

Отчёт по лабораторной работе Задание 1 — структуры данных

Цель работы:

Реализовать три различные структуры данных «с нуля», применить их для хранения записей телефонного справочника и экспериментально сравнить производительность основных операций. Вы должны собственными руками написать код, чтобы понять внутреннее устройство связного списка, хеш-таблицы и двоичного дерева поиска, а также осознать их сильные и слабые стороны на практике.

Введение

В данной работе рассматриваются три структуры данных, используемые для хранения записей телефонного справочника. Каждая запись имеет вид пары:

(имя, номер телефона)

Для всех структур данных реализованы одинаковые базовые операции:

`insert(name, phone)` — добавить или обновить запись.

`find(name)` — `phone` или `None`.

`delete(name)` — удалить запись, игнорировать отсутствие.

`list_all()` — список всех записей, отсортированный по имени

Связный список

Связный список — это структура данных, состоящая из узлов, где каждый узел хранит данные и ссылку на следующий узел.

Хеш-таблица

Хеш-таблица — это структура данных, в которой ключ преобразуется в индекс массива с помощью хеш-функции.

Двоичное дерево поиска

Двоичное дерево поиска — это структура данных, в которой каждый узел хранит ключ и две ссылки: на левое и правое поддерево. Значение левого потомка всегда меньше родителя, правого — больше.

Генерация тестовых данных

Создадим список `records` из `N` элементов (например, `N = 10000`), где каждый элемент — кортеж `(name, phone)`. Имена будем генерировать как `f"User_{i:05d}"` для

получения равномерного распределения. Для проверки влияния порядка подготовим два варианта одного и того же набора:

`records_shuffled` — случайный порядок.

`records_sorted` — отсортированный по имени (по алфавиту).

Проведение замеров

Для каждой структуры данных и для каждого режима входных данных (случайный / отсортированный) выполнено:

- А. Вставка всех записей
- Б. Поиск 100 случайных записей (100 имен существуют, 10 не существуют)
- В. Удаление 50 случайных записей

Для каждой операции было зафиксировано общее время проведения

Эксперименты были проведены 5 раз, результаты всех экспериментов были записаны в файл `results.csv` в формате:

Запуск	Структура	Режим	Операция	Время (сек)
1	LinkedList	shuffled	insert	3.4127272000005178

