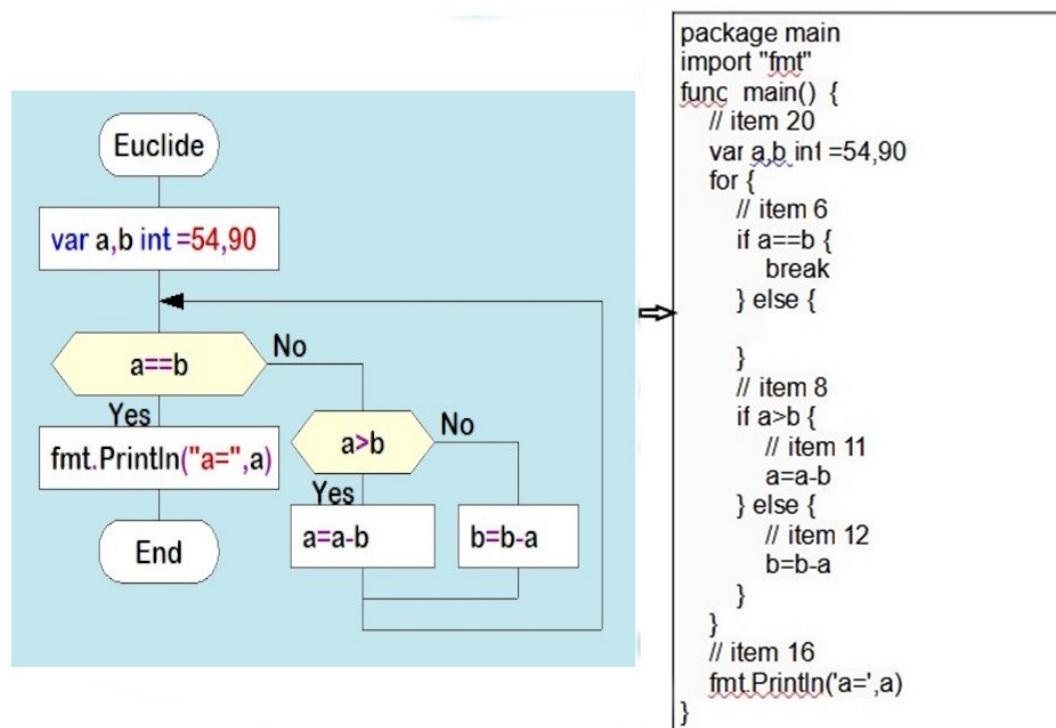
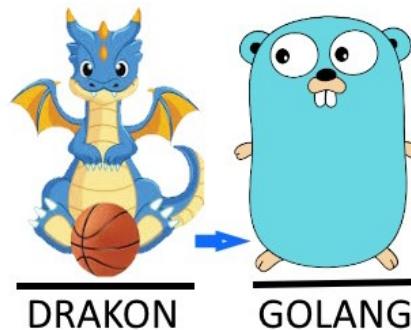


Иванов С.А.

СТРУКТУР ДАННЫХ И АЛГОРИТМЫ. ГИБРИДНЫЙ ПОДХОД



Автогенерация дракон--диаграмм в Golang код

ВВЕДЕНИЕ

Основу процессов познания, понимания и конструирования природной, неприродной и виртуальной реальностей так или иначе составляют ДАННЫЕ, которые можно понимать как неструктурированную информацию. Для структуризации этих данных с целью получения нового знания или новых смыслов необходима их обработка по определенному плану действий. В более строгом определении структура данных – это форма представления свойств и связей предметной области, ориентированная на выражение описания данных средствами формальных языков. Технологически структура данных — это контейнер, информация в котором скомпонована определенным образом, устанавливающим способ размещения данных в памяти компьютера для быстрого и эффективного доступа к ним.

В свою очередь алгоритмы представляют собой наборы инструкций для решения вычислительных проблем путем обработки структур данных. Структуры данных и процессы их обработки по тем или иным алгоритмам неразрывно связаны друг с другом. В одних задачах выбор алгоритма диктуется структурой данных, в других – структура данных определяет выбор алгоритма. Один из основоположников становления информатики как академической науки Н. Вирт сформулировал эту связь в названии книги: «Алгоритмы + структуры данных = программа» [Вирт].

Многообразие предметного, непредметного и виртуального миров и процессов, происходящих в них, предопределяют возникновение множества проблемных ситуаций, эффективное решение которых которые предполагает выбор структур данных и алгоритмов их обработки. Решение считается эффективным , если оно решает проблему в рамках известных ограничений ресурсов. Прежде всего имеется в виду *общее пространство*, доступное для хранения данных, а также *время*, отведенное на выполнение каждого алгоритма. При выборе структуры данных для решения задачи следует,

во-первых, определить основные операции, которые должны поддерживаться, во-вторых, количественно оценить ограничения ресурсов по каждой операции алгоритма (занимаемой памяти и времени выполнения операций).

Структура данных требует определенного объема памяти для каждого хранимого элемента данных, определенного количества времени для выполнения одной базовой операции и определенных усилий по программированию. Каждая задача имеет ограничения по доступному пространству и времени. Каждое решение проблемы использует базовые операции в некоторой относительной пропорции, и процесс выбора структуры данных должен это учитывать. Только после тщательного анализа характеристик вашей задачи вы сможете определить наилучшую структуру данных для этой задачи [].

Исходя из представленных соображений в Пособии даются базовые сведения о наиболее часто используемых структурах данных и алгоритмов, поскольку они составляют основу профессиональной деятельности программиста любого уровня. Уделяется значительное внимание вопросам оценки эффективности структур данных и алгоритмов с точки зрения занимаемой памяти и времени выполнения базовых операций. Организация структур данных и изучение базовых алгоритмов выполняется на языке программирования Golang и сопровождается поясняющими иллюстрациями. В ряде важных случаях применяется технология визуализация алгоритмов на основе графического языка Drakon.

В **РАЗДЕЛЕ 1** раскрывается смысл основных понятий в дисциплине *Структура данных и алгоритмы*: данные, тип данных, абстрактные типы данных, алгоритмы. Высокий уровень компетентности в сфере разработки эффективных алгоритмов необходим каждому программисту не только при решении практических проблем, связных с применением ИТ-технологий. Владение знаниями в области структуры данных и алгоритмов является важнейшей составляющей при собеседовании при приему на работу в крупные ИТ-компании.

Во ВТОРОМ РАЗДЕЛЕ излагаются основы языка программирования Golang в объеме, достаточном для знакомства и практического овладения знаниями и навыками в области структуры данных и алгоритмов. Разумеется, для прочтения этой главы предполагается наличие базовых понятий и определенный уровень программирования.

В РАЗДЕЛЕ 3 дается краткое описание визуального языка DRAKON, позволяющего представлять сколь-угодно сложные алгоритмы в виде дракон-диаграммы с помощью графоэлементов, соответствующих основным конструкциям языка Golang. Приводится технология автоматизированного преобразования дракон-диаграммы в программный код в редакторе DRAKON WEB Editor с последующей реализацией в оболочке Visual Studio Code. Собственно, в этом и заключается гибридный, двухэтапный подход: на первом этапе составляется дракон-диаграмма, гоафоэлементы которой заполняются конструкциями языка Golang, на втором — автоматически генерируется программный код, реализуемый в какой-либо программной средею

В РАЗДЕЛЕ 4 представлены конструкции языка Golang, реализующие основные абстрактные типы данных, описанные в Разделе 1: массив, срез, списки и т. д. Приводятся пояснения в отношении использования соответствующих конструкций языка Golang.

В РАЗДЕЛЕ 5 изложены основные представления об анализе сложности алгоритмов и об оценке и эффективности алгоритмов с позиций использования компьютерной памяти (пространство) и затрат времени. Теоретические представления об оценке сложности алгоритмов имеют определяющее значение при выборе наиболее эффективного из них.

РАЗДЕЛ 6 посвящен базовым алгоритмам сортировки данных. Каждый алгоритм представлен с помощью соответствующей дракон-диаграммы, сопровождающейся необходимыми пояснениями о процессе обработки и оценкой сложности.

В РАЗДЕЛЕ 7 рассматриваются базовые алгоритмы поиска элементов в различных структурах данных (срезы, списки). Кроме того, раздел включает описание и дракон-диаграммы алгоритмов хэширования, позволяющие значительно сократить время поиска за счет уменьшения количества сравнений значений элементов.

РАЗДЕЛ 8 содержит понятийную информацию о двоичных деревьях - нелинейных структурах данных, включая двоичные деревья поиска и самобалансирующиеся деревья. В разделе рассматриваются базовые ДРАКОН-диаграммы алгоритмов вставки новых, поиска узлов с указанными значениями и удаление узлов.

В РАЗДЕЛЕ 9 излагаются базовые сведения о наиболее сложных структурах данных — графах. Приводятся основные понятия теории графов и основные их виды. Рассматриваются алгоритмы представления графов, их обхода, выбора пути между вершинами.

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 классическом 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 АДТ 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» [Мас, С. 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.

1.2. Линейные абстрактные типы данных

1.2.1. 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).

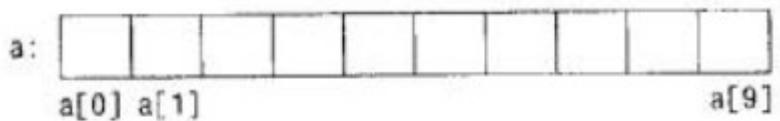


Рис. 1.2. Способы объявления массива (pa - адрес ячейки $a[0]$)

Выбор вида объявления предопределяет способы доступа к отдельным элементам. Запись $a[i]$, где a - 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 , а $pa-i$ — на i -й элемент перед pa . Таким образом, если pa указывает на $a[0]$, то $*(pa+1)$ — это содержимое $a[1]$, $a+i$ — адрес $a[i]$, а $*(pa+i)$ — 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:

- Все элементы массива принадлежат одному типу (однородность);

- Размер массива задается один раз и не меняется на протяжении работы (постоянство);
- Ко всем элементам массива имеется одинаковый доступ (равнодоступность);
- Все элементы располагаются последовательно в ячейках оперативной памяти (последовательность расположения);
- Элементы массива однозначно идентифицируются 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.):

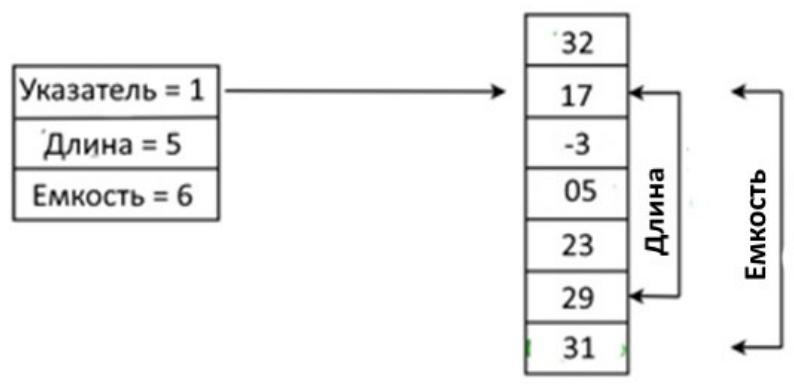


Рис. 13. Срез и его компоненты

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):

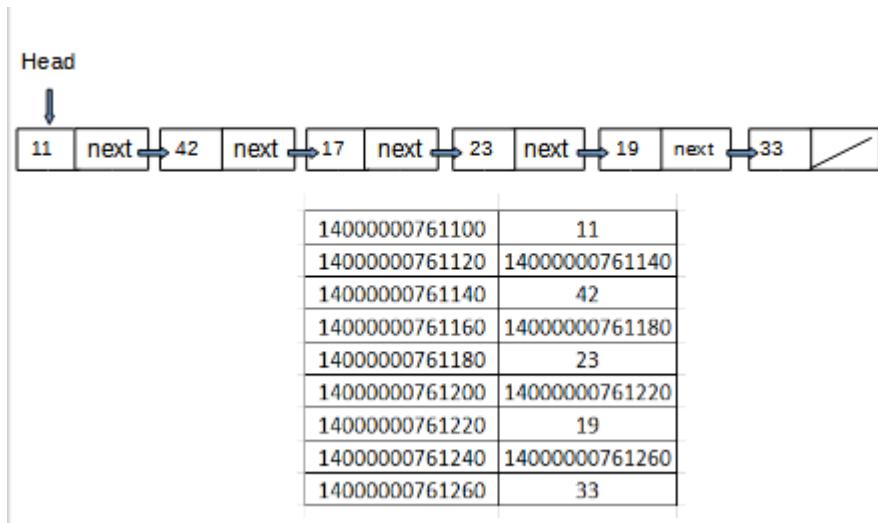


Рис. 1.4. Структура односвязного списка

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.

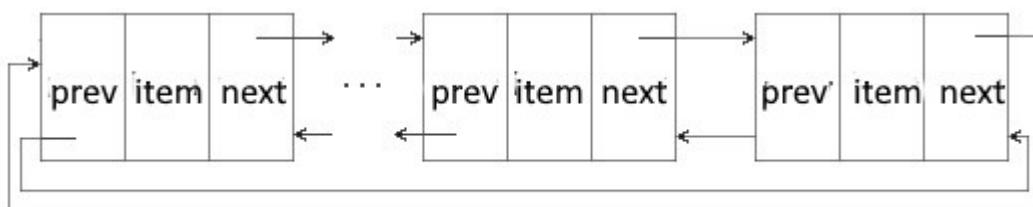


Рис. 1.6. Структура двусвязного списка

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.

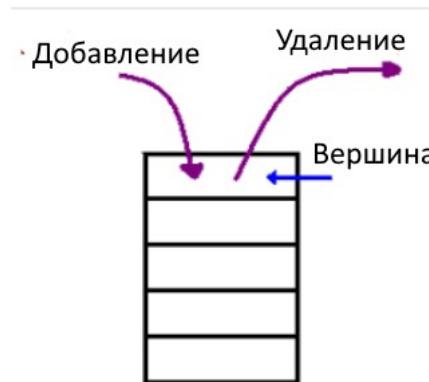


Рис. 1.7. Структура стека

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 интерфейсов, 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 вкладывались (рис.1.8).

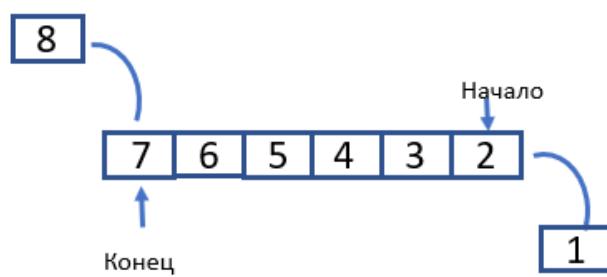


Рис.1.8. Структура данных «очередь»

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, 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)].

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).

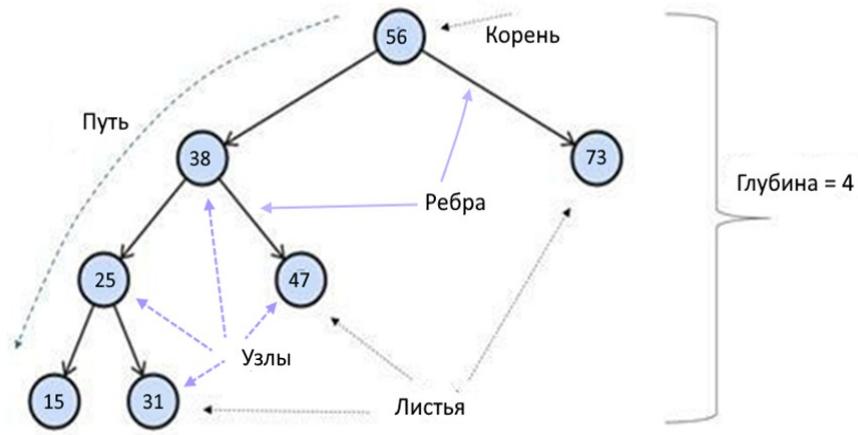


Рис. 1.9. Основная терминология дерева

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 Т. 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).

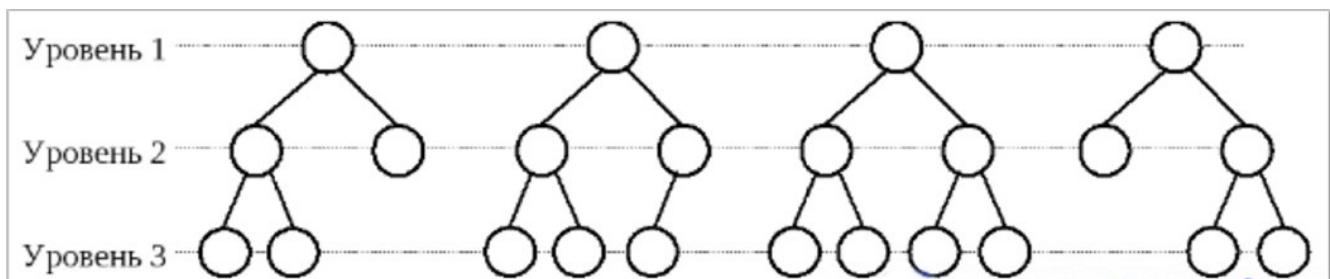


Рис. 1.10. Binary tree species

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)*:

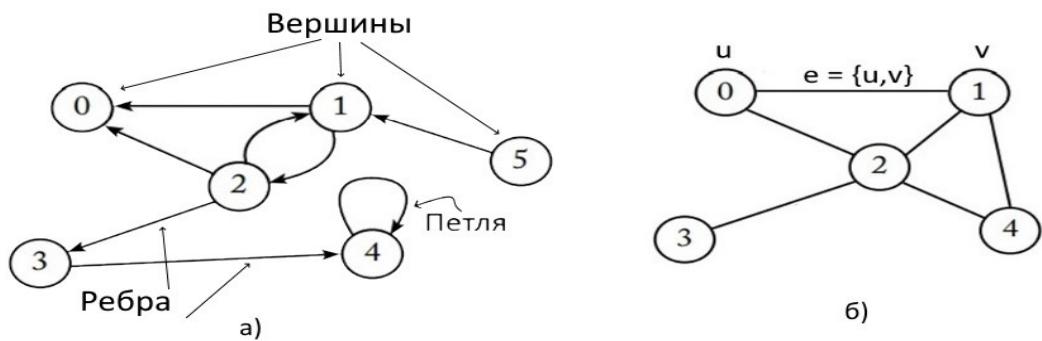


Рис. 1.11. Вид графа а)- ненаправленного; б) – направленного

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 совпадают. Ребро называется *петлей*, если его концы совпадают, то есть, $e = \{u,u\}$. Если вершина v_i является началом или концом ребра e_k , 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.

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.

РАЗДЕЛ 2. ОБЗОР ЯЗЫКА ПРОГРАММИРОВАНИЯ GOLANG

2.1. Общая характеристика

Поскольку в этой работе используется гибридный подход к программной реализации алгоритмов, необходимо хотя бы кратко описать основные концепции и конструкции его составляющих: визуальный алгоритмический язык Drakon и язык программирования Golang. Думается, что такой подход является весьма перспективным, прежде всего, в сфере образования, поскольку способствует формированию и развитию алгоритмического, компьютерного мышления и вполне оправдан для применения в такой области компьютерных наук как структуры данных и алгоритмы. Однако при этом предполагается, что читатель знаком хотя бы с одним из современных языков программирования. В то же время описание языка Golang ограничивается конструкциями, достаточными для понимания алгоритмов структур данных.

Основное внимание в этом разделе уделено особенностям программных конструкций: *переменные, массивы, срезы, карты, указатели, логические операторы, циклы, структуры, рекурсии и т. д.* Особый акцент делается на описании типов данных как базовых, так и пользовательских. Рассматривается также такая конструкция языка как *интерфейс*. Все, что выходит за рамки направления «Структуры данных и алгоритмы» представлено в различных материалах, которые приведены как по ходу текста, так и в перечне литературы.

Golang (Go) - язык общего назначения, разработанный с учетом системного программирования. Первоначально язык был разработан в Google в 2007 году Робертом Гриземером, Роб Пайком и Кен Томпсоном. Он сильно и статически типизирован (требуется точное указание типов переменных), обеспечивает встроенную поддержку сбора мусора и поддерживает параллельное программирование [].

Ниже перечислены наиболее важные особенности программирования Go:

- *Простой и понятный синтаксис.* Это делает написание кода приятным занятием.
- *Статическая типизация.* Позволяет избежать ошибок, допущенных по невнимательности, упрощает чтение и понимание кода, делает код однозначным.
- *Скорость и компиляция.* Скорость у Go в десятки раз быстрее, чем у скриптовых языков, при меньшем потреблении памяти. При этом, компиляция практически мгновенна. Весь проект компилируется в один бинарный файл, без зависимостей. Как говорится, «просто добавь воды». И вам не надо заботиться о памяти, есть сборщик мусора.
- *Отход от концепции объектно-ориентированного программирования* (ООП). В языке нет классов, но есть структуры данных с методами. Наследование заменяется механизмом встраивания. Существуют интерфейсы, которые не нужно явно имплементировать, а лишь достаточно реализовать методы интерфейса.
- *Параллелизм.* Параллельные вычисления в языке делаются просто и изящно. Горутины (что-то типа потоков) легковесны, потребляют мало памяти.
- *Богатая стандартная библиотека.* В языке есть все необходимое для веб-разработки и не только. Количество сторонних библиотек постоянно растет. Кроме того, есть возможность использовать библиотеки C и C++.
- *Возможность писать в функциональном стиле.* В языке есть замыкания (closures) и анонимные функции. Функции являются объектами первого порядка, их можно передавать в качестве аргументов и использовать в качестве типов данных.

2.2. Структура программы на языке Go

Структура программы на языке Golang представлена на рис. 2.1.

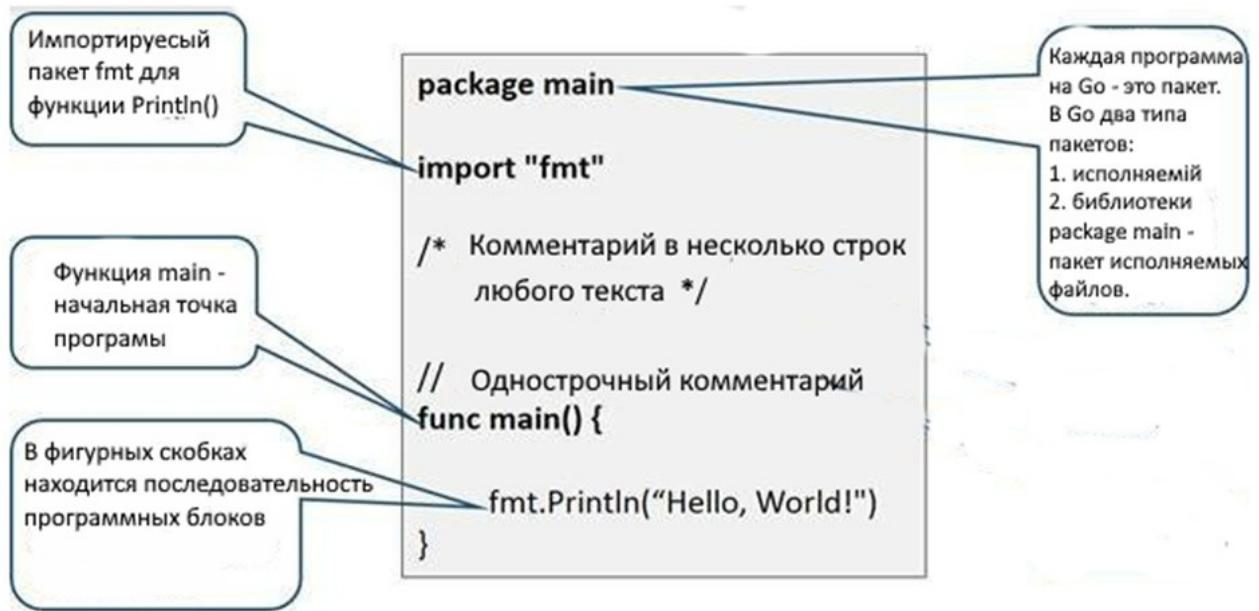


Рис. 2.1. Структура программы на языке Go

Каждая программа на Go — это пакет. В зависимости от его содержимого, этот пакет может стать основным файлом программы, а может — библиотекой с проблемно-ориентированным кодом.

```

// название программы (пакета)

package main

.......

// подгружаем нужные пакеты, если команды из этих пакетов понад-
// обятся в нашей программе

import (
    "fmt",
    "flag"
)

```

```
// основная функция — main — означает, что её содержимое и будет
выполняться после запуска программы

// кроме неё в программе может быть сколько угодно других функций

func main() {

    // содержимое функции

    fmt.Println("Hello World")

}
```

Комментарии в Go существуют в двух видах: односторонние и многострочные.

```
// - односторонний комментарий;

/* */ многострочный комментарий, расположенный между этими
символами
```

2.3. Переменные и константы

Переменная в любом языке программирования назначает место хранения значения, связанного с символическим именем или идентификатором. Одна переменная или список переменных, используемых в программе, объявляется через ключевое слово – *var*. Присваивание значений переменным осуществляется знаком «*=*».

При объявлении переменной обязательно указывается ее тип, чтобы компилятор знал, как обрабатывать эти данные. Именно поэтому язык Go называется строго типизированным.

```
var i int  
var s string  
var f bool
```

здесь *i* – целочисленная переменная; *s* - строковая переменная, *f* – логическая переменная.

При объявлении переменной допускается присвоение начальных значений, которые в дальнейшем могут изменяться:

```
var a int = 25  
var a, b, c int 100, 200 300  
var y = float32 = 32.5  
var z string = "Введите новое значение "
```

При отсутствии начальных значений переменные либо обнуляются (для числовых типов), либо заполняются пустыми строками (для строковых данных).

Возможно короткое объявление переменных через оператор `:=`, например, `a := 2.5`. Однако в этом случае переменная является локальной, то есть доступной только в отдельном фрагменте программы.

Присвоение имен переменным должно ориентироваться на определенные правила и стиль.

- имена переменных могут состоять только из одного слова (без пробелов).
- имена переменных могут состоять только из букв, цифр и символов подчеркивания (`_`).
- имена переменных не могут начинаться с цифр.

В переменных учитывается регистр, однако регистр первой буквы имени переменной в Go имеет особое значение. Если имя переменной начинается с заглавной буквы, это означает, что данная переменная доступна за пределами пакета, где она была декларирована (экспортируемая переменная). Если имя переменной начинается со строчной буквы, она будет доступна только в том пакете, где она декларирована.

Константы представляют собой объекты программы, значения которых в программном коде не меняются. Константы объявляются через ключевое слово: *const*:

```
const item string  
const n int  
const y float64
```

При этом допускается присвоение начальных значений без объявления типа:

```
const item = "name"  
const n = 25  
const y = 45.5
```

2.4.. Ввод и вывод

Ввод данных осуществляется с помощью функций *Scan(&имя_переменной)* чтобы просто поместить введённое значение в переменную или *Scanf(%формат, &имя_переменной)* чтобы заранее указать тип данных, которые будем вводить.

Для реализации вывода данных в начало программы (после *package main*) импортируется соответствующий пакет *fmt*, в котором находится множество функций вывода в зависимости от типа переменных, расположения и пояс-

няющей строки, заключенной в двойные кавычки (`\n` – перевод строки). Для связи с соответствующим пакетом используется префикс `fmt`.

```
fmt.Print ("Привет") // вывод без перевода на новую строку  
fmt.Println () // вывод с переводом курсора на новую строку  
fmt.Printf() // вывод значений переменных в текст  
fmt.fmt.Printf("Hello %d\n", 23) // вывод строки и числа целого  
типа  
fmt.fmt.Fprint("Hello ", 23, "\n") // тоже с переводом на новую  
строку  
fmt.fmt.Println("Hello", 23) // вывод строки и числа  
fmt.Printf("В этом году я проехал %v городов, ", &output)
```

Например,

```
package main  
import "fmt"  
  
func main() {  
  
    var name string  
    var salary int  
    fmt.Println("Введите фамилию")  
    fmt.Scanf("%s \n", &name)  
    fmt.Println("Введите зарплату")  
    fmt.Scanf("%d \n", &salary)  
    fmt.Printf("%s \n", name)  
    fmt.Printf("%d \n", salary)  
    fmt.Printf("Зарплата %s составляет %d рублей", name,  
salary)  
}
```

Результат:

```
Введите зарплату  
Степанова  
Введите зарплату  
32000
```

Степанова

32000

Зарплата Степанова составляет 32000 рублей

2.5. Операторы принятия решений

При решении множества задач возникает проблема выбора дальнейшего пути решения в зависимости от некоторых условий. В языке Go такая возможность реализуется следующими конструкциями:

- выбор по условию: *if else; switch case; select* ;
- повтор команд с помощью итераций: *for (range)*;
- изменение поведения в процессе итераций: *break* и *continue*.

a). Оператор выбора *if-else*

В языке Go используются три конструкции принятия решений или, иными словами, выбора варианта дальнейшего вычислительно процесса: *if-else*, *switch* и *select*. Синтаксис первого оператора имеет вид:

```
if condition {  
    // исполняемый код при условии condition == true  
} else {  
    // исполняемый код при условии condition == false  
}
```

При одиночном сравнении фигурные скобки можно не ставить. Открывающая скобка остаётся на той же строке, что и условие:

```
package main  
import "fmt"
```

```
func main() {
    // если условие верно
    if ID == "Apple" {
        // то выполняется то, что заключено в скобках:
            fmt.Println("Введите свой логин и пароль")
        // если условие не выполнилось, то можно сразу проверить
        ещё одно условие
        // так можно делать сколько угодно раз
    } else if ID == "Google" {
        // выводим второй ответ
            fmt.Println("Ваша операционная система не поддержи-
вается")
        // выполняем то, что относится к последнему if
    } else {
            fmt.Println("Ошибка ввода")
    }
}
```

6). Оператор переключения switch

Переключатель выражений *switch* предоставляет собой простую конструкцию обращения к различным частям программы на основе значения выражения. Синтаксис оператора *switch*:

```
switch optstatement; optexpression{
    case expression1: Statement..
    case expression2: Statement..
    ...
    default: Statement..
}
```

После первого найденного варианта оператор выполняет нужные действия и прекращает работу.

```
switch ID {  
    // проверяется первое значение  
    case "Apple":  
        fmt.Println("Введите свой логин и пароль")  
    // проверяется второе значение  
    case "GoogLe":  
        fmt.Println("Ваша операционная система не поддерживает")  
    // если ничего нужного не нашлось  
    default:  
        fmt.Println("Ошибка ввода")  
}
```

в). Оператор выбора *select*

Оператор *select* позволяет выбирать для выполнения одно из нескольких выражений. Главное различие между *select* и *switch* заключается в том, что *select* работает по принципу ожидания. Это означает, что команда *select* не будет выполняться до тех пор, пока не будет выполнено сообщение, связанное с отправкой и получением по любому каналу. Следует отметить, что в визуальном алгоритмическом языке DRAKON в реализации редактора Drakon Web Editor используется подобный оператор Select, что будет рассмотрено в следующем разделе.

2.6. Циклы

В языке Golang принят только один формат цикла - *for*, имеющий следующие варианты.

а). Классический, С-подобный цикл с переменной, условием и шагом цикла:

```
for i := 0; i < 8; i++ { Тело цикла }
```

б). Цикл с предусловием.

Самый простой цикл — объявить только условие, а остальное разместить внутри цикла:

```
package main
import "fmt"
func main() {
    // переменная для цикла
    var count = 10
    // пока переменная больше 0 – цикл работает
    for count > 0 {
        // выводим текущее значение переменной
        fmt.Println(count)
        // уменьшаем её на единицу
        count = count - 1
    }
}
```

в). Цикл в диапазоне

Существует еще одна разновидность цикла for, выполняющая итерации для диапазона значений для типа данных. Ключевое слово range используется в цикле для итерации по элементам массива, среза или карты. При использовании массива и среза цикл возвращает индекс элемента в виде целого числа. При использовании карт он возвращает ключ следующей пары ключ-значение:

```
package main
```

```

import "fmt"
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    kvs := map[string]string{"ca": "Paris", "co":
"France"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
}

```

The following table provides a complete list of Golang keywords:

break	case	chan	const	continue
default	defer	else	fallthrough	for
func	go	goto	if	import
inter-	map	pack-	range	return

face		age		
select	struct	switch	type	var

2.7. Типы данных

Типы данных определяют виды значений, которые сохраняются переменными при написании программы. Типы данных также помогают определить операции, которые можно выполнять с использованием данных. Различают следующие виды типов данных.

- Базовый тип (Fundamental types). К этой категории относятся *числа, строки и логические (булевы) значения*.
- Агрегированный тип (Aggregate types). К этой категории относятся *массив и структуры.. struct*
- Ссылочный тип (Reference types). К этой категории относятся *указатели, срезы, ассоциативный массив (карта) и, функции*.
- Встраиваемый тип (Embedding types). Go поддерживает определяемые пользователем типы в виде типов псевдонимов или структур.

2.7.1. Базовые типы

- a). Целочисленные и вещественные типы переменных и констант

Целочисленные типы Go делятся на целые со знаком и целые без знака. Целочисленные типы со знаком включают int, int8, int16, int32 и int64; целые типы без знака включают uint, uint8, uint16, uint32, uint64, uintptr. Диапазон значений и объем занимаемой памяти целочисленных типов представлены в табл. 2.1:

Табл. 2.1

Тип	Диапазон	Занимаемая память
int8	-128 — 127	8 бит (1 байт)
uint8	0 — 255	
int16	-32 768 — 32 767	16 бит (2 байта)
uint16	0 — 65535	
int32	-2 147 483 648 — 2 147 483 647	32 бита (4 байта)
uint32	0 — 4 294 967 295	
int64	-9 223 372 036 854 775 808 — 9 223 372 036 854 775 807	64 бита (8 байт)
uint64	0 — 18 446 744 073 709 551 615	

В языке Go принято два формата чисел с плавающей запятой:

- float32 - Самым большим float32 является константа `math.MaxFloat32`, которая составляет около $3,4e^{38}$. Наименьшее положительное значение - $1,4e^{-45}$.
- float64 - Самым большим float64 является константа `math.MaxFloat64`, которая составляет около $1,8e^{308}$. Наименьшее положительное значение $4,9e^{-324}$.

Float32 обеспечивает примерно шесть десятичных знаков точности, в то время как float64 предоставляет около 15 цифр. Тип float64 является более предпочтительным для большинства задач, поскольку при использовании типа float32 во многих итерационных алгоритмах быстро накапливается ошибка округления.

Для определения к какому типу данных компилятор Go относит ту или иную переменную используется функция `Printf()`, в параметрах которой присутствует специальный символ `%T`:

```
a := 234,45
fmt.Printf("Type %T for %v\n", a, a)
Результат: Type float64 for 234.45
```

б) Строковые переменные и константы

Строка - это неизменяемая последовательность байтов. Строки могут содержать произвольные данные, включая байты со значением 0, но обычно они содержат читабельный текст. Текстовые строки обычно интерпретируются как UTF-8-кодированные последовательности кодовых точек. Встроенная функция `Len` возвращает число байт в строке, а оператор печати по спецификации `%x` возвращает один байт строки `s` для латиницы и два байта для кириллицы:

```
func main() {
    s := "Hello, World"
    fmt.Println("Строка >> ", s)
    fmt.Println("Длина строки = ", Len(s))
    fmt.Println("Hex bytes: ")
    for i := 0; i < Len(s); i++ {
        fmt.Printf(" %x", s[i])
    }
}
```

```

fmt.Println()
q := "Привет, Мир"
fmt.Println()
fmt.Println("Строка >> ", q)
fmt.Println("Длина строки = ", Len(q))
for i := 0; i < Len(q); i++ {
    fmt.Printf(" %x", q[i])
}

```

Результат:

```

Строка >> Hello, World
Длина строки = 12
Hex bytes:
48 65 6c 6c 6f 2c 20 57 6f 72 6c 64
Строка >> Привет, Мир
Длина строки = 20

```

d0 9f d1 80 d0 b8 d0 b2 d0 b5 d1 82 2c 20 d0 9c d0 b8 d1 80

в) Логические переменные и константы

Логический тип данных (*bool*) может иметь одно из двух значений: **true** (истина) или **false** (ложь). Булевые операторы используются в программировании для сравнения и для контроля потока данных:

```

func main() {
x := 5
y := 8
fmt.Println("x == y:", x == y) // равно
fmt.Println("x != y:", x != y) // не равно

```

```
fmt.Println("x < y:", x < y)      // меньше
fmt.Println("x > y:", x > y)      // больше
fmt.Println("x <= y:", x <= y) // меньше или равно
fmt.Println("x >= y:", x >= y) // больше или равно
}
```

Результат:

```
x != y: true
x < y: true
x > y: false
x <= y: true
x >= y: false
```

2.7.2. Агрегированные типы

a) Массив (Array).

Массив представляет собой последовательность данных фиксированной длины, которая используется для хранения в памяти однородных элементов. Массивы в Go практически аналогичны массивам в других языках программирования. Элементы массива индексируются с помощью оператора `[] index` с их нулевой позицией, что означает, что индекс первого элемента равен `array[0]`, а индекс последнего элемента — `array[len(array)-1]`, где `len(array)` — длина массива. Однако из-за фиксированной длины массивы не очень популярны в отличие от конструкции `Slice` (Срез), которая несравненно чаще используется в Go. Синтаксис массива выглядит таким образом:

`[N]T{value1, value2, value3, ...value n}, где N – число элементов`

б) Структура (Structure)

Конструкция **Structure** представляет собой тип данных, определяемый разработчиком и служащий для представления каких-либо реальных объектов. Структуры содержат набор полей, которые представляют различные атрибуты объекта. Для определения структуры применяются ключевые слова **type** и **struct**.

```
type имя_структуры struct {  
    поля_структуры  
}
```

Ниже представлен пример структуры данных о сотрудниках предприятия:

```
package main

import "fmt"

type Employee struct {
    firstName, lastName, address string // поля строкового типа
    age, phone, salary int // поля целого типа
}

func main() {
    // инициализация структуры
```

```
emp := Employee{firstName: "Евгений", lastName: "Смирнов", age: 42,
phone: 123456789, salary: 34000, address: "Курск"}

    fmt.Println("Имя и фамилия сотрудника: ", emp.firstName,
emp.lastName)

    fmt.Println("Возраст сотрудника: ", emp.age)

    fmt.Println("Зарплата сотрудника : ", emp.salary)

    fmt.Println("Телефон сотрудника : ", emp.phone)

    fmt.Println("Адрес сотрудника : ", emp.address)

}
```

Память для новой переменной типа *struct* выделяется с помощью функции *new*, которая возвращает указатель на выделенную память:

```
var w *T = new(T),
```

или в разных строчках, если структура объявлена в области пакета:

```
var w *T
```

```
w = new(T)
```

При использовании краткой формы присвоения значения переменной (`:=`), то есть, `w := new(T)` переменная `w` является указателем на участок памяти, в котором поля структуры содержат нулевые значения в соответствии с их типами:

```
type structEmpl struct {
    name string
    age int
    salary float64
```

```

}

func main() {
    ms := new(structEmpL)
    fmt.Println(ms)
    ms.name = "Рыжоб"
    ms.age = 45
    ms.salary = 1200.5
    //fmt.Printf("Фамилия: %s\n", ms.name)
    fmt.Printf("Возраст: %d\n", ms.age)
    fmt.Printf("Зарплата: %8.1f\n", ms.salary)
    fmt.Println(ms)
}

```

2.7.3. Ссылочные типы

a) Указатели (Pointer)

Указатели в языке программирования Golang — это переменные, которые используются для хранения адреса памяти другой переменной. Переменные используются для хранения некоторых данных по определенному адресу памяти в системе. Адрес памяти всегда представляется в шестнадцатеричном формате (начиная с 0x, например, 0xFFAAFF и т. д.) (рис. 2.2):



Рис. 2. Переменная, указатель, память

Указатель обычно называют особым видом переменной. Основными и единственными операторами указателей являются: оператор разыменования (*) и оператор адреса (&). Оператор разыменования (*) используется для объявления переменной указателя и доступа к значению, хранящемуся в адресе. Оператор адреса (&) используется для возврата адреса переменной или для доступа к адресу переменной указателю.

Синтаксис объявления указателя:

```
var pointer_name *Data_Type,
```

где Data_Type - любой допустимый тип данных, например, var pos *string.

Для работы с указателем его необходимо инициализировать с адресом памяти другой переменной с помощью оператора адреса &, как показано в следующем примере:

```
var a = 124  
var s *int = &a
```

Неинициализированный указатель всегда будет иметь нулевое значение <nil>. Ниже приведен пример инициализации и обработки указателя:

```
func main() {  
  
    var z1, z2 int = 64, 128  
    var p1, p2 *int  
    p1 = &z1 // указатель p1 инициализирован  
    fmt.Println("Значение переменной z1 = ", z1)
```

```

fmt.Println("Адрес z1 = ", &z1)
fmt.Println("Значение переменной z2 = ", z2)
fmt.Println("Адрес z2 = ", &z2)
// указатель p2 неинициализирован
fmt.Println("Значение, сохраненное в переменной p1 = ", p1)
fmt.Println("Значение, сохраненное в переменной p2 = ", p2)
}

```

Результат:

```

Значение переменной z1 = 64
Адрес z1 = 0xc000086080
Значение переменной z2 = 128
Адрес z2 = 0xc000086088
Значение, сохраненное в переменной p1 = 0xc000086080
Значение, сохраненное в переменной p2 = <nil>

```

Оператор разыменования * используется для объявления переменной указателя и для доступа к значению, хранящемуся в переменной, на которую указывает указатель:

```

func main() {
    var y = 157
    var p = &y
    fmt.Println("Значение переменной в y = ", y)
    fmt.Println("Адрес переменной y = ", &y)
    fmt.Println("Значение, сохраненное в p = ", p)
    fmt.Println("Значение, сохраненное в y(*p) = ", *p)
}

```

Результат:

```

Значение переменной в y = 157
Адрес переменной y = 0xc000014078
Значение, сохраненное в p = 0xc000014078
Значение, сохраненное в y(*p) = 157

```

б) Срез (Slice)

Срезы представляют собой последовательности переменной длины, все элементы которых имеют один и тот же тип. Записывается тип среза `[] T`, где элементы имеют тип `T`.

Срез состоит из трех компонентов:

Указатель: указатель указывает на элемент массива, доступный через срез, который не обязательно является первым элементом массива.

Длина: количество элементов среза. Он не может превышать емкость.

Емкость: емкость обычно представляет собой количество элементов между началом среза и концом базового массива, представляющего может меняться.

Встроенные функции `len` и `cap` содержат значения длины и емкости среза соответственно. Несколько срезов могут совместно использовать один и тот же базовый массив и могут ссылаться на перекрывающиеся части этого массива.

Первая позиция индекса в срезе всегда равна 0, а последняя равна длине среза минус единица (`len - 1`). Срез объявляется так же, как массив, но он не содержит размера среза. Таким образом, он может расти или уменьшаться в соответствии с тем или иным алгоритмом.

Синтаксис конструкции среза имеет вид:

`[]T` или `[]T{}` или `[]T{value1, value2, value3, ...value n}`

Здесь `T` — это тип элементов. Например: `var my_slice [] int`

На рисунке (рис. 2.3) показано визуальное представление среза через его компоненты – **указатель (ptr), длину (len) и емкости (cap)**:

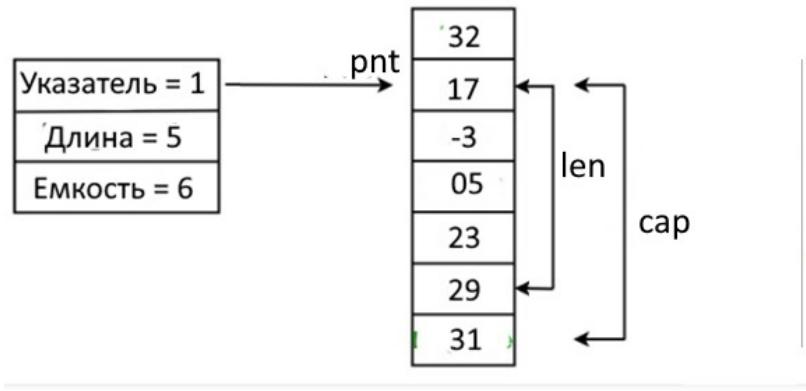


Рис.2.3. Представление среза в памяти через компоненты

Срезы можно создавать из уже существующих срезов, указав соответствующие индексы (рис. 2.4):

```

package main
import (
    "fmt"
)

func main() {
    var sr = []int{17, -21, 5, 62, 24, 48, 78, -43}
    sr0 := sr[2:6]
    fmt.Println("sr0 = ", sr0)
    sr1 := sr[2:]
    fmt.Println("sr1 = ", sr1) // prints [5 7 9 11 13
    15]
    sr2 := sr1[:3]
    fmt.Println("sr2 = ", sr2)
}

```

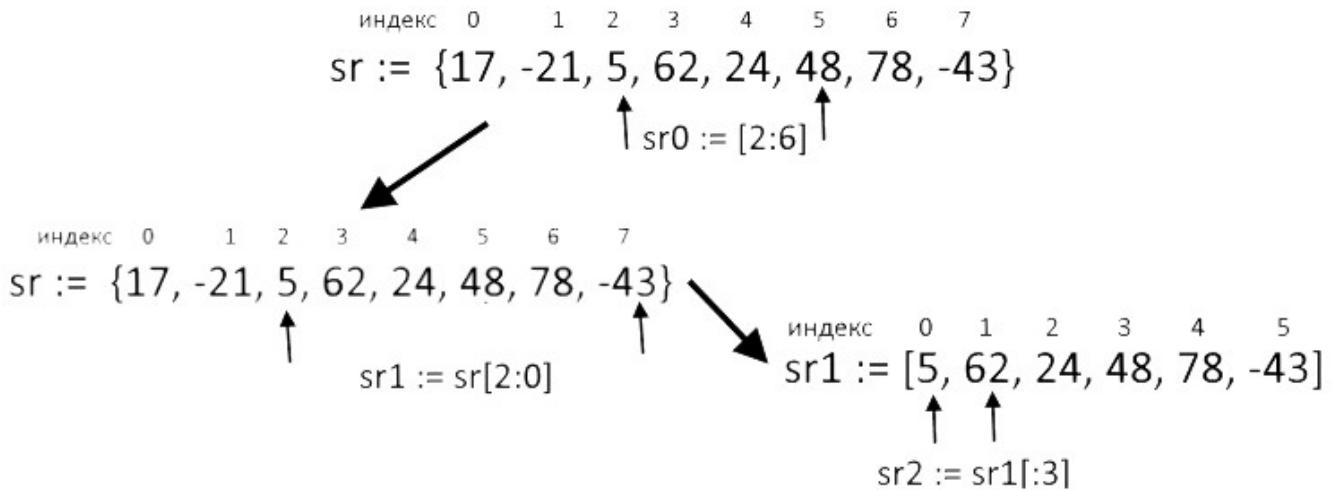


Рис. 2.4. Создание новых срезов из существующих срезов

Важно отметить, что каждому созданному срезу $sr0$, $sr1$, $sr2$ отводится свой участок памяти, то есть, все созданные срезы имеют различные адреса, даже в том случае, когда они являются фрагментами одного среза.:

```
sr --> 0xc000004078; sr0 --> 0xc000004090; sr1 --> 0xc0000040c0;
sr2 > 0xc0000040f0
```

2.7.3. Ассоциативные массивы (Map)

Ассоциативные массивы (в дальнейшем *карты*) представляют собой контейнеры хранения пары "ключ-значение". Карты являются одной из самых распространенных и полезных структур данных. В отличие от таких структур данных как массив или срез карта обеспечивает быстрый и эффективный поиск элементов. Карта не допускает дублирования ключей, но может иметь повторяющиеся значения.

Синтаксис объявления карты, в которой ключ имеет тип *int*, а значение - *string*:

```
var names map[int]string
```

Инициализация карты может быть выполнена двумя способами:

С помощью функции *make()*: *var names = make(map[int]string);*

С помощью синтаксиса литералов: *var films = map[int]string;*

Ниже приведен пример, иллюстрирующий оба метода инициализации карты:

```
func main() {
    names := map[int]string{0: "Жаров", 1: "Лавров", 2: "Ульянов", 3: "Матвеев"}
    fmt.Println(names)
    films := make(map[string]string)
    films["Жаров"] = "Путевка в жизнь"
    films["Лавров"] = "Так победим"
    films["Ульянов"] = "Председатель"
    films["Матвеев"] = "Любовь земная"
    fmt.Sprintln(films)
    for key, value := range films {
        fmt.Println(" \n", key, ">", value)
    }
}
```

Результат:

```
Матвеев > Любовь земная
Жаров > Путевка в жизнь
Лавров > Так победим
Ульянов > Председатель
```

2.8. Функции и методы

2.8.1. Функции

Функция представляет собой группу операторов, совместно выполняющих ту или иную задачу. С помощью функций можно многократно вызывать ее блок операторов как единое целое в других частях программы. Функции в Go могут быть назначены переменным, переданы в качестве аргумента и могут быть возвращены из другой функции. При объявлении функции нужно указать какого типа переменные передаются в функцию (*параметры*) и какого типа данные возвращает функция (*возвращаемые типы*). В Go необходимо указывать тип данных для каждого параметра. Для объявления функции используется ключевое слово *func*. Общая структура объявления функции показана ниже:

```
func Имя функции [Параметры] (Возвращаемые типы)
{
    Тело функции
}
```

Функция умножения двух вещественных чисел *multiply(x,y)* выглядит следующим образом:

```
package main
import "fmt"
func multiply(x,y float64) float64 {
    var res
    res = x * y
    return res      // возвращение результата
}
func main() {      // выполняемая функция
    var x float64 = 15
    var y float64 = 120.6
```

```

        var mult float64      // переменная для результата
        mult = multiply(x,y)    // результат работы функции
        fmt.Println(mult)      // вывод результата
    }

```

Функция может возвращать столько значений, сколько требуется:

```

func main() {
    var a, b int = 60, 20
    vAdd, vSub := addSub(a, b)
    fmt.Printf("a + b = %d\n", vAdd) // prints "35 + 25 = 60"
    fmt.Printf("a - b = %d\n", vSub) // prints "35 - 25 = 10"
}

func addSub(x, y int) (int, int) { // multiple return values
    return x + y, x - y
}

```

Важной особенностью функций в Go является возможность принимать параметры по значению или по ссылке [].

2.8.2. Методы (Methods)

Методы вместе с функциями предоставляют различные способы организации программного кода. Метод в языке Go - это функция, связанная с определенным типом, которая действует на переменную определенного типа, называемую приемником (получателем). Язык Golang не поддерживает концепцию Class, общепринятую в других языках. Именно с помощью метода объект предоставляет свои свойства, включая поведение объекта. Метод должен удовлетворять следующим условиям: он должен быть определенного типа и должен быть определен в том же пакете.

Объявление метода похоже на объявление функции, но оно имеет дополнительную часть объявления параметра. Дополнительный параметр может содержать один и только один параметр типа получателя метода. Параметр получателя должен быть заключен в круглые скобки и объявлен между ключевым словом *func* и именем метода.

```
Func (Имя_приемника Type) Имя_метода(Параметры)(Возвращаемый тип){  
    // Блок операторов  
}
```

В языке Go разрешено определять метод, приемник которого имеет тип структуры. Этот *приемник* доступен внутри метода, как показано в приведенном ниже примере. Объявлена структура-приемник *rec*, описывающая объект *rect*:

```
type rect struct {  
    var width, height int  
}
```

С этой структурой связываются два метода, определяющих поведение объекта, описываемого структурой *rect*:

```
func (r rect) area() int {  
    return r.width * r.height  
}
```

```
func (r rect) perim() int {
```

```
    return 2*(r.width + r.height)
}
```

2.9. Интерфейсный тип (Interface type)

Golang не является классическим объектно-ориентированным языком, то есть, он не поддерживает реализацию в прямом виде концепции «классов» и «наследования». Однако Go содержит очень гибкую концепцию интерфейсов, которая обеспечивает многие аспекты объектно-ориентированного программирования.

Интерфейсы в Go обеспечивают способ указания поведения объекта с помощью набора методов, определяемых типом интерфейса. Переменная типа интерфейса может хранить значение любого типа с набором методов, который является любым надмножеством интерфейса. Концепция интерфейса позволяет организовывать разные группы методов, применяемых к объектам разной природы. Иными словами, интерфейсы представляют собой коллекции сигнатур методов, декларирующих имя, параметры типов и возвращаемые типы методов в интерфейсе.

Синтаксис интерфейса имеет вид:

```
type Namer interface { // Namer – интерфейсный тип
    Method1(param_list) return_type
    Method2(param_list) return_type
```

Например, нужно создать метод для определения площади геометрических фигур – круга и прямоугольника.

```
type shape interface {
    area() float64
    perimeter() float64
```

```
}
```

Этот код определяет интерфейс для фигур и объявляет две функции, area() и perimeter() с возвращаемым типом float64.

```
package main
import "fmt"
// Объявление интерфейса
type shape interface {
    area() float64
    perimeter() float64
}
// Объявление структуры «прямоугольник»
type rectangle struct{
    length, height float64
}
// Объявление структуры «круга»
type circle struct{
    radius float64
}
// Объявление методов для прямоугольника
func (r rectangle) area() float64 {
    return r.length * r.height
}
func (r rectangle) perimeter() float64 {
    return 2 * r.length + 2 * r.height
}
// Объявление методов для круга
func (c circle) area() float64 {
    return 3.142 * c.radius * c.radius
}
func (c circle) perimeter() float64 {
    return 2 * 3.142 * c.radius
}

func main() {
    r := rectangle{length: 10.0, height: 5.0}
    c := circle{radius: 5.0}

    fmt.Printf("Площадь прямоугольника - %8.1f\n", r.area())
    fmt.Printf("Периметр прямоугольника - %8.1f\n",
r.perimeter())
    fmt.Printf("Площадь круга - %8.1f\n", c.area())
    fmt.Printf("Периметр круга - %8.1f\n", c.perimeter())
```

}

Результат:

<i>Площадь прямоугольника -</i>	<i>50.0</i>
<i>Периметр прямоугольника -</i>	<i>30.0</i>
<i>Площадь круга -</i>	<i>78.5</i>
<i>Периметр круга -</i>	<i>31</i>

Заключение

Следует еще раз отметить, что в данном разделе приведены элементарные сведения по языку программирования Golang, достаточные для понимания материала, изложенного в дальнейшем. За пределами раздела остались такие сведения как Go-подпрограммы и каналы, параллельность и совместно используемые переменные, пакеты и инструменты Go, рефлексия и т.д. Далее, в соответствующих разделах будут более подробно объясняться инструменты и конструкции, изложенные в этом разделе. Кроме того, будет детально пояснен инструмент рекурсии в силу его особого значения для итерационных процессов в алгоритмах обработки структур данных.

Кроме того, необходимо отметить, что в данной книге, реализующей концепцию гибридного программирования, линейные конструкции языка Go используются как заполнение графоэлементов визуального алгоритмического языка DRAKON. Другие конструкции, в частности, циклы в диапазоне или оператор *Select* отличается от Go-конструкций. Более того, некоторые конструкции языка DRAKON в ходе автоматической генерации программного кода преобразуются в другие конструкции. В частности, программная реализация оператора *Select* преобразуется в составной оператор выбора *if-else*. Это связано с стремлением авторов языка DRAKON сделать генерируемый код более эффективным, повышая его быстродействие и экономя компьютерную память.

РАЗДЕЛ 3. ВИЗУАЛЬНЫЙ ЯЗЫК DRAKON

Как показывает опыт преподавания информатики в средней школе, владение искусством программирования и алгоритмизации наталкивается на серьезные трудности. В значительной степени они связаны с отсутствием навыков формулирования постановки задачи и разработки алгоритма ее решения. Еще в начале массового обучения программированию в школах и ВУЗах было замечено, что программистов условно можно разделить на две группы: «полевой офицер» и «штабной офицер». Полевой офицер, как правило, владея определенным запасом знаний программирования сразу же бросается писать программный код. Без комментариев и пояснений. Естественно, что в процессе программирования возникают непредвиденные ситуации, приводящие к постоянным изменениям в тексте программы. В конце концов и во многих случаях «полевой офицер» сдается или начинает подумывать о полезности побыть «штабным офицером». В свою очередь «штабной офицер» вначале обдумывает алгоритм программы. И здесь ему в помощь различные методы визуализации алгоритмов.

3.1. Визуализация представления алгоритмов

В теории и на практике сформировались следующие формы представления алгоритмов :

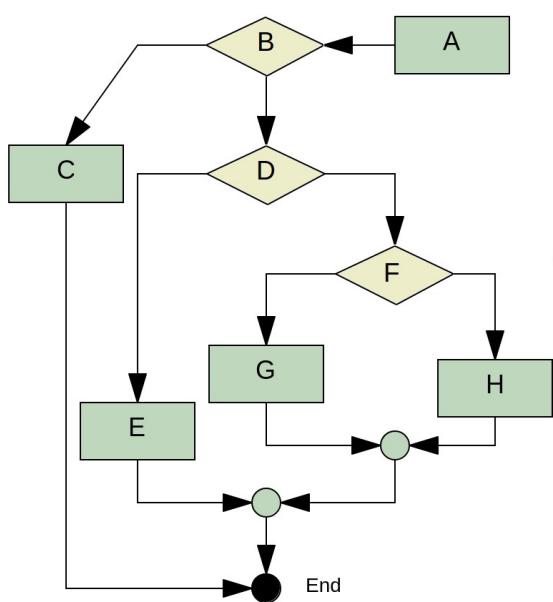
1. словесная (запись на естественном языке);
2. визуальная (изображения из графических символов);
3. псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке);
4. программная (тексты на языках программирования).

Наиболее эффективными и распространеными с точки зрения наглядности и понимания получили визуальные способы представления алгоритмов. В теоретическом отношении визуализация это форма представления информации, данных, знаний в виде изображений с целью достижения

максимального удобства восприятия, понимания и анализа. Визуальное представление позволяет лучше видеть и понимать структурные элементы, используемые при проектировании алгоритма, включая логику их взаимодействия. Наиболее распространенными средствами визуализации алгоритмических конструкций являются блок-схемы и дракон-диаграммы. Преимуществом применения блок-схем является наличие соответствующего Государственного стандарта. Однако в последнее время блок-схемы подвергаются критике: утверждаются, что они непригодны для структурного программирования, с трудом поддаются формализации, их нельзя использовать для генерации программного кода.

Альтернативой визуализации алгоритмов с помощью блок-схем является технология конструирования алгоритмов на основе *дракон-технологии*, результатом которой являются дракон-диаграммы. Дракон-диаграммы пригодны для формализованной записи, автоматического получения кода и выполнения его на компьютере. Однако более важным аспектом применения дракон-технологии является ее когнитивное отличие от технологии блок-схем. Хотя блок-схемы время действительно улучшают ясность программ, однако это происходит не всегда, причем степень улучшения невелика. Кроме того, есть немало случаев, когда неудачно выполнены блок-схемы запутывают дело и затрудняют понимание. В отличие от них дракон-диаграммы гораздо более высоким уровнем понимания (рис.3.1.).

Неупорядоченная
блок-схема старого образца



Современная блок-схема
ДРАКОН

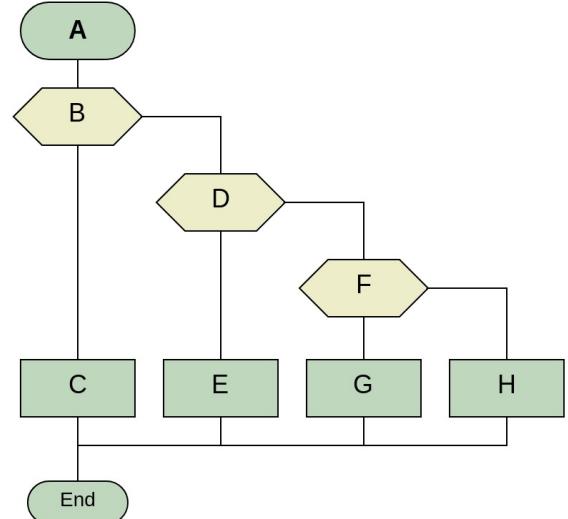


Рис. 3.1. Сравнение визуального представления алгоритма в виде блок-схемы и дракон-диаграммы

Благодаря использованию специальных формальных и неформальных когнитивных приемов дракон-диаграммы позволяют визуально представить решение любого сколь угодно сложного алгоритма или технологической проблемы в предельно ясной, наглядной и доходчивой форме. Особое значение визуализация алгоритмов с помощью дракон-диаграмм приобретает в учебном процессе как при изучении информатики, так и на уроках естественных, а также гуманитарных дисциплин.

3.2. Редактор алгоритмического языка DRAKON Editor

В настоящее время существует несколько платформ для построения дракон-диаграмм (DRAKON Editor, DRAKONHUB, Drakon.tech, «ИС Дракон»). Генерация программного кода на одном из известных языков программирования, в том числе и на языке Golang, реализуется с помощью свободно распространяемого редактора DRAKON Editor WEB. В настоящее время

актуальной версией редактора является DRAKON- Editor 1.31, которую можно загрузить с сайта <http://drakon-editor.sourceforge.net/> (рис. 3.2.).

Download		
Platform	Language	File
Windows, Linux, Mac	English and Russian	drakon_editor1.31.zip

Рис. 3.2. Архивированный файл редактора

Для запуска редактора DRAKON Editor 1.31. в операционной системе **Linux and Mac необходимо:**

- извлечь файлы из архива drakon editor 1.31.zip;
- запустить редактор: /drakon_editor.tcl с терминала (внутри неархивированной папки).

Для операционной системы Windows необходимо:

- извлечь файлы из архива drakon editor 1.31.zip;
- открыть исполняемый файл редактора двойным кликом на фрагменте drakon_tcl (рис. 3.3.):



Рис. 3.3. Запуск
редактора DRAKON Editor 1.31.

После активизации редактора (DRAKON_Editor) выбирается опция «Создать новый файл диаграммы», после чего открывается окно для сохранения файла диаграммы. Сразу же после создания нового файла его нужно обязательно сохранить, поскольку в редакторе DRAKON Editor нет необходимости сохранять все правки, - файлы сохраняются автоматически.

