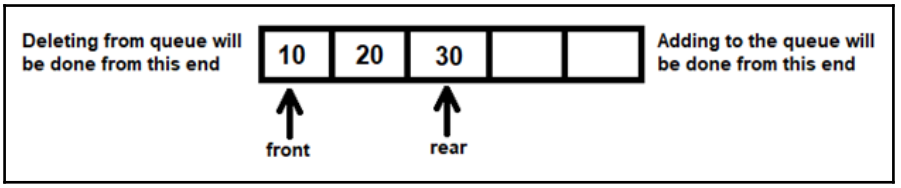
Appendix B

In this appendix, we will be covering some additional recipes related to advanced data structures, which we originally covered in `Chapter 11`, *Advanced Data Structures and Algorithms*:

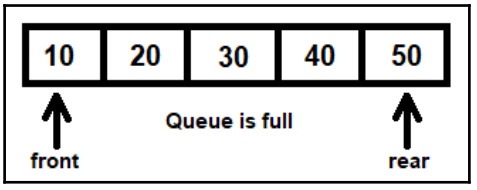
- Implementing queues using arrays
- Implementing circular queues using arrays
- Implementing dequeues using circular queues

Implementing queues using arrays

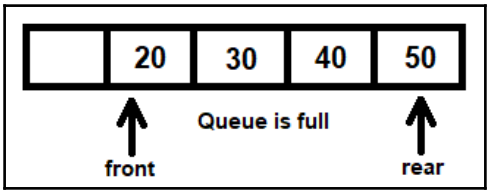
A queue is a linear structure where the addition of elements is done at one end and the removal of elements is done at the other. Basically, a queue has a **First In First Out** (**FIFO**) structure, that is, whatever element is added to the queue first is the first to be removed from it as well. To implement a queue, we need two pointers or variables called `front` and `rear` to keep track of the items that are inserted or deleted. Addition in a queue is done at the rear end, while deletion is done at the front end, as follows:



When the rear index reaches the maximum size of the queue, the queue is declared as full. As shown in the following image, you the value of the rear has become equal to the size of the queue; hence, no more elements can be accommodated in the queue and the queue is declared as full:



However, the biggest disadvantage with this linear queue system is that the queue is shown as *full* even if it has only one element. As shown in the following screenshot, an element is being deleted from the front and the queue has space to accommodate an element, but because the rear index is equal to the maximum size of the queue, the queue is declared as full. We can say that the queue does not use the storage space optimally:



In this recipe, we will learn how to implement a queue using an array. We will be making use of two variables, `front` and `rear`, where the front index will point to the location of the queue where an element can be deleted and the rear index will point at the location where the new element can be added to the queue.

How to do it...

Follow these steps to implement a queue using arrays:

1. Two variables, `front` and `rear`, are initialized to values of `-1`. When the values of the `front` and `rear` are `-1`, it means the queue is empty.
2. An integer array, `que`, is defined as a size 5 element. You can always increase the size of the queue by increasing the size of the `que` array if desired.
3. A menu is displayed to the user and the user is asked to press *1* to add an element to the queue, *2* to delete an element from the queue, and *3* to quit. If the user enters *1*, go to *step 4*; if the user enters *2*, go to *step 9*; if the user enters *3*, that means they want to terminate the program, so go to *step 13*.
4. We check whether the rear index has reached the maximum size of the queue. If it has, that means the queue is full. No element will be added to the queue. Jump to *step 3* to display the menu again.
5. If the rear is less than the queue size, then the user is prompted to enter the value to be added to the queue.
6. The value of `rear` is incremented by 1 and the value that's entered by the user is assigned to the `que[rear]` index location.

7. If the value of the `front` index is `-1`, that is, whether it is the first value being added to the queue, then the front index is also set to point at the first value that was added to the queue. This means that the value of the `rear` index is assigned to the front index.
8. Jump to *step 3* to display the menu again.
9. We check whether the `front` is equal to `-1`. If the `front` is equal to `-1`, that means the queue is empty, so no value is deleted from the queue and the control jumps to *step 3* to display the menu again.
10. Then, we check whether the front is equal to the rear, that is, whether the element to be deleted from the queue is the last element of the queue. If it is, the value at the `que[front]` location is removed and returned. To declare that the queue is empty, the values of the front and rear indices are set to `-1`.
11. If the values of the front and rear are not equal, the value at the `queue[front]` location is removed and the value of the front is incremented by 1 to make it point to the next element in the queue to be deleted.
12. Jump to *step 3* to display the menu again.
13. Terminate the program.

The program for implementing a queue using arrays is as follows:

```
//queuearray.c

#include <stdio.h>

#define max 5
int front = -1, rear = -1;
void addq(int Que[], int Val);
int delq(int Que[]);
int IsQueueEmpty();

int main() {
  int que[max];
  int val, n = 0;
  while (n != 3) {
    printf("\n1. Adding an element into the queue\n");
    printf("2. Deleting an element from the queue\n");
    printf("3. Quit\n");
    printf("Enter your choice 1/2/3: ");
    scanf("%d", & n);
    switch (n) {
    case 1:
```

```
        if (rear >= max - 1) printf("Sorry the queue is full\n");
        else {
          printf("Enter the value to add to queue: ");
          scanf("%d", & val);
          addq(que, val);
          printf("Value %d is added to queue\n", val);
        }
        break;
      case 2:
        if (!IsQueueEmpty()) {
          val = delq(que);
          printf("Value removed from queue %d\n", val);
        } else printf("Sorry, the queue is empty\n");
        break;
      }
    }
    return 0;
}

void addq(int Que[], int Val) {
    rear++;
    Que[rear] = Val;
    if (front == -1)
      front = rear;
}

int IsQueueEmpty() {
    if (front == -1)
      return (1);
    else
      return (0);
}

int delq(int Que[]) {
    int val;
    if (front == rear) {
      val = Que[front];
      front = -1;
      rear = -1;
    } else {
      val = Que[front];
      front++;
    }
    return (val);
}
```

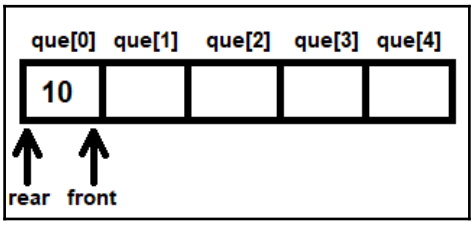
How it works...

Let's assume that the user doesn't enter more than 5 elements; hence, a macro is defined that's 5 in size. An integer array, `que`, is defined as the `max` size. You can always increase the size of `max` if required. Two index pointers, `front` and `rear`, are defined and initialized to `-1`. The values of the front and rear index locations are set to `-1` to indicate that the queue is empty.

