Санкт-Петербургский государственный университет телекоммуникаций им. проф. М. А. Бонч-Бруевича

Кафедра Безопасности информационных систем

Алгоритмы и структуры данных

©***Моисеев Игорь Анатольевич***
mig1256@.mail.ru

Рекомендуемая литература:

Роберт Седжвик. Алгоритмы на C++. Анализ/Структуры данных/
Сортировка/Поиск/Алгоритмы на графах: Пер. с англ./Руководство / Ван Вик Кристофер Дж., Седжвик Роберт. – Издательство «Вильямс», 2019. – 1056 с.

Лекция 1. Алгоритмы. Типы данных.

Понятие, виды и свойства алгоритмов.

Термин «Алгоритм» связывают с именем хорезмского учёного Аль-Хорезми Мухаммед бен-Муса [1]. Около 825 года он написал сочинение «Книга о сложении и вычитании», из оригинального названия которого происходит слово «алгебра» (аль-джебр — восполнение). В этой книге он впервые дал описание придуманной в Индии позиционной десятичной системы счисления. В первой половине XII века книга аль-Хорезми в латинском переводе проникла в Европу. Переводчик, имя которого до нас не дошло, дал ей название «Algoritmi de numero Indorum» («Алгоритмы о счёте индийском») — таким образом, латинизированное имя среднеазиатского учёного было вынесено в заглавие книги. Сегодня считается, что слово «алгоритм» попало в европейские языки именно благодаря этому переводу. Вообще-то, алгоритмы решения арифметических задач были придуманы задолго до появления самого термина «алгоритм». Одним из старейших численных алгоритмов считается алгоритм Евклида (III век до н.э.) – нахождения наибольшего общего делителя двух чисел. Кроме того можно упомянуть и «решето Эратосфена» для поиска простых чисел, описанное во «Введении в арифметику» Никомахом Герасским (60-120 г.г. н.э.).

Эпохой в развитии теории алгоритмов стал XX век. Началом этого развития стали работы немецкого математика Курта Гёделя (1931г.). Первые фундаментальные работы по теории алгоритмов появились в 1936 году и связаны с именами Тьюринга, Поста, Чёрча [1]. Так как данная дисциплина развивается и в настоящее время, часто используются следующие определения алгоритма:

- «Алгоритм – это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определённость, ввод, вывод, эффективность» (Д. Э. Кнут).
- «Алгоритм – это всякая система вычислений, выполняемых по строго определённым правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи» (А. Колмогоров).
- «Алгоритм – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату» (А. Марков).
- «Алгоритм — точное предписание о выполнении в определённом порядке некоторой системы операций, ведущих к решению всех задач данного типа» (Философский словарь под ред. М. М. Розенталя).
- В теоретических исследованиях используют формализованное определение понятие алгоритма [2].

Определение 1. *Алгоритм – это сформулированное на некотором языке конечное предписание, задающее конечную последовательность выполнимых*

элементарных операций для решения задачи, общее для класса возможных исходных данных.

Если *D* – область (множество) исходных данных задачи, а *R* – множество возможных результатов, тогда алгоритм осуществляет отображение *D*→*R*. Это отображение может быть не полным. Алгоритм называется частичным алгоритмом, если получен результат только для некоторых *d* из *D* и полным алгоритмом, если алгоритм получает правильный результат для всех *d* .

Определение 2. *Алгоритм – точный набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное время.*

В 1950-е годы существенный вклад в теорию алгоритмов внесли работы Колмогорова и Маркова. К 1960-1970 годам оформились следующие направления исследований в теории алгоритмов:

- Классическая теория алгоритмов.
- Теория асимптотического анализа алгоритмов (понятие сложности и трудоёмкости алгоритма, критерии оценки алгоритмов, методы получения асимптотических оценок, в частности для рекурсивных алгоритмов, асимптотический анализ трудоемкости или времени выполнения);
- Теория практического анализа вычислительных алгоритмов (получение явных функций трудоёмкости, интервальный анализ функций, практические критерии качества алгоритмов, методика выбора рациональных алгоритмов), основополагающей работой в этом направлении, можно считать фундаментальный труд Д. Кнута «Искусство программирования для ЭВМ» алгоритмов.

В соответствии с изложенным выше, в теории алгоритмов ставятся и решаются следующие задачи [2]:

- формализация понятия «алгоритм» и исследование формальных алгоритмических систем;
- формальное доказательство алгоритмической неразрешимости задач;
- классификация задач, определение и исследование сложных классов;
- асимптотический анализ сложности алгоритмов;
- исследование и анализ рекурсивных алгоритмов;
- получение явных функций трудоемкости в целях сравнительного анализа алгоритмов;
- разработка критериев сравнительной оценки качества алгоритмов.

