**Queue | Set 1 (Introduction and Array Implementation)**

Like [Stack](http://quiz.geeksforgeeks.org/stack-set-1/), [Queue](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)is a linear structure which follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO).  A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

**Operations on Queue:**  
Mainly the following four basic operations are performed on queue:

* **Enqueue:**Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
* **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
* **Front:**Get the front item from queue.
* **Rear:** Get the last item from queue.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2014/02/Queue.png)

**Applications of Queue:**  
Queue is used when things don’t have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property of Queue makes it also useful in following kind of scenarios.

1. When a resource is shared among multiple consumers. Examples include CPU Scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

See [this](http://introcs.cs.princeton.edu/43stack/)for more detailed applications of Queue and Stack.

**Array implementation Of Queue**

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner (See [this](http://www.cs.colostate.edu/~anderson/cs200/index.html/doku.php?id=recit:array_based_queue)for details)

[**Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.**](https://practice.geeksforgeeks.org/problems/implement-queue-using-array/1)

|  |
| --- |
| // C program for array implementation of queue  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>    // A structure to represent a queue  struct Queue  {      int front, rear, size;      unsigned capacity;      int\* array;  };    // function to create a queue of given capacity.  // It initializes size of queue as 0  struct Queue\* createQueue(unsigned capacity)  {      struct Queue\* queue = (struct Queue\*) malloc(sizeof(struct Queue));      queue->capacity = capacity;      queue->front = queue->size = 0;      queue->rear = capacity - 1;  // This is important, see the enqueue      queue->array = (int\*) malloc(queue->capacity \* sizeof(int));      return queue;  }    // Queue is full when size becomes equal to the capacity  int isFull(struct Queue\* queue)  {  return (queue->size == queue->capacity);  }    // Queue is empty when size is 0  int isEmpty(struct Queue\* queue)  {  return (queue->size == 0); }    // Function to add an item to the queue.  // It changes rear and size  void enqueue(struct Queue\* queue, int item)  {      if (isFull(queue))          return;      queue->rear = (queue->rear + 1)%queue->capacity;      queue->array[queue->rear] = item;      queue->size = queue->size + 1;      printf("%d enqueued to queue\n", item);  }    // Function to remove an item from queue.  // It changes front and size  int dequeue(struct Queue\* queue)  {      if (isEmpty(queue))          return INT\_MIN;      int item = queue->array[queue->front];      queue->front = (queue->front + 1)%queue->capacity;      queue->size = queue->size - 1;      return item;  }    // Function to get front of queue  int front(struct Queue\* queue)  {      if (isEmpty(queue))          return INT\_MIN;      return queue->array[queue->front];  }    // Function to get rear of queue  int rear(struct Queue\* queue)  {      if (isEmpty(queue))          return INT\_MIN;      return queue->array[queue->rear];  }    // Driver program to test above functions./  int main()  {      struct Queue\* queue = createQueue(1000);        enqueue(queue, 10);      enqueue(queue, 20);      enqueue(queue, 30);      enqueue(queue, 40);        printf("%d dequeued from queue\n\n", dequeue(queue));        printf("Front item is %d\n", front(queue));      printf("Rear item is %d\n", rear(queue));        return 0;  } |

Output:

10 enqueued to queue

20 enqueued to queue

30 enqueued to queue

40 enqueued to queue

10 dequeued from queue

Front item is 20

Rear item is 40

**Time Complexity:** Time complexity of all operations like enqueue(), dequeue(), isFull(), isEmpty(), front() and rear() is O(1). There is no loop in any of the operations.

Linked list implementation is easier, it is discussed here: [Queue | Set 2 (Linked List Implementation)](http://quiz.geeksforgeeks.org/queue-set-2-linked-list-implementation/)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Queue – Linked List Implementation**

In the [previous post](http://quiz.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/), we introduced Queue and discussed array implementation. In this post, linked list implementation is discussed. The following two main operations must be implemented efficiently.

In a [Queue data structure](https://www.geeksforgeeks.org/queue-data-structure/), we maintain two pointers, *front* and *rear*. The *front* points the first item of queue and *rear* points to last item.

1. **enQueue()** This operation adds a new node after *rear*and moves *rear* to the next node.
2. **deQueue()** This operation removes the front node and moves *front* to the next node.

[**Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.**](https://practice.geeksforgeeks.org/problems/implement-queue-using-linked-list/1)

|  |
| --- |
| // A C program to demonstrate linked list based implementation of queue  #include <stdio.h>  #include <stdlib.h>    // A linked list (LL) node to store a queue entry  struct QNode {      int key;      struct QNode\* next;  };    // The queue, front stores the front node of LL and rear stores the  // last node of LL  struct Queue {      struct QNode \*front, \*rear;  };    // A utility function to create a new linked list node.  struct QNode\* newNode(int k)  {      struct QNode\* temp = (struct QNode\*)malloc(sizeof(struct QNode));      temp->key = k;      temp->next = NULL;      return temp;  }    // A utility function to create an empty queue  struct Queue\* createQueue()  {      struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));      q->front = q->rear = NULL;      return q;  }    // The function to add a key k to q  void enQueue(struct Queue\* q, int k)  {      // Create a new LL node      struct QNode\* temp = newNode(k);        // If queue is empty, then new node is front and rear both      if (q->rear == NULL) {          q->front = q->rear = temp;          return;      }        // Add the new node at the end of queue and change rear      q->rear->next = temp;      q->rear = temp;  }    // Function to remove a key from given queue q  void deQueue(struct Queue\* q)  {      // If queue is empty, return NULL.      if (q->front == NULL)          return;        // Store previous front and move front one node ahead      struct QNode\* temp = q->front;        q->front = q->front->next;        // If front becomes NULL, then change rear also as NULL      if (q->front == NULL)          q->rear = NULL;        free(temp);  }    // Driver Program to test anove functions  int main()  {      struct Queue\* q = createQueue();      enQueue(q, 10);      enQueue(q, 20);      deQueue(q);      deQueue(q);      enQueue(q, 30);      enQueue(q, 40);      enQueue(q, 50);      deQueue(q);      printf("Queue Front : %d \n", q->front->key);      printf("Queue Rear : %d", q->rear->key);      return 0;  } |

**Output:**

Queue Front : 30

Queue Rear : 50

**Time Complexity:** Time complexity of both operations enqueue() and dequeue() is O(1) as we only change few pointers in both operations. There is no loop in any of the operations.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above