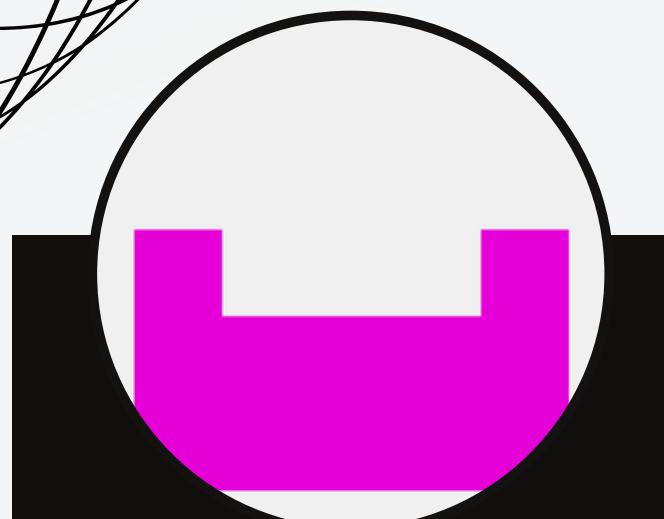


CACHE LRU-K PROJECT

<https://github.com/UsoltsevI/Cache>

OUR TEAM



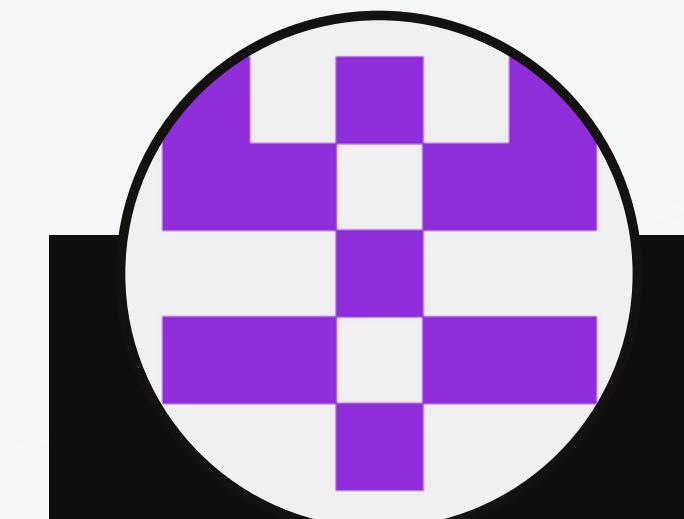
Титов
Артём
Tester,
Developer

Hash,
Comparing
Program



Усольцев
Иван
Team leader,
Technical leader

Project
Structure,
Cache



Деканин
Андрей
Tester

Testing Program,
Test Generator



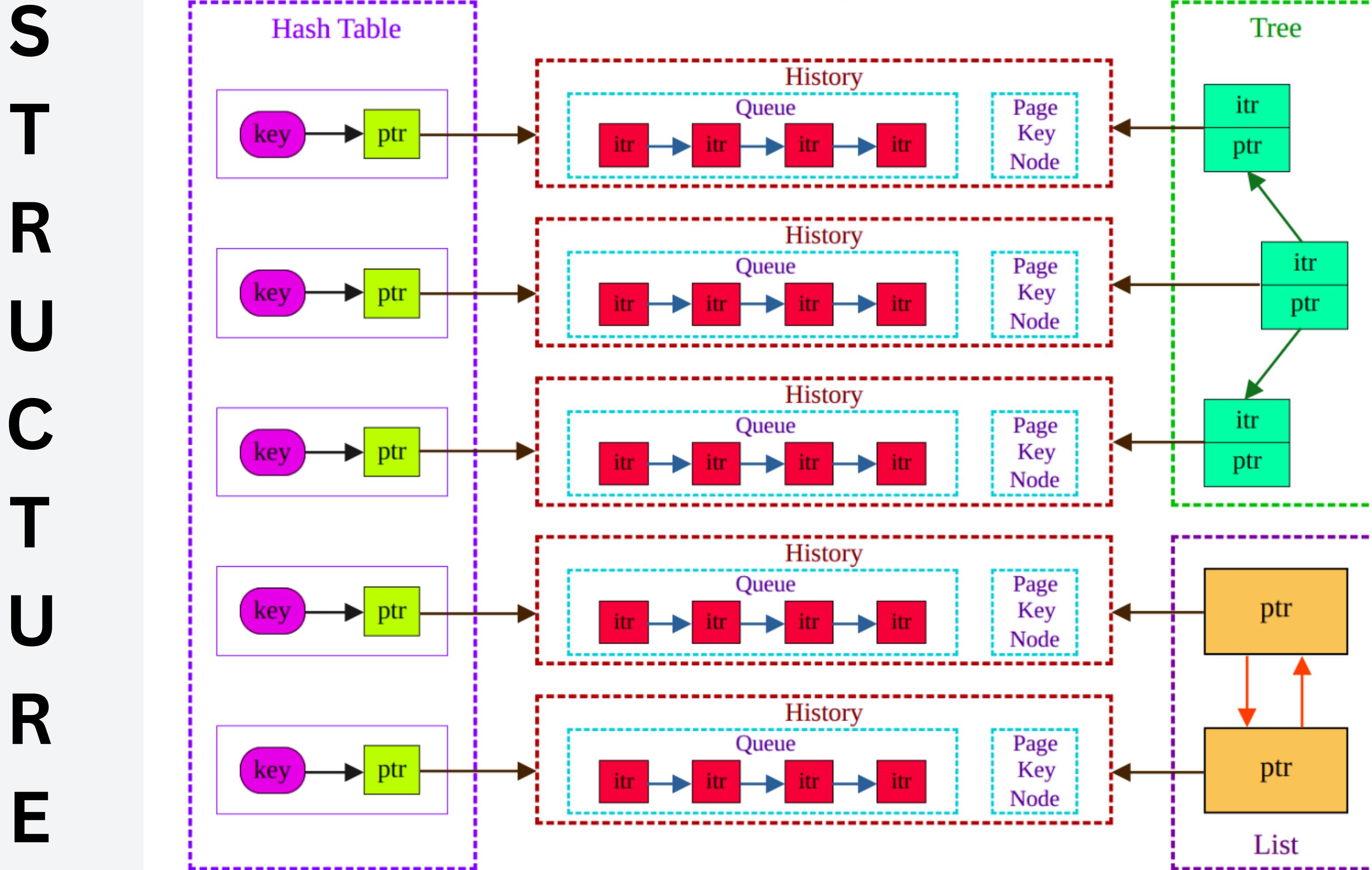
Редькин
Денис
Developer

List, Queue,
Tree

ALGORITHM

- В cache хранится заранее заданное пользователем количество страниц M .
- Для каждой из этих страниц хранится история из K последних записей.
- В случае промаха мы отбрасываем ту страницу, K -ое обращение к которой было самым давним.
- Если в истории страницы хранится менее K элементов, то мы считаем, что K -ое обращение было сделано бесконечное количество вызовов назад.
- Если несколько элементов имеют равное минус бесконечности K -ое значение, то в таком случае удаление идёт по принципу обычного LRU.

LRU-K implementation



MODULES



Обеспечивает поиск страницы с наиболее ранним К-ым обращением, совершённым не бесконечность назад, за $O(\log(M))$.

BINARY TREE



Хранит страницы, К-ое обращение к которым находится на минус бесконечности, в порядке LRU.

LIST



Обеспечивает быстрый поиск страницы по ключу среди набора хранимых страниц.

