NPRG041 – 2025/26 ZS – Cvičení MS – **Velký úkol A02**

Grafy

Gumové pole

Pro potřeby zamýšlené reprezentace grafu se nejprve očekává, že naprogramujete vlastní šablonový kontejner gumového pole `lib::Array<T>`, který umožní ukládání prvků libovolného datového typu `T` způsobem zaručujícím neměnnost umístění všech prvků po celou dobu jejich existence. Jinými slovy při manipulaci s tímto kontejnerem (např. při vkládání nebo odebírání prvků) nikdy nedojde k nutnosti realokace vnitřního úložiště. To celkově přinese nejenom šetrnější práci s dynamicky alokovanou pamětí, současně ale nikdy nedojde k potřebě přemisťování existujících prvků v paměti (v lepším případě přemisťováním, v horším kopírováním dle schopností těchto prvků), a tedy k zneplatnění uživatelem držených referencí nebo ukazatelů na tyto prvky.

Rozsah funkcionality kontejneru gumového pole a přidružených iterátorů se předpokládá na úrovni příslušných dvou malých domácích úkolů. V praktické rovině však není potřeba mít tuto implementaci zcela kompletní, postačí mít k dispozici jen ty funkce, které opravdu v rámci grafu používat budete chtít. Zejména pokud jde o iterátory, plně postačí dosažení jen kategorie dopředných iterátorů. V případě potřeby je naopak možné stávající funkcionalitu pole ještě rozšířit. Nadále pak předpokládáme i možnost aktivace ladicího režimu pomocí makra `__DEBUG__`, díky čemuž bude gumové pole v odpovídajících krajních situacích vyhazovat příslušné výjimky.

Pokud gumové pole implementované nemáte vůbec nebo alespoň ne dostatečně odladěné, můžete ve skutečnosti v rámci tohoto úkolu použít i obyčejný standardní kontejner `std::vector`. Akorát je potřeba si uvědomit, že standardní vektor výše uvedené záruky nenabízí, což povede k nutnosti vypořádat se s určitými z toho vyplývajícími omezeními. V každém případě tedy použití gumového pole je vhodné a vítané, ale ne povinné ani nutné.

Reprezentace grafu

Hlavním cílem tohoto domácího úkolu je naprogramovat šablonovou třídu pro reprezentaci grafu, a to orientovaného i neorientovaného. V obou případech platí, že mezi libovolnou dvojicí vrcholů může (v daném směru) existovat nejvýše jedna hrana, uvažujeme tedy jen obyčejný graf, nikoli multigraf. Důležité rovněž je, že na každý vrchol i hranu potřebujeme umět navázat další libovolné informace, které si u nich budeme chtít uložit.

Třída pro graf jako takový `Graph<NData, EData>` bude poskytovat základní společné rozhraní pro obě uvedené varianty grafů. Bude však abstraktní, nebude od ní možné vytvářet instance. To bude možné až pro konkrétní odvozenou variantu orientovaného grafu `DirectedGraph<NData, EData>` a neorientovaného grafu `UndirectedGraph<NData, EData>`.

Šablonový parametr `NData` určuje datový typ použitý pro reprezentaci zmíněných informací navázaných na jednotlivé vrcholy, `EData` analogicky pro hrany. V obou případech může jít o libovolné základní typy stejně jako o standardní nebo uživatelsky definované třídy, ukazatele však povoleny nejsou. Ať už bude zvolen jakýkoli typ, bude kopírovatelný i přesouvatelný (v tomto případě v režimu `noexcept`), bude mít defaultní konstruktor a bude implementovat svoje operátory `<<` a `>>` pro práci se streamy.

Třída grafu bude z hlediska logického členění obsahovat dvě oddělené komponenty, a to instanci třídy `Nodes<NData, EData>` pro řešení veškeré funkcionality týkající se vrcholů grafu a analogicky instanci třídy `Edges<NData, EData>` pro hrany. Namísto samostatných tříd je možné obě komponenty alternativně dle vlastního uvážení implementovat i jako vnitřní třídy `Graph<NData, EData>::Nodes` resp. `Graph<NData, EData>::Edges`. Každý jednotlivý vrchol bude realizován jako instance třídy `Node<NData>` a hrana jako instance třídy `Edge<EData>`, a to bez ohledu na to, jestli je, nebo není orientovaná. Jinými slovy hrany samy o sobě tento fakt rozlišovat nepotřebují a nebudou.

