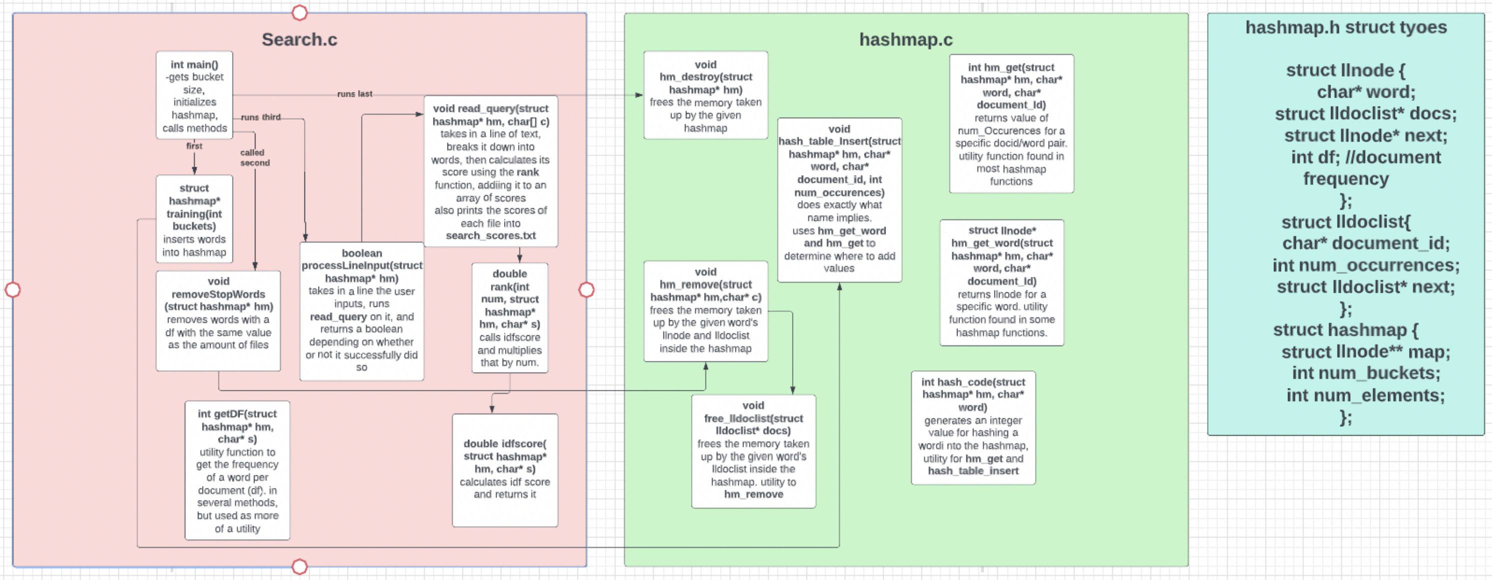
Search.c
int main() -gets bucket size, initializes hashmap, calls methods
runs last
runs third
first
called second
struct hashmap* training(int buckets) inserts words into hashmap
void removeStopWords (struct hashmap* hm) removes words with a df with the same value as the amount of files
boolean processLineInput(struct hashmap* hm) takes in a line the user inputs, runs read_query on it, and returns a boolean depending on whether or not it successfully did so
void read_query(struct hashmap* hm, char[] c) takes in a line of text, breaks it down into words, then calculates its score using the rank function, addiing it to an array of scores also prints the scores of each file into search_scores.txt
double rank(int num, struct hashmap* hm, char* s) calls idfscore and multiplies that by num.
int getDF(struct hashmap* hm, char* s) utility function to get the frequency of a word per document (df). in several methods, but used as more of a utility
double idfscore(struct hashmap* hm, char* s) calculates idf score and returns it
hashmap.c
void hm_destroy(struct hashmap* hm) frees the memory taken up by the given hashmap
void hash_table_Insert(struct hashmap* hm, char* word, char* document_id, int num_occurences) does exactly what name implies. uses hm_get_word and hm_get to determine where to add values
int hm_get(struct hashmap* hm, char* word, char* document_Id) returns value of num_Occurences for a specific docid/word pair. utility function found in most hashmap functions
struct llnode* hm_get_word(struct hashmap* hm, char* word, char* document_Id) returns llnode for a specific word. utility function found in some hashmap functions.
void hm_remove(struct hashmap* hm,char* c) frees the memory taken up by the given word's llnode and lldoclist inside the hashmap
void free_lldoclist(struct lldoclist* docs) frees the memory taken up by the given word's lldoclist inside the hashmap. utility to hm_remove
int hash_code(struct hashmap* hm, char* word) generates an integer value for hashing a wordi nto the hashmap, utility for hm_get and hash_table_insert
hashmap.h struct tyoes

```
struct llnode {
    char* word;
struct lldoclist* docs;
struct llnode* next;
int df; //document
frequency
};
struct lldoclist{
char* document_id;
int num_occurrences;
struct lldoclist* next;
};
struct hashmap {
struct llnode** map;
int num_buckets;
int num_elements;
};
```