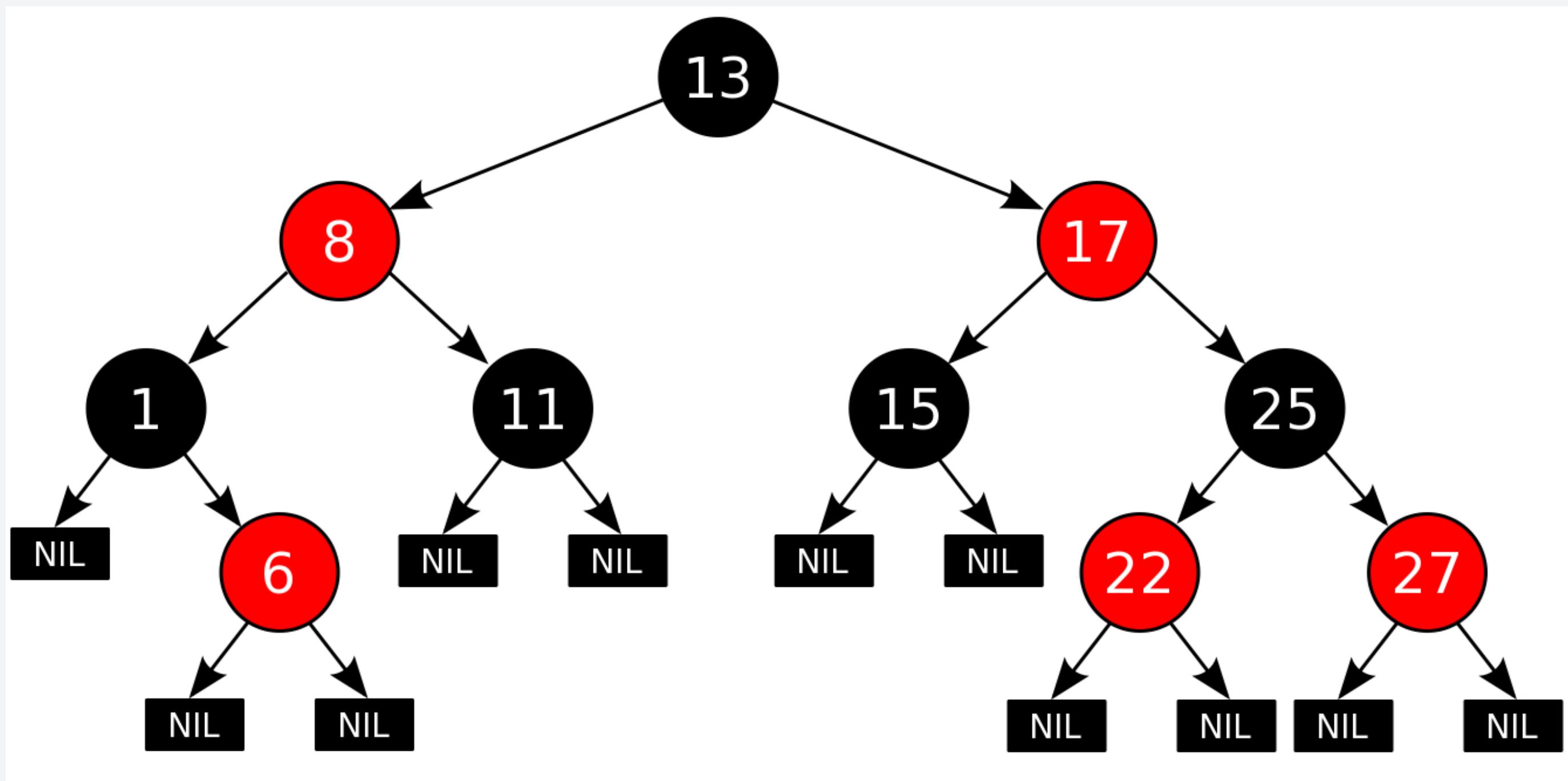
HASH TABLE



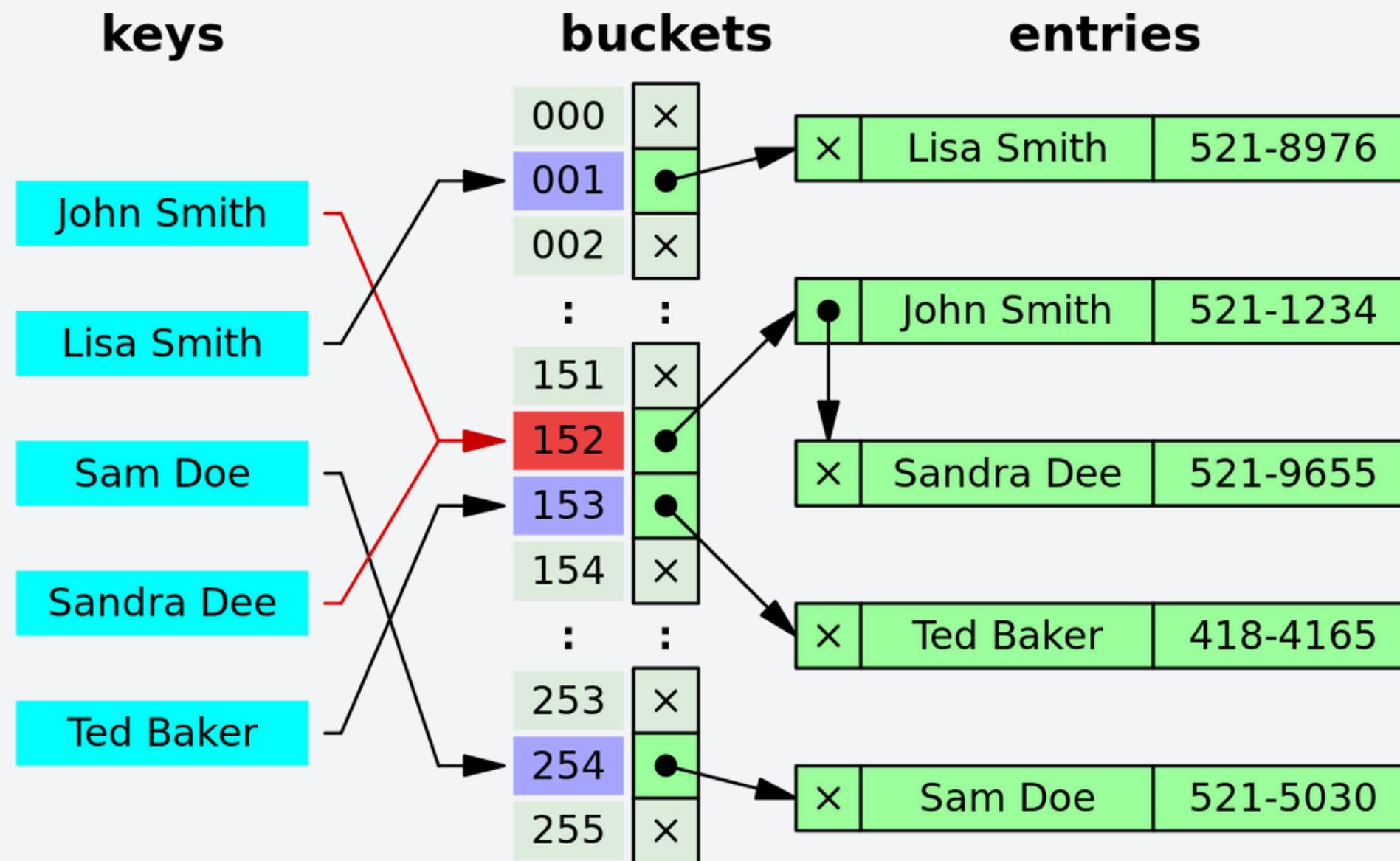
Хранит историю из последних К обращений к данной странице.