Přestože velká část návrhu celého řešení nebude nijak omezena, některé aspekty očekávaného rozhraní je jako obvykle nutné dodržet. V řadě případů budeme navíc potřebovat varianty funkcí pro modifikovatelné, ale i konstantní grafy nebo jejich součásti. Abychom hlavičky takových funkcí nemuseli v dalším textu

neustále zbytečně zdvojovat, jen je označíme pomocí symbolu [♠]. Analogicky použijeme symbol [♣] v situacích, kdy budeme chtít pro konkrétní parametr nabídnout dvě varianty předávání pomocí konstantní lvalue reference resp. modifikovatelné rvalue reference.

Vrcholy a hrany

Každý vrchol i hrana mají přidělený unikátní identifikátor, a to z množiny $\mathbb{N}_0$, tedy přirozených čísel včetně nuly. Tyto identifikátory jsou imutabilní, tedy jakmile jsou jednou přiděleny, nikdy později se už změnit nemohou. Pokud graf obsahuje $n \in \mathbb{N}_0$ vrcholů, platí, že jejich identifikátory pokrývají všechna čísla od 0 do $n - 1$. Jinými slovy tyto identifikátory na sebe souvisle navazují, žádné nepřeskakujeme. Totéž analogicky (odděleně) platí pro hrany. Technicky pro identifikátory použijeme typ `size_t`, v kódu však budeme kvůli usnadnění případných změn v budoucnu pracovat s aliasem `Identifier`. Ať už ale budeme předpokládat jakýkoli typ, vždy bude mít implementované operátory `<<` a `>>` a jeho hodnoty budou implicitně konvertovatelné na nezáporné hodnoty `size_t`.

Pokud jde konkrétně o třídu pro vrchol, je potřeba implementovat následující rozhraní:

- `Identifier getId() const`: vrátí hodnotu identifikátoru daného vrcholu
- `NData& getData()` [♠]: vrátí referenci na datový obsah navázaný na daný vrchol

V případě třídy pro hranu je rozhraní analogické, jen přidáme možnost přistupovat k vrcholům určujícím danou hranu:

- `Identifier getId() const`: vrátí hodnotu identifikátoru dané hrany
- `Identifier getSource() const`: vrátí identifikátor zdrojového vrcholu
- `Identifier getTarget() const`: vrátí identifikátor cílového vrcholu
- `EData& getData()` [♠]: vrátí referenci na datový obsah navázaný na danou hranu

Pro vrcholy i hrany dále naprogramujeme operátory `<<`, pomocí kterých je budeme schopni vypsat do libovolného výstupního streamu. V obou případech je potřeba respektovat následující formát výstupu. Máme-li vrchol s identifikátorem n, serializujeme jej do řetězce `node (`n `{`*data*`})`, kde *data* (tedy řetězec mezi uvedeným párem složených závorek) reprezentuje serializovaný datový obsah. Ať už bude vypadat jakkoli, nikdy nebude obsahovat další takové zanořené složené závorky. Předpokládá se, že každý typ `NData`, který pro datový obsah vrcholů budeme chtít použít, rovněž implementuje příslušný operátor `<<`, a tedy se sám umí tímto způsobem vypsat. Podobně pro hranu s identifikátorem m mezi vrcholy s identifikátory i a j budeme výstup očekávat v podobě `edge (`i`)-[`m `{`*data*`}]->(`j`)`, kde *data* opět tvoří serializovaný obsah instance `EData` získaný příslušným operátorem `<<`. Uvedenou šipku vypíšeme u všech hran bez ohledu na to, jestli jsou, nebo nejsou orientované.

Vypsání jednotlivých vrcholů a hran nebudeme doplňovat koncem řádku. Pokud bychom například měli graf `DirectedGraph<std::string, std::string>`, mohl by obsahovat vrchol `node (1 {one})` nebo hranu `edge (1)-[3 {one-four}]->(4)` mezi vrcholy 1 a 4.