HASHMAP:

struct hashmap* hm_create(int num_buckets);

-allocates memory for the hashmap and the array that contains each llnode bucket

-for loop that sets each bucket to null

int hm_get(struct hashmap* hm, char* word, char* document_id);

-uses hash_code to get to the right bucket, then uses a while loop to go through every word, and compares it to the argument. If the word exists, it proceeds to go through word->docs->document_Id and compaeres it to the argument of the same name. If it exists, it returns docs->num_occurences. Else, it returns 01.

struct llnode* hm_get_word(struct hashmap* hm, char* word);

--uses hash_code to get to the right bucket, then uses a while loop to go through every word, and compares it to the argument of the same name. If it finds it, it returns the llnode. Else, it returns null.

void free_lldoclist(struct lldoclist* docs);

-goes through a lldoclist and frees every value inside with a while loop

void hash_table_insert(struct hashmap* hm, char* word, char* document_id, int num_occurrences);

-uses **hm_get, hm_get_word,** and an else statement to determine if the word exists in the hashmap (get_word), the word/document pair exists in the hashmap (hm_get), or if neither (else).

-if hm_get_word is null, then it uses **hash_code** to find the right bucket, then inserts the new llnode/lldoclist into the head of the bucket.

-else if hm_get equals -1, then it uses **hash_code** and a while loop to get to the right word, then adds a new lldoclist item to the head of the current word's lldoclist

-else, it uses **hash_code** and a while loop to get to the right word, with a while loop inside to find the right document_id. It then adds 1 to the current doc's num_occurences. .

void hm_remove(struct hashmap* hm, char* word);

-uses **hash_code** to get to the right bucket for the word. Then, it checks to see if the head contains said word. If it does, it calls **free_lldoclist**, then frees the node's word and then the node itself before returning.

-if it is not in the head, it goes through a while loop and checks each llnode in the bucket to see if it contains said word. If it does, it calls **free_lldoclist**, then frees the node's word and then the node itself before returning.

void hm_destroy(struct hashmap* hm);

-uses a for loop to go through each bucket in the hashmap. If the bucket is not null, it goes through a while loop that first frees the word, then calls an inner while loop that frees each document. It then proceeds to free the current llnode and moves on to the next one. After this loop, it frees the map and the hashmap itself.

int hash_code(struct hashmap* hm, char* word);

-takes in a word, then adds the ASCII value of each character. It then mods that by the amount of buckets in the hashmap and returns the result.

SEARCH:

struct hashmap* training(int s);

-runs **hm_create,** then uses **fopen()** to open up each doc and extract each word, using a while loop to ensure each character is not EOF. in this while loop, it adds each char to an array, and uses the null terminator when it hits a space. It then hm_adds the word and the document. It then resets the char* and does it for the next word. It repeats this for each file.

void read_query(struct hashmap* hm,char* s);

-first, it initializes a double array of size 3 with zeros for score. It then runs a for loop that goes for the length of s, and adds each char to another array, stopping when it hits a space.

-If the word exists in the hashmap, it then looks for it using a while loop and **hash_code.** Once it gets to the correct word, it goes through that llnode's docs, and sets score[i] to **rank** of the doc's num occurrences, the hashmap, and s. It repeats this until it runs out of words in the query.

double rank(int num, struct hashmap* hm,char* s);

Takes num, multiplies it by idfscore, and returns the result.

int getDF(struct hashmap* hm,char* s);

-returns the document frequency of a word by running **hash_code** of s, then running a while loop of that bucket until it reaches the target. It then returns cur->df.

double idfScore(struct hashmap* hm,char* s);

-takes a hm and char pair, runs **getDF on it, and** if DF = 0, then it returns log(N). Else, it returns Log(N/df), where N is the number of documents.

void removeStopWords(struct hashmap* hm);

-runs a for loop that goes through each bucket of hm,
Then, if the bucket is not null, it goes through a while loop to go through each llnode. If the DF of any word is 3, it runs **hm_remove** to get rid of it, then resets the current to hm->map[i] to prevent it skipping any possible words.

bool processLineInput(struct hashmap* hm);

-Prints a line to insert the search string, then uses **fgets()** to take in a input. If its not null, it first checks to see if it equals "X", in which case it return false and exits to main. If it doesn't it adds a space and a text character at the end of the inserted value to make it easier to arse in **read_query**. It then runs **read_query** using fgets's input plus the " c" and returns true.

Int main()

-inside of a while loop attached to a boolean, it checks to see if a user inputs an integer and stores it into buckets. If they do not, it returns and prints an error. If they insert a value less than 1, it gives the user the option to try again. If they insert a number greater than 1, it sets the boolean to false, breaking the while loop. It then runs **training(**buckets) with the value from the while loop to create the hashmap. It then runs **removestopwords** to remove all words that are in every document.

With another boolean and a while loop attached,it proceeds to run **processLineInput** until it returns false. It they runs **hm_destroy** and returns 0.