МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Сучасні операційні системи

Лабораторна робота 1

На тему: “Алокатор пам’яті загального застосування”

*Виконав студент групи ІC-72*

*Цилюрик Дмитро*

Київ

НТУУ «КПІ»

2020

**Тема роботи**: Аллокатор пам’яті загального призначення.

**Мета роботи**: розробити аллокатор загального призначення.

**Опис розробленого алгоритму**:

1. В якості об’єкту для збереження інформації використовується масив:
2. **int** \*mas = **new** **int**[n+1];

При цьому, кожен блок виділеної пам’яті містить заголовок:

1. **struct** BlockHeader{
2. **bool** state;
3. **size\_t** size;
4. **size\_t** prevsize;
5. };

Під час ініціалізації виділяється 1 вільний блок, який займає всю доступну пам’ять масиву.

1. Виділення пам’яті заданого розміру mem\_alloc(size\_t size):

Вибирається перший знайдений підходящий блок, тобто такий, розмір якого не менший за size. Якщо розмір більший необхідного, блок розбивається на зайнятий і вільний блоки, і користувачу повертається вказівник на початок зайнятого блоку. Якщо потрібний блок не знайдений, то повертається NULL.

1. Перевиділення пам’яті mem\_realloc(void \*addr, size\_t size).

Якщо addr = NULL, то виконується виклик mem\_alloc(size). Інакше, відбувається перевірка сусідніх блоків. Якщо хоча б один з них вільний, відбувається об’єднання блоків. Далі перевіряється, чи цей новий блок достатнього розміру для перевизначення. Якщо так, то в ньому створюється 2 блоки: зайнятий і вільний. У зайнятий копіюються дані з колишнього блоку користувача. Якщо ж ні – йде пошук нового вільного блоку і, якщо він знайдений, виділяється блок для користувача і копіюються данні. Інакше – повертається NULL, а данні користувача не змінюютсья.

1. Звільнення пам’яті mem\_free(void \*addr). Помічаємо блок як вільний і об’єднуємо даний блок с сусідніми вільними блоками (максимум – 2).

**Оцінка часу пошуку вільного блоку пам’яті**:

O(N), де N – кількість блоків у пам’яті.

**Оцінка часу звільнення занятого блоку**:

O(1).

**Оцінка витрати пам'яті для зберігання службої інформації:**

12 \* N байт, де N – кількість блоків у пам’яті.

**Переваги аллокатора:**

1. Невеликий об’єм службової інформації.
2. Малий час виконання звільнення блоку.
3. Об’єднання сусідніх вільних блоків в один.

**Недоліки аллокатора:**

1. Великий час пошуку нового блоку( лінійна складність)
2. При виділенні блоку можливе надмірне виділення пам’яті у розмірі, меншому за розмір заголовку.

**Посилання на GitHub проекту:**

https://github.com/lost-dt/os-course/tree/master/Task1

**Розроблене програмне забезпечення:**

* Файл конфігурації *CLion*:

1. cmake\_minimum\_required(VERSION 3.3)
2. project(hello)
4. set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11")
6. set(SOURCE\_FILES main.cpp Allocator.cpp Allocator.h)
7. add\_executable(os-task-1 ${SOURCE\_FILES})

* Файл *main.cpp*:

1. #include <iostream>
2. #include "Allocator.h"
4. **using** **namespace** std;
6. **void** fillBlock(**void** \*start, **int** size, **int** filler) {
7. **for** (**int** i = 0; i < size; i++) {
8. \*((**int** \*) start + 1) = filler;
9. }
10. }
12. **int** main() {
13. cout << "start!\n";
14. **const** **int** n = 2000;
15. **const** **int** calls = 15;
16. **const** **int** bSize = 50;
17. Allocator al = Allocator(n);
18. **void** \*curBlock;
19. **void** \*mas[calls];
20. **for** (auto &ma : mas) {
21. curBlock = al.mem\_alloc(bSize);
22. ma = curBlock;
23. fillBlock(curBlock, 50, 170);
24. }
25. cout << "alloc 15 blocks (length = 50) \n";
26. al.mem\_dump();
27. cout << "min 3th el to 20\n"; // good
28. al.mem\_realloc(mas[3], 20);
29. al.mem\_dump();
30. cout << "max 3th el to 30\n"; // good
31. al.mem\_realloc(mas[3], 30);
32. al.mem\_dump();
33. cout << "max 3th el to 49\n";
34. al.mem\_realloc(mas[3], 49);
35. al.mem\_dump();
36. cout << "free 2th and 4th els \n";
37. al.mem\_free(mas[2]);
38. al.mem\_free(mas[4]);
39. al.mem\_dump();
40. cout << "...and max 3th el to 156 \n";
41. al.mem\_realloc(mas[3], 156);
42. al.mem\_dump();
43. cout << "min 2th el to 53 \n";
44. al.mem\_realloc(mas[2], 53);
45. al.mem\_dump();
46. cout << "finished! \n";
47. **return** 0;
48. }

