

1. Introduction. K-NOBEL is a project to try to predict the future Laureates of Nobel prize of Physics using K -index to rank the researchers. Another parameter, h -index, is used to evaluate the error threshold, since h -index is used by Web of Science as one of the indices to predict the Laureates of Nobel prize.

The program has the following structure:

```
#include <stdio.h>
#include <stdlib.h>
  < Include files 24 >
  < Macro declarations 5 >
  < Data structures 18 >
  < Internal variables 7 >
  < Static functions 3 >

2. int main(int argc, char **argv)
{
  < Local variables 9 >
  < Parse program arguments 6 >
  < Load the ids of Nobel Laureates 28 >
  < Load authors information 21 >
  < Calculate K index 34 >
  < Sort the authors 46 >
  < Write results to a file 47 >
  < Write a table with the twelve larger ks in latex format 48 >
  < Free up memory 49 >
  < Print information about flags 8 >
  return 0;
}
```

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

```
< Static functions 3 > ≡
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 4, 13, 14, 15, 16, 23, 26, 30, 43, 44, and 45.

This code is used in section 1.

```
4. < Static functions 3 > +≡
static void Fclose(FILE *f)
{
  if (f) fclose(f);
}
```

5. 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 5⟩ ≡

```
#undef panic
#ifdef NDEBUG
#define panic(msg) ((void) 0)
#else
extern void panic(int msg);
#define panic(msg) (fprintf(stderr, "%s:%d: PANIC: %s\n", __FILE__, (int) __LINE__, msg), abort())
#endif
```

This code is used in section 1.

6. 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 6 ⟩ ≡
    if (argc ≡ 2 ∧ ¬strcmp(argv[1], VERBOSE_FLAG, 3))
        verbose = 1;
```

See also section 10.

This code is used in section 2.

7. 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 7 ⟩ ≡
    static int verbose = 0;
```

See also sections 12, 20, 27, and 42.

This code is used in section 1.

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

```
⟨ Print information about flags 8 ⟩ ≡
    if (¬verbose)
        fprintf(stderr, "-use \"%s-v\" to print information about data set.\n", argv[0]);
```

See also section 11.

This code is used in section 2.

9. 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 calculate the indices. The Boolean variable used to mark the mode is *confess*.

```
#define CONFESS_FLAG "-vvv"
⟨ Local variables 9 ⟩ ≡
    int confess;
```

See also sections 19 and 36.

This code is used in section 2.

10. 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 6 ⟩ +≡
    if (argc ≡ 2 ∧ ¬strcmp(argv[1], CONFESS_FLAG, 5))
        confess = 1;
```

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

```
⟨ Print information about flags 8 ⟩ +≡
    if (¬confess)
        fprintf(stderr, "-use \"%s-vvv\" to show details about K-index calculation.\n", argv[0]);
```

12. 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 7> +=
static struct queue_struct {
    int array[QLEN];
    int front, rear;
} queue;
```

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

```
<Static functions 3> +=
static void enqueue(int idx)
{
    queue.array[queue.rear++] = idx;
    if (queue.rear == QLEN)
        queue.rear = 0;
}
```

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

```
<Static functions 3> +=
static int queue_is_empty()
{
    return queue.rear <= queue.front;
}
```

15. 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 3> +=
static int dequeue_from_rear()
{
    int idx;
    if (queue.rear <= queue.front) {
        panic("Queue_is_empty");
    }
    idx = queue.array[--queue.rear];
    return idx;
}
```

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

```
<Static functions 3> +=
static void queue_reset()
{
    queue.front = queue.rear = 0;
}
```

17. Input data. 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". The data files were saved inside in the value of `DATA_DIRECTORY` macro directory.

