NPRG041 – 2024/25 Winter – Labs MS – **Big Assignment A02**

Graphs

Flexible Array

For the purpose of the intended graph representation, it is first expected to implement a custom container template of a flexible array `lib::Array<T>`, which will allow for storing elements of any data type `T` in a way that guarantees the immutability of element locations throughout their existence. In other words, when manipulating this container (e.g., when adding or removing individual elements), there will never be a need to perform the internal storage reallocation. This will not only result in a more friendly work with dynamically allocated memory, but, at the same time, there will never be a need to relocate the existing elements in memory (at best, by moving, at worst, by copying, according to the capabilities of these elements), and thus invalidate any references or pointers to these elements held by the users.

The extent of functionality of the flexible array container and the associated iterators is assumed to be at the level of the two relevant small homework assignments. On a practical level, however, it is actually not necessary to have this implementation fully complete, as it will suffice to only have those functions that you really want to use within the graph. As for the iterators, in particular, it will suffice to reach just the category of forward iterators. On the other hand, if need be, it is possible to extend the existing functionality of the array even further. We continue to assume the possibility of activating the debug mode via the macro `__DEBUG__`, thanks to which the flexible array will throw exceptions in the corresponding edge situations.

If you do not have the flexible array implementation at all or at least not sufficiently debugged, you can actually use an ordinary standard container `std::vector` within this task. However, it is important to realize that the standard vector does not provide the aforementioned guarantees and you will thus have to deal with certain implied limitations. In any case, use of a flexible array is suitable and desirable, yet not mandatory nor necessary.

Graph Representation

The main goal of this assignment is to program a class template that would allow us to represent a graph, both directed and undirected. In both these cases, there can be at most one edge only (in a given direction) between any pair of nodes, so we only consider an ordinary graph, not a multigraph. It is also important to note that we also need to be capable of binding these nodes and edges with any additional information we want to store with them.

The graph class as such `Graph<NData, EData>` will provide a basic common interface for both the mentioned variants of graphs. It will, however, be abstract, it will not be possible to create instances from it. This will only be possible for a particular derived variant of a directed graph `DirectedGraph<NData, EData>` and an undirected graph `UndirectedGraph<NData, EData>`.

Template parameter `NData` determines a data type used for the representation of the information associated with individual nodes, and, analogously, `EData` with edges. In both cases, any basic type as well as an arbitrary standard or user-defined class can be chosen, but pointers are not allowed. Whatever type is selected, it will be copyable and moveable (in this case in the `noexcept` mode), have a default constructor, and implement its own `<<` and `>>` operators for working with streams.

In terms of the logical decomposition, the graph class will contain two separate components, namely an instance of a `Nodes<NData, EData>` class for handling all the functionality related to the graph nodes, and, analogously, an instance of an `Edges<NData, EData>` class for the edges. Instead of standalone classes, you can alternatively implement both the components as inner classes `Graph<NData, EData>::Nodes` and `Graph<NData, EData>::Edges` as well. Each individual node will be implemented as an instance of a `Node<NData>` class, an edge as an instance of an `Edge<EData>` class, regardless of whether or not it is directed. In other words, edges themselves do not need to and will not distinguish this fact.

Although a large part of the entire solution design will not be restricted in any way, certain aspects of the expected interface must, as usual, be observed. Moreover, in many situations, we will need variants of functions for modifiable but also constant graphs or their components. In order to avoid duplication of the headers of such functions in the following text, we just mark them with the [♠] symbol. Analogously, we will use the [♣] symbol in situations where, for a particular parameter, we want to offer two variants of its forwarding via a constant lvalue reference or a modifiable rvalue reference, respectively.

Nodes and Edges

Each node and edge is assigned a unique identifier from the set $\mathbb{N}_0$, i.e., natural numbers including zero. These identifiers are immutable, i.e., once assigned, they can never be changed later. If the graph contains $n \in \mathbb{N}_0$ nodes, then their identifiers cover all the numbers from 0 to $n - 1$. In other words, these identifiers follow each other continuously, we do not skip any. The same applies analogously (separately) to the edges. Technically, we will use the `size_t` data type for these identifiers, but we will work with an `Identifier` alias in the code to facilitate possible future changes. Whatever type we will assume, though, it will always have the `<<` and `>>` operators implemented, and its values will be implicitly convertible to non-negative `size_t` values.

