North South University
CSE-225L(Data Structures & Algorithm)
Summer - 2018
Lab-14 (Graph)

main.cpp

```
#include <iostream>
#include "graphtype.h"
#include <string>
using namespace std;
int main()
{
    GraphType<string> g(10); // a graph with 10 vertices

     /*
          write the necessary codes here for the tasks
     */
    return 0;
}
```

graphtype.h

```
#ifndef GRAPHTYPE_H_INCLUDED
#define GRAPHTYPE_H_INCLUDED
#include "stacktype.h"
#include "quetype.h"

template<class VertexType>
class GraphType
{
     public:
          GraphType(int maxV);
          ~GraphType();
          void MakeEmpty();
          bool IsEmpty();
          bool IsFull();
          void AddVertex(VertexType);
          void AddEdge(VertexType,VertexType,int);
          int WeightIs(VertexType,VertexType);
          void GetToVertices(VertexType,QueType<VertexType>&);
          void ClearMarks();
          void MarkVertex(VertexType);
          bool IsMarked(VertexType);
          void DepthFirstSearch(VertexType,VertexType);
          void BreadthFirstSearch(VertexType,VertexType);

     private:
          int numVertices;
          int maxVertices;
          VertexType* vertices;
          int **edges;
          bool* marks;
};
#endif // GRAPHTYPE_H_INCLUDED
```

graphtype.cpp

```
#include "graphtype.h"

const int NULL_EDGE = 0;

template<class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
      numVertices = 0;
      maxVertices = maxV;

      vertices = new VertexType[maxV];
      edges = new int*[maxV];

      for(int i=0;i<maxV;i++)
            edges[i] = new int [maxV];
      marks = new bool[maxV];
}

template<class VertexType>
GraphType<VertexType>::~GraphType()
{
      delete [] vertices;
      delete [] marks;

      for(int i=0;i<maxVertices;i++)
            delete [] edges[i];
      delete [] edges;
}


template<class VertexType>
void GraphType<VertexType>::MakeEmpty()
{
      numVertices = 0;
}


template<class VertexType>
bool GraphType<VertexType>::IsEmpty()
{
      return (numVertices == 0);
}


template<class VertexType>
bool GraphType<VertexType>::IsFull()
{
      return (numVertices == maxVertices);
}
```

```
template<class VertexType>
void GraphType<VertexType>::AddVertex(VertexType vertex)
{
      vertices[numVertices] = vertex;

      for (int index=0; index<numVertices; index++)
      {
            edges[numVertices][index] = NULL_EDGE;
            edges[index][numVertices] = NULL_EDGE;
      }

      numVertices++;
}



template<class VertexType>
int IndexIs(VertexType* vertices, VertexType vertex)
{
      int index = 0;
      while (!(vertex == vertices[index]))
            index++;
      return index;
}


template<class VertexType>
void GraphType<VertexType>::ClearMarks()
{
      for(int i=0; i<maxVertices; i++)
            marks[i] = false;
}


template<class VertexType>
void GraphType<VertexType>::MarkVertex(VertexType vertex)
{
      int index = IndexIs(vertices, vertex);
      marks[index] = true;
}


template<class VertexType>
bool GraphType<VertexType>::IsMarked(VertexType
vertex)
{
      int index = IndexIs(vertices, vertex);
      return marks[index];
}
```

```
template<class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex,
VertexType toVertex, int weight)
{
      int row = IndexIs(vertices, fromVertex);
      int col= IndexIs(vertices, toVertex);
      edges[row][col] = weight;
}


template<class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex,
VertexType toVertex)
{
      int row = IndexIs(vertices, fromVertex);
      int col= IndexIs(vertices, toVertex);
      return edges[row][col];
}


template<class VertexType>
void GraphType<VertexType>::GetToVertices(VertexType vertex,
QueType<VertexType>& adjVertices)
{
      int fromIndex, toIndex;

      fromIndex = IndexIs(vertices, vertex);

      for (toIndex = 0; toIndex < numVertices; toIndex++)
            if (edges[fromIndex][toIndex] != NULL_EDGE)
                  adjVertices.Enqueue(vertices[toIndex]);
}

template<class VertexType>
void GraphType<VertexType>::DepthFirstSearch(VertexType
startVertex, VertexType endVertex)
{
            StackType<VertexType> stack;
            QueType<VertexType> vertexQ;

         int cost = 0;
            bool found = false;
            VertexType vertex, item;

            ClearMarks();
            stack.Push(startVertex);

            do
            {
                  vertex = stack.Top();
                  stack.Pop();
```

```
                else
                {

                      if (!IsMarked(vertex))
                      {
                            MarkVertex(vertex);
                            cout << vertex << " ";


                            GetToVertices(vertex,vertexQ);

                            while (!vertexQ.IsEmpty())
                            {
                                  vertexQ.Dequeue(item);
                                  if (!IsMarked(item))
                                  stack.Push(item);
                            }
                      }

                }

           } while (!stack.IsEmpty() && !found);

           cout << endl;

           if (!found)
                cout << "Path not found." << endl;
        else
        {
            cout<<"Path Found. Path Cost = "<<cost<<endl;
        }
}




template<class VertexType>
void GraphType<VertexType>::BreadthFirstSearch(VertexType
startVertex, VertexType endVertex)
{

      QueType<VertexType> queue;
      QueType<VertexType> vertexQ;

      bool found = false;
      VertexType vertex, item;

      ClearMarks();
      queue.Enqueue(startVertex);
```

```
      do
      {
            queue.Dequeue(vertex);

            if (vertex == endVertex)
            {
                  cout << vertex << " ";
                  found = true;
            }

            else
            {
                  if (!IsMarked(vertex))
                  {
                        MarkVertex(vertex);
                        cout << vertex << " ";
                        GetToVertices(vertex, vertexQ);

                        while (!vertexQ.IsEmpty())
                        {
                              vertexQ.Dequeue(item);
                              if (!IsMarked(item))
                              queue.Enqueue(item);
                        }
                  }
            }

      } while (!queue.IsEmpty() && !found);

      cout << endl;

      if (!found)
            cout << "Path not found." << endl;
}

template class GraphType<char>;
```