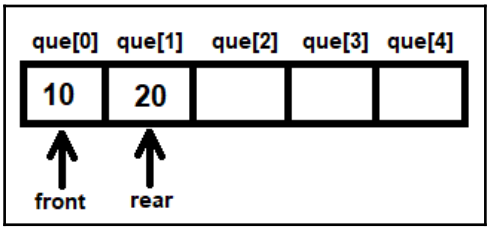
This is a menu-driven program; that is, three options will be displayed on the screen: **1** to add elements to the queue, **2** to delete elements from the queue, and **3** to quit. Assuming the user presses *1* to add elements to the queue, it is ensured that the value of the `rear` index is less than `max-1`, that is, `rear` is less than the maximum size of the queue. If the value of `rear` is greater than or equal to `max-1`, it means the queue is full, so a value won't be added and the menu will be displayed again.

If the value of `rear` is found to be less than max-1, the user will be prompted to enter the value to be added to the queue. Let's assume that the user enters the value `10` so that it will be added to the queue. The value `10` is assigned to the `val` variable. The `addq` function will be invoked and the array queue and the `val` variable are passed to it as arguments.

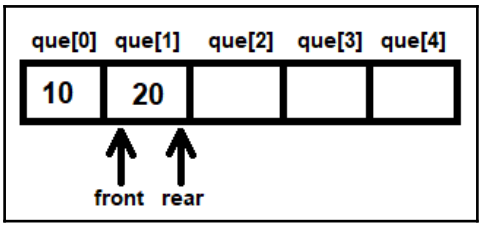
In the `addq` function, the value of the rear index is incremented by 1, making its value `0`. Regarding the `que[rear]` location, that is, the `que[0]` location, the value in the `val` variable, `10`, is assigned. Also, because the value of the `front` variable is `-1`, the value of `rear`, that is, `0`, is assigned to the `front` index too. Whenever the first value is added to the queue, both the rear and front indices are set to point at it, as shown here:



After adding the value to the `que` array, the `addq` function ends and control goes back to the `main` function. In the `main` function, a confirmation message is displayed, informing the user that the value was successfully added to the queue. The menu is then displayed again, showing the two options. Assuming the user wants to add one more element, the value *1* is entered as a menu choice. Again, we check whether the value of `rear` is less than the maximum size of the queue. If there is enough space, the user is prompted to enter another value that will be added to the queue. Suppose the user enters the value `20`; it will be assigned to the `val` variable. Again, the `addq` function is invoked and the `que` array and `val` variable are passed to it as arguments. In the `addq` function, the value of the `rear` index location is incremented by 1, making its value `1`. Regarding the `que[1]` index location, the value `20` is assigned to it, as follows:

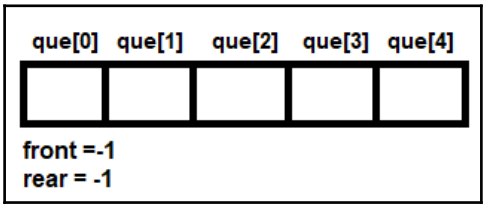


The `addq` function moves over and control jumps back to the `main` function. Again, the menu is displayed. Suppose this time the user enters the value *2* as a menu choice in order to delete an element from the queue. First, the `IsQueueEmpty` function is invoked to check whether the queue has some elements to be deleted in it or whether it's already empty. If the value of `front` is `-1`, this means the queue is empty. However, because the value of front is 0, the `delq` function is invoked and the `que` array is passed to it as an argument. In the `delq` function, we check whether the value to be deleted from the queue is the last value or not. If the front and rear pointers are pointing at the same element in the queue array, this means it is the last element in the array. However, because the values of `front` and `rear` are different, the value at the `que[front]` or `que[0]` index location, that is, `10`, is accessed and assigned to the `val` variable. After this, the value of the front index is incremented to `1`:



The value `10`, which is accessed from the queue, is returned to the `main` function and is displayed on the screen.

After that, the menu is displayed once more, asking the user to enter the desired option. Let's assume that the user enters the value *2* as a menu choice in order to delete one more element from the queue. Again, the `IsQueueEmpty` function is invoked to confirm that the queue isn't empty. Once it is ensured that the queue has elements in it, the `delq` function is called and the `que` array is passed to it. In the `delq` function, we check whether the values of the front and rear indices are the same. The values of `front` and `rear` are the same, that is, `1`, which means there is only one element in the queue. So, the value at the `que[1]` location is accessed, that is, value `20` is accessed and assigned to the `val` variable and the values of the front and rear indices are set to `-1` to declare that the queue is now empty:



The value `20` in the `val` variable is returned to the `main` function, where it is displayed on the screen. The menu is displayed on the screen once more, asking the user to enter valid options. Although we know the queue is empty, let's see what happens if the user still enters the value *2* as a menu option. The `IsQueueEmpty` function will be invoked. In the `IsQueueEmpty` function, we check whether the value of the front index is equal to `-1`. Because the value of `front` is equal to `-1`, the `IsQueueEmpty` function returns value `1` to the `main` function. When `IsQueueEmpty` returns the value `1` to the `main` function, an `else` block will be executed displaying the message `Sorry, the queue is empty`. Then, the menu will be displayed, asking the user to enter valid options. Assuming that the user wants to quit the program, they will enter the value *3* as the menu choice. On entering the value *3*, the program terminates.

The program is compiled using GCC with the following statement:

```
D:\CAdvBook>GCC queuearray.c – queuearray
```

Because no error appears upon compilation, this means the `queuearray.c` program has successfully compiled into the `queuearray.exe` file. On executing the file, you will be shown options for adding as well as deleting an element from the queue. Because a queue is a FIFO structure, you will notice that the value that was added to the queue first is the first one to be removed from the queue. Here, we get the output shown in the following screenshot:

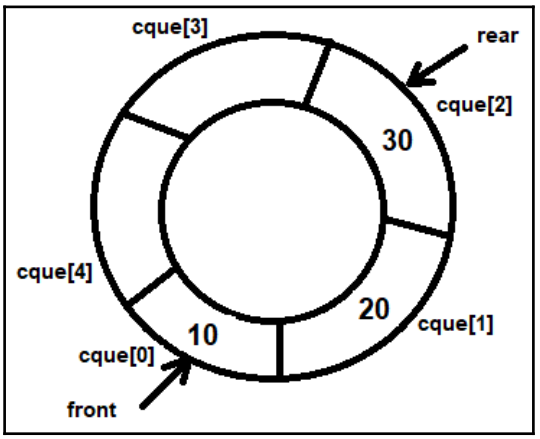
```
D:\CAdvBook>queuearray

1. Adding an element into the queue
2. Deleting an element from the queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to queue: 10
Value 10 is added to queue

1. Adding an element into the queue
2. Deleting an element from the queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to queue: 20
Value 20 is added to queue

1. Adding an element into the queue
2. Deleting an element from the queue
3. Quit
Enter your choice 1/2/3: 2
Value removed from queue 10

1. Adding an element into the queue
2. Deleting an element from the queue
3. Quit
Enter your choice 1/2/3: 2
Value removed from queue 20

1. Adding an element into the queue
2. Deleting an element from the queue
3. Quit
Enter your choice 1/2/3: 2
Sorry, the queue is empty

1. Adding an element into the queue
2. Deleting an element from the queue
3. Quit
Enter your choice 1/2/3: 3

D:\CAdvBook>
```

The queue structure has a disadvantage associated with it. Even if a few of the values are deleted from the queue and there is enough space at the front of the queue to accommodate more elements, no element can be added to the queue if the rear index is pointing at the last location of the queue. Hence, you can say that, storage-wise, the queue is quite an inefficient structure.

