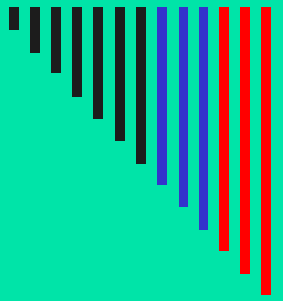


Chapter 4

Queues

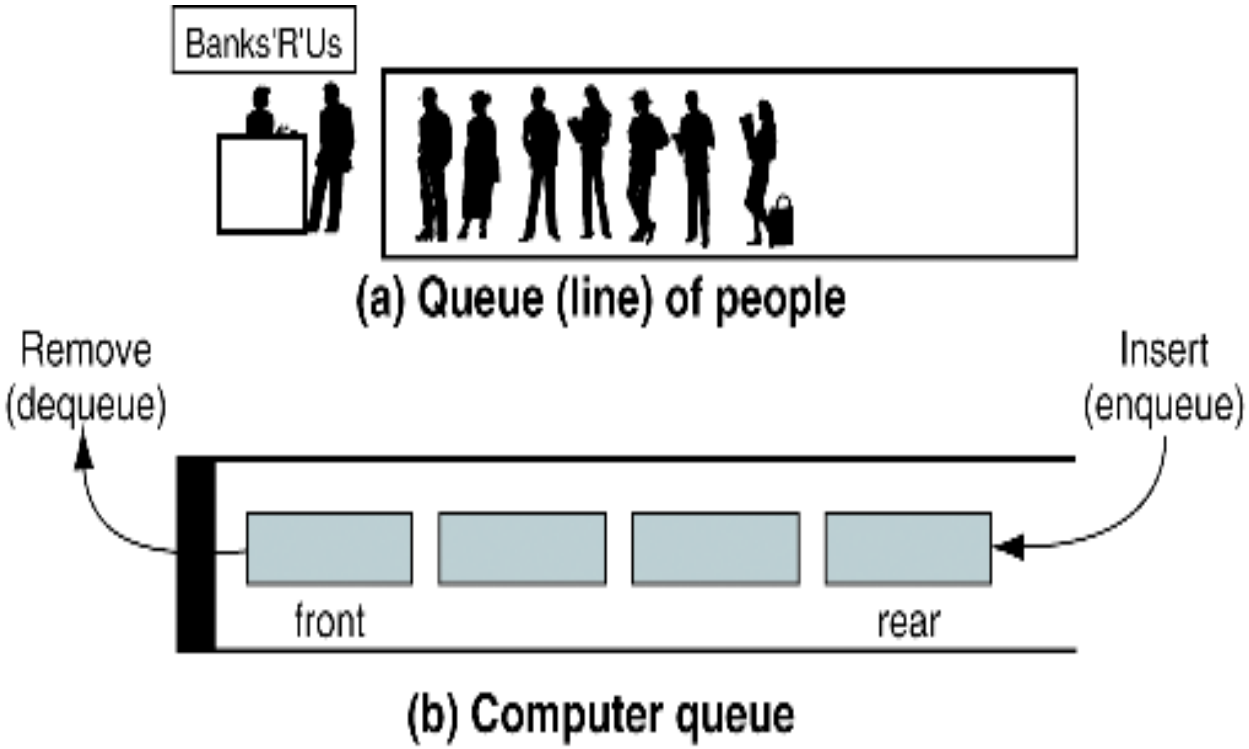
Objectives

Upon completion you will be able to:

- **Explain the design, use, and operation of a queue**
- **Implement a queue using a linked list structure**
- **Understand the operation of the queue ADT**
- **Write application programs using the queue ADT**
- **Explain categorizing data and queue simulation**

Queue Concept

- A queue is a linear list in which data can only be inserted at one end, and deleted from the other end
- A queue is a first in-first out (FIFO) restricted data structure



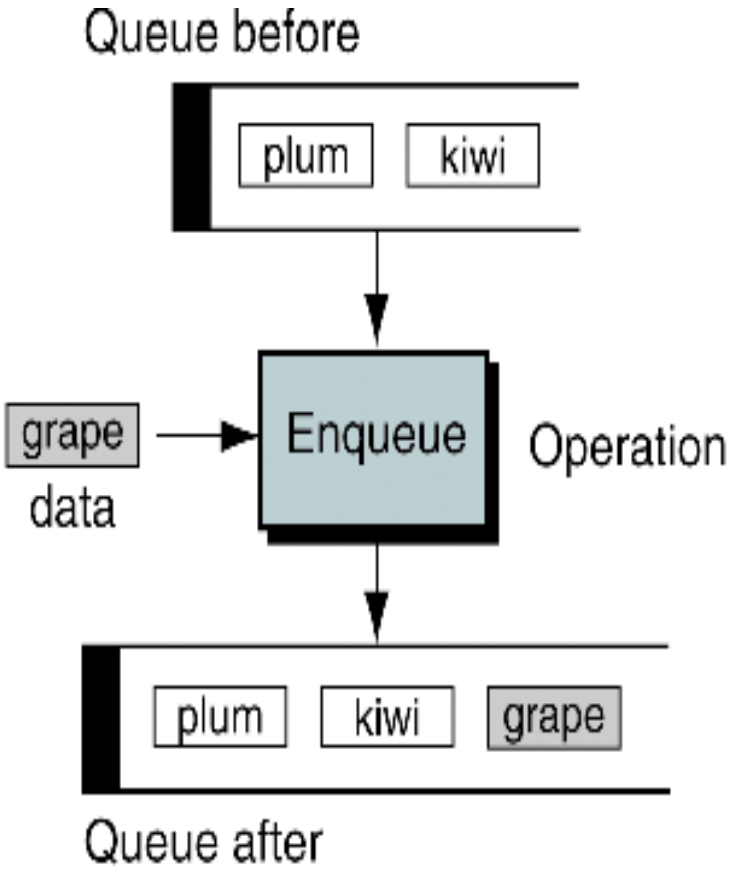
4-1 Queue Operations

This section discusses the four basic queue operations. Using diagrammatic figures, it shows how each of them work. It concludes with a comprehensive example that demonstrates each operation.

- **Enqueue**
- **Dequeue**
- **Queue Front**
- **Queue Rear**
- **Queue Example**

