

## Ход работы

В рамках задания было реализовано хранение данных различными структурами данных (Связный список, хеш-таблица и двоичное дерево поиска) и измерено время выполнения операций вставки в массив, поиска в массиве, удаления из массива для сравнения характеристик этих структур, а также реализована функция вывода всех элементов массива. Данные на вход подавались в случайном и в отсортированном порядке в размере 10000 уникальных, неповторяющихся элементов. Экспериментальные измерения проводились пять раз для каждого случая структуры и входных данных. На основе экспериментальных данных построены графики.

## Измерения

Структура	Режим	Операция	Среднее время (сек)
Linked list	shuffled	Insert	2.9601924599846825
Linked list	shuffled	Find	0.02592504001222551
Linked list	shuffled	Delete	0.01210075996350497
Hash-table	shuffled	Insert	0.004802840016782284
Hash-table	shuffled	Find	3.839998971670866e-05
Hash-table	shuffled	Delete	2.0040012896060943e-05
BST	shuffled	Insert	0.0542771200183779
BST	shuffled	Find	0.0003209599992260337
BST	shuffled	Delete	0.00015207994729280472
Linked list	sorted	Insert	2.5840181399835274
Linked list	sorted	Find	0.021389540005475282
Linked list	sorted	Delete	0.010239119990728796
Hash-table	sorted	Insert	0.004485400021076202
Hash-table	sorted	Find	3.3660023473203185e-05
Hash-table	sorted	Delete	1.8039997667074203e-05
BST	sorted	Insert	0.03127337999176234
BST	sorted	Find	0.00029561996925622227
BST	sorted	Delete	0.00014696004800498485

Csv – таблицу программа создаёт автоматически.

Программы, отвечающие за построение всех графиков (12 обычных и 12 с функцией `ln` для уменьшения масштаба) приложены к работе в папке *data/lab\_1\_data* под названием *graphics1-1.py* и *graphics1-2.py* соответственно. В отчёте же были использованы только необходимые графики.

Был использован только `matplotlib`.

