Task 1

a. *add_elt()* adds a new struct element with the value *val* to the linked list.
The first element of the list given as the argument *head*.
If *val* is NULL, the function does nothing, simply returning the head of the list.
Otherwise, a new struct is created using *malloc()*, with the *val* item set as the argument *val*, and the *next* item set as the pointer to the current head of the linked list. The newly created element is then returned.

free_list() walks through the linked list with the head given as argument *list*.
For each element, the function given as argument *free_elt* is called with the value of the element. The element is then *free()*'d to allow the memory it used to be allocated for other purposes.

b. *read_list()* opens the file with the name given as argument *file_name* in read mode.
If the open operation failed, an error is printed, and the function returns NULL.
Otherwise, the file is read in chunks into the character buffer *buffer*, then a new element is added to a linked list using *add_elt()* until the end of the file.
The file is then closed and the head of the new linked list is returned.

The argument *parse* is a pointer to a function which takes a character pointer as an argument, and returns a void pointer. It is called while reading the file in buffers in order to provide the value (second argument) that should be added to the element in *add_elt()*. It takes the buffer's value to determine this.
This is a function that could be used, with the second argument of *read_list* being *void *(*readInt)(char *)*:

```
void *readInt( char *buf ) {
  void *i = malloc( sizeof( int ) );
  sscanf( buf, "%d\n", (int *)i );
  return i;
}
```

c. (Assuming ascending for explanation)
The merge sort algorithm separates all elements of the linked list into halves repeatedly until all elements are separated. For each adjacent pair, the values are compared, and the last significant value is added first, then the other next. This produces multiple two-element lists which are each in the desired order.
This operation is repeated, with each newly created adjacent 'sublist' compared with one another. The elements of the first list are walked through and added if they are smaller then the next element of the second, if the next element of the second if smaller, the second is then walked. This operation swaps between both lists until all elements have been added to the new list.
The algorithm implements the divide and conquer methodology by starting with sublists of length one and forming small sublists which are built up into larger ones.

d. *sort()* is used to sort the linked list with first element given by argument *x*, using the merge algorithm. If the given argument *x* is NULL, or if it is a single element, *x* is returned. Otherwise, the function finds the middle element *y* of *x* (using *divide)* and calls *merge()* with the results of calling itself again with arguments *x* and *y*. When *merge()* or *sort()* is called, the argument *comp()* is given so that in the recursive calling the comparison function is still available.

merge() implements the actual sorting, using the *comp()* function to compare the two arbitrary values of each element, given as void pointers. *comp()* returns a non-zero value if the first argument is greater than the second. It allows *merge()* to add elements from the two given lists to a new one in a desired order.

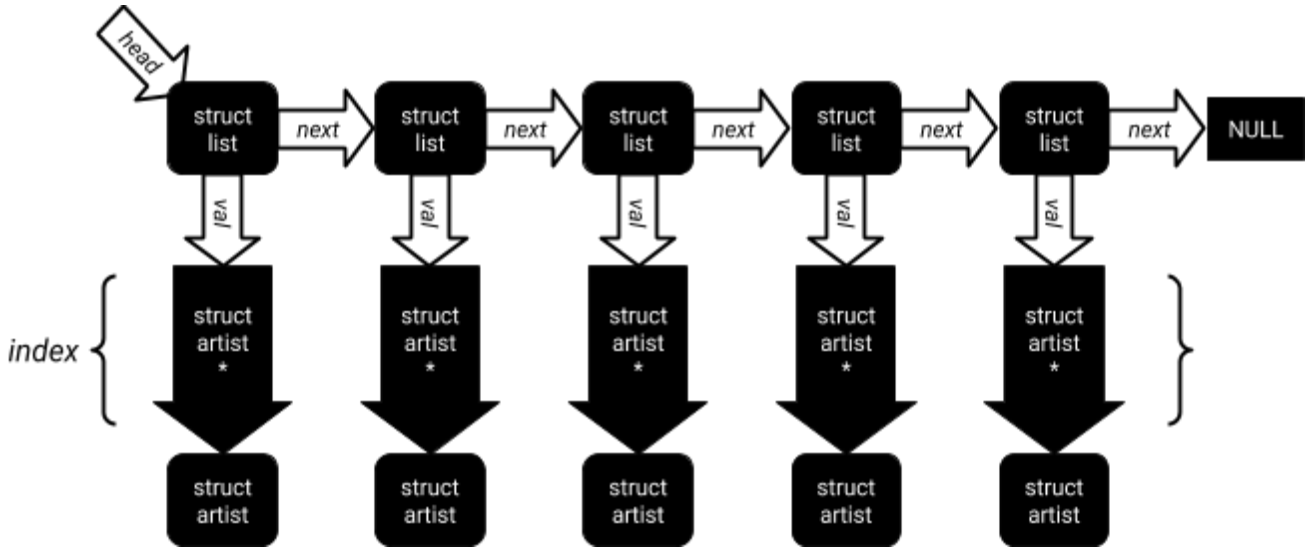
An example function that could be used as the second argument for *sort()* could compare two integers to see if a is larger than b, which would sort the list descending:

```
int comp_ints( void *a, void *b ) {
  return *(int *) a > *(int *) b;
}
```

Task 2

a. *create_index()* first creates an array *index* using *malloc* of size *n *(size of struct pointer)* so that a pointer of each list element can be inserted into it.
If the memory was successfully allocated, it then walks the linked list. The *val* member of each element is casted to a *struct artist* pointer, then appended to the array.
The array is then returned.

b. Black shapes are data objects, white shapes are arguments and members of structs.
Pointers are represented as arrows. The NULL pointer has been labelled as such.
The contents of the array *index* is between the curly brackets.



Task 4

a. *register_fans()* walks list of plays (given by *plays*), for each play using *lookup_artist()* to find a pointer to the artist object that the play's artist is found in the array *index*.
If the artist is in the top 100 most played by the user, the user is registered as a fan of that artist, using *register_fan()*, otherwise, the playcount of the play is added to the *count* member of the artist struct.
This effectively creates a list of fans for each artist.
The list of fans of each artist is then sorted by ascending user ID.

b. *similarity()* determines and returns similarity as the percentage of people that are among both artists' fans.
To do this, the list of both artists' fans is walked, and for each fan a counter for the respective group is incremented, calculating the total number of fans that are in each group.
The user IDs of the fans are then compared, a third counter is incremeneted if they are equal, calculating the number of people who are fans of both artists (i.e. the intersection).
If the users were the same, we move on to the next element of both lists, otherwise we move onto the next element of the list with the currently lowest user ID.
Finally, the intersection is divided by the total unique number of users, which is the sum of fans of both artists take away the intersection. This result is then returned.

c. *Recommend* first reads all artist and play data into their own linked lists. The length of the artist linked list is found for use later on.
The plays are then sorted by increasing user ID, and decreasing play count for each user, while artists are sorted by increasing user ID.
An index array is then created for the artists so that it can be searched to easily find the artist struct that has the input artist ID, allowing the program to find similar artists.
The fan lists are then created for each artist, identifying the users with at least 100 plays for each artist.
An artist ID is then input, which is lookup up in the artist index. If no matching artist struct is found, an error is printed and input is restarted. If a valid artist is given, the artist list is sorted by similarity and the 25 most similar artists are printed.
If the input is 0 the input loop is broken, and finally the artists, plays, and artist index is free()'d.