Основы программного конструирования

ЛЕКЦИЯ №9

17 АПРЕЛЯ 2023

Очень быстрый поиск

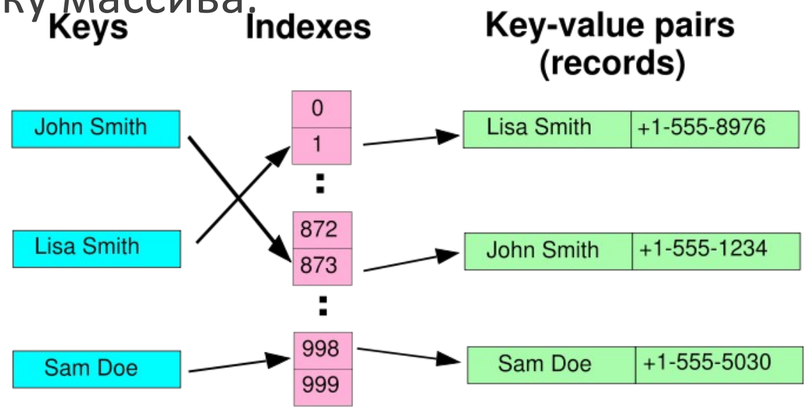
Хеш-таблица

Каждому ключу ставится в соответствие целочисленный код (хеш-код): h(key) → n.

to hash (англ.) – мелко нарубить и перемешать.

Пары *(ключ, значение)* помещаются в массив, индексированный по хеш-коду.

Поиск: вычисляем хеш-код и отправляемся прямо в нужную точку массива.



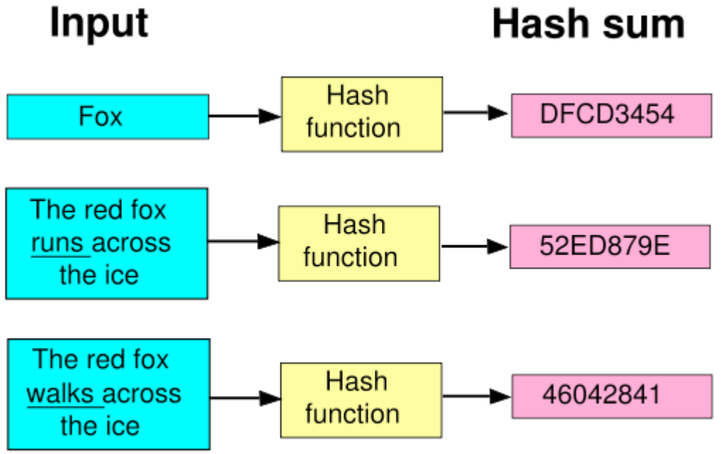
Хеш-функция

Для массива длиной N должна выдавать значения
$0 \leq h_N(k) < N$.

Должна быть детерминистичной.

Должна вычисляться быстро.

Отсутствие кластеризации:



- Очень желательно $h_N(k_1) \neq h_N(k_2)$ для $k_1 \neq k_2$.
- Малое изменение ключа (1 бит) должно давать большое изменение хеш-кода.

Интерфейс хеш-таблицы

get(key) → data.

put(key, data).

has_key(key).

all_keys(). // список не отсортирован

Размещение в массиве

- Метод деления:

  $h_N(k) = h(k) \bmod N$.

- Метод умножения:

  $h_N(k) = [N \bullet \{h(k) \bullet A\}]$ для $0 < A < 1$.

  Например, $A = (\mathrm{sqrt}(5) - 1) / 2 \approx 0{,}6180339887\ldots$

Тривиальные хеш-функции

Если k – целое число, $h(k) = k$.

Если $k \in [0,1)$, то $h_N(k) = [N \bullet k]$.

Если k – произвольное число с плавающей точкой, можно взять мантиссу, приведенную к $[0,1)$.

Общий случай: цепочка байт

Хорошая ли хеш-функция?
$h(b) = \sum b_i$.

Нет.

h("abc") = h("bac") =
h("aad")

хеш-функция дженкинса

```
def jenkins_one_at_a_time_hash(s):
    h = 0
    for c in s:
        h = (h + ord(c)) & 0xFFFFFFFF
        h = (h + (h << 10)) & 0xFFFFFFFF
        h = (h ^ (h >> 6))

    h = (h + (h << 3)) & 0xFFFFFFFF
    h = (h ^ (h >> 11))
    h = (h + (h << 15)) & 0xFFFFFFFF
    return h

jenkins_one_at_a_time_hash(
        "The quick brown fox jumps over the lazy dog")
# 0x519e91f5
```

Примечание: на C будет выглядеть лучше.

