Class	CMPS 261
Section	001
Problem	Programming Assignment #4
Name	McKelvy, James Markus
CLID	Jmm0468
Due Date	12:30pm November 15, 2005

II. Design Documentation

II.1 System Architecture Description

The object(s) within the program are:
UnionFind: UnionFind.h
minHeap: minHeap.h
Edge: Edge.h
AdjacencyList: AdjacencyList.h adjacentVertex.h

The main objective of the application is to utilize the implementation of the UnionFind class, the minHeap class, and the AdjacencyList class, as well as Kruskal’s Algorithm to generate the Minimal Spanning Tree of a given graph. The application's main driver will be used to prompt the user for the total number of edges, the total number of vertices, and the information for each edge in order to use Kruskal’s Algorithm. The program will output to the user the minimal spanning tree created from the given graph and will output each edge added to the tree in the order in which it was added.

II.2 Information about the Objects

II.2.1 Class Information
Name: UnionFind
Description: A class representing the Union/Find construct containing partitions. It has member functions allowing it to perform a find on an element, as well as union on members of different partitions.
Base Class: N/A

II.2.2 Class Attributes
Name: array
Description: Represents the Union/Find data structure in an array
Type: int *
Range of acceptable values: Any value of type "int"

Name: max
Description: Represents the maximum number of partitions
Type: int
Range of acceptable values: Any value greater than 0

Name: currentPartitions
Description: Represents the current number of partitions
Type: int

Range of acceptable value: Any value greater than 0 and less than or equal to "max".

II.2.3 Class Operations
Prototype: UnionFind(int maxPartitions);
Description: Default Constructor
Preconditions:
1. The value of "maxPartitions" must be > 0
2. There must be enough memory to allocate an array of size "maxPartitions"

Postcondition:
1. If "maxPartitions" is > 0
   a. The value of "max" is set to the value of "maxPartitions"
   b. An array of size "maxPartitions" is allocated in memory
2. If "maxPartitions" is <= 0
   a. The value of "max" is set to the value of "1"
   b. An array of size "1" is allocated in memory

Cost Analysis: O(n)
Visibility: public

Prototype: ~UnionFind();
Description: Destructor
Precondition: The dynamic int array "array" has been allocated
Postcondition: Deallocates the memory used by this object
Cost Analysis: O(1)
Visibility: public

Prototype: int Find(int x);
Description: Finds the partition that an element is in
Precondition: The UnionFind object must be initialized and the item must be of type "int".
Postcondition:
1. If the value of "x" < "0" or >= "max" a "-1" is returned indicating that there is no leader for an invalid partition
2. If the element at the current array position is a negative number, then the current index is returned indicating that the position in the array is a leader of "x"
3. If the element at the current array position is a positive number or zero, then through a series of recursive steps until the partition leader is found, each node not pointing to the partition leader is made to do so (path compression).

Cost Analysis: O(log*(n)) ~ O(1)
Visibility: public

Prototype: void Union(int x, int y);
Description: Unions (merges) two partitions together into a new partition
Precondition: The UnionFind object must be initialized and the "x" and "y" must be integers
Postcondition:

1. If the values of "x" or "y" are invalid, nothing is done
2. If the partition leaders of "x" and "y" are the same, nothing is done
3. If the partition leaders of "x" and "y" are unique, then merge them by making the smaller partition leader a leader of the larger partition.
4. Change the value of "currentPartitions" if two partitions are merged.

Cost Analysis: O(log*(n)) ~ O(1)
Visibility: public

Prototype: int numberOfPartitions();
Description: Finds the number of partitions in the UnionFind's current state
Precondition: The UnionFind object must be initialized.
Postcondition: The value of "currentPartitions" is returned.
Cost Analysis: O(1)
Visibility: public

II.2.1 Class Information
Name: minHeap
Description: Simulates a minimum heap with the ability to have items inserted into it and its minimum element removed.
Base Class: N/A

II.2.2 Class Attributes
Name: heap
Description: This is a pointer to an array of elements that represent the minimum heap tree.
Type: Type *
Range of acceptable values: Any value that corresponds to the template class "Type".

