UNIVERSITÉ DE MONTPELLIER

FACULTÉ DES SCIENCES DE MONTPELLIER

MASTER DE MATHÉMATIQUES

PARCOURS *"Modélisation et Analyse Numérique"*

PROGRAMMATION II - HAX011X - 2025/2026

HW - sparse_matrix

A naive implementation of matrices (arrays) is suitable for dense matrices, of which most elements are nonzero. For sparse matrices, on the other hand, a much more efficient approach is to store only the nonzero elements and ignore all the zero elements. This approach makes arithmetic operations much more efficient and saves a lot of time and storage.

Sparse matrices are thus a most useful tool in practical implementations. The data structures used in their implementation, however, should not be simple arrays anymore but more elaborated containers, like for instance linked_lists.

The items T in the linked_list should be objects that contain two fields: one to store the actual value of the element and the other to store the index of the column in which it lies. The sparse matrix should thus be implemented as a list of such linked_lists.

In summary, we need three new objects, which are studied below:

- the row_element object, which contains data about a particular element in a given row;
- the row object, which is a linked_list of row_elements objects and models a given row of the matrix;
- the sparse_matrix object, which is a list of rows objects.

Exercice 1. The row_element object

(1) The row_element class contains two **protected** data fields. The first field value_ is of type T, to be specified later in compilation time. This field contains the value of the element. The second field column_ is of type **int** and contains the index of the column in which the element is located in the matrix.
(2) As usual, although the data fields are protected, they can still be read/set by the public value() and column() member functions.
(3) The constructor should have two arguments corresponding to the two protected fields. Default values should be provided: 0 for value_, -1 for column_. Of course -1 is a meaningless value which should be updated during the construction of the matrix.
(4) The class should have a copy constructor and a definition of **operator**=.
(5) Next, some useful arithmetic operations should be defined. For example, the **operator**+=() can take either a T or a row_element argument. In either case, its value_ field is incremented by the corresponding value of the argument (note that this **operator**+=() will be used in the **operator**+=() of the linked_list merging algorithm, in order to add coefficients). In a similar way, **operator**−=, **operator**$_*$= and **operator**/= should be defined.

(6) Next, binary operator<(), operator>(), and operator==() as nonmember operators should be introduced. These operators take two row_element arguments and compare their column_ fields. For example, operator< returns 1 if the column index of the first argument is smaller than that of the second one, and 0 otherwise (this priority order will be used later to preserve increasing column order in matrix rows). Nnote that this operatorn=() will be used in the operator+=() of the linked_list merging algorithm, in order to check if coefficients have the same column before adding their value_.

(7) We should also define some useful nonmember binary arithmetic operators that involve a row_element object and a scalar. For instance:

```
const row_element<T> operator+(row_element<T> const& e, T const& t)
```

should return a row_element with updated value_ field. here is the required list:

```
const row_element<T> operator+(row_element<T> const& e, T const& t);
const row_element<T> operator+(T const& t, row_element<T> const& e);
const row_element<T> operator-(row_element<T> const& e, T const& t);
const row_element<T> operator-(T const& t, row_element<T> const& e);
const row_element<T> operator*(row_element<T> const& e, T const& t);
const row_element<T> operator*(T const& t, row_element<T> const& e);
const row_element<T> operator/(row_element<T> const& e, T const& t);
```

(8) Finally, a function that prints a row_element to the screen is needed.

Test your class in a main program.

Exercice 2. The row object

In this Exercice, we introduce the row class that implements a row in a sparse matrix. The row object is actually derived from linked_list<row_element<T>>.

The row class is also a template class. The template T that indicates the type of value of the element is to be specified later in compilation time. It is assumed that the elements in the row object are ordered in increasing order. The priority order of row_element objects used for this purpose is induced from the priority order of their corresponding columns.

(1) the constructor should be defined as follows:

```
row(T const& val=0, int col=-1){
        item=row_element<T>(val,col);
}
```

so that it calls the constructor of the row_element class, and insert the corresponding instance of row_element as the first element of the list (note that the default constructor of the base class list<row_element> is automatically called in priority when this constructor is invoked, creating an empty list, which will then be populated through methods of the derived class),

(2) a definition of the operator operator()(), declared as follows

```
row_element<T> const& operator()() const;
```

should return the first item of the list only,

(3) a method declared as follows

```
T const& value() const;
```

should return the first-item value,

(4) a method int column() const should return the first-item column index.

Since the row class is derived from the linked_list class, it can use its public and protected members. In particular, it can use the functions that insert or drop items. Still, the row class should contains its own local versions of the insert_next_item(), insert_first_item(), and append() functions. Indeed, these versions

should be different from the original versions coming from the class linked_list, since they take two arguments to specify the value_ and column_ of the inserted element.