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

18. 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 "authors.idx"
#define MAX_STR_LEN 256
⟨Data structures 18⟩ ≡
    struct author {
        char name[MAX_STR_LEN];
        char researchid[MAX_STR_LEN];
        char url[MAX_STR_LEN];
        int h;
        int k;
        char timestamp[MAX_STR_LEN];    /* last modification of record */
    };

```

This code is used in section 1.

19. `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 9⟩ +≡
    struct author *aut;    /* temporary variable */
    char *fn, *p;    /* file name and generic pointer */
    FILE *fp;    /* file pointer */
    char buffer[MAX_STR_LEN];    /* buffer to store strings */
    char
    line [MAX_LINE_LEN] ;    /* store file lines */
    int i = 0, j = 0;    /* general-purpose counters */

```

20. 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 7⟩ +≡
    static struct author **authors;    /* store authors' info */
    static int A = 0;    /* number of authors */

```

21. 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 *authors.idx* that is opened to load this information.

```
⟨Load authors information 21⟩ ≡
    fp = Fopen(AUTHORS_DATA_FN, "r");
    while ( fgets ( line , MAX_LINE_LEN, fp ) ≠ Λ ) {
        if ( is_comment ( line ) )
            continue;
        ⟨Reallocate the array of authors structure with to pointer elements 22⟩
        ⟨Begin to fill authors structure 25⟩
    } Fclose(fp);

```

This code is used in section 2.

22. To add a new author to the *authors* array, a reallocation of space to it is needed because the array has the capacity exactly equals to the current number of authors.

⟨Reallocate the array of authors structure with to pointer elements 22⟩ ≡

```
authors = (struct author **) realloc(authors, get_no_authors() * sizeof(struct author *));
```

This code is used in section 21.

23. The number of research authors is calculated by adding one to global variable *A* that is the next free array index.

⟨Static functions 3⟩ +≡

```
static int get_no_authors()  
{  
    return A + 1;  
}
```

24. ⟨Include files 24⟩ ≡

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

This code is used in section 1.

25. 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 25> ≡
aut = (struct author *) malloc(sizeof(struct author));
i = 0; /* information index */
char *p; p = strtok ( line , IDX_SEP ) ;
while (p ≠ Λ) {
    switch (i) {
        case 0: strncpy(aut→researchid,p,MAX_STR_LEN);
                break;
        case 1: strncpy(aut→name,p,MAX_STR_LEN);
                break;
        case 2: strncpy(aut→url,p,MAX_STR_LEN);
                break;
        case 3: aut→h = atoi(p);
                if (aut→h ≤ 0) {
                    fprintf(stderr, "=>h=%d<==\n");
                    panic("Wrong value of h-index, run confess mode.");
                }
                break;
        case 4: strncpy(aut→timestamp,p,MAX_STR_LEN);
                break;
        default: break;
    }
    p = strtok(Λ, IDX_SEP);
    i++;
}
if (¬is_nobel_laureate(aut)) {
    authors[A++] = aut;
}
```

This code is used in section 21.

26. 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 3> +≡
int is_comment ( char * line ) { if ( ¬ line ) goto exit_is_comment; if ( line [0] ≡ '#' ) { if (verbose)
    printf ("%s", line ) ;
    return 1; }
exit_is_comment: return 0; }
```


27. Fetching Nobel Laureates. We have to discard researchers that already was laureated with the Nobel Prize. Up to 2018, there was 935 laureates that awarded Nobel Prize. We put more chairs in the room to accomodate future laureated researchers. 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
<Internal variables 7> +=
static struct arr {
    char array[N_LAUREATES + MORE_ROOM][MAX_STR_LEN];
    int n; /* number of elements used */
} list;
```

28. 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.dat"
<Load the ids of Nobel Laureates 28> =
fp = Fopen(NOBEL_FN, "r"); while ( fgets ( line , MAX_LINE_LEN, fp ) != \Lambda ) { if ( is_comment ( line ) )
    continue; /* Remove the new line */
line [ strcspn ( line , "\r\n" ) ] = 0;
<Insert research id in the list 29>
} Fclose(fp);
```

This code is used in section 2.

29. 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 research id in the list 29> =
strncpy (list.array[list.n++], line , sizeof ( line ) );
```

This code is used in section 28.