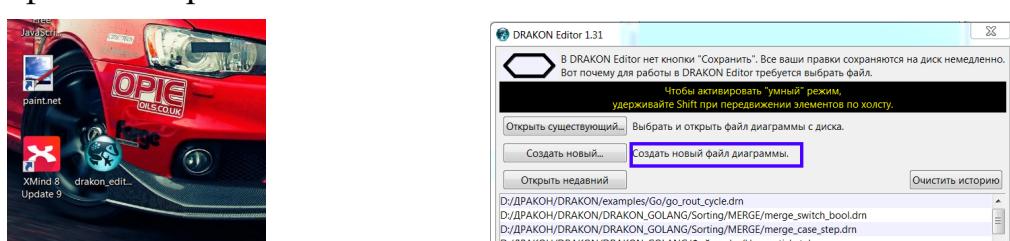




Рис. 3.4. Создание нового файла диаграммы

После создания нового файла открывается окно интерфейса редактора (Рис.3.5.):

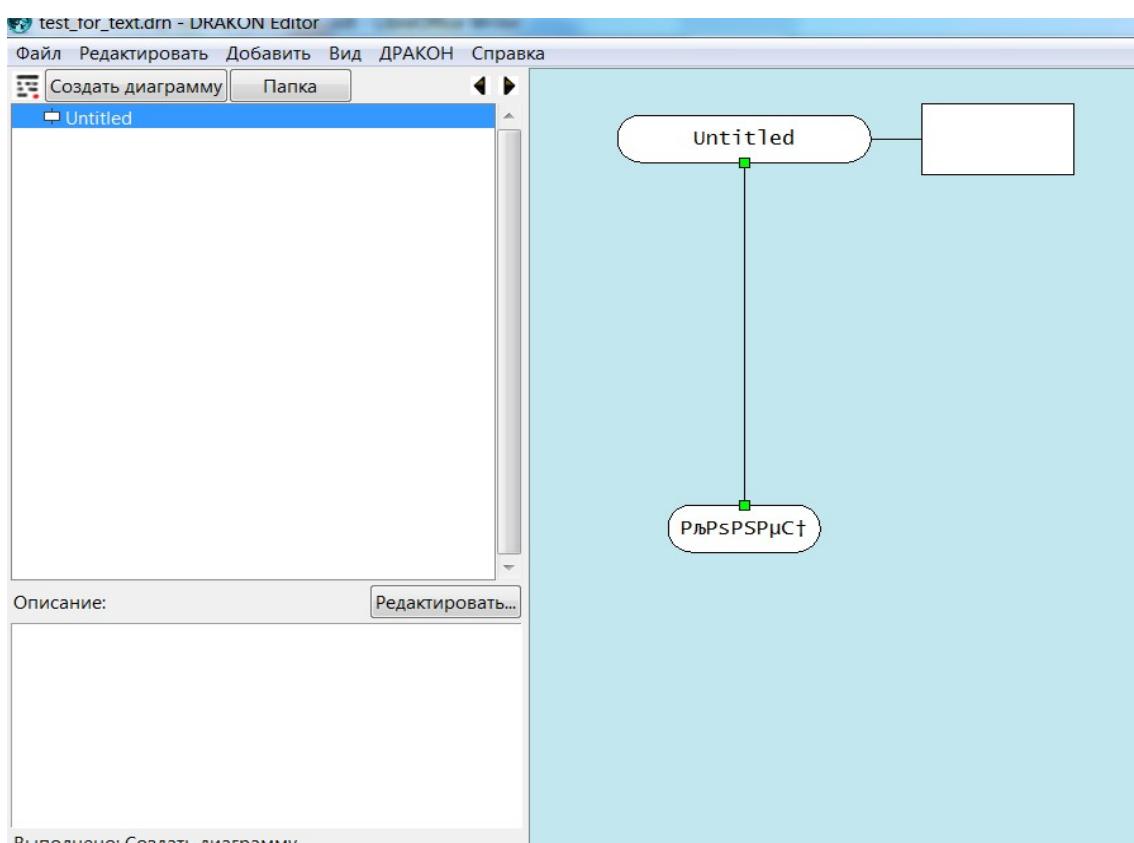


Рис. 3.5. Окно интерфейса редактора DRAKON WEB Editor

Далее нужно нажать иконку “New diagram” и во вновь открывшемся меню выбрать вид диаграммы (Примитив /Primitive/ или Силуэт /Silhouette/) (рис.3.6.).

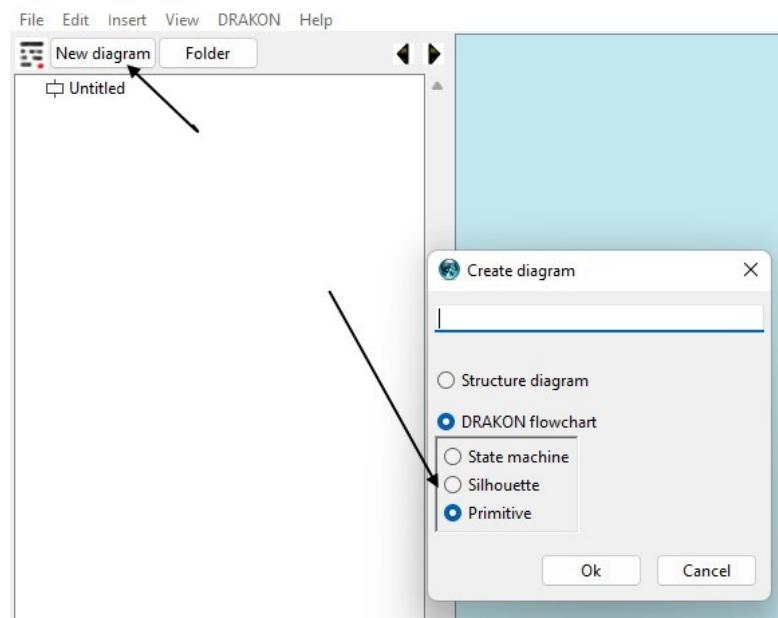
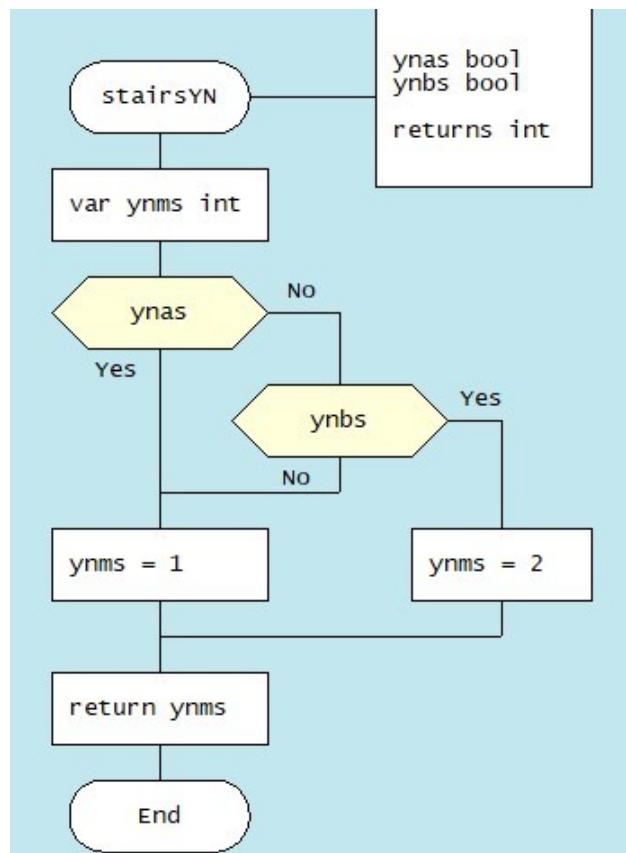
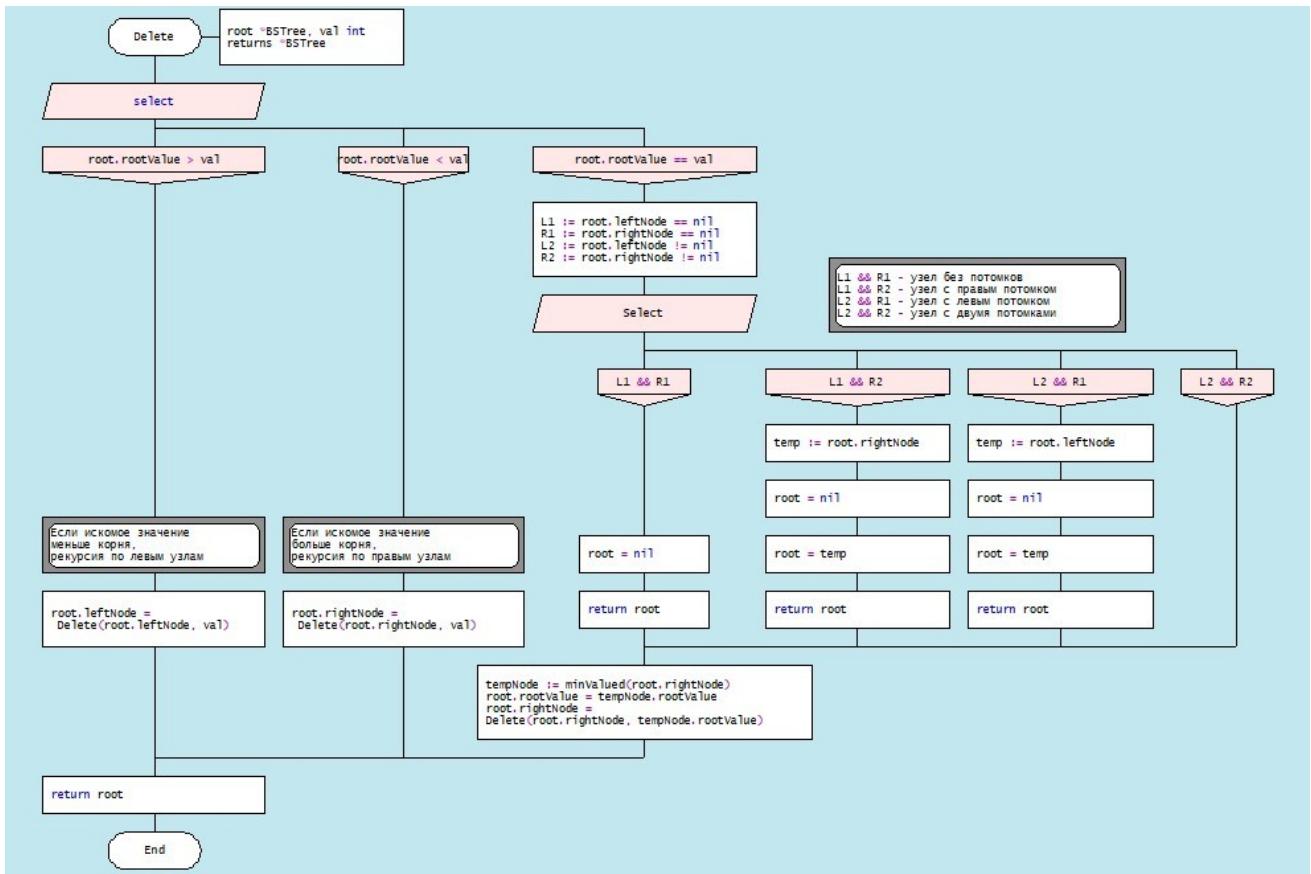


Рис.3.6. Создание новой диаграммы

Выбор вида диаграммы определяется сложностью алгоритма (рис.3.7.):



а) «Примитив»



6) «Силуэт»

Рис. 3.7. Виды диаграмм: а) «Притимив»; б) «Силуэт»

Для создания дракон-диаграммы пользователь вызывает контекстное меню (правая кнопка мыши), и в соответствии с алгоритмом выбирает необходимые графоэлементы, наполняя их соответствующими операторами выбранного языка программирования (Рис.3.6).

	- action	- действие (присвоение, печать...)
	- if	- условие
	- vertical	- вертикальная линия
—	- horizontal	- горизонтальная линия
	- select	- оператор выбора
	- case	- добавление нового варианта
	- loop	- цикл
	- arrow	- обратная стрела для цикла
	- begin/end	- графоэлемент "начало"/"конец"
	- branch header	- заголовок ветви
	- branch footer	- "подошва" ветви
	- inline comment	- встроенный комментарий
	- standard comment	- стандартный комментарий

Рис. 3.8. Массив графоэлементов редактора DRAKON WEBrakon Web Editor

Выбранный графоэлемент перемещается мышью на вертикальную линию (шампур — по терминологии DRAKON-технологии) в нужное место. После этого графоэлемент заполняется соответствующим текстом, отображающим тот или иной фрагмент на конкретном языке программирования.

Создание дракон-диаграммы должно происходить по определенным правилам:

1. Создание дракон-диаграммы начинается с названия, которое должно отражать назначение алгоритма (функции) и располагаться на самом верху.
 2. На диаграмме должны быть только одно начало и один конец.
- Графоэлемент «Конец» помещается внизу диаграммы (рис.3.8):

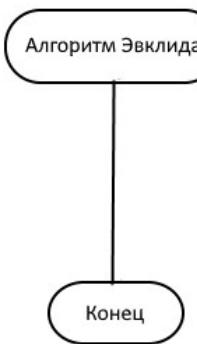


Рис. 3.8. Начало диаграммы

3. Поток действий представленный на диаграмме должен идти только сверху вниз. Такой подход является более удобным, поскольку в нашем культурном ареале тексты читаются именно таким образом.

4. Необходимо избегать поворотов. Единственный случай когда линии должны изменить направление, - это место принятия решений. Повороты нужны только тогда когда по алгоритму требуется осуществить выбор между различными действиями. Если же решений нет, нужно идти вниз. Во всяком случае необходимо минимизировать число поворотов.

5. **Категорически** не допускается пересечений линий. Все попытки применить пересечения должны пресекаться. Впрочем в случае пересечения редактор выдаст ошибку.

6. Выполнение действий сверху вниз позволяет не прибегать к использованию стрелок. Единственное исключение — цикл типа while (рис.3.8.):

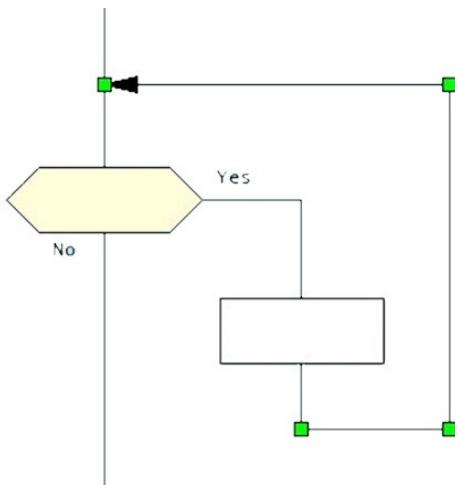


Рис. 3.9. Поток данных снизу-вверх (цикл типа While)

7. При создании дракон-диаграммы должны использоваться исключительно прямые вертикальные и горизонтальные линии, поскольку прямые линии понятней кривых.

8. Необходимо придерживаться одинакового расстояния между соседними графоэлементами. Редактор DRAKON WEB Editor позволяет «умно» оптимизировать диаграмму с помощью клавиш «Shift + мышь»).

9. Разветвление осуществляется только направо. Разветвление налево должно быть исключено. Выполнение этого правила существенно повышает предсказуемость диаграмм и их однообразие.

У читателя может сложиться впечатление, что создание дракон-диаграмм является очень сложным процессом. Следует напомнить еще раз, что в редакторе DRAKON WEB Editor все происходит достаточно просто и понятно. Редактор не допустит нарушение изложенных правил, что проверяется опцией *verify* (*на рисунке указана ошибка — нет пространства между графоэлементами* (рис.3.10)):

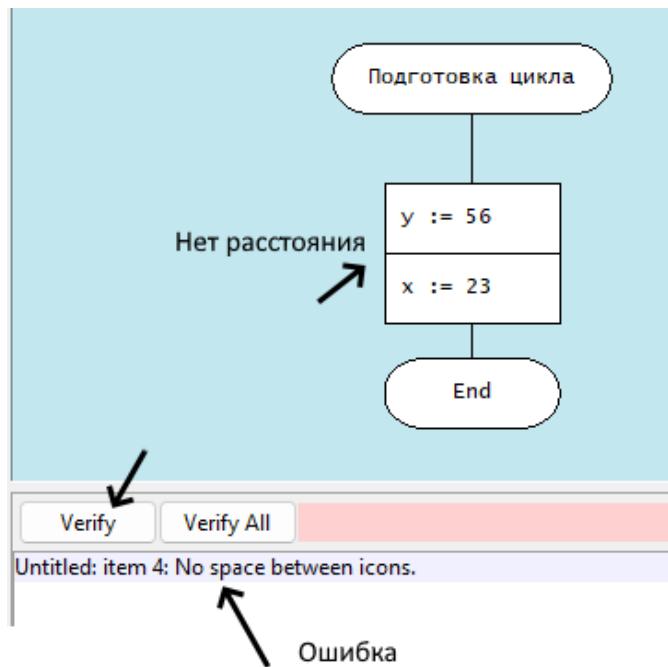


Рис. 3.10. Работа редактора, указывающего ошибку

В редакторе DRAKON WEB Editor реализован полный набор правил визуального синтаксиса, что освобождает пользователя от необходимости детально помнить синтаксические правила. DRAKON WEB Editor создает только правильно созданные диаграммы и не допускает диаграммы, в которых нарушен синтаксис, в генерацию программного кода.

3.3. Базовые структуры алгоритмов редактора DRAKON WEB Editor

Логическая структура какого-либо алгоритма может быть представлена комбинацией трех базовых структур: *линейной, разветвленной и циклической*. Характерной особенностью базовых структур является наличие в них одного входа и одного выхода.

3.3.1. Линейная структура

Линейная структура алгоритмического процесса реализует операции, выполняемые последовательно в порядке их записи. Типичным примером такого процесса является стандартная вычислительная схема, состоящая из трех этапов: а) введение входных данных; б) вычисление по формулам; в) вывод результата.

Графическое изображение базового элемента линейной структуры в программных продуктах линейки DRAKON имеет вид простых прямоугольников. Например, в алгоритме нахождения минимального и максимального значений массива переменным `minim` и `maxim` присваивается значение нулевого элемента массива `array` (рис.3.11).

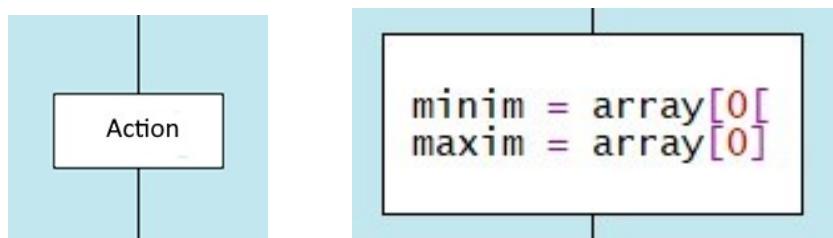


Рис.3.11. Линейная структура дракон-диаграммы

3.3.2. Базовые конструкции разветвленной структуры

Разветвленная конструкция содержит хотя бы одну проверку условия, в результате которой обеспечивается переход на один из возможных вариантов решения. Каждый из вариантов ведет к общему выходу, то есть работа алгоритма будет продолжаться независимо от того, какой путь будет выбран.

Структура разветвления существует в двух основных вариантах:

а). дракон-фрагмент конструкции “если (условие), то (действия) иначе (действия)”, то-есть в языках программирования это оператор *if...else*. Пример применения конструкции if-else в дракон-диаграмме в алгоритме нахождения минимального и максимального значений массива (рис. 3.12):

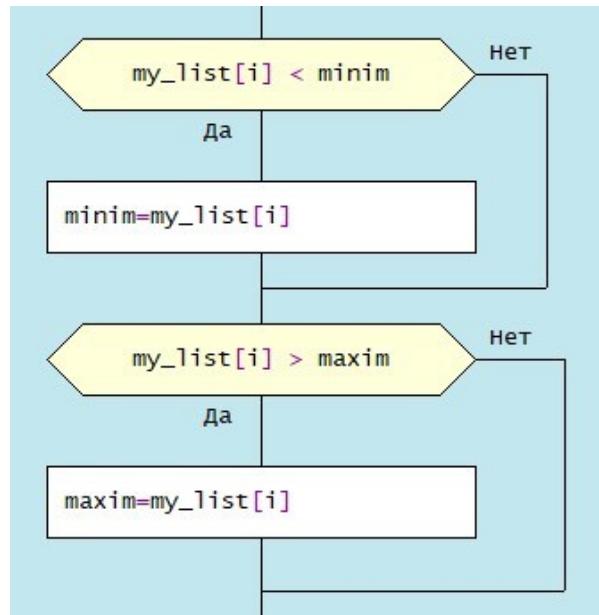


Рис. 3.12. Фрагмент диаграммы с конструкцией *if-else*

б). дракон-фрагмент конструкции “Выбор”

Значения, с которыми будет сравниваться выражение в графоэлементе «Выбор», находятся в графоэлементах «Вариант». Если в крайнем правом варианте отсутствует текст, это означает «все другие значения». Ниже — пример выбора разветвления алгоритма сортировки (merge sort) (3.13).

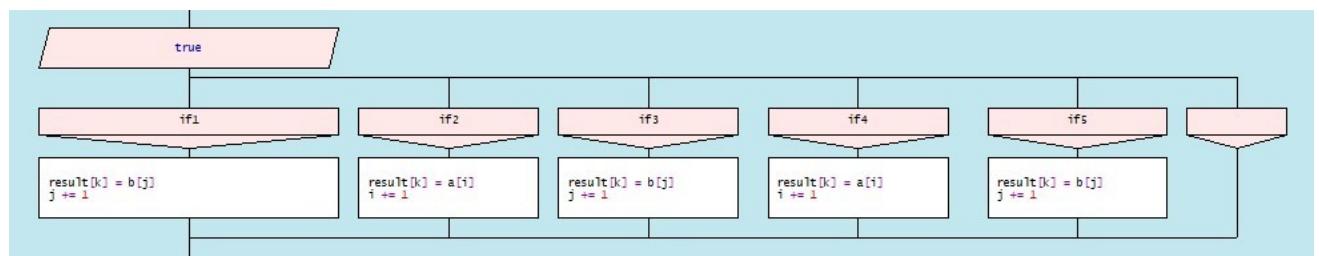


Рис.3.13 Фрагмент выбора разветвления (оператор Select)

3.1.3. Базовые конструкции циклической структуры

Циклическая структура предполагает многократное повторение одной и той же последовательности действий. Количество повторений определяется входными данными или условиями задачи. К циклическим структурам относятся, прежде всего, конструкция "Цикл-Для" ("for"), "Цикл для каждого" ("foreach"), составные конструкции "Цикл-До" ("while-do") и "Цикл-Пока" ("do-while").

а). "Цикл-Для" состоит из трех частей. В первой части фиксируется инициализация цикла. Во второй следует проверка условия завершения цикла. Если оно истинно, то выполняются операторы тела цикла до тех пор, пока это выражение не станет ложным. Если оно ложно, цикл заканчивается и управление передается следующему оператору. В третьей части увеличивается параметр цикла. Фрагмент дракон-диаграммы с конструкцией "Цикл-Для" имеет вид (рис. 3.14):

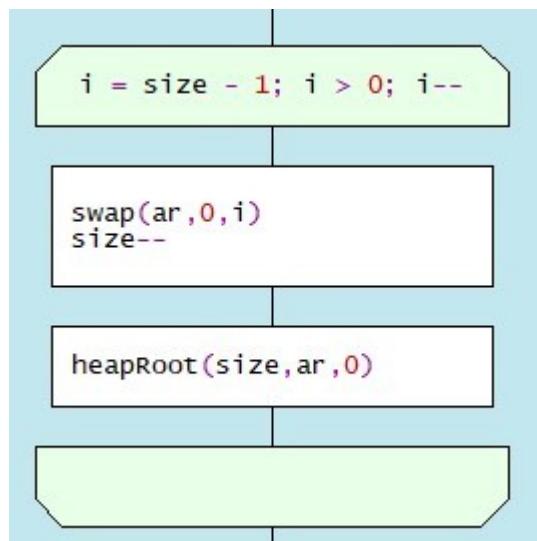


Рис.3.14. Фрагмент «Цикл-Для»

б). "Цикл для каждого" ("foreach") выполняет оператор или блок операторов для каждого элемента массива или списка данных (3.15).

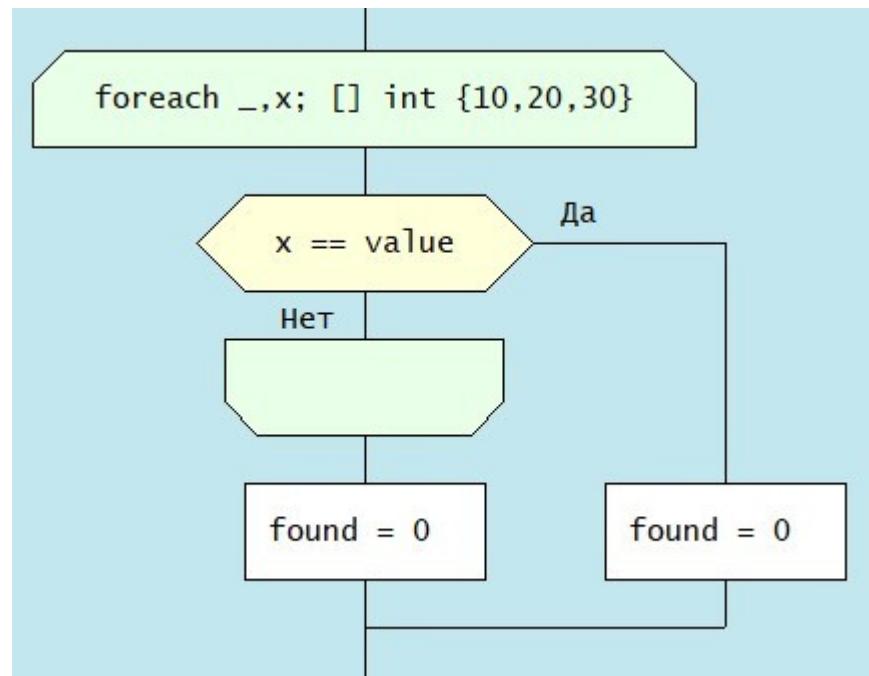


Рис. 3.15. Фрагмент диаграммы «Цикл для каждого»

в). пример составной конструкции “Цикл-До” (рис.3.16):

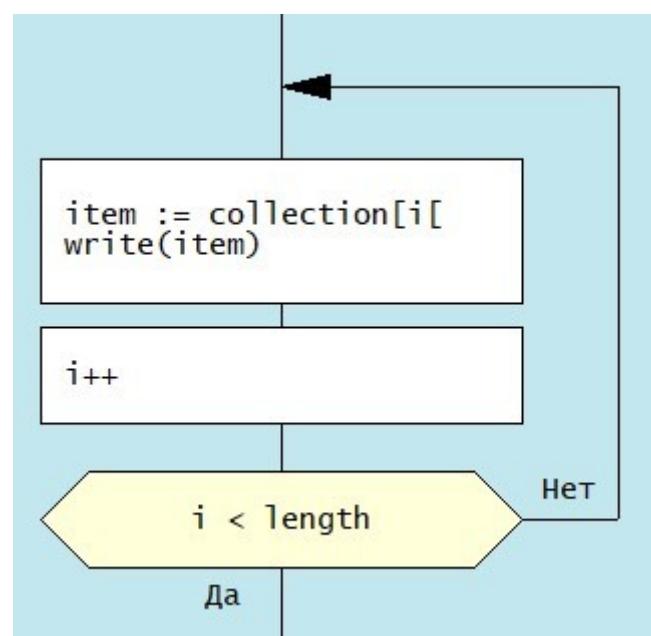


Рис. 3.16. Фрагмент диаграммы «Цикл-До»

г). пример составной конструкции “Цикл-После” (рис.3.17):

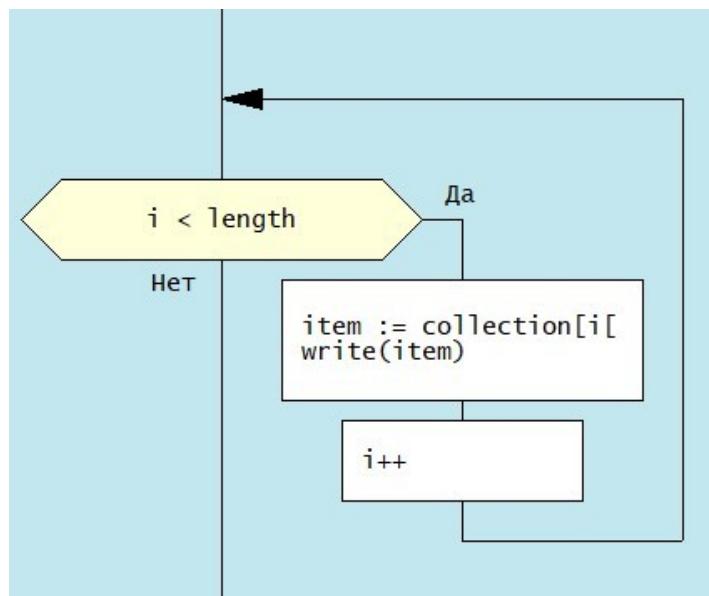


Рис. 3.18 Фрагмент диаграммы «Цикл-После»

3.1.4. Пример построения DRAKON-диаграммы

Покажем на конкретном примере построение диаграммы алгоритма вычисления наибольшего общего делителя методом Евклида. **Наибольший общий делитель (НОД)** – это число, которое делит без остатка два числа и делится само без остатка на любой другой делитель данных двух чисел. Проще говоря, это самое большое число, на которое можно без остатка разделить два числа, для которых ищется НОД.

Алгоритм нахождения НОД делением методом Евклида заключается в следующем:

1. Большее число делим на меньшее.
2. Если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла).
3. Если есть остаток, то большее число заменяем на остаток от деления.
4. Переходим к пункту 1.

Дракон-диаграмма алгоритма нахождения наибольшего общего делителя показана на рис.3.19.

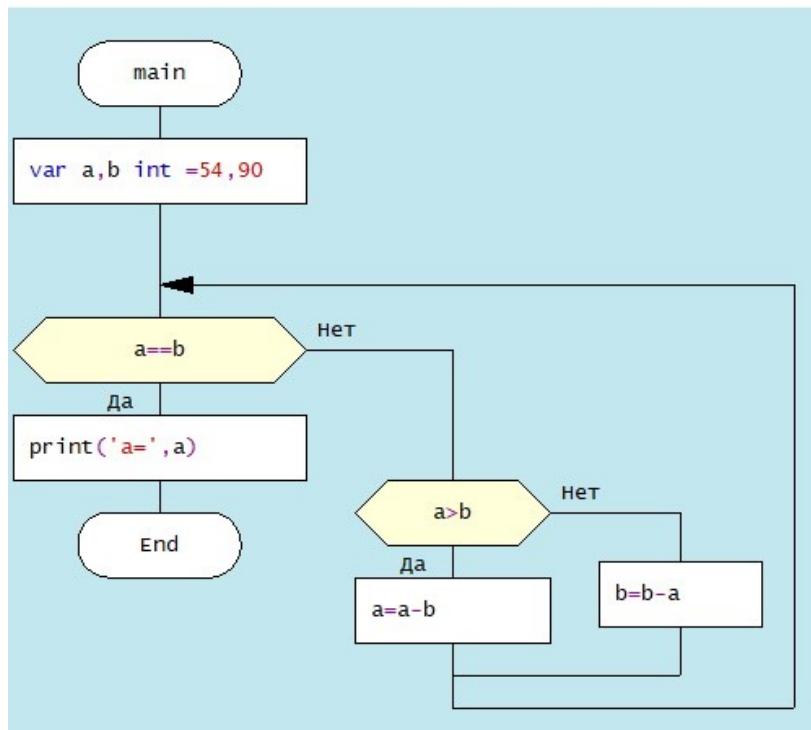


Рис.3.19 Дракон-диаграмма алгоритма Эвклида

3.4. От дракон-диаграммы до программного кода

Каждая дракон-диаграмма соответствует программному модулю. На рис.3.20 показана структура алгоритма сортировки массива, состоящая из ряда отдельных модулей (дракон-диаграмм) и дракон-диаграммы *main*, содержащей конструкции головной программы *main* на языке Go:

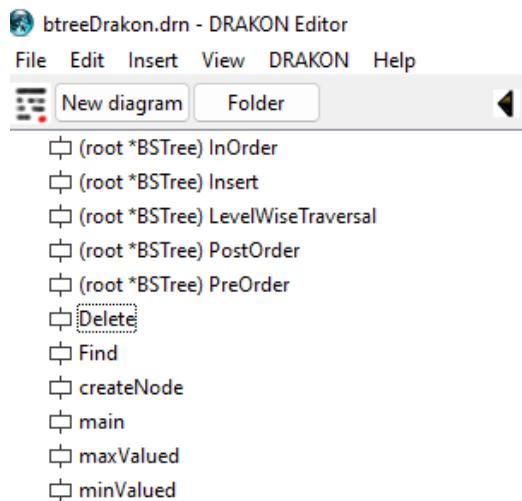


Рис. 3.20. Структура программы обработки деревьев

Как уже было изложено выше, структура программы на языке программирования Golang состоит из пакетов. Конструкции языка, входящие в пакет заносятся в раздел File/Description программного интерфейса редактора DRAKON WEB Editor (рис. 3.21.):

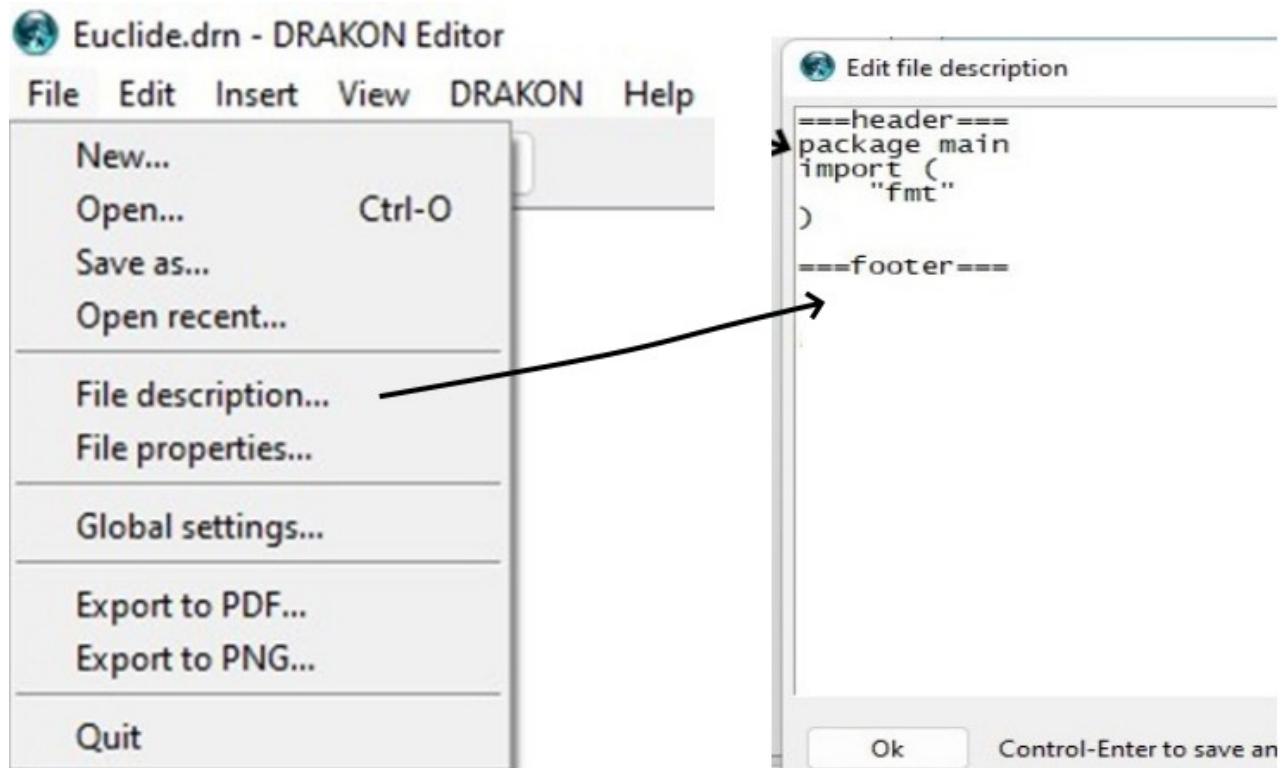


Рис.3.21. Порядок описания структуры программы через File/Description

В поле между метками **====header====** и **====footer====** следует располагать конструкции языка, доступные в каждом модуле программы, например, глобальные переменные.

Далее пользователь должен указать используемый язык программирования. Для этого нужно открыть опцию *File properties* (рис. 3.22):

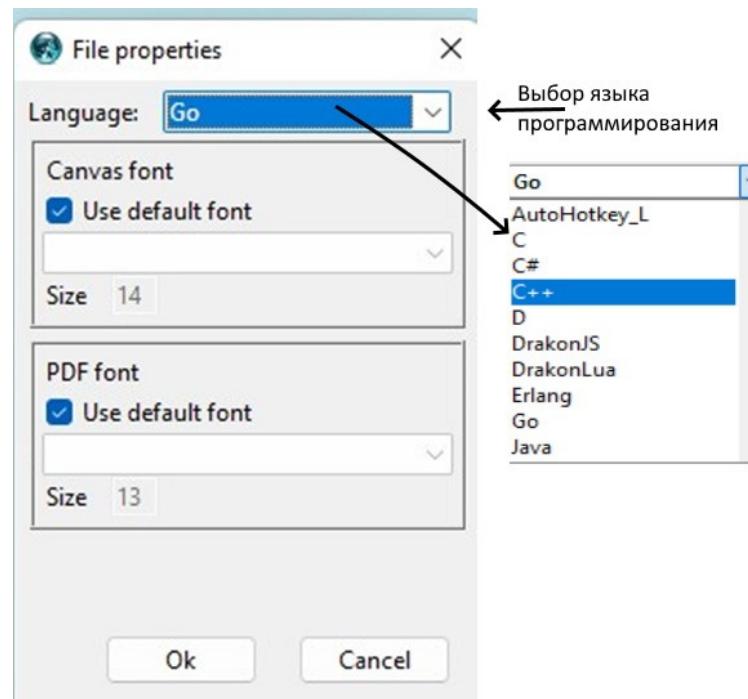


Рис. 3.22. Выбор языка программирования

Завершающим этапом разработки является генерирование программного кода (рис.3.23):

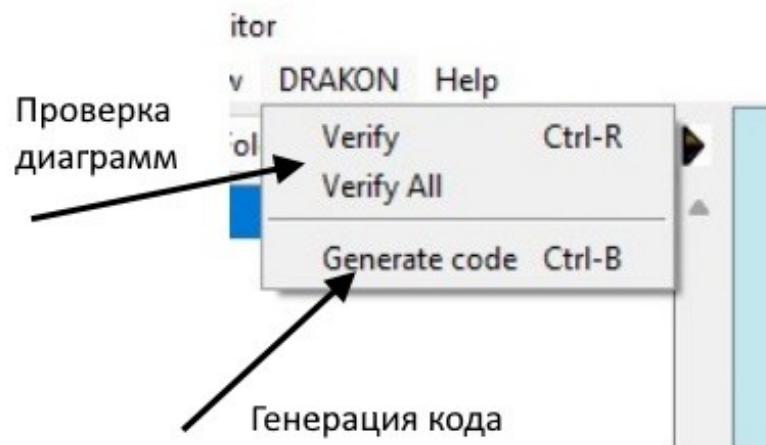


Рис. 3.23. Опция «Генерация кода»

Исходный код в формате .drn и сгенерированный код в формате .go сохраняются в одной и той же той директории (рис. 3.24):

3.5. Работа с сгенерированным кодом

Сгенерированный код можно открыть в одной из интегрированных сред разработки (IDE): Visual Studio Code, VIM, Eclipse, Atom, Sublime Text и ряда других. В данной работе используется IDE Visual Studio Code (VSC), описание которой не входит в содержание книги. Внешний вид среды программирования VSC представлен на рис.3.24:

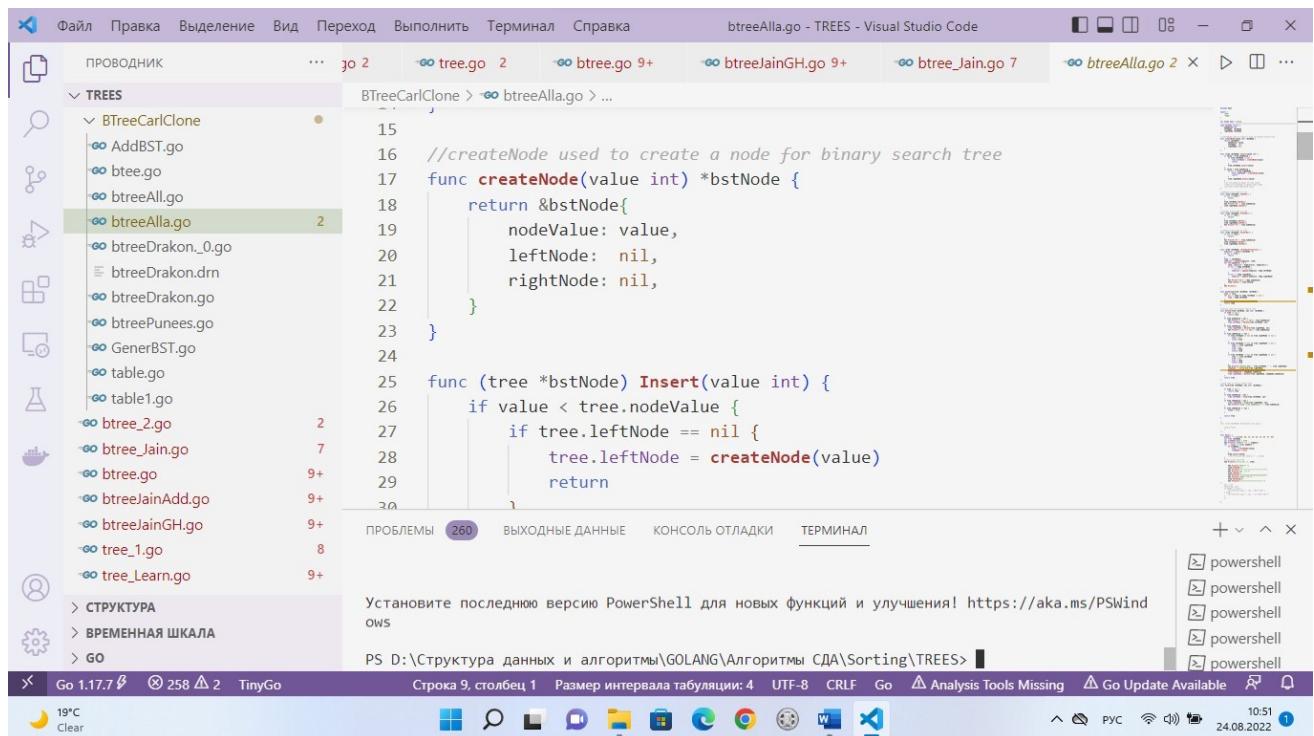


Рис.3.24. Внешний вид программы в среде Visual Studio Code

Практически в рамках гибридного подхода отладку программы зачастую приходится выполнять параллельно в двух средах: DRAKON WEB Editor и Visual Studio Code. Естественно, что при наличии графических синтаксических ошибок в дракон-диаграмме генерация кода производиться не будет, в то же время наличие смысловых ошибок в алгоритме код сгенерируется, но результаты окажутся неверными.

Процесс отладки сгенерированного кода рекомендуется проводить с помощью инструментария соответствующей среды программирования. После за-

вершения отладки все исправления необходимо внести в соответствующие графоэлементы дракон-диаграммы. Во всяком случае необходимо всегда следить за соответствием содержимого графоэлементов диаграммы и программного кода в соответствующей среде программирования.

РАЗДЕЛ 4 РЕАЛИЗАЦИЯ АБСТРАКТНЫХ ТИПОВ ДАННЫХ В GOLANG

4.1. Признаки структурированности данных

Напомним, - абстрактный тип данных представляет собой математическую модель данных и различные операторы, определенные в рамках этой модели. Алгоритмы обработки данных могут быть разработаны в терминах АТД, но для их реализации в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов (методов), поддерживаемых данным языком программирования.

В практической плоскости структура данных - это данные, структурированные таким образом, чтобы обеспечить их эффективное использование пользователями, что предопределяет необходимость определенного упорядочения данных, в первую очередь, на уровне хранения в компьютерной памяти. Уменьшение пространства и повышение уровня его организованности в совокупности с уменьшением временной сложности различных задач является основной целью работы со структурами данных.

Значительный опыт развития компьютерной техники и вычислительных технологий позволил классифицировать структуры данных по различным признакам. Во-первых, **по признаку сложности** различают *простые* структуры и *интегрированные*. Критерием простоты является неделимость данного, то есть в компьютерной реализации – простая цепочка битов. К простым, базовым структурам относятся переменные различных типов: целые, вещественные числа, логические, строковые.

Интегрированные (композитные, сложные) – это структуры данных, составными частями которых являются другие структуры данных, включая простые и интегрированные. Многие базовые интегрированные структуры данных предопределены конкретным языком программирования: массивы, срезы, струк-

туры и т.д. Такие структуры могут создаваться пользователями под конкретную проблемную задачу, используя базовые интегрированные структуры.

По способу представления структуры данных делятся на логическую и физическую. Логическая структура - это абстрактная схема расположения данных, которую представляет себе пользователь или программист. *Физическая структура* – это способ (схема) конкретного размещения данных в памяти вычислительной машины. В общем случае логическая и физическая структуры одних и тех же данных не совпадают. В логической (абстрактной) структуре данные как правило располагаются смежно по отношению друг к другу, в то время как в компьютерной реализации эти данные могут располагаться на различных участках памяти.

Важным признаком структуры данных является **наличие связей между элементами** структуры. По этому признаку различают *несвязные и связные*. Несвязные структуры характеризуются отсутствием связей между элементами структуры, в то время как связные структуры характеризуются наличием связи. К несвязным структурам относятся массивы, строки, стеки, очереди; к связным – связные списки.

Во многих случаях при работе с данными может играть роль такая их характеристика как *изменчивость*, то есть изменение числа элементов и (или) связей между элементами структуры. По этому признаку различают *статические, полустатические и динамические структуры*. К статическим относят массивы, множества, записи, таблицы, к полустатическим – стеки, очереди, деревья, к динамическим – линейные и разветвленные связные списки, графы, дерева.

По характеру упорядоченности элементов в структуре различают *линейные и нелинейные* структуры данных. Линейные структуры в зависимости от характера взаимного расположения элементов в памяти разделяют на структуры с последовательным распределением элементов в памяти (векторы, строки,

массивы, стеки, очереди) и структуры с произвольным связным распределением элементов в памяти (односвязные и двусвязные линейные списки). Нелинейные структуры - многосвязные списки, дерева, графы.

Одним из определяющих признаков структур данных является *способы доступа к данным*. В методе доступа важен механизм поиска – алгоритм, определяющий путь доступа, который возможен в рамках заданной структуры памяти, и количество шагов вдоль этого пути для нахождения искомых данных. Среди методов доступа к данным выделяют две основные группы: *последовательный* и *прямой* [<https://www.geeksforgeeks.org/memory-access-methods/>].

Последовательный доступ означает, что доступ к группе элементов осуществляется в заранее определенной упорядоченной последовательности. Примером последовательного доступа является односвязный список. *Прямой доступ* к различным элементам структуры данных осуществляется путем указания уникального адреса этих элементов.

Наконец следует отметить такой признак структур данных как *однородность*. К однородным относятся структуры, содержащие множество простых данных одного типа (числовые, логические, строковые и т.д.). Неоднородные структуры объединяют данные различных типов. Примерами однородных структур являются массивы, срезы, стеки. К неоднородным структурам относятся записи и множества.

Далее рассмотрим как реализуются структуры данных в языке программирования Golang.

4.2. Линейные структуры данных

Линейные структуры данных - это структуры, в которых элементы данных расположены в последовательном порядке. Линейные структуры можно различать по способам доступа к отдельным элементам той или иной коллекции данных и по признаку однородности данных (однородные и неоднородные).

4.2.1 Линейные структуры прямого доступа

К базовым линейным структурам прямого доступа относятся однородные (массивы и срезы) и неоднородные (записи, словари, хэш-таблицы).

a) Массив (Array)

Массив представляет собой коллекцию данных, принадлежащих к одному типу. Например, коллекция целых чисел 24, 12, 36, 6, 47, 11 образует массив (рис. 4.1.).

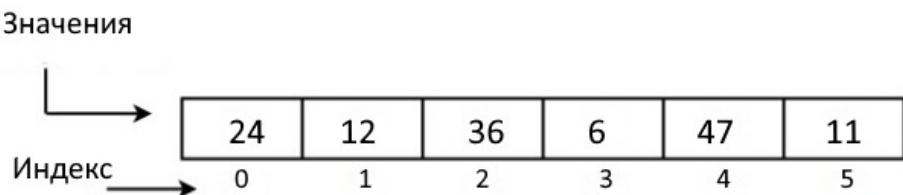


Рис.4.1. Одномерный массив

Смешивание значений разных типов, например массива, содержащего как символы, так и целые числа, в Go не допускается. Существуют различные способы объявления массивов:

```
var array_name []Type
```

или

```
var array_name[length]Type{item1, item2, item3, ...itemN}
```

Кроме того, в языке Go массивы могут быть объявлены в сокращенном виде:

```
array_name := [length]Type{item1, item2, item3,...itemN}
```

В языке Go можно создать многомерный массив, используя следующий синтаксис:

```
array_name[Length1][Length2]..[LengthN]Type
```

Однако данные в виде массива в Go используются достаточно редко. На многое более удобна такая коллекция данных как *срез*.

б) Срез (Slice)

Срез это коллекция данных переменной длины, в которой хранятся элементы однородного типа. Срез можно рассматривать как фрагмент массива.

Синтаксис среза (T – тип) данных:

`[]T`

или

`[]T{ }`

или

`[]T{value1, value2, value3, ...valuem}`

Срез состоит из трех компонентов: *указателя, длины и емкости*. Для создания среза в таком виде используется функция *make*. Для примера на рис.4.2. создание среза *parentSlice* выглядит так:

```
parentSlice = make([]int, 20, 20)
```

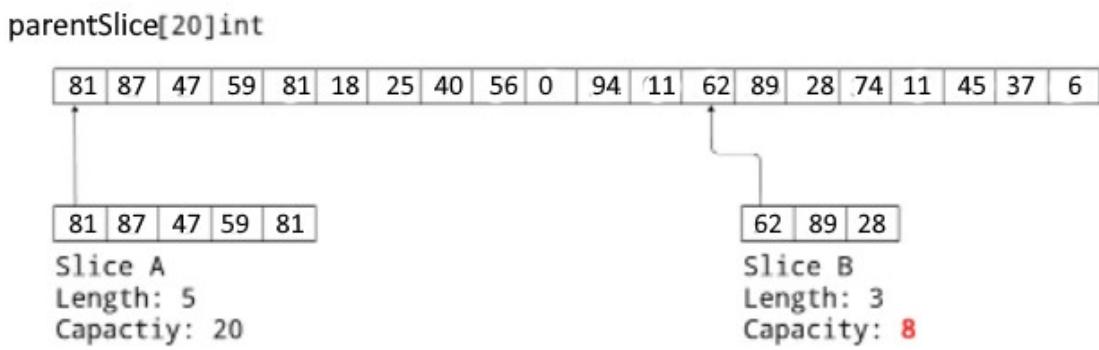


Рис.4.2. Создание разных срезов SliceA и SliceB

Указатель (*Pointer*) указывает на первый элемент массива, доступный через срез (который не обязательно совпадает с первым элементом массива). *Длина* (*Length*) — это количество элементов среза; она не может превышать *емкость* (*capacity*), которая, как правило, представляет собой количество элементов между началом среза и концом базового массива. Эти значения возвращаются встроенными функциями *len* и *cap*. На базе одного массива можно создать несколько срезов с разными значениями указателя, длины и емкости. На рис. 4.2. показан первичный срез/массив *parentSlice*, на базе которого создаются два среза разной длины, начинающиеся с разных мест. Для этого используется запись *sliceA := parentSlice[:5]* и *sliceB := parentSlice[12:15]*. Здесь емкость среза *SliceA* равна 20, а *SliceB* равна 8, поскольку Go определяет это значение как разность между длиной первичного массива (20) и индексом первого элемента среза *Slice B* (12).

Для работы со срезами используются такие функции:

1. Встроенная функция `append()` используется для добавления элементов в срез. Если размера базового среза недостаточно, то автоматически создается новый срез и в него копируется содержимое старого среза.
 2. Функция `len()` возвращает количество элементов, присутствующих в срезе.

3. Функция `cap()` возвращает емкость базового среза.

4. Функция `copy()`, содержимое исходного слайса копируется в срез назначения.

в) Структура (Structure)

Golang поддерживает коллекции данных в виде структур, состоящих из набора множественных типов данных (полей), представляемых как единая сущность. В Golang структура реализуется с помощью типа данных *Struct*:

```
package main
import (
    "fmt"
)
type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}
func main() {
    //создание структуры с указанием полей
    emp1 := Employee{
        firstName: "Петр",     age:      35,      salary:
    20000,   lastName:  "Семенов",
    }
    // создание структуры без указания имен полей
    emp2 := Employee{"Павел", "Кудинов", 49, 35000, }
    fmt.Println("Employee 1", emp1)
    fmt.Println("Employee 2", emp2)
}

Результат.
```

Employee 1 {Петр Семенов 35 20000}

Employee 2 {Павел Кудинов 49 35000}

Замечание. При выводе на монитор (*emp1*) порядок следования полей необязательно должен совпадать с порядком при создании структуры. Во втором случае создания структуры (*emp2*) порядок должен совпадать.

Golang позволяет создавать указатели на структуру.

```
package main

import (
    "fmt"
)

type Employee struct {
    firstName string
    lastName  string
    age       int
    salary    int
}

func main() {
    emp3 := &Employee{
        firstName: "Николай",
        lastName:  "Степин",
        age:        55,
        salary:    22000,
    }
    fmt.Println("First Name:", (*emp3).firstName)
    fmt.Println("Age:", (*emp3).age)
```

```
}
```

Результат.

```
First Name: Sam
```

```
Age: 55
```

г). Карта (Map)

Карты представляют собой коллекции, в которых хранятся неупорядоченные пары *ключ-значение*, где ключи – это уникальные идентификаторы, связанные с каждым значением в карте. Карты особенно эффективны в алгоритмах поиска данных. Ключи карты могут быть практически любого типа, в отличие от массивов и срезов, где для ключей используется последовательность целых чисел. Тип для ключей и значений в Go нужно уточнять. Для объявления карты с ключами типа *string* и значениями типа *int* используется синтаксис *map[string]int*:

map [string] int
 └─────────┘ └─────────┘
 тип тип
 ключа значения

Синтаксис	Смысл
<i>(mapName - имя карты)</i> <i>var mapName map [KeyType] ValueType</i>	объявить карту
<i>var mapName = map [KeyType] ValueType {}</i>	объявить и назначить пустую карту
<i>var mapName = map [KeyType] ValueType {key1: val1, key2: val2}</i>	объявить и назначить карту
<i>mapName: = make (map [KeyType] ValueType)</i>	объявлять и инициализировать карту размера по умолчанию
<i>mapName: = make (map [KeyType]</i>	объявить и инициализировать карту

<code>ValueType, length)</code>	размера длины
---------------------------------	---------------

Различные операции при работе с картами показаны на конкретном примере. Создадим карту, содержащую данные о сотрудниках и их возрасте `EmployeeAge`:

```
employeeAge := map[string]int{}
```

Инициализация этой карты:

```
employeeAge = map[string]int{
    "П. Петров": 45,
    "В. Маркин": 47
}
```

Добавление новой записи:

```
employeeAge["К. Маркин"] = 34
```

Вывод на монитор:

```
map[A. Костин:34 В. Маркин:47 П. Петров:45]
```

Карта может быть создана с помощью функции `make`:

```
employeeAge := make(map[string]int)
employeeAge["П. Петров"] = 45
```

Другие операции с картой.

Обновление ключа:

Получение значения, соответствующего ключу:

```
age := employeeAge["К. Маркин"]
```

Удаление пары "ключ-значение"

```
delete(employeeSalary, "Tom")
```

Проверка существования ключа:

```
val, ok := mapName[key]
```

Если ключ существует, переменная `val` будет значением ключа в `map`, а переменная `ok` будет `true`. Если ключ не существует, переменная `val` получит нулевое значение по умолчанию типа `value`, а переменная `ok` будет `false`.

3.2.1. Линейные структуры последовательного доступа

Последовательный доступ к данным означает, что за каждый конкретный момент времени можно обратиться лишь к одному элементу структуры, причем доступ к элементам происходит в определённом порядке. Классическими примерами структуры последовательного доступа являются *односвязный список, стек и очередь*.

a) Связный список (Linked List)

Связный список представляет собой динамическую структуру данных, элементами которых называют узлы, состоящие из двух частей: *содержательной* и *ссыпочной*. Содержательная часть, предназначенная для хранения значения данных, может быть представлена одним из базовых типов данных, такими как целое число, число с плавающей запятой, строка или какого-либо другого типа данных. Ссыпочная часть представляет собой ссылку, которая используется для хранения адреса следующего элемента в списке.

Связные списки находят свое применение при решении многих вычислительных задач от организации операционных систем до создания плейлистов. В частности, они полезны при обработке файловой системы: иногда трудно найти дисковое пространство для размещения всего файла, поэтому он может разбиваться на разрозненные фрагменты. Для организации работы с этими фрагментами формируется связный список участков, в которых фрагменты файла хранятся на диске. Этим они отличаются от массивов или срезов, где все элементы расположены смежно по отношению друг к другу.

Существуют различные типы связных списков. Прежде всего, они различаются по количеству полей (односвязные и двусвязные списки) и по способу связи элементов (линейные и циклические). В простейшем случае односвязного списка каждый узел (кроме последнего узла) содержит ссылку (указатель) на следующий узел того же списка. Именно в ссыпочной части содержится адрес

следующего узла. Сылочная часть последнего узла содержит значение *nil* (рис.4.3).

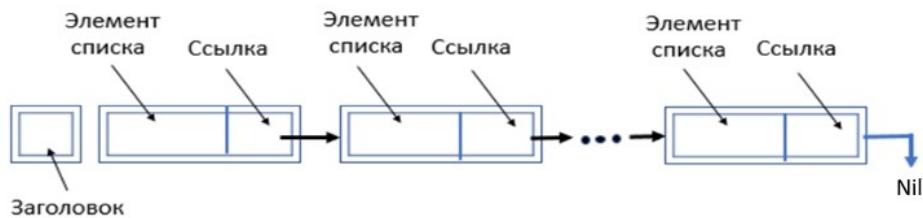


Рис. 4.3. Структура односвязного списка

Структуру отдельного узла связного списка можно описать типом данных *struct* в таком виде:

```
type Node struct {
    data string
    nextNode *Node
}
```

Структура связного списка содержит длину списка, головной узел и хвостовой узел:

```
type LinkedList struct {
    Len      int
    headNode *Node
}
```

Поле «*Len*» в структуре связного списка содержит его длину. В поле «*headNode*» хранится адрес памяти заголовка (первого узла связного списка). Инициализация (создание экземпляра) структуры типа *LinkedList* выполняется так:

```
var LL LinkedList = LinkedList{}
```

Язык Go предоставляет возможность реализовывать различные операции со связными списками, среди которых основными являются:

- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке;

Рассмотрим операцию вставки элемента в связный список (рис.4.4):

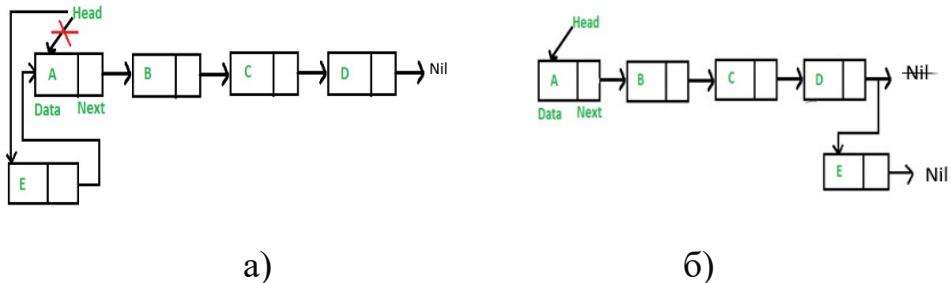


Рис. 4.4. Вставка нового узла в начало (а) и в конец (б) связного списка

Для вставки новых узлов используются три метода: в начало списка – *PushFront(val)*, в конец списка – *PushBack(val)* и в указанную позицию *PushVal(nodeVal, val)*, где параметр *val* – значение элемента списка того или иного типа (в данном случае – *string*), *nodeVal* – значение элемента списка, после которого вставляется новый узел. Рассмотрим последовательно алгоритмы всех трех методов. Удаление узлов реализует метод *RemoveVal(val)*

а) Метод PushFront – вставка нового узла в начало списка

Этот метод реализует процесс вставки нового узла в начало списка. Алгоритм метода представлен в виде дракон-диаграммы с последующей автоматической генерацией программного кода в редакторе DRAKON WEB Editor и его выполнения в редакторе Visual Studio Code (рис.4.5).

