are crucial learning tools.

The assignments are to be completed in *groups of two*. The main reason behind this dictum is to expose you to the experience of team work. In the real world you will definitely be called upon to work with others towards a common end. It is very difficult to convey all aspects of team work in a classroom setting. One has to learn from experience.

You are welcome to discuss with other groups *but* the implementation should be your own. Please be sincere regarding this. Contrary to what you might expect, it is relatively easy to find out if codes have been copied! **Any case of unethical conduct will result in an F in the class.**

1.2 What should I submit?

For each assignment you should submit the following by the due dates and times.

1. *Demonstration:* You are expected to demonstrate your working code to Min Shin. *The demonstrations should be done during his office hours: 2:30 to 3:30 pm on Wednesday or 2pm to 3pm on Thursday.* (If you have problems with the days then let me or Min know.) You should sign up for a demonstration. Min will put up a sign up sheet outside ENB 325 two days prior to the due time. For any demonstration with out sign up, Min reserves the right to refuse appointments and to consider the demo part of the assignment as being turned in late.

2. *On Paper:* Submit, on paper (to me) by the due date and time:

(a) A written five (maximum) page report discussing your results and experience. The report should include the following: (i) introduction, (ii) flowchart of the algorithm, (iii) description of how would one run and use your code, (iv) some typical results or outputs, (v) a section describing the programming bugs and errors you ran into (vi) a section describing what you learned from the assignment and (vii) a statement of division of work between the partners.

Note that you are expected to submit a professionally appearing report and it should be well organized, clear, neat, with adequate visual aids such as figures and images. Writing is a very important part of the scientific process and should not be ignored.

(b) A print out of the source code.

1.3 Grading:

The grading of the assignment will be along the following lines.

1. The writeup is out of 5 points. One points each for the following:
   (a) Flowchart
   (b) Description of usage
   (c) Results
   (d) Programming bugs and errors
   (e) Overall presentation
2. The source code is out of 5 points. One point each for the following:
   (a) The existence of the code itself
   (b) Proper indentation of the code
   (c) Comments
   (d) Proper naming of the procedures, functions, and variables.
   (e) Modularity of the code.

   For proper program style, formatting, and documentation see the guidelines in Appendix H of your text book. You should adhere to these guidelines.
3. The demonstration is out of 5 points. You get three points if the program does *exactly* what it is supposed to do. Two points are reserved for the ease of use, the type of user interface, the ability to handle various user input errors, or any extra features that your system might have.