(5) For instance, here is the definition of the insert_next_item method

```
void insert_next_item(T const& val, int col){
        row_element<T> e(val,col);
        linked_list<row_element<T> >::insert_next_item(e);
}  //  insert a row_element as second item
```

Note that this new version calls the method from the base class. Based on this example, define modified versions of the insert_first_item and append methods,

(6) Define a function **const** T row_sum() **const** using recursion in order to compute the sum of the elements in the row.
We have seen that the recursive structure of the base linked_list class is particularly useful. The definitions of many member functions use recursive calls applied to the p_next_ field that points to the rest of the row. However, the p_next_ field inherited from the base linked_list class is of type pointer-to-linked_list rather than pointer-to-row. Therefore, it must be converted explicitly to pointer-to-row before the recursive call can take place. This is done by adding the prefix (row*). Note that usually, this conversion is considered risky because in theory p_next_ can point to a linked_list object or any object derived from it, with a completely different interpretation of the recursively called function. Fortunately, here p_next_ always point to a row object, so the conversion is safe.

(7) Recursion may also be used to define again **operator**[]. This function should take an integer argument i and returns a copy of the value of the element in column i, if it exists. If, on the other hand, there is no such element in the current row, then it returns 0.
This should be carried out using the recursive pattern of the row object. First, the column_ field in the first element is examined. If it is equal to i, then the required element has been found, and its value is returned as output. If, on the other hand, it is greater than i, then there is no hope of finding the required element because the elements are ordered in increasing column order, so 0 is returned. Finally, if it is smaller than i, then the **operator**[] is applied recursively to the rest of the row.
As before, the p_next_ field must be converted explicitly from pointer-to-linked_list to pointer-to-row before recursion can be applied to it. Again, this is done by adding the prefix (row*).
Note also that the value returned by the **operator**[] function is of type constant-T rather than reference-to-constant-T. This is because, as discussed above, the function may also return the zero value. This local constant cannot be referred to and must be stored in the temporary unnamed variable returned by the function.

(8) Recursion should also used in the following arithmetic operators that involve a row and a scalar:

```
const row& operator*=(T const& t);
const row& operator/=(T const& t);
```

(9) Fortunately, the **operator**+= that adds a row to the current row (while preserving increasing column order) is already available from the base linked_list class, so there is no need to write it again. However, we have to adapt it to possibly *add* the values of the coefficients (remember the remarks about it during PW2, when writing the merging algorithm):

```
// hey : overloaded comparison operator==() for item_ (actually: row) may be used
if (p_scan2 && p_scan1->item() == p_scan2->item())
{
   // hey : overloaded comparison operator+=() for item_ (actually: row) may be used
     p_scan1->item() += p_scan2->item();
     p_scan2 = p_scan2->p_next();
 }
```

(10) A binary operator that computes the inner product of a row and a dynamic_vector is also required. This operation will be used later to compute the product of a sparse matrix and a vector. It may be defined as follows:

```
const T operator*(dynamic_vector<T> const& v) const;
```

(11) Also required is a function that re-numbers the columns with new numbers contained in a vector of integers named renumber. To increase efficiency, this vector is passed to the function by reference. As before, recursion is applied to the p_next_ field after its type is converted from pointer-to-linked_list to pointer-to-row. The corresponding declaration is:

```
void renumber_columns(dynamic_vector<int> const& renumber);
```

(12) We also need to define some non-member binary arithmetic operators that involve a row and a scalar:

```
const row<T> operator*(row<T> const& r, T const& t);
const row<T> operator*(T const& t, row<T> const& r);
const row<T> operator/(row<T> const& r, T const& t);
```

(13) Finally, we offer the following drop_items function, which takes as argument a vector of integers named mask. Happy Christmas ! The zeroes in this vector indicate that the row elements in the corresponding columns should be dropped. This is done by looking ahead to the next element and dropping it if appropriate. For this purpose, the row object that contains the rest of the elements in the row is first accessed as *p_next_. Then, the first item in this row object (which is actually the second element in the current row) is accessed as (*next)(), using the **operator**() in the base linked_list class. Now, the column of this element can be read, and if the corresponding component in mask vanishes, then this element is dropped by the drop_next_item function of the base linked_list class. This leads to the following code:

```
template<class T>
void row<T>::drop_items(dynamic_vector<int> const& mask){
    if(p_next()){
      if(!mask[(*p_next().column()]){
        drop_next_item();
        drop_items(mask);
      }
      else
        (*(row<T>*)p_next()).drop_items(mask);
       if(!mask[column()]) drop_first_item();
    }}   //   "masking" the row by a vector of integers
```

Exercice 3. The sparse_matrix object

In this Exercice, we introduce the sparse_matrix class. By implementing a sparse matrix as a list of linked_lists or, more specifically, as a list of row objects, only the nonzero matrix elements are stored and participate in calculations, whereas the zero matrix elements are ignored. Although linked_lists have their own drawbacks in terms of efficiency (because they use indirect indexing that may slow down the performance due to more expensive data access), this drawback is far exceeded by the advantage of avoiding trivial calculations. Furthermore, in some cases it is possible to map the linked_list to a more continuous data structure and make the required computations in it. The hierarchy of objects used to implement the sparse matrix is as follows:

- the sparse_matrix object, in the highest level, is implemented as a list of rows objects,
- the row object is by itself implemented as a linked_list of row_elements objects,
- the row_element object, at the lowest level, contains a template parameter T to store the value of the element.