Komponenty grafu

Každý graf musí logicky (ne nutně i fyzicky) z hlediska svého datového obsahu přímo či nepřímo uchovávat (ve smyslu vlastnictví) minimálně následující tři datové položky (a tranzitivně i jejich obsah): 1) vnitřní úložiště pro instance vrcholů, 2) vnitřní úložiště pro instance hran a dále také 3) matici sousednosti (anglicky *adjacency matrix*) pro rychlé nalezení hran na základě znalosti dvojice identifikátorů vrcholů, které by měly propojovat. Ve všech těchto třech případech primárně předpokládáme, že se bude používat naše gumové pole `lib::Array`, postačí ale i obyčejný `std::vector`. Kromě uvedených položek je samozřejmě v případě potřeby možné uchovávat i další potřebné údaje.

Třída grafu zpřístupní komponenty pro vrcholy a hrany následujícím způsobem:

- `Nodes<NData, EData>& nodes()` [♠]: vrátí referenci na komponentu starající se o vrcholy
- `Edges<NData, EData>& edges()` [♠]: analogicky pro hrany

Pokud jde o umístění výše uvedených datových položek, nabízejí se dvě základní řešení: buď je všechny umístíme na jedno místo přímo do třídy grafu `Graph<NData, EData>`, nebo je můžeme rozdělit a vhodně umístit až do tříd obou uvedených komponent `Nodes<NData, EData>` resp. `Edges<NData, EData>`. Obě varianty přitom nabízí určité výhody, mají však i svá úskalí. Dobře si proto rozmyslete, který přístup je vám bližší. V každém případě musí být obě komponenty zachovány, i kdyby to bylo jen kvůli zapouzdření očekávané funkcionality, kterou jednoduše nelze poskytovat jen na jednom místě přímo v grafu jako takovém.

Pokud jde o vnitřní úložiště pro objekty vrcholů, předpokládá se, že všechny vrcholy v tomto kontejneru budou uspořádány přesně v pořadí jejich identifikátorů. Jinými slovy vrchol v poli na pozici n musí mít identifikátor n. Totéž bude analogicky platit i pro vnitřní kontejner objektů hran.

Matici sousednosti můžeme realizovat jen obyčejnou datovou položkou stejně jako pomocí samostatné třídy. Technicky pak může obsahovat ukazatele na příslušné hrany nebo lépe jejich identifikátory. Všechny varianty opět nabízí určité výhody i nevýhody, volbu konkrétního řešení není vhodné podcenit. Tak či onak je potřeba si uvědomit, že náš graf má být dynamický (musí umožňovat přidávání nových vrcholů), takže určitě není dobrý nápad obsah matice linearizovat. Na závěr jen poznamenejme, že pozice (i, j) odpovídá hraně vedoucí z vrcholu s identifikátorem i do vrcholu s identifikátorem j. V případě neorientovaného grafu musí platit, že matice je symetrická podle hlavní diagonály. Smyčky, tedy hrany začínající i končící ve stejném vrcholu, jsou rovněž podporovány.

Serializace grafu

Graf jako celek, myšleno jeho vrcholy a hrany, musíme být schopni vypsat na standardní výstup (nebo i do jiného streamu). Platí, že vždy nejprve vypíšeme všechny vrcholy, a to ve vzestupném pořadí podle jejich identifikátorů, teprve pak všechny hrany, opět vzestupně podle jejich identifikátorů. Vždy jeden vrchol resp. jedna hrana na řádek, každý pak ukončíme pomocí `std::endl`. Podle očekávání bude vrcholy umět vypsat třída `Nodes<NData, EData>`, naopak hrany třída `Edges<NData, EData>`. V obou případech toho dosáhneme pomocí následujícího stejného rozhraní:

- `void print(std::ostream& stream = std::cout) const`: vypíše komponentu vrcholů resp. hran

Následně do třídy grafu jako celku už snadno doplníme následující funkce:

- `void print(std::ostream& stream = std::cout) const`: vypíše celý graf (tedy nejprve všechny vrcholy a následně všechny hrany) do uvedeného streamu
- `void print(const std::string& filename) const`: stejným způsobem vypíše graf do výstupního souboru s uvedeným názvem