## Ответы на вопросы:

### 1. Как порядок входных данных влияет на скорость вставки в BST?

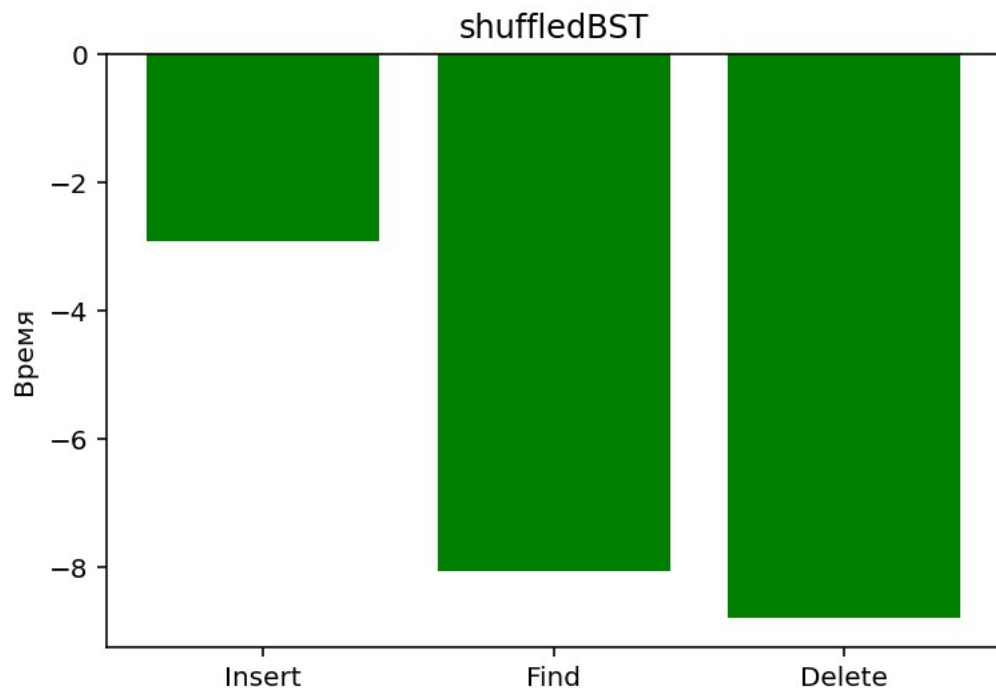


График времени операций с неупорядоченными данными для BST

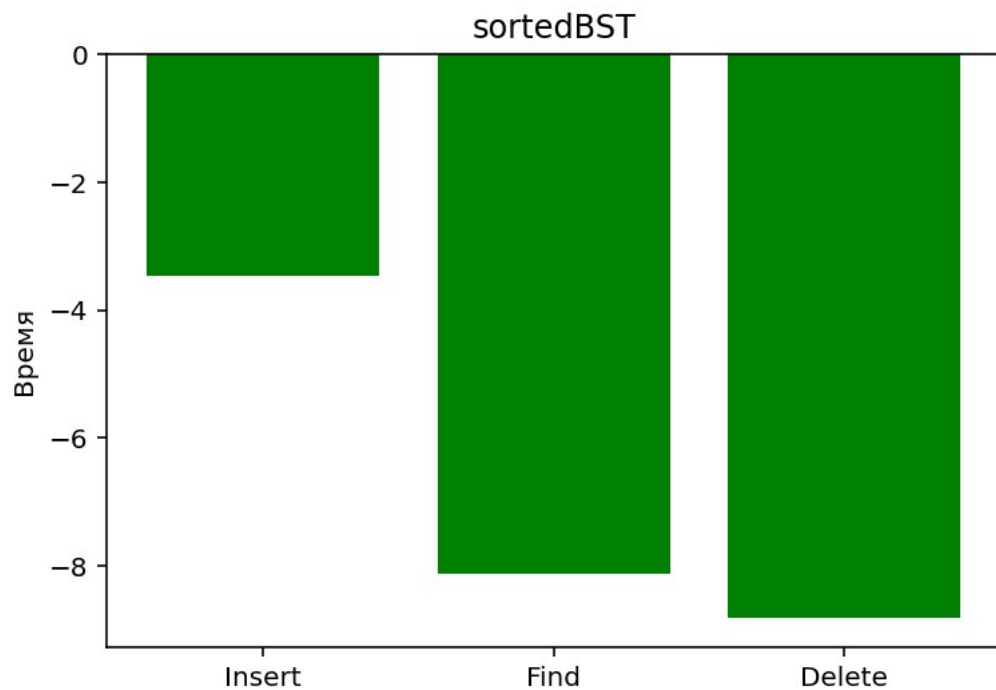


График времени операций с упорядоченными данными для BST

Как мы можем видеть на графиках, вставка для упорядоченных данных в BST в нашем случае получилась чуть быстрее, чем для неупорядоченных. Это не очень совпадает с логикой работы данной структуры (возможность создания длинных ветвей в одну сторону для упорядоченных данных должна замедлять все операции). Неточность средних показателей скорее всего вызвана недостаточностью экспериментальных данных. Стоит также отметить, что в нашем случае различие по времени настолько незначительно, что ответ скорее: “никак”.

