Hannah Kerr and Emma Steinman
CS 271 Project 6
November 1, 2017

1.a To prove that a complete binary tree with height h will have $2^{h+1} - 1$ total nodes we will start with a base case where $h = 0$. A complete binary tree with height zero is just the root, which has $2^{0+1} - 1 = 1$ total nodes. Assume that all complete binary trees with heigh k, where $0 \leq k \leq h - 1$, have $2^{h+1} - 1$ total nodes. Now assume we have a complete binary tree with height h. Each of the subtrees of the root have height $h - 1$. Since this is in the range of k, we can apply the inductive hypothesis, so each subtree has $2^{(h-1)+1} - 1 = 2^h - 1$ total nodes. Thus there are $2(2^h - 1) + 1 = 2^{h+1} - 1$ total nodes in the complete binary tree with height h.

1.b To prove that a complete binary tree with n nodes will have $\frac{n-1}{2}$ internal nodes, we start with a base case where $n = 1$. A complete binary tree with one node is just the root, which is not an internal node if it has no children. Thus the complete binary tree has $\frac{1-1}{2} = 0$ internal nodes. Assume this is true for all complete binary trees with k nodes, where $0 \leq k \leq n - 1$. Now assume we have a complete binary tree with n nodes. Each of the subrees will have $\frac{n-1}{2}$ nodes and since this is in the range of k we can apply the inductive hypothesis. Thus each subtree will have $\frac{\frac{n-1}{2}-1}{2} = \frac{n-1-2}{4} = \frac{n-3}{4}$ internal nodes and the entire tree will have $2(\frac{n-3}{4})+1 = \frac{n-3}{2}+1 = \frac{n-3+2}{2} = \frac{n-1}{2}$ internal nodes in the complete binary tree with n nodes.

2 For a node x with an empty right subtree, to find the successor, y you find the ancestor up to the left as far as possible and then one up to the right. This, y will be the lowest ancestor of x whose left child is also an ancestor of x because the left child of the successor is as far up and to the left you can travel from x. The successor, y, is the lowest ancestor of x whose left child is also an ancestor of x because the left child of the left child of the successor will not be on the path travelling up and to the left from x.
If x is the child of its successor y, it still upholds the properties that the successor is the lowest ancestor of x whose left child is also an ancestor of x because every node is an ancestor to itself.
Alternatively, if y is the successor of x, x is the predecessor of y. To find the predecessor when the left subtree exists, you find the greatest item in the left subtree. x will be in the left subtree because it exists and the predecessor will be less than the successor. Once in the left subtree, travelling all the way to the right will give you x because as the predecessor of y, x is the greatest element in the left subtree.

```
//---------------------------------------------------------------------------
// Hannah Kerr and Emma Steinman
//bst.h
//method declarations for the bst class
//November 1, 2017
//---------------------------------------------------------------------------


#include <iostream>
#include <stdlib.h>
#include <sstream>
#include <string>
using namespace std;
// =============================================================
#ifndef BST_H
#define BST_H

template <class KeyType>
struct Node
{
    KeyType *data;
    Node *  left;
    Node *  right;

};

template <class KeyType>
class BST
{
protected:
  string          inHelper      (Node<KeyType> * r, stringstream &s)const;
  string          preHelper     (Node<KeyType> * r, stringstream &s) const;
  string          postHelper    (Node<KeyType> * r, stringstream &s) const;
  Node<KeyType>*  find           (Node<KeyType> *ptr, KeyType item) const;
  Node<KeyType>   *copy       (Node<KeyType> *ptr);
  Node<KeyType>   *findParent   (Node<KeyType> *, KeyType item) const;
  void            clear       (Node<KeyType> *&ptr );
public:
    Node<KeyType> * root;        // root pointer for binary search tree
                BST          (void);
               ~BST          (void);
                BST          (const BST<KeyType> & );

    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;

    BST<KeyType>    operator=   (const BST<KeyType> &ptr);

    friend ostream & operator<< ( ostream &os, const BST<KeyType> &bst )
    {
        os << bst.inOrder();
        return os;
    };
};
```

```
#include "bst.cpp"
#endif
```

```
//--------------------------------------------------------------------------
//Hannah Kerr and Emma Steinman
//bst.cpp
//implements methods for bst class
//November 1, 2017
//--------------------------------------------------------------------------


#include <string>
#include <sstream>
using namespace std;


//----------------------------------------------------------------
// default constructor
// creates an empty tree with root set to NULL
//----------------------------------------------------------------
template <class KeyType>
BST<KeyType>::BST (void)
{
        root = NULL;
}
//----------------------------------------------------------------
// destructor
//----------------------------------------------------------------
template<class KeyType>
BST<KeyType>::~BST (void)
{
        Node<KeyType> *ptr = root;
  if ( ptr != NULL )
        {
                clear(ptr->left);
                clear(ptr->right);
                ptr = NULL;
                delete ptr;
        }
}
//----------------------------------------------------------------
// copy constructor
// creates new bst from a preexisting bst