Pro obě komponenty stejně jako celý graf dodáme i analogické implementace operátorů `<<`. Serializace ukázkového grafu by pak mohla vypadat např. takto:

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

Třída komponenty hran bude mít pro účely ladění a testování ještě následující funkci, která bude umět na vyžádání vypsat obsah již diskutované matice sousednosti:

- `void printMatrix(std::ostream& stream = std::cout) const`: vypíše obsah matice sousednosti do daného streamu; každý řádek matice vypíšeme na samostatný řádek výstupu; hodnoty (jednotlivé sloupce) v řádku matice pak vypisujeme za sebou a oddělujeme je symbolem `|`; pokud na dané pozici je hrana, vypíšeme její identifikátor; v opačném případě vypíšeme symbol `-`

Serializace matice sousednosti pro předchozí graf (ve variantě neorientovaného grafu) by pak vypadala konkrétně takto:

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

Vkládání vrcholů a hran

Nově zkonstruovaný graf bude vždy prázdný, tedy nebude obsahovat žádný vrchol ani hranu. Ty pak můžeme přidávat jednotlivě pomocí funkcí, na které se teď zaměříme. Komponenta pro vrcholy konkrétně nabídne následující rozhraní:

- `Node<NData>& add(Identifier id, NData [♣] data)`: vloží do grafu nový vrchol s explicitně daným identifikátorem a navázanými daty; v tuto chvíli se předpokládá, že uvedený identifikátor je validní, tedy odpovídá první další dosud nevyužité pozici v interním poli; pro účely testování a obsluhy výjimek se dále předpokládá, že nejprve dojde ke vložení instance nového vrcholu do tohoto pole a teprve následně k úpravám matice sousednosti; vrátí se reference na vytvořený objekt vrcholu
- `Node<NData>& add(NData [♣] data)`: totéž, jen identifikátor nově vkládaného vrcholu implicitně určíme jako další volný

Komponenta pro hrany analogicky nabídne následující funkce:

- `Edge<EData>& add(`
  `Identifier id, Identifier source, Identifier target, EData [♣] data`
  `)`: vloží do grafu novou hranu s explicitně specifikovaným identifikátorem a navázanými daty, a to hranu spojující uvedenou dvojici vrcholů; opět pro tuto chvíli předpokládáme korektnost všech argumentů; vrátí se reference na vytvořený objekt hrany
- `Edge<EData>& add(`
  `Identifier source, Identifier target, EData [♣] data`
  `)`: totéž, jen identifikátor vkládané hrany opět implicitně určíme jako další volný

Pokud by se z nějakého důvodu stalo, že nový vrchol nebo novou hranu není možné do grafu vložit (z důvodu nepodařené alokace paměti v rámci našeho interního kontejneru pole nebo kvůli nevydařenému kopírování instancí navázaných dat), je potřeba takovou situaci korektně ošetřit a docílit atomicity. Tedy že operace vkládání se buď celá podaří, nebo nikoli. V případě dílčího neúspěchu je tedy nutné všechny struktury grafu navrátit do stavu, který bude opět konzistentní.

Import grafu

Pro snadnější vytváření grafů implementujeme i funkce, pomocí kterých budeme moci obsah grafu načíst a importovat ze vstupního souboru (nebo i jiného streamu). V tomto případě platí, že importovat vrcholy a hrany můžeme jen do již vytvořené instance grafu. Také platí, že importovací funkce je možné volat opakovaně, a tedy instanci grafu poskládat např. z více vstupních souborů obsahujících jednotlivé menší části celého grafu, stejně jako je libovolně prokládat již diskutovanými funkcemi na manuální přidávání jednotlivých vrcholů nebo hran.

Funkce pro import definujeme na úrovni třídy pro graf jako celek:

- `void import(std::istream& stream = std::cin)`: importuje vrcholy a hrany z daného vstupního streamu
- `void import(const std::string& filename)`: importuje vrcholy a hrany ze vstupního souboru s uvedeným názvem

Ve vstupních datech mohou být vrcholy a hrany libovolně promíchány. Nemusí tedy platit, že jsou nejprve uvedeny jen vrcholy a pak už jen hrany. Vždy ale platí, že každý řádek obsahuje jen jeden vrchol, nebo jednu hranu. Do grafu je pak vkládáme postupně beze změny jejich pořadí. Současně se také můžeme spolehnout na to, že záznam každého vrcholu i hrany je syntakticky správně formovaný a odpovídá struktuře, kterou jsme popisovali u příslušných funkcí na serializaci.