Разработанные методы и методики теории алгоритмов позволяют осуществить:

- рациональный выбор из известного множества алгоритмов решения задачи с учетом особенностей их применения (например, при ограничениях на размерность исходных данных или объема дополнительной памяти);
- получение временных оценок решения сложных задач;
- получение достоверных оценок невозможности решения некоторой задачи за определенное время, что важно для криптографических методов;

- разработку и совершенствование эффективных алгоритмов решения задач в области обработки информации на основе практического анализа.

Различные определения алгоритма в явной или неявной форме влекут за собой ряд требований:

- алгоритм должен содержать конечное количество элементарно выполнимых предписаний, т.е. удовлетворять требованию конечности записи;
- алгоритм должен выполнять конечное количество шагов при решении задачи, т.е. удовлетворять требованию конечности действий;
- алгоритм должен быть единым для всех допустимых исходных данных, т.е. удовлетворять требованию универсальности;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению, т.е. удовлетворять требованию правильности.

Отсюда следует, что любой алгоритм имеет следующие свойства:

- Дискретность — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. Для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.
- Детерминированность — определённость. В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных.

- Понятность — алгоритм для исполнителя должен включать только те команды, которые ему (исполнителю) доступны, которые входят в его систему команд.
- Завершаемость (конечность) — при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов. Вероятностный алгоритм может и никогда не выдать результат, но вероятность этого равна 0.
- Массовость — универсальность. Алгоритм должен быть применим к разным наборам исходных данных.
- Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не даёт результатов вовсе.
- Алгоритм не содержит ошибок, если он даёт правильные результаты для любых допустимых исходных данных.

Существуют вероятностные алгоритмы, в которых следующий шаг работы зависит от текущего состояния системы и генерируемого случайного числа. Однако при включении метода генерации случайных чисел в список «исходных данных», вероятностный алгоритм становится подвидом обычного.

По виду входных и выходных данных можно выделить

- Алгоритмы для решения численных задач (появились первыми);
- Нечисловые алгоритмы.

С началом выпуска электронных вычислительных машин стали разрабатываться в большом количестве различные алгоритмы численного решения задач, которые невозможно было решить аналитически, например задачи автоматического управления летательными аппаратами и т.п. объектами.

С появлением ЭВМ третьего и четвертого поколений цифровые вычислительные машины стали использоваться для хранения и обработки больших объемов различных данных. Стала быстро развиваться теория реляционных баз данных, появились различные СУБД (системы управления базами данных). В настоящее время цифровые компьютеры пятого поколения (к ним часто относят персональные и современные суперкомпьютеры) помимо выполнения сложных и длинных вычислений в большинстве приложений используют предоставляемую им возможность хранить и обеспечивать доступ к большим массивам информации, что рассматривается как их главная характеристика, а выполнение арифметических операций, во многих случаях становится не главной задачей. В таких приложениях большой массив обрабатываемой информации является абстрактным представлением некоторой части реального мира. Информация, доступная компьютеру, представляет собой специально подобранный набор данных, относящихся к решаемой задаче, причем допускается, что этот набор достаточен для получения нужных результатов. Данные являются абстрактным представлением. реальности в том смысле, что некоторые свойства реальных объектов игнорируются, так как они несущественны для этой задачи. Поэтому абстракция – это еще и упрощение реальности. Для примера можно рассмотреть файл с данными о служащих некоторой фирмы. Каждый служащий (абстрактно) представлен в этом файле набором данных, который нужен либо для руководства

компании, либо для бухгалтерских расчетов. Такой набор может содержать некоторую идентификацию служащего, например данные и зарплату. Но в нем почти наверняка не будет несущественной информации о цвете волос, весе или росте и т.п.

Таким образом, для решения задачи необходимо выбрать абстрактное представление реальности, то есть определить набор данных, который будет представлять реальную ситуацию. Этот выбор можно сделать, руководствуясь решаемой задачей. Затем нужно определиться с представлением информации. Здесь выбор определяется средствами вычислительного устройства. В большинстве случаев эти два шага не могут быть полностью разделены.

