МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Тульский государственный университет»

Интернет-институт

КОНТРОЛЬНАЯ РАБОТА

по дисциплине

«БАЗЫ ДАННЫХ 1»

на тему

«ОСНОВНЫЕ ТИПЫ СТРУКТУР ДАННЫХ»

Семестр 5

Вариант 3

Выполнил: студент гр. ИБ262521-ф

Артемов Александр Евгеньевич

Проверил: канд. техн. наук, доц.

Французова Юлия Вячеславовна

Тула 2024

Оглавление

Введение

Структура данных — это способ организации информации для более эффективного использования. В программировании структурой обычно называют набор данных, связанных определённым образом. Например, массив — это структура.

Со структурой можно работать: добавлять данные, извлекать их и обрабатывать, например изменять, анализировать, сортировать. Для каждой структуры данных — свои алгоритмы. Работа программиста — правильно выбирать уже написанные готовые либо писать свои.

Главное свойство структур данных в том, что у любой единицы данных должно быть чёткое место, по которому её можно найти. Как определяется это место и как происходит поиск, зависит от конкретной структуры.

Характеристики структур данных следующие:

- данные в памяти представлены определённым образом, который однозначно позволяет определить структуру;
- чаще всего внутрь структуры можно добавить элемент или извлечь оттуда. Это свойство не постоянное — бывают структуры, которые нельзя изменять после создания;
- существуют алгоритмы, которые позволяют взаимодействовать с этой структурой.

При этом данных необязательно должно быть много. Массив из одного элемента — уже структура данных.

Структуры нужны, чтобы упорядочивать, искать, анализировать и использовать данные с применением алгоритмов программирования.

Фактически использование структур данных в программировании начинается ещё с задания переменной. Формат переменной — определённая структура данных, так как в память переменная записывается конкретным способом. Но на практике программисты работают с другими структурами, которые объединяют в себе разные переменные и типы данных.

Основные типы структур данных

Определение структуры данных

Структура данных (англ. data structure) — программная единица, позволяющая хранить и обрабатывать (машиной) однотипные и/или логически связанные данные. Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс.

Термин «структура данных» может иметь несколько близких, но тем не менее различных значений:

- Абстрактный тип данных (это математическая модель для типов данных, где тип данных определяется поведением (семантикой) с точки зрения пользователя данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций);
- Реализация какого-либо абстрактного типа данных;
- Экземпляр типа данных, например, конкретный список;
- В контексте функционального программирования — уникальная единица (англ. unique identity), сохраняющаяся при изменениях. О ней неформально говорят как об одной структуре данных, несмотря на возможное наличие различных версий.

Структуры данных формируются с помощью примитивных типов данных, ссылок и операций над ними в выбранном языке программирования. Следует понимать, что в каждом языке программирования имеется собственный набор примитивных типов и модель работы с памятью. Например, в C++ данные могут храниться как на стеке, так и в «куче», из-за чего данные имеют различный жизненный цикл.

Различные виды структур данных подходят для различных приложений; некоторые из них имеют узкую специализацию для определённых задач. Например, B-деревья обычно подходят для создания баз данных, в то время как хеш-таблицы используются повсеместно для создания различного рода словарей, например, для отображения доменных имён в интернет-адресах компьютеров.

При разработке программного обеспечения сложность реализации и качество работы программ существенно зависят от правильного выбора структур данных. Это понимание дало начало формальным методам разработки и языкам программирования, в которых именно структуры данных, а не алгоритмы, ставятся во главу архитектуры программного средства. Большая часть таких языков обладает определённым типом модульности, позволяющим структурам данных безопасно переиспользоваться в различных приложениях. Объектно-ориентированные языки, такие как Java, C# и C++, являются примерами такого подхода.

Многие классические структуры данных представлены в стандартных библиотеках языков программирования или непосредственно встроены в языки программирования. Например, структура данных хеш-таблица встроена в языки программирования Lua, Perl, Python, Ruby, Tcl и др. Широко используется стандартная библиотека шаблонов (STL) языка C++.

Фундаментальными строительными блоками для большей части структур данных являются массивы, записи (struct в Си и record в Паскале), размеченные объединения (union в Си) и ссылки. Например, двусвязный список может быть построен с помощью записей и ссылок, где каждая запись (узел) будет хранить данные и ссылки на «левый» и «правый» узлы.

Описание структуры данных можно условно поделить на интерфейс и реализацию. Интерфейс — это набор операций, который описывает поведение конкретной структуры. То есть это то, что можно сделать с конкретной структурой данных. А реализация — то, как данные структурированы внутри.

Кроме того, типичная структура данных отвечает нескольким критериям:

– корректность, то есть грамотная реализация своего технического описания;

– пространственная сложность — реализация должна занимать минимум места;

– временная сложность — операции должны выполняться за минимум времени.

«Минимум места» и «минимум времени» могут различаться для разных структур. Это нормально: у каждой из них свое назначение и оптимальный способ использования.

Структуры данных в общем можно разделить на линейные и нелинейные. Разница в том, как они хранят свои элементы — данные, которые в них расположены.

В линейных структурах элементы расположены как бы «по цепочке», друг за другом. Они выстраиваются в последовательность. Например, слово можно представить как линейную структуру данных, где данные — буквы.

В нелинейных элементы могут ветвиться, образовывать таблицы или схемы. Пример нелинейной структуры данных из жизни — схема дорог.

Массив.

Массив — структура данных, хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов, принимающих целые (или приводимые к целым) значения из некоторого заданного непрерывного диапазона. Одномерный массив можно рассматривать как реализацию абстрактного типа данных — вектор. Двумерный массив может называться также таблица, ряд, вектор, матрица.

Размерность массива — это количество индексов, необходимое для однозначной адресации элемента в рамках массива. По количеству используемых индексов массивы делятся на одномерные, двумерные, трёхмерные и т.д.

