Fundamentals of Programming 2

Queues and Their Applications

Arkadiusz Chrobot

Department of Computer Science

March 16, 2020

Outline

1. Queues and Their Classification
2. The FIFO Queue
   - Implementing As Dynamically Allocated Data Structure
   - Implementing With the Use of Arrays
3. Testing some of the operations on Dynamically Allocated Data Structures
4. Summary

Queues and Their Classification

The FIFO Queue

Queues, just like the stack, are abstract data structures consisting of linked together elements that store data. However, the elements are managed differently than in the stack. The term *queue* usually is interpreted as a FIFO *queue* and so it is going to be used for the most part of the lecture. The FIFO stands for *First In First Out*. The rule implies that elements are added to the queue at one of its ends and removed on the other. The end where the elements are removed is called the *head* or *front* of the queue, and the end where the elements are added is called a *tail* or *rear* of the queue.

Queues and Their Classification

Double-ended Queue

Aside from the FIFO queues there also exist *double-ended queues* or *deques* for which the operations of adding and removing of an element are defined for both ends. Among them the following types are distinguished:

- *an input-restricted deque* — the elements can be removed at both ends, but added only at one,
- *an output-restricted deque* — the elements can be added at both ends, but removed only at one.

The FIFO Queue

The rest of the lecture is about FIFO queues. Those queues can be implemented as dynamically allocated data structures or with the use of an array. Both possibilities are presented in the lecture. In the last part of the lecture a simple way of testing functions that perform some of the operations on dynamically allocated data structures is introduced. Implementations of queues are described starting with the dynamically allocated data structures. All of them store only `int` numbers.

The FIFO Queue

Like in the case of the stack or any other abstract data structure, the definitions of the base type and functions that implement the basic operations are necessary for implementing a queue. At least two operations need to be implemented: adding an element to the queue and removing an element from the queue. They are called *enqueue* and *dequeue* respectively. To simplify their implementation two special pointers are used. One points to the current first element of the queue and it is called a HEAD and the second one points to the current last element of the queue and it is called a TAIL. The first one is used when an element is added to a queue and the second one when an element is removed from the queue. Some programmers call them a FRONT and a REAR respectively.

Implementing As Dynamically Allocated Data Structure

The program that presents the implementation of the FIFO queue uses functions that manage the heap and display messages on the screen. That's why it includes the `stdio.h` and `stdlib.h` header files.

Implementing As Dynamically Allocated Data Structure

Header Files

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

Implementing As Dynamically Allocated Data Structure

The Base Data Type of The FIFO Queue

The base data type for the FIFO queue is based on a structure and its definition is the same as the definition of the stack base type, with the exception of name. It can be modified to suite the needs of a programmer, but it has to contain at least one pointer that allows for liking an element of the queue with another such an element. The definition of the queue base type is presented in the next slide.

Implementing As Dynamically Allocated Data Structure

The Base Type of FIFO Queue

```
1  struct fifo_node
2  {
3      int data;
4      struct fifo_node *next;
5  };
```

Implementing As Dynamically Allocated Data Structure

The HEAD and TAIL Pointers

As it has been mentioned before, to make the implementation of a queue effective, two pointers are needed. One of them should point to the current first element of the queue and the second one to the current last element of the queue. Those pointers can be declared as either global or local variables. However, in the presented program they are declared as fields of a separated structure. The next slide contains a definition of a type of the structure and a declaration of a global variable of this type. The pointers are global therefore they default value is zero (**NULL**) and thus the queue is initially empty.

Implementing As Dynamically Allocated Data Structure

The Structure With Pointers

```
1  struct fifo_pointers
2  {
3      struct fifo_node *head, *tail;
4  } fifo;
```

Implementing As Dynamically Allocated Data Structure

The *Enqueue* Operation

Now, that the structure with pointers and the base type of FIFO queue is defined, the functions that perform operations on the queue can be also defined, starting with the one that enqueues a new element. It has to satisfy the following assertions:

- If the queue exists (has at least one element), the function adds a new element at the back of it, and if the queue doesn't exist (is empty), the function creates and adds its first element.
- If the function fails to create a new element, then the queue stays the same as it was.
- If the operation of adding a new element is successful then the queue grows by one element or if it was not existing, it is created.