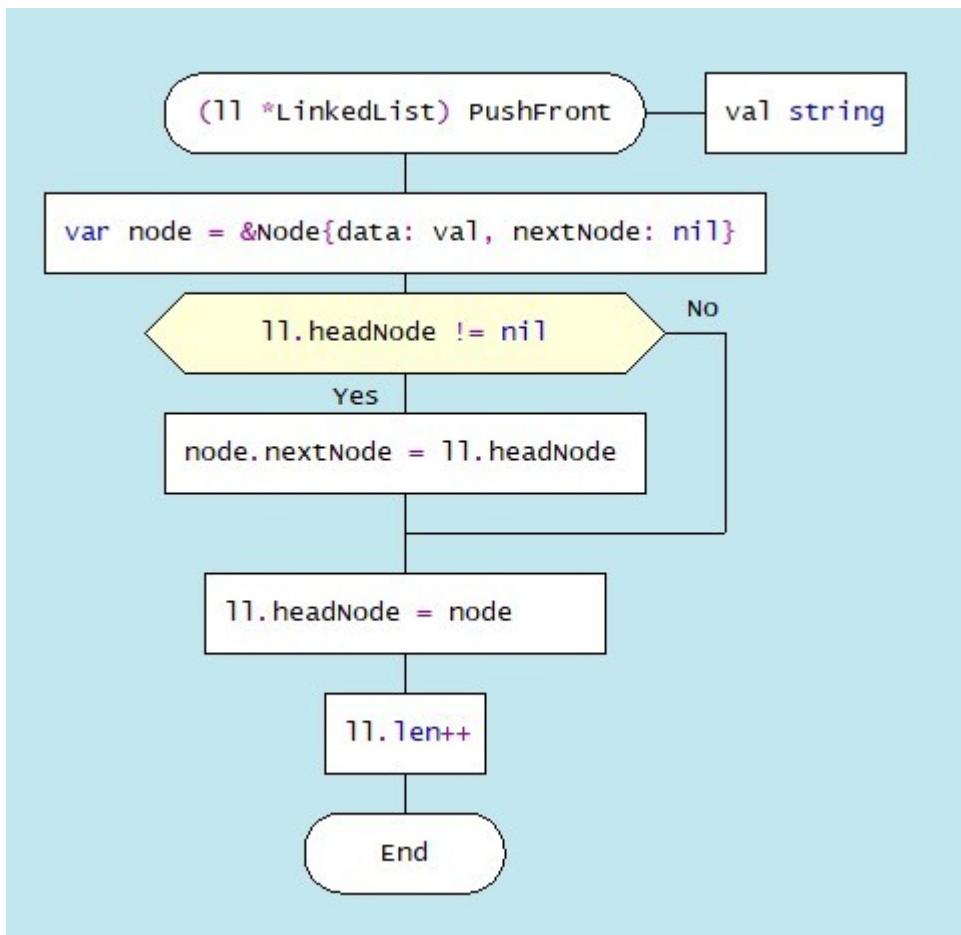


Рис.4.5. Дракон-диаграмма метода PushFront(val)

Программный код метода вставки в начало списка *PushFront*:

```

func (LL *LinkedList) PushFront(val string) {
    // LL - экземпляр данного типа LinkedList
    var uzel = &Uzel{data: val, next: nil}
    // объявляем переменную типа структура Uzel
    if LL.headUzel != nil {
        uzel.next = LL.headUzel
    }
    LL.headUzel = uzel // включение нового узла в список
    // новому начальному узлу присваивается адрес следующего узла
    LL.Len++ // прирост длины списка
}

```

Метод PushBack – вставка нового узла в конец списка

Вставка нового узла в конец односвязного списка реализуется методом *PushBack(val)*. Алгоритм вставки нового узла в конец списка заключается в следующих шагах. Создается новый узел (*newNode*), которому передается параметр (*val*). Если список пуст, то новый узел будет являться одновременно и первым, и последним в списке. Если список не пуст, то перебираются все узлы до конца списка и новый узел добавляется в конец списка. дракон-диаграмма метода *PushBack(val)* (рис. 4.5).

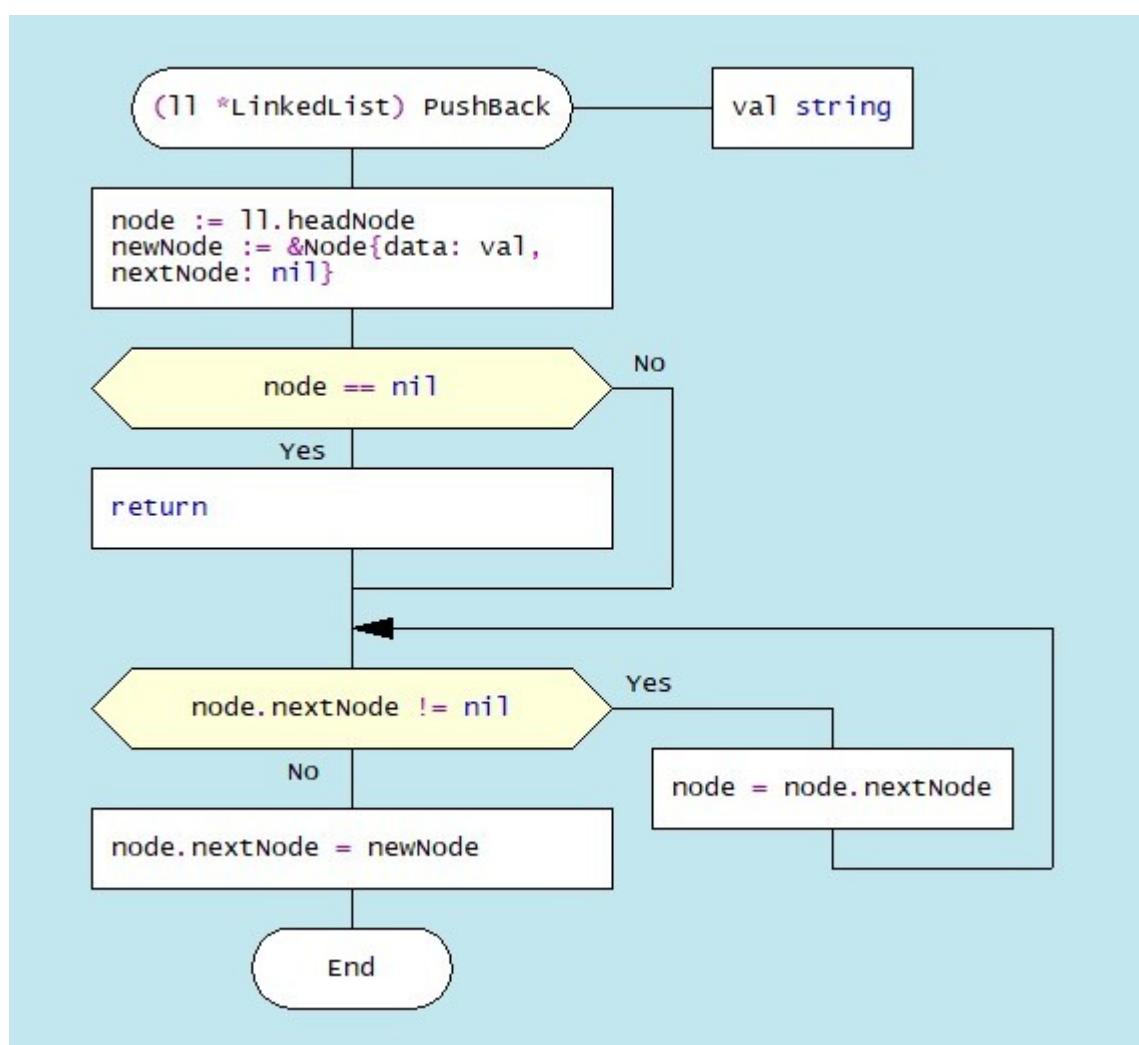


Рис.3.5. Дракон-диаграмма алгоритма вставки в конец списка *PushBack(val)*

```

func (LL *LinkedList) PushBack(val string) {
    uzel := ll.headUzel
    newUzel := &Uzel{data: val, next: nil}
    if uzel == nil {
        return
    }
    for uzel.next != nil { // если вставка после единствен-
        узла, то ==nil
        uzel = uzel.next
    }
    uzel.next = newUzel
    fmt.Println("PushBack > uzel.next = newUzel >>",
newUzel, uzel)
}

```

Метод *PushVal (nodeData, val)* - вставка после заданного узла

Третий метод *PushVal (nodeVal, val)* реализует процесс вставки нового узла с параметром *val* после узла со значением *nodeVal*. Алгоритм метода вначале обращается к модулю *NodeWithNode (nodeVal)*, определяющего узел, после которого нужно вставить новый узел. В нашем случае вставляется слово «наверное» в созданный выше список после узла с содержимым «Это». Дракон-диаграмма алгоритма такой вставки и соответствующий программный код показаны на рис. 4.7.

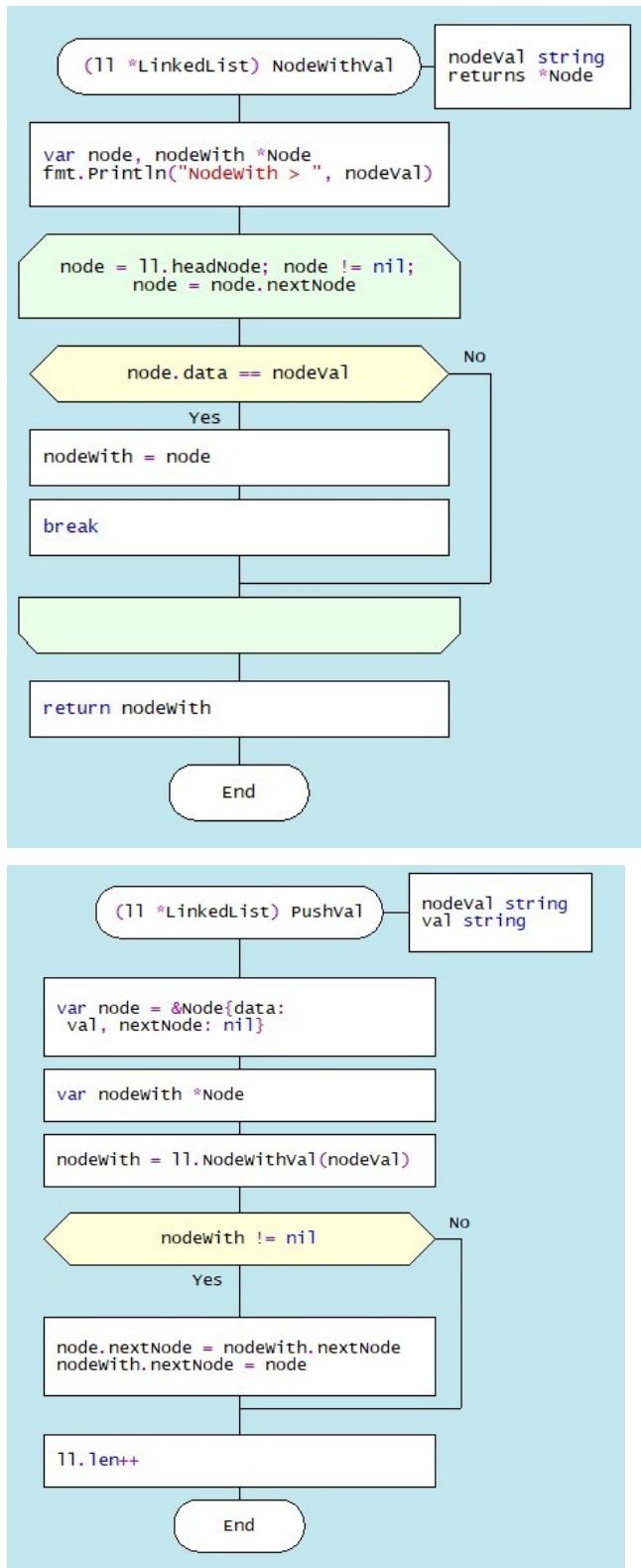


Рис. 4.7. Дракон-диаграммы алгоритмов вставки узла NodeWithVal PushVal

К основным методам работы с узлами связного списка относится методы удаления одного или несколько узлов либо удаление узлов по условию. Рассмотрим алгоритм удаления узла по заданному значению.

Метод RemoveKey (val) – удаление заданного узла

Удаление узла из связного списка после заданного значения (*k*) выполняется с помощью метода *RemoveVal (val)*, параметром которого является ключ *val* (поле *data* структуры узла списка *Node*). Для удаления узла списка с ключом *val* необходимо вначале найти этот узел. В этом модуле вначале проверяется первый узел (*ll.head.data == key*) и если ключ совпадает со значением поля *ll.head.data*, то адрес первого узла заменяется адресом следующего узла, который становится главным (*ll.head.next*). Далее в цикле происходит поиск узла с искомым ключом и сдвиг узлов (рис.4.8).

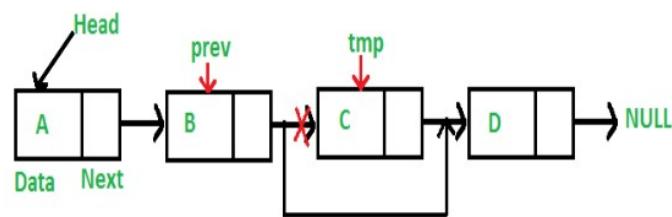


Рис. 4.8. Иллюстрация удаления узла из списка

Дракон-диаграмма алгоритма удаления узла по значению показана на рис. 4.9.

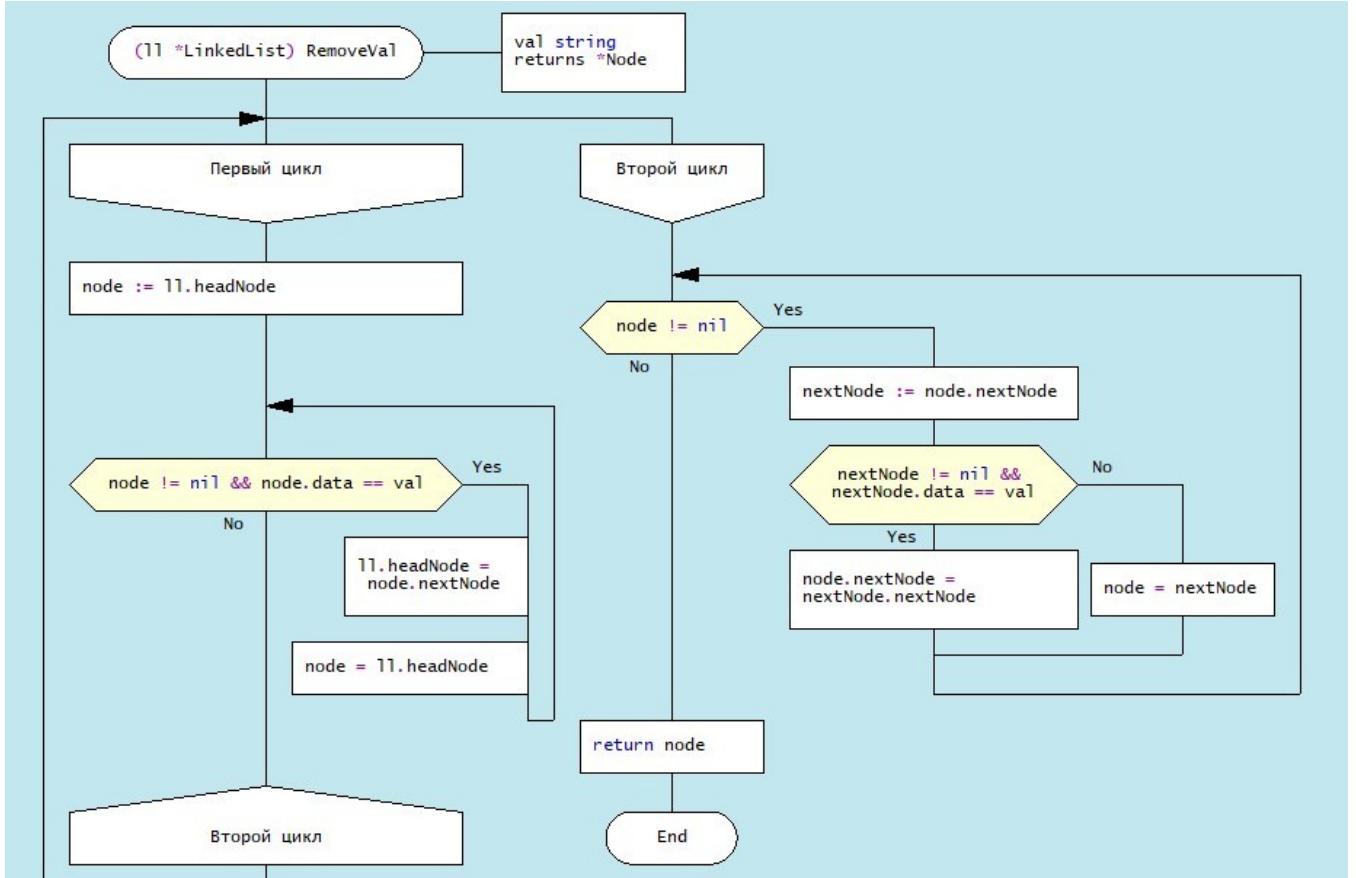


Рис. 4.9. Дракон-диаграмма удаления узла из списка RemoveVal

В качестве примера создадим список, состоящий из трех узлов, содержащих три значения: “Это”, “мой”, “список”. Для формирования списка необходимо создать тип `Node{data,nextNode}`, где `data` – значение, а `nextNode` – адрес следующего узла и тип `LinkedList(len, headNode)`, где `len` – длина списка, а `headNode` – заголовок списка с типом `*Node`:

Далее в функции `main()` нужно инициализировать и создать экземпляр списка `ll` и вставить первый узел (заголовок), вызвав метод `PushFront("Это")`. Далее вставляются новые узлы в конец списка с помощью метода `PushBack("мой")` и `"список"`:

```

func main() {
    var ll LinkedList = LinkedList{}
    ll.PushFront("Это")
    ll.PushBack("мой")
    ll.PushBack("список")
}
    
```

```
LL.PushBack("группы")
```

Далее нужно вставить новый узел со значением «наверное» после первого узла “Это”, после чего удаляем узел с содержанием “мой”:

```
LL.PushKey("Это", "наверное")
LL.RemoveVal("мой")
```

Результаты обработки узлов списка выводятся на монитор с помощью метода *IterateList()* (рис. 4.10.):

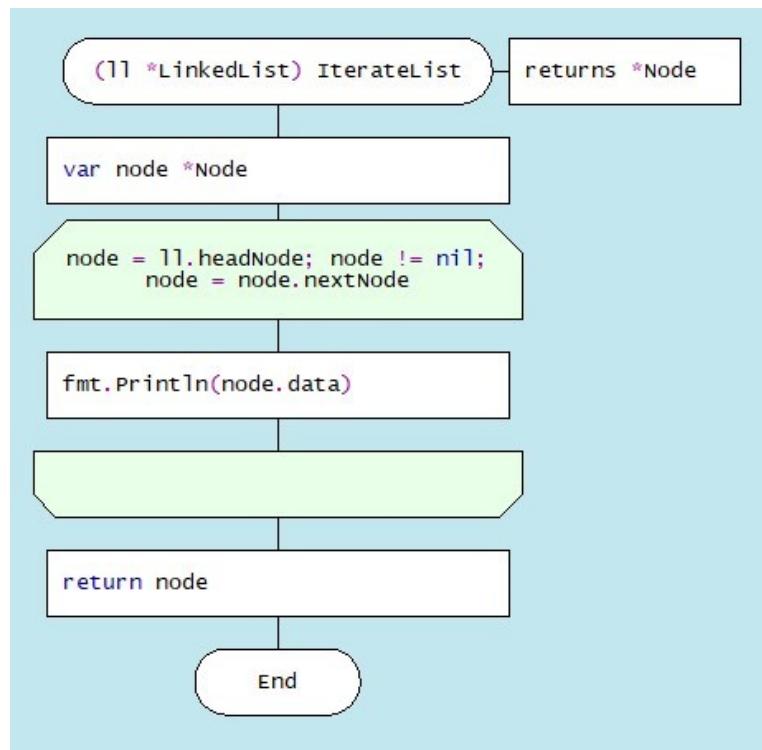


Рис. 4.10. Дракон-диаграмма метода *IterateList()*

Реализация этих модулей выполняется в функции *main()*:

```
func main() {
    var ll LinkedList = LinkedList{}
    ll.PushFront("Это")
    ll.PushBack("мой")
    ll.PushBack("список")
    ll.PushBack("группы")
    ll.PushVal("Это", "наверное")
    ll.IterateList()
```

```
    LL.RemoveVal("группы")
    LL.IterateList()
}
```

В представленном программном коде элементами списка являются данные типа *string*. Для того чтобы программа работала со списками, элементами которых являются числа (целые или вещественные), необходимо вместо встроенного типа *string* использовать сигнатуру `interface{}`` [Manuel]

Результат:

```
Длина связного списка = 5
Заполненный список
20 -> 30 -> 40 -> 50 -> 70 ->
Длина связного списка = 4
Связный список после удаления узла со значением 40
20 -> 30 -> 50 -> 70 ->
```

б). Стек (Stack)

Стек — это абстрактный тип данных, содержащий элементы с двумя основными операциями: *Push*, который добавляет элемент в коллекцию, и *Pop*, который удаляет последний добавленный элемент. В набор носителей этого типа включается набор всех стеков, содержащих элементы типа T, включая пустой стек, стек с одним элементом типа T, стек с двумя элементами типа T и так далее. С технологической точки зрения стек — это память, в которой значения данные загружаются и извлекаются согласно стратегии «последний пришел – первый ушел» (LIFO – Last-In-First-Out). Данные поступают в стек лишь с одной стороны, называющейся вершиной стека (рис. 4.11.):

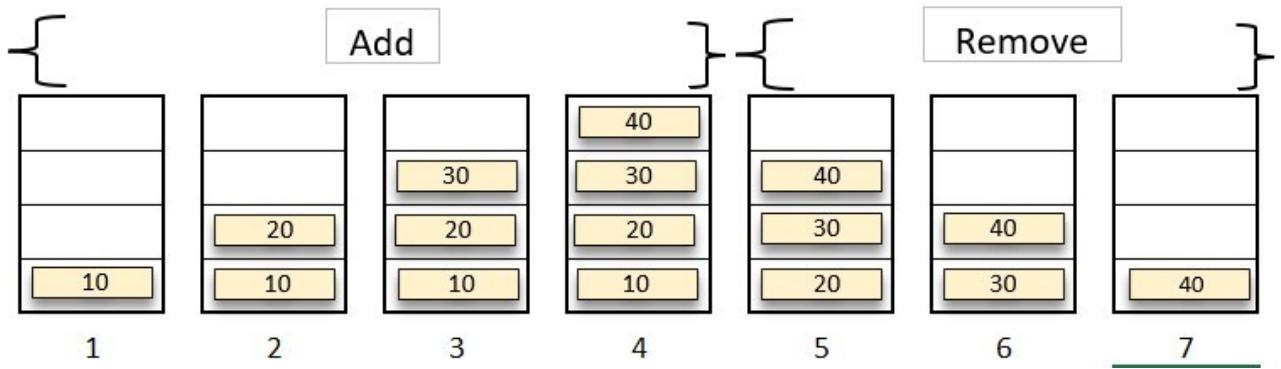


Рис.4.11. Работа стека «последний пришел – первый ушел»

Обычной иллюстрацией стека является стопка тарелок: в любой момент времени доступна только одна тарелка — та, которая поставлена на стопку последней, т. е. верхняя. Для того, чтобы получить доступ к произвольной тарелке внутри стопки, необходимо сначала последовательно снять все тарелки, находящиеся над ней. Наиболее частым применением этой структуры является реализация операции “Отмена последнего действия” в различных редакторах (Ctrl+Z). Другой пример применения организации данных в стеке — рекурсивные обращения функций. Каждый адрес обращения к функции запоминается в стеке и извлекается в обратном порядке.

Согласно предназначению стека основными операциями с его элементами являются функция (метод) добавления *Push(item)* и функция (метод) извлечения *Pop()*:

```

func (s *Stack) Push(str string) {
    *s = append(*s, str) // append встроенная функция добавления
}

func (s *Stack) Pop() (string, bool) {
    if s.IsEmpty() {
        return "", false
    } else {
        index := len(*s) - 1 // Получить адрес верхнего элемента.
        element := (*s)[index] // Указать на срез и получить элемент
    }
}

```

```

    *s = (*s)[:index]      // срез от 0 до index.
    return element, true
}
}

```

Здесь `*s` – указатель на адрес стека; при каждом обращении определяется длина стека, вычисляется параметр `index` для формирования нового среза от 0 до `index`. В результате извлечений всех элементов стек оказывается пустым.

В процессе вычислений используется дополнительная функция `IsEmpty()`, которая определяет состояние заполненности стека: пуст стек или заполнен. Возвращает 1, если стек пуст или возвращает 0:

```

func (s *Stack) IsEmpty() bool {
    return Len(*s) == 0
}

```

Прежде чем выполнять какие-либо операции со стеком рассмотрим способы программного создания стеков в языке программирования Golang. Существует два основных подхода к созданию стеков: первый - на основе среза (*slice*), второй – на основе односвязного списка (*Linked List*). Согласно первому подходу в функции `main()` вводится новый тип `Stack` в виде среза:

```
s.data = s.data[:len(s.data)-1]:
```

Это ключевая строка, которая удаляет элемент из стека. Она работает, изменяя срез `s.data`, чтобы исключить последний элемент. В Go срезы являются динамическими, и вы можете изменить их размер, используя синтаксис `s.data[:n]`, который возвращает первые `n` элементов среза. Здесь `n` равно `len(s.data)-1`, что означает “все элементы, кроме последнего”. Таким образом, последний элемент удаляется из стека.

в). Очередь (Queue)

Очередь (*Queue*) — это линейная структура данных, отличающаяся от стека порядком удаления элементов: в стеке удаляется последний добавленный элемент; в очереди - наоборот, удаляется элемент, добавленный первым (рис. 4.12).

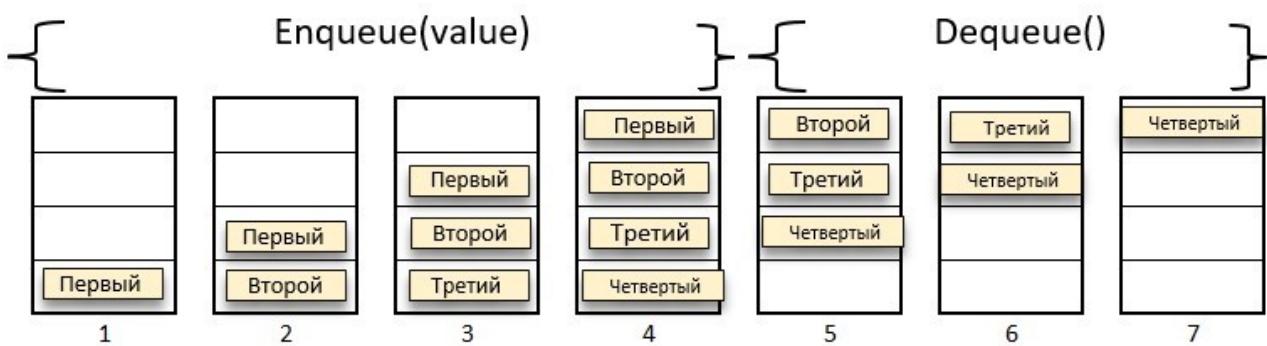


Рис. 4.12. Работа структуры «очередь»

Структура данных в виде очереди находит свое применение в многозадачных системах в системах связи (сетях с промежуточным хранением), в сетях массового обслуживания. Очереди играют важную роль в вычислениях, когда ресурсы предоставляются в порядке очереди, например, задания, отправленные на принтер, или процессы, ожидающие процессора в операционной системе.

Абстрактный тип данных *Queue* определяется как класс, объекты которого реализуют принцип FIFO или First-In-First-Out для добавляемых и удаляемых элементов. С точки зрения концепции АТД очередь — это контейнер, содержащий значения определенного типа. Набор носителей этой структуры данных — это набор всех очередей, содержащих элементы типа Т, включая пустую очередь. Набор основных операций, поддерживаемых очередью, включает в себя:

Add(s) – добавление одного элемента в конец очереди.

`Remove()` - удаление одного элемента из начала очереди.

Создание очереди реализуется в Golang либо с помощью среза (*Slice*) либо на основе связного списка (*Linked List*). В первом случае для создания очереди вводится тип в виде структуры :

```
type SliceQueue struct {
    queue []string
}
```

Здесь поле `queue []string` - целочисленный срез, в котором будут храниться данные.

Ниже представлены основные методы, связные с обработкой элементов очереди - добавление в конец очереди нового элемента `Add(s)` и удаление элемента `Remove()`:

```
func (q *SliceQueue) Add(s string) {
    q.queue = append(q.queue, s)
}

func (q *SliceQueue) Remove() string {
    temp := q.queue[0]
    q.queue = q.queue[1:]
    return temp
}
```

Дополнительными методами по отношению к структуре *Очередь* являются методы `IsEmpty()`, возвращающий `true` в случае пустой очереди, `Size()`, возвращающий длину очереди и некоторые другие.

Работа с очередью в функции `main()` начинается с создания экземпляра очереди типа *SliceQueue* с нулевой длиной:

```
queueSlice := &SliceQueue{make([]string, 0)}
```

Далее реализуем иллюстрацию работы двух экземпляров очереди, первый – для числовых данных, второй – для строковых.

Результат работы программного кода:

Очередь можно создавать на основе связного списка. Каждый новый узел создается методом *newQueue()*, имеющим тип **Queue*:

```
func newQueue() *Queue {
    return &Queue{
        nil,
        nil,
        0,
    }
}
```

Ввод каждого нового узла реализуется методом *enqueue(val)*:

```
func (LL *Queue) enqueue(value int) {
    var node *Uzel = getUzel(value) // Создать новый узел
    if LL.headUzel == nil {
        LL.headUzel = node // Добавить узел в начало очереди
    } else {
        LL.tailUzel.next = node // Добавить узел в конец очереди
    }
    LL.count++ // увеличить длину очереди
    LL.tailUzel = node
}
```

РАЗДЕЛ 5. АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ ПО ВРЕМЕНИ И ПРОСТРАНСТВУ/ СТАБИЛЬНОСТЬ АЛГОРИТМОВ

5.1. Свойства алгоритмов

В предыдущих разделах основное внимание уделялось описанию наиболее распространенных структур данных, с одной стороны как абстрактных сущностей, а, с другой – их конкретную реализацию в языке программирования Golang. В этом разделе речь пойдет о теоретическом анализе алгоритмов обработки структур данных с позиций оценки затрат времени на выполнение программного кода и занимаемого объема компьютерной памяти.

В теории и на практике алгоритмы обработки структур данных определяется как набор последовательно выполняемых, устойчивых процедур, обеспечивающих достижение конечного вычислительного результата, исходя из фиксированного и ограниченного набора исходных данных. Большинство алгоритмов обработки данных включают в себя такие базовые процедуры обработки данных как *поиск элементов, их сортировка, добавление, обновление, удаление и т.д.*

Прежде всего следует отметить, что «хороший алгоритм», в смысле способности получения конечного результата за приемлемое время, используя приемлемый объем памяти, должен обладать следующими основными характеристиками: *корректность и конечность*,

Корректность означает, что если алгоритм создан для решения определенной задачи, то для всех исходных данных он должен всегда давать правильный результат и ни для каких исходных данных не будет получен неправильный результат.

Конечность алгоритма означает, что для получения результата нужно выполнить конечное число вычислительных операций, т. е. исполнение программного кода в некоторый момент времени должен прекратиться.

После обеспечения *корректности* и *конечности* алгоритма важнейшей его характеристикой является *эффективность*, достижение которой обеспечиваются выполнением следующих требований:

1. Алгоритм должен эффективно использовать ресурсы, доступные системе;
2. Вычислительное время (время, затрачиваемое на генерацию выхода, соответствующего конкретному входу) должно быть минимальным;
3. Память, используемой алгоритмом, также должна быть как можно меньше.

В большинстве вычислительных задач необходимо стремиться к компромиссу между вычислительным временем и занимаемой памятью. Иными словами, при выборе метода решения вычислительной задачи нужно определиться с приоритетами: что важнее – вычислительное время или объем занимаемой памяти. Может показаться, что для современных компьютеров это не так уж и важно. Однако при решении сложных практических задач, связанных с обработкой огромных массивов данных, например, для задач искусственного интеллекта или для задач биоинформатики (например, расшифровка генома) поиск оптимального сочетания вычислительного времени и компьютерной памяти становится определяющим. В связи с этим будем определять термин "сложность алгоритма" как мера количества времени и/или пространства, необходимого для решения задачи в зависимости от размера данных.

Такой анализ включает в себя определение функции, которая связывает длину входных данных алгоритма с временем, которое он затрачивает (его временная сложность) или количеством мест хранения, которые он использует (сложность пространства). Алгоритм считается эффективным, когда значения этой функции либо ничтожны либо медленно растут относительно роста размера входных данных.

5.2. Теоретический анализ сложности алгоритмов

В конечном счете созданные алгоритмы должны быть реализованы на каком-либо вычислительном устройстве с помощью сгенерированного программного кода. Поэтому следует различать сложность самого алгоритма и сложность его компьютерной реализации. Самый быстрый алгоритм, реализованный на медленном вычислительном устройстве может оказаться менее эффективным, чем не очень удачный алгоритм, реализованный на компьютере, обладающим большей вычислительной мощностью или языком программирования, способным выполнять параллельные вычисления.

Таким образом при оценке сложности алгоритмов проводят на абстрактной машине с произвольным доступом к памяти, что позволяет не учитывать низкоуровневые параметры вычислительного устройства (размер памяти процессора, многозадачность и т.д.). Модель такой машины состоит из памяти и процессора, которые работают следующим образом:

- память состоит из ячеек, каждая из которых имеет адрес и может хранить один элемент данных;
- каждое обращение к памяти занимает одну единицу времени, независимо от номера адресуемой ячейки;
- количество памяти достаточно для выполнения любого алгоритма;
- процессор выполняет любую элементарную операцию за один временной шаг;
- циклы и функции не считаются элементарными операциями.

Трудоемкость алгоритма T_n , связанной с оценками его сложности определяется подсчетом количества выполняемых операций. Для примера рассмотрим алгоритм поиска максимального элемента массива.

```
package main
```

```

import "fmt"

func main() {

    array:= []int{11, 9, 17, 45, 411}
    N := len(array)
    maxNumber := array[0]
    for item:= 1; item < N; item++ {
        if array[item] > maxNumber {
            maxNumber = array[item]

    }

    }
    fmt.Println("Наименьшее значение элемента массива = ", maxNumber)
}

```

При реализации этого алгоритма будут выполнены:

1. $(N - 1)$ операция присваивания счетчику цикла i нового значения;
2. $(N - 1)$ операция сравнения счетчика со значением N;
3. $(N - 1)$ операция сравнения элемента массива со значением maxNumber;
4. от 1 до N операций присваивания значения переменной maxNumber.

Является очевидным, что такой подсчет для определения трудоемкости алгоритма и оценки его сложности является индивидуальным и зависит от размера входных данных. Поэтому совершенно естественно возникло предложение о качественной оценке сложности: в зависимости от «качества» исходных данных: *наилучшего* варианта, *наихудшего* и *среднего*.

5.3. Качественная оценка сложности

Для качественной оценки сложности алгоритмов используются следующие асимптотические обозначения: *омега-нотация*, *тета-нотация*, и *нотация Big-O*, отражающие три границы времени работы алгоритма (нижнюю, среднюю и

верхнюю). *Омега-нотация* представляет собой нижнюю границу времени работы алгоритма, что характеризует наилучшую сложность алгоритма в смысле достижения эффективности. Для любого значения n исходных данных минимальное время, требуемое алгоритмом, задается обозначением $\Omega(f(n))$.

Нотация *Big-O* представляет верхнюю границу времени выполнения алгоритма, которая характеризует наихудшую сложность алгоритма в смысле эффективности. В большинстве практических случаях именно эта нотация используется для оценки эффективности алгоритмов, поскольку разработчик программного обеспечения должен оценивать именно наихудший сценарий вычислительного процесса. Наконец, *тета-нотация (Θ-нотация)* представляет собой верхнюю и нижнюю границу времени выполнения алгоритмов и используется для их анализа средней сложности. Рассмотрим более подробно нотацию *Big O*, определяющую верхнюю границу любого алгоритма, то есть алгоритм не может занять больше времени, чем время верхней границы.

Начнем с простейшего **алгоритма с постоянным временем** — это алгоритм, который требует одинакового количества времени, независимо от его входных данных, сложность которого обозначается как $O(1)$. Например, даны два числа, требуется вычислить сумму.

Далее следует **логарифмическая временная сложность** $O(\log n)$. Когда время, затрачиваемое алгоритмом на выполнение, пропорционально логарифму входного размера n , говорят, что он имеет логарифмическую временную сложность. В качестве примера можно привести простой подсчет числа операций печати индекса цикла:

```
func main() {
    n := 16
    for i:= 1; i < n; i=i*2 {
        fmt.Println("i = ", i)
    }
}
```

```
}
```

Напомним, что $\log_2 16 = \log_2 2^4 = 4$ $\log_2 2 = 1$, поскольку $\log_2 2 = 1$). Алгоритмы с логарифмической сложностью считаются высокоэффективными, поскольку отношение времени выполнения одной операции к размеру массива данных уменьшается с увеличением его размера. Алгоритмы, работающие за логарифмическое время, обычно встречаются при операциях с двоичными деревьями или при использовании двоичного поиска. Такие алгоритмы будут рассмотрены в следующем разделе.

Линейная времененная сложность ($O(n)$) характеризует алгоритмы, время выполнения которых линейно зависит от размера входных данных (n). Простейшим примером может служить задача на вычисления суммы n чисел:

```
func main() {  
    n := 16  
    sum := 0  
    for i:= 0; i < n; i++ {  
        sum = sum + i  
    }  
    fmt.Println("sum =", sum)  
}
```

Типичными примерами линейной временной сложности являются алгоритмы, связные с сортировкой массивов, что будет показано в следующем разделе. При этом важно заметить, что алгоритмам с линейной сложностью не требуется дополнительной памяти.

Далее следует **квадратичная сложность алгоритмов $O(n^2)$** , число операций в которых находится в квадратичной зависимости от размера входных данных. В качестве примера можно привести алгоритм умножения двух чисел в интервале $[1:n; 1:m]$:

```

func main() {
    n := 3
    m := 5
    pow := 1
    for i:= 1; i < n; i++ {
        for j:= 1; j < m; j++ {
            pow = pow*i*j
        }
    }
    fmt.Println("pow =", pow)
}

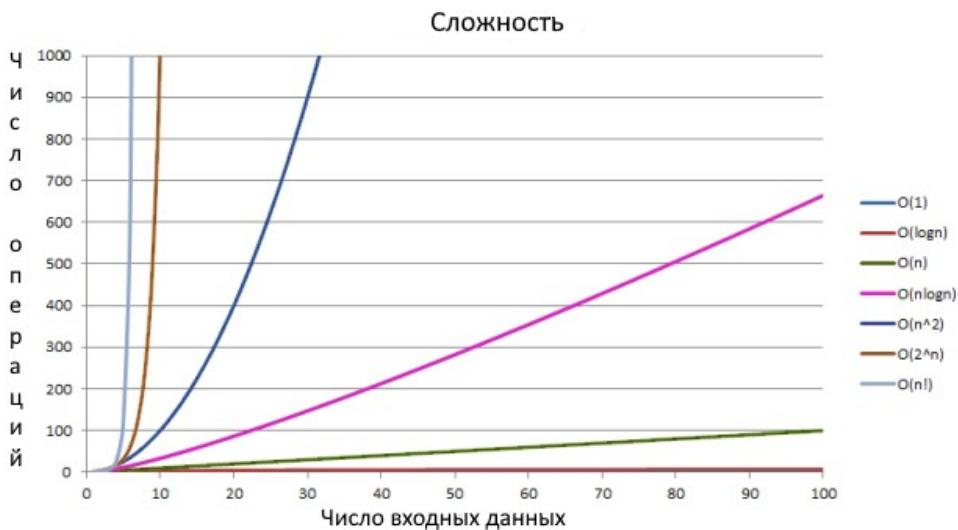
```

Сравнение рассмотренных сложностей представлено в виде таблицы.

Табл. Сравнение общих сложностей

n	Константа O(1)	Логарифмическая O(log n)	Линейная O(n)	Линейная логарифмическая O(n log n)	Квадратичная O(n^2)
1	1	1	1	1	1
2	1	1	2	2	4
4	1	2	4	8	16
8	1	3	8	24	64
16	1	4	16	64	256

Или в виде графика (рис.5.1.).



Как видно из этой таблицы и графика, по мере увеличения сложности функции количество вычислений или времени, необходимого для выполнения функции, может значительно возрасти. Далее при описании алгоритмов будет представлены оценки их сложности.

Другая характеристика вычислительной сложности алгоритмов связана с пространственной сложностью, которая оценивается как объем памяти, необходимой алгоритму в его жизненном цикле. Это пространство состоит из фиксированной и переменной частей. Фиксированная часть, представляет собой пространство, необходимое для хранения простых переменных и констант, а также размера программы, которые не зависят от сложности задачи. Переменная часть — это пространство, требуемое переменными, размер которого полностью зависит от размера задачи. Например, пространство стека рекурсий, динамическое выделение памяти и т.д.

Пространственная сложность оценивается как $O(1)$ в случае использования только входного массива данных. Если же для решения задачи потребуется вспомогательная память такого же размера как и входная память (память для хранения исходного массива), то пространственная сложность оценивается например, как $O(n^2)$.

РАЗДЕЛ 6. БАЗОВЫЕ АЛГОРИТМЫ СОРТИРОВКИ

6.1. Общая характеристика алгоритмов сортировки

Операции сортировки находят свое применение практически во всех сферах человеческой деятельности. Алгоритмы сортировок являются одними из наиболее распространенных в ИТ-технологиях обработки данных. В общем случае задачу сортировки можно сформулировать следующим образом: существует последовательность однотипных записей, одно поле из которых выбрано в качестве ключевого (*ключ сортировки*). Требуется преобразовать исходную последовательность в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа. Цель сортировки – облегчение последующего поиска элементов в отсортированном множестве.

В настоящее время разработано достаточно большое количество алгоритмов сортировки, обладающих различными возможностями применения в зависимости от числа элементов в наборе. Одни алгоритмы просты в реализации и подходят для небольших наборов входных данных, однако в случае больших наборов они требуют большего времени. Другие алгоритмы эффективны для сортировки больших наборов данных, но их использование для малых наборов просто не эффективно. Кроме того, алгоритмы различаются по сложности реализации. Понимание некоторых из них требует визуального сопровождения, в частности, с помощью дракон-диаграмм для представления логики алгоритма и поясняющих иллюстраций. Наборы данных в виде среза практически охватывают и массивы, поэтому в данном разделе рассматривается сортировка среза.

6.2. Сортировка «пузырьком»

Сортировка «пузырьком» - простейший алгоритм, легко реализуемый для наборов с небольшим количеством элементов. Свое название алгоритм получил вследствие того, что большие значения постепенно «всплывают» в конце набора. Дракон-диаграмма алгоритма представлена на рис. 6.1. Реализация алгоритма сортировки методом «пузырька» состоит из двух модулей: *main()* и *bubblesort (ar [] int)*.

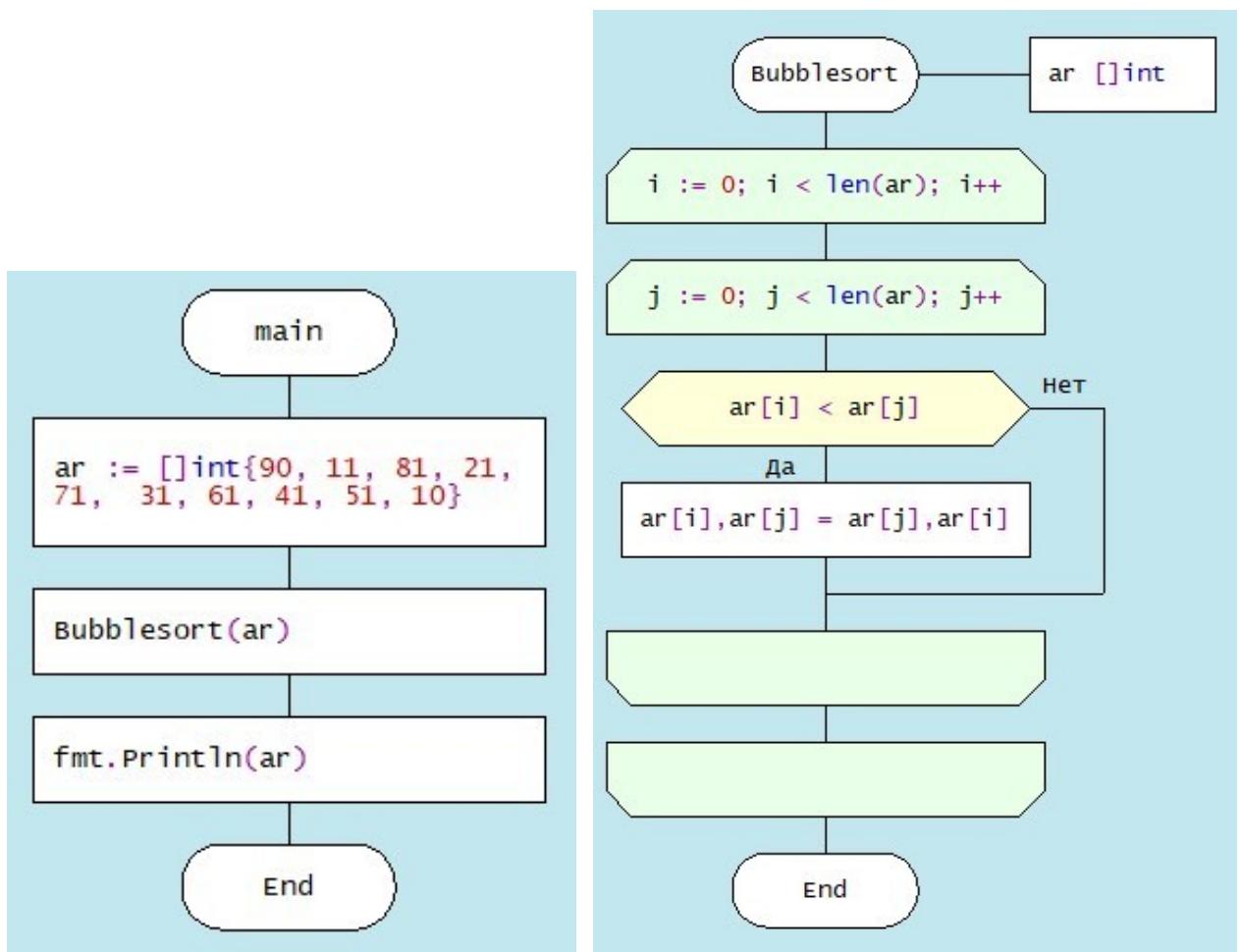


Рис. 6.1. Дракон-диаграмма алгоритма «пузырьковой» сортировки

В этом алгоритме проход по набору осуществляется по индексу (*j*) для каждого индекса (*i*). При этом сравниваются каждые пары значений *ar[i]* и *ar[j]*. Если ставится задача отсортировать значения в порядке возрастания, тогда два элемента меняются местами, если значение элемента *ar[j]* меньше значения элемента *ar[i]*. В противном случае происходит переход к следующему проходу

по индексу (i). Таким образом, самые большие значения появляются в конце набора. На рис.6.2. представлены фрагмент алгоритма при $i = 8$, а на рис.6.3. отображены только строки, в которых произошел обмен элементами.

Индекс j	Фрагмент алгоритма при $i = 8$									
	0	1	2	3	4	5	6	7	8	9
11	21	31	41	61	71	81	90	51	10	
$\text{ar}[8] > \text{ar}[j]$										$\uparrow \text{ar}[8] < \text{r}[4]$
11	21	31	41	51	71	81	90	61	10	
$\text{ar}[8] > \text{ar}[j]$										$\uparrow \text{ar}[8] < \text{r}[5]$
11	21	31	41	51	61	81	90	71	10	
$\text{ar}[8] > \text{ar}[j]$										$\uparrow \text{ar}[8] < \text{r}[6]$
11	21	31	41	51	61	71	90	81	10	
$\text{ar}[8] > \text{ar}[j]$										$\uparrow \text{ar}[8] < \text{r}[7]$
11	21	31	41	51	61	71	81	90	10	
$\text{ar}[8] > \text{ar}[j]$										$\uparrow \text{ar}[8] < \text{r}[8]$
11	21	31	41	51	61	71	81	90	10	
$\text{ar}[8] > \text{ar}[j]$										$\uparrow \text{ar}[8] < \text{r}[9]$

Рис.6.2. Фрагмент процесса сортировки «пузырьком»

```
[90 11 81 21 71 31 61 41 51 10]
[11 90 81 21 71 31 61 41 51 10]
[11 81 90 21 71 31 61 41 51 10]
[11 21 81 90 71 31 61 41 51 10]
[11 21 71 81 90 31 61 41 51 10]
[11 21 31 71 81 90 61 41 51 10]
[11 21 31 61 71 81 90 41 51 10]
[11 21 31 41 61 71 81 90 51 10]
[11 21 31 41 51 61 71 81 90 10]
[10 11 21 31 41 51 61 71 81 90]
[10 11 21 31 41 51 61 71 81 90]
```

Рис.6.3. Строки набора, в которых состоялся обмен элементами

Очевидно, что временная сложность этого алгоритма достаточно высока $O(n^2)$, поскольку определяется числом проверок условия $\text{ar}[i] < \text{ar}[j]$ и числом обменов $\Leftrightarrow \text{ar}[i] \quad \text{ar}[j]$. В то же время пространственная сложность «пузырькового» алгоритма составляет $O(1)$, поскольку не требует дополнительной памяти для организации вычислительного процесса. Алгоритм обладает высоким уровнем стабильности.

Благодаря своей простоте, Bubble-сортировка часто используется, например, в компьютерной графике, где он популярен за свою способность обнаруживать незначительную ошибку в почти отсортированных массивах и исправлять её с линейной сложностью $(2n)$. Однако для сортировки больших наборов данных «пузырьковый» алгоритм крайне неэффективен.

Пути доступа к программным файлам (Bubble sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6.3. Сортировка выбором (Selection Sort)

Алгоритм сортировки выбором основан на операциях сравнения, в котором набор данных делится на две части: отсортированную - в левой части и неотсортированную - в правой. Дракон-диаграмма алгоритма сортировки выбором представлена на рис. 6.4.

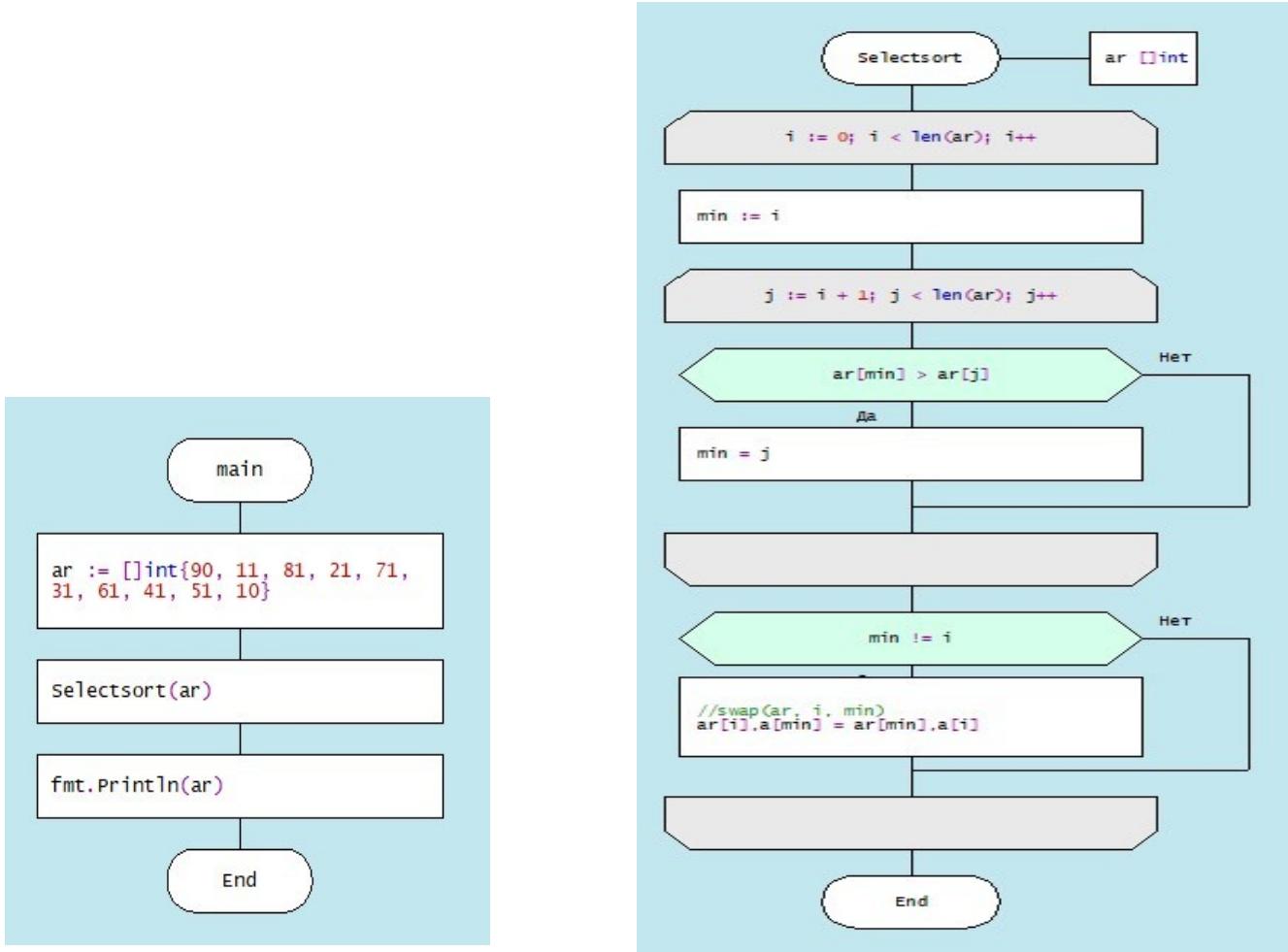


Рис. 6.4. Дракон-диаграммы алгоритма сортировки методом выбора

Сортировка выбором выполняется путём получения наименьшего значения в каждой итерации, а затем его замены на текущий индекс. Изначально отсортированная часть пуста, а несортированная часть — весь набор. Наименьший элемент выбирается из несортированного массива и заменяется самым левым элементом, и этот элемент становится частью отсортированного массива. Этот процесс продолжает перемещение неотсортированной границы массива на один элемент вправо. Например, дан набор целых чисел [90, 12, 83, 24, 75, 38, 62, 41, 59, 10]. В первой позиции, где в настоящее время хранится 90, алгоритм проходит весь набор и находит наименьшее значение — 10, после чего эти два значения меняются местами. Далее такой процесс применяется к остальным элементам в срезе (рис. 6.5.):

<u>9</u>	7	8	2	1	3	6	4	5	<u>1</u>
<u>0</u>	5	3	4	2	8	2	1	9	<u>0</u>
1	7	8	2	1	3	6	4	5	9

0	5	3	4	2	8	2	1	9	0
1	<u>1</u>	8	2	<u>7</u>	3	6	4	5	9
0	<u>2</u>	3	4	<u>5</u>	8	2	1	9	0
1	1	<u>2</u>	<u>8</u>	7	3	6	4	5	9
0	2	<u>4</u>	<u>3</u>	5	8	2	1	9	0
1	1	2	<u>3</u>	7	<u>8</u>	6	4	5	9
0	2	4	<u>8</u>	5	<u>3</u>	2	1	9	0
1	1	2	3	<u>4</u>	8	6	<u>7</u>	5	9
0	2	4	8	<u>1</u>	3	2	<u>5</u>	9	0
1	1	2	3	4	<u>5</u>	6	7	<u>8</u>	9
0	2	4	8	1	<u>9</u>	2	5	<u>3</u>	0
1	1	2	3	4	5	6	7	8	9
0	2	4	8	1	9	2	5	3	0
1	1	2	3	4	5	6	7	8	9
0	2	4	8	1	9	2	5	3	0
1	1	2	3	4	5	6	7	8	9
0	2	4	8	1	9	2	5	3	0

Рис. 6.5. Процесс выбора и замены элементов среза

Оценка сложности алгоритма сортировки выбором представлена в таблице.

Временная сложность:	
Худший случай	$O(n^2)$
Средний случай	$O(n^2)$
Лучший случай	$O(n^2)$
Пространственная сложность:	$O(1)$
Сортировка выбором - нестабильна	

Пути доступа к программным файлам (Selection Sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6.4. Сортировка методом вставки (Insertion Sort)

Сортировка вставки происходит путем многоократного извлечения элемента из несортированной части набора и последующей вставки в отсортированную часть набора до тех пор, пока не будут вставлены все элементы. Этот алгоритм обычно используется людьми при сортировке стопок бумаг. Дракон-диаграмма алгоритма сортировки методом вставки представлена на рис. 6.6.

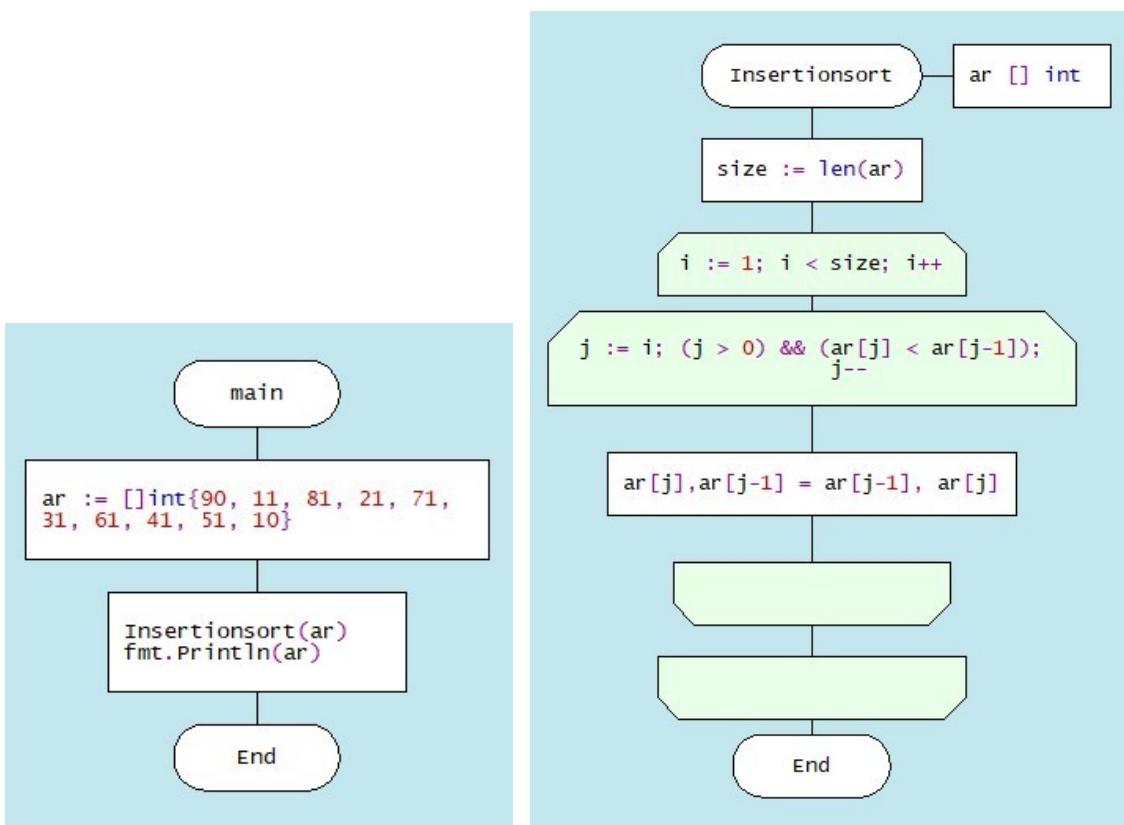


Рис. 6.6. Дракон-диаграмма алгоритма сортировки методом вставки

В данном примере элементы с максимальным значением «продвигаются» вправо, после чего выполняется цикл, в котором сравниваются соседние элементы и при необходимости меняются местами (отмечено подчеркиванием “ ”).

i = 1	i = 4	i = 8
[11 90 81 21 71 31 61 41 51 10]	[11 21 71 81 90 31 61 41 51 10]	[11 21 31 41 51 61 71 81 90 10]
[11 81 90 21 71 31 61 41 51 10]	[11 21 71 <u>81</u> <u>31</u> 90 61 41 51 10]	[11 21 31 41 51 61 71 81 10 90]
i = 2	[11 21 <u>71</u> <u>31</u> <u>81</u> 90 61 41 51 10]	[11 21 31 41 51 61 71 10 81 90]
[11 81 90 21 71 31 61 41 51 10]	[11 <u>21</u> <u>31</u> 71 81 90 61 41 51 10]	[11 21 31 41 51 61 10 71 81 90]
[11 <u>81</u> <u>21</u> 90 71 31 61 41 51 10]	i = 5	[11 21 31 41 51 10 61 71 81 90]
[11 21 <u>81</u> 90 71 31 61 41 51 10]	[11 21 31 71 81 90 61 41 51 10]	[11 21 31 41 10 51 61 71 81 90]

i = 3	i = 7	[11 21 31 10 41 51 61 71 81 90]
[11 21 81 90 71 31 61 41 51 10]	[11 21 31 41 61 71 81 90 51 10]	[11 21 10 31 41 51 61 71 81 90]
[11 21 <u>81 71</u> 90 31 61 41 51 10]	[11 21 31 41 61 71 <u>81 51</u> 90 10]	[11 10 21 31 41 51 61 71 81 90]
[11 21 <u>71 81</u> 90 31 61 41 51 10]	[11 21 31 41 61 71 <u>51 81</u> 90 10]	[10 11 21 31 41 51 61 71 81 90]
	[11 21 31 41 <u>61 51</u> 71 81 90 10]	i = 9
	[11 21 31 41 <u>51 61</u> 71 81 90 10]	[10 11 21 31 41 51 61 71 81 90]

Рис. 6.7. Вставка и замена элементов срезов

Сложность алгоритма сортировки вставкой определяется количеством сравнений и перемещений элементов, которые необходимо выполнить для упорядочивания массива. Это зависит от того, как изначально отсортирован массив.

Наихудший сценарий — когда массив сортируется в обратном порядке. В этом случае каждый элемент следует сравнить со всеми предыдущими элементами и переместить в начало массива. Количество сравнений и ходов равно

$$(n(n-1)/2,$$

что ведет к $O(n^2)$.

Средний случай: когда массив частично отсортирован. При этом каждый элемент следует сравнить в среднем с половиной предыдущих элементов и переместить на соответствующую позицию. Количество сравнений и движений равно

$$n^2/4,$$

что ведет к $O(n^2)$.

Лучше всего, когда массив уже отсортирован. При этом каждый элемент следует сравнить только с одним предыдущим элементом и оставить на своем месте. Количество сравнений и ходов равно

$$(n - 1),$$

что ведет к $O(n)$.

Оценка сложности алгоритма сортировки вставкой представлена в таблице.

