1. Introduction. This is K-INDEX, an implementation and a testbed of a scientometric measure to rank research authors using the citations of articles' citations. The testbed is its use to try to predict the the recipient of scientific awards. We use the Web of Science data as input and use the results for the prediction of Nobel Laureates of Physics.

2. The following code is structured in a way that each section has a well defined boundary in terms of use of variables and functionality. We try to separate the technicalities, like memory management, from logical statements related to the index calculation.

```
#include <stdio.h>
#include <stdlib.h>
  ⟨ Preprocessor definitions ⟩
  ⟨ Include headers 32 ⟩
  ⟨ Macro declarations 8 ⟩
  ⟨ Type definitions 38 ⟩
  ⟨ Data structures 26 ⟩
  ⟨ Internal variables 13 ⟩
  ⟨ Static functions 5 ⟩
  ⟨ Functions 58 ⟩
```

3.
```
int main(int argc, char **argv)
  {
    ⟨ Local variables 27 ⟩
    ⟨ Parse program arguments 12 ⟩
    ⟨ Initialize the variables that need memory allocation 29 ⟩
    ⟨ Load the ids of Nobel Laureates 47 ⟩
    ⟨ Load authors information 30 ⟩
    ⟨ Calculate K index 55 ⟩
    ⟨ Sort the authors 70 ⟩
    ⟨ Write results to a file 72 ⟩
    ⟨ Write a table with the twelve larger ks in latex format 74 ⟩
    ⟨ Free up memory 31 ⟩
    ⟨ Print information about flags 14 ⟩
    return 0;
  }
```

4. Some internal functions are defined to embed repetitive tasks like check null pointers and print error messages.

5. *Fopen* function try to open a file named *filename* and if it fails the error message is thrown out, stopping the execution of the program.

```
⟨ Static functions 5 ⟩ ≡
  static FILE *Fopen(char *filename, char *mode)
  {
    FILE *f;
    f = fopen(filename, mode);
    if (¬f) {
      fprintf(stderr, "Could␣not␣open␣%s\n", filename);
      exit(−1);
    }
    return f;
  }
```

See also sections 6, 7, 9, 19, 20, 21, 22, 35, 39, 40, 41, 42, 43, 44, 45, 51, 67, 68, and 69.

This code is used in section 2.

6. The function *Fclose* tests it the file pointer in not null before closing it.

```
⟨ Static functions 5 ⟩ +≡
  static void Fclose(FILE *f)
  {
    if (f) fclose(f);
  }
```

7. The *mem_free* function check the nullity of the address pointed by *ptr* before deallocation.

```
⟨ Static functions 5 ⟩ +≡
  static void mem_free (void *ptr, int line )
  {
    if (ptr) free(ptr);
  }
```

8. `CALLOC` macro hide the use of macro `__LINE__` that is the number of current line where it is used. When an memory allocation error occurs the line where `CALLOC` is used is informed by *mem_calloc*.

```
⟨ Macro declarations 8 ⟩ ≡
#define CALLOC(count, nbytes)mem_calloc ((count), (nbytes), (int) __LINE__)
```

See also sections 10 and 11.

This code is used in section 2.

9. The function *mem_calloc* is an wrapper to *calloc* to check it the pointer returned is not null.

```
⟨ Static functions 5 ⟩ +≡
  static void * mem_calloc (int count, int nbytes, int line ) { void *ptr;
  ptr = calloc(count, nbytes); if (¬ptr) { fprintf (stderr, "%d:␣Null␣pointer\n", line ) , abort( ); }
      return ptr; }
```

10. `FREE` macro wraps *mem_free* with proper arguments and zeroed *ptr*.

```
⟨ Macro declarations 8 ⟩ +≡
#define FREE(ptr) ((void)(mem_free((ptr), __LINE__), (ptr) = 0))
```