30. The function *is_nobel_laureate* check in the laureated list 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 3> +=
static int is_nobel_laureate(struct author *a)
{
    int i;
    char *id = a->researchid;
    for (i = 0; i < list.n; i++) {
        if (strncmp(list.array[i], id, sizeof (id)) == 0) return 1;
    }
    return 0;
}
```

31. 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 proposed 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].

32. 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.

33. *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, stopping 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`.

34. `< Calculate K index 34 > ≡`
`for (i = 0; i < A; i++) { /* for each author */`
`< Process tsv file 35 >`
`}`

This code is used in section 2.

35. 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 35 > ≡
    strncpy(buffer, DATA_DIRECTORY, sizeof (DATA_DIRECTORY));
    strncat(buffer, authors[i]-researchid, sizeof (authors[i]-researchid));
    strncat(buffer, K_EXT, sizeof (K_EXT));
    fn = buffer;
    fp = Fopen(fn, "r");
    pos = 1;
    ncits = 0, old_ncits = 1000000;
    if (confess) {
        fprintf(stderr, "%d. □ %s\n", i + 1, authors[i]-name);
        queue_reset();
    }
    while ( fgets ( line , sizeof ( line ) , fp ) ≠ Λ )
    {
        < Parse the line counting citings 37 >
    }
    Fclose(fp);
```

This code is used in section 34.

36. `< Local variables 9 > +≡`
`int pos; /* temporary variable to store the paper position */`
`int ncits, old_ncits; /* current and old value of number of citings */`

37. 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 37 ⟩ ≡
  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 38 ⟩
  }

```

This code is used in section 35.

38. K_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 K_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 K_SKIP 7 /* number of fields that can be skipped with safety */
⟨ Find the citings and check if the K-index was found 38 ⟩ ≡
  { int ncits = 0;
    j = 0; p = strtok ( line , TSV_SEP );
    while ( p ≠ Λ ) {
      if ( j > K_SKIP ) {
        push(p);
      }
      j++;
      p = strtok(Λ, TSV_SEP);
    }
    for ( j = 0; j < 3; j++)
      p = pop();
    ncits = atoi(p);
    stack_reset();
    ⟨ Check parsing integrity of citings 39 ⟩
    ⟨ Enqueue temporary index value 40 ⟩
    old_ncits = ncits;
    if ( pos > ncits ) { /* found k */
      pos--;
      authors[i]-k = pos;
      ⟨ Write the last values 41 ⟩
      break;
    }
    pos++; }

```

This code is used in section 37.

39. 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 39 ⟩ ≡

```

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 38.

40. ⟨ Enqueue temporary index value 40 ⟩ ≡

```

if (confess)
    enqueue(ncits);

```

This code is used in section 38.

41. ⟨ Write the last values 41 ⟩ ≡

```

if (confess) {
    register int ii;
    fprintf(stderr, "=>_found_K=%d<==\n_<>_Last_values\n", authors[i]-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 38.

42. 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 7> +=
```

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

43. Elements are inserted at the top of the stack by invoking *enstack* and using **char** **p* 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 3> +=
```

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

44. 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 3> +=
```

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

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

```
<Static functions 3> +=
```

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

46. 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 46 ⟩ ≡
  for ( $i = 1$ ;  $i < A$ ;  $i++$ ) {
     $aut = authors[i]$ ;
    for ( $j = i - 1$ ;  $j \geq 0 \wedge aut \rightarrow k > authors[j] \rightarrow k$ ;  $j--$ ) {
       $authors[j + 1] = authors[j]$ ;
    }
     $authors[j + 1] = aut$ ;
  }

```

This code is used in section 2.

47. 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 "k-nobel.md"
```

⟨ Write results to a file 47 ⟩ ≡

```
fp = fopen(RANK_FN, "w");
if (!fp) {
    perror(fn);
    exit(-4);
}
fprintf(fp, "|_N_|_Author_|_h_|_K_|\\n");
fprintf(fp, "|---|-----|---|---|\\n");
for (i = 0; i < A; i++) {
    fprintf(fp, "|_%d_|_[%s] (%s)|_%d_|_%d_|\\n", i + 1, authors[i]-name, authors[i]-url, authors[i]-h,
            authors[i]-k);
}
fclose(fp);
fprintf(stderr, "*_Wrote_\\\"%s\\\"\\n", RANK_FN);
```

This code is used in section 2.

48. A table with the twelve larger K s to be included in the manuscript is written in LaTeX format.

⟨ Write a table with the twelve larger ks in latex format 48 ⟩ ≡