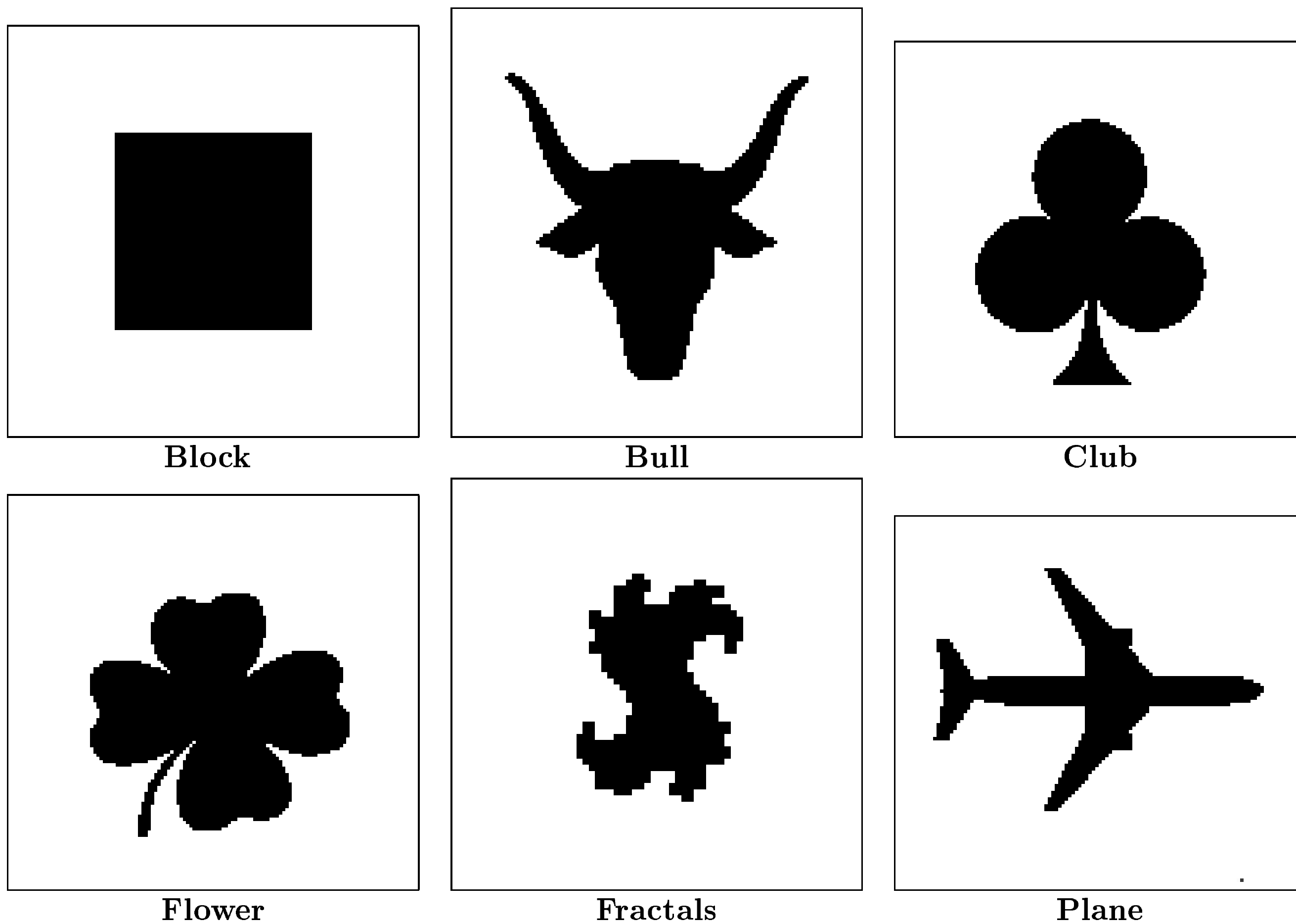

Figure 1.1: Samples of images to be used in the assignments.

Late assignments (or parts thereof) will be penalized 20% of the maximum possible grade for each extra day. Note that computer systems have a tendency go down unexpectedly. So, plan accordingly and do not wait for the last minute.

1.4 Data

All image data for the assignments will be available from the class homepage on Netscape. Figure 1.1 shows a sample of images that are available.

1.5 How are images represented in the computer?

An image (black and white) as defined on, say a photographic film, is a continuous function of brightness values. Each point on the developed film can be associated with a gray level value representing how bright that particular point is. To store images in a computer we have to sample and quantize (digitize) the image function. *Sampling* refers to considering the image only at a finite number of points. And *quantization* refers to the representation of the gray level value at the sampling point using finite number of bits. Each image sample is called a *pixel*. Your typical desktop image scanner does sampling and quantization for you.

One of the simpler scheme is sampling on a *regular* grid of squares. We can visualize the process as overlaying an uniform grid on the image and sampling the image function at the center of each grid square. Finer the grid, better the resolution of the image; coarser the grid, the more is the observed "pixelization" (see Fig. 1.2).

At each pixel (or at each grid square) we usually represent the gray level value using an integer ranging between 0 for black to 255 for fully white. All the images for our assignments will have just two levels: 0 and 255.

1.6 In what format are the images stored in a computer?

A digital image is essentially a collection of pixel values. There are various formats of storing an image in a file. Essentially they all involve storing the pixels values along with other relevant information about the image. The image format we will be using is called PGM or Portable Gray Map.

The extended format for a PGM file is given in Appendix of this document. For our example images we have a simplified PGM format. A typical image file will have data as shown below

```
P2
64 64
255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
```