QUEUE

RED-BLACK TREE



HASH TABLE



separate chaining

OUR REPOSITORY



<https://github.com/UsoltsevI/Cache>

STRUCTURE

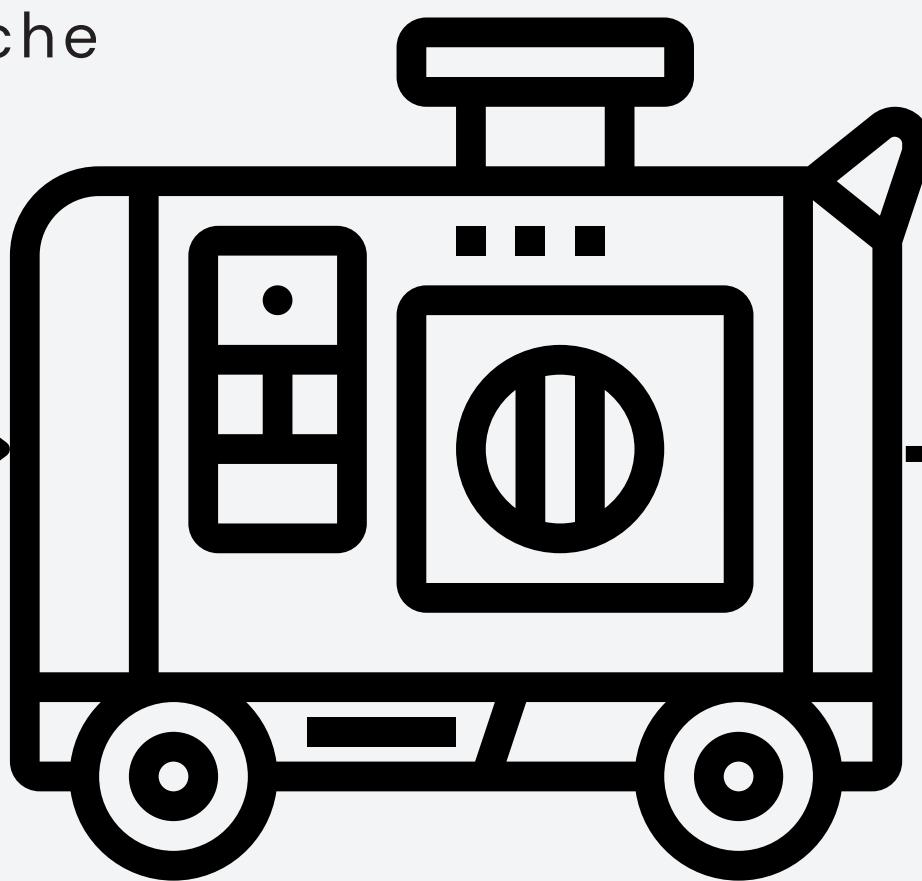
```
Cache/
    ├── include/
    │   ├── cache.h
    │   ├── hash.h
    │   ├── list.h
    │   ├── queue.h
    │   └── tree.h
    ├── source/
    │   ├── cache.c
    │   ├── hash.c
    │   ├── list.c
    │   ├── queue.c
    │   └── tree.c
    ├── tests/
    │   ├── resources/
    │   │   ├── test*.dat
    │   │   └── test*.ans
    │   └── results/
    │       └── res_compare.csv
    ├── cache_compare.c
    ├── test_generator.c
    ├── CMakeList.txt
    └── main.c
```

TEST GENERATOR

- Программа для создания тестирующих данных для cache tester.

1 2 3 5 2 5 7 1 5 7 3 1

input



- Содержит примитивную реализацию cache LRU-K внутри, которая работает за $O(M*K)$

0 0 0 0 1 1 0 1 1 1 1 0

output

- Из-за своей простоты работает очень долго, поэтому вывод пишется с файл, которым далее многократно использует tester

- Получает на вход набор страниц, на выход подаёт последовательность из 0 и 1. 0 - если miss, 1 - если hit.

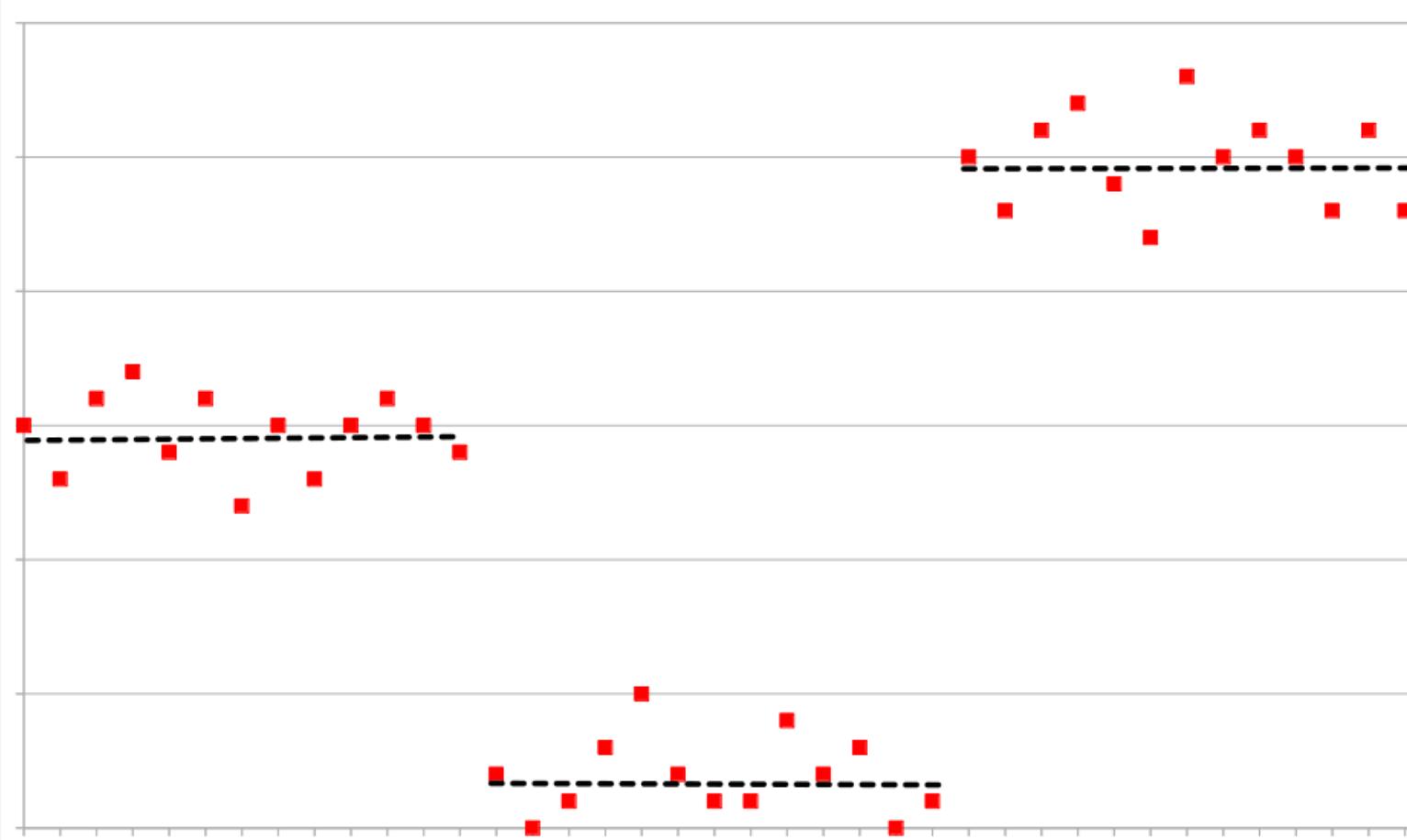
CACHE TESTER

- При несовпадении выводит сообщение с информацией о ошибке
- Запускает настоящий cache LRU-K, сопоставляя результаты его работы с ожидаемыми, полученными с помощью генератора тестов

