QUEUES

Definition

- A **queue** is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.
- Hence, the data are processed through the queue in the order in which they are received (first in → first out – FIFO)

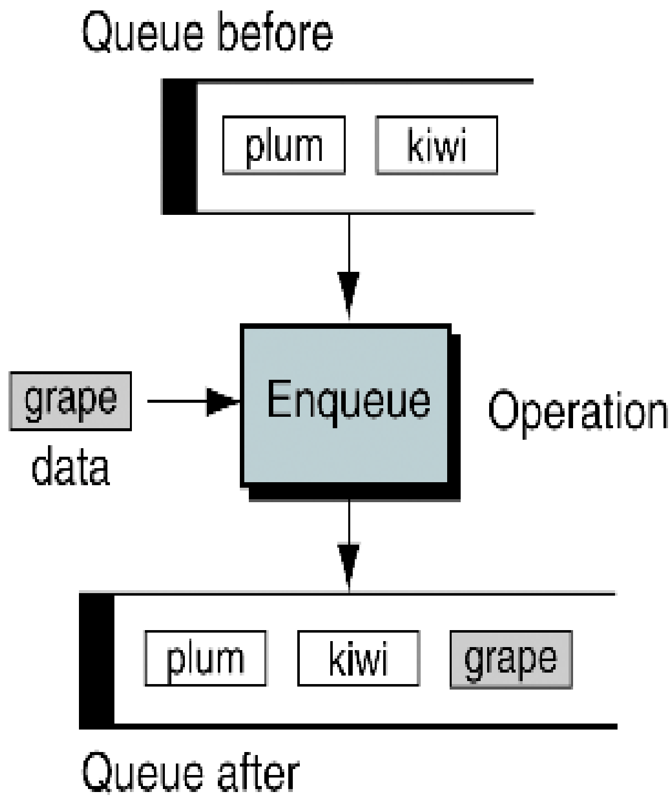


FIGURE 4-2 Enqueue

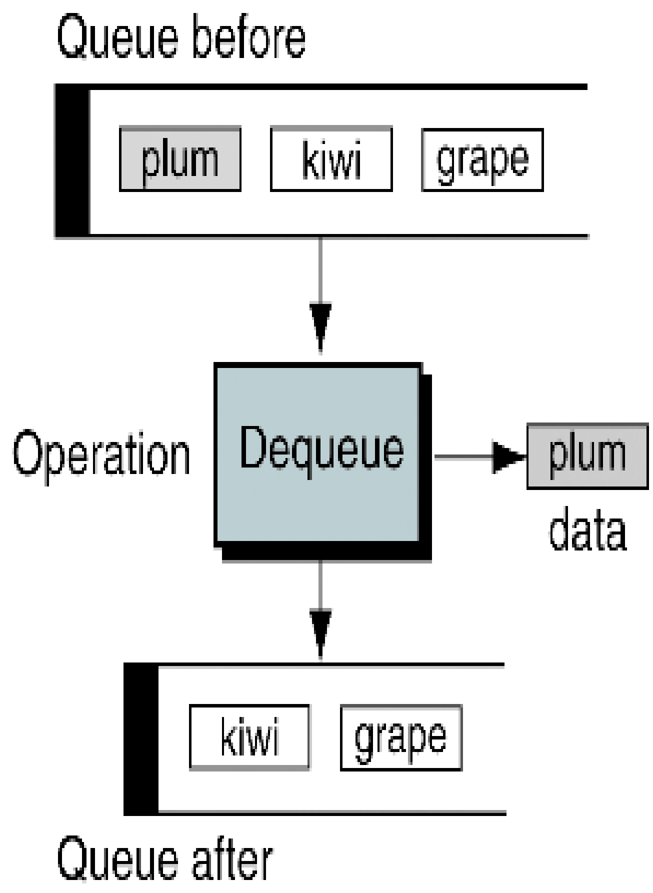