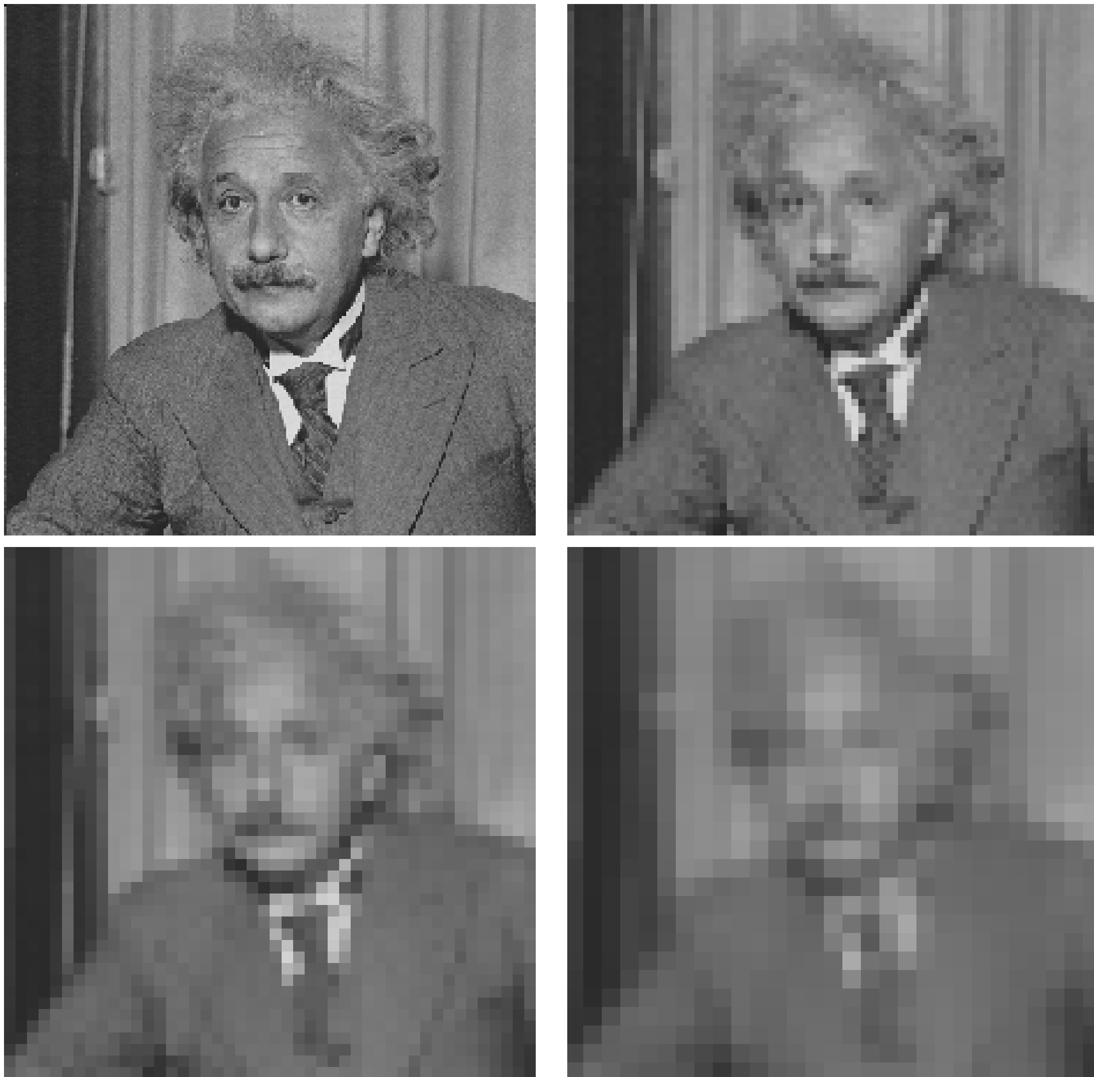

Figure 1.2: Images at different resolutions. Note the pixelization effect.

The first line is a "magic number" for identifying the file type. A pgm file's magic number is the two characters "P2".

The second line contains two numbers denoting the image width and image height, respectively.

The third line has the maximum number of gray levels which is, in our case, 255.

The numbers in fourth line onwards are the pixel values. The number 255 denotes a white pixel and 0 denotes a black pixel. There are a total of (width * height) gray values. The pixels are stored in row-scanned order, starting at the top-left corner of

the image, proceeding in normal English reading order.

1.7 How do I "see" images on the computer?

You would be using the PCs in ENB 118 or 116 to see the images on screen. The software on the PCs use a different format to store images. So first you have to convert the PGM image files into PC format and then FTP them over to the PCs and view them. The steps are:

1. Type on suntan:

   ```
   /home/sunburn/cs.dept/sarkar/EEL_4851/convert block.pgm block.pcx
   ```

   This will convert the PGM image `block.pgm` into a PCX formatted image file `block.pcx`

2. ftp `block.pcx` over to the local PC.

3. Use the Paintbrush software to read in `block.pcx` and view it.

1.8 How do I get a print out of the image?

The `.pcx` formatted image can be imported into a Microsoft Word or a WordPerfect document and printed along with the document.

If you want independent print out of the images, then you can again use suntan to convert the PGM file into a Postscript file which can be sent to the printer. The steps are (all on suntan)

```
convert block.pgm block.eps

lp -d lw118 block.eps
```

2. Assignment1: Arrays (Due at 3:30 pm on Sept 25)

What is the point of this assignment? The goal of this assignment is to familiarize you with arrays. You will have to write Ada code to read, write and compute simple statistics about images.

What much code do I have to write? Your code writing task can be divided into two parts:

1. You have to write a package to represent the image abstract data type implemented using arrays. The ADT should conform to the specifications in Fig. 2.1.

2. Write an application program which uses the above Image ADT to compute statistics about the pixel values such as the mean pixel value, the number of white pixels, the number of black pixels, and the average number of black pixels in each row. The application should have a menu interface with options to read in an image file, save to an image file, and allow the user to set pixel values at specified locations in the image.

What should I submit? For details about what and how to submit see Chapter 1.

```
with text_io;

package Image_Data_Type is

-- This package implements an ADT to represent images

-- Structure:  The image is represented as an array of pixel values

   OUT_OF_BOUNDS : Exception;    -- Raised if an attempt is made to access
                                 -- image pixels outside the image

   type Image_Array is private;
-------------------------------------------------------------------------------
   procedure Read (File : in  Text_IO.File_Type;
                   Image: out Image_Array);
   -- Function:        Reads in the Image from File
   -- Preconditions:   None
   -- Postconditions:  All the image pixels are read in and the image
   --                  width and height recorded in the appropriate fields.
--------------------------------------------------------------------------------
   procedure Save (File : in  Text_IO.File_Type;
                   Image: in Image_Array);
   -- Function:        Write out the Image to the File in PGM format
   -- Preconditions:   None
   -- Postconditions:  The written image file is in PGM format

--------------------------------------------------------------------------------
   function Get_Pixel (Image:in Image_Array;
                       Row: positive;
                       Col: positive) return integer;

   -- Function:   Returns the pixel value at Image(Row, Col)
   -- Preconditions:   None
   -- Postconditions:  Should not change the Image
   -- Exception:  OUT_OF_BOUNDS exception raised if (Row, Col) falls
   --             outside the image
```

```
-----------------------------------------------------------------------------
   procedure Set_Pixel (Image:in out Image_Array;
                        Row: positive;
                        Col: positive;
                        Value: integer);

   -- Function:   Sets the pixel value at Image(Row, Col) to Value
   -- Preconditions:   None
   -- Postconditions:  Should change the Image at (Row, Col)
   -- Exception:  OUT_OF_BOUNDS exception raised if (Row, Col) falls
   --              outside the image
-----------------------------------------------------------------------------
   procedure Get_Size (Image:in Image_Array;
                        Row_Num: out positive;
                        Col_Num: out positive);

   -- Function:   Returns the total number of Rows and Columns in the Image
   -- Preconditions:   None
   -- Postconditions:  The values are returned through the procedure parameters
-----------------------------------------------------------------------------
private

  type Image_array_type is array(1..250,1..250) of integer;

  type Image_Array is record
    Rows: positive; -- stores the total number of rows in the image
    Cols: positive; -- stores the total number of cols in the image
    Image: Image_array_type;
  end record;

end Image_Data_Type;
```