Временнаá сложность:	
Худший случай	$O(n^2)$
Средний случай	$O(n^2)$
Лучший случай	$O(n)$
Пространственная сложность:	$O(1)$
Сортировка вставками - стабильна	

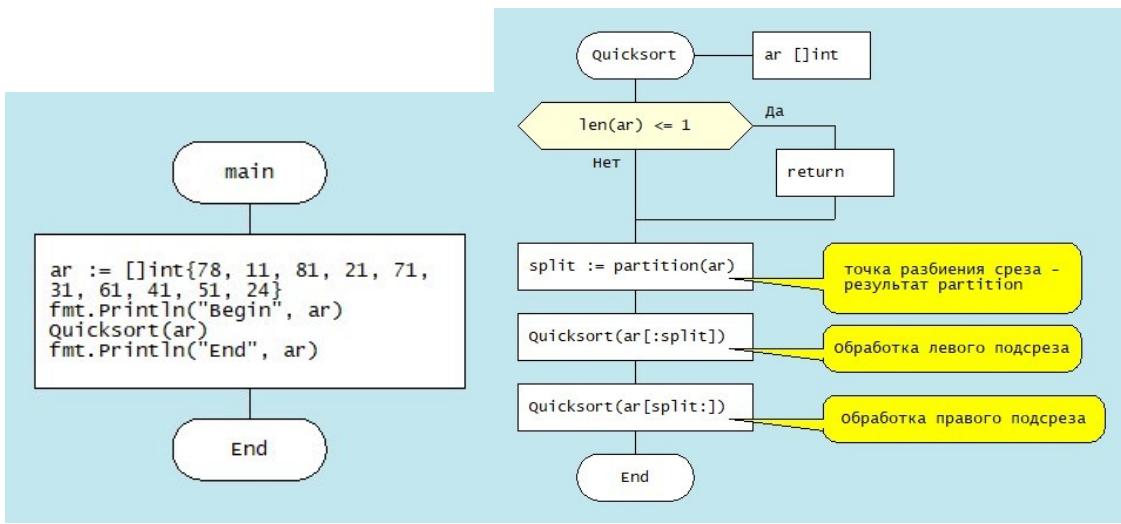
Пути доступа к программным файлам (Insetion Sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6.5. Быстрая сортировка выбором (Quick Sorting)

Быстрая сортировка является высокоэффективным алгоритмом сортировки, общая схема которого состоит из следующих этапов:

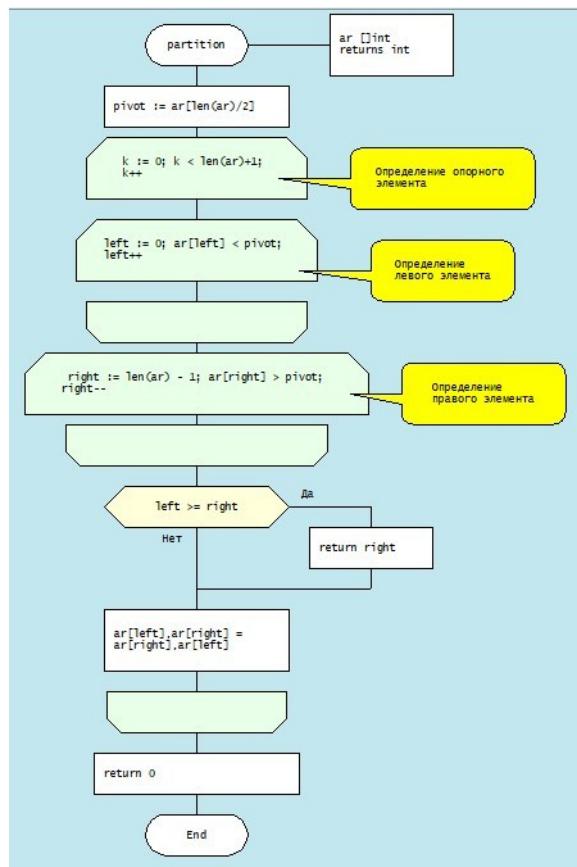
1. Выбор опорного элемента из среза.
2. Перераспределение элементов в срезе таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.
3. Рекурсивное применение первых предыдущих шагов к фрагментам среза слева и справа от опорного элемента.
4. В результате формируется полностью отсортированный массив.

Дракон-диаграммы алгоритма быстрой сортировки представлена на рис.6.6.



а) функция `main`

б) функция `Quicksort`



в) функция `Partition`

Рис. 6.8. Дракон-диаграмма алгоритма быстрой сортировки

Рассмотрим этот алгоритм подробнее. В функции `main` представлена коллекция (набор) целочисленных данных. Опорным элементом `pivot` выбран элемент среза со значением 31.

78	11	81	21	71	31	61	41	51	28
----	----	----	----	----	----	----	----	----	----

Введем два указателя: *left* и *right*. В начале алгоритма они указывают, соответственно, на левый и правый конец набора. В представленном алгоритме указатель *left* сдвигается с шагом в 1 элемент по направлению к концу среза, пока значение текущего элемента будет меньше значения опорного элемента. Индекс первого элемента, значение которого будет больше *pivot* ($ar[left] > pivot$), фиксируется в переменной *left*. Затем аналогичным образом в алгоритме указатель *right* перемещается от конца среза к началу, пока не будет найден элемент, для которого выполняется условие $ar[right] \leq pivot$. Этот фрагмент алгоритма выполняется до тех пор, пока слева не будет обнаружен элемент, значение которого превысит значение правого элемента справа. В этом случае эти элементы меняются местами. В приведенном примере первый же (левый) элемент больше опорного ($78 > 31$) и последний (правый) элемент меньше опорного ($24 < 31$). Во второй строчке таблицы они поменялись местами. Далее процесс продолжается (рис.6.7.):

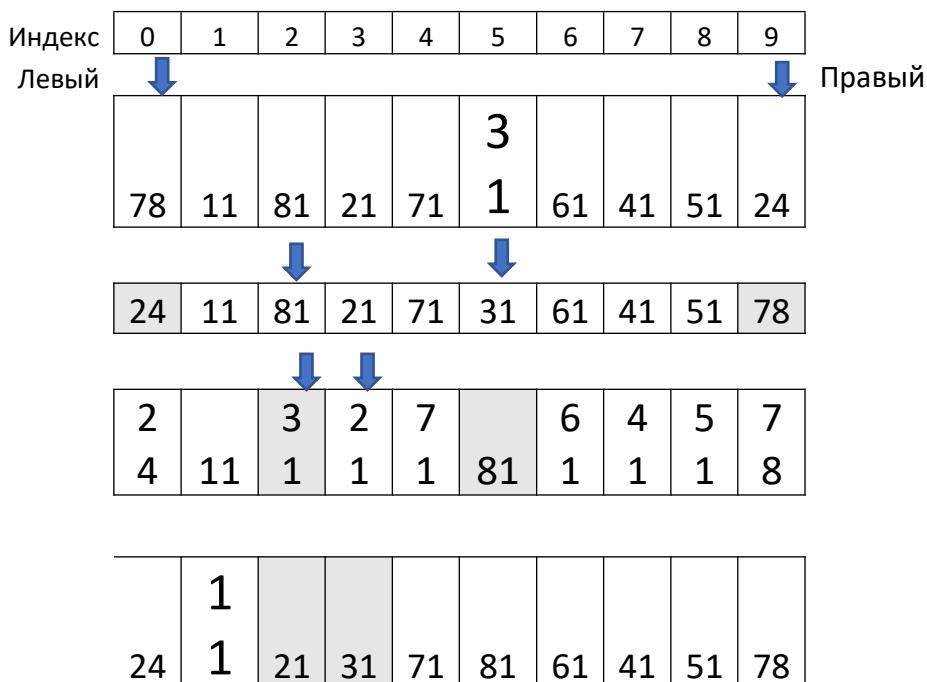
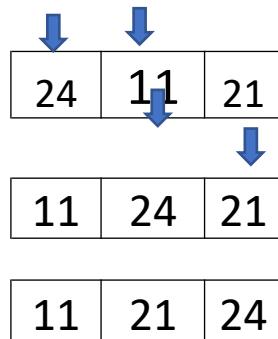


Рис. 6.9. Фрагмент выполнения алгоритма QuickSort

Затем формируется новый фрагмент исходного набора, в котором переопределется опорный элемент. В этом фрагменте $pivot = 11$.



После обработки левой части набора таким же образом обрабатывается правая часть. В конечном счете, срез становится отсортированным:

			3			6			
11	21	24	1	41	51	1	71	78	81

Этот алгоритм довольно эффективен для больших наборов данных, так как его средняя и наихудшая сложность $O(n^2)$, соответственно.

Анализ сложности:

Временная сложность:	
Худший случай	$O(n^2)$
Средний случай	$O(n \log n)$
Лучший случай	$O(n \log n)$
Пространственная сложность:	$O(n \log n)$
Быстрая сортировка - нестабильна	

Worst Case Time Complexity $O(n^2)$ Best Case Time Complexity $O(n \log n)$ Average Time Complexity $O(n \log n)$ Space Complexity $O(n \log n)$ Stable Sorting No

Пути доступа к программным файлам (Quick Sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6.6. Сортировка слиянием (Merge sort)

Сортировка слиянием выполняется с помощью рекурсивного разбиения набора (среза) на фрагменты до тех пор, пока не останется фрагмент, состоящий из двух элементов. Два элемента легко сравнить между собой и упорядочить в зависимости от требования: по возрастанию или убыванию. После разбиения следует обратное слияние, при котором в один момент времени (или за проход цикла) выбираются по одному элементу от каждого фрагмента среза с последующим их сравнением. Наименьший (или наибольший) элемент сохраняется в результирующем наборе, оставшийся элемент остается актуальным для сравнения с элементом из другого фрагмента на следующем шаге (рис.6.8.):

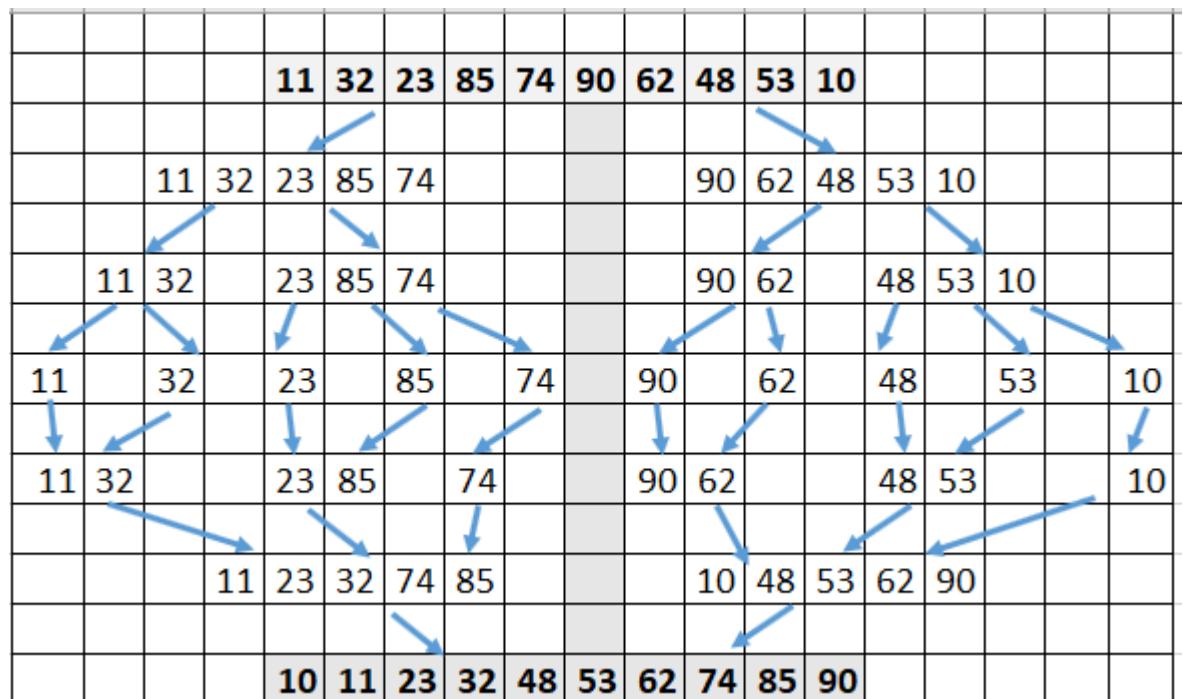
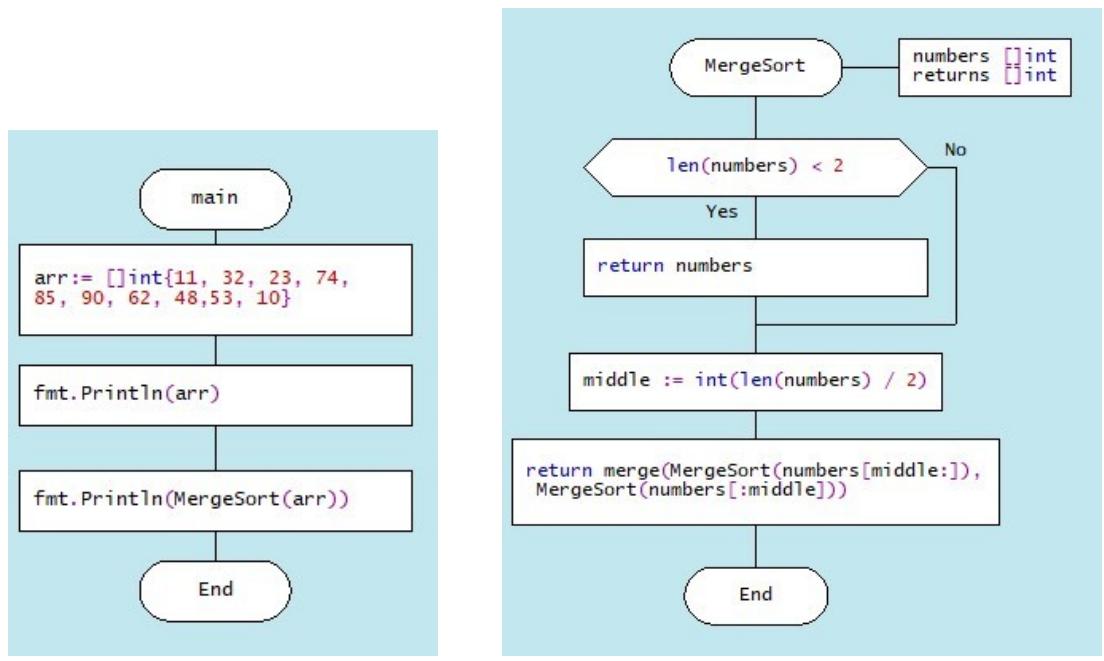


Рис. 6.8. Визуализация алгоритма сортировки слиянием

Дракон-диаграмма алгоритма сортировки слиянием представлена на рис. 6.9.



a)

б)

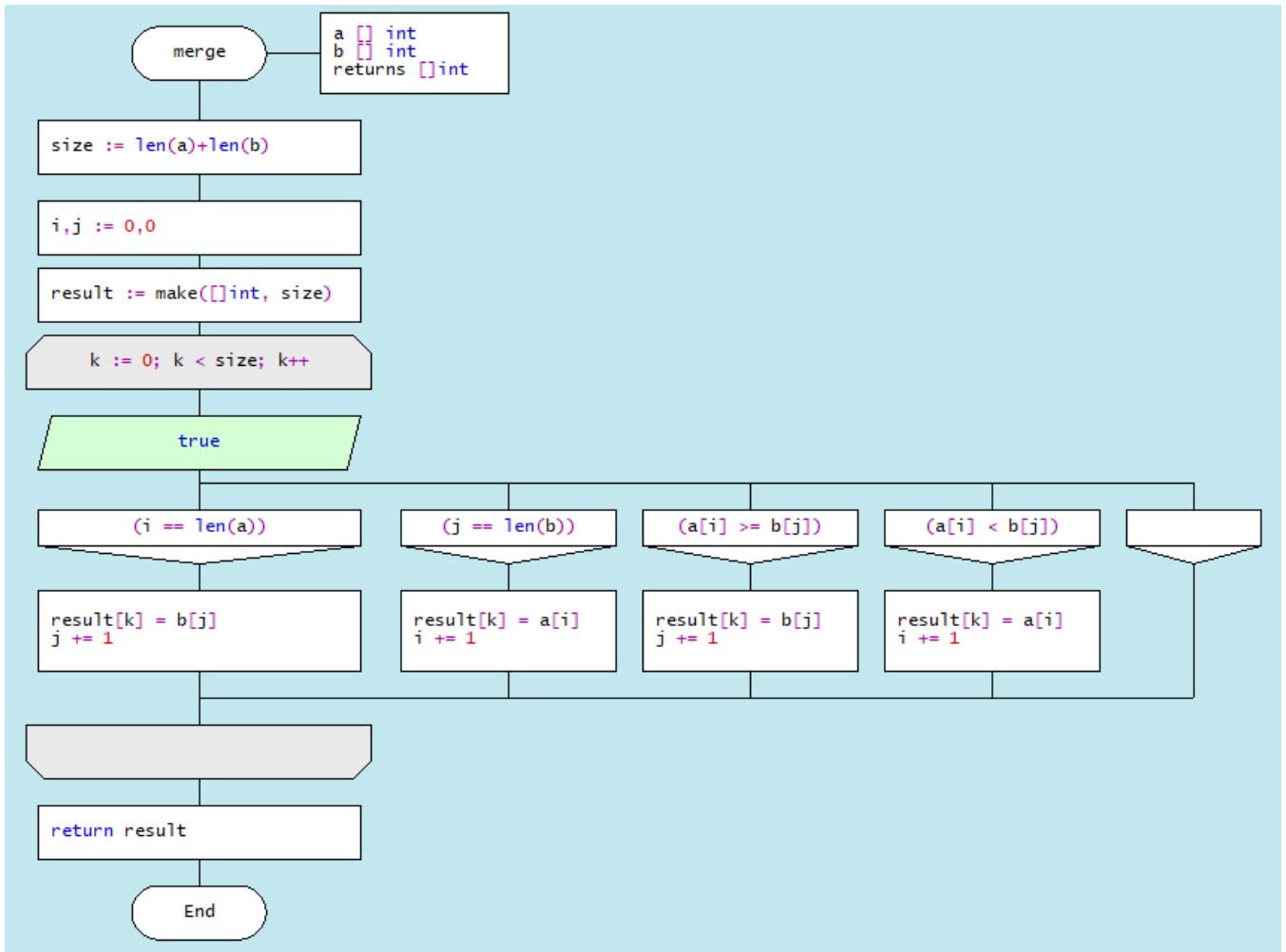


Рис. 6.9. Дракон-диаграмма алгоритма сортировки слиянием:

а) функция `main`; б) функция `MergeSort`; в) функция `merge`

Анализ сложности:

Временнаá сложность:	
Худший случай	$O(n \log n)$
Средний случай	$O(n \log n)$
Лучший случай	$O(n \log n)$
Пространственная сложность:	$O(n)$
Сортировка слиянием - стабильна	

Пути доступа к программным файлам (Merge Sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6.6. Shell сортировка

Шелл-сортировка (ShellSort) представляет собой разновидность сортировки вставками (Insertion Sort). При сортировке Шелла сначала сравниваются и сортируются между собой значения, отстоящие один от другого на некотором расстоянии d . После этого процедура повторяется для некоторых меньших значений d до тех пор пока расстояние не станет $d=1$ (то есть обычной сортировкой вставками). Дракон-диаграмма алгоритма представлена на рис. 6.10.

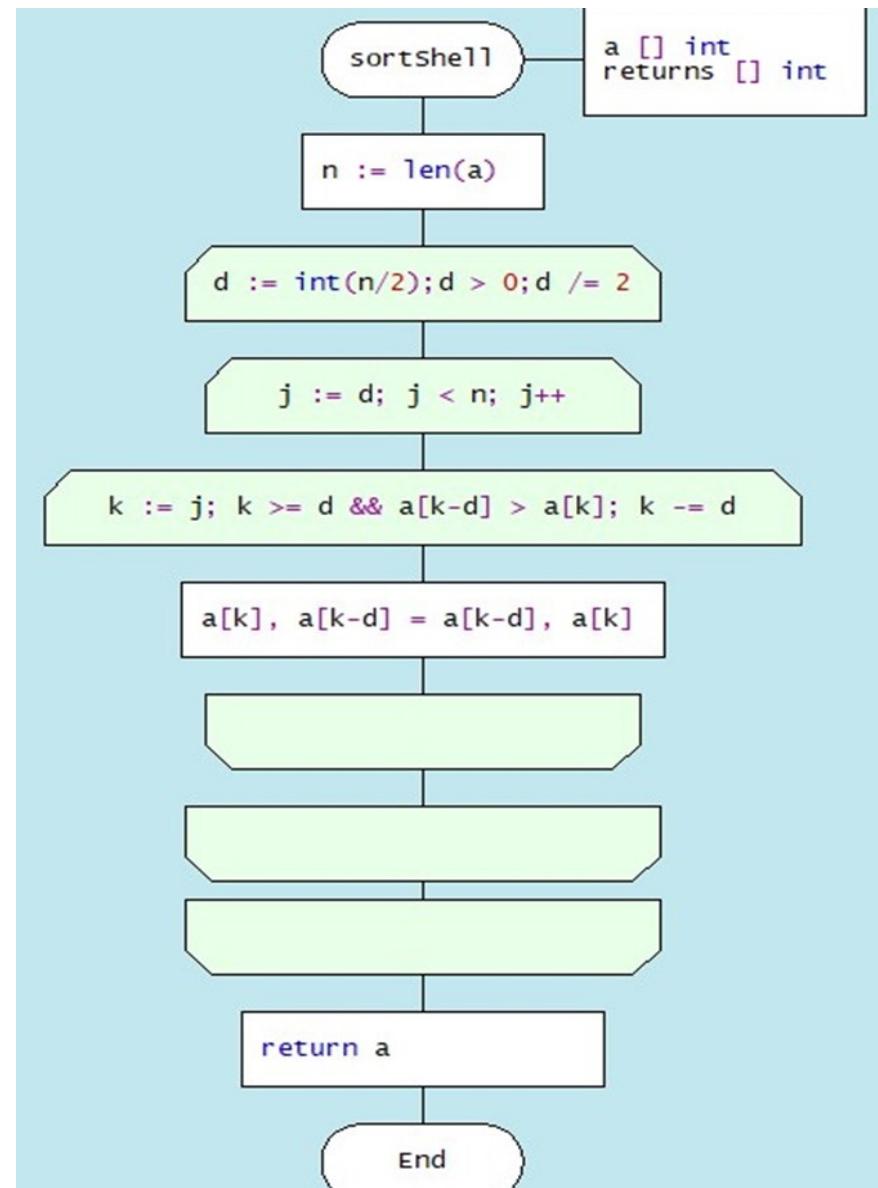
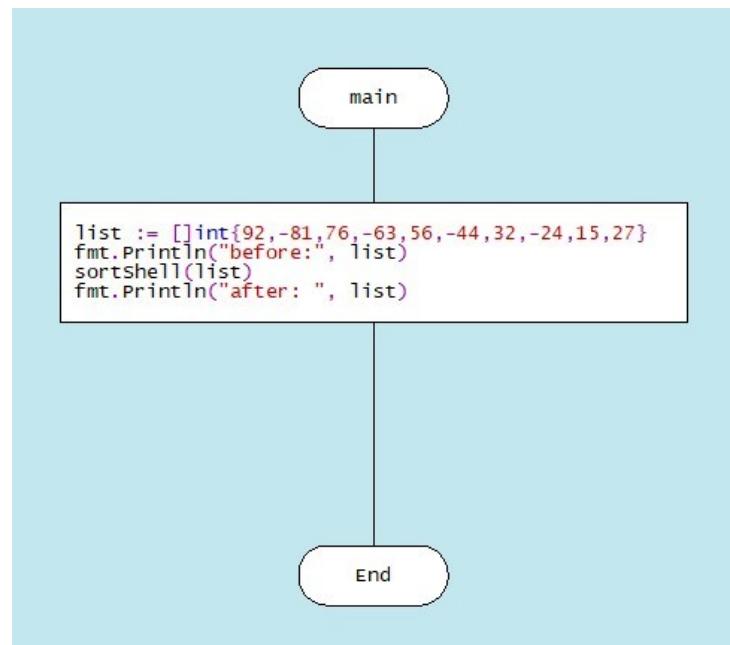


Рис. 6.10. Дракон-диаграммы алгоритма Shell-сортировки

Shellsort использует процесс, который лежит в основе многих представленных сортировок: последовательное разбиение данных на сегменты, сортировка этих сегментов и, наконец, объединение в отсортированный набор. Процесс разбиения происходит так, чтобы каждый элемент в сегменте представлял собой фиксированное количество позиций друг от друга. При этом возникает неопределенность в выборе этого числа позиций, иными словами расстояния между элементами в сегменте (d). Самый простой пример это $d = n / 2, d_2 = d/2 \dots d_n=1$.

На рис. 6.11 показан обмен элементами при условии $a[k] < a[k-d]$, где $d=5$ при первом проходе среза.

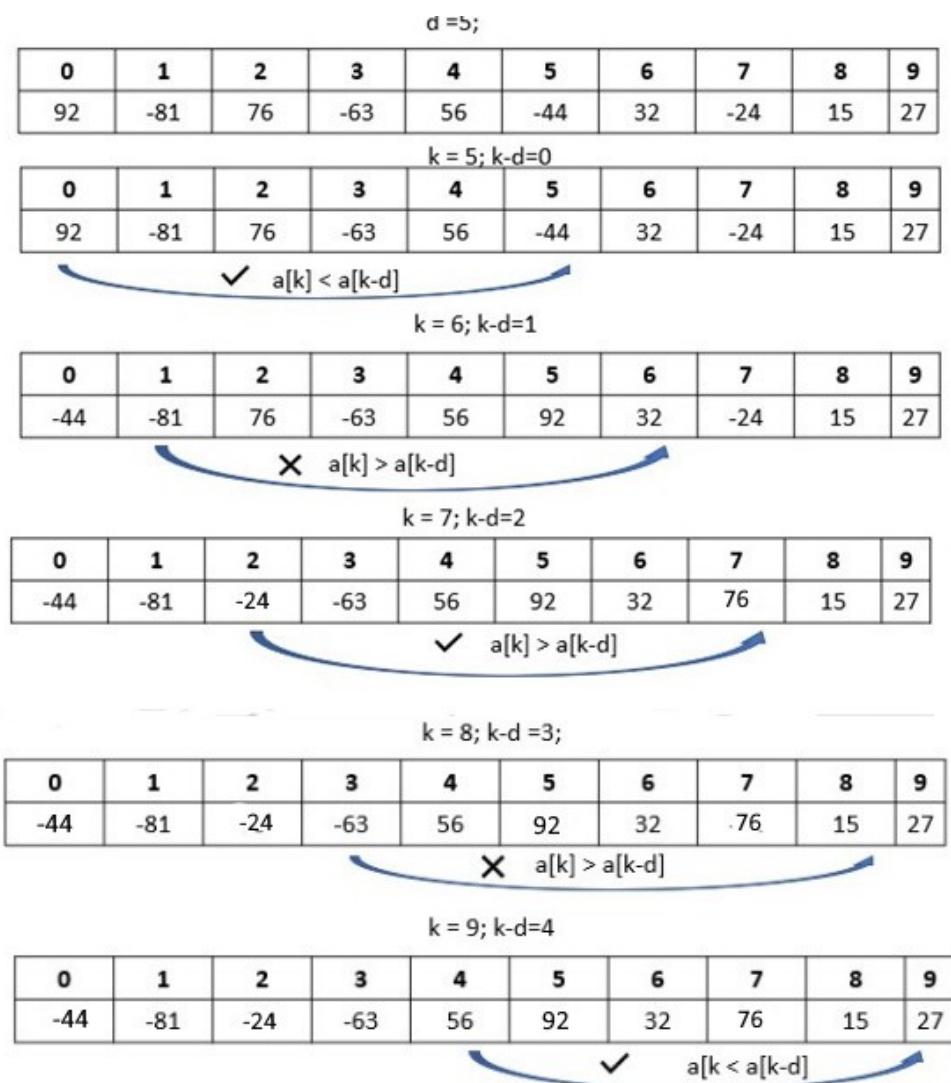


Рис. 6.11. Фрагмент алгоритма обмена элементами набора

Анализ сложности:

Временная сложность:	
Худший случай	$O(n \log n)^2$
Средний случай	$O(n \log^2 n)$
Лучший случай	$O(n)$
Пространственная сложность:	$O(1)$
Сортировка слиянием - нестабильна	

Пути доступа к программным файлам (Shell Sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6.7. Пирамидальная сортировка (Heap Sort)

Алгоритм пирамидальной сортировки можно рассматривать как улучшенную версию алгоритма сортировки выбором (Select Sort): он делит входные данные на отсортированную и несортированную области, а затем последовательно уменьшает несортированную область, извлекая самый большой элемент и перемещая его в отсортированную область. Улучшение состоит в том, что для нахождения наибольшего значения используется бинарная куча, а не алгоритм линейного поиска. Этот алгоритм выполняется с использованием понятия *кучи*, являющейся полным двоичным деревом (см. подраздел 1.3.). Все узлы кучи либо больше, чем ее дочерние элементы, либо меньше, чем ее дочерние элементы. Кучное двоичное дерево может быть двух типов: минимальная куча (MinHeap), в которой родительский узел всегда меньше дочерних узлов, и максимальная куча (MaxHeap), в которой родительский узел всегда больше или равен дочерним узлам (рис. 6.12).

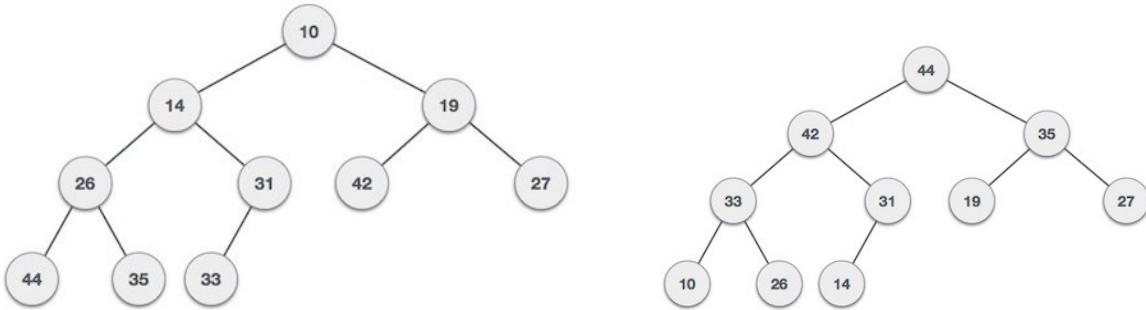


Рис. 6.12. Примеры двоичного дерева

Последовательность замены узлов дерева, начиная с корневого узла, осуществляется по формуле:

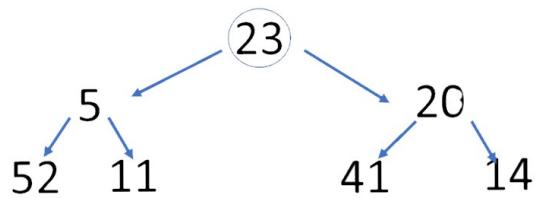


Рис. 6.13. Пример построения двоичного дерева

Покажем последовательность замены узлов дерева, начиная с корневого узла, определяемого по формуле:

$$i_n = (\text{размер массива} / 2) - 1;$$

Вначале по алгоритму меняются местами узлы (20) и (41), затем узлы (5) и (52), далее представим этот процесс в таблице:

23	5	20	52	11	41	14
23	5	41	52	11	20	14
23	5	2	41	5	11	20

	2						
52	3	41	5	11	20	14	
14	23	41	5	11	20	52	
41	23	14	5	11	20	52	
20	23	14	5	11	41	52	
	2						
23	0	14	5	11	41	52	
11	20	14	5	23	41	52	
20	11	14	5	23	41	52	
			2				
5	11	14	0	23	41	52	
11	5	14	20	23	41	52	
	1						
5	1	14	20	23	41	52	

Алгоритм сортировки кучи использует три функции: *heap_Sort*, осуществляющая перебор узлов, функция *heapify* выполняет сравнение смежных узлов, а функция *swap* меняет местами два узла. Последовательность перемещения узлов в куче показана на рис. 6.14:

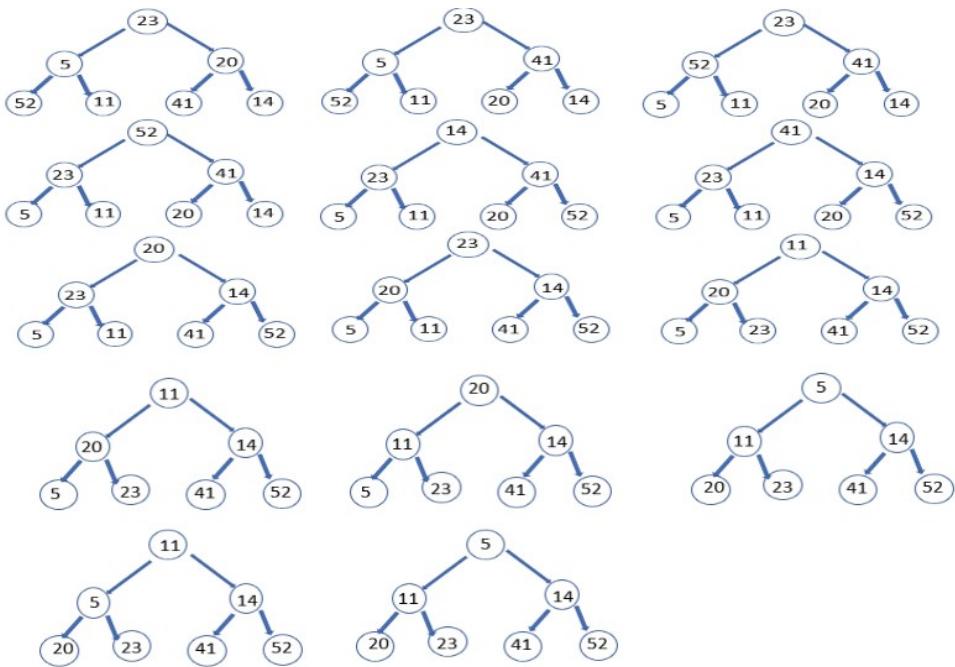


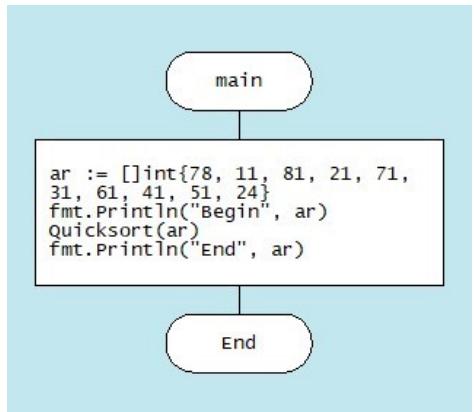
Рис. 6.14. Последовательность перемещения узлов в куче

DRAKON-диаграмма алгоритма кучи представлена на рис. 6.15:

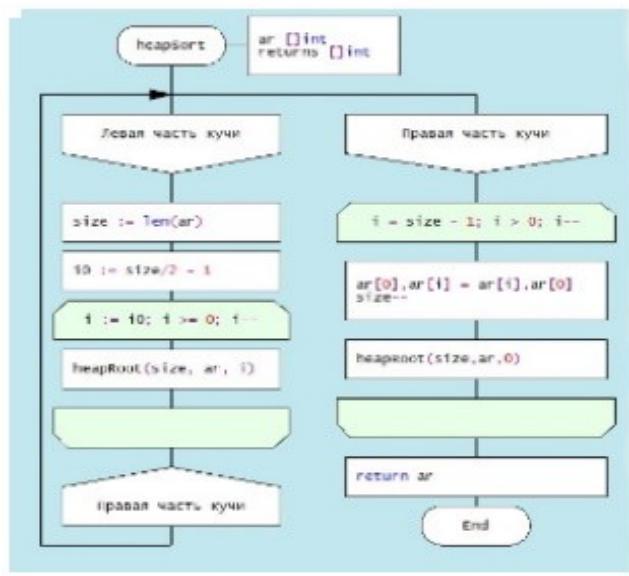
Анализ сложности:

Временная сложность:	
Худший случай	$O(n \log n)$.
Средний случай	$O(n \log n)$.
Лучший случай	$O(n \log n)$.
Пространственная сложность:	$O(1)$
Сортировка слиянием - нестабильна	

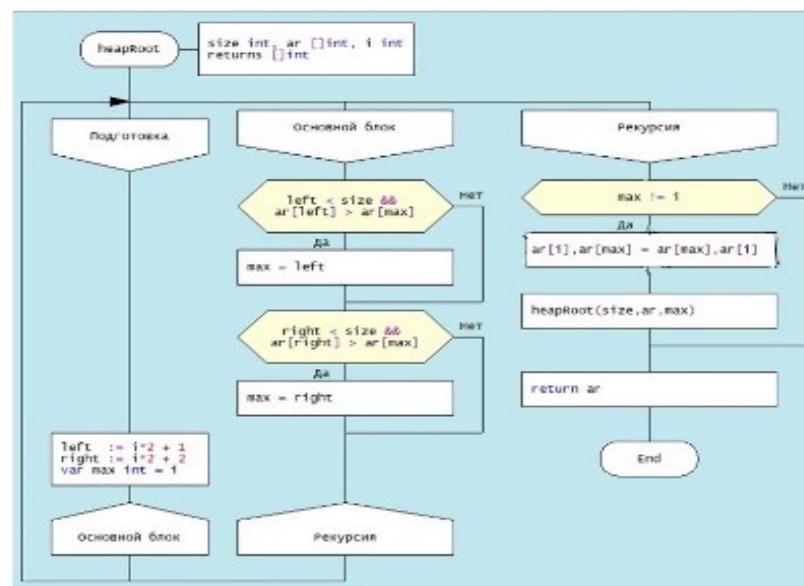
Пути доступа к программным файлам (Heap Sort)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git



а) функция main



б) функция heapsort



в) функция heapRoot

Рис. 6.15. Дракон-диаграммы алгоритма сортировки кучи

6.8. Сравнительный анализ сортировок

Выбор того или иного алгоритма сортировки определяется следующими факторами:

- Временная сложность;
- Пространственная сложность;
- Стабильность/нестабильность;

Знание сильных и слабых сторон каждого из рассмотренных алгоритмов позволяет сделать выбор в пользу того или иного вида сортировки. Каждый алгоритм уникален и лучше всего работает при определенных условиях.

6.8.1. Сравнение временной сложности

Алгоритм сортировки	Средняя	Лучшая	Худшая
<u>Bubble Sort</u>	$O(n^2)$	$O(n)$	$O(n^2)$
<u>Selection Sort</u>	$O(n^2)$	$O(n^2)$	$O(n^2)$
<u>Insertion Sort</u>	$O(n^2)$	$O(n)$	$O(n^2)$
<u>Quick Sort</u>	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n^2)$
<u>Merge Sort</u>	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$
<u>Shell Sort</u>	$n(\log(n))^2$	$O(n)$	$n (\log n)^2$
<u>Heap Sort</u>	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$

Некоторые распространенные алгоритмы сортировки являются стабильными по своей природе, такие как сортировка слияний, сортировка по количеству, сортировка вставок и сортировка пузырей. Другие, такие как Quicksort, Heapsort и Selection Sort, нестабильны. ... Например, мы можем использовать дополнительное пространство для поддержания стабильности в Quicksort.

РАЗДЕЛ 7. БАЗОВЫЕ АЛГОРИТМЫ ПОИСКА

7.1. Основные характеристики алгоритмов

Поиск – это один из важнейших алгоритмов обработки структур данных. Алгоритмы поиска предназначены для проверки наличия элемента или извлечения элемента из любой структуры данных, в которой он хранится. Более строго задачу поиска можно сформулировать следующим образом: найти один или несколько элементов в множестве, причем искомые элементы должны обладать определенным свойством. Это свойство может быть абсолютным или относительным. Относительное свойство характеризует элемент по отношению к другим элементам: например, минимальный элемент в множестве чисел.

Существует огромное количество алгоритмов поиска. Их сложность варьируется от простых алгоритмов поиска методом последовательного сравнения, в чрезвычайно эффективных, но ограниченных алгоритмов бинарного поиска. Особого внимания заслуживают алгоритмы, основанные на представлении базового набора данных в другой, более подходящей для поиска форме, и которые применяются в реально действующих приложениях обработки массивов информации в огромных базах данных.

Для решения задачи поиска также как и для задач сортировки не существует единого алгоритма, который бы лучше подходил для всех случаев. Некоторые из алгоритмов выполняются быстрее других, но для их работы требуется дополнительная оперативная память. Другие выполняются очень быстро, но их можно применять только для заранее отсортированных массивов. В то же время анализ алгоритмов поиска несколько отличается от алгоритмов сортировки. В частности, для них не существует проблемы устойчивости. При этом могут возникать ситуации, требующие введения новых критериев сложности и ее оценка.

В принципе, алгоритмы поиска отличаются между собой именно методами перебора и стратегии поиска. Исходя из типа поисковой операции, эти алгоритмы обычно классифицируются в двух категориях:

Последовательный поиск: при этом список или массив выполняется последовательно, и каждый элемент проверяется.

Интервальный поиск: Эти алгоритмы специально разработаны для поиска в отсортированных структурах данных. Эти типы поисковых алгоритмов намного эффективнее, чем линейный поиск, поскольку они многократно нацелены на центр структуры поиска и делят пространство поиска пополам.

В данном разделе рассмотрим основные алгоритмы поиска в структурах данных. Для более глубокого понимания практического использования алгоритмов в разделе приводятся их дракон-диаграммы и оценка временной и пространственной сложности.

7.2. Линейный поиск данных

а). Линейный поиск неотсортированных данных

В случае линейного поиска элемента в неупорядоченном наборе данных, например, в массиве или в срезе простейшим алгоритмом является просмотр всех элементов до тех пор пока не будет найдено искомое значение (рис.7.1).

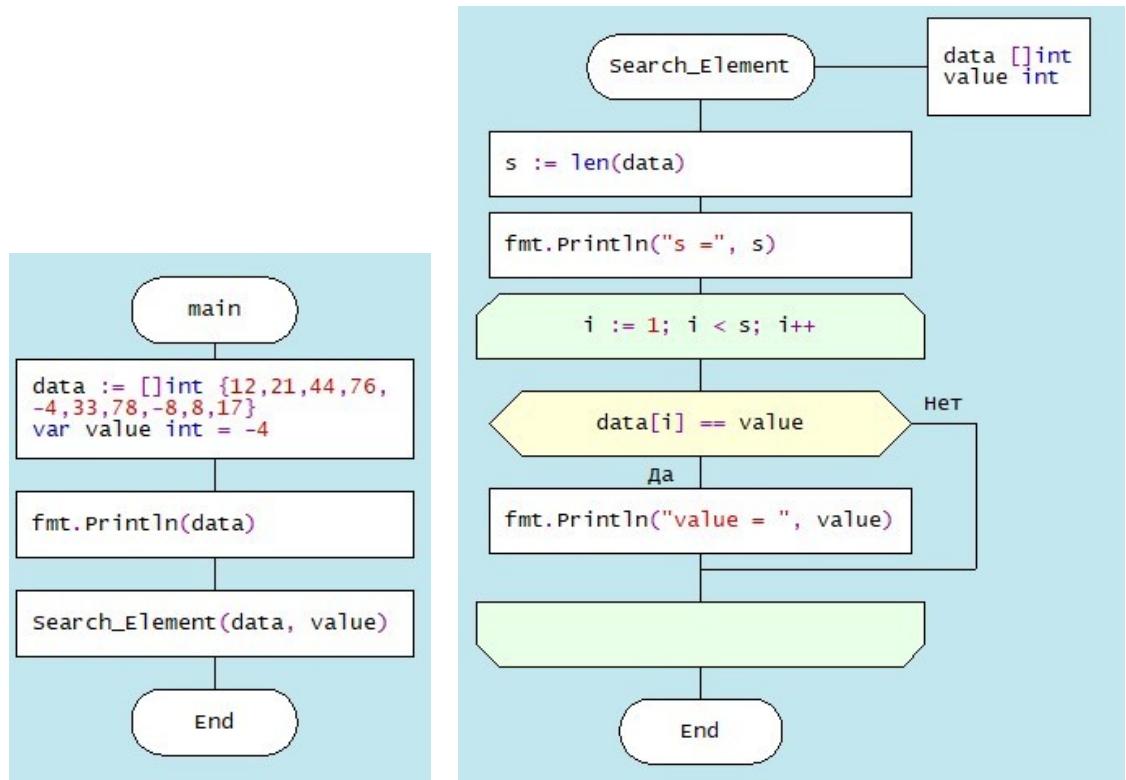


Рис. 7.1 Дракон-диаграмма алгоритма линейного поиска

Этот алгоритм не очень эффективен, однако он работает на произвольных коллекциях.

Временная сложность: $O(n)$. В худшем случае (нужный элемент находится в последней позиции) для поиска элемента нужно пройти все элементы среза. Здесь « n » — размер среза. дополнительная память. В принципе возможен и другой наихудший случай – отсутствие нужного элемента.

Пространственная сложность: $O(1)$. Для размещения среза дополнительной памяти не требуется.

Пути доступа к программным файлам (Search Element)

Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

6). Линейный поиск отсортированных данных в срезе

Если элементы набора данных отсортированы по возрастанию, либо по убыванию, то поиск нужного элемента будет намного эффективнее, чем в неупорядоченном линейном поиске. Поскольку во многих случаях не нужно проходить весь список. Например, когда в результате прохода по возрастающему отсортированному списку обнаруживается элемент с большим значением, дальнейший поиск прекращается. Такой подход экономит время и повышает производительность. На рис. 7.2. показана дракон-диаграмма такого алгоритма.

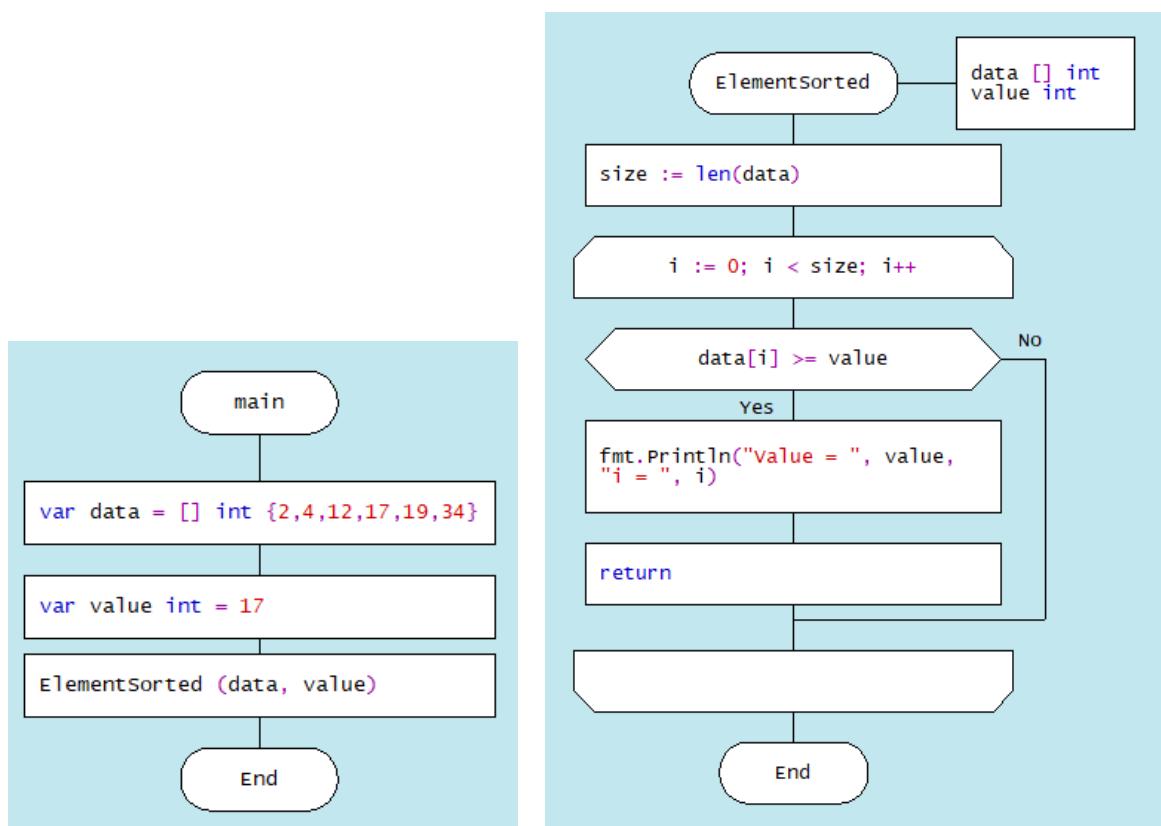


Рис. 7.2. Дракон-диаграмма алгоритма поиска данного отсортированного среза

Пути доступа к программным файлам (Data Sorted)

Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

7.3. Двоичный поиск данных в отсортированном срезе

Алгоритм двоичного поиска реализуется следующим образом:

1. Определяется значение элемента в середине структуры данных.
Полученное значение сравнивается с искомым значением.
2. Если искомое значение меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.
3. Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4. Процесс продолжается до тех пор, пока не будет найден элемент с искомым значением или не станет пустым интервал для поиска.

Дракон-диаграмма алгоритма бинарного поиска представлена на рис.

7.3. (модуль main() аналогичен предыдущему алгоритму):

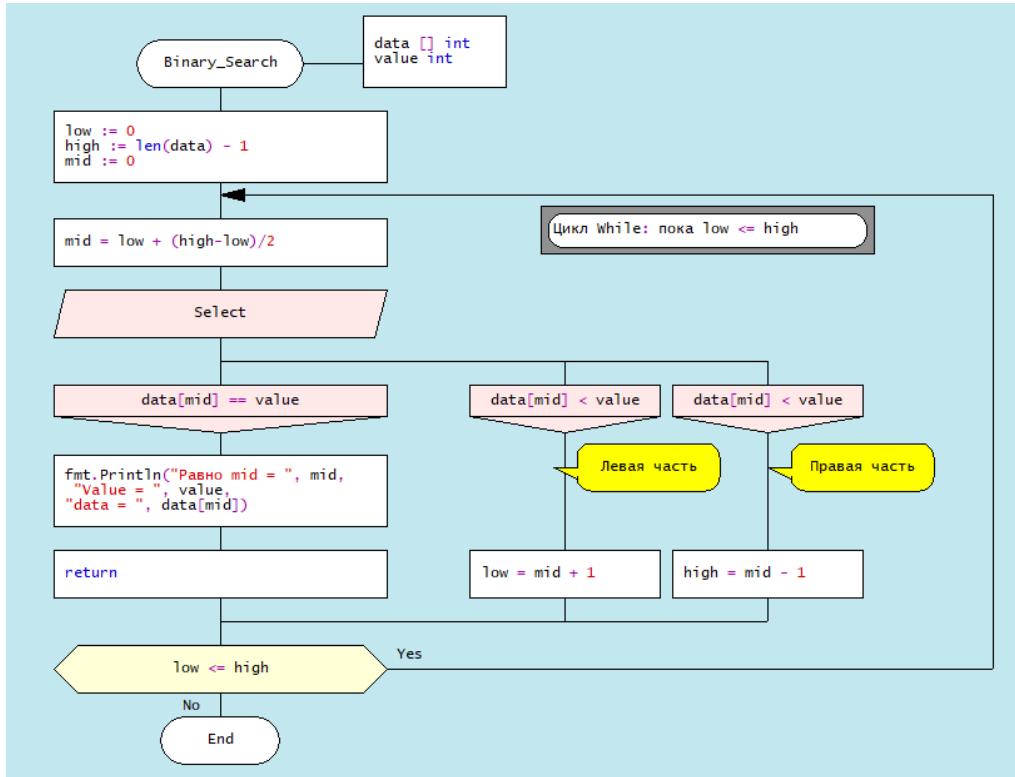


Рис.7.3. Дракон-диаграмма алгоритма бинарного поиска

Временная сложность алгоритма бинарного поиска принадлежит классу O(log n). Способ, которым следует это интерпретировать, заключается в том, что асимптотический рост времени, затрачиваемого функцией на выполнение, данного входного набора размера n, не будет превышать log n.

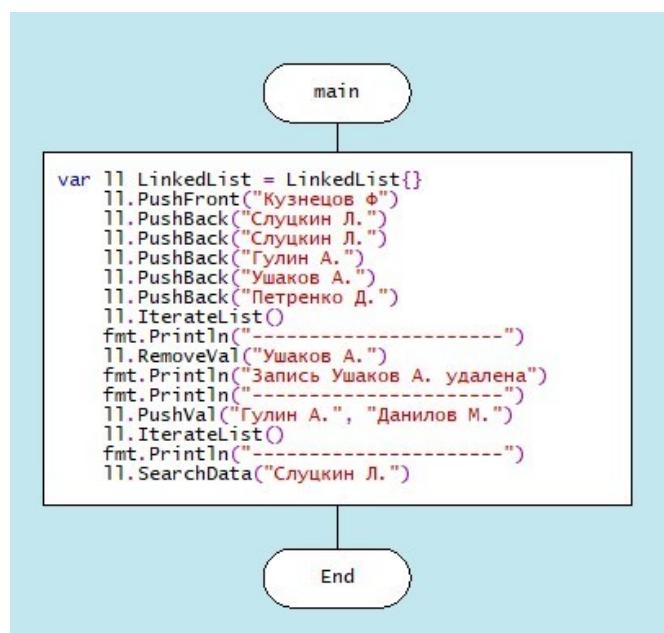
Пространственная сложность: O(1). То есть дополнительного пространства не требуется.

Пути доступа к программным файлам (Binary Sorted)	
Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

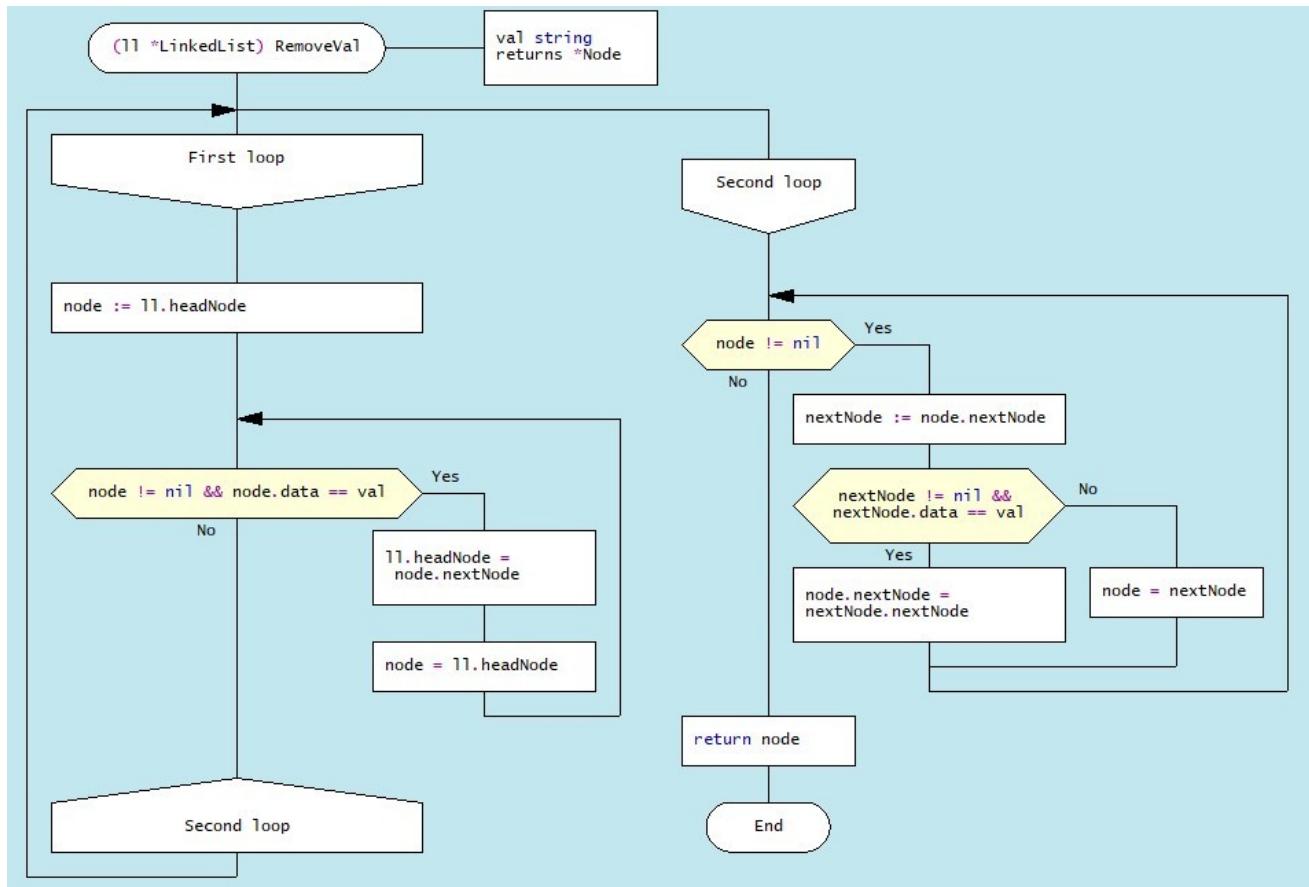
7.4. Поиск данных в односвязном списке

В отношении односвязного списка возможны три варианта. Во-первых, искомое значение в списке отсутствует, во-вторых, искомое значение встречается один раз и, в-третьих, искомое значение встречается неоднократно. Можно также поставить задачу удаления дубликатов, то есть лишних узлов с повторным значением.

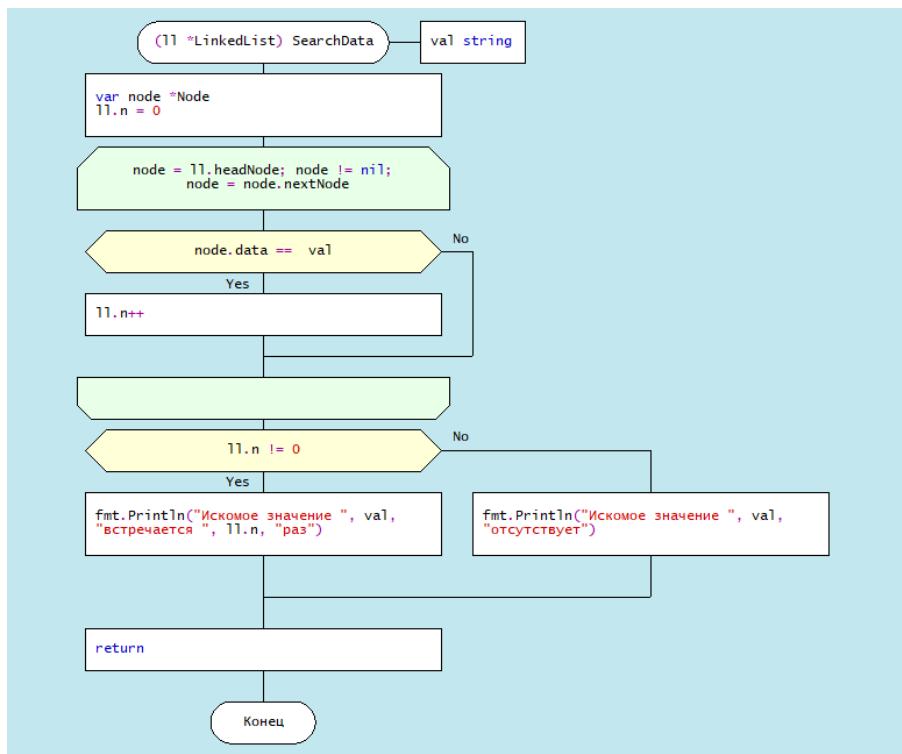
Для решения этих задач необходимо создать односвязный список, элементы которого содержат значения «Кузнецов Ф.», «Слуцкин Л.», «Гулин А.», «Ушаков А.», «Петренко Д.». В этом списке нужно удалить запись «Ушаков А.», после чего включить новую запись «Данилов М.», разместив ее после записи «Гулин А.». После этого нужно удалить дубликаты записи «Слуцкин Л.», оставив только одну. Соответствующие дракон-диаграммы представлены на рис. 7.4 а,б,в,г,д.:



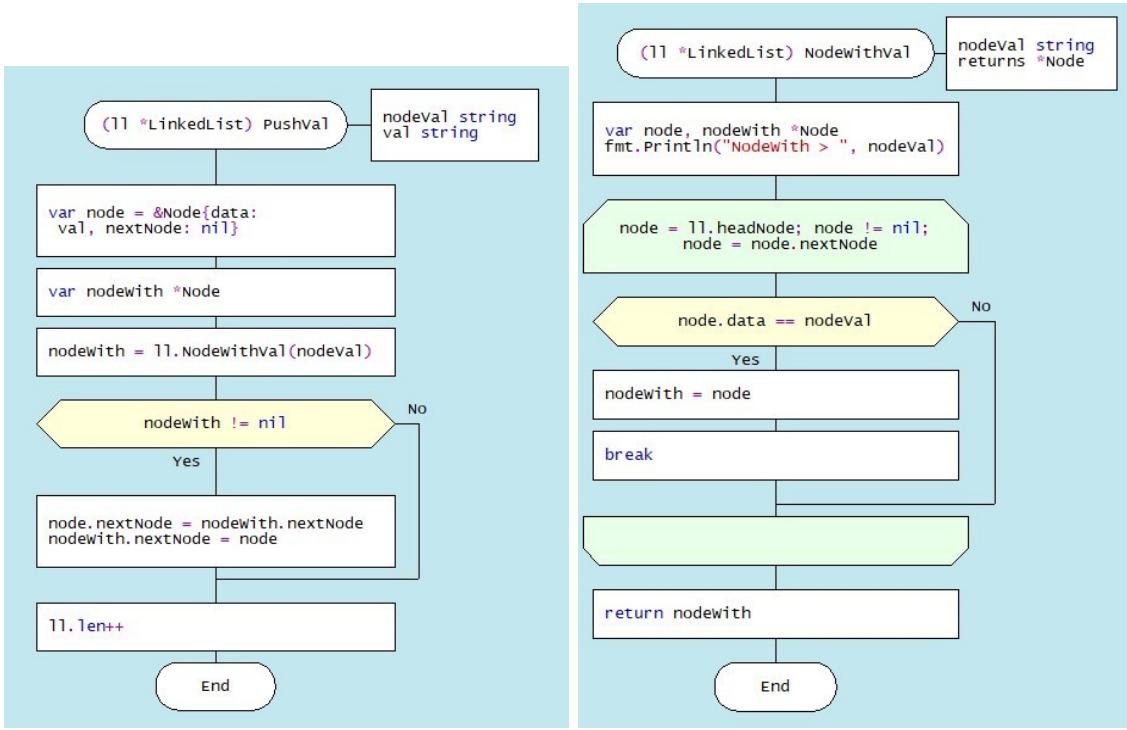
а) функция `main()`



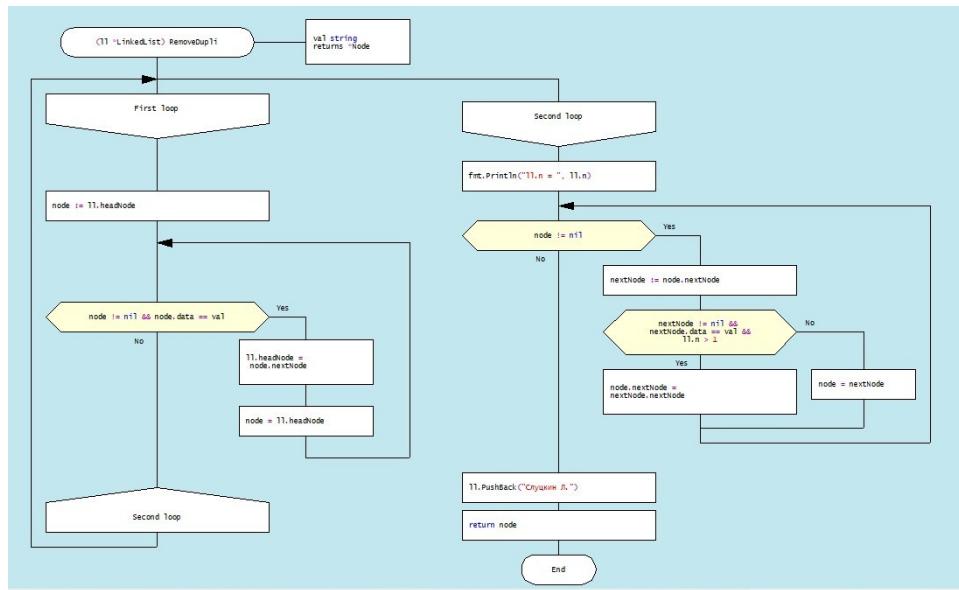
б) функция удаления RemoveVal



в) функция поиска значения searchData



г) функции вставки по значению PushVal + NodeWithVal



д). функция удаления дубликатов RemoveDupli

Рис. 7.4. Дракон-диаграммы алгоритмов удаления, поиска, вставки по значению и удаление дубликатов

Пути доступа к программным файлам (LinkedList Search)

Сгенерированный код

<https://github.com/ISA-victory/dsa-dg.git>

7.5. Хеширование

Время поиска элемента в наборе данных зависит от количества сравнений значений элементов. Для сокращения времени поиска, а, следовательно, повышения эффективности вычислительного процесса, необходимо уменьшить число сопоставлений. Этого можно достичь, преобразовав набор данных большего диапазона в набор меньшего диапазона, получившего название хэширования, результатом которого является хэш-таблицы. .

