Notes on Assignment 7

CSCI 2270: Data Structures

Section: 305
TA: Varad Deshmukh

Section: 202
TA: Sanskar Katiyar

Brief Overview, Motivation

This assignment is an extension to the previous one — you'll be building an IMDB database of movies using a BST — however, the header files and the structures are defined differently. In the last assignment, each node in the BST represented a node, i.e., each node in the BST was indexed by the movie title. Here, your BST will be structured a bit differently — each node in the BST will be indexed by the first alphabet of the title (instead of the title itself).

We have been shirking off the question of inserting duplicate keys in the BST so far. This was not an issue in Assignment 6, since each movie had a unique title. However, if the BST node is indexed by the first alphabet of the title, entries with the same key is an important problem. *For example*, the movies "*Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb*" and "Django Unchained" would be individual nodes in Assignment 6, but would fall under the same node with key "*D*" in Assignment 7.

To take care of this, you'll build a hybrid data structure that maintains a sorted linked list of individual movies at each BST node.[1] Thus, all the movies starting with "*D*" will be inserted at the BST node "D", and maintained as individual movie nodes in a linked list at that node. While this solves the issue of duplicate keys, search, insertion and deletion of a movie in this hybrid data structure is a bit more complex than a simple BST.

Searching a Movie in this Hybrid Data Structure

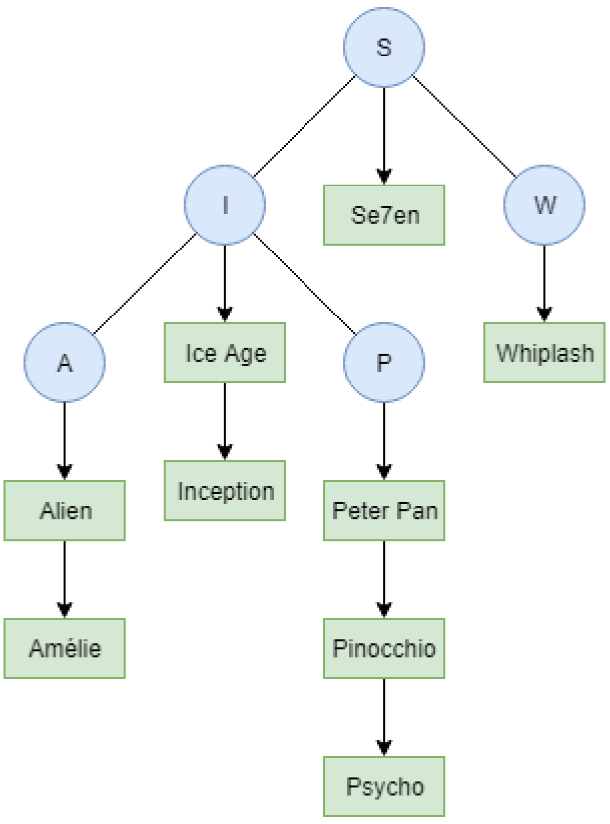
Below is the example tree mentioned in the assignment writeup. Let's perform a search operation for the movie — **Pinnochio** and observe the required steps.

Since the BST is ordered by the first character of the movie titles, for our movie we will have to search for the node which consists of movies starting with

[1] Note that it is possible to have repeated key values in a BST. A common practice is to adopt a convention, such as, which subtree will consist of the repeated values, or maintaining a count at each node of the number of duplicate items.

character ***P***.

Once we have found the BST node which consists of all movie titles starting with ***P***, we can perform a search on the attached linked list for the required movie title — Pinnochio.



Let's observe the steps we performed to search for some **`title`**:

a. Using *key* = first alphabet from the title, we search for the appropriate node in the BST. If the BST key is not found, the search function returns **`NULL`**, else continues to the next step.

b. From the found BST node, access the accompanying linked list, and search through the linked list by comparing the title with the linked list node title. If you reach the end of the linked list, the movie is not found and the search function returns nothing. *Remember, since the linked list is sorted you don't necessarily need to traverse the entire list for each operation.*

c. If the movie is found, the node is returned; else **`NULL`** is returned.

Implementation Notes

Through these helper notes, we will give you useful hints to implement these functions. Just like Assignment 6, you will be implementing functions in the file **`MovieTree.cpp`** (**`driver.cpp`** is available to you).