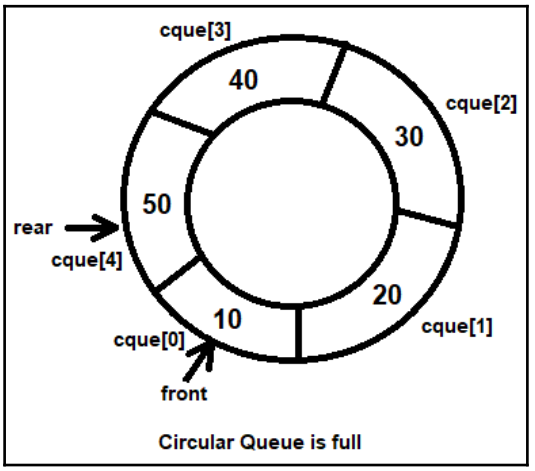
Implementing circular queues using arrays

A circular queue is a linear structure where the element that is added first is the one that will be removed first, that is, using the FIFO principle. To give it a circular shape, the rear end of the queue is connected to the front end of the queue. The benefit of a circular queue is that it uses memory quite efficiently. If the elements are placed until the rear end and there is a vacant location at the front end, the new value can be placed at the front end too.

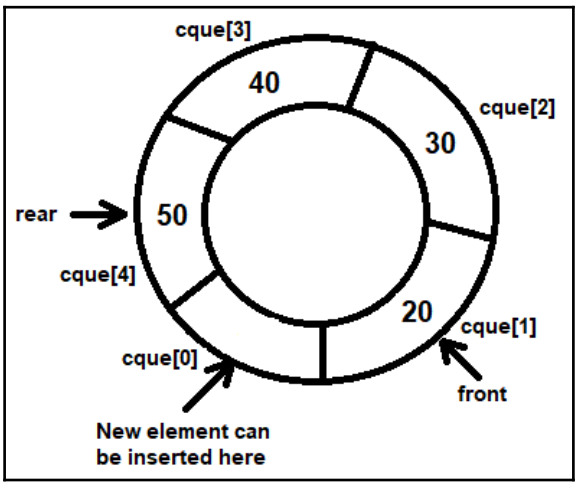
Here, the queue is assumed to be a circular array. When the first element is added, both the front and rear pointers will point at the same element. Afterward, the new value is added at the rear index location and the element is deleted from the front index location:



When the value of the rear index becomes equal to the size of the circular queue or if any value is deleted from the front of the queue, it resets to the value `0` to fill in the empty space in front of the circular queue, if any. The circular queue is considered full when the rear reaches the back of the front index, as follows:



To make the rear index revolve around the array, its value should become `0` after reaching the maximum length of the circular queue. This is done with the help of the `%` (mod) operator. Considering the maximum size of a circular queue is 5 elements, we use the formula *(r+1)%5*. By using this formula, when the rear (*r*) index reaches `4` and a new element is about to be added, it will become `0` (because *(4+1) % 5 = 0*):



In this recipe, we will learn how to implement a circular queue. A circular queue has the advantage that the rear index, after reaching the maximum size of the queue, can reset to the front of the queue. The benefit of doing so is that if any element is deleted from the front of the circular queue, that empty location can be used to accommodate new elements, making the best utilization of the storage space of the queue.

How to do it...

Follow these steps to learn how to implement circular arrays using queues:

1. Two variables, `front` and `rear`, are initialized to `-1`. Initially, the circular queue is empty and when the values of the front and rear indices are `-1`, this indicates that the circular queue is empty.
2. An integer array, `cque`, is defined as a size of 5 elements. If you want to enter more than 5 elements in the circular queue, you can always increase the size of the `cque` array.
3. A menu is displayed where the user is asked to enter options *1*, *2*, or *3*, where option *1* is for adding an element to the circular queue, option *2* is for deleting an element from the circular queue, and option *3* is for quitting the program. If the user enters *1*, go to *step 4*; if the user enters *2*, go to *step 11*; if the user enters *3*, that means they want to terminate the program, so go to *step 16*.
4. Ask the user to enter the value to be added to the circular queue.
5. Then, we check whether the rear index has reached the maximum size of the circular queue. If it has, then follow these steps:
    1. Apply the mod operator and make the value of `rear` equal to 0.
    2. If the value of `rear` is less than `front`, then add the new value at the `cque[rear]` index location.
    3. If the value of `rear` is not less than `front`, that means the circular queue is full. In that case, no new element will be added to the circular queue. Jump to *step 3* to display the menu again.
6. If the value of `rear` is not less than the maximum size of the circular queue, then increment the value of `rear` by one.
7. If the values of `rear` and `front` are not the same, add the new element at the `cque[rear]` index location.

8. If the values of `rear` and `front` are the same, decrement the value of the rear index by one and display a message stating that the circular queue is full. No value will be added to the circular queue. Go to *step 3* to display the menu.
9. If the value of `front` is `-1`, that is, if this is the first element being added to the circular queue, the front index location is also set to point at the first element that was added to the circular queue. This means the value of the rear index location is assigned to the front index.
10. Jump to *step 3* to display the menu again.
11. We check whether the front is equal to `-1`. If so, that means the circular queue is empty, so no value is deleted from the circular queue and control jumps to *step 3* to display the menu again.
12. Then, we check whether `front` is equal to `rear`, that is, whether the element to be deleted from the circular queue is the last element of the circular queue. If it is, the value at the `cque[front]` location is removed and returned. To declare that the circular queue is empty, the values of the front and rear indices are set to `-1`.
13. If the value of the front is greater than the size of the circular queue, then follow these steps:

    1. Apply the mod operator and make the value of the front index equal to `0`.
    2. If the value of `front` is less than the rear index, then remove the value from the circular queue from the `cque[front]` index location and increment the value of the front index by one so that it points at the next element to be deleted.
    3. If the value of `front` is not less than the rear index, check whether the value of `front` is equal to `rear`. If it is, this means the element to be deleted is the last element of the circular queue, so remove the value from the circular queue at the `cque[front]` location. By doing this, the circular queue becomes empty. To indicate that circular queue is empty, set the values of the front and rear indices to `-1`.