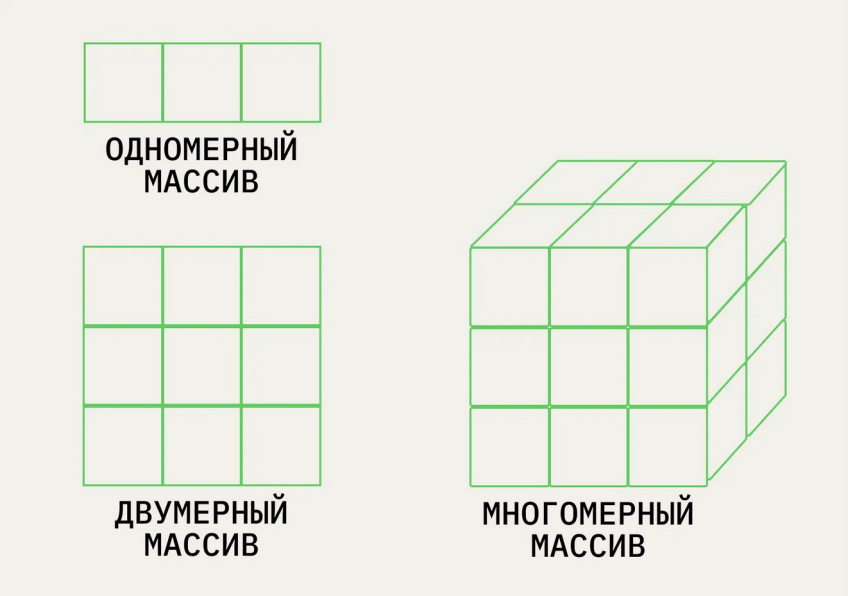

Рисунок 1: Размерность массива

Форма или структура массива — сведения о количестве размерностей и размере (протяжённости) массива по каждой из размерностей, может быть представлена одномерным массивом.

Особенностью массива как структуры данных (в отличие, например, от связного списка) является константная вычислительная сложность доступа к элементу массива по индексу. Массив относится к структурам данных с произвольным доступом.

В информатике под произвольным доступом (также называемым случайным доступом, англ. random access) понимают возможность обратиться к любому элементу последовательности за равные промежутки времени, не зависящие от размеров последовательности (в отличие от последовательного доступа, когда чем дальше расположен элемент, тем больше требуется времени для доступа).

В простейшем случае массив имеет константную длину по всем размерностям и может хранить данные только одного, заданного при описании, типа. Ряд языков поддерживает также динамические массивы, длина которых может изменяться во время выполнения программы, и гетерогенные массивы, которые могут в разных элементах хранить данные различных типов. Некоторые специфичные типы массивов, используемые в различных языках и реализациях — ассоциативный массив, дерево отрезков, V-список, параллельный массив, разреженный массив.

Основные достоинства использования массивов — лёгкость вычисления адреса элемента по его индексу (поскольку элементы массива располагаются один за другим), одинаковое время доступа ко всем элементам, малый размер элементов (они состоят только из информационного поля). Среди недостатков — невозможность удаления или добавления элемента без сдвига других при использовании статических массивов, а при использовании динамических и гетерогенных массивов — более низкое быстродействие из-за накладных расходов на поддержку динамики и разнородности. При работе с массивами с реализацией по типу языка C (с указателями) и отсутствии дополнительных средств контроля типичной ошибкой времени выполнения является угроза выхода за границы массива и повреждения данных.

Варианты реализации

Массив — упорядоченный набор элементов, каждый из которых хранит одно значение, идентифицируемое с помощью одного или нескольких индексов. В простейшем случае массив имеет постоянную длину и хранит единицы данных одного и того же типа, а в качестве индексов выступают целые числа.

Количество используемых индексов массива может быть различным: массивы с одним индексом называют одномерными, с двумя — двумерными, и так далее. Одномерный массив — нестрого соответствует вектору в математике; двумерный («строка», «столбец») — матрице. Чаще всего применяются массивы с одним или двумя индексами; реже — с тремя; ещё большее количество индексов — встречается крайне редко.

Первый элемент массива, в зависимости от языка программирования, может иметь различный индекс. Различают три основных разновидности массивов: с отсчётом от нуля (zero-based), с отсчётом от единицы (one-based) и с отсчётом от специфического значения заданного программистом (n-based). Отсчёт от нуля более характерен для низкоуровневых языков программирования, хотя встречается и в языках высокого уровня, например, используется почти во всех языках семейства Си. В ряде языков (Паскаль, Ада, Модула-2) диапазон индексов может определяться как произвольный диапазон значений любого типа данных, приводимого к целому, то есть целых чисел, символов, перечислений, даже логического типа (в последнем случае массив имеет два элемента, индексируемых значениями «Истина» и «Ложь»).

Поддержка индексных массивов (свой синтаксис объявления, функции для работы с элементами и так далее) есть в большинстве высокоуровневых языков программирования. Максимально допустимая размерность массива, типы и диапазоны значений индексов, ограничения на типы элементов определяются языком программирования или конкретным транслятором.

В языках программирования, допускающих объявления программистом собственных типов, как правило, существует возможность создания типа «массив». В определении такого типа задаются типы и/или диапазоны значений каждого из индексов и тип элементов массива. Объявленный тип в дальнейшем может использоваться для определения переменных, формальных параметров и возвращаемых значений функций. Некоторые языки поддерживают для переменных-массивов операции присваивания (когда одной операцией всем элементам массива присваиваются значения соответствующих элементов другого массива).

Динамические массивы

Динамическими называются массивы, размер которых может изменяться во время выполнения программы. Обычные (не динамические) массивы называют ещё фиксированными или статическими.

Динамические массивы могут реализовываться как на уровне языка программирования, так и на уровне системных библиотек. Во втором случае динамический массив представляет собой объект стандартной библиотеки, и все операции с ним реализуются в рамках той же библиотеки. Так или иначе, поддержка динамических массивов предполагает наличие следующих возможностей:

1. Описание динамического массива. На уровне языка это может быть специальная синтаксическая конструкция, на уровне библиотеки — библиотечный тип данных, значение которого объявляется стандартным образом. Как правило, при описании (создании) динамического массива указывается его начальный размер, хотя это и не обязательно.
2. Операция определения текущего размера динамического массива.
3. Операция изменения размера динамического массива.

Гетерогенные массивы

Гетерогенным называется массив, в разные элементы которого могут быть непосредственно записаны значения, относящиеся к различным типам данных. Массив, хранящий указатели на значения различных типов, не является гетерогенным, так как собственно хранящиеся в массиве данные относятся к единственному типу — типу «указатель». Гетерогенные массивы удобны как универсальная структура для хранения наборов данных произвольных типов. Реализация гетерогенности требует усложнения механизма поддержки массивов в трансляторе языка.

