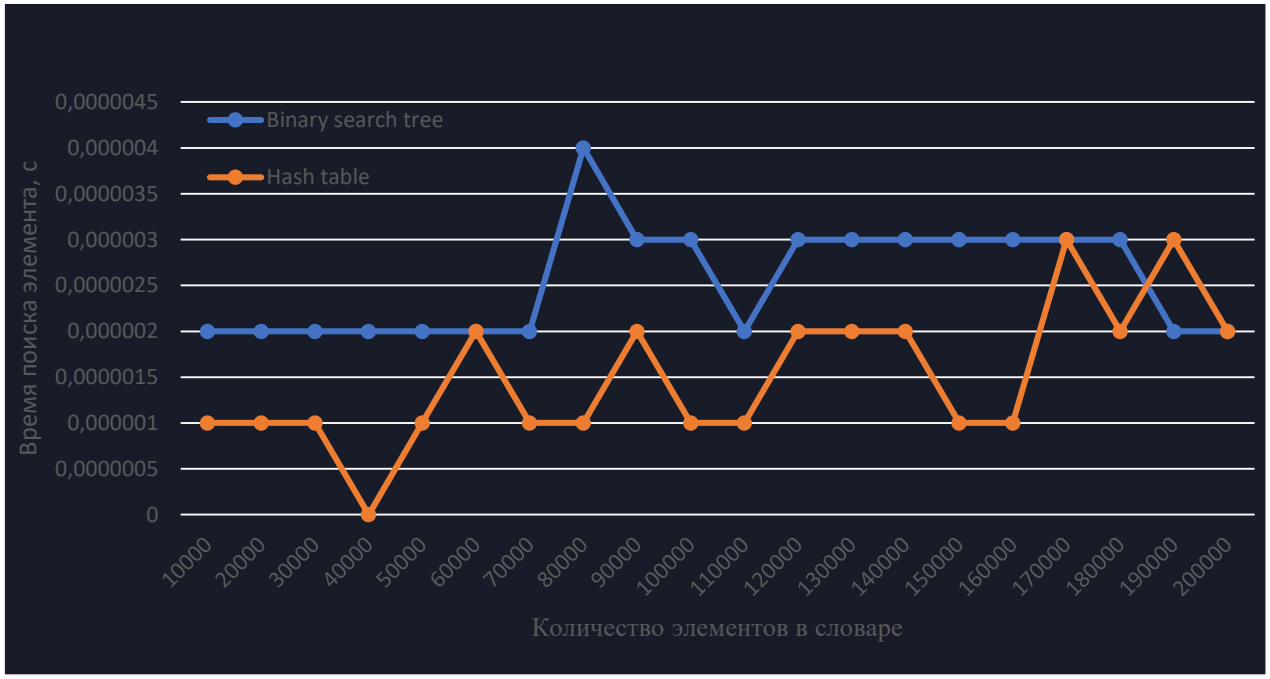
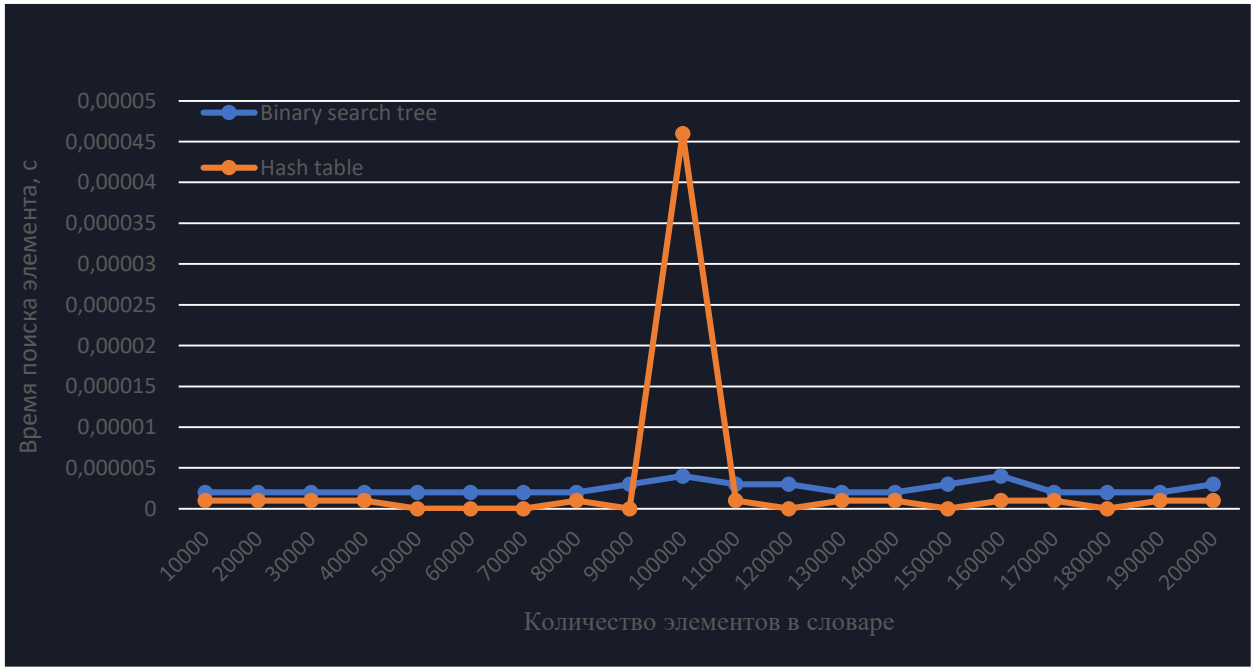
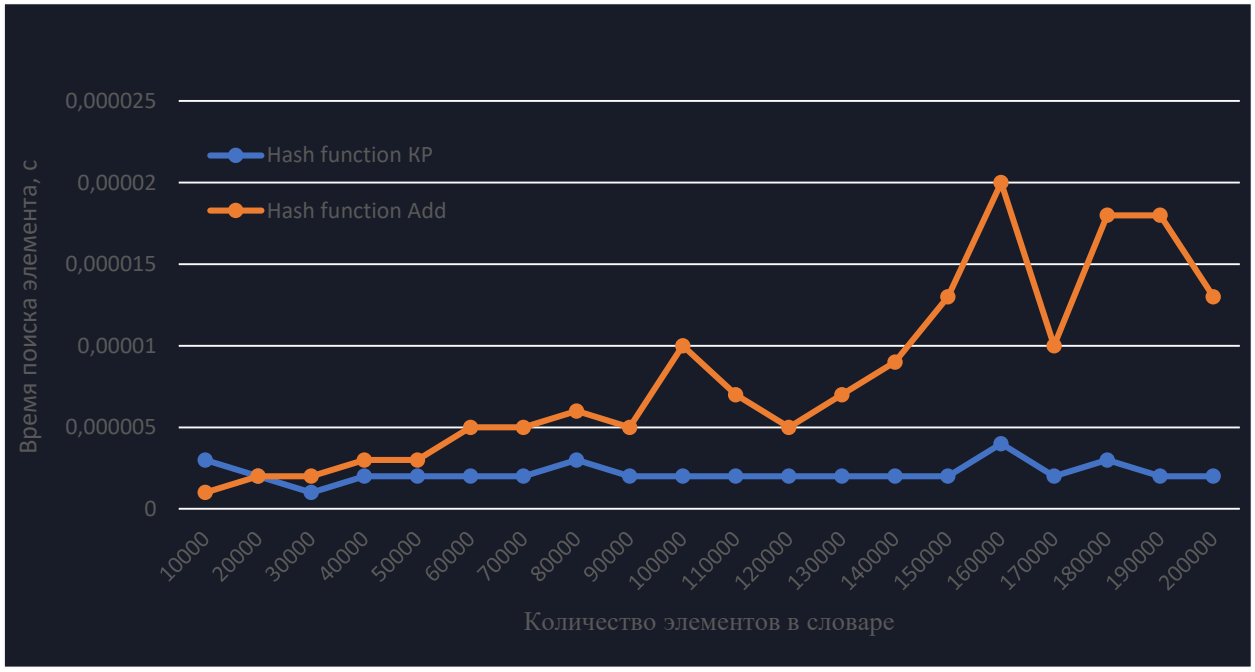
#	Количество элементов в словаре	Время выполнения функции bstree_lookup, с	Время выполнения функции hashtab_lookup, с
1	10000	0,000002	0,000001
2	20000	0,000002	0,000001
3	30000	0,000002	0,000001
4	40000	0,000002	0,000000
5	50000	0,000002	0,000001
6	60000	0,000002	0,000002
7	70000	0,000002	0,000001
8	80000	0,000004	0,000001
9	90000	0,000003	0,000002
10	100000	0,000003	0,000001
11	110000	0,000002	0,000001
12	120000	0,000003	0,000002
13	130000	0,000003	0,000002
14	140000	0,000003	0,000002
15	150000	0,000003	0,000001
16	160000	0,000003	0,000001
17	170000	0,000003	0,000003
18	180000	0,000003	0,000002
19	190000	0,000002	0,000003
20	200000	0,000002	0,000002

