Estructuras de Datos

Queue Data Structure

In This Session

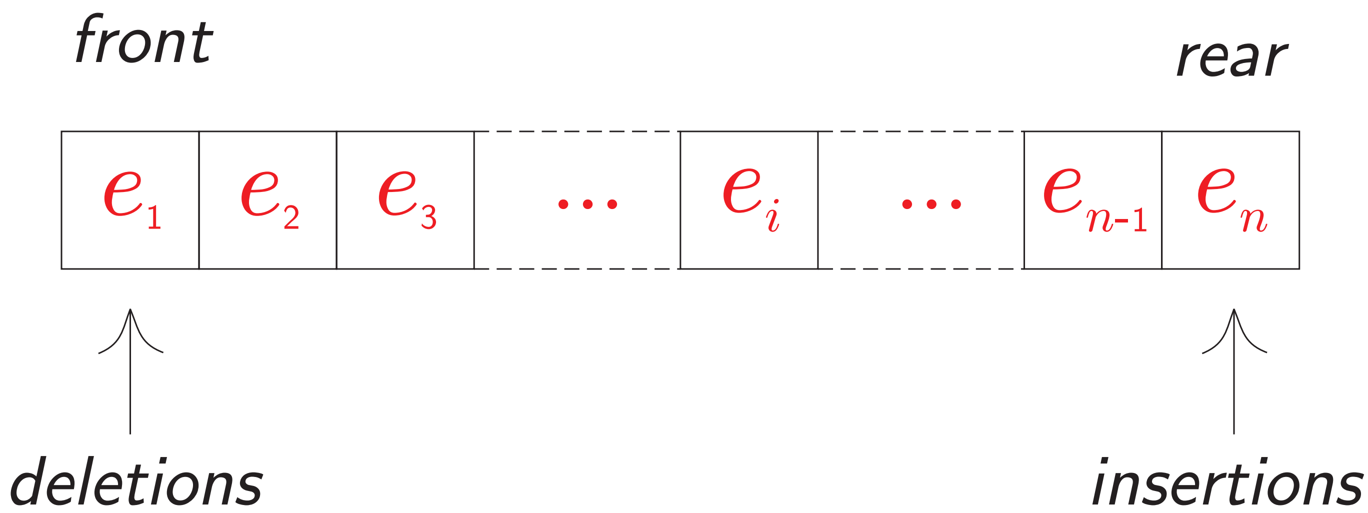
- **Queue Data Structure**
  - ▷ Array-based Representation
  - ▷ Linked Representation
  - ▷ Applications
    - ◇ Image-Component Labeling
    - ◇ Lee's Wire Router

Queue Data Structure

A *queue* is a special case of linear list where insertions and deletions take place at *different* ends

rear: end at which a new element is added.

front: end at which an element is deleted.



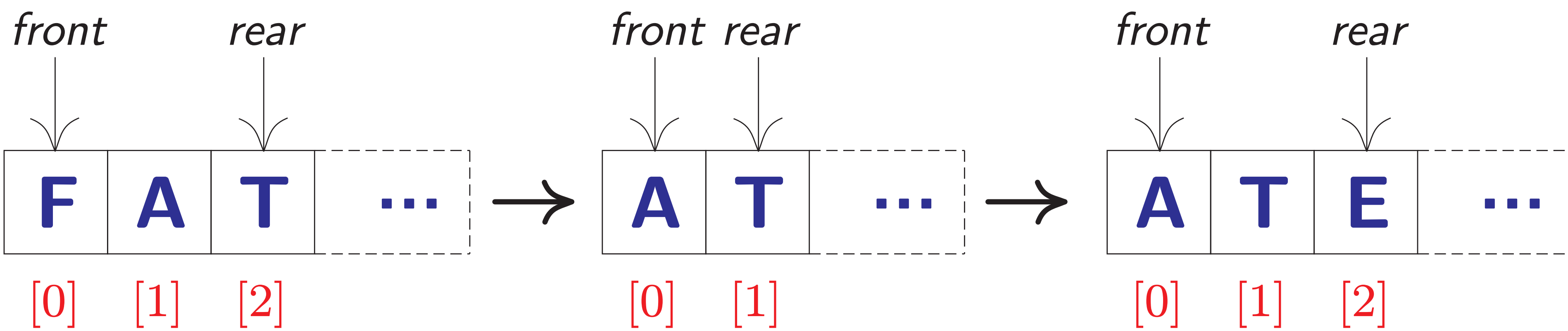
In other words, a queue is a FIFO (first-in-first-out) list

