CS 2412 Data Structures

Chapter 3

Queues

3.1 Definitions

A queue is an ordered collection of data in which all additions to the collection are made at one end (called rare or tail) and all deletions from the collection are made at the other end (called front or head).

A stack is FILO, while a queue is FIFO (first-in first-out).

Applications:

- CPU waiting list.
- Network router packets sequence.
- Simulations

Main operations:

- Creation
- Clear (damage)
- Append (add element)
- Serve (remove element)

Specification:

```
void CteateQueue(Queue *q);
```

precondition: None.

postcondition: The queue `q` has been initialized to be empty.

(Also can use `Queue* CreateQueue(void);`)

```
void ClearQueue(Queue *q);
```

precondition: The queue `q` has been created.

postcondition: All entries have been removed from `q` and it is now empty.

```
Boolean QueueEmpty(Queue *q);
```

precondition: The queue has been created.

postcondition: The function returns true or false according as queue `q` is empty or not.

```
Boolean QueueFull(Queue *q);
```

precondition: The queue has been created.

postcondition: The function returns true or false according as queue `q` is full or not.

`void Append(QueueEntry x, Queue *q);`

precondition: The queue has been create and is not full.

postcondition: The entry `x` has been stored in the queue as last entry.

`void Serve(QueueEntry *x, Queue *q);`

precondition: The queue has been create and is not empty.

postcondition: The first entry has been removed and returned as the value of `x`.

Some other operations which are not usual operations of a queue.

`int QueueSize(Queue *q);`

precondition: The queue has been created.

postcondition: The function returns the number of entries in the queue.

`void QueueFront(QueueEntry *x, Queue *q);`

precondition: The queue has been created and it is not empty.

postcondition: The value of first entry of `q` is copied to `x`. The queue remains unchanged.

3.2 Implementations

Using an array to implement a queue has some difficulties.

- If after each Serve every item moves forward, then time consume is big.
- If the item not moves, the space will be wasted.

A circular array can be used to solve that problem.

Circular array implementation

To let an array be circular in C, we can use:

```
if(i>=MAX-1)
 i=0;
else
 i++;
```

or

```
i=(i+1)% MAX
```

The difficulty is how to know the queue is empty or full if we use a circular array.

We can use a variable `count` to track of the numbers of entries in a queue to avoid the confusion of full or empty.

```
#define MAXQUEUE 10
typedef char QueueEntry;
typedef struct queue{
  int count;
  int front;
  int rear;
  QueueEntry entry[MAXQUREUE];
} Queue;
```

Prototypes:

```
void CreateQueue(Queue *);
void Append(QueueEntry,Queue *);
void Serve(QueueEntry *,Queue *);
Boolean QueueEmpty(Queue *);
Boolean QueueFull(Queue *);
```

```
void CreateQueue(Queue *q)
{
 q->count = 0;
 q->front = 0;
 q->rear = -1;
}

void Append(QueueEntry x,Queue *q)
{ if(QueueFull(q))
   Error("Cannot append an entry to a full queue.");
 else{
   q->count++;
   q->rear = (q->rear +1)%MAXQUEUE;
   q->entry[q->rear)]=x;
   }
}
```

```
void Serve(QueueEntry *x,Queue *q)
{
 if(QueueEmpty(q))
   Error("Can't serve from a an empty queue.");
 else{
   q->cont--;
   *x = q->entry[q->front];
   q->front=(q->front+1)%MAXQUEUE;
   }
}
```

```
Boolean QueueEmpty(Queue *q)
{
 return q->count <=0;
}

Boolean QueueFull(Queue *q)
{
  return q->cout >=MAXQUEUE;
}

int QueueSize(Queue *q)
{
  return q->count;
}
```

3.3 Computer simulation

Simulation is the use of one system to imitate the behavior of another system, when it is too expensive or dangerous to experiment with the real system.

A computer simulation uses the steps of a program to imitate the behavior of a system under study.

We study a simple but useful computer simulation which concentrates on queues as its basic data structure. These simulations imitate the behavior of systems in which there are queues of objects waiting to be served by various processes.

Example

Consider a small but busy airport with only one runway:

- In each unit of time, one plane can land or one plane can take off, but not both.
- Planes arrive ready to land or to take off at random times.
- It is better to keep a plane waiting on the ground than in the air, so the airport allows a plane to take off only if there are no planes waiting to land.

