CSCI2100 Data Structures

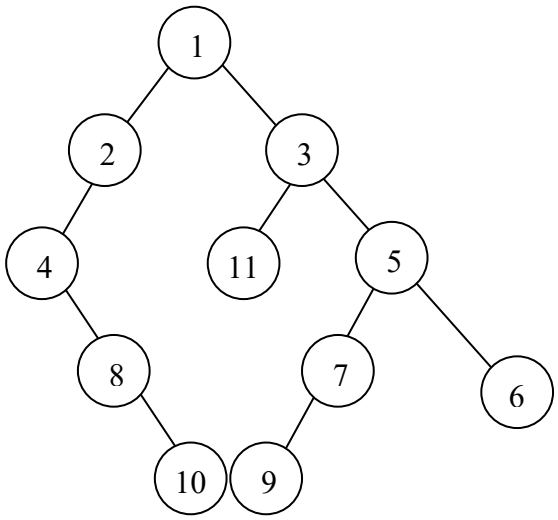
Assignment 4
Due Date: 3 May 2022

Written Exercises

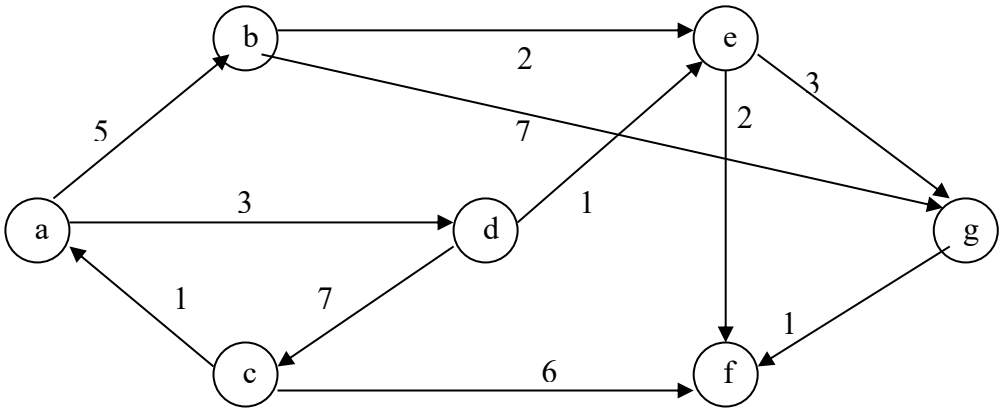
1. Perform the following sequence of **union** operations, using Union-by-height. Show clearly the final resultant disjoint-set forest as well as each step.

   - union(1,2)
   - union(2,3)
   - union(4,5)
   - union(7,8)
   - union(8,9)
   - union(9,10)
   - union(1,6)

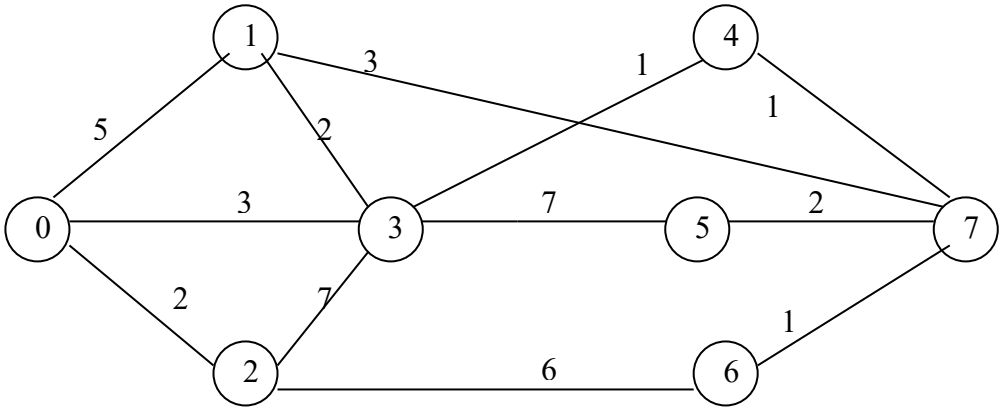
2. Consider the tree below. What is the sequence of nodes visited if
   a) Pre-order traversal;
   b) In-order traversal;
   c) Post-order traversal
   is performed?



