**РАЗДЕЛ 1. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ (АТД)**

1. **Концепция АТД**

Прежде чем перейти к описанию структур данных в языке программирования Golang следует начать с понятий «*данные»*, «*тип данных*» и «*структура данных*». В самом общем понимании, данные — это любой неструктурированный набор (коллекция) символов, которые собираются и обрабатываются с какой-либо целью, прежде всего для извлечения информации для последующего получения знания и смыслов. . Многообразие данных предопределило появление категории тип данных. Тип данных следует понимать как атомарное, неделимое единство данных, определяемое либо стандартом языка программирования либо пользователем и представляющее собой совокупность объектов реального мира на уровне их наиболее существенных параметров. Обобщение опыта применения различных типов данных в различных языках программирования привело к возникновению категории «абстрактный тип данных».

Типы данных впервые были описаны Д. Кнутом в его классичесом труде «Искусство программирования» [Кнут], в котором автор описал *структуры данных*, определяемые как способы организации данных внутри программы. Вместе с описанием самих структур данных Кнут приводит «алгоритмы обработки» этих структур на языке специальных терминов, отражающих те или иные действия с элементами этой структуры, такие как:

* размер структуры (size),
* добавление нового элемента (append),
* выталкивание (pop) и вталкивание (push) элементов и т. д.

Такой подход стал одним из побудительных мотивов для осознания важности концепции абстрактных типов данных (АТД). В большинстве литературных источников абстрактный тип данных понимается как математическая модель, определяющая набор значений данных (carrier set) и набор методов работы с этими данными (method set). Вот несколько примеров АТД:

Boolean — несущий набор логического АТД представляет собой набор значений {true, false}. В набор допустимых операций (методов) входят: отрицание, конъюнкция, дизъюнкция, импликация, эквивалентность и некоторые другие.

Булев - несущее множество Булева ADT является множеством { true, false }. Операции по этим значениям являются отрицанием, объединением, дизъюнкцией, условной и, возможно, некоторыми другими

Integer — несущий набор Integer АТД представляет собой набор {..., -2, -1, 0, 1, 2, ...}, а набор методов включает сложение, вычитание, умножение, деление, остаток, равно, меньше, больше и так далее.

String — несущий набор строкового АТД представляет собой набор всех конечных последовательностей символов некоторого алфавита, включая пустую последовательность (пустую строку). Набор методов включает конкатенацию, длину, подстроку, индекс и т. д.

Bit string. Несущим набором АТД строки битов является набор всех конечных последовательностей битов, включая пустые строки битов. Набор методов АТД строки битов включает в себя дополнение (которое переворачивает все биты), сдвиги (которые поворачивают строку битов влево или вправо), соединение и дизъюнкцию (которые объединяют биты в соответствующих местах в строках), а также конкатенацию и усечение.

Автор выдающейся работы «Совершенный код» С. Макконнелл существенно расширил понятие АТД, подчеркивал тот факт, что абстрактные типы данных «позволяют работать с сущностями реального мира, а не с низкоуровневыми сущностями реализации» [Mac, C. 122 ]. В частности, такими сущностями являются структуры данных для организации вычислительного процесса. Упрощенную классификацию абстрактных структур данных можно представить в таком виде (рис.1.1.)



Рис. 1.1. Классификация структур данных

Рассмотрим подробнее классификацию. Отметим, что каждая из этих структур имеет свои преимущества и недостатки, что предопределяет необходимость их тщательного анализа с точки зрения затрат памяти компьютера и времени доступа к их элементам. В данной работе основное внимание сосредоточено на структурах данных, представленных в оперативной памяти. Прежде всего, структуры данных делятся на линейные и нелинейные. В линейной структуре данных ее элементы являются смежными, то есть данные упорядочены последовательно. Такие конструкции достаточно просты в реализации. В то же время линейные структуры данных не обеспечивают эффективного использования памяти.

В *нелинейных структурах данных* организация элементов не является последовательной. Элементы данных в нелинейной структуре могут быть соединены с несколькими другими элементами данных с целью отражения особых отношений между ними. Кроме того, в нелинейных структурах невозможно пройти по элементам за один проход. К нелинейным структурам данных относятся *карты, словари,* *деревья* и *графы*. К однородным структурам данных относятся большинство указанных структур данных, к неоднородным - структуры, которые состоят из данных разной природы. Например, основу списков составляют узлы, представляемые структурами, которые включают два поля: численное значение и адрес следующего узла. Типичными примерами неоднородной структуры являются *словари, карты и хэш-таблицы*. Рассмотрим основные абстрактные типы данных по принципу линейности.