14. If the value of `front` isn't larger than the circular queue, remove the element at the `cque[font]` index location and increment the value of `front` by 1 to make the front index point at the next element to be deleted from the circular queue.

15. Jump to *step 3* to display the menu again.
16. Terminate the program.

The program for implementing a circular queue using arrays is as follows:

```
//circularquearray.c

#include<stdio.h>

#define max 5
int cque[max];

void addq(int * Front, int * r, int h);
int delq(int * Front, int * Rear);
int isQueueEmpty(int * Front);

int main() {
  int rear, front, n = 0, fromq, val;
  rear = front = -1;
  while (n != 3) {
    printf("\n1. Adding an element into the circular queue\n");
    printf("2. Deleting an element from the circular queue\n");
    printf("3. Quit\n");
    printf("Enter your choice 1/2/3: ");
    scanf("%d", & n);
    switch (n) {
    case 1:
      printf("Enter the value to add to the circular queue: ");
      scanf("%d", & val);
      addq( & front, & rear, val);
      break;
    case 2:
      if (!isQueueEmpty( & front)) {
        fromq = delq( & front, & rear);
        printf("The value removed from the circular queue is %d\n",
fromq);
      } else
        printf("The circular queue is empty\n");
      break;
    }
  }
  return 0;
}

void addq(int * Front, int * Rear, int Val) {
  int oldR;
  if ( * Rear >= max - 1) {
    oldR = * Rear;
```

```
      * Rear = ( * Rear + 1) % max;
      if ( * Rear < * Front) {
        cque[ * Rear] = Val;
        printf("Value %d is added in circular queue\n", Val);
      } else {
        * Rear = oldR;
        printf("Sorry the circular queue is full\n ");
      }
    } else {
      ( * Rear) ++;
      if ( * Rear != * Front) {
        cque[ * Rear] = Val;
        printf("Value %d is added in circular queue\n", Val);
      } else {
        ( * Rear) --;
        printf("Sorry, the circular queue is full\n");
      }
    }
    if ( * Front == -1)
      *
      Front = * Rear;
}

int isQueueEmpty(int * Front) {
    if ( * Front == -1)
      return (1);
    else
      return (0);
}

int delq(int * Front, int * Rear) {
    int val = 0;
    if ( * Front == * Rear) {
      val = cque[ * Front];
      * Front = * Rear = -1;
    } else {
      if ( * Front >= max) {
        * Front = * Front % max;
        if ( * Front < * Rear) {
          val = cque[ * Front];
          ( * Front) ++;
        } else {
          if ( * Front == * Rear) {
            val = cque[ * Front];* Front = * Rear = -1;
          }
        }
      } else {
        val = cque[ * Front];
```

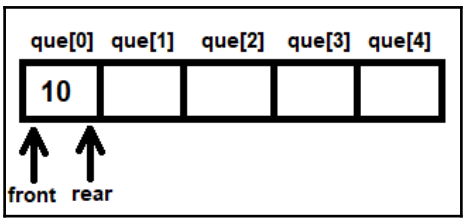
```
            ( * Front) ++;
          }
        }
        return (val);
    }
```

How it works...

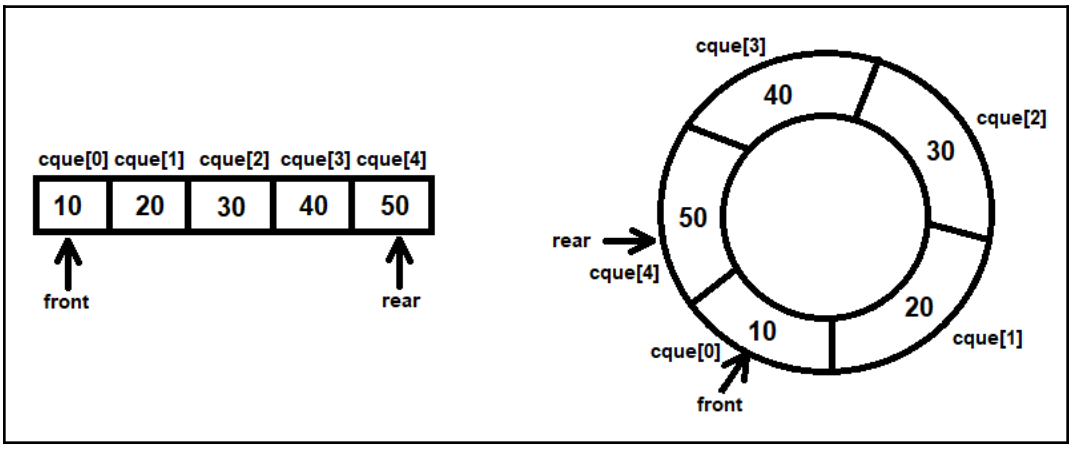
The values of the rear and front index locations are initially set to `-1`. When the circular queue is empty, it is mandatory to set the values of `rear` and `front` to `-1`. Assuming the maximum size of the circular queue is 5, a macro, `max`, is defined with the value `5`. An integer array, `cque`, is defined with the size `max`. If you think the user can enter more than 5 elements in the circular queue, you can always increase the value of max to any desired value.

A menu is displayed showing three options: **1** for adding an element to the circular queue, **2** for deleting an element from the circular queue, and **3** for quitting the program. Let's assume that the user enters the value 1 to add elements to the circular queue. On entering the value 1 as the menu choice, the user will be prompted to enter the value to be added to the circular queue. Assuming the value that's entered by the user is 10, it will be assigned to the `val` variable. The `addq` function is invoked and the addresses of `front`, `rear`, and `val` are passed to it as arguments. In the `addq` function, we check whether the value of rear is greater than or equal to max-1, that is, 4.

Because the value of rear is currently `-1`, an `else` block will be executed. Within the `else` block, the value of `rear` is incremented to `0`. It is ensured that the value of `rear` has not become equal to the front. If the value of `rear` becomes equal to the value of `front`, this means there is no space in the circular queue and it is full. Because the value of rear is `0` and the front is `-1`, the value that's entered by the user, `10`, is assigned to the `cque[0]` index location. Whenever the first value is added to the circular queue, both the front and rear indices are set to point at it, so the value of `front` is also set to `0` to point at the first value that was added to the circular queue:



After adding a value to the circular queue, the `addq` function terminates and the menu will be displayed. The user can enter value 1 as a menu choice repetitively to add as many elements to the circular queue as desired. Let's assume the user has added four more values to the circular queue: `20`, `30`, `40`, and `50`. The circular queue with five elements in it might appear as follows:



At the moment, the value of `rear` is `4`, while the value of `front` is `0`. If, in this situation, the user enters the value `1` as a menu choice to add one more element to the circular queue, they will be prompted to enter the value to be added to the circular queue. Suppose the user enters value `60`, which will be assigned to the `val` variable. The value 60 will not be added to the circular queue.