* Файл *Allocator.h*:

1. #pragma once
2. #ifndef HELLO\_ALLOCATOR\_H
3. #define HELLO\_ALLOCATOR\_H
5. **struct** BlockHeader {
6. **bool** state;
7. **size\_t** size;
8. **size\_t** prevsize;
9. };

12. **class** Allocator {
13. **public**:
14. **explicit** Allocator(**int** n);
15. **void** \*mem\_alloc(**size\_t** size);
16. **void** \*mem\_realloc(**void** \*addr, **size\_t** size);
17. **void** mem\_free(**void** \*addr);
18. **void** mem\_dump();
20. **private**:
21. **size\_t** bSize;
22. BlockHeader \*begin;
23. **int** \*endOfMemory;
25. **bool** checkDamage(**int** filler);
26. BlockHeader \*nextBlockHeader(BlockHeader \*current);
27. BlockHeader \*previousBlockHeader(BlockHeader \*current);
28. **bool** isLast(BlockHeader \*h);
29. **void** mergeWithNext(BlockHeader \*current, BlockHeader \*next);
30. **void** mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current);
31. **void** mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next);
32. **void** mergeBoth(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next);
33. **void** copyData(**void** \*from, **void** \*to, **size\_t** quantity);
34. **void** \*searchNewBlock(**void** \*addr, **size\_t** size);
35. **void** \*expandLeft(**void** \*addr, **size\_t** size);
36. **void** \*expandRight(**void** \*addr, **size\_t** size);
37. **void** \*expandBoth(**void** \*addr, **size\_t** size);
38. **static** **void** initBlockHeader(BlockHeader \*bh, **bool** state, **size\_t** previous, **size\_t** size, **int** mask);
39. **void** \*separateOnUseAndFree(BlockHeader \*current, **size\_t** size);
40. **static** **void** \*getBlock(BlockHeader \*h);
41. };

44. #endif //HELLO\_ALLOCATOR\_H

* Файл *Allocator.cpp*:

1. #include <iostream>
2. #include "Allocator.h"
4. **using** **namespace** std;
6. Allocator::Allocator(**const** **int** n) {
7. **int** \*mas = **new** **int**[n + 1];
8. bSize = **sizeof**(BlockHeader) / **sizeof**(**int**);
9. begin = (BlockHeader \*) (&mas[0]);
11. begin->prevsize = 0;
12. begin->size = &mas[n] - &mas[0] - bSize;
13. begin->state = **false**;
15. endOfMemory = &mas[n];
16. }
18. **void** \*Allocator::mem\_alloc(**size\_t** size) {
19. BlockHeader \*current = begin;
20. **while** (current->size < size || current->state) {
21. **if** (isLast(current))
22. **return** nullptr;
23. current = nextBlockHeader(current);
24. }
25. **if** ((current->size == size) || ((current->size - size) < bSize)) {
26. current->state = **true**;
27. **return** getBlock(current);
28. } **else** {
29. **return** separateOnUseAndFree(current, size);
30. }
31. }
33. **void** \*Allocator::mem\_realloc(**void** \*addr, **size\_t** size) {
34. **if** (addr == nullptr) {
35. **return** mem\_alloc(size);
36. }
38. BlockHeader \*current = (BlockHeader \*) addr - 1;
39. **size\_t** deltaSize = current->size - size; // size\_t - unsigned int!!! fix it!
41. **if** (deltaSize == 0)
42. **return** addr;
44. BlockHeader \*previous = previousBlockHeader(current);
45. BlockHeader \*next = nextBlockHeader(current);
47. **if** (previous == nullptr)
48. **if** (next == nullptr) {
49. //prev=NULL, next=NULL
50. **return** searchNewBlock(addr, size);
51. } **else** **if** (next->state)
52. //prev=NULL, next=1
53. **return** searchNewBlock(addr, size);
54. **else**
55. //prev=NULL, next=0
56. **return** expandRight(addr, size); //check later - fixed
57. **else** **if** (previous->state)
58. **if** (next == nullptr)
59. //prev=1, next=NULL
60. **return** searchNewBlock(addr, size);
61. **else** **if** (next->state)
62. //prev=1, next=1
63. **return** searchNewBlock(addr, size);
64. **else**
65. //prev=1, next=0
66. **return** expandRight(addr, size);
67. **else** **if** (next == nullptr)
68. //prev=0, next=NULL
69. **return** expandLeft(addr, size);
70. **else** **if** (next->state)
71. //prev=0, next=1
72. **return** expandLeft(addr, size);
73. **else**
74. //prev=0, next=0
75. **return** expandBoth(addr, size);
76. }
78. **void** Allocator::mem\_free(**void** \*addr) {
79. BlockHeader \*current = (BlockHeader \*) addr - 1;
80. BlockHeader \*next = nextBlockHeader(current);
81. BlockHeader \*previous = previousBlockHeader(current);

84. **if** (previous == nullptr)
85. **if** (next == nullptr)
86. //prev=NULL, next=NULL
87. current->state = **false**;
88. **else** **if** (next->state)
89. //prev=NULL, next=1
90. current->state = **false**;
91. **else**
92. //prev=NULL, next=0
93. mergeWithNext(current, next);
94. **else** **if** (previous->state)
95. **if** (next == nullptr)
96. //prev=1, next=NULL
97. current->state = **false**;
98. **else** **if** (next->state)
99. //prev=1, next=1
100. current->state = **false**;
101. **else**
102. //prev=1, next=0
103. mergeWithNext(current, next);
104. **else** **if** (next == nullptr)
105. //prev=0, next=NULL
106. mergeWithPrevious(previous, current);
107. **else** **if** (next->state)
108. //prev=0, next=1
109. mergeWithPrevious(previous, current, next);
110. **else**
111. //prev=0, next=0
112. mergeBoth(previous, current, next);
113. }
115. BlockHeader \*Allocator::nextBlockHeader(BlockHeader \*current) {
116. **if** (isLast(current))
117. **return** nullptr;
119. **return** (BlockHeader \*) ((**int** \*) ((**char** \*) current + **sizeof**(BlockHeader)) + current->size);
120. }
122. BlockHeader \*Allocator::previousBlockHeader(BlockHeader \*current) {
123. **if** (current->prevsize == NULL)
124. **return** nullptr;
125. **return** (BlockHeader \*) ((**int** \*) ((**char** \*) current - **sizeof**(BlockHeader)) - current->prevsize);
126. }
128. **bool** Allocator::isLast(BlockHeader \*h) {
129. **return** ((**int** \*) ((**char** \*) h + **sizeof**(BlockHeader)) + h->size) == endOfMemory;
130. }
132. **void** Allocator::mergeWithNext(BlockHeader \*current, BlockHeader \*next) {
133. current->size += next->size + **sizeof**(BlockHeader) / **sizeof**(**int**);
134. current->state = **false**;
135. BlockHeader \*next2 = nextBlockHeader(next);
136. **if** (next2 != nullptr)
137. next2->prevsize = current->size;
138. }
140. **void** Allocator::mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current) {
141. previous->size += current->size + **sizeof**(BlockHeader) / **sizeof**(**int**);
142. }
144. **void** Allocator::mergeWithPrevious(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next) {
145. previous->size += current->size + **sizeof**(BlockHeader) / **sizeof**(**int**);
146. next->prevsize = previous->size;
147. }
149. **void** Allocator::mergeBoth(BlockHeader \*previous, BlockHeader \*current, BlockHeader \*next) {
150. previous->size += current->size + next->size + 2 \* **sizeof**(BlockHeader) / **sizeof**(**int**);
151. BlockHeader \*next2 = nextBlockHeader(next);
152. **if** (next2 != nullptr)
153. next2->prevsize = previous->size;
154. }
156. **void** Allocator::copyData(**void** \*from, **void** \*to, **size\_t** quantity) {
157. **int** \*f = (**int** \*) from;
158. **int** \*t = (**int** \*) to;
159. **for** (unsigned **int** i = 0; i < quantity; i++)
160. t[i] = f[i];
161. }
163. **void** \*Allocator::searchNewBlock(**void** \*addr, **size\_t** size) {
165. BlockHeader \*current = (BlockHeader \*) addr - 1;
166. **size\_t** deltaSize = current->size - size;
168. **if** ((deltaSize > 0) && (deltaSize < 3)) {
169. **return** addr;
170. }
172. **if** (deltaSize > 0) {
173. **return** separateOnUseAndFree((BlockHeader \*) addr - 1, size);
174. } **else** {
175. **void** \*nBlock = mem\_alloc(size);
176. **if** (nBlock != nullptr) {
177. copyData(addr, nBlock, current->size);
178. mem\_free(addr);
179. **return** nBlock;
180. }
181. **return** nullptr;
182. }
183. }

