

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ СОЦИОКУЛЬТУРНЫХ КОММУНИКАЦИЙ
КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

А. В. Овсянников, Ю.А. Пикман

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Часть I

**УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС
ДЛЯ СПЕЦИАЛЬНОСТИ 1-31 03 07
ПРИКЛАДНАЯ ИНФОРМАТИКА
(по направлениям)**

Минск 2015

УДК004.421(075.8)+004.422.63(075.8)
О-345

А в т о р ы:

профессор кафедры информационных технологий
факультета социокультурных коммуникаций
Белорусского государственного университета
к. т. н., доцент А. В. Овсянников,
старший преподаватель кафедры информационных технологий
факультета социокультурных коммуникаций
Белорусского государственного университета
Ю. А. Пикман

Рецензенты:

заведующий кафедрой систем управления Белорусского государственного
университета информатики и радиоэлектроники, к.т.н., доцент Марков А.В.,
кафедра информационных радиотехнологий Белорусского государственного
университета информатики и радиоэлектроники, В. М. Козел, к. т. н., доцент;

Решение о депонировании документа вынес

Совет гуманитарного факультета БГУ
02.03.2015 г., протокол № 8

Овсянников, А. В. Алгоритмы и структуры данных : учебно-методический комплекс для специальности 1-31 03 07 «Прикладная информатика (по направлениям)». Ч. 1 / А. В. Овсянников, Ю. А. Пикман ; БГУ, Фак. социокультурных коммуникаций, Каф. информационных технологий. – Минск : БГУ, 2015. – 124 с. : ил. – Библиогр.: с. 122

Учебно-методический комплекс содержит материал, включающий фундаментальные понятия алгоритма, размерности задачи, трудоемкости алгоритмов, структур данных. Комплекс представлен теоретическим разделом с контрольными вопросами и лабораторным практикумом, а также учебной программой дисциплины. В теоретическом разделе рассматриваются общие вопросы методологии теории алгоритмов и структур данных, абстрактные вычислительные машины, анализ трудоемкости алгоритмов и введение в структуры данных. Лабораторный практикум содержит материал для самостоятельного выполнения и исследования организации работы линейных списков и бинарного дерева.

Адресуется студентам специальности «Прикладная информатика» гуманитарного факультета БГУ. Может быть использован в процессе изучения методики анализа алгоритмов и основ программирования студентами других специальностей.

УДК004.421(075.8)+
004.422.63(075.8)

© А.В. Овсянников, Ю.А. Пикман

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Учебно-методический комплекс включает:

1. Овсянников А.В. Учебная программа учреждения высшего образования по учебной дисциплине для специальности: 1-31 03 07-03 Прикладная информатика Алгоритмы и структуры данных № УД-1499/р. Режим доступа: <http://elib.bsu.by/handle/123456789/85554>

2. Данное пособие «Алгоритмы и структуры данных (часть I)», содержание которого полностью соответствует учебной программе дисциплины и включает 9 учебных модулей по следующим разделам: 1) теоретическую часть: методологические основы теории алгоритмов и структур данных; абстрактные вычислительные машины; анализ алгоритмов; введение в структуры данных; 2) лабораторный практикум: организация и принципы работы с односвязным линейным списком; изучение принципов организации и работы абстрактной структуры данных стек; изучение принципов организации и работы абстрактной структурой данных очередь; изучение принципов организации и работы с абстрактной структурой данных двухсвязный линейный список; изучение принципов организации и работы с абстрактной структурой данных бинарное дерево. В каждом теоретическом модуле в достаточной мере изложен основной теоретический материал, который подкрепляется значительным количеством иллюстраций и примеров. Каждый модуль содержит: 1) теоретическую часть, представленную в виде краткого конспекта лекций; 2) систему контрольных вопросов и заданий для аудиторной и самостоятельной работы студентов.

Учебно-методический комплекс предназначен для организации аудиторной и самостоятельной работы студентов информационных специальностей вузов в процессе изучения дисциплины «Алгоритмы и структуры данных». Сформированная система знаний, умений и навыков необходима студентам для осуществления профессиональной деятельности и развития сферы их научно-исследовательских интересов в области веб-программирования, освоения принципов программирования веб-сайтов и веб-интерфейсов.

Methodical complex is designed to organize classroom and independent work of students of information professions schools in the process of studying the discipline "Algorithms and Data Structures." Formed a system of knowledge and skills necessary for students to carry out professional activities and development of the scope of their research interests in the field of web programming, mastering the principles of programming websites and web interfaces.

1. МЕТОДОЛОГИЯ ТЕОРИИ АЛГОРИТМОВ И СТРУКТУР ДАННЫХ

ИСТОРИЯ ВОЗНИКНОВЕНИЯ ПОНЯТИЯ «АЛГОРИТМ»

Термин «Алгоритм» происходит от имени хорезмского учёного **Аль-Хорезми Мухаммед бен-Муса (825 г н.э.)**. В явном виде понятие алгоритма сформировалось в начале XX века благодаря работам таких учёных, как Д. Гильберт, К. Гёдель, С. Клини, А. Чёрч, Э. Пост, А. Тьюринг, Н. Винер, А.А. Марков.

Одним из старейших численных алгоритмов считается **алгоритм Евклида** (III век до н.э.) – нахождения наибольшего общего делителя двух чисел.

Современная теория алгоритмов началась с работ немецкого математика Курта Гёделя (1931 год), в которых было показано существование задач, не разрешимых в рамках своей формальной, непротиворечивой системы аксиом.

Первые фундаментальные работы по теории алгоритмов появились в 1936 году. Предложены машина Тьюринга, Поста и λ -исчисление Чёрча. Эти машины были эквивалентными формализациями алгоритма.

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

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

- **Классическая теория алгоритмов** (формулировка задач в терминах формальных языков, понятие задачи разрешимости, введение

сложностных классов, формулировка проблемы $P=NP(?)$, открытие класса NP-полных задач и его исследование);

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

ЦЕЛИ И ЗАДАЧИ ТЕОРИИ АЛГОРИТМОВ

Цели и задачи, решаемые в теории алгоритмов:

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

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ РЕЗУЛЬТАТОВ ТЕОРИИ АЛГОРИТМОВ

Теоретический аспект

При исследовании некоторой задачи позволяет ответить на вопросы:

- 1) является задача в принципе алгоритмически разрешимой?

