Introduction:

First you should know that I wrote the functions for a distributed system simulator. The number of CPUs in the system is dynamically changeable. Since your requirement was to implement everything using only Linked Lists, I didn't use array for the list of CPUs in the system. Also I provided you the whole system because you didn't specify the structure of the system and the implementation of the system. So to make you know that the functions are working correctly, I developed a simple distributed system simulator regarding this project. Without the design/structure of the system what data the functions will use? So I used data structures for the system.

Design:

The design of the system is simple. A distributed system has a list of CPUs [nodes]. Each CPU has its own job queue. In ***dsfuncs.h*** header file I declared types for them as follows:

```
/* linked list types */
typedef void element_t; /* an element of any type (void *) */

struct node;
typedef struct node node_t;
typedef node_t *position_t;
typedef node_t *list_t;

/* a single node */
struct node {
  element_t e; /* each node has an element */
  list_t next; /* each node has a pointer to its next node */
};

/* queue data structure */
struct queue_struct {
  position_t front; /* front pointer */
  position_t rear; /* tail/rear pointer */
};
typedef struct queue_struct queue_t;


/* encapsulated type for job */
struct job_struct {
  int jid; /* job identifier */
  int secid5; /* job section [as specified in the problem doc] */
  int secid2; /*                "                             */
  int secid3; /*                "                             */
  int secid6; /*                "                             */
};
typedef struct job_struct job_t;
```

Now we Node, List, Queue, and Job data types. We can declare our distributed system.

First the building blocks are the CPUs. So declaring type for CPU:

```
/* encapsulated type for cpu */
struct cpu_struct {
  int cid; /* cpu id [0 indexed] */
  queue_t jobs; /* list of jobs on this cpu */
  int njobs; /* how many jobs the CPU has */
};
typedef struct cpu_struct cpu_t;
```

As you can see, each CPU has unique CPU ID. A queue of jobs. And a count of how many jobs currently in the job queue. I am using this counter so that we don't have to count the length of the queue every time a job is inserted or deleted. Just incrementing/decrementing this value is enough after inserting the job to the queue.

Then the declaration for our system. It's very simple, just a list of CPUs:

```
struct system_struct {
  list_t cpus; /* list of the system's  CPUs  */
};
typedef struct system_struct system_t;
```

I maintained another structure 'struct cpuinfo'. get_cid() returns list of CPUs and their lengths. Since the list of CPUs is a Linked List I used this data type. It contains linear array of the CPUs and their lengths. So you can iterate through the array to get the specific CPU and its length. If number of CPUs for the system would be fixed [array], then it wouldn't be needed. But your requirement was to use only Linked Lists. Your requirement was that get_cid() should return a list of CPUs and their lengths. So 'struct cpuinfo' takes care of that. It has following structure:

```
/* data type to be returned by get_cid() */
struct cpuinfo {
  cpu_t **cpu; /* the CPU pointer */
  int *cid; /* list of CPUs */
  int *len; /* lengths of the CPU list */
  int n; /* length of each of the above the lists */
};
typedef struct cpuinfo cpuinfo_t;
```

The 'cpu' member contains linear array of CPUs available to the system.
The 'cid' member contains linear array of CPU Ids available to the system.
The 'len' member contains linear array of length of queue for CPUs.

The 'n' member contains length of each array. It's equal to the number of CPUs in the system. So after 'get_cid()' returns, we can simply iterate through the linear arrays for getting values.

List and Queue data structures are abstract and applicable to any type ofdata. Some linked list function calls 'element_compare' function to compare two elements in the list. You can write any custom function for that. Since our current project only uses List of CPUs, the element_compare compares the CPU Ids.

So now you know the data structures used in the project. When you need to apply the functions to some other code, you need to take care of the internal members of the structures.

In 'least_loaded_cpu()' after getting list of CPUs by calling 'get_cid()' the minimum of the length of queue is calculated. It's straightforward, simple code. I commented lines that needs commenting. Here's the code fragment to calculate the minimum length of the lengths of queue of CPUs:

```
/* get list of CPUs and their lengths */
cinfo = get_cid(sys);

min_cid = cinfo->cid[0];
min_len = cinfo->len[0];
n_cpu = 0;
for (i = 0; i < cinfo->n; i++) {
  if (cinfo->len[i] < min_len) { /* less than min_len found */
    min_cid = cinfo->cid[i];
    min_len = cinfo->len[i];
    n_cpu = i;
  }
}
```

The 'increment_cpu()' and 'least_loaded_cpu()' calculates minimum length of queue same way. In the case of 'decrement_cpu()' the maximum length of queue is needed because which CPU we should decrement is the heavily loaded CPU.

The 'qlength' function is straightforward. It just sarts from 'front' pointer and iterates to the end of the queue. Length counter is incremented on each iteration. And returned when iteration to the whole queue is done [NULL found].

<u>Conclusion:</u>

I wish the above discussion will help you understanding the code clearly. When you'll know the structure of the system I developed for the project, you can add more functionality to this. 'dsfuncs.c' contains the whole distributed system simulator code.