The next slide contains a definition of a function that implements the *enqueue* operation.

Implementing As Dynamically Allocated Data Structure

The enqueue() Function

```
1  void enqueue(struct fifo_pointers *fifo, int data)
2  {
3      struct fifo_node *new_node =
4              (struct fifo_node *)malloc(sizeof(struct fifo_node));
5      if(new_node) {
6          new_node->data = data;
7          new_node->next = NULL;
8          if(fifo->head==NULL)
9              fifo->head = fifo->tail = new_node;
10         else {
11             fifo->tail->next=new_node;
12             fifo->tail=new_node;
13         }
14     } else
15         fprintf(stderr,"No new element has been created!\n");
16 }
```

Implementing As Dynamically Allocated Data Structure

The `enqueue()` Function

The *enqueue* operation is implemented in the program in a form of a function of the same name. The function doesn't return any value. If it fails to create and add a new element to the queue it only prints a message on the screen. The state of the queue stays the same. If the queue stays empty the behaviour of other functions that perform operations on it is unaffected. They all check if the queue is not empty before they perform any operation on it. The structure with the `head` and `tail` pointers is passed to the `enqueue()` function by a pointer parameter. The values of those pointers can be modified by the function and the modifications have to be preserved when the function terminates, thus the use of the pointer parameter is necessary. The second parameter of the function is used for passing the value which is to be stored in the new element of the queue.

Implementing As Dynamically Allocated Data Structure

The `enqueue()` Function

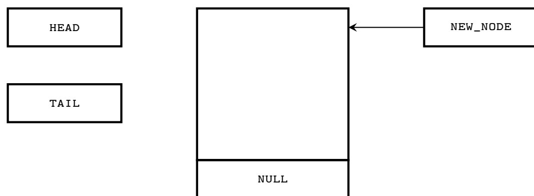
In the lines of the function no. 3 and no. 4, a memory area is allocated for the queue new element. After the function makes sure that the allocation was successful (line no. 5) it initializes the fields of the element. The `NULL` value is assigned to the `next` field of the element, to indicate that the element will be the last one in the queue. There are two cases that have to be taken into consideration when implementing the part of the `enqueue()` function that adds a new element to the tail of the queue:

1. the element is added at the end of an existing queue,
2. the element is added to an empty (nonexistent) queue.

They are distinguished in the line no. 8 of the function. If both queue pointers have the value of `NULL` then the second case applies and both pointers are assigned the address of a new element, which becomes the first and the last element of the queue. It is illustrated by an animation in the next slide.

Implementing As Dynamically Allocated Data Structure

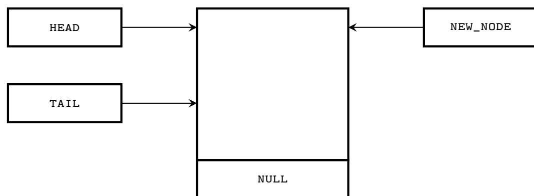
The `enqueue()` Function — Creating a Queue



The queue before the line no. 9 of the `enqueue()` function is performed

Implementing As Dynamically Allocated Data Structure

The `enqueue()` Function — Creating a Queue



The queue after the line no. 9 of the `enqueue()` function is performed

Implementing As Dynamically Allocated Data Structure

The `enqueue()` Function

Adding an element to the existing queue is implemented differently. In the line no. 11 the `enqueue()` function uses the `tail` pointer to reach the current last element of the queue and to store in its `next` field the address of the new element. That's how the new element becomes the last one in the queue. Before exiting, the function has to ensure the correct state of the queue, or more precisely, that the `tail` pointer is still pointing to the last element of the queue. Therefore, in the line no. 12 the function assigns the address of the new element to the pointer. Please note, that the lines no. 11 and no. 12 are related, and cannot switch their places in the function. On the other hand, the line no. 12 could be replaced by the `fifo->tail=fifo->tail->next;` statement, but then the function would be less legible. However, similar expressions will be used in the future lectures, when necessary. The message from the line no. 15 is displayed only if creating the new element fails. In that case the queue stays as it was, before the function started.