```
fn = "table.tex";
fp = fopen(fn, "w");
if (!fp) {
    perror(fn);
    exit(-8);
}
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++) {
    fprintf(fp, "\\_d_&\\_s_&\\_d_&\\_d_\\_\\_\\_\\_\\n", i + 1, authors[i]-name, authors[i]-h, authors[i]-k);
}
fprintf(fp, "\\hline\\end{tabular}\\n");
fclose(fp);
fprintf(stderr, "*_Wrote_\\\"%s\\\"\\n", fn);
```

This code is used in section 2.

49. Memory allocated for the array of pointers *authors* is freed. As the memory deallocation is the last task to be executed, a simple usage notification is appended before the task.

⟨ Free up memory 49 ⟩ ≡

```
for (i = 0; i < A; i++) free(authors[i]);
free(authors);
```

This code is used in section 2.

50. Index.

__FILE__: [5](#).
 __LINE__: [5](#).
 A: [20](#).
 a: [30](#).
 abort: [5](#).
 argc: [2](#), [6](#), [10](#).
 argv: [2](#), [6](#), [8](#), [10](#), [11](#).
 arr: [27](#).
 array: [12](#), [13](#), [15](#), [27](#), [29](#), [30](#).
 atoi: [25](#), [38](#).
 aut: [19](#), [25](#), [46](#).
 author: [18](#), [19](#), [20](#), [22](#), [25](#), [30](#).
 authors: [20](#), [21](#), [22](#), [25](#), [35](#), [38](#), [41](#), [46](#), [47](#), [48](#), [49](#).
 AUTHORS_DATA_FN: [18](#), [21](#).
 buffer: [19](#), [35](#).
 Cardoso, George Cunha: [31](#).
 confess: [9](#), [10](#), [11](#), [35](#), [40](#), [41](#).
 CONFESS_FLAG: [9](#), [10](#).
 data: [42](#), [43](#), [44](#).
 DATA_DIRECTORY: [17](#), [35](#).
 dequeue_from_rear: [15](#), [41](#).
 enqueue: [13](#), [40](#).
 enstack: [43](#).
 exit: [3](#), [47](#), [48](#).
 exit_is_comment: [26](#).
 f: [3](#), [4](#).
 fclose: [4](#), [47](#), [48](#).
 Fclose: [4](#), [21](#), [28](#), [35](#).
 fgets: [21](#), [28](#), [35](#).
 filename: [3](#).
 fn: [19](#), [35](#), [47](#), [48](#).
 Fopen: [3](#), [21](#), [28](#), [35](#).
 fopen: [3](#), [47](#), [48](#).
 fp: [19](#), [21](#), [28](#), [35](#), [47](#), [48](#).
 fprintf: [3](#), [5](#), [8](#), [11](#), [25](#), [35](#), [39](#), [41](#), [47](#), [48](#).
 free: [49](#).
 front: [12](#), [14](#), [15](#), [16](#).
 get_no_authors: [22](#), [23](#).
 h: [18](#).
 Hirsch, Jorge Eduardo: [31](#).
 i: [19](#), [30](#).
 id: [30](#).
 idx: [13](#), [15](#), [21](#), [25](#), [43](#).
 IDX_SEP: [25](#).
 ii: [41](#).
 is_comment: [21](#), [26](#), [28](#).
 is_nobel_laureate: [25](#), [30](#).
 j: [19](#).
 k: [18](#).
 K_EXT: [35](#).
 K_SKIP: [38](#).
 Kinouch, Osame: [31](#).
 list: [27](#), [29](#), [30](#).
 main: [2](#).
 malloc: [25](#).
 MAX_LINE_LEN: [19](#), [21](#), [28](#).
 MAX_STR_LEN: [18](#), [19](#), [25](#), [27](#).
 mode: [3](#).
 MORE_ROOM: [27](#).
 msg: [5](#).
 n: [27](#).
 N_LAUREATES: [27](#).
 name: [18](#), [25](#), [35](#), [47](#), [48](#).
 ncits: [35](#), [36](#), [38](#), [39](#), [40](#).
 NDEBUG: [5](#).