You should be comfortable with both BST and Linked List operations. If you're rusty, please refer to the recitation, lecture slides and code examples.[2,3]

`MovieTree()`
`~MovieTree()`
`void MovieTree::printMovieInventory();`
`void MovieTree::addMovie(int ranking, std::string title, int year, float rating)`
`void MovieTree::deleteMovie(std::string title)`
`void MovieTree::leftRotation(TreeNode * curr)`

Some tips to approach the problem

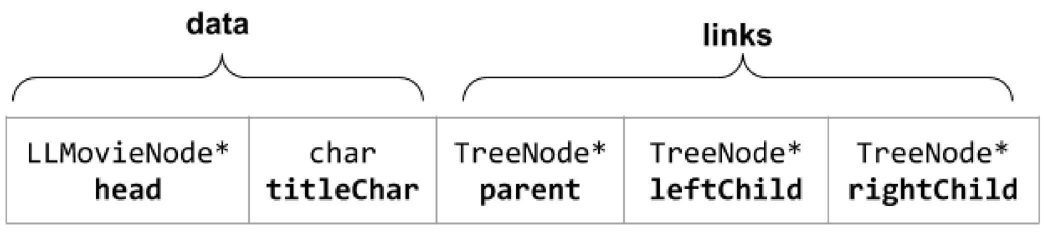
[1] You're constructing a BST and LL hybrid data structure. Therefore, you will be using two different node structures: **`struct TreeNode`** (the BST node) and **`struct LLMovieNode`** (the linked list node). The BST node structure in this assignment is different than the BST node structure in assignment 6:

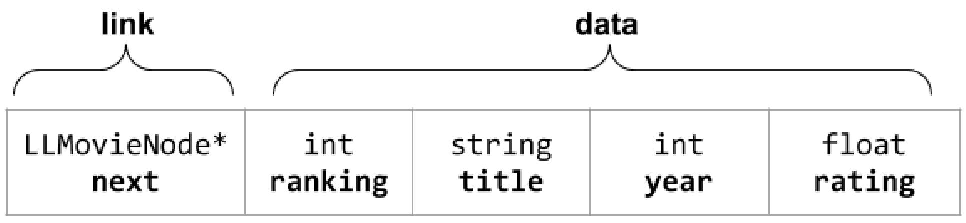
a. The BST node uses the first letter of the movie title as the key.

b. Apart from the left and right child, each BST node contains a parent pointer (for the root, parent = NULL). ***Remember to appropriately set the parent pointer during insertion and deletion operations.***

c. The BST contains the head pointer of the associated linked list: **`LLMovieNode* head == NULL; // true for empty Linked List`**

[2] Binary Search Tree Operations [Code]
[3] Linked List Operations [Code]





The linked list node is a node which should contain all the details of the movie: the title, the ranking, the rating and the year released. As any linked node, it should contain a pointer to the next node.

The layout of both the nodes is available to you as a structure – **struct MovieNode** – in the file **MovieTree.hpp**. *Refer image above. Carefully read* `MovieTree.hpp` *file once before starting.*

[2] Searching, adding and deleting nodes in the BST will involve recursive functions similar to what you implemented in the past assignment and the recitation. The difference here is that – instead of simply modifying the tree node, you will need to perform additional operations on the associated linked list as well.

For example, adding a movie in this hybrid structure may involve both inserting a new node in the BST (with the first alphabet in the title as key), as well as inserting a new node in the linked list.

A function has been provided to you that searches for the BST node with the queried first alphabet of the title: **`TreeNode* MovieTree::searchChar(char key)`** in **`MovieTree.cpp`**. *Carefully go through MovieTree.cpp as well before starting.*

[3] We *strongly* recommend modularizing your approach in terms of separate Linked List and BST operations. Refer your code from the past exercises, recitations or lectures. However, remember to modify it to the suitable node structures and class.

~MovieTree

Refer the code from the previous assignment's BST destructor (and helper, if applicable) and adapt it to the current node structures.

Note that now you need to **delete the associated linked list first** before you delete the BST node. One way to delete the entire linked list is to keep deleting the head node until it is empty (or `head == NULL`).

printMovieInventory

```
void printMovieInventory()
```

This function requires a traversal implementation on a BST and the associated Linked List.

[1] Assume that we *just* want to print the `titleChar` of each BST node (not the movies), in alphabetical order. ***Which tree traversal will you use for printing node items in an alphabetical order in a BST?*** Start off by implementing this traversal on just the BST.

[2] Now, instead of just printing the `titleChar` in the BST node, we also need to print the contents of the entire linked list associated with the BST node. ***Review: Traversal in a linked list***.

addMovie

```
void addMovie(int ranking, string title, int year, float rating)
```

Adding a new movie in this hybrid data structure requires you to follow a series of sequential steps:

[1] Using the first alphabet from the title, search for the appropriate node in the BST. If the node is missing, create a new BST node in the tree. ***You may want to implement a recursive helper for this. Utilize the searchChar function to search the BST.***
For example, if we are inserting the movie "Coherence", but the BST doesn't contain a node with key "C", create the "C" node first.