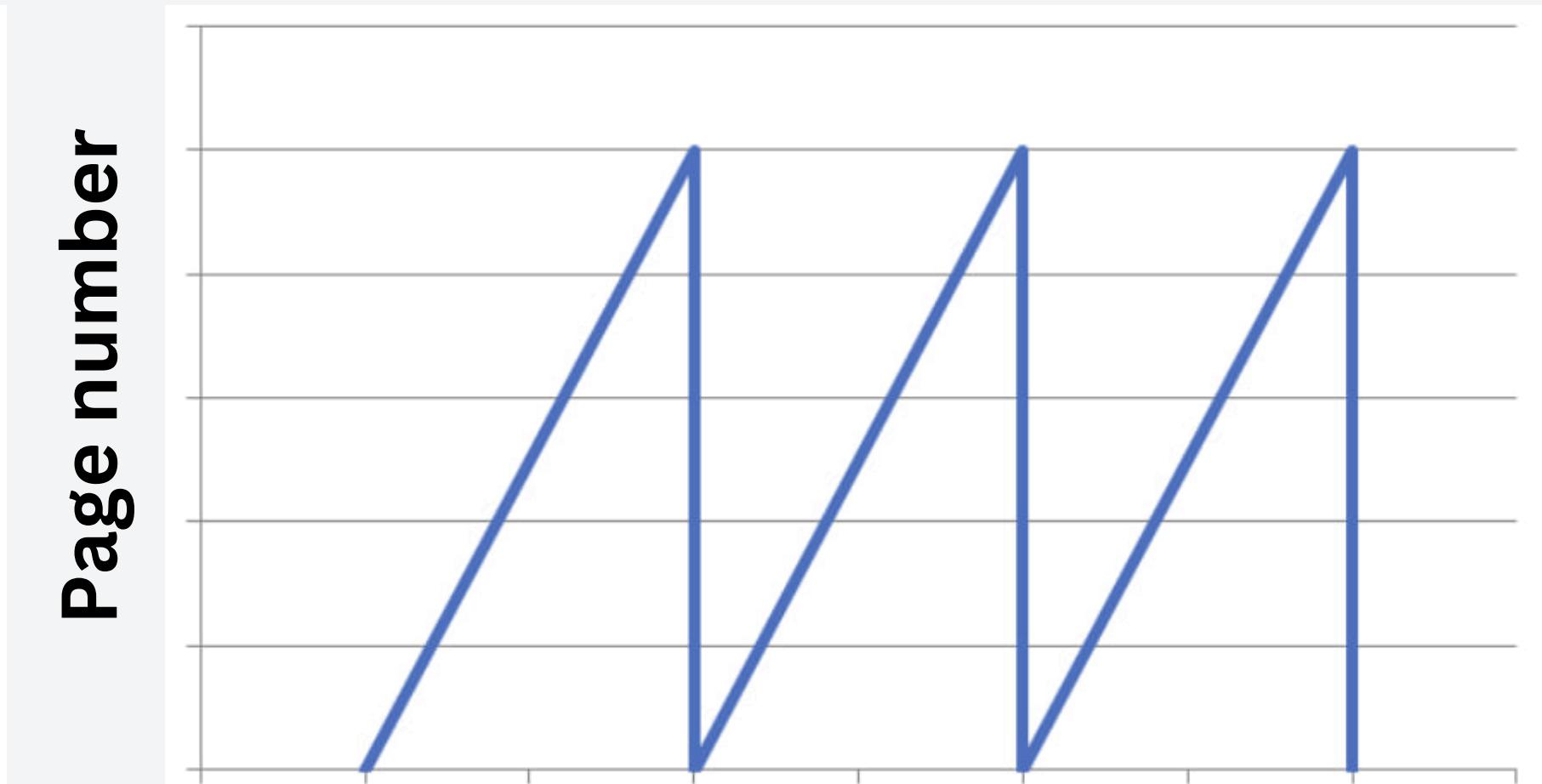
мем с котиком

CACHE COMPARATOR

Виды входных данных



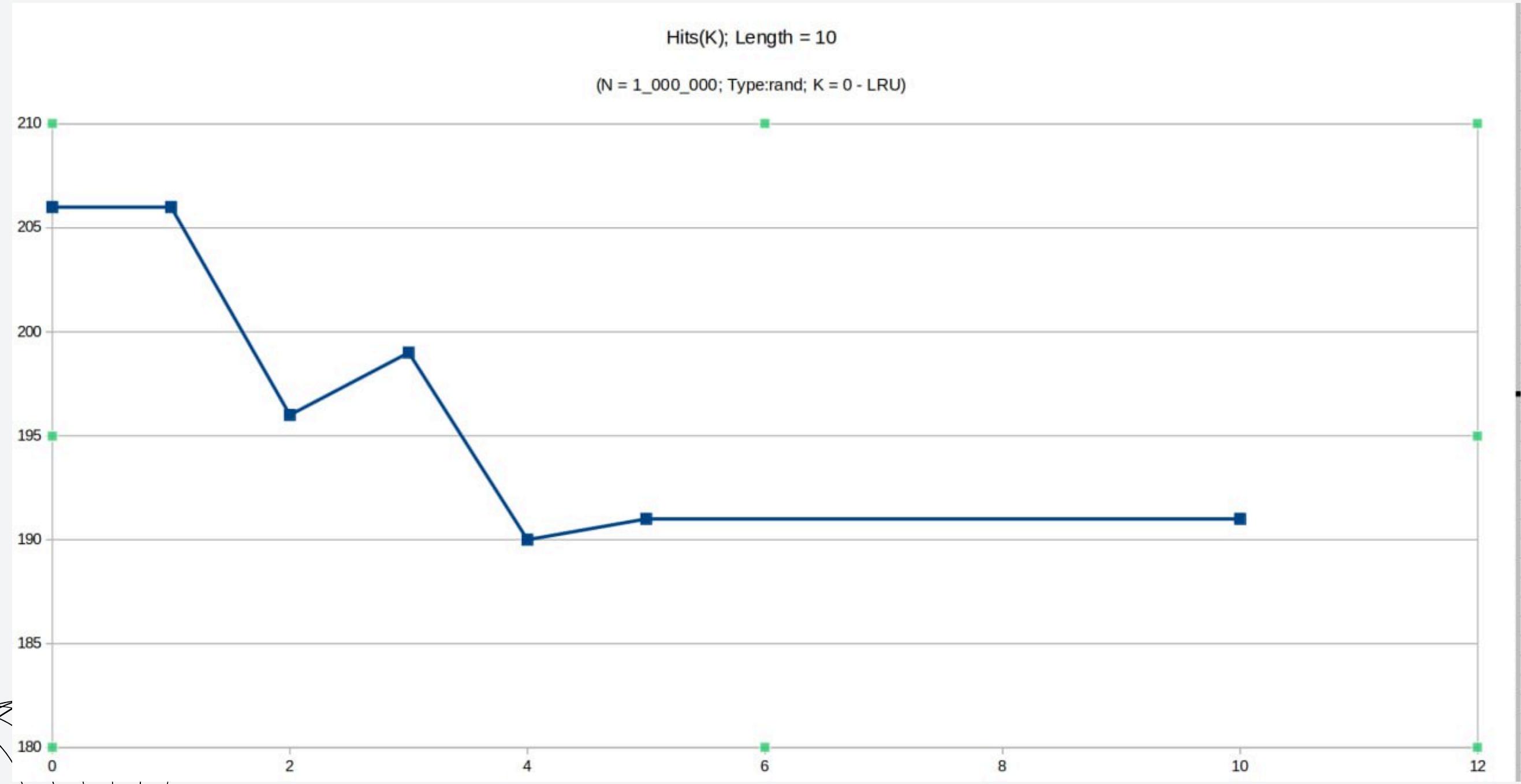
Флуктуирующие



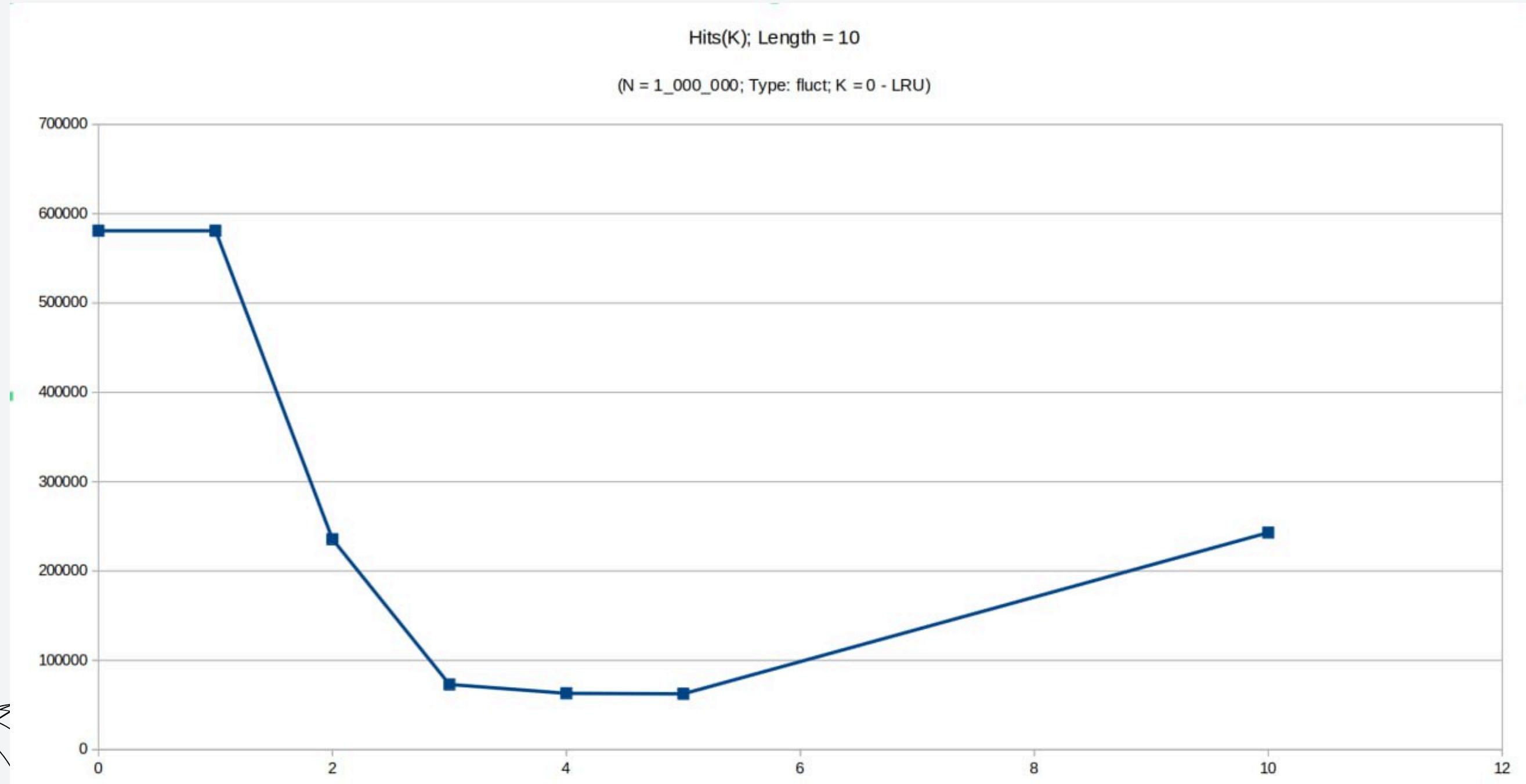
Пилообразные

Случайные

