Assignment 1

N9723099

Joshua want

cab301

# Top 10 Algorithm

*mainpage.cs*

public class glob //class containing global variables

{

public static string fname; //glob used for storing logged in users full name

public static Movie[] TopArray = new Movie[40]; //Array used to calculate top movies, assumes 40 or less movies in BST

static int loc = 0; //variable used to properly manipulate variable order in top array

public static void Add(Movie mov) //function for adding movie to correct position

{

if (loc < TopArray.Length) //checks its not trying to index out of array length

{

TopArray[loc] = mov; //set current index to inpout movie

loc++; //iterate

}

}

}

*MovieCollection.cs*

public Movie[] ArrayMovies() //this one is the only one that works slightly differently

{

if (root != null) {

root.ArrayMovie(); //calls arraymovie on root which recursivley modifies the TopArray global

return glob.TopArray; //returns top array to func after its been modified by arraymovie

}

return null;

}

*TreeNode.cs*

public void ArrayMovie() //works the same as printmovies but instead of printing adds the movie to a global array using my own .Add function

{

if (leftNode != null)

{

leftNode.ArrayMovie();

}

glob.Add(movie); //add movie to global array using defined .Add

if (rightNode != null)

{

rightNode.ArrayMovie();

}

}

*MemberMenus.cs*

public static void Top10()

{

for (int i = 0; i < glob.TopArray.Length; i++) //this loop isn't necessary, it just clears all entries in the global array when top10 is called

{ //Allows for the top 10 function to be run back to back to compare effects of borrowing a movie on the top 10

glob.TopArray[i] = null;

}

Movie[] tops = Store.movies.ArrayMovies(); //calls ArrayMovies, which returns all nodes from BST to an array of movies

for (int i = 0; i < tops.Length -1; i++) //bubble sort on array

{

for (int j = i+1; j < tops.Length; j++)

{

if (tops[i] != null && tops[j] != null) //doesn't sort null values, Null represents the extra spaces in the array (has enough length to sort 40 movies).

{

if (tops[i].timesBorrowed < tops[j].timesBorrowed) //compare

{

Movie tempMov = tops[i]; //swap the values and then iterate

tops[i] = tops[j];

tops[j] = tempMov;

}

}

}

}

int k = 0; //k represents the number of movies written to console. Used to make sure only 10 are printed (no point in iterating through rest of array if 10 already printed)

for (int i = 0; i < tops.Length && k < 10; i++) //iterates through tops which has been sorted with bubblesort. terminates if K will exceed array length or k is greater than 9 (ie 10 things have been printed).

{

if (tops[i] != null) //checks that movie isn't null, this would mean its one of the unsorted blank spots that would allow for up to 40 movies to be sorted

{

Console.WriteLine((k+1)+".\t"+tops[i].title + " has been borrowed " + tops[i].timesBorrowed); //write movie to console

k++; //something was printed so iterate k

}

}

Console.WriteLine("");

MainPage.MemberMenu();

}

# Time Complexity Analysis

**ALGORITHM** Add(Movie Mov)

//Adds movie object to global array TopArray

//Input: Movie object, uses global int loc, global array TopArray[0..n-1]

**If** loc < n **then**

TopArray[loc] <- mov

loc <- loc + 1

**ALGORITHM** ArrayMovie

//traverses BST and adds nodes to global array ArrayTop[0..n-1]

**If** leftnode != null **then**

leftNode ArrayMovies

Add(node.movie)

**If** rightnode != null **then**

rightNode ArrayMovies

**ALGORITHM** ArrayMovies  
 //Starts BST traversal at tree root  
 //returns global array TopArray[0..n-1] or null  
 **If** root != null **then**

root ArrayMovies //traverses tree and adds all nodes to TopArray

return TopArray[]

return null

**ALGORITHM** Top10(A[0..n-1])

//Sorts an array of length n into descending order

//Output: Writes sorted array to console

**for** i <- 0 **to** n – 1 **do**

//set all objects in array to null so array is clean

A[i] <- null

tops <- ArrayMovies

**for** i <- 0 **to** n – 2 **do**

**for** j <- i+1 **to** n – 1 **do**

**If** tops[i] != null **then**

**If** tops[j] != null **then**

temp <- tops[i]

tops[i] <- tops[j]

tops[j] <- temp

k <- 0

**for** i <- 0 **to** n – 1 **do**

**If** k < 10 **then**

**If** tops[i] != null **then**  
 write tops[i]

k <- k + 1

As demonstrated in appendix 1, the highest order of complexity for the Top10 algorithm is the sorting algorithm used to sort the movie array into the correct order. While completing further analysis, this allows us to discard the complexity of all other algorithms used in this function as their impact on the complexity is insignificant when compared to the sorting algorithm.