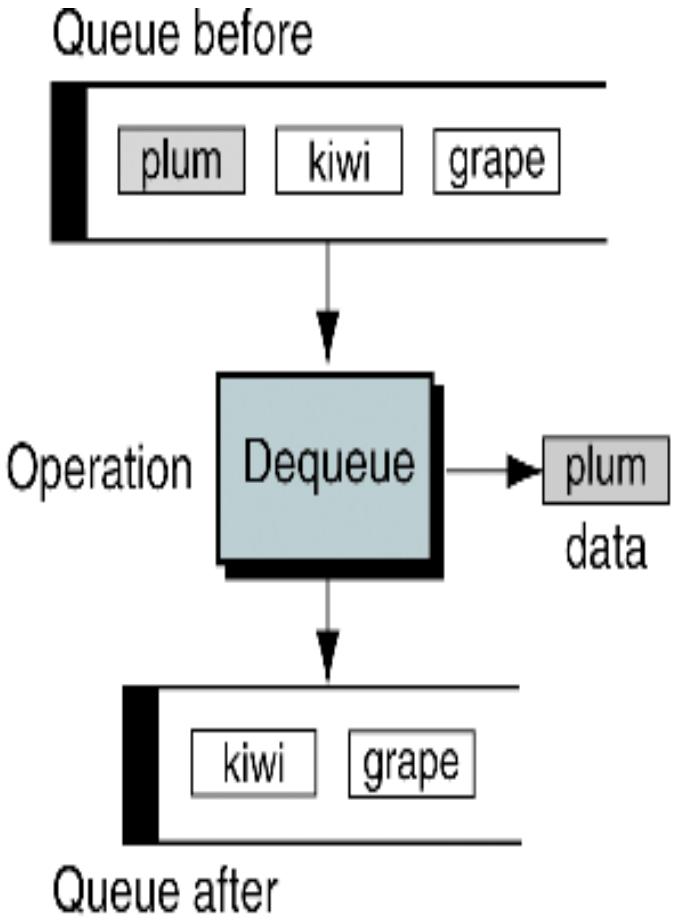
Enqueue

- Potential problem with enqueue is ***overflow***



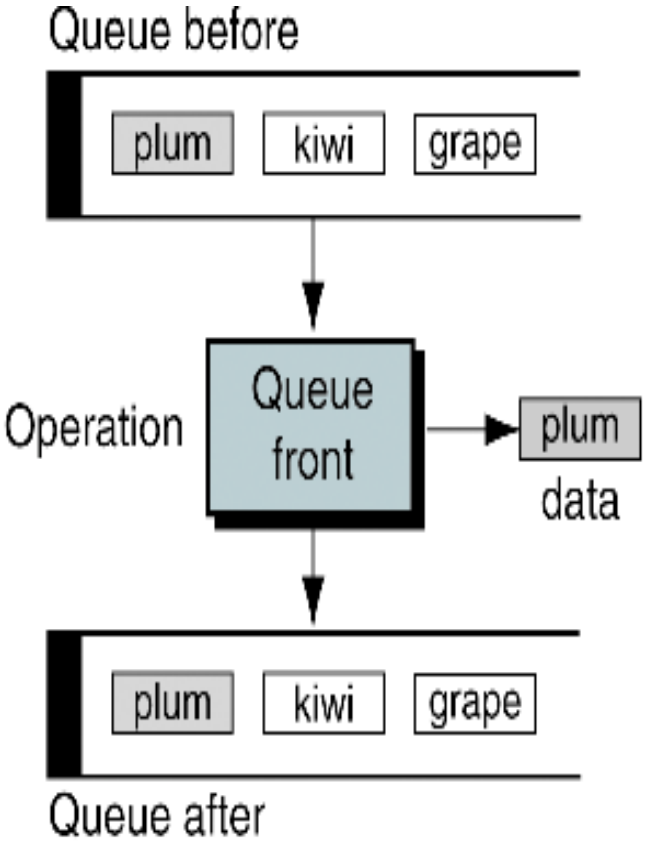
Dequeue

- **Potential problem with dequeue is *underflow***



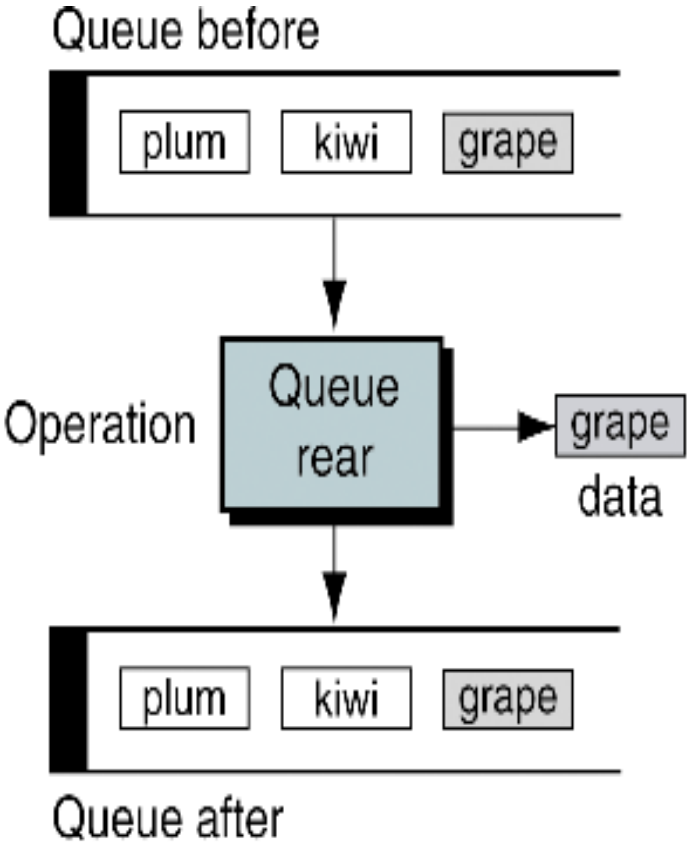
Queue Front

- Potential problem with queue front is ***underflow***

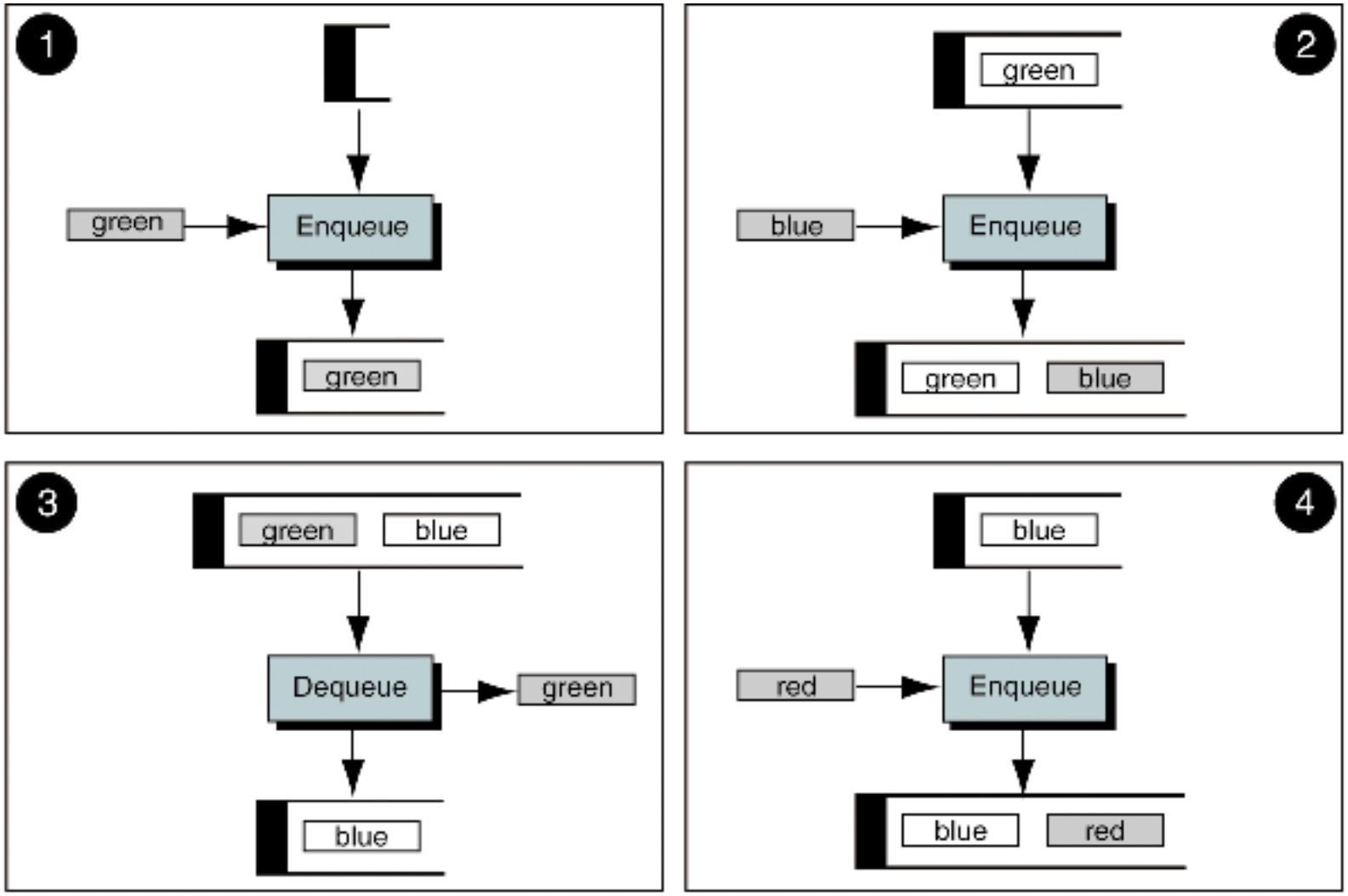


Queue Rear

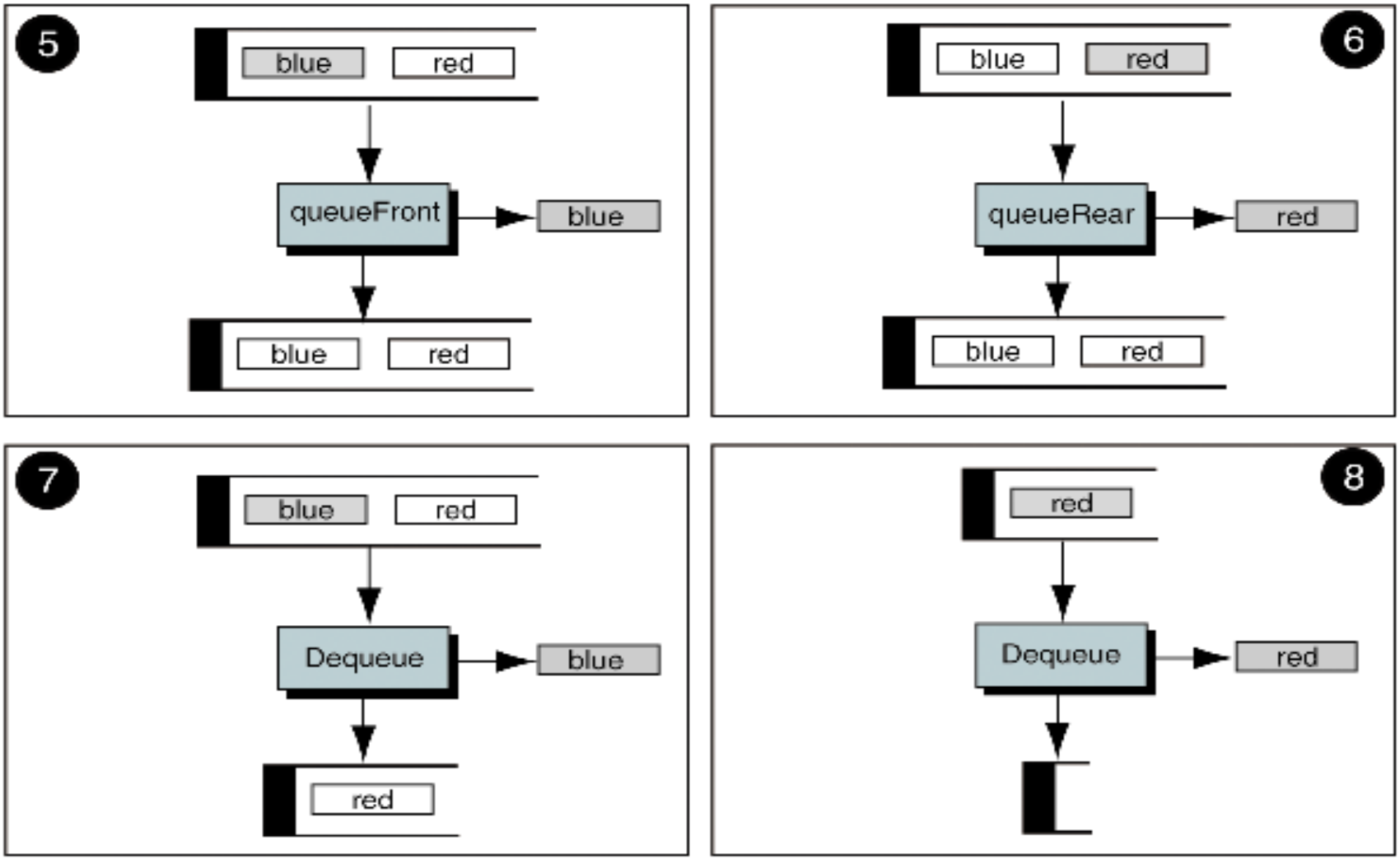
- Potential problem with queue rear is ***underflow***



Queue Example



Queue Example (cont.)



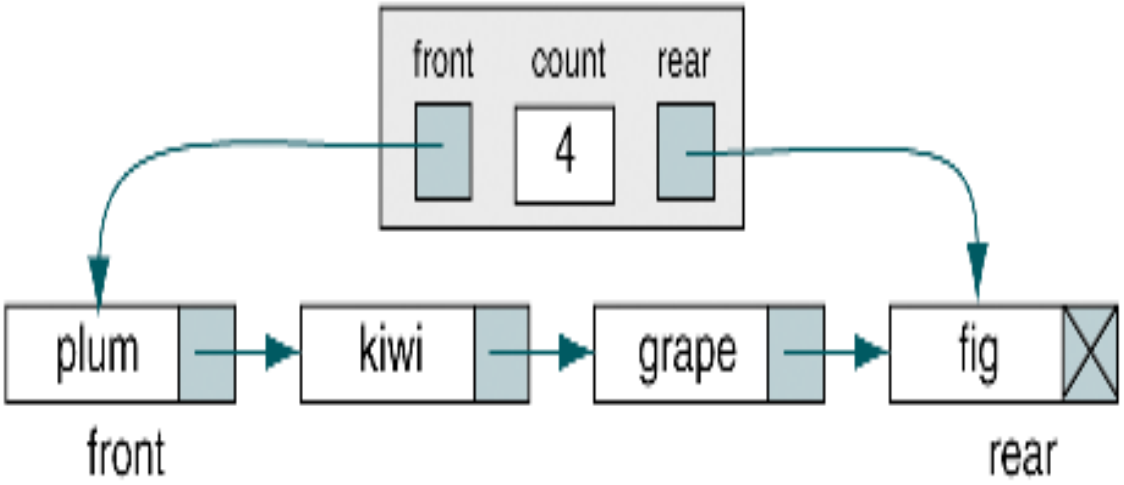
4-2 Queue Linked List Design

We first discuss the data structure for a linked-list implementation. We then develop the eight algorithms required to implement a queue.