We will use two queues: `takeoff` and `landing` in the simulation.

A plane is appended to a queue at random with certain probability in a unit time.

The simulation runs for many units of time. It starts for `curtime` from 1 to a variable `endtime`.

We will use a function `RandomNumber()` to output a random number between 0 and `INT_MAX` with given average, which will denote the number of planes at a unit of time.

Outline of the simulation program:

Settings:

```
Queue landing, takeoff;
Queue *pl = &landing;
Queue *pt = &takeoff;
Plane plane; //abstrat of a plane
int curtime; //current time unit
int endtime; //total number of time units to run
```

Initialization:

```
CreateQueue(pl);
CreateQueue(pt);
```

Simulate queue `landing` at a unit of time:

```
for(i = 1; i <= RandomNumber();i++){
  NewPlane(&plane); //new planes arrive
  if (QueueFull(pl)
    Refuse(plane);
  else
    Append(plane,pl);
 }
```

Simulation of queue `takeoff` at a unit of time is similar.

Simulation of the control:

```
if(!QueueEmpty(pl)){
  Serve(&plane,pl);
  Land(plane);
} else if(!QueueEmpty(pt)){
  Serve(&plane,pt);
} else
  Idle();
```

Now we consider some details.

First, the queues we used are queue of planes.

```
#define MAXQUEUE 5 //use a small value for testing
typedef enum action {ARRIVE, DEPART} Action;
typedef struct plane {
  int id;  //id number of plane
  int tm;  //time of arrival in queue
} Plane;
typedef Plane QueueEntry;
typedef struct queue{
   int count; //number of planes in the queue
   int front;
   int rear;
   QueueEntry entry[MAXQUEUE];
} Queue;
```

The purpose of the simulation is to obtain some data about the airport. So we need to record the number of planes, the waiting time, the numbers of departure and arriving planes, etc.

The simulation also wants to know the different data in different settings of the probability of arriving planes and departing planes.

```
/*NewPlane: make a new record for a plan, update nplanes.
Pre: None.
Post: Makes a new  plane and update nplanes.*/
void NewPlane(Plane *p,int *nplanes,int curtime,Action kind)
{(*nplanes)++;
 p->id=*nplanes;
 p->tm=curtime;
 switch(kind){
 case ARRIVE:
   printf(" Plane %3d ready to land.\n",*nplanes);
   break;
 case DEPART:
   printf(" Plane %3d ready to take off.\n",*nplanes);
   break;
 }
}
```

```
/*Refuse: processes a plan when the queueis full.
Pre: None.
Post: processes a plane requesting runway, but queue is full*/
void Refuse(Plane p, int *nrefuse,Action kind)
{
  switch(kind) {
  case ARRIVE:
    printf(" Plane %3d directed to another airport.\n",p.id);
    break;
  case DEPART:
    printf(" Plane %3d told to try later.\n".p.id);
    break;
  }
  (*nrefuse)++;
}
```

```
/*Land: process a plane that is actually landing.
Pre: None.
Post: Precesses a plane p that is landing.*/

void Land(Plane p,int curtime,int *nland,int *landwait)
{
  int wait;
  wait = curtime - p.tm;
  printf("%3d: Plane %3d landed; in queue %d units.
         \n",curtime,p.id,wait);
  (*nland)++;
  *landwait+= wait;
}
```

The function `Fly` is similar.

```
/* Idle: undate variables for idle runway.
Pre: None.
Post: Update variables for a time unit when the
      runway is idle. */
void Idel(int curtime,int *idletime)
{
  printf("%3d: Runway is idel.\n",curtime);
  (*idletime)++;
}
```

Two functions: `Start` and `Conclude` are used to initialize and conclude statistical data for the simulation.

```
/*Start: print messages and initialize the parameters.
Pre:None.
Post: Asks user for responses and initializes all
     variables specified as parameters.
Uses: UserSaysYes.*/
void Start(int *endtime,double *expectarrive,
           double *expectdepart)
```

```
/*Conclude: write out statistics and conclude simulation.
Pre: None.
Post: Writes out all the statistics and concludes the
      simulation.*/
void Conclude(int nplanes, int nland, int ntakeoff,
              int nrefuse, int landwait, int takeoffwait,
              int idletime,int endtime,
              Queue *pt, Queue *pl)
```

We omitted the details of these two functions.