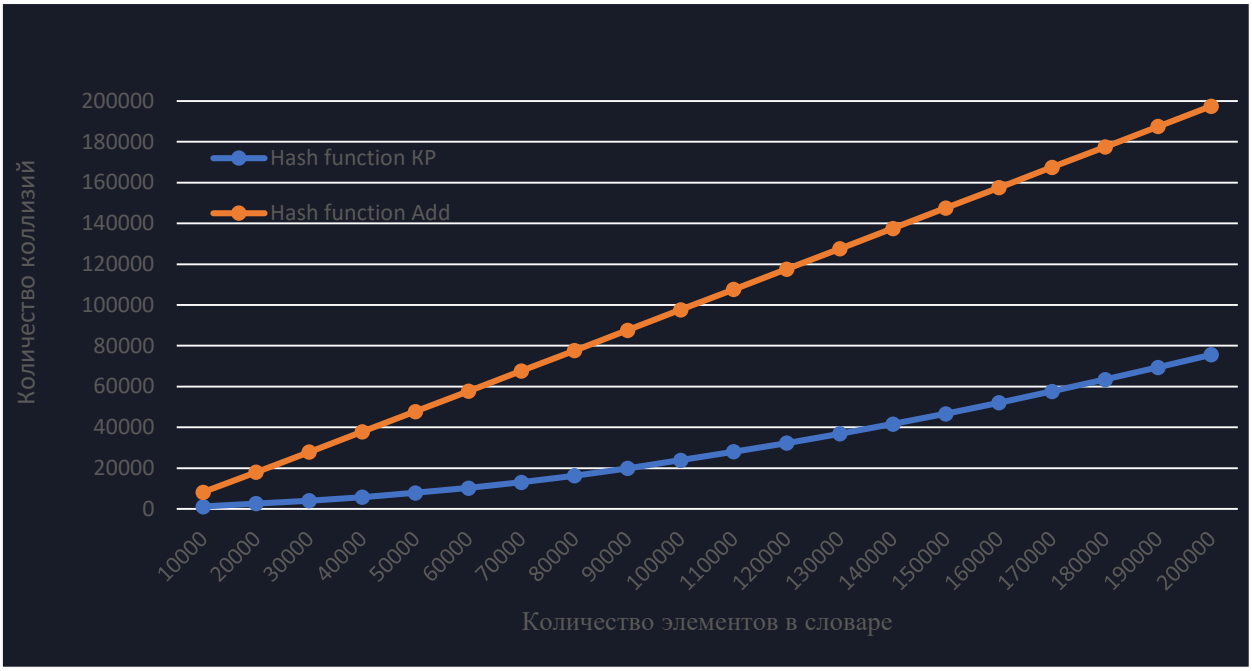


#	Количество элементов в словаре	Время выполнения функции bstree_add, с	Время выполнения функции hashtab_add, с
1	10000	0,000002	0,000001
2	20000	0,000002	0,000001
3	30000	0,000002	0,000001
4	40000	0,000002	0,000001
5	50000	0,000002	0,000000
6	60000	0,000002	0,000000
7	70000	0,000002	0,000000
8	80000	0,000002	0,000001
9	90000	0,000003	0,000000
10	100000	0,000004	0,000046
11	110000	0,000003	0,000001
12	120000	0,000003	0,000000
13	130000	0,000002	0,000001
14	140000	0,000002	0,000001
15	150000	0,000003	0,000000
16	160000	0,000004	0,000001
17	170000	0,000002	0,000001
18	180000	0,000002	0,000000
19	190000	0,000002	0,000001
20	200000	0,000003	0,000001



#	Количество элементов в словаре	Хеш-функция KP		Хеш-функция Add	
		Время выполнения функции hashtab_lookup, с	Число коллизий	Время выполнения функции hashtab_lookup, с	Число коллизий
1	10000	0,000003	1119	0,000001	8224
2	20000	0,000002	2634	0,000002	18021
3	30000	0,000001	4053	0,000002	27919
4	40000	0,000002	5754	0,000003	37859
5	50000	0,000002	7866	0,000003	47812
6	60000	0,000002	10282	0,000005	57774
7	70000	0,000002	13048	0,000005	67735
8	80000	0,000003	16308	0,000006	77707
9	90000	0,000002	19888	0000005	87682
10	100000	0,000002	23904	0,000010	97654
11	110000	0,000002	28075	0,000007	107629
12	120000	0,000002	32311	0,000005	117613
13	130000	0,000002	36878	0,000007	127602
14	140000	0,000002	41631	0,000009	137588
15	150000	0,000002	46694	0,000013	147580
16	160000	0,000004	52103	0,000020	157568
17	170000	0,000002	57684	0,000010	167553
18	180000	0,000003	63440	0,000018	177540
19	190000	0,000002	69442	0,000018	187529
20	200000	0,000002	75634	0,000013	197519





Контрольные вопросы

1. Ассоциативный массив (associative array, map) – это вид множества, элементами которого являются пары вида «(ключ, значение)». Ассоциативные массивы также называют словарями (dictionary). Поиск элемента в ассоциативном массиве осуществляется по ключу (key). Значение (value), ассоциированное с каждым ключом, может иметь тип данных, отличный от типа данных ключа.

Единственное отличие ассоциативных массивов от множеств в том, что множества хранят только ключи, а ассоциативные массивы с каждым ключом хранят и некоторое значение (данные из предметной области решаемой задачи).

2. Бинарное дерево (двоичное дерево, binary tree) – это корневое дерево, в котором каждый узел имеет не более двух дочерних вершин (0, 1 или 2 вершины).

Полное бинарное дерево (full binary tree) – это бинарное дерево, в котором каждый узел имеет 0 или 2 дочерних узла. Пример такого дерева приведен на рис. 9.1.

Завершенное бинарное дерево (complete binary tree) – это бинарное

дерево, в котором каждый уровень, возможно за исключением последнего, полностью заполнен узлами, а заполнение последнего уровня осуществляется слева направо (рис. 9.2).

Высота завершенного бинарного дерева из n узлов равна $\lfloor \log_2 n \rfloor$, а число вершин с высотой $h \in \{0, 1, \ldots, \lfloor \log_2 n \rfloor\}$ есть $\lceil n/2^{h+1} \rceil$.

Совершенное бинарное дерево (perfect binary tree) – это бинарное дерево, в котором все листья имеют одинаковую глубину (рис. 9.3).