## 2. Почему хеш-таблица почти не чувствительна к порядку?

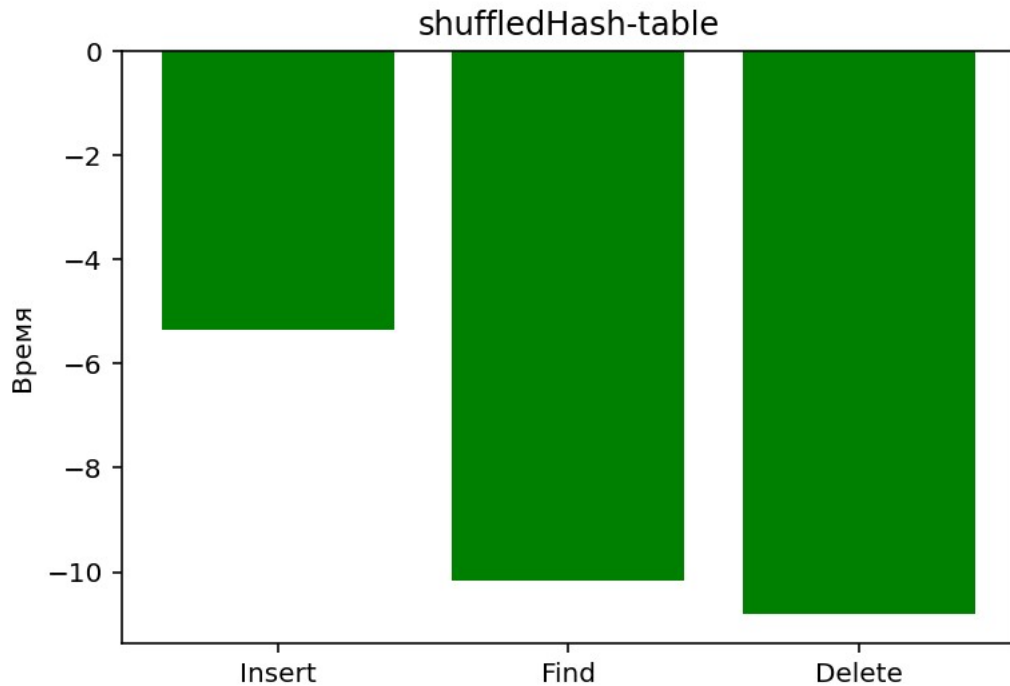


График времени операций с неупорядоченными данными для хеш-таблицы

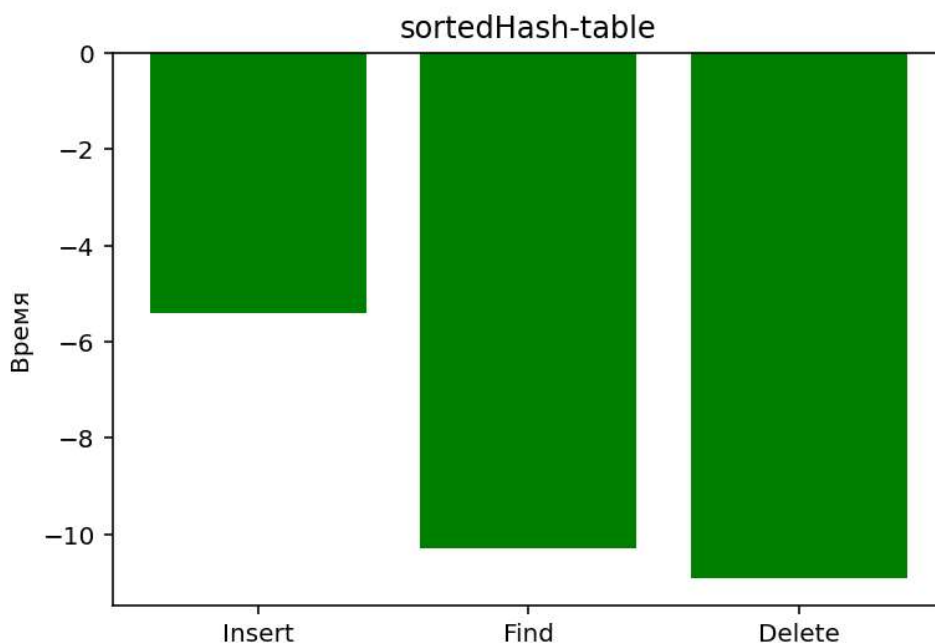


График времени операций с упорядоченными данными для хеш-таблицы

Грамотно построенная хеш-таблица не чувствительна к порядку, так как количество её бакетов – простое число, от которого количество элементов (в нашем случае  $N = 10\,000$ ) составляет примерно 0.7 или меньше. Это называется коэффициент заполнения. В программе выбрано число бакетов – 15013. Так как это число простое, а в поиске индекса элемента как раз применяется деление на него, обеспечивается защита от коллизий и повышается случайность распределения элементов по хеш-таблице.

Проще говоря – каждый элемент (в основном) имеет уникальный индекс.