Queue Data Structure

– Array-based Representation –

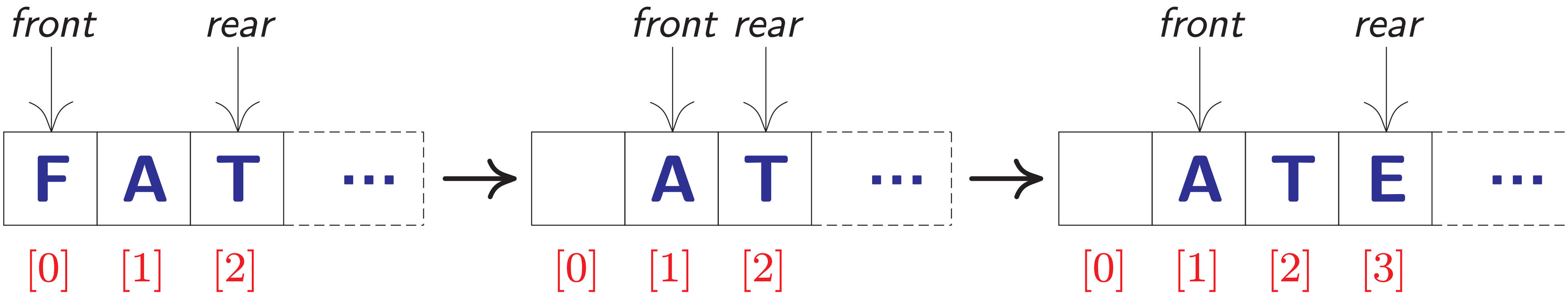
We can use three different approaches:

1) Using the formula $location(i) = i - 1$



- **Empty queue:** rear = -1
- **Addition:**
  rear = rear + 1
  queue[rear] = new_element
  $O(1)$ time
- **Deletion:**
  Shift all elements one position to the left.
  $\Theta(n)$ time

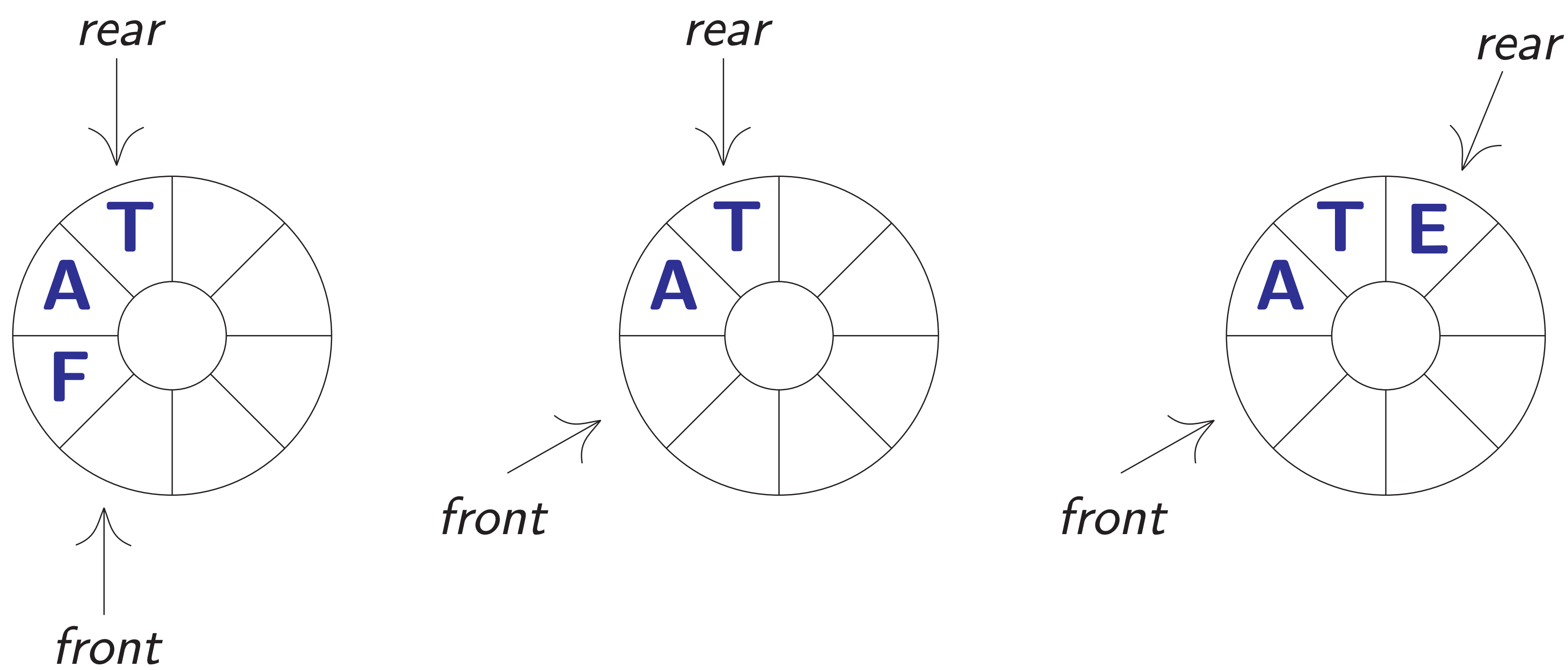
2) Using the formula $location(i) = location(1) + i - 1$