В случае «ДА» возникает следующий вопрос:

Принадлежит ли эта задача к классу NP-полных задач?

При утвердительном ответе, анализируют временные затраты на получения точного решения для больших размерностей исходных данных.

- 2) для алгоритмически неразрешимых задач возможно ли их сведение к задаче останова машины Тьюринга?

Практический аспект

Методы и методики теории алгоритмов позволяют осуществить:

- рациональный выбор из известного множества алгоритмов решения задачи с учетом особенностей их применения (например, при ограничениях на размерность исходных данных или объема дополнительной памяти);
- получение временных оценок решения сложных задач;
- получение достоверных оценок невозможности решения некоторой задачи за определенное время, что важно для криптографических методов;
- разработку и совершенствование эффективных алгоритмов решения задач в области обработки информации на основе практического анализа.

ФОРМАЛИЗАЦИЯ ПОНЯТИЯ АЛГОРИТМА

Определение 1. *Алгоритм – это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых элементарных операций для решения задачи, общее для класса возможных исходных данных.*

Пусть D – область (множество) исходных данных задачи, а R – множество возможных результатов, тогда алгоритм осуществляет отображение $D \rightarrow R$. Это отображение может быть не полным.

Алгоритм называется **частичным алгоритмом**, если получен результат только для некоторых $d \in D$ и **полным алгоритмом**, если алгоритм получает правильный результат для всех $d \in D$.

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

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

- алгоритм должен содержать конечное количество элементарно выполнимых предписаний, т.е. удовлетворять требованию конечности записи;

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

ФОРМАЛЬНЫЕ СВОЙСТВА АЛГОРИТМОВ

Дискретность — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. Для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.

Детерминированность — определённая. В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных.

Понятность — алгоритм для исполнителя должен включать только те команды, которые ему (исполнителю) доступны, которые входят в его систему команд.

Завершаемость (конечность) — при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов. Вероятностный алгоритм может и никогда не выдать результат, но вероятность этого равна 0.

Массовость — универсальность. Алгоритм должен быть применим к разным наборам исходных данных.

Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не даёт результатов вовсе.

Алгоритм не содержит ошибок, если он даёт правильные результаты для любых допустимых исходных данных.

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

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

ПОНЯТИЕ СТРУКТУРЫ ДАННЫХ

Структура данных — множество элементов данных и множество связей между ними.

Структура данных — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных с помощью вычислительной техники.

Для добавления, поиска, изменения и удаления данных структура данных предоставляет некоторый набор функций, составляющих её интерфейс. Структура данных является реализацией какого-либо **абстрактного типа данных**.

Для точного описания абстрактных структур данных и алгоритмов программ используются системы формальных обозначений – языки программирования, в которых смысл всякого предложения определяется точно и однозначно.

Структура данных относится к "пространственным" понятиям: ее можно свести к схеме организации информации в памяти компьютера (рис.1).



Рис. 1. Схема организации информации в памяти компьютера

Понятие **"физическая структура данных"** соответствует способу физического представления данных в памяти машины и называется ещё **структурой хранения, внутренней структурой** или **структурой памяти**.

Структуры данных, которая рассматривается без учета ее представления в машинной памяти называется абстрактной или **логической структурой**(рис.2).

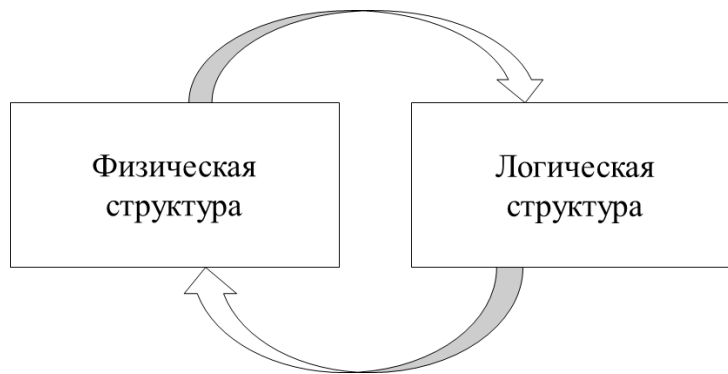


Рис.2. Способ представления структуры данных

С понятием «**структура данных**» тесно связано понятие «**тип данных**». Например, натуральные и целые числа, вещественные числа (*в виде приближенных десятичных дробей*), символы алфавита, строки и т.п. – это **типы данных**. В языках программирования тип данных (*констант и переменных*)

- определяется компилятором автоматически
- или
- задается явно.

Информация по каждому типу данных однозначно определяет:

- **способ хранения** данных указанного типа, т.е. выделение памяти и представление данных в ней и интерпретирование двоичного представления;
- **множество допустимых значений**, которые может принимать тот или иной объект указанного типа;
- **множество допустимых операций**, которые применимы к объекту указанного типа.

КЛАССИФИКАЦИЯ СТРУКТУР ДАННЫХ

По сложности:

- Простые (*базовые, примитивные*);
- Интегрированные (*композиционные, сложные*).

По связи между элементами:

- Связные (*связные списки*);
- Несвязные (*векторы, массивы, строки, очереди*).

Линейные структуры

по расположению элементов в памяти:

- Последовательные (*векторы, строки, массивы, стеки, очереди*);
- Произвольные (*односвязные, двусвязные списки*).

Нелинейные структуры

- Многосвязные списки, деревья, графы.