Specifically for the node class, the following interface needs to be implemented:

- `Identifier getId() const`: returns the identifier of a given node
- `NData& getData()` [♠]: returns a reference to the data content associated with a given node

In the case of the edge class, the interface is analogous, we just add the possibility of accessing nodes determining a given edge:

- `Identifier getId() const`: returns the identifier of a given edge
- `Identifier getSource() const`: returns the identifier of the source node
- `Identifier getTarget() const`: returns the identifier of the target node
- `EData& getData()` [♠]: returns a reference to the data content associated with a given edge

We will further implement the `<<` operators for nodes as well as edges so that we can print them to any output stream. In both cases, the following output format needs to be respected. If we have a node with identifier n, we serialize it into a string `node (`*n* `{`*data*`})`, where *data* (that is, the string between the pair of curly braces) represents the serialized data content. Whatever it looks like, it will never contain other embedded curly braces. It is assumed that each `NData` type we will want to use also implements the corresponding operator `<<`, and thus can itself be printed in this way. Similarly, for an edge with identifier m between nodes with identifiers i and j, we will expect the output in the form `edge (`*i*`)-[`*m* `{`*data*`}]->(`*j*`)`, where *data* again constitutes the serialized content of the `EData` instance obtained by the appropriate `<<` operator. We will print the suggested arrow for all edges, regardless of whether they are directed or not.

The output of individual nodes and edges shall not be terminated by line endings. For example, if we had a graph `DirectedGraph<std::string, std::string>`, it could contain a node `node (1 {one})` or an edge `edge (1)-[3 {one-four}]->(4)` between nodes 1 and 4.

Graph Components

In terms of its data content, each graph must logically (not necessarily also physically) directly or indirectly store (in the sense of ownership) at least the following three data members (including transitively their content): 1) internal storage for node instances, 2) internal storage for edge instances, and also 3) adjacency matrix for efficient retrieval of edges based on the knowledge of identifiers of pairs of nodes they should connect. In all three cases, we assume that our flexible array `lib::Array` will primarily be used, yet standard `std::vector` will suffice as well. In addition to the listed items, it is, of course, possible to store other data if need be.

The graph class will expose the components for nodes and edges as follows:

- `Nodes<NData, EData>& nodes()` [♠]: returns the reference to the component handling nodes
- `Edges<NData, EData>& edges()` [♠]: analogously for edges

When it comes to placing the mentioned data members, there are two basic solutions: either you put them all in one place directly in the graph class `Graph<NData, EData>`, or you can split them up and place them accordingly into the classes of both the components `Nodes<NData, EData>` and `Edges<NData,`

EData>. Both variants offer certain advantages, but they also have their drawbacks. Therefore, think carefully about which approach is closer to you. In any case, both components must be preserved, if only to encapsulate the expected functionality, since it simply cannot be provided in just one place directly in the graph itself.

As for the internal storage for node objects, all nodes in that container are assumed to be arranged exactly in the order of their identifiers. In other words, node at a position n in the array must have an identifier n. The same will analogously be satisfied for the internal container with edge objects.

The adjacency matrix can be implemented via an ordinary data member as well as via a separate class. Technically, it can contain pointers to the respective edges, or, better, their identifiers. Again, all variants offer certain advantages and disadvantages, the choice of a particular solution should thus not be underestimated. One way or the other, we need to realize that our graph is supposed to be dynamic (it must allow for the addition of new nodes), it would therefore definitely not be a good idea to linearize the matrix content. To conclude, let us just note that a position (i, j) corresponds to an edge leading from a node with an identifier i to a node with an identifier j. In the case of an undirected graph, the matrix must be symmetric along the main diagonal. Loops, i.e., edges starting and ending at the same node, are also supported.