- **Empty queue:** rear < front
- front $= location(1)$; rear $= location(n)$
- **Deletions & Insertions:** $O(1)$ time

What happens if rear = MaxSize - 1 and front > 0 ?

3) Using the formula $location(i) = (location(1) + i - 1)\%MaxSize$



- front points one position before the position of the first element in the queue.
- **Empty queue:** front = rear
(initially front=rear=0)
- **Full queue:** (rear+1)%MaxSize = front

This approach is also called *Circular Queue*

arrayqueue.h

```
#ifndef ARRAYQUEUE
#define ARRAYQUEUE

#include <stdio.h>
#include <stdlib.h>
#define CAP 100

typedef struct Queue Queue;

struct Queue{
   int a[ CAP ];
   int front, rear, n;
   void (*put) ( Queue *, int );
   void (*delete) ( Queue * );
   int (*getFront) ( Queue * );
   int (*getRear) ( Queue * );
   void (*display) ( Queue * );
   int (*empty) ( Queue * );
};

void put( Queue * x, int e ){ ... }
void delete( Queue * x ){ ... }
int getFront( Queue * x ){ ... }
int getRear( Queue * x ){ ... }
void display( Queue * x ){ ... }
int empty( Queue * x ){ ... }
Queue createQueue( ){ ... }

#endif
```

createQueue

```
Queue createQueue( ){
   Queue q;
   q.front = q.rear = q.n = 0;
   q.put = &put;
   q.delete = &delete;
   q.getFront = &getFront;
   q.getRear = &getRear;
   q.display = &display;
   q.empty = &empty;
   return q;
}
```

put, delete

```
void put( Queue * x, int e ){
   if( x->n == CAP ){
      fprintf( stderr, "Error: Queue is full\n" );
      return;
   }
   x->rear = ( x->rear + 1 ) % CAP;
   x->a[ x->rear ] = e;
   x->n++;
   return;
}


void delete( Queue * x ){
   if( empty( x ) ){
      fprintf( stderr, "Error: Queue is empty\n" );
      return;
   }
   x->front = ( x->front + 1 ) % CAP;
   x->n--;
   return;
}
```

getFront, getRear

```
int getFront( Queue * x ){
   if( empty( x ) ){
      fprintf( stderr, "Error:␣Queue␣is␣empty\n" );
      exit( 1 );
   }
   return x->a[ ( x->front + 1 ) % CAP ];
}
```

```
int getRear( Queue * x ){
   if( empty( x ) ){
      fprintf( stderr, "Error:␣Queue␣is␣empty\n" );
      exit( 1 );
   }
   return x->a[ x->rear ];
}
```

display, empty

```
void display( Queue * x ){
   int i;
   printf( "Queue:␣[front]␣" );
   for( i = 1; i <= x->n; i++ )
      printf( "\%d␣", x->a[ ( x->front + i ) % CAP ] );
   printf( "[rear]\n" );
   return;
}


int empty( Queue * x ){
   return !x->n;
}
```