Работа с памятью

Типовым способом реализации статического гомогенного (хранящего данные одного типа) массива является выделение непрерывного блока памяти объёмом $S \cdot m_1 \cdot m_2 \cdot \ldots m_n$, где S — размер одного элемента, а $m_1, \ldots m_n$ — размеры диапазонов индексов (то есть количество значений, которые может принимать соответствующий индекс). При обращении к элементу массива с индексом $(i_1, i_2, \ldots i_n)$ адрес соответствующего элемента вычисляется как $B + S \cdot \left(\left(\ldots (i_{1\text{p}} m_1 + i_{2\text{p}}) \cdot m_2 + \cdots + i_{(n-1)\text{p}} \right) \cdot m_{n-1} + i_{n\text{p}} \right)$, где B — база (адрес начала блока памяти массива), $i_{k\text{p}}$ — значение k-го индекса, приведённое к целому с нулевым начальным смещением. Порядок следования индексов в формуле вычисления адреса может быть различным. (Этот способ соответствует реализации в большинстве компиляторов языка Си; в Фортране порядок индексов противоположен).

Таким образом, адрес элемента с заданным набором индексов вычисляется так, что время доступа ко всем элементам массива с теоретической точки зрения одинаково; однако, могут сказываться различные значения задержек отклика от оперативной памяти к ячейкам, расположенным на разных элементах памяти, но в практике высокоуровневого программирования такими тонкостями, за редкими исключениями, пренебрегают.

Обычным способом реализации гетерогенных массивов является отдельное хранение самих значений элементов и размещение в блоке памяти массива (организованного как обычный гомогенный массив, описанный выше) указателей на эти элементы. Поскольку указатели на значения любых типов, как правило, имеют один и тот же размер, удаётся сохранить простоту вычисления адреса, хотя возникают дополнительные накладные расходы на размещение значений элементов и обращение к ним.

Для динамических массивов может использоваться тот же механизм размещения, что и для статических, но с выделением некоторого объёма дополнительной памяти для расширения и добавлении механизмов изменения размера и перемещения содержимого массива в памяти.

Также динамические и гетерогенные массивы могут реализовываться путём использования принципиально иных методов хранения значений в памяти, например, одно- или двухсвязных списков. Такие реализации могут быть более гибкими, но требуют, как правило, дополнительных накладных расходов. Кроме того, в них обычно не удаётся выдержать требование константного времени доступа к элементу.

Операции. К массиву можно добавить элемент или несколько — в конец или в определенный участок ряда. Также можно вернуть элемент по индексу, в некоторых случаях — удалить. По прямому обращению к элементу по индексу его можно изменить. Существуют также операции, позволяющие узнать длину массива, и операции, выдающие весь массив целиком.

Применение. Массивы применяют в большинстве ситуаций, где требуется организованное хранение данных. Не используют их только в сложных структурах: там логичнее будет применить другой формат. Но и тут массивы нужны как строительные «кирпичики» для реализации более комплексных структур.

Стек.

Стек (англ. stack — стопка) — абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять вторую сверху, нужно снять верхнюю.

В цифровом вычислительном комплексе стек называется магазином — по аналогии с магазином в огнестрельном оружии (стрельба начнётся с патрона, заряженного последним).

В некоторых языках (например, Lisp, Python) стеком можно назвать любой список, так как для них доступны операции pop и push. В языке C++ стандартная библиотека имеет класс с реализованной структурой и методами.

Операции со стеком

Возможны три операции со стеком: добавление элемента (иначе проталкивание, push), удаление элемента (pop) и чтение головного элемента (peek).

При проталкивании (push) добавляется новый элемент, указывающий на элемент, бывший до этого головой. Новый элемент теперь становится головным.

При удалении элемента (pop) убирается первый, а головным становится тот, на который был указатель у этого объекта (следующий элемент). При этом значение убранного элемента возвращается.

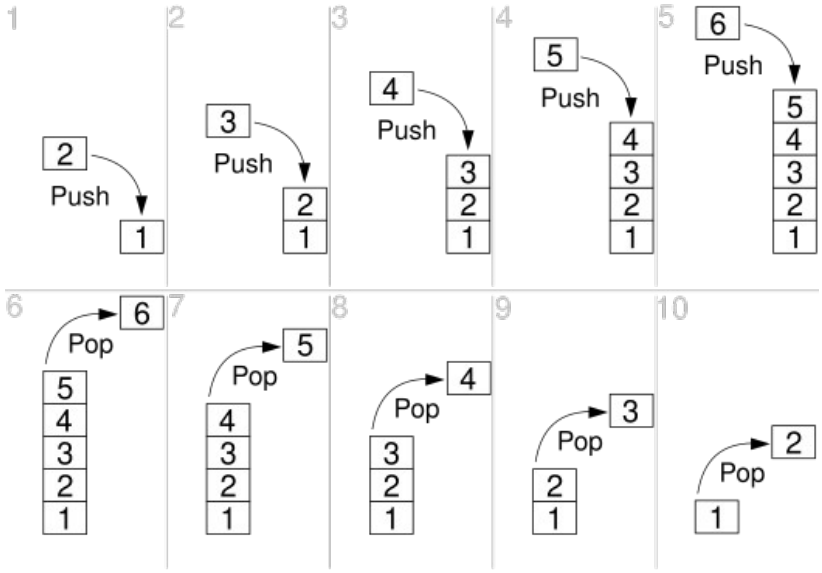

Рисунок 2: Операции вталкивания и выталкивания данных из стека операциями push и pop.

Организация в памяти

Зачастую стек реализуется в виде однонаправленного списка (каждый элемент в списке содержит помимо хранимой информации в стеке указатель на следующий элемент стека).

Но также часто стек располагается в одномерном массиве с упорядоченными адресами. Такая организация стека удобна, если элемент информации занимает в памяти фиксированное количество слов, например, 1 слово. При этом отпадает необходимость хранения в элементе стека явного указателя на следующий элемент стека, что экономит память. При этом указатель стека (Stack Pointer, — SP) обычно является регистром процессора и указывает на адрес головы стека.

Предположим для примера, что голова стека расположена по меньшему адресу, следующие элементы располагаются по нарастающим адресам. При каждом вталкивании слова в стек SP сначала увеличивается на 1 и затем по адресу из SP производится запись в память. При каждом извлечении слова из стека (выталкивании) сначала производится чтение по текущему адресу из SP и последующее уменьшение содержимого SP на 1.