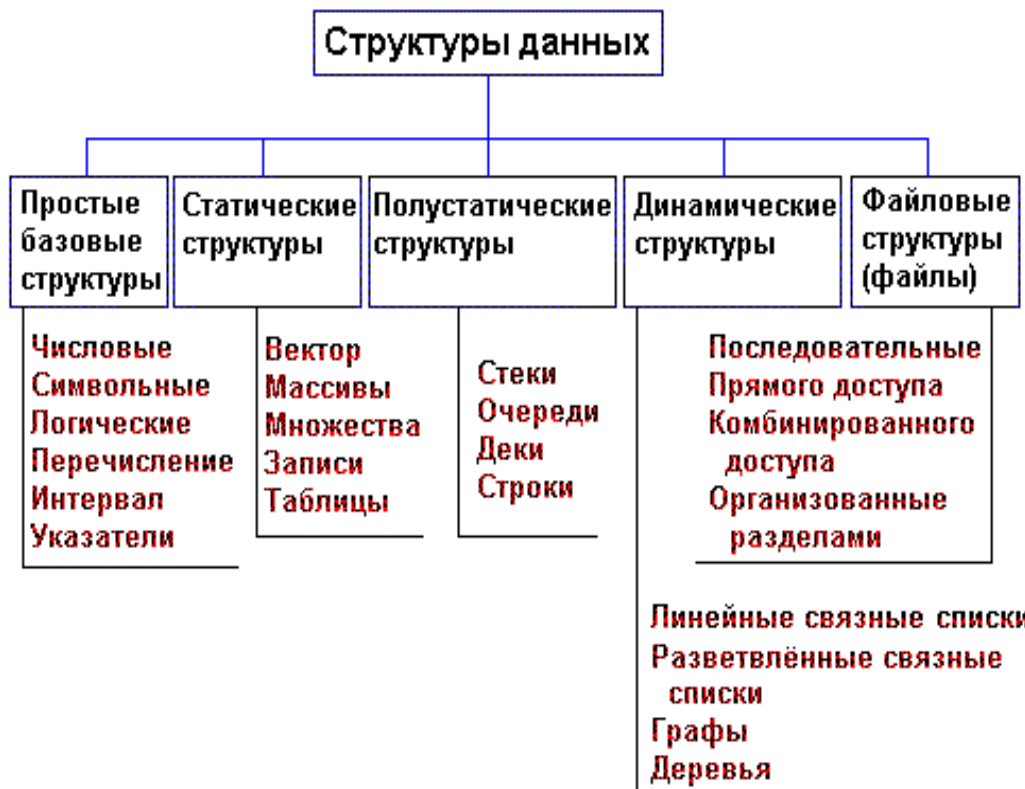


Рис.3. Классификация структур данных

ОПЕРАЦИИ НАД СТРУКТУРАМИ ДАННЫХ

- **Создание** – выделение памяти для структуры данных.
- **Уничтожение** – противоположна по своему действию операции создания.
- **Выбор** – обеспечивает доступ к данным внутри самой структуры.
- **Обновление** – позволяет изменять значения данных в структуре и добавлять (удалять) выбранные данные.

СТРУКТУРНОСТЬ ДАННЫХ И ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

- Структуры данных позволяют организовать их хранение и обработку максимально эффективным образом, с точки зрения минимизации затрат памяти и процессорного времени;
- Правильный выбор структуры программного продукта дает возможность сосредоточиться на одной обозримой части программного продукта на каждом этапе разработки и обеспечить реализацию разных его частей одновременно;

Известно, что от 50 до 100% времени программиста тратится на исправление и модификацию программ. Поэтому современная индустрия программирования предлагает системные подходы к программированию, использование которых:

- уменьшает вероятность ошибок в программах;
- упрощает их понимание;
- облегчает модификацию;

Структурное программирование – одна из популярных методик, фундаментом которой является теорема о структурировании.

Теорема. *Как бы сложна ни была задача, блок-схема соответствующей программы (алгоритма) всегда может быть представлена с использованием ограниченного числа элементарных управляющих структур (последовательность, ветвление, цикл).*

Идея доказательства:

- преобразование каждой части алгоритма в одну из трех основных структур или их комбинацию;
- после достаточного числа таких преобразований оставшаяся неструктурированная часть либо исчезнет, либо станет ненужной;
- в результате получится алгоритм, эквивалентный исходному и использующий лишь 3 управляющие структуры.

Цель структурного программирования – выбор структуры программы путем разложения исходной задачи на подзадачи (*декомпозиция*). Программы должны иметь простую структуру. Разработка алгоритма упрощается на каждом уровне шаг за шагом.

При этом используется **метод пошагового уточнения** (Рис.4):

- задача рассматривается в целом, выделяются наиболее крупные ее части, алгоритм, указывающий порядок выполнения этих частей, описывается в структурированной форме, не вдаваясь в мелкие детали;
- от общей структуры переходят к описанию отдельных частей и, таким образом, разработка алгоритма состоит из последовательности шагов в направлении уточнения алгоритма.

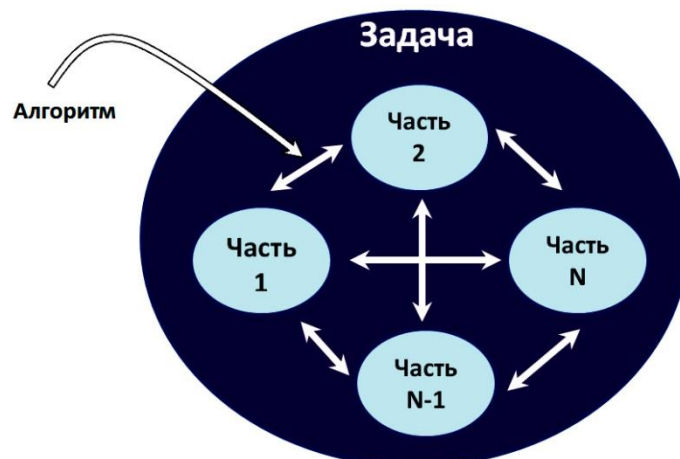


Рис.4. Метод пошагового уточнения

ПРИНЦИП МОДУЛЬНОГО ПРОГРАММИРОВАНИЯ

- Модульное программирование позволяет ускорить процесс за счет привлечения к работе нескольких специалистов сразу.
- Модульное программирование предполагает использования разработанных стандартных программ (библиотек стандартных подпрограмм).

