Erik Sjoberg  
[esjoberg@cmu.edu](mailto:esjoberg@cmu.edu)

# Assignment 5: Six Degrees of Separation

## ADT Design

**ADT Choices**

For this assignment, the functionality was implemented with three key data structures: A linked list (LinkedList.h), a sorted mapping from strings to linked lists representing the graph (ActorMap in GraphActors.h), and a queue (pathQueue in GraphActors.cpp::bfs) containing discrete paths through the network represented by vectors of actor-movie pairs. A set of strings (actor names) was also used to keep track of visited nodes in the network, and avoid duplicate / looping paths during breadth-first search.

The actor-movie pairs which represent the “node” elements in all of these data structures (called “relationships” and defined in LinkedList.h) contain an actor and associated movie title. The actor and movie title represent the vertexes and edges of the graph structure.

**ADT Design**

The sorted mapping is implemented natively in the C++ standard library using red-black trees, although from the outside it can be treated similarly to a hash table. This was chosen as the key representation of the graph because it is an ordered mapping of items which allows it to be iterated over in order while containing references to arbitrarily complex objects that can be looked up by their key.

The linked list was implemented in the standard manner of a singly-linked list, with each node containing the key data (actor name and movie title), as well as a pointer to the next node. A pointer to the end of the list is also maintained. Linked lists were a sensible choice as the vertex-to-vertex mappings, because it allows any vertex to contain a set of nodes of any length. The order(n) worst-case element access of a linked-list is not as much of an issue in this application because we expect each actor to only have a limited number of movies they have appeared in, due to the real-world limitations on the number of movies one actor could appear in during their lifetime.

The queue of vectors of actor-movie relationships was chosen as the primary data structure for the breadth-first search because each vector compactly represents the complete path through the tree traveled to reach any particular node. When paired with the set containing already-visited nodes, this made the bookkeeping involved with returning the complete sequence of actor-movie relationships straightforward. The cost is paid in terms of memory usage, but given the large amount of memory available on personal computers and the limited depth of the search, memory efficiency was not a major limiting factor for the design.

**Information Hiding**

The linked list ADT specifically hides its internal structure, only providing insertEnd(), getFirstRelationship(), and getAllConnection() functions which provide the functionality required for this assignment. Internal details such as the pointers to the head and the tail are explicitly hidden from the user via the private member variable feature of C++.

Similarly, the ActorMap data structure itself is completely hidden inside of the GraphActors object, and only accessable via methods such as addActorConnection(), getLength(), and bfs() for performing the actual search upon the graph data structure.

**How the ADT is used**

The user is given access to initialization routines for each data structure, as well as the following:

* Linked List:
  + Insert at end
  + Get head of list
  + Get all items in list as vector
  + Display list
* ActorMap graph:
  + Add actor connections to graph
  + Get number of actors inserted
  + Test for key existence
  + Breadth-first search between two actors

To use the ActorMap graph, the user first inserts actor / movie combinations into the data structure, which automatically uses a linked list internally to append these actor/movie combinations to the graph structure.

Once the graph has been built, the user can extract a vector of actor/movie relationships from the ActorMap using ActorMap.bfs(), which searches up to 6 levels of the graph in a breadth-first manner.

## How to Use the Program

The program can be run at the terminal via the following executable:

sixdegrees.exe

Pressing Enter once when prompted will cause the program to proceed. When asked to input the name of an actor, please input a the actors full name using proper capitalization. For example:

Jim Carrey

To compile the program, go to the directory containing the source code files and use the g++ compiler with the following command:

g++ --std=c++11 main.cpp GraphActors.cpp GraphActors.h LinkedList.cpp LinkedList.h -o sixdegrees

## Program Analysis

**Correctness**

This program builds upon the std::map(), std::set(), and std::queue() ADTs which can be trusted to maintain their respective correctness as a result of the tremendous amount of code which relies upon them. Furthermore, the correctness of our LinkedList implementations has been demonstrated in previous assignments.

This leaves primarily the process of the graph search to be proved correct.

For the graph search, the first point to make is that the top-level while() loop behaves in the following manner:

1. While pathQueue not empty
   1. Pop item from queue
   2. Get all connections from item
   3. For connection in connections
      1. If connection has not been visited:
         1. Mark connection as visited
         2. Put connection into queue

From this behavior, proper termination can be demonstrated because for every loop, the queue length decreases by 1, while the queue can only ever increase by a total maximum of n = number of actors times. This means the queue will be empty after at most n iterations since each actor can only be added once.

The post-condition of this queue loop above is that either a) all possible paths up to 6 degrees have been explored, or b) the shortest path to the goal actor has been found.

**Complexity: Performance**O(n6) (unrealistic worst case)

The process for performing breadth-first search in the actor graph is as follows:

1. For link from actor:
   1. If link not already visited:
   2. Find links from linked actor
      1. If link not already visited:
      2. Find links from linked actor
         1. … six times total

If one sixth of the actors are visited each loop, this comes out to O((n/6)\*(n/6)\*(n/6)\*(n/6)\*(n/6)\*(n/6)) = O((n/6)^6) = O(n^6) comparison operations to see if a given actor is already visited. This however assumes the worst cast where every actor has been in a movie with every other actor, which is clearly overly pessimistic.

In addition, a single lookup in the ActorMap is O(log n) due to the use of a red-black tree implementation, but because lookups only occur at most once per actor this term is at most O(n \* log n). Pulling all actor relationships (other actors who’ve shared a movie) from the ActorMap also takes O(n), so on each iteration the total contribution from ActorMap manipulations could add up to O(n^2 \* logn).

This puts final, worst case performance at O(n^2 \* logn) + O(n^6) = O(n^6).

**Complexity: Memory**   
O(n2)

In the worst case, the graph structure ActorMap would need to contain not only an entry for every actor, but also another linked-list entry in each actor to every other actor. This results in O(n2) memory requirement to store the graph structure itself in the worst case where every actor has been in a movie with every other actor.

For the graph search itself, there is an additional memory requirement for an array of links for every actor which is connected to the starting actor, adding another constant O(n) term.

This results in O(n) + O(n2) = O(n2) memory usage.

## Build / Run Instructions

1. Change path to the folder containing the source code
   1. cd /path/to/sourcecode/folder/
2. Compile with gcc (use MinGW if on Windows)
   1. g++ --std=c++11 main.cpp GraphActors.cpp GraphActors.h LinkedList.cpp LinkedList.h -o sixdegrees
3. Run sixdegrees.exe (from console, or double click)
4. Press Enter as prompted
5. Input full name of first actor
6. Input full name of second actor
7. View degrees of separation and list of movies and actors making the connection