При организации стека в виде однонаправленного списка значением переменной стека является указатель на его вершину — адрес вершины. Если стек пуст, то значение указателя равно NULL.

Область применения

Программный вид стека используется для обхода структур данных, например, дерево или граф. При использовании рекурсивных функций также будет применяться стек, но его аппаратный вид. Кроме этих назначений, стек используется для организации стековой машины, реализующей вычисления в обратной польской записи. Примером использования стековой машины является программа Unix dc.

Для отслеживания точек возврата из подпрограмм используется стек вызовов.

Арифметические сопроцессоры, программируемые микрокалькуляторы и язык Forth используют стековую модель вычислений.

Идея стека используется в стековой машине среди стековых языков программирования.

Аппаратный стек

Другое название аппаратного стека — машинный стек. Работа с ним поддерживается аппаратно центральным процессором. Машинный стек используется для нужд выполняющейся программы: хранения переменных и вызова подпрограмм. При вызове подпрограммы (процедуры) процессор помещает в стек адрес команды, следующей за командой вызова подпрограммы «адрес возврата» из подпрограммы. По команде возврата из подпрограммы

из стека извлекается адрес возврата в вызвавшую подпрограмму программу и осуществляется переход по этому адресу.

Аналогичные процессы происходят при аппаратном прерывании (процессор X86 при аппаратном прерывании сохраняет автоматически в стеке ещё и регистр флагов). Кроме того, компиляторы размещают локальные переменные процедур в стеке (если в процессоре предусмотрен доступ к произвольному месту стека).

В архитектуре X86 аппаратный стек — непрерывная область памяти, адресуемая специальными регистрами ESP (указатель стека) и SS (селектор сегмента стека).

До использования стека он должен быть инициализирован так, чтобы регистры SS:ESP указывали на адрес головы стека в области физической оперативной памяти, причём под хранение данных в стеке необходимо зарезервировать нужное количество ячеек памяти (очевидно, что стек в ПЗУ, естественно, не может быть организован). Прикладные программы, как правило, от операционной системы получают готовый к употреблению стек. В защищённом режиме работы процессора сегмент состояния задачи содержит четыре селектора сегментов стека (для разных уровней привилегий), но в каждый момент используется только один стек.

Операции. Среди операций специально для стеков — так называемый push, то есть добавление элемента в конец, и pop — извлечение элемента из конца. Извлечение означает, что элемент удаляется из стека, но возвращается как значение.

Также есть проверка пустоты стека и операция, возвращающая последний элемент без его удаления.

Применение. Стек используется при выделении памяти программам и процессам. Функции, работающие внутри программы, как бы «укладываются» в системный стек. Популярное выражение stack overflow — это распространенная ошибка, возникающая, когда стек переполняется из-за неправильного использования памяти. Чаще всего такое бывает при рекурсии.

Очередь.

Очередь (англ. queue) — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, англ. first in, first out). Добавление элемента (принято обозначать словом enqueue — поставить в очередь) возможно лишь в конец очереди, выборка — только из начала очереди (что принято называть словом dequeue — убрать из очереди), при этом выбранный элемент из очереди удаляется.

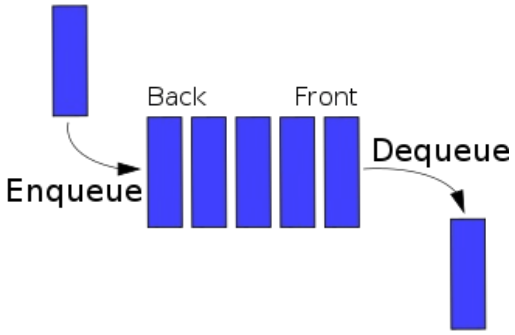

Рисунок 3: Принцип работы очереди

Способы реализации очереди

Существует несколько способов реализации очереди в языках программирования. Первый способ представляет очередь в виде массива и двух целочисленных переменных start и end.

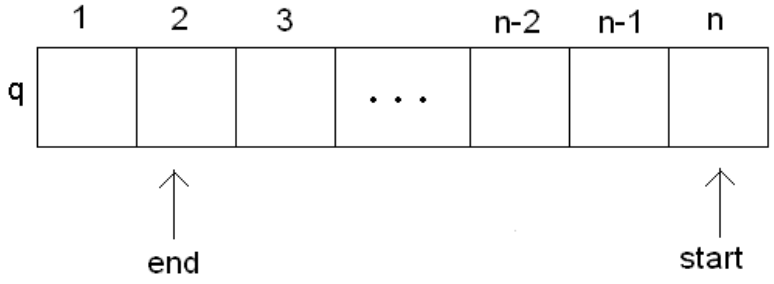

Рисунок 4: Реализация очереди при помощи массива

Обычно start указывает на голову очереди, end — на элемент, который заполнится, когда в очередь войдёт новый элемент. При добавлении элемента в очередь в q[end] записывается новый элемент очереди, а end уменьшается на единицу. Если значение end становится меньше 1, то мы как бы циклически обходим массив, и значение переменной становится равным n. Извлечение элемента из очереди производится аналогично: после извлечения элемен-

та q[start] из очереди переменная start уменьшается на 1. С такими алгоритмами одна ячейка из n всегда будет незанятой (так как очередь с n элементами невозможно отличить от пустой), что компенсируется простотой алгоритмов.

Преимущества данного метода: возможна незначительная экономия памяти по сравнению со вторым способом; проще в разработке.

Недостатки: максимальное количество элементов в очереди ограничено размером массива. При его переполнении требуется перевыделение памяти и копирование всех элементов в новый массив.

Второй способ основан на работе с динамической памятью. Очередь представляется в качестве линейного списка, в котором добавление/удаление элементов идет строго с соответствующих его концов.

Преимущества данного метода: размер очереди ограничен лишь объёмом памяти.

Недостатки: сложнее в разработке; требуется больше памяти; при работе с такой очередью память сильнее фрагментируется; работа с очередью несколько медленнее.

Разновидностью очередей являются деки. Деками называют двусторонние очереди: они объединяют возможности и очереди, и стека. Такие структуры данных могут работать и по принципу FIFO, и по принципу LIFO. Доступ к элементам возможен с любого конца.