FIGURE 4-3 Dequeue

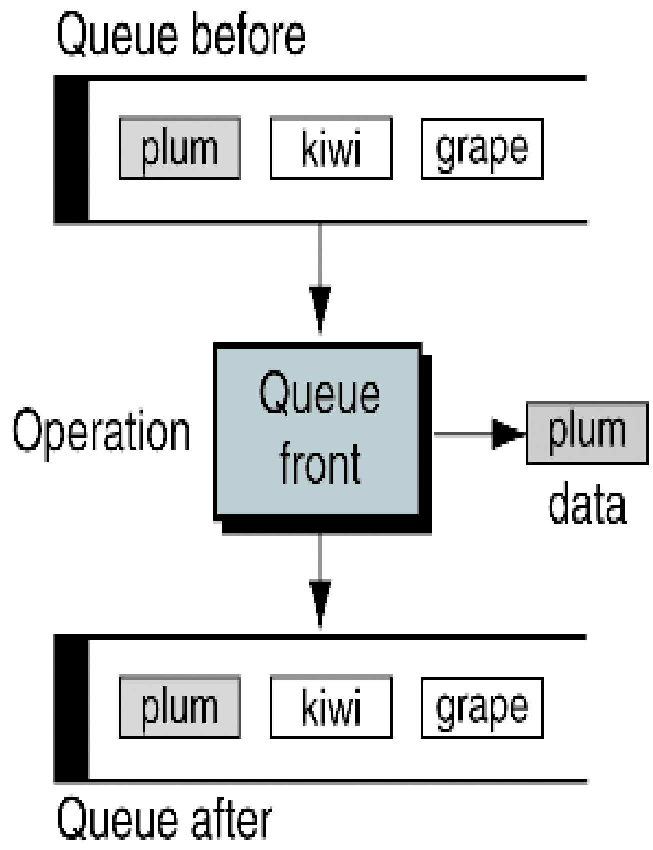


FIGURE 4-4 Queue Front

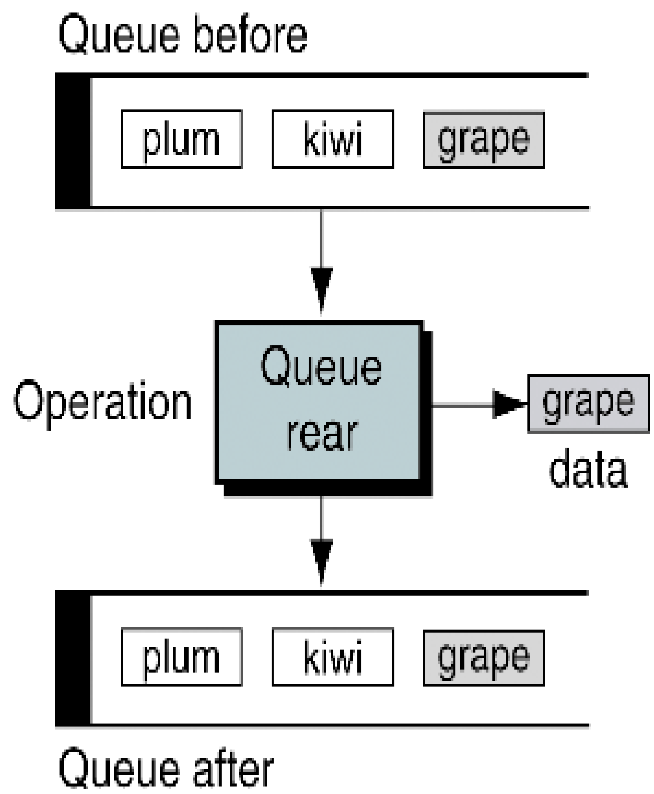


FIGURE 4-5 Queue Rear