Хеш-функция для набора элементов

Не отличная, но вполне пригодная хеш-функция для набора элементов (массив, структура и др.):

$h(e_1, e_2, \ldots, e_n) = h(e_1) + 31 \cdot (h(e_2) + 31 \cdot (\ldots 31 \cdot h(e_n)))$

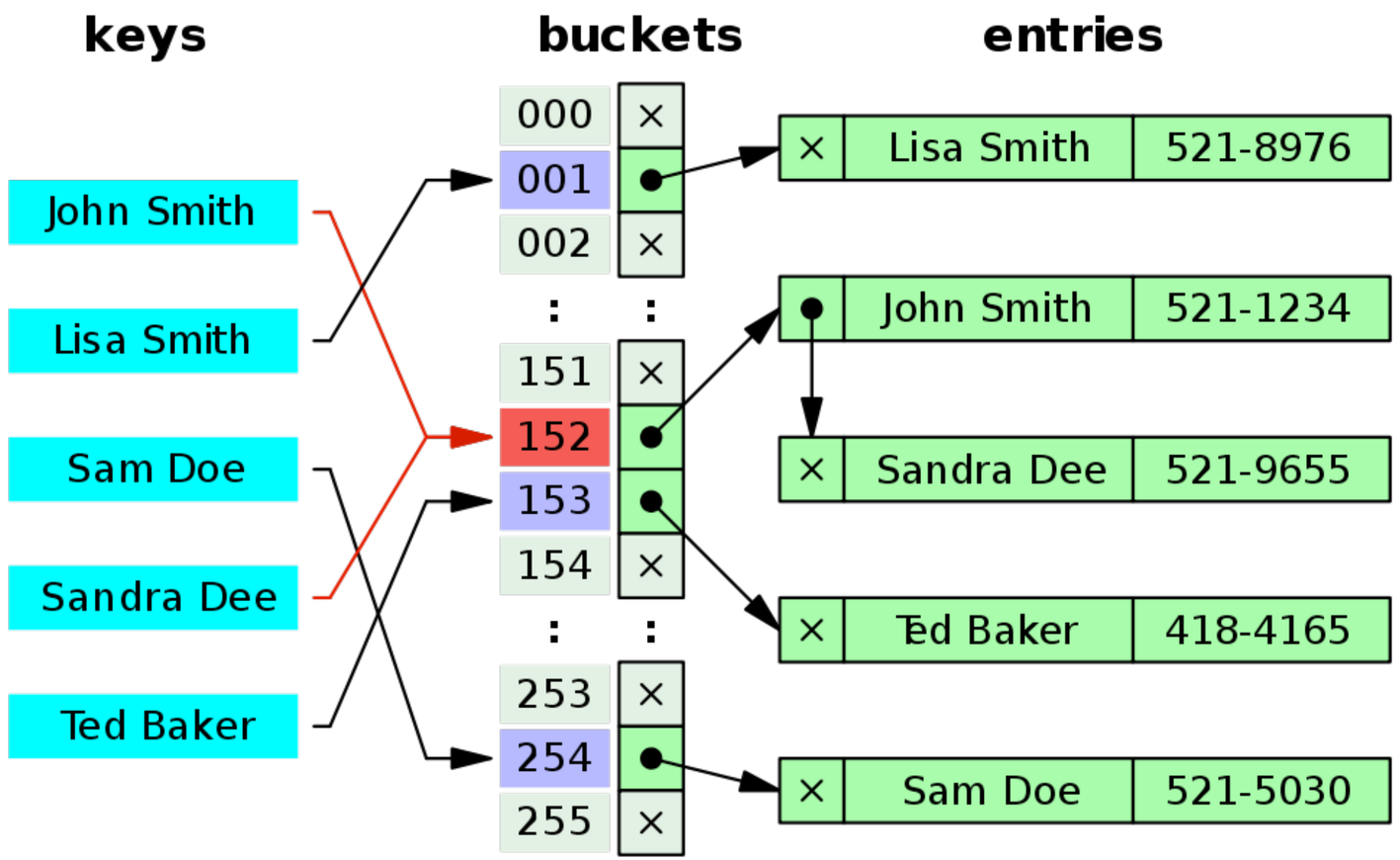
Конфликты

Конфликт – ситуация, когда $h_N(k_1) = h_N(k_2)$ для $k_1 \neq k_2$.