Можно сказать, что деки объединяют в себе операции, характерные для очередей и стеков. В каком-то смысле эти структуры данных напоминают массивы и приближены к ним по функциональности.

Деки используют, когда важно обеспечить доступ и к первым, и к последним элементам. Например, при оптимизации выполнения процессов.

Операции. Основные операции, характерные для очереди — добавление элемента в конец, получение элемента из начала без удаления или с удалением, а также проверка ее на пустоту.

Применение. Очередь в программировании используется, как и в реальной жизни, когда нужно совершить какие-то действия в порядке их поступления, выполнив их последовательно. Примером может служить организация событий в Windows. Когда пользователь оказывает какое-то действие на приложение, то в приложении не вызывается соответствующая процедура (ведь в этот момент приложение может совершать другие действия), а ему присылается сообщение, содержащее информацию о совершенном действии, это сообщение ставится в очередь, и только когда будут обработаны сообщения, пришедшие ранее, приложение выполнит необходимое действие.

Клавиатурный буфер BIOS организован в виде кольцевого массива, обычно длиной в 16 машинных слов, и двух указателей: на следующий элемент в нём и на первый незанятый элемент.

Очереди так же применяют для организации многопоточных процессов, когда несколько действий выполняются одновременно. Эта структура

данных позволяет задать действиям последовательность и очередность выполнения. Так балансируется нагрузка на программу и предотвращается зависание. По такому же принципу очереди используют при выполнении запросов — если их много, и они поступают очень быстро.

Множество.

Множество — тип и структура данных в информатике, которая является реализацией математического объекта множество.

Данные типа множество позволяют хранить ограниченное число значений определённого типа без определённого порядка. Повторение значений, как правило, недопустимо. За исключением того, что множество в программировании конечно, оно в общем соответствует концепции математического множества. Для этого типа в языках программирования обычно предусмотрены стандартные операции над множествами.

В зависимости от идеологии, разные языки программирования рассматривают множество как простой или сложный тип данных.

Ее можно представить не как последовательность или список, а как «облако тегов»: неупорядоченный набор уникальных значений. Это определение близко к математическому понятию множества. И работают с ним так же: не сортируют и не структурируют, но хранят, получают и анализируют данные.

Множество еще называют сетом. По-английски название звучит как Set.

Операции. Характерные для этой структуры данных операции рассчитаны на работу с двумя множествами. Например, это операции объединения множества, сравнения, поиска пересечений между двумя сетами. Похоже на круги Эйлера, но программно реализованные.

Также есть операция, которая возвращает только элементы множества, не пересекающиеся с другим, и операция поиска подмножества. Она показывает, содержится ли одно множество в другом.

Применение. Множества используют для хранения наборов данных, которые должны быть уникальными, но не нуждаются в упорядочивании. Обычно это наборы, с которыми нужно проводить определенные операции: искать в них подмножества, объединять, сравнивать с другими и выделять пересечения.

Связанный список.

Связный список — базовая динамическая структура данных в информатике, состоящая из узлов, содержащих данные и ссылки («связки») на следующий и (или) предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями. Первый элемент называется головой списка, последний — хвостом.

Виды связных списков:

Линейный однонаправленный список — это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на NULL. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

В информатике линейный список обычно определяется как абстрактный тип данных (АТД), формализующий понятие упорядоченной коллекции данных. На практике линейные списки обычно реализуются при помощи массивов и связных списков. Иногда термин «список» неформально используется также как синоним понятия «связный список». К примеру, АТД нетипизированного изменяемого списка может быть определён как набор из конструктора и основных операций:

- операция, проверяющая список на пустоту.
- три операции добавления объекта в список (в начало, конец или внутрь после любого (n-го) элемента списка);
- операция, вычисляющая первый (головной) элемент списка;
- операция доступа к списку, состоящему из всех элементов исходного списка, кроме первого.

Характеристики:

- Длина списка. Количество элементов в списке.
- Списки могут быть типизированными или нетипизированными. Если список типизирован, то тип его элементов задан, и все его элементы должны иметь типы, совместимые с заданным типом элементов списка. Чаще списки типизированы.
- Список может быть сортированным или несортированным.
- В зависимости от реализации может быть возможен произвольный доступ к элементам списка.

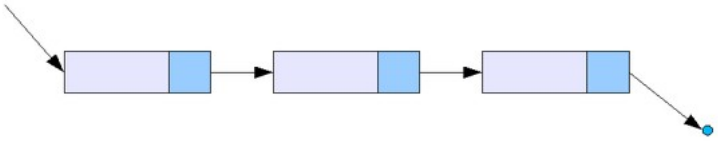

Рисунок 5: Линейный однонаправленный список

Двунаправленный связный список — ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещения в обе стороны. В этом списке проще производить удаление и перестановку элементов, так как легко доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

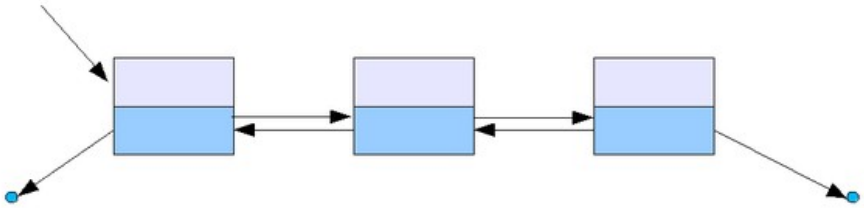

Рисунок 6: Двунаправленный связный список

Кольцевой связный список. Разновидностью связных списков является кольцевой (циклический, замкнутый) список. Он тоже может быть односвязным или двусвязным. Последний элемент кольцевого списка содержит указатель на первый, а первый (в случае двусвязного списка) — на последний.

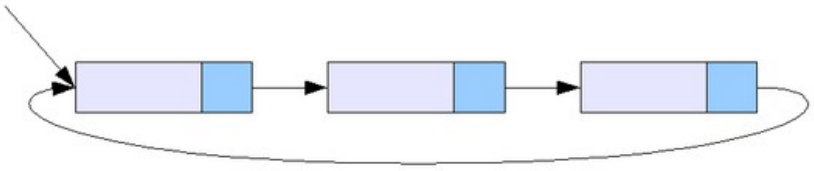

Рисунок 7: Кольцевой связный список

Преимущества связных списков:

– эффективное (за константное время) добавление и удаление элементов;

– размер ограничен только объёмом памяти компьютера и разрядностью указателей;

– динамическое добавление и удаление элементов.

Недостатки. Недостатки связных списков вытекают из их главного свойства — последовательного доступа к данным:

– сложность прямого доступа к элементу, а именно определения физического адреса по его индексу (порядковому номеру) в списке;

– на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны);

– некоторые операции со списками медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы;

– соседние элементы списка могут быть распределены в памяти нелокально, что снизит эффективность кэширования данных в процессоре;

– над связными списками, по сравнению с массивами, гораздо труднее (хоть и возможно) производить параллельные векторные операции, такие, как вычисление суммы: накладные расходы на перебор элементов снижают эффективность распараллеливания.

Операции. В список можно добавить элемент — в голову, хвост или заданный участок. Также можно обновить значение элемента или удалить его, провести поиск по структуре данных. Также есть операция-проверка, пуст ли список. Доступ к следующему элементу возможен с помощью указателя next, к предыдущему — prev.

Применение. Связанные списки используют для построения более сложных структур, например, графов и хэш-таблиц. Кроме того, они бывают важны при выделении памяти программам в динамических процессах и операционных системах.

Ассоциативный массив.

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов — например, строки.

Ассоциативный массив — абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида «(ключ, значение)» и поддерживающий операции:

- добавления пары INSERT(ключ, значение);
- поиск FIND(ключ);
- удаление пары по ключу REMOVE(ключ).

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

В паре (k , v) значение v называется значением, ассоциированным с ключом k. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Операция FIND(ключ) возвращает значение, ассоциированное с заданным ключом, или некоторый специальный объект, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением, возможно, информации о том, успешно ли была выполнена данная операция).

Поддержка ассоциативных массивов есть практически во всех языках программирования высокого уровня. Для языков, которые не имеют встроенных средств работы с ассоциативными массивами, существует множество реализаций в виде библиотек. Ассоциативная память это способ прямой поддержки ассоциативных массивов на аппаратном уровне.

Реализации ассоциативного массива. Существует множество различных реализаций ассоциативного массива. Самая простая реализация может быть основана на обычном массиве, элементами которого являются пары (ключ, значение). Для ускорения операции поиска можно упорядочить элементы этого массива по ключу и осуществлять нахождение методом бинарного поиска. Но это увеличит время выполнения операции добавления новой пары, так как необходимо будет «раздвигать» элементы массива, чтобы в образовавшуюся пустую ячейку поместить новую запись.

Наиболее популярны реализации, основанные на различных деревьях поиска. Так, например, в стандартной библиотеке STL языка C++ контейнер map реализован на основе красно-чёрного дерева. В языках D, Java, Ruby, Tcl, Python используется один из вариантов хеш-таблицы. Есть и другие реализации.

У каждой реализации есть свои достоинства и недостатки. Важно, чтобы все три операции выполнялись как в среднем, так и в худшем случае за время O(log n), где n — текущее количество хранимых пар. Для сбалансиро-

ванных деревьев поиска (в том числе для красно-чёрных деревьев) это условие выполнено.

В реализациях, основанных на хеш-таблицах, среднее время оценивается как O(1), что лучше, чем в реализациях, основанных на деревьях поиска. Но при этом не гарантируется высокая скорость выполнения отдельной операции: время операции INSERT в худшем случае оценивается как O(n). Операция INSERT выполняется долго, когда коэффициент заполнения становится высоким и необходимо перестроить индекс хеш-таблицы.

КЛЮЧ	ЗНАЧЕНИЕ
январь	327.2
февраль	368.2
март	197.6
апрель	178.4
май	100.0
июнь	69.9
июль	32.3
август	19.0
сентябрь	21.0
октябрь	37.0
ноябрь	73.2
декабрь	110.8
годовой	1551.0

Рисунок 8: Ассоциативный массив

Операции. С картами можно работать как с массивами, но доступ к ним осуществляется по названию ключа, а не индексу. Можно добавить пару ключ-значение, удалить ее или изменить значение. Также можно обратиться к элементу по его ключу.

У ассоциативных массивов иначе реализован проход по всей структуре, чем у обычных: тут не получится просто увеличивать индекс на 1 на каждом шаге, ведь ключ может быть каким угодно. Поэтому в языках программирования обычно предусмотрены специальные операции для прохода по словарю.

Применение. Ассоциативные массивы используют для хранения данных, которые нужно быстро найти по какому-то неизменному ключу. Еще они применяются для передачи данных — ключ выступает как название параметра, а в самом элементе хранится его значение.

Хэш-таблица.

Хеш-таблица — структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию удаления и операцию поиска пары по ключу.

Существуют два основных варианта хеш-таблиц: с открытой адресацией и списками. Хеш-таблица является массивом H, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица со списками).

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение i = hash(key) играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[i].

Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется коллизией. Такие события не так уж и редки — например, при вставке в хеш-таблицу размером 365 ячеек всего лишь 23 элементов вероятность коллизии уже превысит 50% (если каждый элемент может равновероятно попасть в любую ячейку) — парадокс дней рождения. Поэтому механизм разрешения коллизий — важная составляющая любой хеш-таблицы.

В некоторых специальных случаях удаётся избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую совершенную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без коллизий. Хеш-таблицы, использующие подобные хеш-функции, не нуждаются в механизме разрешения коллизий, и называются хеш-таблицами с прямой адресацией.

Число хранимых элементов, делённое на размер массива H (число возможных значений хеш-функции), называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (добавление, удаление, поиск элемента) в среднем выполняются за время O(1). Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: создать новый массив H′ увеличенного размера и заново добавить в него все пары из старого массива пар (он же старая хеш-таблица).

С помощью хэш-таблицы можно генерировать ключи автоматически, например, в ситуациях, когда их название не должно нести полезной нагрузки. Это удобнее и быстрее, чем ассоциативный массив, если речь идет о

больших объемах данных. Кроме того, использование хэшей позволяет шифровать информацию — правда, одной таблицы для этого недостаточно.

При работе с хэш-таблицами важно избегать коллизий. Чтобы такого не было, нужно грамотно подбирать формулу для каждого случая. Также существуют специальные стратегии для предотвращения коллизий.