Due to the design of the sorting algorithm the best case, average and worst case complexity are all similar. As the sorting algorithm uses two nested for loops with no escape conditions, regardless of even if a pre-sorted array is processed by the function it will still have to complete all iterations through the nested loops, and only constant time operations will differ between the best and worst complexity.

Below is a summation formula which can be used to describe the algorithms best performance for an array of length n, and determine the corresponding efficiency class.

Below is a summation formula which can be used to describe the algorithms best performance for an array of length n, and determine the corresponding efficiency class.

Due to the design of the algorithm, the only times when the null checks will not be satisfied is if the array has been initiated as much longer than the number of movies currently in the BST. As such the average performance will closely resemble for an array of length n.

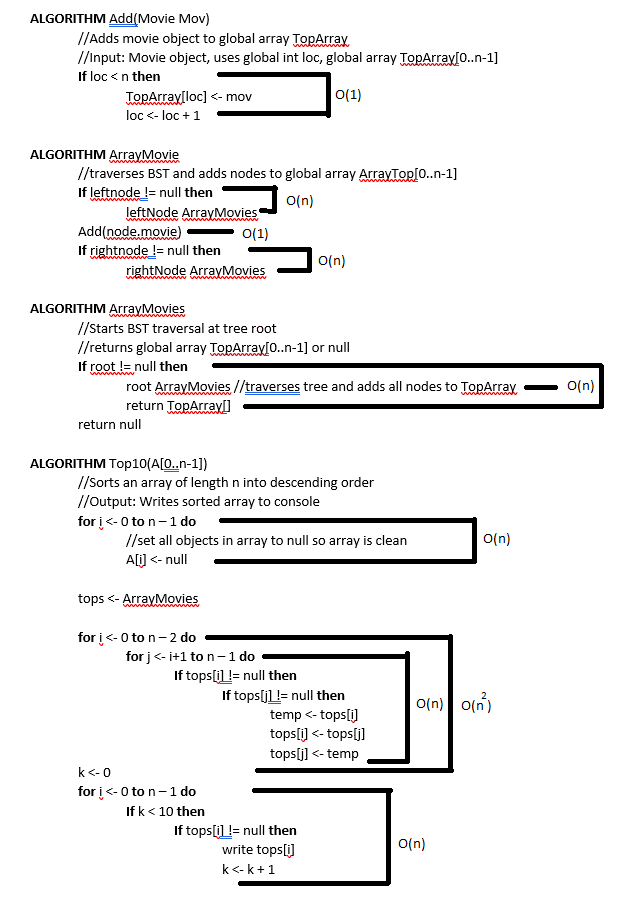
Furthermore, included in appendix 2 is a graph demonstrating the runtime of the algorithm from between n = 100 and n = 200000. It is clear from this graph that computation time does not linearly increase with array size.

# 3. Testing

|  |  |  |  |
| --- | --- | --- | --- |
| **Functionality** | **Expected Result** | **Actual Result** | **Proof** |
| Begin Program | Member or Staff login selection | As expected |  |
| Staff Login correct details | Prompts for username/pass. Allows into staff menu | As expected |  |
| Staff Login incorrect details | No entry to staff menu. Says login details wrong | As expected |  |
| Staff Add movie | Movie added to movie collection in correct alphabetical order | As expected |  |
| Staff remove movie | Movie is removed from movie collection | Remove function not working | Not implemented |
| Staff register member | New member is registered | As expected |  |
| Staff find member phone number (correct) | Real member name input, phone number output | As expected |  |
| Staff find member phone number (correct) | Output saying not a valid member | As expected |  |
| Member login correct details | Proceed to member menu | As expected |  |
| Member login incorrect details | Login fails, output of incorrect login given | As expected |  |
| Member Display movies | All movies in collection displayed in alphabetical order | As expected |  |
| Member borrowed DVDs correct | Input prompting for movie, output of success. In movie collection times rented increases and copies decreases | As expected |  |
| Member borrowed DVDs correct | Prompted for movie. Output saying its invalid | As expected |  |
| Member return movie correct | Prompt asking for movie, output confirming return. Copies goes up. | As expected |  |
| Member movie return incorrect | Prompt asking for movie, output saying movie unable to rent | As expected |  |
| Member list rented | Display title of all rented movies | As expected |  |
| Member Top 10 | Prints top 10 rented movies in descending order | As expected |  |
| Members Top 10 changes with borrowing | If movie is borrowed enough the top 20 order will change accordingly.  This test will borrow star wars twice | As expected |  |
| Members Top 10 when the store has less than 10 movies | Movies are displayed by times rented descending. As there are less than 10 movies less than 10 movies should be sorted | As Expected |  |

# Appendix 1

Visual demonstration of efficiency classes for functions to highlight most computationally complex algorithms



# Appendix 2

Graph demonstrating sorting algorithm runtime depending on array length. Tests were ran on an AMD 3700X at ~4.1GHz