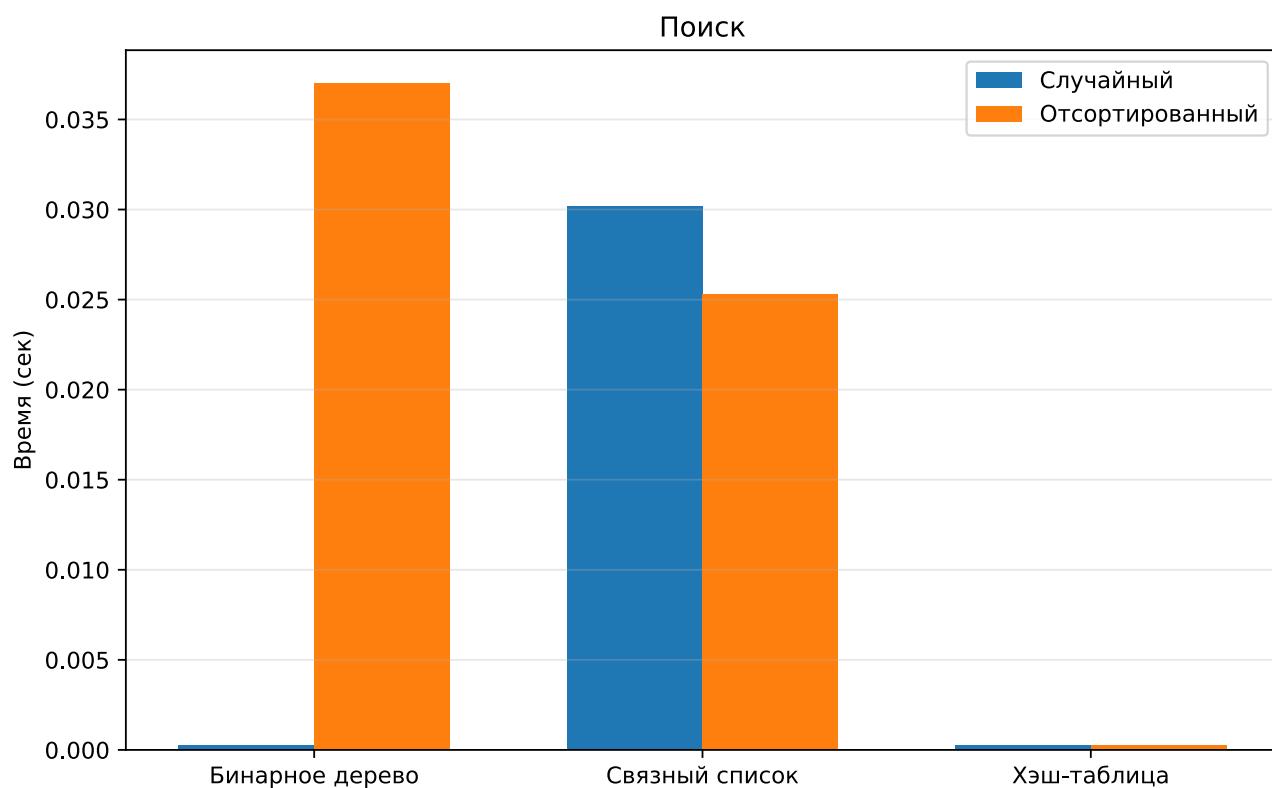
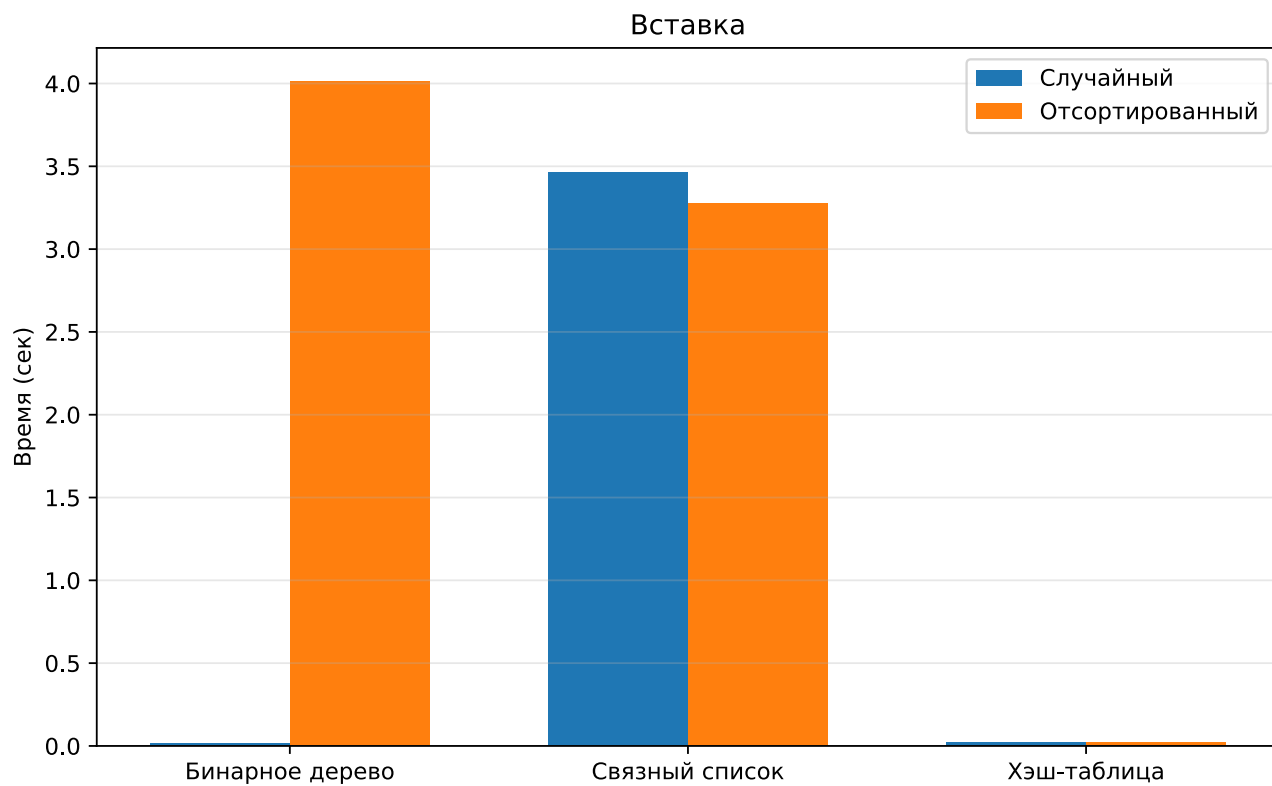
После этого время выполнения для каждой операции было усреднено и записано отдельными строками в тот же файл в формате:

