```
//=============================================================
// header.cpp
// This file contains several header files
//=============================================================

//=============================================================
// LIST AS ARRAY HEADER
//=============================================================
template <class T>
class List
{
private:
        T               *array;
        int             DEFAULT_LIST_SIZE = 10;
        int             capacity;
        int     size;

public:

                                List                (void);             //default
                List                (const List<T> &c);
                ~List               (void);             //destructor
        bool            isEmpty                 (void);
        int             length              (void) const;
        T&                      operator[]              (int i);        //index operator
        string          toString            (void)const ;
        void            append              (T c);
        void            insert              (T, int);
        void            remove              (int);
        List<T>         operator+               (const List<T> c) const ;
        List<T>         operator=               (List<T> c) const ;
        void        clear                   (void);
  friend ostream& operator<<    (ostream &os, List<T> c)
  {
    for (int i = 0; i < c.size; i++)
      os << c[i] << " ";

    return os;
  }
};
class IndexError { };

//=============================================================
// MINHEAP HEADER
//=============================================================

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);
    MinHeap(KeyType initA[], int n);
    MinHeap(const MinHeap<KeyType>& heap);
    ~MinHeap();

    void heapSort(KeyType sorted[]);

    MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);
    std::string toString() const;

  private:
    KeyType *A;     // array containing the heap
    int heapSize;   // size of the heap
```

```
    int capacity;   // size of A

        void heapify(int index);
    void buildHeap();
        int leftChild(int index) { return 2 * index + 1; }
        int rightChild(int index) { return 2 * index + 2; }
        int parent(int index) { return (index - 1) / 2; }
    void swap(int index1, int index2);
    void copy(const MinHeap<KeyType>& heap);
    void destroy();
};

//=============================================================
// HASH TABLE HEADER
//=============================================================
template <class KeyType>
class HashTable
{
public:
                                                HashTable       (int numSlots);
                                                HashTable       (const HashTable<KeyType>& h);
                                                ~HashTable      ();
        KeyType*                        get                     (const KeyType& k) const;
        void                            insert          (KeyType *k);
        void                            remove          (const KeyType& k);
        HashTable<KeyType>& operator=   (const HashTable<KeyType>& h);
        std::string toString                                    (int slot) const;

private:
        int                     slots;
        List<KeyType>   *table; // an array of List<KeyType>â\200\231s
};


//=============================================================
// RBT HEADER
//=============================================================

template <class KeyType>
class Node
{
public:
        KeyType         *data;
        Node                    *left;
        Node                    *right;
        Node                    *p;
        string          color;

        Node            (void)                          //default constructor
        {
                data = NULL;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
        };
        Node            (KeyType *item) //constructor with item
        {
                data = item;
                left = NULL;
                right = NULL;
                p = NULL;
                color = "red";
```

```
        };
        Node            (KeyType *item, Node<KeyType>* nil)//nil pointer
        {                                                                       //cons
tuctor
                data = item;
                left = nil;
                right = nil;
                p = nil;
                color = "red";
        };


};

template <class KeyType>
class RBT
{
protected:
        void    RBTFix                          (Node<KeyType> *current);
        void    LeftRotate                      (Node<KeyType> *z);
        void    RightRotate                     (Node<KeyType> *z);
        string  inOrderHelper   (Node<KeyType> *z,stringstream &s);
        string  preOrderHelper  (Node<KeyType> *z, stringstream &s);
        string  postOrderHelper (Node<KeyType> *z, stringstream &s);
        Node<KeyType>*  copy                                    (Node<KeyType> *z);
        void                                            clear           (Node<KeyType> *z);
        Node<KeyType>*  find    (Node<KeyType> *r, KeyType k) const;

public:

        Node<KeyType> * root;     // root pointer for red black tree
        Node<KeyType> * nil;    // null pointer for leaf nodes

                        RBT                     (void);
                ~RBT                            (void);
                        RBT             (const RBT<KeyType> & r);

    bool                Empty                   (void) const;
    KeyType             *get                    (const KeyType& k);
    void                insert                  (KeyType *k);
    void                remove                  (const KeyType& k);
    KeyType             *maximum                (void) const;
    KeyType             *minimum                (void) const;
    KeyType             *successor              (const KeyType& k)const;
    KeyType             *predecessor            (const KeyType& k)const;
    string              inOrder                 (void) const;
    string              preOrder                (void) const;
    string              postOrder               (void) const;

};


class EmptyError {};
//============================================================
// MOVIE CLASS
//============================================================

class Movie
{
public:
  string title;
  string cast;
```

```
  bool operator< ( Movie& a) const
  {
    return this->title <= a.title;
  };
  bool operator> (const Movie& a) //overloading comparison oper.
  {                               //to compare key specifically
    return this->title > a.title;
  };
  bool operator== (const Movie& a)
  {
    return this->title == a.title;
  };
  bool operator!= (const Movie& a)
  {
    return this->title != a.title;
  };

  friend ostream & operator<< (ostream &os, const Movie &mov)
  {                                       //overloading
      os << mov.cast;                                          //cout operator
      return os;
  };
};
//================================================================
//movieDict
//takes string of text file as parameter and reads in and
//separates
//and creates a dictionary of movie titles and cast
//lists and returns
//said dictionary
//================================================================
Dictionary<Movie> movieDict(string movieFile)
{
  ifstream file;
  file.open(movieFile);
  string line;
  Dictionary<Movie> movieDictionary;
  while (file)
  {
    getline(file, line);
    int index = line.find('\t');
    Movie *mov = new Movie;
    mov->title = line.substr(0, index);
    mov->cast = line.substr(index+1);
    //cout << mov->title << endl;
    movieDictionary.insert(mov);
  }
  file.close();
  return movieDictionary;
}

//================================================================
// DICTIONARY
//================================================================
template <class KeyType>
class Dictionary: public HashTable<KeyType>
{
/*
public:

              Dictionary  (){};         //default constructor
              ~Dictionary (void){};    //destructor
              Dictionary  (const Dictionary<KeyType> &d){};
```

```
  void        insert      (KeyType *k)
  {
    HashTable<KeyType>::insert(k);
  };
  void        remove      (const KeyType &k)
  {
    HashTable<KeyType>::remove(k);
  };
  KeyType *   get         (const KeyType &k) const
  {
    HashTable<KeyType>::get(k);
  };
  bool        Empty       (void)  const
  {
    HashTable<KeyType>::Empty();
  };

private:
  HashTable<KeyType> *d;
*/

public:

        Dictionary():HashTable<KeyType>(){};
        ~Dictionary(void) {};
Dictionary(const Dictionary<KeyType> &d):HashTable<KeyType>(d){};

//Inherited HashTable Class incorrectly, much simpler this way.
};
```