Выбор представления данных часто довольно сложен и не полностью определяется имеющимися вычислительными средствами. Делать такой выбор всегда нужно с учетом операций, которые нужно выполнять с данными. В качестве примера пример можно привести представление чисел. Если единственное действие, которое нужно выполнять, сложение, то представлением числа может быть реализовано с помощью черточек. Правило сложения при таком представлении очевидное и очень простое. На этом основана римская нотация. Правила сложения просты для маленьких чисел. Представление арабскими цифрами требует неочевидных правил сложения (для маленьких чисел), и их нужно запоминать, но ситуация меняется на противоположную, если нужно складывать большие числа или выполнять умножение и деление. Разбиение этих операций на более простые шаги гораздо проще в случае арабской нотации благодаря ее позиционной структуре.

С другой стороны компьютеры используют внутреннее представление, основанное на двоичных цифрах (битах). Это представление непригодно для использования в обычной жизни, так как здесь обычно приходится иметь дело с большим числом цифр, но весьма удобно для электронных схем, так как два значения 0 и 1 можно легко и надежно представить посредством наличия или отсутствия электрических токов, зарядов или магнитных полей.

Использование языка, предоставляющего удобный набор базовых абстракций, общих для большинства задач обработки данных, влияет главным образом на надежность получающихся программ. Легче спроектировать программу, опираясь в рассуждениях на знакомые понятия чисел, множеств, последовательностей и циклов, чем иметь дело с битами, единицами хранения и переходами управления. Конечно, реальный компьютер представляет любые данные – числа, множества или последовательности – как огромную массу битов. Но программист может забыть об этом, если ему не нужно беспокоиться о деталях представления выбранных абстракций и, если он может считать, что выбор представления, сделанный компьютером (или компилятором), разумен для решаемых задач.

В общем случае при создании компьютерной программы мы реализуем метод, который ранее был разработан для решения какой-либо задачи. Часто этот метод не зависит от конкретного используемого компьютера — весьма вероятно, что он будет равно пригодным для многих компьютеров и многих компьютерных языков. Именно метод, а не саму программу и нужно исследовать для выяснения способа решения задачи.

Термин алгоритм используется в компьютерных науках для описания метода решения задачи, пригодного для реализации в виде компьютерной программы. Алгоритмы составляют основу компьютерных наук, они являются основными объектами изучения во многих, если не в большинстве ее областей.

Большинство алгоритмов касаются методов организации данных для конкретной задачи. Созданные таким образом объекты называются структурами данных, и они также являются центральными объектами изучения в компьютерных науках. Следовательно, алгоритмы и структуры данных следуют рука об руку, структуры данных существуют в качестве промежуточных или конечных продуктов алгоритмов и, следовательно, их можно и нужно изучить, чтобы понять алгоритмы. Простые алгоритмы могут порождать сложные структуры данных и наоборот, сложные алгоритмы могут использовать простые структуры данных.

Компьютерные программы часто бывают слишком оптимизированы. Обеспечение наиболее эффективной реализации конкретного алгоритма может не стоить затраченных усилий, если только алгоритм не должен использоваться для решения очень сложной задачи или же многократно. В противном случае вполне достаточно качественной, сравнительно простой реализации: достаточно быть уверенным в ее работоспособности и в том, что, скорее всего, в худшем случае она будет работать в 5-10 раз медленнее наиболее эффективной версии, что может означать увеличение времени выполнения на несколько дополнительных секунд. И, наоборот, правильный выбор алгоритма может ускорить работу в 100—1000 и более раз, что может вылиться во время выполнения в экономию минут, часов и даже более того.

Выбор наилучшего алгоритма выполнения конкретной задачи может оказаться сложным процессом, возможно, требующим сложного математического анализа. Направление компьютерных наук, занимающееся изучением подобных вопросов, называется анализом алгоритмов. Анализ многих изучаемых алгоритмов показывает, что они имеют прекрасную производительность, о хорошей работе других известно просто из опыта их применения. Основная цель курса — изучение приемлемых алгоритмов выполнения важных задач, хотя значительное внимание будет уделено также сравнительной производительности различных методов. Не следует использовать алгоритм, не имея представления о ресурсах, которые могут потребоваться для его выполнения, поэтому необходимо знать, как могут выполняться используемые алгоритмы.

Когда компьютер используется для решения той или иной задачи, как правило, мы сталкиваемся с рядом возможных различных подходов. При решении простых задач выбор какого-либо подхода вряд ли имеет особое значение, если только выбранный подход приводит к правильному решению. Однако, при решении сложных задач (или в приложениях, в которых приходится решать очень большое количество простых задач) мы немедленно сталкиваемся с необходимостью разработки методов, при которых время или память используются с максимальной эффективностью. Основная причина изучения алгоритмов состоит в том, что это позволяет обеспечить экономию ресурсов, вплоть до получения решений задач, которые в противном случае были бы невозможны. В приложениях, в которых обрабатываются миллионы объектов, часто оказывается возможным ускорить работу программы в миллионы раз, используя хорошо разработанный алгоритм.

