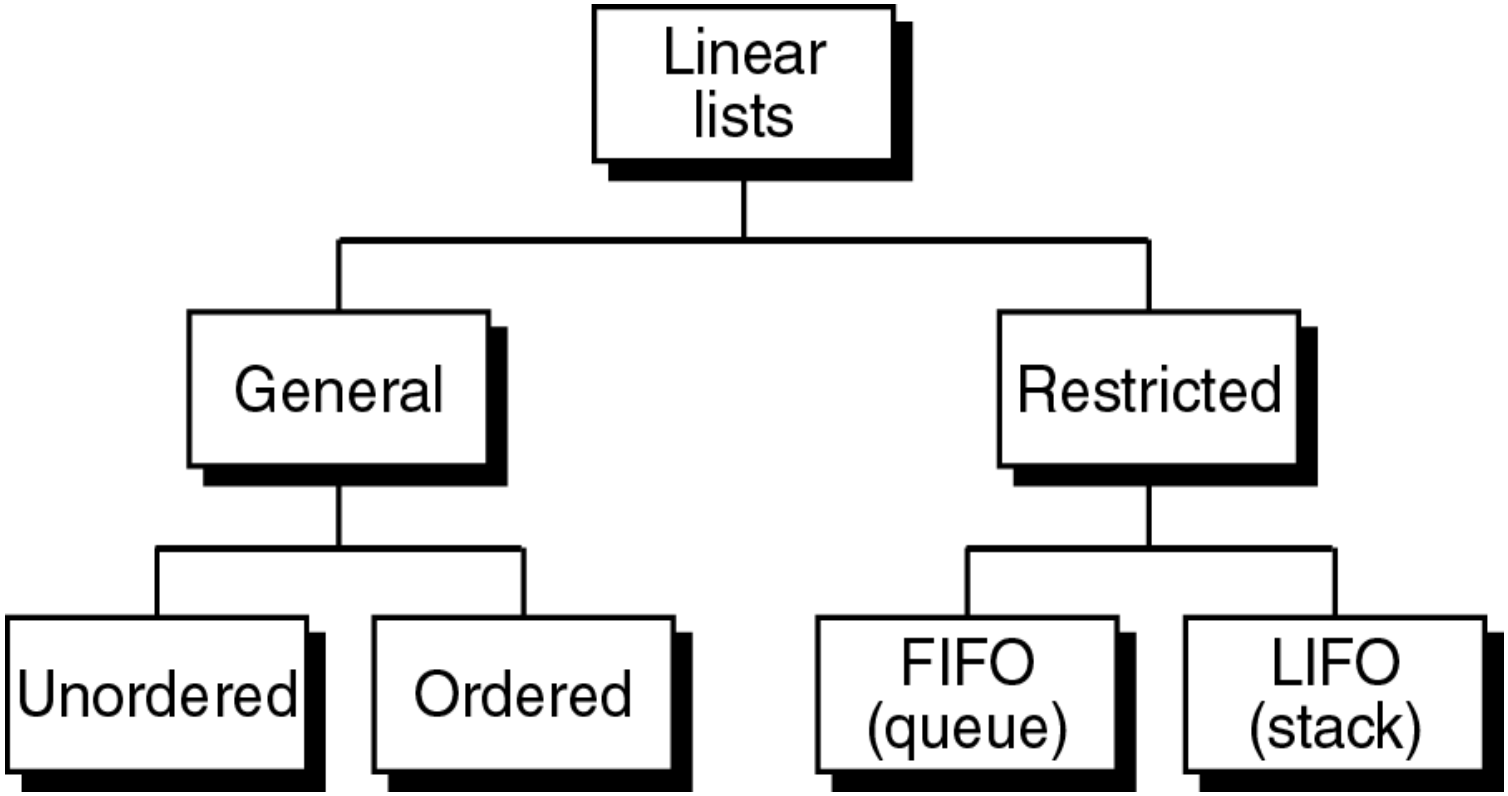
BLM212 Veri Yapıları

Queues
(Kuyruk)

2021-2022 Güz Dönemi

Linear Lists

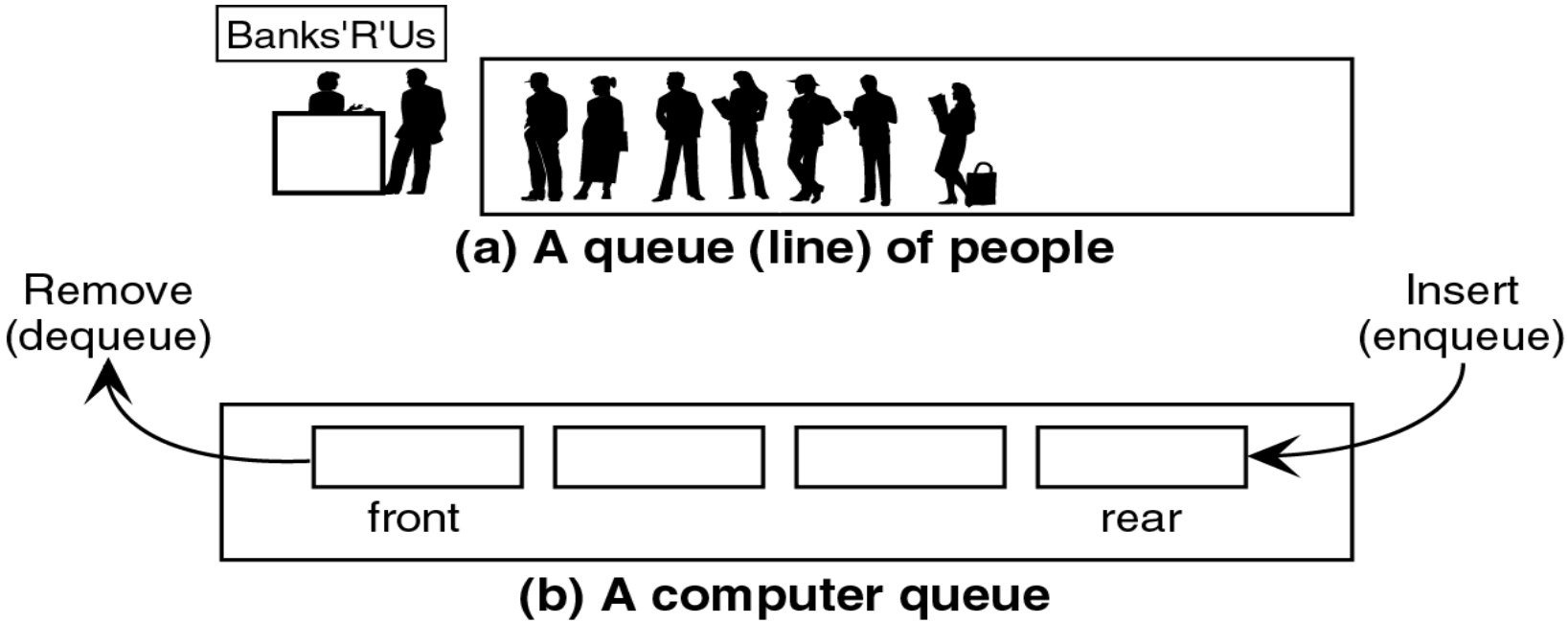


Operations are;

1. Insertion
2. Deletion
3. Retrieval
4. Traversal (exception for restricted lists).

Queue

Kuyruk bir doğrusal listedir.
Veriler bir uçtan eklenir (**rear**) ve diğer uçtan silinir (**front**)



First In First Out - FIFO

Veriler kuyruğa alındıkları sırayla işlenir

Queue Operations

- 4 temel kuyruk işlemi vardır
- Veriler kuyruğa sondan eklenir ve baş taraftan alınıp işlenir.

1. **Enqueue** ; kuyruğun sonuna bir eleman ekler.
2. **Dequeue** ; kuyruğun başından bir eleman siler.
3. **Queue Front**; kuyruğun başındaki elemanı okur.
4. **Queue Rear**; kuyruğun sonundaki elemanı okur.