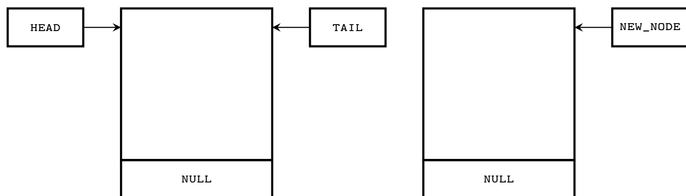
Implementing As Dynamically Allocated Data Structure

The `enqueue()` Function

The next slides presents an animation that illustrates how a new element is added to a queue consisting of a single element. The element would be added in the same way, if the queue contained more than one element.

Implementing As Dynamically Allocated Data Structure

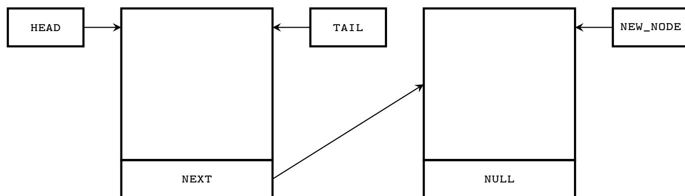
The `enqueue()` Function — Adding a New Element



Before the line no. 11 of the `enqueue()` function is performed

Implementing As Dynamically Allocated Data Structure

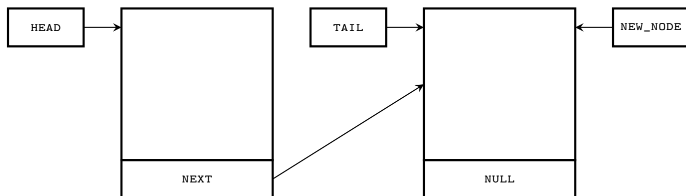
The `enqueue()` Function — Adding a New Element



After the line no. 11 of the `enqueue()` function is performed

Implementing As Dynamically Allocated Data Structure

The `enqueue()` Function — Adding a New Element



After the line no. 12 of the `enqueue()` function is performed

Implementing As Dynamically Allocated Data Structure

The *Dequeue* Operation

The *dequeue* operation removes an element from the front (the beginning) of the FIFO queue. The operation should satisfy the following assertions:

- If the queue doesn't exist then the state of its pointers should not change after the operation is performed — both pointers have to have the value of `NULL`.
- If an element is removed from a queue that has only one element, then after the operation is performed both queue pointers must have the value of `NULL`.
- If an element is removed from a queue consisting of more than one element, then after the operation is successfully completed the queue is reduced by one element and the pointers correctly point to the head and tail of the queue.

An implementation of the operation in a form of a function is presented in the next slide.

Implementing As Dynamically Allocated Data Structure

The dequeue() Function

```
1  int dequeue(struct fifo_pointers *fifo)
2  {
3      if(fifo->head) {
4          struct fifo_node *tmp = fifo->head->next;
5          int data = fifo->head->data;
6          free(fifo->head);
7          fifo->head=tmp;
8          if(tmp==NULL)
9              fifo->tail = NULL;
10         return data;
11     }
12     return -1;
13 }
```

Implementing As Dynamically Allocated Data Structure

The `dequeue()` Function

Please observe, that the `dequeue()` function definition is very similar to the definition of the `pop()` function for the stack. The `dequeue()` function, just like the `pop()` function returns `-1` if it is called for an empty queue. The operation of removing an element from the head of the queue is similar to the operation of removing an element from the top of a stack. There are only two differences. The first one is that the `head` and `tail` pointers are fields of a structure and the second one is that the `tail` pointer has to be assigned the `NULL` value, after an element is removed from a queue, that contained only one element. This is enforced by the assertions given in the previous slides. The assignment is performed in the 8th and 9th lines.