Для проектирования алгоритма решения сложной задачи, состоящей из нескольких подзадач, можно использовать следующие подходы:

- **Нисходящее проектирование**
 - ✓ вначале проектируются функции управляющей программы – драйвера;
 - ✓ затем более подробно представляют каждую подзадачу и разрабатывают другие модули.

При нисходящем проектировании на каждом шаге функционирование модуля описывается с помощью ссылок на последующие, более подробные шаги.

- **Восходящее проектирование**
 - ✓ вначале проектируют программы низшего уровня, иногда в виде самостоятельных подпрограмм;
 - ✓ затем на каждом шаге разрабатываются модули более высокого уровня.

Для структурирования данных применяется технологический прием, называемый *инкапсуляция*.

Декомпозиция данных в сложных программах (рис.5):

1. Данные разбиваются на фрагменты одного типа (*одной структуры*).
2. С фрагментами связываются операции их обработки, формируются подзадачи.
3. Определяются необходимость пересылки данных.
4. Пересечение частей, на которые разбивается задача, должно быть сведено к минимуму.
5. Схема разбиения в дальнейшем может уточняться.
6. Если необходимо уменьшение числа обменов, допускается увеличение степени перекрытия подзадач.
7. Сначала анализируются структуры данных наибольшего размера, либо те, обращение к которым происходит чаще всего.

На разных стадиях расчета могут использоваться разные структуры данных, поэтому могут использоваться как статические, так и динамические схемы декомпозиции этих структур.

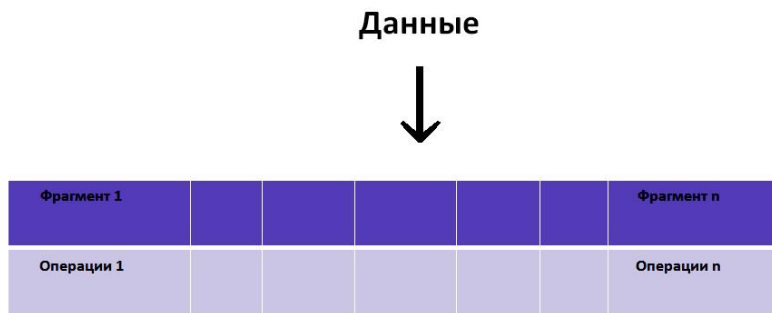


Рис. 5. Декомпозиция данных в сложных программах

Контрольные вопросы

1. Чем, на Ваш взгляд, отличается современное понятие (определение) алгоритма от предшествующих определений? Чем можно объяснить историческую трансформацию этого определения?
2. Какие существуют направления исследований в теории алгоритмов?
3. В чем состоит суть практического применения результатов теории алгоритмов?
4. Раскройте смысл формальных свойств алгоритма на примерах.
5. Поясните сопоставление информационной, логической структуры данных с ее способом организации в компьютере.
6. Дайте развернутое определение физической и логической структуры данных на примерах
7. Каковы основные признаки классификации структур данных?

8. Какие структуры данных будут востребованы в будущем, а какие станут неэффективными? Можно ли предположить появление новых структур данных в будущем?
9. В чем заключается идея структурного программирования?
10. В чем состоит отличие структурного программирования от модульного программирования?
11. В чем заключается разница между восходящим и нисходящим проектированием алгоритмов решения сложных задач?
12. Поясните этапы декомпозиции данных в сложных программах.

2. АБСТРАКТНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ

МАШИНА ПОСТА

Эмиль Пост рассматривает общую проблему (*общую задачу*), состоящую из множества конкретных проблем (*конкретных задач*). Решение общей проблемы – такое решение, которое даёт решение каждой конкретную (рис.1).

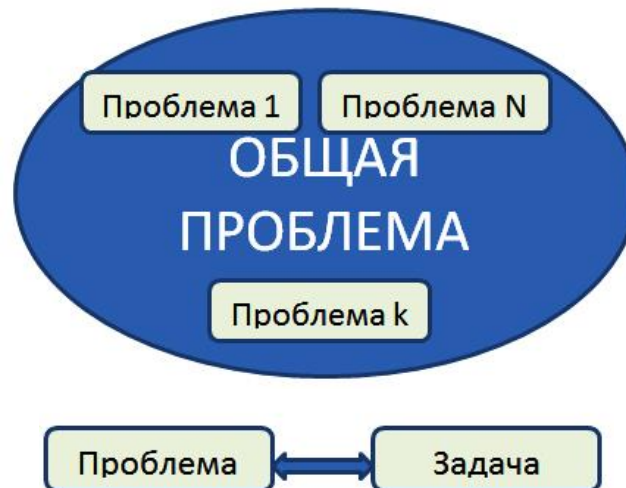


Рис.1. Общая задача Поста

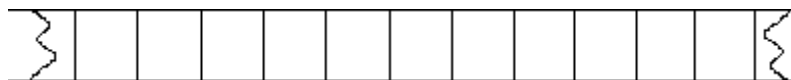
Например, решение уравнения $ax + b = 0$ это общая проблема, а решение уравнения $3x + 9 = 0$ – одна из конкретных (частных) проблем.

Основные понятия алгоритмического формализма Поста:

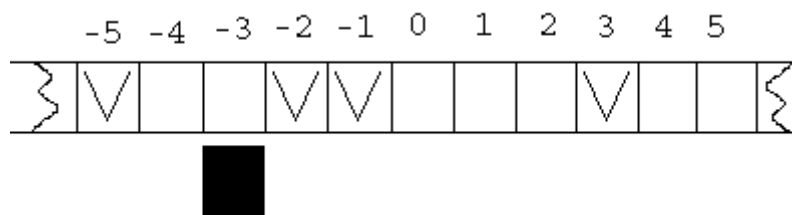
- пространство символов (язык S) в котором задаётся конкретная проблема и получается ответ;
- набор инструкций (*операций*) в пространстве символов, задающих как сами операции, так и порядок их выполнения.