Kuyruğun başı (**front**)

Kuyruğun sonu (**rear**)

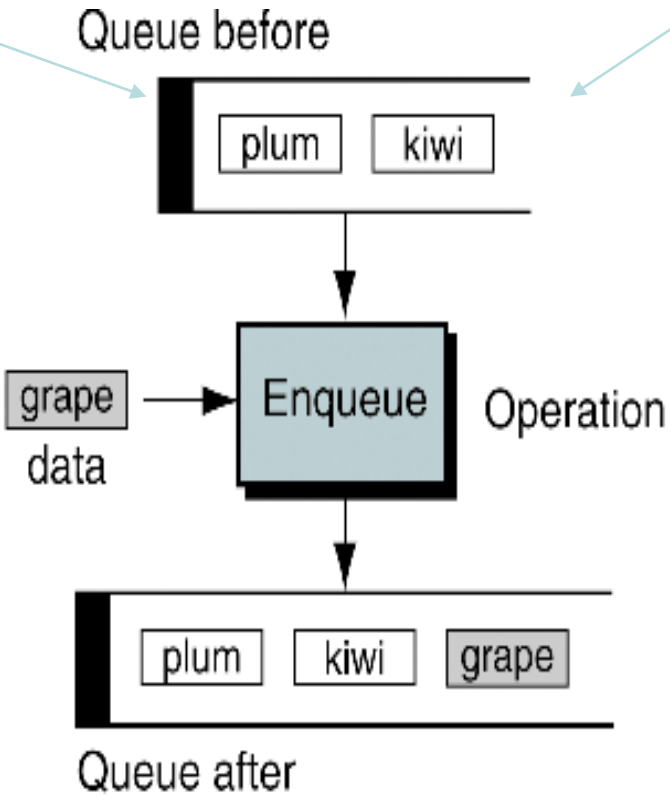


FIGURE 4-2 Enqueue

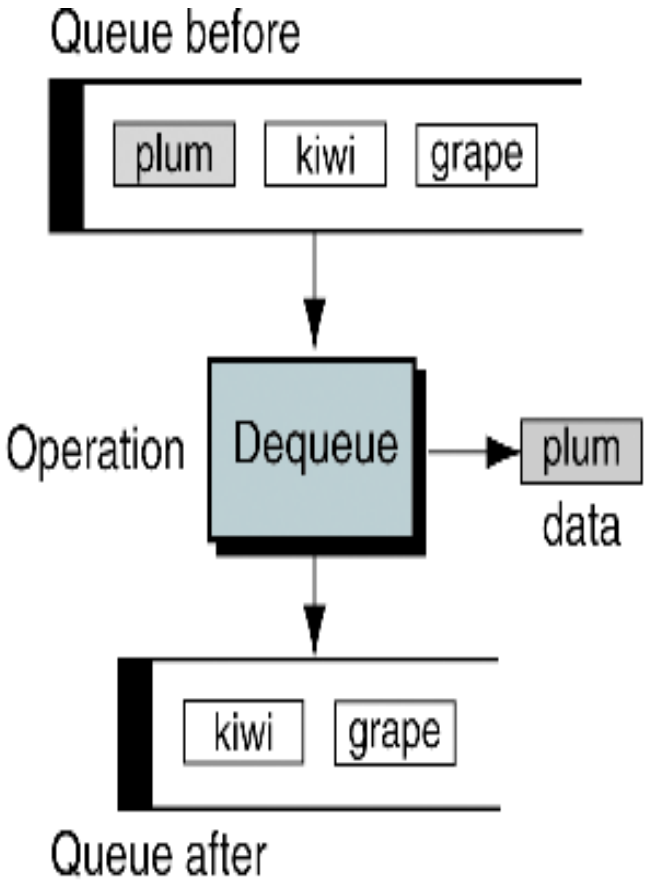


FIGURE 4-3 Dequeue

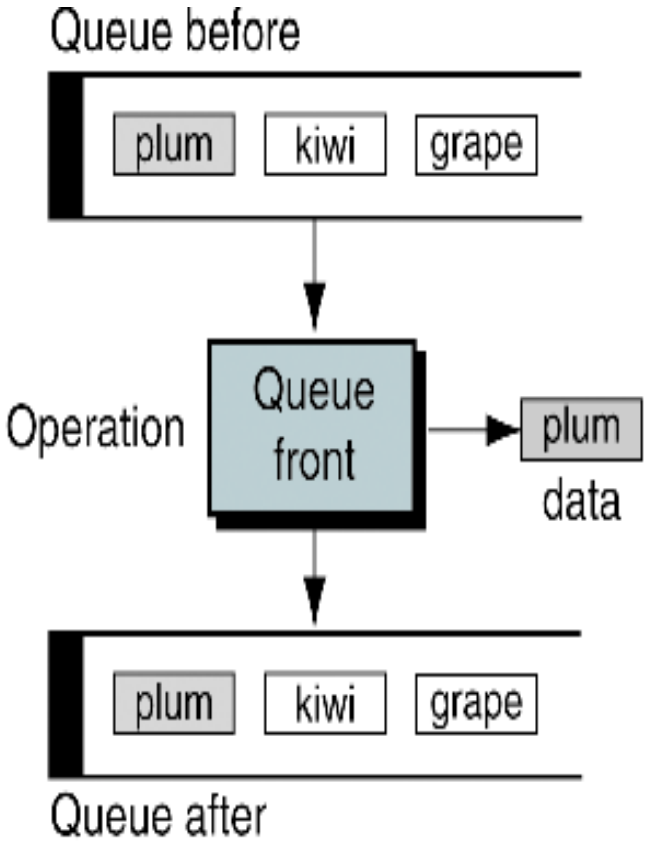


FIGURE 4-4 Queue Front

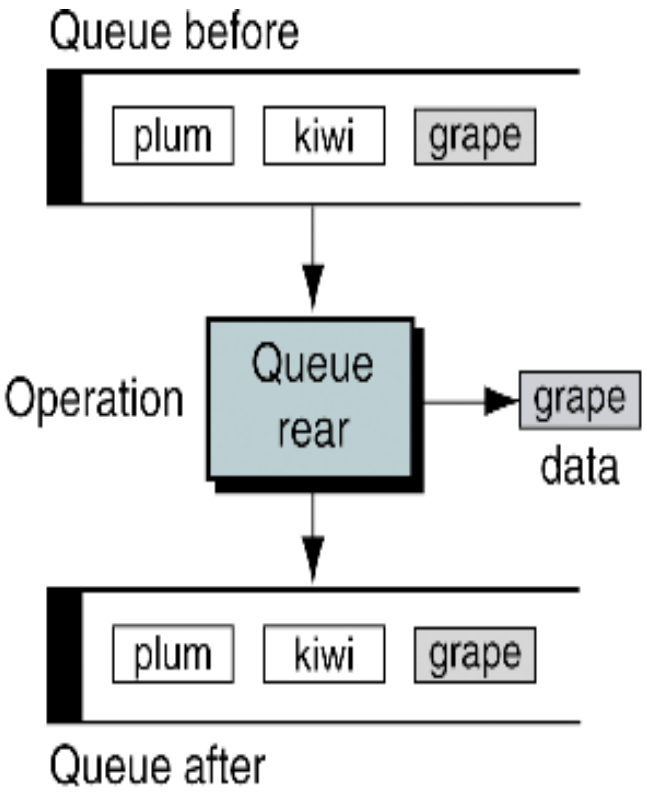


FIGURE 4-5 Queue Rear

FIGURE 4-6 Queue Example

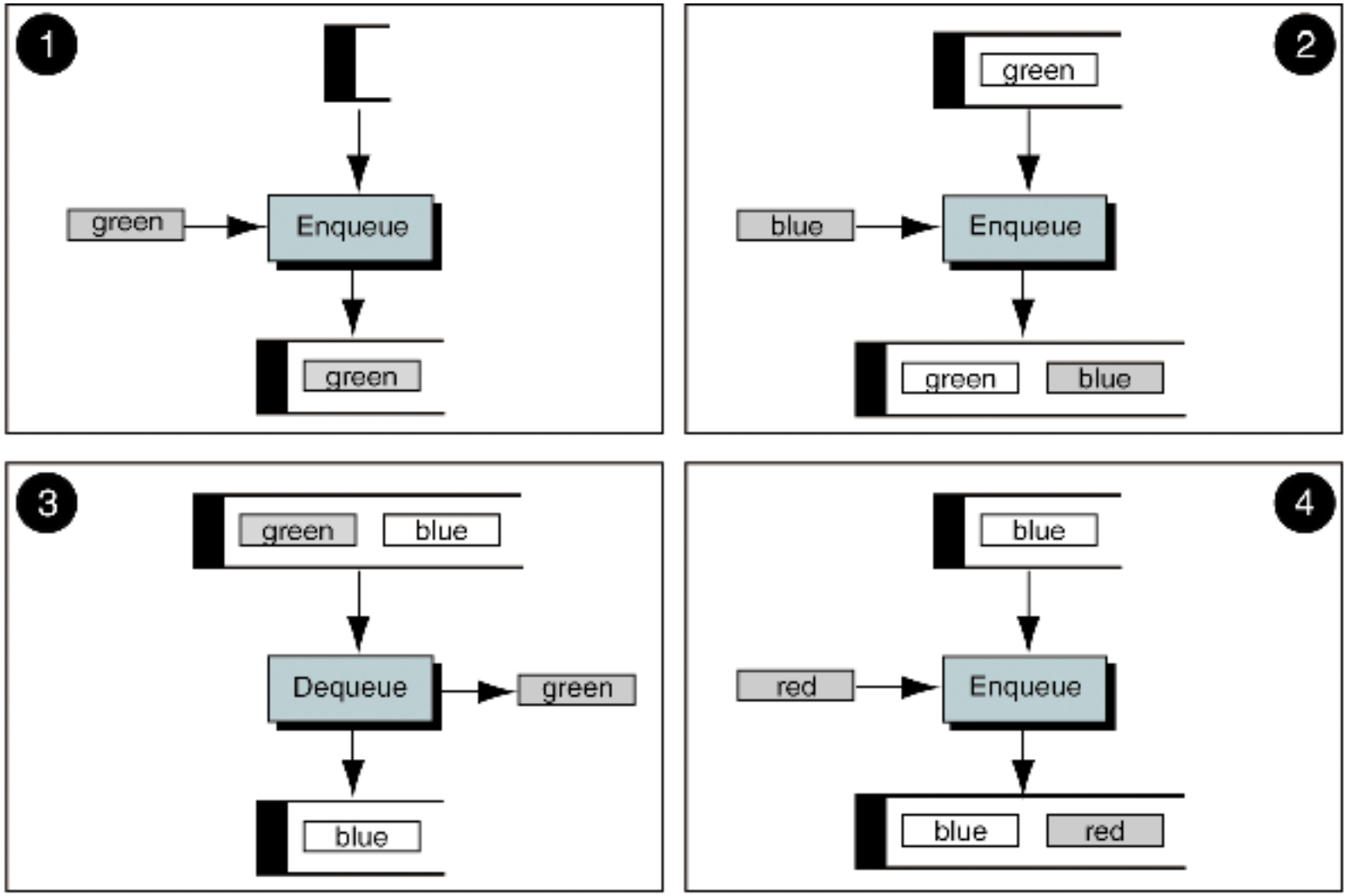