С точки зрения теории абстрактных типов данных (АДТ) хэш-таблица представляет собой структуру данных, реализующую интерфейс ассоциативного массива, которая позволяет хранить пары ключ-значение и выполнять три основных операции: операцию *добавления* новой пары, операцию *поиска* и операцию *удаления* пары ключ-значение.

С позиций программирования хэш-таблица представляет собой коллекцию элементов, содержащих пары ключ-значение, где ключ вычисляется специальной функцией, получившей название *хэш-функции*. Хэш-таблица, в свою очередь, состоит из бакетов (bucket), наборов элементов с совпадающими или близкими значениями хэш-функции. Существуют различные методы построения хэш-функции, простейшим из которых является *метод остатков*, где хэш-функция определяется как остаток от деления двух чисел (x, m), где x – элемент набора, m - число бакетов. В языке golang хэш-функция для этого метода имеет вид: $h = x \% m$.

Разберем более подробно процесс создания хэш-таблицы с помощью инструментария языка Go в редакторе DRAKON WEB Editor . Вначале создается переменная типа *Node*, определяемая как структура, состоящая из двух полей: значение элемента - *Value int* и адрес следующего элемента - *Next *Node*. Фактически это односвязный список (см. Раздел 1)

```
type Node struct {
```

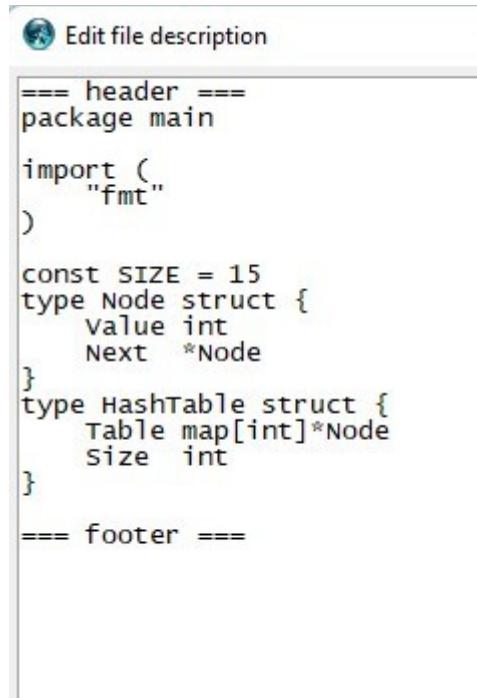
```
Value int
Next *Node

}
```

Затем создается хэш-таблица через структуру, имеющую два поля: первое поле (*Table*) представляет собой карту *map*, которая соотносит целое число (хэш-индекс) со связанным списком (**Node*), а второе – размер хэш-таблицы – *Size int*:

```
type HashTable struct {
    Table map[int]*Node
    Size int
}
```

В результате, эта хэш-таблица должна будет иметь столько связанных списков (бакетов), сколько было задано константой *Size*. В приведенном выше случае число слотов равно 15. Заметим, что объявление типов *Node* и *HashTable*, а также константа *Size* вынесено в опцию File/File description (рис.6.5.).



The screenshot shows a file editor window titled "Edit file description". The code is written in Go and defines two structures: *Node* and *HashTable*, along with a constant *SIZE*.

```
==== header ====
package main

import (
    "fmt"
)

const SIZE = 15
type Node struct {
    Value int
    Next  *Node
}
type HashTable struct {
    Table map[int]*Node
    Size   int
}
==== footer ====
```

Рис. 6.5. Объявление типов *Node*, *HashTable* и константы *SIZE*

В качестве примера рассмотрим построение хэш-таблицы размером $m = 15$ для коллекции целых чисел от 0 до 120. Первоначально слоты хэш-таблицы являются пустыми (рис. 6.5.)

0	1	2	3	4	5	6	7	8	9	0	11	12	13	14
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Рис. 6.5. Пустая хэш-таблица из 15 слотов

Хэш-функция, отражающая связь между элементом и слотом, должна принимать любой элемент из набора данных (0 … 120) и возвращать целое число из диапазона номеров слотов (от 0 до 14). Алгоритм, реализующий метод остатков, просто берёт по очереди элемент из исходного набора и делит его на 15, возвращая остаток в качестве хэш-значения, который заносится в слот $h(item)=item \% 15$. Например, хэш-код для элемента 119 определяется как $119 \% 15 = (119 - 15*7) = 14$, значение 119 заносится в соответствующий слот (рис. 6.6.):

0	1	2	3	4	5	6	7	8	9	0	11	12	13	14
-	-	-	-	-	-	-	-	-	-	-	-	-	-	11 9

Далее алгоритм определяет слоты для других элементов, постепенно заполняя их. Когда же алгоритм встретит в цикле элемент 104, тогда остаток от деления $104 \% 15 = 14$, следовательно этот элемент будет занесен в также в 14-ый слот. Таким образом в каждом слоте будут накапливаться соответствующие элементы, имеющие один индекс-хэш. Например для индекс-хэша, равного 8, слот будет состоять из таких элементов: 113 : 98 : 83 : 68 : 53 : 38 : 23 : 8. А вся хэш-таблица будет иметь следующий вид (табл. 6.1.):

Таблица 6.1. Хэш-таблица из 15 слотов

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	10	10	10	10	11	11	11	11	11	115	116	11	118	11
5	6	7	8	9	0	1	2	3	4	100	101	7	103	9
10	91	92	10	94	95	96	97	98	99	85	86	10	88	10
3	76	77	3	79	80	81	82	83	84	70	71	2	73	4
88	61	62	78	64	65	66	67	68	69	55	56	87	58	89

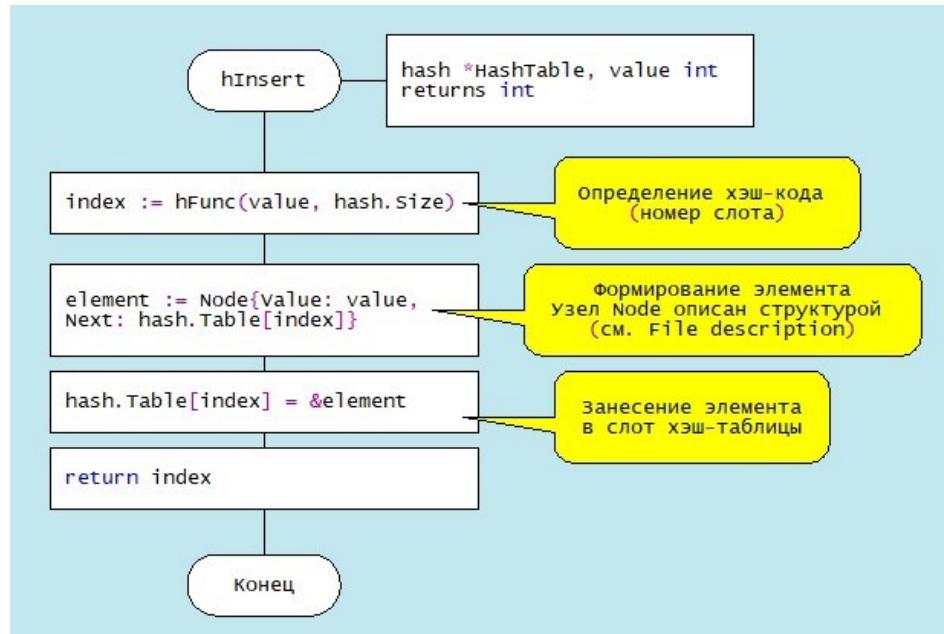
73			63	48	49	50	51	52	53	54			72		74
58	46	47	33	34	35	36	37	38	39	40	41	42	43	57	59
43	31	32	18	19	20	21	22	23	24	25	26	27	28	42	44
28	16	17	3	4	5	6	7	8	9	10	11	12	13	27	29
13	1	2												13	14

Таким образом, с помощью метода остатков коллекция из 120 целых чисел преобразуется в хэш-таблицу, состоящую из 15 слотов. Теперь поиск того или иного элемента значительно ускоряется, поскольку происходит в два шага: вначале по хэш-функции $h = (x \% m)$ вычисляется хэш-индекс, после чего поиск выполняется в слоте из 7 элементов. Алгоритм, основанный на методе остатков представлен следующими дракон-диаграммами (рис. 6.6.). Здесь *hinsert* – модуль заполнения элементами слотов по хэш-функции, *hLookup* – модуль поиска элемента в хэш-таблице, *hTravers* – модуль прохода по хэш-таблице.

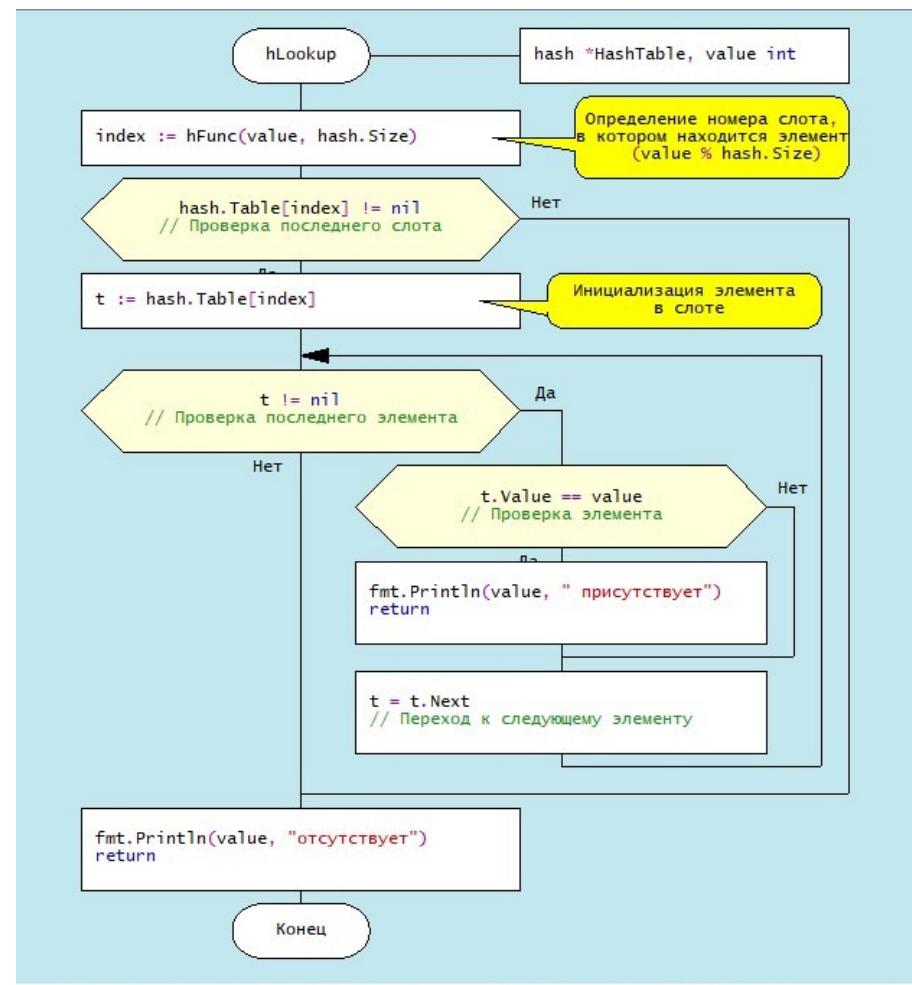
Реализация основных функций хэш-таблицы с использованием хэш-функции выглядит следующим образом:

1. Создать структуру *HashTable* список размера m для хранения объектов.
2. Вычислить хэш-код объекта, передав его через хэш-функцию.
3. Получить хэш-индексы бакета, где будут сохранены объекты.
4. Сохранить эти объекты в назначенному бакете.

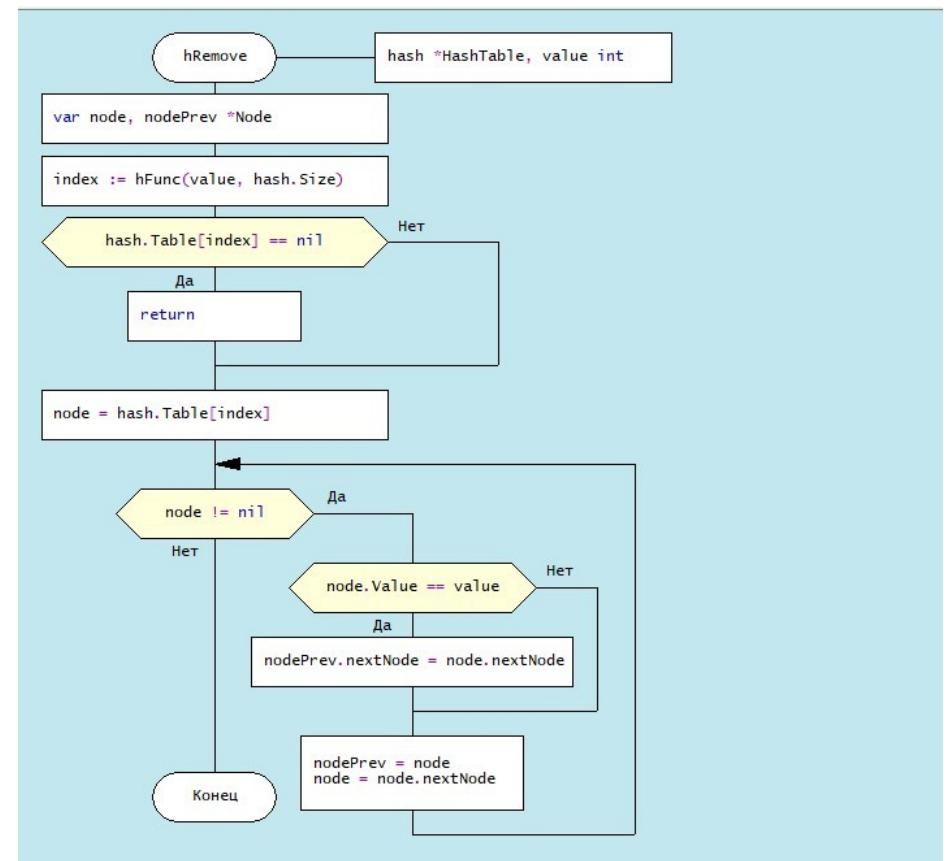
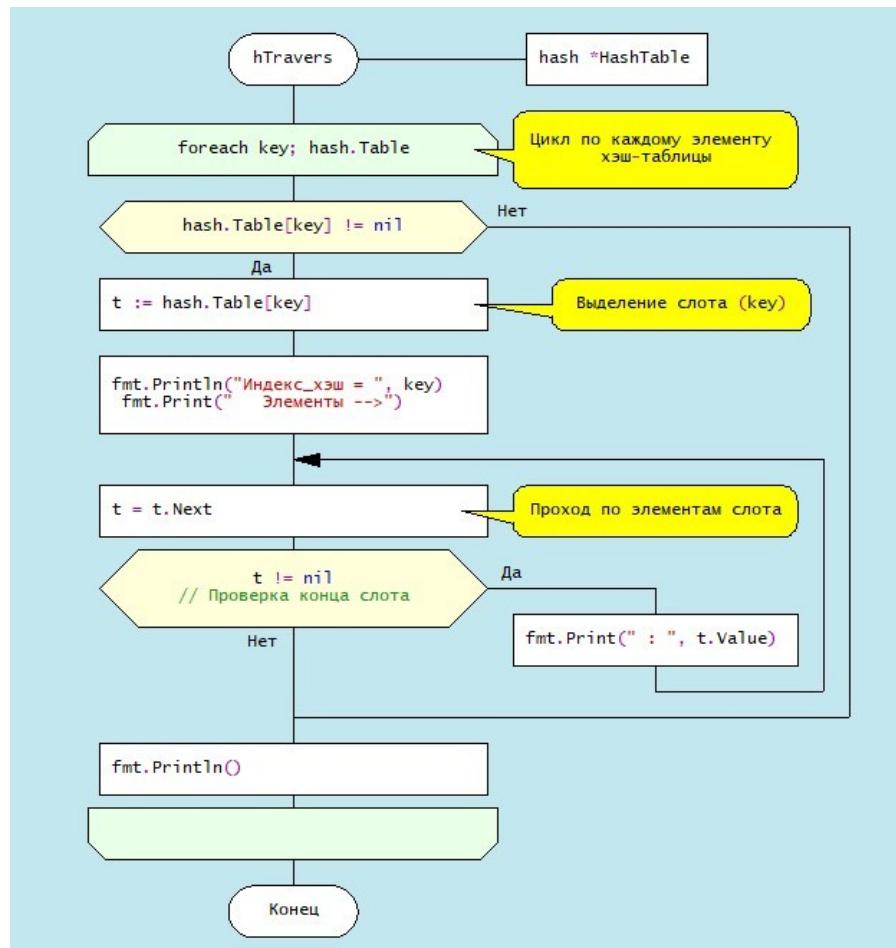
Дракон-диаграммы алгоритмов реализации основных функций работы с хэш-таблицами представлены на рис. 6.6.



а) Модуль формирования хэш-таблицы



б) Модуль поиска элемента



в) Модуль прохода по хэш-таблице

г) Модуль удаления элемента

Рис. 6.6. DRAKON-диаграммы алгоритма работы с хэш-таблицами

Рассмотрим подробнее алгоритм удаления элемента из хэш-таблицы. Предположим необходимо удалить элемент 74. Вначале определяется бакет, в котором находится удаляемый элемент. Затем осуществляется проход по элементам этого бакета, где после каждой проверки условия `node.Value == value` текущий элемент сохраняется в переменной `nodePrev`. При выполнении вышеуказанного условия поле `nodePrev.nextNode` (`0xc0000386d0`) меняется на (`0xc0000384f0`), то есть пропускается удаленный элемент (рис. 6.7.):

<code>nodePrev</code>	<code>node</code>	<code>node.nextNode</code>
89 0xc0000386d0	74 0xc0000385e0	59 0xc0000384f0
<code>nodePrev.nextNode = node.nextNode</code>		
89 0xc0000384f0	xxxx	59 0xc000038400

Рис. 6.7. Удаление элемента из хэш-таблицы

Другим примером «хорошей» хэш-функции является для использования с целочисленными значениями ключей является метод среднего квадрата. Метод среднего квадрата возводит в квадрат значение ключа, а затем извлекает средние цифры результата, давая значение в диапазоне от 0 до M. Программная реализация алгоритма построения хэш-функции на языке Golang сводится к применению встроенных функций конвертации целых чисел в строковые (`strconv.Atoi(i)`) и наоборот (`strconv.Itoa(i)`). Например, для любого четырехзначного числа хэш-функция выглядит так:

```
func hFunc(i,) int {
    var j int
    var s string
    i = i*i
    s = strconv.Itoa(i)
    s = s[3:5]
    j_ = strconv.Atoi(s)
    return j
}
```

Более реальным случаем построения хэш-таблиц является так называемый эффект коллизий, который заключается в том, что в один и тот же бакет могут попасть одинаковые или близкие по значению объекты. В большинстве задач два и более ключей хешируются (то есть преобразуются в структуру меньшего диапазона) одинаково, но они не могут занимать в хеш-таблице одну и ту же ячейку в бакете. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш-таблицы отдельный список, в который помещаются все ключи, отображающиеся в этот индекс. Эти варианты и представляют собой две классические схемы хеширования:

- хеширование методом открытой адресации с линейным опробованием;
- хеширование методом цепочек (со списками), или так называемое, многомерное хеширование.

Однако эта тема находится вне задач настоящего пособия.

РАЗДЕЛ 8. АЛГОРИТМЫ БИНАРНЫХ ДЕРЕВЬЕВ

8.1. Представление бинарных деревьев

Двоичные деревья как абстрактный тип данных были рассмотрены в первом разделе, здесь речь пойдет о компьютерной реализации этого вида структур данных с помощью визуального алгоритмического языка ДРАКОН и языка программирования Golang. Напомним основную терминологию в отношении двоичных деревьев:

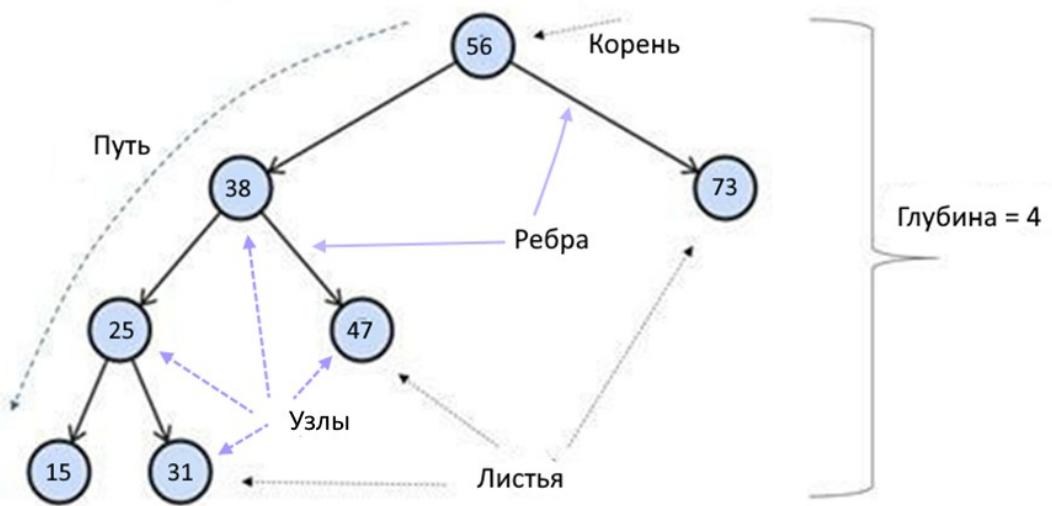


Рис. 8.1. Основная терминология дерева

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

Информационная структура двоичного дерева организуется следующим образом (рис.8.2):

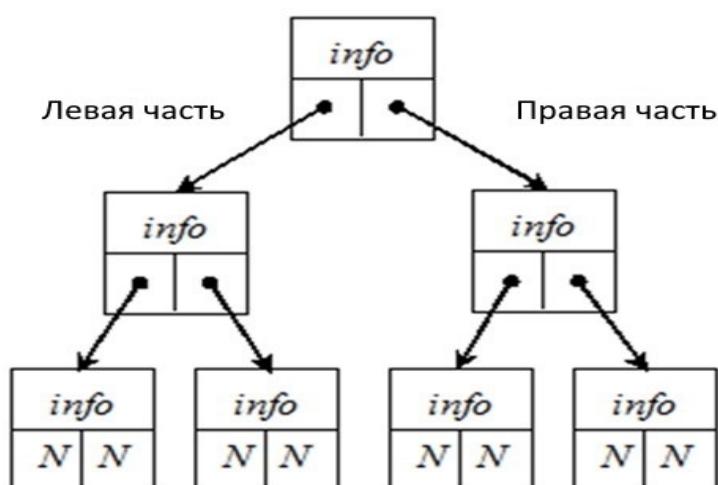


Рис.8.2. Структура бинарного дерева (*info* – значение (ключ), (N – NULL))

В учебной литературе рассматриваются различные виды бинарных деревьев, среди которых наиболее значимой является классификация на основе значений узлов:

- **двоичное дерево поиска;**
- дерево АВЛ (AVL);
- красное черное дерево.

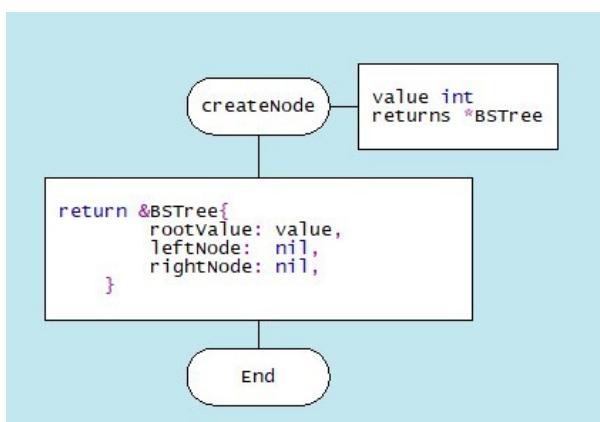
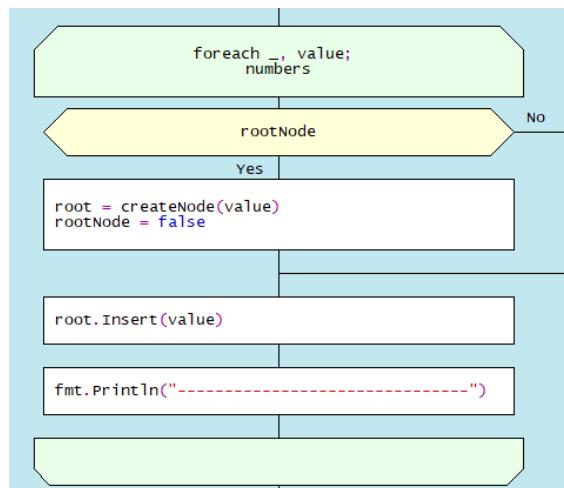
8.2. Двоичное дерево поиска

8.2.1. Построение бинарного дерева

Если дерево организовано таким образом, что для каждого узла все значения узлов его левого поддерева меньше значений этого узла, а все значения его правого поддерева – больше, оно называется *бинарным деревом поиска* (Binary Search Tree). Двоичное дерево поиска по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. Двоичное дерево поиска обладает следующими свойствами:

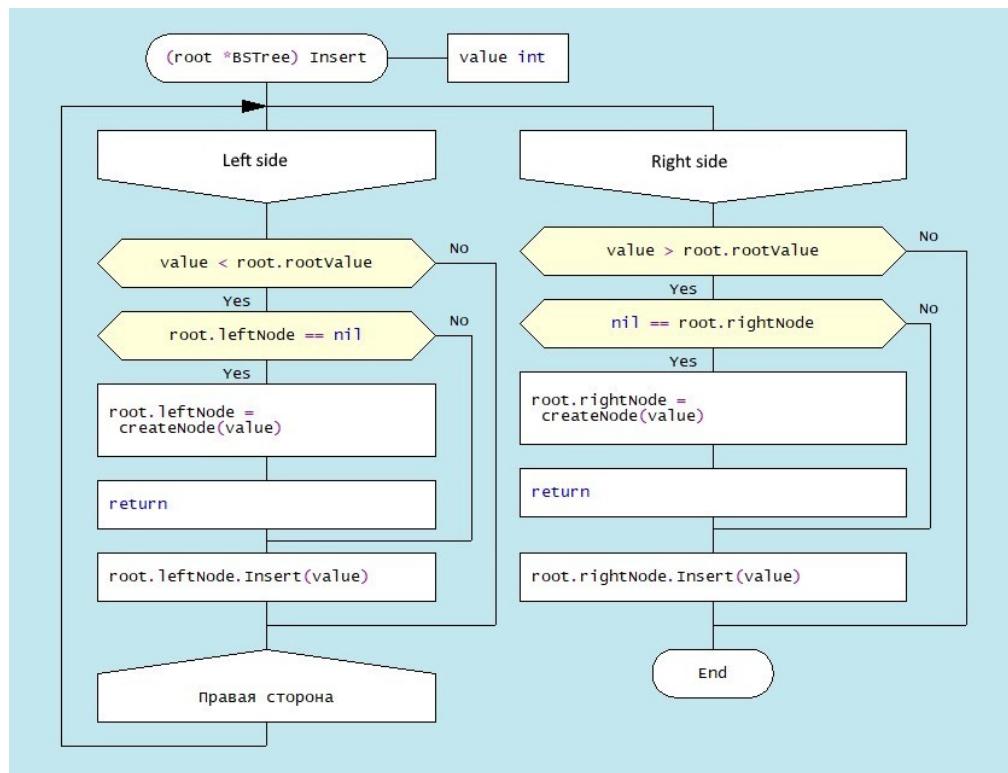
- дерево состоит из узлов, которые сохраняют уникальные значения;
- каждый узел имеет ноль, один или два дочерних узла;
- один из узлов обозначается как корневой узел, который находится в верхней части структуры дерева;
- каждый узел имеет только один родительский узел, за исключением корневого узла, который не имеет родительского;
- значение каждого узла больше, чем значение его левого потомка, но меньше, чем значение его правого потомка;

Построение бинарного дерева поиска осуществляется по определенному правилу (алгоритму). Пусть дана последовательность целых чисел $\{11, 5, 17, 15, 1, 8, 19, 13, 21\}$, представленная как срез $numbers[]$. Вначале формируется корневой узел $\{11\}$, затем в цикле для каждого узла рекурсивно вызывается метод *insertion (value)* (рис. 8.2 а), который в свою очередь вызывает метод *createNode(value)*, создающий новый узел (рис. 8.2 б) и размещающий его в левом или правом поддереве в зависимости от значения *value* (рис. 8.2 в).



а) построение дерева

б) создание узла



в) метод *Insert (value)*

Рис. 8.3. Дракон-диаграммы методов построения дерева

Полный процесс формирования бинарного дерева показан на рис. 8.4. Первое число 11 записывается в корень дерева. Второе число 5 меньше значения в корне дерева, следовательно, это число записывается в левое поддерево. Следующее число 17 больше значения в корне, соответственно оно записывается в правое поддерево. Далее число 15 больше, чем значение в корне дерева, значит, оно записывается в правое поддерево, но правое поддерево уже построено. Число 15 сравнивается со значением в корне правого поддерева — числом 17. Так как добавляемое значение меньше значения в корне правого поддерева, то добавляем левое поддерево уже к этому узлу. В конечном результате формируются бинарное дерево поиска, в котором представлены три варианта: узел 5 является родителем двух потомков (1,8), узел 15 имеет только одного левого потомка, а узел 19 — одного правого потомка. Такое расположение узлов выбрано для

демонстрации работы функции удаления узлов, что будет рассмотрено позже.

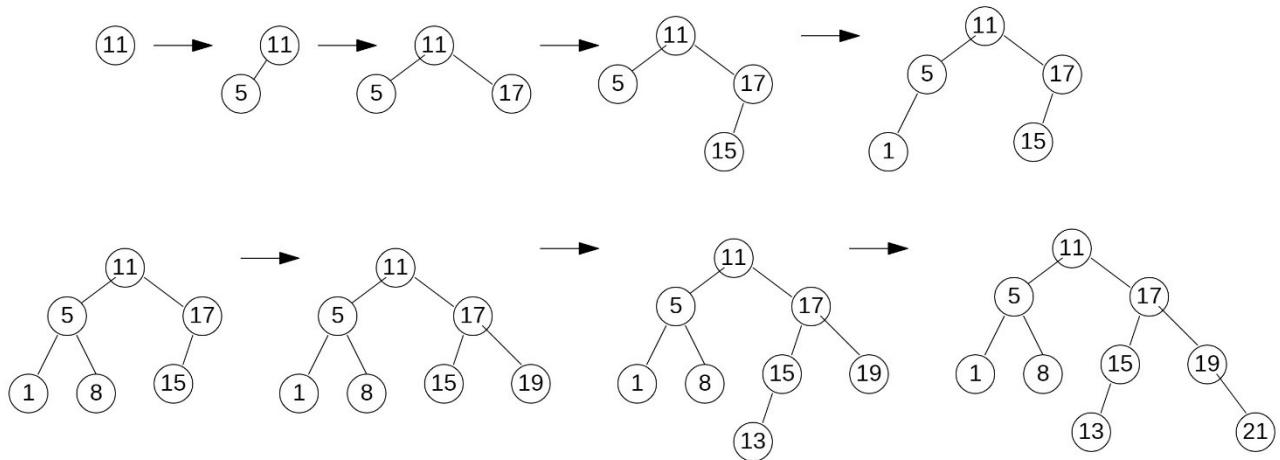


Рис. 8.4. Построение бинарного дерева

8.2.2. Поиск узла по заданному значению

Другой базовой операцией является функция поиск узла по его значению *findNode(value)*. В этой функции используется конструкция языка Golang – *Select*, где искомое значение *val* рекурсивно сравнивается со значениями других узлов в процессе обхода дерева. Заметим, что оператор *Select* в сгенерированном коде представляется операторами *if-else*. Если совпадение найдено, печатается результат “Узел существует”. Отсутствие узла с таким значением обнаруживается по признаку равенства *<nil>* адресов *root.LeftNode* и *root.RightNode* в переменной *root* типа *BSTree*:

```
BSTree struct {
    rootValue int
    leftNode  *BSTree
    rightNode *BSTree
}
```

}

Дракон-диаграмма функции `findNode(root *BSTree, val int)` представлена на рис. 8.5. Поиск осуществляется по всему дереву, конечные узлы определяются по выполнению условия (`root.LeftNode == nil && root.rightNode == nil`).

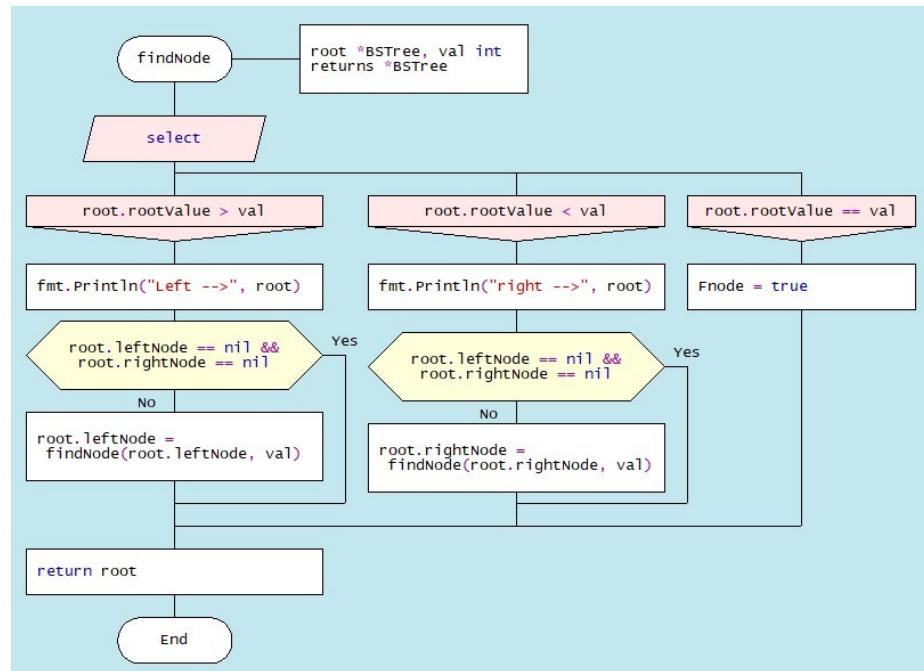


Рис. 8.5. Дракон-диаграмма метода `findNode(root *BSTree, val int)`

8.2.3. Удаление узла с заданным значением

Следующая базовая операция - удаление узла с указанным значением. Здесь рекурсивно используется функция `deleteNode (root *BSTree, val int)`. Алгоритм этой функции осложняется тем, что возможны следующие варианты расположения потомков, как, например, на рис. 8.3:

а) удаляемый узел 21 не имеет потомков;

б) удаляемый узел 19 имеет правого потомка;

в) удаляемый узел 15 имеет левого потомка;

г) удаляемый узел 5 имеет правого потомка.

Рассмотрим более подробно порядок перемещения узлов в указанных вариантах:

Вариант а): Узел (21) не имеет потомков (рис. 8.6.).

В этом случае этот узел удаляется путем изменения в родительском узле значения $root = nil$. На рис.8.6. показан процесс удаления узла (21) и соответствующий фрагмент дракон-диаграммы, где $L1 := root.leftNode == nil$ и $R1 := root.rightNode == nil$.

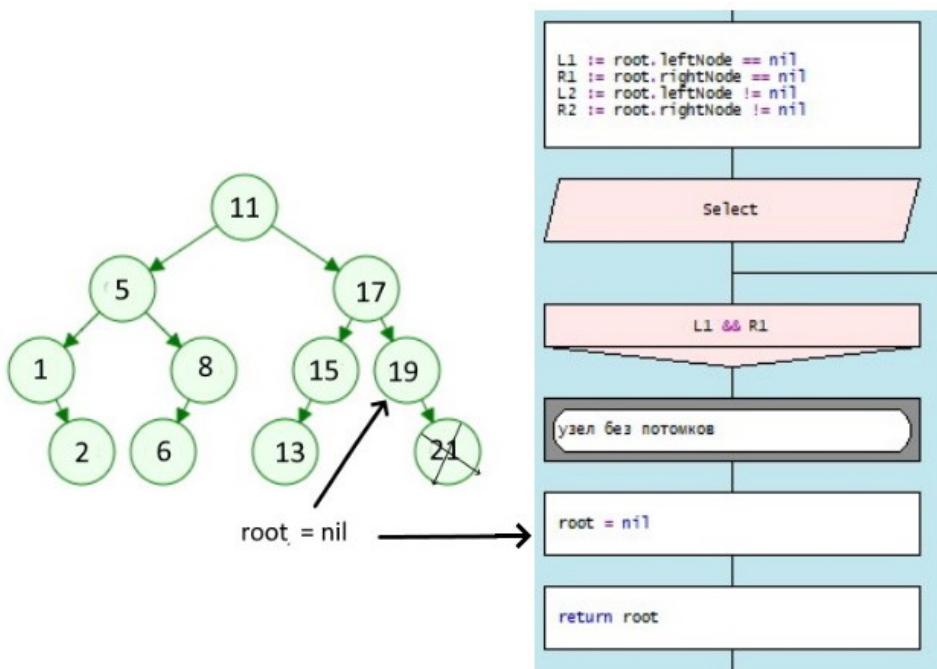


Рис. 8.6. Удаление узла без потомков

Вариант б): Узел (19) (рис. 8.7.) имеет правого потомка.

В этом случае узел (19) удаляется из дерева путем замены его адреса в родительском узле (17) на адрес единственного потомка – узла (21). На рис.

8.7. показан процесс удаления узла (19) и соответствующий фрагмент дракон-диаграммы, где $L1 := \text{root.leftNode} == \text{nil}$ и $R2 := \text{root.rightNode} != \text{nil}$.

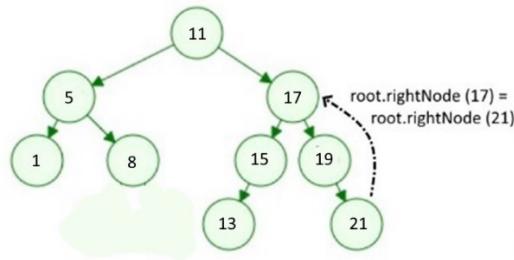


Рис. 8.7. Удаление узла (19) с правым потомком

Вариант в): Узел (15) имеет левого потомка (рис. 8.8.).

В этом случае узел (15) удаляется из дерева путем замены его адреса в родительском узле (17) на адрес единственного потомка – узла (13). На рис. 8.8. показан процесс удаления узла (15) и соответствующий фрагмент дракон-диаграммы, где $L1 := \text{root.leftNode} == \text{nil}$ и $R2 := \text{root.rightNode} != \text{nil}$.

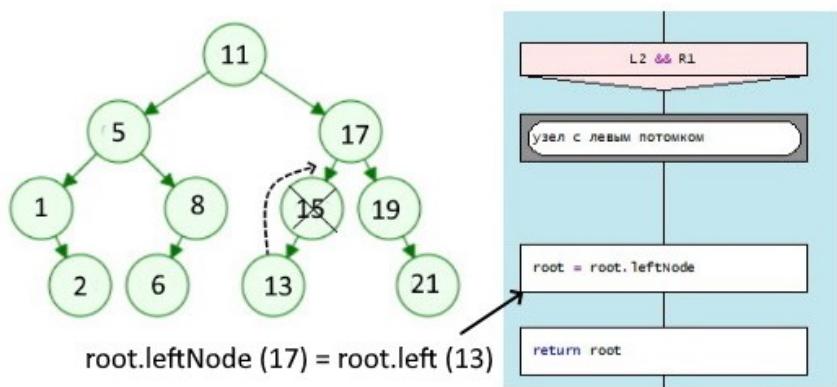


Рис. 8.8. Удаление узла (15) с левым потомком

Вариант г): Узел (5) имеет двух потомков (рис. 8.9.). В этом случае двоичное дерево поиска перестраивается: узел (2) переходит на место узла (5):

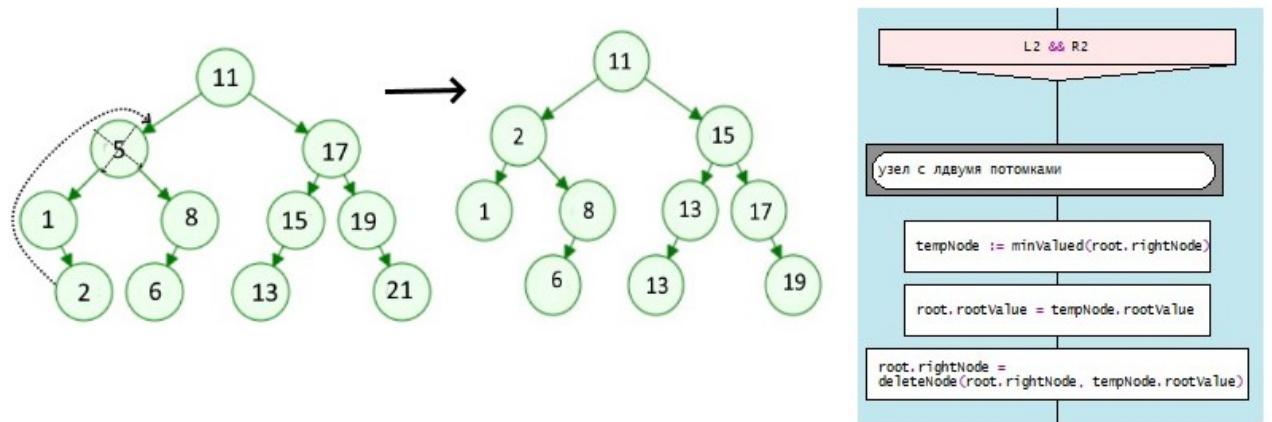


Рис. 8.9. Удаление узла (5) с двумя потомками

Полностью дракон-диаграмма метода удаления узла из двоичного дерева поиска имеет вид (рис. 8.10.):

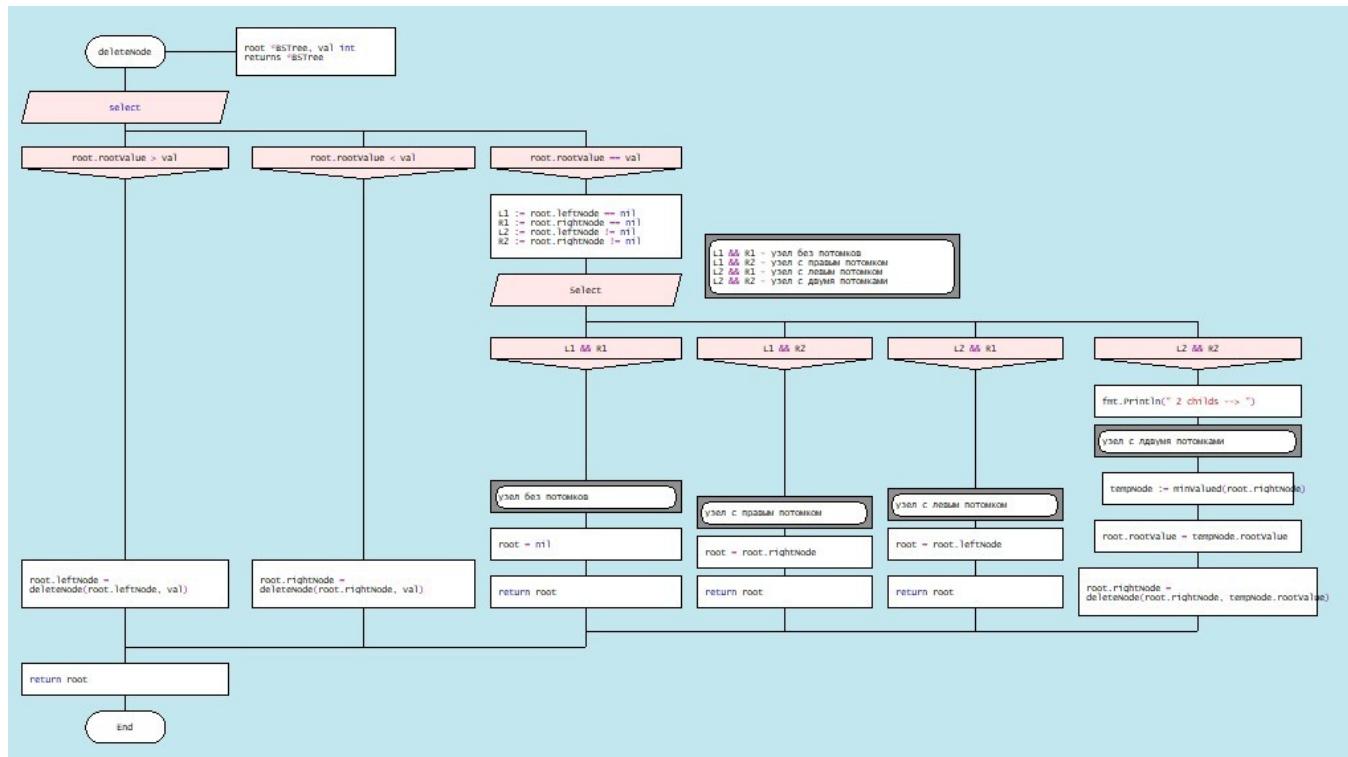


Рис. 8.10. Дракон-диаграмма функции удаления отдельного узла (4 варианта)

8.2.4. Обход двоичного дерева поиска

К базовым операциям с деревьями относится операция обхода дерева по узлам. В отличие от линейных структур данных, в которых обход элементов выполняется в линейном порядке, узлы деревьев можно обходить различными путями. Обход, при котором каждый узел-предок просматривается прежде его потомков, называется *предупорядоченным* обходом или обходом в прямом порядке (pre-order traversal). Обход, при котором вначале просматриваются потомки, а потом предки, называется *поступорядоченным* или обходом в обратном порядке (post-order traversal). Существует также центрированный обход (in-order traversal), при котором посещается сначала левое поддерево снизу-вверх, затем корневой узел, затем — правое поддерево (рис. 8.11, 8.12, 8.13)].

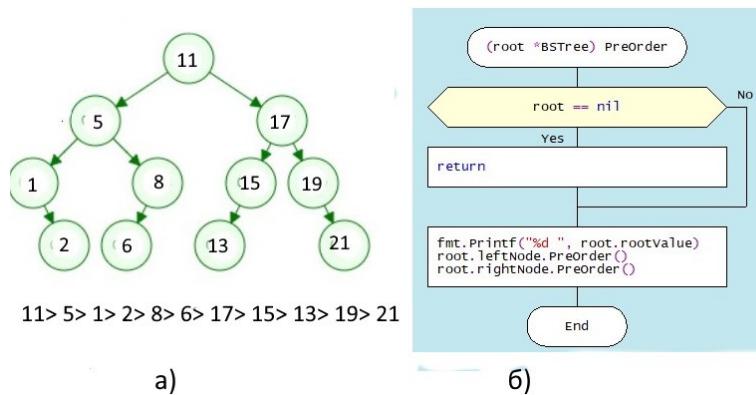


Рис. 8.11. а) Прямой обход дерева ;б) дракон-диаграмма функции PreOrder()

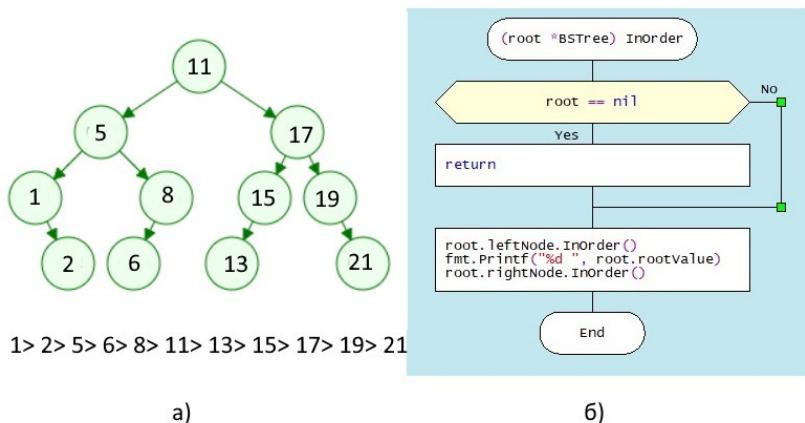


Рис. 8.12. а) Центрированный обход; б) дракон-диаграмма функции In-Order()

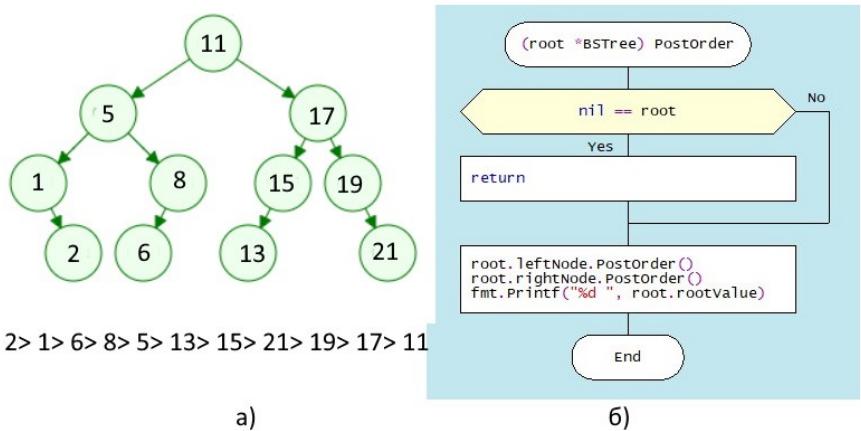


Рис. 8.13. а) Обратный обход; б) Дракон-диаграмма функции PostOrder()

Общий вид программы, включающей все драконовские диаграммы реализации алгоритмов основных и вспомогательных функций, представлен на рис. 8.14.

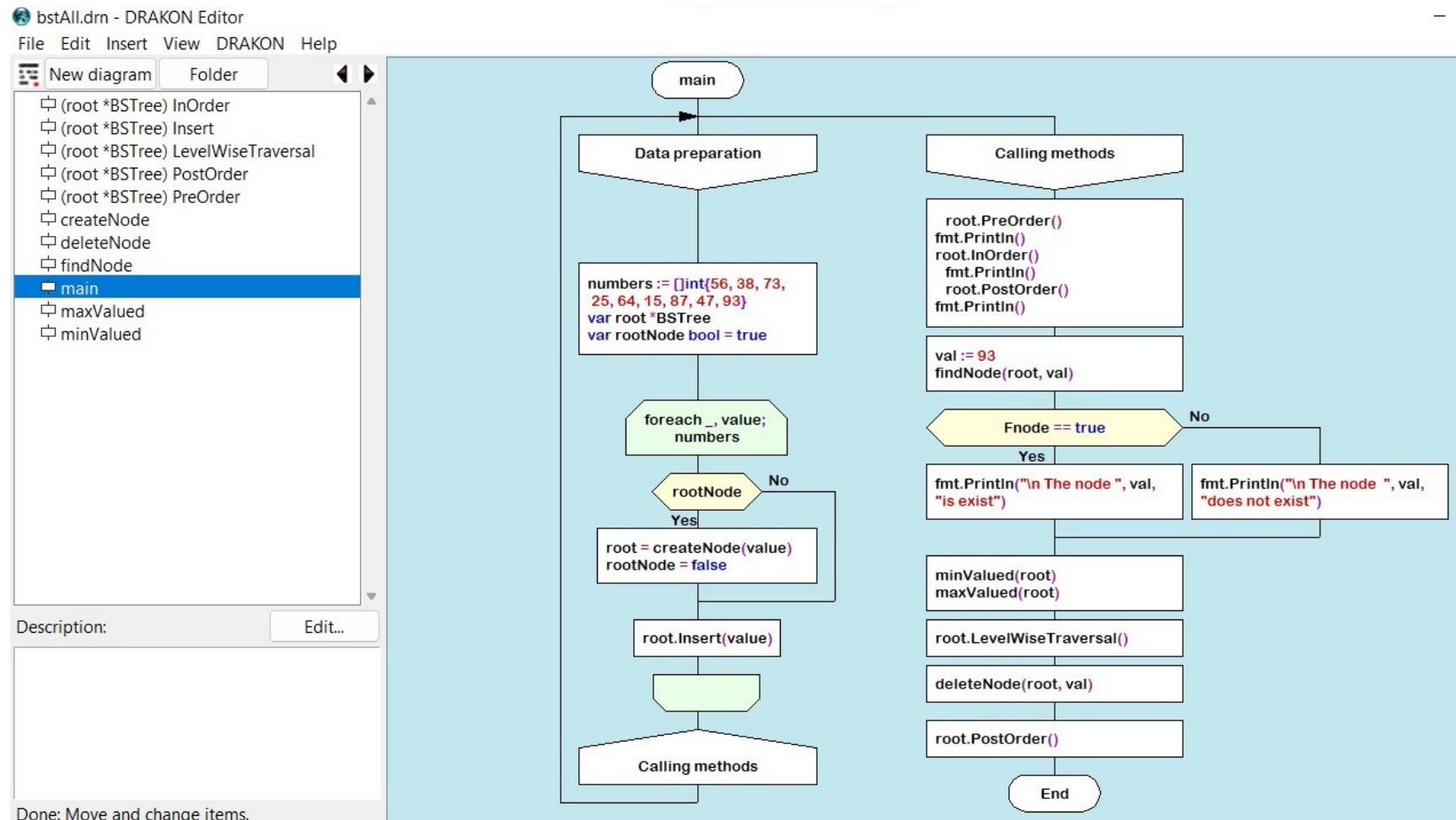


Рис. 8.14. Общий вид программы для реализации основных функций

BST-дерева

8.3. Самобалансирующиеся двоичные деревья (AVL-деревья)

Эффективность выполнения любых операций с деревьями существенно зависит от порядка поступления входных данных. Например, если поступающая последовательность чисел частично отсортирована по возрастанию или убыванию, то внешний вид такой структуры уже не будет похож на дерево (рис. 8.15).

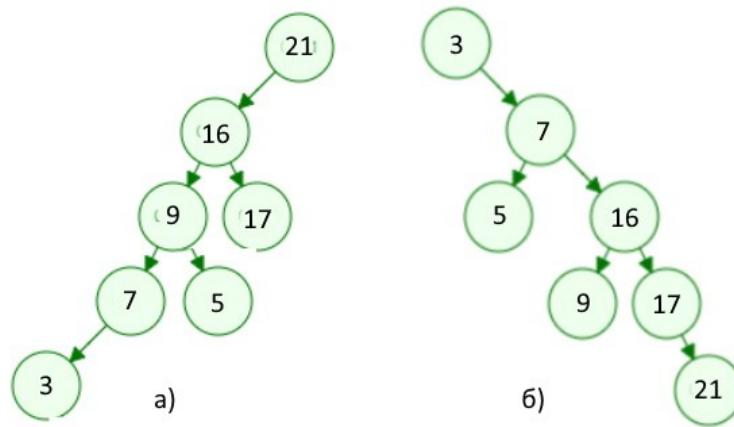


Рис. 8.15. Частично отсортированные входные данные

В таких практически «вырожденных» деревьях сложность операций определяется числом узлов, то есть, близка к линейной - $O(n)$. При этом левое и правое поддерево неуравновешены (несбалансированы), что можно оценить коэффициентом баланса (kb), равным разности высот левого и правого поддеревьев. Напомним, что высота дерева определяется как длина самой длинной ветви в поддереве (сумма ребер). Для идеального двоичного дерева поиска (дерево, в котором число узлов в левом поддереве равно числу узлов в правом поддереве) этот коэффициент равен 0 (рис. 8.16).

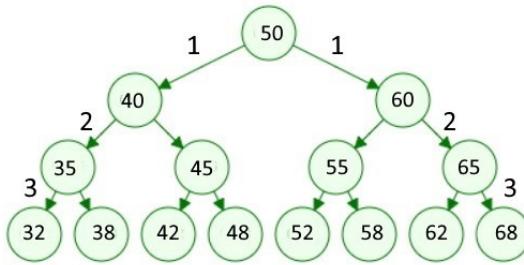


Рис. 8.16. Идеальное двоичное дерево

В реальной жизни идеальные двоичные деревья практически не встречаются, чаще программисты стремятся построить дерево, в котором высота левого поддерева отличается от высоты правого не более чем на 1. Такие деревья получили название AVL-деревьев; для таких деревьев сложность операций определяется как $O(\log n)$, то есть время выполнения базовых операций (поиск, удаление) существенно меньше чем BST-деревья.

Алгоритмы построения таких деревьев основаны на процессе балансировки дерева при вставке нового узла или удалении какого-либо существующего узла. Целью балансировки является перестройка дерева таким образом, чтобы высоты левого и правого поддеревьев не отличались более чем на 1. При этом коэффициент балансировки должен удовлетворять следующим условиям:

- допустимые значения $kb = -1, 0$ и $+1$;
- значение $kb = -1$ указывает на то, что правое поддерево «перевешивает» левое поддерево;
- значение $kb = +1$ указывает, что левое поддерево «перевешивает» правое поддерево;
- значение $kb = 0$ показывает, что дерево содержит равное число узлов с каждой стороны, то есть дерево идеально сбалансировано.

Технология балансировки сводится к выполнению круговых перемещений узлов в четырех вариантах:

- правый поворот (RR);
- Левый поворот (LL);
- Право - левый поворот (RL);
- Лево - правый поворот (LR).

Правый поворот выполняется, когда корневой узел имеет коэффициент баланса $kb = +2$, а его левый потомок имеет коэффициент баланса $kb = +1$ (рис. 8.17):

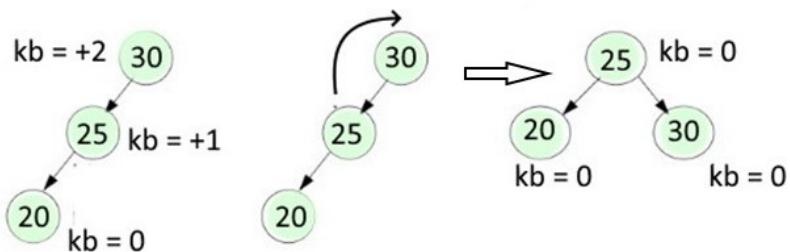


Рис. 8.17. Правый поворот

Левый поворот выполняется, когда корневой узел имеет сбалансированный коэффициент $kb = -2$, а его правый потомок имеет коэффициент баланса $kb = -1$ (8.18):

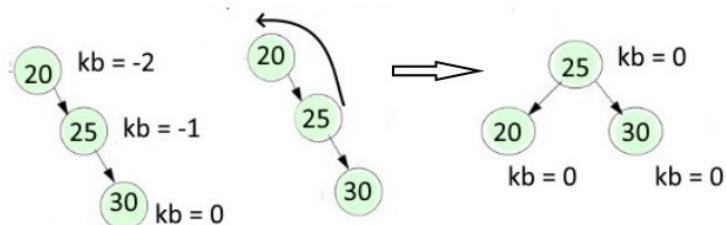


Рис. 8.18. Левый поворот

Право-левый поворот выполняется, когда корневой узел имеет коэффициент баланса $kb = -2$, а его правый потомок имеет коэффициент баланса $kb = +1$ (рис.8.19):

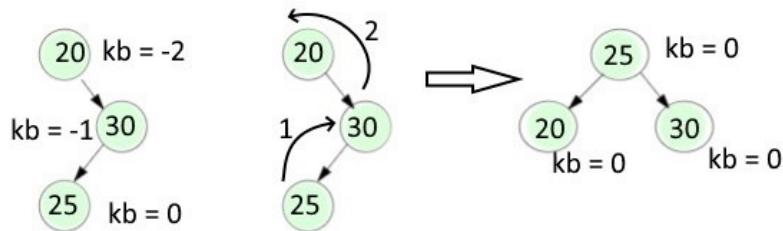


Рис. 8.19. Право-левый поворот

Лево-правое вращение выполняется, когда у узла коэффициент баланса равен $kb = +2$, а у его левого потомка коэффициент баланса равен $kb = -1$ (рис.8.20):

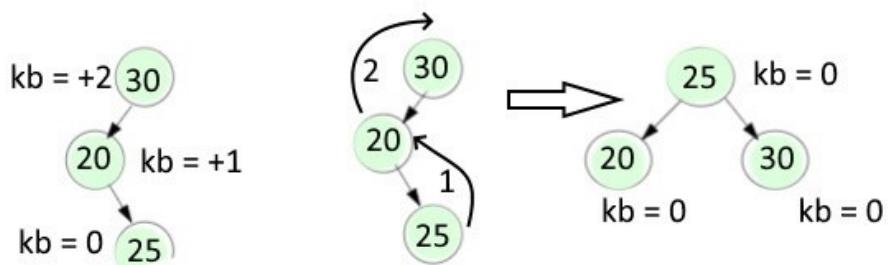


Рис. 8.20. Лево-правый поворот

Рассмотрим процесс балансировки более подробно на примере формирования дерева при поступлении входных данных в таком порядке: 21, 17, 16, 11, 9, 7, 5, 3. Такое дерево несбалансированно, точнее даже представляет собой практически «хворостину» (рис. 8.21):

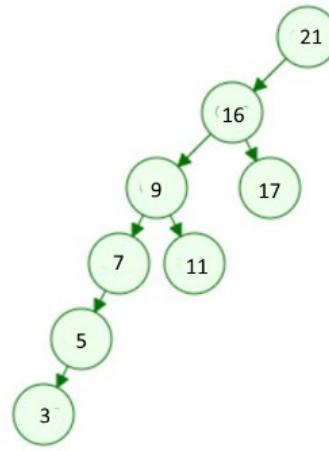


Рис. 8.21. Несбалансированное дерево

Проверка необходимости балансировки начинается при поступлении каждого нового узла с помощью метода *InsertNode(node *node, value int)*, определяющего расположение нового узла в левом или правом поддереве относительно корневого узла (рис. 8.22).