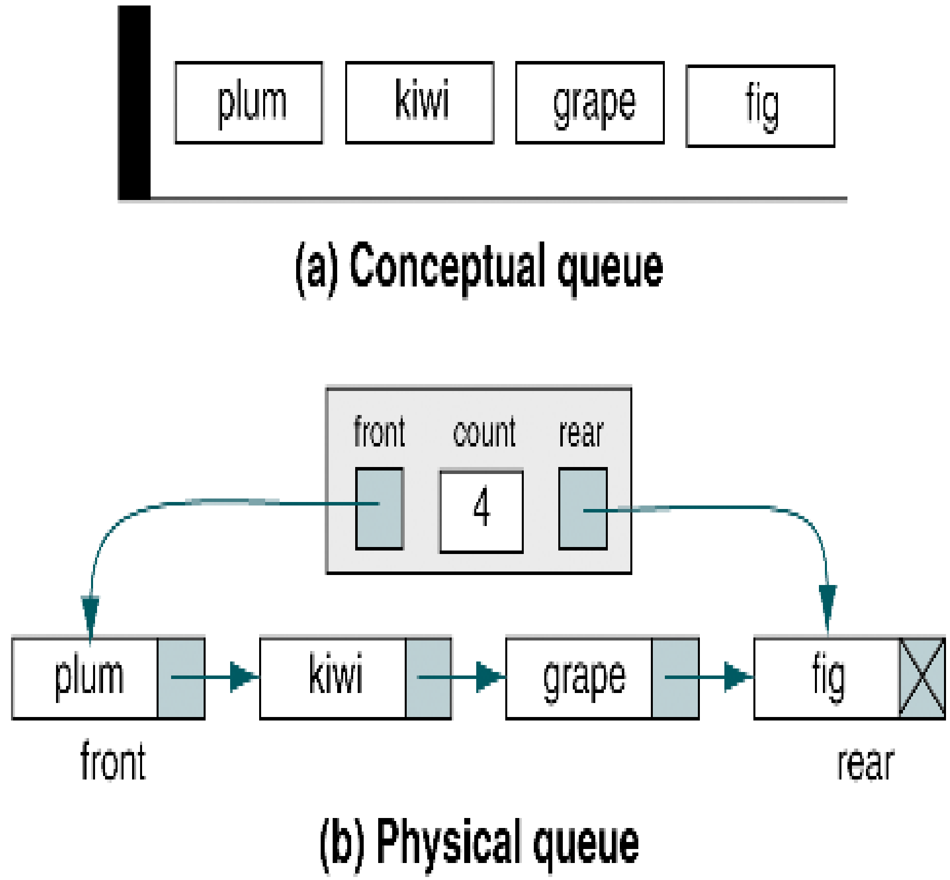


FIGURE 4-7 Conceptual and Physical Queue Implementations

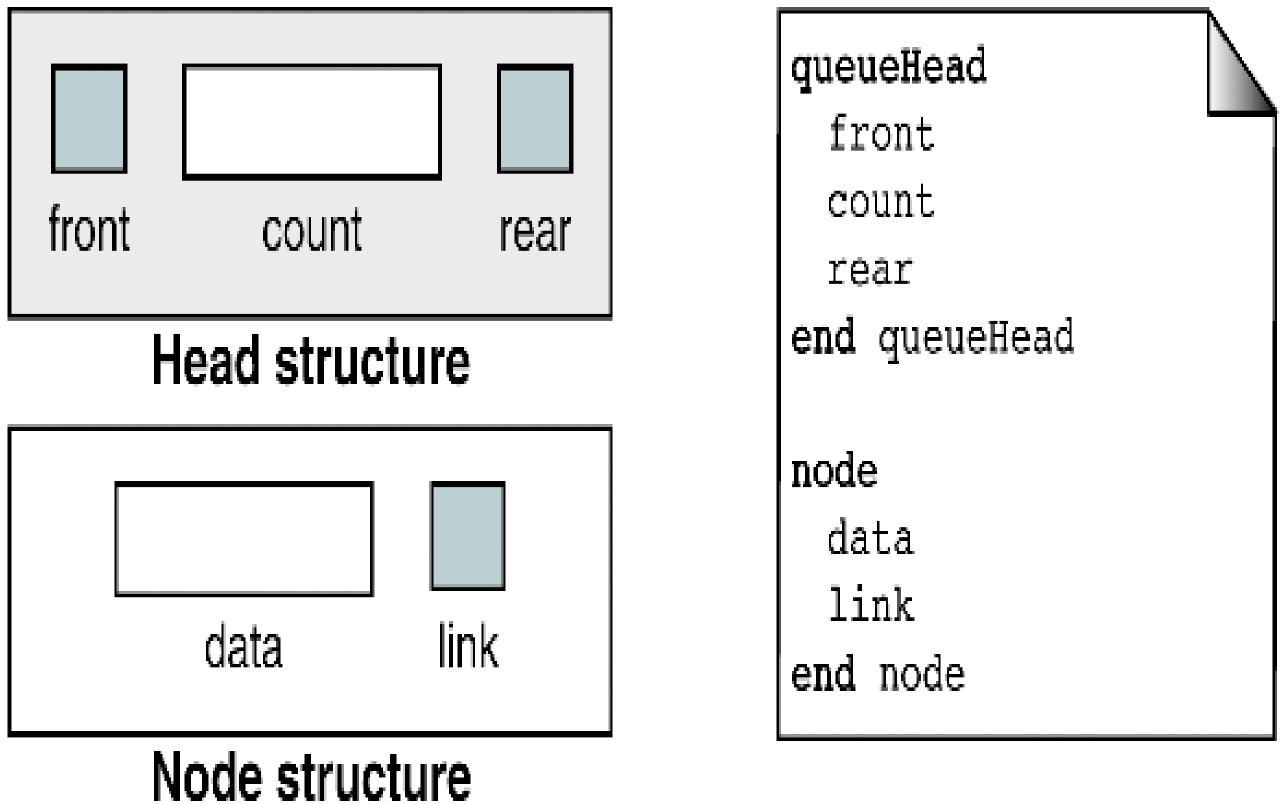