186. **void** \*Allocator::expandLeft(**void** \*addr, **size\_t** size) {
188. BlockHeader \*current = (BlockHeader \*) addr - 1;
189. BlockHeader \*previous = previousBlockHeader(current);
190. **size\_t** area = current->size + current->prevsize + **sizeof**(BlockHeader) / **sizeof**(**int**);
191. **size\_t** deltaSize = current->size - size;
193. **if** (deltaSize > 0) { //minimize
194. //copy
195. **int** \*first = (**int** \*) addr;
196. **for** (**int** i = size - 1; i >= 0; i--) {
197. first[i + deltaSize] = first[i];
198. }
199. //set BlockHeaders
200. previous->size += deltaSize;
202. current = nextBlockHeader(previous);
203. initBlockHeader(current, **true**, previous->size, size, 7);
205. BlockHeader \*next = nextBlockHeader(current);
206. **if** (next != nullptr)
207. next->prevsize = current->size;
209. **return** getBlock(current);
210. } **else** { //maximize
211. **if** (area >= size)
212. **if** ((area == size) || ((area - size) < 3)) {
213. //set BH
214. previous->size = area;
215. BlockHeader \*next = nextBlockHeader(current);
216. **if** (next != nullptr)
217. next->prevsize = previous->size;
218. previous->state = **true**;
219. //copy
220. copyData(addr, getBlock(previous), current->size);
221. **return** getBlock(previous);
222. } **else** {
223. previous->size = size;
224. previous->state = **true**;
225. copyData(addr, getBlock(previous), current->size);
227. current = nextBlockHeader(previous);
228. current->size = area - size - **sizeof**(BlockHeader) / **sizeof**(**int**);
229. current->state = **false**;
230. current->prevsize = previous->size;
231. BlockHeader \*next = nextBlockHeader(current);
232. **if** (next != nullptr)
233. next->prevsize = current->size;
234. **return** getBlock(previous);
235. }
236. **else** {
237. **void** \*p = mem\_alloc(size);
238. **if** (p != nullptr) {
239. copyData(addr, p, current->size);
240. BlockHeader \*next = nextBlockHeader(current);
241. **if** (next == nullptr)
242. mergeWithPrevious(previous, current);
243. **else**
244. mergeWithPrevious(previous, current, next);
245. **return** p;
246. }
247. }
248. }
249. **return** nullptr;
250. }
252. **void** \*Allocator::expandRight(**void** \*addr, **size\_t** size) {
253. BlockHeader \*current = (BlockHeader \*) addr - 1;
254. BlockHeader \*next = nextBlockHeader(current);
255. **size\_t** area = current->size + next->size + **sizeof**(BlockHeader) / **sizeof**(**int**);
256. **int** deltaSize = (**int**) (current->size) - (**int**) (size); //TODO: fis expression to int - fixed
258. **if** (deltaSize > 0) { //minimize
259. //set BlockHeaders
260. current->size = size;
261. next = nextBlockHeader(current);
262. initBlockHeader(next, **false**, current->size, area - size - **sizeof**(BlockHeader) / **sizeof**(**int**), 7);
263. BlockHeader \*next2 = nextBlockHeader(next);
264. **if** (next2 != nullptr)
265. next2->prevsize = next->size;
266. **return** getBlock(current);
267. } **else** { //maximize
268. **if** (area >= size)
269. **if** ((area - size) < 3) {
270. //set BH
271. current->size = area;
272. next = nextBlockHeader(current);
273. **if** (next != nullptr)
274. next->prevsize = current->size;
275. **return** getBlock(current);
276. } **else** {
277. current->size = size;
279. next = nextBlockHeader(current);
280. next->size = area - size - **sizeof**(BlockHeader) / **sizeof**(**int**);
281. next->prevsize = size;
282. next->state = **false**;
283. BlockHeader \*next2 = nextBlockHeader(next);