Figure 2.1: Specification of the Image ADT.

3. Assignment 2: Stacks and Queues (Due 3:30 pm on Oct. 16)

Goal: The goal of this assignment is to use stack and queues to solve the region growing problem in image processing. Given a starting point, the task is to collect all pixels that has the same pixel value as the starting point. The only restriction is that the final set of pixel should be one connected component.

For example, consider the 16 by 16 image shown in Fig 3.1. There are two connected components. If the user provides the initial starting point as (4, 6) then the set of connected black pixels in the upper left part of the image should be chosen. In your assignment, you will mark these pixels in a new blank image with a value of 0. Thus for this example, you will create a separate image with only the figure on the left side of the image.

Algorithm: Each pixel in an image has 8 neighbors. For example, the pixel (4, 6) has the pixels at locations {(4, 5), (4, 7), (5, 5), (5, 6), (5, 7), (3, 5), (3, 6), (3, 7) } as neighbors. Not all the neighbors have the same pixel value. We start by considering the neighbors which have the same value as the starting pixels. There are two ways to proceed: depth first or breadth first. In depth first search, you recursively follow one neighbor of a pixel. In breadth first search you process all the neighbors of one pixel before proceeding to the next one. The pseudo code of the algorithms are as follows:

```
Breadth_First_Search (Input_Image, Output_Image, Start_x, Start_y)

  Q: Queue of (x, y) coordinates of a pixel
  px, py: integers;

  Initialize Q to null.
  Set all pixels of the Output_Image to 255 (white).
  Enqueue(Q, (Start_x, Start_y));
  while (Q is not null) {
    (px, py) = Dequeue(Q);
    Set_Pixel(Output_Image, px, py) = 0;
    for each neighbor, (nx, ny), of (px, py) {
       if (Get_Pixel(Input_Image, nx, ny) ==
             Get_Pixel(Input_Image, px,  py)) and
          (Get_Pixel(Output_Image, nx, ny) == 255)
          Enqueue(Q, nx, ny);
    }
```