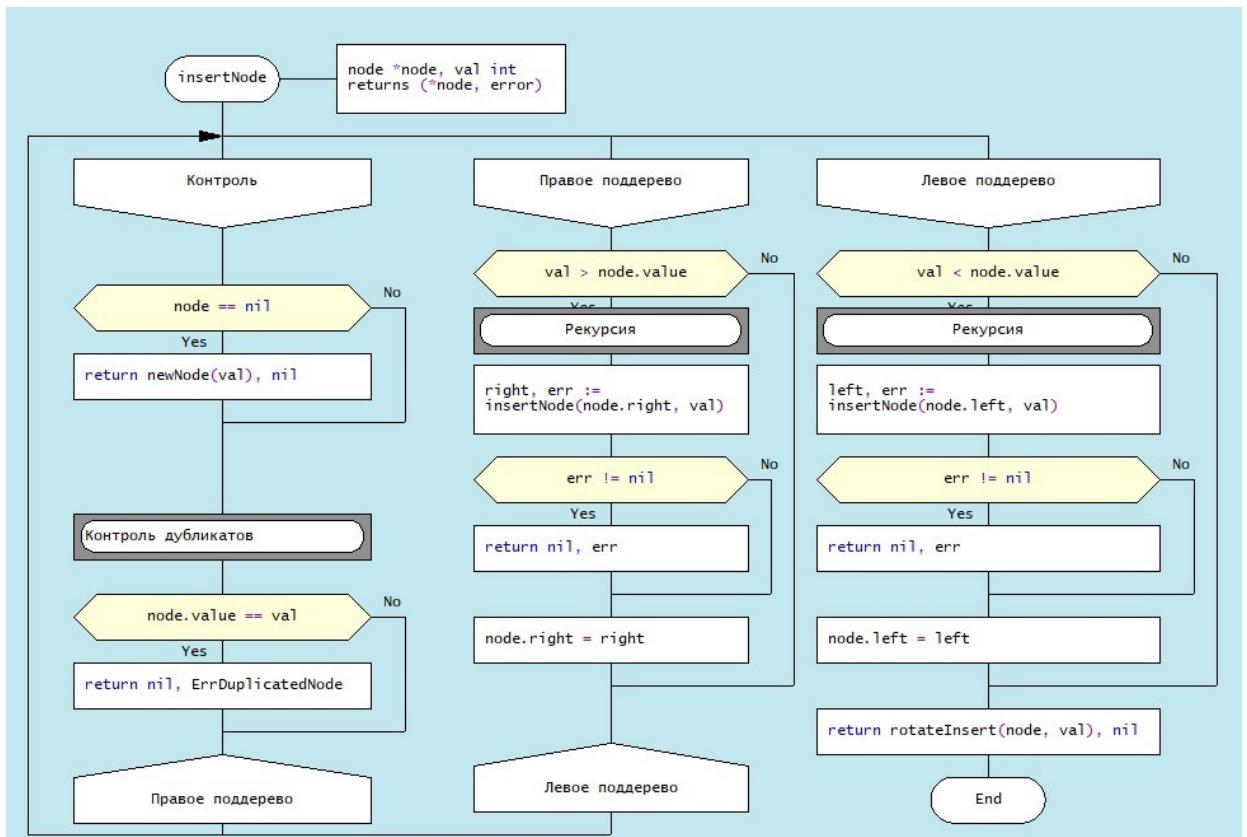


Рис. 8.22. Дракон-диаграммы метода вставки нового узла *insertNode*

Выбор одного из возможных путей балансировки (поворотов) определяется методом *rotateInsert(node *node, val int)* (рис. 8.23).

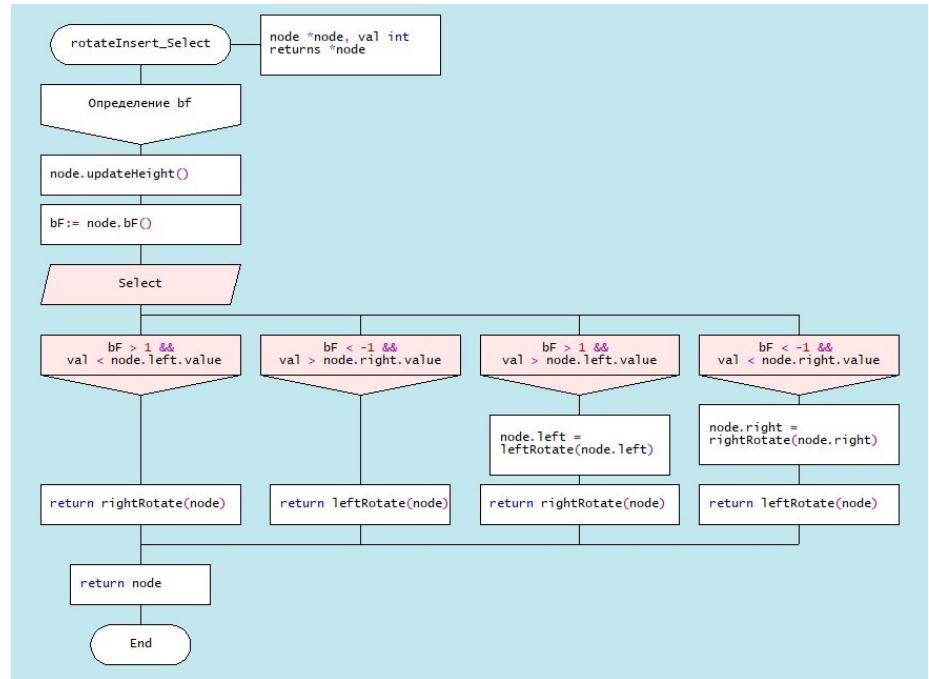


Рис. 8.23. Дракон-диаграмма метода *rotateInsert(node *node, val int)*.

В зависимости от расположения нового узла относительно родительского и от значений коэффициента балансировки выполняется вызов методов левого или правового вращений (рис. 8.24):

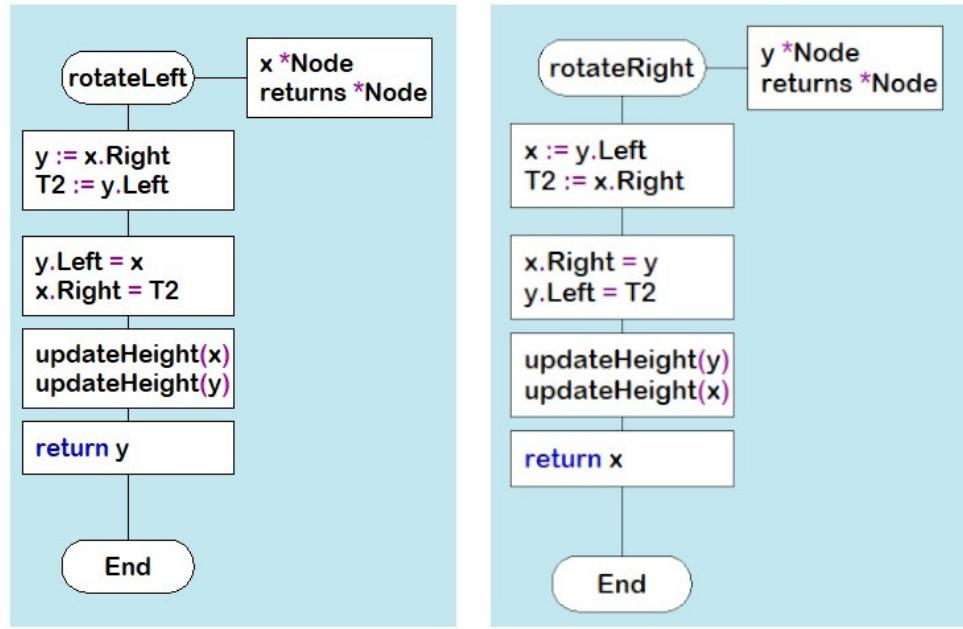


Рис. 8.24. Программные коды вращения узлов

Рассмотрим более подробно процесс перестройки дерева как результат применения метода *rotateInsert(node *node, val int)* на примере поступления значений трех узлов (21,17,16). После введения значения узла (16) вместо дерева появляется «хвостина», требующая балансировки. В данном случае в методе *rotateInsert(node *node, val int)* выполняется условие: коэффициент балансировки равен 2, и значение узла (16) меньше родительского узла (17) и происходит обращение к методу *rightRotate(node)*. Процесс балансировки показан на рис.8.25:

Insert + insertNode						
main	21		17		16	
	0xc000020460	<nil>	0xc0000204e0	<nil>	<nil>	<nil>
	address of node (17)	No childs	address of node (16)	No childs	No childs	No child: s
rotateInsert		rightRotate				
		17	16	21		
factorbalance = 2	val < node.left.value	0xc0000204e0	0xc000020400	<nil>	<nil>	<nil>
		address of node (16)	address of node (21)	No childs	No childs	No childs
				17	16	21

Рис. 8.25. Процесс вращения узлов для балансировки

Как видно из этого рисунка, результатом работы метода *rightRotate(node)* является значений полей адресов элементов структуры *node*. Напомним, что эта структура состоит из четырех полей: высоты дерева, значения узла и адресов левого и правого узлов потомков. На рис.8.18. под значением узла показаны адреса узлов-потомков. После перестройки поля структуры *node.left* и *node.right* узлов (17) и (21) меняются и узел (17) становится корневым, узел (16) остается без изменений, а бывший корневой узел (21) становится правым потомком узла (17). В результате поступления узла (16) дерево становится сбалансированным.

Процесс поступления новых узлов и перестройка возникающего дерева путем балансирования для набора входных данных (21,17,16,11,9,7,5,3) представлен на рис. 8.26.

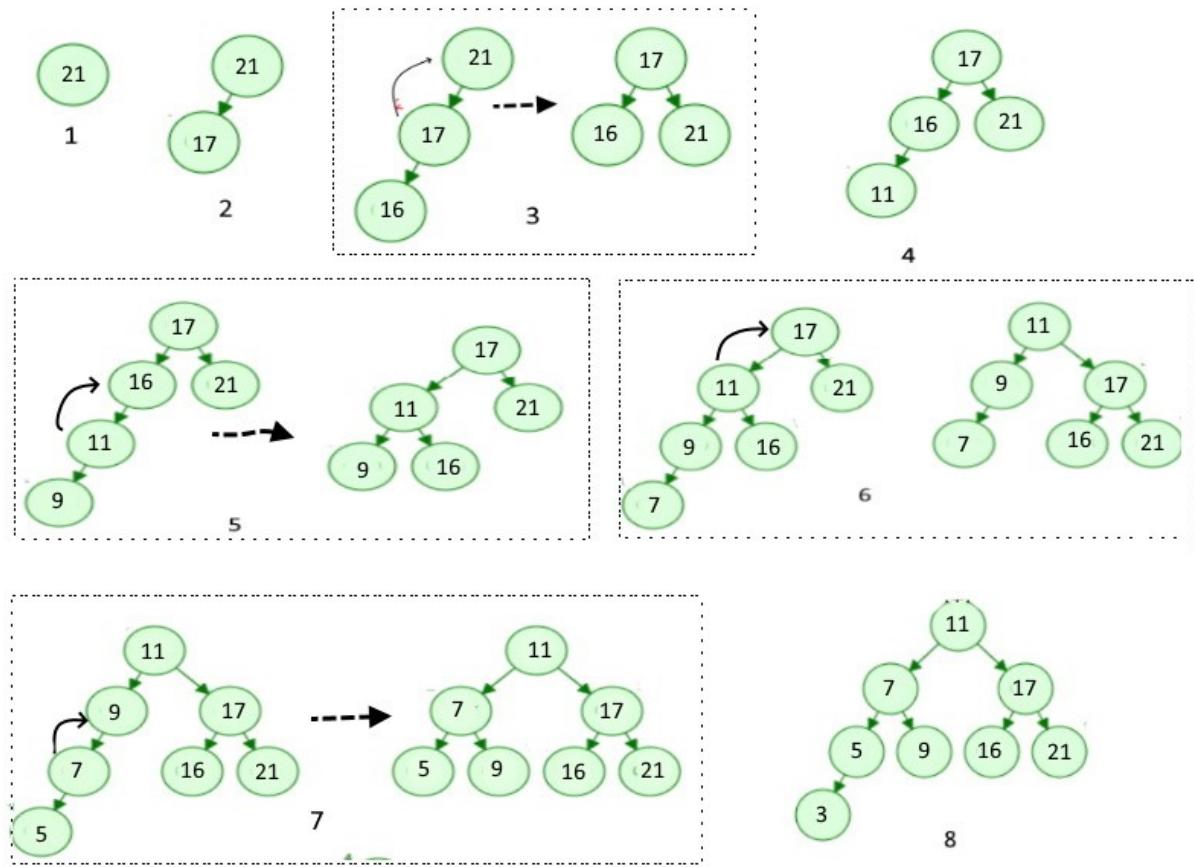


Рис. 8.26. Процесс построения сбалансированного AVL-дерева

Основными операциями с AVL-деревьями являются операция нахождения узла по заданному значению (`findNode(node * node, val)`) и операция удаления (`removeNode(node *node, val)`). Дракон-диаграмма метода `findNode(node * node, val)` представлена на рис. 8.27:

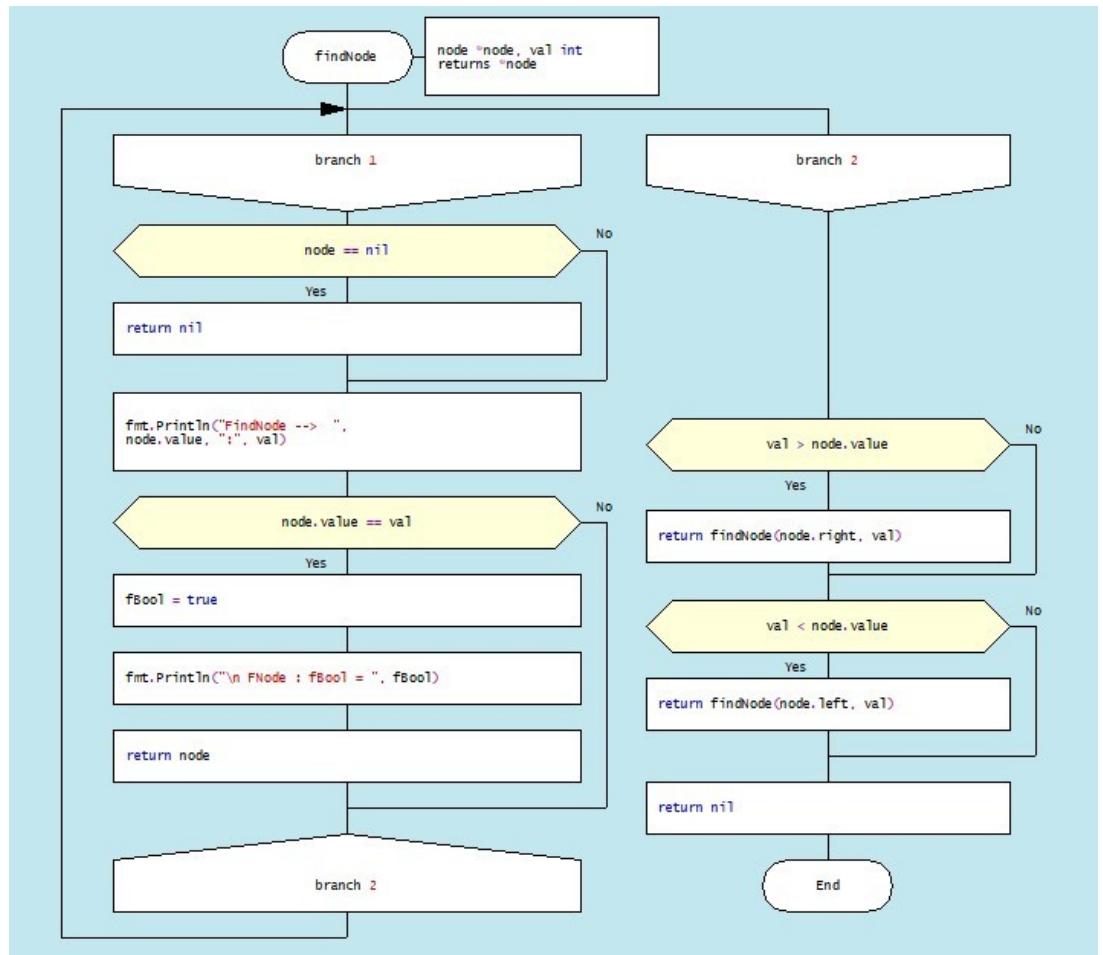


Рис. 8.27. Дракон-диаграмма метода нахождения узла с искомым значением

Другая базовая операция – удаление узла с указанным значением – состоит из следующих этапов. Поиск узла осуществляется с корня вниз по ветвям до удаляемого узла. При этом могут возникнуть следующие ситуации (рис. 8.28):

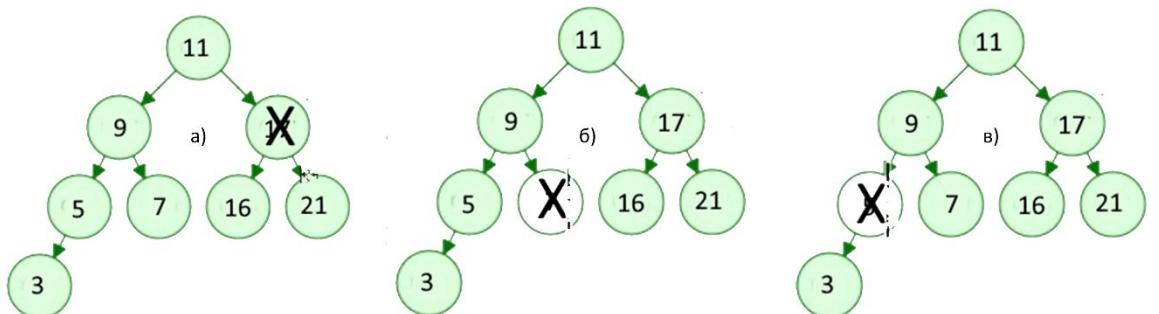


Рис. 8.28. Варианты удаляемых узлов

- а) Удаляемый узел имеет два непустых потомка;
- б) Удаляемый узел не имеет потомков;
- в) Удаляемый узел имеет один потомок (левый или правый).

Как и в других методах вначале рекурсивно определяется узел с заданным значением, а затем выбирается один из представленных вариантов. Дракон-диаграмма алгоритма удаления узла с указанным значением показан на рис. 8.29.

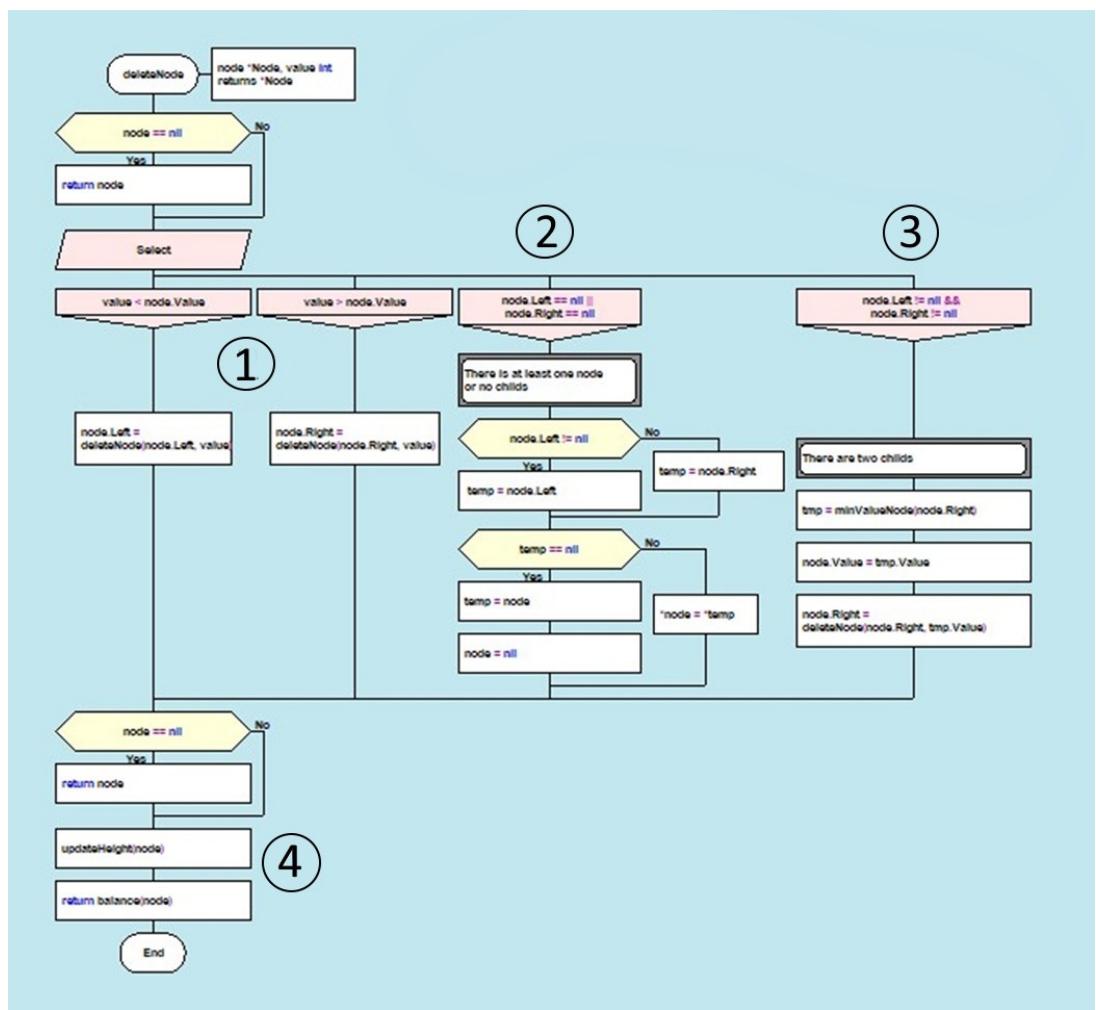


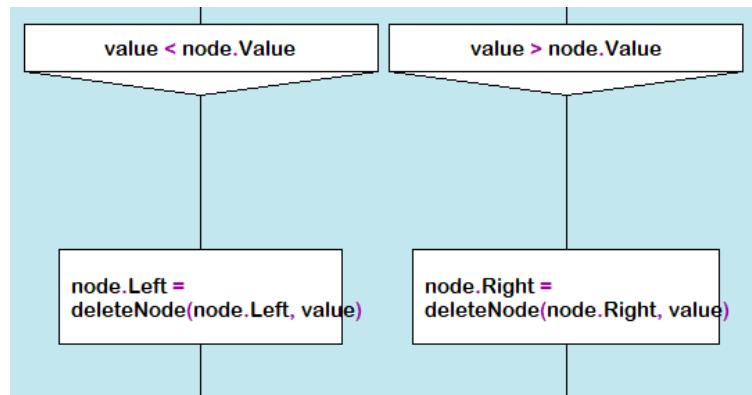
Рис. 8.29. Дракон-диаграмма алгоритма удаления узла с заданным значением

Словесное описание алгоритма удаления узла avl-дерева с конкретным значением.

Фрагмент 1: Обход дерева:

Если удаляемое значение меньше значения текущего узла, алгоритм рекурсивно вызывает функцию `deleteNode` в левом поддереве.

Если удаляемое значение больше значения текущего узла, алгоритм рекурсивно вызывает функцию `deleteNode` в правом поддереве.

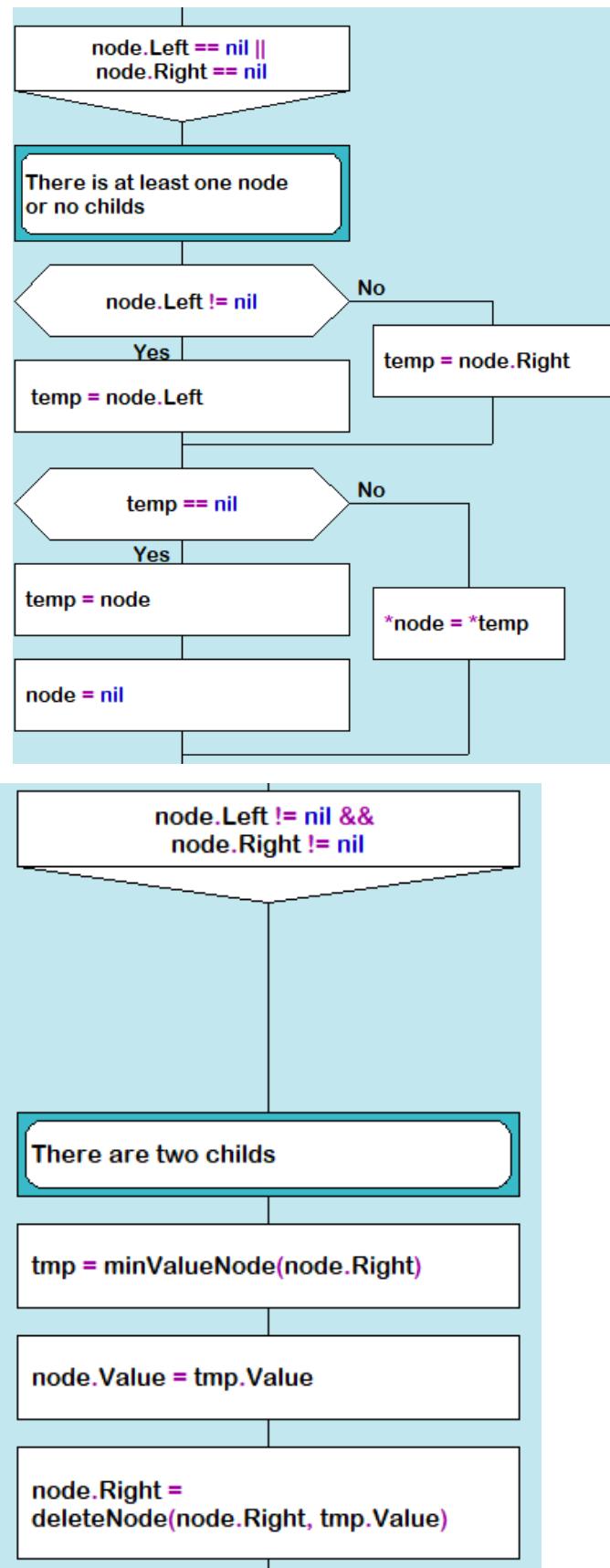


Фрагменты 2 и 3: Работа с удаляемым узлом:

Если удаляемое значение равно значению текущего узла, алгоритм проверяет следующие случаи:

Если у узла нет дочерних элементов или есть только один дочерний элемент, он удаляет узел напрямую, корректируя указатели.

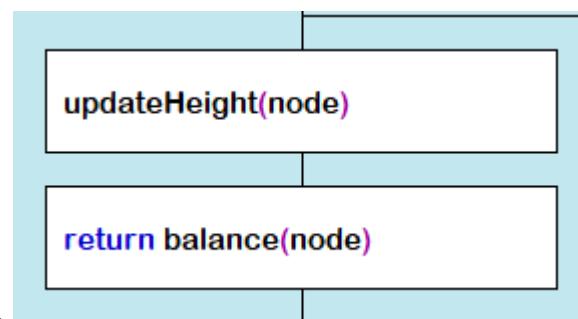
Если у узла есть два дочерних элемента, он находит узел с минимальным значением в правом поддереве, заменяет значение текущего узла на значение минимального узла, а затем рекурсивно удаляет минимальный узел из правого поддерева.



Фрагмент 4: Обновление дерева:

После операции удаления алгоритм обновляет высоту узлов и при необходимости балансирует дерево (*updateHeight(node)*).

Алгоритм использует рекурсию для обхода и изменения древовидной структуры. Он возвращает измененный узел после операции удаления (*return balance(node)*).



Программа *main()* вводит массив данных, ищет узел с заданным значением и удаляет узел с заданным значением. Результат реализации сгенерированного кода показан ниже:

Traversing the tree prior to removal:

preOrder 11 7 5 3 9 17 16 2 1

inOrder 3 5 7 9 11 16 17 21

postOrder 3 5 9 7 16 21 17 11

The number 3 is present

Number 6 is missing

Removing the node 7

Traversing the tree after deleting number 7

preOrder 11 5 3 9 17 16 21

inOrde 3 5 9 11 16 17 21

postOrder 3 9 5 16 21 17 11

Общий вид программы, включающей все драконовские диаграммы реализации алгоритмов основных и вспомогательных функций, представлен на рисунке 8.30:

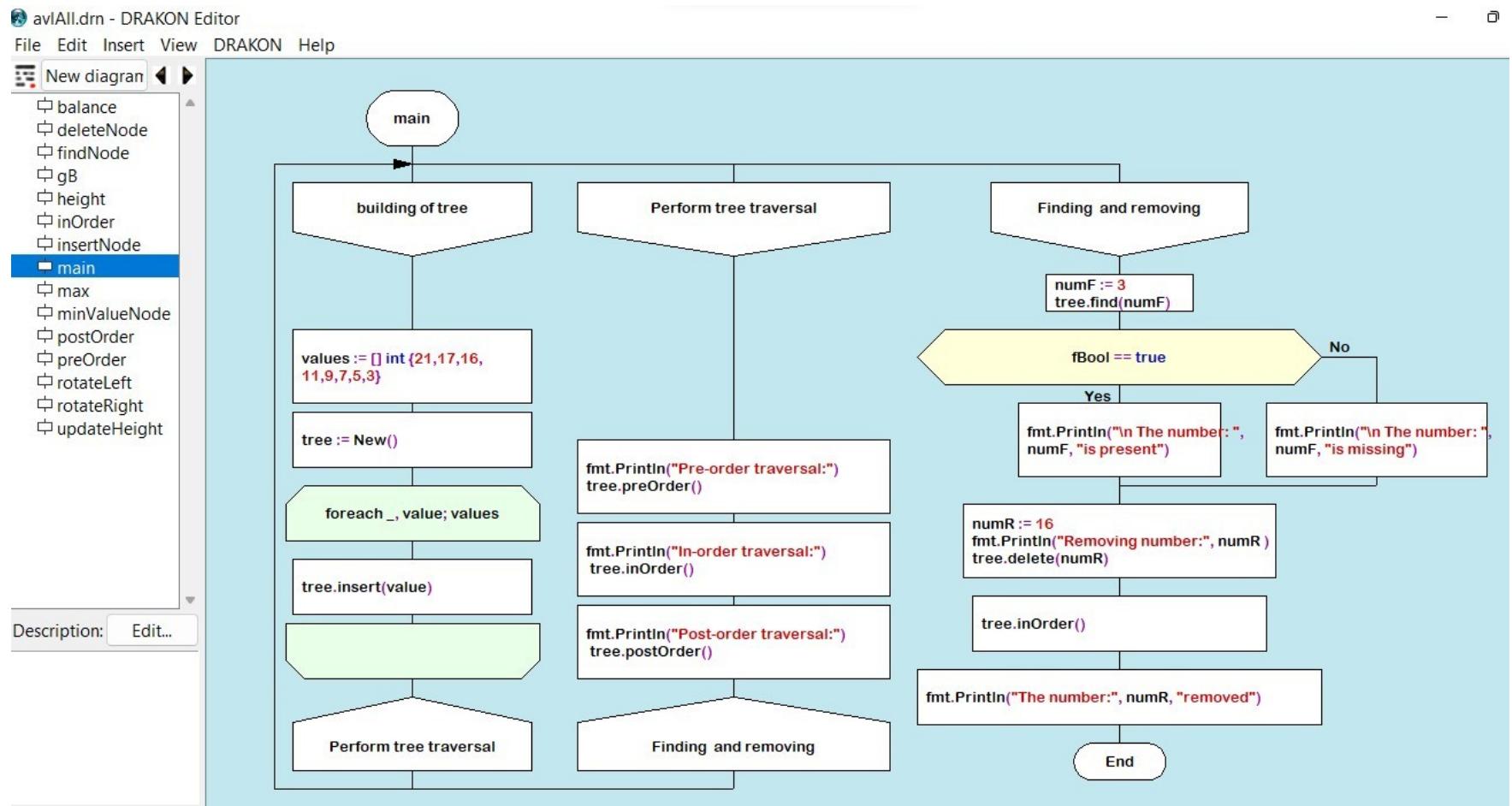


Рис. 8.30 Общий вид программы для реализации основных функцийavl

Описание файла (avlAll)

==== header ==== package main import ("fmt") var fBool bool = false var temp, tmp *Node type Node struct { Value, Height int Left, Right *Node } type Tree struct { root *Node }	func New() *Tree { return &Tree{ } } func (t *Tree) insert(value int) { t.root = insertNode(t.root, value) } func (tree *Tree) find(value int) *Node { return findNode(tree.root, value) } func (t *Tree) delete(value int) { t.root = deleteNode(t.root, value) }	func (t *Tree) pre- Order() { preOrder(t.root) fmt.Println() } func (t *Tree) in- Order() { inOrder(t.root) fmt.Println() } func (t *Tree) pos- tOrder() { postOrder(t.root) fmt.Println() } ==== footer ==== }
---	---	---

--	--	--

8.4. Красно-черные деревья

Пути доступа к программным файлам проекта rbt

Дракон-диаграмма	https://github.com/ISA-victory/dsa-dg.git ,
Сгенерированный код	https://github.com/ISA-victory/dsa-dg.git

Красно-чёрное дерево представляет собой разновидность самобалансирующегося двоичного дерева поиска, в котором узлы размещены по определенному правилу и окрашены в красный или чёрный цвета (рис. 8.31)

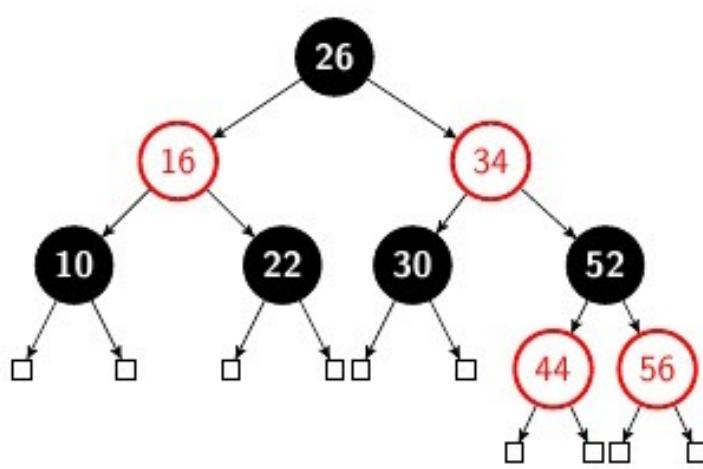


Рис. 8.31. Красно-чёрное дерево

Узлы, содержащие данные (в данном случае – целые числа), являются внутренними. Кроме того, красно-чёрные деревья содержит мнимые, «нулевые» узлы, связанные с листьями дерева (Null – на рис. 8.31). Красно-

черные деревья удовлетворяют всем свойствам двоичного дерева поиска и должны обладать следующими свойствами:

1. Каждый узел окрашен в красный или черный цвет.
2. Корень дерева всегда черный.
3. Все листья черные (Null).
4. Оба потомка красного узла чёрные, то есть не может быть последовательных красных узлов.
5. Все простые пути от узла к нисходящим листьям содержат одинаковое число чёрных узлов.

В отличие от AVL-деревьев, в которых баланс достигается соотношением высот левого и правого поддеревьев, баланс красно-черного дерева достигается свойствами, указанными выше. Добавление или удаление узла из красно-черного дерева может нарушить свойства красно-черного дерева и восстановление баланса достигается путем двух операций: перекрашивания узлов и (или) перестройки всего дерева или его поддеревьев с помощью определенных вращений.

Наиболее важными из перечисленных свойств являются свойства 4 и 5. Следствием свойства 4 является требование одинакового количества красных и чёрных узлов на пути между двумя узлами. Свойство 5 требует, чтобы независимо от выбора пути между двумя узлами число чёрных узлов должно быть одинаковым. Иными словами, в худшем случае высота дерева не должна быть более чем в два раза превосходить высоту кратчайшего пути. В этом случае красно-черное дерево становится достаточно сбалансированным.

8.4.2. Вставка нового узла и процесс балансировки

Для достижения сбалансированного красно-черного дерева процесс вставки нового узла сопровождается проверкой на соответствие вышеуказанным свойствам. Поскольку красно-черный цвет является двоичным деревом поиска (BST), он сначала определяет, где на дереве должен быть размещен узел, окрашивая его в красный цвет (рис. 8.32):

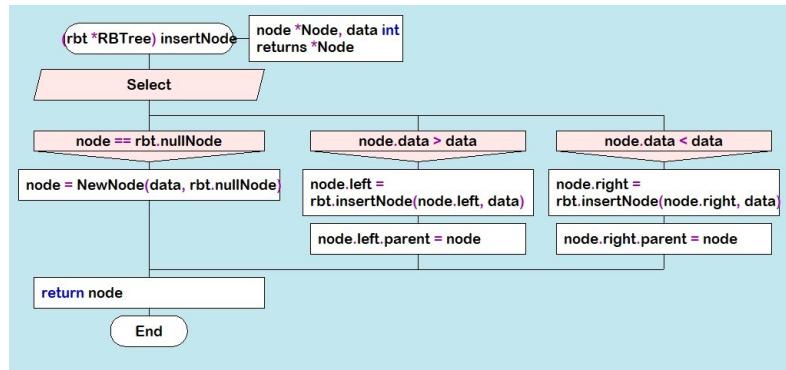


Рис. 8.32. Дракон-диаграмма, изображающая алгоритм вставки узла

Этот метод вызывает функцию *NewNode(d, nullNode)*, которая возвращает полную информацию о новом узле, при этом цветовое поле типа *bool* изначально принимает значение *red*:

<pre> func NewNode(d int, nullNode *Node) (nd *Node) { nd = &Node{} nd.data = d nd.left = nullNode nd.right = nullNode </pre>	<pre> type Node struct { data int colour bool left, right, parent *Node } </pre>
---	--

```

nd.parent = nullNode

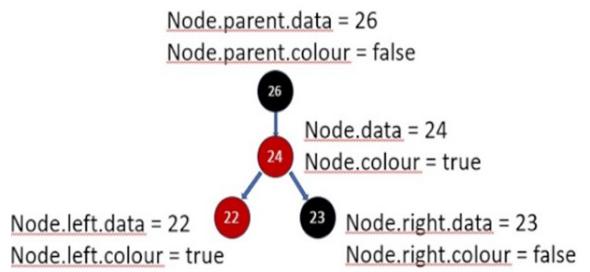
nd.colour = true

// new node - red

return

}

```



Далее рассмотрим положение узлов в дереве в результате каждой вставки нового узла и определим, какие свойства красно-черного дерева нарушаются. Чтобы понять этот процесс, мы вводим следующую нотацию: x - новый узел, P (родитель) - родитель узла x , G (прапородитель) - предок (родитель родителя), U (дядя) - дядя узла x (рис. 8.33):

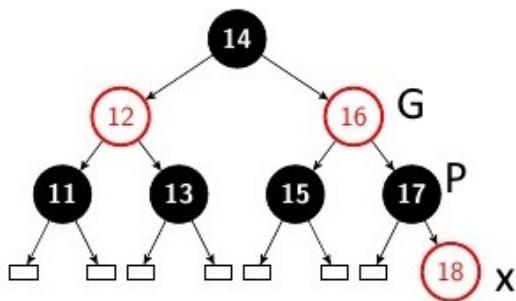


Рис. 8.33. «Изображение ассоциированных узлов x , P , G , U

Когда новый элемент вставляется, ему присваивается красный цвет. Чтобы удовлетворить первые два правила, достаточно просто перекрасить новые вершины в нужный цвет. После каждой вставки необходимо проверять все свойства

красно-черного дерева. Если хотя бы одно свойство не удовлетворяет, выполняются операции поворота и изменения цвета. Рисунок 8.34. показывает все случаи взаимного расположения связанных узлов и фрагменты соответствующей дракон-диаграммы.

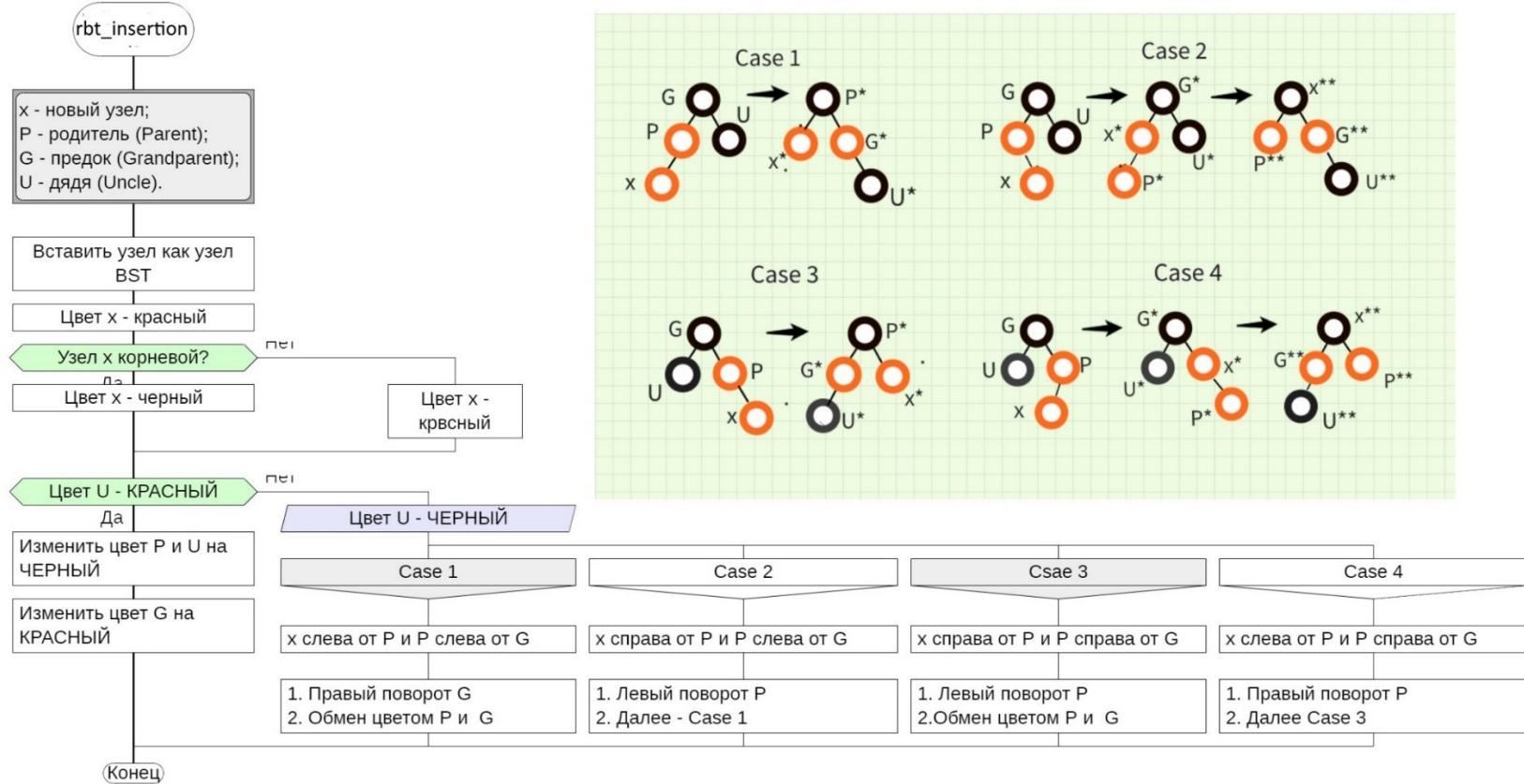


Рис.8.34. Фрагменты дракон-диаграммы вариантов взаимного расположения родственных узлов (x, P, G, U)

Рассмотрим процесс балансировки на примере построения красно-черного дерева из наборов целых чисел, которые являются ключами входных узлов: $\{11, 12, 13, 14, 15, 16, 17, 18, 19\}$ (рис. 8.35):

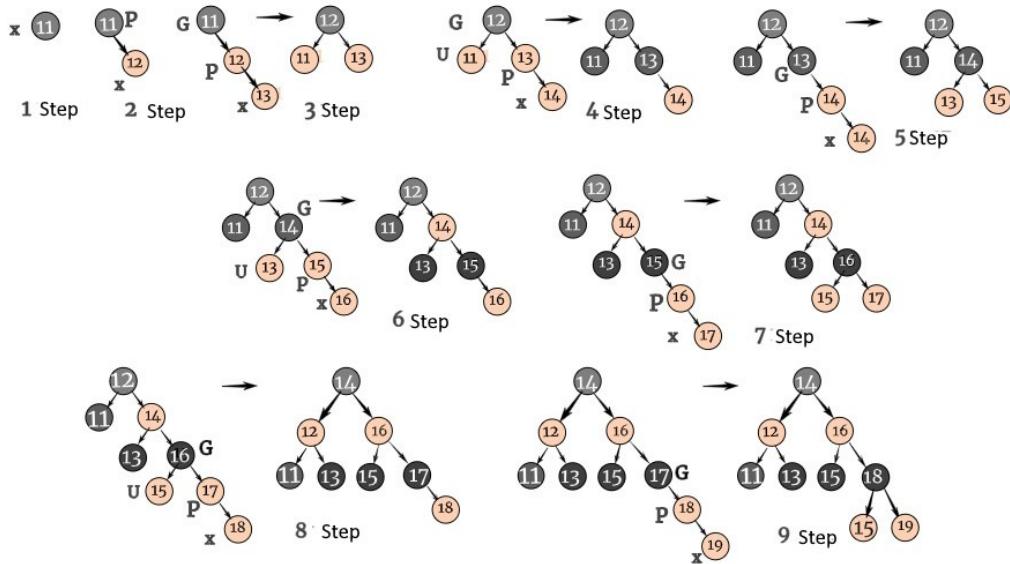


Рис. 8.35. Процесс балансировки красно-черного дерева

Первые два шага не приводят к нарушению свойств. Вход в узел с ключом (13) приводит к нарушению свойства #4. Для восстановления равновесия на третьем шаге выполняется поворот влево против часовой стрелки, в результате чего узел (12) становится корневым узлом черного цвета, а бывший корневой узел (11) становится левым дочерним. На следующем шаге возникает ситуация, когда три узла становятся красными: новый узел $x(14)$, родительский $P(13)$ и $uncle U(11)$, здесь баланс может быть достигнут путем изменения цвета узлов (11) и (13).

Другие наборы ключей создают другие комбинации связанных узлов (x, P, G, U) дерева, которые уравновешиваются вращением узлов и изменением цвета, как показано на дракон-диаграмме (рис. 8.36). Эта диаграмма

создается в режиме «СИЛУЭТ» и состоит из четырех ветвей. В первой ветви корневой узел окрашивается в черный цвет. Вторая ветвь определяет иерархию связанных узлов (x , P , G , U). Третья ветвь раскрашивает эти узлы в соответствии с иерархией. Четвертая ветвь вызывает соответствующие методы поворота.

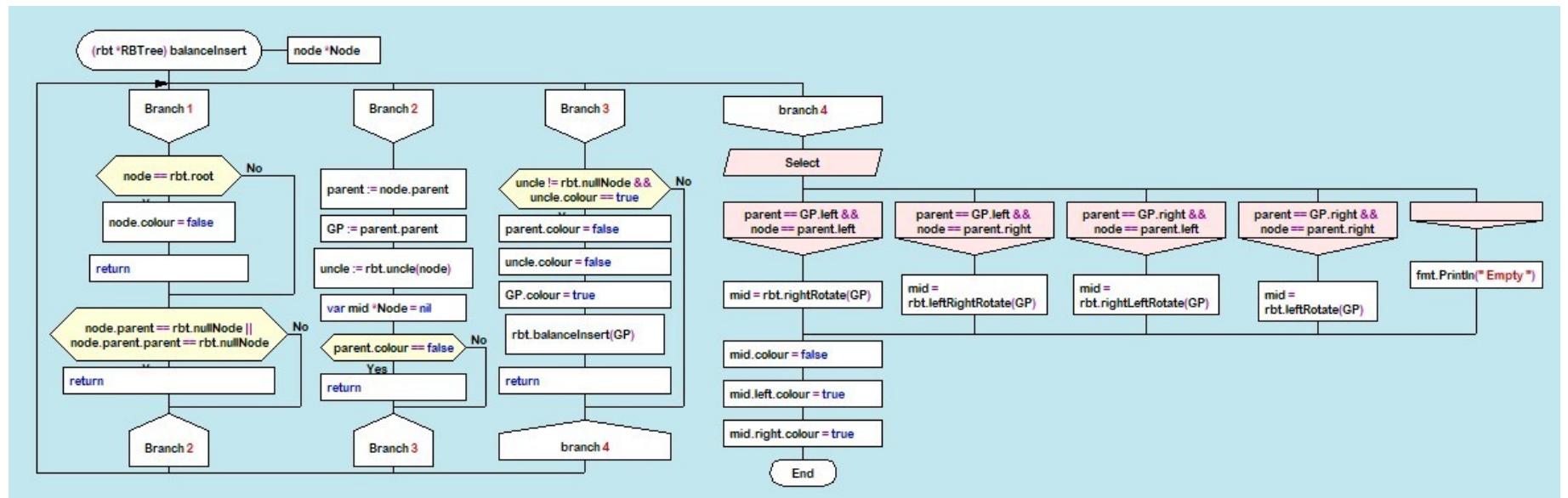


Рис. 8.36. Дракон-диаграмма балансировка красно-черных деревьев

<code>parent == GP.left && node == parent.left</code>	Left child of the left child — right rotation.
<code>parent == GP.left && node == parent.right</code>	Right child of the left child — left-right ro-

	tation.
parent == GP.right && node == parent.left	Left child of the right child — right-left rotation.
parent == GP.right && node == parent.right	Right child of the right child — left rotation.

Дракон-диаграммы поворотов узлов во время балансировки красно-черного дерева при вставке нового узла показаны на рисунке 8.37.

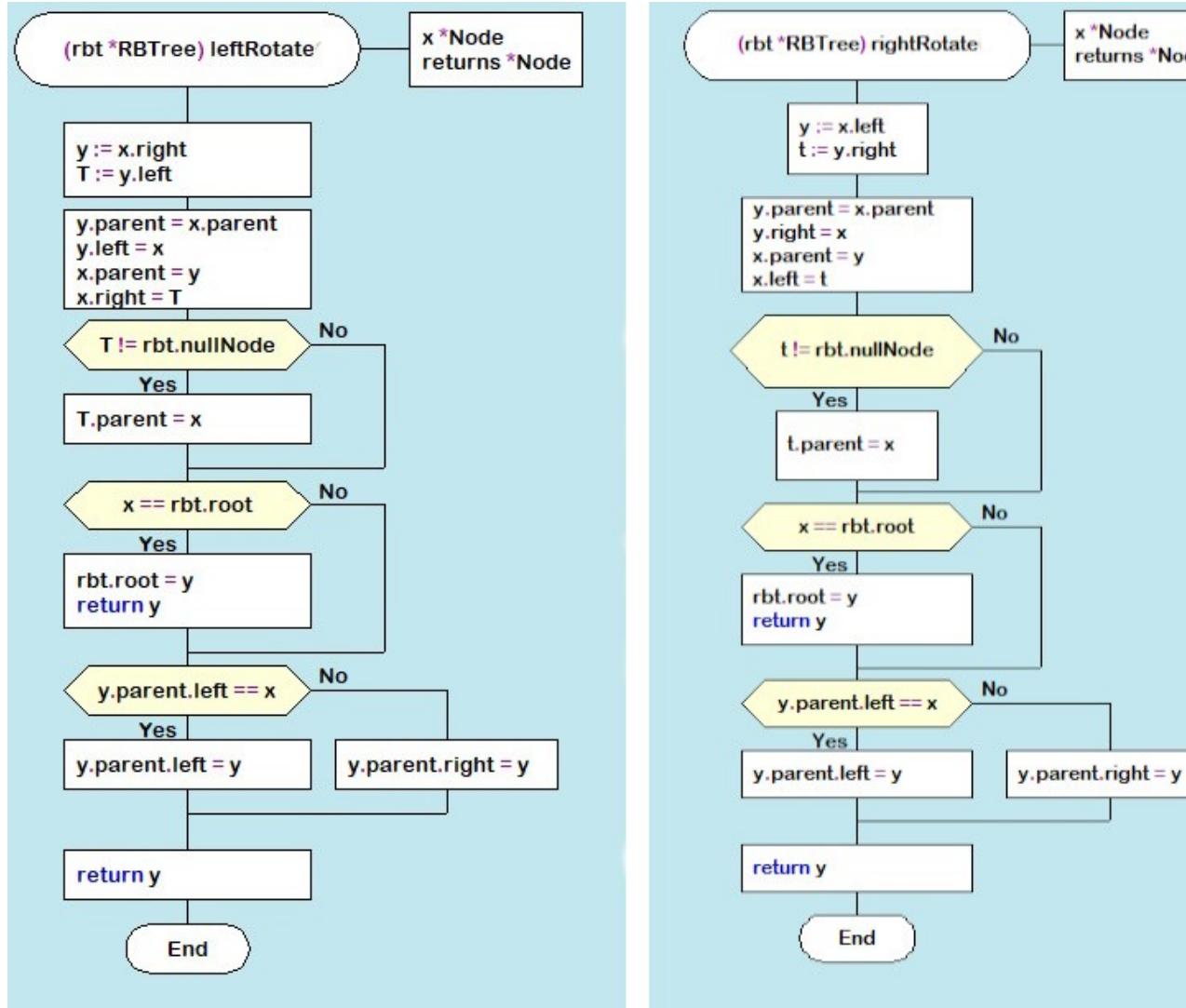


Рис. 8.37. Методы поворота в балансировке `leftRotate` и `rightRotate`

Алгоритм *rightRotation* в процессе балансировки при вставке нового узла аналогичен алгоритму *LeftRotate*.

Важно отметить, что структура красно-черного дерева может варьироваться не только в зависимости от последовательности вставленных элементов, но и от конкретного используемого алгоритма реализации. Несмотря на возможные изменения в структуре дерева, все красно-черные деревья, построенные на основе одного и того же набора данных, будут

поддерживать одну и ту же «черную высоту» (количество черных узлов от корня до любого листа), что является фундаментальной характеристикой красно-черных деревьев. Это свойство обеспечивает сбалансированность дерева, тем самым оптимизируя эффективность операций поиска, вставки и удаления (рис. 8.38):

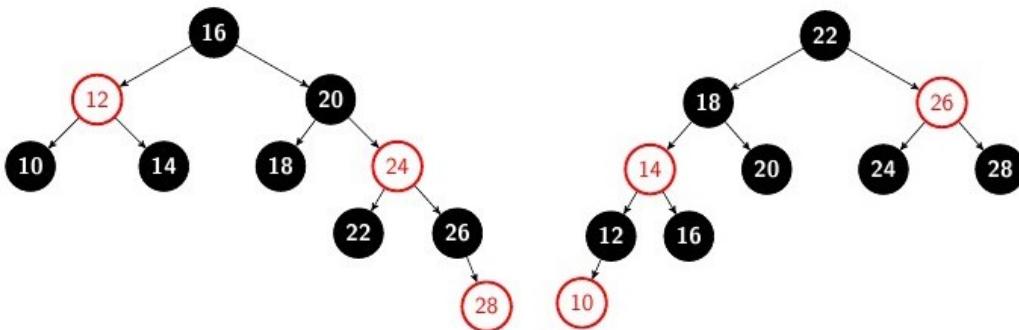


Рис. 8.38. Красно-черные деревья, построенные на одном массиве

*a) [10, 12, 14, 16, 18, 20, 22, 24, 26]; b)
[26, 24, 22, 20, 18, 16, 14, 12, 10]*

8.4.3. Удаление узла из красно-черного дерева

Удаление узла из красно-черного дерева является сложным процессом, поскольку он зависит от расположения узла, наличия дочерних элементов и цвета узлов. Важно помнить, что все преобразования древовидной структуры должны учитывать его свойства. Дракон-диаграмма алгоритма удаления узлов показана на рисунке 8.39. Этот алгоритм можно разделить на три этапа.

Первым делом необходимо удалить назначенный узел. Алгоритм проходит по узлам красно-черного дерева, перемещаясь

влево или вправо, в зависимости от того, больше или меньше заданного значения ключ текущего узла. Когда узел с ключом *ключа* найден, он сохраняется в переменной *z*. Затем указанный узел удаляется из своего исходного положения в дереве.

Вторым этапом является фиксация двойного черного узла. Если удаляемый узел был черным, это может нарушить свойства красно-черного дерева, так как все пути от корня к листьям должны содержать одинаковое количество черных узлов. Чтобы исправить это, вводится понятие «двойной черный» узел. Если узел *x*, заменивший удаленный узел, черный, он становится «двойным черным» узлом.

Третий шаг – коррекция двойного черного узла. Для фиксации "двойного черного" узла вызывается функция *balanceDeleteNode (node * Node)* (Рисунок 8.39). Эта функция исправляет «двойной черный узел» с помощью серии вращений и перерисовок. Он рассматривает несколько случаев, в зависимости от окраски брата узла «двойной черный» и его потомков. *balanceDeleteNode (node * Node)* обрабатывает все эти случаи, чтобы гарантировать, что красно-черное дерево остается сбалансированным после каждой операции удаления.

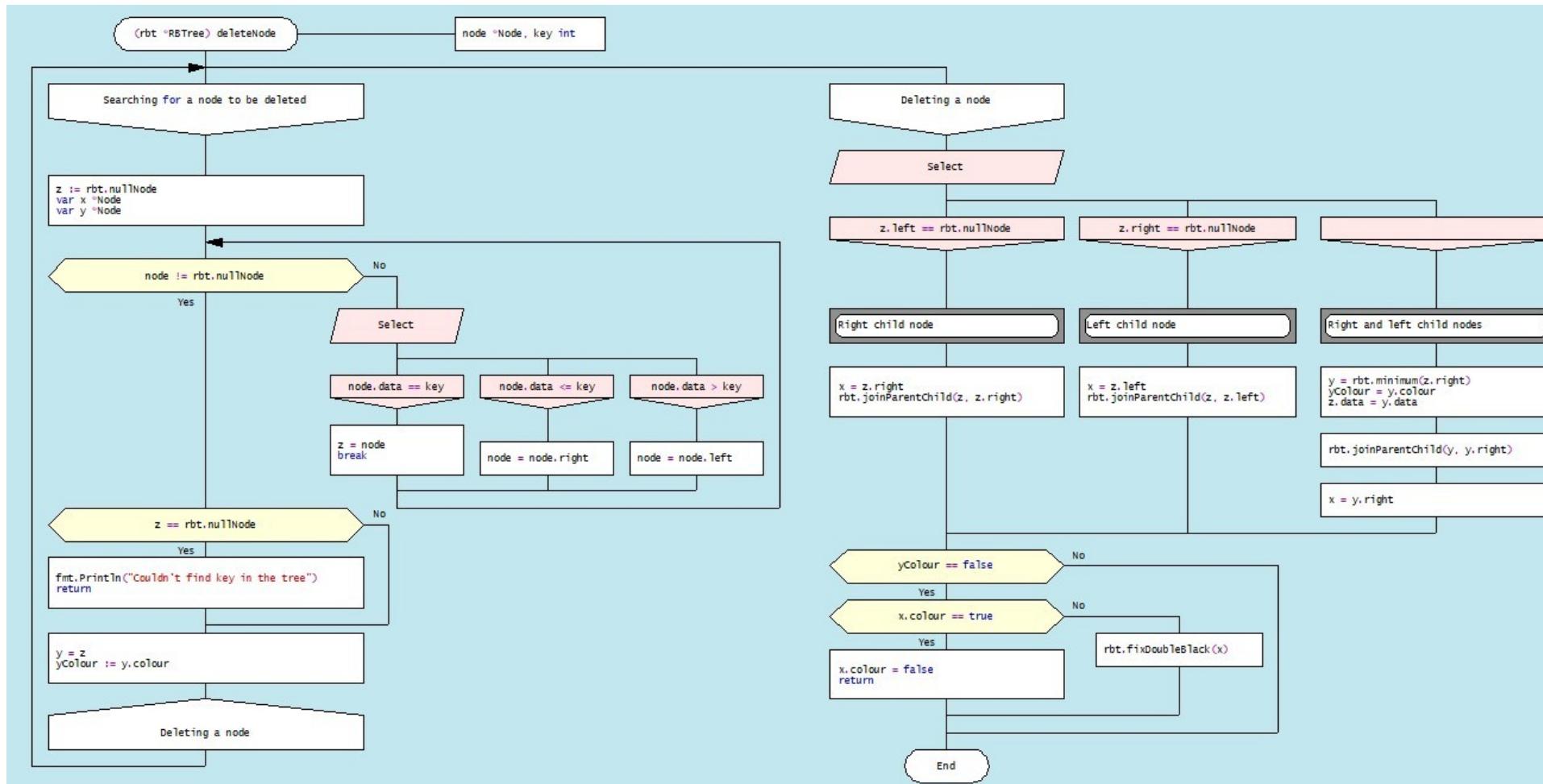


Рис. 8.39. Дракон-диаграмма алгоритма удаления узла из красно-черного дерева

Consider in more detail the different options of the removal node in the structure of the red-black tree. In the first step, the node is removed simply as a node in the binary search tree. If the replacement node is a "red" node, or if the removed node is a "red" node, the removed node is replaced by another node. There is no need to make any further changes to the tree structure (рис. 8.40).