В случае дальнейшего увеличения объёма данных количество бакетов также последовало бы увеличить для сохранения скорости.

### 3. Почему связный список всегда медленен при поиске?

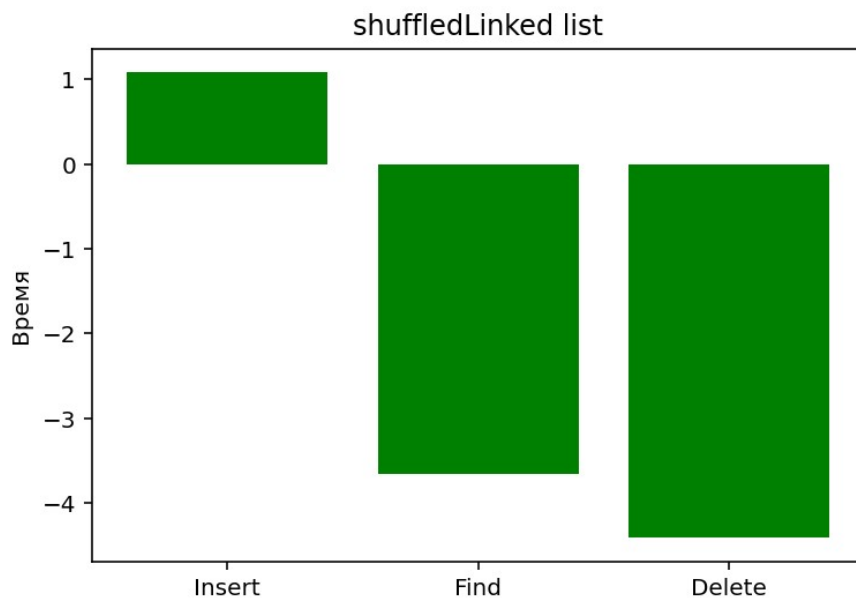
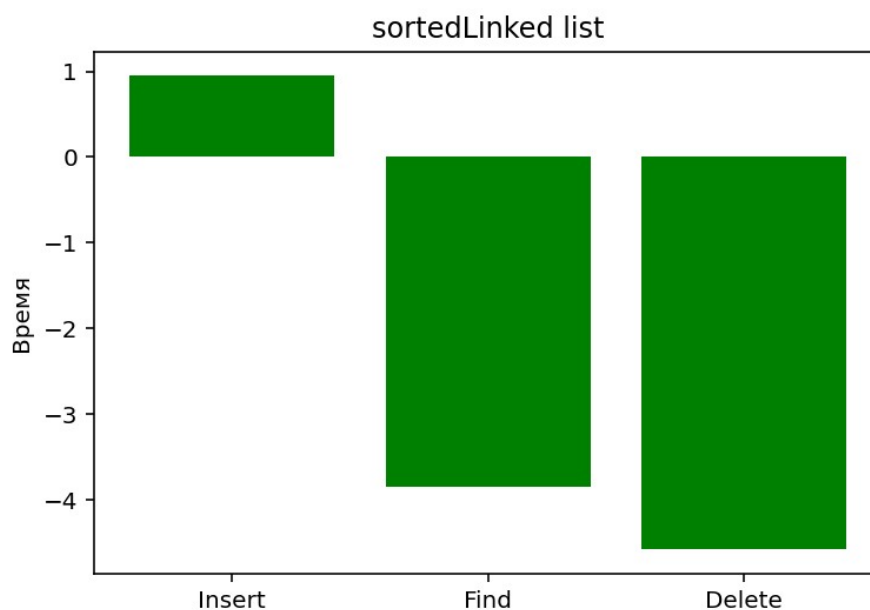


График времени операций с неупорядоченными данными для связного списка



## График времени операций с упорядоченными данными для связного списка

Большие затраты времени на поиск в связном списке вызваны отсутствием индексации и необходимостью перебирать все элементы строго друг за другом. Если хеш-таблица и двоичное дерево поиска выделяют хоть какие-то условные адреса для объектов благодаря хешу и логическим условиям соответственно, то в связном списке отсутствует возможность выбрать наиболее удачное направление поиска. Другими словами, если элемента в списке вовсе нет или он находится в самом конце – нам придётся перебрать весь список.

