Queue ADT

March 16, 2000

1 The Queue ADT

Queue represent another form of linear data structure. As in the case of the list, the idea of a queue is to represent sequences of elements, all having the same type. As in the case of lists, the order in which elements are indicated is part of the definition (thus sequences with the same elements in different order represent distinct values).

The difference between queues and lists comes from the fact that, while lists allow arbitrary insertion and deletion of elements in any point of the sequence, queues allow insertion and deletion to occur only at opposite extremes of the sequence. Addition of new elements can be only performed at one fixed extreme of the sequence, while removal of elements can be only performed at the opposite extreme of the sequence. See Figure 1.

This sort of structure is usually indicated with the term FIFO (First in First out), as elements can be removed only in the same order as they are inserted (the first element inserted will the first to be removed, the second element inserted will be the second to be removed, etc.).

Thus, this data type relies on four key operations:

- enqueue which inserts a new element in the queue; the new element becomes the new tail of the queue;
- dequeue which removes one element from the queue; the element removed is the current head of the queue; the new head will be the immediately preceding element in the sequence;
- front and rear which simply read the value which is located respectively in the head and in the tail of the queue

Figure 2 shows with some examples the effect of the various operations on a queue.
Figure 2: Queue Example
2 Specification

The description in Figure 3 is the specification of the queue ADT.

AbstractDataType Queue
{
instances:
ordered list of elements from a base type; one end
of the sequence is called the front, while the opposite
end is called the rear
operations:
1. Constructors:
   CreateQueue
   input: nothing
   output: a new queue data object containing the empty sequence
2. Destructors:
   DeleteQueue
   input: an existing queue data object
   output: nothing
   effect: destroys the object given in input
3. Inspectors:
   EqualQueues
   input: two queue data objects
   output: a boolean value; TRUE if the two input objects contain
   the same values (same sequences of elements); FALSE otherwise
   EmptyQueue
   input: a queue data object
   output: boolean value; TRUE if the queue contains the empty sequence, FALSE otherwise
   QueueSize
   input: a queue data object
   output: an integer value identifying the length of the sequence currently
   stored in the input data object
   QueueFront
   input: a queue data object
   output: the element in the front position of the sequence stored in the queue
   QueueRear
   input: a queue data object
   output: the element in the rear position of the sequence stored in the queue
   PrintStack
   input: a queue data object
   output: none
   effect: prints on the screen the sequence of elements
   currently stored in the input data object
4. Modifiers
   Enqueue
   input: a queue data object; an element of the base type
   output: a boolean indicating whether the operation was successful
   effect: the element is added to the sequence in the queue, in the rear position
   Dequeue
   input: a queue data object
   output: a boolean indicating whether the operation was successful
   effect: the element in the front position in the sequence in the queue is removed
   AssignQueues
   input: two queue data objects
   output: none
   effect: the value of the second object is copied in the first object
}

Figure 3: Abstract Data Type specification for the Queue ADT

Set of values: the set of possible values is represented by the set of all possible sequences of elements; i.e., each value is of the form \( \langle a_1, \ldots, a_n \rangle \), where \( a_i \) identifies the \( i^{th} \) element in the sequence.

In the queue we need to identify, at every point in time, the two elements that are at the two extremes of the
stack. The first element in the queue is always indicated with the name *head* (see Figure 1). The last element in the queue is indicated with the name *tail*. Thus the *head* is the first element which has been inserted in the queue (the oldest element in the queue) while the tail is the last element inserted in the queue (the youngest element in the queue).

**Operations:**

- ** Constructors:** the only constructor needed is the one to create a new object of type queue.

  **CreateQueue**
  - preconditions: none
  - postconditions: returns a new object of type queue containing, as value, the empty queue (i.e., the sequence containing zero elements);

  This is illustrated in Figure 4. The figure also sketches the implementation via linked lists.

  ![Figure 4: Queue Creation](image)

  - **Destructors:** the only destructor needed is used to trash a queue which is not needed any longer.

