

Project #02: Movie Search with Binary Search Trees

Complete By: Saturday, September 28th @ 11:59pm

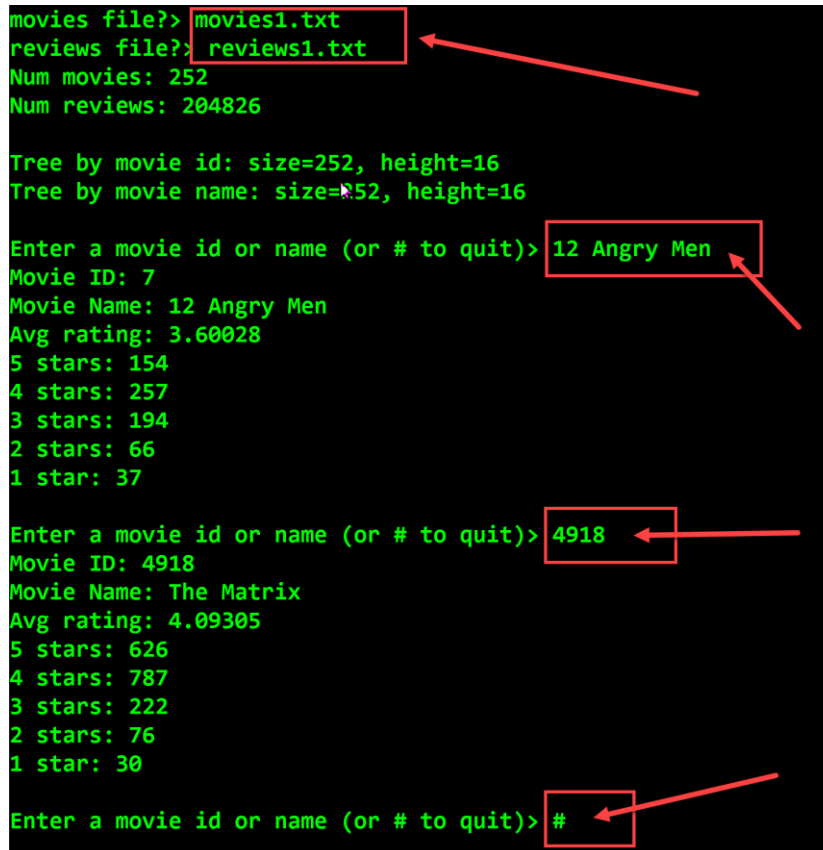
Assignment: BST class + program for searching movie data

Policy: Individual work only, late work **is** accepted (see “Policy” section for more details)

Submission: Two separate Gradescope submissions

Overall Assignment

The assignment is to input movie and review data, allowing the user to search this data quickly for movies by ID or name. Here’s a screenshot of the final program (one possible execution run):



```

movies file?> movies1.txt
reviews file?> reviews1.txt
Num movies: 252
Num reviews: 204826

Tree by movie id: size=252, height=16
Tree by movie name: size=252, height=16

Enter a movie id or name (or # to quit)> 12 Angry Men
Movie ID: 7
Movie Name: 12 Angry Men
Avg rating: 3.60028
5 stars: 154
4 stars: 257
3 stars: 194
2 stars: 66
1 star: 37

Enter a movie id or name (or # to quit)> 4918
Movie ID: 4918
Movie Name: The Matrix
Avg rating: 4.09305
5 stars: 626
4 stars: 787
3 stars: 222
2 stars: 76
1 star: 30

Enter a movie id or name (or # to quit)> #
  
```

The only data structure you may use are binary search trees, in particular the BST class you have developed in the HW and labs.

Programming Environment

You are free to program in any environment you want; final submissions will be collected using Gradescope. We are releasing an initial “main.cpp” file, and sample input files, on Codio. Login to Codio and open the project “**cs251-project02-bst-movie-search**”. You can export the provided files using the Project menu, Export as Zip. Or you can work on Codio using the provided **makefile**: open a terminal window via the Tools menu, and “make build” to compile and “make run” to run.

If you have not yet created a Codio account, register using your UIC email address and join the class (“CS 251 Fall 2019”) via the following URL: <https://codio.com/p/join-class?token=brigade-america>

Part 1 of 5: `binarysearchtree<TKey, TValue>`

From the most recent HW and lab exercises, you should have a working **`binarysearchtree<TKey>`** class in a header file “bst.h”. The first part of this assignment is to extend your **`binarysearchtree`** class so that it works in terms of (key, value) pairs. At this point we have only stored keys in the tree. In reality, trees store a **key** for ordering / searching, and a separate **value** for any data associated with the key. For example, your UIC Netid is a key, while your name and address are values associated with your Netid. A value can be a simple integer, or more typically a structure of values (e.g. Name, Address, Phone, Email, etc.).