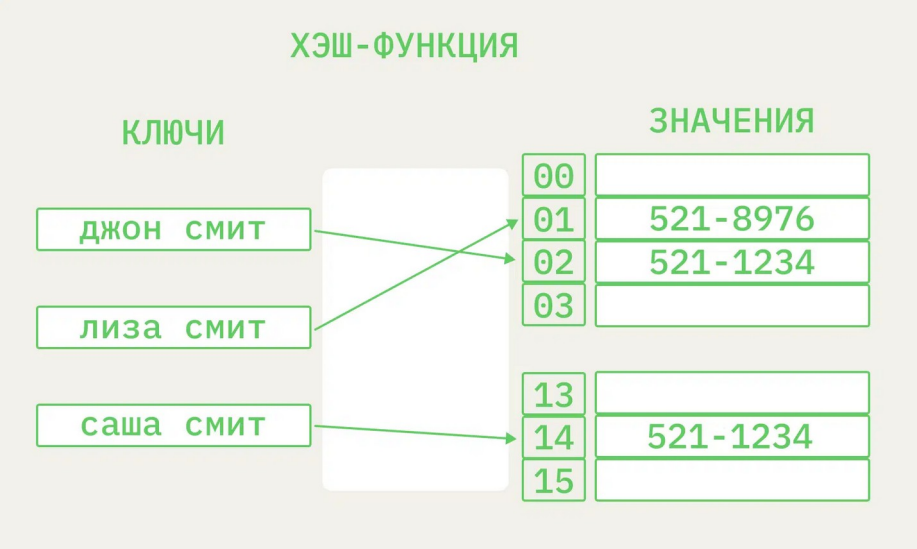

Рисунок 9: Хеш-таблица

Операции. В таблицу можно добавить элемент, удалить или найти по тому или иному признаку. При ее создании также можно и нужно задать формулу, по которой будут генерироваться хэши.

Применение. Хэш-таблицы используются для хранения больших объемов информации, в базах данных, а также для создания кэшей или при построении более сложных структур. Чаще всего таблицы используют, когда нужен быстрый доступ к информации.

Граф.

Граф — нелинейная структура организации данных, которая состоит из «вершин» и «ребер» между ними. Каждая вершина — это значение, а ребра — пути, которые соединяют между собой вершины. Получается своеобразная «сетка», похожая на карту дорог или созвездие.

Графы бывают неориентированными, когда ребра не имеют конкретного направления, и ориентированными. Во втором случае по ребрам можно пройти только в одну сторону — как по дороге с односторонним движением. Есть смешанные графы, в которых есть и ориентированные, и неориентированные ребра.

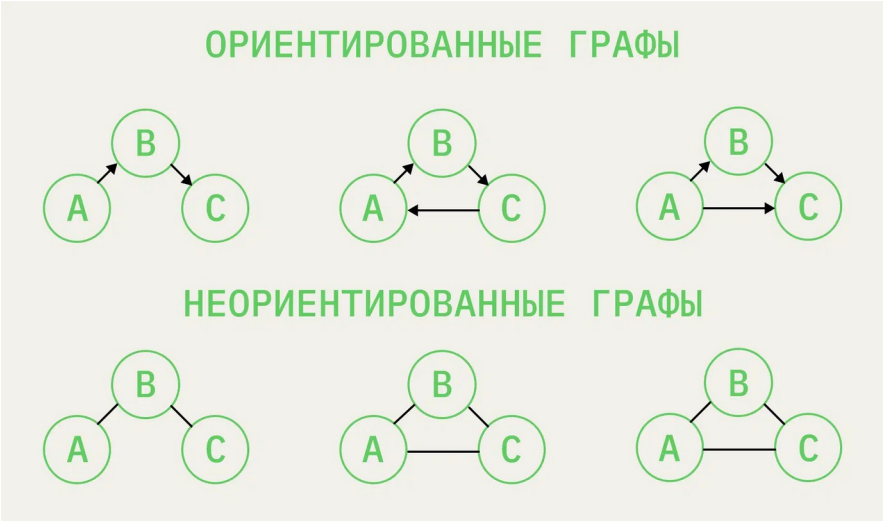

Рисунок 10: Графы ориентированные и неориентированные

Также существуют взвешенные графы, у ребер которых есть «вес» — то или иное значение. Например, в карте дорог весом ребра-дороги можно назвать его длину. Графы часто представляют в виде матрицы смежности.

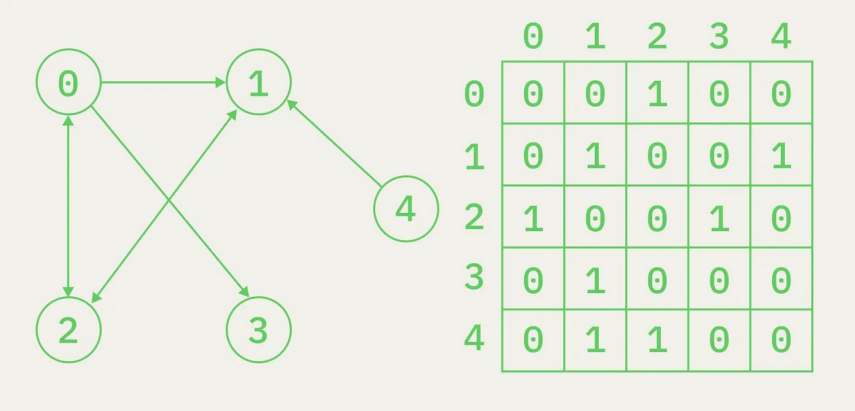

Рисунок 11: Матрица смежности графа

Графы обычно реализуют с помощью связанных списков или матриц — двумерных массивов.

Операции. С этими структурами есть базовые операции: добавление вершины или ребра, отображение вершины или графа целиком, оценка «стоимости» обхода взвешенного графа и так далее.

Существует несколько алгоритмов обхода графов для поиска информации или для нахождения кратчайшего пути от одной вершины до другой. Например, DFS, BFS, алгоритм Дейкстры и другие. Для них не всегда существуют отдельные команды, так что реализовать эти алгоритмы может понадобиться самостоятельно.

Применение. Графы активно используют для хранения моделей машинного обучения, а также при работе с картами, например, для построения маршрутов через онлайн-сервисы. Программное эмулирование электрических цепей — это тоже графы. Также теория графов применяется в поисковых алгоритмах и социальных сетях — например, так рассчитывается «охват» друзей одного человека. Графы можно использовать для распределения ресурсов внутри системы или при организации сложных вычислений.