Implementing As Dynamically Allocated Data Structure

The `enqueue()` and `dequeue()` Functions — Summary

The *enqueue* and *dequeue* operations are the basic ones that should be implemented for the FIFO queue. There are necessary for using the data structure in a program. Their example implementations are presented in the previous slides. However, they may be written differently. For example, the `dequeue()` function could return no value or a value that describes the result of the operation of removing an element. That would require defining a separate function for reading the value of the element at the head of the queue. The way the functions are implemented depends on the preferences and needs of the programmer and the problem that she or he tries to solve.

Implementing As Dynamically Allocated Data Structure

Displaying Values of Elements on the Screen

Implementing the operation of displaying all values store in the elements of the FIFO queue is not mandatory, but it is quite convenient. In the next slides are presented two function that implements such an operation.

Implementing As Dynamically Allocated Data Structure

The print_queue() Function

```
1  void print_queue(struct fifo_pointers fifo)
2  {
3      while(fifo.head) {
4          printf("%d ",fifo.head->data);
5          fifo.head = fifo.head->next;
6      }
7      puts("");
8  }
```

Implementing As Dynamically Allocated Data Structure

The `print_queue()` Function

The structure of the queue pointers is passed by value to the function, because it is handy to use the `head` pointer for iterating over the elements of the queue. That however means, that the value of the pointer is changed inside the function and those modifications cannot "go" outside. Passing the queue pointers structure by value prevents such an issue. If the `head` pointer had a different value after the function exits than it had before the function started, then that would mean that the address of the fist element of the queue has been lost. The `while` loop inside the `print_queue()` function is performed as long as the `head` pointer has a value different than `NULL`, which means as long as the queue has elements containing not yet displayed values. The printing of the elements is performed in the 4th line. In the 5th line the `head` pointer is "moved" to the next element of the queue by storing in it the address stored in the `next` field of the element that it currently points to.

Implementing As Dynamically Allocated Data Structure

The `print_queue()` Function — The `for` Loop Version

```
1  void print_queue_with_for(struct fifo_pointers fifo)
2  {
3      for(;fifo.head;fifo.head=fifo.head->next)
4          printf("%d ",fifo.head->data);
5      puts("");
6  }
```

Implementing As Dynamically Allocated Data Structure

The `print_queue()` Function — The `for` Loop Version

The same operation of printing the values of elements of the FIFO queue can be implemented, in the C language, with the use of the `for` loop, just as it is demonstrated in the previous slide. The `head` pointer is the loop counter in the function. Please note, that the initialization part of the loop has been omitted. The condition part specifies that the loop is performed as long as the `head` pointer is not equal `NULL`. In the increment part the address of the next element in the queue is assigned to the `head` pointer. The function definition is briefer than the previous one, but slightly less legible.

Implementing As Dynamically Allocated Data Structure

An Example of Using — The `main()` Function

```
1  int main(void)
2  {
3       int i;
4       for(i=0;i<20;i++)
5           enqueue(&fifo,i);
6       print_queue_with_for(fifo);
7       while(fifo.head)
8           printf("%d ",dequeue(&fifo));
9       puts("");
10      return 0;
11 }
```

Implementing As Dynamically Allocated Data Structure

An Example of Using — The `main()` Function

In the `main()` function of the program, all defined functions for handling the FIFO queue are called, except for the `print_queue()` function. It can be invoked in place of the `print_queue_with_for()` function or just after the latter is called. In the 4th and 5th lines the `main()` function adds element containing natural numbers raging from `0` to `19` to the queue and then prints the content (the numbers) of the queue on the screen (line no. 6). Next, all the elements of the queue are removed and their values are displayed once more (lines no. 7 and no. 8).

