Complexity Notation and Basic Data Structures

Practical Exercises

Gonçalo Leão

with Contributions from Prof. Liliana Ferreira and Prof. Pedro Diniz

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

A - Graphs

Exercise 1
The following exercises, use the **IntroGraph** class provided in the *exercises.h* file. **IntroGraph** derives from the **Graph** base class (declared in *data_structures/Graph.h*), which implements a graph's basic functionalities and includes all the necessary attributes for the **Vertex**, **Edge** and **Graph**.

All exercises can be solved without modifying the **Graph**, **Vertex** and **Edge** classes and by only adding auxiliary methods to the **IntroGraph** class. No class attribute needs to be created for any class.

The base **Graph** class represents the graph as an adjacency list and already has methods to add vertices and edges to the graph. In this exercise, methods will be created to remove edges and vertices from the derived **IntroGraph** class (as you should not edit the **Graph** class, as explained in the setup tutorial).

In this exercise you are asked to:

a) Write in pseudo-code an algorithm to remove an edge from a graph, given the unique identifiers of its source and destination vertices.
b) Indicate and justify the temporal complexity of the proposed algorithm, in the worst case, with respect to V and E, which denote, respectively, the graph's number of vertices and edges.
c) In the **IntroGraph** class, implement *removeEdge*, which corresponds to the proposed algorithm.

```
bool removeEdge(const int &source, const int &dest)
```

This function removes the edge originating at vertex **source** and ending at vertex **dest**. It returns **true** if the edge was removed successfully and **false** otherwise (edge does not exist).

<u>**Suggestion**</u>: Use the *findVertex* method from the **Graph** base class and the *removeEdge* method from the **Vertex** class. This latter method removes the edge going to a specific vertex and returns **true** if and only if such an edge to be removed exists.

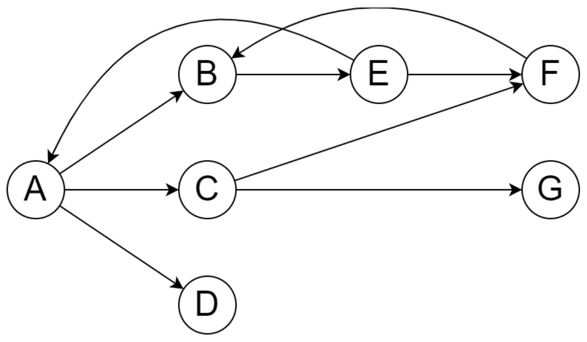
d) Write in pseudo-code an algorithm to remove a vertex from a graph, given its unique identifier.
e) Indicate and justify the temporal complexity of the proposed algorithm, in the worst case, with respect to V and E, which denote, respectively, the graph's number of vertices and edges.
f) In the **IntroGraph** class, implement *removeVertex*, which corresponds to the proposed algorithm.

```
bool removeVertex(const int &id)
```

This function removes the vertex with the identifier **id**. It returns **true** if the vertex was removed successfully and **false** otherwise (the vertex does not exist). Removing a vertex implies removing all edges incident on that vertex or emanating from it.

Exercise 2
Given the directed graph depicted below:

a) Indicate an order that the vertices are visited in using Depth-First Search (DFS) in the graph below, when starting at vertex A.



b) In the **IntroGraph** class, implement the member function below.

```
vector<int> dfs(const int & source) const
```

This function returns a vector containing the graph elements (in this case the identifiers of the vertices) when a Depth-First Search (DFS) is performed on the graph, starting at the vertex whose identifier is the **source**. To avoid ambiguous solutions, if there are multiple child vertices to explore, select the first unexplored vertex as it appears in the vertex's adjacency list.

Suggestion: Use the *visited* attribute (and associated getter and setter methods) from the **Vertex** class.

Exercise 3
Given the directed graph depicted in Exercise 2 above:

a) Indicate an order that the vertices are visited in using Breadth-First Search (BFS) when starting at vertex A.
b) In the **IntroGraph** class, implement the member function below:

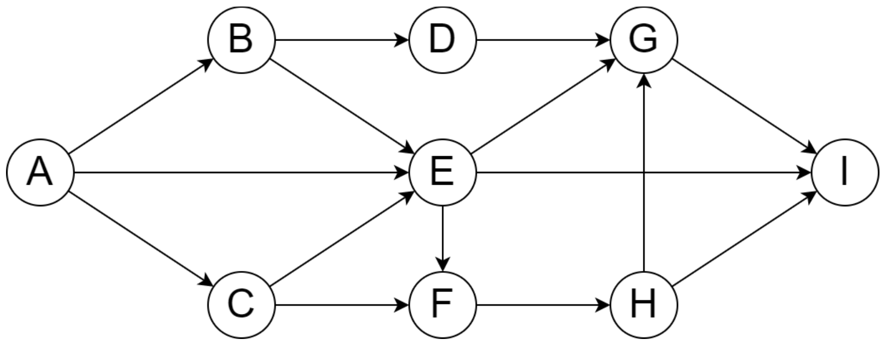
```
vector<int> bfs(const int & source) const
```

This function returns a vector containing the graph elements (in this case the vertex identifiers) when a Breadth-First Search (BFS) is performed on the graph, starting at vertex whose identifier is the **source**. To avoid ambiguous solutions, if there are multiple child vertices to explore, select the first unexplored vertex as it appears in the vertex's adjacency list.