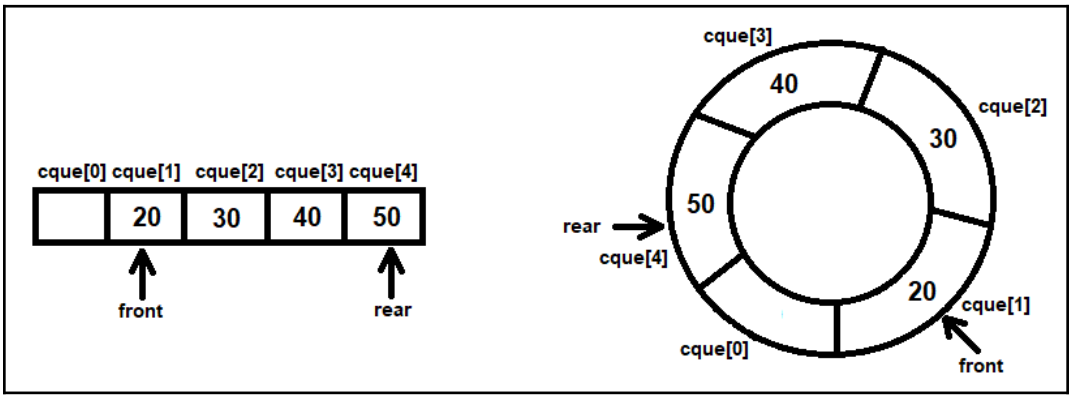
The `addq` function is invoked and the address of `front`, the address of `rear`, and `val` are passed to it as parameters. In the `addq` function, the program will check whether the value of `rear` is greater than or equal to max-1. The value of rear is `4`, so its value is greater than or equal to 4; consequently, an `if` block will be executed. The value of `rear`, `4`, is temporarily saved into a variable called `oldR`. The formula *rear+1 % 5* is applied to get the next `rear` value. The value of `rear` will become `0`. The value of `front` is already 0, which means the `cque[0]` location already has an element, so the new value cannot be accommodated in the circular queue. The old value of rear, `4`, is retrieved from the temporary variable, `oldR`, and a message stating `Sorry, the circular queue is full` is displayed on the screen. The `addq` function will move over and control will go back to the `main` function where, again, the menu will be displayed.

Deleting an element from a circular queue

Let's assume that the user enters the value `2` as a menu choice to delete an element from the circular queue.

First of all, the `isQueEmpty` function will be invoked and the front index will be passed to it by reference. `isQueueEmpty` checks the value of the front index to determine whether the circular queue is empty or not; that is, if the value of the front index is equal to `-1`, `isQueueEmpty` returns `1`; otherwise, it returns `0`. Currently, the value of the front index is `0`, so the `isQueueEmpty` function returns `0`, hence proving that the circular queue is not empty.

Then, the `delq` function will be invoked and the addresses of `front` and `rear` will be passed to it. In the `delq` function, the front and rear values are compared. The program will check whether the values of `front` and `rear` are the same or whether the value of `front` is greater than or equal to `max`. Currently, the value of `front` is less than `rear`, so the `else` block will be executed and the element at `cque[0]` will be extracted and assigned to the `val` variable; that is, `10` will be accessed from the circular queue and assigned to `val`. Thereafter, the value of `front` is incremented to `1`:

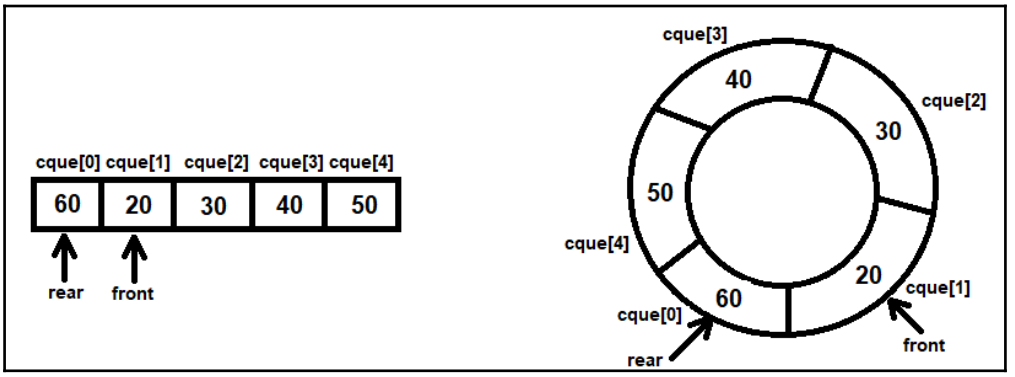


The `delq` function returns the value in the `val` variable to the `main` function and the value is assigned to the `fromq` variable; hence, it's displayed on the screen.

Again, the menu is displayed, asking the user to enter the desired option. Because a value has been deleted from the queue, let's assume that the user enters the value `1` as a menu choice to add an element to the circular queue. On entering the value `1`, the user will be prompted to enter the value to be added to the circular queue.

Let's assume that the user enters the value `60`, which will be assigned to the `val` variable. The `addq` function is invoked and `front`, `rear`, and `val` will be passed to it as arguments.

In the `addq` function, the program checks whether the value of `rear` is greater than or equal to `max-1`. Currently, the value of `rear` is `4`, so the condition is true and an `if` block will be executed. The value of `rear`, that is, `4`, is assigned to a temporary variable, `oldR`. The formula *(rear+1)%5* is applied to make the value of `rear` equal to `0`. That is why it is called a circular queue: because the value of `rear` resets to 0 once it reaches the maximum size of the circular queue. The program checks whether the value of the rear index is less than the front index. If so, the value `60` that was entered by the user is assigned to the `cque[0]` location, as follows:



This confirms that the circular queue is efficient when it comes to storing information. The location that was freed up by deleting an element can be reused for storing new information. Once `addq` moves over, control goes back to the `main` function, and the menu will be displayed once more. The user might press *3* to quit. On entering `3`, the program will terminate.