In the context of this assignment, you’ll need to store movie information in the tree, e.g. the publication year of the movie, along with review data. So you’ll want to define a structure to hold the data:

```
struct MovieData
{
    int   PubYear;
    int   Num5Stars;
    int   Num4Stars;
    int   Num3Stars;
    int   Num2Stars;
    int   Num1Stars;
};
```

Now, to store movie data using the movie id as the key, you declare a binary search tree as follows:

```
binarysearchtree<int,MovieData>   bstMoviesByID;
```

In other words, each entry in the tree is now a (key, value) pair where the key is an integer movie id, and the value is a `MovieData` struct. The tree is still ordered and searched using the key, the only differences are that **insert** will insert both (key, value), and **search** will return a pointer to the value so it can be read / written.

The good news is these changes are fairly straightforward, since only a few modifications are needed. To begin, the template clause at the top of “bst.h” must change to include both key and value:

```
template<typename TKey, typename TValue>
class binarysearchtree
```

Next you'll need to update the **struct NODE** to store both the key and the value:

```
struct NODE
{
    TKey    Key;
    TValue  Value;
    NODE*   Left;
    NODE*   Right;
};
```

Keep in mind the tree is still ordered and searched by the key, so nothing with regards to how the tree is built or searched is changed. For the **insert** function, the only difference is that when the node is inserted, the value must also be stored in the new node. This means the value must be passed as a parameter:

```
void insert(TKey key, TValue value)
{
    .
    . // same as before, though be sure to store value along with key
    .
}
```

The final change involves the **search** function, which currently returns true / false to denote whether the key was found or not found. Now, if the key is found, we want to return a **pointer** (yes, a C-style pointer) to the value so it can be read or written. If the key is not found, search will return nullptr. Here's the declaration:

```
TValue* search(TKey key)
{
    .
    . // return pointer to value if found, nullptr if not
    . // NOTE: to return pointer: return &(cur->Value)
    .
}
```

Make these changes, and then test using different (key, value) combinations. Example:

```
binarysearchtree<int,int>    bst1;
binarysearchtree<int,string> bst2;
binarysearchtree<string,int> bst3;

bst1.insert(123, 456);
.
.
.

int* value = bst1.search(123);
cout << *value << endl; // should output 456
*value = 789;           // change value to 789
```

I would recommend modifying your **inorder** function to output both the key and the value, and use inorder to help confirm your functions are working correctly.

Part 2 of 5: `binarysearchtree<TKey, TValue>` copy constructor

Since we are going to use the `binarysearchtree` class in a program, we need a copy constructor for parameter passing to work properly. Add a copy constructor in the public section of your “bst.h” file:

```
//  
// copy constructor:  
//  
binarysearchtree(binarysearchtree& other)  
{  
    .  
    .  
    .  
}
```

Like we did in the previous project, the goal of the copy constructor is to make a complete and independent copy of the “other” tree. Recall that you can make your life easier if you take advantage of the functions already available in the class [Hint: insert]. To make a copy, you have to traverse the entire “other” tree, and visit every node. Your inorder function does this as well, so your copy constructor should mimic how inorder works --- including the public-private approach. However, think very carefully about how the recursive traversal should be done: inorder, preorder, or postorder? Humm.....

How to test your work? Make a copy of the tree, change the copy, and make sure the original is unchanged. Here’s some code to get you started:

```
binarysearchtree<int,int>  bst1;  
bst.insert(123, 456);  
.  
.  // insert more (key, value) pairs  
.  
  
binarysearchtree<int, int> bst2 = bst1;  // copy construct:  
  
cout << bst1.size() << " vs. " << bst2.size() << endl;  
cout << bst1.height() << " vs. " << bst2.height() << endl;  
  
int* value1 = bst1.search(123);  
int* value2 = bst2.search(123);  
  
cout << *value1 << " vs. " << *value2 << endl;    // both 456  
  
*value2 = 789;  // this should only change bst2:  
  
cout << *value1 << " vs. " << *value2 << endl;    // 456 vs. 789
```

Part 3 of 5: `binarysearchtree<TKey, TValue>` submission

Submit your `binarysearchtree` class to Gradescope for grading. Your work will receive a tentative grade for correctness, and then will undergo manual review for commenting, readability, and approach. This part of the assignment is worth 30 points, and commenting/readability/approach count for 10% of that 30 points.