Для сравнения, вложение дополнительных денег или времени для приобретения и установки нового компьютера потенциально позволяет ускорить работу программы всего в 100 раз. Тщательная разработка алгоритма — исключительно эффективная часть процесса решения сложной задачи в любой области применения. При разработке очень большой или сложной компьютерной программы значительные усилия должны затрачиваться на выяснение и определение задачи, которая должны быть решена, осознание ее сложности и разбиение ее на менее сложные подзадачи, решения которых можно легко реализовать. Часто реализация многих из алгоритмов, требующихся после разбиения, тривиальна.

Следует заметить, что в большинстве случаев существует несколько алгоритмов, выбор которых критичен, поскольку для их выполнения требуется большая часть системных ресурсов. Именно изучением этих типов алгоритмов мы и будем заниматься. Мы изучим ряд основополагающих алгоритмов, которые полезны при решении сложных задач во многих областях применения. Совместное использование программ в компьютерных системах становится все более распространенным, поэтому, хотя возможно использовать придется многие из рассматриваемых в дальнейшем алгоритмов, можно надеяться, что реализовывать придется лишь немногие из них. Например, библиотека стандартных шаблонов (Standard Template Library) C++ содержит реализации множества базовых алгоритмов. Однако реализация простых версий основных алгоритмов позволяет лучше их понять и, следовательно, эффективнее использовать и настраивать более совершенные библиотечные версии. И что еще важнее, повод повторной реализации основных алгоритмов возникает очень часто. Основная причина состоит в том, что мы сталкиваемся, и очень часто, с совершено новыми вычислительными средами (аппаратными и программными) с новыми свойствами, которые не могут наилучшим образом использоваться старыми реализациями.

Поэтому часто приходится реализовывать базовые алгоритмы, приспособленные к конкретной задаче, а не основывающиеся на системных подпрограммах.

Рассмотрим пример реализации простейшей задачи вычисления суммы целочисленной арифметической прогрессии, использующий разные алгоритмы. Требуется вычислить сумму S целых чисел от 1 до N, где N=10000. Это можно сделать так

```
int  S=0,N=10000;
for(int  i=1;i<=N;i++)
S=S+i;
cout << S;
```

Другой вариант программы может иметь следующий вид

```
Int N=10000;
Int S=N/2*N+N/2;
cout << S;
```

Третий вариант

```
Int N=10000, i=1,S=0;
do {S=S+I; i++;} while (i<+N);
cout << S;
```

Ясно, что результат выполнения программ один и тот же. Вопрос заключается в эффективности того или иного алгоритма по какому-либо критерию.

Типы данных.

Организация данных для обработки является важным этапом разработки программ. Для реализации многих приложений выбор структуры данных — единственное важное решение: когда выбор сделан, разработка алгоритмов не вызывает затруднений. Для одних и тех же данных различные структуры будут занимать разный объем дискового пространства. Одни и те же операции с различными структурами данных создают алгоритмы различной эффективности. Выбор алгоритмов и структур данных тесно взаимосвязан. Программисты постоянно изыскивают способы повышения быстродействия или экономии дискового пространства за счет оптимального выбора.

Структура данных не является пассивным объектом: необходимо принимать во внимание выполняемые с ней операции и алгоритмы, используемые для этих операций. Далее будем рассматривать различные массивы, связные списки и строки. Эти классические структуры данных имеют широкое применение: посредством деревьев они формируют основу почти всех алгоритмов, которые мы будем рассматривать далее. Будут рассматриваться различные примитивные операции для управления структурами данных, а также разработки базового набора средств, которые можно использовать для составления сложных алгоритмов. Изучение хранения данных в виде объектов переменных размеров, а также в связных структурах, требует знания того, как система управляет областью хранения, которую она выделяет программам для данных, поэтому мы ознакомимся с принципами управления хранением и некоторыми базовыми механизмами решения этой задачи.

Кроме того, будут рассмотрены специфические методы, в которых используются механизмы выделения области хранения для программ на C++.

Данные, хранящиеся в памяти ЭВМ представляют собой совокупность нулей и единиц (битов). Биты объединяются в последовательности: байты, слова и т.д. Каждому участку оперативной памяти, который может вместить один байт или слово, присваивается порядковый номер (адрес). Все данные, необходимые для решения практических задач, подразделяются на несколько типов, причем понятие тип связывается не только с представлением данных в адресном пространстве, но и со способом их обработки. В любом языке программирования каждая константа, переменная, результат вычисления выражения или функции должны иметь определенный тип данных.