Name: count
Description: Keeps track of the current number of elements in the heap.
Type: int
Range of acceptable values: Any number greater than or equal to zero.

Name: maxSize
Description: Keeps track of the maximum size of this heap
Type: int
Range of acceptable values: Any number greater than zero.

II.2.3 Class Operations
Prototype: minHeap(int size);
Description: Default Constructor
Precondition: There is enough memory to be allocated
Postcondition: Creates an empty minimum heap with a max size of maxSize if and only if maxSize > 0, else the max size will be set to a default value of 15.
Cost Analysis: O(1)
Visibility: public

Prototype: ~minHeap()
Description: Destructor
Precondition: The minimum heap has been allocated
Postcondition: deallocates the memory used by this object
Cost Analysis: O(1)
Visibility: public

Prototype: bool insert(Type item);
Description: Inserts the item into the heap.
Precondition: The heap must be initialized and the item must be of type Type.
Postcondition: The item is inserted to the heap and is moved to the correct level in the tree.
Cost Analysis: log(n)
Visibility: public

Prototype: Type removeMin();
Description: Removes the minimum item from the heap
Precondition: The heap must be initialized and have an item to remove.
Postcondition: The minimum item is removed from the heap (if it exists) and returned, otherwise null is returned.
Cost Analysis: log(n)
Visibility: public

Prototype: bool isLeaf(int index);
Description: Checks to see if the current index is a leaf in the heap.
Precondition: The heap must be initialized.
Postcondition: Returns true if the item is a leaf node, false otherwise.
Cost Analysis: O(1)
Visibility: private

Prototype: int leftChild(int index);
Description: Finds the position of the left child from the given index.
Precondition: The index must be valid and the heap must be initialized
Postcondition: Returns the index of the left child (if it exists), or a -1 otherwise.
Cost Analysis: O(1)
Visibility: private

Prototype: int rightChild(int index);
Description: Finds the position of the right child from the given index.
Precondition: The index must be valid and the heap must be initialized
Postcondition: Returns the index of the right child (if it exists), or a -1 otherwise.
Cost Analysis: O(1)
Visibility: private

Prototype: int parent(int index);
Description: Finds the position of the parent from the given index.
Precondition: The index must be valid and the heap must be initialized
Postcondition: Returns the index of the parent (if it exists), or a -1 otherwise.

Cost Analysis: O(1)
Visibility: private

II.2.1 Class Information
Name: Edge
Description: Represents an Edge in a graph, with values for the edge weight and the vertices connected by the edge.
Base Class: N/A

II.2.2 Class Attributes
Name: vertex1
Description: Holds the name of the first vertex connected by this edge.
Type: int
Range of acceptable values: Any greater than or equal to 1.

Name: vertex2
Description: Holds the name of the second vertex connected by this edge.
Type: int
Range of acceptable values: Any greater than or equal to 1.

Name: weight
Description: Holds the edge weight of this edge.
Type: int
Range of acceptable values: Any greater than or equal to 1.

II.2.3 Class Operations
Prototype: Edge();
Description: Default constructor to create an edge with no data.
Precondition: None
Postcondition: An edge is created with vertex 1 being "-1", vertex 2 being "-1", and the weight being "-1" to symbolize that the edge is not really valid.
Cost Analysis: O(1)
Visibility: public

Prototype: Edge(int v1, int v2, int w);
Description: Default constructor to create an edge with incident vertices v1 and v2, and the weight of the edge.
Precondition: v1, v2, and w are expected to be of type "int" and all greater than zero.
Postcondition: An edge is created with vertex 1 being "v1", vertex 2 being "v2", and the weight being "w", unless one of the values is less than or equal to "0", then they all will have the value of "-1".
Cost Analysis: O(1)
Visibility: public

Prototype: ~Edge();
Description: Destructor.