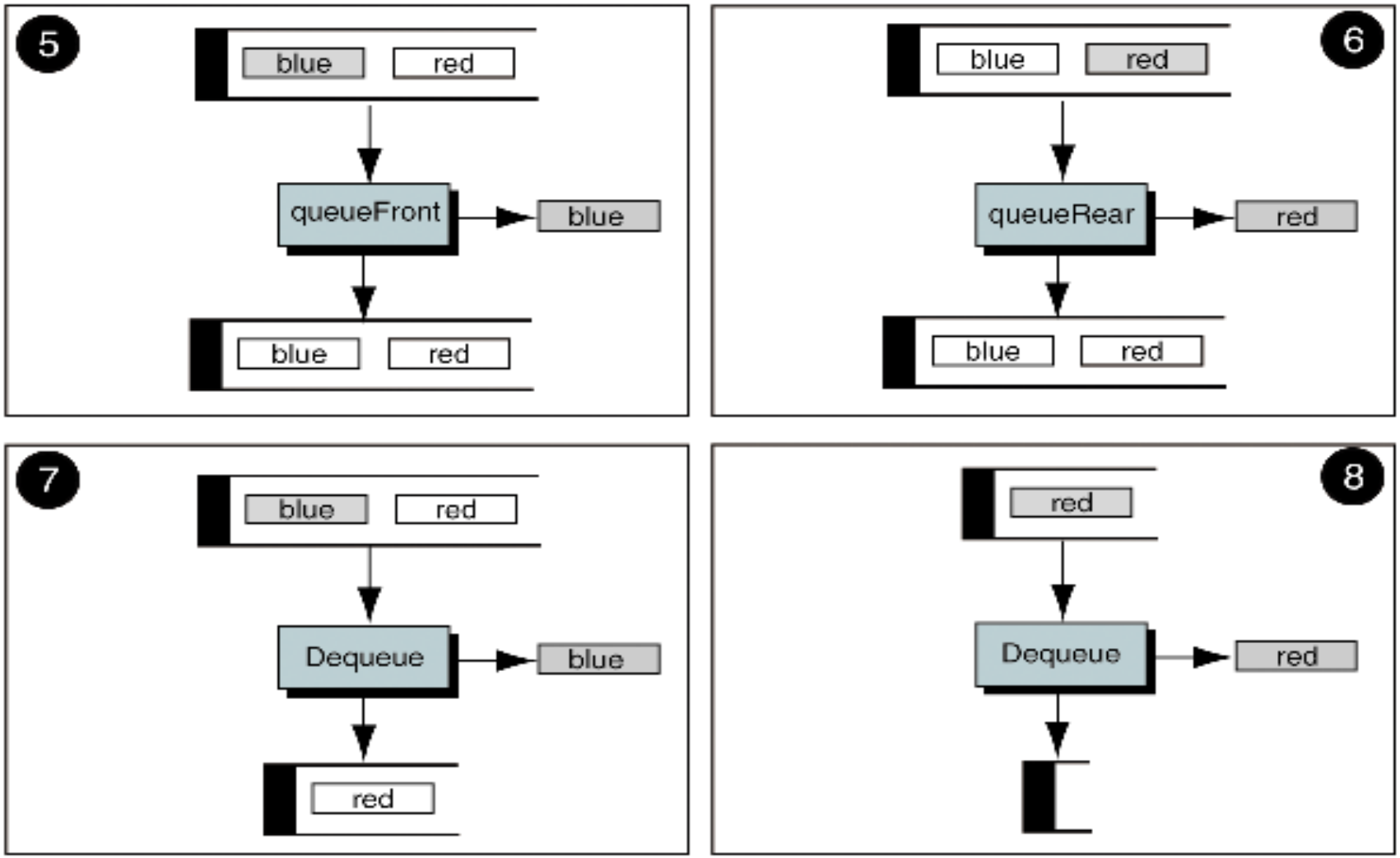


FIGURE 4-6 Queue Example (Continued)



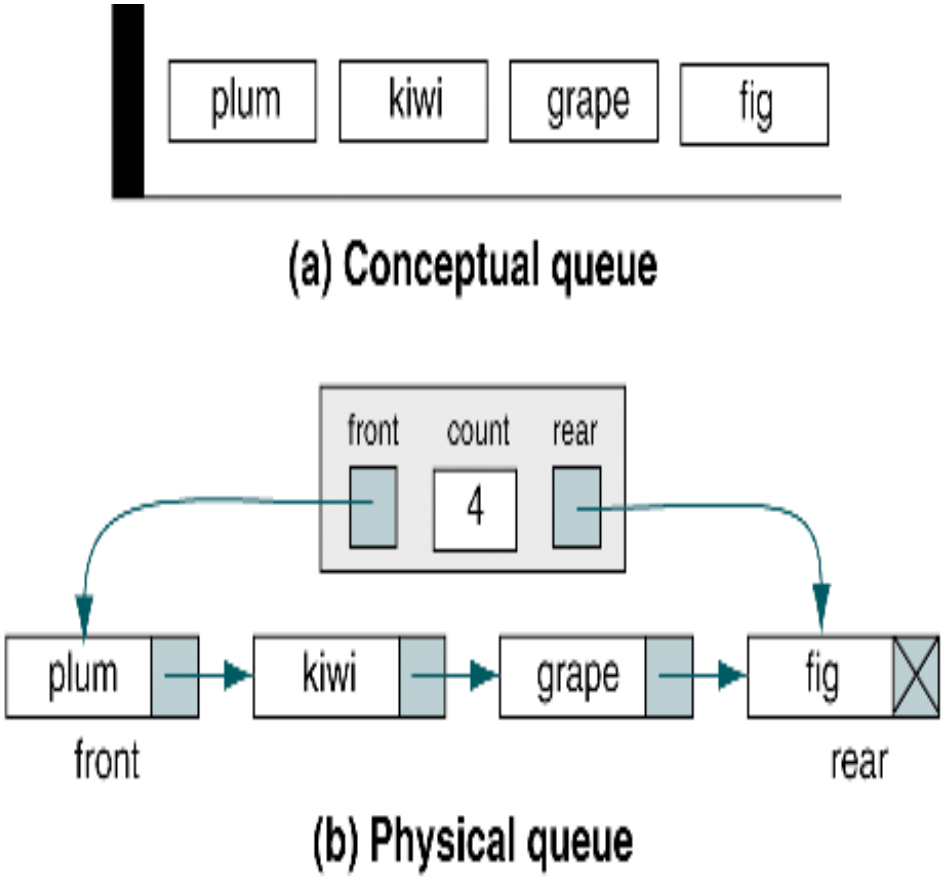


FIGURE 4-7 Conceptual and Physical Queue Implementations

Queue Algorithms - Create Queue

algorithm **createQueue**

Allocates memory for a queue head node from dynamic memory and returns its address to the caller.