testqueue.c

```
#include <stdio.h>
#include "arrayqueue.h"

int main(){
   Queue maria = createQueue( );
   maria.put( &maria, 100 );
   maria.put( &maria, 80 );
   maria.put( &maria, 15 );
   maria.put( &maria, 24 );
   maria.put( &maria, 135 );
   maria.display( &maria );
   maria.delete( &maria );
   maria.display( &maria );
   printf( "Front␣element␣is␣%d\n", maria.getFront( &maria ) );
   printf( "Rear␣element␣is␣%d\n", maria.getRear( &maria ) );
   while( !maria.empty( &maria ) ){
      printf( "Deleting␣front␣element␣%d\n", maria.getFront( &maria ) );
      maria.delete( &maria );
   }
   return 0;
}
```

Compilation and Running

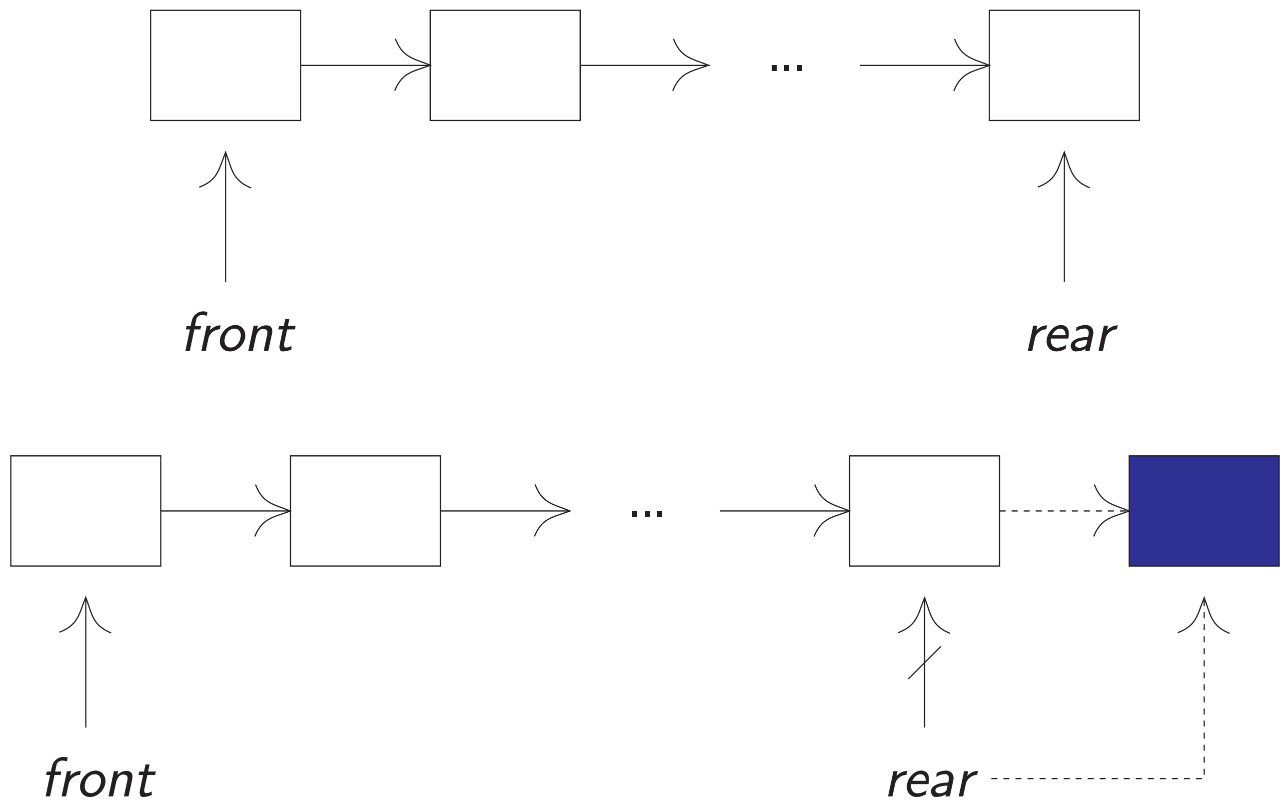
```
C:\Users\Yoan\Desktop\code\progs>gcc testqueue.c

C:\Users\Yoan\Desktop\code\progs>a
Queue: [front] 100 80 15 24 135 [rear]
Queue: [front] 80 15 24 135 [rear]
Front element is 80
Rear element is 135
Deleting front element 80
Deleting front element 15
Deleting front element 24
Deleting front element 135

C:\Users\Yoan\Desktop\code\progs>
```