Способы обхода проблемы:

- Идеальное хеширование.
- Усложнение базовой структуры данных.
- Размещение конфликтующего ключа в том же массиве, но в другом месте.
- *Совсем хитрые способы*.

Добавляем списки коллизий



Динамический массив списков

В ячейках массива хранятся указатели на головы списков.

Двухфазный поиск:

- Вычисление хеш-кода.
- Путешествие по списку.

Удобно вставлять в голову.

Альтернативы:

- Динамический массив^2.
- Сбалансированное дерево.

Открытая адресация

Вставка ключа: если возникает конфликт, начинаем перебирать другие ячейки, пока не найдем свободную:

- $h_N(k) \to < h_N(k, 0), h_N(k, 1), \ldots, h_N(k, N-1) >$.

Линейное исследование: $h_N(k, m) = (h_N(k) + m) \bmod N$.

- *Первичная кластеризация* – длинные последовательности занятых ячеек.

Квадратичное исследование

$h_N(k, m) = (h_N(k) + c_1 \cdot m + c_2 \cdot m^2) \bmod N$.

Требует специального выбора c_1, c_2 и m.

Вторичная кластеризация: при конфликте между k_1 и k_2 последовательности $h_N(k_1, m)$ и $h_N(k_2, m)$ совпадают.

Двойное хеширование

$h_N(k, m) = (h_N(k) + m \bullet h'_N(k)) \bmod N$, где $h'_N(k)$ – вторая хеш-функция.

Значения $h'_N(k)$ должны быть взаимно просты с N, чтобы последовательность перебирала все ячейки таблицы.

- $N = 2^j \rightarrow h'_N(k)$ возвращает нечетные значения.

Нет ни первичной, ни вторичной кластеризации.

Лучший способ использования открытой адресации!

Кукушиное хеширование

Две отдельные таблицы размера N: T_1 и T_2 с хеш-функциями $h_{1,N}$ и $h_{2,N}$.

Любой ключ k находится либо в ячейке $T_1[h_{1,N}(k)]$,
либо в $T_2[h_{2,N}(k)]$.

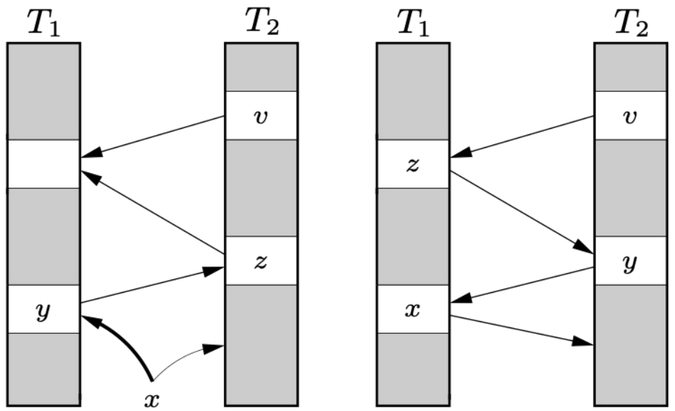
Поиск за O(1) гарантирован!

Кукушка вставляет

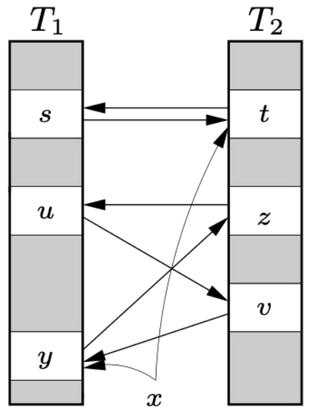
Вставка в $T_1[h_{1,N}(k)]$.

Если ячейка занята, то находящийся в ней ключ перемещается на альтернативную позицию в T_2, вытесняя находившийся там ключ и т.д.

Если процесс зацикливается, выбираются новые хеш-функции, и таблицы перестраиваются заново.



Вставка *X*



«Не лезет»

Хеш vs. дерево

Дерево:

- Нужна операция сравнения: $k_1 \leqslant k_2$.
- Операции за O(log N).
- Упорядоченная структура данных.

Хеш-таблица:

- Нужна хеш-функция hash(k) и операция сравнения $k_1 = k_2$.
- Операции где-то между O(1) и O(N).
- Неупорядоченная структура данных.

