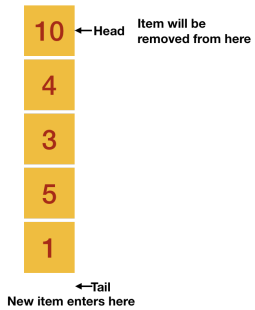
[redacted] (/)

Queue Data Structures

Similar to stacks, a queue is also an Abstract Data Type or ADT. A **queue** follows **FIFO (First-in, First out)** policy. It is equivalent to the queues in our general life. For example, a new person enters a queue at the last and the person who is at the front (who must have entered the queue at first) will be served first.

Similar to a queue of day to day life, in Computer Science also, a new element enters a queue at the last (tail of the queue) and removal of an element occurs from the front (head of the queue).



Similar to the stack, we will implement the queue using a linked list as well as with an array. But let's first discuss the operations which are done on a queue.

Enqueue → Enqueue is an operation which adds an element to the queue. As stated earlier, any new item enters at the tail of the queue, so Enqueue adds an item to the tail of a queue.

Dequeue → It is similar to the pop operation of stack i.e., it returns and deletes the front element from the queue.

isEmpty → It is used to check whether the queue has any element or not.

isFull → It is used to check whether the queue is full or not.

Front → It is similar to the top operation of a stack i.e., it returns the front element of the queue (but don't delete it).

Before moving forward to code up these operations, let's discuss the applications of a queue.

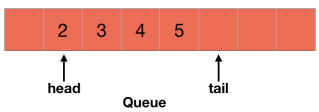
Applications of Queue

Queues are used in a lot of applications, few of them are:

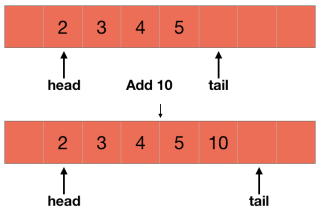
- Queue is used to implement many algorithms like Breadth First Search (BFS), etc.
- It can be also used by an operating system when it has to schedule jobs with equal priority
- Customers calling a call center are kept in queues when they wait for someone to pick up the calls

Queue Using an Array

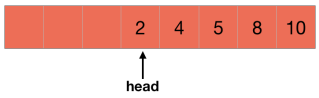
We will maintain two pointers - *tail* and *head* to represent a queue. *head* will always point to the oldest element which was added and *tail* will point where the new element is going to be added.



To insert any element, we add that element at *tail* and increase the *tail* by one to point to the next element of the array.



Suppose *tail* is at the last element of the queue and there are empty blocks before head as shown in the picture given below.

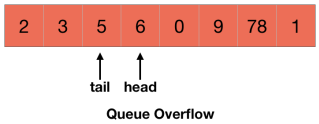


In this case, our *tail* will point to the first element of the array and will follow a circular order.

Initially, the queue will be empty i.e., both *head* and *tail* will point to the same location i.e., at index 1. We can easily check if a queue is empty or not by checking if *head* and *tail* are pointing to the same location or not at any time.


```
IS_EMPTY(Q)
     If Q.tail == Q.head
          return True
     return False
```

Similarly, we will say that if the *head* of a queue is 1 more than the *tail*, the queue is full.



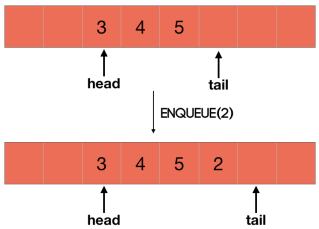

```
IS_FULL(Q)
     if Q.head = Q.tail+1
          return True
     Return False
```

Now, we have to deal with the enqueue and the dequeue operations.

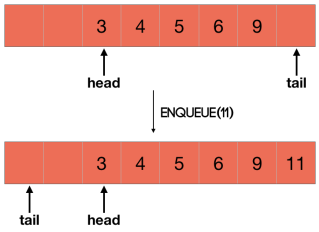
To enqueue any item to the queue, we will first check if the queue is full or not i.e.,

```
Enqueue(Q, x)
     if isFull(Q)
          Error “Queue Overflow”
     else
          …
```

If the queue is not full, we will add the element to the *tail* i.e, `Q[Q.tail] = x`.



