Data Structures
Assignment 2: Due on March 19, 2001 at 12:30pm

(a) **(b)** **(c)**

Figure 1: Images of the walkway near ENB at two time instants are shown in (a) and (b). Difference image, thresholded with a value of 40 gray levels, is shown in (c).

I. Objectives

- You will learn the use the stack and the queue ADTs.
- You will learn how to design a new ADT namely the `ListOfLists` ADT.
- You will learn about depth-first and breadth-first searches.
- You will learn about connected component based coloring operation (akin to "flood fill") in image processing.
- You will learn about source code management in Unix using Makefiles.
- You will reuse the code you wrote from your first assignment. (If you had followed good software programming habits such as modularity, information hiding etc then you would not have much difficulty in this aspect.)
- You will learn to document your programming experience and to describe your program.

II. Programming Goals

This assignment will build on your first assignment, but is much larger in scope than your first assignment, so plan early. The input to this assignment would be the morphologically processed, thresholded, difference image from the first assignment. The output would be an image with each connected black region labeled by a unique number.

III. Overview of the modules

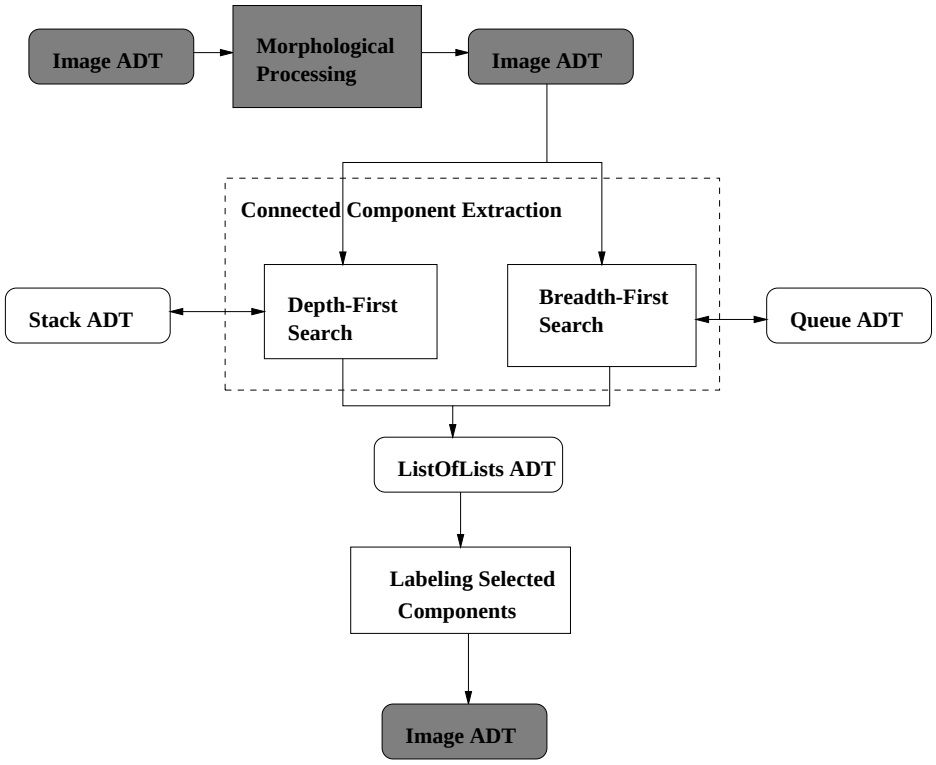

Figure 2: Block diagram of the interaction between data and algorithm.

Figure 2 shows the interaction of the ADTs and the algorithms for this assignment. The shaded blocks are the ones you will reuse from your first assignment. The algorithmic blocks consists of the connected component extraction block and the labeling of the selected components block. The connected component extraction block, in turn, is composed out of depth-first and breadth-first search modules. (These operations are discussed later in this document.) The new ADTs for this assignment are the: Stack ADT, Queue ADT, and the ListOfLists ADT. In the following sections, we first provide more details about the ADTs and then discuss the algorithms.