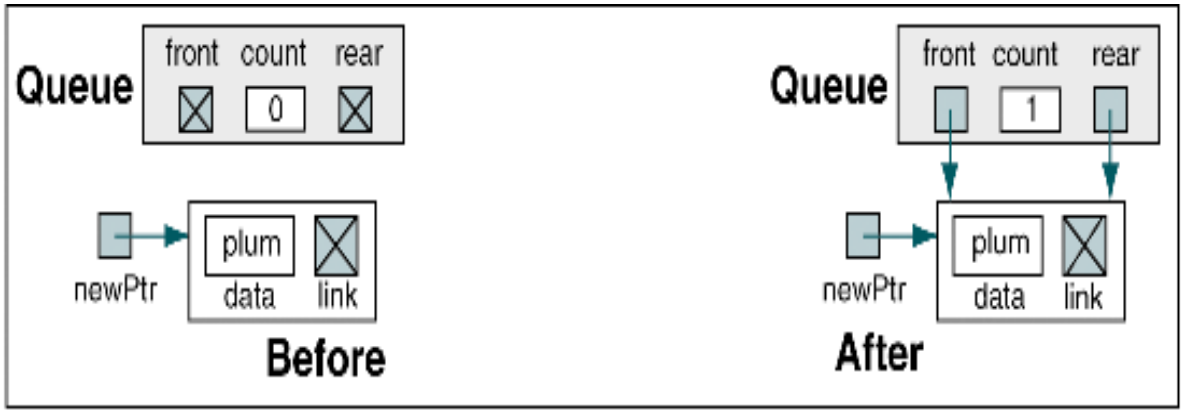
Pre Nothing

Post head has been allocated and initialized

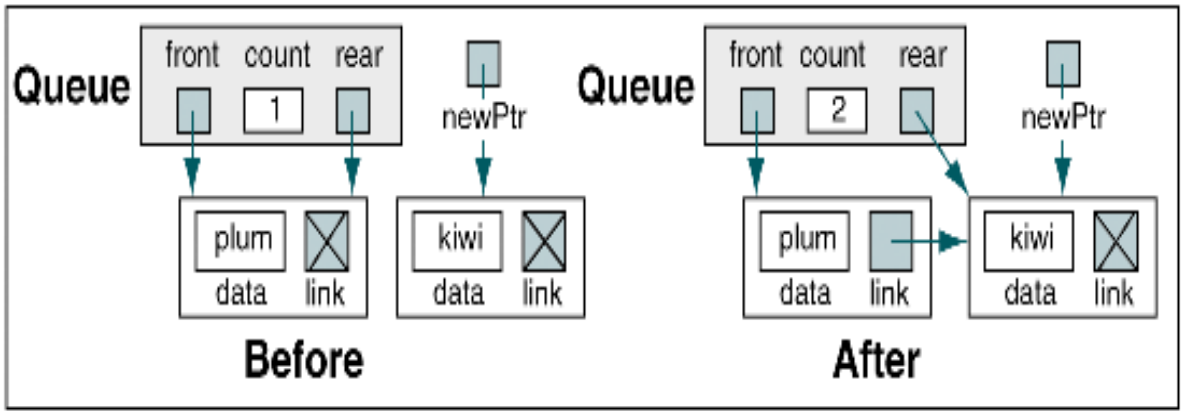
Return head's address if successful, null if memory owerflow.

1. if (memory available)
    1. allocate (newPtr)
    2. newPtr→front = null pointer
    3. newPtr→rear = null pointer
    4. newPtr→count = 0
    5. return newPtr
2. else
    1. return null pointer

end **createQueue**



(a) Case 1: insert into empty queue



(b) Case 2: insert into queue with data

FIGURE 4-10 Enqueue Example

Queue Algorithms - Enqueue

algorithm **enqueue**(val queue <head pointer>, val item <dataType>)

This algorithm inserts data into queue.

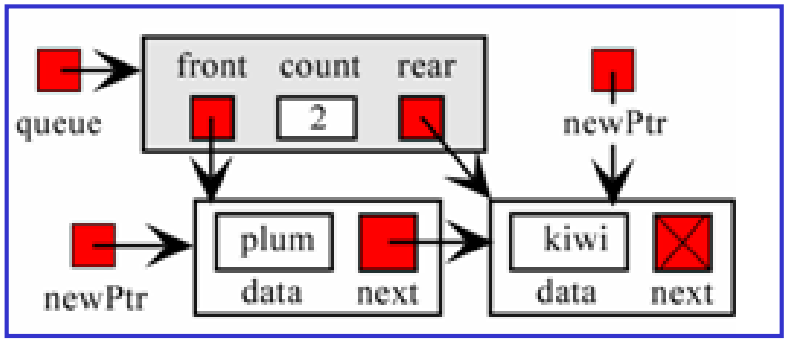
Pre queue has been created

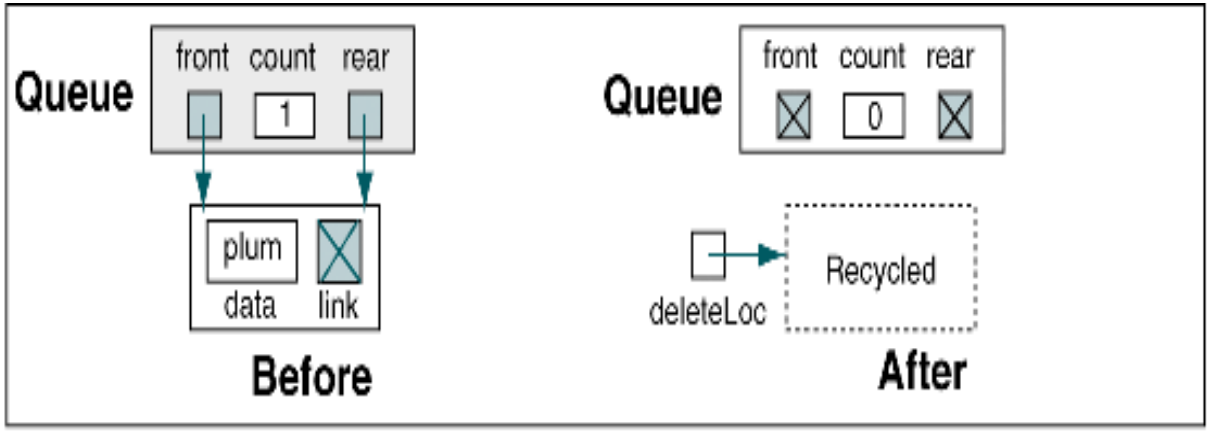
Post item data have been inserted

Return boolean, true if successful, false if overflow.

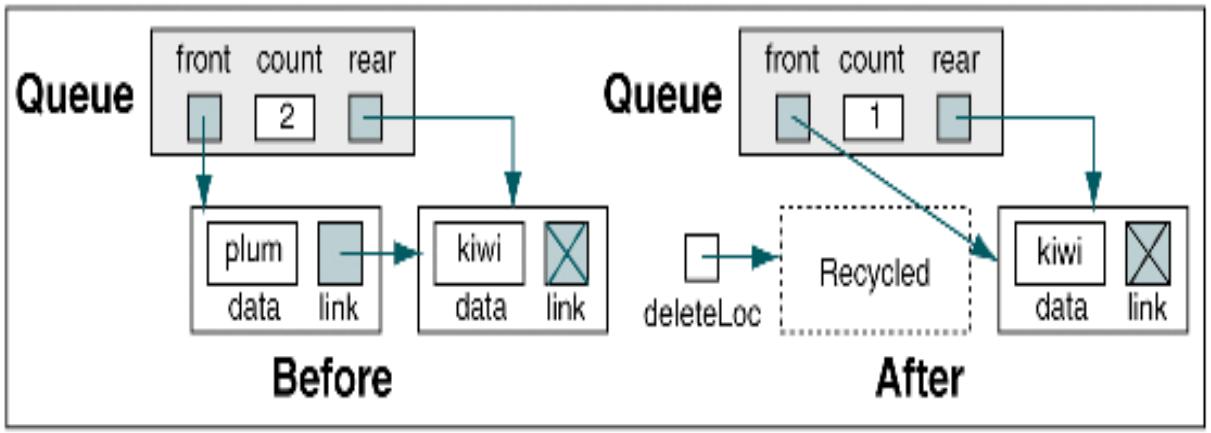
1. if (queue full)
    1. return false
2. allocate (newPtr)
3. newPtr→data = item
4. newPtr→next = null pointer
5. if (queue→count zero) //inserting into null queue
    1. queue→front = newPtr
6. else // insert data and adjust meta data
    1. queue→rear→next = newPtr
7. queue→rear = newPtr
8. queue→count = queue→count +1
9. return true

end **enqueue**





(a) Case 1: delete only item in queue



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

FIGURE 4-11 Dequeue Examples

Queue Algorithms - Dequeue

algorithm **dequeue**(val queue <head pointer>, ref item <dataType>)

This algorithm deletes a node from a queue.

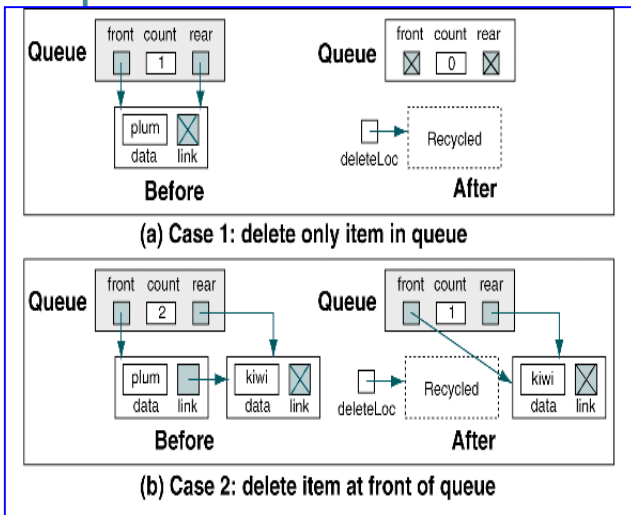
Pre queue has been create

Post data at front of the queue returned to user through item and front element deleted and recycled.