Complexity of Operations

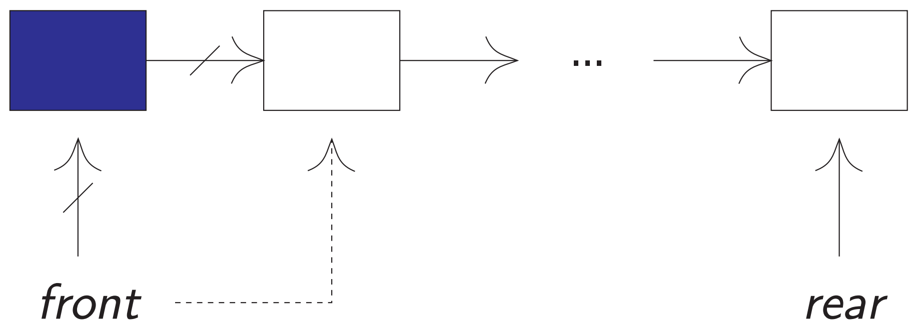
$$
\begin{aligned}
\text{put} &: \Theta(1)\\
\text{delete} &: \Theta(1)\\
\text{getFront} &: \Theta(1)\\
\text{getRear} &: \Theta(1)\\
\text{display} &: \Theta(n)\\
\text{empty} &: \Theta(1)\\
\text{createQueue} &: \Theta(1)
\end{aligned}
$$

Queue Data Structure

Linked Representation



(a) Addition



(b) Deletion

- **Empty queue:** front = null
- **Deletions & Insertions:** $\Theta(1)$ time

linkedqueue.h

```c
#ifndef LINKEDQUEUE
#define LINKEDQUEUE

#include <stdio.h>
#include <stdlib.h>

typedef struct Queue Queue;
typedef struct Node Node;

struct Node{
   int item;
   Node * next;
};

struct Queue{
   Node *front, *rear;
   void (*put) ( Queue *, int );
   void (*delete) ( Queue * );
   int (*getFront) ( Queue * );
   int (*getRear) ( Queue * );
   void (*display) ( Queue * );
   int (*empty) ( Queue * );
};
```

```c
void put( Queue * x, int e ){ ... }
void delete( Queue * x ){ ... }
int getFront( Queue * x ){ ... }
int getRear( Queue * x ){ ... }
void display( Queue * x ){ ... }
int empty( Queue * x ){ ... }
Queue createQueue( ){ ... }

#endif
```

createQueue

```
Queue createQueue( ){
   Queue q;
   q.front = q.rear = NULL;
   q.put = &put;
   q.delete = &delete;
   q.getFront = &getFront;
   q.getRear = &getRear;
   q.display = &display;
   q.empty = &empty;
   return q;
}
```

put, delete

```
void put( Queue * x, int e ){
   Node * y = malloc( sizeof( Node ) );
   y->item = e;
   y->next = NULL;
   if( empty( x ) )
      x->front = y;
   else
      x->rear->next = y;
   x->rear = y;
   return;
}


void delete( Queue * x ){
   if( empty( x ) ){
      fprintf( stderr, "Error:␣Queue␣is␣empty\n" );
      return;
   }
   Node * y = x->front;
   x->front = x->front->next;
   free( y );
   return;
}
```

getFront, getRear

```
int getFront( Queue * x ){
   if( empty( x ) ){
      fprintf( stderr, "Error:␣Queue␣is␣empty\n" );
      exit( 1 );
   }
   return x->front->item;
}


int getRear( Queue * x ){
   if( empty( x ) ){
      fprintf( stderr, "Error:␣Queue␣is␣empty\n" );
      exit( 1 );
   }
   return x->rear->item;
}
```

display, empty

```
void display( Queue * x ){
   Node * y = x->front;
   printf( "Queue:␣[front]␣" );
   while( y ){
      printf( "%d␣", y->item );
      y = y->next;
   }
   printf( "[rear]\n" );
   return;
}


int empty( Queue * x ){
   return x->front == NULL;
}
```