IV. StackADT

The following `stack.h` file specifies the Stack ADT and its implementation. You can use the material from your textbook to implement these functions in `stack.c`.

```
#ifndef Pixel_Has_Been_Defined
typedef struct pixel {
  int X, Y;
} Pixel;
#define Pixel_Has_Been_Defined
#endif

#ifndef Stack_Has_Been_Defined
/* You will have to choose an appropriate size for the maximum stack size*/
#define MAXSTACK 10000
/* You will have to change the following typedef statement to suit your purpose*/
typedef Pixel StackEntry;
typedef struct stack {
    int  top;
    StackEntry entry[MAXSTACK];
} Stack;
#define Stack_Has_Been_Defined
#endif

/* Push: push an item onto the stack.
   Pre:     The stack exists and it is not full.
   Post:   The argument item has been stored at the top of the stack.*/
void Push (StackEntry item, Stack *s);

/* Pop: pop an item from the stack.
   Pre:   The stack exists and it is not empty.
   Post: The item at the top of stack has been removed and returned in *item. */
void Pop(StackEntry *item, Stack *s);

/* StackEmpty: returns non-zero if the stack is empty.
   Pre:    The stack exists and it has been initialized.
   Post:  Return non-zero if the stack is empty; return zero, otherwise. */
Boolean StackEmpty(Stack *s);

/* StackFull: returns non-zero if the stack is full.
   Pre:     The stack exists and it has been initialized.
   Post:   Return non-zero if the stack is full; return zero, otherwise. */
Boolean StackFull(Stack *s);
```

```
/* CreateStack: initialize the stack to be empty.
   Pre:     None.
   Post:   The stack has been initialized to be empty. */
void CreateStack(Stack *s);
```

V. QueueADT

The following `queue.h` file specifies the QueueADT and its array based implementation. You can use the material from your textbook to implement these functions in `queue.c`.

```
#ifndef Pixel_Has_Been_Defined
typedef struct pixel {
  int X, Y;
} Pixel;
#define Pixel_Has_Been_Defined
#endif

#ifndef Queue_Has_Been_Defined
/* You will have to choose an appropriate size for the maximum queue size */
#define MAXQUEUE 10000
typedef Pixel QueueEntry;
typedef struct queue {
    int count;
    int front;
    int rear;
    QueueEntry entry[MAXQUEUE];
} Queue;
#define Queue_Has_Been_Defined
#endif

/* CreateQueue:  create the queue.
   Pre:   None.
   Post: The queue q has been initialized to be empty. */
void CreateQueue(Queue *q);

/* Append:  append (enqueue) an entry to the queue.
   Pre:   The queue q has been created and is not full.
   Post: The entry x has been stored in the queue as its last entry.
   Uses: QueueFull, Error. */
void Append(QueueEntry x, Queue *q);

/* Serve:  remove (dequeue) the first entry in the queue.
```

```
   Pre:   The queue q has been created and is not empty.
   Post: The first entry in the queue has been removed and returned as the value of x.
   Uses: QueueEmpty, Error. */
void Serve(QueueEntry *x, Queue *q);

/* QueueSize: return the number of entries in the queue.
   Pre:   The queue q has been created.
   Post: The function returns the number of entries in the queue q. */
int QueueSize(Queue *q);

/* QueueEmpty: returns non-zero if the queue is empty.
   Pre:   The queue q has been created.
   Post: The function returns non-zero if the queue q is empty,  zero otherwise. */
Boolean QueueEmpty(Queue *q);

/* QueueFull: returns non-zero if the queue is full.
   Pre:   The queue q has been created.
   Post: The function returns non-zero if the queue is full, zero otherwise. */
Boolean QueueFull(Queue *q);
```