3. Use **Dijkstra's algorithm** to find the shortest path from node a to every other node in the following graph. Show each step clearly.



4. Use both Kruskal's algorithm and Prim's algorithm to find a minimum spanning tree of the following graph. Show each step clearly.



Programming Exercises

There are two parts in the programming exercises in this homework assignment.

In the lecture, we briefly introduce the concrete implementation 1.0a for digraphs with weights associated with edges, using an adjacency matrix A. The element A[i,j] stores the weight of the directed arc (i,j), or INFINITY (defined in <math.h>) if the arc (i,j) does not exist. Actually, we can use the same idea to implement undirected graphs with weights associated with edges using an adjacency matrix. To do this, A[i,j] and A[j,i] always store the same value for all nodes i and j. This is because the undirected arc (i,j) is just the same as the undirected arc (j,i).

PART I

In the first part of the programming exercise, you shall complete a concrete implementation of undirected graphs with weights associated with edges using an adjacency matrix. This time we use a refined header file as follows:

```
/* graph.h */
#include "list.h"
#include <stdbool.h>
#include <math.h>

typedef struct GraphCDT * GraphADT;
typedef int Node;

GraphADT EmptyGraph(void);              /* returns an empty undirected graph */
bool GraphIsEmpty(GraphADT);            /* returns whether graph is empty */

void AddNode(GraphADT, Node);           /* add a node */
void DeleteNode(GraphADT, Node);        /* delete a node */
bool NodeExists(GraphADT, Node);        /* returns whether the node exists */
listADT AllNodes(GraphADT);             /* returns a list of all nodes in the graph */

void AddArc(GraphADT, Node, Node, float);
                                        /* add an arc */
                                        /* note that the last argument is the weight */
void DeleteArc(GraphADT, Node, Node);   /* delete an arc */
float ArcWeight(GraphADT, Node, Node);  /* returns the weight of an arc, possibly INFINITY */
bool ArcExists(GraphADT, Node, Node);   /* returns whether the arc exists */

listADT AdjList(GraphADT, Node);        /* returns a list of adjacent nodes of a node */

void PrintAllNodes(GraphADT);           /* prints all existing nodes */
void PrintAllArcs(GraphADT);            /* prints all existing arcs */

int ArctoIndex(GraphADT, Node, Node);   /* see below */
void IndextoArcNodes(GraphADT, int, Node*, Node*);
                                        /* see below */
```

The first a few lines of a straightforward implementation are shown as follows:

```
/* graph.c */

#include "graph.h"
#include "list.h"
#include <stdbool.h>
#include <math.h>
#define MAX_N 100 /* Possible nodes are node 0, node 1, ..., node 99. */

struct GraphCDT {
        float A[MAX_N][MAX_N];
        bool NodeExists[MAX_N];
};
```

However, we note that this straightforward implementation is not efficient in terms of space—almost half of the space used by the array A stores redundant information (A[i,j] is always the same as A[j,i]), and is wasted.

A better implementation is to implement the adjacency matrix using a one-dimensional array W, such that the weight of the undirected arc (i,j), $0 \leq i \leq j < MAX\_N$, is stored in W[(j-i) + i*MAX_N - i*(i-1)/2] (**why?**), or W[j + i*MAX_N - i*(i+1)/2]. Hence, we only need an array W of size MAX_N*(MAX_N+1)/2 to store all the weights.

```
/* graph.c -- more space efficient version */

#include "graph.h"
#include "list.h"
#include <stdbool.h>
#include <math.h>
#define MAX_N 100 /* Possible nodes are node 0, node 1, ..., node 99. */

struct GraphCDT {
        float W[MAX_N*(MAX_N+1)/2];
        bool NodeExists[MAX_N];
};
```

