Assignment 2 - Sparse Matrices

Part A is required. **Part B** is optional but is worth two points extra credit (but must be submitted in addition to, and along with, **Part A**). Make sure you have read and understood

- both ***modules A*** and ***B*** this week, and
- ***module 2R - Lab Homework Requirements***

before submitting this assignment. Hand in only one program, please.

Part A (required) - Sparse Matrices

This is going to be a fun assignment. Imagine trying to allocate a 100000 x 100000 matrix of objects on any computer in a resident program (without using disk storage). Can't be done. But with your knowledge of ***vectors*** and ***lists***, you can create a template that you and your company can use to do just that.

Design a class template **SparseMat** which implements a ***sparse matrix***. Use **FHvector** and **FHlist**, only, as base ADTs on which to build your class. Your primary public instance methods will be:

- **SparseMat(int r, int c, const Object & defaultVal)** - A constructor that establishes a size (row size and column size both $\geq$ 1) as well as a default value for all elements.
- **const Object & get(int r, int c) const** - An accessor that returns the object stored in row **r** and column **c**. It throws a user-defined exception if matrix bounds (row and/or column) are violated.
- **bool set(int r, int c, const Object &x)** A mutator that places **x** in row **r** and column **c**. It returns **false** without an exception if bounds are violated. Also, if **x** is the default value it will remove any existing node (the internal data type used by **SparseMat**) from the data structure, since there is never a need to store the default value explicitly. Of course, if there is no node present in the internal data representation, **set()** will add one if **x** is not default and store **x** in it.
- **void clear()** - clears all the rows, effectively setting all values to the **defaultVal** (but leaves the matrix size unchanged).
- **void showSubSquare(int start, int size)** - a display method that will show a square sub-matrix anchored at **(start, start)** and whose size is **size x size**. In other words it will show the rows from **start** to **start + size -1** and the columns from **start** to **start + size - 1**. This is mostly for debugging purposes since we obviously cannot see the entire matrix at once.

Here is a sample **main()** that will test your template. However, this does not prove that you are correctly storing, adding and removing internal nodes as needed. You'll have to confirm that by stepping through your program carefully. You main should also print the upper left and lower right of the (huge) matrix, so we can peek into it.

```
#include <iostream>
using namespace std;
#include "FHsparseMat.h"

#define MAT_SIZE 100000
typedef SparseMat<float> SpMat;

// --------------- main ---------------
int main()
{
   SpMat mat(MAT_SIZE, MAT_SIZE, 0); // 100000 x 100000 filled with 0

   // test mutators
   mat.set(2, 5, 10);
   mat.set(2, 5, 35);   // should overwrite the 10
   mat.set(3, 9, 21);
   mat.set(MAT_SIZE, 1, 5);  // should fail silently
   mat.set(9, 9, mat.get(3, 9));  // should copy the 21 here
   mat.set(4,4, -9);
   mat.set(4,4, 0);   // should remove the -9 node entirely
   mat.set(MAT_SIZE-1, MAT_SIZE-1, 99);

   // test accessors and exceptions
   try
   {
      cout << mat.get(7, 8) << endl;
      cout << mat.get(2, 5) << endl;
      cout << mat.get(9, 9) << endl;
      cout << mat.get(-4, 7) << endl;  // should throw an exception
   }
   catch (...)
   {
      cout << "oops" << endl;
   }

   // show top left 15x15
   mat.showSubSquare(0, 15);

   // show bottom right 15x15
```

```
   mat.showSubSquare(MAT_SIZE - 15, 15);
}
```

Depending on how your **showSubSquare()** formats numbers, a run might look like this:

```
0
35
21
oops
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0 35  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0 21  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0 21  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0


  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0 99


Press any key to continue . . .
```

As support, you'll want a node class, which you should call **MatNode**, as a building block. I'll give this to you now without charge:

```
template <class Object>
class MatNode
{
protected:
   int col;

public:
   Object data;
   // we need a default constructor for lists
   MatNode(int cl = 0, Object dt = Object()) : col(cl), data(dt) {}
   int getCol() const { return col; }

   // not optimized yet for set() = defaultVal;  refer to forums
   const Object & operator=( const Object &x ) { return (data = x);}
};
```

Notes:

1. Use **protected**, rather than **private**, for your class data, as you will need this for next week.
2. If you have a **const** member function that uses ***iterators***, you'll need to use a **const_iterator** (not a plain iterator) to move through lists or you will get a compiler error.
3. In general, if you declare a const member function, you can only call other const member functions from it.
4. Your **MatNode** class template will need (and has, as you can see, above) a default constructor, as you will be creating an **FHlist** of these things for each row, and **FHlist** instantiates two objects (header and tail nodes) without parameters.
5. If you use a list ***iterator*** to plow through a list looking for something to erase, and find it, then you should break from the loop as soon as you erase the item - otherwise, your loop control which uses the **FHlist::end()** method will undoubtedly be stale after the erasure. This is the kind of think you'll have to do when you find occasion to remove a **MatNode** from some list due the fact that you are asked to store the default value at that location (which should trigger an **FHlist::erase()**).
6. As specified, **showSubSquare(),** only allows sub-matrices along the diagonal. You may change the signature if you wish to make it more flexible. (You can add io manipulators or other formatting in this if you wish - your output does not have to look like mine.)
7. See the note in the modules about using "**typename**" when instantiating an **FHlist** iterator.

Part B (2 points extra credit)

Overload **set()** so that it supports the following optional syntax: