МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта
Базовая кафедра №252 – информационной безопасности

КУРСОВАЯ РАБОТА

По дисциплине
«Математическая логика и теория алгоритмов»

Тема курсовой работы
«Сложные структуры данных и их использование для решения алгоритмических задач»

Студент группы ККСО-03-19 Николенко В.О. _____
(подпись)

Руководитель курсовой работы _____
(подпись)

Работа представлена к защите «__» _____ 2022 г.

Допущен к защите «__» _____ 2022 г.

Москва – 2022

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

Алгоритмы - круто.

2. СЛОЖНЫЕ СТРУКТУРЫ ДАННЫХ

Простыми словами, **структура данных** - это контейнер, который хранит информацию в определенном виде. Из-за такой «компоновки» она может быть эффективной в одних операциях и неэффективной в других. Цель разработчика – выбрать из существующих структур оптимальный для конкретной задачи вариант.

Данные являются самой важной сущностью в информатике, а структуры позволяют хранить их в организованной форме.

Какую бы проблему ни решал инженер, ему приходится иметь дело с данными — будь то зарплата сотрудника, цены на акции, список покупок или даже простой телефонный справочник.

В зависимости от ситуации данные должны храниться в некотором определенном формате. Структуры данных предлагают несколько вариантов таких размещений.

Самые распространённые структуры данных:

1. Массив (Array)
2. Стек (Stack)
3. Очередь (Queue)
4. Связный список (Linked List)
5. Дерево (Tree)
6. Граф (Graph)
7. Префиксное дерево (Trie)
8. Куча
9. Хэш-Таблица (Hash Table)

В данной курсовой работе, мы рассмотрим структуры с 4 по 8 пункт включительно, так как все остальные незаурядны и в дополнительном обзоре не нуждаются. Плюс к этим 5 способам хранения информации я бы хотел добавить передовые технологии: дерево хешей (дерево Меркла). Дерево хешей - технологии реализованные в блокчейне, так что их рассмотрение будет наиболее полезным для понимания его работы.

2.1. Связный список

Первая структура данных, которую мы рассмотрим - **связный список**. На то есть две причины:

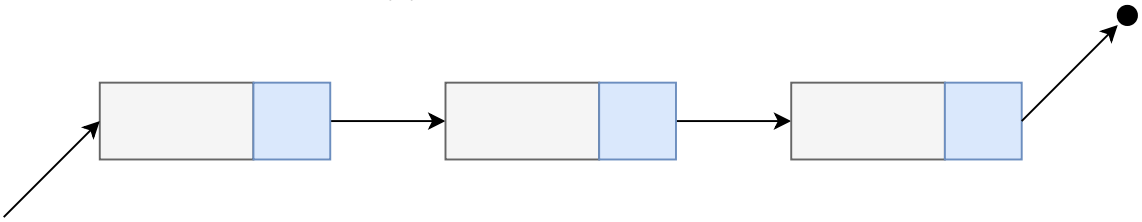
- связный список используется практически везде - от ОС до игр;

- на его основе строится множество других структур данных (тот же HashMap при возникновении коллизий).

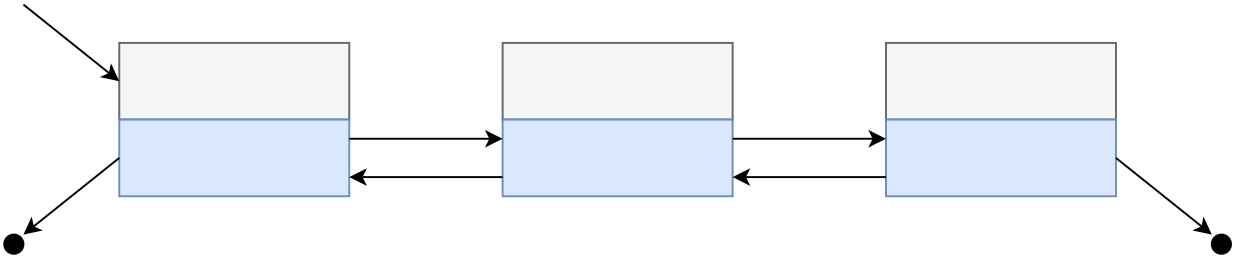
Существует две версии связного списка: односвязный и двусвязный.

Принципиальное отличие двусвязного от односвязного состоит в том, что во втором можно двигаться в двух направлениях, то есть узел (контейнер) указывает на последующий элемент и на предыдущий одновременно.

Односвязный список:



Двусвязный список:



Рассмотрим тогда сложность выполнения операций в связном списке:

Метод:	Добавление	Удаление элемента	Поиск	Удаление списка
Сложность:	O(n)	O(n)	O(n)	O(1)

• Операция добавления элемента в список - итерация до его конца (n эл-тов) и назначение указателя последнего элемента на новый контейнер.

• Операция поиска элемента - итерация по списку до того как мы найдём эл-т, если натыкаемся на конец списка - return false.

• Операция удаления элемента - выполнение операции поика и переназначение указателя предыдущего элемента на последующий контейнер, либо до конца списка (n эл-тов) и возвращаем false, т.к. элемент в списке отсутсвует.

• Операция удаления списка - переназначение указателей _head и _tail в ноль.

Можно дополнительно оптимизировать структуру и, допустим, хранить указатель на последний элемент, так мы сможем добиться ускорения добавления элемента в список до $O(1)$.

Исходя из таблицы, с такой структурой работать над большими объёмами данных будет просто невозможно. Так что если она и используется, то только в особых случаях.

2.2. Деревья (бинарное, AVL, RB tree)

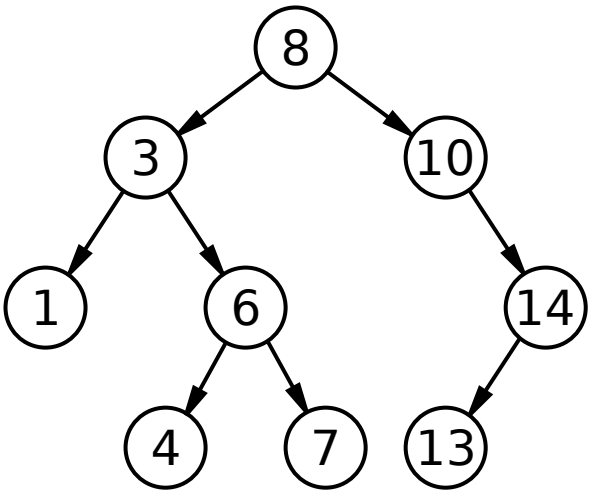
Дерево - одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы. Большинство источников также добавляют условие на то, что рёбра графа не должны быть ориентированными.

Бинарное дерево

Бинарное дерево - это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Оно обладает дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.

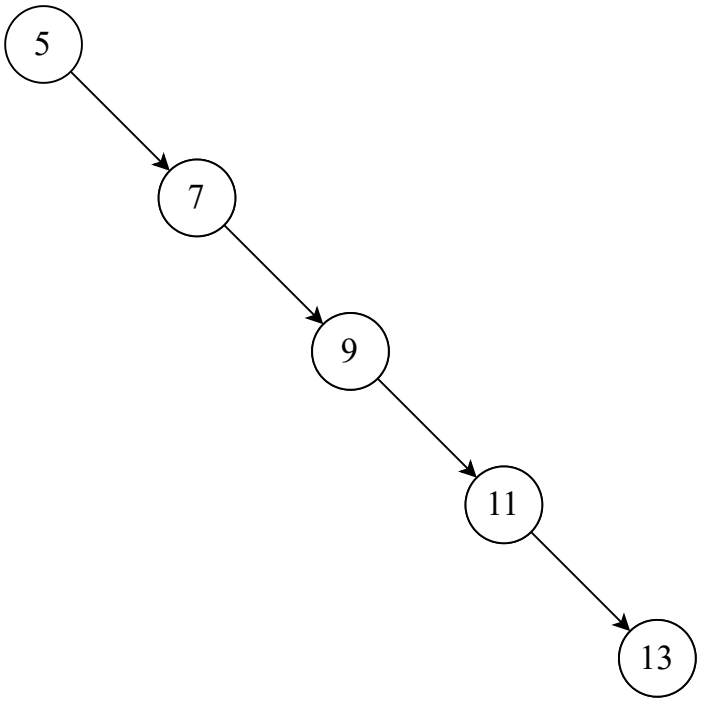
Бинарное дерево:



Рассмотрим тогда сложность выполнения операций в бинарном дереве:

Метод:	Добавление	Удаление эл-а	Поиск	Удаление дер-а
Сложность (ср.):	O(log n)	O(log n)	O(log n)	O(1)
Сложность (худ.):	O(n)	O(n)	O(n)	

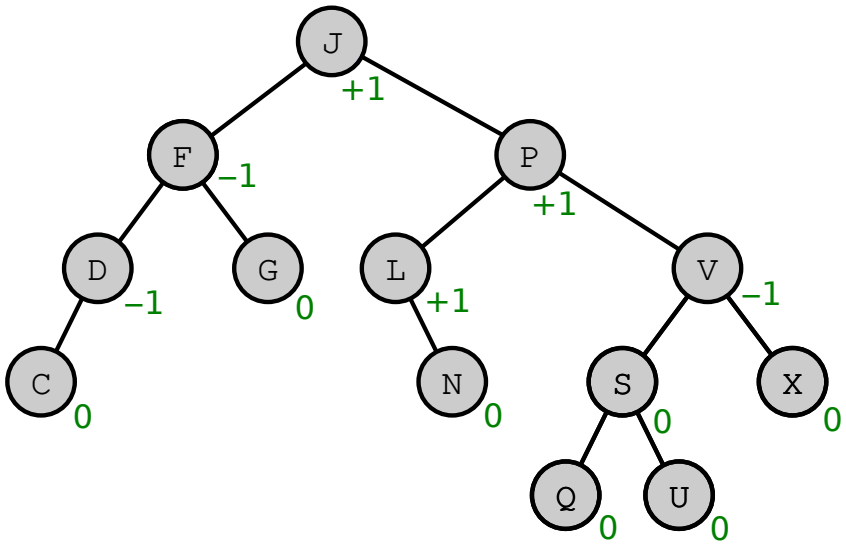
Сложность (худ.) здесь означает, что дерево будет перебрано полностью. Примером может служить экстремально несбалансированное дерево:



Это дерево тоже не очень нам подходит для работы с большими объёмами данных, хоть оно и даёт нам воможность осуществлять поиск / удаление / добавление за $O(log(n))$, но не гарантирует этого, худший вариант всё ещё имеет место быть.

AVL дерево

AVL дерево - самобалансирующееся бинарное дерево поиска. Тут исключён худший вариант, как в обычном бинарном дереве, т.к. для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.



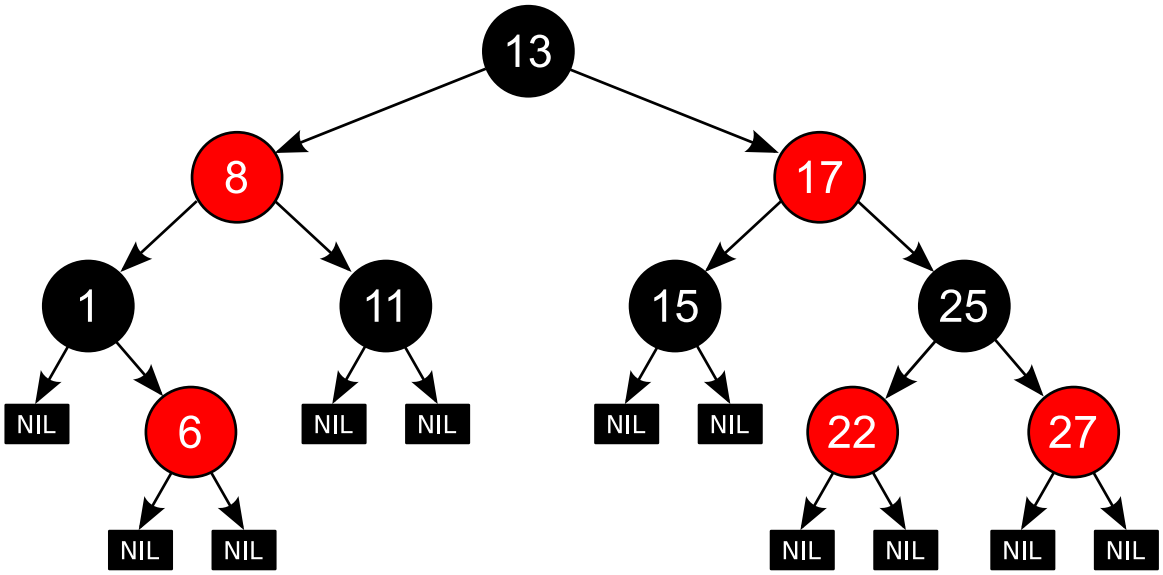
Из таблицы сложности операции уходит строка со сложностью в худшем варианте:

Метод:	Добавление	Удаление эл-а	Поиск	Удаление дер-а
Сложность (ср.):	O(log n)	O(log n)	O(log n)	O(1)

Однако, из-за сложности операций балансировки считается, что сбалансированные деревья следует использовать лишь в том случае, когда поиск информации происходит значительно чаще, чем добавление.

RB tree

Красно-Чёрное дерево (Red-Black tree) - двоичное дерево поиска, в котором баланс осуществляется на основе "цвета"узла дерева, который принимает только два значения: "красный"(англ. red) и "чёрный"(англ. black). При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными. Для экономии памяти фиктивные листья можно сделать одним общим фиктивным листом.



Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен дополнительный атрибут - цвет и для которого выполняются следующие свойства:

- Каждый узел промаркирован красным или чёрным цветом
- Корень и конечные узлы (листья) дерева — чёрные
- У красного узла родительский узел — чёрный
- Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
- Чёрный узел может иметь чёрного родителя

Таблица временной сложности будет аналогична таблице для AVL дерева:

Метод:	Добавление	Удаление эл-а	Поиск	Удаление дер-а
Сложность (ср.):	O(log n)	O(log n)	O(log n)	O(1)

В данном случае, как и с AVL деревом, производительность для всех операций является оптимальной, чтоб уже использовать данную структуру как хранилище для объёмных данных.

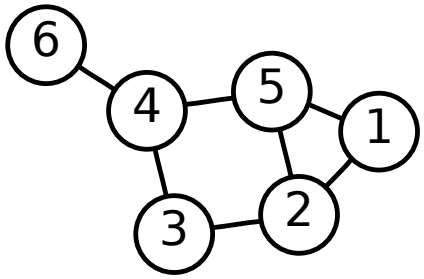
2.3. Граф

Структура данных **графа** представляет собой набор узлов, которые имеют данные и связаны с другими узлами.

Другими словами, граф - это структура данных (V, E), которая состоит из:

- коллекции вершин V.

- набора рёбер E, представленный в виде упорядоченных пар вершин (u, v).



Рассмотрим треминологию графа.

Говорят, что вершина **смежна** с другой вершиной, если есть ребро, соединяющее их.

Последовательность ребер, которая позволяет перейти от вершины A к вершине B, называется **путём**.

Граф, в котором есть ребро (u, v) не обязательно означает, что также имеется ребро (v, u). Граф называется **ориентированным**, если его рёбра представлены стрелками, чтобы показать направление перехода.

Наиболее распространенные операции над графами:

- проверьте, присутствует ли элемент в графе;

- обход графа;

- добавить элементы (вершины, ребра) в граф;

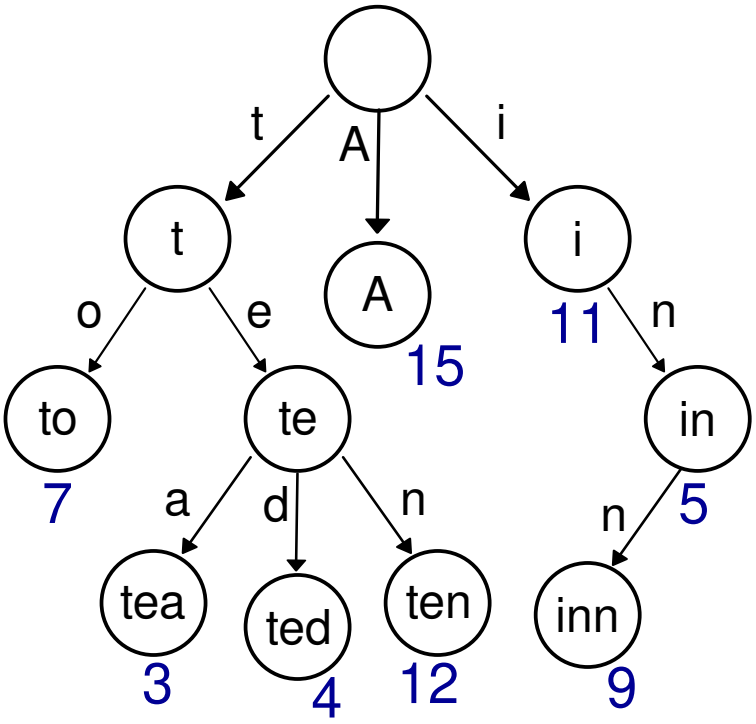
- нахождение пути от одной вершины к другой.

Т.к. граф как структура данных используется редко и сама по себе является довольно экзотической, так что рассматривать сложность операций поиска, вставки и прочих не имеет смысла.

2.4. Префиксное дерево

Префиксное дерево (иначе говоря нагруженное дерево или же trie) - структура данных реализующая интерфейс ассоциативного массива, то есть позволяющая хранить пары «ключ-значение». Сразу следует оговорится, что в большинстве случаев ключами выступают строки, однако в качестве ключей можно использовать любые типы данных, представимые как последовательность байт (то есть вообще любые).

Нагруженное дерево отличается от обычных n-арных деревьев тем, что в его узлах не хранятся ключи. Вместо них в узлах хранятся односимвольные метки, а ключем, который соответствует некоему узлу является путь от корня дерева до этого узла, а точнее строка составленная из меток узлов, повстречавшихся на этом пути. В таком случае корень дерева, очевидно, соответствует пустому ключу.



Нагруженное дерево по показателям потребления памяти/процессорного времени не уступает хэш-таблицам и сбалансированным деревьям, а иногда и превосходит их по этим параметрам. Если обратить внимание на таблицу ниже, можно заметить, что в случае, когда $log(n) > |key|$, операции совершённые в префиксном дереве займут меньше времени, чем в том же AVL дереве.

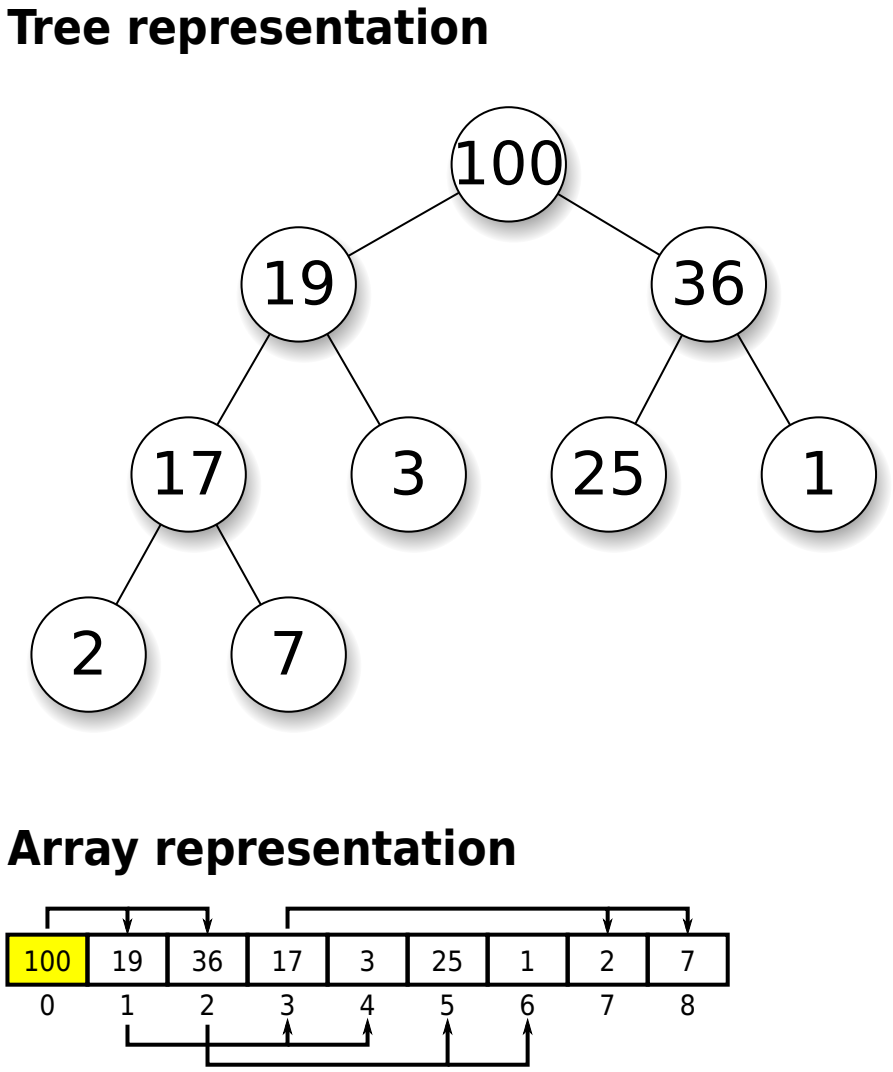
Метод:	Добавление	Удаление эл-а	Поиск	Удаление дер-а
Сложность (ср.):	O(|Key|)	O(|Key|)	O(|Key|)	O(1)

По потреблению памяти нагруженное дерево часто выигрывает у хэш-таблиц и сбалансированных деревьев. Это связано с тем что у множества ключей в нагруженном дереве совпадают префиксы, и вместе с ними память которую они используют. Также, в отличии от сбалансированных деревьев, в нагруженном дереве нет необходимости хранить ключ в каждом узле.

2.5. Куча

Куча - полное бинарное дерево, для которого выполняется основное свойство кучи: приоритет каждой вершины больше приоритетов её потомков. В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется max-heap, поскольку корень поддерева является максимумом из значений элементов поддерева.

Дерево называется полным бинарным, если у каждой вершины есть не более двух потомков, а заполнение уровней вершин идет сверху вниз (в пределах одного уровня – слева направо).



Двоичную кучу удобно хранить в виде одномерного массива, причем левый потомок вершины с индексом i имеет индекс $2 \cdot i + 1$, а правый $2 \cdot i + 2$. Корень дерева - элемент с индексом 0. Высота двоичной кучи равна высоте дерева, то есть $log_2 N$, где N - количество элементов массива.

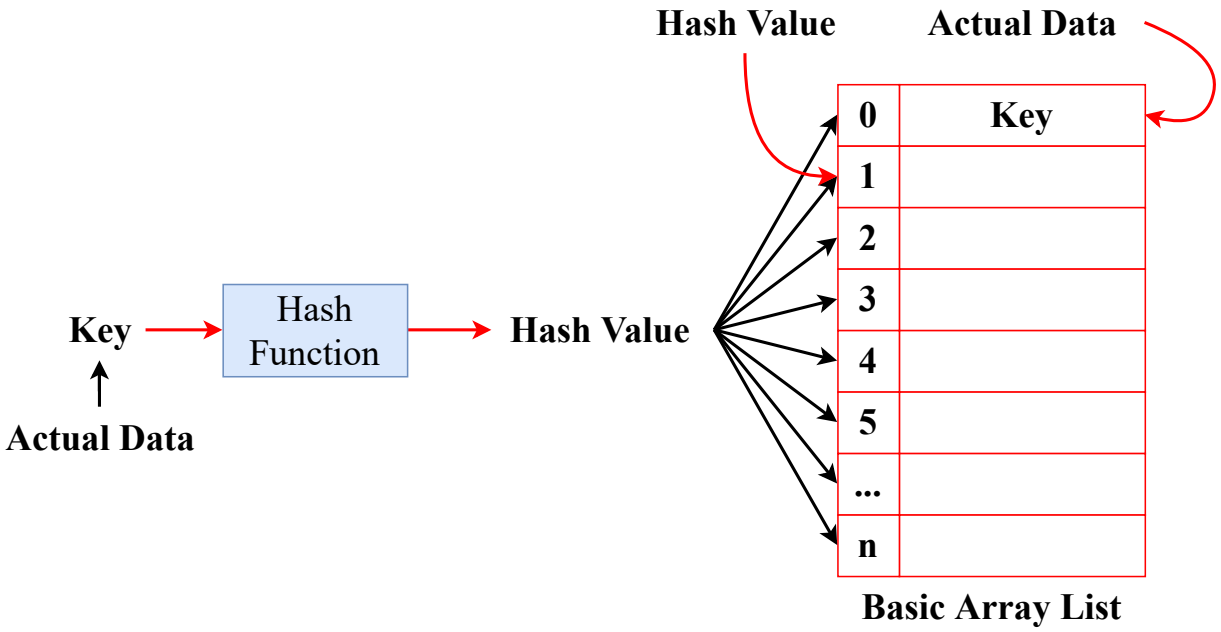
Метод:	Добавление	Удаление эл-а	Поиск	Удаление дер-а
Сложность (ср.):	O(log n)	O(log n)	O(log n)	O(1)

2.6. Хэш-Таблица

Хеш-таблица — это контейнер, который используют, если хотят быстро выполнять операции вставки/удаления/нахождения.

Любая операция внутри хеш-таблицы начинается с того, что ключ каким-то образом преобразуется в индекс обычного массива. Для получения индекса из ключа нужно выполнить два действия: найти хеш (хешировать ключ) и

привести его к индексу (например, через остаток от деления).



Хеширование — операция, которая преобразует любые входные данные в строку (реже число) фиксированной длины. Функция, реализующая алгоритм преобразования, называется "хеш-функцией а результат называют "хешем"или "хеш-суммой". Наиболее известны CRC32, MD5 и SHA (много разновидностей).

Метод:	Добавление	Удаление эл-а	Поиск	Удаление табл.
Сложность (ср.):	O(1)	O(1)	O(1)	O(1)

Однако, хэш-таблица очень требовательна к памяти. А так же существует слабое место - коллизии.

Ключом в ассоциативном массиве может быть абсолютно любая строка (любой длины и содержания). Другими словами, множество всех возможных ключей — бесконечно. В свою очередь, результат работы хеш-функции — строка фиксированной длины, а значит множество всех выходных значений — конечно.

Из этого факта следует, что не для всех входных данных найдётся уникальный хеш. На каком-то этапе возможно появление дублей (где под одним хешем лежит несколько разных значений — как если бы под одним индексом в массиве лежало два разных элемента). Такую ситуацию принято называть коллизией. Есть несколько способов разрешения коллизий (открытая адресация, метод цепочек), и каждому из них соответствует свой тип хеш-таблицы.

2.7. Дерево Меркла

Дерево Меркла - особый вид бинарного дерева, которое строится снизу вверх. А именно, пускай у нас имеется массив неких значений (не важно каких и в каком виде они представлены).

Тогда на первом шаге мы применим функцию хэширования к каждому элементу по отдельности:

$$Hash_{00} = hash(L_1),\ Hash_{01} = hash(L_2), \ldots$$

Так мы создали нижний уровень листов нашего дерева.

На втором и последующих шагах, блоки, находящиеся уровнем выше, заполняются как хеши от суммы их детей:

$$Hash_0 = Hash_{00} + Hash_{01},\ Hash_1 = Hash_{10} + Hash_{11}$$

+ - конкатенация хэшей, обычно в ethereum для этого используется abi.encode ф-ия.

Таким образом мы получаем верхушку дерева Меркла - Root Hash, который в последствии испольуется для проверки корректности данных введённых пользователем. В качестве хэш функции может использоваться как $keccak256$, так и $SHA256$.

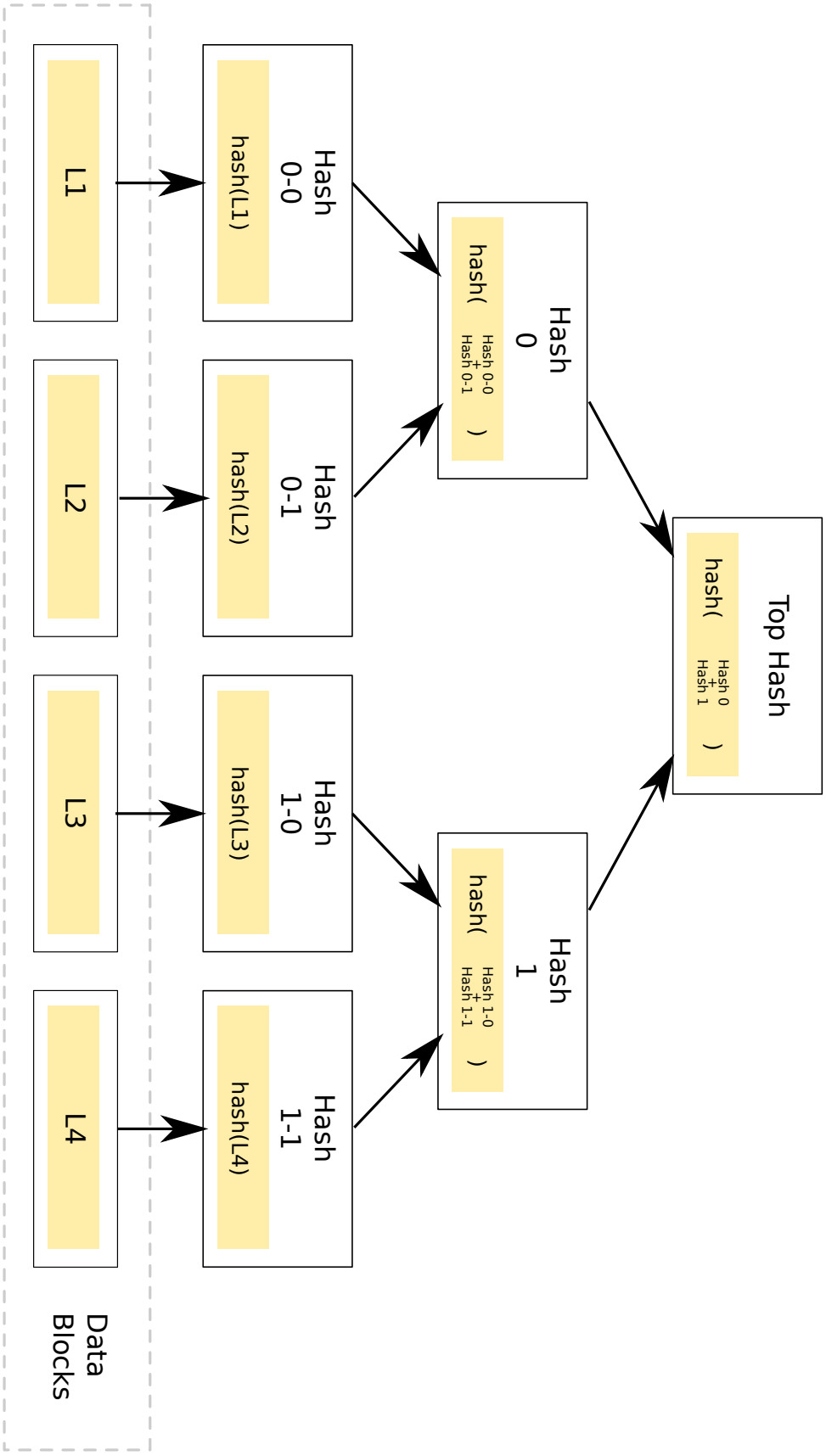
Хеш-деревья имеют преимущество перед хеш-цепочками или хеш-функциями При использовании хеш-деревьев гораздо менее затратным является доказательство принадлежности определённого блока данных к множеству. Поскольку различными блоками часто являются независимые данные, такие как транзакции или части файлов, то нас интересует возможность проверить только один блок, не пересчитывая хеши для остальных узлов дерева.

Пример работы дерва на практике:

Пускай мы хоти доказать что транзакция, содержащаяся в L_3 - наша. Тогда мы должны предоставить сети некий proof, который будет массивом хешей: $[Hash_{11}, Hash_0]$. Тогда путём работы алгоритма полученный результат:

$$Hash(Hash(Hash(L_3) + Hash_{11}) + Hash_0)$$

мы сверяем с root нашего исходного дерева, и в случае совпадения - мы доказали, что являемся владельцем этой транзакции и можем претендовать на какую-то монету, допустим.



В общем случае можно записать формулу проверки так:

$$signature(L) = (L \mid auth_{L_1}, \ldots, auth_{L_{K-1}}),$$

а проверку осуществить как $TopHash = Hash_K$

$$\begin{cases} hash(L), & if\ k = 1 \\ hash(Hash_{k-1} + auth_{L_{k-1}}), & if\ 2 \leq k \leq K \end{cases}$$

Количество транзакций	16	512	2048	65535
Приблизительный размер блока	4 кб	128 кб	512 кб	16 мб
Размер пути (в хешах)	4	9	11	16
Размер пути (в байтах)	128	288	352	512

Таким образом, проверку можно выполнить за $O(K) = log_2 N$, где K - высота дерева.

3. АЛГОРИТМЫ НА СТРУКТУРАХ ДАННЫХ

Алгоритмы - круто! Ведь правда..?

3.1. Поиск высоты дерева

Одна из распространённых алгоритмических задачек - поиск минимальной и максимальной высоты дерева (допустим бинарного). Тут мы вспоминаем, что дерево, какое бы оно ни было, - тоже граф, так что к нему применимы алгоритмы на графах. В данном случае интересен поиск в ширину, который и поможет найти высоты дерева и выяснить, например, достаточно ли дерево сбалансировано.

Поиск в ширину, алгоритм:

1. Поместить узел u, с которого начинается поиск (в нашем случае корень дерева), в изначально пустую очередь.

2. Извлечь из начала очереди узел u и пометить его как посещённый и записать шаг на котором мы его посетили (расстояние от корня до узла).

• Если узел u является целевым узлом, то завершить поиск с результатом «успех».

• В противном случае, в конец очереди добавляются все потомки узла u, которые ещё не были посещены и не находятся в очереди.

3. Если очередь пуста, то все узлы связного графа были просмотрены, следовательно, целевой узел недостижим из начального; завершить поиск с результатом «неудача» (в случае с бинарным деревом такое недостижимо).

4. Вернуться к пункту №2.

3.2. Поиск в глубину

Применяется:

- В качестве подпрограммы в алгоритмах поиска одно- и двусвязных компонент;

- В топологической сортировке;

- Для поиска точек сочленения, мостов;

- Для преобразования синтаксического дерева в строку (любую: префиксную, инфиксную, обратную польскую).

Алгоритм:

1. Пройдём по всем вершинам $v \in V$.

• Если вершина v белая, выполним для неё $DFS(v)$.

Процедура DFS (параметр - вершина $u \in V$):

1. Перекрашиваем вершину u u в серый цвет.

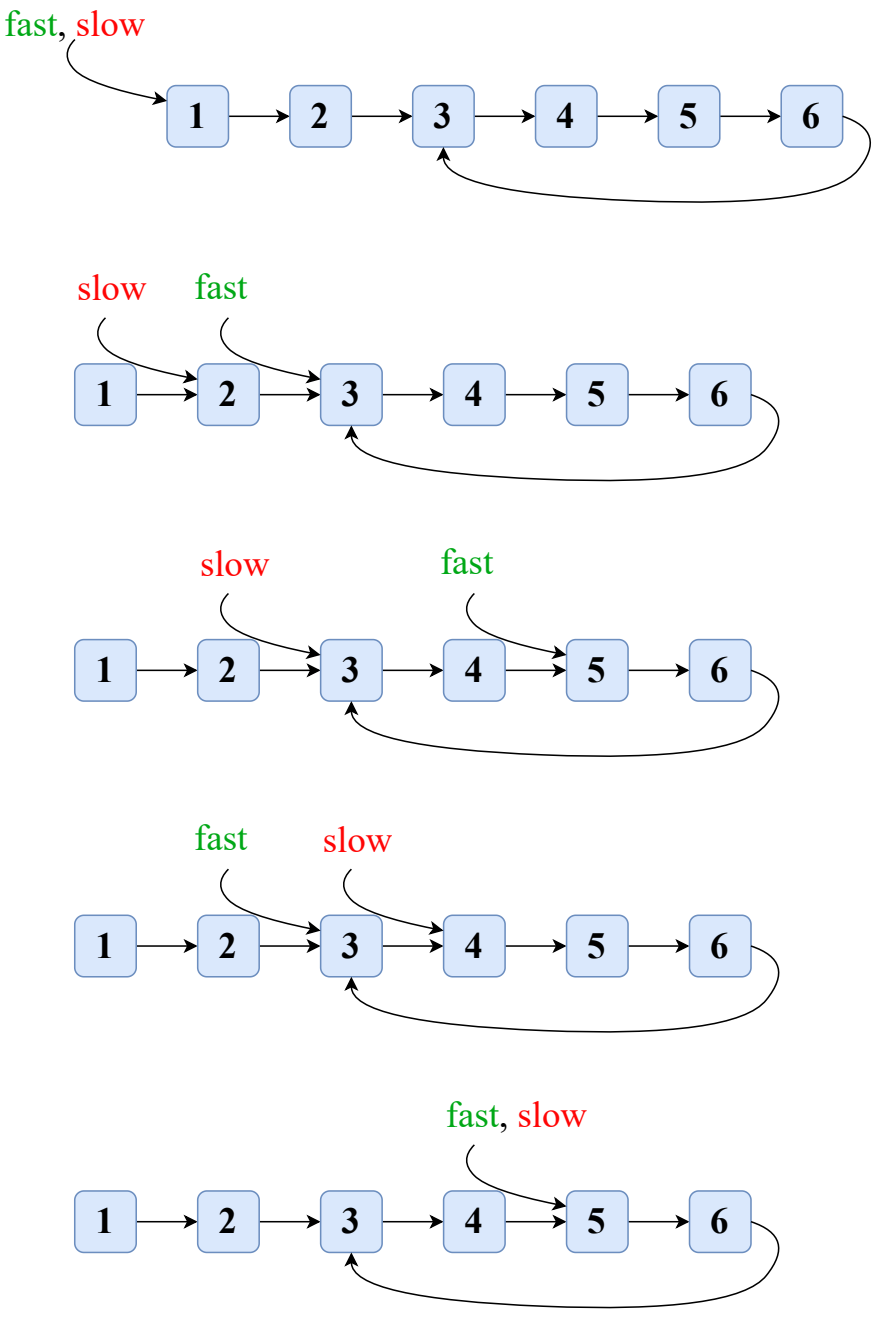
2. Для всякой вершины w, смежной с вершиной u и окрашенной в белый цвет, рекурсивно выполняем процедуру $DFS(w)$.

3. Перекрашиваем вершину u в чёрный цвет.

Часто используют двухцветные метки — без серого, на 1-м шаге красят сразу в чёрный цвет.

3.3. Нахождение цикла

Контекст: Еще этот алгоритм называют алгоритмом черепахи и зайца. Тут два указателя движутся с разной скоростью. Алгоритм применяется к односвязному списку. Первый указатель движется с шагом 1, второй с шагом в 2, и если они встретились, значит цикл существует.



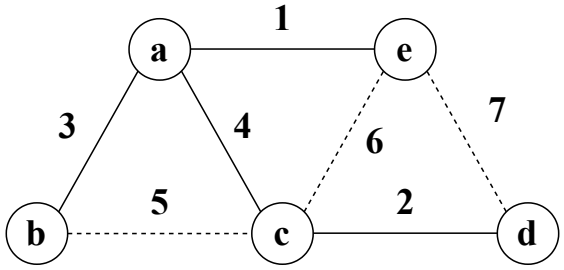
Алгоритм Прима

Алгоритм использует граф и предназначен для поиска минимального **остовного дерева**. Искомый минимальный остов строится постепенно, добавлением в него рёбер по одному. Остовное дерево строится путём отстраивания рёбер минимальных весов до того момента, пока последняя вершина не будет соеденина с деревом.

Пускай у нас есть некая вершина, из которой мы и начинаем построение дерева. Мы смотрим все исходящие рёбра из этой вершины и выбираем наименьшее, занося остальные веса рёбер в массив. Так же смотрим следующую вершину и добавляем все веса рёбер в массив. Выбираем ребро с наименьшим весом и вытаскиваем его из массива (обязательно убедившись в том, что ребро соединяет нас с НЕпосещённой вершиной дерева). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины (или, что то же самое, $n-1$ ребро).

В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связен, то остов найден не будет (количество выбранных рёбер останется меньше $n-1$).

Граф после алгоритма Прима:



4. ЗАКЛЮЧЕНИЕ

5. СПИСОК ЛИТЕРАТУРЫ