```c
// fifo
void fifo(unsigned int page){
    // make a struct of the current page
    S_page *current = malloc(sizeof(S_page));
    current->page = page;

    // search if the page is in memory
    S_page *found = queue_find(memory, inner_equalitor, current);

    if(found == NULL){   // page was not found in memory

        // if memory is not full, just add (page fault)
        if(queue_size(memory) < frames_){
            queue_enqueue(memory, current);
        }
        // if memory is full, remove based on (fifo) (page fault)
        else{
            queue_dequeue(memory);
            queue_enqueue(memory, current);
        }
        sim_get_page(page); // page fault occurs
    }
    else{ // if page is found, free the current sturct
        free(current);
    }
}
```

```c
// for searching
static bool inner_equalitor(void  *a, void *b) {
    return ((S_page*)a)->page == ((S_page*)b)->page;
  }
// for sorting based on usage count
static int last_used_comparator(void *a, void *b) {
    return ((S_page*)a)->last_used - ((S_page*)b)->last_used;
}
// for sorting based on usage count (lfu)
static int usage_count_comparator(void *a, void *b) {
    return ((S_page*)a)->usage_count - ((S_page*)b)->usage_count;
}
// algorithms
void fifo(unsigned int page);
void lru(unsigned int page);
void lfu(unsigned int page);
void sc(unsigned int page);
void cq(unsigned int page);
////////////////////////////////////////////////////////////////////

void pager_init(enum algorithm algorithm, unsigned int frames) {

    memory= queue_create();
    algorithm_ = algorithm;
    frames_= frames;

    spage_arr= calloc(frames_,sizeof(S_page)); // array for the (cq) algorithm
    spage_arr_size = 0; // array size for the (cq) algorithm
    cq_pointer = 0;

    // initialize all the pages to -1;
    for(int i = 0; i<frames_;i++){
        spage_arr[i].page = -1;
        spage_arr[i].reference_bit = 0;
    }
}
```

```
void pager_destroy() {
    int size= queue_size(memory);
    for(int i = 0; i< size; i++){
        S_page *current;
        current = queue_dequeue(memory);
        free(current);
    }
    //queue_destroy(memory);
    for(int i = 0; i< frames_; i++){
        free(&spage_arr[i]);
    }
    free(spage_arr);
}
/**
 * A request has been made for virtual page PAGE. If your pager does
 * not currently have PAGE in physical memory, request it from the
 * simulator.
 */
void pager_request(unsigned int page) {

    switch(algorithm_){
        case FIRST_IN_FIRST_OUT:
            fifo(page);
            break;
        case LEAST_RECENTLY_USED:
            lru(page);
            break;
        case LEAST_FREQUENTLY_USED:
            lfu(page);
            break;
        case SECOND_CHANCE:
            sc(page);
            break;
        case CIRCULAR_QUEUE:
            cq(page);
            break;
    }
}
```

```c
/*
Abdel Rahman Alnajjar
*/
#include <stdlib.h>
#include <stdbool.h>
#include "queue.h"

#include "simulator.h"

/**
 * Initialise your ALGORITHM pager with FRAMES frames (physical pages).
 */

static enum algorithm algorithm_;
static unsigned int frames_;

static void *memory;


////////////////////////////////////////////////////////////////
typedef struct _S_page {
    int page;
    int last_used; // for lru
    int usage_count; // for lfu
    int reference_bit; // for sc and cq

}S_page;

// pointer for the next page to be removed (cq) algorithm
int cq_pointer;
// array for the pages for the (cq) algorithm
S_page* spage_arr;
// keep track of the size of array for the (cq) algorithm
int spage_arr_size;


```

```
// cq
void cq(unsigned int page){
    // search if the page is in memory
    int found=0;
    int tr=1;
    for(int i = 0; i<frames_;i++){
        if(spage_arr[i].page == page){
            found =1;
            spage_arr[i].reference_bit=1;
            break;
        }
    }

    // page was not found in memory
    if(found == 0){
            while(tr==1){
                if((cq_pointer) == frames_){
                    cq_pointer=0;
                }
                if(spage_arr[cq_pointer].reference_bit == 0){
                    spage_arr[cq_pointer].page = page; // just change the page number
                    spage_arr[cq_pointer].reference_bit=1;
                    cq_pointer+=1;
                    tr=0;
                }
                else{
                spage_arr[cq_pointer].reference_bit = 0;
                cq_pointer+=1;
                }
            }

        sim_get_page(page); // page fault occurs
    }
}
```

```c
// sc
void sc(unsigned int page){
    // make a struct of the current page
    S_page *current = malloc(sizeof(S_page));
    current->page = page;
    current->reference_bit = 0; //initially each page has a chance bit of 0
    // search if the page is in memory
    S_page *found = queue_find(memory, inner_equalitor, current);
    if(found == NULL){   // page was not found in memory
        // if memory is not full, just add (page fault)
        if(queue_size(memory) < frames_){
            queue_enqueue(memory, current);
        }
        // if memory is full, remove based on (sc) (page fault)
        else{

            for(int i=0 ; i<frames_;i++){
                S_page *temp = queue_head(memory);
                if(temp->reference_bit == 1){
                                                // give another chance
                    temp->reference_bit=0;
                    queue_dequeue(memory);
                    queue_enqueue(memory, temp);
                }
                else{
                    break;
                }
            }
            queue_dequeue(memory);
            queue_enqueue(memory, current);
        }
        sim_get_page(page); // page fault occurs
    }
    else{ // if page is found, free the current sturct
        // page is refrenced again so make sure its chance bit is 1
        found->reference_bit=1;
        free(current);
    }
}

```

```c
// lfu
void lfu(unsigned int page){
    // make a struct of the current page
    S_page *current = malloc(sizeof(S_page));
    current->page = page;
    //current->usage_count = 0;

    // search if the page is in memory
    S_page *found = queue_find(memory, inner_equalitor, current);

    if(found == NULL){   // page was not found in memory
        current->usage_count=0; //update page usage count
        // if memory is not full, just add (page fault)
        if(queue_size(memory) < frames_){
            queue_enqueue(memory, current);
            queue_sort(memory, usage_count_comparator);
        }
        // if memory is full, remove based on (lfu) (page fault)
        else{
            //queue_sort(memory, last_used_comparator);

            queue_dequeue(memory);
            queue_enqueue(memory, current);
            queue_sort(memory, usage_count_comparator);
        }
        current->last_used=sim_time(); // update when page was last_used

        sim_get_page(page); // page fault occurs
    }
    else{
        found->last_used=sim_time();
        found->usage_count+=1;
        queue_sort(memory, usage_count_comparator);
        free(current);
    }

}
```

```c
// lru
void lru(unsigned int page){

    // make a struct of the current page
    S_page *current = malloc(sizeof(S_page));
    current->page = page;


    // search if the page is in memory
    S_page *found = queue_find(memory, inner_equalitor, current);

    if(found == NULL){   // page was not found in memory

        // if memory is not full, just add (page fault)
        if(queue_size(memory) < frames_){
            queue_enqueue(memory, current);
        }
        // if memory is full, remove based on (lfu) (page fault)
        else{
            queue_sort(memory, last_used_comparator);
            queue_dequeue(memory);
            queue_enqueue(memory, current);
        }
        current->last_used=sim_time(); // update when page was last_used
        sim_get_page(page); // page fault occurs
    }
    else{
        found->last_used=sim_time();
        free(current);
    }

}
```