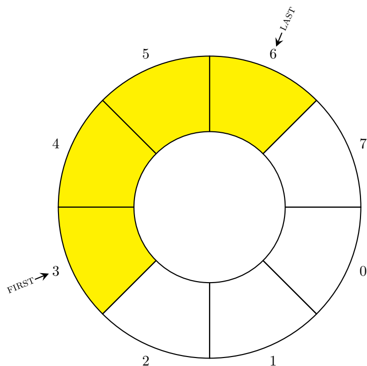
The FIFO Queue

Implementing With the Use of an Array

The FIFO queue can be implemented with the use of an array. In that case its capacity is limited by the number of the element in the array, but other than that it should behave in the same way as a dynamically allocated queue. If a new element cannot be added then the queue is called a *full queue*. The implementation of a queue based on an array is explained with the use of a program that stores integer numbers in such a data structure. The `head` and `tail` pointers are replaced int the queue by the `fist` and `last` indices. To simplify the implementation of such a queue the underlining array can be organized as a circular array, such that has no start or end. The queue implemented with the use of a circular array is depicted in the next slide.

The FIFO Queue

Implementing With the Use of Array



A partially filled FIFO queue

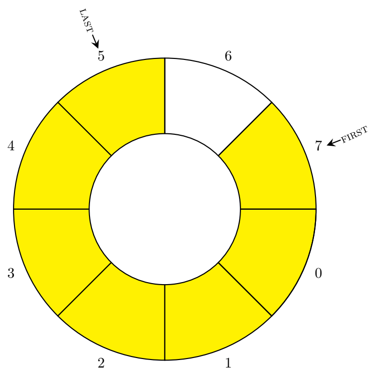
The FIFO Queue

Implementing With the Use of an Array

Using circular array for implementing a FIFO queue has two consequences. The values of both indices are only incremented by one regardless of the performed operation (adding or removing an element). On the other hand a way for detecting if the queue is empty or full has to be defined. One of the possibilities is using a separate variable for counting the elements of the queue. The other one is described by Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman in the book "Algorithms and Data Structures". The presented program is based on their solution. The full and empty queues are depicted in the next slides.

The FIFO Queue

Implementing With the Use of an Array



A full FIFO queue

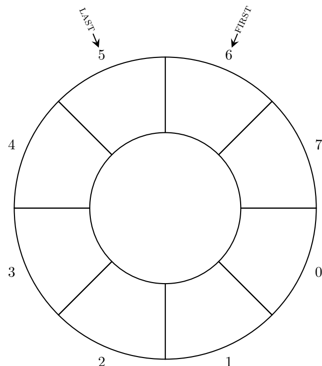
The FIFO Queue

Implementing With the Use of an Array

A FIFO queue is full when the values of the `last` and `first` indices differ by `2` modulo the number of the array elements. Please observe, that according to this definition in the full queue one element of the array remains unused, just as it is showed in the picture.

The FIFO Queue

Implementing With the Use of an Array



An empty FIFO queue

The FIFO Queue

Implementing With the Use of an Array

A FIFO queue is empty when the values of the `last` and `first` indices differ by `1` modulo the number of elements of the array.

The FIFO Queue

Implementing With the Use of an Array — The Queue Structure

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

#define FIFO_SIZE 20

struct queue
{
    int elements[FIFO_SIZE], first, last;
} fifo;
```

The FIFO Queue

Implementing With the Use of an Array — The Queue Structure

The previous slide contains the beginning of the example program that implements a queue with the use of an array. The `stdlib.h` file is replaced by the `stdbool.h` header file, because one of the functions is returning a value of the `bool` type and the program doesn't need functions for managing the heap. The `FIFO_SIZE` constant defines the number of the elements of the array. The capacity of the queue is smaller by one element. The array and the queue indices are defined in the program as fields of a structure of the `fifo` type. It can be stated that the structure is the queue itself.

The FIFO Queue

Implementing With the Use of an Array — The `add_one()` Function