Graph Serialization

We must be able to print the graph as a whole, meaning its nodes and edges, to the standard output (or to another stream). It applies that we always print all the nodes first, in ascending order according to their identifiers, only then all the edges, again in ascending order according to their identifiers. Always one node or one edge per line, terminating each one with `std::endl`. As expected, the `Nodes<NData, EData>` class will be able to print the nodes, while the `Edges<NData, EData>` class will be able to print the edges. In both cases, we achieve this via the following interface:

- `void print(std::ostream& stream = std::cout) const`: prints the component of nodes or edges

We can then easily add the following functions to the graph class as a whole:

- `void print(std::ostream& stream = std::cout) const`: prints the entire graph (i.e., first all the nodes and then all the edges) into a specified stream
- `void print(const std::string& filename) const`: prints the graph in the same way to an output file with a given name

Both the components as well as the entire graph will also supply analogous implementations of the `<<` operators. Sample graph serialization could then look like this:

```
node (0 {zero})
node (1 {one})
node (2 {two})
edge (0)-[0 {zero-one}]->(1)
edge (0)-[1 {zero-two}]->(2)
```

For debugging and testing purposes, the edges class will also have the following function, using which will be able to separately print the content of the already discussed adjacency matrix:

- `void printMatrix(std::ostream& stream = std::cout) const`: prints the current content of the adjacency matrix to a given stream; we write each matrix row on a separate line of the output; we then list the values (individual columns) in the matrix row one after the other and separate them with the `|` symbol; if there is an edge at a given position, we print its identifier; otherwise we print the `-` symbol

Serialization of the adjacency matrix from the previous sample graph (in the undirected variant) would then, in particular, look like this:

```
-|0|1
0|-|-
1|-|-
```

Inserting Nodes and Edges

The newly constructed graph will always be empty, i.e., it will not contain any nodes or edges. We can then add them individually using functions we will focus on now. In particular, the component of nodes will offer the following interface:

- Node<NData>& add(Identifier id, NData [♣] data): inserts a new node with an explicitly given identifier and associated data into the graph; at this moment, it is assumed that the specified identifier is valid, i.e., corresponds to the first not yet used position in the flexible array; for the purposes of testing and exceptions handling, it is further assumed that a new node instance is first inserted into this array and only then the adjacency matrix is modified; reference to the created node object is returned
- Node<NData>& add(NData [♣] data): the same, only the identifier of the newly inserted node is implicitly specified as the next free

The component of edges will similarly offer the following functionality:

- Edge<EData>& add(
  Identifier id, Identifier source, Identifier target, EData [♣] data
  ): inserts a new edge with an explicitly given identifier and associated data into the graph, namely an edge connecting the specified pair of nodes; again, for this moment, we assume that all parameters are correct; reference to the created edge object is returned
- Edge<EData>& add(
  Identifier source, Identifier target, EData [♣] data
  ): the same, only the identifier of the inserted edge is implicitly determined as the next free

If for some reason it happens that a new node or a new edge cannot be inserted into the graph (due to failed memory allocation within our internal array container or due to failed copying of bound data instances), it is necessary to treat such a situation correctly and achieve the expected atomicity. That is, the insertion operation will either succeed in its entirety, or it will not at all. In case of a partial failure, it is therefore necessary to return all graph structures to a state that will again be consistent.

Graph Import

To make creating graphs easier, we also implement functions with which we will be able to load and import graph content from an input file (or another stream). In this case, we can only import nodes and edges into a graph instance already created. Moreover, the import function can be called repeatedly, and thus the graph instance can be composed, for example, from several input files containing individual smaller parts of the entire graph, as well as interleaved arbitrarily with calls of the previously discussed functions for the manual addition of individual nodes or edges.

We define the import functions at the level of the graph class as a whole:

- void import(std::istream& stream = std::cin): imports nodes and edges from a given input stream
- void import(const std::string& filename): imports nodes and edges from an input file with a given name

Nodes and edges can be mixed arbitrarily in the input data. Therefore, it may not hold that all the nodes are listed first and only then all the edges. It is, however, always true that each line contains only one node or one edge. We then insert them into the graph gradually without changing their order. At the same time, we can also rely on the fact that the records of each node and edge are syntactically correctly formed and conform to the structure we already described within the respective serialization functions.

When it comes to data content linked to nodes and edges, we need to retrieve it from a string between the pair of curly braces using the >> operator. In other words, we assume that each specific data type we want to use for nodes or edges in this sense must implement the corresponding stream extraction operator.