- Data Structure Queue
- Algorithms

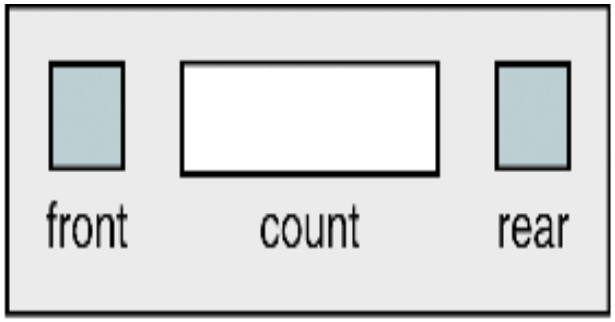
Conceptual and Physical Queue Implementations

(a) Conceptual queue

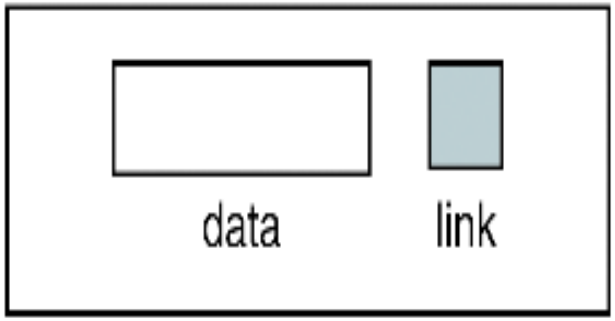


(b) Physical queue

Queue Data Structure



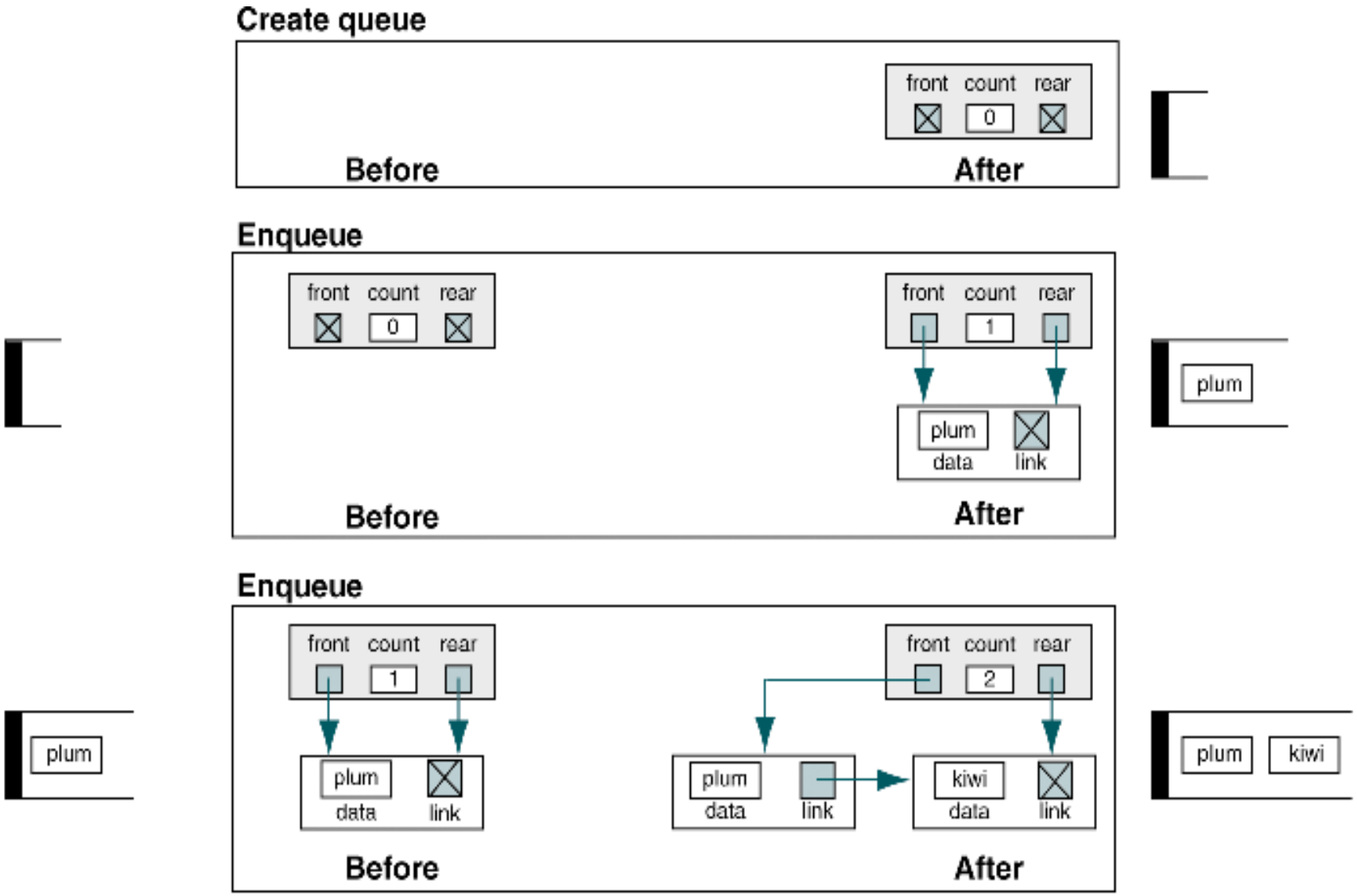
Head structure



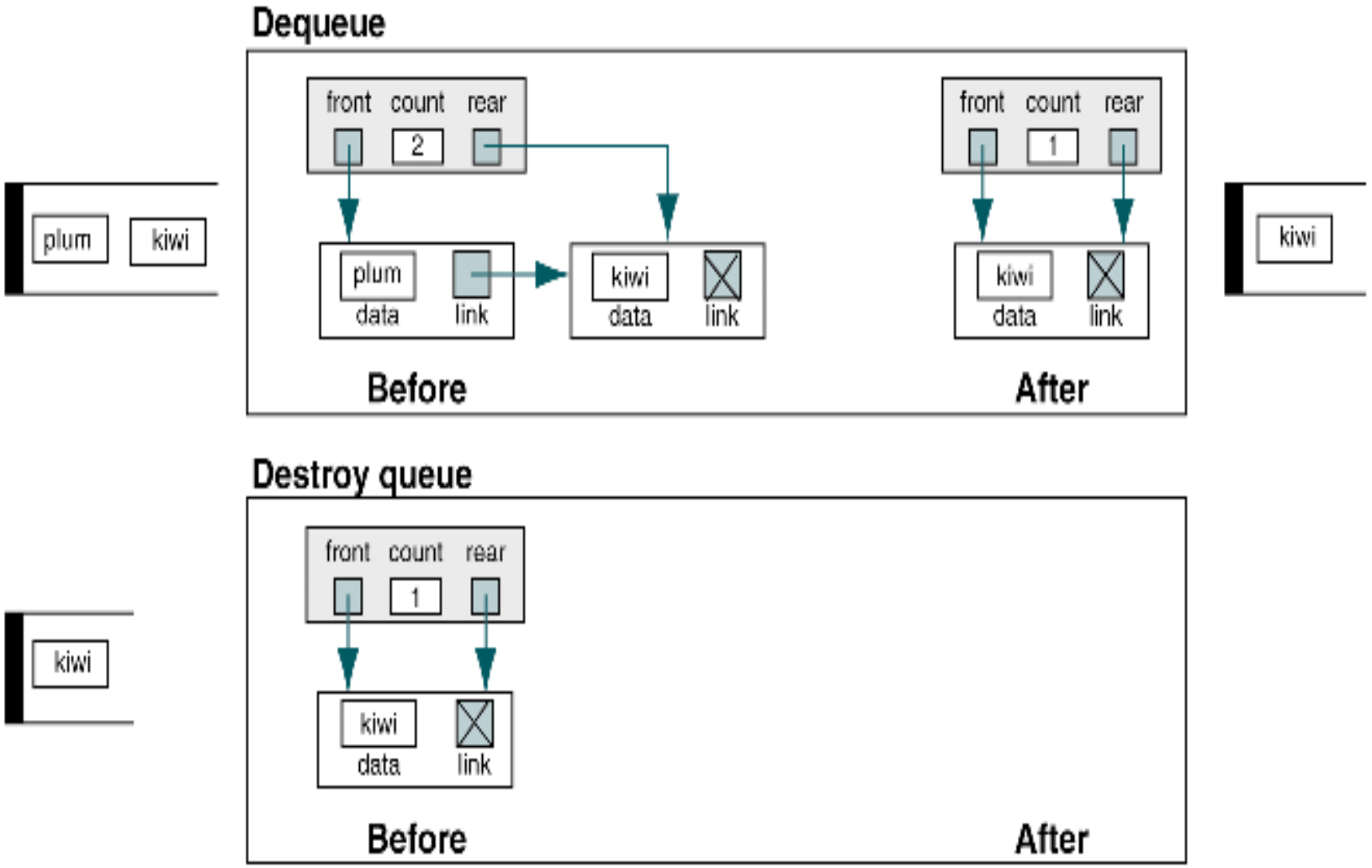
Node structure

```
queueHead
  front
  count
  rear
end queueHead

node
  data
  link
end node
```

Basic Queue Functions



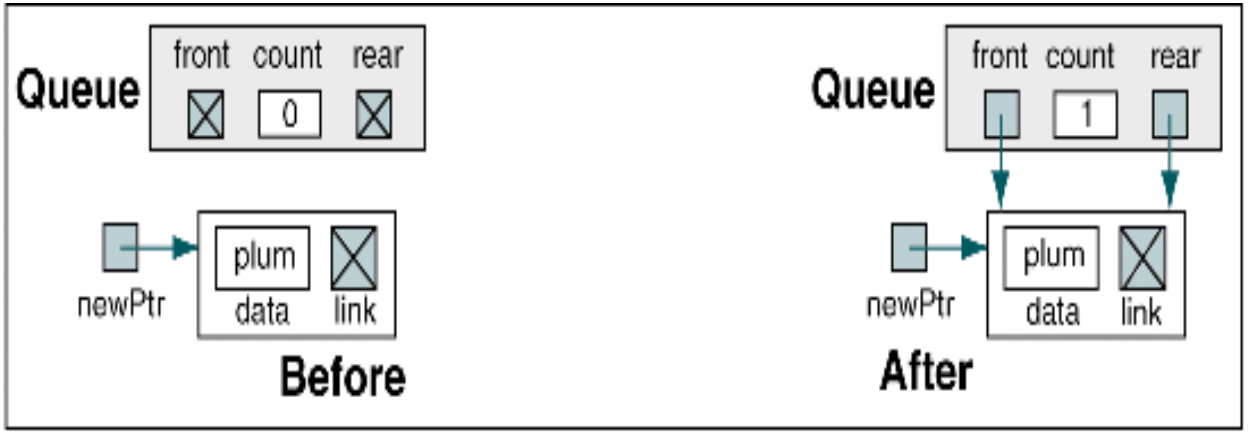
Basic Queue Functions (cont.)



