# Big-(O)

## Class : DocumentProcessing

| Method | Time Complexity | Space Complexity |
|---|---|---|
| setupFilter(File f) | O(n) | O(n) |
| filterSpecialChars(String str) | O(m) | O(m) |
| filterStopword(String word) | O(1) | O(1) |

## Class : Document_Processing

| Method | Time Complexity | Space Complexity |
|---|---|---|
| Load(String fileName) | O(n * (k + m) + T * (p + W)) | Depends on index & filter size |
| getDocumentCount() | O(1) | O(1) |

# Class : Index

**1.Constructor (`Index`)**

**Time Complexity: O(1)**

**Complexity: O(1)**

**2.index(Documentd):**

**Time Complexity: O(1) Space Complexity: O(1)**

**3.add_Document(Documentd): Time Complexity: O(n)**

**Space Complexity: O(1)**

**4. getTotalTokens():**

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**5. getUniqueWords():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

# Class : Document

**1. Constructor Document(LinkedList<String> words, int id, String document)):**

**Time Complexity: O(n)**

**Space Complexity: O(n)**

**2. getContent():**

**Time Complexity O(1)**

**Space Complexity: O(1)**

**3. getId():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**4. getWords():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**5. isWordAvailable(String w):**

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**6. toString():**

**Time Complexity: O(m) where m is the length of the `document` string, Space Complexity: O(1)**

# Class : InvertedIndexBST

**1.Constructor_Document(LinkedList<String> words, int id, String document):**

**Time Complexity: O(n)**

**Space Complexity: O(n)**

**2. getContent():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**3. getId():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**4. getWords():**

**Time Complexity: O(1)**

**Space Complexity: O(1).**

**5. isWordAvailable(String w):**

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**6. toString():**

**Time Complexity: O(m)**

**Space Complexity: O(1)**

# Class : Word

1. **searchIfExists(Integer id):**

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**2. addId(int id):**

**Time Complexity: O(n)**

**Space Complexity: O(1)**

**3. getText():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**4. getIds():**

**Time Complexity: O(1)**

**Space Complexity: O(1)**

**5. toString():**

**Time Complexity: O(n)**

**Space Complexity: O(n)**

# Class : InvertedIndexNode

1. **Space Complexity**: - Each `InvertedIndexNode` contains a `LinkedList` of `Frequency` objects, a `String` for the word, and two pointers for the left and right child nodes. - The space used by the `LinkedList` depends on the number of unique sentence IDs associated with the word. If there are `n` unique sentence IDs, the space complexity for the `LinkedList` is O(n). - The overall space complexity for a single `InvertedIndexNode` is O(n) for the linked list plus O(1) for the other attributes, resulting in O(n) for the node itself.

2. **Time Complexity**: - The constructor `InvertedIndexNode(String word, int sentenceId)` initializes the node and adds the first `Frequency` object to the linked list. This operation takes O(1) time. - The `addId(int sentenceId)` method performs the following operations: - It checks if the last element in the linked list has the same document ID as the one being added. This operation involves accessing the tail of the linked list, which is O(1). - If the document ID matches, it increments the frequency, which is also O(1). - If the document ID does not match, it adds a new `Frequency` object to the linked list. The insertion operation in a linked list is O(1). - Therefore, the time complexity for the `addId` method is O(1) for both checking and adding.

# Class : QueryProcessing

1. **andQuery Method:** - **Time Complexity:** The method uses a two-pointer technique to traverse both linked lists. In the worst case, each pointer will traverse the entire length of its respective list. If `n` is the length of `list1` and `m` is the length of `list2`, the time complexity is O(n + m). This is efficient since each list is only traversed once. - **Space Complexity:** The space complexity is O(k), where `k` is the number of common document IDs found in both lists. This is because the result list stores only the intersecting document IDs. The space used by the input lists is not counted in this analysis.

2. **orQuery Method:** - **Time Complexity:** The `orQuery` method is not fully implemented in the provided code, but if it follows a similar two-pointer approach as `andQuery`, the time complexity would also be O(n + m) for traversing both lists. If it involves additional operations to ensure uniqueness of document IDs, the complexity could increase, but typically it remains linear. - **Space Complexity:** Similar to `andQuery`, the space complexity would be O(k), where `k` is the total number of unique document IDs found in both lists. If the implementation requires additional data structures to handle duplicates, the space complexity could increase accordingly.

# Class : Ranking

**Time Complexity:**

1. **Merge Sort:** The merge sort algorithm has a time complexity of O(n log n), where n is the number of nodes in the linked list. This is because the list is repeatedly divided in half (log n divisions), and each division requires a linear time merge operation (O(n)).

2. **Overall Complexity:** The `rankedRetrival` function calls `mergeSort`, so the overall time complexity for sorting the linked list remains O(n log n).

3. The time complexity of the merge function is O(n + m), where n is the number of nodes in the left list and m is the number of nodes in the right list. This is because the function iterates through both lists once, comparing their elements and appending them to the result list. Each node from both lists is processed exactly once.

**Space Complexity:**

1. **Auxiliary Space:** The merge sort algorithm requires O(n) space in the worst case for the merged list. However, since the algorithm is implemented on a linked list, it does not require additional arrays or lists for sorting, and the space used is primarily for the recursive stack.

2. **Recursive Stack Space:** The recursive calls in merge sort will use O(log n) space for the call stack due to the depth of the recursion.

3. **Overall Space Complexity:** The overall space complexity is O(n) due to the need to create new nodes during the merge process, although the actual space used can vary based on the implementation details.

4. The space complexity is O(n + m) as well, primarily due to the creation of new nodes for the merged list. For each node in the left and right lists, a new node is created in the merged list. Therefore, if there are n nodes in the left list and m nodes in the right list, the total number of new nodes created will be n + m. Additionally, the temporary node used to hold the result does not significantly affect the overall space complexity.

## Class : Frequency

- The constructor `Frequency(int documentId)` initializes the `documentId` and sets `frequency` to 1. This operation is performed in constant time, O(1). - The `increment()` method increases the `frequency` by 1. This operation is also performed in constant time, O(1).

- The getter methods `getDocId()` and `getFrequency()` return the values of `documentId` and `frequency`, respectively. Both of these methods operate in constant time, O(1).

n: Number of lines in the file.

k: Average length of a line.

m: Average number of characters in a line.

T: Total number of words in the file.

p: Average length of a word.

W: Number of unique words in the inverted index.