While adding the element, it might be possible that we have added the element at the last of the array and in this case, the *tail* will go to the first element of the array.



Otherwise, we will just increase the *tail* by 1.

```
Enqueue(Q, x)
     if isFull(Q)
          Error “Queue Overflow”
     else
          Q[Q.tail] = x
          if Q.tail == Q.size
               Q.tail = 1
          else
               Q.tail = Q.tail+1
```

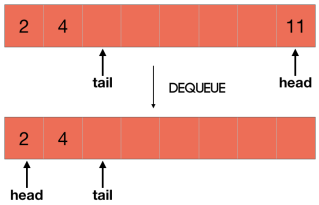
To dequeue, we will first check if the queue is empty or not. If the queue is empty, then we will throw an error.

```
Dequeue(Q, x)
     if isEmpty(Q)
          Error “Queue Underflow”
     else
          …
```

To dequeue, we will first store the item which we are going to delete from the queue in a variable because we will be returning it at last.

```
Dequeue(Q, x)
     if isEmpty(Q)
          Error “Queue Underflow”
     else
          x = Q[Q.head]
          …
```

Now, we just have to increase the head pointer by 1. And in the case when the head is at the last element of the array, it will go 1.



```
Dequeue(Q, x)
     if isEmpty(Q)
          Error “Queue Underflow”
     else
          x = Q[Q.head]
          if Q.head == Q.size
               Q.head = 1
          else
               Q.head = Q.head+1
          return x
```

C **Python** **Java**

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct queue {
  int head;
  int tail;
  int size;
  int Q[];
}queue;

queue* new_queue(int size) {
  queue *q = malloc(sizeof(queue) + size*sizeof(int));

  q->head = 1;
  q->tail = 1;
  q->size = size;

  return q;
}

int is_empty(queue *q) {
  if(q->tail == q->head)
    return 1;
  return 0;
}

int is_full(queue *q) {
  if(q->head == q->tail+1)
    return 1;
  return 0;
}

void enqueue(queue *q, int x) {
  if(is_full(q)) {
    printf("Queue Overflow\n");
  }
  else {
    q->Q[q->tail] = x;
    if(q->tail == q->size)
      q->tail = 1;
    else
      q->tail = q->tail+1;
  }
}

int dequeue(queue *q) {
  if(is_empty(q)) {
    printf("Underflow\n");
    return -1000;
  }
  else {
    int x = q->Q[q->head];
    if(q->head == q->size) {
      q->head = 1;
    }
    else {
      q->head = q->head+1;
    }
    return x;
  }
}

void display(queue *q) {
  int i;
  for(i=q->head; i<q->tail; i++) {
    printf("%d\n",q->Q[i]);
    if(i == q->size) {
      i = 0;
    }
  }
}

int main() {
  queue *q = new_queue(10);
  enqueue(q, 10);
  enqueue(q, 20);
  enqueue(q, 30);
```

```
    enqueue(q, 40);
    enqueue(q, 50);
    display(q);

    printf("\n");

    dequeue(q);
    dequeue(q);
    display(q);

    printf("\n");

    enqueue(q, 60);
    enqueue(q, 70);
    display(q);
    return 0;
}
```

We have covered all the major operations while implementing a queue using an array. Let's move ahead and implement these operations with a queue made from a linked list.

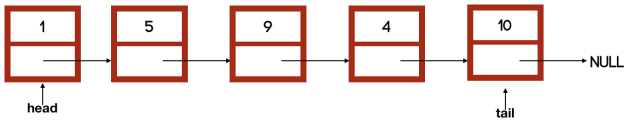
Queue Using Linked List

As we know that a linked list is a dynamic data structure and we can change the size of it whenever it is needed. So, we are not going to consider that there is a maximum size of the queue and thus the queue will never overflow. However, one can set a maximum size to restrict the linked list from growing more than that size.

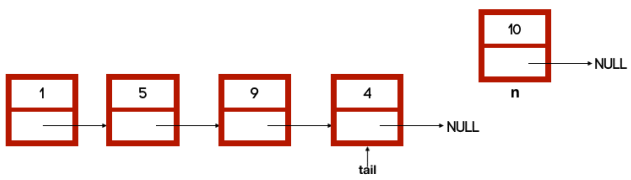
As told earlier, we are going to maintain a *head* and a *tail* pointer to the queue. In the case of an empty queue, *head* will point to `NULL`.