Pokud jde o datový obsah navázaný na vrcholy resp. hrany, musíme jej načíst z řetězce mezi párem složených závorek, a to výhradně pomocí operátoru `>>`. Jinými slovy předpokládáme, že každý konkrétní datový typ, který v tomto smyslu chceme u vrcholů nebo hran používat, musí implementovat příslušný operátor extrakce ze streamu.

Předpokládá se, že při realizaci importu budeme vnitřně používat již dříve uvedené funkce na přidávání jednotlivých vrcholů a hran. Případné zcela prázdné řádky se přeskočí. Pokud import v průběhu selže, již úspěšně přidané vrcholy nebo hrany v grafu ponecháme. Očekáváme tedy atomické chování jen na úrovni

jednotlivých vrcholů resp. hran. Pro ilustraci uveďme validní obsah souboru, který i přes promíchané pořadí vrcholů a hran odpovídá našemu prvnímu příkladu:

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

Přístup k vrcholům a hranám

Třída pro komponentu vrcholů nabídne následující funkce, pomocí kterých budeme schopni přistupovat ke konkrétním vrcholům nebo se nad nimi dotazovat:

- `size_t size() const`: vrátí aktuální počet vrcholů v grafu
- `bool exists(Identifier id) const`: otestuje existenci vrcholu aneb vrátí `true`, pokud v grafu existuje vrchol s uvedeným identifikátorem, jinak `false`
- `Node<NData>& get(Identifier id)` [♠]: vrátí referenci na vrchol s uvedeným identifikátorem; pro tuto chvíli předpokládejme, že můžeme přistupovat jen k existujícím vrcholům
- `Node<NData>& operator[](size_t id)` [♠]: totéž

Analogicky u třídy pro komponentu hran připravíme následující funkce:

- `size_t size() const`: vrátí aktuální počet hran v grafu
- `bool exists(Identifier id) const`: otestuje existenci dané hrany aneb vrátí `true`, pokud v grafu existuje hrana s uvedeným identifikátorem, jinak `false`
- `bool exists(Identifier source, Identifier target) const`: totéž, jen pro detekci existence hrany mezi uvedenou dvojicí vrcholů určených jejich identifikátory
- `Edge<EData>& get(Identifier id)` [♠]: vrátí referenci na hranu s daným identifikátorem; pro tuto chvíli opět předpokládejme jen validní identifikátor
- `Edge<EData>& get(Identifier source, Identifier target)` [♠]: totéž, jen pro danou dvojici vrcholů podle jejich identifikátorů

Do komponenty hran dále přidáme i operátor `[]`, který nám umožní přistupovat ke konkrétním hranám. Nikoli však jednoúrovňově pomocí identifikátorů takových hran, ale dvouúrovňově pomocí identifikátorů dvojice odpovídajících vrcholů `source` resp. `target` (v tomto pořadí). Chceme tedy umět používat kód ve tvaru `graph.edges()[source][target]` [♠].

Iterátory nad grafem

Komponenty vrcholů a hran dále nabídnou iterátory umožňující iterovat nad jednotlivými vrcholy resp. hranami v grafu. Pochopitelně není nutné implementovat nic nového, stačí totiž jen zpřístupnit iterátory z interního pole, které už máme. Jen připomeňme, že z hlediska jejich schopností očekáváme alespoň úroveň dopředných iterátorů. Konkrétně tedy potřebujeme definovat dvě dvojice funkcí `begin()` [♠] a `end()` [♠] se zřejmým významem.

Manipulace s grafy

Nad třídou grafu je dále potřeba naprogramovat standardní kopírovací a přesouvací (vykrádací) konstruktory a operátory přiřazení, díky nimž budeme schopni manipulovat s grafy jako celky. Pro jistotu dodejme, že každý vrchol i hrana jsou v logickém vlastnictví grafu, do kterého patří. Pokud tedy kopírujeme nebo přesouváme grafy jako celky, musíme kopírovat resp. přesouvat i veškerý jejich obsah. Pamatovat musíme i na zajištění očekávané atomicity. V tomto smyslu nám může pomoci i možnost obalení celého těla funkce (včetně inicializačního seznamu položek v konstruktoru) pomocí try/catch bloků.