//----------------------------------------------------------------
template <class KeyType>
BST<KeyType>::BST (const BST<KeyType>& ptr)
{
        root = copy(ptr.root);
}
//----------------------------------------------------------------
// Empty
// tests if the bst is empty or not, returns a bool
// true (1) if the bst is empty
// false(0) if the bst is not empty
//----------------------------------------------------------------
template <class KeyType>
bool BST<KeyType>::Empty (void) const
{
        return ( root == NULL );
}

//----------------------------------------------------------------
// get
// returns a KeyType pointer to the first instance of the  parameter value
//----------------------------------------------------------------
template <class KeyType>
```

```
KeyType* BST<KeyType>::get (const KeyType& k)
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL){
                cout << "Error: empty tree" << endl;
                exit(1);
        }
        while (ptr != NULL and *(ptr->data) != k)
        {
                if (k < *(ptr->data))
                        ptr = ptr->left;
                else
                        ptr = ptr->right;
        }
        return ptr->data;
}
//-----------------------------------------------------------------
// copy
// private function
// copies the bst from a given node
//-----------------------------------------------------------------
template <class KeyType>
Node<KeyType> * BST<KeyType>::copy (Node<KeyType> *ptr) //changed return type
{
        if(ptr == NULL)
                return NULL;
        else
        {
                Node<KeyType> *qtr = new Node<KeyType>;
                qtr->data = ptr->data;
                qtr->left = copy(ptr->left);
                qtr->right = copy(ptr->right);
                return qtr;
        }
}
//-----------------------------------------------------------------
// insert
// inserts the value given as the parameter into the correct
// node in the bst.
// pre-condition: It is assumed that the bst is valid
// post-condition: The bst is valid after insertion
//-----------------------------------------------------------------
template <class KeyType>
void BST<KeyType>::insert (KeyType *k)
{
        Node<KeyType> *K = new Node<KeyType>;           //node to be inserted
        K->data = k;
        K->left = NULL;
        K->right = NULL;
        Node<KeyType> *qtr = NULL;
        Node<KeyType> *ptr = root;
        while (ptr != NULL)
        {
                qtr = ptr;
                if (*(K->data) < *(ptr->data))
                        ptr = ptr->left;
                else
                        ptr = ptr->right;
        }
        if (qtr == NULL)
        {
                root = K;
        }
        else if (*(K->data) < *(qtr->data))
        {
                qtr->left = K;
        }
        else
                qtr->right = K;
}
//-----------------------------------------------------------------
// remove
// removes and deletes the node of the first instance of the
// value passed as a parameter
// pre-condition: the bst is valid
// post-condition: the bst is still valid after removal
//-----------------------------------------------------------------
template <class KeyType>
void BST<KeyType>::remove (const KeyType& k)
{
        Node<KeyType> *K = find(root, k);
        Node<KeyType> *parent = findParent(root, k);
        if (K->left == NULL && K->right ==NULL)                  //case 1: no children
        {
                if (*(parent->data) < *(K->data))
                        parent->right = NULL;
                else
                        parent->left = NULL;
                delete K;
        }
        else if (K->left == NULL && K->right != NULL)   //case 2: one child (right)
        {
                if (*(K->right->data) < *(parent->data))
                {
                        parent->left = K->right;
                        delete K;
                }
                else
                {
                        parent->right = K->right;
                        delete K;
                }
        }
        else if (K->left != NULL && K->right == NULL)   //one child (left)
        {
                if (*(K->left->data) < *(parent->data))
                {
                                parent->left = K->left;
                                delete K;
                }
                else
                {
                        parent->right = K->left;
                        delete K;
                }
        }
        else                            //case 3: two children
        {
                Node<KeyType> *succ = find(root, *successor(k));
                KeyType temp = *(succ->data);
                remove(*(succ->data));
                *(K->data) = temp;
        }
}
//-----------------------------------------------------------------
// findParent
// private function
// finds and returns the node that is the parent of the first instance
// of the KeyType parameter value
//-----------------------------------------------------------------
template <class KeyType>
Node<KeyType> * BST<KeyType>::findParent (Node <KeyType> *ptr, KeyType item) const
{
        if (ptr->left == NULL && ptr->right == NULL)
           return NULL;
        else if (ptr->left && *((ptr->left)->data) == item)
                  return ptr;
        else if (ptr->right && *((ptr->right)->data) == item)
           return ptr;
        else
        {
                Node<KeyType> *qtr = NULL;
                if(ptr->left != NULL)