testqueue.c

```
#include <stdio.h>
#include "linkedqueue.h"

int main(){
   Queue maria = createQueue( );
   maria.put( &maria, 100 );
   maria.put( &maria, 80 );
   maria.put( &maria, 15 );
   maria.put( &maria, 24 );
   maria.put( &maria, 135 );
   maria.display( &maria );
   maria.delete( &maria );
   maria.display( &maria );
   printf( "Front␣element␣is␣%d\n", maria.getFront( &maria ) );
   printf( "Rear␣element␣is␣%d\n", maria.getRear( &maria ) );
   while( !maria.empty( &maria ) ){
      printf( "Deleting␣front␣element␣%d\n", maria.getFront( &maria ) );
      maria.delete( &maria );
   }
   return 0;
}
```

Compilation and Running

```
C:\Users\Yoan\Desktop\code\progs>gcc testqueue.c

C:\Users\Yoan\Desktop\code\progs>a
Queue: [front] 100 80 15 24 135 [rear]
Queue: [front] 80 15 24 135 [rear]
Front element is 80
Rear element is 135
Deleting front element 80
Deleting front element 15
Deleting front element 24
Deleting front element 135

C:\Users\Yoan\Desktop\code\progs>
```

Complexity of Operations

$$\begin{aligned} \text{put} &: \Theta(1) \\ \text{delete} &: \Theta(1) \\ \text{getFront} &: \Theta(1) \\ \text{getRear} &: \Theta(1) \\ \text{display} &: \Theta(n) \\ \text{empty} &: \Theta(1) \\ \text{createQueue} &: \Theta(1) \end{aligned}$$

Queue Application

– Image-Component Labelling –

(a) Input

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(b) Output

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

- Digitised image: $m \times m$ matrix of pixels (0,1). 0-pixel represents image background; 1-pixel represents a point on an image component.
- Two pixels are adjacent if one is to the left, above, right, or below the other.
- Two 1-pixels (component pixels) that are adjacent belong to the same image component.
- **Objective:** label the components pixels such that two pixels get the same label if and only if they are pixels of the same image component.

Image-Component Labelling
– Implementation –

imagecomponets.c

```
#include <stdio.h>
#include "arrayqueue.h"

int **pixel, size, i, j;

typedef struct Position Position;

struct Position{
   int row, col;
};

void inputImage( ){ ... }
void labelComponents( ){ ... }
void outputImage( ){ ... }

int main( ){
   inputImage( );
   labelComponents( );
   outputImage( );
   return 0;
}
```

inputImage

```
void inputImage( ){
   printf( "Enter␣image␣size:␣" );
   scanf( "%d", &size );
   pixel = malloc( ( size + 2 ) * sizeof( int * ) );
   for( i = 0; i < size + 2; i++ )
      pixel[i] = malloc( ( size + 2 ) * sizeof( int ) );
   printf( "Enter␣the␣pixel␣array␣in␣row-major␣order\n" );
   for( i = 1; i <= size; i++ )
      for( j = 1; j <= size; j++ )
         scanf( "%d", &pixel[ i ][ j ] );
   return;
}
```

labelComponents

```
void labelComponents( ){
   int numOfNbrs = 4, id = 1, r, c;
   Position nbr, here, offset[ 4 ] = {
      { 0, 1 },  // right
      { 1, 0 },  // down
      { 0, -1 }, // left
      { -1, 0 }  // up
   };
   for( i = 0; i <= size + 1; i++ ){
      pixel[ 0 ][ i ] = pixel[ size + 1 ][ i ] = 0;
      pixel[ i ][ 0 ] = pixel[ i ][ size + 1 ] = 0;
   }
   Queue qx = createQueue();
   Queue qy = createQueue();
   for( r = 1; r <= size; r++ )
      for( c = 1; c <= size; c++ )
         if( pixel[ r ][ c ] == 1 ){
            pixel[ r ][ c ] = ++id;
            here = (Position) { r, c };
            while( 1 ){
               for( i = 0; i < numOfNbrs; i++ ){
                  nbr.row = here.row + offset[ i ].row;
                  nbr.col = here.col + offset[ i ].col;
```

```
            if( pixel[ nbr.row ][ nbr.col ] == 1 ){
               pixel[ nbr.row ][ nbr.col ] = id;
               qx.put( &qx, nbr.row );
               qy.put( &qy, nbr.col );
            }
         }
         if( qx.empty( &qx ) ) break;
         here.row = qx.getFront( &qx );
         here.col = qy.getFront( &qy );
         qx.delete( &qx );
         qy.delete( &qy );
      }
   }
   return;
}
```