При числе узлов n минимальной высотой $h = \log_2 (n + 1) - 1$ обладает совершенное бинарное дерево, а максимальной высотой – бинарное дерево, в котором каждый внутренний узел имеет единственную дочернюю вершину (дерево вырождается в цепочку высоты $n - 1$). Следовательно, высота h любого бинарного дерева из n узлов ограничена снизу и сверху $\lceil \log_2(n + 1) \rceil - 1 \le h \le n - 1$.

3. Хеш-таблица (hash table) – это массив $T[0..m - 1]$, в котором номер ячейки для ключа $key \in U$ вычисляется с использованием хеш-функции hash(key). Причем размер m хеш-таблицы значительно меньше размера исходной совокупности ключей U. Ячейки хеш-таблицы также называют слотами (slot, bucket).

4. Хеш-функция (hash function) принимает в качестве аргумента целочисленный ключ из множества U и возвращает соответствующий ему номер ячейки таблицы $T[0..m - 1]$. Другими словами, хеш-функция отображает совокупность ключей U на множество ячеек хеш-таблицы $T[0..m-1]$:

$$\text{hash(key)}: U \to \{0, 1, \ldots, m - 1\}.$$

Значение хеш-функции для ключа называют его хеш-кодом (hash code), хеш-значением (hash value) или просто хешем (hash).

Заметим, что в определении хеш-функции мы ограничились лишь ключами целочисленного типа. Далее мы рассмотрим способы преобра-

зования ключей других типов данных (строки и вещественные числа) в целочисленные ключи из фиксированного диапазона.

Из-за того, что $|U| > m$, некоторым ключам из U неизбежно будет соответствовать одна и та же ячейка таблицы $T[0..m-1]$. Такая ситуация, при которой для двух и более разных ключей хеш-функция возвращает одинаковый хеш-код, называется коллизией (collision). Формально

$$\forall key1, key2 \in U, key1 \neq key2 : hash(key1) = hash(key2).$$

Хорошая хеш-функция должна иметь следующие свойства:

Уникальность: хеш-функция должна генерировать уникальные хеш-коды для каждого набора входных данных.

Равномерность распределения: хеш-функция должна равномерно распределять значения хеш-кодов в зависимости от входных данных. Это означает, что изменение небольшой части входных данных должно приводить к изменению большой части хеш-кода.

Быстродействие: хеш-функция должна работать быстро, чтобы обеспечивать высокую производительность системы.

Хорошей хеш-функцией считается, например, SHA-256 (Secure Hash Algorithm 256-bit), которая широко используется для целей шифрования и безопасности данных в интернете. Она генерирует уникальные 256-битные хеш-коды для любых входных данных и имеет хорошую производительность.

5. Широкое распространение получили два основных метода разрешения коллизий (collision resolution): метод цепочек (chaining) и открытая адресация (open addressing). Задачей любого метода разрешения коллизий является формирование правил хранения в хеш-таблице ключей с одинаковыми хеш-кодами.

В методе цепочек каждая ячейка h хеш-таблицы $T[0..m-1]$ содержит указатель на голову связного списка, в который помещаются все ключи имеющие одинаковый хеш-код h. В каждом узле списка содержатся ключ и некоторое значение. Заметим, что хранение значений в узлах списков не является обязательным и необходимо только при реализации АТД ас-

социативный массив. При реализации неупорядоченных множеств в узлах списков достаточно хранить только ключи.

Основной недостаток метода цепочек – использование порядка Θ(m + n) ячеек памяти для хранения вспомогательных указателей связных списков. Основная идея открытой адресации – хранение ключей и ассоциированных с ними значений непосредственно в ячейках таблицы T[0..m − 1]. Такой подход позволяет более экономно использовать память.
Нетрудно заметить, что при использовании такой стратегии в хеш-таблице может храниться не более m ключей и, как следствие, значение коэффициента α = n/m не может превышать 1.
Коллизия (collision) — это совпадение значений хеш-функции для двух разных ключей.

Эффективность хеш-таблиц

- Хеш-таблица требует предварительной инициализации ячеек значениями NULL — трудоёмкость O(*h*)
- Ключ — это строка из *m* символов

Операция	Вычислительная сложность в среднем случае	Вычислительная сложность в худшем случае
Add(*key*, *value*)	*O*(*m*)	*O*(*m*)
Lookup(*key*)	*O*(*m* + *mn* / *h*)	*O*(*m* + *nm*)
Delete(*key*)	*O*(*m* + *mn* / *h*)	*O*(*m* + *nm*)
Min()	*O*(*m*(*n* + *h*))	*O*(*m*(*n* + *h*))
Max()	*O*(*m*(*n* + *h*))	*O*(*m*(*n* + *h*))

17

Хеш-таблицы vs. бинарные деревья поиска

- Эффективность реализации словаря хеш-таблицей (метод цепочек) и бинарным деревом поиска
- Ключ — это строка из m символов
- Оценка сложности для **худшего случая** (worst case):

Операция	Хеш-таблица (неупорядоченный словарь)	Бинарное дерево поиска (упорядоченный словарь)
Add(*key*, *value*)	$O(m)$	$O(nm)$
Lookup(*key*)	$O(m + nm)$	$O(nm)$
Delete(*key*)	$O(m + nm)$	$O(nm)$
Min()	$O(m(n + h))$	$O(n)$
Max()	$O(m(n + h))$	$O(n)$

Хеш-таблицы vs. бинарные деревья поиска

- Эффективность реализации словаря хеш-таблицей (метод цепочек) и бинарным деревом поиска
- Ключ — это строка из m символов
- Оценка сложности для **среднего случая** (average case):

Операция	Хеш-таблица (неупорядоченный словарь)	Бинарное дерево поиска (упорядоченный словарь)
Add(*key*, *value*)	$O(m)$	$O(m\log n)$
Lookup(*key*)	$O(m + mn / h)$	$O(m\log n)$
Delete(*key*)	$O(m + mn / h)$	$O(m\log n)$
Min()	$O(m(n + h))$	$O(\log n)$
Max()	$O(m(n + h))$	$O(\log n)$

АТД «Словарь» (dictionary)

Операция	Описание
Add(*map*, *key*, *value*)	Добавляет в словарь *map* пару (*key*, *value*)
Lookup(*map*, *key*)	Возвращает из словаря *map* значение *value*, ассоциированное с ключом *key*
Delete(*map*, *key*)	Удаляет из словаря *map* пару с ключом *key*
Min(*map*)	Возвращает из словаря *map* минимальный ключ
Max(*map*)	Возвращает из словаря *map* максимальный ключ

Реализация словаря на основе массива

Операция	Неупорядоченный массив	Упорядоченный массив
Add(*map*, *key*, *value*)	$O(1)$ (добавление в конец)	$O(n)$ (поиск позиции)
Lookup(*map*, *key*)	$O(n)$	$O(\log n)$ (бинарный поиск)
Delete(*map*, *key*)	$O(n)$ (поиск элемента и перенос последнего на место удаляемого)	$O(n)$ (перемещение элементов)
Min(*map*)	$O(n)$	$O(1)$ (элемент $a[1]$)
Max(*map*)	$O(n)$	$O(1)$ (элемент $a[n]$)

Реализация словаря на основе связного списка

Операция	Неупорядоченный список	Упорядоченный список
Add(*map, key, value*)	*O*(1) (добавление в начало)	*O*(*n*) (поиск позиции)
Lookup(*map, key*)	*O*(*n*)	*O*(*n*)
Delete(*map, key*)	*O*(*n*) (поиск элемента)	*O*(*n*) (поиск элемента)
Min(*map*)	*O*(*n*)	*O*(1)
Max(*map*)	*O*(*n*)	*O*(*n*) или *O*(1), если поддерживать указатель на последний элемент