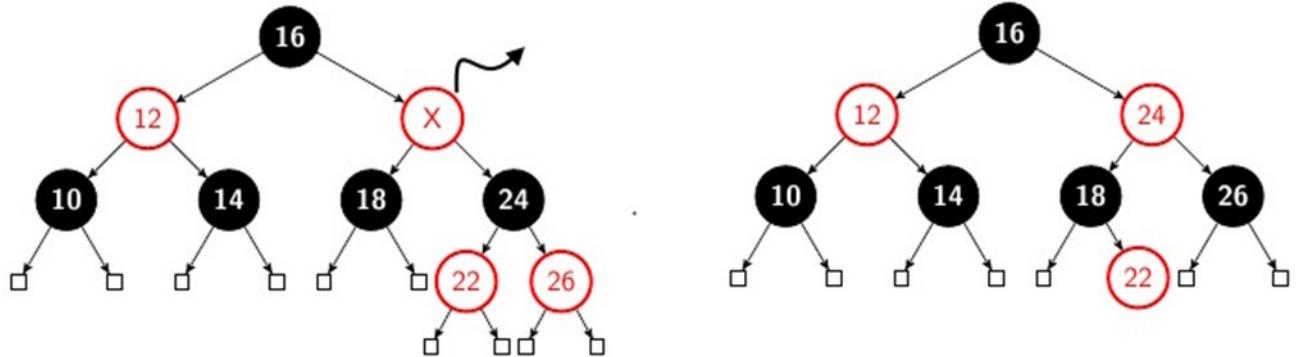


Рис. 8.40. Удаление красного узла (20)

Если удаляемый узел и узел, который его заменяет, являются черными узлами, возникает ситуация, называемая «двойным черным узлом». В этом случае необходимо выполнить некоторые дополнительные операции для того, чтобы свойства красно-черного дерева сохранились. Дракон-иаграмма этого алгоритма показана на рисунке 8.41.

Если заменяющий узел является конечным узлом (т.е. не имеет дочерних узлов), он заменяется на «нулевой» листовой узел и окрашивается в черный цвет. Если у заменяющего узла есть один дочерний узел, он заменяется на дочерний узел и окрашивается в черный цвет. Если у заменяющего узла есть два дочерних узла, то он заменяется на его последующий узел по порядку, а затем удаляется следующий по порядку узел (который представляет собой не более одного узла с одним дочерним узлом) с помощью описанного выше метода.

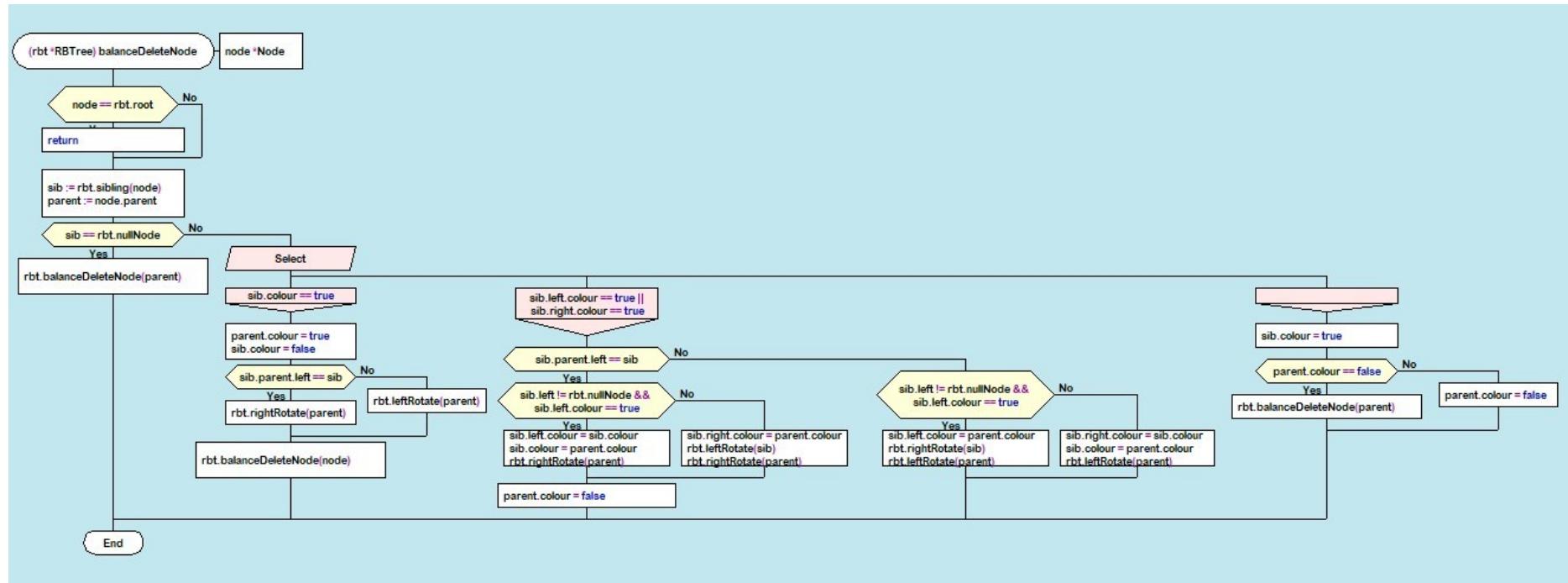


Рис. 8.41. Дракон-диаграмма алгоритма коррекции структуры узла при удалении *balanceDeleteNode(node)*

Рассмотрим варианты восстановления свойств красно-черного дерева. Введите следующие обозначения: удаляемый узел – « x », дочерний узел удаленного узла – « y », а одноуровневый узел удаляемого – « s » (рис. 8.42).

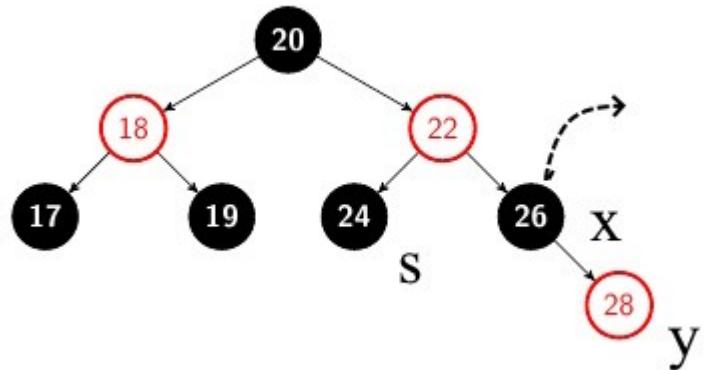


Рис. 8.42. Связь между древовидными узлами

Вариант 1: Если дочерний элемент (y) удаляемого узла (x) окрашен в красный цвет, то после удаления его следует перекрасить в черный цвет, в результате чего количество черных узлов будет восстановлено (рис. 8.43).

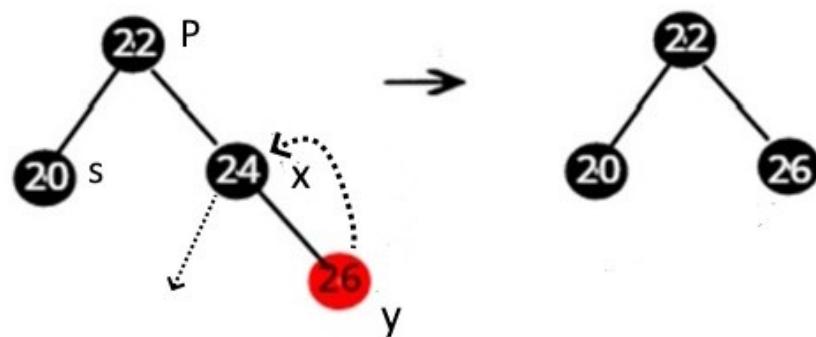


Рис. 8.43. Удаление и перерисовка дочернего элемента удаляемого узла (у)

Вариант 2: Когда родственники (s) удаляемого узла (x) окрашены в черный цвет, и хотя бы один из потомков окрашен в красный, то возможны четыре различные комбинации:

Вариант 2(a): Брат (s) удаляемого узла (x) черного цвета. В этом случае брат (братья) является левым потомком родителя (P), а левый дочерний элемент брата окрашен в красный цвет. Это так называемая «лево-левая конфигурация», при которой баланс достигается за счет вращения вправо, после чего цвет ребенка должен быть заменен на черный (рисунок 8.44).

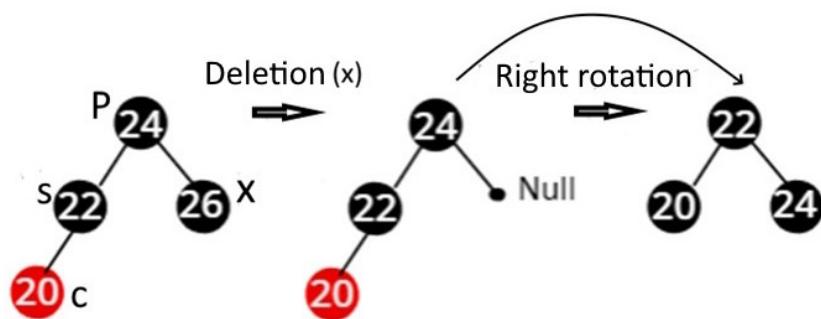


Рис. 8.44. Ллево-левая конфигурация удаления узлов

Вариант 2(b): Зеркальное отражение случая 2(a). Брат удаляемого узла «s» является правым ребёнком родителя, а правый ребёнок узла (s) — красный ребёнок. «Право-правая» конфигурация выполняет вращение влево. Затем дочерний узел заменяется на черный. В результате свойство счетчика черного узла восстанавливается (рис. 8.45).

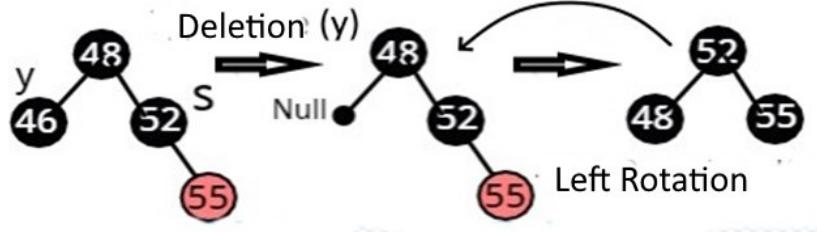


Рис. 8.45. Право-правая конфигурация удвляемого узла

Вариант 2 с). Брат (братья) удаленного узла (у) черного цвета. Брат (братья) является левым потомком своего родителя, а его правый ребенок – красный. Это конфигурация слева направо (лево-правая), поэтому выполняется вращение влево, а затем вправо. Дочерний узел окрашен в черный цвет (рис. 8.46):

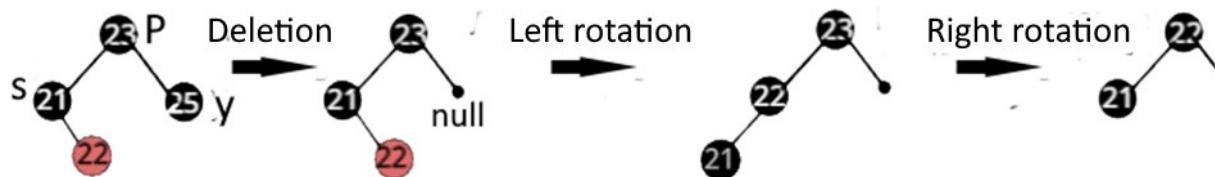


Рис. 8.46. Лево-правая конфигурация удаляемого узла

Вариант 2(d): Брат (братья) удаленного узла (у) окрашен в черный цвет. Брат (братья) является правым ребенком своего родителя (), а его левый ребенок является красным ребенком. Затем выполняется вращение вправо налево, за которым следует вращение влево. Цвет дочернего узла изменится на черный (рис. 8.47).

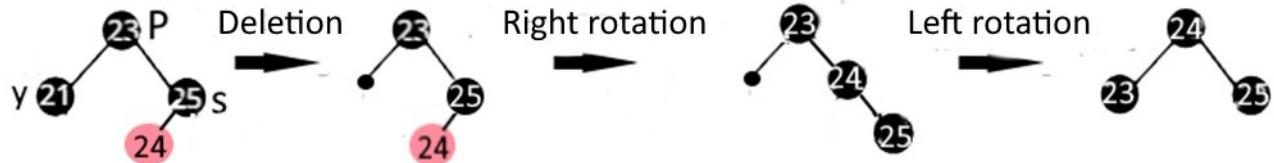


Рис. 8.47 Право-левая конфигурация удаляемого узла

Вариант 3: Если удаляемый узел (у) и его брат (братья) окрашены в черный цвет и оба его потомка отсутствуют, т.е. черные по определению, то нужно перекрасить дочерние элементы в красный цвет и рекурсивно добавить черный цвет родительскому. Если родитель был красным, то он становился черным. Если родитель был черным, он становился двойным черным. Если родителем является корень, то он остается черным. Например, ниже приведен случай, когда удаляемый узел (у) и его брат (братья) черные (рис. 8.48):

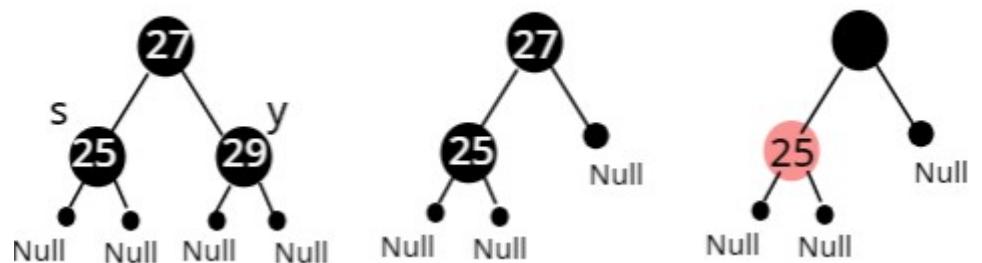


Рис. 8.48 Процесс перестроения дерева с двумя черными узлами (у) и (s)

Вариант 4: Если родственник (s) удаляемого узла (у) окрашены в красный цвет, то выполняется вращение для достижения баланса дерева.

Затем сестринские узлы перерисовываются. Например, рассмотрим следующий вариант (рисунок 8.49):

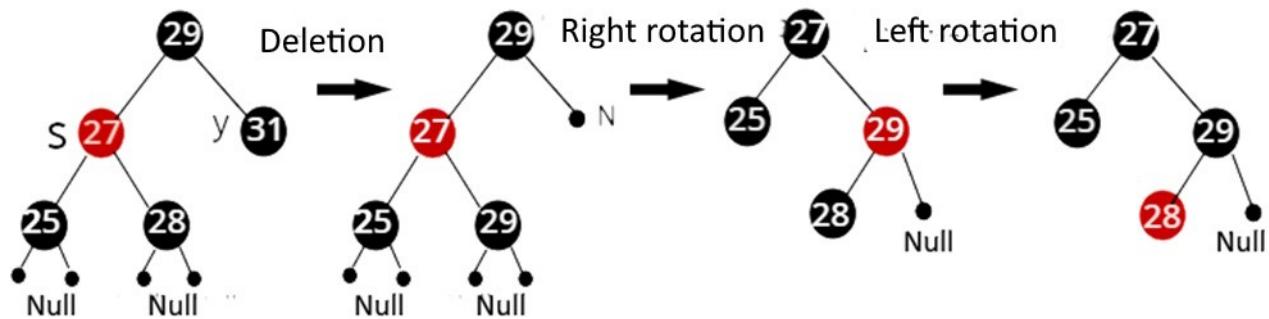


Рис. 8.49. Процесс перестроения дерева с двумя узлами (у) и (s) разных цветов

Общий вид программы, включающей все драконовские диаграммы реализации алгоритмов основных и вспомогательных функций, представлен на рисунке 8.50:

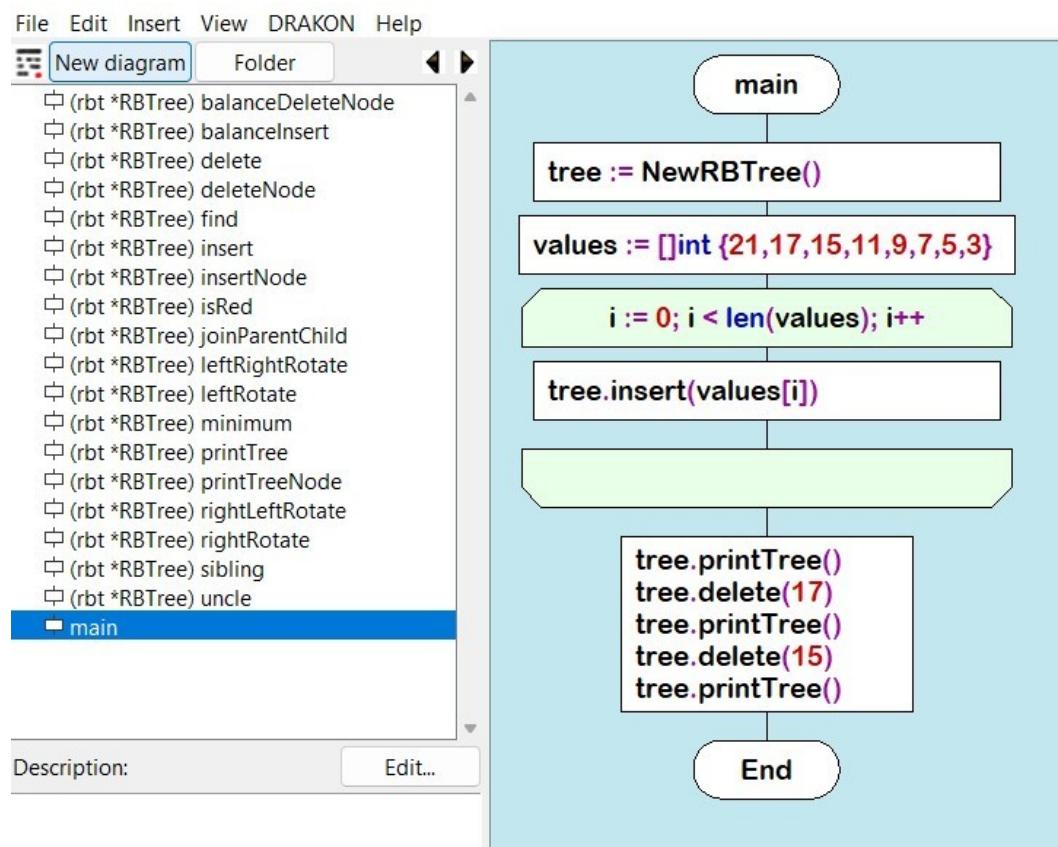


Рис. 8.50. Общий вид программы для реализации основных функций красно-черного дерева

Описания типов, используемых в программе, показаны на рисунке 8.51:

```
==== header ====
package main

import "fmt"

type RBTree struct {
    root      *Node
    nullNode *Node
}

func NewRBTree() (rbt *RBTree) {
    rbt = &RBTree{}
    rbt.nullNode = NewNode(0, nil)
    rbt.nullNode.colour = false
    rbt.root = rbt.nullNode
    return
}

type Node struct {
    data          int
    colour        bool |
    left, right, parent *Node
}
==== footer ====
```

Рис. 8.51. Описание типов переменных в программе *rbtAll.drn*

Дракон-диаграммы сопутствующих алгоритмов реализации основных функций красно-черных деревьев представлены ниже:

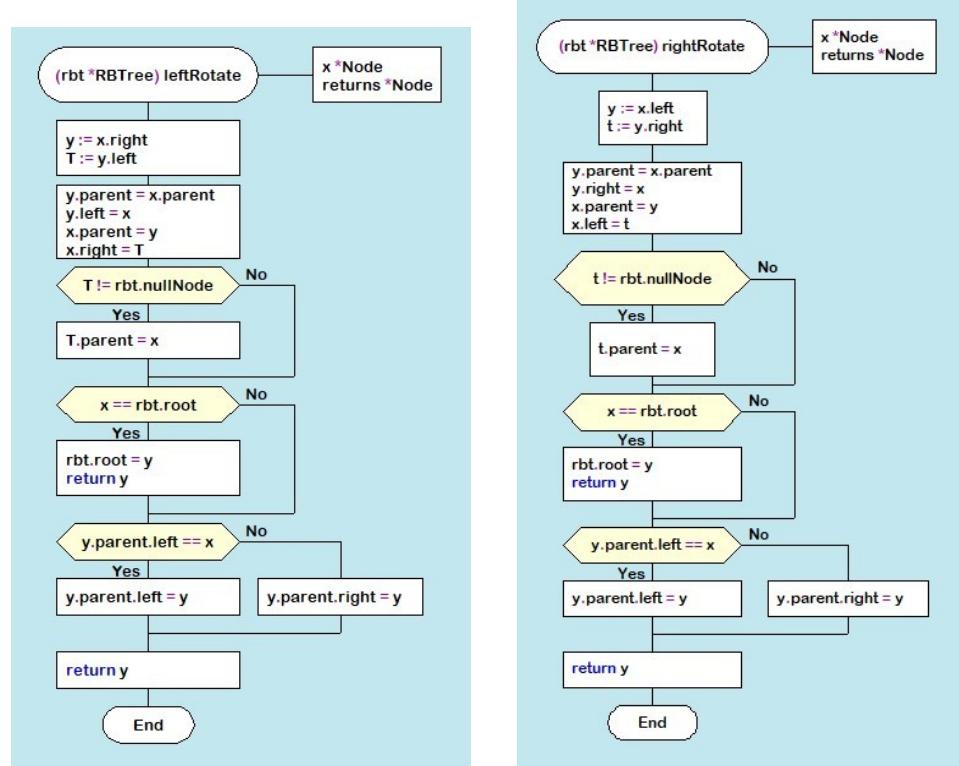


Рис. 8.52. Дракон-диаграммы алгоритмов вращения красно-черных узлов

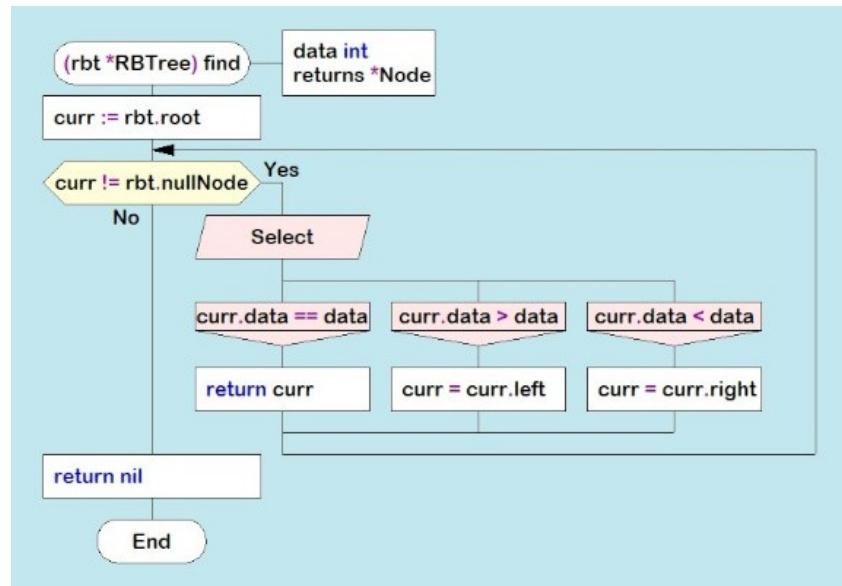


Рис. 8.53. Дракон-диаграмма алгоритма нахождения узла `find(node)`

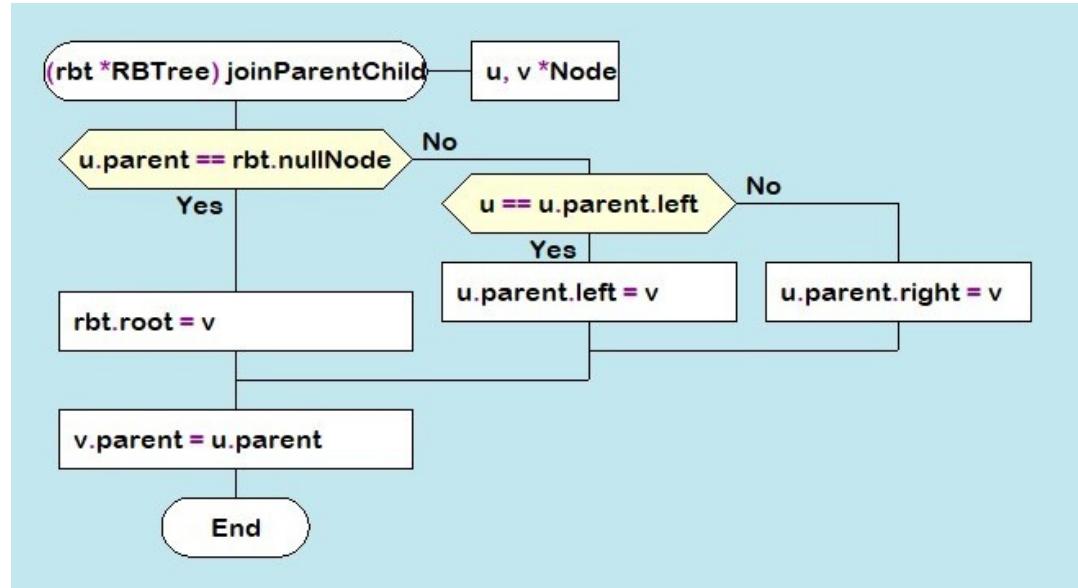


Рис. 8.54. Дракон-диаграмма алгоритма корректного слияния двух узлов

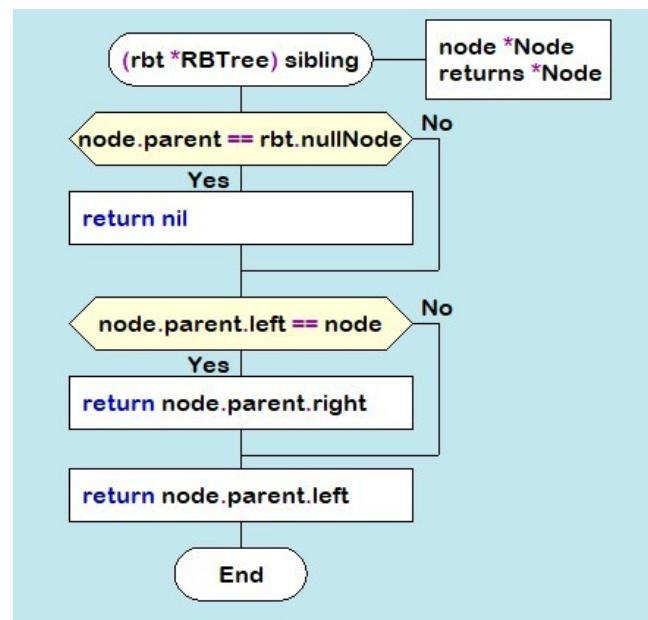


Рис. 8.55. Дракон-диаграмма нахождения родственного узла в красно-черном дереве

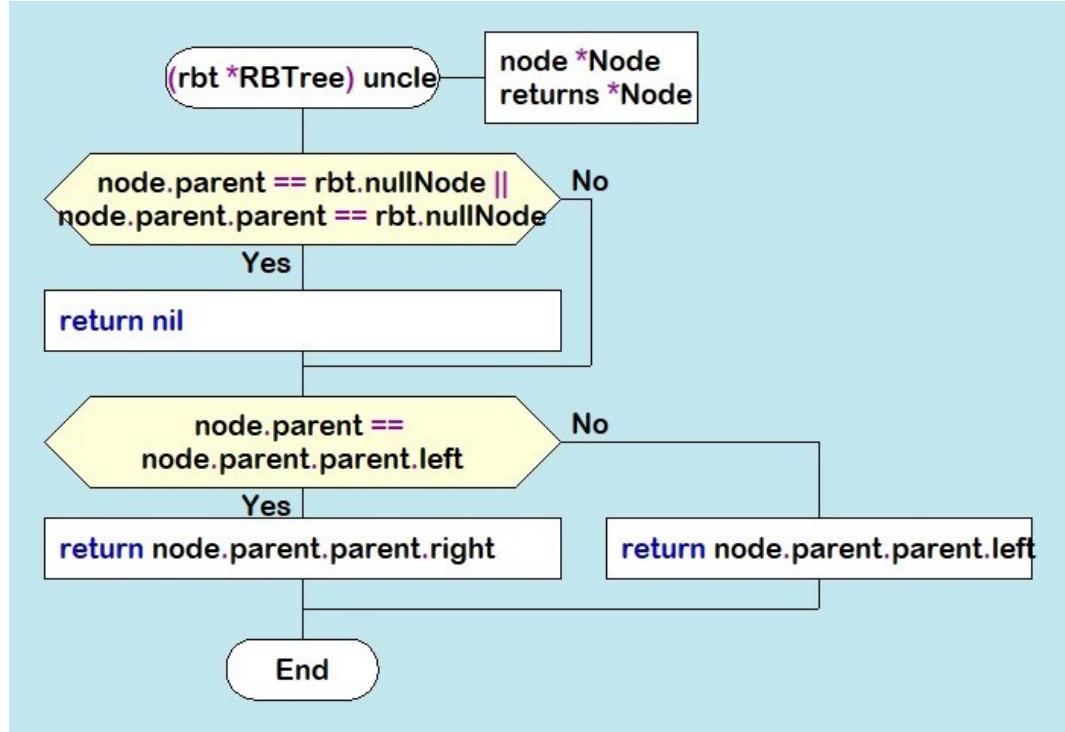


Рис. 8.56 Дракон-диаграмма нахождения алгоритма uncle-узла заданного узла

8.5. Пространственно-временная сложность различных пород деревьев

Пространственная и временная сложность операций поиска, вставки и удаления в BSTree зависит от уровня, на котором сбалансировано дерево. Рассмотрим оценки сложности основных операций поиска, вставки и удаления для различных типов деревьев (BSTree, AVL-дерево, красно-черное дерево).

BSTree

Пространственная и временная сложность операций поиска, вставки и удаления в этих деревьях во многом зависит от уровня их

сбалансированности. Для BST сложность пространства для операций поиска, вставки и удаления в наилучшем случае равна $O(1)$, так как эти операции не требуют дополнительного пространства, которое масштабируется в зависимости от количества узлов в дереве. В худшем случае сложность пространства равна $O(n)$, что происходит, когда дерево сильно разбалансировано, что приводит к линейной структуре, похожей на односвязный список. Сложность пространства в среднем случае для сбалансированной BST равна $O(\log n)$, где n представляет количество узлов в дереве.

Временную сложность для операций поиска, вставки и удаления в BST можно резюмировать следующим образом: Наилучшая времененная сложность для сбалансированной BST равна $O(\log n)$. Наихудшая времененная сложность для несбалансированной BST равна $O(n)$. Средневременная сложность для операций поиска, вставки и удаления в сбалансированном BST также равна $O(\log n)$.

AVL-дерево

Для дерева AVL это можно резюмировать следующим образом:

Сложность пространства для операций поиска, вставки и удаления в наилучшем случае равна $O(\log n)$, так как эти операции не требуют дополнительного пространства, которое масштабируется в зависимости от количества узлов в дереве. В худшем случае сложность пространства равна $O(n)$, что происходит, когда дерево сильно разбалансировано, что приводит к линейной структуре, похожей на односвязный список. Сложность пространства в среднем случае для сбалансированного AVL-дерева равна $O(\log n)$, где n представляет количество узлов в дереве.

Временную сложность операций поиска, вставки и удаления в AVL-дереве можно резюмировать следующим образом: наилучшая времененная сложность для сбалансированного AVL-дерева равна $O(\log n)$. Наихудшая времененная сложность для несбалансированной BST равна $O(n)$. Средневременная сложность операций поиска, вставки и удаления в сбалансированном AVL-дереве также равна $O(\log n)$

Красно-черное дерево

Для дерева AVL это можно резюмировать следующим образом:

Наилучшая сложность пространства для операций поиска, вставки и удаления составляет $O(1)$, так как эти операции не требуют дополнительного пространства, которое масштабируется в зависимости от количества узлов в дереве. В худшем случае сложность пространства равна $O(\log n)$, что происходит, когда дерево сильно разбалансировано, что приводит к линейной структуре, похожей на односвязный список. Сложность пространства в среднем случае для сбалансированного красно-черного дерева равна $O(\log n)$.

Временную сложность для операций поиска, вставки и удаления в красно-черном дереве можно резюмировать следующим образом: наилучшая времененная сложность для сбалансированного красно-черного дерева равна $O(\log n)$. Наихудшая времененная сложность для несбалансированной BST равна $O(n)$. Средневременная сложность для операций поиска, вставки и удаления в сбалансированном AVL-дереве также равна $O(\log n)$.

РАЗДЕЛ 9. ГРАФЫ

9.1. Общие сведения. Базовая терминология

В этом разделе рассматриваются такие структуры данных как *графы*. Граф в математике представляет собой структуру, состоящую из вершин (узлов) и рёбер (связей), которые соединяют эти вершины. Вообще говоря, графы имеют много общего с *деревьями*, - можно сказать, что деревья являются частным случаем графа. Однако их практическая ценность при решении многих реальных задач намного выше. Многие задачи сводятся к рассмотрению совокупности объектов, свойства которых описываются связями между ними. В числе таких объектов – электрические и электронные схемы, печатные платы, карты дорог, авиационные маршруты, описание конструкций, игры, а в числе задач - поиск кратчайшего пути из одной вершины до другой, решение задачи максимальной пропускной способности трубопровода или дорожной сети или компьютерной сети, распределение N работников для выполнения M различных типов работ, выбор наиболее эффективного метода решения задачи и т.д. [Задачи, которые можно решить с помощью графов | Вики справка [Graph Online](#)]

Более строго, граф G задан множеством вершин $\{V\}$ и множеством рёбер $\{E\}$, соединяющих все или часть этих вершин. Таким образом, граф G полностью определяется как $\{V, E\}$. Если рёбра ориентированы, то они называются дугами, а граф с такими рёбрами называется ориентированным графом (рисунок 9.1 а). Если рёбра не имеют ориентации, то граф называется неориентированным:

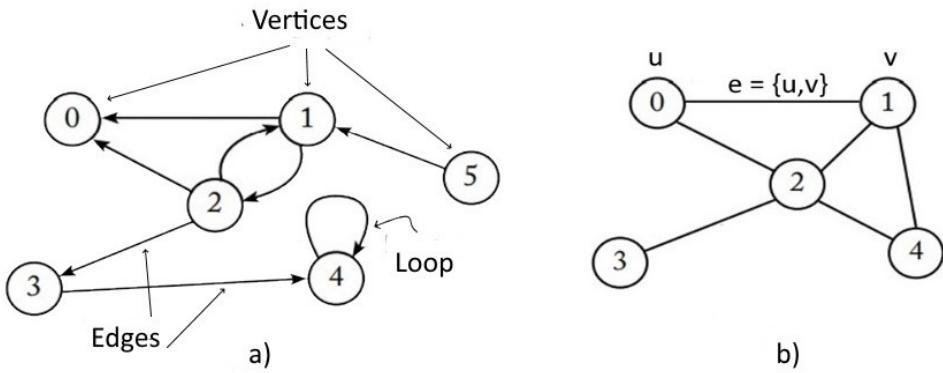


Рис. 9.1 Виды графов а) - неориентированный; б) – направленный

Вершины и рёбра называются элементами графа, число вершин в графе является порядком, число рёбер - размером графа. Вершины (u, v) называются конечными точками $e = \{u, v\}$, а две конечные точки одного ребра называются смежными. Рёбра называются смежными, если они имеют общую конечную точку. Рёбра называются кратными, если наборы конечных точек одинаковы. Ребро называется петлей, если его концы совпадают, то есть $e = \{u, u\}$. Если вершина является началом или концом ребра, то они (вершина и ребро) инцидентны. Число рёбер, инцидентных вершине, называется степенью вершины (рис. 9.2).



Рис. 9.2. Основные параметры графа

9.2. Методы представления графов

Решение задач, связанных с обработкой совокупности данных, организованных в виде графов, требует их моделирования в компьютерных программах. В принципе совокупность вершин можно хранить в массиве и обращаться к ним по индексу. Хранение вершин также можно организовать с помощью односвязного списка или другой структуры данных. На практике для моделирования структуры графа обычно применяются две структуры: *матрица смежности* и *список смежности*. Смежными в том смысле, что такие вершины соединены одним ребром.

Матрица смежности представляет собой двумерный массив, элементы которого обозначают наличие связи между двумя вершинами. Если граф содержит V вершин, то матрица смежности представляет собой массив $V \times V$. На рис. 9.3. показан неориентированный и ориентированный графы и

соответствующие матрицы смежности. В частности, для ориентированного графа матрица смежности строится таким образом, что 1 обозначает наличие связи между вершинами, 0 – ее отсутствие. Так, для вершины (2) смежными вершинами являются вершины (1) и (3), а для вершины (5) – вершина (2).

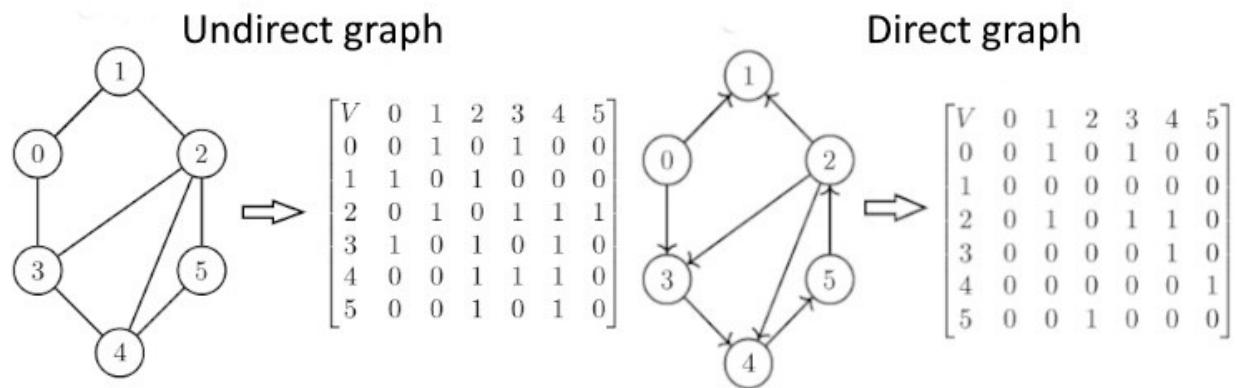


Рис. 9.3. Представление графа в матрице смежности

Представление графа в матрице смежностей связано с определенными проблемами. Прежде всего при построении матрицы нужно заранее знать количество вершин в графе, что приводит к необходимости каждый раз строить новую матрицу при внесении новых вершин. Кроме того, матрица смежностей состоит в основном из нулей, что приводит к неэффективному использованию памяти, если граф содержит V вершин, то должна быть отведена память для V^2 элементов.

Во многих случаях более эффективным способом представления графа является использование списка смежностей. Неориентированный и ориентированный графы и их списки смежности изображены на рисунке 9.4.

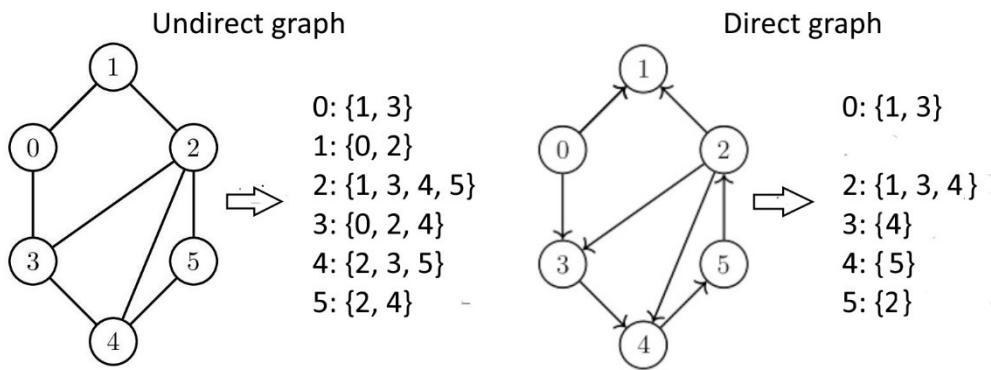


Рис. 9.4. Представление графа в списке смежностей

Матрица и список смежности для ненаправленного графа отличаются от направленного тем, что метки ‘1’ в матрице проставляются для всех смежных вершин (рис.9.4.).

В большинстве практических случаев задачи обработки графов содержат значения, которые связаны с либо с вершинами графа, либо с его ребрами. Например, в задаче поиска оптимального пути между двумя пунктами, ребра графа нужно нагрузить такими данными как расстояние между этими вершинами и стоимостью проезда за 1 км (Рис. 9.5):

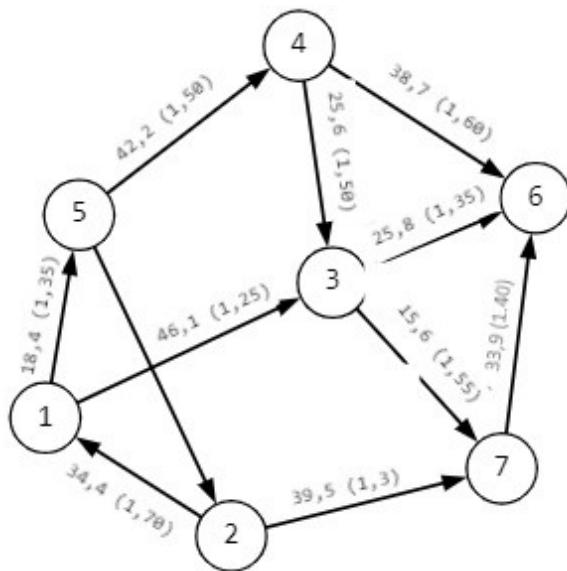


Рис. 9.5. Нагруженный граф

9.3. Интерфейсы графов

Интерфейс графа как структуры данных представляет собой набор методов, которые определяют базовые операции, доступные для работы с графом. Этот интерфейс включает в себя методы для получения количества вершин и рёбер в графе, доступа к конкретному ребру между вершинами, добавления и удаления вершин и рёбер, а также методы для работы с итераторами, позволяющие получить информацию о входящих и исходящих рёбрах для конкретной вершины. Интерфейс графа позволяет работать с графовыми структурами данных и реализовывать их в соответствии с требованиями конкретной задачи.

В языке Golang интерфейс графа может быть реализован с использованием структур и методов, содержание которых определяется конкретной задачей и выбором базовых структур данных, из которых будут создаваться новые типы. В простейшем случае для работы с графиками требуется включение в структуру графа полей для отображения вершин и/или ребер. Для создания типа *Graph* с поддержкой представления графа с помощью списка смежностей используют различные базовые структуры языка Golang (срезы, элементами которых являются односвязные списки (Linked List), карты (map), двумерный срез [] []).

Покажем примеры построения ориентированного графа, приведенного на рис. 9.6., с помощью указанных выше базовых структур (Табл. 9.1.).

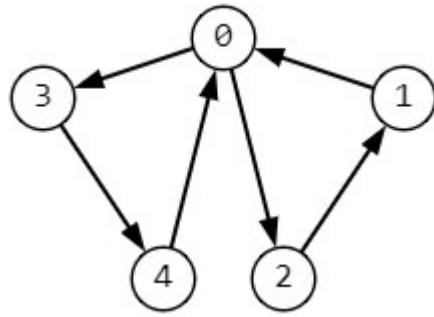


Рис. 9.6. Ориентированный граф для иллюстрации интерфейса

При выборе способа реализации интерфейса графа средствами языка Golang следует руководствоваться следующими критериями:

1. Linked List:

- Используется, если необходимо хранить связи между вершинами графа в виде списка смежности.
- Подходит для реализации графов с большим количеством ребер и небольшим количеством вершин.
- Позволяет эффективно добавлять и удалять ребра между вершинами.

2. Map:

- Используется, если необходимо хранить связи между вершинами графа в виде словаря, где ключами являются вершины, а значениями - их соседи.
- Подходит для реализации графов с произвольным количеством вершин и ребер.

- Позволяет быстро получать список соседей для заданной вершины.

3. Slice:

- Используется, если необходимо хранить связи между вершинами графа в виде срезов, где индексы срезов соответствуют вершинам, а значения - их соседи.

- Подходит для реализации графов с фиксированным количеством вершин.

- Позволяет эффективно получать список соседей для заданной вершины.

Выбор конкретного способа реализации зависит от конкретных требований к графу, таких как количество вершин и ребер, операции, которые чаще всего будут выполняться (добавление вершин, добавление ребер, поиск соседей и т.д.), а также от особенностей самой задачи, в которой будет использоваться граф.

Табл. 9.1. Базовые структуры создания интерфейсов графа

Linked List	map	Slice
<pre>package main import "fmt" type Node struct { value int next *Node } type Graph struct { vertices []*Node } func (g *Graph) addVertex(value int) { newNode := &Node{value: value} g.vertices = append(g.vertices, newNode) } func (g *Graph) addEdge(v1, v2 int) { node1 := g.vertices[v1] node2 := g.vertices[v2] newNode := &Node{value: v2} newNode.next = node1.next }</pre>	<pre>package main import "fmt" type Graph struct { vertices map[int][]int } func (g *Graph) addVertex(vertex int) { g.vertices[vertex] = []int{} } func (g *Graph) addEdge(v1, v2 int) { g.vertices[v1] = append(g.vertices[v1], v2) g.vertices[v2] = append(g.vertices[v2], v1) } func (g *Graph) getNeighbors(vertex int) { fmt.Println(g.vertices[vertex]) } func main() { graph := NewGraph() graph.addVertex() graph.addVertex() graph.addVertex() graph.addEdge(0, 1) graph.addEdge(1, 2) graph.getNeighbors(0) }</pre>	<pre>package main import "fmt" type Graph struct { vertices [][]int } func (g *Graph) addVertex() { g.vertices = append(g.vertices, []int{}) } func (g *Graph) addEdge(v1, v2 int) { g.vertices[v1] = append(g.vertices[v1], v2) } func (g *Graph) getNeighbors(vertex int) { fmt.Println(g.vertices[vertex]) } func main() { graph := Graph{ vertices: [][]int{{}, {}, {}}, } graph.addVertex() graph.addVertex() graph.addVertex() graph.addEdge(0, 1) graph.addEdge(1, 2) graph.getNeighbors(0) }</pre>

<pre> node1.next = newNode newNode = &Node{value: v1} newNode.next = node2.next node2.next = newNode func (g *Graph) getNeighbors(vertex int) { node := g.vertices[vertex] for node != nil { fmt.Println(node.value) node = node.next } } func main() { graph := Graph{} graph.AddVertex(0) graph.AddVertex(1) graph.AddVertex(2) graph.AddVertex(3) graph.AddVertex(4) } </pre>	<pre> graph.AddVertex(0) graph.AddVertex(1) graph.AddVertex(2) graph.AddVertex(3) graph.AddVertex(4) graph.AddEdge(0, 1) graph.AddEdge(0, 2) graph.AddEdge(0, 3) graph.AddEdge(1, 4) graph.AddEdge(2, 4) graph.getNeighbors(0) } </pre>	<pre> } func main() { graph := Graph{} graph.addVertex() graph.addVertex() graph.addVertex() graph.addVertex() graph.addVertex() graph.addEdge(0, 1) graph.addEdge(0, 2) graph.addEdge(0, 3) graph.addEdge(1, 4) graph.addEdge(2, 4) graph.getNeighbors(0) } </pre>
--	---	---

```
graph.AddEdge(0, 1)
graph.AddEdge(0, 2)
graph.AddEdge(0, 3)
graph.AddEdge(1, 4)
graph.AddEdge(2, 4)
graph.getNeighbors(4)
}
```

9.4. Базовые алгоритмы графов

Вследствие широкого распространения графов существует большое количество алгоритмов их обработки. Среди задач, решаемых в рамках теории графов, таковыми являются:

- Определение графа и его свойства;
- Действия с графиками;
- Маршруты, цепи и циклы, контуры;
- Вычисление характеристик графа;

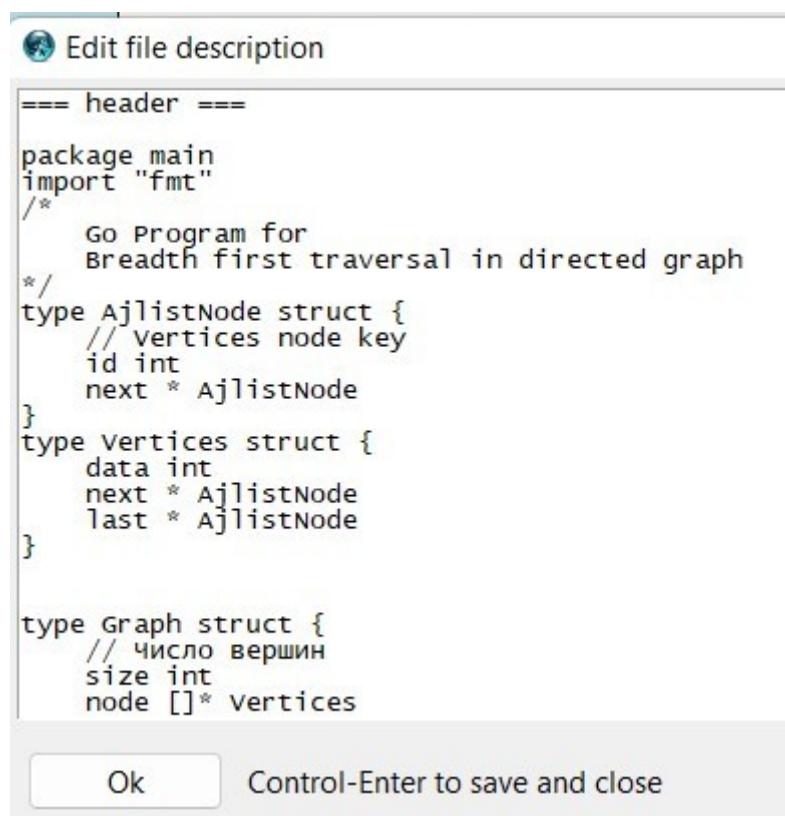
Рассмотрим основные алгоритмы, используемые в этих задачах и реализованные, как и прежде, в рамках гибридного программирования DRAKON + Golang. После того, как график построен и его свойства проверены, естественно может возникнуть задача обхода графа по его вершинам. Общая задача формулируется следующим образом: обойти график, начиная с заданной вершины и двигаясь по ребрам к другим вершинам, посетить все вершины.

Существует два основных способа обхода графов: обход по глубине и обход по ширине, которые обеспечивают перебор всех связных вершин. Разница между поиском в глубину и поиском в ширину заключается в том, что результатом алгоритма поиска в глубину является некоторый маршрут, по которому можно последовательно обойти все вершины графа, доступные из начальной вершины. Это принципиально отличается от поиска в ширину, где вначале обходятся все вершины одного уровня, а затем последовательно обходятся вершины следующих уровней.

9.4.1. Обход в глубину (In depth)

9.4.1.1 Формирование графа

Напомним, что алгоритм поиска в глубину заключается в последовательном обходе вершин графа, доступных из начальной вершины. Для реализации этого алгоритма граф будем представлять с помощью связных списков (Табл. 9.1). Описание типов переменных, используемых в программе обработки графа представлены на рис. 9.7.



The screenshot shows a code editor window titled "Edit file description". The code is written in Go and defines several structures:

```
==== header ====
package main
import "fmt"
/*
   Go Program for
   Breadth first traversal in directed graph
*/
type AjlistNode struct {
    // Vertices node key
    id int
    next * AjlistNode
}
type Vertices struct {
    data int
    next * AjlistNode
    last * AjlistNode
}

type Graph struct {
    // Число вершин
    size int
    node [] * Vertices
}
```

At the bottom of the editor window, there are two buttons: "Ok" and "Control-Enter to save and close".

Рис. 9.7. Описание типов переменных программы обхода в глубину

В этом фрагменте описываются следующие структуры:

1. *AjlistNode* - структура, которая представляет узел в связанном списке для представления смежности графа. Она содержит следующие поля:

- *id int* - целочисленное значение, которое представляет идентификатор вершины графа.

- *next *AjlListNode* - указатель на следующий узел в связанным списке.

2. *Vertices* - структура данных, которая представляет вершину графа в программе обработки графа. Она содержит следующие поля:

- *data int* - это целочисленное значение, которое представляет данные, связанные с вершиной графа.

- *next *AjlListNode* - это указатель на первый узел в связанным списке смежных вершин.

- *last *AjlListNode* - это указатель на последний узел в связанным списке смежных вершин.

3. *Graph* – структура описывает граф, состоящий из узлов типа *Vertices* и имеющий два поля:

- *size* представляет количество узлов в графе (размер графа);

- *node* представляет собой срез указателей на эти узлы.

На рис.9.8. представлена полная Дракон-диаграмма алгоритма обхода графа в глубину:

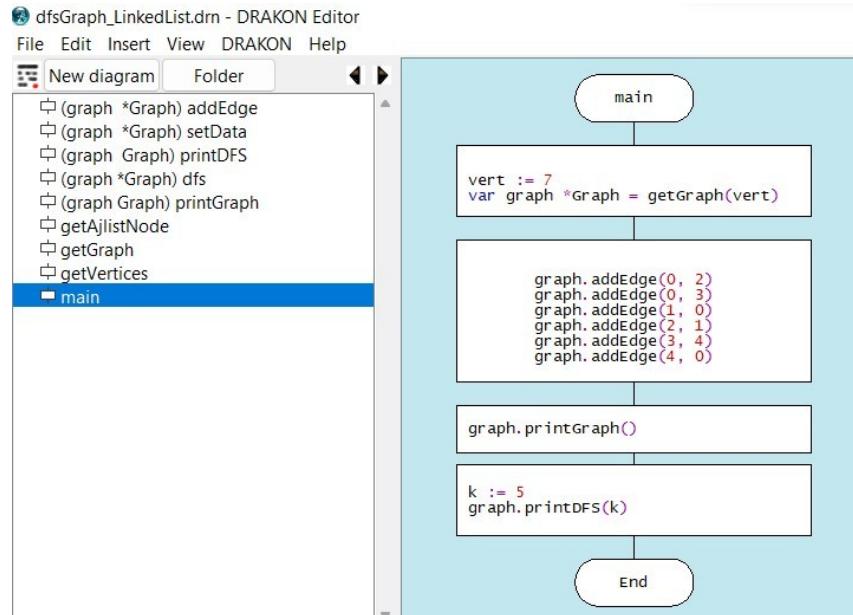


Рис. 9.8. Дракон-диаграмма функции *main()*

В функции *main ()* создается переменная *graph* типа указателя **Graph*, описывающая граф, состоящий из структур данных типа *Vertices*, где *size* представляет количество узлов в графе, а *node* представляет собой срез указателей на эти узлы (Рис. 9.9.).

После создания нового экземпляра графа выполняется, метод *addEdge(start, last)*, который добавляет ребро между узлами *start* и *last*. Для создания узла для списка смежности в методе используется переменная *var edge *AjlListNode*, инициализирующая уникальный идентификатор (*id*) и указатель на следующий узел (*next*).

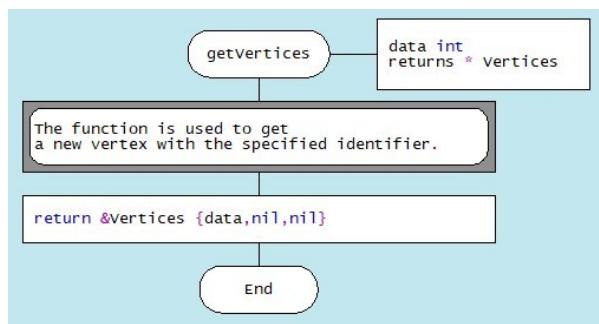
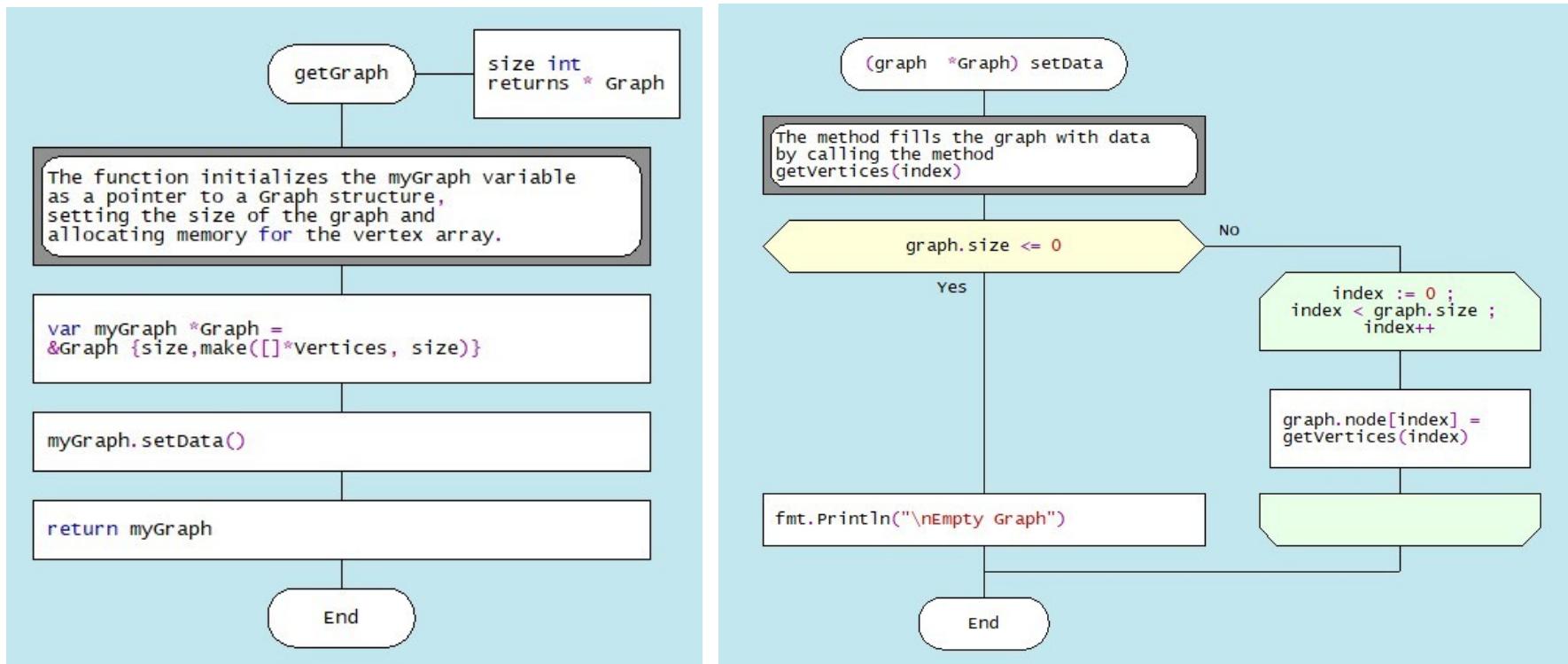
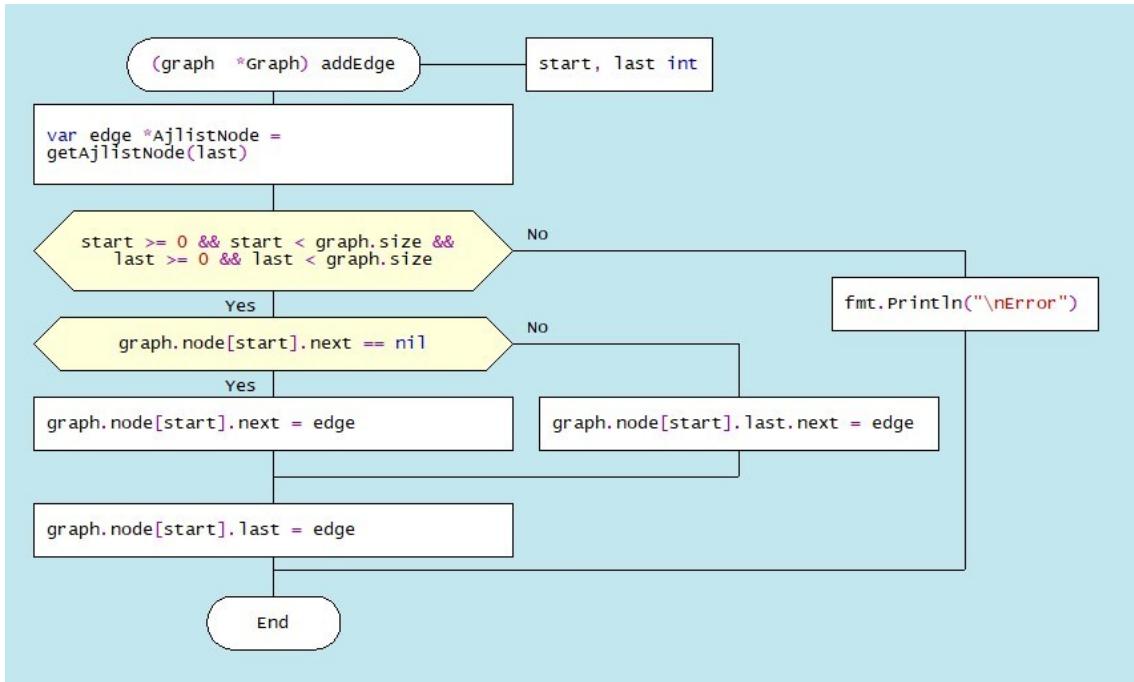
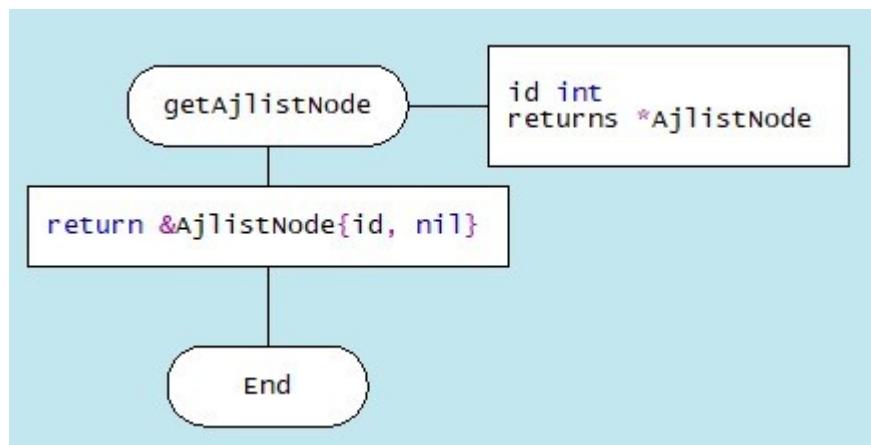


Рис. 9.9. Дракон-диаграммы методов формирования графа



a) метод addEdge((start, last int))



б) метод getAjlistNode (id int)

Рис. 9.10. Дракон-диаграммы заполнения графа данными

Метод printGraph() позволяет вывести список смежности графа, где каждый узел связан со своими соседями (рис. 9.11).