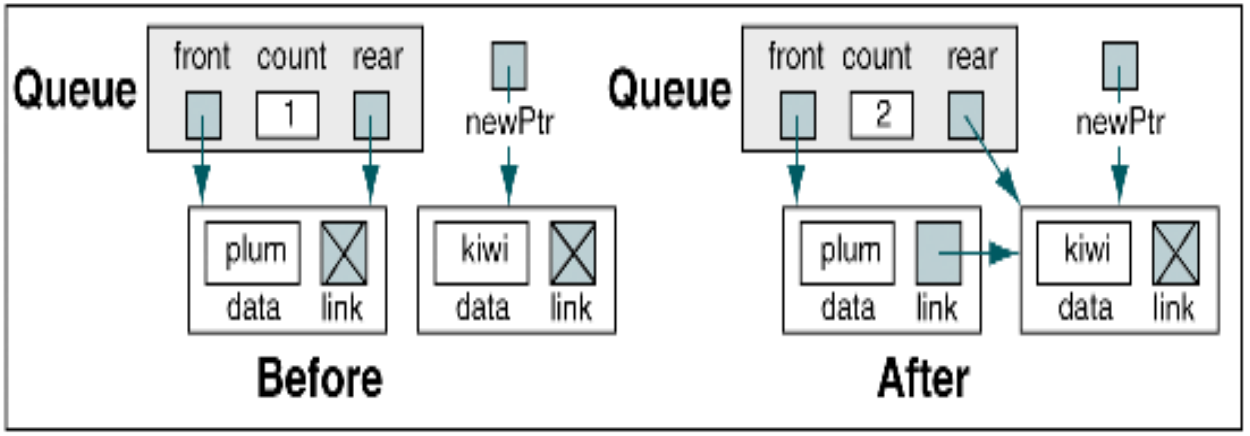
Create Queue: Pseudocode

```
Algorithm createQueue
Creates and initializes queue structure.
  Pre    queue is a metadata structure
  Post   metadata elements have been initialized
  Return queue head
1 allocate queue head
2 set queue front to null
3 set queue rear  to null
4 set queue count to 0
5 return queue head
end createQueue
```

Enqueue Design



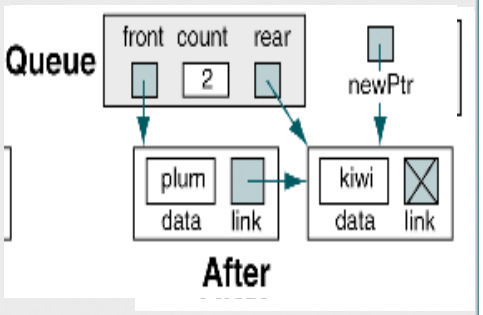
(a) Case 1: insert into empty queue



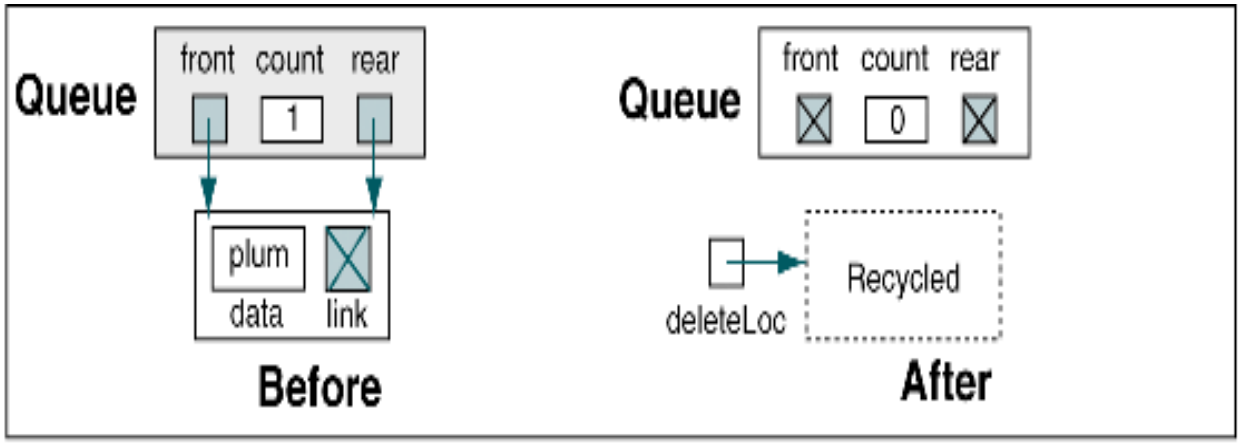
(b) Case 2: insert into queue with data

Enqueue: Pseudocode

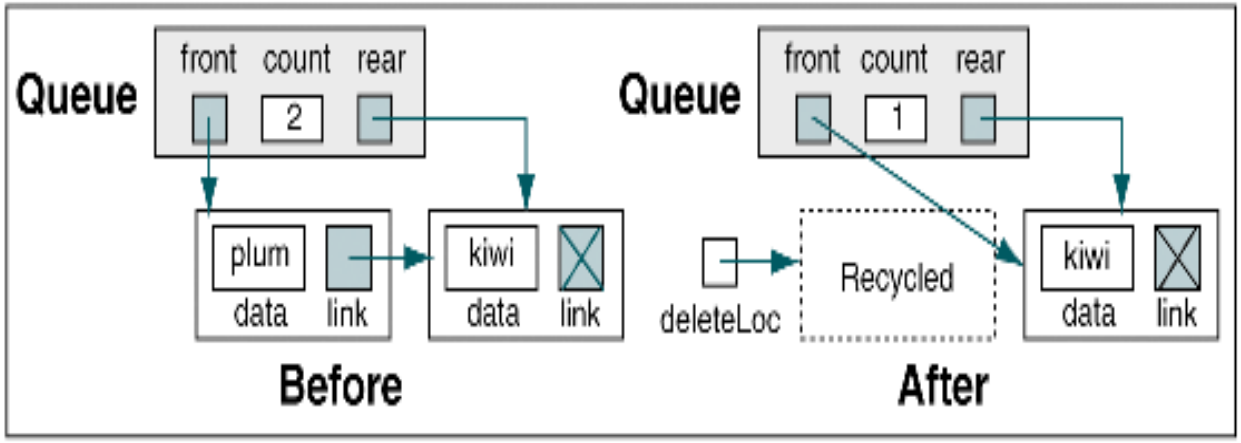
```
Algorithm enqueue (queue, dataIn)
This algorithm inserts data into a queue.
   Pre    queue is a metadata structure
   Post   dataIn has been inserted
   Return true if successful, false if overflow
 1 if (queue full)
   1  return false
 2 end if
 3 allocate (new node)
 4 move dataIn to new node data
 5 set new node next to null pointer
 6 if (empty queue)
   Inserting into null queue
   1  set queue front to address of new data
 7 else
   Point old rear to new node
   1  set next pointer of rear node to address of new node
 8 end if
 9 set queue rear to address of new node
10 increment queue count
11 return true
end enqueue
```



Dequeue Design



(a) Case 1: delete only item in queue



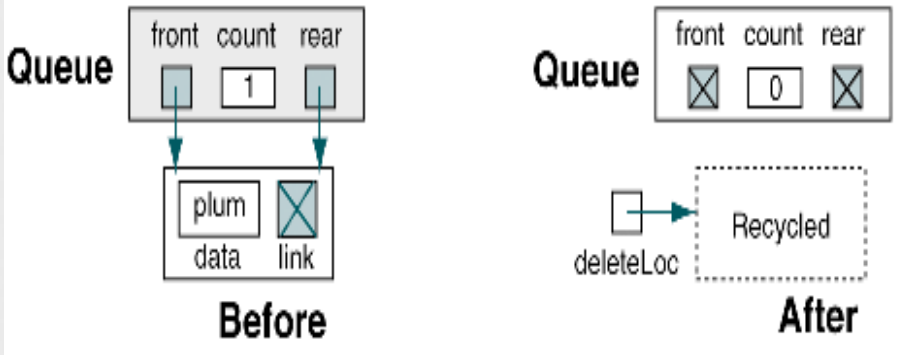
(b) Case 2: delete item at front of queue

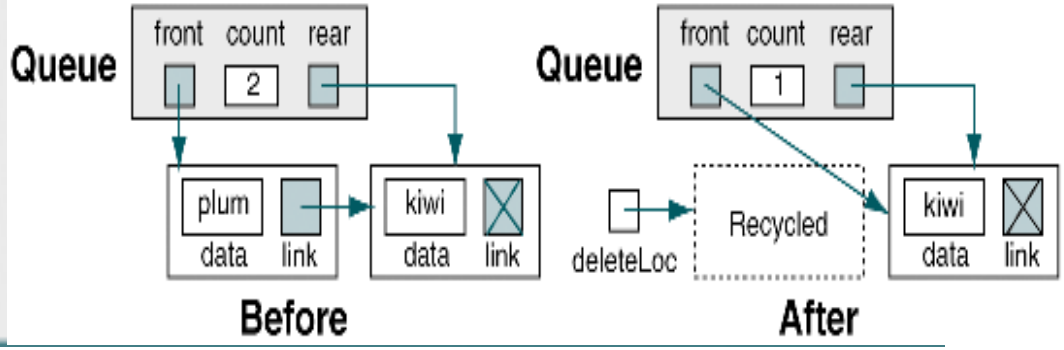
Dequeue: Pseudocode

```
Algorithm dequeue (queue, item)
This algorithm deletes a node from a queue.
  Pre    queue is a metadata structure

         item is a reference to calling algorithm variable
  Post   data at queue front returned to user through item
         and front element deleted
  Return true if successful, false if underflow
1 if (queue empty)
  1  return false
2 end if
```

Dequeue: Pseudocode (cont.)

```
3 move front data to item
4 if (only 1 node in queue)
  Deleting only item in queue
  1  set queue rear to null
5 end if
6 set queue front to queue front next
7 decrement queue count
8 return true
end dequeue
```





Retrieve Data at Front of Queue

```
Algorithm queueFront (queue, dataOut)
Retrieves data at the front of the queue without changing
queue contents.
  Pre    queue is a metadata structure
         dataOut is a reference to calling algorithm variable
  Post   data passed back to caller
  Return true if successful, false if underflow
1 if (queue empty)
  1  return false
2 end if
3 move data at front of queue to dataOut
4 return true
end queueFront
```

Queue Empty

```
Algorithm emptyQueue (queue)
This algorithm checks to see if a queue is empty.
  Pre    queue is a metadata structure
  Return true if empty, false if queue has data
1 if (queue count equal 0)
  1  return true
2 else
  1  return false
end emptyQueue
```