11. The *panic* function is used when the program enters in a condition that was not expected to be in. It stops the program execution and prints a message *msg*. If there was a sure expectation that nothing bad can occurs, a definition of `NDEBUG` as macro turn off the *panic* function.

```
⟨ Macro declarations 8 ⟩ +≡
#undef panic
#ifdef NDEBUG
#define panic(msg) ((void) 0)
#else
  extern void panic(int msg);
#define panic(msg) (fprintf(stderr, "%d: PANIC: %s\n", (int) __LINE__, msg), abort())
#endif
```

12. Verbose mode. The flag `-v` is provided to print the existing comments inside data files and any other useful information to the user.

```
#define VERBOSE_FLAG "-v"
```

⟨ Parse program arguments 12 ⟩ ≡

```
if (argc ≡ 2 ∧ ¬strncmp(argv[1], VERBOSE_FLAG, 3))
    verbose = 1;
```

See also sections 16 and 24.

This code is used in section 3.

13. The *verbose* Boolean variable marks if the output of the program is extended with the comments inside data files. The default behavior is to write to the output the name the generated files.

⟨ Internal variables 13 ⟩ ≡

```
static int verbose = 0;
```

See also sections 15, 18, 28, 37, 66, and 73.

This code is used in section 2.

14. Warn the user about the `-v` if the flag was not used.

⟨ Print information about flags 14 ⟩ ≡

```
if (¬verbose)
    fprintf(stderr, "- use \"%s -v\" to print information about data set.\n", argv[0]);
```

See also section 17.

This code is used in section 3.

15. The flag `-vvv` causes the program to print the values of the indices moments before they are reached. It's used to check the correctness of the algorithms used to calculated the indices. The Boolean variable used to mark the mode is *confess*.

```
#define CONFESS_FLAG "-vvv"
```

⟨ Internal variables 13 ⟩ +≡

```
static int confess = 0;
```

16. The program doesn't accept both flags, `-v` and `-vvv`, to avoid an output complexity in terms of information and to set a boundary between the two tasks.

⟨ Parse program arguments 12 ⟩ +≡

```
if (argc ≡ 2 ∧ ¬strncmp(argv[1], CONFESS_FLAG, 5))
    confess = 1;
```

17. The user of the program is warned about the flag `-vvv` if the flag was not used.

⟨ Print information about flags 14 ⟩ +≡

```
if (¬confess)
    fprintf(stderr, "- use \"%s -vvv\" to show details about K-index calculation.\n", argv[0]);
```

18. In confess mode, a queue is necessary to not lost previous values of some variable already processed. The queue is implemented using a circular array where the field *front* is the index of the first element and *rear* the index of last element. There's no problem in overwriting some queue elements because only a limited number of values `PREV_NVALS` lesser than the queue length `QLEN` are of interest.

```
#define QLEN 32 /* queue length */
#define PREV_NVALS 5 /* number of elements of interest in the queue */
```

⟨ Internal variables 13 ⟩ +≡

```
static struct queue_struct {
  int array[QLEN];
  int front, rear;
} queue;
```

19. Add the value *idx* in the rear of the queue.

⟨ Static functions 5 ⟩ +≡

```
static void enqueue(int idx)
{
  queue.array[queue.rear++] = idx;
  if (queue.rear ≡ QLEN)
    queue.rear = 0;
}
```

20. The function *queue_is_empty* returns 1 when the queue is empty.

⟨ Static functions 5 ⟩ +≡

```
static int queue_is_empty()
{
  return queue.rear ≤ queue.front;
}
```

21. The function *dequeue_from_rear* removes the element in the rear of the queue returning it. There is no need to remove elements in front of the queue.

