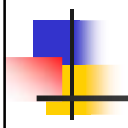


Queue

By:
Mr. Khaleifah Al.jada'
Data Structures
Al- Majma'ah Community College

Introduction

- Queue is a British word for *line*.
- All access is restricted to the least recently inserted elements
- Basic operations are `Append`, `Serve`, `Retrieve`.

Cont..

- Queue has two ends Front and Rear
- Front is used to retrieve and serve operations
- Rear is used to append

The principle of the queue is First In First Out (FIFO).

Examples

- Line for bank's customers
- Line printer queue
- Expect $O(1)$ time per queue operation because it is similar to stack.
- Queues are useful for storing pending work.

Cont…

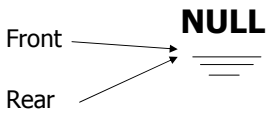
- There are several examples in computer science about queue:
  - Shortest paths problem (minimize the number of connecting flights between two arbitrary airports)
  - Topological ordering: given a sequence of *events*, and pairs (*a*,*b*) indicating that event *a* MUST occur prior to *b*, provide a schedule

Implementation

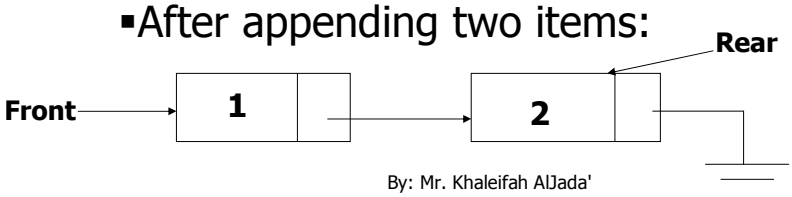
- Queue can be implemented by using :
  - Array.
  - Linked List.
- The problem in array implementation occurs when we try to serve because all the remaining items in the queue must be shifted up.

Example from A to Z

- After declaring a queue:

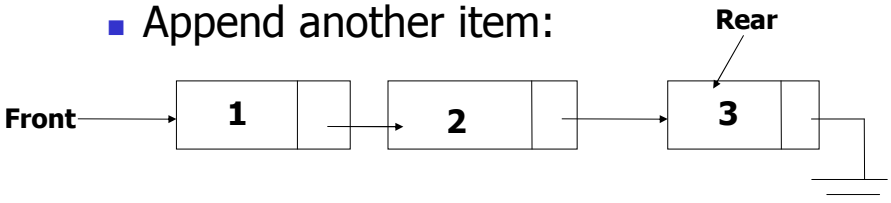


- After appending two items:

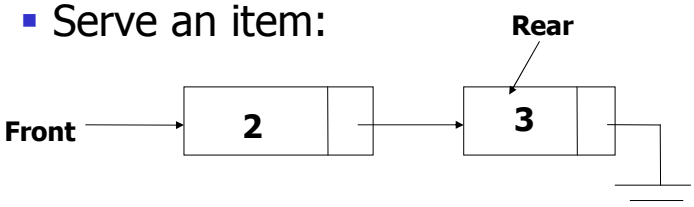


Cont…

- Append another item:



- Serve an item:



Class Linked Queue

```
class queue
{
   public:
        bool empty() const;
        bool serve();
        bool append(const Node_entry &item);
        bool retrieve(Node_entry &item) const;
        ¬queue();
   private:
        Node * front, *rear;
};
```


Methods

Constructor:

```
queue::queue()
{
   front = rear = NULL;
}
```

Empty:

```
bool queue::empty() const
{
   return rear == NULL;
}
```

Cont..

- ***Append***

```
bool queue::append(const Node_entry &item)
{
   Node * new_node = new Node(item);
   if(new_node == NULL)
        return false;
   if(rear == NULL)
   {
        rear = new_node;
        front = new_node;
   }
```


Cont..

```
        else
            {
                rear→next = new_node;
                rear = new_node;
            }
        return true;
    }
```

Cont…

Serve:

```
Bool queue::serve()
{
   if(rear == NULL)
        return false;
   else
        {
                Node *temp = front;
                front = front → next;
                delete temp;
        }
   return true;
}
```


Cont..

Retrieve:

```
bool queue::retrieve(Node_entry &item) const
{
   if( rear == NULL)
        return false;
   else
        item = front → entry;
   return true;
}
```

Cont…

Destructor:

```
queue:: ¬queue()
{
  while( !empty() )
      serve();
}
```


Extra Functions

Function Full:

```
bool Full() const
{
  Node *temp = new Node;
  if(temp==NULL)
      return true;
  else
      delete temp;
  return false;
}
```

Cont..

Method Size:

```
int Queue::size() const
{
   int count=0;
   Node *temp = front;
   while(temp != NULL)
   {
        count++;
        temp=temp→next;
   }
   return count;
}
```


Cont..

- ***Function Clear:***

```
void clear(queue Q)
{
   while( ! Q.empty() )
        serve();
}
```

Cont…

Method serve and retrieve:

```
bool queue::serve_retrieve(Node_entry& item)
{
   if(empty())
        return false;
   item = front → entry;
   node * temp = front;
   front = front →next;
   delete temp;
   return true;
}
```