The program is compiled with GCC using the following statement:

```
D:\CAdvBook>gcc circularquearray.c -o circularquearray
```

Because no error appears on compilation, this means the `circularquearray.c` program has successfully compiled into the `circularquearray.exe` file. Let's run the executable file and try adding and deleting some elements from the circular queue. If we do this, we'll get the following output:

```
D:\CAdvBook>circularquearray

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 10
Value 10 is added in circular queue

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 20
Value 20 is added in circular queue

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 30
Value 30 is added in circular queue
```

Let's add a few more values to the circular queue:

```
1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 40
Value 40 is added in circular queue

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 50
Value 50 is added in circular queue

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 60
Sorry the circular queue is full
```

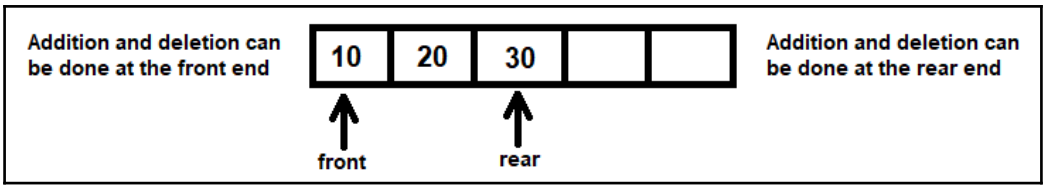
The circular queue is full, so no more values can be added to it. Let's delete a value from it:

```
1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 2
The value removed from the circular queue is 10

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 1
Enter the value to add to the circular queue: 60
Value 60 is added in circular queue

1. Adding an element into the circular queue
2. Deleting an element from the circular queue
3. Quit
Enter your choice 1/2/3: 3
```

As you can see, the circular queue is more efficient than a linear queue as far as storage is concerned. Even if the rear index has reached the maximum size of the circular queue, if any of the elements are deleted from the front in a circular queue, the rear index will reset to the beginning of the circular queue to accommodate the new element.

Implementing dequeues using circular queues

A dequeue, also known as a double-ended queue, is a structure similar to a queue but after removing all queue constraints. The queue has a constraint stating that addition can only be done at one end, while deletion will be done from the other. A dequeue is different from a queue in the sense that you can add or delete elements from both the ends, that is, from the front as well as the rear end, as follows:



In this recipe, we will learn how to implement a dequeue using a circular queue.

How to do it...

Follow these steps to implement a dequeue using circular queues:

1. Two variables, `front` and `rear`, are initialized to `-1`. Initially, the dequeue is empty and when the values of the front and rear indices are `-1`, this indicates that the dequeue is empty.
2. An integer array, `dq`, is defined with a size of 5 elements. If you want to enter more than 5 elements to the dequeue, you can always increase the size of the `dq` array.
3. A menu is displayed where the user is shown five options to choose from: **1** to add an element to the front of the dequeue, **2** to add an element to the rear, **3** to delete an element from the front, **4** to delete an element from the rear, and **5** to quit the program. If the user enters *1* as a menu choice, go to *step 4*; if the user enters *2*, go to *step 10*; if the user enters *3*, go to *step 17*; if the user enters *4*, go to *step 21*; and if the user enters *5*, that means they want to terminate the program, so go to *step 24*.
4. Ask the user to enter the value to be added to the front of the dequeue.
5. The program checks whether the value of `front` is `-1`; if it is, that means it is the first element that is being added to the dequeue. So, the value of the front and rear index locations are set to `0` to make both of them point to the first value being added to the dequeue. The new value is added to the `dq[front]` index location.
6. If the value of `front` is not `-1`, the program whether the value of `front` is equal to `0`. If it is, the value of `front` is set equal to max-1, where max is the maximum size of the dequeue.

7. If the value of `front` is not equal to 0, the value of the front index is decremented by 1.
8. The value to be added to the dequeue is assigned to the `dq[front]` index location.
9. Jump to *step 3* to display the menu.
10. Ask the user to enter the value to be added to the rear of the dequeue.
11. Then, the program checks whether the rear index has reached the maximum size of the dequeue. If it has, then follow these steps:
    1. Apply the mod operator and make the value of `rear` equal to 0.
    2. If the value of `rear` is less than the front, then add the new value to the `dq[rear]` index location.
    3. If the value of `rear` is not less than `front`, that means the dequeue is full. In that case, no new element will be added to the dequeue. Jump to *step 3* to display the menu again.
12. If the value of `rear` is not less than the maximum size of the dequeue, then increment the value of `rear` by one.
13. If the values of `rear` and `front` are not the same, add the new element to the `dq[rear]` index location.
14. If the values of `rear` and `front` are the same, decrement the value of the rear index by one and display a message stating that the dequeue is full. A value won't be added to the dequeue. Go to *step 3* to display the menu.
15. If the value of `front` is `-1`, that is, if this is the first element being added to the dequeue, the front index location is also set to point at the first element that was added to the dequeue. This means the value of the rear index location is assigned to the front index.
16. Jump to *step 3* to display the menu again.
17. The program checks whether `front` is equal to `rear`, that is, whether the element to be deleted from the dequeue is the last element of the dequeue. If it is, the value at the `dq[front]` location is removed and returned. To declare that the dequeue is empty, the values of the front and rear indices are set to `-1`.

18. If the value of `front` is greater than the size of the dequeue, then follow these steps:
    1. Apply the `mod` operator and make the value of the front index equal to `0`.
    2. If the value of `front` is less than the rear index, then remove the value in the dequeue from the `dq[front]` index location and increment the value of the front index by 1 so that it points to the next element to be deleted.
    3. If the value of `front` is not less than the rear index, check whether the value of `front` is equal to `rear`. If it is, this means the element to be deleted is the last element of the dequeue, so remove the value from the dequeue at the `dq[front]` location. Now, the dequeue will be empty. To indicate that the dequeue is empty, set the values of the front and rear indices to `-1`.
19. If the value of `front` is not bigger than the size of the dequeue, remove the element at the `dq[font]` index location and increment the value of `front` by 1 to make the front index point at the next element to be deleted from the dequeue.
20. Jump to *step 3* to display the menu again.
21. The program checks whether `front` is equal to `rear`. If it is, that means it is the last element of the dequeue; in this case, the value at the `dq[front]` index location is removed. To indicate that the dequeue is empty, the values of the front and rear indices are set to `-1`.
22. If the values of `front` and `rear` are not the same, the program checks whether the value of `rear` is equal to `0`. If it is, the element at the `dq[rear]` index location is removed and the value of `rear` is set equal to `max-1`, where the value of max is equal to the maximum size of the dequeue.
23. If the value of `rear` is not equal to `0`, the element at the `dq[rear]` index location is removed from the dequeue and the value of rear is decremented by 1.
24. Terminate the program.