⟨ Static functions 5 ⟩ +≡

```
static int dequeue_from_rear()
{
  int idx;
  if (queue.rear ≤ queue.front) {
    panic("Queue is empty");
  }
  idx = queue.array[--queue.rear];
  return idx;
}
```

22. The queue fields *front* and *rear* are initialized using *queue_reset*.

⟨ Static functions 5 ⟩ +≡

```
static void queue_reset()
{
  queue.front = queue.rear = 0;
}
```

23. Input. The data to be processed comes from CSV (comma-separated values) and TSV (tab-separated vaules) files containing, among other data, the papers and its number of citations (CSV) or number of citings (TSV) of researchers. Each file stores data about one researcher. The citing is the number of citations received by a paper that cites the researcher paper in question. The CSV files are used to calculate the h-index and TSV are used to find the K-index. An index file with author's identification and some information like his/her homepage is used to associate the data files. For example, an author with an Researcher ID equals to "Z-1111-1900" has the papers' citations in a file called "Z-1111-1900.csv" and the papers' citings in a file named "Z-1111-1900.tsv".

24. One or more files as input.

```
⟨ Parse program arguments 12 ⟩ +≡
  if (argc ≥ 2) {
    for (i = 0; i < argc; i++) {
      fn = argv[i];
      fprintf(stderr, "␣%s,␣k=%d\n", fn, do_k(fn));
    }
  }
```

25. A directory with files as input. The data files were saved inside in the value of `DATA_DIRECTORY` macro directory.

```
#define DATA_DIRECTORY "data"
```

26. Fetching authors' record. The macro `AUTHORS_DATA_FN` is set with the file name that contains information about researchers (authors). Each line of the file has the name, Web of Science, Google Scholar or Publons research id and a link to a page containing more information about the author's publications. Not all authors have researcher id, when this occurs, we assign a number and link to the Web of Science page. The author's h-index and K-index are assigned to the fields h and K, respectively.

```
#define AUTHORS_DATA_FN "identifiers.dat"
#define MAX_STR_LEN 256
```

⟨ Data structures 26 ⟩ ≡

```
typedef struct author_struct {
  char id[MAX_STR_LEN];
  int h;
  int k;
  char timestamp[MAX_STR_LEN];      /* last modification of record */
} Author;
```

This code is used in section 2.

27. `MAX_LINE_LEN` is the maximum length of each line, the value is very high because some papers have too many authors.

```
#define MAX_LINE_LEN 1 << 16
```

⟨ Local variables 27 ⟩ ≡

```
char buffer[MAX_LINE_LEN];      /* buffer to store strings */
int i = 0, j = 0;      /* general-purpose counters */
```

See also sections 34, 56, 59, 62, and 71.

This code is used in section 3.

28. An array of structures is used to store the *authors*' information. The global variable A is set with the number of authors processed at the time it is read.

⟨ Internal variables 13 ⟩ +≡

```
char *fn;      /* file name */
FILE *fp;      /* file pointer */
char
line [MAX_LINE_LEN] ;      /* store file lines */
char *ptr;
static Array*authors;      /* store authors' info */
```

29. The maximum number of authors is dictated by the macro `MAX_N_AUTHORS` and the array of *authors* is initialized using this value.

```
#define MAX_N_AUTHORS #2000
```

⟨ Initialize the variables that need memory allocation 29 ⟩ ≡

```
authors = Array_new(MAX_N_AUTHORS, sizeof(Author));
```

See also section 48.

This code is used in section 3.

30. Authors basic information was picked from the Web of Science page, more specifically at `https://hcr.clarivate.com/#categories%3Dphysics` that is the page of most cited authors in physics. They are stored in a file named *ids.idx* that is opened to load this information.

```
⟨ Load authors information 30 ⟩ ≡
  fp = Fopen(AUTHORS_DATA_FN, "r");
  while ( fgets ( line , MAX_LINE_LEN, fp ) ≠ Λ ) {
  if ( is_comment ( line ) )
  continue;
  ⟨ Begin to fill authors structure 33 ⟩
  } Fclose(fp);
```

This code is used in section 3.

31. Memory allocated for the array of pointers *authors* is freed.

```
⟨ Free up memory 31 ⟩ ≡
  Array_free(authors);
```

See also section 50.

This code is used in section 3.

32. ⟨ Include headers 32 ⟩ ≡

```
#include <string.h>      /* strtok() */
```

See also section 46.

This code is used in section 2.

