Inlämningsuppgift 2

Inledning

I denna uppgift ska du implementera en **Graf** vilken ska representeras med **närhetslistor** (dvs. ”adjacency lists”). För närhetslistorna ska listan som du implementerade i första inlämningsuppgiften användas. Vidare ska du bestämma ett **minimalt uppspännande träd** (minimun spanning tree) för grafer.

En grafs noder representeras i detta fall av heltalen 0, 1, 2, … n-1 om grafen innehåller n noder.

Förberedelser

Läs om graf (Graph) och grafrepresentation i avsnitt 9.1 samt minimala uppspännande träd (”minimum spanning trees”) i avsnitt 9.5 i kursboken.

Delta på föreläsningarna som behandlar dessa delar, dvs. i synnerhet föreläsningen på onsdag vecka 16 samt måndag vecka 17. Måndag kl. 15.15 den 23 april i Multisalen kommer vi att gå igenom uppgiften och svara på eventuella frågor.

Implementation

Du ska utgå från stommen till klassen Graph nedan och definiera alla medlemsfunktioner för denna inklusive konstruktor och destruktor. Kopieringskonstruktorn och tilldelningsoperatorn ska inte definieras. Vill du trots allt göra det är det tillåtet, men du behöver då ta bort *”= delete”* från källkoden i header-filen. Vidare är det tillåtet att lägga till privata hjälpfunktioner men inte några ytterligare medlemsfunktioner.

```
enum GraphType {DIRECTED, UNDIRECTED}; // used for member variable in the Graph

class Graph
{
private:
	GraphType						graphType;

	// for you to decide: member variables and helper functions
	// but you must use the List-class from the first
	// assignment for the adjacency lists that will
	// be used for representation of the Graph
public:
	Graph(GraphType graphType = DIRECTED, int nrOfVert = 0);
	Graph(const Graph& origin) = delete;
	Graph& operator=(const Graph& origin) = delete;
	virtual ~Graph();
	void reset(int nrOfVert, GraphType graphType);
	bool isDirected() const;
	bool addArc(int sourceVertex, int destinationVertex, int arcWeight);
	bool hasArc(int sourceVertex, int destinationVertex) const;
	bool removeArc(int sourceVertex, int destinationVertex, int arcWeight);

	int getNrOfVertices() const;
	int outDegreeOfVertex(int theVertex) const;
	int inDegreeOfVertex(int theVertex) const;

	List<int> getAllVerticesAdjacentTo(int theVertex) const;
	void minSpanTree(List<AdjInfo> minSpanTree[], int cap, int &totalCost) const;
```

```
            void printGraph() const;
};
```

Klassen **AdjacencyInfo** (som finns i filerna AdjacencyInfo.h resp AdjacencyInfo.cpp) är definierad enligt:

```
class AdjacencyInfo
{
private:
            int neighbourVertex;
            int arcWeight;
public:
            AdjacencyInfo(int destination = -1, int arcWeight = 0);
            virtual ~ AdjacencyInfo ();
            int getNeighbourVertex() const;
            void setNeighbourVertex(int neighbourVertex);
            int getArcWeight() const;
            void setArcWeight(int arcWeight);

            bool operator==(const AdjacencyInfo & other) const;
            bool     operator!=(const     AdjacencyInfo     &     other)     const;

            //it is ok to add more comparison operators
};
```

Objekt av denna klasstyp ska skapas och placeras i de närhetslistor som grafen ska vara uppbyggd av.

Du ska använda din egen implementation av klassmallen List som du implementerat i inlämningsuppgift 1 för

- att representera närhetslistor (adjacency lists)
- övriga list-objekt (ex-vis för parametrar och returdatatyper som är av datatypen List)

