ಕರ್ನಾಟಕ ರಾಜ್ಯ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ
ಮಾನಸಗಂಗೋತ್ರಿ, ಮೈಸೂರು – 570 006

KARNATAKA STATE OPEN UNIVERSITY
Manasagangotri Mysore - 570 006

M.Sc. Computer Science

Second Semester

Course:11
Practical-3:
MSCS-511 ADA & DBMS

ಉನ್ನತ ಶಿಕ್ಷಣಕ್ಕಾಗಿ ಇರುವ ಅವಕಾಶಗಳನ್ನು ಹೆಚ್ಚಿಸುವುದಕ್ಕೆ ಮತ್ತು ಶಿಕ್ಷಣವನ್ನು ಪ್ರಜಾತಂತ್ರೀಕರಿಸುವುದಕ್ಕೆ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ ವ್ಯವಸ್ಥೆಯನ್ನು ಆರಂಭಿಸಲಾಗಿದೆ.

ರಾಷ್ಟ್ರೀಯ ಶಿಕ್ಷಣ ನೀತಿ 1986

The Open University System has been initiated in order to augment opportunities for higher education and as instrument of democrating education.

National Educational Policy 1986

ವಿಶ್ವ ಮಾನವ ಸಂದೇಶ

ಪ್ರತಿಯೊಂದು ಮಗುವು ಹುಟ್ಟುತ್ತಲೇ – ವಿಶ್ವಮಾನವ, ಬೆಳೆಯುತ್ತಾ ನಾವು ಅದನ್ನು 'ಅಲ್ಪ ಮಾನವ'ನನ್ನಾಗಿ ಮಾಡುತ್ತೇವೆ. ಮತ್ತೆ ಅದನ್ನು 'ವಿಶ್ವಮಾನವ'ನನ್ನಾಗಿ ಮಾಡುವುದೇ ವಿದ್ಯೆಯ ಕರ್ತವ್ಯವಾಗಬೇಕು.

ಮನುಜ ಮತ, ವಿಶ್ವ ಪಥ, ಸರ್ವೋದಯ, ಸಮನ್ವಯ, ಪೂರ್ಣದೃಷ್ಟಿ ಈ ಪಂಚಮಂತ್ರ ಇನ್ನು ಮುಂದಿನ ದೃಷ್ಟಿಯಾಗಬೇಕಾಗಿದೆ. ಅಂದರೆ, ನಮಗೆ ಇನ್ನು ಬೇಕಾದುದು ಆ ಮತ ಈ ಮತ ಅಲ್ಲ; ಮನುಜ ಮತ. ಆ ಪಥ ಈ ಪಥ ಅಲ್ಲ ; ವಿಶ್ವ ಪಥ. ಆ ಒಬ್ಬರ ಉದಯ ಮಾತ್ರವಲ್ಲ; ಸರ್ವರ ಸರ್ವಸ್ತರದ ಉದಯ. ಪರಸ್ಪರ ವಿಮುಖವಾಗಿ ಸಿಡಿದು ಹೋಗುವುದಲ್ಲ; ಸಮನ್ವಯಗೊಳ್ಳುವುದು. ಸಂಕುಚಿತ ಮತದ ಆಂಶಿಕ ದೃಷ್ಟಿ ಅಲ್ಲ; ಭೌತಿಕ ಪಾರಮಾರ್ಥಿಕ ಎಂಬ ಭಿನ್ನದೃಷ್ಟಿ ಅಲ್ಲ; ಎಲ್ಲವನ್ನು ಭಗವದ್ ದೃಷ್ಟಿಯಿಂದ ಕಾಣುವ ಪೂರ್ಣ ದೃಷ್ಟಿ.

ಕುವೆಂಪು

Gospel of Universal Man

Every Child, at birth, is the universal man. But, as it grows, we trun it into "a petty man". It should be the function of education to turn it again into the enlightened "universal man".

The Religion of Humanity, the Universal Path, the Welfare of All, Reconciliation, the Integral Vision - these ***five mantras*** should become view of the Future. In other words, what we want henceforth is not this religion or that religion, but the Religion of Humanity; not this path or that path, but the Universal Path; not the well-being of this individual or that individual, but the Welfare of All; not turning away and breaking off from one another, but reconciling and uniting in concord and harmony; and above all, not the partial view of a narrow creed, not the dual outlook of the material and the spiritual, but the Integral Vision of seeing all things with the eye of the Divine.

Kuvempu

Karnataka State Open University

Manasagangothri, Mysore - 570 006

M.Sc in Computer Science

Second Semester

LABORATORY EXERCISES

MSCS-511: Practical-3

ADA AND DBMS LABORATORY

Foreword

C++ is a general purpose programming language with close to machine fundamental semantics, suitable for kernel and systems programming. It supports data abstraction, object-oriented programming and generic programming. It is used in a very wide range of applications and it is popular among professional software developers. A good knowledge of C++ is very much essential for students to better understand the basic concepts of object-oriented programming and to exploit the language features to develop better and efficient software products.

The book entitled "Programming in C++" is brought out to cater the needs of the beginners who want to learn the concept of object-oriented programming. The book introduces the basic concepts of object-oriented programming as simple as possible with illustrative examples. The book is written with a philosophy that a good book needs to treat the subject as simple as possible and should support self learning. Each and every chapter is well organized and presented interactively with suitable examples to make the concepts complete, clear and concrete. The solved problems, review questions and programming exercises discussed at the end of each chapter will definitely help the students to learn the subject by practice. The book has covered almost all the features of object-oriented programming concepts essential for students of computer science and management courses at both undergraduate and postgraduate level, which offer C++ language as part of their syllabus. I am sure that after having studied this book the students will gain sufficient knowledge about programming to develop better and efficient object-oriented programs.

