1. What are the heights of your movie (hm) and actor (ha) trees when you load the entire data set? Given the heights hm and ha, how many nodes can be stored in each tree, how many nodes are actually stored in these trees? In a short paragraph, describe how this differs from the results in your project 4.

For this project there is a hm of 6, and a ha of 3. The number of nodes that can be stored in 2^(h+1) -1 . For hm, this would be 2^7 -1 = 127 movies; for ha, it is 2^4 -1 = 31 actors. In actuality, there are only 5 actors stored and 39 movies. This is to be compared tp the last project in which the height was much larger to store the same amount of data. This caused a lot of space to be allocated for this. It was able to store 131071 movies and 7 actors. This shows that it will be take less time to find specifc objects since it is more compact.

2. Assume that you are given a binary search tree which uses lazy deletion.

a. Give a pesudo code algorithm for a new private member function in our tree class. This function must rebuild the BST in O(N) time. Assume that you are given the following private member function: vector<Node*>* inOrder(). Where inOrder returns a pointer to a vector containing pointers to all of the nodes in the tree, in sorted order (obviously, this is the result of an in-order traversal). Assume that inOrder also deallocates the "lazily" deleted nodes and runs in O(N) time.

First call would be BuildRecursive(Vector, m_root)

```
void BuildRecursive(vector<Node*>* V, Node *root){
        int halfway = floor(V.Size()/2)
        if(halfway>0){
                root=V[halfway];
                BuildRecursive(V[from 0 to halfway-1], root->left);
                BuildRecursive(V[from halfway+1 to V.size()], root->right);
        }
}
```

b. Give a good heuristic for deciding when to rebuild our tree without the "lazily" deleted nodes. Explain why your heuristic is reasonable, and how it effects the asymptotic bounds on both space and time. NOTE: a heuristic is simply a rule, it need not be perfectly optimal, but you should be able to explain in what ways it will impact time and space management.

If a count was added for the number of existing nodes, En, and number of deleted nodes, Dn, then when Dn>=En. This would be a good time to call it because half the tree is being allocated to non-existant nodes, so this would free up a lot of space, but at the same time would not be called too often to be a lag on the overall running time. An extreme of when to call it would be after every delete, and while this would free up a lot of memory, it would extend the bound of time expontentially. The method proposed would be a good middle ground because it isn't an extreme either way.

3. Write a pesudo code algorithm to perform an in-order traversal of a BST that does not use a recursive function call (i.e. the algorithm must be contained within a single looping structure in a single function). HINT: What data structure do you know of that will preserve the state variables in a recursive fashion?

   Sorry, I'm not sure what the hint is getting at, but here is a

```
voidInOrderWithoutRecursion(Node* root, std::ostream &p_str){
        if(root == NULL)
            return; //empty list
        Node * node = root;
        Node * prev;
        while(node !=NULL){
                //Check if left is null
                if(node->left==null){
                        //No more left branches, print and move to right
                        p_str<<node->Data();
                        node = node->right;

                }
                else{
                        prev = node->left;
                        while(prev->right && prev->right != node){
                                //move more to the right
                                prev = prev->right;
                        }
                        if(prev->right == NULL){
                                prev->right = node;
                                node = node->left;
                        }
                        else{
                                prev->right = NULL;
                                p_str<<node->Data();
                                node = node->right;
                        }
                }
        }
}
```

4. Give a big-Theta bound on the running time for your algorithm in number 3. Justify you answer with a short paragraph.

It would have a O(N) and a Omega(N), therefore it would also be Theta(N). It is big O(N), because no matter how well the algorithm is, it still must go to every single node, so it will go to N nodes. The Omega is also n because the only thing that is occuring is iterating to the nodes. There is nothing going on at each node besides adding it to the string. There are no nested for loops, but there are nested whiles loops, but those will not go through n-times for each.