Medlemsfunktioner i klassen Graph

- **reset** – återställer grafen så att den representerar en graf med *nrOfVert* antal noder men inga bågar
- **isDirected** – returnerar sant eller falskt beroende på om grafen är riktad eller ej
- **addArc** – lägger till en båge från *sourceVertex* till *destinationVertex* och om grafen är oriktad även från *destinationVertex* till *sourceVertex.* Om *sourceVertex* eller *destinationVertex* inte motsvarar en nod i grafen returneras falskt och annars sant
- **hasArc** – returnerar sant eller falskt beroende på om det finns en båge eller ej från *sourceVertex* till *destVertex*
- **removeArc**- tar bort bågen theArc som går från *sourceVertex* till *destinationVertex* och om grafen är oriktad även bågen som går från *destinationVertex* till *sourceVertex.* Sant eller falskt returneras beroende på om det fanns en båge som togs bort eller ej
- **getNrOfVertices** – returnerar antalet noder
- **outDegreeOfVertex** – returnerar utgraden (out degree) för noden *theVertex*
- **inDegreeOfVertex** - returnerar ingraden (in degree) för noden *theVertex*
- **getAllVerticesAdjacentTo** – returnerar en lista innehållande de noder som är grannar till *theVertex*

- **minSpanTree** – bestämmer ett minimalt uppspännande träd (minimum spanning tree) för grafen vilket därefter finns representerat genom närhetslistor i parametern *minSpanTree*. Parametern *totalCost* ska innehålla den totala kostnaden för det minimala uppspännande trädet. Kapaciteten på arrayen finns i *cap*. Om kapaciteten inte är tillräcklig eller om grafen är riktad kastas undantag (t.ex som en teckenföljd).
- **printGraph** – ger en utskrift av grafen så att det framgår hur grafen är uppbyggd. Både noder och bågar ska skrivas ut

Din implementation av klassen Graph får inte ge upphov till några minnesläckor.

Du ska dessutom implementera en fil innehållande main-funktionen i vilken det åtminstone finns funktioner som löser följande delproblem:

- läser uppgifter för en graf från en textfil (innehållet beskrivs längre fram i uppgiften)
- testar och visar resultat av anrop av alla medlemsfunktioner för Graf-objekt förutom medlemsfunktionen minSpanTree
- bestämmer ett minimalt uppspännande träd för grafen (om den är oriktad) och presenterar (skriver ut) detta på ett sådant sätt att det framgår vilka noder som är sammanbundna samt vikten på den båge som förbinder noderna. Vidare ska den totala kostnaden för det minimala uppspännande trädet presenteras.

Funktionerna ovan ska anropas från main-funktionen och ska vara fungerande för flera grafer så länge innehållet på textfilerna följer den bestämda uppbyggnaden.

Skriftlig rapport

Rapporten ska ha strukturen av en teknisk rapport, som ni lärt er skriva i tidigare kurser. Den kan vara på engelska eller svenska, men säkerställ att den är korrekturläst och att ni har använt relevant program för stavnings- och grammatikkontroll innan ni lämnar in rapporten. Utöver ert namn, studentakronym och datum ska rapporten innehålla följande:

- En redogörelse för hur du hanterade närhetslistorna i din lösning och varför, med hänvisning till kurslitteraturen och föreläsningarna.
- En diskussion om tidskomplexitet och ange tidskomplexiteten för alla medlemsfunktioner i Graph-klassen förutom funktionen minSpannTree.
- En beskrivning av din algoritm (i form av pseudokod) och de datastrukturer du använt i din lösning för att bestämma ett minimalt uppspännande träd.
- En bilaga i rapporten där du klistrar in lösningen (källkoden) till funktionen minSpannTree.

Inlämning

Du ska dels lämna in den skriftliga rapporten enligt ovan (antingen som ett pdf-dokument eller ett worddokument). Du ska även lämna in alla h-filer och cpp filer som du konstruerat och som krävs för din lösning av uppgiften, dvs. minst:

- Graph.h
- Graph.cpp
- AdjacencyInfo.h

- AdjacencyInfo.cpp
- Filen innehållande main-funktionen

Lämna inte in en zip/rar-fil utan samtliga filerna ska lämnas in som separata filer via Canvas.

Regler för inlämningsuppgiften

Inlämningsuppgifterna **ska göras individuellt**. Det ska vara egen implementering i C++. Det är **inte tillåtet** att enbart modifiera eller anpassa kod hämtad från internet eller från annan person eller annan källa. Om otillåten kopiering eller annan form av fusk misstänks, är läraren skyldig att anmäla detta till högskolans disciplinnämnd.