Машина Поста состоит из:

- **бесконечной ленты**, поделенной на одинаковые ячейки (*секции*), ячейка может быть пустой (0 или *пустота*) или содержать метку (1 или V);



- **головки** (карыетки), способной передвигаться по ленте на одну ячейку в ту или иную сторону, а также проверять наличие метки, стирать и записывать метку.



Текущее состояние машины Поста описывается состоянием ленты и положением каретки.

Состояние ленты— информация о том, какие секции пусты, а какие отмечены.

Шаг — это движение каретки на одну ячейку влево или вправо. Состояние ленты может изменяться в процессе выполнения программы. Конкретная проблема задается «внешней силой» посредством пометки конечного количества ячеек. При этом любая конфигурация начинается и заканчивается помеченным ящиком.

Машина работает в единичной системе счисления ($0=V$; $1=VV$; $2=VVV$; $3=VVVV$), т.е. ноль представляется одним помеченным ящиком, а целое положительное число — помеченными ящиками в количестве на единицу больше его значения.

После применения к проблеме набора инструкций решение представляется в виде набора помеченных и непомеченных ячеек, распознаваемое «внешней силой».

Пост предложил **набор инструкций**. Этот набор инструкций является, минимальным набором битовых операций:

- 1) пометить ячейку, если она пуста;
- 2) стереть метку, если она есть;
- 3) переместить головку влево/вправо на 1 ячейку;
- 4) определить помечена ячейка или нет и, в зависимости от результата, выполнить инструкцию 3;
- 5) остановиться.

Программа представляет собой нумерованную последовательность инструкций, причем переходы в инструкции 3 производятся на указанные в ней номера других инструкций.

МАШИНА ТЬЮРИНГА И АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ПРОБЛЕМЫ

Статья «О вычислимых числах в приложении к проблеме разрешения» может считаться одной из основ современной теории алгоритмов. **Посвящена проблеме «разрешимости»** (Гильберт, 1900г.) — **нахождения общего метода** для определения выполнимости высказывания на языке формальной логики.

Тьюринг показал:

- этот метод (*алгоритм, процедура*) может быть выполнен механически, без творческого вмешательства;
- как эту идею можно воплотить в виде подробной модели вычислительного процесса.

Машина Тьюринга – логическая конструкция модели вычислений, в которой алгоритм разбивается на последовательность элементарных шагов.

Машина Тьюринга является расширением модели конечного автомата, который включает в себя потенциально бесконечную память с возможностью перехода (*движения*) от обозреваемой в данный момент ячейки к ее левому/правому соседу.

Формально машина Тьюринга может быть описана следующим образом:

- конечное множества букв алфавита $S = \{s_1, \dots, s_n\}$;
- конечное множества состояний $Q = \{q_1, \dots, q_k\}$;
- движения головки $D = \{d_1, d_2, d_3\}$: на ячейку влево, вправо, остаться на месте;
- набором правил (*функций перехода*), по которым работает машина:

$$\delta: q_i s_j \rightarrow q_m s_n d_k .$$

Для каждой возможной конфигурации $\langle q_i, s_j \rangle$ имеется ровно одно правило. Для заключительного состояния, попав в которое машина останавливается, правил δ нет.

- один из символов алфавита пустой e (*принадлежит* S);
- конечное и начальное состояния, начальную конфигурацию на ленте и расположение головки машины.

Конечная проблема задается записью конечного количества символов на ленте.

e	s_1	s_2			s_n	e
-----	-------	-------	--	-----	-----	--	-------	-----

Детерминированная машина Тьюринга имеет функцию перехода, которая по комбинации текущего состояния и символа на ленте определяет три вещи:

- символ, который будет записан на ленте;
- направление смещения головки по ленте;
- новое состояние конечного автомата.

Вероятностная машина Тьюринга представляет собой детерминированную машину Тьюринга, имеющую дополнительно

аппаратный источник случайных битов, любое число которых, например, она может «заказать» и «загрузить» на отдельную ленту и потом использовать в вычислениях обычным для МТ образом.

В случае **недетерминированной машины Тьюринга**, комбинация текущего состояния автомата и символа на ленте может допускать несколько переходов. НМТ распознаёт, какой из возможных путей приведёт в допускающее состояние двумя способами:

1. Можно считать, что НМТ – «чрезвычайно удачлива»; то есть всегда выбирает переход, который в конечном счете приводит к допускающему состоянию, если такой переход вообще есть.
2. Можно представить, что в случае неоднозначности перехода (текущая комбинация состояния и символа на ленте допускает несколько переходов) НМТ делится на копии, каждая из которых следует за одним из возможных переходов.

В отличие от ДМТ, которая имеет единственный «путь вычислений», НМТ имеет «дерево вычислений» (в общем случае — экспоненциальное число путей). Говорят, что НМТ допускает входные данные, если какая-нибудь ветвь этого дерева останавливается в допускающем состоянии, иначе НМТ входные данные не допускает.

АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ПРОБЛЕМЫ

Доказательство алгоритмической неразрешимости некоторой задачи принято сводить к классической задаче – «задаче останова».

Теорема. Алгоритмическая неразрешимость. *Не существует общего алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.*

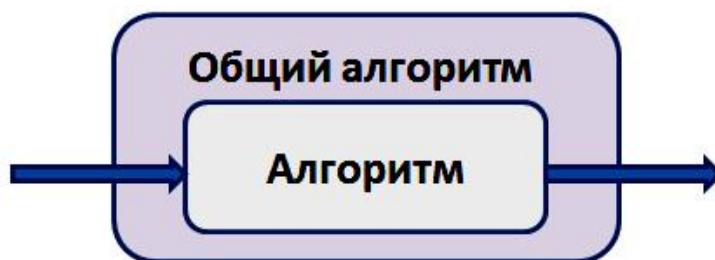


Рис. 2. Алгоритмическая неразрешимость

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, т.е. невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов.

