CSCI2100 Data Structures

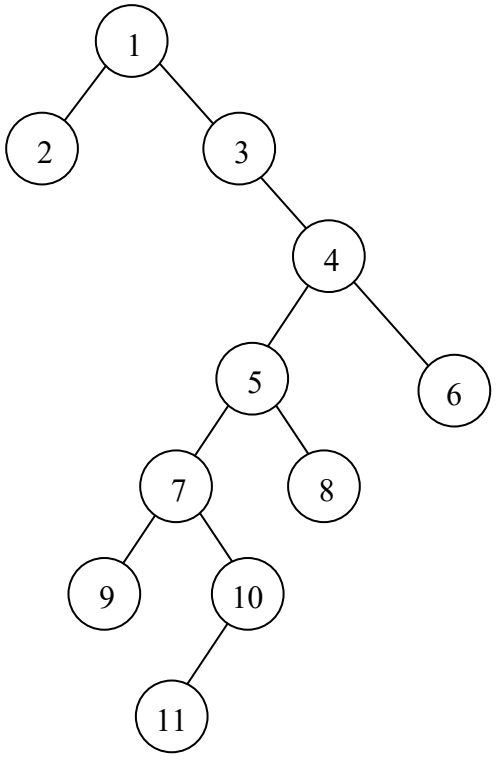
Assignment 4
Due Date: 12 May 2020

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(3,8)
   - 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?

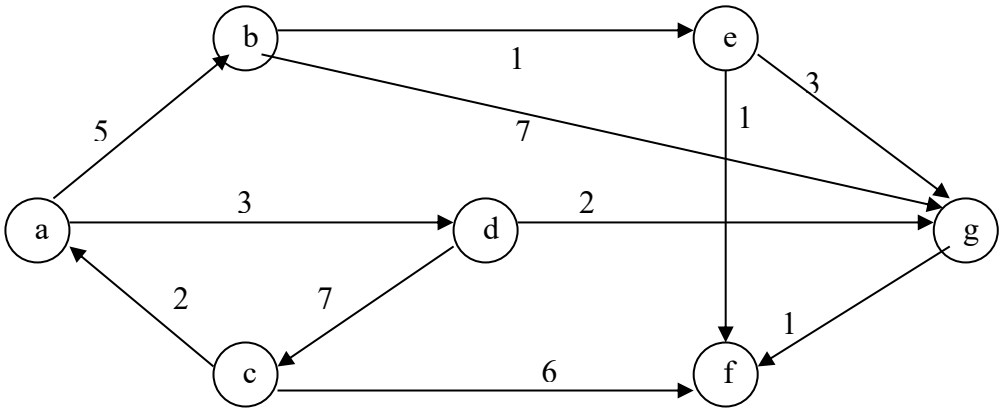


3. Show the resultant red-black tree when nodes with the following keys are inserted, in the following sequence:

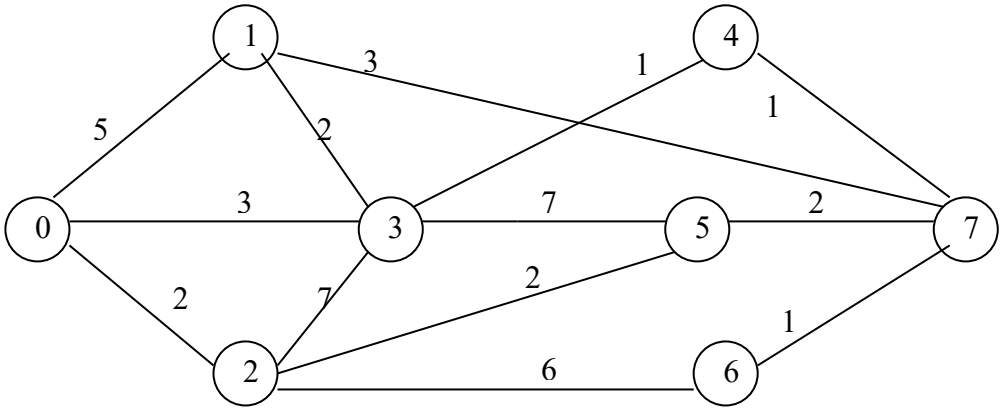
   5, 6, 3, 9, 12, 1, 8, 7, 2

   Show clearly each step.