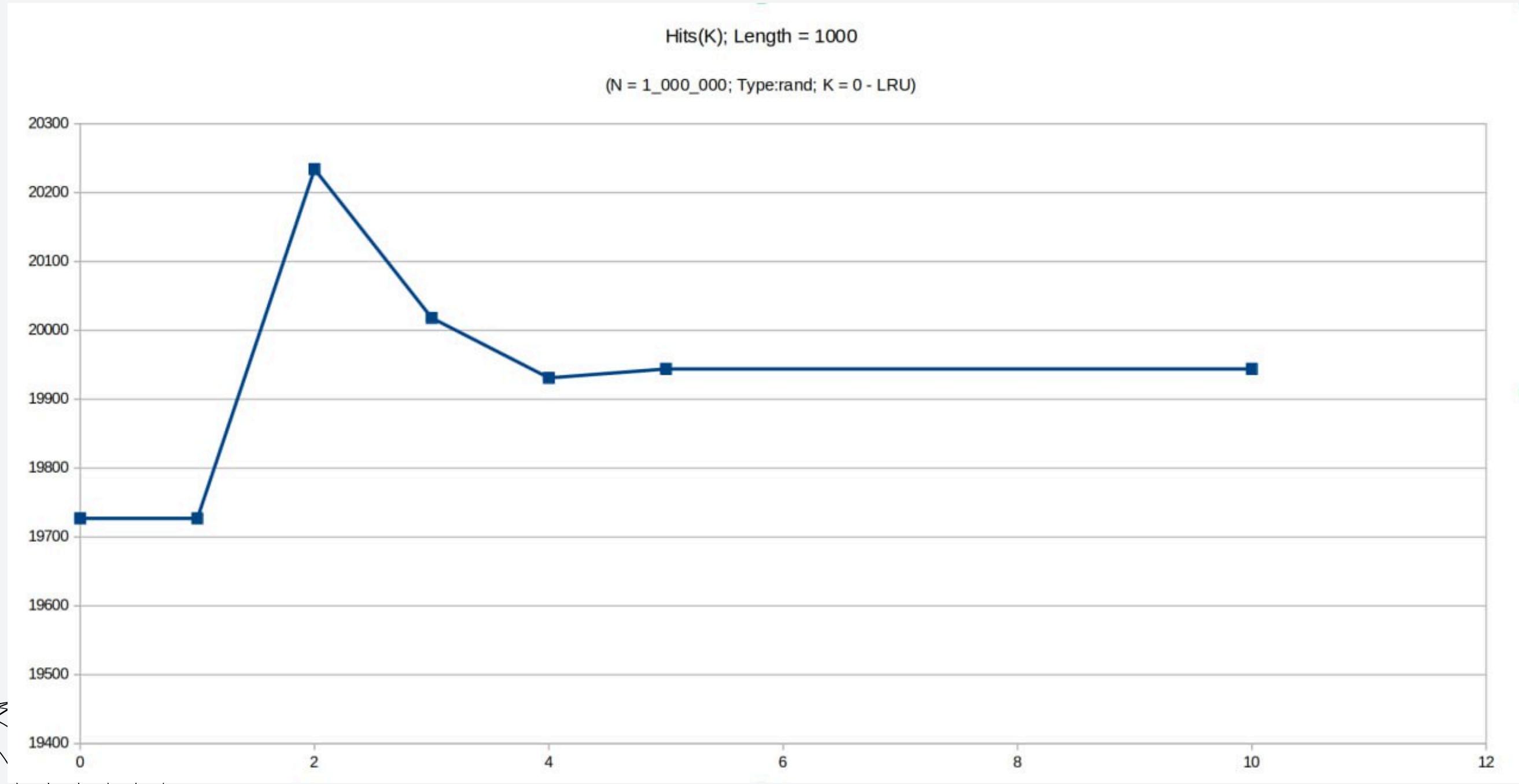
TEST RESULTS



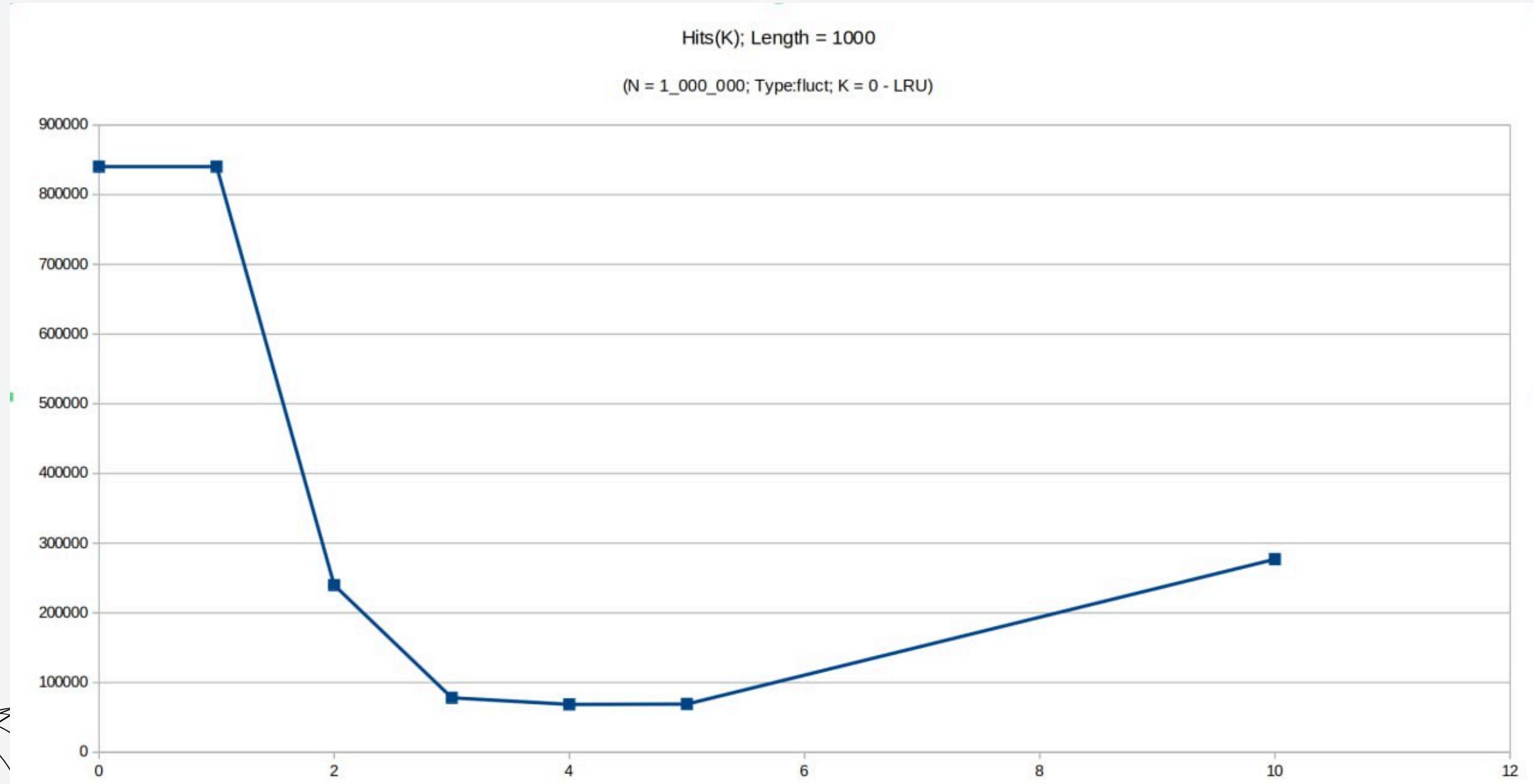
TEST RESULTS



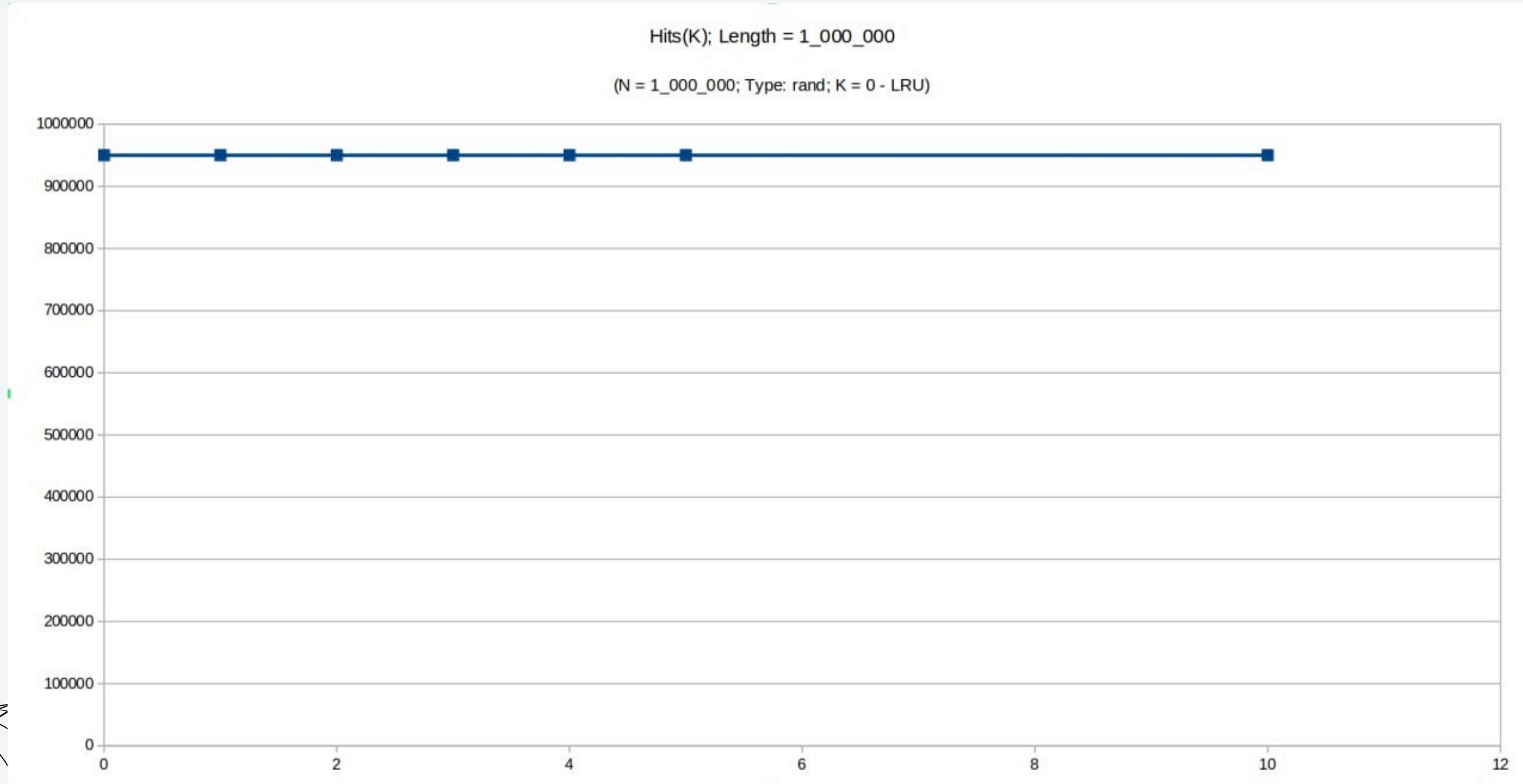
TEST RESULTS



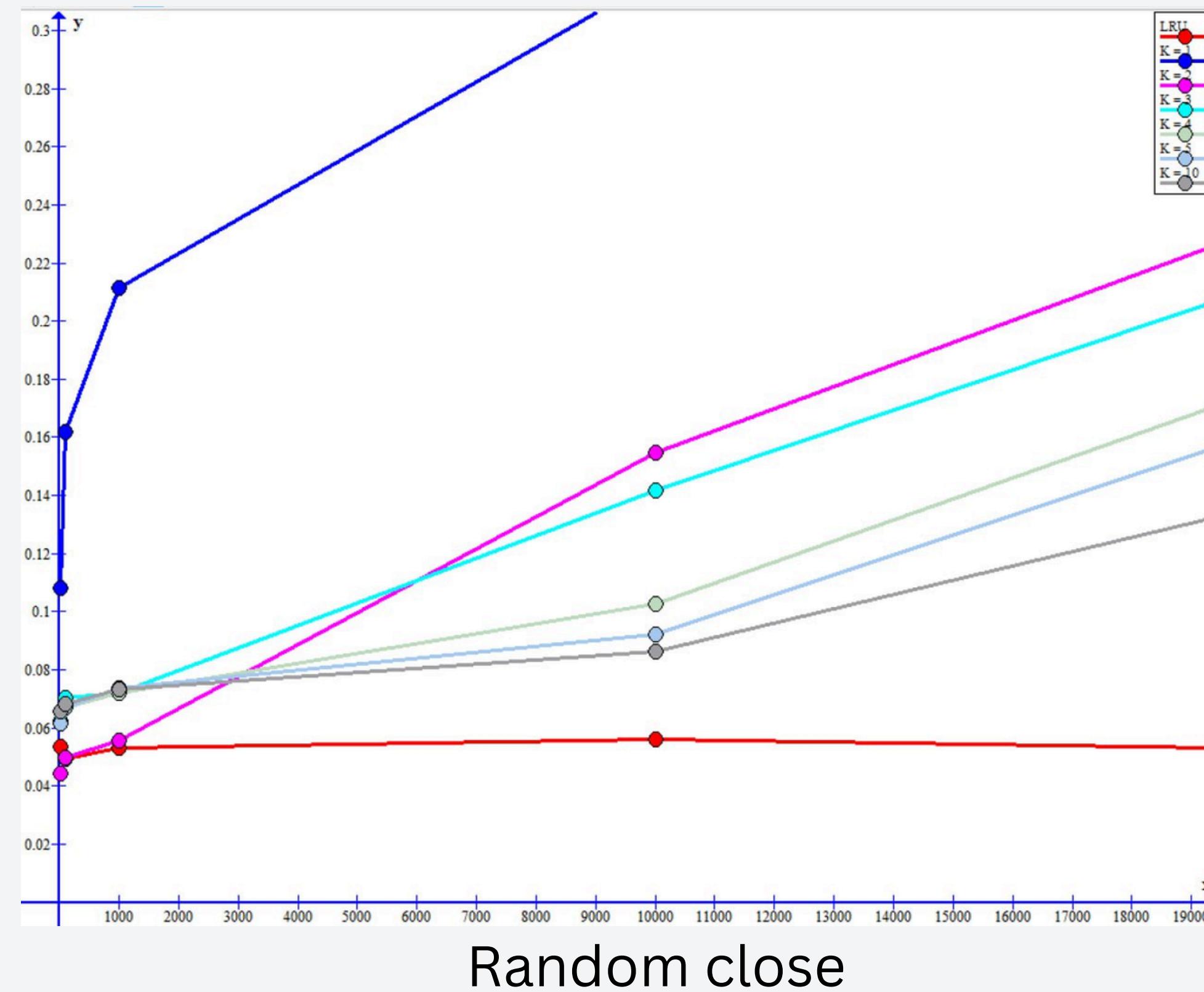
TEST RESULTS



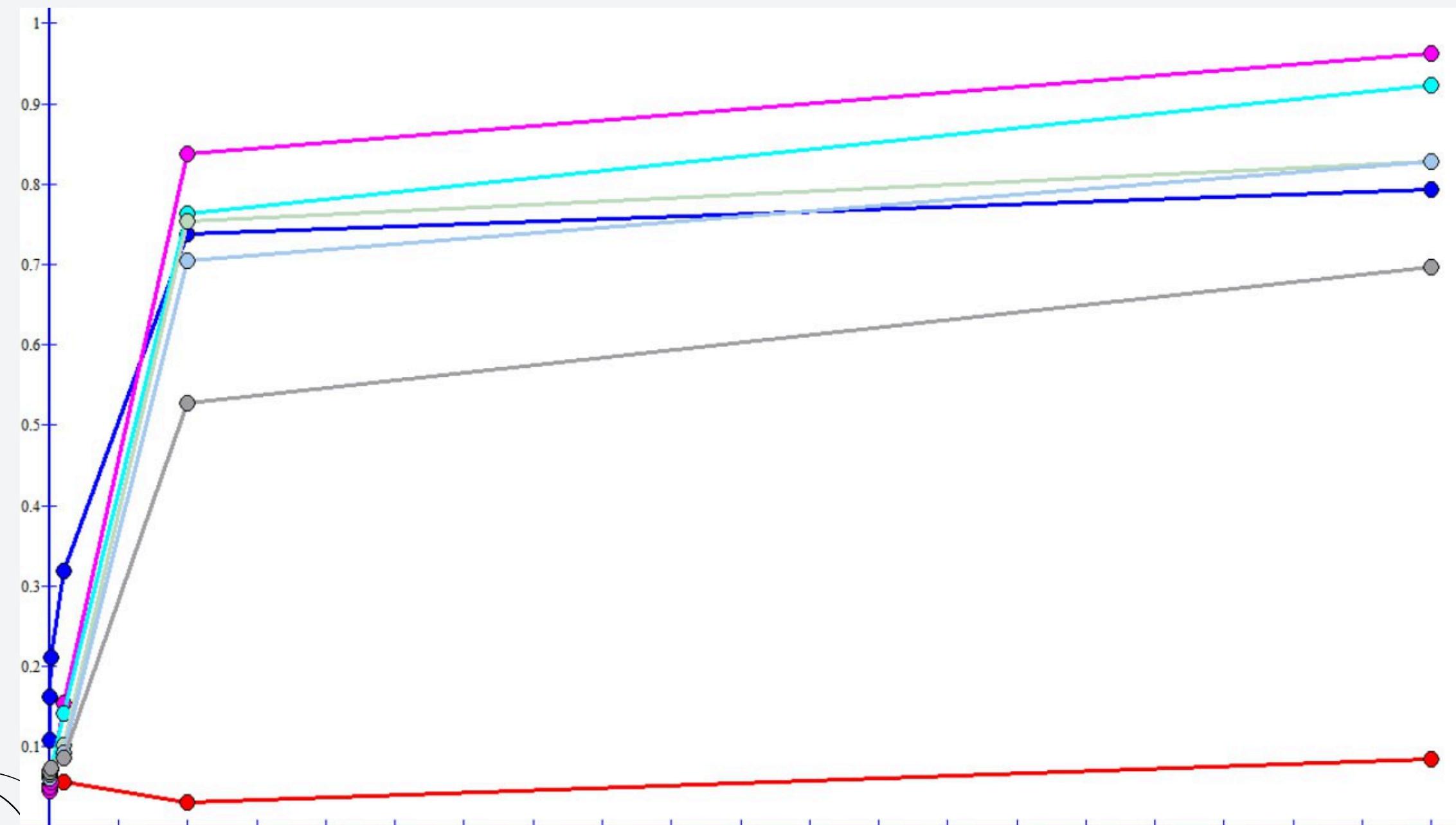
