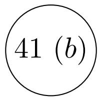
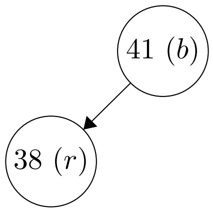
Ben Liepert and Gezim Saciri - Project 7

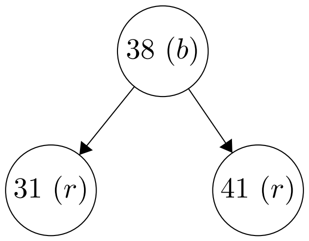
CS271 Data Structures

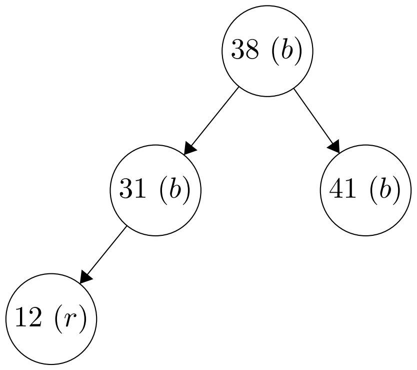
Wednesday, April 19th

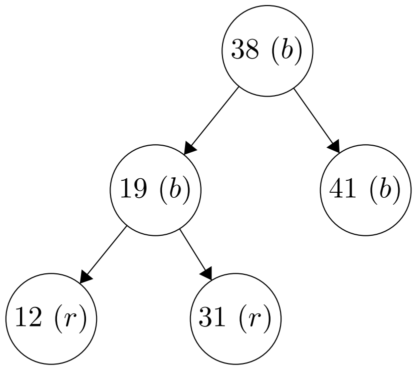
1.

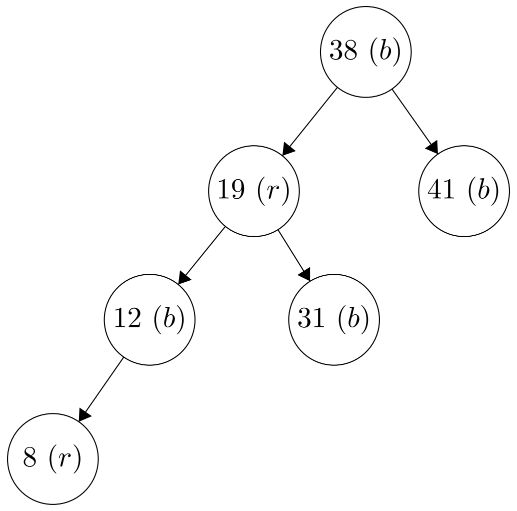












2. Proof Remember that if a node in an RBT is red, its children must be black. Thus, the longest possible path from a node to its leaf must alternate between red and black nodes. If we think of this path as the right subtree of the root, our left subtree must have equal black height. The shortest this path can possibly be, in order for the black height property to be satisfied, is if it is only comprised of black nodes. Thus, the right subtree has one red node for every black node on the path, because they are alternating. In other words, there would be 2*b nodes in this path where b is the number of black nodes. We know because of the black height property that there must be exactly b black nodes in the path down the left subtree, meaning there are two times the number of nodes in the right. The largest path is bound by 2 times the length of the shortest path because we don't necessarily need red nodes in the right subtree.

5.) On the *un-randomized* movies file, Binary Search Tree average time to insert over 100 trials: 4.9064
Hash Table average time to insert over 100 trials: 0.0535483
Red Black Tree average time to insert over 100 trials: 0.0258046
On the words file, the times averaged over 100 trials are:
BST: took so long for one trial (more than a minute) that we gave up and concluded it is terrible.
Hash Table: .485501 seconds
RBT: .194656 seconds
The fact that the RBT implementation of our dictionary class is faster than the Hash Table, which is faster than the BST, makes sense. For the BST, the alphabetic order of the movies causes an the BST to insert all values on the next right subtree, effectively creating a linked list. This will cause the BST to insert in $O(n^2)$ time. However, when we randomize our moviesmpaa.txt, our BST preforms much faster at avg time of 0.0711234, which is comparable to our Hash Table. Our Hash Table implementation is faster due to the fact that it inserts values at the head of its linked lists, which allows it to preform in constant time $O(1)$. However, due to implementation of our dictionary class, we must first see if a movie is already in the hash table, which in the worst case, where all movies are in the same slot, causes the runtime to be linear. Due to this, our runtime was not the optimal constant time a bare-bones hash table is capable of. The RBT, which is the most efficient implementation, runs insert in $O(lgn)$. This is due to the fact that a RBT can only have height $O(lgn)$ for n nodes, it only takes $O(lgn)$ to traverse to the farthest leaf and insert it. The insert-fixup function's while loop runs in $O(lgn)$ time for worst case, which doesn't affect the overall runtime. Therefore, we can see that RBT is the most efficient in terms of the dictionary implementation, due to its balanced tree which allows us to search for duplicates quickest in $O(lgn)$ worst case time and insert in $O(lgn)$ time.