Тип данных – это множество допустимых значений, которые может принимать тот или иной объект, а также множество допустимых операций, которые применимы к нему. В современном понимании тип также зависит от внутреннего представления информации. Таким образом, данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

- внутреннее представление данных в памяти компьютера;
- объем памяти, выделяемый под данные;
- множество (диапазон) значений, которые могут принимать величины этого типа;
- операции и функции, которые можно применять к данным этого типа.

Исходя из данных характеристик, необходимо определять тип каждой величины, используемой в программе для представления объектов. Обязательное описание типа позволяет компилятору производить проверку допустимости различных конструкций программы. От выбора типа величины зависит последовательность машинных команд, построенная компилятором.

Существует два основных способа хранения информации в оперативной памяти.

Первый заключается в использовании глобальных и локальных переменных. В случае глобальных переменных выделяемые под них области памяти остаются неизменными во все время выполнения программы. Под локальные переменные программа отводит память из стекового пространства.

Второй способ заключается в использовании системы динамического распределения. При этом способе память распределяется для информации из свободной области по мере необходимости. Область свободной памяти находится между кодом программы с ее постоянной областью памяти и стеком.

<table>
<tr><td>Полустатические данные</td><td>Область стека</td><td>Старший адрес</td></tr>
<tr><td>Динамические данные(«куча»)</td><td>Область свободной памяти</td><td>,,,,,,,,,,,,,,,,,</td></tr>
<tr><td rowspan="2">Статические данные</td><td>Область глобальных переменных</td><td>,,,,,,,,,,,,,,,,,</td></tr>
<tr><td>Область программ, констант</td><td>Младший адрес</td></tr>
</table>

Динамическое размещение удобно, когда неизвестно, сколько элементов данных будет обрабатываться. По мере использования программой стековая область увеличивается вниз, то есть программа сама определяет объем стековой памяти. Например, программа с большим числом рекурсивных функций займет больше стековой памяти, чем программа, не имеющая рекурсивных функций, так как локальные переменные и возвращаемые адреса хранятся в стеках.

Память под саму программу и глобальные переменные выделяется на все время выполнения программы и является постоянной для конкретной среды. Память, выделяемая в процессе выполнения программы, называется динамической. После выделения динамической памяти она сохраняется до ее явного освобождения, что может быть выполнено только с помощью специальной операции или библиотечной функции.

Если динамическая память не освобождена до окончания программы, то она освобождается автоматически при завершении программы. Тем не менее, явное освобождение ставшей ненужной памяти является признаком хорошего стиля программирования.

В процессе выполнения программы участок динамической памяти доступен везде, где доступен указатель, адресующий этот участок. Таким образом, возможны следующие три варианта работы с динамической памятью, выделяемой в некотором блоке (например, в теле неглавной функции):

- Указатель (на участок динамической памяти) определен как локальный объект автоматической памяти. В этом случае выделенная память будет недоступна при выходе за пределы блока локализации указателя, и ее нужно освободить перед выходом из блока.

- Указатель определен как локальный объект статической памяти. Динамическая память, выделенная однократно в блоке, доступна через указатель при каждом повторном входе в блок. Память нужно освободить только по окончании ее использования.
- Указатель является глобальным объектом по отношению к блоку. Динамическая память доступна во всех блоках, где «виден» указатель. Память нужно освободить только по окончании ее использования.

Все переменные, объявленные в программе, размещаются в одной непрерывной области – сегменте данных (стеке). Они не меняют своего размера в ходе выполнения программы и называются статическими. Стек растет вниз и размера этого сегмента данных может быть недостаточно для размещения больших объемов информации.

Динамическая память – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы. Для хранения динамических переменных выделяется специальная область памяти, называемая «кучей».

Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программы, либо до тех пор, пока не будет освобождена выделенная под них память также с помощью специальных функций или операций. То есть время жизни динамических переменных – от точки создания до конца программы или до явного освобождения памяти.

Структура данных — это некоторый контейнер, который хранит данные в определенной форме представления .