The four authors, who crafted this book, have got rich experience both in teaching as well as in research. Their class room interactions with students and the students' feedback have been incorporated to make the material more effective. I feel this book can be good source material for any academic institution, which offers a course on "Programming with C

I congratulate all the authors for their interest and effort in bringing this book to cater the needs of the beginners and I wish they continue to write many such books in future.

Sample Programs for Analysis and Design of Algorithms

Note: Related header files to be used in the programs are listed below with their name.

```
// Header file name weightedEdge.h
// Edge in a weighted graph

#ifndef weightedEdge_
#define weightedEdge_

#include "edge.h"

using namespace std;

template <class T>
class weightedEdge : public edge<T>
{
  public:
    weightedEdge() {};
    weightedEdge(int theV1, int theV2, T theW)
      {v1 = theV1; v2 = theV2; w = theW;}
    ~weightedEdge() {};
    int vertex1() const {return v1;}
    int vertex2() const {return v2;}
    T weight() const {return w;}
```

```
    operator T() const {return w;}
    void output(ostream& out) const
    {// Put the edge into the stream out.
      out << "(" << v1 << ", " << v2 << ", " << w << ")";
    }

  private:
    int v1,
        v2;
    T w;
};

// overload <<
template <class T>
ostream& operator<<(ostream& out, const weightedEdge<T>& x)
  {x.output(out); return out;}
#endif
```

```
// Header file name: unweightedEdge.h
// Edge in an unweighted graph

#ifndef unweightedEdge_
#define unweightedEdge_

#include "edge.h"
#include "myExceptions.h"

using namespace std;

class unweightedEdge : public edge<bool>
```

```
{
  public:
    unweightedEdge(int theV1, int theV2)
      {v1 = theV1; v2 = theV2;}
    ~unweightedEdge() {};
    int vertex1() const {return v1;}
    int vertex2() const {return v2;}
    bool weight() const {throw undefinedMethod("weight");}
  private:
    int v1,
        v2;
};
#endif
```

```
// Headerfile name: vetexIterator.h
// Abstract class for graph vertex iterator

#ifndef vertexIterator_
#define vertexIterator_
using namespace std;

template<class T>
class vertexIterator
{
  public:
    virtual ~vertexIterator() {}
    virtual int next() = 0;
    virtual int next(T&) = 0;
};
```

```
#endif
```

```
// Headerfile name: delete2dArray.h
// Delete a two-dimensional array

#ifndef delete2dArray_
#define delete2dArray_
using namespace std;

template <class T>
void delete2dArray(T ** &theArray, int numberOfRows)
{// Delete the two-dimensional array theArray.

  // delete the memory for each row
  for (int i = 0; i < numberOfRows; i++)
    delete [] theArray[i];

  // delete the row pointers
  delete [] theArray;
  theArray = 0;
}
#endif
```

```
// Header file name: graphChain.h
// Extension of chain to include a method to erase by matching element field
// This class is for use by the linked graph classes

#ifndef graphChain_
#define graphChain_
```

```
#include "chainWithIterator.h"
#include "chainNode.h"

using namespace std;

template <class T>
class graphChain : public chain<T>
{
  public:
    T* eraseElement(int theVertex)
    {// Delete node with element == theVertex. Return pointer to
     // deleted element. Return NULL if no matching element.
      chainNode<T> *current = firstNode,
                   *trail = NULL; // one behind current

      // search for match
      while (current != NULL && current->element != theVertex)
      {
        trail = current;
        current = current->next;
      }

      if (current == NULL) // no match
        return NULL;

      // match found in node current
      T* theElement = &current->element; // save matching element

      // remove current from chain
      if (trail != NULL)
```

```
        trail->next = current->next;
      else
        firstNode = current->next; // current is first node

      delete current;
      listSize--;
      return theElement;
    }
};
#endif
```

```
// Header file name: minHeap.h
// heap implementation of a min priority queue derives from the ADT
minPriorityQueue

#ifndef minHeap_
#define minHeap_

#include "minPriorityQueue.h"
#include "myExceptions.h"
#include "changeLength1D.h"
#include <sstream>
#include <algorithm>

using namespace std;

template<class T>
class minHeap : public minPriorityQueue<T>
{
  public:
```

```
    minHeap(int initialCapacity = 10);
    ~minHeap() {delete [] heap;}
    bool empty() const {return heapSize == 0;}
    int size() const
      {return heapSize;}
    const T& top()
      {// return min element
        if (heapSize == 0)
          throw queueEmpty();
        return heap[1];
      }
    void pop();
    void push(const T&);
    void initialize(T *, int);
    void deactivateArray()
      {heap = NULL; arrayLength = heapSize = 0;}
    void output(ostream& out) const;
  private:
    int heapSize;       // number of elements in queue
    int arrayLength;    // queue capacity + 1
    T *heap;            // element array
};

template<class T>
minHeap<T>::minHeap(int initialCapacity)
{// Constructor.
  if (initialCapacity < 1)
  {ostringstream s;
   s << "Initial capacity = " << initialCapacity << " Must be > 0";
   throw illegalParameterValue(s.str());
  }
```

```
  arrayLength = initialCapacity + 1;
  heap = new T[arrayLength];
  heapSize = 0;
}

template<class T>
void minHeap<T>::push(const T& theElement)
{// Add theElement to heap.

  // increase array length if necessary
  if (heapSize == arrayLength - 1)
  {// double array length
    changeLength1D(heap, arrayLength, 2 * arrayLength);
    arrayLength *= 2;
  }

  // find place for theElement
  // currentNode starts at new leaf and moves up tree
  int currentNode = ++heapSize;
  while (currentNode != 1 && heap[currentNode / 2] > theElement)
  {
    // cannot put theElement in heap[currentNode]
    heap[currentNode] = heap[currentNode / 2]; // move element down
    currentNode /= 2;                          // move to parent
  }

  heap[currentNode] = theElement;
}

template<class T>
void minHeap<T>::pop()
```

```
{// Remove max element.
  // if heap is empty return null
  if (heapSize == 0)  // heap empty
    throw queueEmpty();

  // Delete min element
  heap(1).~T();

  // Remove last element and reheapify
  T lastElement = heap(heapSize--);

  // find place for lastElement starting at root
  int currentNode = 1,
     child = 2;    // child of currentNode
  while (child <= heapSize)
  {
    // heap(child) should be smaller child of currentNode
    if (child < heapSize && heap(child) > heap(child + 1))
      child++;

    // can we put lastElement in heap(currentNode)?
    if (lastElement <= heap(child))
      break;  // yes

    // no
    heap(currentNode) = heap(child); // move child up
    currentNode = child;          // move down a level
    child *= 2;
  }
  heap(currentNode) = lastElement;
}
```

```
template<class T>
void minHeap<T>::initialize(T *theHeap, int theSize)
{// Initialize max heap to element array theHeap(1:theSize).
  delete [] heap;
  heap = theHeap;
  heapSize = theSize;

  // heapify
  for (int root = heapSize / 2; root >= 1; root--)
  {
    T rootElement = heap(root);

    // find place to put rootElement
    int child = 2 * root; // parent of child is target
                          // location for rootElement
    while (child <= heapSize)
    {
      // heap(child) should be smaller sibling
      if (child < heapSize && heap(child) > heap(child + 1))
        child++;

      // can we put rootElement in heap(child/2)?
      if (rootElement <= heap(child))
        break;  // yes

      // no
      heap(child / 2) = heap(child); // move child up
      child *= 2;                    // move down a level
    }
    heap(child / 2) = rootElement;
```

```
  }
}

template<class T>
void minHeap<T>::output(ostream& out) const
{// Put the array into the stream out.
  copy(heap + 1, heap + heapSize + 1, ostream_iterator<T>(cout, " "));
}

// overload <<
template <class T>
ostream& operator<<(ostream& out, const minHeap<T>& x)
  {x.output(out); return out;}

#endif
```

```
// Header file name: dictionary.h
// Abstract data type specification for dictionary data structure
// all methods are pure virtual functions
// K is key type and E is value type

#ifndef dictionary_
#define dictionary_

using namespace std;

template<class K, class E>
class dictionary
{
  public:
```

```
    virtual ~dictionary() {}
    virtual bool empty() const = 0;
            // return true iff dictionary is empty
    virtual int size() const = 0;
            // return number of pairs in dictionary
    virtual pair<const K, E>* find(const K&) const = 0;
            // return pointer to matching pair
    virtual void erase(const K&) = 0;
            // remove matching pair
    virtual void insert(const pair<const K, E>&) = 0;
            // insert a (key, value) pair into the dictionary
};
#endif
```

```
// Header file name: hash.h
// Functions to convert from type K to nonnegative integer derived from
similar classes in SGI STL

#ifndef hash_
#define hash_

#include <iostream>
#include <string>

using namespace std;

template <class K> class hash;

template<>
class hash<string>
```

```
{
  public:
    size_t operator()(const string theKey) const
    {// Convert theKey to a nonnegative integer.
      unsigned long hashValue = 0;
      int length = (int) theKey.length();
      for (int i = 0; i < length; i++)
        hashValue = 5 * hashValue + theKey.at(i);

      return size_t(hashValue);
    }
};

template<>
class hash<int>
{
  public:
    size_t operator()(const int theKey) const
    {return size_t(theKey);}
};

template<>
class hash<long>
{
  public:
    size_t operator()(const long theKey) const
    {return size_t(theKey);}
};

#endif
```

```
// Header file name: sortedChain.h
// Sorted chain, implements dictionary

#ifndef sortedChain_
#define sortedChain_

#include <iostream>
#include "dictionary.h"
#include "pairNode.h"

using namespace std;

template<class K, class E>
class sortedChain : public dictionary<K,E>
{
  public:
    sortedChain() {firstNode = NULL; dSize = 0;}
    ~sortedChain();

    bool empty() const {return dSize == 0;}
    int size() const {return dSize;}
    pair<const K, E>* find(const K&) const;
    void erase(const K&);
    void insert(const pair<const K, E>&);
    void output(ostream& out) const;

  protected:
    pairNode<K,E>* firstNode;  // pointer to first node in chain
    int dSize;                 // number of elements in dictionary
```

```
};

template<class K, class E>
sortedChain<K,E>::~sortedChain()
{// Destructor.  Delete all nodes.
  while (firstNode != NULL)
  {// delete firstNode
    pairNode<K,E>* nextNode = firstNode->next;
    delete firstNode;
    firstNode = nextNode;
  }
}

template<class K, class E>
pair<const K,E>* sortedChain<K,E>::find(const K& theKey) const
{// Return pointer to matching pair.
 // Return NULL if no matching pair.
  pairNode<K,E>* currentNode = firstNode;

  // search for match with theKey
  while (currentNode != NULL &&
         currentNode->element.first != theKey)
    currentNode = currentNode->next;

  // verify match
  if (currentNode != NULL && currentNode->element.first == theKey)
    // yes, found match
    return &currentNode->element;

  // no match
  return NULL;
```

```
}

template<class K, class E>
void sortedChain<K,E>::insert(const pair<const K, E>& thePair)
{// Insert thePair into the dictionary. Overwrite existing
 // pair, if any, with same key.
  pairNode<K,E> *p = firstNode,
               *tp = NULL; // tp trails p

  // move tp so that thePair can be inserted after tp
  while (p != NULL && p->element.first < thePair.first)
  {
    tp = p;
    p = p->next;
  }

  // check if there is a matching pair
  if (p != NULL && p->element.first == thePair.first)
  {// replace old value
    p->element.second = thePair.second;
    return;
  }

  // no match, set up node for thePair
  pairNode<K,E> *newNode = new pairNode<K,E>(thePair, p);

  // insert newNode just after tp
  if (tp == NULL) firstNode = newNode;
  else tp->next = newNode;

  dSize++;
```

```
  return;
}

template<class K, class E>
void sortedChain<K,E>::erase(const K& theKey)
{// Delete the pair, if any, whose key equals theKey.
  pairNode<K,E> *p = firstNode,
                *tp = NULL; // tp trails p

  // search for match with theKey
  while (p != NULL && p->element.first < theKey)
  {
    tp = p;
    p = p->next;
  }

  // verify match
  if (p != NULL && p->element.first == theKey)
  {// found a match
    // remove p from the chain
    if (tp == NULL) firstNode = p->next;  // p is first node
    else tp->next = p->next;

    delete p;
    dSize--;
  }
}

template<class K, class E>
void sortedChain<K,E>::output(ostream& out) const
{// Insert the chain elements into the stream out.
```

```
  for (pairNode<K,E>* currentNode = firstNode;
                  currentNode != NULL;
                  currentNode = currentNode->next)
    out << currentNode->element.first << " "
        << currentNode->element.second << "  ";
}

// overload <<
template <class K, class E>
ostream& operator<<(ostream& out, const sortedChain<K,E>& x)
  {x.output(out); return out;}

#endif
```

```
// adjacency matrix representation of an undirected graph
// Program name: adjacencyGraph.h

#ifndef adjacencyGraph_
#define adjacencyGraph_

#include <iostream>
#include <sstream>
#include <iterator>
#include "adjacencyWGraph.h"
#include "unweightedEdge.h"
#include "maxHeap.h"

using namespace std;

class adjacencyGraph : public adjacencyWGraph<bool>
```

```
public:
  adjacencyGraph(int numberOfVertices = 0)
    : adjacencyWGraph<bool> (numberOfVertices, false) {}

  void insertEdge(edge<bool> *theEdge)
  {// Insert an edge.
    int v1 = theEdge->vertex1();
    int v2 = theEdge->vertex2();
    if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2)
    {
      ostringstream s;
      s << "(" << v1 << "," << v2  << ") is not a permissible edge";
      throw illegalParameterValue(s.str());
    }

    if (!a(v1)(v2))  // new edge
      e++;
    a(v1)(v2) = true;
    a(v2)(v1) = true;
  }

  bool weighted() const {return false;}

  int btMaxClique(int *maxClique)
  {// Solve max-clique problem using backtracking.
   // Set maxClique() so that maxClique(i) = 1 iff i is in max clique.
   // Return size of max clique.
    // initialize for rClique
    currentClique = new int (n + 1);
    sizeOfCurrentClique = 0;
```

```
  sizeOfMaxCliqueSoFar = 0;
  maxCliqueSoFar = maxClique;

  // find max clique
  rClique(1);
  return sizeOfMaxCliqueSoFar;
}

// structs used by max-profit branch-and-bound max clique
struct bbNode
{
  // instance data members
  bbNode* parent;
  bool leftChild;   // true iff left child of parent

  // constructors
  bbNode() {}

  bbNode(bbNode* theParent, bool theLeftChild)
  {
    parent = theParent;
    leftChild = theLeftChild;
  }
};

struct heapNode
{
  // data members
  bbNode* liveNode;
  int upperSize;    // upper bound on clique size in this subtree
  int cliqueSize;   // size of clique at this node
```

```
  int level;

  // constructors
  heapNode() {}

  heapNode(bbNode* theLiveNode, int theUpperSize,
           int theCliqueSize, int theLevel)
  {
    liveNode = theLiveNode;
    upperSize = theUpperSize;
    cliqueSize = theCliqueSize;
    level = theLevel;
  }

  operator<(const heapNode right)
    {return upperSize < right.upperSize;}

  operator int() {return upperSize;}
};

int maxProfitBBMaxClique(int *maxClique)
{// Max-profit branch-and-bound code to find a max clique.
 // maxClique(i) is set to 1 iff i is in max clique.
 // Return size of max clique.
  // initialize for level 1 start
  bbNode* eNode = NULL;
  int eNodeLevel = 1;
  int sizeOfCliqueAtENode = 0;
  int sizeOfMaxCliqueSoFar = 0;

  // search subset space tree
```

```
while (eNodeLevel != n + 1)
{// while not at leaf
  // see if vertex eNodeLevel is connected to all
  // vertices in the current clique
  bool connected = true;
  bbNode* currentNode = eNode;
  for (int j = eNodeLevel - 1; j > 0;
       currentNode = currentNode->parent, j--)
    if (currentNode->leftChild && !a(eNodeLevel)(j))
    {// j is in the clique but no edge between eNodeLevel and j
      connected = false;
      break;
    }

  if (connected)
  {// left child is feasible
    if (sizeOfCliqueAtENode + 1 > sizeOfMaxCliqueSoFar)
      sizeOfMaxCliqueSoFar = sizeOfCliqueAtENode + 1;
    addLiveNode(sizeOfCliqueAtENode + n - eNodeLevel + 1,
      sizeOfCliqueAtENode + 1, eNodeLevel + 1, eNode, true);
  }

  if (sizeOfCliqueAtENode + n - eNodeLevel >= sizeOfMaxCliqueSoFar)
    // right child has prospects
    addLiveNode(sizeOfCliqueAtENode + n - eNodeLevel,
      sizeOfCliqueAtENode, eNodeLevel + 1, eNode, false);

  // get next E-node, heap cannot be empty
  heapNode nextENode = liveNodeMaxHeap.top();
  liveNodeMaxHeap.pop();
  eNode = nextENode.liveNode;
```

```
      sizeOfCliqueAtENode = nextENode.cliqueSize;
      eNodeLevel = nextENode.level;
    }

    // construct maxClique() by following path from eNode to the root
    for (int j = n; j > 0; j--)
    {
      maxClique(j) = (eNode->leftChild) ? 1 : 0;
      eNode = eNode->parent;
    }

    return sizeOfMaxCliqueSoFar;
  }

protected:
  // recursive backtracking code to compute largest clique
  void rClique(int currentLevel)
  {// search from a node at currentLevel
    if (currentLevel > n)
    {// at leaf, found a larger clique
     // update maxCliqueSoFar and sizeOfMaxCliqueSoFar
      for (int j = 1; j <= n; j++)
        maxCliqueSoFar(j) = currentClique(j);
      sizeOfMaxCliqueSoFar = sizeOfCurrentClique;
      return;
    }

    // not at leaf; see whether vertex currentLevel
    // is connected to others in current clique
    bool connected = true;
    for (int j = 1; j < currentLevel; j++)
```

```
      if (currentClique[j] == 1 && !a[currentLevel][j])
      {// vertex currentLevel not connected to j
        connected = false;
        break;
      }

    if (connected)
    {// try left subtree
      currentClique[currentLevel] = 1;  // add to clique
      sizeOfCurrentClique++;
      rClique(currentLevel + 1);
      sizeOfCurrentClique--;
    }

    if (sizeOfCurrentClique + n - currentLevel > sizeOfMaxCliqueSoFar)
    {// try right subtree
      currentClique[currentLevel] = 0;
      rClique(currentLevel + 1);
    }
  }

  // class data members used by backtracking max clique
  static int *currentClique;
  static int sizeOfCurrentClique;
  static int sizeOfMaxCliqueSoFar;
  static int *maxCliqueSoFar;

  // class data member used by max-profit branch-and-bound max clique
  static maxHeap<heapNode> liveNodeMaxHeap;

  void addLiveNode(int upperSize, int theSize, int theLevel,
```

```
                bbNode* theParent, bool leftChild)
    {// Add a new live node to the max heap.
     // Also add the live node to the solution space tree.
     // theSize = size of clique at this live node.
     // theParent = parent of new node.
     // leftChild = true iff new node is left child of theParent.
      // create the new node of the solution space tree
      bbNode* b = new bbNode(theParent, leftChild);

      // insert corresponding node into max heap
      liveNodeMaxHeap.push(heapNode(b, upperSize, theSize, theLevel));
    }
};

int* adjacencyGraph::currentClique;
int adjacencyGraph::sizeOfCurrentClique;
int adjacencyGraph::sizeOfMaxCliqueSoFar;
int* adjacencyGraph::maxCliqueSoFar;
maxHeap<adjacencyGraph::heapNode>
adjacencyGraph::liveNodeMaxHeap;

#endif
-------------------------------------------------------------------------------------------
----------

// Test adjacency matrix representation of an undirected weighted graph
// Program name: adjacencyWGraph.h


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