Suggestion: Use the *visited* attribute (and associated getter and setter methods) from the **Vertex** class.

Exercise 4
Given the directed and acyclic graph (DAG) below:

a) Indicate two different topological sorts of its vertices.


$$\begin{tikzcd}
 & B \arrow[r] \arrow[dr] & D \arrow[r] & G \arrow[dr] & \\
A \arrow[ur] \arrow[r] \arrow[dr] & & E \arrow[r] \arrow[ur] \arrow[d] & & I \\
 & C \arrow[ur] \arrow[r] & F \arrow[r] & H \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

b) In the **IntroGraph** class, implement the member function below:

```
vector<int> topsort() const
```

"

This function returns a vector containing the elements of the graph (in this case the identifiers of the vertices) ordered topologically. When a topological sort is not possible for the graph in question, that is, when it is not a DAG, the vector to be returned will be empty.

Suggestion: Use the *indegree* attribute (and associated getter and setter) from the **Vertex** class.

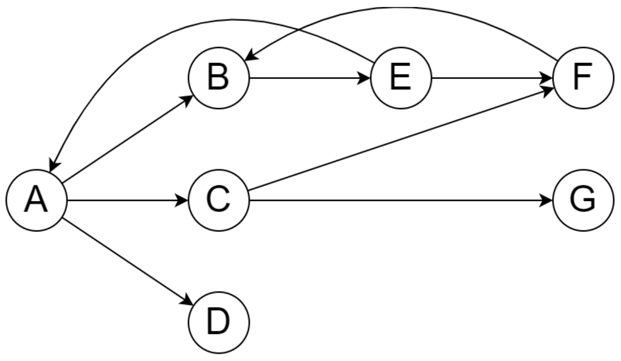
Exercise 5
For the **IntroGraph** class, implement the function below.

```
bool isDAG() const
```

This function determines whether a directed graph is DAG, that is, it does not contain cycles.

Suggestions:
- Adapt the Depth-First Search algorithm.
- Use the *visited* and *processing* attributes (and associated getters and setters) from the **Vertex** class.

Exercise 6
Consider directed graph G depicted below.



For this graph G:

a) Compute G's Strongly Connected Components (SCCs). An SCC is a subset of a directed graph's vertices, where, from any vertex, it is possible to reach any other vertex.
b) In the **IntroGraph** class, implement a member function that computes a directed graph's SCCs.

```
std::vector<std::vector<int>> computeSCCs() const
```

Suggestion: Create new auxiliary member functions for the **IntroGraph** class, based on the Depth-First Search (DFS) algorithm.

Exercise 7

Considering directed graphs:

a) Draw a directed graph with 3 vertices, at least 3 edges and (i) 1 SCC; (ii) 2 SCCs; (iii) 3 SCCs.
b) What happens to SCC's when you add an edge to a directed graph?
c) Show an example when adding an edge does not change the SCC's, and one where it does.

B - Heaps

Exercise 8

Consider the min-heap data structure using a binary tree. For this data structure:

a) Show the organization of this min-heap before and after the insertion of the numbers 6 and 1, regarding the following sequence of insertion 10, 7, 15, 17, 12, 20, 6, 32, 1. Then, consider the insertion order 1, 32, 6, 20, 12, 17, 15, 7, 10. What do you conclude?
b) Implement the function below:

```
std::vector<int> heapSort(std::vector<int> v)
```

This function sorts the provided vector in increasing order by using a heap as an auxiliary data structure.

c) Indicate and justify the temporal complexity of *heapSort*, with respect to n, which denotes the number of elements of the vector to sort.

Suggestion: Use the **Heap** class provided in the *data_structures* directory. No modifications are needed in this class.

Exercise 9

Implement an algorithm that computes the kth smallest element in a set of n positive integers in O(n+klgn) time, with the aid of a heap. If there are less than k elements in the set, then the algorithm should return -1.

```
int kthSmallest(unsigned int k, std::vector<int> v)
```

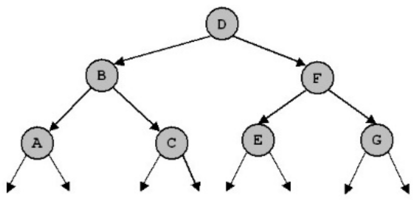
Justify why the algorithm has the desired complexity.

Exercise 10
What are the minimum and maximum number of elements in a heap of height h? Prove that these two values are correct by induction. **Note**: the height of a heap is the number of edges on the longest root-to-leaf path.

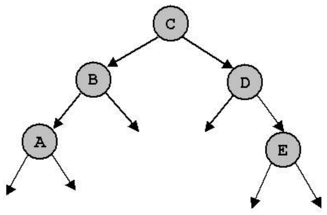
Exercise 11
In a min-heap, where might the largest element be, assuming that all elements are distinct?

C - Trees

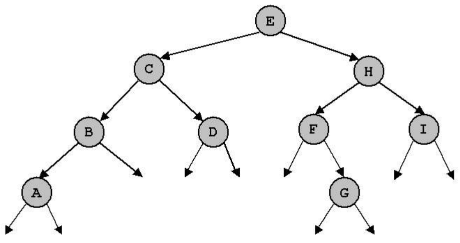
Exercise 12
Give the preorder, inorder, postorder and level order traversals of the following binary trees.



(a)



(b)



(c)