Full Queue

```
Algorithm fullQueue (queue)
This algorithm checks to see if a queue is full. The queue
is full if memory cannot be allocated for another node.
  Pre    queue is a metadata structure
  Return true if full, false if room for another node
1 if (memory not available)
  1  return true
2 else
  1  return false
3 end if
end fullQueue
```

Queue Count

```
Algorithm queueCount (queue)
This algorithm returns the number of elements in the queue.
  Pre    queue is a metadata structure
  Return queue count
 1 return queue count
end queueCount
```

Destroy Queue

```
Algorithm destroyQueue (queue)
This algorithm deletes all data from a queue.
  Pre     queue is a metadata structure
  Post    all data have been deleted
1 if (queue not empty)
  1  loop (queue not empty)
     1  delete front node
  2  end loop
2 end if
3 delete head structure
end destroyQueue
```

4-2.5 Parameter Passing for Functions

- Call by value (C and C++)
- Call by address (C and C++)
- Call by reference (C++)
- Call by constant reference (C++)
- Call by Pointer to pointer (C and C++)
- Call by Reference to pointer (C++)

Call by Value (C ~~an~~ and C++)

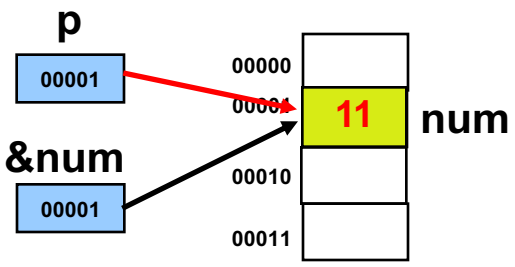
- Main copies the value 10 to bump which does not change the value of num

```
void bump (int m) {
 ++m;                                  // increment by one
cout << m << endl;                     // displays 11
}

void Main {
 int num = 10;
 bump(num);                            // call by value
 cout<< num << endl;                   // displays 10
}
```

Call by Address (C and C++)

- Main copies the address of memory containing 10 to bump
- Still a type of call by value in which the value is the address

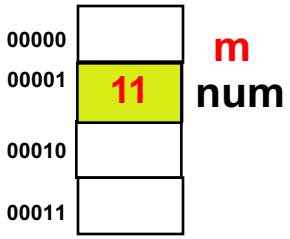
```
void bump (int *p) {
++*p;                                  // increment by one
cout << *p << endl;                    // displays 11
}

void Main {
int num = 10;
 bump(&num);                           // call by value
 cout<< num << endl;                   // displays 11
}
```



Call by Reference (C++)

- Same results as call by address did but simplyfy the program writing

```
void bump (int & m) {
++m;                                  // increment by one
 << m << endl;                        // displays 11
}

void Main {
int num = 10;
 bump(num);                           // call by value
 cout<< num << endl;                  // displays 11
}
```



Call by Constant Reference (C++)

```
#include <iostream.h>
const int MaxBuf = 20;
struct block {
char buf[MaxBuf];
int used;
};

int main() {

void display(const block &);                    // prototype for display()
block data;
int i;

data.used = 5;
for (i = 0; i < data.used; i++)                 // assign some values
   data.buf[i] = i + 'a' ;
display(data);                                  // call by reference
data.used = MaxBuf;
```

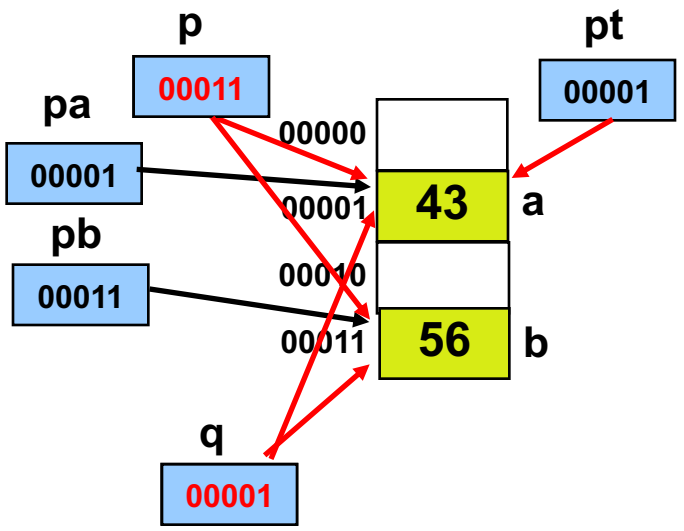
Call by Constant Reference (C++)

```
for (i = 0; i < data.used; i++)
   data.buf[i] = i + 'a';
display(data);                                  // call by reference
return 0;
 }

void display(const block & blockref) {
for (int i = 0; i < blockref.used; i++)
cout << blockref.buf[i] << ' '; cout << endl;
}

-----------------------------------------------------------
Reault:
a b c d e
a b c d e f g h i j k l m n o p q r s t
```

- Compiler reports errors if function display modifies a data member
- Efficient (no local copies) and Safe (no modifications)

Call by Address Does NOT Work ?

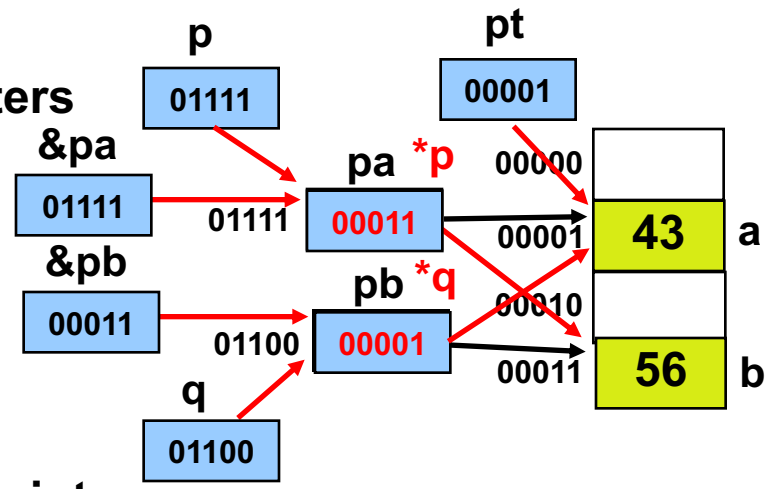
```
// exchange pointers to integers in place using pointers

#include <iostream.h>
int main() {
int a = 43, b = 56;
int *pa = &a,  *pb = &b;
void pxchg(int *, int *);

cout << *pa << ' ' << *pb << endl;
pxchg(pa, pb);
cout << *pa << ' ' << *pb << endl;
return 0; }

void pxchg(int *p, int *q){
int *pt ;
pt = p;
p = q;
q = pt; }
```



Result: 43 56 43 56

Call by Pointer to pointer (C and C++)

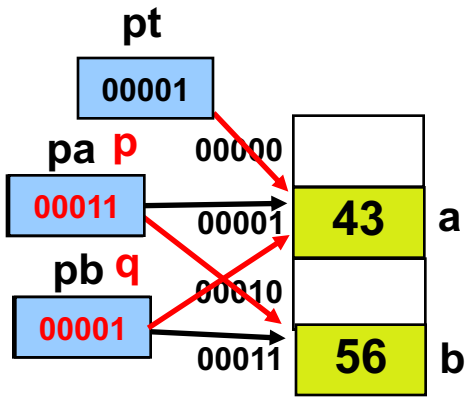
```
// exchange pointers to integers in place using pointers

#include <iostream.h>
int main() {
int a = 43, b = 56;
int *pa = &a,  *pb = &b;
void pxchg(int **, int **);  // double indirect pointers

cout << *pa << ' ' << *pb << endl;
pxchg(&pa, &pb);    // pass pointers to pointers
cout << *pa << ' ' << *pb << endl;
return 0; }

void pxchg(int **p, int **q){    // double indirect pointers
int *pt;
pt = *p;                         // exchange pointers
*p = *q;
*q = pt; }
```



Result: 43 56 56 43

Call by Reference to Pointer (C++)

```
// exchange pointers to integers in place using pointers

#include <iostream.h>
int main() {
int a = 43, b = 56;
int *pa = &a,  *pb = &b;
void pxchg(int *&, int *&);  // reference to pointers

cout << *pa << ' ' << *pb << endl;
pxchg(pa, pb);
cout << *pa << ' ' << *pb << endl;
return 0; }

void pxchg(int *& p, int *& q){    // reference to pointers
int *pt;
pt = p;                            // exchange pointers
p = q;
q = pt; }
```



Result: 43 56 56 43

4-3 Queue ADT

This section develops the data structures and C code to implement a Queue ADT. The first program contains the data structure declarations and a list of the prototypes for all of the functions. We then develop the C code for the algorithms discussed in Section 4.2

- Queue Structure
- Queue ADT Algorithms

Queue ADT Data Structures

```
1  //Queue ADT Type Defintions
2    typedef struct node
3      {
4       void*         dataPtr;
5       struct node* next;
6      } QUEUE_NODE;
7    typedef struct
8      {
9       QUEUE_NODE* front;
10      QUEUE_NODE* rear;
11      int         count;
12     } QUEUE;
13
14 //Prototype Declarations
15   QUEUE* createQueue  (void);
16   QUEUE* destroyQueue (QUEUE* queue);
17
18   bool   dequeue    (QUEUE* queue, void** itemPtr);
19   bool   enqueue    (QUEUE* queue, void*  itemPtr);
20   bool   queueFront (QUEUE* queue, void** itemPtr);
21   bool   queueRear  (QUEUE* queue, void** itemPtr);
22   int    queueCount (QUEUE* queue);
23
24   bool   emptyQueue (QUEUE* queue);
25   bool   fullQueue  (QUEUE* queue);
26 //End of Queue ADT Definitions
```

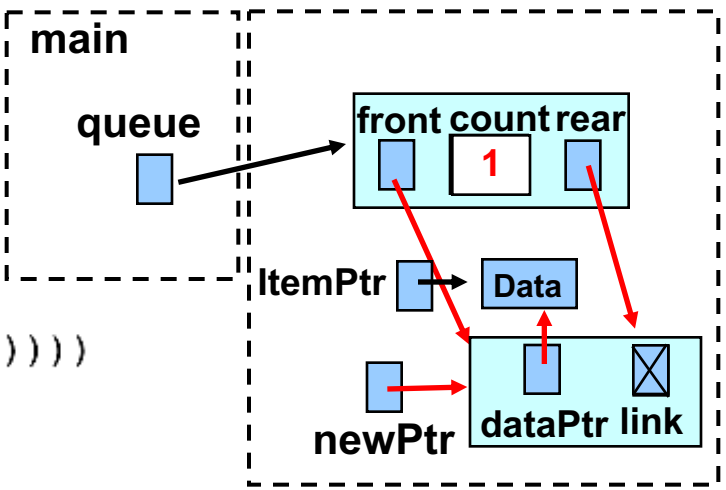
之前stack是直接return pointer

Create Queue in C

```
1  /*================= createQueue ================
2    Allocates memory for a queue head node from dynamic
3    memory and returns its address to the caller.
4       Pre    nothing
5       Post   head has been allocated and initialized
6       Return head if successful; null if overflow
7  */
8  QUEUE* createQueue (void)
9  {
10 //Local Definitions
11   QUEUE* queue;
12
13 //Statements
14   queue = (QUEUE*) malloc (sizeof (QUEUE));
15   if (queue)
16      {
17       queue->front  = NULL;
18       queue->rear   = NULL;
19       queue->count  = 0;
20      } // if
21   return queue;
22 } // createQueue
```