33. The fields are separated by semicolon inside *authors.idx*, a record in the file looks like

```
L-000-000;Joe Doe;http//joedoe.joe
```

where the first field `L-000-000` is the Research ID or ORCID, when the author doesn't have an identifier, a custom number is assigned. The second field `Joe Doe` is the author name and the third field is the link to the page that contains information about author's publications. A structure is loaded with these data and a pointer to this structure is passed to the array *authors*. Lately, h-index and K-index will be calculated and assigned to the proper field in the structure.

```
#define IDX_SEP ";\n"
```

⟨ Begin to fill authors structure 33 ⟩ ≡

```
i = 0;      /* information index */
ptr = strtok ( line , IDX_SEP ) ;
while (ptr ≠ Λ) {
  switch (i) {
  case 0: strncpy(aut.id, ptr, MAX_STR_LEN);
    break;
  case 1: aut.h = atoi(ptr);
    if (aut.h ≤ 0) {
      fprintf(stderr, "==> h=%d <==\n", aut.h);
      panic("Wrong value of h-index, run confess mode.");
    }     /* Initialize K too */
    aut.k = 0;
    break;
  case 4: strncpy(aut.timestamp, ptr, MAX_STR_LEN);
    break;
  default: break;
  }
  ptr = strtok(Λ, IDX_SEP);
  i++;
}
Array_append(authors, &aut);
```

This code is used in section 30.

34. *aut* is used to point to new allocated **Author** structure adress while the fields are assigned with the proper values.

⟨ Local variables 27 ⟩ +≡

```
Author aut;      /* temporary variable */
```

35. In all custom files used to parse the data, the hash character "#" is used to indicate that after it the following tokens must be interpreted as comments.

⟨ Static functions 5 ⟩ +≡

```
int is_comment ( char * line ) {
if ( ¬ line ) goto exit_is_comment;
if ( line [0] ≡ '#' ) {
if (verbose) fprintf (stderr, "%s", line ) ;
return 1; }
exit_is_comment: return 0; }
```

36. Fetching Nobel laureates. We have to discard researchers that already was awarded with the Nobel Prize. Up to 2018, there was 935 laureates. We put more chairs in the room to accommodate future laureates. A simple array is used to store the IDs and a linear search is performed. As the number of winners is not high, this simple scheme, even though not so efficient, is used to avoid complexities.

```
#define N_LAUREATES 935
#define MORE_ROOM 128
```

37. The Nobel Laureates identifier are inserted in the *list* array;

```
⟨ Internal variables 13 ⟩ +≡
  static Array*list;
```

38. The *Array* data structure is used to manage sequential allocation of related elements. The data is copied to *array* field, the *cap* field is the maximum number of elements provided by the array, *length* is the number of elements occupied in the array and *size* is the number of bytes occupied by each element. All elements are of the same size.

```
⟨ Type definitions 38 ⟩ ≡
  typedef struct array_struct {
    void *array;
    int cap;      /* capacity of the array in number of elements */
    int length;     /* number of elements used */
    int size;     /* size in bytes of each element of the array */
  } Array;
```

This code is used in section 2.

39. To create an array, memory is allocated for the structure and the data.

```
⟨ Static functions 5 ⟩ +≡
  static Array *Array_new(int capacity, int size)
  {
    Array *ary;
    ary = CALLOC(1, sizeof(Array));
    ary→array = CALLOC(capacity, size);
    ary→cap = capacity;
    ary→size = size;
    ary→length = 0;
    return ary;
  }
```

40. An element is get by accessing the ith element in the array taking into account the size of each element.

```
⟨ Static functions 5 ⟩ +≡
  static void *Array_get(Array *ary, int i)
  {
    assert(ary);
    assert(i ≥ 0 ∧ i < ary→length);
    return ary→array + i * ary→size;
  }
```

