It is a FIFO (first in first out) data structure.
It is a set of objects such that you can insert a new object at one end (called front) and delete from another (called rear).

We will show a linked list representation for a queue (write the array-based representation yourself). We will maintain two types of header nodes—"front" and "rear". They will contain data. In an empty queue i.e. we will not maintain any header nodes unlike what we did so far. In an empty queue, we will mark "front" and "rear" both as NULL.

```cpp
template <class T>
class LinkedQueue
{
private:
   Node<T> * front, * rear;
public:
   LinkedQueue() { front = rear = NULL; }

   bool IsEmpty()
   {
      return (front == NULL);
   }
};
```
It is a FIFO (first in first out) data structure. It is a set of objects such that you can insert a new object at one end (called "front") and delete from another (called "rear").

We will show a linked list representation for a queue (write the array-based representation yourself). We will maintain two types of header nodes — "front" and "rear". They will contain no data, in an empty queue i.e. we will not maintain any header nodes unlike what we did so far. In an empty queue, we will mark "front" and "rear" both as NULL.

```cpp
template <class T>
class LinkedQueue {
  private: Node<T> *front,*rear;
  public: LinkedQueue(){ front = rear = NULL; }

  bool IsEmpty () {
    return (front == NULL);
  }
};
```
void Insert (T & x)
{
    Node<T> * p = new Node<T>;
    p->data = x; p->next = NULL;
    if (front != NULL) // queue is not empty
    {
        rear->next = p;
    }
    else
    {
        front = p; // queue is empty.
        rear = p; // if there is only one node in
                   // the queue, front = rear
    }
}

void Delete ()
{
    if (IsEmpty()) throw exception;

    Node<T> * p = front;
    front = front->next;
    delete p;
    // in some variants, you might want to
    // save or return the data in p, before
    // deleting p
}
Now, we need to declare a linked queue as a friend class of Node here.

Also, we need to write a destructor for this class. The destructor is as follows:

```cpp
template <class T>
class linkedQueue<T>: public queue<T> {
public:
    linkedQueue() {}
    ~linkedQueue() {
        Node<T> *p;
        while (front != NULL) {
            p = front;
            front = front->next;
            delete p;
        }
    }

    // destructor
};
```

**Applications:**

Queue is widely used in all kinds of scheduling problems. For instance, customer care operators at a call center receive a number of calls. When the customer is being served, customers that call earlier are served.
Applications in Image Processing

Imagine an image of a tissue from the human body. Suppose the tissue contains a tumor. A pathologist observes the image on the screen and marks a point inside the tumor (with a mouse). We want to develop an algorithm which will tell the pathologist the total area of the tumor. We will assume that the pixels inside the tumor region have a fixed intensity (say 0) whereas the background has some other intensity. Of course, situations in actual image processing are much more complicated than this, but we will not get into those details. We will develop an algorithm that will tell the pathologist the area of the tumor starting from use of the information he gave us—namely the pixel inside the tumor that he marked. This pixel is called "seed pixel" and such algorithms are called seed-growing algorithms.
We will maintain a boolean array called "visited-flag", having the same size as the original image. This array will be set to "not-visited" for each pixel location initially. As the seed-growing algorithm proceeds, it will mark each pixel it visits as "visited".

```
init tumorseize (image I, visited flag, seed) {
  ts = 0;
  insert Q (seed);
  while (1, Q is empty Q (1)) {
    (x,y) = delete Q;
    for each neighbor (x',y') of (x,y) having same intensity as (x,y) and having coordinates (x',y') {
      if (visited-flags (x',y') \neq VISITED)
        insert Q (x',y'); visited-flags (x',y') = VISITED;
    }
  }
}
```
\[ ts = ts + 1; \] // increase area count

\[ \text{visited}_{-\text{flags}}(x, y) = \text{VISITED}; \] // mark visited

```
return ts;
```

Very similar algorithms can be written for other similar applications in image processing. Sometimes, there exist images that contain several regions each of which contain distinct color values. A "region" is basically a set of contiguous pixels. Given an image, we may need to determine the number of such regions, and create a list of pixel locations for each region. This is called as "connected components analysis". For example, the following image contains 5 regions (besides a background).