Криптографические хеш-функции

Создают «отпечаток» данных, имеющий фиксированный размер.

Для $H = h(k)$ очень трудно найти $k = h^{-1}(H)$.

Имея $H = h(k)$, очень трудно организовать коллизию (найти такое k_2, чтобы $H = h(k) = h(k_2)$).

MD5 (128 бит), SHA-1 (160 бит), SHA-2 (224-512 бит), SHA-3/Keccak (произвольная длина), GOST и т.д.

Checksum

Контрольная сумма используется для проверки целостности данных, передаваемых по *незащищенным* каналам. При этом контрольная сумма передается по *защищенному* каналу.

Пользователь может проверить корректность полученных данных, вычислив контрольную сумму и сравнив ее с опубликованной.

$ curl http://releases.ubuntu.com/21.04/SHA256SUMS

567210e81e01b1ddb0f135b4fde90ff20a89b717608e4e916d3f44f1205b66c6
 *ubuntu-21.04-beta-desktop-amd64.iso
394d4de1b19efb61bd0bd9bc3fb9d71a4d36b460e2fe1dbfd6ec040f09ed7662
 *ubuntu-21.04-beta-live-server-amd64.iso

Адресация по содержимому

Вместо «имени» объекта используем значение криптографической хеш-функции от содержимого.

Дубликаты отсутствуют как факт!

Используется в BitTorrent, в системе управления версиями файлов Git и др.

Теоретический минимум

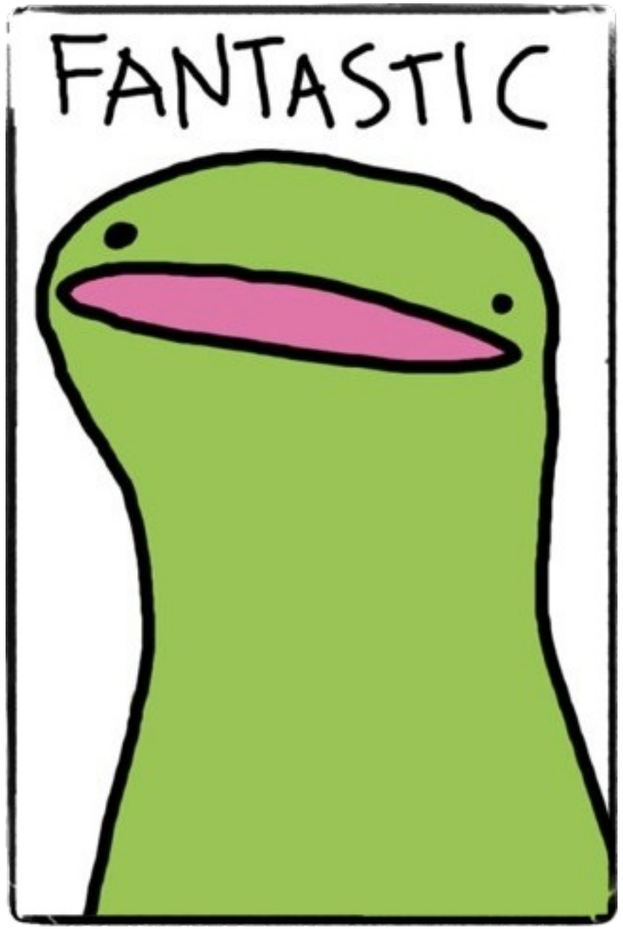
1. Чем отличаются компилируемые языки программирования от интерпретируемых?
2. Что такое динамическая и статическая типизация?
3. Область видимости переменных и время жизни объектов.
4. Сложность добавления/удаления/поиска элементов в различных структурах данных (дерево, хэш таблица, список, динамический массив).

Практический минимум

- Понятные названия переменных и функций (читаемость кода).
- Наличие применения подхода «разделяй и властвуй» / принципа декомпозиции.
- Наличие тестов.

Хороший проект

- Функциональность (решение заявленной задачи).
- Отсутствие ошибок, устойчивость.
- Понятный интерфейс, наличие документации.
- Качество кода:
  - Функции и модули.
  - Отсутствие copy-paste.
  - Отступы и т. д.
- Наличие тестов.
- Клёвый!



Устойчивость

Аксиома. Входным данным нельзя доверять.

Следствие. Данные нужно проверять и либо заменять некорректные значения корректными, либо сигнализировать об ошибке.

Чем проще программа, тем она, как правило, устойчивее.