284. **if** (next2 != nullptr)
285. next2->prevsize = next->size;
286. **return** getBlock(current);
287. }
288. **else** {
289. **void** \*p = mem\_alloc(size);
290. **if** (p != nullptr) {
291. copyData(addr, p, current->size);
292. next = nextBlockHeader(current);
293. mergeWithNext(current, next);
294. **return** p;
295. }
296. }
297. }
298. **return** nullptr;
299. }
301. **void** \*Allocator::expandBoth(**void** \*addr, **size\_t** size) {
303. BlockHeader \*current = (BlockHeader \*) addr - 1;
304. BlockHeader \*previous = previousBlockHeader(current);
305. BlockHeader \*next = nextBlockHeader(current);
306. **size\_t** area = current->size + previous->size + next->size + 2 \* **sizeof**(BlockHeader) / **sizeof**(**int**);
307. **int** deltaSize = (**int**) (current->size) - (**int**) (size); //fix unsigned - fixed
309. **if** (deltaSize > 0) { //minimize
310. mergeBoth(previous, current, next);
311. //copy
312. copyData(addr, getBlock(previous), size);
313. **return** separateOnUseAndFree(previous, size);
314. } **else** { //maximize
315. **if** (area >= size)
316. **if** ((area == size) || ((area - size) < 3)) {
317. //set BH
318. previous->size = area;
319. BlockHeader \*next2 = nextBlockHeader(next);
320. **if** (next2 != nullptr)
321. next2->prevsize = previous->size;
322. previous->state = **true**;
323. //copy
324. copyData(addr, getBlock(previous), current->size);
325. **return** getBlock(previous);
326. } **else** {
327. mergeBoth(previous, current, next);
328. //copy
329. copyData(addr, getBlock(previous), current->size);
330. **return** separateOnUseAndFree(previous, size);
331. }
332. **else** {
333. **void** \*p = mem\_alloc(size);
334. **if** (p != nullptr) {
335. copyData(addr, p, current->size);
336. mergeBoth(previous, current, next);
337. **return** p;
338. }
339. }
340. }
341. **return** nullptr;
342. }
344. **void** Allocator::initBlockHeader(BlockHeader \*bh, **bool** state, **size\_t** previous, **size\_t** size, **int** mask) {
345. **if** (bh == nullptr)
346. **return**;
347. **if** (mask & 4) {
348. bh->state = state;
349. }
350. **if** (mask & 2) {
351. bh->prevsize = previous;
352. }
353. **if** (mask & 1)
354. bh->size = size;
355. }
357. **void** \*Allocator::separateOnUseAndFree(BlockHeader \*current, **size\_t** size) {
358. **size\_t** curSize = current->size;
359. BlockHeader \*next = nextBlockHeader(current);
360. **if** (next == nullptr) {
361. initBlockHeader(current, **true**, 0, size, 5);
362. BlockHeader \*next1 = nextBlockHeader(current);
363. initBlockHeader(next1, **false**, size, (curSize - size - bSize), 7);
364. **return** getBlock(current);
365. } **else** {
366. initBlockHeader(current, **true**, 0, size, 5);
367. BlockHeader \*next1 = nextBlockHeader(current);
368. initBlockHeader(next1, **false**, size, (curSize - size - bSize), 7);
369. next->prevsize = next1->size;
370. **return** getBlock(current);
371. }
372. }
374. **void** \*Allocator::getBlock(BlockHeader \*h) {
375. **return** (**void** \*) (h + 1);
376. }
378. **void** Allocator::mem\_dump() {
379. BlockHeader \*current = begin;
380. **int** i = 0;
381. cout << "--- Out all BlockHeaders:" << endl;
382. **while** (current != nullptr) {
383. cout << i << ". " << current << "   " << current->state << "   " << current->size << "   " << current->prevsize
384. << endl;
385. i++;
386. current = nextBlockHeader(current);
387. }
388. cout << "--- --- --- - --- --- ---" << endl;
389. }

