1. Implement the term weighting formula: Calculate the tf-idf (term frequency-inverse document frequency) score for each term in the inverted file. Divide the tf by the maximum tf in the document and multiply it by the idf score.

- Initialize a dictionary to store the term frequency (tf) for each term in each document.

- Iterate over all documents:
  - Initialize the tf dictionary for the current document.
  - Iterate over all terms in the inverted file.
  - Calculate the term frequency for the current term in the current document.
  - Store the term frequency in the tf dictionary.

- Initialize a dictionary to store the maximum term frequency in each document.

- Iterate over all documents:
  - Calculate the maximum term frequency in the current document.
  - Store the maximum term frequency in the max_term_frequency dictionary.

- Initialize a dictionary to store the document frequency for each term.

- Iterate over all terms in the inverted file:
  - Calculate the document frequency for the current term.
  - Store the document frequency in the document_frequency dictionary.

- Initialize a dictionary to store the idf score for each term.

- Calculate the inverse document frequency (idf) for each term:
  - Iterate over all terms in the inverted file.
  - Calculate the idf score using the formula: idf = log(total_documents / (1 + document_frequency[term])).
  - Store the idf score in the idf dictionary.

- Initialize a dictionary to store the tf-idf score for each term in each document.

- Iterate over all documents:

- Initialize the tf-idf dictionary for the current document.
- Iterate over all terms in the inverted file.
- Retrieve the term frequency, maximum term frequency, and idf score for the current term and document.
- Calculate the tf-idf score using the formula: tf-idf = (tf / max_tf) * idf.
- Store the tf-idf score in the tf_idf dictionary.

2. Implement the document similarity measure: Calculate the cosine similarity between the query vector and each document vector. The document vector consists of the tf-idf scores for each term in the document.

- Create a function that calculates the cosine similarity between two vectors. The cosine similarity is a measure of similarity between two non-zero vectors based on the cosine of the angle between them.

- Define a function that computes the document vector for each document using the tf-idf scores. The document vector will consist of the tf-idf scores for each term in the document.

- Construct the query vector using the tf-idf scores for each term in the query.

- Iterate over each document:

- Calculate the document vector for the current document.
- Calculate the cosine similarity between the query vector and the document vector using the cosine similarity function.
- Store the cosine similarity score for the current document.
- Sort the documents based on their cosine similarity scores in descending order.

3. Implement the mechanism to favor matches in the title: Boost the rank of a document if the query terms are found in the title. You can assign a higher weight to terms found in the title when calculating the document score.

- Assign a higher weight to terms found in the title when calculating the document score. You can define a weight factor to increase the importance of terms appearing in the title.

- Iterate over each document:

- Check if any of the query terms are present in the title of the document.
- If a query term is found in the title, multiply its tf-idf score by the weight factor.
- Calculate the document score incorporating the modified tf-idf scores.

4. Implement PageRank algorithm: Assign an initial rank score to each document. Then, iteratively update the rank scores based on the inbound links from other documents. The more inbound links a document has, the higher its rank score.

To implement the PageRank algorithm and assign rank scores to each document based on inbound links from other documents, you can follow these steps:

1. Assign an initial rank score to each document. The initial rank score can be a uniform value or any other suitable value depending on your preference. For simplicity, you can assign an initial rank score of 1.0 to each document.

2. Iterate over a fixed number of iterations or until convergence is reached:
    - Initialize a dictionary to store the updated rank scores for each document.
    - Iterate over each document:
        - Calculate the inbound link contribution to the rank score for the current

document.

- Iterate over all other documents:
  - Check if the current document is linked from the other document.
  - If there is a link, add the rank score of the other document divided by the total number of outbound links from the other document to the inbound link contribution.
- Update the rank score for the current document by combining the inbound link contribution with a damping factor (typically 0.85) and adding a small constant (e.g., 0.15) to ensure that all documents have a non-zero rank score.
- Store the updated rank score for the current document in the dictionary.

3. Normalize the rank scores to ensure they sum up to 1.0 or any other desired value.
   - Calculate the sum of all rank scores.
   - Divide each rank score by the sum to normalize them.

Here's a Python code skeleton to help you get started:

```python
# Assign an initial rank score to each document
rank_scores = {document: 1.0 for document in documents}

# Iterate over a fixed number of iterations or until convergence
for iteration in range(num_iterations):
    updated_rank_scores = {}
    for document in documents:
        inbound_link_contribution = 0.0
        for other_document in documents:
            if document != other_document and is_linked(other_document, document):
                outbound_links = get_outbound_links(other_document)
                inbound_link_contribution += rank_scores[other_document] / len(outbound_links)
        updated_rank_scores[document] = damping_factor * inbound_link_contribution + (1 - damping_factor) / len(documents)
    rank_scores = updated_rank_scores

# Normalize the rank scores
sum_rank_scores = sum(rank_scores.values())
```

```
normalized_rank_scores = {document: score / sum_rank_scores for document, score in rank_scores.items()}
```

In the code skeleton above, `documents` represents the list of all documents in your collection. The `is_linked` function checks if there is a link from one document to another. The `get_outbound_links` function retrieves the outbound links from a given document. Adjust these functions according to your specific implementation and data structures.

You can modify the `num_iterations` variable to control the number of iterations for the PageRank algorithm. Additionally, you can adjust the damping factor as needed to balance the influence of inbound links and the uniform rank score.

Remember to adapt the code to fit your specific implementation and data structures.

5. Update the retrieval function: Incorporate the PageRank scores into the document scores calculated using the term weighting formula and document similarity measure. Consider the PageRank as an additional factor when ranking the documents.

To update the retrieval function and incorporate the PageRank scores as an additional factor when ranking the documents, you can modify the existing code by following these steps:

1. Retrieve the PageRank scores for each document.

2. Adjust the document scores calculated using the term weighting formula and document similarity measure by incorporating the PageRank scores.

3. Rank the documents based on the adjusted document scores.

6. Rank the documents: Sort the retrieved documents in descending order of their scores, taking into account both the document scores and the PageRank scores.

7. Implement the web interface: Create a JSP page with a text box for the user to enter the query. Submit the query to the retrieval function and display the returned results.

Make sure to test your implementation thoroughly to ensure it meets the requirements and functions correctly, taking into account the PageRank algorithm for result ranking.