41. An element is put in the array by copying the bytes of the element *elem* starting at the proper position i and taking into account the size of each element.

```
⟨ Static functions 5 ⟩ +≡
  static void Array_put(Array *ary, int i, void *elem)
  {
    assert(ary);
    assert(i ≥ 0 ∧ i < ary→cap);
    assert(elem);
    memcpy(ary→array + i * ary→size, elem, ary→size);
  }
```

42.
```
⟨ Static functions 5 ⟩ +≡
  static void Array_append(Array *ary, void *elem)
  {
    assert(ary);
    assert(elem);
    memcpy(ary→array + ary→length * ary→size, elem, ary→size);
    ary→length++;
  }
```

43. Memory of array structure are freed by deallocating the data field *array* and the structure itself.

```
⟨ Static functions 5 ⟩ +≡
  static void Array_free(Array *ary)
  {
    FREE(ary→array);
    FREE(ary);
  }
```

44. *Array_length* returns the number of elements in the *array*.

```
⟨ Static functions 5 ⟩ +≡
  static int Array_length(Array *array)
  {
    assert(array);
    assert(array→length ≥ 0);
    return array→length;
  }
```

45.
```
⟨ Static functions 5 ⟩ +≡
  static int Array_size(Array *array)
  {
    assert(array);
    assert(array→size > 0);
    return array→size;
  }
```

46. The function *assert* is used to check some invariants or integrity constraints of the data.

```
⟨ Include headers 32 ⟩ +≡
#include <assert.h>
```

47. A file `NOBEL_FN` with the identification number (id) of the Nobel Laureates is used to check if the researcher already win the prize.

```
/* file name with ids of Nobel Laureates */
#define NOBEL_FN "laureates_in.dat"
⟨ Load the ids of Nobel Laureates 47 ⟩ ≡
  fp = Fopen(NOBEL_FN, "r"); while ( fgets ( line , MAX_LINE_LEN, fp ) ≠ Λ ) { if ( is_comment ( line ) )
      continue;      /* Remove the new line */
  line [ strcspn ( line , "\r\n" ) ] = 0;
  ⟨ Insert the hash id at the end of the list 49 ⟩
  } Fclose(fp);
```

This code is used in section 3.

48. The *list* array is initialized with enough space to put lareuates.

```
⟨ Initialize the variables that need memory allocation 29 ⟩ +≡
  list = Array_new(N_LAUREATES + MORE_ROOM, MAX_STR_LEN);
```

49. Each new Laureate id is inserted in the array list and the number of elements in the list is incremented. No overflow checking is done.

```
⟨ Insert the hash id at the end of the list 49 ⟩ ≡
  Array_append (list, line ) ;
```

This code is used in section 47.

50.

```
⟨ Free up memory 31 ⟩ +≡
  Array_free(list);
```

51. The function *is_nobel_laureate* check in the list od laureates with IDs if the author *a* id is in the list. The string comparison does not take into account if an id is prefix of another one because this is very unlikely to occur.

```
⟨ Static functions 5 ⟩ +≡
  static int is_nobel_laureate(Author *aut)
  {
    int i;
    char *id = aut→id;
    for (i = 0; i < list→length; i++) {
      if (strncmp(Array_get(list, i), id, MAX_STR_LEN) ≡ 0) return 1;
    }
    return 0;
  }
```

52. Indices calculation. There is procedure in this program to calculate the scientometric index K. The h is the Hirsch index proposed by Hirsch [J. E. Hirsch, "An index to quantify an individual's scientific research output," *PNAS* **102** (15) 16569–16572, 2005]. It is obtained at Web of Science, so no further procedure is needed. The K stands for Kinouchi index and was prososed by O. Kinouchi *et al.* [O. Kinouchi, L. D. H. Soares, G. C. Cardoso, "A simple centrality index for scientific social recognition", *Physica A* **491** (1), 632–640].

53. h-index. The number of papers is in decreasing order of citations that the number of citations is greater than the paper position is the h-index. On Web of Science homepage, the procedure to find the h of an author is as follows:

- Search for an author's publications by *Author* or *Author Identifiers*;
- Click on the link *Create Citation Report*;
- The h-index is showed at the top of the page.