Přesouvací konstruktor a operátor přiřazení musí garantovat zachování referencí na všechny jednotlivé vrcholy i hrany, nikoli však na komponenty jako takové. Vykradené vzorové grafy ponecháme prázdné (bez

jakýchkoli vrcholů a hran) a vnitřně konzistentní (aby je pro účely testování bylo stále možné korektně používat). Kopírovací operátor přiřazení ponechá v případě jakéhokoli selhání cílový graf prázdný a opět vnitřně konzistentní (dosažení atomicity na úrovni *Strong Exception Guarantee* tedy nechceme). Jelikož budeme z uvedených důvodů pro graf tak jako tak potřebovat vyprazdňovací funkci, rovnou ji přidáme i do veřejného rozhraní celého grafu v podobě `void clear()`.

Chybové situace

Doteď jsme u většiny funkcí předpokládali, že jejich parametry byly validní a ani nic jiného se nemůže pokazit. V této části popíšeme, jak všechny dotčené části kódu upravíme takovým způsobem, abychom mohli většinu krajních a chybových situací detekovat a také korektně ošetřit.

Za tímto účelem vytvoříme jednoduchou hierarchii vlastních výjimek. Abstraktním předkem bude naše vlastní třída `Exception`. Poskytne jedinou funkci, a to `const char* message() const`. Jejím účelem bude vrátit konkrétní textový popis vzniklé chyby. Z praktických důvodů budeme potřebovat rozlišit dva druhy odvozených výjimek, a to pro výjimky s fixními textovými zprávami (známými v době kompilace) a variabilními zprávami (dynamicky poskládanými a zkonstruovanými až za běhu).

V první kategorii bude jediná odvozená třída, a to `MemoryException`. Důvodem je, že bude vyvolávána při nedostatku paměti pro dynamickou alokaci. V takovém okamžiku samozřejmě není možné vyhazovat takové výjimky, které by ji samy potřebovaly. Ve druhé kategorii pak budou všechny ostatní odvozené typy výjimek, konkrétně `IdentifierException`, `ElementException`, `ConflictException` a `FileException`.

Následující přehled obsahuje konkrétní ošetřované situace, jejich pořadí, očekávané typy výjimek, stejně jako očekávané zprávy. Ty pak obsahují zástupné symboly ve tvaru `$něco`, které jen nahradíme za příslušnou konkrétní hodnotu:

- Přístup ke konkrétnímu vrcholu pomocí funkce `get` nebo operátoru `[]` na základě jeho identifikátoru:
  - `ElementException("Node with identifier $id does not exist");`
    pokud se přistupuje k vrcholu, který neexistuje
- Přístup ke konkrétní hraně pomocí funkce `get` na základě jejího identifikátoru:
  - `ElementException("Edge with identifier $id does not exist");`
    pokud se přistupuje k hraně, která neexistuje
- Přístup ke konkrétní hraně pomocí funkce `get` nebo operátoru `[]` na základě dvojice vrcholů:
  - `ElementException("Source node with identifier $source does not exist");`
    pokud zdrojový vrchol neexistuje
  - `ElementException("Target node with identifier $target does not exist");`
    pokud cílový vrchol neexistuje
  - `ElementException("Edge between nodes $source and $target does not exist");`
    pokud se přistupuje k neexistující hraně mezi jinak validní dvojicí vrcholů
- Test existence hrany pomocí funkce `exists` na základě dvojice vrcholů:
  - `ElementException("Source node with identifier $source does not exist");`
    pokud zdrojový vrchol neexistuje
  - `ElementException("Target node with identifier $target does not exist");`
    pokud cílový vrchol neexistuje
- Přidání nového vrcholu pomocí funkce `add`:
  - `IdentifierException("Invalid node identifier $id requested");`
    pokud je použit neplatný identifikátor (identifikátor s nejvyšší možnou hodnotou typu `size_t`)
  - `ConflictException("Node with identifier $id already exists");`
    pokud je použit identifikátor již existujícího vrcholu
  - `IdentifierException("Non-successive node identifier $id requested");`
    pokud je použit nenavazující identifikátor
  - Původní výjimka v případě, že selže vytváření kopie navázaných dat

