Data Structures Used

Format: Below is a list of the most important data structures in the program. Each item consists of a name (as it appears in the code) and a description. Also a comment on why it was used may be included.

The Stemmer

tokens: An array created by the tokenization of the documents. Contains all words form the document collection.

filtered_sentences: Same as the token array but with all stop words removed

The Indexer

The index of an IR system is what allows you to search documents without having to store copies of every document and iterate over the entire collection. Using the standard terminology, each distinct term in the document corpus is mapped to a variable size postings list (just called posting in the Indexer class). The postings list tells us each document that a term occurs in and how many times in each document. The index must support fast term lookup as it is queried very frequently during search. It is less important that adding documents to the index be as quick as the querying part is user facing and time critical. The choice was made to have an unsigned int be the primary document specifier rather than its name, this choice was made because it's more space efficient than using an arbitrary length string and also it's a lot quicker to compare ints than strings. There is of course efficient accesor methods to convert between doc_id and doc_name for displaying to the user.

For the postings list (posting) I chose to use a std::vector filled with pairs of unsigned integers. In each pair the first element is the doc_id and the second element is the frequency in that doc. The vector was chosen as it has an extremely dense representation in memory (reduces to an array). I believe this density is preserved when the elements are pairs because as far as i know there are no pointers in the pair implementation. This means we have an ideal collection for iterating over, which is exactly how the posting is used.

The Indexer class has three, private, data members.

The core of the Indexer is in the 'index' member variable. I chose to use a std::unordered_map for this. 'index' maps a string(term) to its posting. I chose to use unordered_map as it has constant time lookup, this is crucial as it will be queried so often.

The second data member is another unordered_map called 'doc_stats'. This maps document id's(the main way of referencing docs internally) to a pair of string(doc_name) and unsigned int(doc_length). Lookup speed is less crucial here as this map's size is proportional to the number of docs rather than the number of terms, though using an unordered_map still makes internal lookup fast.

The third data member is just a float 'avg_doc_length'. A float was chosen as it's half the size of a double(negligible saving really) and we would never actually need to know it's precision past a couple of decimal places. I chose to make this a data member as it's needed in all the BM25 score calculations and to compute it is quite inefficient(O(#OfDocs)) so it's only recomputed when a new document is added to the index.

The Rank Generator

queries: A vector of multisets. Each multiset contains strings (query words). In C++ a vector is just an array. A multiset is a structure where the contents are automatically sorted upon insertion.
Reasons for use: A vector stores its contents in a compact way so that is best for efficient space storage. A multiset provides a count method which has a logarithmic complexity (lookup) * linear complexity (count item). This count function was required to get term frequency in a query of a certain term

ranks: A vector of maps. In C++ a map is an object that stores key value pairs sorted by key. In this case the keys were document ids and the values were BM25 scores.
Reasons for use: Vector reasons are same as above. A map was used in this case because this allows for efficient lookup times. Maps don't do well when they are being iterated over so this was kept to a minimum. Every time I updated a BM25 score I used a lookup. Since I was doing lots of updating I chose to use a map.

The evaluator

A slightly modified version of ranks was used here but the core data structure was the same as above.

rel: A vector of sets. Each set contains integers (documents considered relevant to a query by outside judgement). In C++ a set is the same as a multiset (discussed above) except that multiple instances of a value are not allowed.
Reasons for use: A set was used because of the count function which in the case of a set has logarithmic complexity. The existence of elements in the set needed to be checked on a regular basis and set allowed me to do this most efficiently.

results: A vector of vectors. Each inner vector stored doubles (precision scores for later graphing).