* 1. **Линейные абстрактные типы данных** 
     1. **Массив и срез**

Начнем с фундаментального абстрактного типа данных - массив. Фундаментальность массива, как структуры данных, заключается в их прямом соответствии системам памяти на всех компьютерах. Для извлечения содержимого слова из памяти машинный язык требует указания адреса. Таким образом, всю память компьютера можно рассматривать как массив, где адреса памяти соответствуют индексам. Большинство процессоров машинного языка транслируют программы, использующие массивы, в эффективные программы на машинном языке, в которых осуществляется прямой доступ к памяти.

Массив является фиксированным набором однотипных данных, которые хранятся в виде непрерывной последовательности, индексация элементов которых может начинаться с 0 или 1. Индексация может начинаться с 0 или 1. При создании и инициализации массив объявляется через идентификатор или через указатель адреса начального (0 или 1) элемента (рис.1.2).

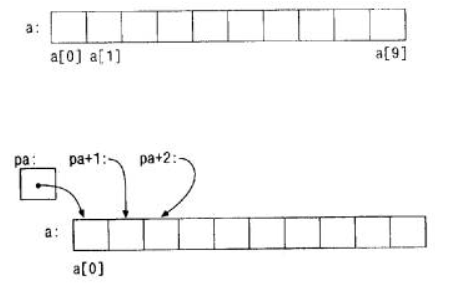


Рис. 1.2. Способы объявления массива (*pa*  - адрес ячейки *a[0]*)

Выбор вида объявления предопределяет способы доступа к отдельным элементам. Запись a[i], где а - идентификатор, отсылает нас i-му элементу массива. В то же время массив может быть объявлен с помощью указателя, который отсылает программный код на адрес начального элемента массива. В большинстве языков, использующих концепцию указателей массив задается выражением: \*pa, а реальный адрес в 16-ричном выражении определяется через символ &: pa = &a[0]. Если *pa* указывает на некоторый элемент массива, то *pa+1* по определению указывает на следующий элемент *pa+i —* на *i*-й элемент после *pa*, a *pa-i —* на *i-*й элемент перед *pa*. Таким образом, если *pa* указывает на *a[0]*, то \*(pa+1) – это содержимое *a[1], a+i —* адрес a[i], a \*(pa+i) — содержимое a[i].

Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. Указатель - это переменная, поэтому можно написать *pa = a*, но имя массива не является переменной, и записи вроде a = pa не допускаются.

Рассмотрим наиболее общие свойства массива как абстрактного типа данных:

* Bce элементы массива принадлежат одному типу (однородность);
* Paзмер массива задается один раз и не меняется на протяжении работы (постоянство);
* Ко всем элементам массива имеется одинаковый доступ (равнодоступность);
* Bce элементы располагаются последовательно в ячейках оперативной памяти (последовательость расположения);
* Элементы массива однозначно идентефицируются своими индексами (индексация);
* Индексы должны быть простым порядковым типом данных упорядоченность.

Напомним, что любой абстрактный тип данных состоит из набора значений и набора методов. Для массива основными методами являются:

* Получить элемент с номером N;
* Записать элемент с номером N;
* Получить размер массива.

К достоинствам массивов относят

* Доступ за фиксированное время к любому элементу;
* Пам'ять тратится только на пам'ять.

К недостаткам – статичность, неизменность структуры.: в массиве нельзя произвольно менять объявленный размер.

Указанный недостаток может быть преодолен введением такого абстрактного типа данных как *срез* (*slice*). Срез — это гибкая и расширяемая структура данных, состоящая из нескольких элементов одного типа. Как и массивы, срез индексируется и имеет размер (длину), который может программно регулироваться (увеличиваться и уменьшаться) за счет наличия такого свойства как емкость. Емкость среза — это количество элементов в базовом массиве, считая от первого элемента в срезе. Длину среза можно увеличивать, повторно нарезав срез, если он имеет достаточную емкость. Срез может состоять из трех элементов: указатель, указывающий на первый элемент среза, длину (количество элементов в срезе) и емкость - максимальный размер, до которого срез может расширяться (рис. 1.3.):

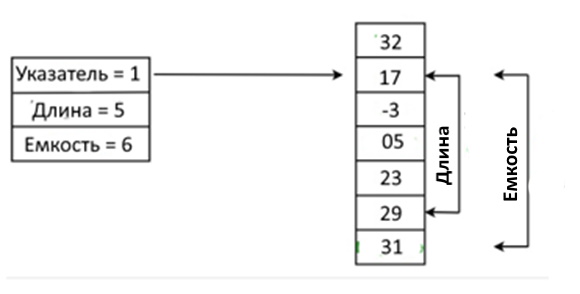


Рис. 13. Срез и его компоненты

Основными методами для среза являются операции добавления, удаления, определение длины среза, изменение емкости. Более подробно программная реализация среза в языке Golang (объявление, инициализация, основные методы работы с этим типом данных) изложено в следующем разделе.

**Связный список (Linked List)**

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения и удаления. Список, отражающий отношения соседства между элементами, называется линейным. Длина списка равна числу элементов, содержащихся в списке. Если компонента не связана ни с какой другой, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем - null.

Существует три распространенных типа связных типов:

Односвязный список;

Двухсвязный список;

Кольцевой связный список

Односвязный список. На рис. 1.4. приведена структура односвязного списка. На нем поле item - информационное поле, данные, next - указатель на следующий элемент списка. Каждый список должен иметь особый элемент, называемый указателем начала списка или головой списка, который обычно по формату отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак null, свидетельствующий о конце списка (рис. 1.4):

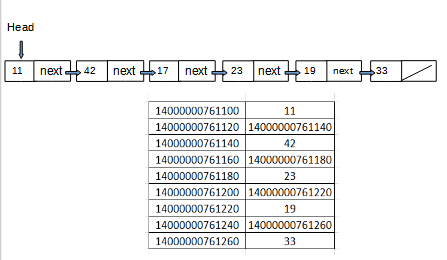


Рис. 1.4. Структура односвязного списка

Определенный таким образом односвязный список обладает рядом свойств:

Размер списка — количество элементов в нем, исключая последний «нулевой» элемент, являющийся по определению пустым списком.

Тип элементов — тот самый тип, над которым строится список; все элементы в списке должны быть этого типа.

Отсортированность — список может быть отсортирован в соответствии с некоторыми критериями сортировки (например, по возрастанию целочисленных значений, если список состоит из целых чисел).

Возможности доступа — некоторые списки в зависимости от реализации могут обеспечивать программиста селекторами для доступа непосредственно к заданному по номеру элементу.

Сравниваемость — списки можно сравнивать друг с другом на соответствие, причем в зависимости от реализации операция сравнения списков может использовать разные технологии.

Ниже приведен некоторый набор функций, которые можно производить над АТД списка:

* Инициализировать список;
* Вставить новый элемент;
* Удалить какой-либо элемент;
* Найти k-тый элемент;
* Прочитать следующий элемент;
* Печать элементов.

Двусвязный список. В некоторых случаях более удобно указывать ссылки с обеих сторон. В этом случае возможен как прямой, так и обратный доступ. Он определяется как последовательность узлов, в которой каждый узел указывает области памяти. Каждый узел разделен на три поля: item - поле данных, next – адрес следующего поля или прямое поле и предыдущее или обратное поле. Поле данных используется для хранения значений данных. Поле prev – используется для хранения адреса предыдущего элемента, next - для хранения адреса следующего элемента. Первый узел предыдущего поля и последний узел следующего поля всегда равны null (рис. 1.5.).



Рис. 1.5. Структура двусвязного списка

Свойства и функции для двусвязного списка практически те же, что и для односвязного списка. Основное преимущество двусвязного списка заключается в упрощении многих операций; основной недостаток – затраты дополнительной памяти на указатели адресов.

Кольцевой связный список. Разновидностью рассмотренных видов линейных списков является кольцевой список, который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются, как показано на рис. 1.6.

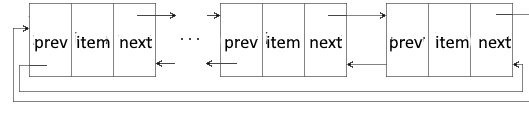


Рис. 1.6. Структура двусвязного списка

При работе с кольцевыми списками также упрощаются некоторые процедуры; однако, при просмотре такого списка следует принять некоторые меры предосторожности, чтобы не попасть в бесконечный цикл [www.algolist.manual.list].

При работе с такими списками несколько упрощаются некоторые процедуры. Однако, при просмотре такого списка следует принять некоторых мер предосторожности, чтобы не попасть в бесконечный цикл.

**Стек (Stack) и Очередь (Queue)**

Стек представляет собой динамический, постоянно меняющийся набор данных, добавление и удаление элементов которого выполняются исключительно по принципу «последний вошел, первый вышел» *(Last-In-First-Out или LIFO* (рис. 1.7). Работа стека напоминает манипуляции со стопкой книг, которую формируют действия по удалению и добавлению книг.

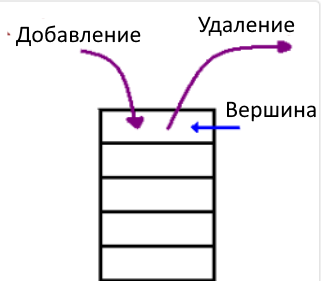


Рис. 1.7. Структура стека

Стеки находят широкое применение в различных сценариях обработки данных, например, реализация механизма «отмены» в текстовых редакторах; эта операция выполняется путем хранения всех текстовых изменений в стеке. Стек часто используется в организации вызовов методов и рекурсий.

Основные операции, которые можно выполнять над стеком:

* Добавить новые данные;
* Удалить данные из стека;
* Возвращать (без удаления) верхнее значение;
* Проверить состояние заполнения стека.

Во многих языках программирования стек создается на основе либо массива либо связного списка. В языке Golang может применяться концепция интерфейсв, что будет подробно рассмотрено далее.

В отличие от *стека* АТД *очередь,* с которым стек имеет много общего, элементы добавляются (enqueue) и удаляются (dequeue) с разных концов. Такой метод называется *«первый вошел, первый вышел»* (First-In-First-Out или FIFO). То есть элементы из очереди забираются в том же порядке, в каком что вкладываются (рис.1.8).

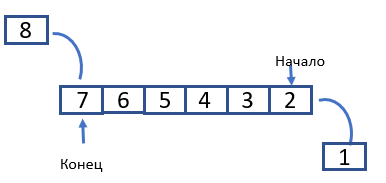


Рис.1.8. Структура данных «очередь»

Функции для манипулирования компонентами очереди представлены ниже.

* Создать новую пустую очередь;
* Добавить новый элемент в конец очереди;
* Удалить из очереди передний элемент;
* Проверить очередь на пустоту;
* Проверить очередь на заполнение;
* Определить длину (количество элементов) в очереди.

Тип данных *очередь* может иметь несколько разновидностей:

*Круговая очередь* представляет собой расширенную версию [обычной очереди](https://www.programiz.com/data-structures/queue), в которой последний элемент подключен к первому элементу, вследствие чего образуется кругообразная структура. Круговая очередь решает основное ограничение обычной очереди – неэффективное использование памяти.

*Приоритетная очередь* — это особый тип очереди, в котором каждый элемент связан со значением приоритета. И элементы обслуживаются на основе их приоритетности. То есть в первую очередь обслуживаются более приоритетные элементы. Однако, если элементы с одинаковым приоритетом встречаются, они обслуживаются в соответствии с их порядком в очереди. В обычной очереди реализуется правило «первым пришел-первым вышел», тогда как в очереди с приоритетом значения удаляются на основе приоритета. Элемент с наивысшим приоритетом удаляется первым [[Очередь как структура данных, виды, реализация, применение,... (intellect.icu)](https://intellect.icu/ochered-kak-struktura-dannykh-vidy-realizatsiya-primenenie-tsiklicheskaya-ochered-4494)].

Хранение данных в очереди находит широкое применение при планировании задач центрального процессора, принтера, других устройств вывода. Метод FIFO используется при обработке прерываний.

**Нелинейные абстрактные типы данных**

Этот подраздел касается нелинейных абстрактных структур данных, из которых *деревья* и *графы* являются наиболее распространенными []. Напомним, что нелинейные структуры данных позволяют выразить более сложные иерархические отношения между элементами в отличие от простых отношений в линейных структурах. Рассмотрим эти структуры в самых общих чертах, то есть как абстрактные типы данных.

**Деревья как АТД.** В абстрактном представлении *деревья* понимаются как нелинейные структуры данных, состоящие из узлов, представляющих собой иерархию отношений (родители-потомки). Каждая вершина дерева при наличии потомков соединяется с ними ориентированными ребрами (рис. 1.9).

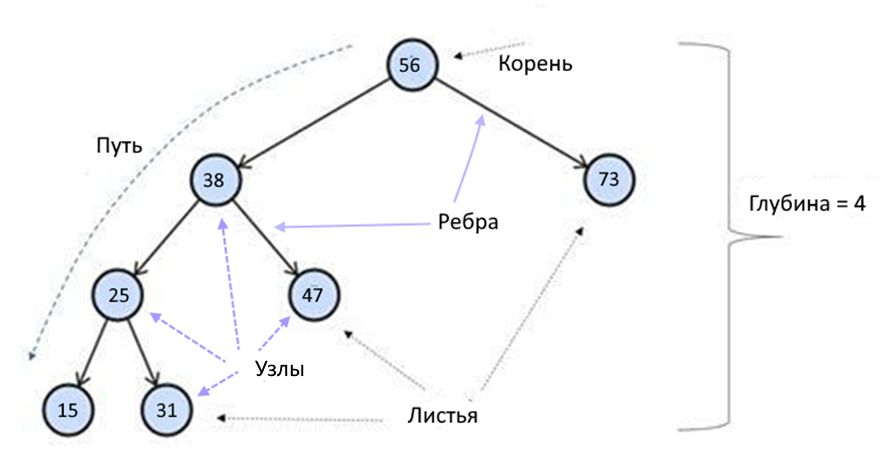


Рис. 1.9. Основная терминология дерева

Приведем определение основных понятий теории деревьев.

Корень: корень дерева является единственным узлом без каких-либо входящих рёбер. Это верхний узел дерева;

Узел: это фундаментальный элемент дерева. Каждый узел имеет данные и две ссылки, которые могут указывать на нуль или его потомков;

Ребро: Это также фундаментальная часть дерева, которая используется для соединения двух узловые точки.

Путь: Путь - это упорядоченный список узлов, соединенных рёбрами.

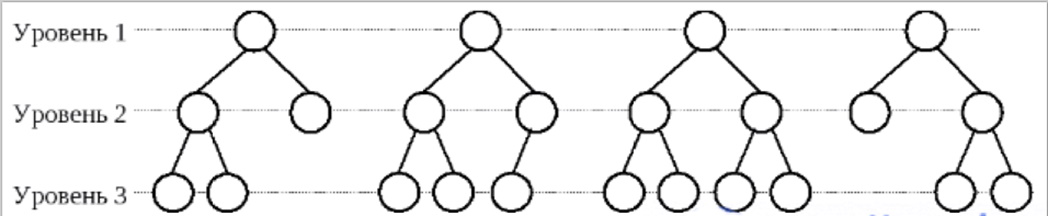
Лист: Листовой узел - это узел, не имеющий потомков.

Высота дерева: Высота дерева - число рёбер на самом длинном пути между корнем и листом.

Уровень узла: Уровень узла - число рёбер на пути от корневой узел этого узла

При решении ряда вычислительных задач возникает необходимость в организации данных в различных видах *деревьев*, однако особенно наиболее хорошо изученными и распространенными являются *двоичные деревья* (binary tree) [DSA].

В рамках концепции абстрактных типов данных *двоичное* *дерево* представляет собой множество связанных узлов, в котором каждый узел содержит значение (собственно элемент данных) и имеет не более двух потомков [DSA]. Это означает, что степень двоичного дерева равна нулю, одному или двум. С позиций АТД двоичные деревья содержат значения элементов типа T. Несущим множеством этого типа является множество всех двоичных деревьев, у которых вершины имеют значение типа Т. Несущее множество, таким образом, включает пустое дерево, деревья только с корнем со значением типа Т, деревья с корнем и левым потомком, деревья с корнем и справа дочерний, и так далее (рис.1.10).