Интерфейс молуля (библиотеки)

Вопросы, на которые ответ должен быть ясен:

- Кто создает и освобождает ресурсы?
- Как возвращаются ошибки?
- Самый простой способ ответить на вопросы: поместить в документацию небольшой пример.

Интерфейс командной строки

Программа, запущенная без аргументов (или с ключами

-h, --help, /?), выдает справку: «я такая-то программа, меня запускать так-то».

Использование кода возврата sys.exit() для сигнализирования об ошибках и результатах.

Текстовые сообщения об ошибках на разные случаи.

Пример справки (ZIP)

```
$ zip
Copyright (c) 1990-2008 Info-ZIP - Type 'zip "-L"' for software license.
Zip 3.0 (July 5th 2008). Usage:
zip [-options] [-b path] [-t mmddyyyy] [-n suffixes] [zipfile list] [-xi list]
  The default action is to add or replace zipfile entries from list, which
  can include the special name - to compress standard input.
  If zipfile and list are omitted, zip compresses stdin to stdout.
  -f   freshen: only changed files  -u   update: only changed or new files
  -d   delete entries in zipfile    -m   move into zipfile (delete OS files)
  -r   recurse into directories     -j   junk (don't record) directory names
  -0   store only                   -l   convert LF to CR LF (-ll CR LF to LF)
  -1   compress faster              -9   compress better
  -q   quiet operation              -v   verbose operation/print version info
  -c   add one-line comments        -z   add zipfile comment
  -@   read names from stdin        -o   make zipfile as old as latest entry
  -x   exclude the following names  -i   include only the following names
  -F   fix zipfile (-FF try harder) -D   do not add directory entries
  -A   adjust self-extracting exe   -J   junk zipfile prefix (unzipsfx)
  -T   test zipfile integrity       -X   eXclude eXtra file attributes
  -y   store symbolic links as the link instead of the referenced file
  -e   encrypt                      -n   don't compress these suffixes
  -h2  show more help
```

Пример вывода zip и unzip

```
$ zip src *.c *.h
  adding: assoc_list.c (deflated 68%)
  adding: phonebook.c (deflated 62%)
  adding: reader.c (deflated 67%)
  adding: assoc_list.h (deflated 57%)
  adding: reader.h (deflated 39%)
$ unzip -t src.zip
Archive:  src.zip
    testing: assoc_list.c                     OK
    testing: phonebook.c                      OK
    testing: reader.c                         OK
    testing: assoc_list.h                     OK
    testing: reader.h                         OK
No errors detected in compressed data of src.zip.
$ unzip -t notfound.zip
unzip:  cannot find or open notfound.zip, notfound.zip.zip or notfound.zip.ZIP.
```

Пользовательский интерфейс

Должно быть понятно, как выйти из программы!

Подтверждение деструктивных операций.

Диалоговый режим: приглашения ко вводу.

Консольный интерфейс: меню, справка по горячим клавишам.

Самостоятельная документация.