outputImage

```
void outputImage( ){
   printf( "\nThe␣labeled␣image␣is\n\n" );
   for( i = 1; i <= size; i++ ){
      for( j = 1; j <= size; j++ )
         printf( "%d␣", pixel[ i ][ j ] );
      printf( "\n" );
   }
}
```

imagecomponents.input

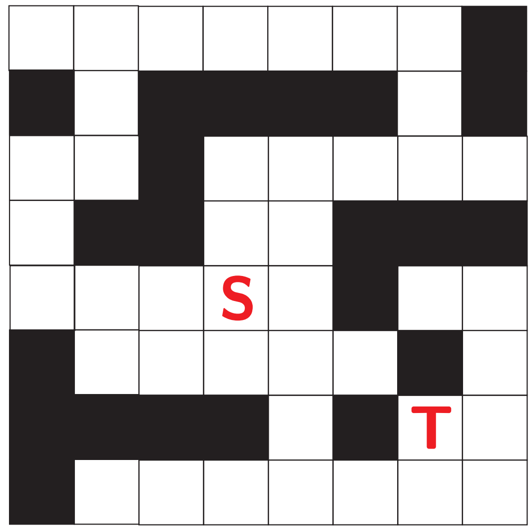
```
7
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 0 0
1 1 1 0 0 0 0
1 1 1 0 0 0 0
```

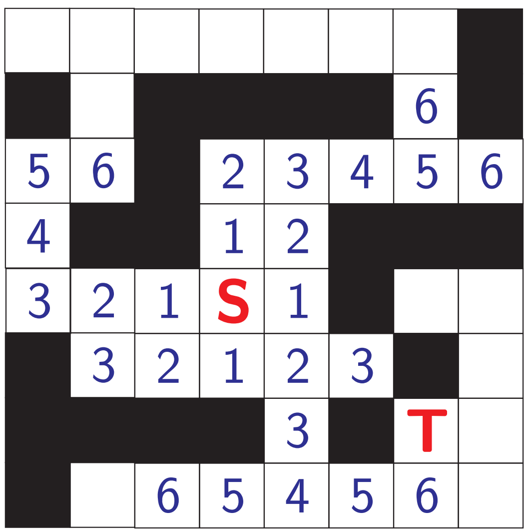
Compilation and Running

```
C:\Users\Yoan\Desktop\code\progs>gcc imagecomponents.c

C:\Users\Yoan\Desktop\code\progs>a < imagecomponents.input
Enter image size: Enter the pixel array in row-major order

The labeled image is

0 0 2 0 0 0 0
0 0 2 2 0 0 0
0 0 0 0 3 0 0
0 0 0 3 3 0 0
4 0 0 0 3 0 0
4 4 4 0 0 0 0
4 4 4 0 0 0 0
```

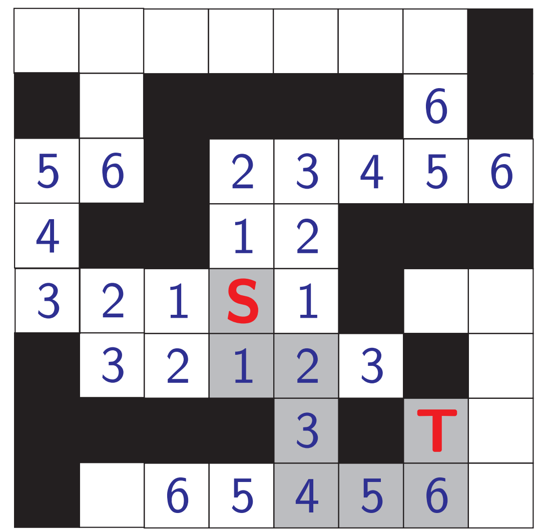
Queue Application
– Lee's Wire Router –

Find a path from S to T by *wave propagation*.