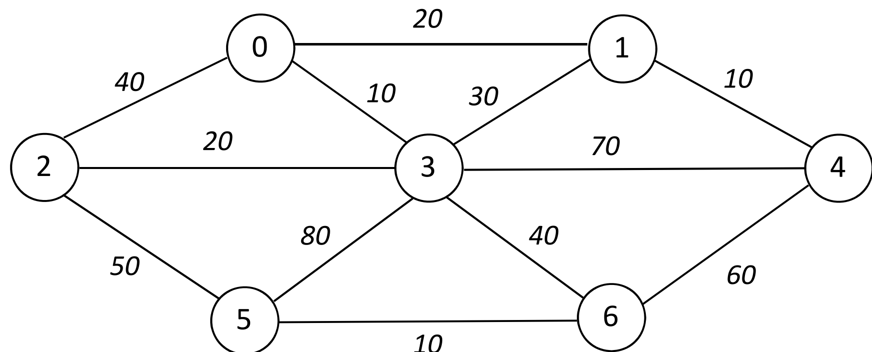
Grafer och filinnehåll

Innehållet på textfilerna ska vara enligt

Rad 1: Antalet noder som ingår i grafen
Rad 2: U eller D beroende på om grafen är oriktad (U) eller riktad (D)
Övriga rader: **nod1 nod2 vikt** där **nod1** är den nod från vilken bågen utgår och **nod2** den nod bågen ansluter till och **vikt** är vikten på bågen

Nedan är ett exempel på filinnehåll för den uppritade grafen.

```
7
U
0 1 20
0 3 10
0 2 40
3 2 20
5 2 50
5 3 80
6 5 10
3 6 40
4 3 70
3 1 30
1 4 10
4 6 60
```



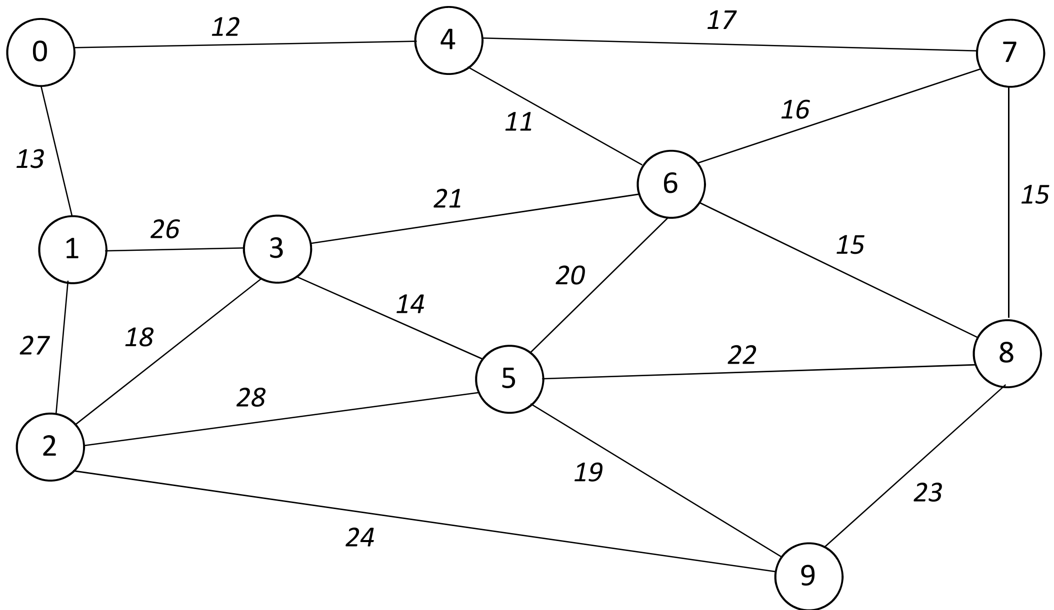
Tips:

Anta att **in** är en inström för textfil i vilken det finns 3 heltal på en och samma rad samt att **nr1**, **nr2** och **nr3** är heltalsvariabler. Då kan läsning från textfilen genomföras enligt

```
in >> nr1 >> nr2 >> nr3;
```

Exempel på grafer

Oriktad graf:



Riktad graf: