```
template<class TV, class TE>
class graphVertex
{
  private:
    int vKey;
    TV vContent;
    list < graphEdge <TV,TE> > eList;

  public:
    graphVertex();
    graphVertex(const TV& vContent, int vKey=-1);
    graphVertex(const graphVertex<TV,TE>& v);
    ~graphVertex();

    TV getVContent() const ;
    void setVContent(const TV& vContent);

    int getVKey() const ;
    void setVKey(int vKey) ;

    bool getEdgeContentByVDestination(TE& eContent, typename
                                      list<graphVertex <TV,TE> >::iterator vDestination);

    typename list < graphEdge <TV,TE> >::iterator getAdjacenciesBegin() ;
    typename list < graphEdge <TV,TE> >::iterator getAdjacenciesEnd()  ;

    int getAdjacenciesSize() const ;
    void addAdjacency( const TE& econtent, typename
                                list< graphVertex <TV,TE> >::iterator vDestination );
    bool operator ==(const graphVertex <TV,TE> &v) const;
    void write(ostream &o) const;
};


template<class TV,class TE>
class graphEdge
{
  private:
     TE eContent;
     typename list < graphVertex <TV,TE> >::iterator vDestination;

  public:
    graphEdge();
    graphEdge(const TE& eContent, typename list < graphVertex <TV,TE> >::iterator
                                                                    vDestination);
    graphEdge(const graphEdge<TV,TE>& e);
    ~graphEdge();

    TE getEContent() const ;
    void setEContent(const TE& eContent);

    typename list < graphVertex <TV,TE> >::iterator getVDestination();
    void setVDestination(typename list <graphVertex <TV,TE>>::iterator vDestination);
    void write(ostream &o) const;
};
```

```cpp
template<class TV,class TE>
class graphStl
{
private:
  TE infinite; // Dijkstra's Infinite
  int keys;

protected:
  list < graphVertex <TV,TE> > vlist;  //Vertice list

  TE getInfinite() const;
  virtual bool compareVertices(TV const &vContent1, TV const &vContent2);

  bool getVertexIteratorByContent(typename list<graphVertex <TV,TE> >::iterator
                                            &vIterator, const TV &vContent );
  bool getVertexIteratorByKey(typename list < graphVertex <TV,TE> >::iterator &itv,
                                                                  int vKey );
public:
  graphStl();
  bool getVertexContentBySubContent(TV &vContent, const TV &vSubContent );
  bool getVertexContentByKey(TV &vContent, int vKey );
  bool getVertexKeyByContent(int &vKey, const TV &vContent);
  bool getEdgeByVertexContents(TE &eContent, const TV &vOrigin, const TV &vDestination);
  bool getEdgeByVertexKeys(TE &eContent, int vKeyOrigin, int vKeyDestination);

  int getEntranceDegree(const TV& vContent);
  int getExitDegree(const TV& vContent);

  bool addGraphVertex(const TV& vContent);
  bool addGraphEdge(const TE& eContent, const TV& vOrigin, const TV& vDestination);

  virtual void write(ostream &o);
};


template<class TV,class TE>
class graphStlPath : public graphStl <TV,TE>
{
  protected:
    void lengthFirstVisitRecursive(typename list<graphVertex <TV,TE>>::iterator itv,
                                   bitset <MAX_VERTICES> &taken, queue <TV> &q) const;
    void distinctPathsRecursive(typename list < graphVertex <TV,TE> >::iterator itvo,
                                typename list < graphVertex <TV,TE> >::iterator itvd,
                                bitset <MAX_VERTICES> &taken, stack <TV> &s,
                                queue<stack<TV>> &qr);

  public:
      graphStlPath();

      queue <TV> lengthFirstVisit(const TV &vContent);
      queue <TV> breadthFirstVisit(const TV &vContent);
      queue < stack <TV> > distinctPaths(const TV &vOrigin, const TV &vDestination);
      bool dijkstrasAlgorithm(const TV &vContent, vector<int> &pathKeys, vector<TE> &dist);
      queue <TV> getDijkstrasPath(int vKeyOrigin, vector <int> pathKeys);
};
```

```
template<class TN>
class treeNode
{
  private:
    TN key;
    treeNode <TN> *left, *right;
    int hLeft, hRight;

   public:
    treeNode();
    treeNode(const TN &key, int hLeft=0, int hRight=0, treeNode <TN> *left=NULL,
                                                       treeNode <TN> *right=NULL );
    treeNode(const treeNode <TN> &n);
    ~treeNode();

    TN getKey() const;

    treeNode <TN> *getLeft() const;
    void setLeft(treeNode <TN> *left);

    treeNode <TN> *getRight() const;
    void setRight(treeNode <TN> *right);

    int getLeftHeight() const;
    void setLeftHeight(int lh);
    void setLeftSide(treeNode <TN> *left, int lh);

    int getRightHeight() const;
    void setRightHeight(int rh);
    void setRightSide(treeNode <TN> *right, int rh);

    int getBalanceFactor() const;
};


template<class TN>
class bst
{
  protected:
    treeNode <TN> *root;
    int nodeCount;

    virtual bool lessThanKeys(const TN &key1, const TN &key2);
    virtual bool equalToKeys(const TN &key1, const TN &key2);

    treeNode <TN> *copyRecursive(const treeNode <TN> *node);
    int getTreeHeight(treeNode <TN> *node) const;

    treeNode <TN> * deleteAndUpdate(treeNode <TN> *toDelete, treeNode <TN> *node,
                                    treeNode <TN> * &substitute);
    treeNode <TN> * popOneNode(treeNode <TN> *node);
    virtual treeNode <TN> * balanceNode(treeNode <TN> *node) { return node; }
    treeNode <TN> * pushRecursive(treeNode <TN> *node, const TN &key, bool &isInserted);
    treeNode <TN> * popRecursive(treeNode <TN> *node, const TN &key, bool &isDeleted);
    void clearRecursive(treeNode <TN> *node);
```

```cpp
  public:
    bst();
    bst(const bst<TN> &bst);
    ~bst();

    int getTreeHeight() const;
    int size() const;
    bool push(const TN &key);
    bool pop(const TN &key);
    void clear();

    bstIterator<TN> begin();
    bstIterator<TN> end();
};


template<class TN>
class avlst : public bst <TN>
{
  private:
      treeNode <TN> * RotationLeft (treeNode <TN> * node);
      treeNode <TN> * RotationRight (treeNode <TN> * node);
      treeNode <TN> * twoRotations (treeNode <TN> * node);
      treeNode <TN> * balanceNode (treeNode <TN> * node);

      bool perfeitEquilib(treeNode <TN>* node) ;
      void byLevel(treeNode <TN>* rz) const ;
  public:
      avlst();
      avlst(const avlst<TN> &a);
      ~avlst() ;
};



template<class TN>
class bstIterator
{
  protected:
     treeNode <TN> *now;

  public:
    bstIterator();
    bstIterator(treeNode <TN> *now);
    bstIterator(const bstIterator<TN> &i);

    TN operator*() const;
    TN operator->() const;
    bool operator==(const bstIterator<TN> &i) const;
    bool operator!=(const bstIterator<TN> &i) const;

    virtual bstIterator<TN> & operator++(int) { return *this;}; // dummy
    virtual bstIterator<TN> & operator=(const bstIterator<TN> &i);
};
```

```
template<class TN>
class bstIteratorInOrder : public bstIterator<TN>
{
  private:
       stack <treeNode <TN> * > unVisited;
       void findFirstElement();

   public:
       bstIteratorInOrder();
       bstIteratorInOrder(const bstIterator<TN> &i);
       bstIteratorInOrder(const bstIteratorInOrder<TN> &i);

       bstIterator<TN> & operator++(int);
       bstIterator<TN> & operator=(const bstIterator<TN> &i) ;
};



template<class TN>
class bstIteratorPreOrder : public bstIterator<TN>
{
   private:
       stack <treeNode <TN> * > unVisited;

   public:
       bstIteratorPreOrder();
       bstIteratorPreOrder(const bstIterator<TN> &i);
       bstIteratorPreOrder(const bstIteratorPreOrder<TN> &i);

       bstIterator<TN> & operator++(int);
       bstIterator<TN> & operator=(const bstIterator<TN> &i);
};



template<class TN>
class bstIteratorLevel : public bstIterator<TN>
{
  private:
       queue <treeNode <TN> * > unVisited;

  public:
       bstIteratorLevel();
       bstIteratorLevel(const bstIterator<TN> &i);
       bstIteratorLevel(const bstIteratorLevel<TN> &i);

       bstIterator<TN> & operator++(int);
       bstIterator<TN> & operator=(const bstIterator<TN> &i);
};
```

```
template<class TN>
class bstIteratorPosOrder : public bstIterator<TN>
{
  private:
       stack <treeNode <TN> * > unVisited;
       stack <bool> unVisitedRight;

       void findDeepestPosOrder();

  public:
       bstIteratorPosOrder();
       bstIteratorPosOrder(const bstIterator &i);
       bstIteratorPosOrder(const bstIteratorPosOrder &i);

       virtual bstIterator<TN> & operator++(int);
       bstIterator<TN> & operator=(const bstIterator<TN> &i);
};
```