Whenever we want to access A[i,j] in the straightforward version, we now instead access W[x], where x = j + i*MAX_N - i*(i+1)/2. There is an additional advantage of this approach: we can use the number x as the unique index of the arc (i,j).

```
int ArctoIndex(GraphADT G, Node i, Node j) {
        if (!ArcExists(G, i, j)) exit(EXIT_FAILURE);
        return j + i*MAX_N - i*(i+1)/2;
}
```

Therefore, the weight of arc (i,j) of graph G is stored in W[ArctoIndex(G,i,j)]. Note that if x is the index of the arc (i,j), then the following function returns the nodes given the index x.

```
#include <stdlib.h>

void IndextoArcNodes(GraphADT G, int x, Node* i, Node* j) {
        int m = 1; int n;
        int x1; while (x >= (x1 = m * MAX_N - m * (m - 1) / 2)) m++;
        m--; n = MAX_N + x - x1;
        (*i) = m; (*j) = n;
        /* The arc is (m,n), or ((*i),(*j))--whether the arc exists is not checked! */
}
```

Your task now is to complete the rest of this more space-efficient implementation, using the struct GraphCDT with the float array W and the bool array NodeExists as the only fields.

To test the implementation, write an **application program** that makes use of the Graph ADT. The program first reads data about a graph from the data file **GraphData.dat**. Each line starts with a character '+' or '-' (possibly with some preceding spaces). Suppose m and n are two integers, and w a float number, then a line

+ m

indicates that a node m should be added. On the other hand, a line

- m

indicates that a node m should be deleted. Note that when a node is deleted, all the arcs connected to it are all deleted.

The line

+ m n w

indicates that an undirected arc (m, n) with weight w should be added. On the other hand, a line

- m n

indicates that an undirected arc (m, n) should be deleted. Note that the nodes m and n are not deleted.

If a line does not start with '+' or '-', then the line will be ignored.

The following shows a possible data file for graph construction.

```
+  0
+  1
+ 0 1 5
+ 2
  +    3
+ 0 2 2
 + 4
+ 5
 -   5
+  7
+ 0 3    3
+ 3 1 2
  - 1 3
+ 3 4       1
  - 3
+ 2  5 2
+ 5  7 2
    + 4 7 1
```

PART II

[IMPORTANT: This function should be written as an application program. It should not be written in the ADT implementation file graph.c.]

Next, write a function mstP that finds the minimum spanning tree of a graph using Prim's algorithm, and returns the minimum spanning tree as a graph:

GraphADT mstP(GraphADT);

For simplicity, you may assume that the graph in the argument is a connected graph.

Hints:

You may consider to follow the following pseudocodes:

1. Use a float array *K* and a node array *N*. Intuitively, *K*[*v*] stores the lowest weight known so far among all arcs connecting *v* to the minimum spanning tree—and *N*[*v*] will store *w* if the corresponding arc is (*v*,*w*). Hence we initialize *K*[*v*] to INFINITY, and *N*[*v*] to -1 (note that -1 intuitively means 'none').

2. Store all nodes in a set *Q*.

3. Repeat until *Q* is empty:
    a. Find in *Q* the node *v* with the minimum value of *K*[*v*].
    b. Remove *v* from *Q* and add *v* to the minimum spanning tree being constructed.
    c. For each adjacent node *w* of *v*, such that *w* is still in *Q* (**how do we know?**), update *K*[*w*]: if the weight of (*v*,*w*) is less than *K*[*w*],
        i. Set *K*[*w*] = weight of (*v*,*w*) (because *v* is in the minimum spanning tree)
        ii. Set *N*[*w*] = *v* (the corresponding arc is (*v*,*w*))

Note
1. In step 2, the set *Q* can actually be a queue, a list, a stack, or even simply an array.
2. In step 3a, if there are more than one node with the minimum value of *K*[*v*], then any one of them can be chosen as *v*. In particular, as initially *K*[*v*] is INFINITY for all nodes, an arbitrary node will be chosen as the first node to remove from *Q*.

Last but not least, write a simple main program to test it.

—End of Assignment—