```
void main(void)
{
  adjacencyWGraph<int> g(4);
  cout << "Number of Vertices = " << g.numberOfVertices() << endl;
  cout << "Number of Edges = " << g.numberOfEdges() << endl;
  cout << endl;

  g.insertEdge(new weightedEdge<int>(2, 4, 1));
  g.insertEdge(new weightedEdge<int>(1, 3, 2));
  g.insertEdge(new weightedEdge<int>(2, 1, 3));
  g.insertEdge(new weightedEdge<int>(1, 4, 4));
  g.insertEdge(new weightedEdge<int>(4, 2, 5));
  cout << "The graph is" << endl;
  cout << "Number of Vertices = " << g.numberOfVertices() << endl;
  cout << "Number of Edges = " << g.numberOfEdges() << endl;
  cout << g << endl;
  cout << endl;

  g.eraseEdge(2,1);
  cout << "The graph after deleting (2,1) is" << endl;
  cout << "Number of Vertices = " << g.numberOfVertices() << endl;
  cout << "Number of Edges = " << g.numberOfEdges() << endl;
  cout << g << endl;

  cout << "existsEdge(3,1) = " << g.existsEdge(3,1) << endl;
  cout << "existsEdge(1,3) = " << g.existsEdge(1,3) << endl;
```

```
  cout << "inDegree(3) = " << g.inDegree(3) << endl;
  cout << "outDegree(1) = " << g.outDegree(1) << endl;
  cout << "Number of Vertices = " << g.numberOfVertices() << endl;
  cout << "Number of Edges = " << g.numberOfEdges() << endl;
  cout << endl;

  // test iterator
  cout << "Edges incident to vertex 4" << endl;
  vertexIterator<int>* gi = g.iterator(4);
  pair<int,int>* thePair;
  while((thePair = gi->next()) != NULL)
    cout << thePair->first << " " << thePair->second << endl;
}
```