Return boolean, true if successful, false if overflow.

1. if (queue→count == 0)
    1. return false
2. item = queue→front→data
3. deleteLoc = queue→front
4. if (queue→count == 1) // deleting only item in queue
    1. queue→rear = null pointer
5. queue→front = queue→front→next
6. queue→count = queue→count – 1
7. recycle (deleteLoc)
8. return true

end **dequeue**



Queue Algorithms - **queueFront**

Kuyruk Verilerini Alma/Okuma (Retrieving Queue Data)

- Mantık, verilerin kuyruktan silinmemesi dışında, **Dequeue** ile aynıdır.
- İlk önce kuyruk boş mu diye kontrol eder ve boşsa «**false**» döndürür.
- Kuyrukta veri varsa, veriyi dataOut üzerinden geri iletir ve «**true**» değerini döndürür.

Queue Algorithms - queueFront

algorithm **queueFront**(val queue <head pointer>, ref dataOut <dataType>)

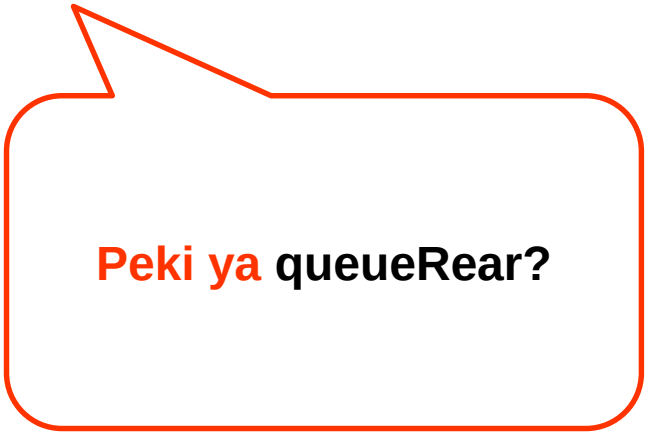
Retrieves data at the front of the queue without changing
queue contents.

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→count == 0) // (if queue is empty)
    1. return false
2.dataOut = queue→front→data
3.return true
end **queueFront**



ALGORITHM 4-5 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
```

ALGORITHM 4-6 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
```

ALGORITHM 4-7 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
```

ALGORITHM 4-8 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
```

PROGRAM 4-1 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
```

PROGRAM 4-2 Create Queue

```
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
```

PROGRAM 4-3 Enqueue

```
/*================= enqueue ================
  This algorithm inserts data into a queue.
     Pre    queue has been created
     Post   data have been inserted
     Return true if successful, false if overflow
*/
bool enqueue (QUEUE* queue, void* itemPtr)
{
//Local Definitions
  QUEUE_NODE* newPtr;

//Statements
  if (!(newPtr =
     (QUEUE_NODE*)malloc(sizeof(QUEUE_NODE))))
     return false;

  newPtr->dataPtr = itemPtr;
  newPtr->next    = NULL;

  if (queue->count == 0)
     // Inserting into null queue
     queue->front  = newPtr;
  else
     queue->rear->next = newPtr;

  (queue->count)++;
  queue->rear = newPtr;
  return true;
} // enqueue
```

PROGRAM 4-4 Dequeue

```
dequeue(queue, (void*)&dataPtr)
```

Dequeue kodu basit olmasına karşın bu fonksiyonu çağırma işi incelik ister. **Dequeue** fonksiyonunun data parametresi bir *void* pointer to pointer olduğu için bu fonksiyona çağrı yaparken bu data pointer'ı ***void*** türüne cast etmek gerekir.

```
/*================= dequeue ================
  This algorithm deletes a node from the queue.
     Pre    queue has been created
     Post   Data pointer to queue front returned and
            front element deleted and recycled.
     Return true if successful; false if underflow
*/
bool dequeue (QUEUE* queue, void** itemPtr)
{
//Local Definitions
  QUEUE_NODE* deleteLoc;

//Statements
  if (!queue->count)
      return false;

  *itemPtr  = queue->front->dataPtr;
  deleteLoc = queue->front;
  if (queue->count == 1)
     // Deleting only item in queue
     queue->rear  = queue->front = NULL;
  else
     queue->front = queue->front->next;
  (queue->count)--;
  free (deleteLoc);

  return true;
} // dequeue
```

PROGRAM 4-5 Queue Front

```
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
```

PROGRAM 4-7 Empty Queue

```
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
```

PROGRAM 4-9 Queue Count

```
1  /*================== queueCount ================
2    Returns the number of elements in the queue.
3       Pre    queue is pointer to the queue head node
4       Return queue count
5  */
6  int queueCount(QUEUE* queue)
7  {
8  //Statements
9    return queue->count;
10 } // queueCount
```

PROGRAM 4-8 Full Queue

```
 1  /*================== fullQueue ================
 2    This algorithm checks to see if queue is full. It
 3    is full if memory cannot be allocated for next node.
 4       Pre    queue is a pointer to a queue head node
 5       Return true if full; false if room for a node
 6  */
 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
```

PROGRAM 4-10 Destroy Queue