stacktype.h

```
#ifndef STACKTYPE_H_INCLUDED
#define STACKTYPE_H_INCLUDED

class FullStack
{};

class EmptyStack
{};

template <class ItemType>
class StackType
{
```

```
      struct NodeType
      {
      ItemType info;
      NodeType* next;
      };

public:

      StackType();
      ~StackType();
      void Push(ItemType);
      void Pop();
      ItemType Top();
      bool IsEmpty();
      bool IsFull();

private:
      NodeType* topPtr;
};
#endif // STACKTYPE_H_INCLUDED
```

stacktype.cpp

```
#include <iostream>
#include "stacktype.h"
using namespace std;

template <class ItemType>
StackType<ItemType>::StackType()
{
      topPtr = NULL;
}

template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
      return (topPtr == NULL);
}


template <class ItemType>
ItemType StackType<ItemType>::Top()
{
      if (IsEmpty())
            throw EmptyStack();
      else
            return topPtr->info;
}


template <class ItemType>
```

```
bool StackType<ItemType>::IsFull()
{
      NodeType* location;

      try
      {
            location = new NodeType;
            delete location;
            return false;
      }
      catch(bad_alloc& exception)
      {
            return true;
      }
}


template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{

      if (IsFull())
            throw FullStack();
      else
      {
            NodeType* location;
            location = new NodeType;
            location->info = newItem;
            location->next = topPtr;
            topPtr = location;
      }
}


template <class ItemType>
void StackType<ItemType>::Pop()
{
      if (IsEmpty())
            throw EmptyStack();
      else
      {
            NodeType* tempPtr;
            tempPtr = topPtr;
            topPtr = topPtr->next;
            delete tempPtr;
      }
}



template <class ItemType>
```

```
StackType<ItemType>::~StackType()
{
      NodeType* tempPtr;

      while (topPtr != NULL)
      {
            tempPtr = topPtr;
            topPtr = topPtr->next;
            delete tempPtr;
      }
}


template class StackType<char>;
```

quetype.h

```
#ifndef QUETYPE_H_INCLUDED
#define QUETYPE_H_INCLUDED
#include <iostream>
using namespace std;
class FullQueue
{};
class EmptyQueue
{};
    template <class ItemType>
    class QueType
    {
        struct NodeType
        {
            ItemType info;
            NodeType* next;
        };

        public:
            QueType();
            ~QueType();
            void MakeEmpty();
            void Enqueue(ItemType);
            void Dequeue(ItemType&);
            bool IsEmpty();
            bool IsFull();

        private:
            NodeType *front, *rear;
    };
#endif // QUETYPE_H_INCLUDED
```

quetype.cpp

```
#include "quetype.h"

template <class ItemType>
QueType<ItemType>::QueType()
{
    front = NULL;
    rear = NULL;
}


template <class ItemType>
bool QueType<ItemType>::IsEmpty()
{
    return (front == NULL);
}


template<class ItemType>
bool QueType<ItemType>::IsFull()
{
    NodeType* location;
    try
    {
    location = new NodeType;
    delete location;
    return false;
    }
    catch(bad_alloc& exception)
    {
    return true;
    }
}


template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
    if (IsFull())
        throw FullQueue();
    else
    {
        NodeType* newNode;
        newNode = new NodeType;
        newNode->info = newItem;
        newNode->next = NULL;
        if (rear == NULL)
            front = newNode;
        else
```

```
            rear->next = newNode;
        rear = newNode;
        }
}


template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
    if (IsEmpty())
    throw EmptyQueue();
    else
    {
        NodeType* tempPtr;
        tempPtr = front;
        item = front->info;
        front = front->next;
        if (front == NULL)
            rear = NULL;
        delete tempPtr;
    }
}


template <class ItemType>
void QueType<ItemType>::MakeEmpty()
{
    NodeType* tempPtr;
    while (front != NULL)
    {
        tempPtr = front;
        front = front->next;
        delete tempPtr;
    }
    rear = NULL;
}


template <class ItemType>
QueType<ItemType>::~QueType()
{
    MakeEmpty();
}


template class QueType<char>;
```