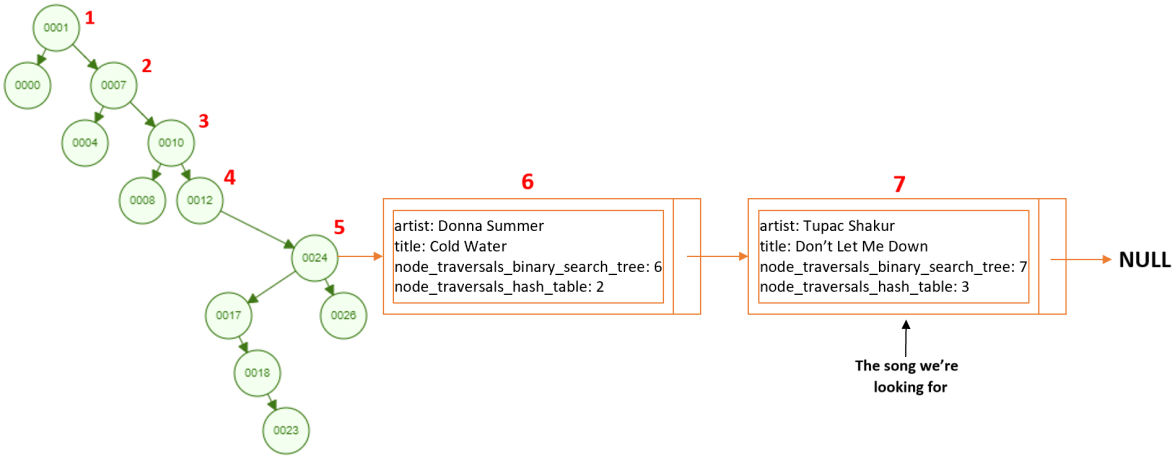
Project 4

This project consists of Python code that helps to store and retrieve songs, extracted from a text file, to and from two different data structures: a hash table and a hashed binary search tree. A hash function was used to map the song data (artist and title) to an index, and linked lists were used to resolve all collisions.

In order to compare the efficiency of both data structures, during the `search` operation to retrieve a song object, the number of node traversals within the data structures was monitored until the song was found.

The Hashed Binary Search Tree

Below, you can see what the hashed binary search tree data structure looks like for the first 20 lines of the `songlist.txt` file for `SIZE_OF_HASH_TABLE = 27`:



To easily explain what the data structure holds, only the linked list that was used to handle the collisions at index `24` is displayed in the figure above.

Essentially, in this project, a binary search tree node consists of four attributes: the index returned by the hash function, the linked list object to handle collisions, as well as the `left` and `right` pointers. Additionally, a linked list node consists of two attributes: the song object, and the `next` pointer.

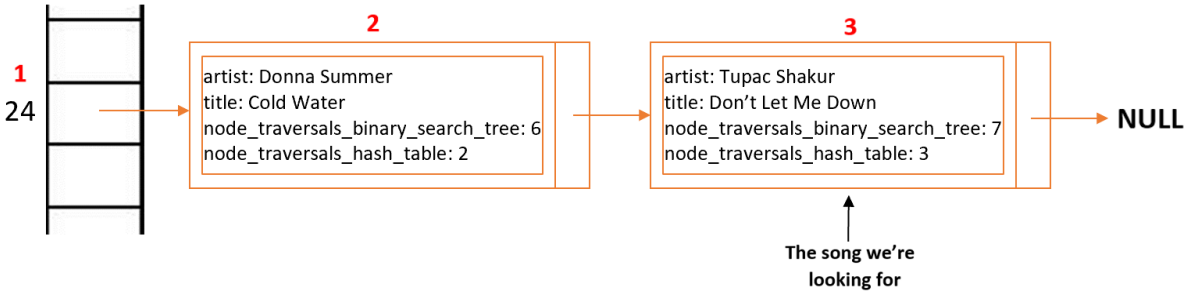
The song object is what we want to retrieve from the data structure. It consists of four attributes: the artist of the song, the title of the song, the count of traversals needed to fetch the song object from the binary search tree, and the count of traversals needed to fetch the song object from the hash table.

So, let us say that we want to retrieve *Tupac Shakur - Don't Let Me Down* from the hashed binary search tree above, using this project's code. As you can see, we would need to traverse seven nodes during the `search` operation to arrive to this song object in the data structure.

> To clarify, in the submitted code, the songs are searched right after they are inserted in the respective data structures, because each linked list node that holds the song object is appended to the linked list during insertion. This means that the position of a node in the linked list will not change as more nodes are inserted. This is why searching immediately after insertion produces an accurate node traversal count.

The Hash Table

Now, let us see what the hash table would look like for the same data:



As the figure above shows, a hash table location would point to the same linked list; however, in this case, to retrieve *Tupac Shakur - Don't Let Me Down*, there would need to be three traversals during the `search` operation (if we count finding the linked list that corresponds to the calculated index as one traversal).

> Note that, in both data structures, the song object is encapsulated in a linked list node. This allows us to retrieve the same song object, whether we search for it in the binary search tree, or in the hash table.

Comparison

From the figures above, it is clear that organizing the information in a hash table offers us a faster `search` operation, since the program would need less traversals than what it would need with a hashed binary search tree to find a song object, in general. Even if we insert all of the 7000 songs of the `songlist.txt` file in the respective data structures, on average, using this project's code, we would need ~17.3 node traversals to search and retrieve a song in the binary search tree, compared to ~2.4 node traversals needed to search and retrieve a song in the hash table (these results are obtained from the Python code).

However, it is also clear that the faster `search` operation comes with a memory downside: the hash table uses more memory than the hashed binary search tree to store the data. Although the number of collisions is the same in both data structures - we are using the same hash function for both of them - the hashed binary search tree uses exactly as many tree nodes as needed, based on the indexes obtained from the hash function, while the hash table, being fixed in size, will use more array locations to ensure a fast search operation, even though this may mean that some of these array locations may not be used at all to store song objects. For example, in the figures above, the binary search tree uses 12 tree nodes, while the hash table uses 27 array locations, which means that 15 of these locations are unused.

Therefore, **compared to the hash table, the binary search tree will offer a slower search time, but it will not use more memory than it needs**.

Additional Information

The project is organized into four different files:

- `main.py`: Run this file in order to see the result in your terminal.
- `settings.py`: This file defines the location of the `songlist.txt` file, as well as the size of the hash table.
- `utils.py`: This file contains utility functions needed to calculate the output of the hash function, read the data from the songs list file into the respective data structures, and calculate the average number of node traversals needed to find a song object in these data structures.
- `data_structures.py`: This file contains the necessary data structures and methods needed to complete this project.

In addition, inside the zipped folder, you will also find a file, called `songlist_20.txt`, which consists of the first twenty songs of the `songlist.txt` file. To verify the results I presented above, you simply need to change the size of the hash table, as well as the name of the songs list file in `settings.py`, and then run the `main.py` file.