  **DeleteQueue**
  - preconditions: receives an existing queue as input
  - postconditions: none

- **Inspectors:** the following inspectors are included in the type specification:

  1. **QueueSize** used to compute the size of a queue (i.e., the number of elements present in a given queue)
     - preconditions: receives a queue as input
     - postconditions: produces a number representing the count of the elements in the queue
  2. **EmptyQueue** used to verify whether a given queue is empty or not
     - preconditions: receives a queue as input
     - postconditions: produces true (i.e., 1 in C) if the queue is empty, false (i.e., 0 in C) otherwise
  3. **EqualQueues** used to verify whether two queues have the same value (i.e., they contain the same elements and in the same order).
     - preconditions: receives two queues as input
postconditions: returns true if the values of the two queues contain the same elements and in the same order, false otherwise.

4. **Front** returns the information which is currently stored as first element (the head) in the queue (i.e., the oldest element in the queue, which is also the element which is going to be removed in the next removal operation).
   
   - preconditions: receives a queue s
   - postconditions: if the queue is not empty, then it returns the first element in the queue s (the head), otherwise it gives an error

5. **Rear** returns the information which is currently stored as last element (the tail) in the queue (i.e., the youngest element in the queue, which is also the element which has been inserted in the last enqueue operation).
   
   - preconditions: receives a queue s
   - postconditions: if the queue is not empty, then it returns the last element in the queue s (the tail), otherwise it gives an error

- **Modifiers:** the following modifiers are included

1. **Enqueue** used to insert a new element at the end of the queue
   
   - preconditions: receives as input a queue s, and an element x
   - postconditions: if x is of the correct type (i.e., it has the same type as all the other elements in s), then the queue is modified introducing the element x in the last position of the queue (i.e., the element x becomes the new tail of the queue); otherwise it returns an error

   This is illustrated in Figure 5.

   ![Enqueue Operation Diagram](image)

   **Figure 5: Enqueue Operation**

2. **Dequeue** removes the first element from the queue (the head of the queue)
   
   - preconditions: receives as input a queue s
   - postconditions: if the queue is not empty, then the queue is modified by removing the element in last position in s (the head of s); otherwise it returns an error

   This is illustrated in Figure 6.

3. **AssignQueues** assign the value of a queue to another queue object
   
   - preconditions: receives as input two queue objects;
   - postconditions: copies the value of the second queue in the first queue object (thus making them identical—i.e., they have the same value)
3 Implementation using Linked Lists

Most of the ideas on how to implement queues using linked lists have been depicted in the previous figures. The Header file which contains the interface of the Queue ADT will contain the following instructions:

```c
#include "genlib.h"

/* Base type of the queue ADT */
typedef int ElementType;

/* Definition of the Queue data type */
/* a queue is thus a pointer to a queue descriptor */
typedef struct queue_type *Queue;
```

```c
/* Operations */

/* Constructors */
/* create new queue */
Queue CreateQueue();
```
void DeleteQueue (Queue s);

bool EmptyQueue (Queue s);

void PrintQueue (Queue q);

int QueueSize (Queue s);

ElementType QueueFront (Queue s);
ElementType QueueRear (Queue s);

void Enqueue (ElementType element, Queue s);

void Dequeue (Queue s);

void AssignQueues (Queue destination, Queue source);

The implementation part of the ADT is very similar to what we have seen in the case of stacks. This is reported in the following code:
/* QUEUE ADT - IMPLEMENTATION */
/* This implementation makes use of linked lists */
/*-----------------------------------------------*/

/* Needed header files */
/*-----------------------------------------------*/
#include <stdio.h>
#include <genlib.h>
#include <simpio.h>
#include "queue.h"    /* include interface */

/*-----------------------------------------------*/
/* Data Section */
/*-----------------------------------------------*/

/*-----------------------------------------------*/
/* Structure representing a node */
/*-----------------------------------------------*/
typedef struct node_type {
    ElementType data;  /* data part of node */
    struct node_type *next;  /* pointer to next element in queue */
} basic_node;

typedef struct node_type *Node;

/*-----------------------------------------------*/
/* Structure representing a queue */
/*-----------------------------------------------*/
typedef struct queue_type {
    struct node_type *head;  /* pointer to front of queue */
    struct node_type *tail;  /* pointer to rear of queue */
    int size;  /* count of elements in queue */
} basic_queue;

/*-----------------------------------------------*/
/* OPERATIONS */
/*-----------------------------------------------*/

/*-----------------------------------------------*/
/* CreateQueue: */
/*    no input; */
/*    returns a brand new and empty queue */
/*-----------------------------------------------*/