The h-index value is stored in the author record structure and saved in "authors.idx" file.

54. K-index. If an author receives at least K citations, where each one of these K citations have get at least K citations, then the author's K-index was found. On Web of Science homepage, the procedure to find the K of an author looks like below:

- ⋆ Search for an author's publications;
- ⋆ Click on the link *Create Citation Report*;
- ⋆ Click on the link *Citing Articles without self-citations*;
- ⋆ Traverse the list, stoping when the rank position of the article were greater than the *Times Cited*;
- ⋆ Subtract on from the rank position, this is the K value.

To calculate in batch mode, we downloaded a file with the data to calculate the K by clicking on the button *Export...* and selecting *Fast 5K* format that saves the same data, with limit of 5.000 records, where each field is separated by one or more tabs that is assigned to the macro `TSV_SEP`.

55. ⟨ Calculate K index 55 ⟩ ≡

```
N = Array_length(authors);
for (i = 0; i < N; i++) {      /* for each author */
  ⟨ Process tsv file 57 ⟩
}
```

This code is used in section 3.

56. ⟨ Local variables 27 ⟩ +≡

```
int N = 0;
```

57. To open the proper file the Researcher ID is concatenated with `DATA_DIRECTORY` as prefix and the file extension `K_EXT` as suffix.

```
#define K_EXT "tsv"
```

⟨ Process tsv file 57 ⟩ ≡

```
paut = Array_get(authors, i);
snprintf(buffer, MAX_LINE_LEN, "%s/%s.%s", DATA_DIRECTORY, 39paut→id, K_EXT);
fn = buffer;
paut→k = do_k(fn);
```

This code is used in section 55.

58. ⟨ Functions 58 ⟩ ≡

```
int do_k(char *filename){ int pos;      /* temporary variable to store the paper position */
    int ncits, old_ncits;      /* current and old value of number of citings */
    int i = 0, j = 0;
    fp = Fopen(fn, "r");
    pos = 1;
    ncits = 0, old_ncits = 1000000;
    if (confess) {
      fprintf(stderr, "%d. %s\n", i + 1, paut→id);
      queue_reset( );
    }
    while ( fgets ( line , sizeof ( line ) , fp ) ≠ Λ )
    {
      ⟨ Parse the line counting citings 60 ⟩
    }
    Fclose(fp);
    return pos; }
```

This code is used in section 2.

59. ⟨ Local variables 27 ⟩ +≡

60. The file with citings has few lines to ignore, basically it is only one that begins with "PT \t" (ignore double quotes). A line that begins with new line command ignored too, but only for caution.

```
⟨ Parse the line counting citings 60 ⟩ ≡
  if ( strstr ( line , "PT\t" ) ≠ Λ )
  {
    continue;
  }
  else if ( line [0] ≡ '\n' )
  {     /* start with new line */
    continue;
  }
  else {
    ⟨ Find the citings and check if the K-index was found 61 ⟩
  }
```

This code is used in section 58.

61. SKIP represents the fields to be skipped before *Times Cited* value is reached. Its value is not fixed and for this reason it was implemented a tricky way to get the *Times Cited* value: after SKIP is passed, each field is accumulated in a queue and when the end of the record is reached, the queue is dequeue three times to get the *Times Cited* value. This position offset of *Times Cited* value from the end is fixed for all files.

```
#define TSV_SEP "\t"
#define SKIP 7     /* number of fields that can be skipped with safety */
⟨ Find the citings and check if the K-index was found 61 ⟩ ≡
  { ncits = 0;
  j = 0; ptr = strtok ( line , TSV_SEP ) ;
  while (ptr ≠ Λ) {
    if (j > SKIP) {
      push(ptr);
    }
    j++;
    ptr = strtok(Λ, TSV_SEP);
  }
  for (j = 0; j < 3; j++) {
    if (queue_is_empty( )) break;
    else ptr = pop( );
  }
  ncits = atoi(ptr);
  stack_reset( );
  ⟨ Check parsing integrity of citings 63 ⟩
  ⟨ Enqueue temporary index value 64 ⟩
  old_ncits = ncits;
  if (pos > ncits) {     /* found k */
    pos−−;
    ⟨ Write the last values 65 ⟩
    break;
  }
  pos++; }
```

This code is used in section 60.