### 4. Как удаление работает в каждой структуре?

Удаление в связном списке:

Если мы хотим удалить элемент А, то у нас есть три варианта развития событий:

- 1) Элемент А оказался во главе списка, тогда новой главой списка выбираем идущий позади него (next от него)
- 2) Элемент А оказался между началом и концом списка. Перед ним стоит элемент В, после него стоит элемент С. Тогда мы соединяем В напрямую с С вместо А. (next для С станет В)
- 3) Элемент А оказался в конце списка. Перед ним стоит элемент В. Приравниваем next в В к None, таким образом любая вставка обойдёт А.

Удаление в хеш-таблице:

Вычисляется хеш удаляемого элемента А и по остатку от его деления на количество бакетов ищется ячейка таблицы, в которой элемент предположительно содержится (если он существует)

В этой ячейке может лежать один лишь этот самый элемент А, и в таком случае он просто заменится на None.

Может в ней лежать и связный список элементов, испытавших коллизию, тогда будет повторяться процедура удаления из связного списка, описанная выше.

Данный способ удаления в нашем случае работает быстрее так как количество бакетов достаточно для избежания образования длинных списков. Однако даже небольшое количество бакетов всё равно было бы достаточно для скорости удаления большей, чем у связного списка, ведь в хеш-таблице как-бы происходит разбиение по хешу для упрощения поиска удаляемого элемента.

Удаление в двоичном дереве поиска:

Производится поиск ячейки, содержащей удаляемый элемент с помощью перенаправления в left и right ячейки (в моём случае налево перенаправляется меньший, а направо - больший или равный изначальной ячейке хеш). Искомую ячейку условимся называть подкорнем.

В случае успешного обнаружения подкорня возможно 4 исхода:

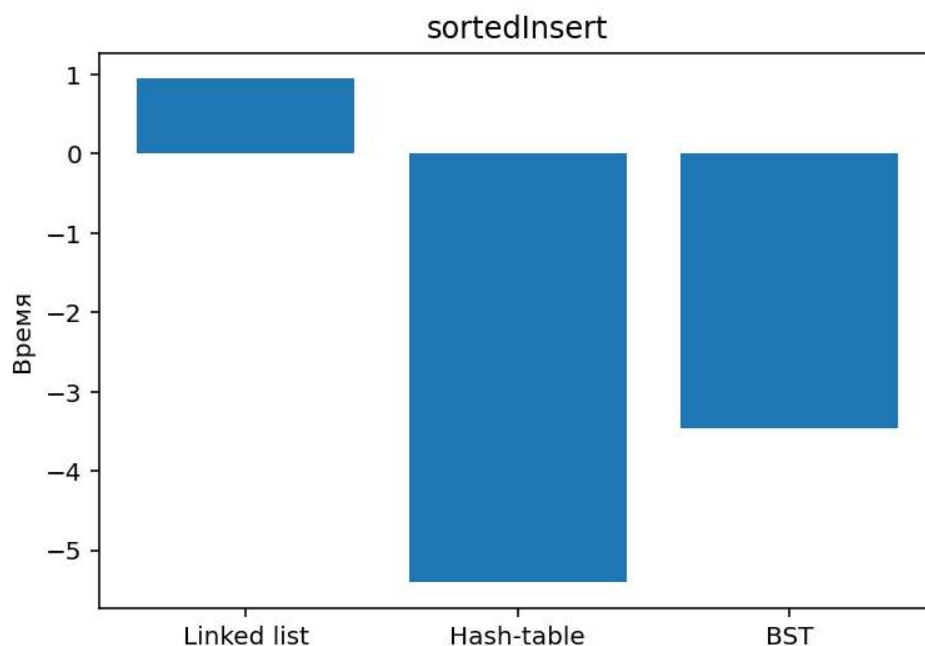
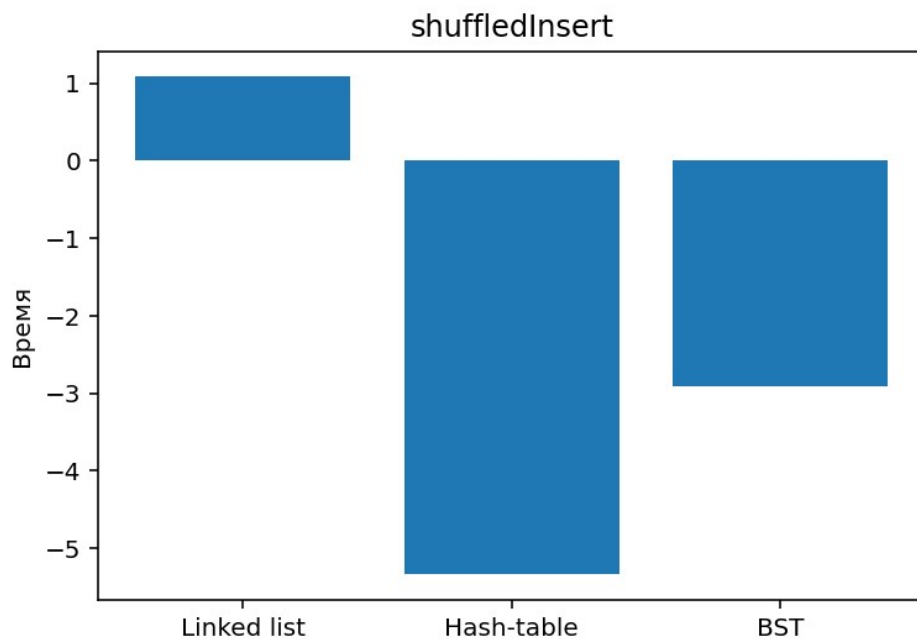
- 1) У подкорня не имеется ответвлений. Тогда подкорень просто заменяется на None
- 2) У подкорня только левое ответвление. Тогда подкорень просто заменяется на первую левую ячейку после себя.
- 3) У подкорня только правое ответвление. Тогда подкорень заменяется на первую правую ячейку.
- 4) У подкорня оба ответвления. Тогда производится поиск самой правой из левых ячеек относительно него. То есть, сначала берётся первая левая ячейка и проверяется, есть ли у неё правое ответвление. Если правого ответвления у неё нет, тогда мы действуем аналогично шагу 2.

Если же у неё есть правое ответвление, тогда мы проходим направо по этому ответвлению до упора (до тупика или до следующего поворота налево). Ячейку на которой мы остановимся переместим на место корня, а ячейки, которые могли ответвляться от неё налево, отдадим под правое ответвление родительской ей ячейки. Таким образом мы совершаем так называемый поворот двоичного дерева.