Precondition: None
Postcondition: None, no memory was allocated, no memory needs to be deleted
Cost Analysis: O(1)
Visibility: public

Prototype: int first();
Description: Member function to return the integer representation of the
first incident vertex.
Precondition: None
Postcondition: The integer form of the first vertex is returned. If the Edge
has not been initialized with proper values, "-1" may be returned.
Cost Analysis: O(1)
Visibility: public

Prototype: int second();
Description: Member function to return the integer representation of the
second incident vertex.
Precondition: None
Postcondition: The integer form of the second vertex is returned. If the Edge
has not been initialized with proper values, "-1" may be returned.
Cost Analysis: O(1)
Visibility: public

Prototype: int edgeWeight();
Description: Member function to return the integer weight of the edge.
Precondition: None
Postcondition: Returns the weight of this edge. If the Edge has not been
initialized with proper values, "-1" may be returned.
Cost Analysis: O(1)
Visibility: public

Prototype: void operator=(Edge rhs);
Description: Overloaded assignment operator.
Precondition: The right hand side Edge is expected to be initialized already
with proper values.
Postcondition: This edge will take on the exact same values as the right
hand side Edge, even if it has improper values.
Cost Analysis: O(1)
Visibility: public

Prototype: bool operator<=(Edge rhEdge);
Description: Overloaded less than or equal to operator
Precondition: The right hand side Edge is expected to be initialized already
with proper values.
Postcondition: Returns true if the value of the weight of this Edge is less
than or equal to the weight of right hand side Edge.
Cost Analysis: O(1)
Visibility: public

Prototype: bool operator<(Edge rhEdge);
Description: Overloaded less than operator
Precondition: The right hand side Edge is expected to be initialized already with proper values.
Postcondition: Returns true if the value of the weight of this Edge is less than the weight of right hand side Edge.
Cost Analysis: O(1)
Visibility: public

Prototype: bool operator>(Edge rhEdge);
Description: Overloaded greater than or equal to operator
Precondition: The right hand side Edge is expected to be initialized already with proper values.
Postcondition: Returns true if the value of the weight of this Edge is greater than or equal to the weight of right hand side Edge.
Cost Analysis: O(1)
Visibility: public

Prototype: ostream & operator<<(ostream & out, Edge & e);
Description: Overloaded extraction operator
Precondition: A reference to an ostream object is received, a reference to an edge object is received.
Postcondition: Print out the values of this edge in the format of "(vertex1, vertex2, weight)" and returns a reference to an ostream object.
Cost Analysis: O(1)
Visibility: friend

Prototype: istream & operator>>(istream & in, Edge * e);
Description: Overloaded insertion operator
Precondition: A reference to an istream object is received, a pointer alreay initialized of an Edge is received.
Postcondition: Sets the values of vertex1, vertex2, and weight as taken from the input. Returns a reference to an istream object.
Cost Analysis: O(1)
Visibility: friend

II.2.1 Class Information
Name: AdjacencyList
Description: An implementation of of an Adjacency List of a sparse graph. Keeps track of vertices adjacent to each vertex, as well as their weights. Allows for adding of edges, and printing of the current list.
Base Class: N/A

II.2.2 Class Attributes
Name: edgesAdded
Description: Represents the total number of edges that have been added to the graph.

Type: int
Range of acceptable values: Any value greater than or equal to "0".

Name: totalVertices
Description: Represents the total number of vertices in the graph.
Type: int
Range of acceptable values: Any value greater than or equal to "1".

Name: arrayOfVertices
Description: Represents the all of the vertices of the graph, as well as
their adjacent vertices.
Type: node *
Range of acceptable values: Values that conform to the "node" type.