Queue CreateQueue()
{
Queue new;

/* allocate space for new queue */
new = malloc (sizeof(basic_queue));
if (new == NULL)
    Error("Creation of Queue has failed\n");
else
{
    /* need to initialize the fields of the queue descriptor */
    /* to describe an empty queue */
    new->size = 0;
    new->head = NULL;
    new->tail = NULL;
}
return new;
}

/***********************/
/* DeleteQueue      
    takes a queue as input; 
    destroys the given queue */
/***********************/
void DeleteQueue (Queue s)
{
    Node scan,temp;

    /* remove each element of queue */
    scan = s->head;
    while (scan != NULL)
    {
        temp = scan;
        scan = scan->next;
        free(temp);
    }

    /* need to simply release the memory used by the queue descriptor */
    free(s);
}

/***********************/
/* EmptyQueue       
    takes a queue as input; 
    returns true if queue empty */
/***********************/
bool EmptyQueue (Queue s)
{
    return (s->size == 0);
}
/*---------------------------------*/
/* QueueSize
   takes a queue as input;
   returns number of elements
   in the queue */
/*---------------------------------*/
int QueueSize (Queue s)
{
    return (s->size);
}

/*---------------------------------*/
/* QueueFront
   takes a queue as input;
   returns the element currently
   on the head of the queue */
/*---------------------------------*/
ElementType QueueFront (Queue s)
{
    if (EmptyQueue(s))
        Error("Attempting to access empty queue\n");
    else
        return (s->head->data);
}

/*---------------------------------*/
/* QueueRear
   takes a queue as input;
   returns the element currently
   on the tail of the queue */
/*---------------------------------*/
ElementType QueueRear (Queue s)
{
    if (EmptyQueue(s))
        Error("Attempting to access empty queue\n");
    else
        return (s->tail->data);
}

/*---------------------------------*/
/* Enqueue
   takes a queue as input;
   takes an element as input;
   modifies the queue by inserting
   the element as new tail */
/*---------------------------------*/
void Enqueue (ElementType element, Queue s)
{
    Node new;
    /* create new node */
    new = malloc(sizeof(basic_node));
    if (new == NULL)
        Error("cannot create new node for enqueue\n");
    else
    {
        new->data = element;
        new->next = NULL; /* it will be last element in queue */
        /* add element to the linked list */
        if (s->size == 0)
            s->head = new;
        else
            s->tail->next = new;
        /* update the tail of the queue */
        s->tail = new;
        s->size++;
    }
}

/*---------------------------------*/
/* Dequeue */
/* Dequeue takes a queue as input; */
/* modifies the queue by */
/* removing the current head */
/*---------------------------------*/

void Dequeue (Queue s)
{
    Node temp;
    if (EmptyQueue(s))
        Error("Cannot dequeue; queue is empty\n");
    else
    {
        temp = s->head;
        /* remove from linked list */
        if (s->size == 1)
        {
            s->tail = NULL;
            s->head = NULL;
        }
        else
            /*...*/
    }
{  
    s->head = s->head->next;  
}
    free(temp);  
    s->size--;  
}

/*---------------------------------*/
/* AssignQueues
   takes two queues as input;
   copies content of second
   queue into first queue   */
/*---------------------------------*/

void AssignQueues (Queue destination, Queue source)  
{
    int i;
    Node new,temp,scan;

    /* destroy old content of destination */
    new = destination->head;
    for (i=0; i < destination->size; i++)  
    {
        temp = new;
        new = new->next;
        free(temp);
    }

    destination->size = source->size;

    scan = source->head;
    new = malloc(sizeof(basic_node));
    new->data = scan->data;
    destination->head = new;
    new->next = NULL;
    for (i=1; i < source->size; i++)  
    {
        scan = scan->next;
        temp = malloc(sizeof(basic_node));
        temp->data = scan->data;
        temp->next = NULL;
        new->next = temp;
        new = temp;
    }
    destination->tail = new;
}
4 Implementation Using Arrays

The following implementation is for queue of integers. We consider the use of arrays for implementing the queue data structure. The intuition is to store the elements of the queue in consecutive locations in an array. The head index denotes the first element inserted, while the tail index denotes the last element inserted. This is illustrated in Figure 7.