ПРИЧИНЫ (СВОДИМЫЕ К ПРОБЛЕМЕ ОСТАНОВА) ВЕДУЩИЕ К АЛГОРИТМИЧЕСКОЙ НЕРАЗРЕШИМОСТИ

1. Отсутствие общего метода решения задачи

Проблема 1. Распределение числа x в записи числа π ;

$$\pi = 3,141592 \dots, f_9(1) = 5.$$

$$\pi = 48 \arctg(1/18) + 24 \arctg(1/57) - 20 \arctg(1/239).$$

Задача состоит в вычислении функции $f_x(n)$ для произвольного n .

Проблема 2. Вычисление совершенных чисел;

Совершенные числа – это числа, которые равны сумме своих делителей, например:

$$S(1) = 1 = 1$$

$$S(2) = 6 = 1 + 2 + 3$$

$$S(3) = 28 = 1 + 2 + 4 + 7 + 14.$$

Задача вычисления $S(n)$ по произвольно заданному n .

Проблема 3. Десятая проблема Гильберта;

Пусть задан многочлен n -ой степени с целыми коэффициентами $P_n(x)$. Существует ли алгоритм, который определяет, имеет ли уравнение $P_n(x) = 0$ решение в целых числах?

2. Информационная неопределенность задачи

Проблема 4. Позиционирование машины Поста на последнюю помеченную ячейку. Информационная неопределенность – отсутствие информации либо о количестве последовательности ячеек, либо о максимальном расстоянии между ячейками (*кортежами*) – при наличии такой информации задача становится алгоритмически разрешимой.

3. Логическая неразрешимость (в смысле теоремы Гёделя о неполноте)

Проблема 5. Проблема «останова» (см. теорема).

Проблема 6. Проблема эквивалентности алгоритмов. По двум произвольным заданным алгоритмам (*например, по двум машинам Тьюринга*) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.

Проблема 7. Проблема тотальности. По произвольному заданному алгоритму определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи – является ли частичный алгоритм P всюду определённым?

Контрольные вопросы

1. В чем состоят основные принципы работы машины Поста?
2. В чем состоят основные принципы работы машины Тьюринга?
3. В чем разница между машиной Поста и Тьюринга?
4. Охарактеризуйте виды машин Тьюринга.
5. В чем заключается смысл понятия «алгоритмическая неразрешимость» задачи?
6. Перечислите причины, приводящие к алгоритмической неразрешимости задачи
7. Является ли задача нахождения простых чисел алгоритмически неразрешимой? Дайте подробное пояснения.
8. Приведите примеры алгоритмически неразрешимых проблем.

3. АНАЛИЗ АЛГОРИТМОВ

3.1. СРАВНИТЕЛЬНЫЕ ОЦЕНКИ АЛГОРИТМОВ

При использовании алгоритмов для решения практических задач возникает **проблема рационального выбора алгоритма** (рис. 1).



Рис. 1. Проблема рационального выбора алгоритма

Решение проблемы выбора связано с построением системы сравнительных оценок, которая в свою очередь существенно зависит от формальной модели алгоритма.

Для оперирования с формальной моделью алгоритма рассматривается абстрактная машина, включающая:

- процессор, поддерживающий адресную память;
- набор элементарных операций, соотнесенных с языком высокого уровня.

Допущения:

- каждая команда выполняется не более чем за фиксированное время;
- исходные данные алгоритма представляются N машинными словами по α битов каждое.

На входе алгоритма:

$$N_{\alpha} = N * \alpha \text{бит информации}$$

Программа, реализующая алгоритм состоит из M машинных инструкций по β битов:

$$M_{\beta} = M * \beta \text{бит информации}$$

Дополнительные ресурсы абстрактной машины на реализацию алгоритма:

- S_d – память для хранения промежуточных результатов;
- S_{γ} – память для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов).

При решении конкретной задачи, заданной $N + M + S_d + S_{\gamma}$ словами памяти, алгоритм выполняет конечное количество «элементарных» операций абстрактной машины. В связи с этим вводится определение трудоёмкости алгоритма.

Под трудоёмкостью алгоритма $F_a(n)$

- для данного конкретного входа;
- для решения конкретной проблемы (*задачи*);
- в данной формальной системе

понимается количество «элементарных» операций (n), совершаемых алгоритмом.

Комплексный анализ алгоритма может быть выполнен на основе комплексной оценки ресурсов формальной машины, требуемых алгоритмом для решения конкретных задач.

$$\varphi_A = c_1 F_a(n) + c_2 N_{\alpha} + c_3 M_{\beta} + c_4 S_d + c_5 S_{\gamma}$$

Для различных областей применения веса ресурсов c_i будут различны.

3.2. КЛАССИФИКАЦИЯ АЛГОРИТМОВ ПО ВИДУ ФУНКЦИИ ТРУДОЁМКОСТИ

1. Количественно-зависимые по трудоёмкости алгоритмы.

Их функция трудоёмкости только от размерности входных данных и не зависит от их значений:

$$F_a(n), \quad n = f(N).$$

Пример. Алгоритмы для стандартных операций с массивами и матрицами: умножение матриц, умножение матрицы на вектор и т.д.

2. Параметрически-зависимые по трудоемкости алгоритмы.

Их функция трудоемкости определяется конкретными значениями обрабатываемых слов памяти:

$$F_a(n), \quad n = f(p_1, \dots, p_i).$$

У таких алгоритмов на входе два числовых значения: аргумент функции и точность.

Пример. Алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов.

- а) Вычисление x^k последовательным умножением: $F_a(a, k) = F_a(k)$.
- б) Вычисление $e^k = \sum(x^n/n!)$, с точностью до $\varepsilon > 0$: $F_a = F_a(x, \varepsilon)$

3. Количественно-параметрические по трудоемкости алгоритмы.

В большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от их значений:

$$F_a(n), \quad n = f(N, p_1, \dots, p_i)$$

Пример. Алгоритмы численных методов, в которых существует параметрически-зависимый цикл по точности и цикл количественно-зависимый по размерности.

Среди параметрически-зависимых алгоритмов выделяют группу алгоритмов, для которой количество операций зависит от порядка расположения исходных объектов.