FIGURE 4-8 Queue Data Structure

FIGURE 4-9 Basic Queue Functions

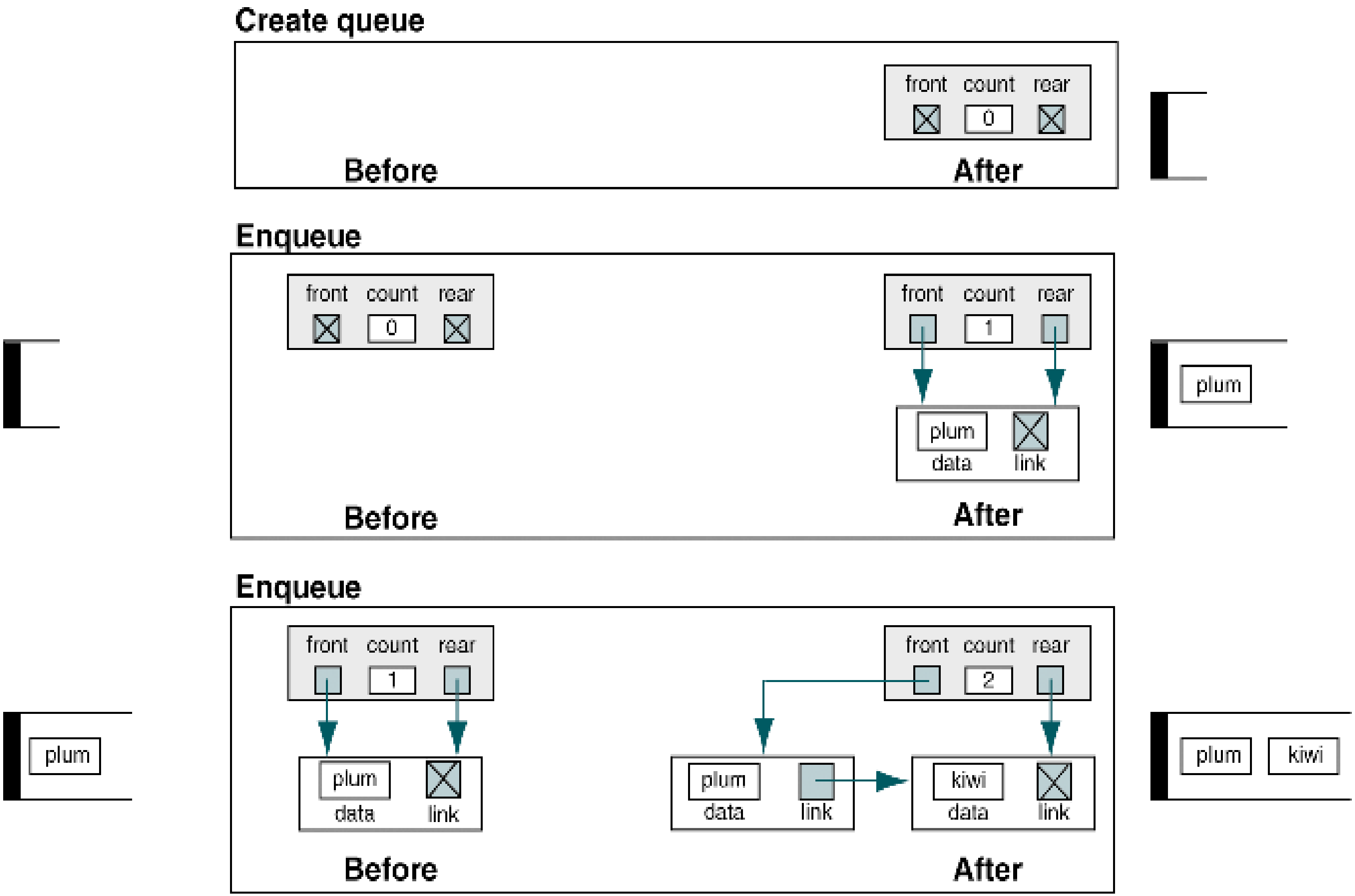
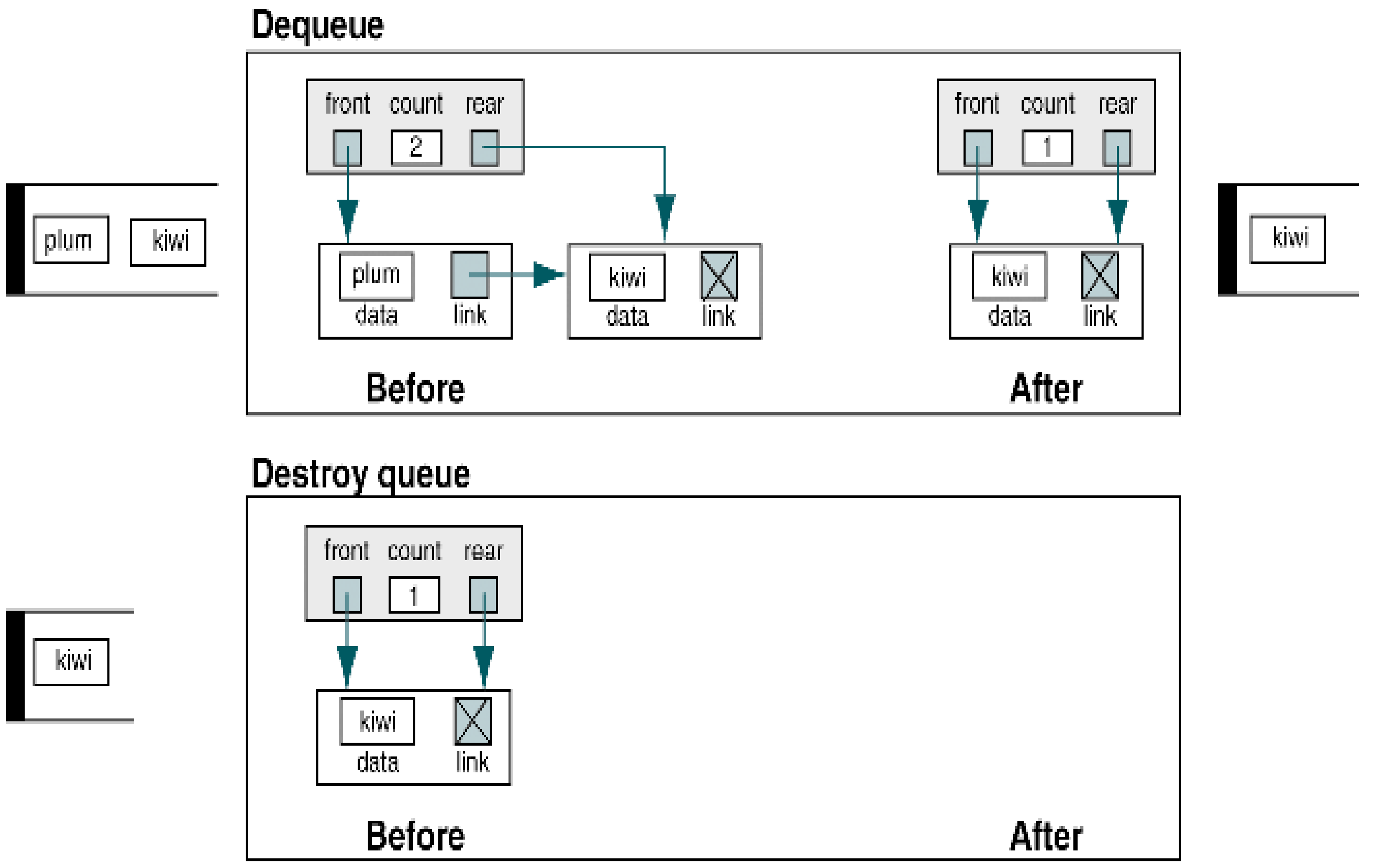