Hence, the sparse_matrix class is a **template** class derived from a list of row<T> objects.Therefore, it enjoys access to the **public** and **protected** members of the list class. The additional member functions are defined below in the sparse_matrix class. In these methods, you may often loop over all the items

in the underlying list of rows. In this loop, member functions of the row class are used to access the row_elements.
Your new sparse_matrix class should include:

(1) a first (default) constructor sparse_matrix(**int** Nrows=0) which proceeds to the memory allocation for the list of row<T> according to the value of Nrows (precisely: allocate an array row<T>*[Nrows] and set all the pointers values to zero),

(2) a second constructor declared as sparse_matrix(**int** Nrows, T **const**& a); which deals with the memory allocation for the list<row<T>> according to the value of Nrows, and set the i^{th} row to be item_[i] = **new** row<T>(a,i);. (this leads to a scalar matrix stored as a sparse_matrix: only the diagonal terms are stored),

(3) a third constructor, allowing to directly read a sparse_matrix from a provided file, in a formatted way like the *Harwell-Boeing* file format, should be provided for practical use and left as an Exercice (not required for the HW to be validated),

(4) a definition of **const** T **operator**()(**int** i, **int** j) that returns the $(i, j)^{th}$ matrix element (if this element is of course non-zero),

(5) functions that returns the number of rows in the matrix, the number of columns, and the order (actually max(row_number(), column_number())) of a non-square matrix, declared as follows:

```
int row_number() const;
int column_number() const;
int order() const;
```

(6) some member functions dedicated to arithmetic operations:

```
const sparse_matrix& operator+=(sparse_matrix<T> const&);
const sparse_matrix& operator−=(sparse_matrix<T> const&);
const sparse_matrix& operator*=(T const&);
```

The first two ones add (resp. subtract) a sparse_matrix to (resp. from) the current sparse_matrix. The third one multiplies the current sparse_matrix by a scalar,

(7) some **friend** free functions declared as follows:

```
template<class S>
friend const sparse_matrix<S> operator+(sparse_matrix<S> const& M1, sparse_matrix<S> const& M2);
template<class S>
friend const sparse_matrix<S> operator−(sparse_matrix<S> const& M1, sparse_matrix<S> const& M2);
template<class S>
friend const sparse_matrix<S> operator*(sparse_matrix<S> const& M, const S& t);
template<class S>
friend const sparse_matrix<S> operator*(const S& t, sparse_matrix<S> const& M);
```

(8) a sparse_matrix × dynamic_vector product function. The returned value should be a dynamic_vector of suitable size. The function should be declared **friend** as follows:

```
template<class S>
friend const dynamic_vector<S> operator*(sparse_matrix<S> const& M, dynamic_vector<S> const& v)
```

Note that the structure of the sparse_matrix as a list of rows should be used. The product method of the list class (declared as **operator***(dynamic_vector<T> **const**& v) **const**) can be useful,

(9) a sparse_matrix × sparse_matrix product should also be defined, introduced as follows;

```
friend const sparse_matrix<T> operator*<T>(sparse_matrix<T> const&, sparse_matrix<T> const&);
```

In order to take benefit from the data structure of the sparse_matrix, a product algorithm that relies on rows operations should be implemented. Specifically, we observe that if A is a matrix with N rows and B is a matrix with N columns, with B_i the i^{th} row of B, then the i^{th} row of BA

can be written as a linear combination of rows in A with coefficients in b_i as follows:

$$(BA)_i = B_i A = \sum_j B_{ij} A_i.$$

(10) a function that computes the transposition of a sparse_matrix, declared as follows:

```
friend const sparse_matrix<T> transpose<T>(sparse_matrix<T> const&);
```

(11) a function that extract the diagonal of a sparse_matrix and return it as another sparse_matrix of same size, declared as follows:

```
template<class T>
const sparse_matrix<T> diagonal(sparse_matrix<T> const& M);
```

(12) a print free function declared as follows:

```
template<class S>
friend void print(sparse_matrix<S> const& M);
```

which prints on screen the non-zero coefficients, together with their row-column indices,

(13) a printf free function declared as follows:

```
template<class S>
friend void printf(sparse_matrix<S> const& M, ofstream& os);
```

which prints the non-zero coefficients row, column and value in an "out-file-stream" for plotting purpose (see the next Exercice below).

Test your class in a main program: show me with many examples that your hierarchy of classes is correctly operating.