The details of creating pseudo-random numbers are omitted.

Implementation using dynamic memory

```
// Queue ADT Type Defintions
typedef struct node
  {
   void*        dataPtr;
   struct node* next;
  } QUEUE_NODE;

typedef struct
  {
   QUEUE_NODE* front;
   QUEUE_NODE* rear;
   int         count;
  } QUEUE;
```

```
QUEUE* createQueue (void)
{
// Local Definitions
QUEUE* queue;

// Statements
queue = (QUEUE*) malloc (sizeof (QUEUE));
if (queue)
   {
    queue->front  = NULL;
    queue->rear   = NULL;
    queue->count  = 0;
   } // if
return queue;
} // createQueue
```

```
bool enqueue (QUEUE* queue, void* itemPtr)
{
QUEUE_NODE* newPtr;
if (!(newPtr =
   (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE))))
   return false;
newPtr->dataPtr = itemPtr;
newPtr->next    = NULL;
if (queue->count == 0)
   queue->front  = newPtr;
else
   queue->rear->next = newPtr;
(queue->count)++;
queue->rear = newPtr;
return true;
} // enqueue
```

```
bool dequeue (QUEUE* queue, void** itemPtr)
{
QUEUE_NODE* deleteLoc;
if (!queue->count)
    return false;
*itemPtr  = queue->front->dataPtr;
deleteLoc = queue->front;
if (queue->count == 1)
   queue->rear  = queue->front = NULL;
else
   queue->front = queue->front->next;
(queue->count)--;
free (deleteLoc);
return true;
} // dequeue
```

```
bool queueFront (QUEUE* queue, void** itemPtr)
{
// Statements
if (!queue->count)
    return false;
else
   {
    *itemPtr = queue->front->dataPtr;
     return true;
   } // else
} // queueFront
```

```
bool queueRear (QUEUE* queue, void** itemPtr)
{
// Statements
if (!queue->count)
    return true;
else
   {
    *itemPtr = queue->rear->dataPtr;
    return false;
   } // else
} // queueRear
```

```
bool emptyQueue (QUEUE* queue)
{
// Statements
return (queue->count == 0);
} // emptyQueue
```

```
bool fullQueue (QUEUE* queue)
{
// Local Definitions *
QUEUE_NODE* temp;

// Statements
temp = (QUEUE_NODE*)malloc(sizeof(*(queue->rear)));
if (temp)
   {
    free (temp);
    return true;
   } // if
// Heap full
return false;
} // fullQueue
```

```
int queueCount(QUEUE* queue)
{
// Statements
return queue->count;
} // queueCount
```

```
QUEUE* destroyQueue (QUEUE* queue)
{
QUEUE_NODE* deletePtr;
if (queue)
   {
    while (queue->front != NULL)
       {
        free (queue->front->dataPtr);
        deletePtr     = queue->front;
        queue->front = queue->front->next;
        free (deletePtr);
       } // while
    free (queue);
   } // if
return NULL;
} // destroyQueue
```

Application: Polynomial arithmetic

Using a reverse Polish calculator to do polynomial arithmetics. basic idea is using a stack. The basic idea of reverse Polish calculator is to input operands first and input operation.

For example: if we want to calculate $(a+b)*(c+d)$, then the inputs are: $a\ b + c\ d + *$. If we want to calculate $a*(b+c)$, then the inputs are: $b\ c + a\ *$.

To do the calculations, push the operands to a stack. If an operation is input, then pop out two operands and do the operation and then push the result to the stack. When "=" is input, pop out the result.

The outline of the program is quite simple: mainly use `ReadCommand` to get the inputs and use `DoCommand` to do the calculation.

To represent polynomials, we use a queues. For a polynomial $3x^5 - 2x^3 + x^2 + 4$, we can use a queue of size 4 to represent it. Each item of the queue represent a term.

```
typedef struct term{
  double coef;
  int exp;
} Term;
```

The term $-2x^3$ can be stored as:

```
Term *newterm;

newterm->coef = -2.0;
newterm->exp = 3;
```