Для реализации многих приложений правильный выбор структуры данных – единственное важное решение: когда выбор сделан, разработка алгоритмов не вызывает затруднений. Для одних и тех же данных различные структуры занимают различный объем памяти. Одни и те же операции с различными структурами данных приводят к алгоритмам различной эффективности. Для каждой структуры характерны выполняемые с ней операции и алгоритмы, используемые для этих операций. И операции, и алгоритмы учитывают не только форму представления данных, но и тип этих данных. Все обрабатываемые компьютером данные в конечном счете состоят из отдельных битов.

Типы позволяют указывать, как будут использоваться определенные наборы битов, а функции позволяют задавать операции, выполняемые над данными. Любые данные могут быть отнесены к одному из двух типов: основному (простому), форма представления которого определяется архитектурой ЭВМ, или сложному, конструируемому пользователем для решения конкретных задач.

Данные простого типа – символы и числа – это основные строительные блоки, из которых конструируются другие типы. Они могут быть представлены в ЭВМ битами и, поэтому, их дальнейшее дробление не имеет смысла. Для них определены элементарные операции: арифметические, логические, присваивания и др., определенные алфавитом языка. Из элементарных данных формируются структурированные (составные) типы данных, и первыми в этом списке конструкций идут массивы и записи. Без этих конструкций редко можно обойтись при практическом программировании, они являются основой для нетривиальных структур данных, обладая при этом простотой машинной реализацией. Все языки программирования высокого уровня включают массивы в качестве встроенных средств.

Массив– простая совокупность элементов данных одного типа. Отдельный элемент массива задается индексом. Массив может быть одномерным, двумерным и т.д. Помимо массивов есть и другие структуры данных, обладающие семантикой линейных последовательностей элементов: записи, списки, очереди и т. д. О них речь пойдет далее.

Набор основных операций над массивами:

- Индексирование— доступ к элементу для чтения или записи по его номеру в последовательности;

- Итерация— переход от текущего к последующему или предыдущему элементу;
- Вставка— добавление нового элемента в пространство между парой уже существующих, а также важный частный случай добавления в начало и конец последовательности;
- Удаление— уничтожение элемента последовательности, которое тоже имеет важный частный случай удаления из начала и конца.

Семантика массива позволяет представлять его в виде непрерывного блока памяти, разделяемой на отдельные переменные, адреса которых образуют арифметическую прогрессию.

Отсюда следует, следующее:

- По индексу элемента процессор может очень быстро определить соответствующий адрес в памяти и произвести операцию чтения или записи значения. Более того, время выполнения этой операции не связано с размером массива (на фундаментальном уровне), а значит, ее алгоритмическая сложность минимальна.
- Переход от текущего элемента к следующему заключается в увеличении целочисленного индекса на константное значение. Эта операция также должна выполняться за время, независящее от размера массива. Ее алгоритмическая сложность опять же минимальна Переход к предыдущему элементу удовлетворяет аналогичным требованиям.

- При этом интересным является тот факт, что в отличие от некоторых других структур данных, поддерживающих итерацию за постоянное время, массив позволяет осуществлять переход на несколько элементов вперед (назад) за время, не зависящее от их количества. Эта возможность появляется благодаря линейному соответствию между индексами и адресами элементов, а также тому факту, что арифметические операции и доступ к ячейке памяти считаются постоянными по времени. Иначе говоря, зная начальный адрес представления массива в памяти и размер памяти для хранения одного элемента, можно вычислить адрес хранения любого элемента массива (а значит получить к нему доступ) за одно и то же время.

Как видно, простые операции доступа выполняются весьма быстро. Алгоритмы, в которых заранее известен требуемый (максимальный) размер последовательности, не получают никаких, связанных с этим, дополнительных расходов. Ситуация радикально ухудшается, когда по тем или иным причинам программа в процессе работы должна изменять размер массива, применяя операции вставки или удаления. Такие операции возможны, когда массив в памяти объявлен динамической переменной.

Под физическим размером массива будем понимать количество элементов, которые он в принципе способен хранить на данный момент. Под логическим размером — только то, что актуально используется. Предположим, что эти элементы имеют наименьшие положительные индексы: 0, ... , n − 1, где n — логический размер. Для того, чтобы произвести вставку элемента в некоторую позицию нужно сначала сдвинуть все элементы, начиная с нее и заканчивая логическим концом массива, на одну позицию вправо. При этом логический размер увеличивается на единицу. Поскольку в худшем случае добавления элемента в начало, придется сдвигать весь массив, сложность операции вставки существенно увеличивается.

Для того, чтобы сохранялась возможность применения индексной арифметики массив не должен иметь «дыр», образовавшихся в результате операции удаления. Значит, эта операция также должна приводить к сдвигу под последовательности элементов на одну позицию влево, которая, в худшем случае, совпадет со всем массивом без одного элемента. Таким образом, операция удаления также требует от процессора увеличения количества операций.