VI. List Of Lists ADT

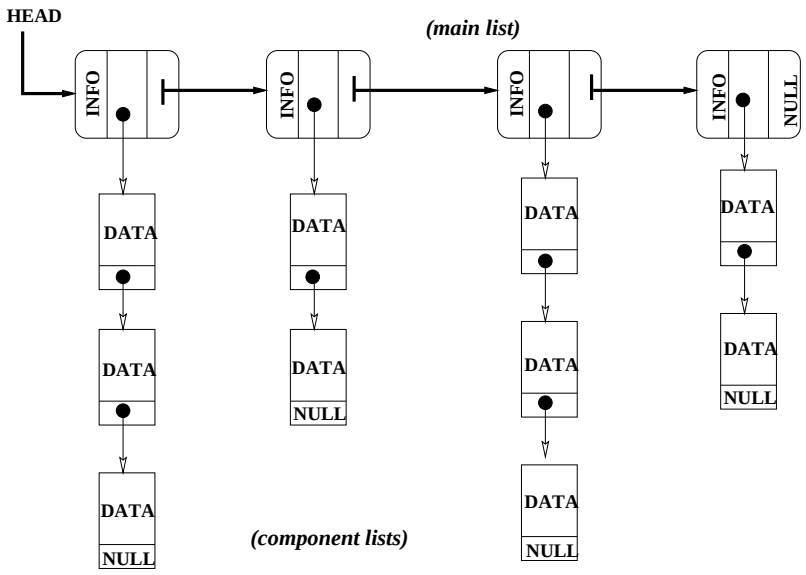

Figure 3: Schematic of the structure of the ListOfLists ADT.

This is a new ADT that you have to design and implement. Partial specifications are as follows. The component objects of this ADT are of two types, lists and individual data objects, nested within each other.

The structure of the ADT is depicted in Figure 3. At the basic level we have individual data objects that are strung together into multiple lists. These lists are then strung together in another list, thus giving us a list of lists structures. For purposes of discussion we will refer to one list as the `main` list and the others as components lists, as labeled in the figure.

The operations that would have to be supported for this ADT are the following.

- `Clear` : This procedure removes all elements from Main list and the Component lists.
- `CreateNewComponent` : Inserts a new node at the head of the Main list with a Component list of length zero. The info field of the new Main list node is set to zero.
- `InsertData` : Insert data at the head of the Component list of the first node in the Main list. The info field of the new Main list node is incremented by one.
- `Traverse` : A function that will allow you to traverse the List of Lists structure and perform user specified operations, such as display on screen or write to an Image ADT, at each data item identified by the component number in the main list.

VII. Connected Component Extraction

Consider the 16 by 16 image shown in Fig 4. There are two connected black components. You will collect these components into a List Of Lists structure, where each connected component is a list of pixels. Thus for the image in Fig 4, you would have a List of Lists structure with two component lists, one for the component in the top-left connected component and the other for the bottom-right component.

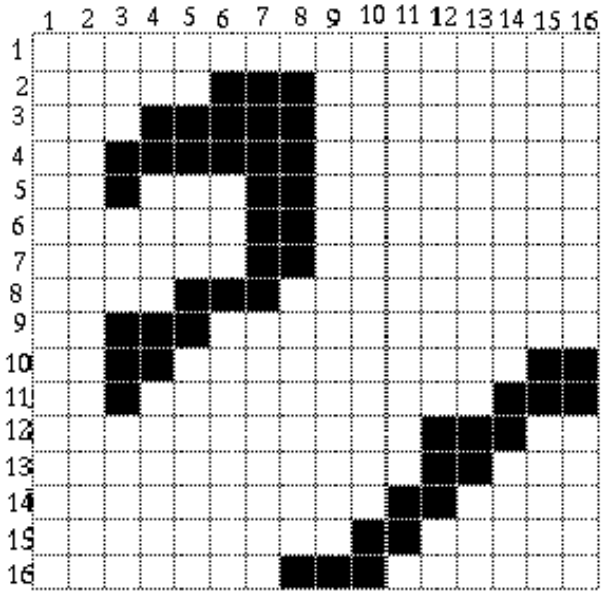

Figure 4: An example image with two connected components. The image size is 16 by 16.

Algorithm: You will build the connected component extraction algorithm around a seed region growing algorithm. Given a starting pixel with black value (the seed) the region growing algorithm finds all pixels that are black and connected to it. The idea is to iteratively invoke this seed region growing algorithm one for each of the black regions. The region growing algorithm can be described as follows.