Деревья.

Деревья можно назвать частным случаем графов. Это тоже структуры из вершин и ребер, но имеющие древовидный формат. Вершины деревьев называются узлами. От одного узла может отходить несколько вершин-потомков, но предок у каждого узла может быть только один. Так и получается древовидная иерархическая структура.

В программировании применяют несколько видов деревьев. Большинство из них реализуют с помощью графов.

Бинарные деревья поиска — самый распространенный формат деревьев. У каждого узла может быть не более двух потомков. Если значение узла-потомка меньше предка, он располагается слева. Если равно или больше — справа. С помощью таких деревьев удобно сортировать данные и искать в них нужное значение с помощью бинарного поиска.

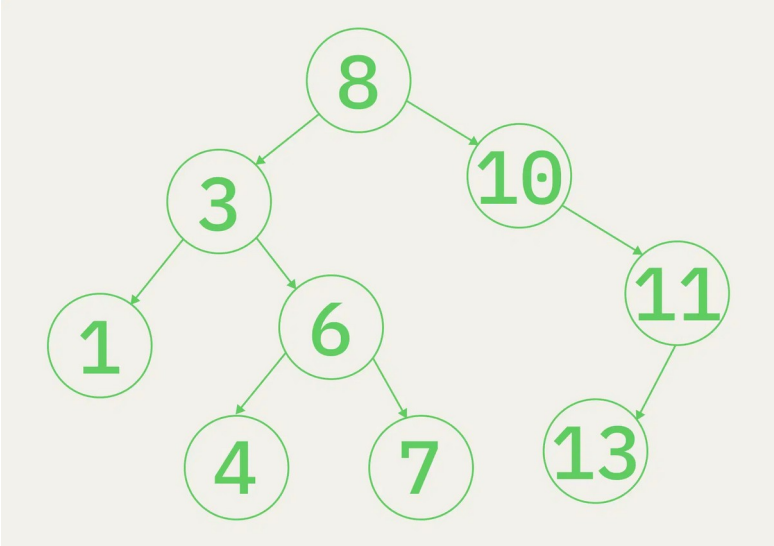

Рисунок 12: Бинарное дерево поиска

Префиксные деревья, они же нагруженные деревья или боры — это деревья, которые хранят данные «по цепочке». Узлы-предки — префиксы, которые нужны, чтобы перейти к потомкам. При прохождении дерева «собираются» полные данные. Например, в каждом узле — буква. При прохождении от начала до конца получаются слова. Обычно такие деревья используют для хранения повторяющихся данных или, например, для реализации автодополнения при вводе.

Также существуют N-арные деревья, красно-черные деревья, AVL-деревья, сбалансированные, 2-3-4-деревья и так далее.

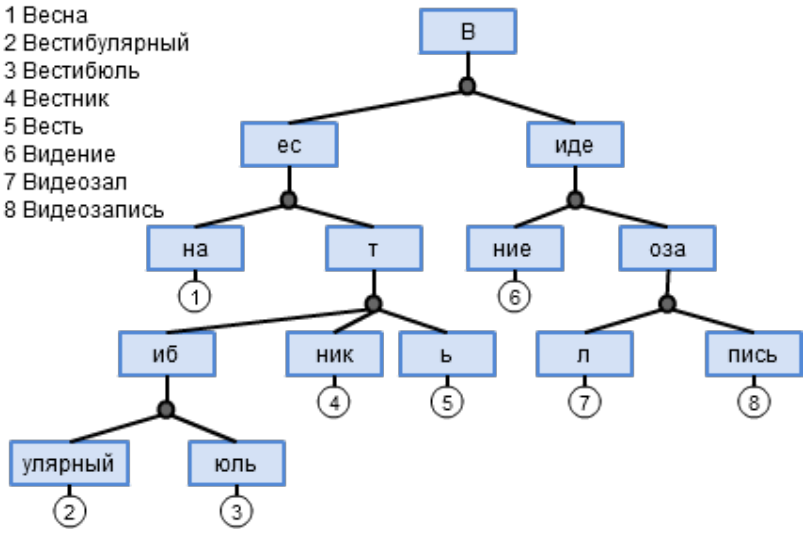

Рисунок 13: Пример хранения слов в префиксном дереве

Операции. По деревьям можно проходить снизу вверх или сверху вниз, в ширину или в глубину. Для этого существуют специальные алгоритмы — как и для графов. Так как обычно в деревьях расположение узла зависит от его значения, произвольно добавить узел куда угодно не получится. Поэтому при добавлении, удалении или изменении узла происходит перебалансировка — изменение структуры с учетом добавленного значения.

Применение. Деревья используют для хранения информации, для которой важна иерархичность. Также их могут применять для эффективной сортировки, поиска нужных значений или хранения повторяющихся данных, как в случае с префиксными деревьями. Часто деревья встречаются в алгоритмах машинного обучения.

Заключение.

Если говорить простым языком, то структура данных представляет собой контейнер, в котором информация скомпонована специальным образом. Чего удается достичь благодаря такому строению? Для сравнения, при выполнении одних операций определенная структура данных будет весьма эффективна, в то время как при выполнении других — не очень. Задача разработчика вне зависимости от языка программирования (language of programming) как раз в том и состоит, чтобы уметь выбирать наиболее подходящую структуру и знать, как сравнить их между собой с учетом поставленной задачи. Но это невозможно без ясного и единого представления о существующих способах организации информации.

Список используемых источников.

1. «10 структур данных, которые должен знать каждый разработчик» https://practicum.yandex.ru/blog/10-osnovnyh-struktur-dannyh/
2. «Основные понятия структуры данных программы» https://otus.ru/journal/osnovnye-ponyatiya-struktury-dannyh-programmy/
3. «Структура данных» https://ru.wikipedia.org/wiki/Структура_данных
4. «Массив (тип данных)» https://ru.wikipedia.org/wiki/Массив_(тип_данных)
5. «Стек» https://ru.wikipedia.org/wiki/Стек
6. «Множество» https://ru.wikipedia.org/wiki/Множество_(тип_данных)
7. «Связный список» https://ru.wikipedia.org/wiki/Связный_список
8. «Ассоциативный массив» https://ru.wikipedia.org/wiki/Ассоциативный_массив