FIGURE 4-9 Basic Queue Functions (Continued)



PROGRAM 4-1 Queue ADT Data Structures

```
//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;

//Prototype Declarations
  QUEUE* createQueue  (void);
  QUEUE* destroyQueue (QUEUE* queue);

  bool  dequeue    (QUEUE* queue, void** itemPtr);
  bool  enqueue    (QUEUE* queue, void*  itemPtr);
  bool  queueFront (QUEUE* queue, void** itemPtr);
  bool  queueRear  (QUEUE* queue, void** itemPtr);
  int   queueCount (QUEUE* queue);

  bool  emptyQueue (QUEUE* queue);
  bool  fullQueue  (QUEUE* queue);
//End of Queue ADT Definitions
```

PROGRAM 4-2 Create Queue

```
/*================= 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 if successful; null if overflow
*/
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
```

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
*/
```

continued

PROGRAM 4-3 Enqueue *(continued)*

```
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
```

```
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

```
/*==================== queueFront ===================
  This algorithm retrieves data at front of the
  queue without changing the queue contents.
     Pre    queue is pointer to an initialized queue
     Post   itemPtr passed back to caller
     Return true if successful; false if underflow
*/
bool queueFront (QUEUE* queue, void** itemPtr)
{
//Statements
  if (!queue->count)
      return false;
  else
     {
      *itemPtr = queue->front->dataPtr;
       return true;
     } // else
} // queueFront
```

PROGRAM 4-6 Queue Rear

```
/*=================== queueRear ==================
  Retrieves data at the rear of the queue
  without changing the queue contents.
     Pre    queue is pointer to initialized queue
     Post   Data passed back to caller
     Return true if successful; false if underflow
*/
bool queueRear (QUEUE* queue, void** itemPtr)
{
//Statements
  if (!queue->count)
      return true;
```

continued

PROGRAM 4-6 Queue Rear *(continued)*

```
  else
     {
      *itemPtr = queue->rear->dataPtr;
      return false;
     } // else
} // queueRear
```

PROGRAM 4-8 Full Queue

```
/*=================== fullQueue ==================
  This algorithm checks to see if queue is full. It
  is full if memory cannot be allocated for next node.
     Pre    queue is a pointer to a queue head node
     Return true if full; false if room for a node
*/
bool fullQueue (QUEUE* queue)
{
```

PROGRAM 4-8 Full Queue *(continued)*

```
//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
```

PROGRAM 4-9 Queue Count

```
/*==================== queueCount ==================
  Returns the number of elements in the queue.
     Pre    queue is pointer to the queue head node
     Return queue count
*/
int queueCount(QUEUE* queue)
{
//Statements
  return queue->count;
} // queueCount
```

PROGRAM 4-10 Destroy Queue *(continued)*

```
     Post   All data have been deleted and recycled
     Return null pointer
*/
QUEUE* destroyQueue (QUEUE* queue)
{
//Local Definitions
  QUEUE_NODE* deletePtr;

//Statements
  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
```

Drawback of Linear Queue

- Once the queue is full, even though few elements from the front are deleted and some occupied space is relieved, it is not possible to add anymore new elements, as the rear has already reached the Queue's rear most position.

Circular Queue

- This queue is not linear but circular.

- Its structure can be like the following figure:

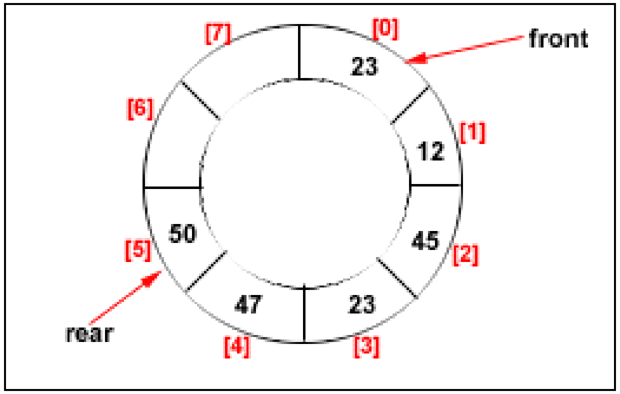


Figure: Circular Queue having Rear = 5 and Front = 0

Insert-Circular-Q(CQueue, Rear, Front, N, Item)