The intuition is to use a circular array, as depicted in Figure 8.

The interface to the ADT, called `queue.h` is the same used for the linked list implementation. For the sake of simplicity the implementation does not allow extension of the array (this can be achieved as shown in the case of stacks and lists—by reallocating the array each time it fills up).

The second file contains the implementation of the ADT. The file is named `queue.c` and contains the following:

```c
/*---------------------------------------------------*/
/* QUEUE ADT - IMPLEMENTATION */
/* This implementation makes use of arrays */
/*---------------------------------------------------*/

/*---------------------------------------------------*/
/* Needed header files */
/*---------------------------------------------------*/

#include <stdio.h>

#include <stdio.h>
```
#include "genlib.h"
#include "simpio.h"
#include "queue.h"     /* include interface */

#include "queue.h"     /* include interface */

#define MAXSIZE 10

/*---------------------------------*/
/* Structure representing a queue */
/*---------------------------------*/
typedef struct queue_type {
    ElementType space[MAXSIZE]; /* space for elements */
    int head; /* position of head in array */
    int tail; /* position of tail in array */
    int size; /* count of elements in queue */
} basic_queue;

/*-------------------------------------------------------*/
/* OPERATIONS */
/*-------------------------------------------------------*/

/*-------------------------------------------------------*/
/* CreateQueue:
    no input;
    returns a brand new and
    empty queue */
/*-------------------------------------------------------*/

Queue CreateQueue()
{
    Queue new;

    /* allocate space for new queue */
    new = malloc (sizeof(basic_queue));
    if (new == NULL)
        Error("Creation of Queue has failed\n");
    else
    {
        /* need to initialize the fields of the queue descriptor */
        /* to describe an empty queue */
        new->size = 0;
        new->head = -1; /* there is no head */
        new->tail = -1; /* there is no tail */
    }
    return new;
}
/*---------------------------------*/
/* DeleteQueue 
   takes a queue as input; 
   destroys the given queue */
/*---------------------------------*/

void DeleteQueue (Queue s)
{
    /* need to simply release the memory used by the queue descriptor */
    free(s);
}

/*---------------------------------*/
/* EmptyQueue 
   takes a queue as input; 
   returns true if queue empty */
/*---------------------------------*/

bool EmptyQueue (Queue s)
{
    return (s->size == 0);
}

/*---------------------------------*/
/* QueueSize 
   takes a queue as input; 
   returns number of elements 
   in the queue */
/*---------------------------------*/

int QueueSize (Queue s)
{
    return (s->size);
}

/*---------------------------------*/
/* QueueFront 
   takes a queue as input; 
   returns the element currently 
   on the head of the queue */
/*---------------------------------*/

ElementType QueueFront (Queue s)
{
    if (EmptyQueue(s))
        Error("Attempting to access empty queue\n");
    else
        return (s->space[s->head]);
}
/*---------------------------------*/
/* QueueRear */
   takes a queue as input; 
   returns the element currently 
on the tail of the queue */
/*---------------------------------*/

ElementType QueueRear (Queue s)
{
   if (EmptyQueue(s))
      Error("Attempting to access empty queue\n");
   else
      return (s->space[s->tail]);
}

/*---------------------------------*/
/* Enqueue */
   takes a queue as input; 
   takes an element as input; 
   modifies the queue by inserting 
the element as new tail */
/*---------------------------------*/

void Enqueue (ElementType element, Queue s)
{
   if (s->size == MAXSIZE)
      Error("Cannot enqueue; queue is full\n");

   s->tail = (s->tail + 1) % MAXSIZE;
   s->space[s->tail] = element;

   if (s->size == 0)
      s->head = s->tail;
   s->size++;
}

/*---------------------------------*/
/* Dequeue */
   takes a queue as input; 
   modifies the queue by 
removing the current head */
/*---------------------------------*/

void Dequeue (Queue s)
{
   if (EmptyQueue(s))
      Error("Cannot dequeue; queue is empty\n");
   else
   {
      s->head = (s->head + 1) % MAXSIZE;
      s->size--;
QUESTION: what happen if during an Enqueue we reach a queue of size greater than MAXSIZE ??