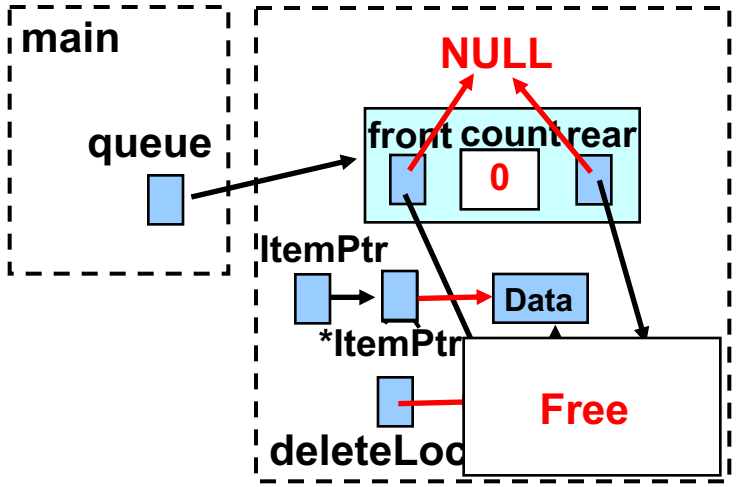
Enqueue in C

```
 7  bool enqueue (QUEUE* queue, void* itemPtr)
 8  {
 9  //Local Definitions
10    QUEUE_NODE* newPtr;
11
12  //Statements
13    if (!(newPtr =
14       (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE))))
15       return false;
16
17    newPtr->dataPtr = itemPtr;
18    newPtr->next    = NULL;
19
20    if (queue->count == 0)
21       // Inserting into null queue
22       queue->front  = newPtr;
23    else
24       queue->rear->next = newPtr;
25
26    (queue->count)++;
27    queue->rear = newPtr;
28    return true;
29  } // enqueue
```



Dequeue in C

```
 8  bool dequeue (QUEUE* queue, void** itemPtr)
 9  {
10  //Local Definitions
11    QUEUE_NODE* deleteLoc;
12
13  //Statements
14    if (!queue->count)
15        return false;
16
17    *itemPtr   = queue->front->dataPtr;
18    deleteLoc = queue->front;
19    if (queue->count == 1)
20       // Deleting only item in queue
21       queue->rear  = queue->front = NULL;
22    else
23       queue->front = queue->front->next;
24    (queue->count)--;
25    free (deleteLoc);
26
27    return true;
28  } // dequeue
```



Queue Front in C

```
1  /*================== queueFront ================
2    This algorithm retrieves data at front of the
3    queue without changing the queue contents.
4       Pre    queue is pointer to an initialized queue
5       Post   itemPtr passed back to caller
6       Return true if successful; false if underflow
7  */
8  bool queueFront (QUEUE* queue, void** itemPtr)
9  {
10 //Statements
11   if (!queue->count)
12       return false;
13   else
14      {
15       *itemPtr = queue->front->dataPtr;
16        return true;
17      } // else
18 } // queueFront
```

Queue Rear in C

```
 8  bool queueRear (QUEUE* queue, void** itemPtr)
 9  {
10  //Statements
11    if (!queue->count)
12        return true;
13    else
14       {
15        *itemPtr = queue->rear->dataPtr;
16        return false;
17       } // else
18  } // queueRear
```

Empty Queue in C

```
1  /*================== emptyQueue ================
2    This algorithm checks to see if queue is empty.
3    Pre    queue is a pointer to a queue head node
4    Return true if empty; false if queue has data
5  */
6  bool emptyQueue (QUEUE* queue)
7  {
8  //Statements
9    return (queue->count == 0);
10 } // emptyQueue
```

Full Queue in C

```
 7  bool fullQueue (QUEUE* queue)
 8  {
 9  //Local Definitions
10  QUEUE_NODE* temp;
11
12  //Statements
13    temp = (QUEUE_NODE*)malloc(sizeof(*(queue->rear)));
14    if (temp)
15       {
16        free (temp);
17        return true;
18       } // if
19    // Heap full
20    return false;
21  } // fullQueue
```

Destroy Queue in C

```
 5        Post   All data have been deleted and recycled
 6        Return null pointer
 7  */
 8  QUEUE* destroyQueue (QUEUE* queue)
 9  {
10  //Local Definitions
11    QUEUE_NODE* deletePtr;
12
13  //Statements
14    if (queue)
15       {
16        while (queue->front != NULL)
17           {
18            free (queue->front->dataPtr);
19            deletePtr    = queue->front;
20          queue->front = queue->front->next;
21            free (deletePtr);
22           } // while
23        free (queue);
24       } // if
25    return NULL;
26  } // destroyQueue
```

4-5 Queue Applications

We develop two queue applications. The first shows how to use a queue to categorize data. The second is a queue simulator, which is an excellent tool to simulate the performance and to increase our understanding of its operation.

- Categorizing Data
- Queue Simulation

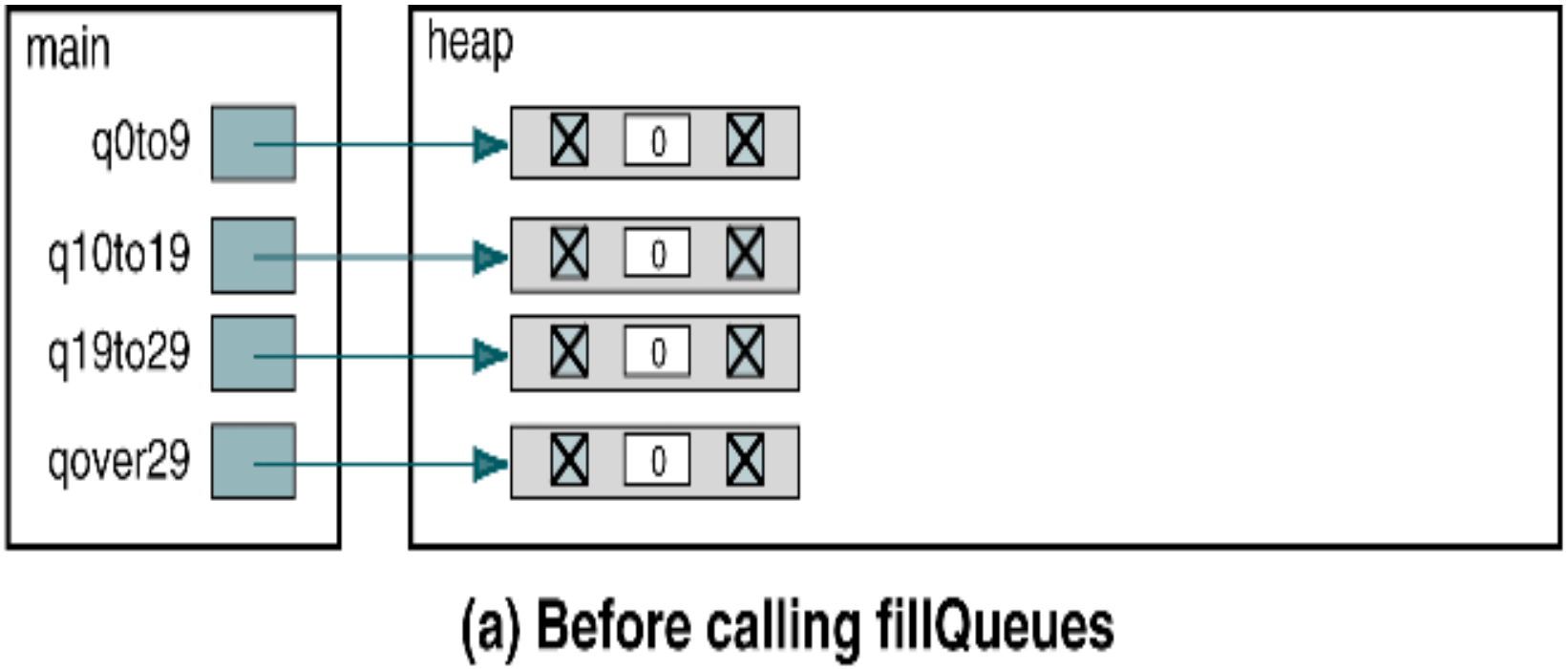
Category Queues

```
Algorithm categorize
  Group a list of numbers into four groups using four queues.
     Written by:
     Date:
 1 createQueue (q0to9)
 2 createQueue (q10to19)
 3 createQueue (q20to29)
 4 createQueue (qOver29)
 5 fillQueues  (q0to9, q10to19, q20to29, qOver29)
 6 printQueues (q0to9, q10to19, q20to29, qOver29)
end categorize
```