II.2.3 Class Operations
Prototype: AdjacencyList();
Description: Default constructor for an Adjacency List representing a graph
of "numberOfVertices" vertices.
Precondition: The value of "numberOfVertices" is expected to be greater than
or equal to "1".
Postcondition: Allocates memory for an adjacency list that has "numberOfVertices"
nodes to represent all the vertices.
Cost Analysis: O(n)
Visibility: public

Prototype: ~AdjacencyList();
Description: Destructor.
Precondition: "arrayOfVertices" has been initilized.
Postcondition: Deallocates the memory used by this object.
Cost Analysis: O(n^2)
Visibility: public

Prototype: void addEdge(Edge * e);
Description: Member function to add an edge to the adjacency edge.
Precondition: The value of "e" is not NULL and has been allocated. Also, "e"
does not contain any vertices in it that are out of the range of this
adjacency list.
Postcondition: Adds an adjacent vertex to the vertex specified by calling
e->first(). Adds an adjacent vertex to the vertex specifid by calling
e->second().
Cost Analysis: O(n^2)
Visibility: public

Prototype: void printAll();
Description: Prints the values that are stored in the adjacency list,
starting with the first vertex and listing all adjacent vertices, and
proceeding to the last vertex and listing all adjacent vertices.
Precondition: The variable "arrayOfVertices" has been allocated.

Postcondition: Calls "printNextNode()" on the first vertex.
Cost Analysis: O(n^2)
Visibility: public

Prototype: adjacentVertex * createNode(Edge * e, bool useFirst);
Description: Creates a pointer to an adjacentVertex using an Edge.
Precondition: The value of "e" is not NULL and has been allocated. "useFirst"
  is either given as true or false.
Postcondition: Creates a new adjacent vertex with either the name of e->first()
  or e->second() depending on the value of "useFirst". Sets the value of
  weight to be e->edgeWeight(), and sets "next" to be NULL.
Cost Analysis: O(1)
Visibility: private

Prototype: void removeNextNode(node * current);
Description: Helper of the destructor, deletes the chain of vertices.
Precondition: "current" is not equal to NULL.
Postcondition: Recursively deletes the "node" nodes connected to
  the original "node" passed, including itself.
Cost Analysis: O(n)
Visibility: private

Prototype: void removeNext(adjacentVertex * current);
Description: Helper of the destructor, deletes the chain hanging off a vertex.
Precondition: "current" is not equal to NULL.
Postcondition: Recursively deletes the "adjacentVertex" nodes connected to
  the original "adjacentVertex" passed, including itself.
Cost Analysis: O(n)
Visibility: private

Prototype: void printNextNode(node * current);
Description: Prints the values that are stored in the adjacency list,
  starting with the "current" vertex and listing all adjacent vertices, and
  proceeding to the last vertex and listing all adjacent vertices.
Precondition: The variable "current" has been allocated.
Postcondition: Prints information about the current vertex, recursively calls
  "printNext()" and then recursively calls "printNextNode()".
Cost Analysis: O(n^2)
Visibility: private

Prototype: void printNext(adjacentVertex * current);
Description: Prints the values that are stored in adjacent vertices.
Precondition: The variable "current" has been allocated.
Postcondition: Prints information about the current adjacent vertex and then
  recursively calls "printNext()".
Cost Analysis: O(n)
Visibility: private

II.2.1 Class Information
Name: node
Description: Represents a vertex by name, with a pointer to (possibly)
adjacent vertices, and a pointer to (possibly) the next vertex in the graph

II.2.1 Class Information
Name: adjacentVertex
Description: Represents an adjacent vertex by name and weight, with a pointer
to (possibly) other adjacent vertices.