Пример.

- алгоритмы сортировки;
- алгоритмы поиска минимума/максимума в массиве.

3.3. АСИМПТОТИЧЕСКИЙ АНАЛИЗ ФУНКЦИЙ

Цель анализа трудоёмкости алгоритмов – нахождение оптимального алгоритма для решения данной задачи (рис. 2).

Критерий оптимальности – количество элементарных операций, которые необходимо выполнить для решения задачи с помощью данного алгоритма.



Рис. 2. Анализ трудоёмкости алгоритмов

$$\Theta = \frac{S_{\text{и}}}{S_{\text{и}} + S_r} \rightarrow 1, S_{\text{и}} \rightarrow \min$$

$$\Theta_m = \frac{S_{\text{и}}}{S_{\text{и}} + S_r/m} = \frac{mS_{\text{и}}}{mS_{\text{и}} + S_r} \rightarrow 1$$

Цель асимптотического анализа - сравнение затрат ресурсов системы различными алгоритмами, предназначенными для решения:

- одной и той же задачи;
- при больших объемах входных данных.

Используемая в асимптотическом анализе **оценка функции трудоёмкости, называется сложностью алгоритма** и позволяет определить, как быстро растет трудоёмкость алгоритма с увеличением объема данных.

В асимптотическом анализе используются обозначения, позволяющие показать скорость роста функции:

1. Оценка Θ (тетта)
2. Оценка O (О большое)
3. Оценка Ω (Омега)

1. Оценка Θ (тетта)

Пусть $f(n)$ и $g(n)$ – положительные функции положительного аргумента, $n \in \mathbb{N}$, тогда (рис. 3):

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{существуют положительные константы } c_1, c_2 \text{ и } n_0 \\ \text{такие что } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ для всех } n \geq n_0 \end{array} \right\}$$

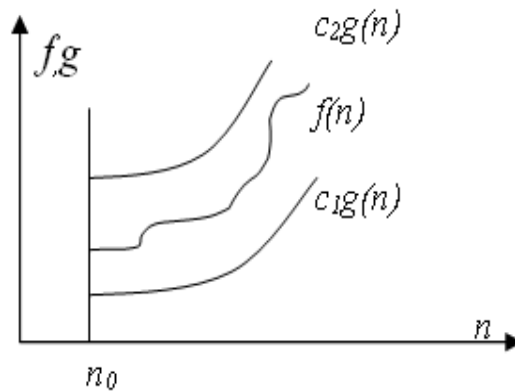


Рис. 3. Оценка Θ

Обычно говорят, что при этом функция $g(n)$ является асимптотически точной оценкой функции $f(n)$, т.к. по определению функция $f(n)$ не отличается от функции $g(n)$ с точностью до постоянного множителя.

Примеры.

$$1) f(n) = 4n^2 + n \ln(n) + 174, f(n) \rightarrow \Theta(n^2);$$

$$2) f(n) \rightarrow \Theta$$

Следующая запись означает, что $f(n)$ или равна константе, не равной нулю, или $f(n)$ ограничена константой на Θ :

$$f(n) = 7 + 1/n \rightarrow \Theta.$$

Пример.

$$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Неравенства выполняются, если выбрать:

$$n \geq 7, c_1 \leq 1/14$$

$$n \rightarrow \infty, c_2 \geq 1/2$$

Таким образом:

$$\left(\frac{2}{7}\right) n^2 \leq n^2/2 - 3n \leq \left(\frac{1}{2}\right) n^2$$

Пример.

$$c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2, a, b, c > 0,$$

$$an^2 \leq an^2 + bn + c \leq (a + b + c)n^2.$$

Для произвольных a, b, c :

$$c_1 = \min\{a, a + b + c\}, c_2 = \max\{a, a + b + c\}$$

2. Оценка O (O большое)

В отличие от оценки Θ , оценка O требует только, чтобы функция $f(n)$ не превышала $g(n)$, начиная с некоторого $n > n_0$, с точностью до постоянного множителя (рис. 4):

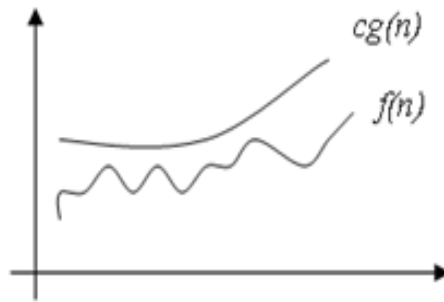


Рис. 4. Оценка O

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leq f(n) \leq cg(n) \text{ для всех } n \geq n_0 \end{array} \right\}$$

Запись $O(g(n))$ обозначает класс функций, которые растут **не быстрее, чем** функция $g(n)$ с точностью до постоянного множителя, поэтому иногда говорят, что $g(n)$ **мажорирует** функцию $f(n)$.

Например, для всех функций:

$$\begin{aligned} f(n) = \frac{1}{n}, \quad f(n) = 12, \quad f(n) = 3n + 17, \\ f(n) = n \cdot \ln(n), \quad f(n) = 6n^2 + 24n + 77 \end{aligned}$$

будет справедлива оценка $O(n^2)$.

3. Оценка Ω (Омега)

Оценка Ω является оценкой снизу, т.е. определяет класс функций, которые растут **не медленнее, чем** $g(n)$ с точностью до постоянного множителя (рис. 5):

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \text{существуют положительные константы } c \text{ и } n_0 \\ \text{такие что } 0 \leq cg(n) \leq f(n) \text{ для всех } n \geq n_0 \end{array} \right\}$$

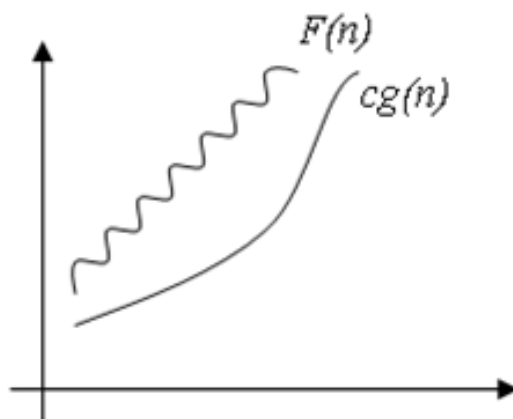


Рис. 5. Оценка Ω

Теорема. Для любых двух функций $f(n)$ и $g(n)$ соотношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$ (рис. 6).