Fill Category Queues

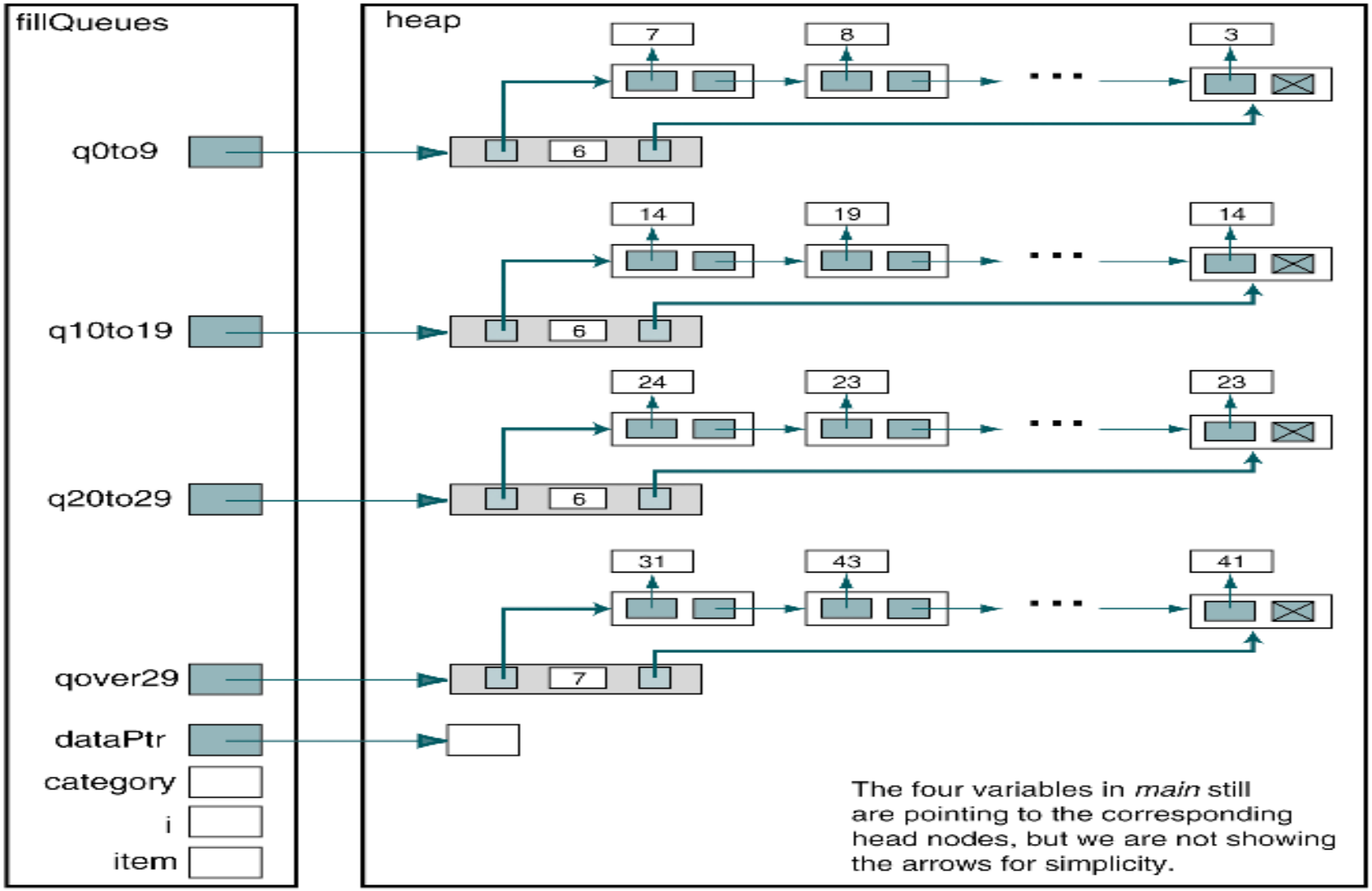
```
Algorithm fillQueues (q0to9, q10to19, q20to29, qOver29)
This algorithm reads data from the keyboard and places them in
one of four queues.
  Pre  all four queues have been created
  Post queues filled with data
1 loop (not end of data)
  1  read (number)
  2  if (number < 10)
     1  enqueue (q0to9, number)
  3  elseif (number < 20)
     1  enqueue (q10to19, number)
  4  elseif (number < 30)
     1  enqueue (q20to29, number)
  5  else
     1  enqueue (qOver29, number)
  6  end if
2 end loop
end fillQueues
```

Structures for Categorizing Data



(a) Before calling fillQueues

Structures for Categorizing Data (cont.)



(b) After calling fillQueues

Categorizing Data Machine

```
1  /*Groups numbers into four groups using four queues.
2       Written by:
3       Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <stdbool.h>
8  #include "queues.h"
9
10 //Prototype Statements
11 void fillQueues  (QUEUE*, QUEUE*, QUEUE*, QUEUE*);
12 void printQueues (QUEUE*, QUEUE*, QUEUE*, QUEUE*);
13
```

Categorizing Data Machine (cont.)

```
14  void printOneQueue (QUEUE* pQueue);
15
16  int main (void)
17  {
18  //Local Definitions
19    QUEUE* q0to9;
20    QUEUE* q10to19;
21    QUEUE* q20to29;
22    QUEUE* qOver29;
23
```

Categorizing Data Machine (cont.)

```
24  //Statements
25    printf("Welcome to a demonstration of categorizing\n"
26           "data. We generate 25 random numbers and then\n"
27           "group them into categories using queues.\n\n");
28
29    q0to9   = createQueue ();
30    q10to19 = createQueue ();
31    q20to29 = createQueue ();
32    qOver29 = createQueue ();
33
34    fillQueues  (q0to9, q10to19, q20to29, qOver29);
35    printQueues (q0to9, q10to19, q20to29, qOver29);
36
37    return 0;
38  } // main
```

Categorizing Data : Fill Queues

```
1  /*================= fillQueues ================
2    This function generates data using rand() and
3    places them in one of four queues.
4      Pre:  All four queues have been created
5       Post: Queues filled with data
6  */
7  void fillQueues (QUEUE* q0to9,   QUEUE* q10to19,
8                   QUEUE* q20to29, QUEUE* qOver29)
9  {
10 //Local Definitions
11   int  category;
12   int  item;
13   int* dataPtr;
```

Categorizing Data : Fill Queues (cont.)

```
14
15  //Statements
16    printf("Categorizing data:\n");
17    srand(79);
18
19    for (int i = 1; i <= 25; i++)
20        {
21        if (!(dataPtr = (int*) malloc (sizeof (int))))
22            printf("Overflow in fillQueues\a\n"),
23                    exit(100);
24
25         *dataPtr = item = rand() % 51;
26         category = item / 10;
27         printf("%3d", item);
28         if (!(i % 11))
29            // Start new line when line full
30            printf("\n");
31
```

Categorizing Data : Fill Queues (cont.)

```
32          switch (category)
33              {
34               case 0 : enqueue (q0to9, dataPtr);
35                        break;
36               case 1 : enqueue (q10to19, dataPtr);
37                        break;
38               case 2 : enqueue (q20to29, dataPtr);
39                        break;
40               default: enqueue (qOver29, dataPtr);
41                        break;
42              } // switch
43        } // for
44    printf("\nEnd of data categorization\n\n");
45    return;
46  } // fillQueues
```

Categorizing Data : Print Queues

```
1  /*================= printQueues =================
2    This function prints the data in each of the queues.
3       Pre  Queues have been filled
4       Post Data printed and dequeued
5  */
6  void printQueues (QUEUE* q0to9,   QUEUE* q10to19,
7                    QUEUE* q20to29, QUEUE* qOver29)
8  {
9  //Statements
10   printf("Data   0.. 9:");
11   printOneQueue (q0to9);
12
13   printf("Data  10..19:");
14   printOneQueue (q10to19);
15
16   printf("Data  20..29:");
17   printOneQueue (q20to29);
18
19   printf("Data over 29:");
20   printOneQueue (qOver29);
21
22   return;
23 } // printQueues
```

Print One Queue

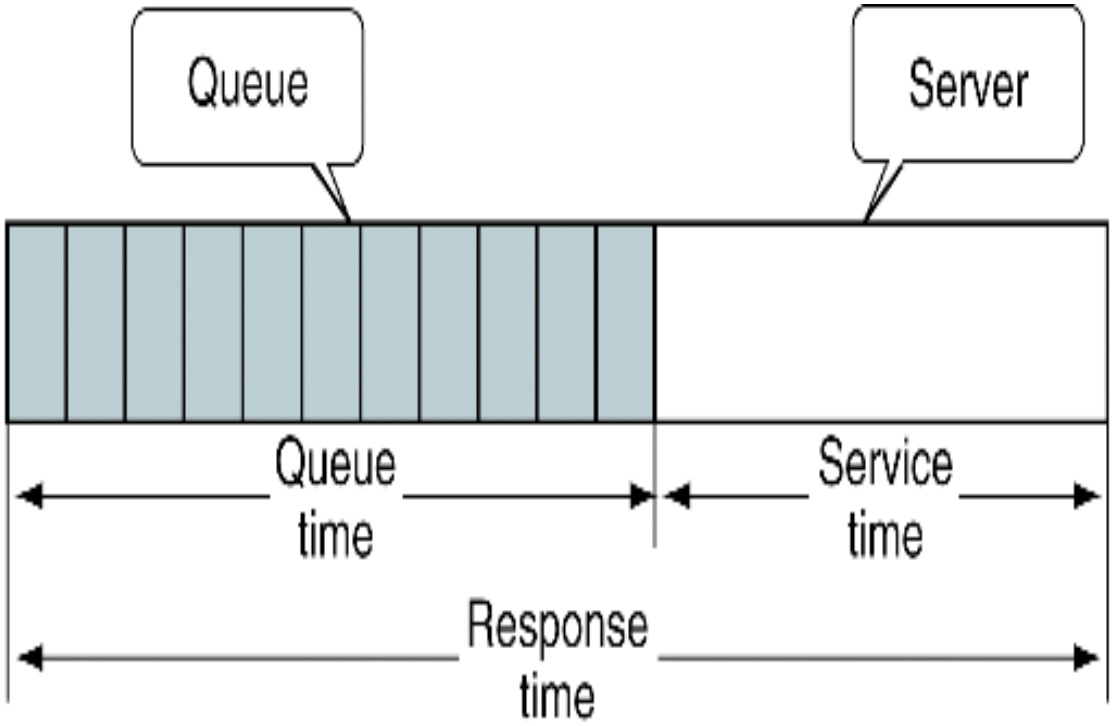
```
 7  void printOneQueue (QUEUE* pQueue)
 8  {
 9  //Local Definitions
10    int  lineCount;
11    int* dataPtr;
12
13  //Statements
14    lineCount = 0;
15    while (!emptyQueue (pQueue))
16         {
17          dequeue (pQueue, (void*)&dataPtr);
18          if (lineCount++ >= 10)
19              {
20               lineCount = 1;
21               printf ("\n                ");
22              } // if
23          printf("%3d ", *dataPtr);
24         } // while !emptyQueue
25    printf("\n");
26
27    return;
28  } // printOne Queue
```

Not necessary !

