

```

127 // fifo
128 void fifo(unsigned int page){
129     // make a struct of the current page
130     S_page *current = malloc(sizeof(S_page));
131     current->page = page;
132
133     // search if the page is in memory
134     S_page *found = queue_find(memory, inner_equalitor, current);
135
136     if(found == NULL){ // page was not found in memory
137
138         // if memory is not full, just add (page fault)
139         if(queue_size(memory) < frames_){
140             queue_enqueue(memory, current);
141         }
142         // if memory is full, remove based on (fifo) (page fault)
143         else{
144             queue_dequeue(memory);
145             queue_enqueue(memory, current);
146         }
147         sim_get_page(page); // page fault occurs
148     }
149     else{ // if page is found, free the current struct
150         free(current);
151     }
152 }

```

```

44 // for searching
45 static bool inner_equalitor(void *a, void *b) {
46     return ((S_page*)a)->page == ((S_page*)b)->page;
47 }
48 // for sorting based on usage count
49 static int last_used_comparator(void *a, void *b) {
50     return ((S_page*)a)->last_used - ((S_page*)b)->last_used;
51 }
52 // for sorting based on usage count (lfu)
53 static int usage_count_comparator(void *a, void *b) {
54     return ((S_page*)a)->usage_count - ((S_page*)b)->usage_count;
55 }
56 // algorithms
57 void fifo(unsigned int page);
58 void lru(unsigned int page);
59 void lfu(unsigned int page);
60 void sc(unsigned int page);
61 void cq(unsigned int page);
62 ///////////////////////////////////////////////////////////////////
63
64 void pager_init(enum algorithm algorithm, unsigned int frames) {
65
66     memory= queue_create();
67     algorithm_ = algorithm;
68     frames_= frames;
69
70     spage_arr= calloc(frames_,sizeof(S_page)); // array for the (cq) algorithm
71     spage_arr_size = 0; // array size for the (cq) algorithm
72     cq_pointer = 0;
73
74     // initialize all the pages to -1;
75     for(int i = 0; i<frames_;i++){
76         spage_arr[i].page = -1;
77         spage_arr[i].reference_bit = 0;
78     }
79 }

```

```

84 void pager_destroy() {
85     int size= queue_size(memory);
86     for(int i = 0; i< size; i++){
87         S_page *current;
88         current = queue_dequeue(memory);
89         free(current);
90     }
91     //queue_destroy(memory);
92     for(int i = 0; i< frames_; i++){
93         free(&spage_arr[i]);
94     }
95     free(spage_arr);
96 }
97 /**
98  * A request has been made for virtual page PAGE. If your pager does
99  * not currently have PAGE in physical memory, request it from the
100  * simulator.
101  */
102 void pager_request(unsigned int page) {
103
104     switch(algorithm_){
105         case FIRST_IN_FIRST_OUT:
106             fifo(page);
107             break;
108         case LEAST_RECENTLY_USED:
109             lru(page);
110             break;
111         case LEAST_FREQUENTLY_USED:
112             lfu(page);
113             break;
114         case SECOND_CHANCE:
115             sc(page);
116             break;
117         case CIRCULAR_QUEUE:
118             cq(page);
119             break;
120     }
121 }

```

```

1  /*
2  Abdel Rahman Alnajjar
3  */
4  #include <stdlib.h>
5  #include <stdbool.h>
6  #include "queue.h"
7
8  #include "simulator.h"
9
10 /**
11  * Initialise your ALGORITHM pager with FRAMES frames (physical pages).
12  */
13
14 static enum algorithm algorithm_;
15 static unsigned int frames_;
16
17 static void *memory;
18
19
20 ///////////////////////////////////////////////////////////////////
21 typedef struct _S_page {
22     int page;
23     int last_used; // for lru
24     int usage_count; // for lfu
25     int reference_bit; // for sc and cq
26 }S_page;
27
28 // pointer for the next page to be removed (cq) algorithm
29 int cq_pointer;
30 // array for the pages for the (cq) algorithm
31 S_page* spage_arr;
32 // keep track of the size of array for the (cq) algorithm
33 int spage_arr_size;
34
35
36