                        qtr = findParent(ptr->left, item);
                if(qtr == NULL && ptr->right != NULL)
                        qtr = findParent(ptr->right, item);
                return qtr;
        }
        return NULL;
}
//-----------------------------------------------------------------
// maximum
// returns a KeyType pointer to the maximum element in the bst
//-----------------------------------------------------------------
template<class KeyType>
KeyType* BST<KeyType>::maximum (void) const
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL)
        {
                cout << "Error: empty tree" << endl;
                //exit(1);
        }
        while (ptr->right != NULL)
                ptr = ptr->right;
        return ptr->data;
}
//-----------------------------------------------------------------
// minimum
// returns a KeyType pointer to the minimum element in the bst
//-----------------------------------------------------------------
template<class KeyType>
KeyType* BST<KeyType>::minimum (void) const
{
        Node<KeyType> *ptr = root;
        if (ptr == NULL)
        {
                cout << "Error: empty tree" << endl;
                //exit(1);
        }
        while (ptr->left != NULL)
    ptr = ptr->left;
  return ptr->data;
}
//-----------------------------------------------------------------
// successor
// returns a KeyType pointer to the successor of the first instance
// of the KeyType parameter value
//-----------------------------------------------------------------
```

```
template<class KeyType>
KeyType* BST<KeyType>::successor  (const KeyType& k) const
{
        Node<KeyType> *K = find(root, k);               //case 1: find smallest value in right
 subtree
        if (K->right != NULL)
        {
                K = K->right;
                while (K->left != NULL)
                {
                        K = K->left;
                }
                return K->data;
        }
        Node<KeyType> *parent = findParent(root, k);
        while (parent != NULL && K == parent->right)
        {
                K = parent;
                parent = findParent(root, *(parent->data));
        }
        return parent->data;
}
//-----------------------------------------------------------------
// predecessor
// returns a KeyType pointer to the predecessor of the first
// instance of the KeyType parameter value
//-----------------------------------------------------------------
template<class KeyType>
KeyType* BST<KeyType>::predecessor  (const KeyType& k) const
{
        Node<KeyType> *K = find(root, k);
        if (K->left != NULL)
        {
                K = K->left;
                while (K->right != NULL)
                {
                        K = K->right;
                }
                return K->data;
        }
        Node<KeyType> *parent = findParent(root, k);
        while (parent != NULL && K == parent->left)
        {
                K = parent;
                parent = findParent(root, *(parent->data));
        }
        return parent->data;
}
//-----------------------------------------------------------------
// inHelper
// private inOrder helper function.
// recursively creates a string of KeyType values by inOrder traversal
// base case: if r is NULL, returns the string parameter with no changes
//-----------------------------------------------------------------
template<class KeyType>
string BST<KeyType>::inHelper (Node<KeyType> *r, stringstream &s) const
{
        if (r != NULL)
        {
                inHelper(r->left, s);
                s << (*r->data);
                s << ' ';
                inHelper(r->right, s);
        }
        return s.str();
}
//------------------------------------------------------------------
// inOrder
// returns a string of KeyType values in inOrder traversal
//------------------------------------------------------------------
template<class KeyType>
string BST<KeyType>::inOrder (void) const
{
        stringstream s;
        if (root == NULL)
        {
                return "Empty tree";
        }
        string str = inHelper(root, s);
        str.pop_back();                 //to get rid of the trailing space created in helper f
unc.
        return str;
}
//------------------------------------------------------------------
// preHelper
// private preOrder helper function.
// recursively creates a string of KeyType values by preOrder traversal
// base case: if r is NULL, returns the string parameter with no changes
//------------------------------------------------------------------
template<class KeyType>
string BST<KeyType>::preHelper (Node<KeyType> * r, stringstream &s) const
{
        if (r != NULL)
        {
                s <<(*r->data);
                s << ' ';
                preHelper(r->left, s);
                preHelper(r->right, s);
        }
        return s.str();
}
//------------------------------------------------------------------
// preOrder
// returns a string of KeyType values in preOrder traversal
//------------------------------------------------------------------
template<class KeyType>
string BST<KeyType>::preOrder (void) const
{
        stringstream s;
        if (root == NULL)
        {
                return "Empty tree";
        }
        string str = preHelper(root, s);
        str.pop_back();                 //to get rid of the trailing space created in helper f
unc.
        return str;
}
//------------------------------------------------------------------
// postHelper
// private postOrder helper function.