It is assumed that the import function will internally use the previously introduced functions for adding individual nodes and edges. Any entirely empty line will be skipped. If the import process fails, nodes or edges that have already been successfully added into the graph will be preserved. We therefore expect atomic behavior only at the level of individual nodes and edges. For illustration, let us provide the following valid input file, which, despite the mixed order of nodes and edges, corresponds to our first example:

```
node (0 {zero})
node (1 {one})
edge (0)-[0 {zero-one}]->(1)
node (2 {two})
edge (0)-[1 {zero-two}]->(2)
```

Accessing Nodes and Edges

The nodes component class will offer the following functions, with which we will be able to access or query particular nodes:

- `size_t size() const`: returns the current number of nodes in the graph
- `bool exists(Identifier id) const`: tests for the existence of a node, i.e., returns `true` if a node with a given identifier exists in the graph, otherwise `false`
- `Node<NData>& get(Identifier id)` [♠]: returns a reference to a node with a given identifier; for now, let us assume we can only access the existing nodes
- `Node<NData>& operator[](size_t id)` [♠]: the same

Analogously, we will prepare the following functions for the edges component class:

- `size_t size() const`: returns the current number of edges in the graph
- `bool exists(Identifier id) const`: tests for the existence of an edge, i.e., returns `true` if an edge with a given identifier exists in the graph, otherwise `false`
- `bool exists(Identifier source, Identifier target) const`: the same, only for the existence of an edge between the specified pair of nodes determined by their identifiers
- `Edge<EData>& get(Identifier id)` [♠]: returns a reference to an edge with a requested identifier; for now let us again assume valid identifiers only
- `Edge<EData>& get(Identifier source, Identifier target)` [♠]: the same, only for a given pair of nodes according to their identifiers

We will also add the `[]` operator to the edges component so that we can access particular edges. However, not as a one-level approach using the identifiers of such edges, but as a two-level approach using the identifiers of a pair of the corresponding nodes `source` and `target` (in that order). We thus want to be able to use code fragments of the form `graph.edges()[source][target]` [♠].

Graph Iterators

The components of nodes and edges will also offer iterators that will allow for the iteration over individual nodes and edges in the graph. Of course, it is not necessary to implement anything new, we just need to utilize and expose the iterators we already have within the internal array. Just to remind, at least the level of forward iterators in terms of their capabilities is expected. We specifically need to define two pairs of functions `begin()` [♠] and `end()` [♠] with an obvious meaning.

Graph Manipulation

As for the graph class, it is also necessary to implement its standard copy and move (stealing) constructors and assignment operators, thanks to which we will be able to manipulate graphs as a whole. To avoid any misunderstandings, let us add that every node and edge is in the logical ownership of a graph instance to which it belongs. Therefore, if we are about to copy or move graphs as a whole, we have to copy or move their entire content as well. We must also not forget to ensure the expected atomicity. The possibility of wrapping the entire function body (including a member initializer list within a constructor) using try/catch blocks can also help us in this sense.

The move constructor and assignment operator must guarantee to preserve references to all individual nodes and edges, but not to the components themselves. We will leave the stolen source graphs empty (without any nodes or edges) and internally consistent (so that they can still be used correctly for testing purposes). The copy assignment operator leaves the target graph empty and internally consistent again in case of any failure (achieving atomicity at the level of *Strong Exception Guarantee* is thus not wanted). Since we will need a graph emptying function anyway because of the mentioned reasons, we will add it into the public interface of the entire graph in the form of `void clear()`, too.

Error Situations

Until now, we have mostly assumed that parameters of our functions were valid, as well as that nothing else could get wrong. Within this section, we describe how to modify all the affected parts of our code in such a way that we will be able to detect the majority of edge and error situations and also treat them appropriately.

To achieve this, we will introduce a simple hierarchy of custom exceptions. The abstract ancestor will be our own `Exception` class. It will provide a single function, `const char* message() const`. Its purpose will be to return a specific textual description of the error that occurred. For practical reasons, we will need to distinguish between two types of derived exceptions, namely exceptions with fixed text messages (known at compile time), and variable messages (dynamically composed and constructed at runtime).

In the first category, there will only be a single derived class `MemoryException`. This is because it will be invoked when there is insufficient memory for dynamic allocation. At such a moment, of course, it is not possible to throw exceptions that would need it by themselves. The second category will contain all other derived exception types, namely `IdentifierException`, `ElementException`, `ConflictException`, and `FileException`.