```
1  /*================== destroyQueue ================
2    Deletes all data from a queue and recycles its
3    memory, then deletes & recycles queue head pointer.
4       Pre    Queue is a valid queue
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
```

QUEUING THEORY

Kuyruk Teorisi (Queuing Theory), kuyrukların performansını tahmin etmek için kullanılan uygulamalı matematik ve bilgisayar bilimleri alanıdır.

Queuing Theory

- single-server queue: Aynı anda sadece bir müşteriye hizmet sunabilir (Örneğin köşe başında hizmet veren büfe).

- multi-server queue: Aynı anda birden fazla sayıda müşteriye hizmet verebilir (Örneğin banka, postane).

- Multiqueues: – (multiple single-server queues) birden çok tek sunuculu kuyruklar

Queuing Theory

- customer: Hizmete ihtiyacı olan herhangi bir kişi veya şeydir.
- service: (Servis/Hizmet) istenen sonucu elde etmek için gereken her türlü faaliyettir.

Queuing Theory

- Müşterilerin servis kuyruğuna geliş oranı/sıklığı/sürati "arrival rate" olarak bilinir. Rastgele veya düzenli olabilir.

- Service time (Hizmet süresi), bir müşteri isteğinin işlenmesini tamamlamak için gereken ortalama süredir.

Queuing Theory

- İdeal durumda, müşteriler hizmet süresine uygun bir oranda/sıklıkta gelirler.
- Ancak, işler nadiren ideale uygun olur. Hizmet verilecek müşteri olmadığında, bazen sunucu boşta olabilir. Diğer zamanlarda da hizmet verilecek çok sayıda müşteri olabilir.
- Eğer bu örüntüleri tahmin edebiliyorsak, boşta kalan sunucuları ve bekleyen müşterileri en aza indirebiliriz.

Queuing Theory

- Kuyruk Teorisi'ndeki ana görevlerden biri bu örüntüleri **tahmin etmektir**.

- Özellikle, kuyruk süresini (queue time – bu, müşterilerin kuyrukta beklediği ortalama süre olarak tanımlanır), kuyruğun ortalama boyutunu ve maksimum kuyruk boyutunu tahmin etmeye çalışır.

- Bu tahminler iki faktöre dayanmaktadır: geliş hızı (**arrival rate**) ve ortalama hizmet süresi (**average service time** - boşta kalma süreleri arasındaki toplam hizmet süresinin ortalaması)

Queuing Theory

- Kuyruk süresi (queue time) ve hizmet süresi (service time) verilirse, tepki/cevap süresini (response time) bulabiliriz- müşterilerin kuyruğa girdikleri andan sunucudan çıktıkları ana kadar olan ortalama sürenin bir ölçüsü.
- Tepki süresi, özellikle online bilgisayar sistemlerinde önemli bir istatistiktir.

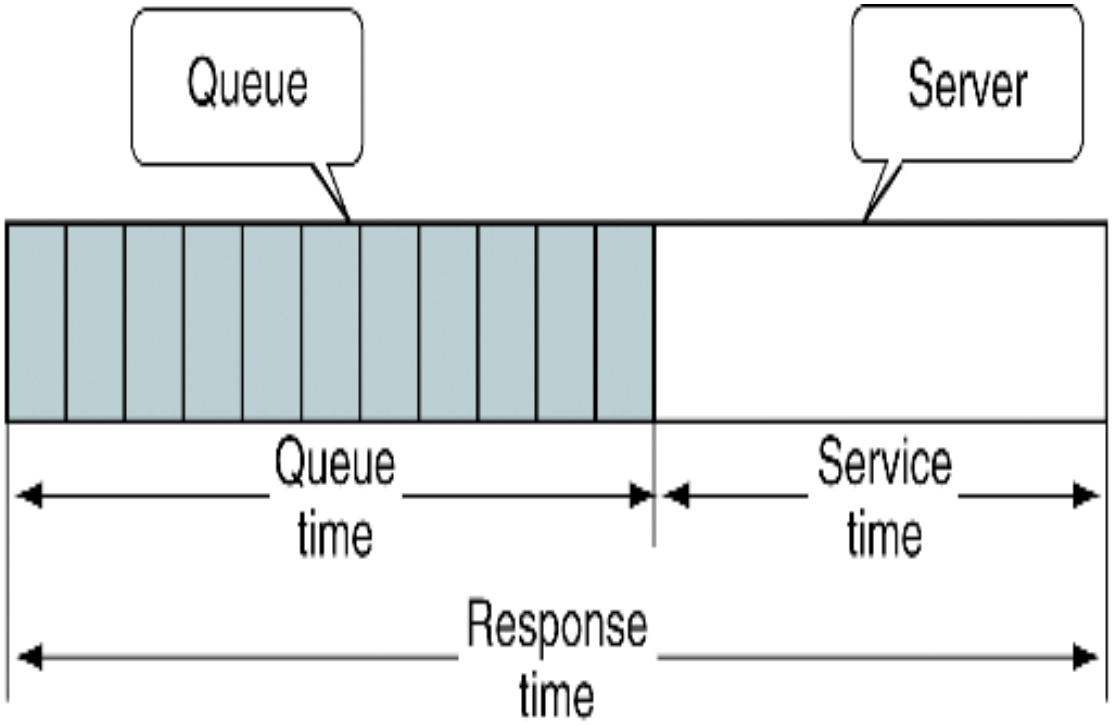


FIGURE 4-12 Queuing Theory Model

Queuing Theory

- Bir kuyruk modeli oluşturulduktan sonra, sistemde önerilen değişiklikleri incelemek için kullanılabilir.

- Örneğin, banka gişe kuyruğunda, ortalama servis süresini %15 azaltacak otomasyon kullanabilseydik, ne kadar daha az sayıda kişiye ihtiyaç duyardık?
- Veya, şu anda kapasite altında olan, büyüyen bir sistemin modeli göz önüne alındığında, başka bir sunucu eklememiz gerekmeden bu düzen ne kadar daha sürdürülebilirdi?

  gibi sorular…

Queuing Theory

- Kuyrukların performansını en çok etkileyen iki faktör
  - geliş oranı (**arrival rate**) ve
  - hizmet süresidir (**service time**).

Queue Applications

İki tane kuyruk uygulaması:
İlki, verileri sınıflandırmak için kuyruğun nasıl kullanılacağını gösterir.
İkincisi kuyruk simülatörüdür.

- **Categorizing Data**
- **Queue Simulation**

Categorizing Data

- Verileri Kategorize etme, temel sırayı bozmadan verilerin yeniden düzenlenmesidir.

- Örnek olarak: her grupta orijinal sıralamayı koruyarak, bir sayı dizisini gruplayıp yeniden düzenlenmesinin gerekli olduğunu varsayalım (multiple-queue application).

Categorizing Data

Initial list of numbers:

3 22 12 6 10 34 65 29 9 30 81 4 5 19 20 57 44 99

Onları dört farklı gruba ayırmamız gerekiyor:

Group 1: less than 10

Group 2: between 10 and 19

Group 3: between 20 and 29

Group4: 30 and greater

Bu, sıralama değil. Sonuç, sıralanmış bir liste değil, belirtilen kurallara göre kategorize edilmiş bir listedir.

| 3 6 9 4 5 | 12 10 19 | 22 29 20 | 34 65 30 81 57 44 99

Categorizing Data

- **Çözüm**: Dört kategorinin her biri için bir kuyruk oluştururuz. Sayıları okudukça uygun olan kuyrukta saklıyoruz.

- Tüm veriler işlendikten sonra, verileri doğru şekilde kategorize ettiğimizi göstermek için her kuyruğu yazdırırız.

ALGORITHM 4-9 Category Queues

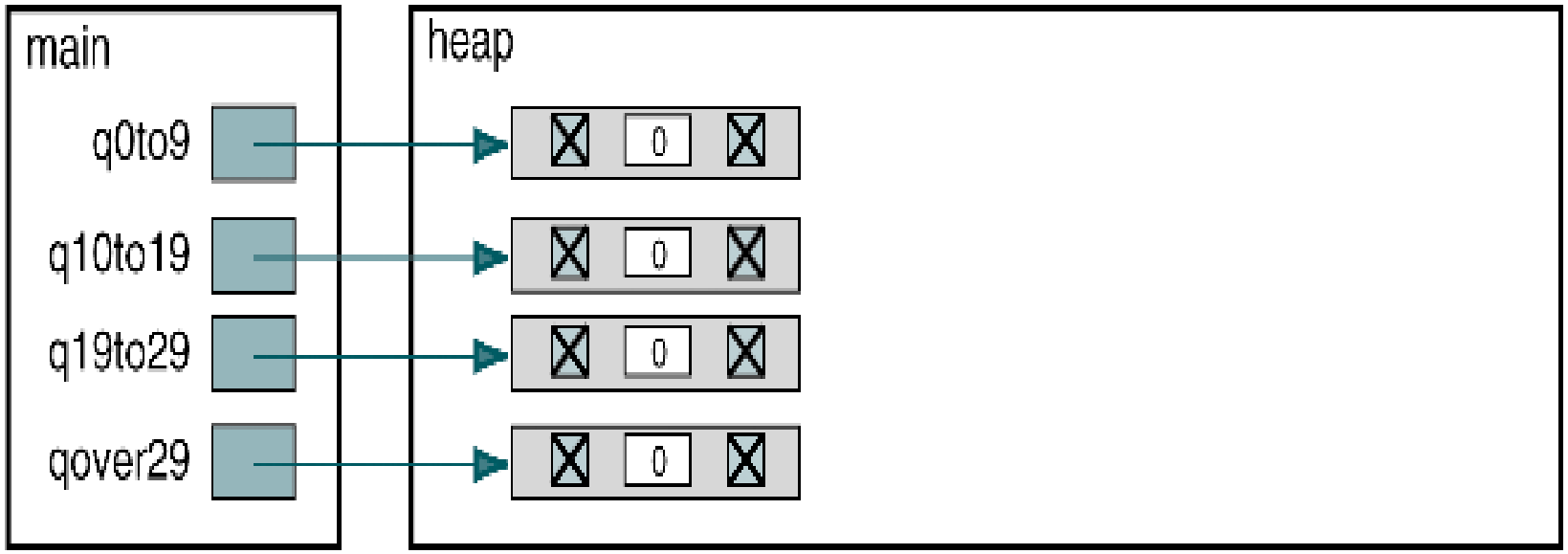
```
Algorithm categorize
  Group a list of numbers into four groups using four queues.
     Written by:
     Date:
1 createQueue (q0to9)
2 createQueue (q10to19)
3 createQueue (q20to29)
4 createQueue (qOver29)
5 fillQueues  (q0to9, q10to19, q20to29, qOver29)
6 printQueues (q0to9, q10to19, q20to29, qOver29)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