62. ⟨ Local variables 27 ⟩ +≡
```
char *ptr;      /* Generic pointer */
```

63. The articles are listed in descending order of number of citings. For this reason, the old value of number of citings *old_ncits* must not be lesser than current value just parsed *ncits*. The verification stops the program execution if this invariant is not obeyed.

⟨ Check parsing integrity of citings 63 ⟩ ≡
```
if (old_ncits < ncits) {
  fprintf(stderr, "==>␣%d<%d␣<==\n", old_ncits, ncits);
  panic("Previous␣number␣of␣citings␣is␣lesser␣the␣the␣current␣one.");
}
```
This code is used in section 61.

64. ⟨ Enqueue temporary index value 64 ⟩ ≡
```
if (confess)
  enqueue(ncits);
```
This code is used in section 61.

65. ⟨ Write the last values 65 ⟩ ≡
```
if (confess) {
  register int ii;
  paut = Array_get(authors, i);
  fprintf(stderr, "==>␣found␣K=%d␣<==\n␣<>␣Last␣values\n", paut→k);
  for (ii = 0; ii < PREV_NVALS; ii++) {
    if (queue_is_empty()) break;
    fprintf(stderr, "␣K:␣pos=%d,␣ncits=%d\n", (pos -- + 1), 39dequeue_from_rear());
  }
}
```
This code is used in section 61.

66. Stack. A humble stack is implemented to store few pointers using FIFO policy. The stack is composed by an array of pointers named *data* and an index named *top* to point to the next index to add element in the stack. Three stacks are declared, one for storing temporary values of the fields during *K*-index calculation, other to store temporary values of citation and other to store temporary values of citings.

```
#define STACK_LEN #10000
```

⟨ Internal variables 13 ⟩ +≡

```
static struct {
  char *data[STACK_LEN];
  int top;
} stack;
```

67. Elements are inserted at the top of the stack by invoking *push* and using **char** *∗ptr* as parameter. The index *idx* is incremented to the number of elements in the stack and *top* − 1 is the index of the element in the top.

⟨ Static functions 5 ⟩ +≡

```
static void push(char *ptr)
{
  if (ptr ≡ Λ) {
    panic("Tring␣to␣push␣NULL␣value␣to␣the␣stack");
  }
  stack.data[stack.top++] = ptr;
  if (stack.top ≡ STACK_LEN) {
    panic("Stack␣overflow");
  }
}
```

68. Elements from the top of the stack are removed by *pop* function. If there is no element in the stack, Λ is returned.

⟨ Static functions 5 ⟩ +≡

```
static char *pop( )
{
  if (stack.top ≤ 0) panic("Stack␣underflow");
  else return stack.data[--stack.top];
}
```

69. To reset the stack, *top* is assigned to zero.

⟨ Static functions 5 ⟩ +≡

```
static void stack_reset( )
{
  stack.top = 0;
}
```

70. Sorting. The authors are classified in descending order according to their K-index. The insertion-sort algorithm is used to simplify the code and according to the number of entries is not so large.

```
⟨ Sort the authors 70 ⟩ ≡
  N = Array_length(authors);
  for (i = 1; i < N; i++) {
    memcpy(&aut, Array_get(authors, i), Array_size(authors));
    for (j = i − 1; j ≥ 0; j−−) {
      qaut = (Author *) Array_get(authors, j);
      if (aut.k < qaut→k) break;
      Array_put(authors, j + 1, qaut);
    }
    Array_put(authors, j + 1, &aut);
  }
```

This code is used in section 3.