The following overview contains particular situations we want to handle, their order, expected exception types, as well as anticipated messages. These messages will contain placeholders in the form of `$something`, which we just replace with the appropriate particular values:

- Access to a particular node using the `get` function or the `[]` operator based on its identifier:
  - `ElementException("`Node with identifier `$id` does not exist`");`
    when accessing a node that does not exist
- Access to a particular edge using the `get` function based on its identifier:
  - `ElementException("`Edge with identifier `$id` does not exist`");`
    when accessing an edge that does not exist
- Access to a particular edge using the `get` function or `[]` operator based on a pair of nodes:
  - `ElementException("`Source node with identifier `$source` does not exist`");`
    when the source node does not exist
  - `ElementException("`Target node with identifier `$target` does not exist`");`
    when the target node does not exist
  - `ElementException("`Edge between nodes `$source` and `$target` does not exist`");`
    when a non-existent edge is accessed between an otherwise valid pair of nodes
- Existence test of an edge using the `exists` function based on a pair of nodes:
  - `ElementException("`Source node with identifier `$source` does not exist`");`
    when the source node does not exist
  - `ElementException("`Target node with identifier `$target` does not exist`");`
    when the target node does not exist
- Addition of a new node using the `add` function:
  - `IdentifierException("`Invalid node identifier `$id` requested`");`
    when an invalid identifier is used (identifier with the highest possible `size_t` value)
  - `ConflictException("`Node with identifier `$id` already exists`");`
    when an identifier of an already existing node is used

  - `IdentifierException("`Non-successive node identifier `$id` requested`");`
    when a not-following-up identifier is used
  - Original exception when making a copy of the bound data instance fails
  - `MemoryException("`Unavailable memory for a new node in the nodes container`");`
    when it is not possible to insert a node object into the internal array for nodes
  - `MemoryException("`Unavailable memory for the adjacency matrix extension`");`
    when it is not possible to add a new column and row into the adjacency matrix

- Addition of a new edge using the `add` function:
  - `IdentifierException("`Invalid edge identifier `$id` requested`");`
    when an invalid identifier is used (identifier with the highest possible `size_t` value)
  - `ConflictException("`Edge with identifier `$id` already exists`");`
    when an edge with the specified identifier already exists
  - `IdentifierException("`Non-successive edge identifier `$id` requested`");`
    when a not-following-up identifier is used
  - `ElementException("`Source node with identifier `$source` does not exist`");`
    when the source node does not exist
  - `ElementException("`Target node with identifier `$target` does not exist`");`
    when the target node does not exist
  - `ConflictException("`Edge between nodes `$source` and `$target` already exists`");`
    when an edge between a given pair of nodes already exists
  - Original exception when making a copy of the bound data instance fails
  - `MemoryException("`Unavailable memory for a new edge in the edges container`");`
    when it is not possible to insert an edge object into the internal array for edges

- Printing a graph to a specified file using the `print` function:
  - `FileException("`Unable to open output file `$filename`");`
    when a given output file cannot be opened

- Importing a graph from a specified file or stream using the `import` function:
  - `FileException("`Unable to open input file `$filename`");`
    when a given input file cannot be opened (only for the first variant)
  - Exceptions thrown by the functions for adding individual nodes and edges

- Graph copy constructor:
  - `MemoryException("`Unavailable memory for constructing a source graph copy`");`
    when a copy of the source graph could not be created due to lack of memory
  - Original exception when making a copy of data instances bound to nodes or edges fails

- Graph copy assignment operator:
  - `MemoryException("`Unavailable memory for assigning a source graph copy`");`
    when a copy of the source graph could not be created due to lack of memory
  - Original exception when making a copy of data instances bound to nodes or edges fails

Final Instructions

Submit all the created source files (probably only `Graph.h` and `Array.h`) except the `Main.cpp` file, which is already part of the prepared test. It contains a single directive `#include <Graph.h>` and also the `main` function, which will control the course of the test as usual.

The objective of the task is to show the ability to work with the constructs we have encountered since the beginning of the semester. In addition to basic skills, it is mainly about working with text files, streams, design of classes, use of constructors and destructors, inheritance, virtual methods, dynamic allocation, templates, pointers, operators, iterators or exceptions.

The submitted implementation must, of course, be correct and stable, the compilation must take place without any warnings. The overall quality of the code will also be evaluated, though. It means, especially, but not exclusively, organization of code into individual files, classes and functions, use of header files, naming of files, functions and variables, overall visual style of the code and indentation, passing of parameters by value or reference, quality of the class design and use of inheritance and virtual methods, not repeating the same code fragments unnecessarily, using named constants, handling error situations, as well as using standard libraries, containers or functions.