Another thing to note: **How can you update the node's parent pointer?** Once each recursive call is complete, you will update the child's parent pointers to the current node? *This is something we will also cover in the recitation, this week.*

[2] If the BST node exists, access the accompanying linked list. This linked list is alphabetically sorted, so march through the list until you reach the right spot to insert into. ***Take inspiration from Assignment 1 (Insert into Sorted Array problem).***

[3] Create a new linked list node with all the movie details, and insert into the linked list at the correct position. Use the algorithm for insertion into the linked list from the past assignments.

[4] Edge cases: (i) New node is inserted in an empty list, (ii) New node is inserted before head in an existing linked list.

NOTE: If you are trying to implement a separate helper function to insert a node in a linked list, make sure you are passing the **head** as a double pointer so that you can modify the **head** when required.

deleteMovie

```
void deleteMovie(std::string title)
```

[1] Using the first alphabet from the **title**, search for the appropriate node in the BST. If the node is missing, return without deleting. ***Use searchChar helper function.***

[2] If the BST node is found, search and delete the node in the accompanying linked list using the node deletion algorithm. **(Handle all the corner cases for deletion in the linked list – *delete the head, delete the last node, node not found, etc.*)**

[3] If, post-deletion, the accompanying linked list is empty, i.e., the last node was deleted – delete the BST node as well. ***Review how to delete a node in a BST. Remember to adjust the parent pointers if you delete a BST node.***

Just like in the case of insertion, we need to update the parent pointers after deletion. This is something we need to take care of once a recursive call is complete.
Beware of Segfaults: *In order to avoid segfaults, make sure the pointers you are accessing – exist.*

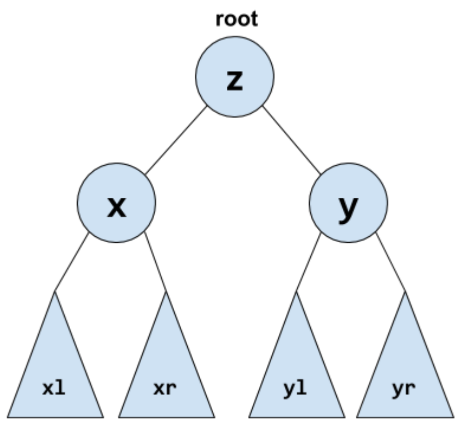
leftRotation

```
void leftRotation(TreeNode * curr)
```

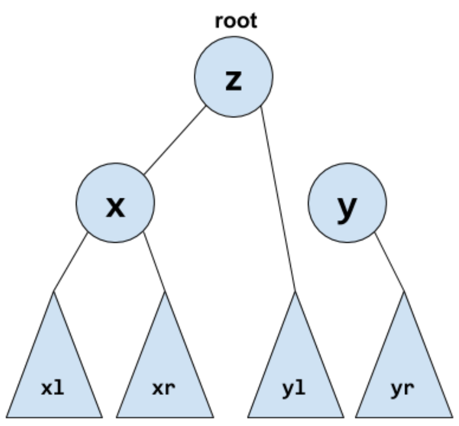
Tree rotations, coupled with the certain conditions, allow us to balance a BST. This is the principle that RBTs use to maintain their height property.

In order to perform left rotation, observe the possible (general) configurations of the tree, pre and post leftRotate:

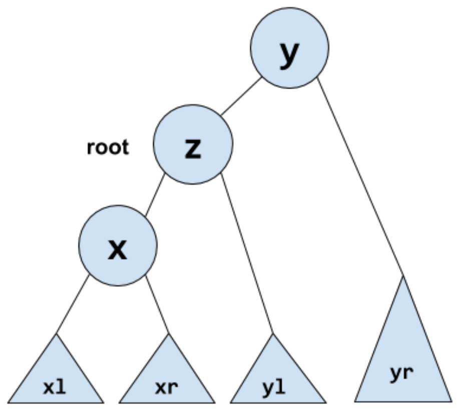
[A] leftRotate on root



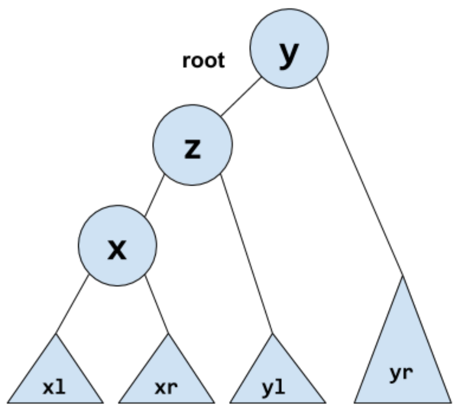
1. Original Tree



2. Modify y's left subtree connection



3. Modify y's left connection to z



4. Modify root

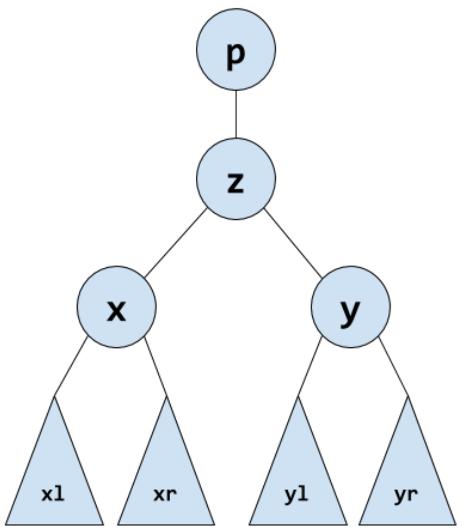
Notice that x and x's subtrees do not participate in any modification. Narrow down your list of links that you need to modify.

As usual, keep track of the nodes of interest using additional pointers. ***Remember to modify both child and parent pointers for each node.***

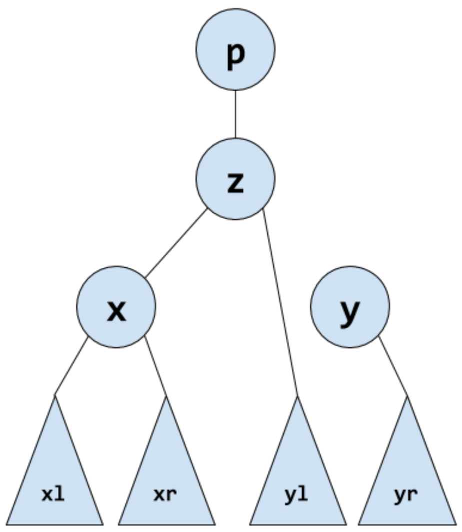
[B] leftRotate on any other node

Key Question: How will you figure out which child of **p** was **z** (Left or Right?), and later **y**.

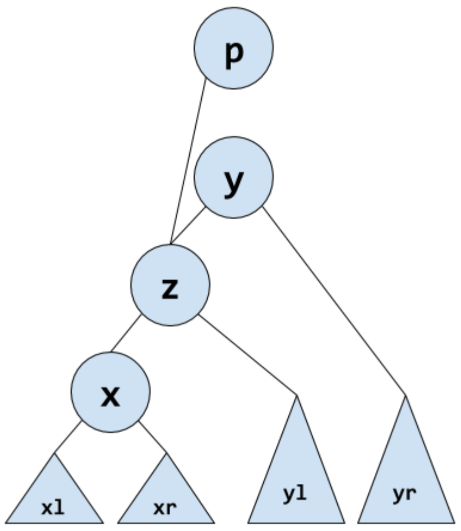
As usual, keep track of the nodes of interest using additional pointers. ***Remember to modify both child and parent pointers for each node.***



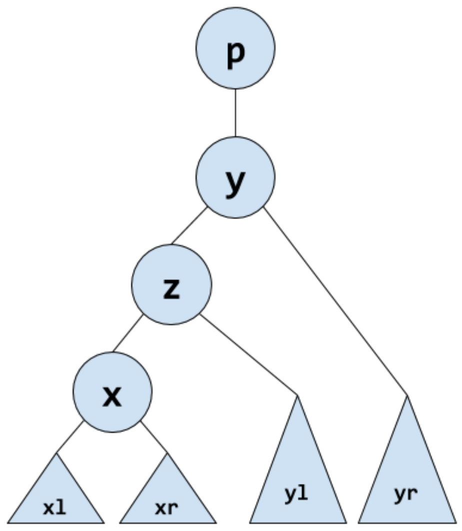
1. Original Tree



2. Modify y's left subtree connection



3. Modify y's left connection to z



4. Modify y's parent link and p's child

Consider both the cases**, **A and B:
It seems like the only difference between both cases is how the parent pointer of **y** is updated, and in the other case how **`root`** is also updated.

Tips to avoid segfaults
The above illustration states the most general case: What if either of $\mathbf{y}_{\texttt{left}}$ or $\mathbf{y}_{\texttt{right}}$ subtree does not exist?

Make sure you are not dereferencing a NULL address.