6

Реализация словаря на основе бинарного дерева поиска

Операция	Средний случай (average case)	Худший случай (worst case)
Add(*map, key, value*)	*O*(log*n*)	*O*(*n*)
Lookup(*map, key*)	*O*(log*n*)	*O*(*n*)
Delete(*map, key*)	*O*(log*n*)	*O*(*n*)
Min(*map*)	*O*(log*n*)	*O*(*n*)
Max(*map*)	*O*(log*n*)	*O*(*n*)

30

КР, Add, XOR, FNV, Jenkins, ELF, DJB
DJB (также известна как djb2 или Dan Bernstein's hash) - это хеш-функция, разработанная Дэном Бернштейном, которая использует простые операции побитового сдвига, побитового ИЛИ (OR) и побитового И (AND) для комбинирования значений байтов во входных данных и возвращает результат в качестве хеш-значения.

КР (Kr) - это простая хеш-функция, которая суммирует коды символов в строке и возвращает сумму в качестве хеш-значения. Она основана на принципе сложения кодов символов в строке и может использоваться для простых случаев, где требуется простая и быстрая

хеш-функция, например, для хеширования небольших объемов данных.

Add (Additive) - это хеш-функция, которая суммирует значения байтов во входных данных и возвращает сумму по модулю определенного числа (обычно степени двойки) в качестве хеш-значения. Она основана на принципе сложения значений байтов и может использоваться для простых случаев, где требуется простая и быстрая хеш-функция.

XOR (Xclusive OR) - это хеш-функция, которая выполняет побитовое исключающее ИЛИ (XOR) операции над значениями байтов во входных данных и возвращает результат в качестве хеш-значения. Она основана на принципе побитового исключающего ИЛИ операций и может использоваться для простых случаев, где требуется простая и быстрая хеш-функция.

FNV (Fowler-Noll-Vo) - это хеш-функция, которая комбинирует значения байтов во входных данных с использованием простых операций побитового сдвига и побитового исключающего ИЛИ (XOR) и возвращает результат в качестве хеш-значения. Она широко используется в различных приложениях, таких как хеширование строк, хеширование данных в хеш-таблицах и т.д.

Jenkins - это хеш-функция, разработанная Бобом Дженкинсом, которая использует сложные операции побитового сдвига, побитового ИЛИ (OR), и побитового И (AND) для комбинирования значений байтов во входных данных и возвращает результат в качестве хеш-значения. Она обеспечивает хорошее распределение хеш-значений и высокую степень случайности, и широко используется в различных приложениях, таких как хеширование паролей, хеширование данных в хеш-таблицах и т.д.

ELF (Executable and Linkable Format) - это хеш-функция, Суть которой заключается в комбинации байтов имени символа с использованием взвешенной суммы (weighted sum) или побитового сдвига. Входные данные (имя символа) разбиваются на байты, которые затем комбинируются с использованием определенных весов (констант) или сдвигов. Затем полученное значение подвергается дополнительной обработке, такой как побитовые операции, для получения окончательного хеш-значения.

Выражение "m(n + h)" в контексте хеш-таблицы обычно используется для оценки сложности операций в хеш-таблице, где:

"m" - размер массива хеш-таблицы, то есть количество ячеек в массиве

"n" - количество элементов (ключей-значений) в хеш-таблице
"h" - среднее число коллизий (конфликтов) в хеш-таблице
Таким образом, "m(n + h)" означает, что сложность операции в хеш-таблице пропорциональна сумме количества элементов в таблице и среднего числа коллизий. Когда "n" и "h" малы по сравнению с "m", то сложность операций в хеш-таблице обычно считается низкой. Однако, если "n" и "h" значительно превышают "m", то это может привести к ухудшению производительности хеш-таблицы из-за увеличения числа коллизий и, как результат, увеличения времени выполнения операций.
https://drive.google.com/drive/folders/1mjztnabGpnIXbty2mmSuwTlHwtRNhReE