ALGORITHM 4-10 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
```

FIGURE 4-13 Structures for Categorizing Data



(a) Before calling fillQueues

PROGRAM 4-14 Print One Queue *(continued)*

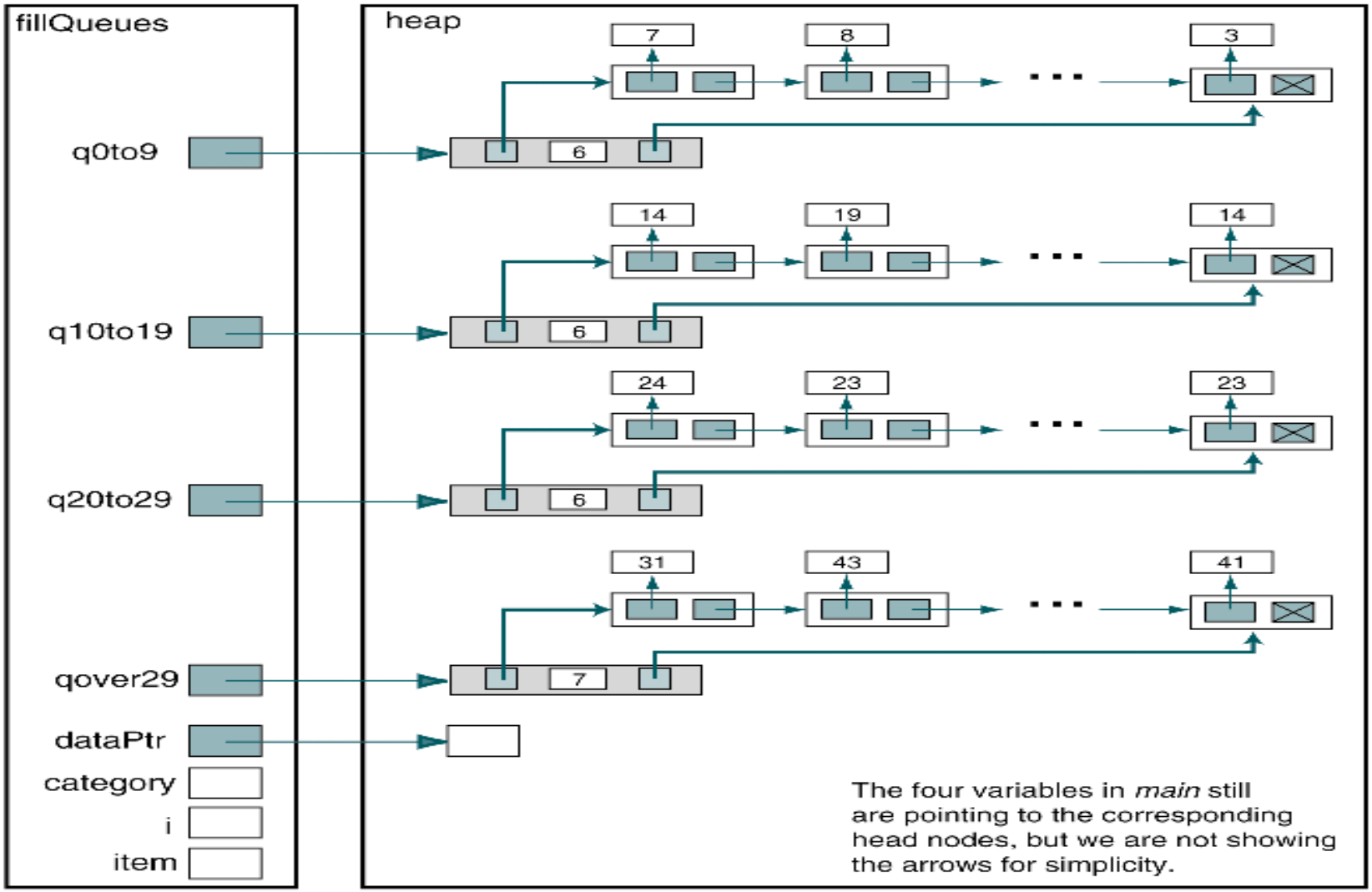
```
Results:
Welcome to a demonstration of categorizing
data. We generate 25 random numbers and then
group them into categories using queues.

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
```

Lab Uygulaması 1: Kitaptaki PROGRAM 4-11 Categorizing Data Mainline C kodu ve buradan çağırılan alt programlar PROGRAM 4-12 ~ 4-14 incelenecek.

FIGURE 4-13 Structures for Categorizing Data **(Continued)**



(b) After calling fillQueues

Queue Simulation

- Kuyruk simülasyonu (Queue simulation), kuyrukların performansı hakkında istatistik üretmek için kullanılan bir modelleme etkinliğidir.
- Tek sunuculu kuyruk (single-server queue) modeli oluşturma

Queue Simulation

- Bir büfenin müşterilerine hizmet vermesini simüle edelim.
- Büfenin bir penceresi vardır. Büfede bir çalışan vardır ve aynı anda bir müşteriye hizmet verebilmektedir. Müşterilere hizmet etme süresi 1-10 dakika arasında değişmektedir.
- Büfenin etkinliğini varsayımsal bir gün boyunca inceleyeceğiz. Büfe haftada 7 gün, her gün 8 saat boyunca açık. Bir günü simüle etmek için, 480 (8x60=480) dakikalık bir model oluşturulur.

- Aksiyonların 1 dakikalık aralıklarla başladığını ve durduğunu varsayıyoruz.

Queue Simulation

- Ortalama her 4 dakikada bir müşterinin geldiğini varsayıyoruz. Geliş sıklığını, 1 ile 4 arasında bir değer döndüren rastgele sayı üreteci kullanarak simüle ediyoruz. (Eğer sayı 4 ise, müşteri geldi. Eğer sayı 1, 2 veya 3 ise müşteri gelmedi)

- Sunucu boştayken bir müşterinin işlemine başlarız.

- Simülasyonun her dakikasında, simülatörün satıcı meşgul mü yoksa boş mu olduğunu belirlemesi gerekir.
  - Boştaysa, bekleyen bir sonraki müşteriye hizmet verilebilir. Meşgulse, bekleyen müşteriler kuyrukta kalır.

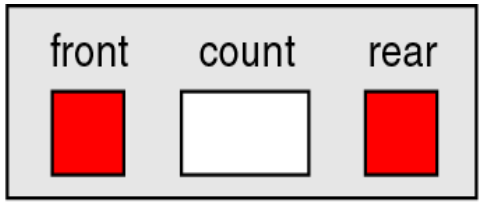
Queue Simulation

- Her dakikanın sonunda, simülasyon mevcut müşteri için işlemi tamamlayıp tamamlamadığını belirler.
- Mevcut müşterinin işlem süresi, müşteri işlemeye başladığında rastgele sayı üreteci tarafından simüle edilir. Ardından, her müşterinin işlemini tamamlamak için gereken dakika sayısını tekrarlıyoruz.
- Müşteriye tamamen hizmet verildiğinde, satışla ilgili istatistikleri toplar ve sunucuyu boşta durumuna getiririz.

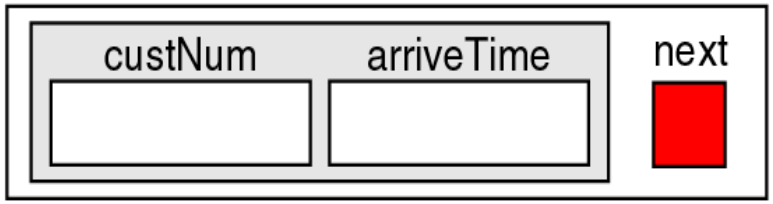
Queue Data Structures

Kuyruk simülasyonu için dört veri yapısı gereklidir:

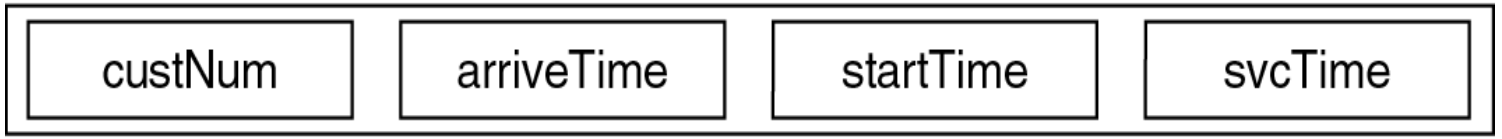
- a queue **head**,
- a queue **node**,
- a current **customer status**,
- a **simulation statistics** structure



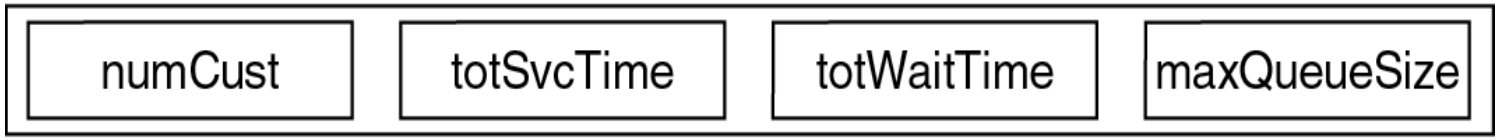
head



node



custStatus



simStats

Simulation Algorithm

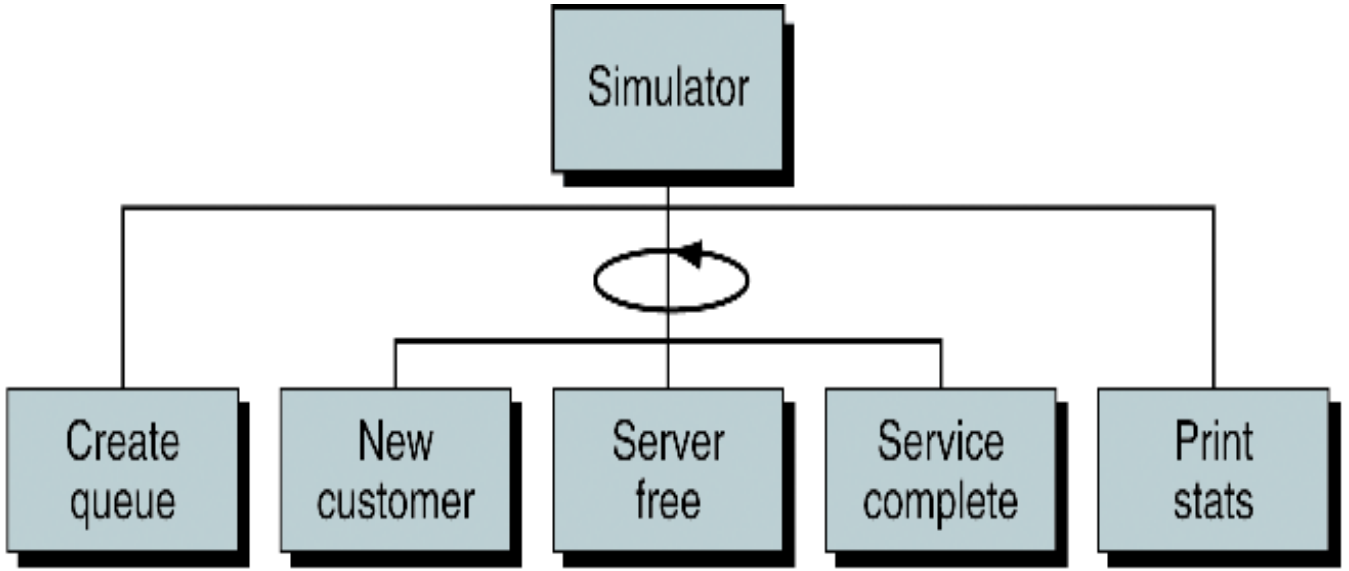


FIGURE 4-15 Design for Queue Simulation

ALGORITHM 4-11 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
```