The program for implementing a dequeue using a circular queue is as follows:

```
//dequearray.c

#include <stdio.h>

#define max 5
int dq[max];
```

```
void addInFront(int * Front, int * r, int h);
void addInRear(int * Front, int * r, int h);
int delFFront(int * Front, int * Rear);
int delFRear(int * Front, int * Rear);
int isDQueueEmpty(int * Front);
int isDQueueFull(int * Front, int * Rear);

int main() {
  int rear, front, n = 0, fromdq, val;
  rear = -1;
  front = -1;
  while (n != 5) {
    printf("\n1. Adding element in front into the dequeue\n");
    printf("2. Adding element at rear into the dequeue\n");
    printf("3. Deleting an element from the front in dequeue\n");
    printf("4. Deleting an element from the rear in dequeue\n");
    printf("5. Quit\n");
    printf("Enter your choice 1/2/3/4/5: ");
    scanf("%d", & n);
    switch (n) {
    case 1:
      printf("Enter the value to be added to the front in dequeue: ");
      scanf("%d", & val);
      addInFront( & front, & rear, val);
      break;
    case 2:
      printf("Enter the value to be added at the rear in dequeue: ");
      scanf("%d", & val);
      addInRear( & front, & rear, val);
      break;
    case 3:
      if (!isDQueueEmpty( & front)) {
        fromdq = delFFront( & front, & rear);
        printf("The value removed from the front in dequeue is %d\n",
fromdq);
      } else printf("The dequeue is empty\n");
      break;
    case 4:
      if (!isDQueueEmpty( & front)) {
        fromdq = delFRear( & front, & rear);
        printf("The value removed from the rear in dequeue is %d\n",
fromdq);
      } else printf("The dequeue is empty\n");
      break;
    }
  }
  return 0;
}
```

```
int isDQueueEmpty(int * Front) {
  if ( * Front == -1)
    return (1);
  else
    return (0);
}

int isDQueueFull(int * Front, int * Rear) {
  if (( * Front == 0) && ( * Rear == max - 1))
    return (1);
  if ( * Front == ( * Rear + 1))
    return (1);
  return (0);
}

void addInRear(int * Front, int * Rear, int Val) {
  int oldR;
  if ( * Rear == max - 1) {
    oldR = * Rear;
    * Rear = ( * Rear + 1) % max;
    if ( * Rear < * Front) {
      dq[ * Rear] = Val;
      printf("Value %d is added in dequeue\n", Val);
    } else {
      * Rear = oldR;
      printf("Sorry the dequeue is full\n ");
    }
  } else {
    ( * Rear) ++;
    if ( * Rear != * Front) {
      dq[ * Rear] = Val;
      printf("Value %d is added in dequeue\n", Val);
    } else {
      ( * Rear) --;
      printf("Sorry, the dequeue is full\n");
    }
  }
  if ( * Front == -1)
    *
    Front = * Rear;
}

void addInFront(int * Front, int * Rear, int Val) {
  if ( * Front == -1) {
    * Front = 0;
    * Rear = 0;
  } else {
    if ( * Front == 0)
```

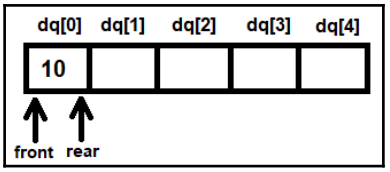
```
        *
        Front = max - 1;
      else
        *Front = ( * Front) - 1;
    }
    dq[ * Front] = Val;
  }

  int delFFront(int * Front, int * Rear) {
    int val = 0;
    if ( * Front == * Rear) {
      val = dq[ * Front];
      * Front = * Rear = -1;
    } else {
      if ( * Front >= max) {
        * Front = * Front % max;
        if ( * Front < * Rear) {
          val = dq[ * Front];
          ( * Front) ++;
        } else {
          if ( * Front == * Rear) {
            val = dq[ * Front];* Front = * Rear = -1;
          }
        }
      } else {
        val = dq[ * Front];
        ( * Front) ++;
      }
    }
    return (val);
  }

  int delFRear(int * Front, int * Rear) {
    int val = 0;
    if ( * Front == * Rear) {
      val = dq[ * Front];
      * Front = * Rear = -1;
    } else {
      if ( * Rear == 0) {
        val = dq[ * Rear];
        * Rear = max - 1;
      } else {
        val = dq[ * Rear];
        * Rear = * Rear - 1;
      }
    }
    return (val);
  }
```

How it works...

The initial conditions of the dequeue are set, that is, the values of the rear and front index locations are initially set to `-1`. Initially, the dequeue is empty and it is mandatory to set the values of `rear` and `front` to `-1` for an empty dequeue. Assuming the maximum size of the dequeue is 5, a macro, `max`, is defined with the value 5. An integer array, `dq`, is defined as the max size. You can always initialize the `max` macro to any larger value if you think the user might want to enter more values in the dequeue.

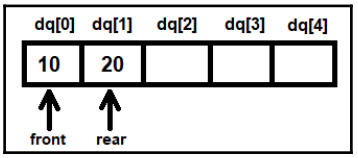
A menu is displayed, showing five options: 1 for adding elements to the front of the dequeue, 2 for adding an element to the rear of the dequeue, 3 for deleting an element from the front of the dequeue, 4 for deleting an element from the rear of the dequeue, and 5 to quit the program. Let's assume that the user enters the value `1` to add elements to the front of the dequeue. On entering the value `1`, the user will be prompted to enter the value to be added to the dequeue.

Assuming that the user enters the value `10`, it will be assigned to the `val` variable. Then, a function, `addInFront`, is invoked and the addresses of the front and rear indices, along with the entered value, `val`, are passed to it. In the `addInFront` functions, the program checks whether the value of `front` is equal to `-1`. The dequeue is currently empty, so the value of front is `-1`, so an `if` block is executed and the values of the front and rear indices are initialized to `0`. This is because whenever the first value is added to the dequeue, both the front and rear indices are set to point at it. Because the value will be added at the 0^{th} index location, the values of the front and rear indices are set to `0`. At the `dq[0]` index location, the value 10 will be assigned, as follows:

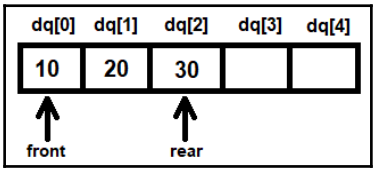


After adding a value to the dequeue, the `addInFront` function moves over and the menu will be displayed. Let's assume that the user enters the value `2` as a menu choice to add the next element to the rear of the dequeue. On entering `2` as the menu choice, the user will be prompted to enter a number to be added to the rear in the dequeue. Assuming the user wants to add the value `20`, it will be assigned to the `val` variable. The `addInRear` function will be invoked and the addresses of the front and rear indices, along with the entered value, `val`, that is, `20`, are passed to it.