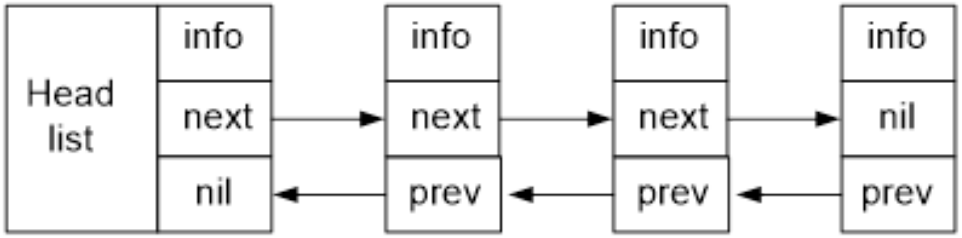
Запись - совокупность (массив) элементов данных разного типа. Также, как и в массиве, элемент этой структуры имеет постоянный размер для ее размещения в памяти. В простейшем случае запись содержит постоянное количество элементов, которые называют полями. Совокупность записей одинаковой структуры называется файлом данных. (Файлом называют также набор данных во внешней памяти, например, на магнитном диске). Для того, чтобы иметь возможность извлекать из файла отдельные записи, каждой записи присваивают уникальное имя или номер, которое служит ее идентификатором и располагается в отдельном поле. Этот идентификатор называют ключом. И все операции поиска, вставки и удаления осуществляются, в отличие от массива, не по индексу элемента, а по этому ключу. Такие структуры данных как массив или запись занимают в памяти ЭВМ постоянный объем, поэтому их называют статическими структурами. Обобщим вычислительную сложность алгоритмов обработки данных, представленных статическими структурами: доступ – простой, поиск, вставка и удаление зависит от количества элементов.

К сожалению, «расширение объема памяти» для статических структур задача достаточно трудоемкая, поэтому широкое применение получили динамические структуры, т.е. структуры, которые могут изменять свою длину.

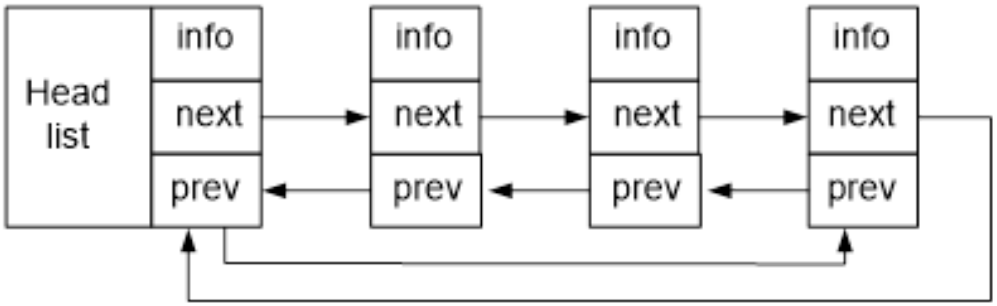
К ним относятся списки, очереди, деревья и др. Некоторые из них имеют линейную форму представления в памяти, другие, такие, как деревья, для размещения в памяти своих элементов требуют нелинейного адресного пространства.

Связные списки. Важная линейная структура данных, которая, на первый взгляд, похожа на массив, но отличается распределением памяти, внутренней организацией и способом выполнения основных операций вставки и удаления. Связные списки бывают однонаправленными (односвязными) и двунаправленными (двусвязными). Односвязный список – это сеть узлов, каждый из которых содержит данные и указатель на следующий узел в цепочке (next). Также есть указатель на первый элемент – head. Если список пуст, то он указывает на null (next=NULL). Односвязный список не слишком удобен, т. к. из одной точки есть возможность попасть лишь в следующую точку, двигаясь тем самым в конец.

В двусвязном линейном списке, кроме указателя на следующий элемент (next) есть указатель и на предыдущий (prev). Возможность двигаться как вперед, так и назад полезна для выполнения некоторых операций, но дополнительные указатели требуют задействования большего количества памяти, чем таковой необходимо в эквивалентном односвязном списке.



Подвидом линейного списка является кольцевой список. Сделать из односвязного списка кольцевой можно добавив всего лишь один указатель в последний элемент, так чтобы он ссылался на первый (next=head). А для двусвязного потребуется два указателя: на первый и последний элементы.



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.