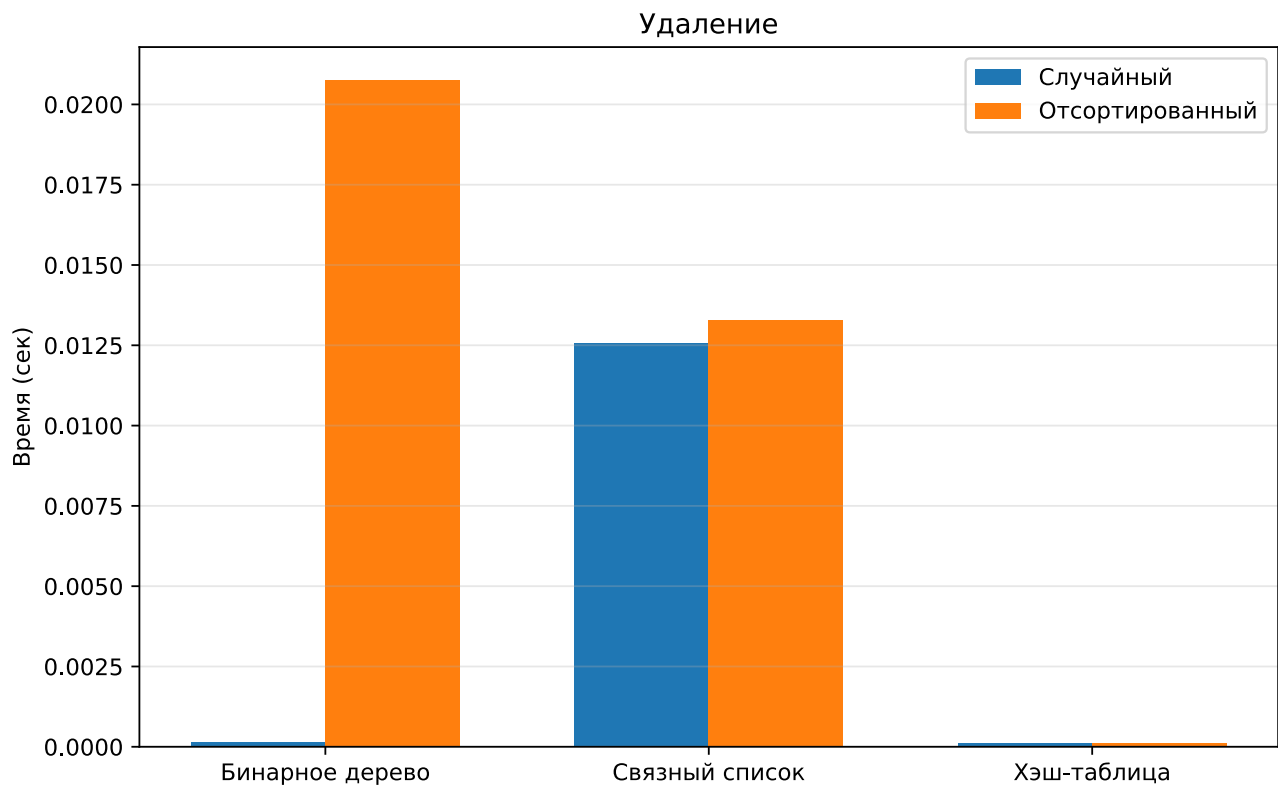
Запуск	Структура	Режим	Операция	Время (сек)
average	LinkedList	shuffled	insert	3.4634849199996096

В результате получены следующие экспериментальные данные (В таблице ниже приведено усредненное время для соответствующих операций и порядка данных):

№	Структура	Режим	Вставка	Поиск	Удаление
1	Бинарное дерево	Случайный	0,01715242	0,00023462	0,00014212
2	Бинарное дерево	Отсортированный	4,01406982	0,03699530	0,02074794
3	Связный список	Случайный	3,46348492	0,03018340	0,01254974
4	Связный список	Отсортированный	3,27399720	0,02528810	0,01326564
5	Хэш-таблица	Случайный	0,01967166	0,00022984	0,00012208
6	Хэш-таблица	Отсортированный	0,01923022	0,00025798	0,00011824

Построение графиков и их анализ





Анализ результатов

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

Вставка в BST на отсортированных данных оказалась примерно:

$4,01406982 / 0,01715242 \approx 234$ раза медленнее

Причина в том, что при случайном порядке дерево получается более-менее сбалансированным. В этом случае путь от корня до места вставки сравнительно короткий, и операция вставки близка к сложности $O(\log n)$.

При отсортированной вставке имена добавляются по возрастанию:

User_00000, User_00001, User_00002,

Из-за чего каждый следующий элемент оказывается больше предыдущего и всегда добавляется в правое поддерево. Это, в свою очередь, приводит к вырождению Бинарного дерева в цепочку, то есть в Связный список. Сложность деградирует до $O(n)$.