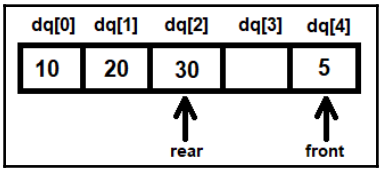
In the `addInRear` function, the program checks whether the value of `rear` is equal to max-1, that is, `4`. The value of `rear` at the moment is `0`, so the `else` block will be executed. In the `else` block, the value of `rear` is incremented by `1`, making it `1`. Then, the program ensures that the values of `rear` and `front` are not equal. If the values of `rear` and `front` are equal, this means the dequeue is full. At the moment, the value of `rear` is `1` and the value of `front` is `0`, so the value that's supplied by the user is assigned to the `dq[1]` index location, as follows:



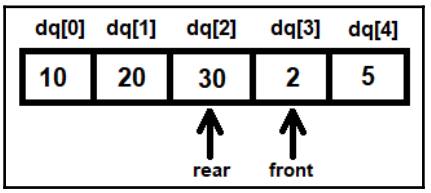
Let's assume the user enters `2` as a menu choice to add one more element to the dequeue at the rear. If the value that's entered by the user is `30`, it will be added to the dequeue at the `dq[2]` index location, as follows:



Again, the menu will be displayed, and this time, the user enters the value `1` to add an element to the front of the dequeue. On entering `1`, the user will be prompted to enter the value to be added to the front of the queue. Let's assume that the user renters the value `5`. The `addInFront` function will be invoked and the values of `front` and `rear` and the value `5` will be passed to it. In the `addInFront` function, the program checks whether the value of `front` is equal to `-1`. The value of `front` is `0` currently, so an `else` block will be executed. Within the `else` block, the program checks whether the value of `front` is `0`. If the value of `front` is `0`, the value of `front` is set equal to `max-1`, that is, equal to `4`. Hence, the value `5` will be assigned at the `dq[4]` index location, as follows:



Similarly, if the user enters the value `2` as a menu choice to add one more element to the front of the dequeue, that value will be added at the `dq[3]` index location. Assuming the value that the user wants to add to dequeue is `2`, the `dq` array will finally appear as follows:

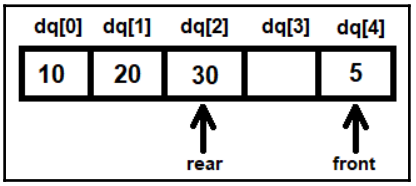


Now, if the user tries to add more values, they won't be able to, and they will get the following message:

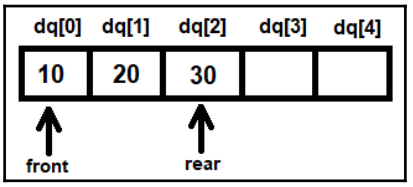
```
Sorry the dequeue is full
```

On displaying the menu, let's assume that the user enters the value `3` as a menu choice to delete an element from the front of the dequeue.

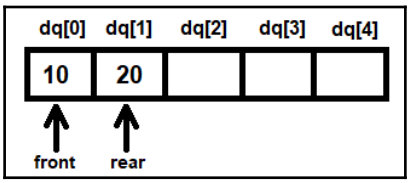
On entering the value `3`, the `isDQueueEmpty` function is invoked and the address of the front index is passed to it. Because the value of `front` is `3` and not `-1`, the `delFFront` function will be invoked and the addresses of the front and rear index locations will be passed to it. In the `delFFront` function, the program checks whether the front and rear index values are the same, that is, whether there is only one element left in the dequeue. However, because the values of `front` and `rear` are `3` and `2`, respectively, they are quite different, so an `else` block will be executed. Within the `else` block, the program checks whether the value of `front` is greater than or equal to max. The value of `front` isn't greater than or equal to 5, so again, an `else` block will be executed and the value at the `dq[3]` index location, that is, the value `2`, will be accessed from the dequeue and the value of `front` will be incremented to `4`. The value `2`, which is returned by the function, is displayed on the screen:



The menu will be displayed again. The value of `front` is `4` now, so if the user enters the value `3` again to delete the value from the front of the dequeue, the value at the `dq[4]` index location, that is, `5`, will be accessed and the value of `front` will be incremented to `5`. Now, by applying the `%5` operator, the value of the front index will be set to `0`:



Again, the menu will be displayed. Let's assume that the user enters the value `4` as a menu choice to delete a value from the rear of the dequeue. The `delFRear` function will be invoked. In the `delFRear` function, the program checks whether the values of `rear` and `front` are the same, that is, whether it is the last element in the dequeue. It isn't; due to this, the values of `front` and `rear` are `5` and `2`, respectively. Then, the function checks whether the value of `rear` is equal to `0`. It isn't, so again, an `else` block will be executed and the value at the `dq[2]` index location, that is, `30`, will be accessed and the value of `rear` will be decremented to `1`:



The value `30` is returned to the `main` function. On displaying the menu, the user might press *5* to quit. On entering *5*, the program terminates.

The program is compiled using GCC with the following statement:

```
D:\CAdvBook>gcc dequeuearray.c -o dequeuearray
```

Because no error appears upon compilation, that means the `dequearray.c` program has successfully compiled into the `dequearray.exe` file. Let's run the executable file to see how the dequeue works. You will get a variety of menu options so that you can add an element to the front of the dequeue, add an element to the rear of the dequeue, delete an element from the front of the dequeue, and delete an element from the rear of the dequeue. You can select the desired option to take the respective action. By doing this, we get the following output:

```
D:\CAdvBook>dequeuearray

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 1
Enter the value to be added to the front in dequeue: 10

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 2
Enter the value to be added at the rear in dequeue: 20
Value 20 is added in dequeue

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 2
Enter the value to be added at the rear in dequeue: 30
Value 30 is added in dequeue
```

We will keep getting the menu repetitively. Let's add a few more elements to the dequeue at both ends:

```
1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 1
Enter the value to be added to the front in dequeue: 5

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 1
Enter the value to be added to the front in dequeue: 2

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 2
Enter the value to be added at the rear in dequeue: 40
Sorry, the dequeue is full
```

Now, let's see how removing elements from the dequeue takes place:

```
1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 3
The value removed from the front in dequeue is 2

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 3
The value removed from the front in dequeue is 5

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 4
The value removed from the rear in dequeue is 30
```

Let's remove a few more elements from the dequeue:

```
1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 4
The value removed from the rear in dequeue is 20

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 3
The value removed from the front in dequeue is 10

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 3
The dequeue is empty

1. Adding element in front into the dequeue
2. Adding element at rear into the dequeue
3. Deleting an element from the front in dequeue
4. Deleting an element from the rear in dequeue
5. Quit
Enter your choice 1/2/3/4/5: 5
```

As you can see, the dequeue overpowers the constraint of a queue, that is, only adding at the rear end and only deleting from the front.