```



```

277 // cq
278 void cq(unsigned int page){
279     // search if the page is in memory
280     int found=0;
281     int tr=1;
282     for(int i = 0; i<frames_;i++){
283         if(spage_arr[i].page == page){
284             found =1;
285             spage_arr[i].reference_bit=1;
286             break;
287         }
288     }
289
290     // page was not found in memory
291     if(found == 0){
292         while(tr==1){
293             if((cq_pointer) == frames_){
294                 cq_pointer=0;
295             }
296             if(spage_arr[cq_pointer].reference_bit == 0){
297                 spage_arr[cq_pointer].page = page; // just change the page number
298                 spage_arr[cq_pointer].reference_bit=1;
299                 cq_pointer+=1;
300                 tr=0;
301             }
302             else{
303                 spage_arr[cq_pointer].reference_bit = 0;
304                 cq_pointer+=1;
305             }
306         }
307
308         sim_get_page(page); // page fault occurs
309     }
310 }
311

```

```

235 // sc
236 void sc(unsigned int page){
237     // make a struct of the current page
238     S_page *current = malloc(sizeof(S_page));
239     current->page = page;
240     current->reference_bit = 0; //initially each page has a chance bit of 0
241     // search if the page is in memory
242     S_page *found = queue_find(memory, inner_equalitor, current);
243     if(found == NULL){ // page was not found in memory
244         // if memory is not full, just add (page fault)
245         if(queue_size(memory) < frames_){
246             queue_enqueue(memory, current);
247         }
248         // if memory is full, remove based on (sc) (page fault)
249         else{
250
251             for(int i=0 ; i<frames_;i++){
252                 S_page *temp = queue_head(memory);
253                 if(temp->reference_bit == 1){
254                     // give another chance
255                     temp->reference_bit=0;
256                     queue_dequeue(memory);
257                     queue_enqueue(memory, temp);
258                 }
259                 else{
260                     break;
261                 }
262             }
263             queue_dequeue(memory);
264             queue_enqueue(memory, current);
265         }
266         sim_get_page(page); // page fault occurs
267     }
268     else{ // if page is found, free the current struct
269         // page is referenced again so make sure its chance bit is 1
270         found->reference_bit=1;
271         free(current);
272     }
273 }
274

```

```

195 // lfu
196 void lfu(unsigned int page){
197     // make a struct of the current page
198     S_page *current = malloc(sizeof(S_page));
199     current->page = page;
200     //current->usage_count = 0;
201
202     // search if the page is in memory
203     S_page *found = queue_find(memory, inner_equalitor, current);
204
205     if(found == NULL){ // page was not found in memory
206         current->usage_count=0; //update page usage count
207         // if memory is not full, just add (page fault)
208         if(queue_size(memory) < frames){
209             queue_enqueue(memory, current);
210             queue_sort(memory, usage_count_comparator);
211         }
212         // if memory is full, remove based on (lfu) (page fault)
213         else{
214             //queue_sort(memory, last_used_comparator);
215
216             queue_dequeue(memory);
217             queue_enqueue(memory, current);
218             queue_sort(memory, usage_count_comparator);
219         }
220         current->last_used=sim_time(); // update when page was last_used
221
222         sim_get_page(page); // page fault occurs
223     }
224     else{
225         found->last_used=sim_time();
226         found->usage_count+=1;
227         queue_sort(memory, usage_count_comparator);
228         free(current);
229     }
230
231 }

```

```

159 // lru
160 void lru(unsigned int page){
161
162     // make a struct of the current page
163     S_page *current = malloc(sizeof(S_page));
164     current->page = page;
165
166
167     // search if the page is in memory
168     S_page *found = queue_find(memory, inner_equalitor, current);
169
170     if(found == NULL){ // page was not found in memory
171
172         // if memory is not full, just add (page fault)
173         if(queue_size(memory) < frames_){
174             queue_enqueue(memory, current);
175         }
176         // if memory is full, remove based on (lfu) (page fault)
177         else{
178             queue_sort(memory, last_used_comparator);
179             queue_dequeue(memory);
180             queue_enqueue(memory, current);
181         }
182         current->last_used=sim_time(); // update when page was last_used
183         sim_get_page(page); // page fault occurs
184     }
185     else{
186         found->last_used=sim_time();
187         free(current);
188     }
189
190 }

```