## 5. Вывод.

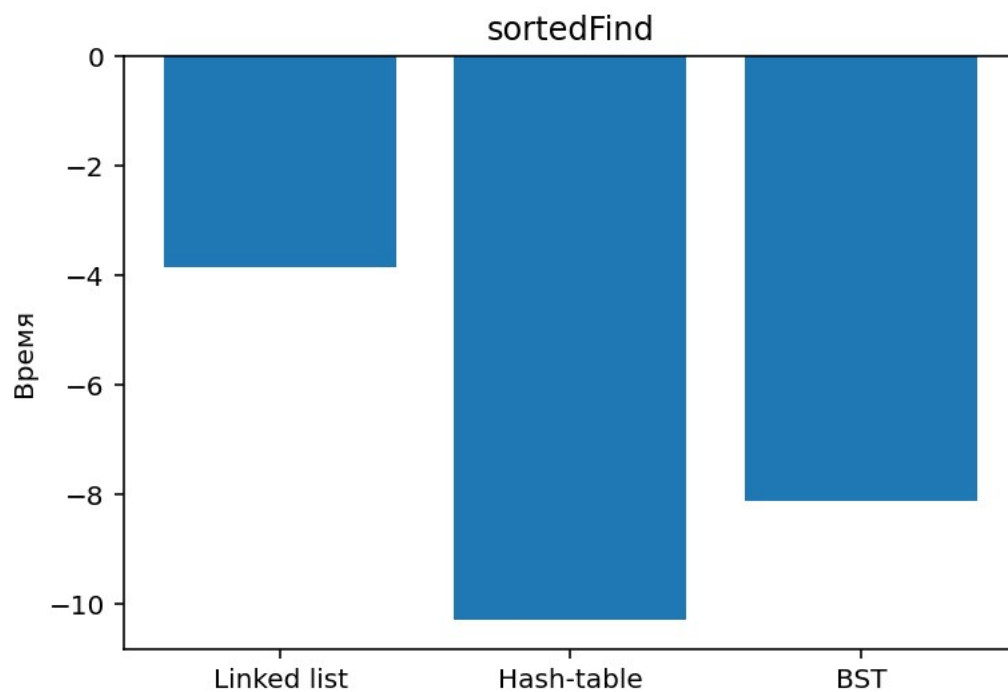
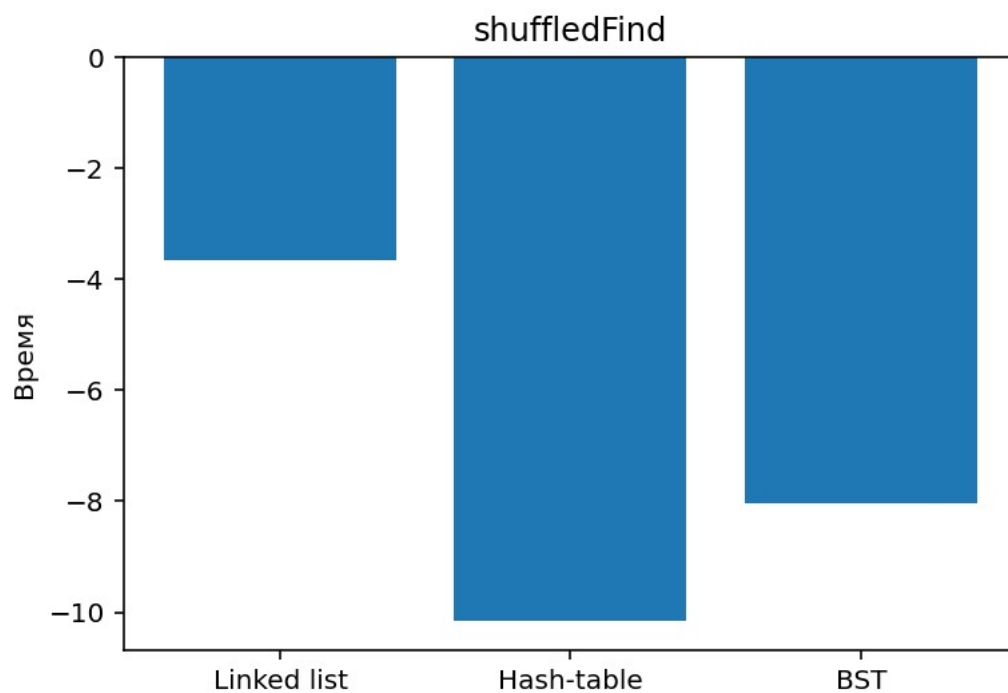
Исходя из проделанной работы и полученных экспериментальных данных, для задач, требующих *частые вставки* лучше всего будет использовать:

- 1) Хеш-таблицу при работе с мало-пересекающимися и часто удаляющимися данными (так как в противном случае потребуются постоянно расширять таблицу, что означает лишние вычислительные ресурсы)
- 2) Двоичное дерево универсально для случаев как удаляющихся, так и долго хранящихся данных (но оно чуть помедленнее хеш-таблицы)



Для задач требующих *частый поиск* лучше всего было бы использовать:

Также хеш-таблицу или двоичное дерево, по той же логике что и со вставками.



Для задач же, требующих *получать данные в порядке* подойдут:

Связный список или хеш-таблица, так как получение данных в случае двоичного дерева поиска требует рекурсивного обращения к ним. В случаях больших объёмов данных это потребует значительных вычислительных мощностей. Стоит отметить что и в хеш-таблице присутствует рекурсия, но далеко не в столь значительном объёме как в дереве.