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 are black. We only consider the neighboring pixels that are black. 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 codes of the algorithms are as follows:

```
Breadth_First_Search (Input_Image, ListOfLists, Start_x, Start_y)
  Q: Queue of (x, y) coordinates of a pixel
  px, py: integers;

  Initialize Q to null.
  Enqueue(Q, (Start_x, Start_y));

  while (Q is not null) {
    (px, py) = Dequeue(Q);
    InsertData, i.e. (px, py) into ListOfLists;

    for each neighbor, (nx, ny), of (px, py) {
       if (Input_Image(nx, ny) == 0) {
         Input_Image(nx, ny) == 1; /* A label to indicate that the pixel has been enqueued*/
         Enqueue(Q, nx, ny);
       }
    }
  }
end Breadth_First_Search;


Depth_First_Search (Input_Image, ListOfLists, Start_x, Start_y)
  S: Stack of (x, y) coordinates of a pixel
  px, py: integers;

  Initialize S to null.
  Push(S, (Start_x, Start_y));

  while (S is not null) {
    (px, py) = Pop(S);
    InsertData, i.e. (px, py) into ListOfLists;

    for each neighbor, (nx, ny), of (px, py) {
       if (Input_Image(nx, ny) ==  0 {
         Input_Image(nx, ny) == 1; /* A label to indicate that the pixel is in stack*/
         Push(S, nx, ny);
       }
    }
  }
end Depth_First_Search;
```

The above algorithms will extract all pixels with the same gray level and connected to the pixel at (`Start_x`, `Start_y`). You have to iteratively invoke the above subroutines to process all the black regions in the image and extract them into a List Of Lists data structure.

VIII. Labeling Selected Component

From the List of Lists structure built by the connected component extraction algorithm, you have to create an output image that has pixels in each connected component marked with an unique gray level value. All pixels in the same connected component would have the same label. No two components will have the same label. The possible values of the labels range from 0 to 254. We, of course, are assuming that we do not have more than 255 connected components.

The user should be able to select the size (in pixels) of the smallest component to be saved.

IX. Source Code Files

Here is a summary of the source code files that you should expect to have.

- Your `Image.c` and `Image.h` files from first assignment
- Files `Stack.h` and `Stack.c` containing the specification and implementation of a stack to handle pixel co-ordinates.
- Files `Queue.h` and `Queue.c` containing the specification and implementation of an queue to handle pixel co-ordinates.
- Files `ListOfLists.h` and `ListOfLists.c` containing the specification and implementation of the List of Lists ADT.
- A file called `ProcessImage.c` containing the following functions
  1. Image subtracting, thresholding, morphological functions from your first assignment.
  2. Breadth First search routine
  3. Depth First search routine
  4. A function that invokes the breadth first and depth first search for each of the black components.
  5. Write to an image, components that are above a certain size.
- A file called `ProcessImage.h` containing the prototype definitions of the above functions
- A file called `Implementation.c` that contain the main routine, which calls functions to operate on the various data structures. The application should have an interface (menu based or command line) that allows the user to select the search algorithm, to specify the image, to label and save connected components above a certain size, and to output the number of black regions, with their sizes. **All user interactions should be done only from this file.**
- A `Makefile` to compile and manage the code.

X. Postscript

```
(Typical Makefile)

EXEC= application
LIBS  = -lm -lc
CC = gcc
OBJECTS = Image.o Stack.o Queue.o ListOfLists.o ProcessImage.o Implementation.o

$(EXEC): $(OBJECTS)
        $(CC)  -o $(EXEC) $(OBJECTS) $(LIBS)

Image.o: Image.c
        $(CC)  -c Image.c
Stack.o: Stack.c
        $(CC)  -c Stack.c
Queue.o: Queue.c
        $(CC)  -c Queue.c
ListOfLists.o: ListOfLists.c
        $(CC)  -c listOflists.c
ProcessImage.o: ProcessImage.c Stack.h Queue.h Image.h ListOfLists.h
        $(CC)  -c ProcessImage.c
Implementation.o: Implementation.c ProcessImage.h Image.h
        $(CC)  -c Implementation.c
```

What to submit? For details about what and how to submit, see Submission Guidelines.

There are up to 20% extra credit points if in your report you include (and discuss) images that demonstrate how the pixels are being labeled by your implementation of the depth-first and the breadth-first searches on an actual connected component.