ALGORITHM 4-12 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
```

ALGORITHM 4-13 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
```

ALGORITHM 4-14 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
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
```

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

TABLE 4-1 Hypothetical Simulation Service Times

ALGORITHM 4-15 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
```

Clock time	Call number	Arrival time	Wait time	Start time	Service time	Queue size
4	1	2	0	2	3	2
6	2	3	2	5	2	4

TABLE 4-2 Sample Output for Project 22

Queues Array Implementation

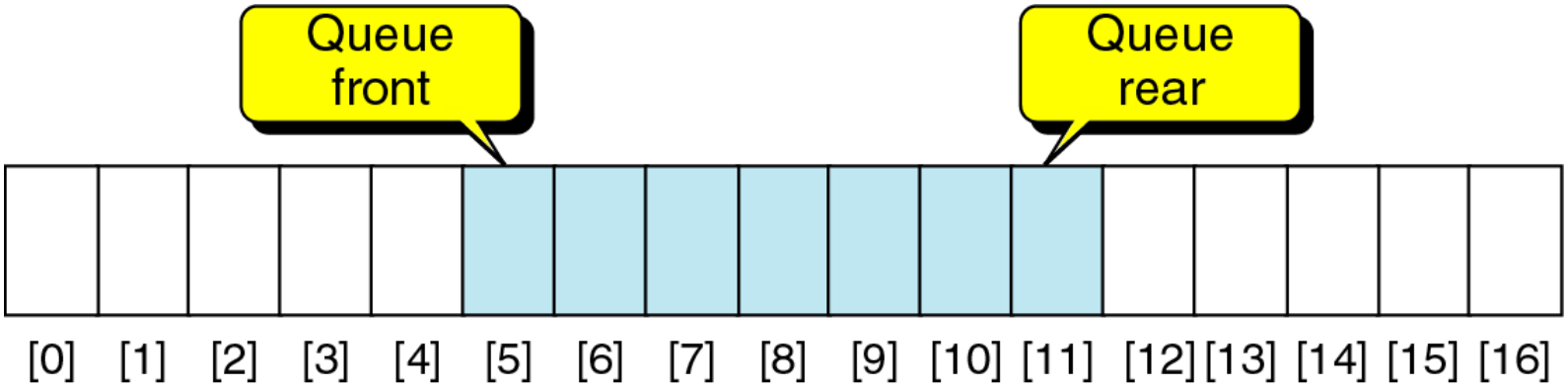


Figure 5-15

Queues Array Implementation

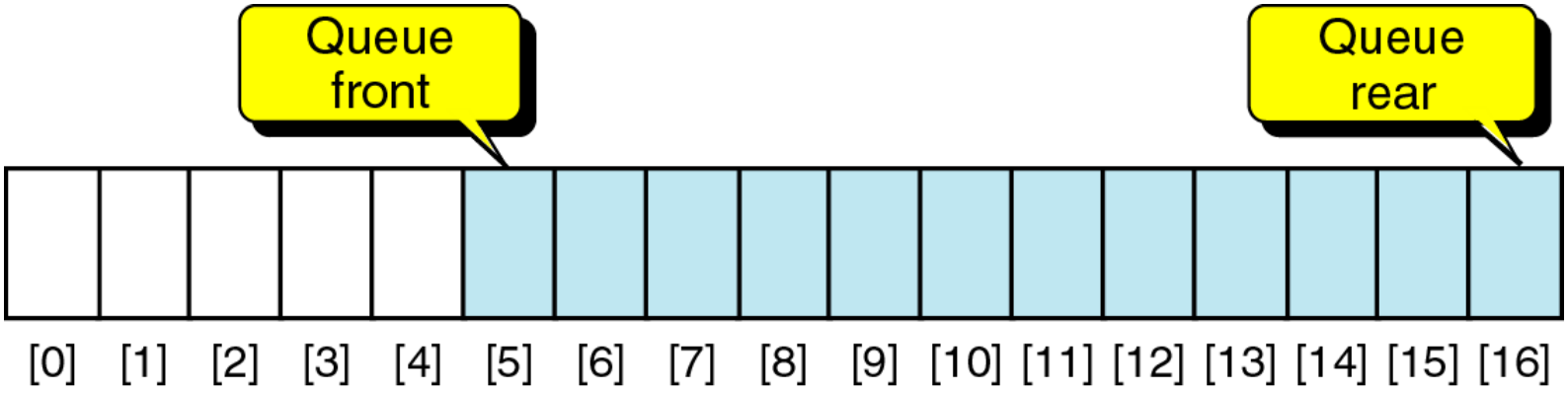


Figure 5-16

Exercise

Q1 ve Q2 kuyruklarının içeriğinin gösterildiği gibi olduğunu varsayın. Aşağıdaki kod çalıştırıldıktan sonra Q3'ün içeriği ne olurdu? Kuyruk içeriği önden (soldan) arkaya (sağdan) gösterilir.

Q1: **42 30 41 31 19 20 25 14 10 11 12 15**

Q2: **1 4 5 4 10 13**

```
1 Q3 = createQueue
2 count = 0
3 loop (not empty Q1 and not empty Q2)
        1 count = count + 1
        2 dequeue (Q1, x)
        3 dequeue (Q2, y)
        4 if (y equal to count)
                1 enqueue (Q3, x)
```

Exercise

Q1 ve Q2 kuyruklarının içeriğinin gösterildiği gibi olduğunu varsayın. Aşağıdaki kod çalıştırıldıktan sonra Q3'ün içeriği ne olurdu? Kuyruk içeriği önden (soldan) arkaya (sağdan) gösterilir.

Q1: **42 30 41 31 19 20 25 14 10 11 12 15**

Q2: **1 4 5 4 10 13**

```
1 Q3 = createQueue
2 count = 0
3 loop (not empty Q1 and not empty Q2)
        1 count = count + 1
        2 dequeue (Q1, x)
        3 dequeue (Q2, y)
        4 if (y equal to count)
                1 enqueue (Q3, x)
```

Step	count	Q1	Q2	Q3	x	y
1	0,1	42,30,..	1,4,5,4,..	42	42	1
2	2	30,41,..	4,5,4,..		30	4
3	3	42,31,19,..	5,4,10,13		42	5
4	4	31,19,20,..	4,10,13	42,31	31	4
5	5	19,20,25,..	10,13		19	10
6	6	20,25,14,..	13		20	13
7		25,14,10,..	NULL			

Q3: 42, 31

Ödev 5

Prefix ifadeleri hesaplamanın bir yolu da kuyruk yapısı kullanmaktır. İfadeyi hesaplamak için nihai değeri bulana kadar ifade tekrar tekrar taranır. Her taramada simgeler (tokens) okunur ve bir kuyrukta tutulur. Her taramada bir operatör ve onu takip eden iki operand varsa değeri hesaplanır ve onunla değiştirilir. Örneğin aşağıdaki ifade değeri 159 olarak hesaplanan bir prefix ifadesidir:

- + * 9 + 2 8 * + 4 8 6 3

İfadeyi tarar ve bir kuyrukta saklarız. Tarama esnasında bir operatörü izleyen 2 operand olduğunda, mesela + 2 8, kuyruğa 10 sonucunu koyarız.

İlk tarama dan sonra şunu elde ederiz:

- + * 9 10 * 12 6 3

İkinci taramadan sonra:

- + 90 72 3

Üçüncü taramadan sonra:

- 162 3

Dördüncü tarama:

159

Kuyruk yapısını kullanarak Prefix ifadelerin sonucunu hesaplayan genel bir uygulama C ile yazılacak.

Ödev 5 teslim tarihi: **?? Aralık 2020 23:59 (GitHub)**