(a) Filing



(b) Retrace

Lee's Wire Router
– Implementation –

wirerouter.c

```
#include <stdio.h>
#include "arrayqueue.h"

int **grid, size, pathLength, i, j;

typedef struct Position Position;

struct Position{
   int row, col;
};

Position start, finish, path[100];

void inputData( ){ ... }
void findPath( ){ ... }
void outputPath( ){ ... }

int main( ){
   inputData( );
   if( findPath( ) ) outputPath( );
   else printf( "There␣is␣no␣wire␣path" );
   return 0;
}
```

inputData

```
void inputData( ){
   printf( "Enter␣grid␣size:␣" );
   scanf( "%d", &size );
   printf( "Enter␣the␣start␣position\n" );
   scanf( "%d␣%d", &start.row, &start.col );
   printf( "Enter␣the␣finish␣position\n" );
   scanf( "%d␣%d", &finish.row, &finish.col );
   grid = malloc( ( size + 2 ) * sizeof( int * ) );
   for( i = 0; i < size + 2; i++ )
      grid[i] = malloc( ( size + 2 ) * sizeof( int ) );
   printf( "Enter␣the␣grid␣array␣in␣row-major␣order\n" );
   for( i = 1; i <= size; i++ )
      for( j = 1; j <= size; j++ )
         scanf( "%d", &grid[ i ][ j ] );
   return;
}
```

findPath

```
void findPath( ){
   int numOfNbrs = 4;
   Position here, nbr, offset[ 4 ] = {
      { 0, 1 },  // right
      { 1, 0 },  // down
      { 0, -1 }, // left
      { -1, 0 }  // up
   };
   for( i = 0; i <= size + 1; i++ ){
      grid[ 0 ][ i ] = grid[ size + 1 ][ i ] = 1;
      grid[ i ][ 0 ] = grid[ i ][ size + 1 ] = 1;
   }
   if( ( start.row == finish.row ) && ( start.col == finish.col ) )
      return 1;
   here = (Position) { start.row, start.col };
   grid[ start.row ][ start.col ] = 2;
   Queue qx = createQueue();
   Queue qy = createQueue();
   while( 1 ){
      for( i = 0; i < numOfNbrs; i++ ){
         nbr.row = here.row + offset[ i ].row;
         nbr.col = here.col + offset[ i ].col;
         if( grid[ nbr.row ][ nbr.col ] == 0 ){
            grid[ nbr.row ][ nbr.col ] = grid[ here.row ][ here.col ] + 1;
            if( ( nbr.row == finish.row ) && ( nbr.col == finish.col ) )
```

```
               break;
            qx.put( &qx, nbr.row );
            qy.put( &qy, nbr.col );
         }
      }
      if( ( nbr.row == finish.row ) && ( nbr.col == finish.col ) )
         break;
      if( qx.empty( &qx ) ) return 0;
      here.row = qx.getFront( &qx );
      here.col = qy.getFront( &qy );
      qx.delete( &qx );
      qy.delete( &qy );
   }
   pathLength = grid[ finish.row ][ finish.col ] - 2;
   here = finish;
   for( j = pathLength - 1; j >= 0; j-- ){
      path[ j ] = here;
      for( i = 0; i < numOfNbrs; i++ ){
         nbr.row = here.row + offset[ i ].row;
         nbr.col = here.col + offset[ i ].col;
         if( grid[ nbr.row ][ nbr.col ] == j + 2 ) break;
      }
      here = (Position) { nbr.row, nbr.col };
   }
   return 1;
}
```

outputPath

```
void outputPath( ){
   printf( "The␣wire␣path␣is\n" );
   for( i = 0; i < pathLength; i++ )
      printf( "%d␣%d\n", path[ i ].row, path[ i ].col );
   return;
}
```

wirerouter.input

```
7
3 2
4 6
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 0 0
1 1 1 0 0 0 0
1 1 1 0 0 0 0
```

Compilation and Running

```
C:\Users\Yoan\Desktop\code\progs>gcc wirerouter.c

C:\Users\Yoan\Desktop\code\progs>a < wirerouter.input
Enter grid size: Enter the start position
Enter the finish position
Enter the grid array in row-major order
The wire path is
4 2
5 2
5 3
5 4
6 4
6 5
6 6
5 6
4 6
```