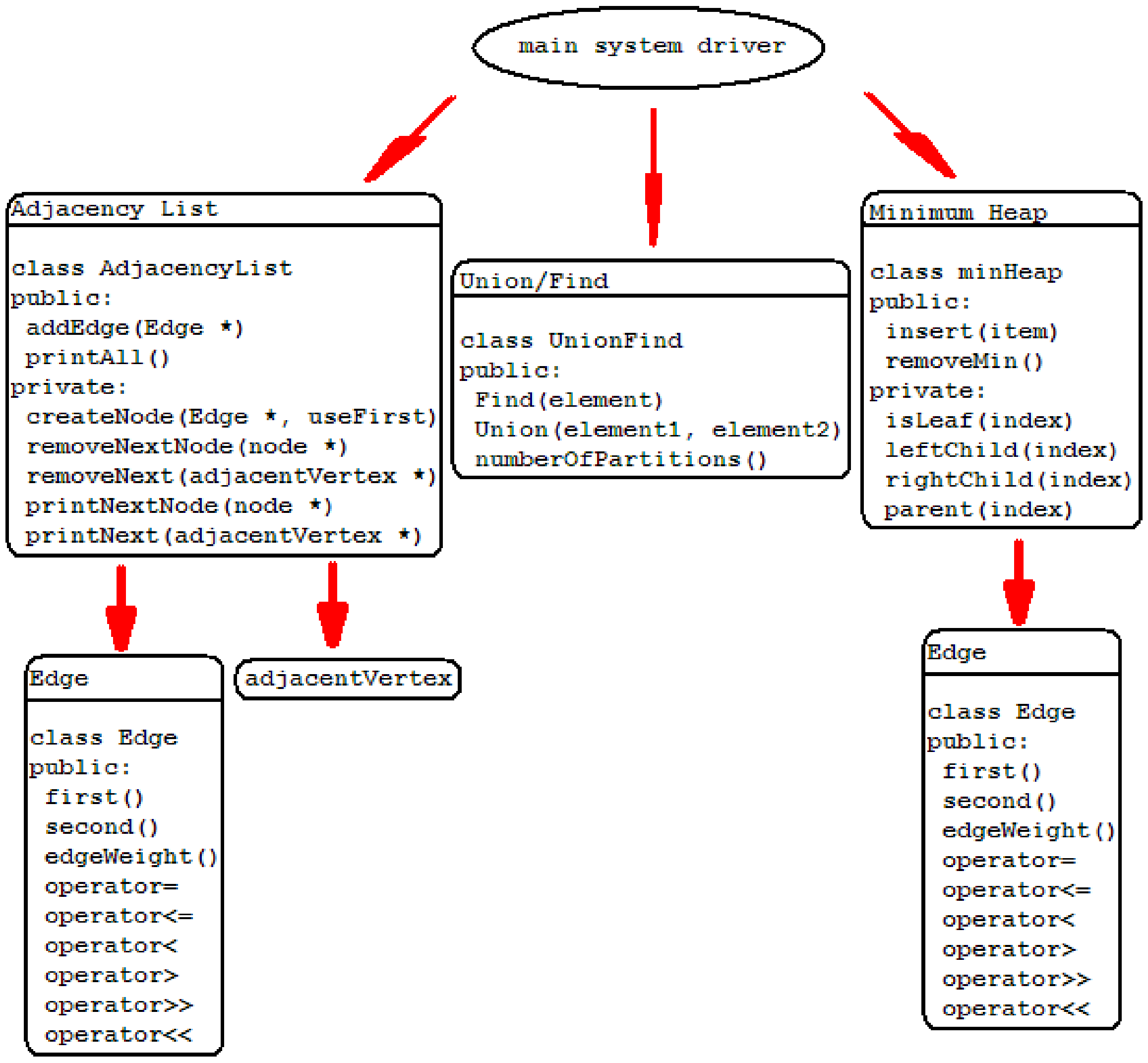
II.3 Information about the Main Application

```
#include<iostream>
#include<cstdlib>
#include "Edge.h"
#include "AdjacencyList.h"
#include "minHeap.h"
#include "UnionFind.h"

using namespace std;

int main(){
   int vertices;          // number of vertices in the graph
   int edges;             // number of edges in the graph
   int edges_accepted = 0;  // total edges in the mst
   int p1, p2;            // used for "find()" on edges
   AdjacencyList * adjlist; // holds the mst
   Edge * e = new Edge;     // an edge on the graph
   minHeap<Edge> * heap;    // for getting the smallest edge
   UnionFind * unionfind;   // checking if an edge will create a cycle

   // how many edges?
   cout << "|E|: ";
   cin >> edges;
   // how many vertices?
   cout << "|V|: ";
   cin >> vertices;
   cout << endl << "Input: vertex1 vertex2 weight"
       << endl << endl;

   // allocate memory for our objects

   // will have "vertices" number of vertices
   adjlist = new AdjacencyList(vertices);
   // will have "edges" number of edges
   heap = new minHeap<Edge>(edges);
   // will have "vertices" number of partitions
   unionfind = new UnionFind(vertices);
```