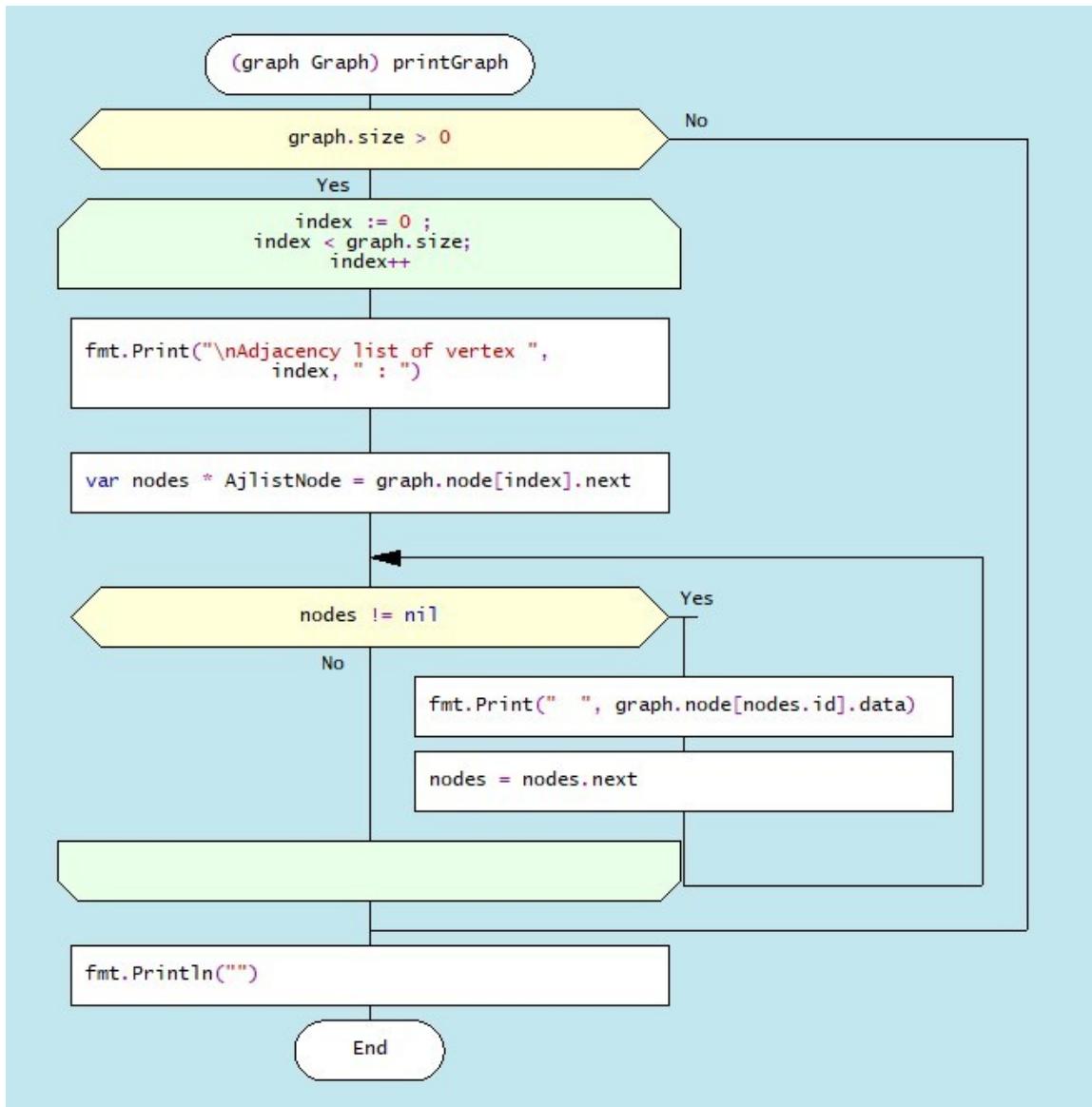


Рис. 9.11. Дракон-диаграмма метода printGraph

9.4.1.2. Алгоритм обхода в глубину (DFS)

Обход графа в глубину является оправданным в ситуациях, когда необходимо исследовать неизвестную структуру реального объекта, представленного в виде графа. Если же граф ориентированный, то поиск в глубину строит дерево путей из начальной вершины во все доступные из нее вершины. Алгоритма обхода графа в глубину можно представить себе следующим образом. Пусть наблюдателю, находящемуся в одной из вершин графа, поставлена задача

обойти все его вершины. Находясь в этой вершине, наблюдатель видит ребра, исходящие из этой вершины. В случае достаточно сложной структуры графа наблюдатель рискует проходить некоторые вершины по нескольку раз и, в конце концов, зациклиться. Для избежания такой ситуации наблюдатель должен отмечать все посещенные вершины и не должен идти в ту вершину, которую он уже посещал. Тогда алгоритм может выглядеть следующим образом:

- Выбрать вершину, с которой начинается обход графа;
- Перейти в любую смежную вершину, не посещенную ранее;
- Запустить из этой вершины алгоритм обхода в глубину;
- Вернуться в начальную вершину;
- Повторить процесс для всех не посещенных ранее смежных вершин.

Таким образом для реализации алгоритма понадобится отмечать, в каких вершинах был исследователь, а в каких — нет. Пометку будем делать в списке `visit`, где `visit[i] == True` для посещенных вершин, и `visit[i] == false` для непосещенных. Пометка «о посещении вершины» ставится при заходе в эту вершину.

Поиск в глубину начинается с посещения исходной вершины `start`, после чего рекурсивно посещаются все смежные вершины. Посещение вершины фиксируется в логическом срезе `visit`. Алгоритм обхода в глубину основан на применении рекурсивной функции, которая извлекает из стека вершину, проверяя ее на посещаемость. Если вершина уже посещалась, обход продолжает поиск по списку смежности до достижения тупиковой вершины. Иллюстрация обхода графа в глубину представлена на рис.9.12.

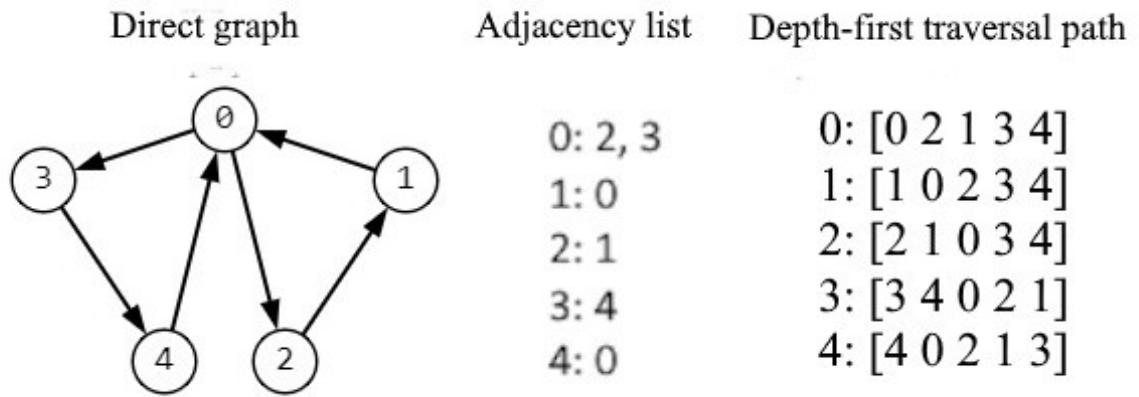


Рис. 9.12. Иллюстрация обхода ориентированного графа в глубину

Алгоритм обхода в глубину в ненаправленных и направленных графах в целом аналогичен, но его выполнение может меняться из-за различий в направленности рёбер и стрелок в графе. Это различие состоит в том, что в ненаправленных графах возможна связь между вершинами в обоих направлениях без явного учёта направления. В направленных графах каждое ребро учитывается только один раз в направлении от начальной вершины к конечной. Дракон-диаграмма алгоритма обхода графа в глубину представлена на рис. 9.13.

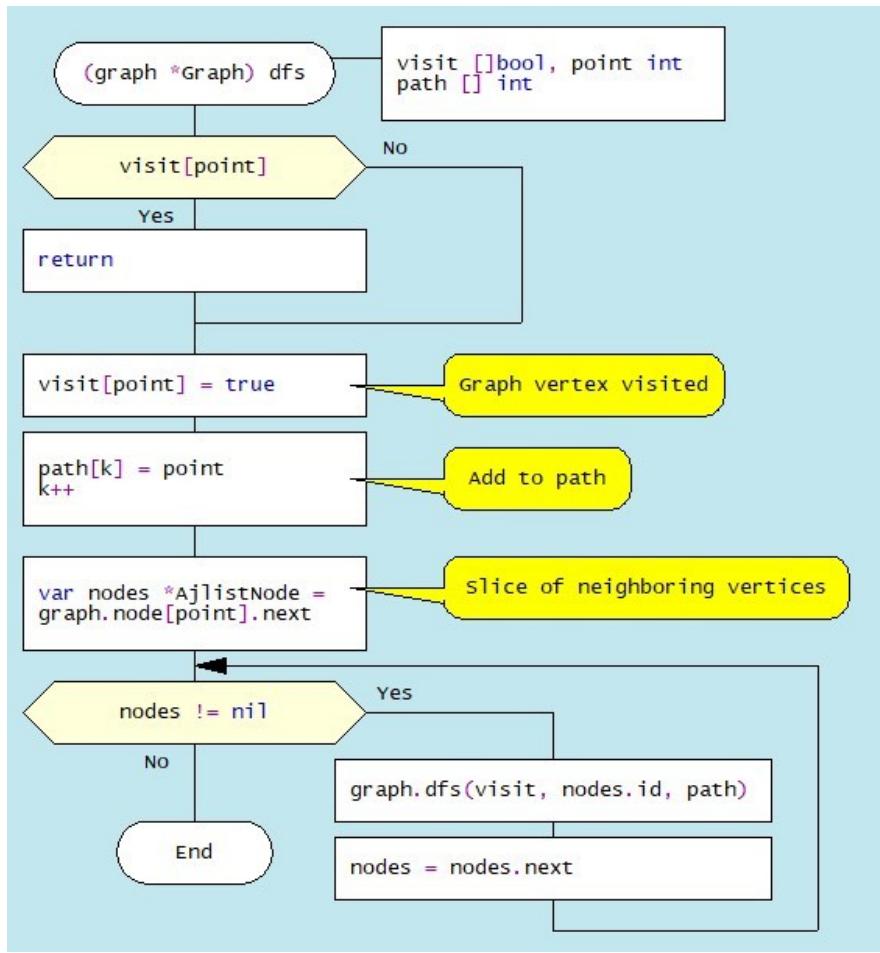


Рис. 9.13. Дракон-диаграмма алгоритма обхода в глубину *dfs*

9.4.2. Обход графа в ширину

Обход графа по ширине широко применяется в различных областях, таких как компьютерные сети, поиск веб-страниц, обработка изображений и других областях, где требуется анализ структуры данных, представляемой в виде графа. При обходе в ширину сначала посещаются все вершины, расположенные на одном уровне, а затем происходит переход к следующему уровню и так далее. Напомним, что уровень графа – это группа вершин, находящихся на одинаковом расстоянии от начальной вершины. В качестве примера рассмотрим алгоритм обхода графа в ширину неориентированного графа (рис. 9.14),

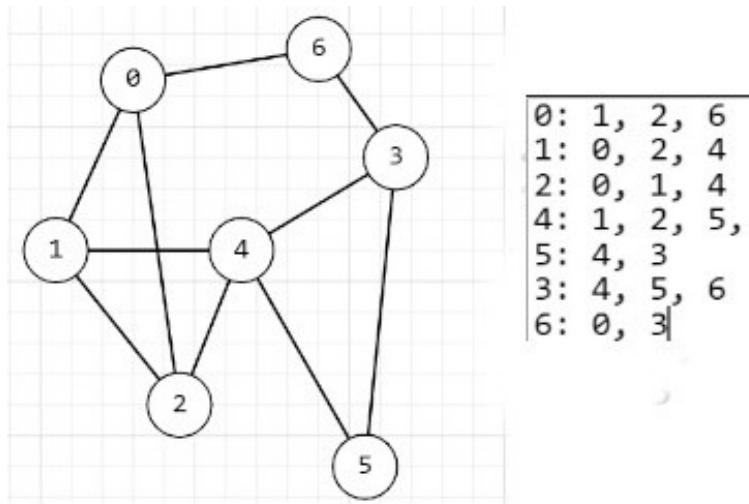


Рис. 9.14. Неориентированный граф и его список смежности

В данном случае, в отличие от предыдущего алгоритма обхода в глубину, для обхода графа в ширину создадим пустую карту *map [int] bool* для отслеживания посещенных вершин и пустой срез *queue []* для добавления посещенных вершин (рис. 9.15).

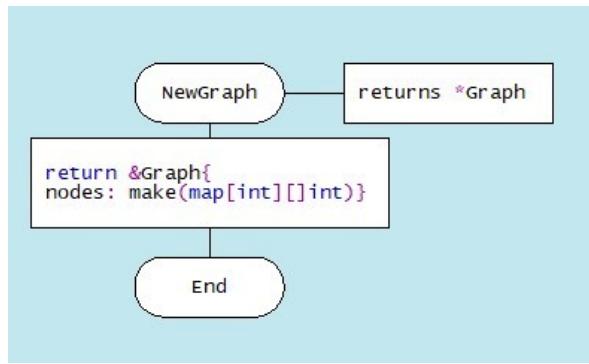
```

==== header ====
package main
import "fmt"
type Graph struct {
    nodes map[int][]int
    size int
}
var path [] int
==== footer ====

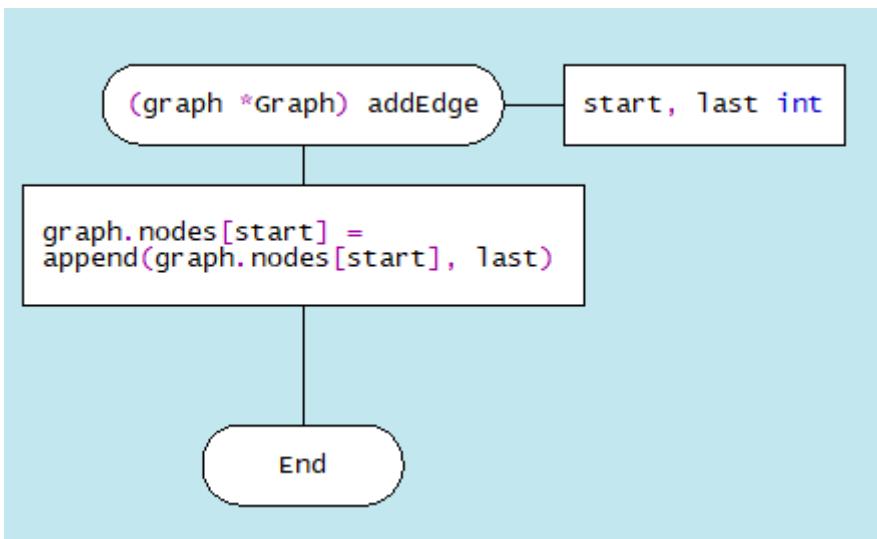
```

Рис. 9.15. Описание типов переменных

Формирование графа осуществляется методами *NewGraph* и *addEdge(start, last)* (рис. 9.16)



a) Метод `newGraph`



b) Метод `addEdge(start, last int)`

Рис. 9.16. Алгоритмы формирования графа

На первом шаге алгоритма начальная вершина (`start`) добавляется в очередь `queue`. На каждом шаге обхода графа по ширине извлекается вершина из начала очереди. Если эта вершина уже посещалась (находится в состоянии `visited`), программа переходит к следующей итерации цикла, прерывая текущую итерацию. Если вершина еще не посещалась, то она выводится и помечается как посещенная. Затем, каждый сосед (`neighbor`) этой вершины, который еще не посещен, добавляется в очередь. Затем обход продолжается до тех пор, пока очередь не станет пустой. Если очередь пуста, алгоритм завершается. Дракон-диаграмма алгоритма обхода графа по ширине представлена на рис. 9.17.

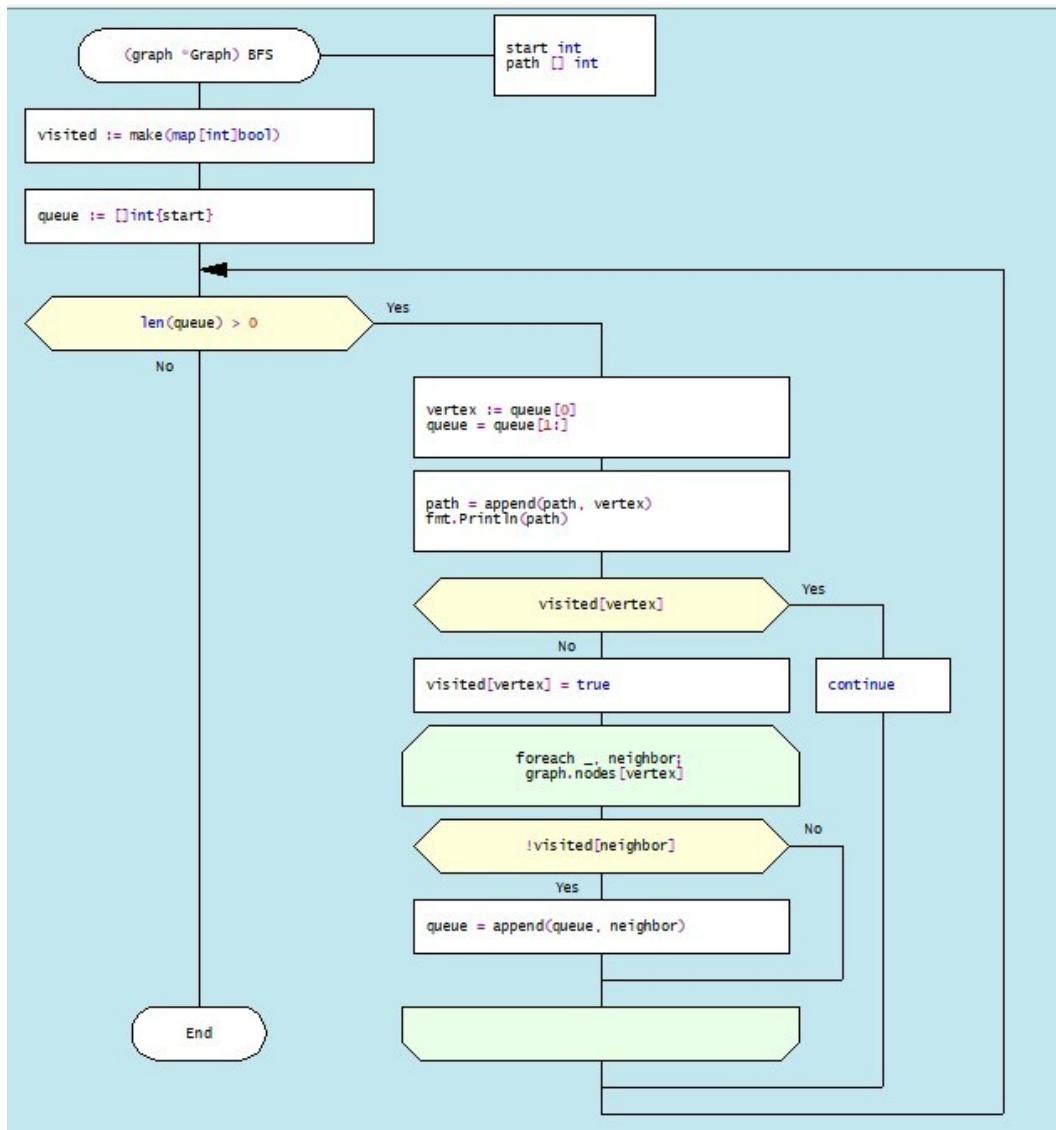


Рис. 9.17. Дракон-диаграмма алгоритма обхода в ширину *BFS*

Результаты обхода направленного графа в глубину и ширину представлены на рис. 9.18.

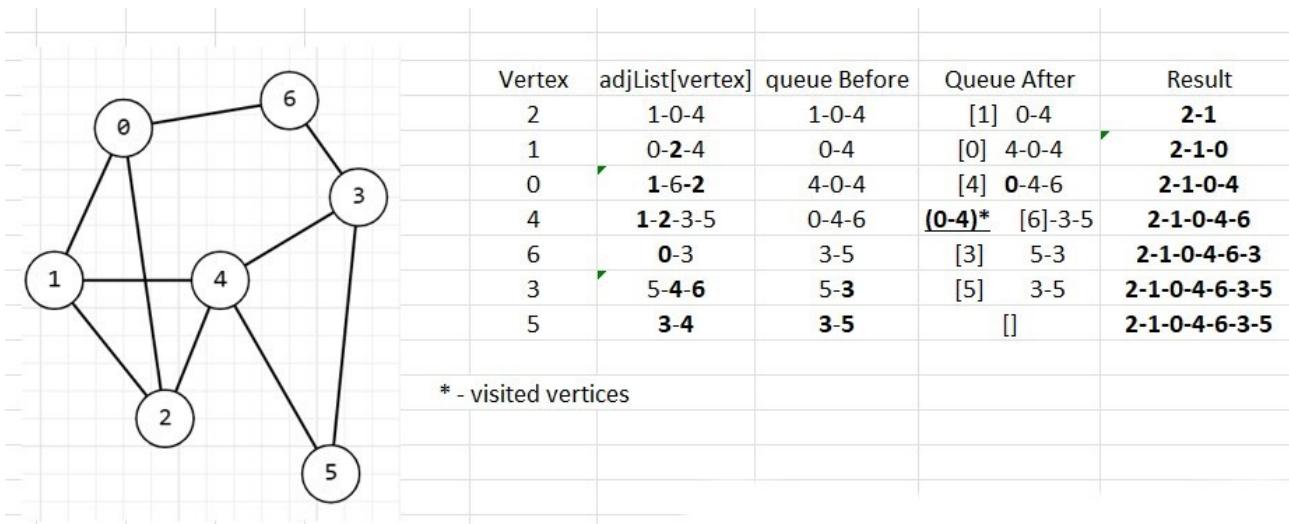


Рис. 9.18. Процесс формирования пути обхода графа по ширине

9.4.3. Удаление вершины из графа

Удаление вершины из графа происходит путем копирования всех вершин до удаляемой вершины, затем удаляемая вершина пропускается, после чего копирование оставшихся вершин продолжается. Дракон-диаграмма алгоритма удаления вершины показана на рис. 9.19.

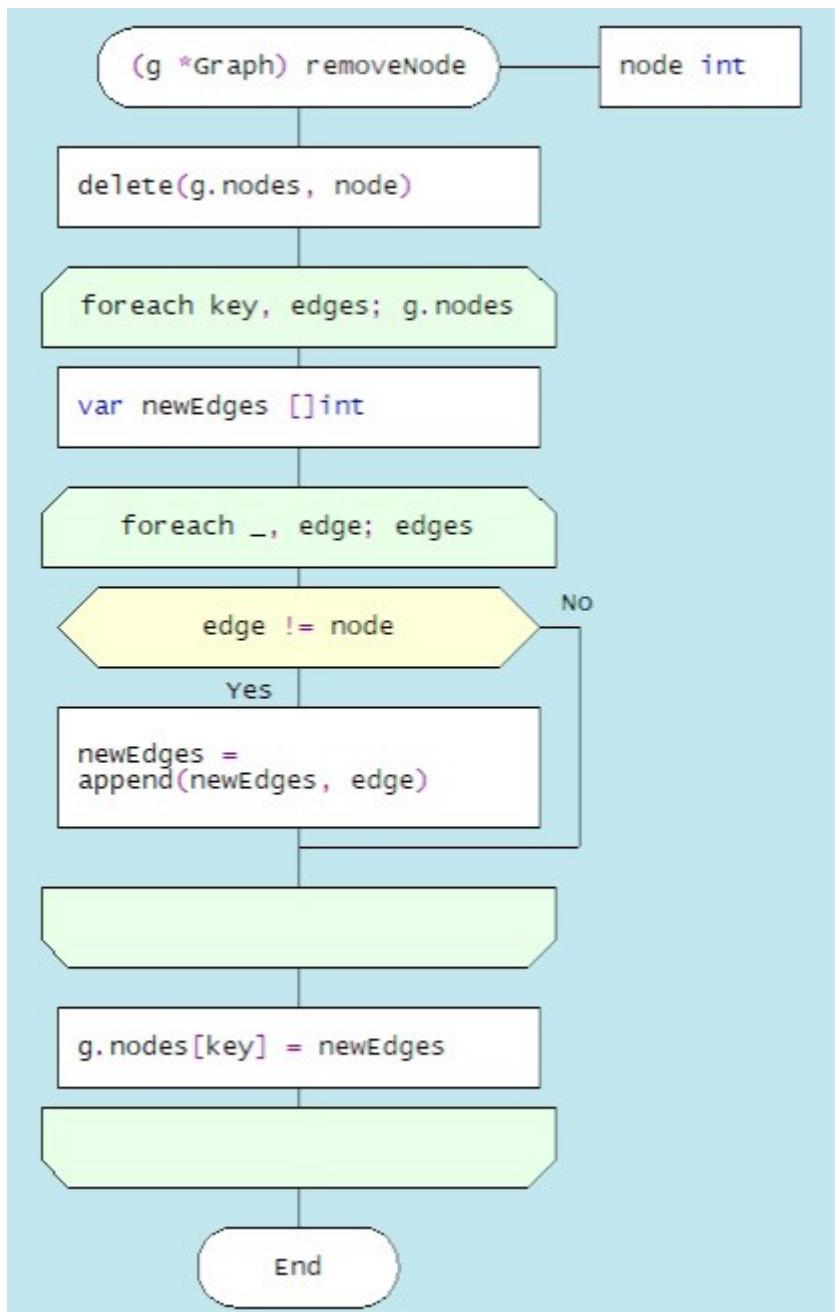


Рис. 9.19. Дракон-диаграмма алгоритма удаления вершины из графа

9.5. Выбор пути между вершинами в направленном графе

Выбор пути между двумя вершинами в направленном графе, имеющий практическое значение в смысле поиска оптимального пути между двумя населенными пунктами, соединенными дорогами с односторонним движением, может осуществляться по различным критериям. Поставим задачу следующим образом: с целью определения оптимального пути найти расстояния всех возможных путей между двумя географическими пунктами, а также суммарную стоимость проезда с учетом различной стоимости за 1 км на различных участках дороги.

Для решения этой задачи необходимо создать нагруженный граф, конкретнее, нагрузить ребра между всеми вершинами значениями расстояний между соседними пунктами и ценой проезда за 1 километр (рис. 9.20).

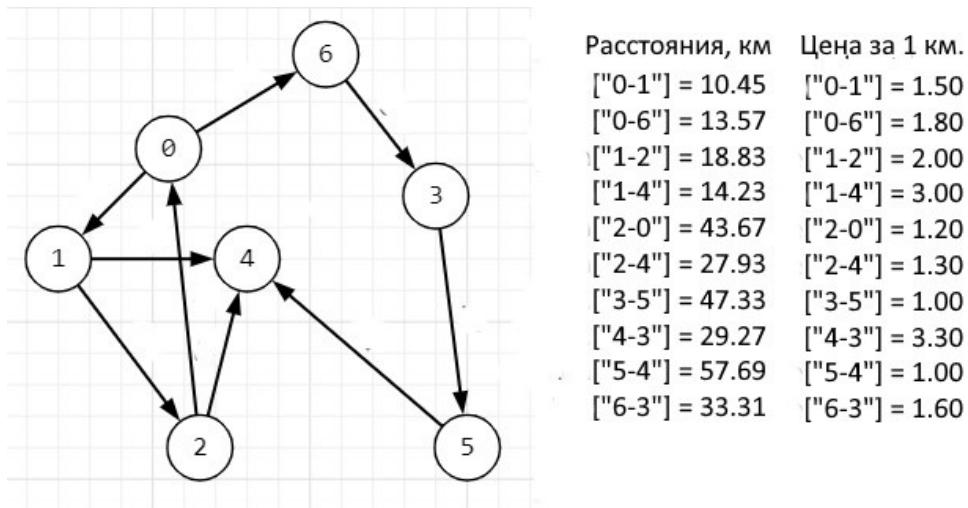


Рис. 9.20. Нагруженный направленный граф с исходными данными

Создание нового экземпляра графа (метод *newGraph*, добавление информации о ребрах (метод *addEdge*) и вызов метода *findPaths* (метод *allPaths*) представлены на рис. 9.21.

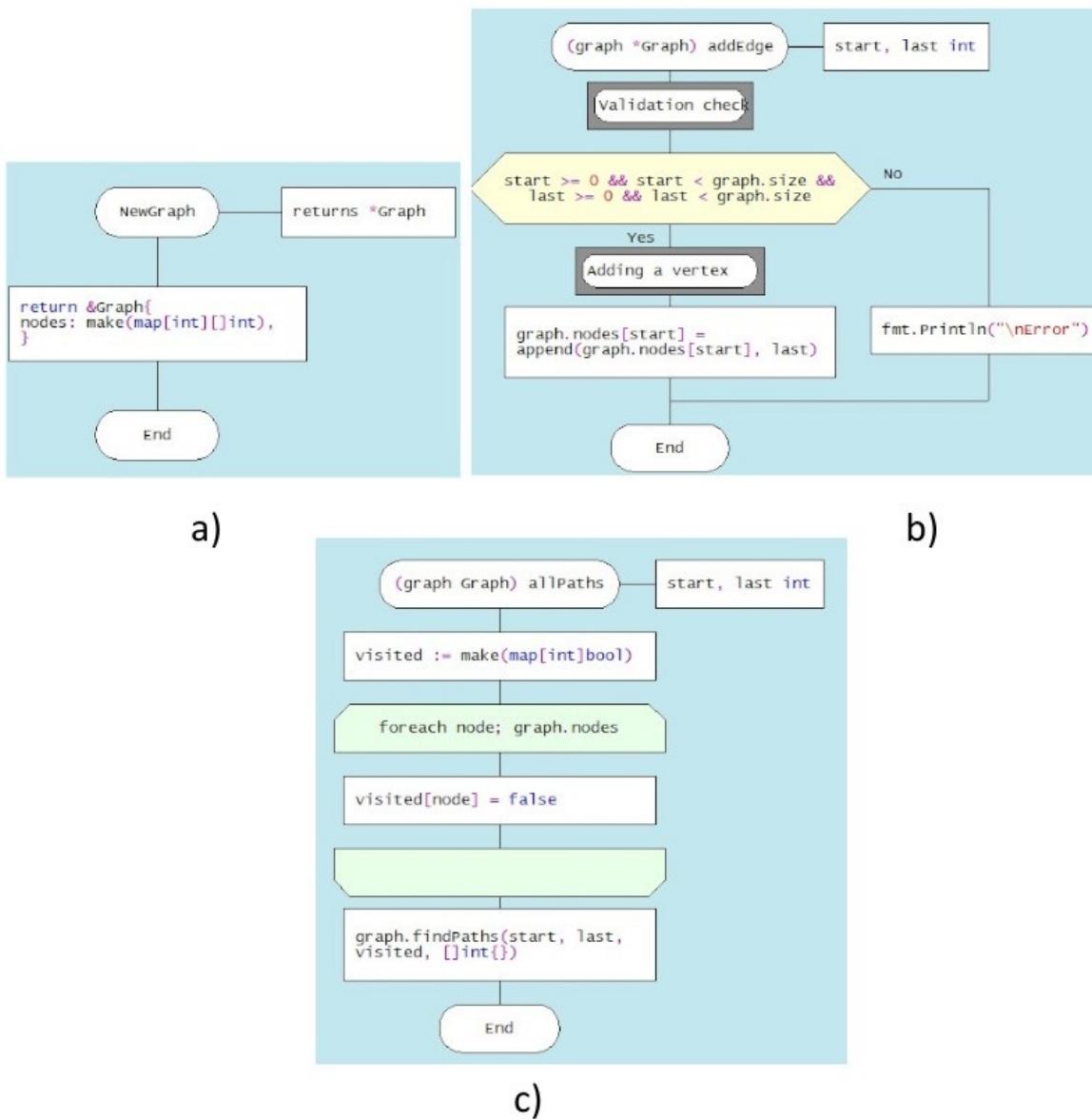


Рис. 9.21. Дракон-диаграммы методов создания графа

a) *newGraph*; **b)** *addEdge*; **c)** *allPaths*

Алгоритм поиска всех возможных путей между двумя вершинами основан на последовательном анализе списка связностей вершин и рекурсивном обращении к модулю *findPath*, реализующим обход узлов с фиксацией посещения каждой вершины. Дракон-диаграмма метода *findPath* представлена на рис. 9.22. Алгоритм состоит из трех частей: в первой части устанавливается корректность параметров модуля (имена начальной и конечной вершин), проверка факта посещения текущей

вершины. Во второй части осуществляется проход по еще не посещенным узлам графа, формируя строку, состоящую из посещенных имен вершин и разделителя между ними, например, «2-0-1-4-3-5» (рис. 9.21). В третьей части при достижении конечной вершины, то-есть, при выполнении условия `«start == last»`, определяется минимальный путь между этими вершинами. Кроме того, строка возможного пути, например, «2-0-1-4-3-5», разбивается на участки типа «2-0» с целью формирования ключей для карт, содержащих расстояния между соответствующими вершинами и стоимости проезда 1 км на этих участках.

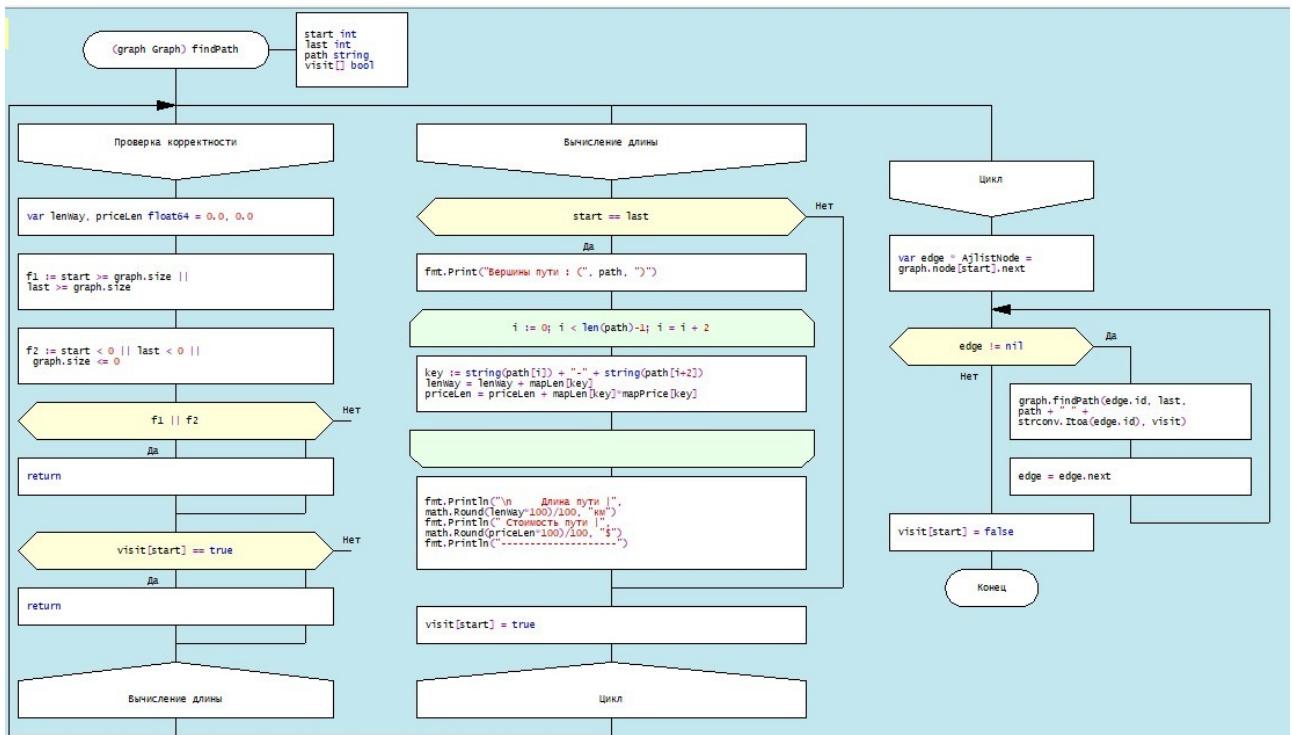


Рис. 9.22. Дракон-диаграмма модуля findPath

Рассмотрим задачу поиска всех возможных путей между двумя вершинами графа (2) и (5), показанного на рис. 9.23:

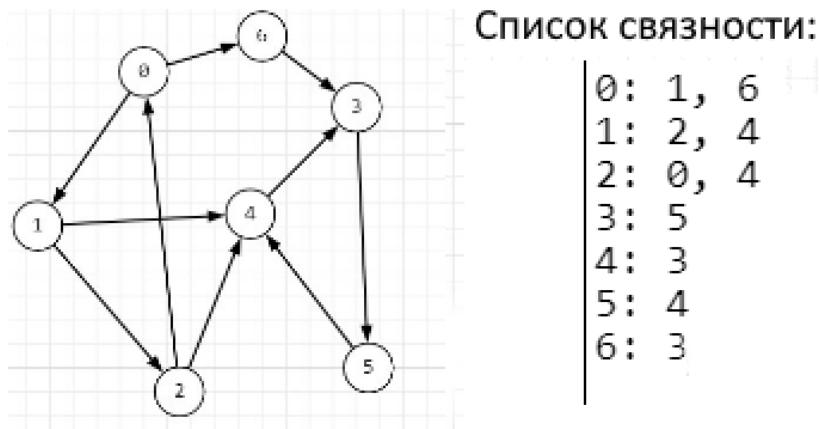


Рис.9.23. Граф и список связности

Результат первого обхода графа между вершинами (2) и (5), включая процесс рекурсии, показан в табл.9.2.

Табл.9.2. Процесс реализации модуля *findPath*

Vertices	Neighbors	Path	Visited vertices
start \neq last			
2	(2, 4)	2	TRUE
0	(1, 6)	2 0	TRUE
1	(2,4)	2 0 1	TRUE
4	(3)	2 0 1 4	TRUE
3	(5)	2 0 1 4 3	TRUE
5	(4)	2 0 1 4 3 5	TRUE

start = last \rightarrow Recursion \rightarrow (path = path[:len(path) - 1]

Vertices	Path	Visited vertices (true); Unvisited vertices (false)
5	2 0 1 4 3 5	0:true 1:true 2:true 3:true 4:true 5:true 6:false

3	2 0 1 4 3	0:true 1:true 2:true 3:true 4:true 5:false 6:false
4	2 0 1 4	0:true 1:true 2:true 3:false 4:true 5:false 6:false
1	2 0 1	0:true 1:true 2:true 3:false 4:false 5:false 6:false
0	2 0	0:true 1:false 2:true 3:false 4:false 5:false 6:false

Все пути обхода ориентированного графа между вершинами (2) и (5) показаны на рис. 9.24.

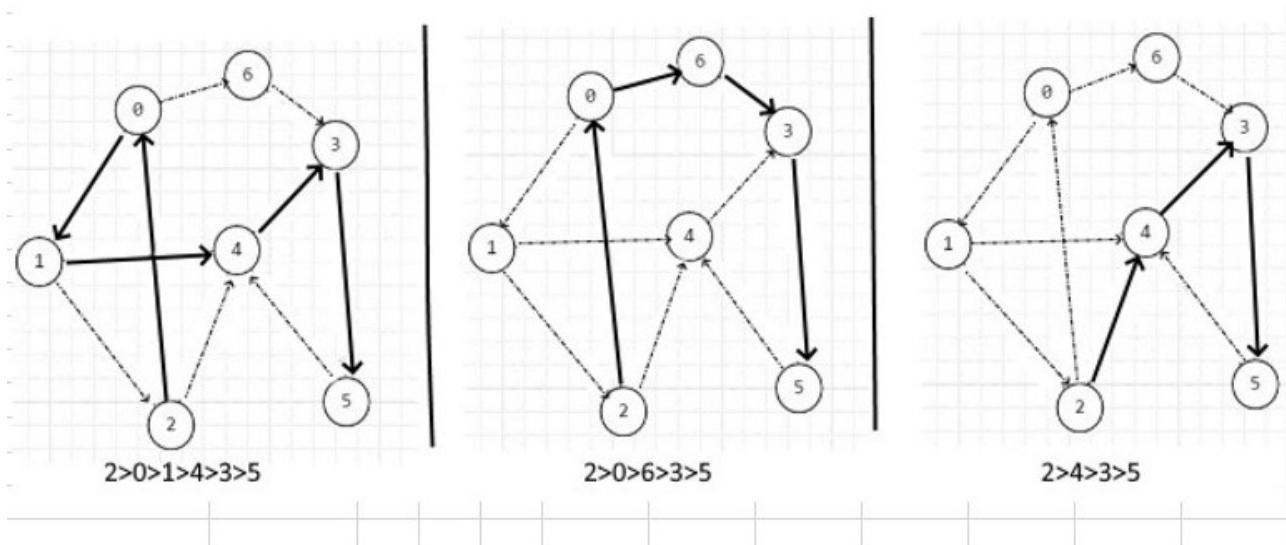


Рис. 9.24. Возможные пути обхода вершин ориентированного графа

Результаты сравнения стоимостей проезда от вершины (2) и (5) показаны в таблице 9.3.

Таблица 9.3. Расчет затрат на проезд

Travel	Length of travel, km	Cost of travel, \$
2-0-1-4-3-5	144.95	254.69
2-0-6-3-5	137.88	177.46
2-4-3-5	104.53	180.23

Из таблицы 9.3. следует, что с точки зрения затрат второй маршрут (2-0-6-3-5) является наиболее выгодным.

9.6. Алгоритм Дейкстры – метод нахождения кратчайшего пути

Алгоритм Дейкстры (англ. *Dijkstra's algorithm*) разработан с целью нахождения кратчайших путей от заданной вершины до всех остальных вершин при условии неотрицательных весов ребер графа. Алгоритм был назван по имени голландского ученого Э. В. Дейкстры в 1956 году. Данный алгоритм получил широкое распространение в различных областях науки, техники и социальной жизни. В GPS-системах алгоритм используется для нахождения наиболее быстрого пути до пункта назначения. В телекоммуникационных сетях он используется для определения оптимального пути передачи данных от источника к получателю. В робототехнике алгоритм может быть использован для планирования пути робота для достижения цели наиболее эффективным образом. В компьютерных играх – для определения пути персонажей или объектов. В сетевом планировании алгоритм может быть применен для нахождения оптимального пути доставки товаров.

Для реализации алгоритма Дейкстры создаются структуры *vertex* типа *Vertex* для описания вершин и *Graph* для описания графа (рис.24)

```
Edit file description  
--- header ---  
package main  
  
import (  
    "fmt"  
    "math"  
)  
  
type Graph struct {  
    vertices []*Vertex  
}  
  
type Vertex struct {  
    key      int  
    adjacent []*Vertex  
    lengths [][]float64  
    dist     float64  
}  
--- footer ---
```

Рис. 9.25 Описание структур вершин и графа

Здесь *key* – ключ вершины графа; *adjacent* – список смежности для вершины; *lengths* - набор длин ребер до вершин из списка смежности; *dist* – суммарная длина пути.

Далее, в программе *main* создается экземпляр графа и в цикле создаются экземпляры вершин (рис. 9.26):

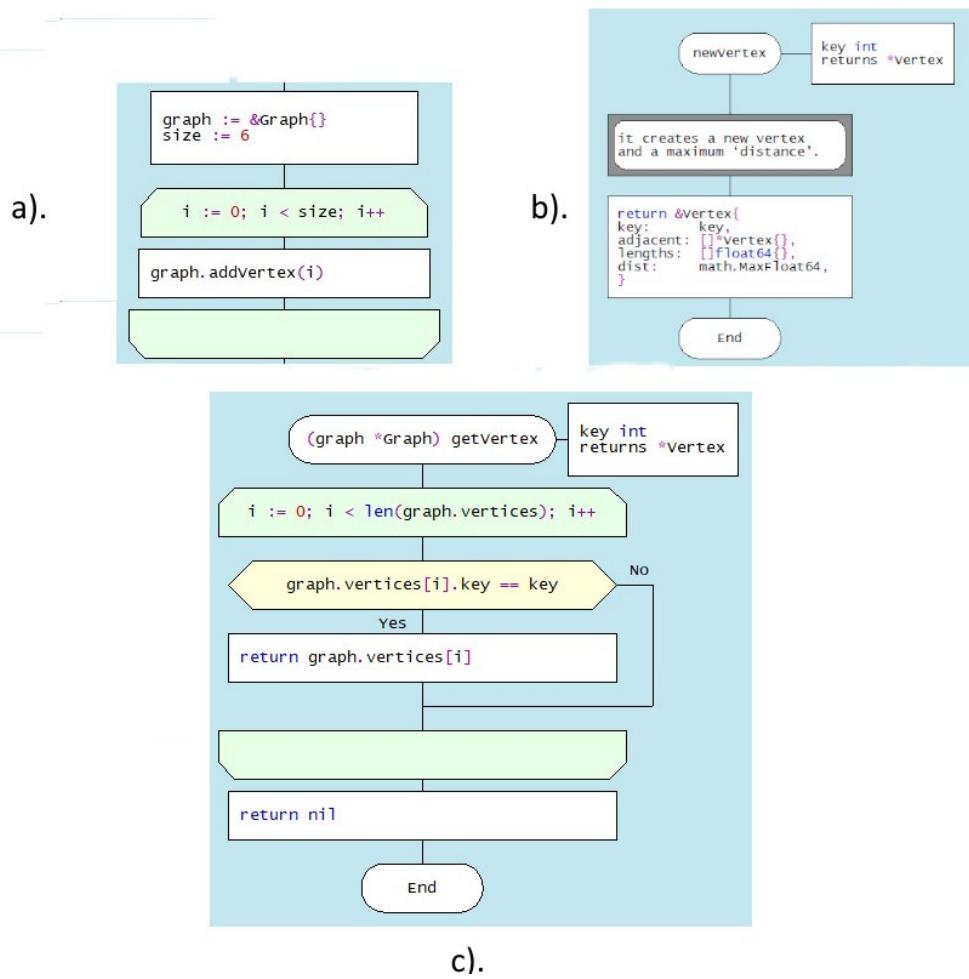


Рис. 9.26. Дракон-диаграммы методов формирования вершин графа

- создание экземпляра графа и шаблонов вершин
- создание новой вершины
- наполнение вершины данными

После создания вершин графа и заполнения их данными (ключи, соседние вершины и расстояния), формируются ребра с помощью метода `addEdge` (Рис. 9.27):

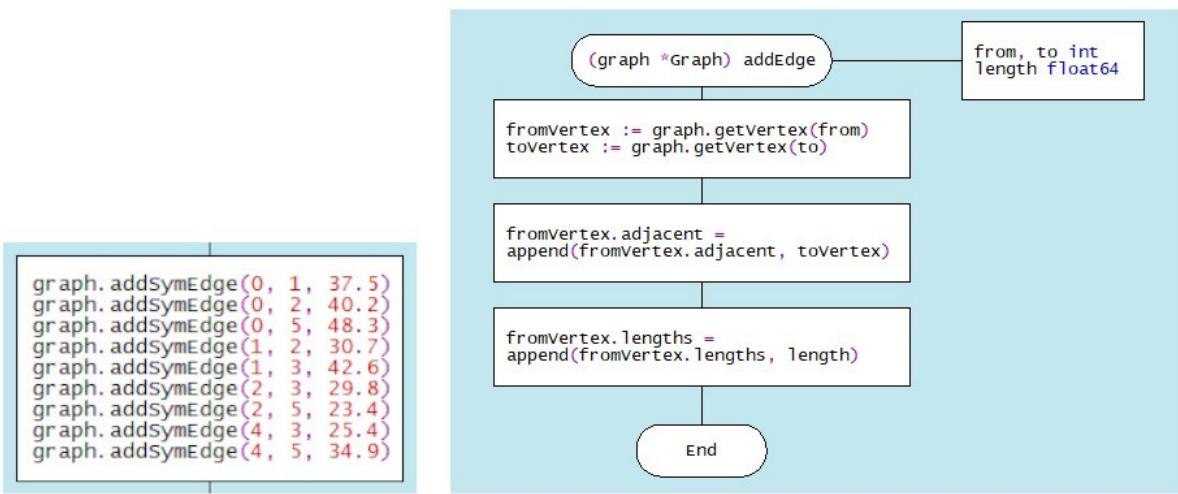


Рис. 9.27. Дракон-диаграммы метода формирования ребер графа *addEdge*

Алгоритм метода *addEdge(from, to, length)* реализуется следующим образом.

Сначала метод получает две вершины, между которыми нужно добавить ребро (*graph.getVertex(from)*, *graph.getVertex(to)*), где *from* и *to* - ключи вершин. Затем метод добавляет вершину *toVertex* в список смежных вершин для вершины *fromVertex* (*append(fromVertex.adjacent, toVertex)*). В результате формируется ребро между этими двумя вершинами *fromVertex* и *toVertex*. На последнем шаге метод добавляет длину ребра в список длин ребер для вершины *fromVertex* (*append(fromVertex.lengths, length)*). Это означает, что длина ребра между вершинами *fromVertex* и *toVertex* теперь известна.

Заметим, что в программе *main* используется метод *addSymEdge* (*u,v,L*) который создает симметричные данные по каждому ребру в ненаправленном графе (рис.9.28):

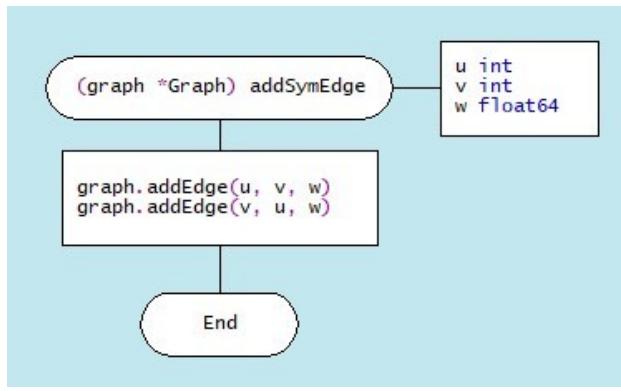


Рис.9.28. Дракон-диаграмма метода симметризации ребра *addSymEdge*

Далее происходит обращение к методу *Dijkstra* (*startKey int*), где *startKey int* – исходная вершина, и вывод конечных результатов (Рис.9.29):

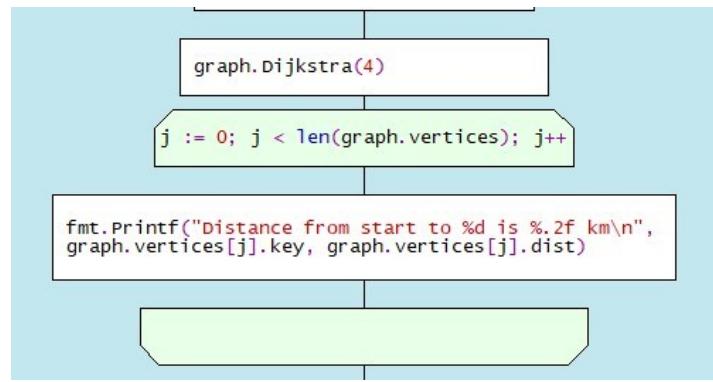


Рис. 29. Дракон-диаграмма обращения к методу *Dijkstra* и вывод результатов

Дракон-диаграмма метода *Dijkstra(startkey)* представлена на рис. 9.30.

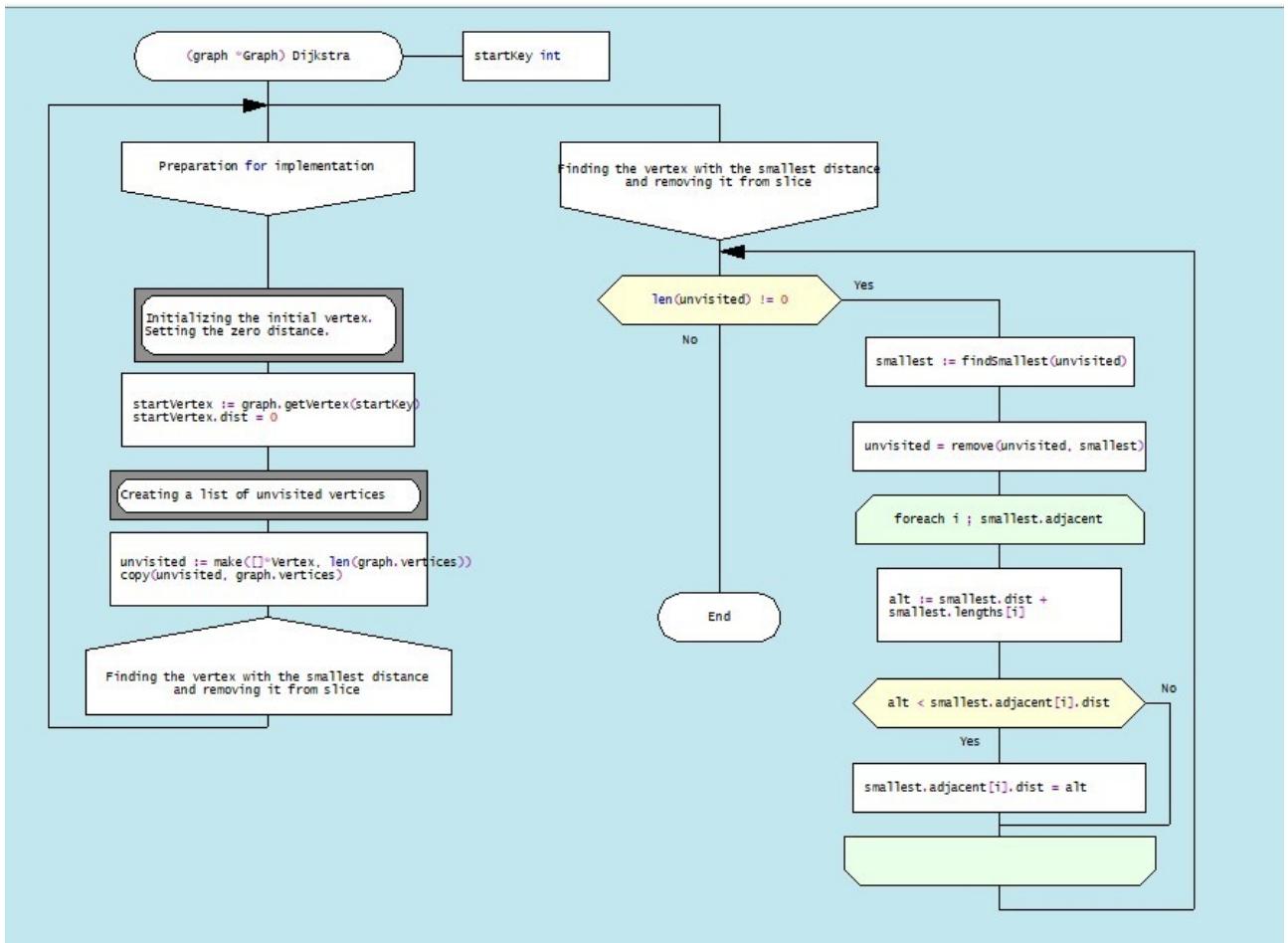


Рис. 9.30. Дракон-диаграмма метода *Dijkstra(startkey)*

Рассмотрим основные шаги реализации этого алгоритма. Вначале происходит инициализация (`startVertex := graph.getVertex(startKey); startVertex.dist = 0`), результатом которой является получение начальной вершины по ключу и установление ее расстояния, равное 0. На втором шаге создается список непосещенных вершин, который затем копируется командой `copy` с целью обеспечения возможности манипулировать этим списком в процессе обработки данных вершин, не меняя сам граф.

Далее выполняется цикл `for len(unvisited) != 0` обхода всех непосещенных вершин с целью поиска вершины с наименьшим расстоянием с помощью обращения к методу `smallest := findSmallest(unvisited)` (Рис. 9.31.)

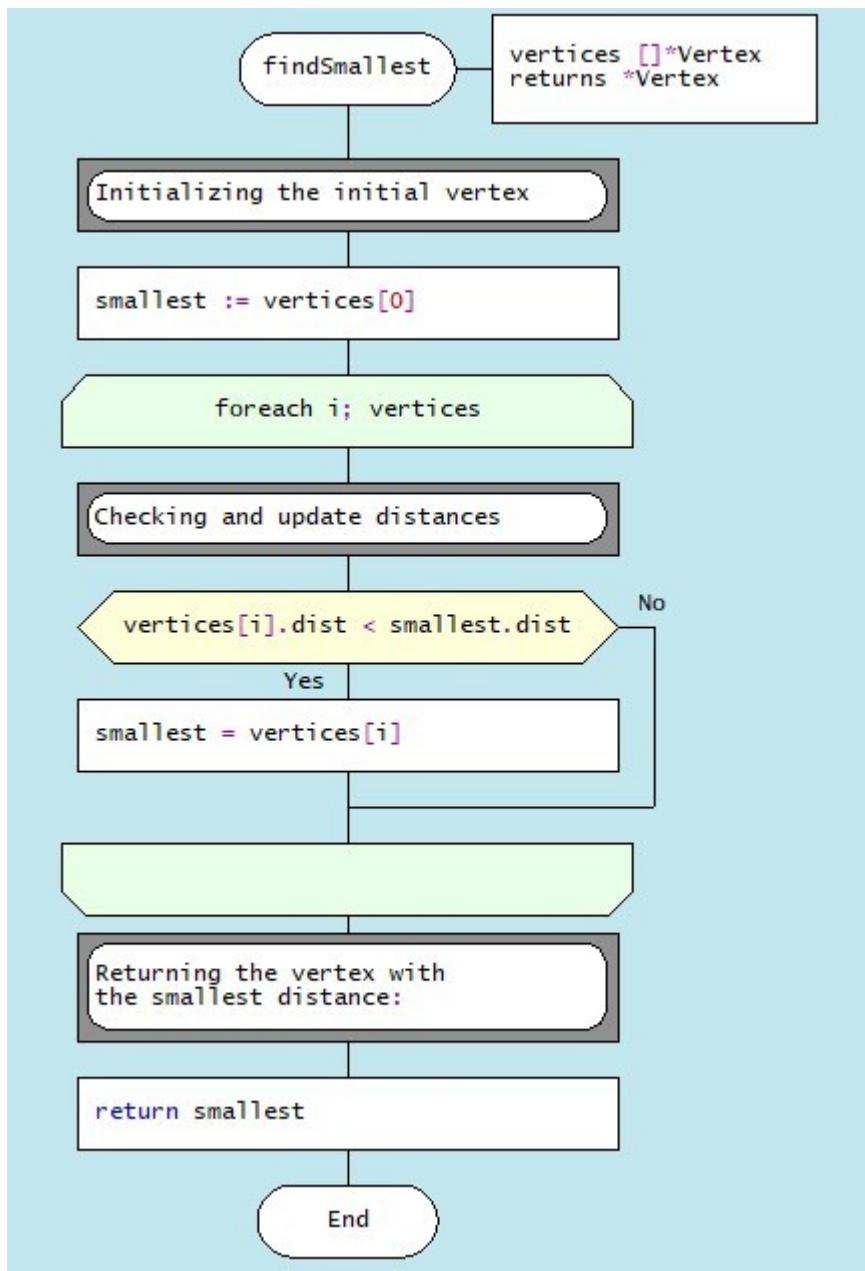


Рис. 9.31. Дракон диаграмма метода *findSmallest(unvisited)*

Затем из списка непосещенных вершин удаляется текущая вершина (*unvisited* = *remove(unvisited, smallest)*). И, наконец, в цикле по всем вершинам происходит обновление расстояний до смежных вершин в результате обнаружения более короткого пути.

В качестве примера рассмотрим применение алгоритма Дейкстры к нахождению кратчайших путей от заданной вершины для неориентированного графа, изображенного на рис. 9.32.

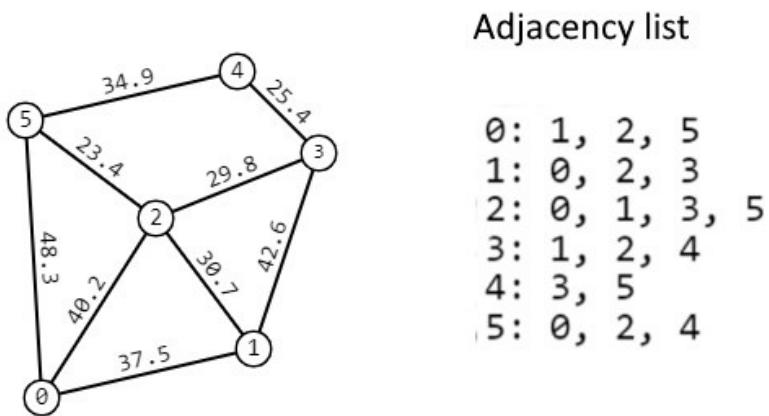


Рис.9.32. Неориентированный нагруженный граф

Результаты расчетов кратчайших путей от вершины 0 до остальных вершин графа (рис. 9.33.):

Distance from start to 0 is 0.00 km

Distance from start to 1 is 37.50 km

Distance from start to 2 is 40.20 km

Distance from start to 3 is 70.00 km

Distance from start to 4 is 83.20 km

Distance from start to 5 is 48.30 km

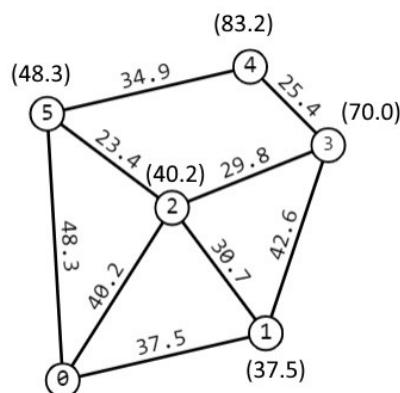


Рис. 9.33. Кратчайшие расстояния от вершины 0 указаны в скобках

И, в завершение, об оценке сложности алгоритма Дейкстры. Эта оценка может варьироваться в зависимости от структуры данных, используемой для представления графа. В случае применения связанного списка (Linked List) времененная сложность алгоритма Дейкстры оценивается как $O(V^2)$, где V - количество вершин в графе. Это связано с тем, что приходится просматривать все вершины при каждом извлечении минимального элемента из очереди приоритетов.

При использовании структуры данных *map* времененная сложность алгоритма Дейкстры будет $O((V+E) \log V)$, где E - количество ребер в графе. Это связано с тем, что извлечение минимального элемента из очереди приоритетов, реализованной с помощью *map*, занимает время $O(\log V)$.

В обоих случаях пространственная сложность будет $O(V + E)$, так как нужно хранить все вершины и ребра графа. Однако, стоит отметить, что *map* обычно занимает больше места, чем связанный список, из-за дополнительной информации, которую он хранит (ключи и значения).