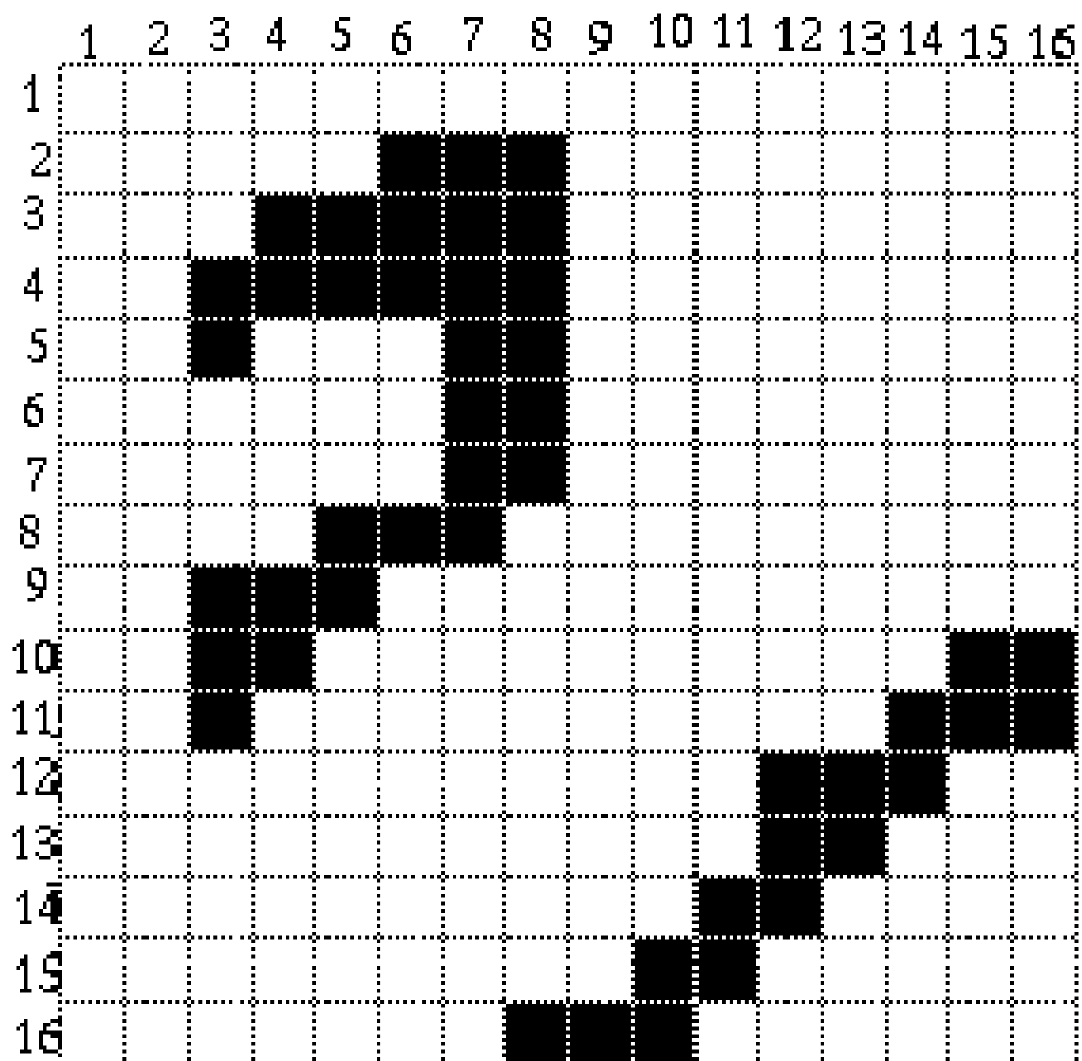

Figure 3.1: An example image with two connected components. The image size is 16 by 16.

```
   }
end Breadth_First_Search;


Depth_First_Search (Input_Image, Output_Image, Start_x, Start_y)

  S: Stack of (x, y) coordinates of a pixel
  px, py: integers;

  Initialize S to null.
  Set all pixels of the Output_Image to 255 (white).
  Push(S, (Start_x, Start_y));
  while (S is not null) {
    (px, py) = Pop(S);
    Set_Pixel(Output_Image, px, py) = 0;
    for each neighbor, (nx, ny), of (px, py) {
       if (Get_Pixel(Input_Image, nx, ny) ==
             Get_Pixel(Input_Image, px,  py)) and
          (Get_Pixel(Output_Image, nx, ny) == 255)
          Push(S, nx, ny);
    }
   }
end Depth_First_Search;
```

What to code? You have to write an Ada application program to implement the above two algorithms. The application should use the generic queue and stack packages from the text book and the image package you developed in the first assignment. The application should have an menu based interface which allows the user to select the search algorithm, to specify the image, and to specify the starting pixel for the region growing.

What to submit? For details about what and how to submit see Chapter 1. Make sure you include outputs which show the states of the Output_Image after 10, 50, 100, and 500 iterations of the depth first search and the breadth first search on the bull image (`bull.pgm`) with a starting point of (60,60).

4. Assignment3: Linked Lists (Due 3:30 pm on Oct. 30)

Goal: In this assignment you will use linked lists (actually a list of lists) to encode an image in a compact format called the run length code and to perform operations on this run length encoded image.

Algorithm: Run Length Coding (RLC) is a very compact way of encoding images for storage and transmission. The idea is to use the redundancy in the pixel value information among neighboring pixels to reduce the amount of information that needs to be stored. The algorithm proceeds row-wise and stores the column indices of contiguous segments of black (0) pixels. For example, the image shown in Fig. 4.1 has the RLC as follows:

```
16 16
-1
6 8 -1
4 8 -1
3 8 -1
3 3  7 8 -1
7 8 -1
7 8 -1
5 7 -1
3 5 -1
3 4 15 16 -1
3 3 14 16 -1
12 14 -1
12 13 -1
11 12 -1
10 11 -1
8 10 -1
```

The first line has the image width and height, respectively. Second line onwards, each line correspond to the image rows. In each line we store the column indices for contiguous segments of black pixels. For example, in row 3 there is a set of black pixels from column 4 to column 8, hence, we have `4 8` in the fourth line of the RLC. The `-1`s demarcate the rows. If there are more than one contiguous black segments in a row then we store the start and end indices of each segment in order as for rows 5, 10 and 11.

What to code? Your task is to write code to implement the above algorithm and to perform simple tasks using the RLC. The code should have the following parts and should have an user friendly menu based interface.