```
// Program adjacency matrix representation of a weighted directed graph
// Header file program name: adjacencyWDigraph.h

#ifndef adjacencyWDigraph_
#define adjacencyWDigraph_

#include <iostream>
#include <sstream>
#include <iterator>
#include "graph.h"
#include "weightedEdge.h"
```

```
#include "vertexIterator.h"
#include "make2dArrayNoCatch.h"
#include "delete2dArray.h"
#include "myExceptions.h"
#include "arrayQueue.h"
#include "graphChain.h"
#include "minHeap.h"

using namespace std;
template <class T>
class adjacencyWDigraph : public graph<T>
{
  protected:
    int n;          // number of vertices
    int e;          // number of edges
    T **a;          // adjacency array
    T noEdge;       // denotes absent edge

  public:
    adjacencyWDigraph(int numberOfVertices = 0, T theNoEdge = 0)
    {// Constructor.
      // validate number of vertices
      if (numberOfVertices < 0)
        throw illegalParameterValue("number of vertices must be >= 0");
      n = numberOfVertices;
      e = 0;
```

```
  noEdge = theNoEdge;
  make2dArray(a, n + 1, n + 1);
  for (int i = 1; i <= n; i++)
    // initialize adjacency matrix
    fill(a[i], a[i] + n + 1, noEdge);
}
~adjacencyWDigraph() {delete2dArray(a, n + 1);}
int numberOfVertices() const {return n;}

int numberOfEdges() const {return e;}
bool directed() const {return true;}
bool weighted() const {return true;}
bool existsEdge(int i, int j) const
{// Return true iff (i,j) is an edge of the graph.
  if (i < 1 || j < 1 || i > n || j > n || a[i][j] == noEdge)
    return false;
  else
    return true;
}

void insertEdge(edge<T> *theEdge)
{// Insert edge theEdge into the digraph; if the edge is already
 // there, update its weight to theEdge.weight().
  int v1 = theEdge->vertex1();
  int v2 = theEdge->vertex2();
  if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2)
```

```
  {
    ostringstream s;
    s << "(" << v1 << "," << v2 << ") is not a permissible edge";
    throw illegalParameterValue(s.str());
  }
    if (a[v1][v2] == noEdge)  // new edge
    e++;
  a[v1][v2] = theEdge->weight();
}

void eraseEdge(int i, int j)
{// Delete the edge (i,j).
  if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] != noEdge)
  {
    a[i][j] = noEdge;
    e--;
  }
}

void checkVertex(int theVertex) const
{// Verify that i is a valid vertex.
  if (theVertex < 1 || theVertex > n)
  {
    ostringstream s;
    s << "no vertex " << theVertex;
    throw illegalParameterValue(s.str());
```

```
  }
}

int degree(int theVertex) const
  {throw undefinedMethod("degree() undefined");}

int outDegree(int theVertex) const
{// Return out-degree of vertex theVertex.
  checkVertex(theVertex);

  // count out edges from vertex theVertex
  int sum = 0;
  for (int j = 1; j <= n; j++)
    if (a[theVertex][j] != noEdge)
      sum++;

  return sum;
}

int inDegree(int theVertex) const
{// Return in-degree of vertex theVertex.
  checkVertex(theVertex);

  // count in edges at vertex theVertex
  int sum = 0;
  for (int j = 1; j <= n; j++)
```

```
    if (a[j](theVertex) != noEdge)
      sum++;

  return sum;
}

class myIterator : public vertexIterator<T>
{
  public:
    myIterator(T* theRow, T theNoEdge, int numberOfVertices)
    {
      row = theRow;
      noEdge = theNoEdge;
      n = numberOfVertices;
      currentVertex = 1;
    }

    ~myIterator() {}
     int next()
    {// Return next vertex if any. Return 0 if no next vertex.
      // find next adjacent vertex
      for (int j = currentVertex; j <= n; j++)
        if (row[j] != noEdge)
        {
          currentVertex = j + 1;
          return j;
```

```
    }
  // no next adjacent vertex
  currentVertex = n + 1;
  return 0;
}

int next(T& theWeight)
{// Return next vertex if any. Return 0 if no next vertex.
 // Set theWeight = edge weight.
  // find next adjacent vertex
  for (int j = currentVertex; j <= n; j++)
    if (row[j] != noEdge)
    {
      currentVertex = j + 1;
      theWeight = row[j];
      return j;
    }
  // no next adjacent vertex
  currentVertex = n + 1;
  return 0;
}

protected:
  T* row;          // row of adjacency matrix
  T noEdge;        // theRow(i) == noEdge iff no edge to i
  int n;           // number of vertices
```

```
    int currentVertex;
};

myIterator* iterator(int theVertex)
{// Return iterator for vertex theVertex.
  checkVertex(theVertex);
  return new myIterator(a[theVertex], noEdge, n);
}

void output(ostream& out) const
{// Output the adjacency matrix.
  for (int i = 1; i <= n; i++)
  {
    copy(a[i] + 1, a[i] + n + 1, ostream_iterator<T>(out, "  "));
    out << endl;
  }
}

void bfs(int v, int reach[], int label)
{// Breadth-first search. reach[i] is set to label for
 // all vertices reachable from vertex v.
  arrayQueue<int> q(10);
  reach[v] = label;
  q.push(v);
  while (!q.empty())
  {
```

```
    // remove a labeled vertex from the queue
    int w = q.front();
    q.pop();