```
1  int add_one(int index)
2  {
3      return (index+1)%FIFO_SIZE;
4  }
```

The FIFO Queue

Implementing With the Use of an Array — The `add_one()` Function

The `add_one()` function is used for incrementing the values of the queue indices by one. Using the reminder operator ensures that the values of each of the indices stay within an acceptable range. The function takes as an argument the current value of an index and returns the next one.

The FIFO Queue

Implementing With the Use of an Array — The `make_empty()` Function

```c
void make_empty(struct queue *fifo)
{
    fifo->first = 0;
    fifo->last = FIFO_SIZE-1;
}
```

The FIFO Queue

Implementing With the Use of an Array — The `make_empty()` Function

The `make_empty()` initializes the queue by "reseting" its indices. After the function is performed the `first` index is indicating the first element of the array and the `last` index indicates the last one.

The FIFO Queue

Implementing With the Use of an Array — The `is_empty()` Function

```
1  bool is_empty(struct queue fifo)
2  {
3      return add_one(fifo.last)==fifo.first;
4  }
```

The FIFO Queue

Implementing With the Use of an Array — The `is_empty()` Function

The `is_empty()` function returns the `true` value when there is no elements in the queue or the `false` value if there is at least one element in the queue. The function verifies if the value of the `last` index incremented with the use of the `add_one()` function is equal to the value of the `first` index[1]. If so, then the queue is empty.

[1] Please refer to the corresponding figure in the previous slides.

The FIFO Queue

Implementing With the Use of an Array — The `first_one()` Function

```
1  int first_one(struct queue fifo)
2  {
3      if(is_empty(fifo)==true)
4          return -1;
5      else
6          return fifo.elements[fifo.first];
7  }
```

The FIFO Queue

Implementing With the Use of an Array — The `first_one()` Function

In this program, the *dequeue* operation only removes the first element from the queue. The `first_one()` function returns the value of such an element. This element is indicated by the `fist` index. If the queue is empty, the function returns the `-1` value.

The FIFO Queue

Implementing With the Use of an Array — The enqueue() Function

```
1  void enqueue(struct queue *fifo, int data)
2  {
3
4      if(add_one(add_one(fifo->last))!=fifo->first)
5      {
6          fifo->last = add_one(fifo->last);
7          fifo->elements[fifo->last] = data;
8      } else
9          fprintf(stderr, "The queue is full!\n");
10 }
```

The FIFO Queue

Implementing With the Use of an Array — The `enqueue()` Function

The `enqueue()` function adds a new element and stores in it the value passed by the `data` parameter. Before it happens the function makes sure that the queue is not full. It accomplishes the task by applying the `add_one()` function to the `last` index twice and comparing the result with the value of the `first` index. If the values are equal then the queue is full and adding a new element is impossible[2]. In that case the function displays on the screen a message informing the user that the queue is full. If the queue is not full the function first increments the value of the `last` index with the use of the `add_one()` function and then it assigns the value of the `data` parameter to the element of the array indicated by the new value of the `last` index (lines no. 6 and no. 7).

[2] Please refer to the corresponding figure in the previous slides.

The FIFO Queue

Implementing With the Use of an Array — The `dequeue()` Function

```
1 void dequeue(struct queue *fifo)
2 {
3     if(is_empty(*fifo))
4         fprintf(stderr, "The queue is empty!\n");
5     else
6         fifo->first = add_one(fifo->first);
7 }
```

The FIFO Queue

Implementing With the Use of an Array — The `dequeue()` Function

The `dequeue()` function in this implementation of the queue returns nothing, just removes the first element. However, first it checks if the queue is empty. If so, the function displays a corresponding message on the screen and exits. Otherwise it removes the element by incrementing the value of the `first` index with the use of the `add_one()` function (line no. 6).

The FIFO Queue

Implementing With the Use of an Array — The `main()` Function