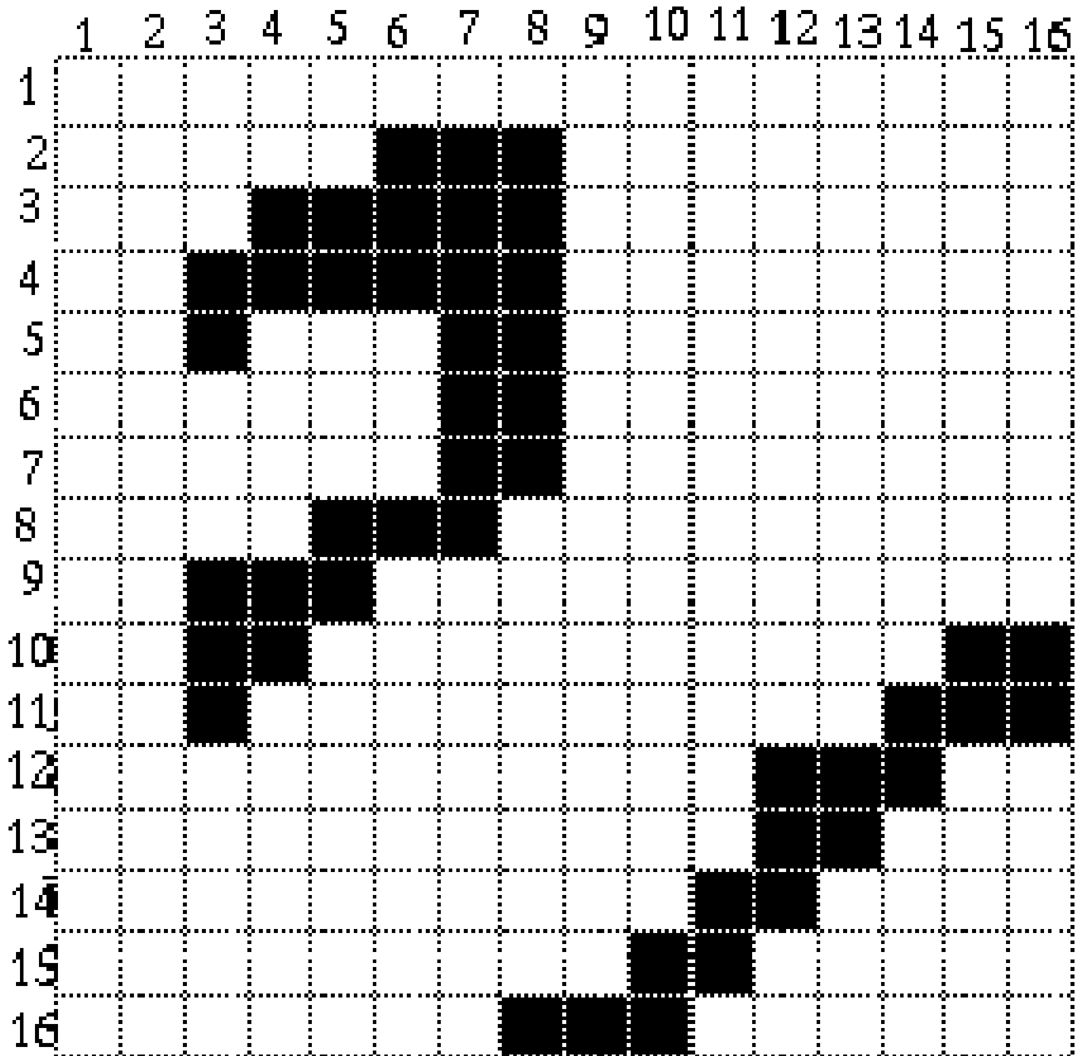

Figure 4.1: An example 16 by 16 image.

1. An abstract data type package which implements the linked list of lists data structures. The operations should allow you to add elements to a list specified by its index, delete elements from a specified list, access elements in a list based on index.

2. An application code which uses the above ADT and the Image ADT from the first assignment to do the following:

   (a) To convert a PGM formatted image file to RLC and to store the RLC in a file.

   (b) To read a RLC formatted image and convert to a PGM formatted file.

   (c) From the RLC compute the number of black pixels in each row. Display the average and the standard deviation of the number of black pixels per row.

   (d) Perform the image inversion operation *using the RLC* (Note: Do not use the array based image representation for this operation.) Convert the transformed RLC to PGM and store the image. Each pixel on the output image should be the opposite of that in the input image. If a pixel had a value of 255 then it will become 0.

What to submit? For details about what and how to submit see Chapter 1.

5. Assignment 4: Trees (Due 3:30 pm on Nov 20)

Goal: The goal of this assignment is to introduce you to the `quad tree` data structure which offers a very efficient methodology for encoding images.

Algorithm: Nodes in a quad tree represents a square region of the image. Each node has three colors, white, black, or gray depending on whether the region has all white, all black, or a mixture of black or white (a "gray" region) pixels. Each "gray" region is broken into four equal parts. Thus, every gray node has *four* children representing these four parts which are also labeled white, black, or gray depending on the status of the pixels in the subregion. The process is repeated until there are no gray nodes without children. Note that the root node represents the whole image and the leaf nodes are either black or white. An example quad tree is shown in Fig. 5.1.

What to code? You will write code to compute the quad tree of an image and perform simple logical operations on them using the algorithms discussed in class.

1. You have to implement the quad tree abstract data structure with operations to compute the quad tree from an image, to compute an image from a quad tree representation, to OR two quad trees, to perform logical NOT of a quad tree, and to display the pre-order and the post-order traversal of the tree. (The OR and the NOT operations will be discussed in class.)
2. You should provide a menu driven interface which allows the user to choose the image files, to specify operations to perform, and to save the resultant image.

What to submit? For details about what and how to submit see Chapter 1.