71. ⟨ Local variables 27 ⟩ +≡

```
  register Author *qaut;
```

72. Output. The results are written as a table in markdown format to the file name assigned to `RANK_FN`. A space is needed between the bars and the content.

```
#define RANK_FN "rank.md"
```

⟨ Write results to a file 72 ⟩ ≡

```
fp = Fopen(RANK_FN, "w");
fprintf(fp, "| N | Author | h | K |\n");
fprintf(fp, "|---|--------|---|---|\n");
N = Array_length(authors);
for (i = 0; i < N; i++) {
  paut = Array_get(authors, i);      /* Mark Nobel Laureates */
  if (is_nobel_laureate(paut)) ptr = "**";
  else ptr = "";
  fprintf(fp, "| %d | %s%s%s | %d | %d |\n", i + 1, ptr, paut->id, ptr, paut->h, paut->k);
}
Fclose(fp);
fprintf(stderr, "* Wrote \"%s\"\n", RANK_FN);
```

This code is used in section 3.

73. ⟨ Internal variables 13 ⟩ +≡

```
static Author *paut;
```

74. A table with the twelve larger Ks to be included in the manuscript is written in LaTeX format.

⟨ Write a table with the twelve larger ks in latex format 74 ⟩ ≡

```
fn = "table.tex";
fp = Fopen(fn, "w");
fprintf(fp, "\\begin{tabular}{cccc} \\\\ \\hline\n");
fprintf(fp, "\\bf N &\\bf Author &\\bg h &\\bf K \\\\ \\hline\n");
for (i = 0; i < 12; i++) {
  paut = Array_get(authors, i);
  fprintf(fp, " %d & %s & %d & %d \\\\\n", i + 1, paut->id, paut->h, paut->k);
}
fprintf(fp, "\\hline\\end{tabular}\n");
Fclose(fp);
fprintf(stderr, "* Wrote \"%s\"\n", fn);
```

This code is used in section 3.

75. Index.

⟨ Begin to fill authors structure 33 ⟩ Used in section 30.
⟨ Calculate K index 55 ⟩ Used in section 3.
⟨ Check parsing integrity of citings 63 ⟩ Used in section 61.
⟨ Data structures 26 ⟩ Used in section 2.
⟨ Enqueue temporary index value 64 ⟩ Used in section 61.
⟨ Find the citings and check if the K-index was found 61 ⟩ Used in section 60.
⟨ Free up memory 31, 50 ⟩ Used in section 3.
⟨ Functions 58 ⟩ Used in section 2.
⟨ Include headers 32, 46 ⟩ Used in section 2.
⟨ Initialize the variables that need memory allocation 29, 48 ⟩ Used in section 3.
⟨ Insert the hash id at the end of the list 49 ⟩ Used in section 47.
⟨ Internal variables 13, 15, 18, 28, 37, 66, 73 ⟩ Used in section 2.
⟨ Load authors information 30 ⟩ Used in section 3.
⟨ Load the ids of Nobel Laureates 47 ⟩ Used in section 3.
⟨ Local variables 27, 34, 56, 59, 62, 71 ⟩ Used in section 3.
⟨ Macro declarations 8, 10, 11 ⟩ Used in section 2.
⟨ Parse program arguments 12, 16, 24 ⟩ Used in section 3.
⟨ Parse the line counting citings 60 ⟩ Used in section 58.
⟨ Print information about flags 14, 17 ⟩ Used in section 3.
⟨ Process tsv file 57 ⟩ Used in section 55.
⟨ Sort the authors 70 ⟩ Used in section 3.
⟨ Static functions 5, 6, 7, 9, 19, 20, 21, 22, 35, 39, 40, 41, 42, 43, 44, 45, 51, 67, 68, 69 ⟩ Used in section 2.
⟨ Type definitions 38 ⟩ Used in section 2.
⟨ Write a table with the twelve larger ks in latex format 74 ⟩ Used in section 3.
⟨ Write results to a file 72 ⟩ Used in section 3.
⟨ Write the last values 65 ⟩ Used in section 61.

K-INDEX