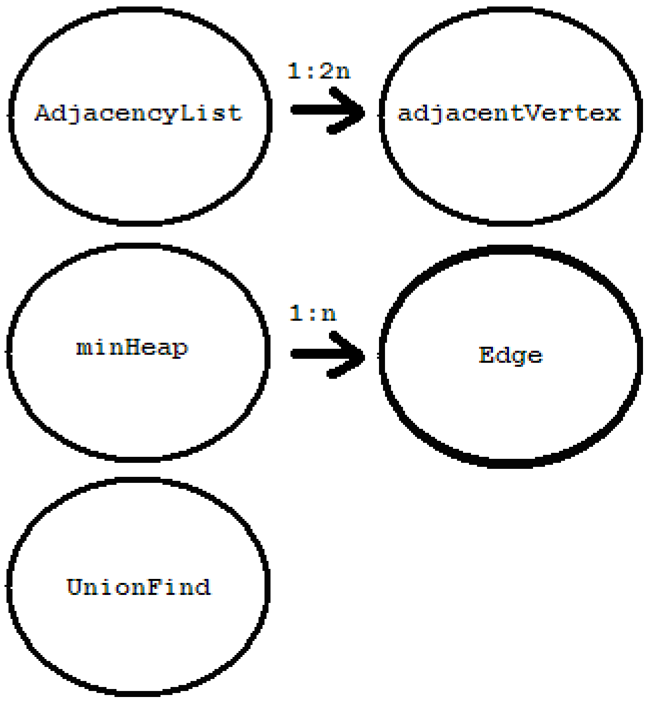
```
  // get the edges from the user
  // will ask only as many times as there are edges
  for(int i = 0; i < edges; i++){
    cout << i+1 << ": ";
    cin >> e;
    heap->insert(*e);
  }

  // prepare to output the mst
  cout << "\n\nThe Minimal Spanning Tree for this graph contains the edges:\n\n";

  // loop until we have all the vertices connected
  while(edges_accepted < (vertices - 1)){
    // get the smallest edge off the heap
    *e = heap->removeMin();
    // check to see that the two vertices are not already connected
    p1 = unionfind->Find(e->first());
    p2 = unionfind->Find(e->second());
    if(p1 != p2){
      // add this edge to the mst
      adjlist->addEdge(e);
      // update our union find so that we won't create a cycle
      unionfind->Union(e->first(), e->second());
      // update the number of edges on our mst
      edges_accepted++;
      // output the newly added edge to our mst to the user
      cout << edges_accepted << ". From vertex " << e->first() << " to "
           << e->second() << ", at a cost of " << e->edgeWeight() << ".\n";
    }
  }

  // and we're done

  return 0;
}
```

II.4 Design Diagrams

II.4.1 Object Interaction Diagram



II.4.2 Aggregation Diagram


AdjacencyList
1:2n
adjacentVertex
minHeap
1:n
Edge
UnionFind