```
int main(void)
{
    int i;
    make_empty(&fifo);
    for(i=0;i<FIFO_SIZE-1;i++)
        enqueue(&fifo,i);
    while(!is_empty(fifo)) {
        printf("%d ",first_one(fifo));
        dequeue(&fifo);
    }
    return 0;
}
```

The FIFO Queue

Implementing With the Use of an Array — The `main()` Function

In the `main()` function of the program the queue is first initialized with the use of the `make_empty()` function and then in the `for` loop elements are added to the queue by calling the `enqueue()` function. The queue can have at most 19 of them. After the `for` loop terminates the `while` loop is performed in which the values of the queue elements are read with the use of the `first_one()` function and then the elements are removed with the use of the `dequeue()` function.

The array based implementations of the FIFO queue were used in programming languages that haven't supported dynamic allocation of the memory. Nowadays they are applied in computer systems with limited size of the RAM, such as microcontrollers. The keyboard buffer is also implemented in such a way. It is a place in memory where the keyboard controller stores data about keystrokes that are latter used by the CPU. It is an example of a limited capacity queue managed by a hardware.

Testing some of the operations on Dynamically Allocated Data Structures

Using dynamically allocated variables and data structures is quite a difficult task. It is relatively easy to make mistakes implementing operations for the stack, queue or any other similar data structure. Locating and removing such defects is a challenging task. There is however an easy way for testing functions, such as the `print_queue()` function, that implement operations on dynamically allocated data structures which don't involve allocating and deallocating memory. It only requires to create a queue or other data structure from elements which are statically allocated global or local variables. A queue created in such a manner can be applied for checking the behaviour of those functions. To some extend the same method can also be applied for testing functions that implement operations requiring allocating and deallocating memory. In the next slide a function is presented that uses the described method to test the behaviour of `print_queue()` function.

Testing some of the operations on Dynamically Allocated Data Structures

```
1  void print_queue_test(struct fifo_pointers *fifo)
2  {
3      struct fifo_node front, middle, rear;
4
5      front.data = 1;
6      front.next = &middle;
7      middle.data = 2;
8      middle.next = &rear;
9      rear.data = 3;
10     rear.next = NULL;
11
12     fifo->head = &front;
13     fifo->tail = &rear;
14     print_queue(*fifo);
15     fifo->head = fifo->tail = NULL;
16 }
```

Testing some of the operations on Dynamically Allocated Data Structures

In the `print_queue_test()` function are defined three structure, named `front`, `middle` and `rear`, of the `struct fifo_node` type. Those variables are used for creating a queue consisting of three elements (lines no. 3–9). To the `data` field of each of the structures is assigned a number. The `next` field of the first element gets the address of the second element. The `next` field of the second element gets the address of the third element. Finally, the `next` field of the third element gets the `NULL` value. The queue pointers are initialized in the 12th and 13th lines. The addresses of the first and the last element of the queue are assigned to them. The created queue can be used by the programmer to test the `print_queue()` function without worrying about damaging the integrity of the queue or causing memory leakages. In the 15th line the function zeros out the pointers of the queue, which means that the queue ceases to exist.

Summary

Queues, which can be implemented either as dynamically allocated data structures or by using arrays, have many applications. Operating systems use them for scheduling threads and processes, for implementing special variables called semaphores, for managing input-output operations and for many other purposes. Also other programs, like compilers or concurrent programs utilizes those data structures. The queues are also implemented in hardware, like the aforementioned keyboard buffer.

Often the programmers create a single element of the queue that exist throughout the whole life cycle of the program, to simplify functions that implement operations on a queue. It is a dummy element, that doesn't store any useful data and it is called a *sentinel node*. The queue that uses such an element is called a *queue with a sentinel*. Such a solution can also be applied for the stack.

Questions

?

The End

Thank You For Your Attention!