    // mark all unreached vertices adjacent from w
    for (int u = 1; u <= n; u++)
      // visit an adjacent vertex of w
      if (a[w][u] != noEdge && reach[u] == 0)
      {// u is an unreached vertex
        q.push(u);
        reach[u] = label; // mark reached
      }
  }
}

void shortestPaths(int sourceVertex,
                   T* distanceFromSource, int* predecessor)
{// Find shortest paths from sourceVertex.
 // Return shortest distances in distanceFromSource.
 // Return predecessor information in predecessor.
  if (sourceVertex < 1 || sourceVertex > n)
    throw illegalParameterValue("Invalid source vertex");

  if (!weighted())
    throw undefinedMethod
```

```
        ("adjacencyWDigraph::shortestPaths() not defined for unweighted
graphs");

      graphChain<int> newReachableVertices;

      // initialize
      for (int i = 1; i <= n; i++)
      {
        distanceFromSource[i] = a[sourceVertex][i];
        if (distanceFromSource[i] == noEdge)
          predecessor[i] = -1;
        else
        {
          predecessor[i] = sourceVertex;
          newReachableVertices.insert(0, i);
        }
      }
      distanceFromSource[sourceVertex] = 0;
      predecessor[sourceVertex] = 0;  // source vertex has no predecessor

      // update distanceFromSource and predecessor
      while (!newReachableVertices.empty())
      {// more paths exist
        // find unreached vertex v with least distanceFromSource
        chain<int>::iterator iNewReachableVertices
                        = newReachableVertices.begin();
```

```
chain<int>::iterator theEnd = newReachableVertices.end();
int v = *iNewReachableVertices;
iNewReachableVertices++;
while (iNewReachableVertices != theEnd)
{
  int w = *iNewReachableVertices;
  iNewReachableVertices++;
  if (distanceFromSource(w) < distanceFromSource(v))
    v = w;
}

// next shortest path is to vertex v, delete v from
// newReachableVertices and update distanceFromSource
newReachableVertices.eraseElement(v);
for (int j = 1; j <= n; j++)
{
  if (a(v)(j) != noEdge && (predecessor(j) == -1 ||
 distanceFromSource(j) > distanceFromSource(v) + a(v)(j)))
  {
    // distanceFromSource(j) decreases
    distanceFromSource(j) = distanceFromSource(v) + a(v)(j);
    // add j to newReachableVertices
    if (predecessor(j) == -1)
      // not reached before
      newReachableVertices.insert(0, j);
    predecessor(j) = v;
```

```
      }
    }
  }
}

template <class T>
void allPairs(T **c, int **kay)
{// Dynamic programming all pairs shortest paths algorithm.
 // Compute c(i)(j) and kay(i)(j) for all i and j.
  if (!weighted())
    throw undefinedMethod
    ("adjacencyWDigraph::allPairs() not defined for unweighted graphs");

  // initialize c(i)(j) = c(i,j,0)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
    {
      c(i)(j) = a(i)(j);
      kay(i)(j) = 0;
    }
  for (int i = 1; i <= n; i++)
    c(i)(i) = 0;

  // compute c(i)(j) = c(i,j,k)
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
```

```
            for (int j = 1; j <= n; j++)
              if (c[i][k] != noEdge && c[k][j] != noEdge &&
                (c[i][j] == noEdge || c[i][j] > c[i][k] + c[k][j]))
              {// smaller value for c[i][j] found
                 c[i][j] = c[i][k] + c[k][j];
                 kay[i][j] = k;
              }
        }

        T btSalesperson(int *bestTour)
        {// Traveling salesperson by backtracking.
         // bestTour[1:n] is set to best tour.
         // Return cost of best tour.
          if (!weighted())
            throw undefinedMethod
            ("adjacencyWDigraph::btSalesperson() not defined for unweighted
graphs");
          // set partialTour to identity permutation
          partialTour = new int [n + 1];
          for (int i = 1; i <= n; i++)
            partialTour[i] = i;

          costOfBestTourSoFar = noEdge;
          bestTourSoFar = bestTour;
          costOfPartialTour = 0;
```

```
  // search permutations of partialTour(2:n)
  rTSP(2);

  return costOfBestTourSoFar;
}

// struct used by least-cost branch-and-bound traveling salesperson
struct heapNode
{
  // data members
  T lowerCost;          // lower bound on cost of tours in subtree
  T costOfPartialTour;  // cost of partial tour
  T minAdditionalCost;  // min additional cost to complete tour
  int sizeOfPartialTour; // partial tour is
  int *partialTour;     // partialTour(sizeOfPartialTour+1:n-1)
                        // gives remaining vertices to be added
                        // to partialTour(0:sizeOfPartialTour)
  // constructors
  heapNode() {}

  heapNode(T lC, T cOPT, T mAC, int sOPT, int* pT)
  {
    lowerCost = lC;
    costOfPartialTour = cOPT;
    minAdditionalCost = mAC;
    sizeOfPartialTour = sOPT;
```

```
        partialTour = pT;
      }
      operator int() {return lowerCost;}
      operator>(const heapNode right)
        {return lowerCost > right.lowerCost;}
    };

    T leastCostBBSalesperson(int *bestTour)
    {// least-cost branch-and-bound code to find a shortest tour
     // bestTour(i) set to i'th vertex on shortest tour
     // Return cost of shortest tour.
      if (!weighted())
        throw undefinedMethod
        ("adjacencyWDigraph::leastCostBBSalesperson() not defined for
unweighted graphs");

      minHeap<heapNode> liveNodeMinHeap;