// recursively creates a string of KeyType values by postOrder traversal
// base case: if r is NULL, returns the string parameter with no changes
//------------------------------------------------------------------
template<class KeyType>
string BST<KeyType>::postHelper (Node<KeyType> * r, stringstream &s) const
```

```
{
        if (r != NULL)
        {
                postHelper(r->left, s);
                postHelper(r->right, s);
                s << (*r->data);
                s << ' ';
        }
        return s.str();
}
//-----------------------------------------------------------------
// postOrder
// returns a string of KeyType values in postOrder traversal
//-----------------------------------------------------------------
template<class KeyType>
string BST<KeyType>::postOrder (void) const
{
        stringstream s;
        if (root == NULL)
        {
                return "Empty tree";
        }
        string str = postHelper(root, s);
        str.pop_back();                  //to get rid of the trailing space created in helper f
unc.
        return str;
}
//-----------------------------------------------------------------
// assignment operator (=)
//-----------------------------------------------------------------
template<class KeyType>
BST<KeyType>     BST<KeyType>::operator=   (const BST<KeyType> &ptr)
{
        clear(root);
        root = NULL;
        root = copy(ptr.root);
        return *this;
}
//-----------------------------------------------------------------
// clear
// private function that clears the subtree from a given node
//-----------------------------------------------------------------
template <class KeyType>
void BST<KeyType>::clear (Node<KeyType> *& ptr)
{
        if ( ptr != NULL )
        {
                clear(ptr->left);
                clear(ptr->right);
                ptr = NULL;
                delete ptr;
        }
}
//-----------------------------------------------------------------
// find
// private funciton that returns the node of the first instance
// of the KeyType parameter
//-----------------------------------------------------------------
template <class KeyType>
Node<KeyType> * BST<KeyType>::find (Node<KeyType> *ptr, KeyType item) const
{
        if (root == NULL){
                cout << "Error: empty tree" << endl;
                exit(1);
        }

        if(ptr == NULL)
                return NULL;

        else if(*(ptr->data) == item)
                return ptr;

        else if(*(ptr->data) != item)
        {
                Node<KeyType> *qtr = find(ptr->left, item);
                if(qtr == NULL)
                        return find(ptr->right, item);
                else
                        return qtr;
        }
        else
                return NULL;
}
```

```
//Hannah Kerr and Emma Steinman
//test file for bst


#include <iostream>
#include <assert.h>
#include "bst.h"
using namespace std;

//--------------------------------------------------------------------------
// defConst
// tests default constructor
//--------------------------------------------------------------------------
int defConst(void)
{
    BST<int> t;
    //cout << t << endl;
    assert(t.inOrder() == "Empty tree");
}
//--------------------------------------------------------------------------
// testInsert
// tests insert
//--------------------------------------------------------------------------
int testInsert (void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  assert(t.inOrder() == "1 2 5 9");
}
//--------------------------------------------------------------------------
// copyConst
// tests copy constructor
//--------------------------------------------------------------------------
int copyConst (void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);

  BST<int> t2(t);
  assert(t2.inOrder() == "1 2 5 9");
}
//--------------------------------------------------------------------------
// testEmpty
// tests if bst is empty
//--------------------------------------------------------------------------
int testEmpty(void)
{
  BST<int> t1;
  BST<int> t2;
  int a = 1;
  int b = 4;
  int c = 7;
  t2.insert(&a);
  t2.insert(&b);
  t2.insert(&c);

  assert(t1.Empty() == 1);
  assert(t2.Empty() == 0);
}
//---------------------------------------------------------------------------
// testGet
// tests get function
//---------------------------------------------------------------------------
int testGet(void)
{
  BST<int> t1;
  int a = 1;
  int b = 4;
  int c = 7;
  t1.insert(&a);
  t1.insert(&b);
  t1.insert(&c);
  int* g = t1.get(4);
  assert(*g == 4);
}
//---------------------------------------------------------------------------
// testRemove
// tests remove
//---------------------------------------------------------------------------
int testRemove(void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);

  t.remove(1);
  assert(t.inOrder() == "2 4 5 9 10 15 20");
  t.insert(&a);
  t.remove(5);
  assert(t.inOrder() == "1 2 4 9 10 15 20");
  t.insert(&z);
  t.remove(9);
  assert(t.inOrder() == "1 2 4 5 10 15 20");
}
//---------------------------------------------------------------------------