5 Implementation Using the LIST ADT

We can easily implement queues using the ADT designed for lists. This can be realized as follows:

```c
#include "list.h"    /* we use list adt */
typedef List Queue;  /* A list is used to represent a queue */

Queue CreateQueue()
{
    return (CreateList());
}

void DeleteQueue(Queue s)
{
    DeleteList(s);
}

int QueueSize(Queue s)
{
    return (Size(s));
}

int EmptyQueue(Queue s)
{
return (EmptyList(s));
}

int Front(Queue s)
{
    return (SelectInList(1,s));
}

int Rear(Queue s)
{
    return (SelectInList(ListSize(s),s));
}

void Enqueue(int elem, Queue s)
{
    InsertInList(Size(s)+1,elem,s);
}

void Dequeue(Queue s)
{
    DeleteInList(1,s);
}

void AssignQueues(Queue s1, Queue s2)
{
    AssignLists(s1,s2);
}

6 Queue Applications

Typical kind of applications:

6.1 Buffer

Two system units communicating. E.g., an HTTP server which receives requests from undetermined clients.

Queue Buffer;

void Server()
{
    ElementType request;

    Buffer = CreateQueue();

    while (TRUE)
    {
        while (EmptyQueue(Buffer));
        request = QueueFront(Buffer);
        Dequeue(Buffer);
6.2 Image Component Labeling

A digitized image is an $m \times m$ matrix of pixels. In a binary image each pixel is either 0 or 1. A 0 pixel represents image background, while a 1 represents a point on an image component. We will refer to pixels whose value is 1 as component pixels. Two pixels are adjacent if one is to the left, above, right, or below the other. Two adjacent component pixels are pixels of the same image component. The objective of component labeling is to label the component pixels so that two pixels get the same label if they are pixels of the same image component.

Consider figure 9. The blank squares represent background pixels and the 1s represent component pixels. Pixels (1,3) and (2,3) are pixels of the same component because they are adjacent. Since component pixels (2,3) and (2,4) are adjacent they are also of the same component. Hence the three pixels (1,3), (2,3), and (2,4) are from the same component. Since no other image pixels are adjacent to these three pixels, these three define an image component. The image in figure 9 has four components. The first component is defined by pixels (1,3), (2,3), and (2,4); the second is (3,5), (4,4), (4,5), (5,5); the third is (5,2), (6,1), (6,2), (6,3), (7,1), (7,2), (7,3); and the fourth is (5,7), (6,7), (7,6), (7,7). In the table on the right in figure 9 the components have been labeled with the numbers 2, 3, 4, 5.

The components are determined by scanning the pixels by rows and within rows by columns. When an unlabeled component pixel is encountered, it is given a component label. The pixel forms the seed of a new component. We determine the remaining pixels in the component by identifying and labeling all component pixels that are adjacent to the seed. Call the pixels that are adjacent to the seed the distance 1 pixels. Then unlabeled component pixels that are adjacent to the distance 1 pixels are identified and labeled. These newly labeled pixels are the distance 2 pixels. This process continues until no new unlabeled adjacent component pixels are found.

In the implementation, for simplicity we surround the original image with a wall of blank (0) pixels.
We use a queue and we assume that the base type of the queue is

typedef struct
{
    int x;
    int y;
} Point;

So each element in the queue is one element of type Point.

void labelcomponents()
{
    int i;
    Queue q;
    int row, column;
    int label = 2;
    Point here;

    // initialize wall of 0 pixels
    for (i=0; i <= size+1; i++)
    {
        pixel[0][i] = pixel[size+1][i] = 0;
        pixel[i][0] = pixel[i][size+1] = 0;
    }

    q = CreateQueue();

    for (row=1; row <= size; row++)
    for (column=1; column <= size; column++)
    if (pixel[row][column] == 1)
    {
        // new component
        here.x = row;
        here.y = column;
        Enqueue(here, q);

        while (! EmptyQueue(q))
        {
            here = Front(q);
            Dequeue(q);
            pixel[here.x][here.y] = label;
            if (pixel[here.x-1][here.y] == 1)
                Enqueue(up(here),q);
            if (pixel[here.x+1][here.y] == 1)
                Enqueue(down(here),q);
            if (pixel[here.x][here.y-1] == 1)
                Enqueue(left(here),q);
            if (pixel[here.x][here.y+1] == 1)
                Enqueue(right(here),q);
        }
        label++;
    }
}