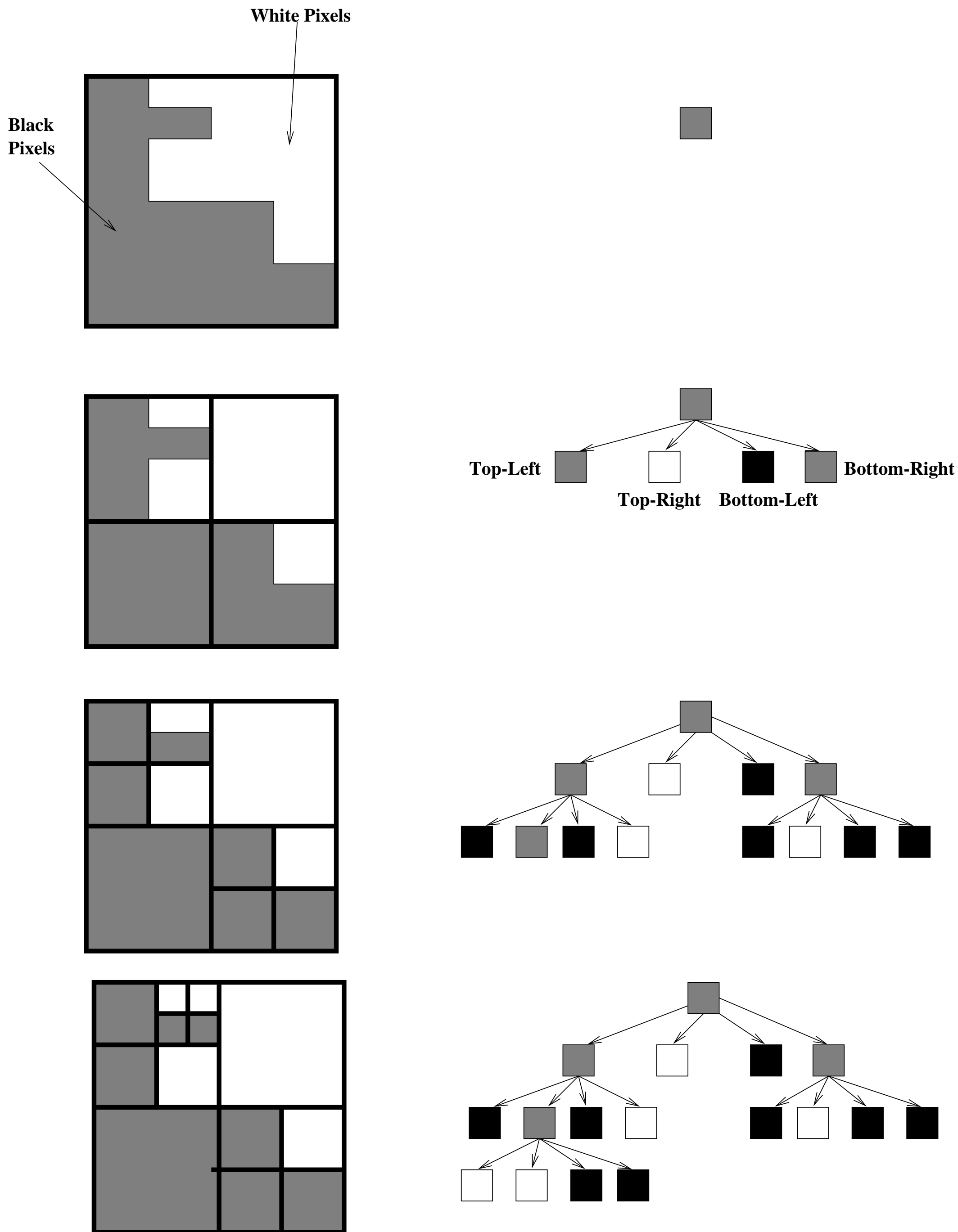

Figure 5.1: Stages through the building of a quad tree. The image is shown on the left and the quad tree is shown on the right. The regions are marked in dark outlines. Note that at each stage we break regions which have a mixture of black and white pixels. (Adapted from Jain, Kasturi, and Schunck)

6. Assignment 5: Hashing (Due 3:30 pm on Dec 9)

Goal: The goal of this assignment is to use hashing to address the problem of 2D shape recognition. We will attempt to design a very simple pattern recognition algorithm.

Algorithm: The data in our case are the black and white images in our image set. We would like to pre-store a set of images in the hash table and then given a "new" image we would like to answer if it exists in the database or not. For this, we should be able to extract a key index for every image. All pixels should be involved in calculating the key index so that the probability of collision is minimized. We will adopt the following strategy.

First, we will compute the horizontal and vertical projections of the images. The horizontal projection of an image is basically an array whose each cell corresponds to an image row and stores the number of black pixels in that row. Similarly the vertical projection is an array which stores the number of black pixels in each column. An example vertical and horizontal projection is shown in Fig. 6.1

Next, we use these horizontal and vertical projection arrays to compute the key as discussed in class. We basically concatenate the horizontal and vertical array and use four digit folding followed by a modulus operation with the hash table size.

What to code? The coding for this assignment can be divided into the following parts:

1. Write a generic hashing package that will accept any element types and any hashing function. The package will include the standard procedures that go with a hashed ADT such as functions to add, delete, search, and display the hash table. Use the quadratic collision resolution scheme.

2. Modify the Image ADT to include a function which computes the hash key index using the vertical and horizontal projection as discussed above.

3. Design an application which uses the above two packages to implement a mini-recognition module. The program should ask the user to read in a set of "database" images into the hash table. The hash table should store the image filename at the key index. For this problem assume an hash table size of 467. Given a new image file name, the program should search for it in the hash table and display its key location and the number of collisions encountered in the process.