 NOBEL_FN: [28](#).
 old_ncits: [35](#), [36](#), [38](#), [39](#).
 p: [19](#), [25](#), [43](#).
 panic: [5](#), [15](#), [25](#), [39](#), [43](#), [44](#).
 perror: [47](#), [48](#).
 pop: [38](#), [44](#).
 pos: [35](#), [36](#), [38](#), [41](#).
 PREV_NVALS: [12](#), [41](#).
 printf: [26](#).
 push: [38](#), [43](#).
 QLEN: [12](#), [13](#).
 queue: [12](#), [13](#), [14](#), [15](#), [16](#).
 queue_is_empty: [14](#), [41](#).
 queue_reset: [16](#), [35](#).
 queue_struct: [12](#).
 RANK_FN: [47](#).
 realloc: [22](#).
 rear: [12](#), [13](#), [14](#), [15](#), [16](#).
 researchid: [18](#), [25](#), [30](#), [35](#).
 Soares, Leonardo D. H.: [31](#).
 stack: [42](#), [43](#), [44](#), [45](#).
 STACK_LEN: [42](#), [43](#).
 stack_reset: [38](#), [45](#).
 stderr: [3](#), [5](#), [8](#), [11](#), [25](#), [35](#), [39](#), [41](#), [47](#), [48](#).
 strcspn: [28](#).
 strncat: [35](#).
 strncmp: [6](#), [10](#), [30](#).
 strncpy: [25](#), [29](#), [35](#).
 strstr: [37](#).
 strtok: [25](#), [38](#).
 timestamp: [18](#), [25](#).
 top: [42](#), [43](#), [44](#), [45](#).
 TSV_SEP: [33](#), [38](#).
 url: [18](#), [25](#), [47](#).
 verbose: [6](#), [7](#), [8](#), [26](#).
 VERBOSE_FLAG: [6](#).

⟨Begin to fill authors structure 25⟩ Used in section 21.
⟨Calculate K index 34⟩ Used in section 2.
⟨Check parsing integrity of citings 39⟩ Used in section 38.
⟨Data structures 18⟩ Used in section 1.
⟨Enqueue temporary index value 40⟩ Used in section 38.
⟨Find the citings and check if the K-index was found 38⟩ Used in section 37.
⟨Free up memory 49⟩ Used in section 2.
⟨Include files 24⟩ Used in section 1.
⟨Insert research id in the list 29⟩ Used in section 28.
⟨Internal variables 7, 12, 20, 27, 42⟩ Used in section 1.
⟨Load authors information 21⟩ Used in section 2.
⟨Load the ids of Nobel Laureates 28⟩ Used in section 2.
⟨Local variables 9, 19, 36⟩ Used in section 2.
⟨Macro declarations 5⟩ Used in section 1.
⟨Parse program arguments 6, 10⟩ Used in section 2.
⟨Parse the line counting citings 37⟩ Used in section 35.
⟨Print information about flags 8, 11⟩ Used in section 2.
⟨Process tsv file 35⟩ Used in section 34.
⟨Reallocate the array of authors structure with to pointer elements 22⟩ Used in section 21.
⟨Sort the authors 46⟩ Used in section 2.
⟨Static functions 3, 4, 13, 14, 15, 16, 23, 26, 30, 43, 44, 45⟩ Used in section 1.
⟨Write a table with the twelve larger ks in latex format 48⟩ Used in section 2.
⟨Write results to a file 47⟩ Used in section 2.
⟨Write the last values 41⟩ Used in section 38.

K-NOBEL

	Section	Page
Introduction	1	1
Verbose mode	6	3
Input data	17	5
Fetching authors' record	18	6
Fetching Nobel Laureates	27	9
Indices calculation	31	10
<i>h</i> -index	32	11
<i>K</i> -index	33	12
Stack	42	15
Sorting	46	16
Output	47	17
Index	50	18