1. If Front = 0 and Rear = 0 then Set Front := 1 and go to step 4.
2. If Front =1 and Rear = N or Front = Rear + 1

   then Print: “Circular Queue Overflow” and Return.
3. If Rear = N then Set Rear := 1 and go to step 5.
4. Set Rear := Rear + 1
5. Set CQueue [Rear] := Item.
6. Return

Delete-Circular-Q(CQueue, Front, Rear, Item)

1. If Front = 0 then

   Print: “Circular Queue Underflow” and Return. /*..Delete without Insertion

2. Set Item := CQueue [Front]

3. If Front = N then Set Front = 1 and Return.

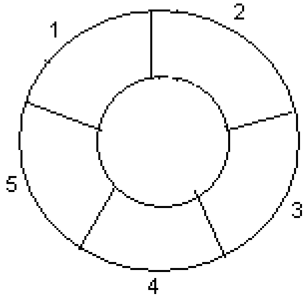
4. If Front = Rear then Set Front = 0 and Rear = 0 and Return.

5. Set Front := Front + 1

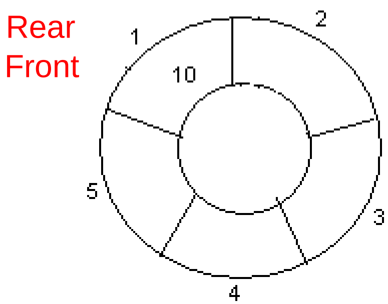
6. Return.

Example: Consider the following circular queue with N = 5.

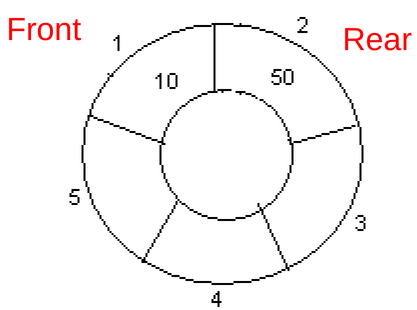
1. Initially, Rear = 0, Front = 0.



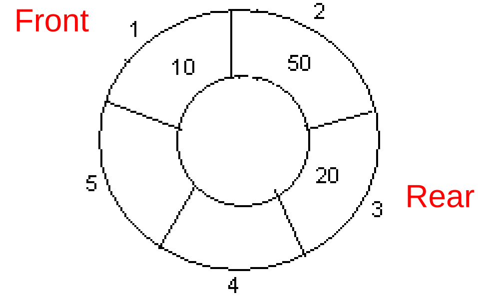
2. Insert 10, Rear = 1, Front = 1.



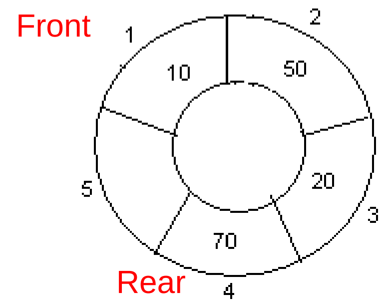
3. Insert 50, Rear = 2, Front = 1.



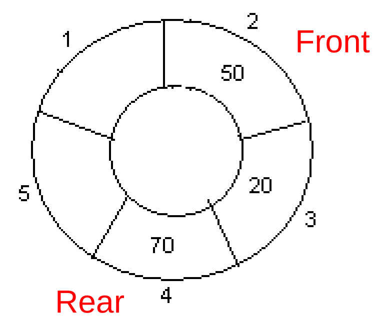
4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.



Priority Queue

• A priority queue is a collection of elements such that each element has been assigned a priority.