Основными операциями над линейными связными списками являются:

- вставка элемента (в начало, в конец, в указанное место);
- удаление указанного элемента
- поиск указанного элемента;
- проверка списка на отсутствие элементов.

Связные списки используются для реализации файловых систем, хэш-таблиц и списков смежности. На основе линейных списков строятся другие линейные динамические структуры.

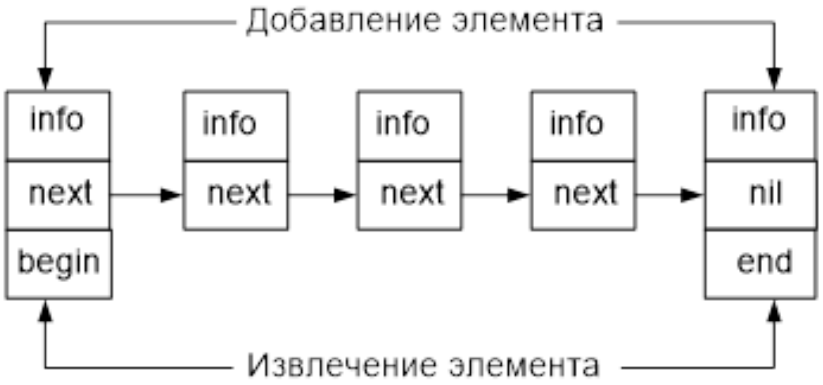
Стек. Как работает опция «отменить», присутствующая практически в каждом приложении? Вначале сохраняются предыдущие состояния приложения (определенное их количество) в памяти в таком порядке, что последнее сохраненное появляется первым. Именно такой способ обработки данных и реализует стек. Стек – абстрактный тип данных, построенный по типу однонаправленного линейного списка элементов, организованных по принципу LIFO (англ. last in — first out, т.е. «последним пришёл — первым вышел»). Данный метод придумал Алан Тьюринг. Примером стека может быть стопка книг, расположенных в вертикальном порядке. Для того, чтобы получить книгу, которая где-то посередине, нужно будет удалить все книги, размещенные на ней. Таким образом, и дополнение новых данных, и извлечение их из стека всегда выполняется с одной стороны – с его вершины.

Основные операции. В общем, стек — это односвязный список, для которого определены только две операции: добавление (push) и удаление (pop) из начала списка. Возможны также операции проверки на пустоту стека, получение всего его содержимого и получение ве.

Очередь. Как и стек, очередь – это линейная структура данных, которая хранит элементы последовательно. Единственное существенное различие заключается в том, что вместо использования метода LIFO, очередь реализует метод FIFO (First in First Out, «первым пришел – первым ушел»). Идеальный пример этих структур в реальной жизни – очереди людей за чем-то. Если придет новый человек, он присоединится к линии с конца, а не с начала. А человек, стоящий впереди, первым получит требуемое и, следовательно, покинет очередь.

В отличие от стека вставка элемента осуществляется только в конец (endqueue), а удаление (delqueue) осуществляется только из начала этого списка. Возможны также операции получения первого элемента и проверки очереди на пустоту.

Дек. Дек (deque — double ended queue, «двухсторонняя очередь») – стек с двумя концами. Дек можно определять не только как двухстороннюю очередь, но и как стек, имеющий два конца. Это означает, что данный вид списка позволяет добавлять элементы в начало и в конец, и то же самое справедливо для операции извлечения.



Эта структура одновременно работает по двум способам организации данных: FIFO и LIFO. Поэтому ее допустимо отнести к отдельной программной единице, полученной в результате суммирования предыдущих видов линейного списка.

Преимущество этих линейных динамических структур перед массивами в том, что несмотря на сложность их алгоритмов доступа и поиска, алгоритмы вставки и удаления элементов имеют и в худшем случае простейший порядок.

А это имеет существенное преимущество в условиях сильно изменяющихся в процессе работы (динамике) структур больших данных. Следует заметить, что линейные списки служат основой для построения более сложных нелинейных структур, например, бинарных деревьев, хеш-таблиц, списков смежности, при работе с графами и т.п.

Список литературы

1. Овсянников, А. В. Алгоритмы и структуры данных: учебно-методический комплекс для специальности 1-31 03 07 «Прикладная информатика (по направлениям)». Ч. 1 / А. В. Овсянников, Ю. А. Пикман; БГУ, Фак. социокультурных коммуникаций, Каф. информационных технологий. – Минск: БГУ, 2015. – 124 с.
2. Никлаус ВиртАлгоритмы и структуры данных. Новая версия для Оберона + CD / Пер.с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010. – 272 с.: ил.