* Файл *результату виконання*:

1. start!
2. alloc 15 blocks (length = 50)
3. --- Out all BlockHeaders:
4. 0. 0x7fe387001600   1   50   0
5. 1. 0x7fe3870016e0   1   50   50
6. 2. 0x7fe3870017c0   1   50   50
7. 3. 0x7fe3870018a0   1   50   50
8. 4. 0x7fe387001980   1   50   50
9. 5. 0x7fe387001a60   1   50   50
10. 6. 0x7fe387001b40   1   50   50
11. 7. 0x7fe387001c20   1   50   50
12. 8. 0x7fe387001d00   1   50   50
13. 9. 0x7fe387001de0   1   50   50
14. 10. 0x7fe387001ec0   1   50   50
15. 11. 0x7fe387001fa0   1   50   50
16. 12. 0x7fe387002080   1   50   50
17. 13. 0x7fe387002160   1   50   50
18. 14. 0x7fe387002240   1   50   50
19. 15. 0x7fe387002320   0   1154   50
20. --- --- --- - --- --- ---
21. min 3th el to 20
22. --- Out all BlockHeaders:
23. 0. 0x7fe387001600   1   50   0
24. 1. 0x7fe3870016e0   1   50   50
25. 2. 0x7fe3870017c0   1   50   50
26. 3. 0x7fe3870018a0   1   20   50
27. 4. 0x7fe387001908   0   24   20
28. 5. 0x7fe387001980   1   50   24
29. 6. 0x7fe387001a60   1   50   50
30. 7. 0x7fe387001b40   1   50   50
31. 8. 0x7fe387001c20   1   50   50
32. 9. 0x7fe387001d00   1   50   50
33. 10. 0x7fe387001de0   1   50   50
34. 11. 0x7fe387001ec0   1   50   50
35. 12. 0x7fe387001fa0   1   50   50
36. 13. 0x7fe387002080   1   50   50
37. 14. 0x7fe387002160   1   50   50
38. 15. 0x7fe387002240   1   50   50
39. 16. 0x7fe387002320   0   1154   50
40. --- --- --- - --- --- ---
41. max 3th el to 30
42. --- Out all BlockHeaders:
43. 0. 0x7fe387001600   1   50   0
44. 1. 0x7fe3870016e0   1   50   50
45. 2. 0x7fe3870017c0   1   50   50
46. 3. 0x7fe3870018a0   1   30   50
47. 4. 0x7fe387001930   0   14   30
48. 5. 0x7fe387001980   1   50   14
49. 6. 0x7fe387001a60   1   50   50
50. 7. 0x7fe387001b40   1   50   50
51. 8. 0x7fe387001c20   1   50   50
52. 9. 0x7fe387001d00   1   50   50
53. 10. 0x7fe387001de0   1   50   50
54. 11. 0x7fe387001ec0   1   50   50
55. 12. 0x7fe387001fa0   1   50   50
56. 13. 0x7fe387002080   1   50   50
57. 14. 0x7fe387002160   1   50   50
58. 15. 0x7fe387002240   1   50   50
59. 16. 0x7fe387002320   0   1154   50
60. --- --- --- - --- --- ---
61. max 3th el to 49
62. --- Out all BlockHeaders:
63. 0. 0x7fe387001600   1   50   0
64. 1. 0x7fe3870016e0   1   50   50
65. 2. 0x7fe3870017c0   1   50   50
66. 3. 0x7fe3870018a0   1   50   50
67. 4. 0x7fe387001980   1   50   50
68. 5. 0x7fe387001a60   1   50   50
69. 6. 0x7fe387001b40   1   50   50
70. 7. 0x7fe387001c20   1   50   50
71. 8. 0x7fe387001d00   1   50   50
72. 9. 0x7fe387001de0   1   50   50
73. 10. 0x7fe387001ec0   1   50   50