По той же причине на отсортированных входных данных ухудшается и поиск, и удаление.

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

Место элементов в Хеш-таблице определяется значением хэш-функции. для каждой записи вычисляется свой индекс, из-за чего каждое имя (например User_00123) попадёт в свой бакет. Даже если хеш-функция допустит коллизию данных, они будут записаны в цепочку в одну ячейку хеш-таблицы. Однако, при правильно подобранной хеш-функции, такие коллизии происходят редко. Поэтому независимо от операции (вставка/поиск/удаление), достаточно лишь вычислять индекс для элемента, а не

проходиться по всему списку или веткам дерева. Из-за детерминированности хеш-таблицы порядок элементов не важен (всё равно для соответствующего имени индекс всегда будет одинаковым).

Получается, что скорость работы хеш-таблицы зависит от:

1. качество хеш-функции;
2. размер таблицы;
3. количество коллизий;
4. длина цепочек в бакетах.

3. Почему связный список всегда медленен при поиске.

Для того, чтобы найти элемент в связном списке, мы неизбежно должны пройти по всем предыдущим элементам списка. Если нужное имя находится в начале, поиск будет быстрым. Но если имя находится ближе к концу или его вообще нет, приходится пройти почти весь или весь список. Поэтому сложность поиска в связном списке: $O(n)$.

4. Как удаление работает в каждой структуре.

4.1 Связный список

В связном списке удаление происходит последовательным проходом от головы списка. Сначала поиском доходим до удаляемого элемента, а затем "соединяем" предыдущий и следующий элементы, относительно удаляемого (то есть происходит перестановка ссылок - предыдущий элемент теперь указывает на следующий после удаляемого элемент). По сложности удаление в связном списке $O(n)$, потому что сначала нужно найти нужный узел.

4.2 Хеш-таблица

Для удаления элемента из хеш-таблице сначала вычисляется индекс его бакета, после чего удаление внутри бакета происходит также, как в связном списке. Но поскольку внутри бакетов лежит (в идеале) один элемент, то удаление происходит сразу и близко к $O(1)$

4.3 Бинарное дерево

Для бинарного дерева существует 3 случая удаления: элемент является листом (нет потомков), у элемента 1 потомок, у элемента 2 потомка.

Для всех трёх случаев вначале мы поиском ищем нужный элемент, который мы хотим удалить. Затем:

- а) для элемента без потомков** мы просто меняем ссылку его предка на None
- б) для элемента с одним потомком** мы меняем ссылку родителя удаляемого узла на его потомка (как бы перепрыгиваем через удаляемый элемент)
- в) для элемента с двумя потомками** мы производим замену удаляемого элемента на

самый левый элемент правого поддерева удаляемого элемента. Такой элемент, исходя из правила построения дерева "*Значение левого потомка всегда меньше родителя, правого — больше.*", всегда будет больше всех элементов из левого поддерева, но меньше всех элементов правого поддерева. То есть, он может встать на место удаляемого элемента и правило построения дерева не нарушится. Удаление в программе происходит следующим образом: сначала в удаляемый элемент копируются данные элемента, выбранного на замену, а затем этот лишний левый нижний узел удаляется как обычный узел с 0 или 1 потомком.

Если дерево сбалансировано (входные данные перемешаны), удаление работает примерно за $O(\log n)$. Если дерево выродилось Связный список (в случае отсортированных входных данных) удаление становится $O(n)$

Вывод

В результате работы были реализованы и сравнены три структуры данных: связный список, хеш-таблицы, бинарное дерево.

Для задач с частым поиском и удалением лучше всего, как показал эксперимент, подходит **Хеш-таблица**. Из построенных графиков и рассуждений выше видно, что для поиска, удаления хеш-таблице требуется меньше времени, чем другим структурам. Также хеш-таблице требуется кратно меньше времени на заполнение.

Для задач, в которых требуется получать данные в отсортированном порядке, то лучше всего подходит **Бинарное дерево**. Это связано с самим правилом заполнения дерева. При центрированном обходе элементы сразу выводятся по возрастанию ключей (в случае имен, как в нашей работе, в алфавитном порядке), без дополнительной сортировки. Однако, как показал эксперимент, время операций для Бинарного дерева сильно зависит от порядка подаваемых данных.

Связный список оказался наименее эффективным для поиска (и, соответственно, для удаления), потому что он требует последовательного просмотра элементов. Такая структура может быть полезна только в простых случаях, когда данных мало или когда важна простая последовательная организация элементов.

Таким образом для реальных задач лучше выбирать Хеш-таблицы или Сбалансированные деревья, в зависимости от поставленной задачи и порядка данных.