What to submit? For details about what and how to submit see Chapter 1.

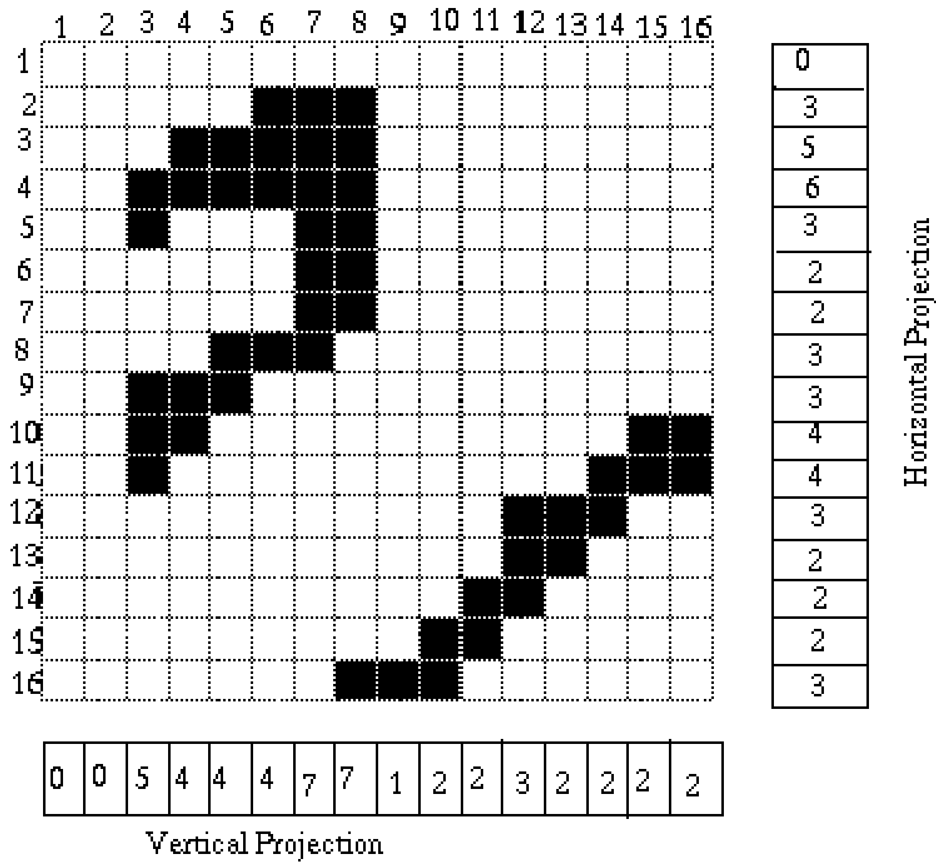

Figure 6.1: Horizontal and vertical projection of the example image.

7. Appendix

7.1 PGM format for images

```
pgm(5)              Headers, Tables, and Macros              pgm(5)



NAME
     pgm - portable gray-map file format

DESCRIPTION
     The portable gray-map format is a lowest  common  denominator
     gray-scale file format.  The definition is as follows:

     - A "magic number" for identifying the  file  type.   A  pgm
       file's magic number is the two characters "P2".

     - Whitespace (blanks, TABs, CRs, LFs).

     - A width, formatted as ASCII characters in decimal.

     - Whitespace.

     - A height, again in ASCII decimal.

     - Whitespace.

     - The maximum gray value, again in ASCII decimal.

     - Whitespace.

     - Width * height gray values, each in ASCII decimal, between
       0  and  the  specified  maximum  value,  separated by whi-
       tespace, starting at the top-left corner of  the  graymap,
       proceding  in  normal English reading order.  A value of 0
       means black, and the maximum value means white.

     - Characters from a "#" to the next end-of-line are  ignored
```

  (comments).

- No line should be longer than 70 characters.

Here is an example of a small graymap in this format:

```
P2
# feep.pgm
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Programs that read this format should be as lenient as possible, accepting anything that looks remotely like a graymap.

There is also a variant on the format, available by setting the RAWBITS option at compile time. This variant is different in the following ways:

- The "magic number" is "P5" instead of "P2".

- The gray values are stored as plain bytes, instead of ASCII decimal.

- No whitespace is allowed in the grays section, and only a single character of whitespace (typically a newline) is allowed after the maxval.

- The files are smaller and many times faster to read and write.

Note that this raw format can only be used for maxvals less than or equal to 255. If you use the _Φp_Φg_Φm library and try to write a file with a larger maxval, it will automatically

fall back on the slower but more general plain format.

SEE ALSO

fitstopgm(1), fstopgm(1), hipstopgm(1), lispmtopgm(1), psidtopgm(1), rawtopgm(1), pgmbentley(1), pgmedge(1), pgmenhance(1), pgmhist(1), pgmnorm(1), pgmoil(1), pgmramp(1), pgmtofits(1), pgmtofs(1), pgmtolispm(1), pgmtopbm(1), pgmtops(1), pnm(5), pbm(5), ppm(5)

AUTHOR

Copyright (C) 1989, 1991 by Jef Poskanzer.