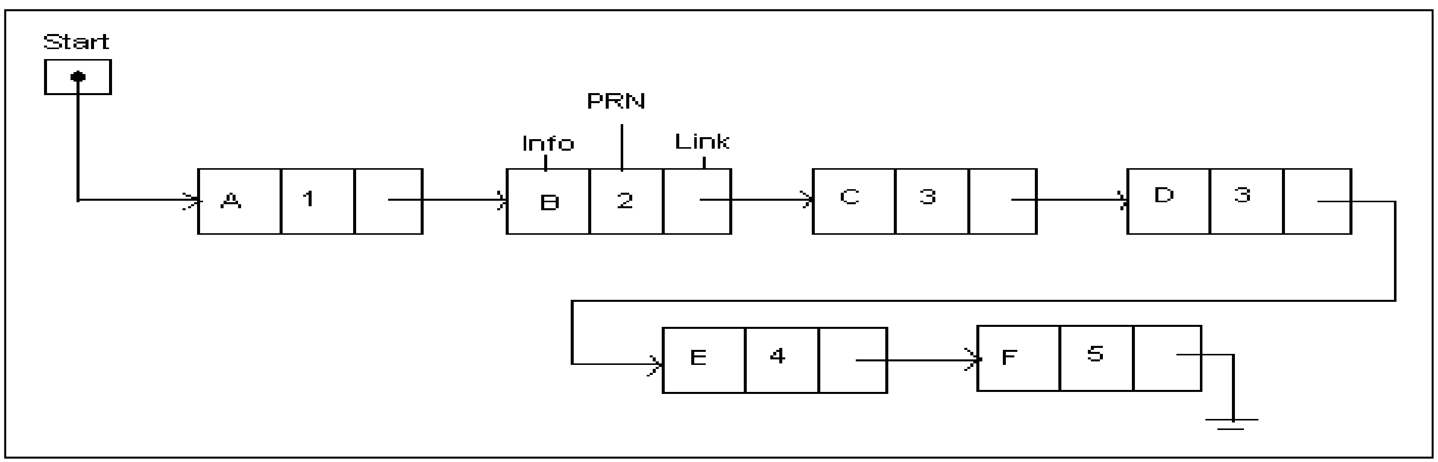
• A priority queue supports inserting new priorities, and removing the highest priority.

• Two elements with the same priority are processed according to the order in which they were added to the queue.

Representation of Priority Queue

• Each node in the list has three fields: an information field INFO, a priority number PRN and a link number Link.

• A node X precedes a node Y in the list when X has higher priority than Y or both have the same priority but X was added to the list before Y.



Priority Queue Implementation Time Complexity Comparison

Implementation	Insert	Extract-Min/Max	Find-Min/Max
Unsorted Array	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(1)	O(1)
Unsorted Linked List	O(1)	O(n)	O(n)
Sorted Linked List	O(n)	O(1))	O(1)

Deques

• A deque is a linear list in which data elements can be added or removed at either end but not in the middle.

• This type of queue is also known as dequeue and double-ended queue.

• There are two types of DEQUE.

Input-Restricted Deque – Allow insertions at only one end of the list but deletions at both ends of the list.

Output-Restricted Deque - Allow insertions at both ends but deletions at only one end of the list.

•**Example:**

0	1	2	3	4	5	6	7
60	20					100	80

Figure: Example of a Deque

Review Questions

- Suppose a circular queue is implemented using an array of size N = 10. Determine the number of elements in the queue for the following cases:
  - FRONT = 3, REAR = 7
  - FRONT = 8, REAR = 2
  - FRONT = 5, REAR = 6, then three elements are deleted.
  - If the queue is full and FRONT = 4, what will be the value of REAR?
- A priority queue is maintained as a sorted linked list. The queue initially contains the following elements: (lower priority values indicate higher priority)

Element	**Priority**
Alpha	2
Beta	5
Gamma	3
Delta	4

  - Insert (Epsilon, 1) and (Zeta, 6) into the queue.
  - Describe the new structure.
  - Delete the highest-priority element and describe the updated queue.
  - What if the priority queue was maintained as an unsorted linked list?

Review Questions

- A deque is implemented using an array of size N = 6. The current deque is represented as:

  FRONT = 1, REAR = 4

  ____, A, B, C, D, ____

  - Insert E at the rear and describe the new structure.
  - Insert Z at the front and describe the updated deque.
  - Delete two elements from the front.
  - What is the new FRONT index?
  - Explain the advantages of using a deque instead of a regular queue.

- Write an algorithm to **reverse a queue** using a stack.
- Compare **circular queues** with **regular queues** in terms of memory efficiency.