 Fig.1.10. Binary tree species

Общими операциями для двоичных деревьев как абстрактных типов данных являются:

* Поиск элемента с указанным значением k;
* Вставка k-того элемента;
* Удаление k-того элемента;
* Нахождение максимального значения, хранящегося в дереве;
* Нахождение минимального значения, хранящегося в дереве.
* Нахождение количества уровней дерева.

Реализация основных алгоритмов для бинарных деревьев поиска представлена в разделе 8.

**Графы как АТД.** В самом общем определении граф G задается множеством узлов (вершин) {V} и множеством ребер {E}, соединяющих между собой все или часть этих вершин, иными словами, граф G полностью задается парой {V, E}. Если ребра ориентированы, что обычно показывается стрелкой, то они называются *дугами*, и граф с такими ребрами называется *направленным (ориентированным) графом* (рис. 1.11 а). Если ребра не имеют ориентации, то граф называется *ненаправленным* (*неориентированным*):

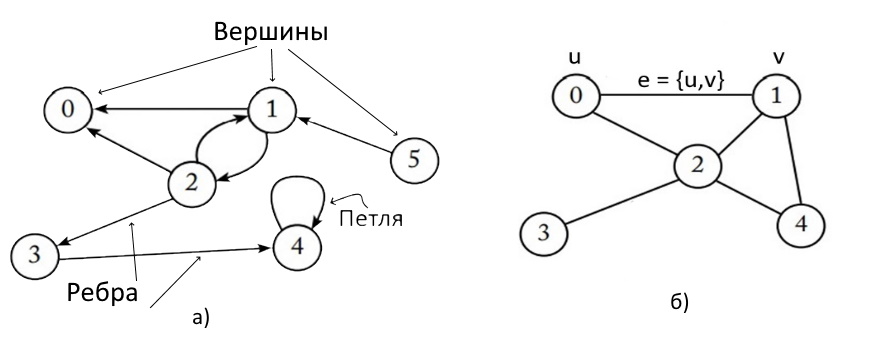


Рис. 1.11. Вид графа а)- ненаправленного; б) – направленного

Вершины и ребра называются *элементами* графа, число вершин в графе – *порядком*, число ребер – *размером* графа. Вершины (u,v) называются *концевыми* вершинами ребра e = {u,v}, две концевые вершины одного и того же ребра называются *соседними*. Два ребра называются *смежными*, если они имеют общую концевую вершину. Два ребра называются *кратными*, если множества их концевых вершин совпадает. Ребро называется *петлей*, если его концы совпадают, то есть, e = {u,u}. Если вершина vi является началом или концом ребра ek, то они (вершина и ребро) – *инцидентны*. Число ребер, инцидентных вершине, называется *степенью вершины* (рис. 1.12.).

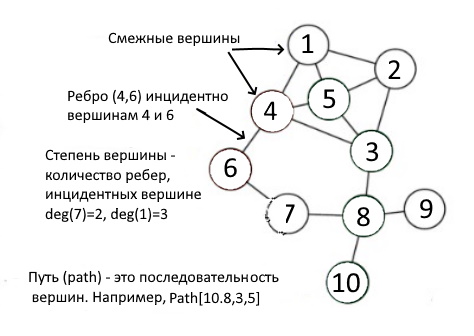


Рис.1.12. Основные параметры графа

Существует множество видов графов, среди которых наиболее значимыми являются:

* *нулевой* граф, в котором нет ребер;
* *тривиальный* граф, в котором только одна вершина;
* *ненаправленный* граф, в котором ребра не имеют направления;
* *направленный* граф, в котором ребра имеют направления;
* *полный* граф, в котором от каждого узла есть ребро к другому узлу;
* *взвешенный* граф, в котором ребра указаны с весом.ром ребра не имеют никакого

Соответственно существует множество операций с элементами графов, в частности:

* + - добавить ребро между двумя вершинами;
    - операция удаления ребра при сохранении всех вершин графа;
    - операция добавления вершины к множеству вершин;
    - операция добавления этого ребра к множеству ребер;
    - операция Дейкстра, определяющая минимальное расстояние между двумя заданными узлами.

Более подробно алгоритмы графов рассматриваются в РАЗДЕЛЕ 9.