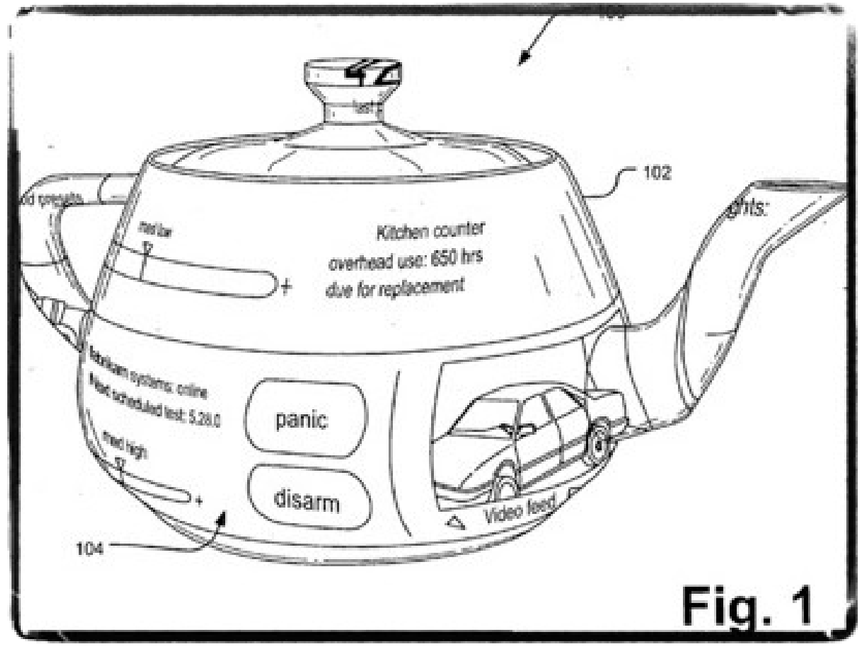
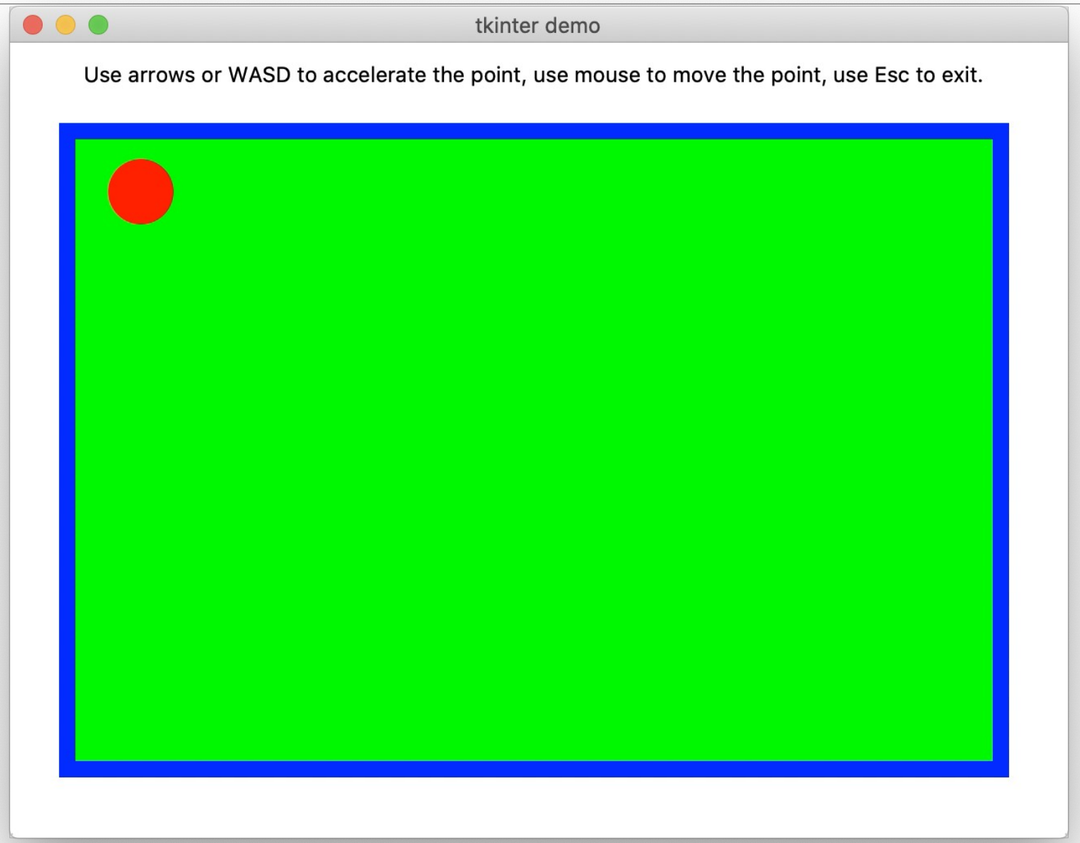


Fig. 1

Графический интерфейс



Документирование

Аксиома. Ваша программа – на Земле не первая.

Следствие. Пользователь, скорее всего, уже имеет опыт взаимодействия с другими программами.

Следствие. Если ваша программа ведет себя так же, как уже известная, пользователь сразу сможет с ней работать, ничего не читая.

Главные вопросы, на которые отвечает документация:

- «Что это? На что это похоже из того, что я уже знаю?»
- «Какие здесь особенности?»

Основные виды документации

- Файл README (что это, зачем нужно и на что похоже).
- Инструкция для пользователя (детальное описание функций и режимов работы).
- Инструкция для разработчиков (внутреннее устройство, модули, соглашения, используемые алгоритмы и т.п.)

Автоматизированное тестирование

Аксиома. Все не протестируешь.

Следствие. Тестировать нужно там, где появление ошибок наиболее вероятно и где ошибки буду

наиболее критичны.

На сегодня все!