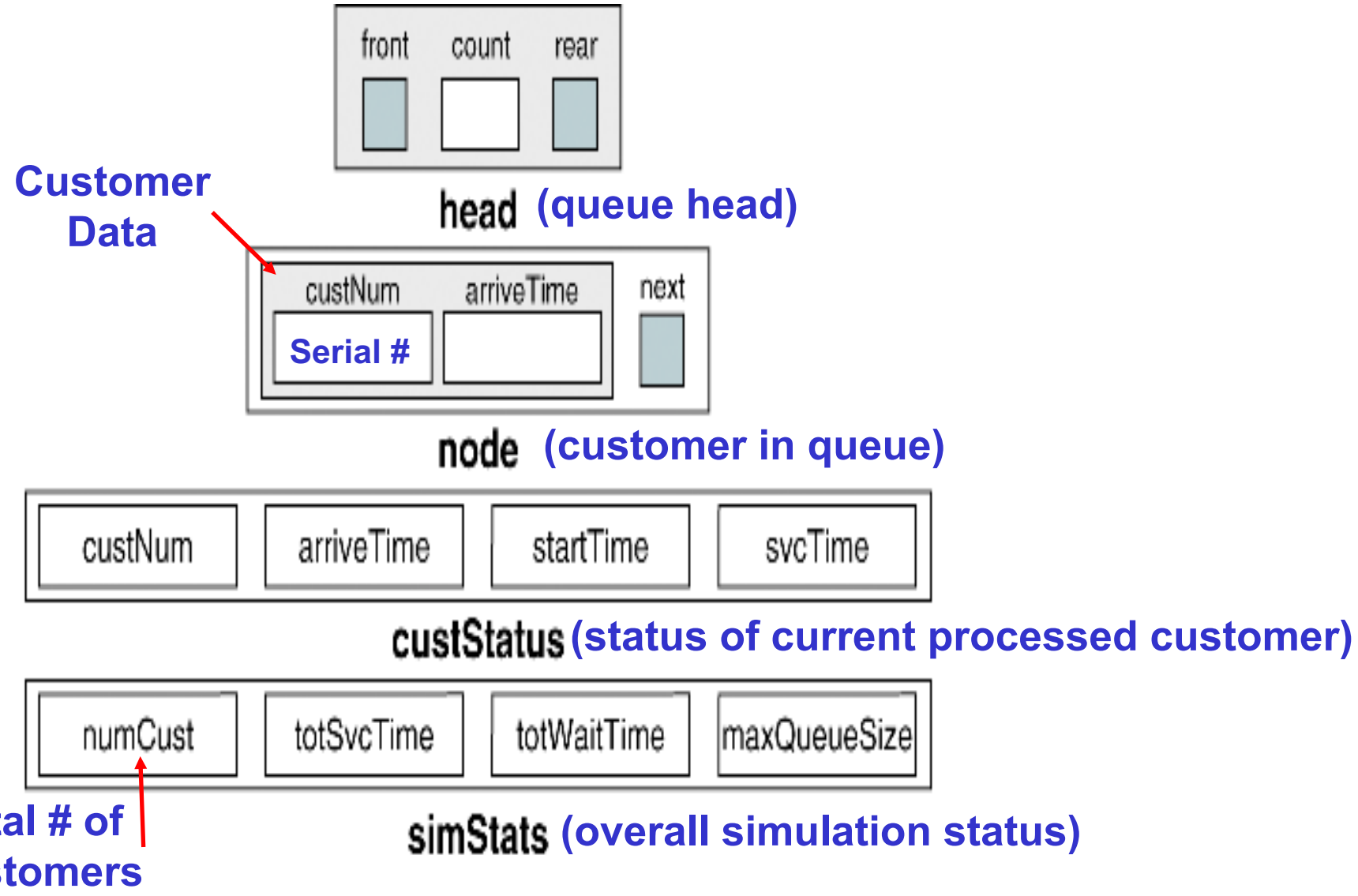
Categorizing Data Result

```
Results:
Welcome to a demonstration of categorizing
data. We generate 25 random numbers and then
group them into categories using queues.

Categorizing data:
 24  7 31 23 26 14 19  8  9  6 43
 16 22  0 39 46 22 38 41 23 19 18
 14  3 41
End of data categorization

Data   0.. 9:  7   8   9   6   0   3
Data  10..19: 14  19  16  19  18  14
Data  20..29: 24  23  26  22  22  23
Data over 29: 31  43  39  46  38  41  41
```

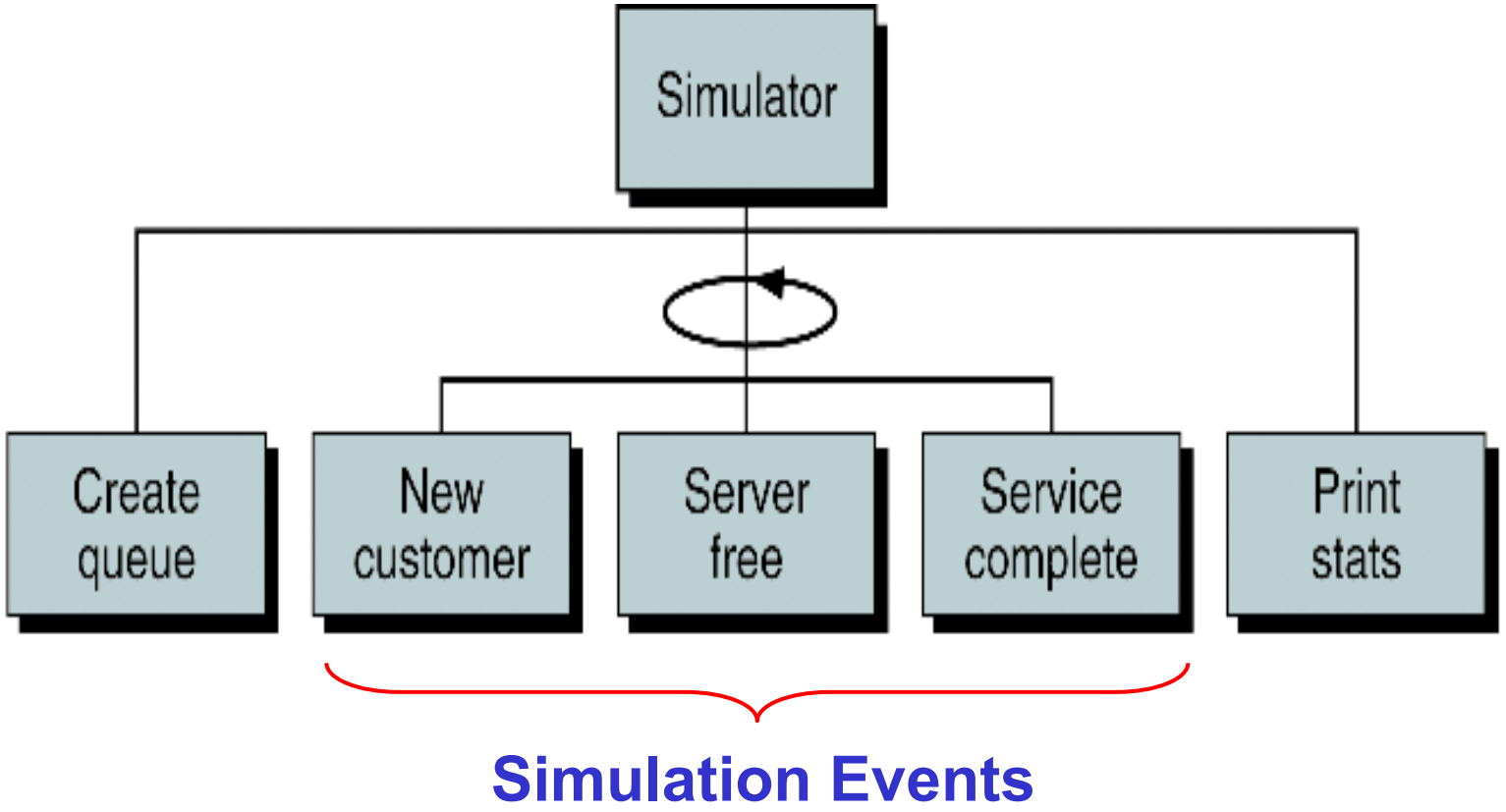
Queue Simulation



Queue Data Structures



Design for Queue Simulation



Queue Simulation: Driver

```
Algorithm taffySimulation
This program simulates a queue for a saltwater taffy store.
      Written by:
      Date:
1 createQueue (queue)
2 loop (clock <= endTime OR moreCusts)
  1  newCustomer  (queue, clock, custNum)
  2  serverFree   (queue, clock, custStatus, moreCusts)
  3  svcComplete  (queue, clock, custStatus,
                   runStats,     moreCusts)
  4  if (queue not empty)
     1  set moreCusts true
  5  end if
  6  increment clock
3 end loop
4 printStats (runStats)
end taffySimulation
```

Customer is being served or customers are waiting in the queue

If server is idle, set moreCusts = true

Queue Simulation: New Customer

```
Algorithm  newCustomer (queue, clock, custNum)
This algorithm determines if a new customer has arrived.
  Pre    queue is a structure to a valid queue
         clock is the current clock minute
         custNum is the number of the last customer
  Post   if new customer has arrived, placed in queue
1 randomly determine if customer has arrived
2 if (new customer)
  1  increment custNum
  2  store custNum in custData
  3  store arrival time in custData
  4  enqueue (queue, custData)
3 end if
end newCustomer
```

Queue Simulation: Server Free

```
Algorithm serverFree (queue, clock, status, moreCusts)
This algorithm determines if the server is idle and if so
starts serving a new customer.
  Pre   queue is a structure for a valid queue
        clock is the current clock minute
        status holds data about current/previous customer
  Post  moreCusts is set true if a call is started
1 if (clock > status startTime + status svcTime - 1)
     Server is idle.
  1  if (not emptyQueue (queue))
     1  dequeue (queue, custData)
     2  set status custNum    to custData number
     3  set status arriveTime to custData arriveTime
     4  set status startTime  to clock
     5  set status svcTime    to random service time
     6  set moreCusts true
  2  end if
2 end if
end serverFree
```

Queue Simulation: Service Complete

```
Algorithm svcComplete (queue, clock, status,
                       stats, moreCusts)
This algorithm determines if the current customer's
processing is complete.
  Pre    queue is a structure for a valid queue
         clock is the current clock minute
         status holds data about current/previous customer
         stats contains data for complete simulation

  Post   if service complete, data for current customer
         printed and simulation statistics updated
         moreCusts set to false if call completed
```

Queue Simulation: Service Complete

```
1 if (clock equal status startTime + status svcTime - 1)
  Current call complete
  1  set waitTime to status startTime - status arriveTime
  2  increment stats numCust
  3  set stats totSvcTime  to stats totSvcTime + status svcTime
  4  set stats totWaitTime to stats totWaitTime + waitTime
  5  set queueSize to queueCount (queue)
  6  if (stats maxQueueSize < queueSize)
     1  set stats maxQueueSize to queueSize
  7  end if
  8  print (status custNum    status arriveTime
            status startTime  status svcTime
            waitTime          queueCount (queue))
  9  set    moreCusts to false
 2 end if
end svcComplete
```

Hypothetical Simulation Service Times

Start time	Service time	Time completed	Minutes served
1	2	2	1 and 2
3	1	3	3
4	3	6	4, 5, and 6
7	2	8	7 and 8

Queue Simulation: Print Statistics

```
Algorithm printStats (stats)
This algorithm prints the statistics for the simulation.
  Pre    stats contains the run statistics
  Post   statistics printed
1 print (Simulation Statistics:)
2 print ("Total customers: "    stats numCust)
3 print ("Total service time: " stats totSvcTime)
4 set avrgSvcTime to stats totSvcTime / stats numCust
5 print ("Average service time: " avrgSvcTime)
6 set avrgWaitTime to stats totWaitTime / stats numCust
7 print ("Average wait time: " avrgWaitTime)
8 print ("Maximum queue size: " stats maxQueueSize)
end printStats
```