TEST RESULTS



TEST RESULTS

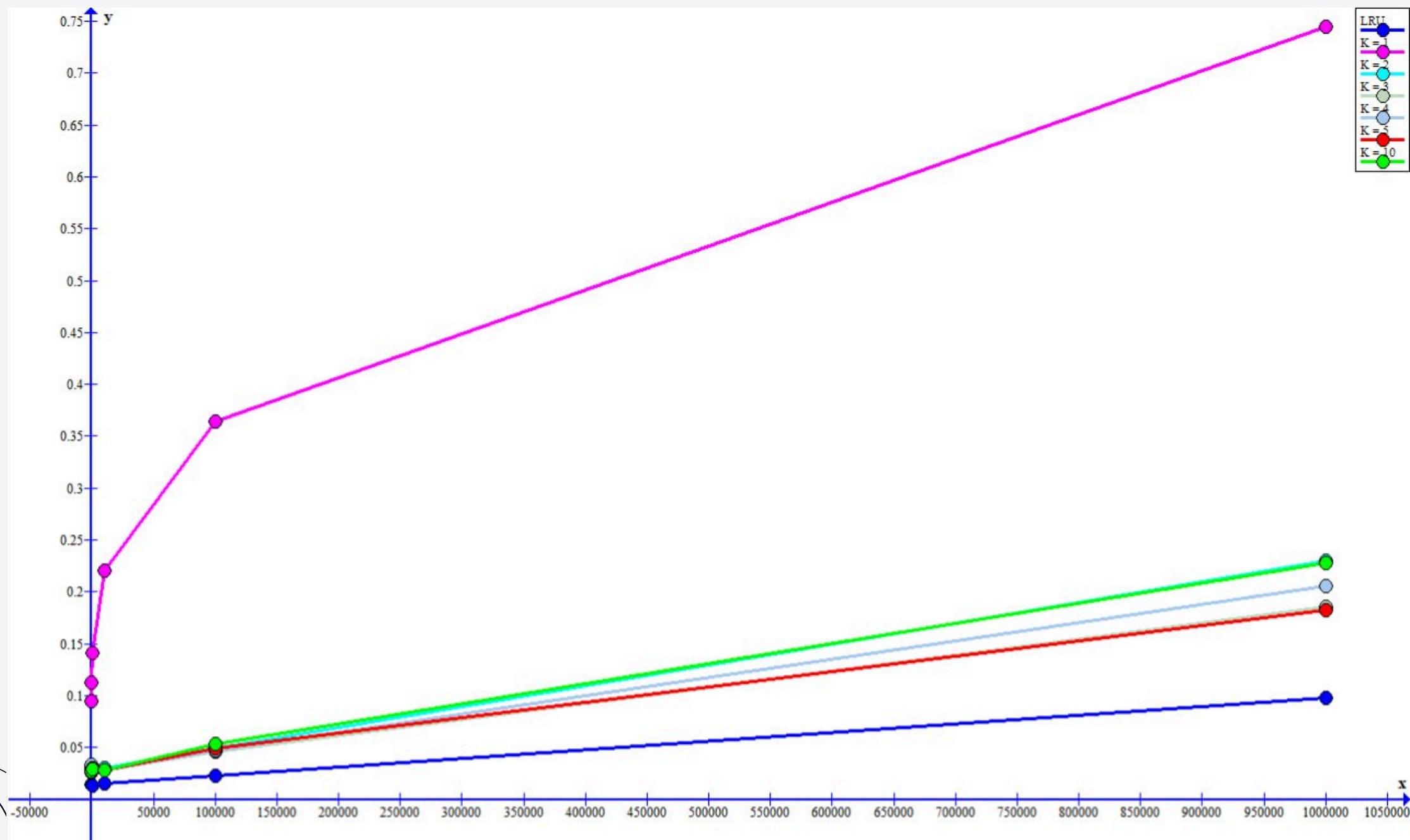


TEST RESULTS



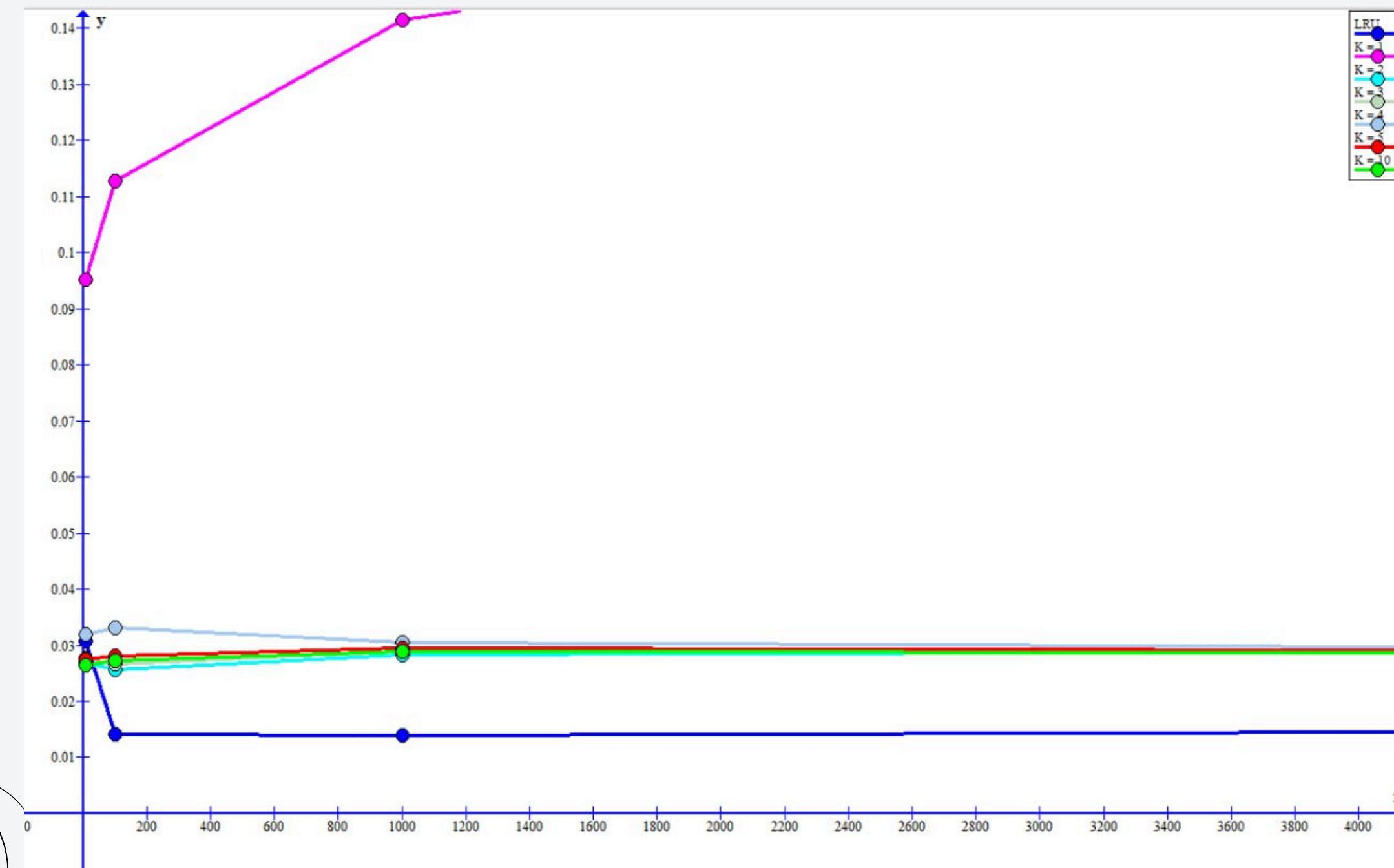
Random far

TEST RESULTS



Saw

TEST RESULTS



Saw closer

THE BEST K

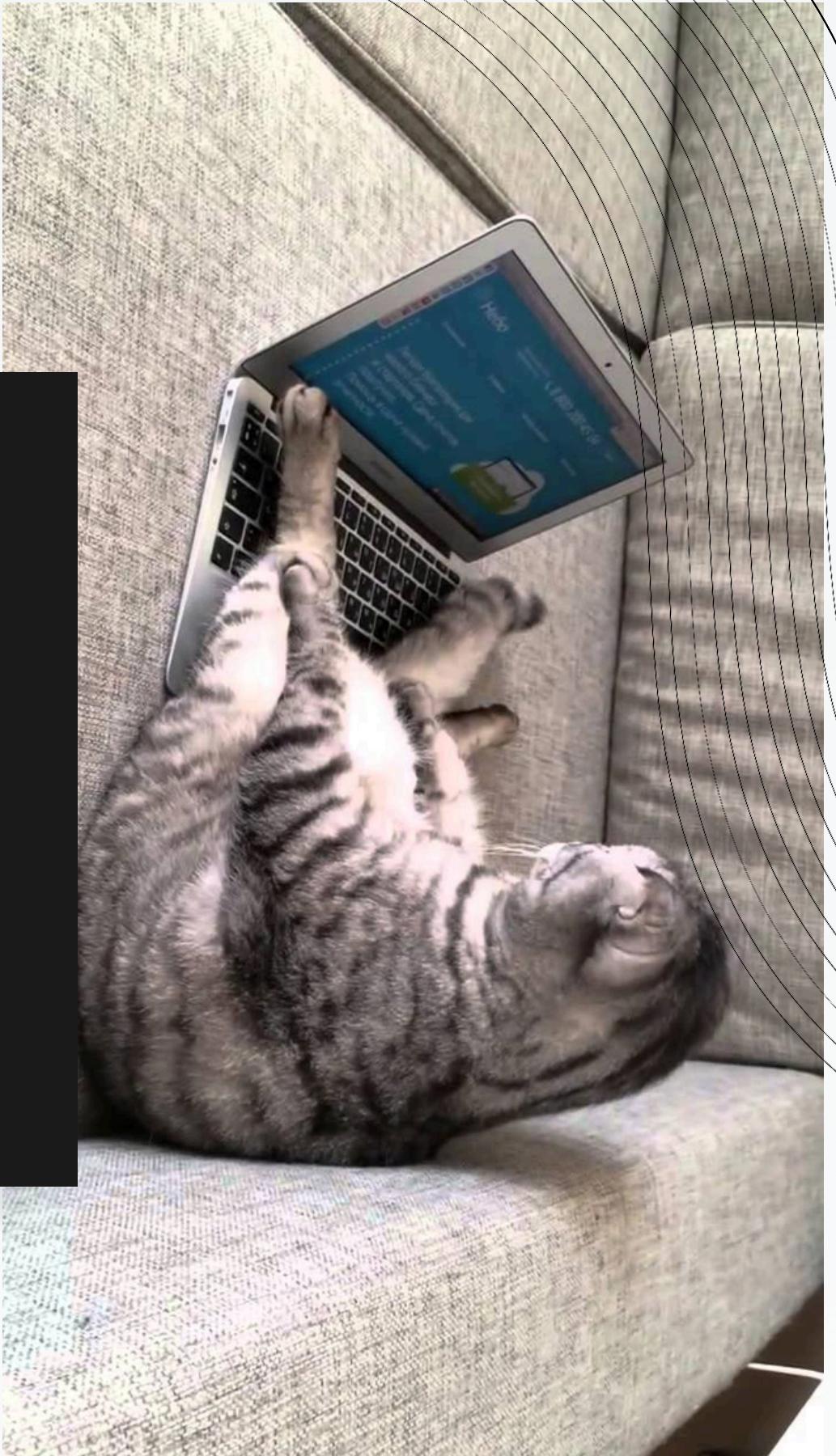
LRU-1 (LRU) показал
наибольшую эффективность
в более чем половине тестов

60% wins



CONCLUSION

- Мы реализовали алгоритм LRU-K, протестировали его и сравнили с обычным LRU.
- По результатам тестирования можно сказать, что наиболее оптимальное K для количества хитов не определяется явно.
- В зависимости от входных данных то или иное значение K становится наиболее приближенным к идеальной модели.



THANK'S FOR WATCHING

Отсутствие мема с котиком

лимит исчерпан