```
IS_EMPTY(Q)
     if Q.head == null
          return True
     return False
```

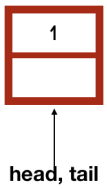
We will point the *head* pointer to the first element of the linked list and the *tail* pointer to the last element of it as shown in the picture given below.



The enqueue operation simply adds a new element to the last of a linked list.



However, if the queue is empty, we will simply make the new node *head* and *tail* of the queue.




```
ENQUEUE(Q, n)
     if IS_EMPTY(Q)
          Q.head = n
          Q.tail = n
     else
          Q.tail.next = n
          Q.tail = n
```

To dequeue, we need to remove the head of the linked list. To do so, we will first store its data in a variable because we will return it at last and then point *head* to its next element.

```
x = Q.head.data
Q.head = Q.head.next
return x
```

We will execute the above codes when the queue is not empty. If it is, we will throw the "Queue Underflow" error.

```
DEQUEUE(Q, n)
     if IS_EMPTY(Q)
          Error "Queue Underflow"
     else
          x = Q.head.data
          Q.head = Q.head.next
          return x
```

C Python Java

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
  int data;
  struct node *next;
}node;

typedef struct linked_list {
  struct node *head;
  struct node *tail;
}queue;

//to make new node
node* new_node(int data) {
  node *z;
  z = malloc(sizeof(struct node));
  z->data = data;
  z->next = NULL;

  return z;
}

//to make a new queue
queue* new_queue() {
  queue *q = malloc(sizeof(queue));
  q->head = NULL;
  q->tail = NULL;

  return q;
}

void traversal(queue *q) {
  node *temp = q->head; //temporary pointer to point to head

  while(temp != NULL) { //iterating over queue
    printf("%d\t", temp->data);
    temp = temp->next;
  }

  printf("\n");
}

int is_empty(queue *q) {
  if(q->head == NULL)
    return 1;
  return 0;
}

void enqueue(queue *q, node *n) {
  if(is_empty(q)) {
    q->head = n;
    q->tail = n;
  }
  else {
    q->tail->next = n;
    q->tail = n;
  }
}

int dequeue(queue *q) {
  if(is_empty(q)) {
    printf("Underflow\n");
    return -1000;
  }
  else {
    int x = q->head->data;
    node *temp = q->head;
    q->head = q->head->next;
    free(temp);
    return x;
  }
}

int main() {
  queue *q = new_queue();
```

```
    node *a, *b, *c;
    a = new_node(10);
    b = new_node(20);
    c = new_node(30);

    dequeue(q);
    enqueue(q, a);
    enqueue(q, b);
    enqueue(q, c);

    traversal(q);
    dequeue(q);
    traversal(q);


    return 0;
}
```

We used a singly linked list to make both stack and queue. We could have made the operations of both the data structures better by using doubly linked list because of the access of the previous node which would prevent us from iterating the entire list in many cases. You must try to make both of these data structures using doubly linked lists on your own.

So, you have studied about linked lists, stacks and queues. In the next chapters, you will learn about one more important data structure - trees.

> “ Everything must be made as simple as possible. But not simpler ”
>
> - Albert Einstein

PREV (/course/data-structures-stacks/) (/course/data-structures-trees/) NEXT

Further Readings

- Queue in C (/blog/article/queue-in-c/)
- Making a queue using linked list in C (/blog/article/making-a-queue-using-linked-list-in-c/)
- Making a queue using an array in C (/blog/article/making-a-queue-using-an-array-in-c/)