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



5. Use **Kruskal'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 (page 34 of LL26). 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>

typedef struct GraphCDT * GraphADT;
typedef struct ArcCDT * ArcADT;
typedef int Node;

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

bool AddNode(GraphADT, Node);           /* returns whether the operation succeeds */
bool NodeExists(GraphADT, Node);        /* returns whether the node exists */
listADT AdjacentNodes(GraphADT, Node); /* returns a list of adjacent nodes */
int Degree(GraphADT, Node);             /* returns the degree of node */

ArcADT Arc(Node, Node);                 /* returns an arc, if it exists; error otherwise */
                                        /* note that Arc(i,j) is the same as Arc(j,i) */
Node NodeI(ArcADT);                     /* returns the node of an arc with smaller index */
Node NodeJ(ArcADT);                     /* returns the node of an arc with larger index */
bool AddArc(GraphADT, ArcADT, float);   /* returns whether the operation succeeds */
                                        /* note that the third argument is the weight */
float ArcWeight(GraphADT, ArcADT);      /* returns the weight of an arc, if it exists */
bool ArcExists(GraphADT, ArcADT);       /* returns whether the arc exists */

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

int ArctoIndex(GraphADT, ArcADT);       /* see below */
ArcADT IndextoArc(GraphADT, int);       /* 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];
};

struct ArcCDT {
        Node i, j; /* make sure that both nodes exist, and i <= j */
};
```

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), $0 \le i \le j < MAX\_N$, 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];

};

struct ArcCDT {
        Node i, j; /* make sure that both nodes exist, and i <= j */
};
```

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, ArcADT a) {
        if (!ArcExists(G, a)) exit(EXIT_FAILURE);
        return a->j + a->i*MAX_N - a->i*(a->i+1)/2;
}
```

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

```
#include <stdlib.h>

ArcADT IndextoArc(GraphADT G, int x) {
        ArcADT a = (ArcADT) malloc(sizeof(*a));
        int x1; a->i = 1;
        while (x >= (x1 = (a->i) * MAX_N - (a->i) * (a->i - 1) / 2)) a->i++;
        a->i--; a->j = MAX_N + x - x1;
        return a; /* 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 'n' or 'a'. Suppose m and n are two integers, and w a float number, then a line

n m

indicates that a node m should be added. This can be done by a function call AddNode(G, m).

The line

a m n w

indicates that an undirected arc (m, n) with weight w should be added. This can be done by a function call AddArc(G, Arc(m,n), w).

For example, the graph shown previously in Question 5 can be constructed using the following data file:

```
n  0
n  1
a 0 1 5
n 2
  n    3
a 0 2 2
 n 4
n 5
 n   6
n  7
a 0 3   3
a 3 1 2
  a 3 2 7
a 1 7 3
a 3 4      1
  a 3 5 7
a 2  5 2
a 6 2 6
a 6 7   1
a 5  7 2
   a 4 7 1
```

When we call PrintAllNodes, the output should look like this:

```
0
1
2
3
4
5
6
7
```

When we call PrintAllArcs, the arcs are printed in the form of "(i,j) w", where i and j are nodes, i ≤ j, and w is the weight of the arc. The arcs are printed in the ascending order of i, and then in the ascending order of j. Hence the output should look like this:

```
(0,1) 5
(0,2) 2
(0,3) 3
(1,3) 2
(1,7) 3
(2,3) 7
(2,5) 2
(2,6) 6
(3,4) 1
(3,5) 7
(4,7) 1
(5,7) 2
(6,7) 1
```

PART II

Next, write a function mspK to find the minimum spanning tree of a graph, and return the minimum spanning tree as a graph:

GraphADT mspK(GraphADT);

Hints:

There are several main steps in this function.

Step 1. We can use a priority queue to store the arcs, so that it is easier to find the shortest arcs one after another. To do so, we need to define and implement an ADT of priority queue of arcs.

Step 2. In order to decide whether the currently shortest arc should be included in the minimum spanning tree, we can use a disjoint-set to store the arcs. To do so, we need to define and implement an ADT of disjoint-set of nodes. We can check whether the two nodes of an arc are in the same class of the disjoint-set of all nodes. Note that if x is the index of an arc a in a graph G, then the two nodes are NodeI(a) and NodeJ(a), or NodeI(IndextoArc(G,x)) and NodeJ(IndextoArc(G,x)). The arc of index x should not be added to the minimum spanning tree if Find(NodeI(IndextoArc(G,x)))==Find(NodeJ(IndextoArc(G,x))). Contrarily, if this arc should be added to the minimum spanning tree, then we must remember to perform SetUnion(NodeI(IndextoArc(G,x)), NodeJ(IndextoArc(G,x))).

Step 3. After the first two steps, the next step is relatively simple. We should keep performing the following steps: doing priority-dequeue to obtain the shortest arc, then either adding it to the minimum spanning tree or discarding it, until the priority-queue is empty.

Finally, print all arcs in the minimum spanning tree using PrintAllNodes and PrintAllArcs.