// testMax
// tests maximum
//---------------------------------------------------------------------------
```

```
void testMax(void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);
  assert(*(t.maximum()) == 20);
}

//--------------------------------------------------------------------------
// testMin
// tests minimum
//--------------------------------------------------------------------------
void testMin(void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);
  assert(*(t.minimum()) == 1);
}

//--------------------------------------------------------------------------
// testSuccessor
// tests successor method
//--------------------------------------------------------------------------
void testSuccessor (void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);
  assert(*(t.successor(9)) == 10);
  assert(*(t.successor(1)) == 2);
}

//--------------------------------------------------------------------------
// testPred
// tests predecessor function
//--------------------------------------------------------------------------
void testPred (void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);
  assert(*(t.predecessor(15)) == 10);
  assert(*(t.predecessor(9)) == 5);
}

//--------------------------------------------------------------------------
// testPre
// tests pre-order
//--------------------------------------------------------------------------
void testPre (void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);
  string pre = t.preOrder();
  assert(pre == "2 1 9 5 4 15 10 20");
}
//-------------------------------------------------------------------------
// testPost
// tests post-order
//-------------------------------------------------------------------------
void testPost (void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);
  string post = t.postOrder();
  assert(post == "1 4 5 10 20 15 9 2");
}
//-------------------------------------------------------------------------
// testEqOp
// tests assignment operator
//-------------------------------------------------------------------------
void testEqOp(void)
{
  BST<int> t;
  int x = 2;
  t.insert(&x);
  int y = 9;
  t.insert(&y);
  int z = 5;
  t.insert(&z);
  int a = 1;
  t.insert(&a);
  int b = 15;
  int c = 4;
  int d = 10;
  int e = 20;
  t.insert(&b);
  t.insert(&c);
  t.insert(&d);
  t.insert(&e);

  BST<int> t2;
  t2 = t;
  assert(t2.inOrder() == t.inOrder());
}
int main(void)
{
  defConst();
  testInsert();
  copyConst();
  testGet();
  testRemove();
  testMax();
  testMin();
  testSuccessor();
  testPred();
  testPre();
  testEqOp();
  return 0;
}
```

```
//Hannah Kerr and Emma Steinman
//dictionary.h
//This file implements a dictionary via inheritance from our bst class
//November 1, 2017

#ifndef DICTIONARY_H
#define DICTIONARY_H

#include "bst.h"

template <class KeyType>
class Dictionary : public BST<KeyType>
{

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


  using       BST<KeyType>::copy;               //uses copy, insert, remove, get, and Empty
  using       BST<KeyType>::insert;            //from BST class
  using       BST<KeyType>::remove;
  using       BST<KeyType>::get;
  using       BST<KeyType>::Empty;

};
#endif
```

```
//Hannah Kerr and Emma Steinman
// movies.cpp
// A program that builds a Movie class and functions to read in
// a text file and create a corresponding dictionary with title as key
// and cast list as value. Finally a function that returns the cast list given
// movie title
// November 1, 2017


#include <iostream>
#include <fstream>
#include <string>
#include "dictionary.h"

using namespace std;

class Movie
{
public:
  string title;
  string cast;

  bool operator< ( Movie& a) const
  {
    return this->title <= a.title;
  };
  bool operator> (const Movie& a)            //overloading comparison operators
  {                                          //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
  {                                                             //cout operator
      os << mov.cast;
      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;
}
//---------------------------------------------------------------------------
//getCast
//takes the key representing movie title for which you want cast
//as a parameter and returns corresponding cast lists
//---------------------------------------------------------------------------
Movie getCast (string movieTitle)
{
  Dictionary<Movie> movies = movieDict("movies_mpaa.txt");
  Movie *search = new Movie;
  search->title = movieTitle;
  Movie castList = *(movies.get(*search));
  return castList;
}
//---------------------------------------------------------------------------
//see it works!
//---------------------------------------------------------------------------
int main(void)
{
  cout << "When Evil calls: " << endl;
  cout << getCast("\"When Evil Calls\" (2006)");
  cout << endl;
  cout << "27 Dresses (2008): " << endl;
  cout << getCast("27 Dresses (2008)");
  cout << endl;
  cout << "300 (2006): " << endl;
  cout << getCast("300 (2006)");
  cout << endl;
  cout << "A Knight’s Tale (2001): " << endl;
  cout << getCast("A Knight’s Tale (2001)");

}
```