Part 4 of 5: movie search program

Once you have a working BST class, use this class to input movie data and efficiently search this data in $O(\lg N)$ time. Since the input data is in random order, tree balancing is not required. Here's another screenshot of the program in action ----->

The movies input file consists of $N > 0$ lines, where each line consists of 3 values: integer movie id, integer publication year, and string movie name. Here's the start of the provided "movies1.txt" file:

```
119 1995 Heat
87 1952 Singin' in the Rain
6 2008 The Dark Knight
.
.
.
104 1988 Die Hard
```

The data is in random order, both in terms of the movie id, and the movie name. This makes it suitable for building binary search trees using the id as a key, and the name as a key. In this assignment, you'll want to create two different trees so you can lookup by id or name in $O(\lg N)$ time. Make no assumptions about the input data, as we'll test your work using different movie files.

The reviews input file consists of $N > 0$ lines, where each line consists of 3 values: an integer review id, an integer movie id, and the integer rating in the range 1..5. The movie id refers to the movie for which this rating is associated, and the rating is the # of stars given in this review. Here's the start of the provided "reviews1.txt" file:

```
140035 188 5
133994 78 4
```

```
movies file?> movies1.txt
reviews file?> reviews1.txt
Num movies: 252
Num reviews: 204826

Tree by movie id: size=252, height=16
Tree by movie name: size=252, height=16

Enter a movie id or name (or # to quit)> finding nemo
not found...

Enter a movie id or name (or # to quit)> Finding Nemo
Movie ID: 1162
Movie Name: Finding Nemo
Avg rating: 4.09641
5 stars: 611
4 stars: 783
3 stars: 202
2 stars: 70
1 star: 35

Enter a movie id or name (or # to quit)> 99712
not found...

Enter a movie id or name (or # to quit)> 123
Movie ID: 123
Movie Name: The General
Avg rating: 3.62968
5 stars: 146
4 stars: 314
3 stars: 222
2 stars: 68
1 star: 25

Enter a movie id or name (or # to quit)>
```

```
123308 199 2
.
.
.
289471 78 4
```

Once again, make no assumptions about the input data, as we'll test your work using different review files.

One of the design decisions you'll need to make is how to associate each review with the correct movie. That's where the movie id comes in. As you input the review data, lookup the movie id in your binary search tree, and store the rating in the value portion of the tree. In other words, use the review's movie id as the search key, and store the review's rating in the value associated with the key. When you are done processing the reviews, you should have a binary search tree that contains all the review data, organized by movie id or name (or both). The goal is to be able to find a movie's information in $O(\lg N)$ time using either the movie's id, or the movie's name. This implies your key may be a simple type like integer or string, but your value will most likely be a struct.

When searching for a movie, note that the user can enter a numeric movie id, or a string name. Use the **getline** function to obtain the input, and then determine if the input is numeric or not:

```
string input;

cout << "Enter a movie id or name (or # to quit)> ";
getline(cin, input); // read entire input line
```

It is not sufficient to simply check the first character of the input to see if it's numeric --- some movie names start with numeric values such as "12 Angry Men". You'll need to write a little function to loop over the string and check all the characters; strings are objects, use their `length()` function to obtain the # of characters.

The program should interact with the user until the sentinel "#" is input. Your program output must match the output shown in the screenshots *exactly*, otherwise the autograding system will report your program as incorrect. It's okay for your program to be case-sensitive, i.e. "Finding Nemo" != "finding nemo".

Additional program requirements:

1. *You must use your own `binarysearchtree` class, a solution will not be provided and you cannot use that of another individual. The entire program must be your own work.*
2. *No other data structures --- no arrays, no vectors, no stacks, no queues, nothing. The *only* data structure you may use is your own `binarysearchtree<TKey, TValue>`.*
3. *Create multiple trees to ensure that all search and insert operations are $O(\lg N)$. Tree balancing is not required as the input data is random.*
4. *You can only open and read each input file once; you cannot solve the program by reading the data from the input files over and over again.*

Part 5 of 5: program submission

Submit your final program files --- “main.cpp” and “bst.h” --- to Gradescope for grading. Your work will receive a tentative grade for correctness, and then will undergo manual review for commenting, readability, approach, and adherence to requirements. This part of the assignment is worth 70 points. However, you must score 30/30 on part 3 to earn credit here for part 5; you need a working `binarysearchtree` class.

Commenting/readability/approach count for 10% of the 70 points. Adherence to requirements may count anywhere from 10% to 100%.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your “iClicker teammates”), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is available here:

<https://dos.uic.edu/conductforstudents.shtml> .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else’s iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at <https://dos.uic.edu/conductforstudents.shtml> .