      // costOfMinOutEdge(i) = cost of least-cost edge leaving vertex i
      T *costOfMinOutEdge = new T (n + 1);
      T sumOfMinCostOutEdges = 0;
      for (int i = 1; i <= n; i++)
      {// compute costOfMinOutEdge(i) and sumOfMinCostOutEdges
        T minCost = noEdge;
        for (int j = 1; j <= n; j++)
          if (a(i)(j) != noEdge && (minCost == noEdge ||
```

```
        minCost > a[i][j]))
      minCost = a[i][j];

  if (minCost == noEdge) return noEdge; // no route
  costOfMinOutEdge[i] = minCost;
  sumOfMinCostOutEdges += minCost;
}

// initial E-node is tree root
heapNode eNode(0, 0, sumOfMinCostOutEdges, 0, new int [n]);
for (int i = 0; i < n; i++)
  eNode.partialTour[i] = i + 1;
T costOfBestTourSoFar = noEdge;          // no tour found so far
int *partialTour = eNode.partialTour;    // shorthand for
                                         // eNode.partialTour
// search permutation tree
while (eNode.sizeOfPartialTour < n - 1)
{// not at leaf
  partialTour = eNode.partialTour;
  if (eNode.sizeOfPartialTour == n - 2)
  {// parent of leaf
    // complete tour by adding two edges
    // see whether new tour is better
    if (a[partialTour[n - 2]][partialTour[n - 1]] != noEdge
      && a[partialTour[n - 1]][1] != noEdge
      && (costOfBestTourSoFar == noEdge ||
```

```
        eNode.costOfPartialTour
        + a(partialTour(n - 2))(partialTour(n - 1))
        + a(partialTour(n - 1))(1)
        < costOfBestTourSoFar))
    {// better tour found
      costOfBestTourSoFar = eNode.costOfPartialTour
          + a(partialTour(n - 2))(partialTour(n - 1))
          + a(partialTour(n - 1))(1);
      eNode.costOfPartialTour = costOfBestTourSoFar;
      eNode.lowerCost = costOfBestTourSoFar;
      eNode.sizeOfPartialTour++;
      liveNodeMinHeap.push(eNode);
    }
  }
  else
  {// generate children
    for (int i = eNode.sizeOfPartialTour + 1; i < n; i++)
      if (a(partialTour(eNode.sizeOfPartialTour))
          (partialTour(i)) != noEdge)
      {
        // feasible child, bound path cost
        T costOfPartialTour = eNode.costOfPartialTour
            + a(partialTour(eNode.sizeOfPartialTour))
              (partialTour(i));
        T minAdditionalCost = eNode.minAdditionalCost
              - costOfMinOutEdge(partialTour
```

```
                (eNode.sizeOfPartialTour));
        T leastCostPossible = costOfPartialTour
                        + minAdditionalCost;
        if (costOfBestTourSoFar == noEdge ||
         leastCostPossible < costOfBestTourSoFar)
        {// subtree may have better leaf, put root in min heap
          heapNode hNode(leastCostPossible,
                        costOfPartialTour,
                        minAdditionalCost,
                        eNode.sizeOfPartialTour + 1,
                        new int (n));
          for (int j = 0; j < n; j++)
            hNode.partialTour(j) = partialTour(j);
          hNode.partialTour(eNode.sizeOfPartialTour + 1) =
                partialTour(i);
          hNode.partialTour(i) =
                partialTour(eNode.sizeOfPartialTour + 1);
          liveNodeMinHeap.push(hNode);
        }
      }
    }

// get next E-node
delete () eNode.partialTour;
if (liveNodeMinHeap.empty()) break;
eNode = liveNodeMinHeap.top();
```

```
        liveNodeMinHeap.pop();
    }

    if (costOfBestTourSoFar == noEdge)
      return NULL; // no route

    // copy best route into bestTour[1:n]
    for (int i = 0; i < n; i++)
      bestTour[i + 1] = partialTour[i];

    return costOfBestTourSoFar;
  }

protected:
  void rTSP(int currentLevel)
  {// Recursive backtracking code for traveling salesperson.
   // Search the permutation tree for best tour. Start at a node
   // at currentLevel.
    if (currentLevel == n)
    {// at parent of a leaf
      // complete tour by adding last two edges
      if (a[partialTour[n - 1]][partialTour[n]] != noEdge &&
          a[partialTour[n]][1] != noEdge &&
          (costOfBestTourSoFar == noEdge ||
           costOfPartialTour + a[partialTour[n - 1]][partialTour[n]]
           + a[partialTour[n]][1] < costOfBestTourSoFar))
```

```
    {// better tour found
      copy(partialTour + 1, partialTour + n + 1, bestTourSoFar + 1);
      costOfBestTourSoFar = costOfPartialTour
                  + a(partialTour(n - 1))(partialTour(n))
                  + a(partialTour(n))(1);
    }
  }
  else
  {// try out subtrees
    for (int j = currentLevel; j <= n; j++)
      // is move to subtree labeled partialTour(j) possible?
      if (a(partialTour(currentLevel - 1))(partialTour(j)) != noEdge
        && (costOfBestTourSoFar == noEdge ||
         costOfPartialTour +
          a(partialTour(currentLevel - 1))(partialTour(j))
          < costOfBestTourSoFar))
      {// search this subtree
        swap(partialTour(currentLevel), partialTour(j));
        costOfPartialTour += a(partialTour(currentLevel - 1))
                    (partialTour(currentLevel));
        rTSP(currentLevel + 1);
        costOfPartialTour -= a(partialTour(currentLevel - 1))
                    (partialTour(currentLevel));
        swap(partialTour(currentLevel), partialTour(j));
      }
  }
```