74. 11. 0x7fe387001fa0   1   50   50
75. 12. 0x7fe387002080   1   50   50
76. 13. 0x7fe387002160   1   50   50
77. 14. 0x7fe387002240   1   50   50
78. 15. 0x7fe387002320   0   1154   50
79. --- --- --- - --- --- ---
80. free 2th and 4th els
81. --- Out all BlockHeaders:
82. 0. 0x7fe387001600   1   50   0
83. 1. 0x7fe3870016e0   1   50   50
84. 2. 0x7fe3870017c0   0   50   50
85. 3. 0x7fe3870018a0   1   50   50
86. 4. 0x7fe387001980   0   50   50
87. 5. 0x7fe387001a60   1   50   50
88. 6. 0x7fe387001b40   1   50   50
89. 7. 0x7fe387001c20   1   50   50
90. 8. 0x7fe387001d00   1   50   50
91. 9. 0x7fe387001de0   1   50   50
92. 10. 0x7fe387001ec0   1   50   50
93. 11. 0x7fe387001fa0   1   50   50
94. 12. 0x7fe387002080   1   50   50
95. 13. 0x7fe387002160   1   50   50
96. 14. 0x7fe387002240   1   50   50
97. 15. 0x7fe387002320   0   1154   50
98. --- --- --- - --- --- ---
99. ...and max 3th el to 156
100. --- Out all BlockHeaders:
101. 0. 0x7fe387001600   1   50   0
102. 1. 0x7fe3870016e0   1   50   50
103. 2. 0x7fe3870017c0   1   156   50
104. 3. 0x7fe387001a48   0   0   156
105. 4. 0x7fe387001a60   1   50   0
106. 5. 0x7fe387001b40   1   50   50
107. 6. 0x7fe387001c20   1   50   50
108. 7. 0x7fe387001d00   1   50   50
109. 8. 0x7fe387001de0   1   50   50
110. 9. 0x7fe387001ec0   1   50   50
111. 10. 0x7fe387001fa0   1   50   50
112. 11. 0x7fe387002080   1   50   50
113. 12. 0x7fe387002160   1   50   50
114. 13. 0x7fe387002240   1   50   50
115. 14. 0x7fe387002320   0   1154   50
116. --- --- --- - --- --- ---
117. min 2th el to 53
118. --- Out all BlockHeaders:
119. 0. 0x7fe387001600   1   50   0
120. 1. 0x7fe3870016e0   1   50   50
121. 2. 0x7fe3870017c0   1   53   50
122. 3. 0x7fe3870018ac   0   103   53
123. 4. 0x7fe387001a60   1   50   103
124. 5. 0x7fe387001b40   1   50   50
125. 6. 0x7fe387001c20   1   50   50
126. 7. 0x7fe387001d00   1   50   50
127. 8. 0x7fe387001de0   1   50   50
128. 9. 0x7fe387001ec0   1   50   50
129. 10. 0x7fe387001fa0   1   50   50
130. 11. 0x7fe387002080   1   50   50
131. 12. 0x7fe387002160   1   50   50
132. 13. 0x7fe387002240   1   50   50
133. 14. 0x7fe387002320   0   1154   50
134. --- --- --- - --- --- ---
135. finished!

**Скриншоти виконання застосунку:**

**A picture containing computer

Description automatically generated**

**A picture containing computer

Description automatically generated**

**A picture containing electronics, circuit, computer

Description automatically generated**

**A screenshot of a cell phone

Description automatically generated**