(a) When a triadiagonal matrix is stored by columns rows a one dimensional array t, the mapping is $t[0:3n-3]$ = $[M(1,1)$, $M(1,2)$, $M(2,1)$, $M(2,2)$, $M(2,3)$, $M(3,2)$, $M(3,3)$, $M(3,4)$, $M(4,3)$, $\cdots]$. If $|i-j| > 1$, then $M(i,j)$ is not on the tridiagonal. For elements in the tridiagonal, we see that the rwo 1 elements are stored in $t[0]$ and $t[1]$. So, when $i = 1$, $M(i,j)$ is at $t[j-1]$. When $i > 1$, there are $1-1$ rows that come before the first element of row 1. These $i-1$ rows contain $3(i-1)-1$ elements. Within row i, $M(i,j)$ is the $(j-i+2)$th element. So, if $i > 1$, $M(i,j)$ is at $t[j+2i-3]$. When $i = 1$, $j + 2i - 3 = j - 1$. So, this formula may also be used for the case $i = 1$.

Note also that the lower diagonal is stored in positions 2, 5, 8, $\cdots$ of the one-dimensional array; the main diagonal occupies positions 0, 3, 6, $\cdots$; and the upper diagonal ocupies positions 1, 4, 7, $\cdots$.

With these bservations in mind, we arrive at the following code:

```
template<class T>
class TriByRows {
   friend ostream& operator<<
          (ostream&, const TriByRows<T>&);
   friend istream& operator>>
          (istream&, TriByRows<T>&);
   public:
      TriByRows(int size = 10)
         {n = size; t = new T [3*n-2];}
      ~TriByRows() {delete [] t;}
      TriByRows<T>& Store
              (const T& x, int i, int j);
      T Retrieve(int i, int j) const;
      TriByRows(const TriByRows<T>& x);
         // copy constructor
      TriByRows<T>&
         operator=(const TriByRows<T>& x);
      TriByRows<T> operator+() const; // unary +
      TriByRows<T>
         operator+(const TriByRows<T>& x) const;
      TriByRows<T> operator-() const; // unary minus
      TriByRows<T>
         operator-(const TriByRows<T>& x) const;
      TriByRows<T>& operator+=(const T& x);
      TriByRows<T> Transpose();
   private:
      int n; // matrix dimension
      T *t;  // 1D array for tridiagonal
};
```

```
template<class T>
TriByRows<T>& TriByRows<T>::
                Store(const T& x, int i, int j)
{// Store x as T(i,j)
   if ( i < 1 || j < 1 || i > n || j > n)
       throw OutOfBounds();

   switch (i - j) {
      case 1: case 0: case -1: // in tridiagonal
            t[2 * i + j - 3] = x;
            break;
      default: if(x != 0) throw MustBeZero();
      }
   return *this;
}

template <class T>
T TriByRows<T>::Retrieve(int i, int j) const
{// Retrieve T(i,j)
   if ( i < 1 || j < 1 || i > n || j > n)
       throw OutOfBounds();

   switch (i - j) {
      case 1: case 0: case -1: // in tridiagonal
            return t[2 * i + j - 3];
      default: return 0;
      }
}
```

```
template<class T>
TriByRows<T>::TriByRows(const TriByRows<T>& x)
{// Copy constructor for tridiagonal matrices.
   n = x.n;
   t = new T[3 * n - 2];                // get space
   for (int i = 0; i < 3 * n - 2; i++) // copy elements
      t[i] = x.t[i];
}


template<class T>
TriByRows<T>& TriByRows<T>::
     operator=(const TriByRows<T>& x)
{// Overload assignment operator.
   if (this != &x) {// not self-assignment
      n = x.n;
      delete [] t; // free old space
      t = new T[3 * n - 2];      // get right amount
      for (int i = 0; i < 3 * n - 2; i++) // copy elements
         t[i] = x.t[i];
      }
   return *this;
}

template<class T>
TriByRows<T> TriByRows<T>::
     operator+(const TriByRows<T>& x) const
{// Return w = (*this) + x.
   if (n != x.n) throw SizeMismatch();

   // create result array w
   TriByRows<T> w(n);
   for (int i = 0; i < 3 * n - 2; i++)
       w.t[i] = t[i] + x.t[i];

   return w;
}
```

```
template<class T>
TriByRows<T> TriByRows<T>::
     operator-(const TriByRows<T>& x) const
{// Return w = (*this) - x.
   if (n != x.n) throw SizeMismatch();

   // create result array w
   TriByRows<T> w(n);
   for (int i = 0; i < 3 * n - 2; i++)
       w.t[i] = t[i] - x.t[i];

   return w;
}

template<class T>
TriByRows<T> TriByRows<T>::operator-() const
{// Return w = -(*this).
   // create result array w
   TriByRows<T> w(n);
   for (int i = 0; i < 3 * n - 2; i++)
       w.t[i] = -t[i];

   return w;
}

template<class T>
TriByRows<T>& TriByRows<T>::
     operator+=(const T& x)
{// Add x to each element of (*this).
   for (int i = 0; i < 3 * n - 2; i++)
       t[i] += x;
   return *this;
   }
```

```
template<class T>
TriByRows<T> TriByRows<T>::
           Transpose()
{// Compute the transpose of *this.

   // create result array w
   TriByRows<T> w(n);
   // copy lower diagonal of *this to
   // upper diagonal of w and upper of
   // *this to lower of w
   for (int i = 1; i < 3 * n - 2; i += 3) {
       w.t[i] = t[i + 1];
       w.t[i + 1] = t[i];
       }

   // copy main diagonal of *this to
   // main diagonal of w
   for (int i = 0; i < 3 * n - 2; i += 3)
       w.t[i] = t[i];

   return w;
}
```

```
template<class T>
ostream& operator<<(ostream& out,
                    const TriByRows<T>& x)
{// Put the elements of x into the stream out.
   out << "Lower diagonal is" << endl;
   for (int i = 2; i < 3 * x.n - 2; i += 3)
      out << x.t[i] << "  ";
   out << endl;

   out << "Main diagonal is" << endl;
   for (int i = 0; i < 3 * x.n - 2; i += 3)
      out << x.t[i] << "  ";
   out << endl;

   out << "Upper diagonal is" << endl;
   for (int i = 1; i < 3 * x.n - 2; i += 3)
      out << x.t[i] << "  ";
   out << endl;

   return out;
}
```

```
// overload >>
template<class T>
istream& operator>>(istream& in,
                    TriByRows<T>& x)
{// Input the tridiagonal matrix.

   cout << "Enter number of rows"
        << endl;
   in >> x.n;
   if (x.n < 0) throw BadInput();

   // input terms
   cout << "Enter lower diagonal" << endl;
   for (int i = 2; i < 3 * x.n - 2; i += 3)
      in >> x.t[i];

   cout << "Enter main diagonal" << endl;
   for (int i = 0; i < 3 * x.n - 2; i += 3)
      in >> x.t[i];

   cout << "Enter upper diagonal" << endl;
   for (int i = 1; i < 3 * x.n - 2; i += 3)
      in >> x.t[i];

   return in;
}
```

(b) The codes are in the files `trirow.*`.

(c) The complexity of the `Store`, `Retrieve`, default constructor, and destructor functions is $\Theta(1)$. The complexity of the remaining functions in $\Theta(n)$.