  - `MemoryException("Unavailable memory for a new node in the nodes container");`
    pokud není možné vložit objekt vrcholu do interního pole vrcholů
  - `MemoryException("Unavailable memory for the adjacency matrix extension");`
    pokud není možné do matice sousednosti přidat nový sloupec a řádek

- Přidání nové hrany pomocí funkce `add`:
  - `IdentifierException("Invalid edge identifier $id requested");`
    pokud je použit neplatný identifikátor (identifikátor s nejvyšší možnou hodnotou typu `size_t`)
  - `ConflictException("Edge with identifier $id already exists");`
    pokud hrana s uvedeným identifikátorem již existuje
  - `IdentifierException("Non-successive edge identifier $id requested");`
    pokud je použit nenavazující identifikátor
  - `ElementException("Source node with identifier $source does not exist");`
    pokud zdrojový vrchol neexistuje
  - `ElementException("Target node with identifier $target does not exist");`
    pokud cílový vrchol neexistuje
  - `ConflictException("Edge between nodes $source and $target already exists");`
    pokud už mezi danou dvojicí vrcholů hrana existuje
  - Původní výjimka v případě, že selže vytváření kopie navázaných dat
  - `MemoryException("Unavailable memory for a new edge in the edges container");`
    pokud není možné vložit objekt hrany do interního pole hran

- Vypsání grafu do uvedeného souboru pomocí funkce `print`:
  - `FileException("Unable to open output file $filename");`
    pokud není možné otevřít daný výstupní soubor

- Import grafu z uvedeného souboru nebo streamu pomocí funkce `import`:
  - `FileException("Unable to open input file $filename");`
    pokud není možné otevřít daný vstupní soubor (pouze pro první variantu)
  - Výjimky vyhazované funkcemi na přidávání jednotlivých vrcholů a hran

- Kopírovací konstruktor grafu:
  - `MemoryException("Unavailable memory for constructing a source graph copy");`
    pokud nebylo možné vytvořit kopii grafu kvůli nedostatku paměti
  - Původní výjimka v případě, že selže vytváření kopie dat navázaných na vrcholy nebo hrany

- Kopírovací operátor přiřazení grafu:
  - `MemoryException("Unavailable memory for assigning a source graph copy");`
    pokud nebylo možné vytvořit kopii grafu kvůli nedostatku paměti
  - Původní výjimka v případě, že selže vytváření kopie dat navázaných na vrcholy nebo hrany

Závěrečné pokyny

Odevzdejte všechny vytvořené zdrojové soubory (patrně jen `Graph.h` a `Array.h`) kromě souboru `Main.cpp`, který již součástí připraveného testu je. Obsahuje jedinou direktivu `#include <Graph.h>` a dále funkci `main`, která bude průběh testu jako obvykle řídit.

Cílem úkolu je ukázat schopnost práce s konstrukty, se kterými jsme se od začátku semestru setkali. Kromě základních dovedností tedy jde zejména o práci s textovými soubory, streamy, návrh tříd, použití konstruktorů a destruktorů, dědičnost, virtuální metody, dynamickou alokaci, šablony, ukazatele, operátory, iterátory nebo výjimky.

Předložená implementace pochopitelně musí být korektní a stabilní, kompilace musí proběhnout bez jakýchkoli varování. Hodnotit se ale bude i celková kvalita kódu. Tedy zejména avšak ne výlučně organizace kódu do jednotlivých souborů, tříd a funkcí, použití hlavičkových souborů, pojmenovávání souborů, funkcí nebo i proměnných, celkový vizuální styl kódu a indentace, předávání parametrů hodnotou nebo referencí, kvalita návrhu tříd a použití dědičnosti a virtuálních metod, zbytečné neopakování stejných fragmentů

kódu, používání pojmenovaných konstant, ošetřování chybových situací stejně jako využívání standardních knihoven, kontejnerů nebo funkcí.