Например, запись $\Omega(n \cdot \ln(n))$ обозначает класс функций, которые растут не медленнее, чем $g(n) = n \cdot \ln(n)$.

В этот класс попадают все полиномы степени $n > 2$ и все степенные функции с основанием большим единицы.

Есть примеры, когда для пары функций не выполняется ни одно из асимптотических соотношений. Например, $f(n) = n^{1+\sin(n)}$ и $g(n) = n$.

В асимптотическом анализе алгоритмов разработаны специальные методы получения асимптотических оценок, особенно для класса рекурсивных алгоритмов. Θ оценка является предпочтительной.

Знание асимптотики поведения функции трудоемкости алгоритма (сложности), дает возможность делать прогнозы по выбору более рационального с этой точки зрения алгоритма для больших размерностей исходных данных.

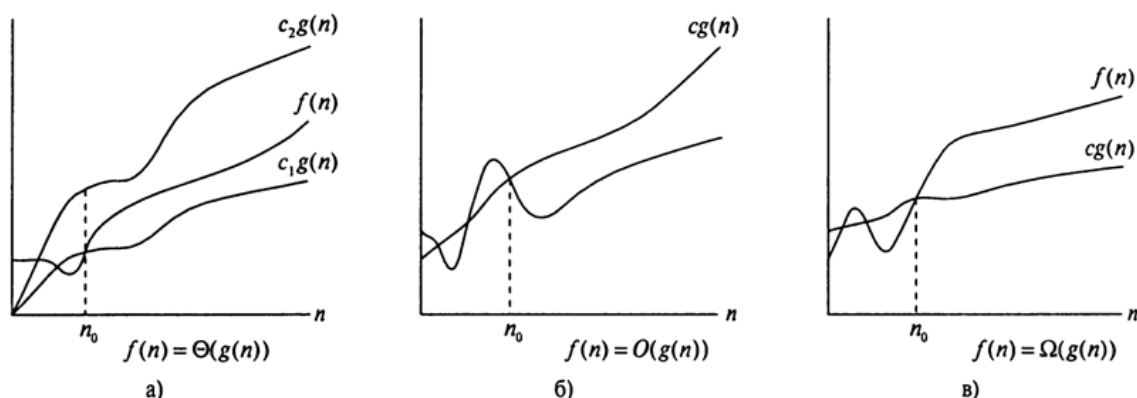


Рис. 6. а, б, в – графические примеры Θ , O и Ω обозначений.

В качестве n_0 используется минимальное возможное значение, т.е. при любом значении $n > n_0$ соответствующая оценка также будет выполняться.

3.4. ТРУДОЕМКОСТЬ АЛГОРИТМОВ И ВРЕМЕННЫЕ ОЦЕНКИ

Элементарные операции на языке записи алгоритмов

При получении функции трудоемкости алгоритма выделяют следующие «элементарные» операции:

1. Простое присваивание: $a \leftarrow b$;
2. Одномерная индексация $a[i]$: (адрес $(a)+i \cdot [\text{длина элемента}]$);
3. Арифметические операции: $(*, /, -, +)$;
4. Операции сравнения: $a < b$;
5. Логические операции (or, and, not).

Из этих операций в анализе исключают команду перехода по адресу, считая ее связанной с операцией сравнения в конструкции ветвления.

1. Конструкция «Следование»

Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом (рис. 7).

$$F_{\text{«следование»}} = f_1 + \dots + f_k, \text{ где } k - \text{количество блоков.}$$

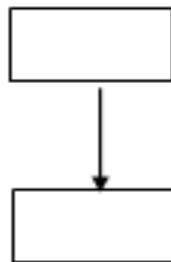


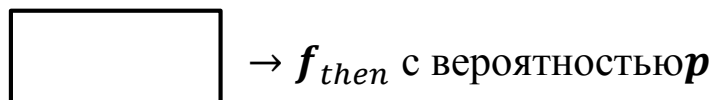
Рис. 7. Конструкция «Следование»

2. Конструкция «Ветвление»

Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения переходов на блоки «Then» и «Else» и определяется как (рис. 8):

$$F_{\text{«ветвление»}} = f_{\text{then}} * p + f_{\text{else}} * (1 - p)$$

if (l) then



else

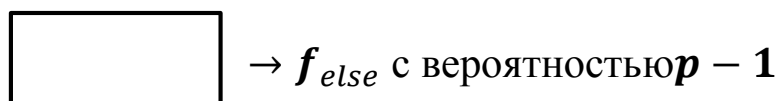


Рис. 8. Конструкция «Ветвление»

3. Конструкция «Цикл 1». Количественно-зависимый алгоритм

После сведения конструкции к элементарным операциям ее трудоемкость определяется как (рис. 9):

$$F_{\langle\langle\text{цикл}\rangle\rangle} = 1 + 3 * N + N * f_{\langle\langle\text{тело цикла}\rangle\rangle}$$

for $i \leftarrow 1$ **to** N



end

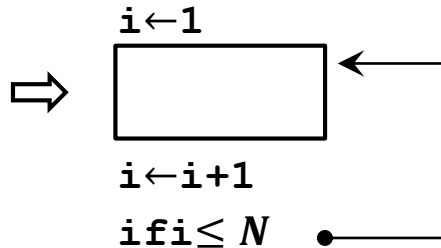


Рис. 9. Конструкция «Цикл 1». Количественно-зависимый алгоритм

4. Конструкция «Цикл2». Количественно-параметрический алгоритм

```
for 1 ← 1 to N
<>           (n1)
  If <условие(i)>
  <>           (n2)
  else
  <>           (n3)
  break
end
end
```

Худший случай	$F_x(n) = 1 + 3N + Nf_{\text{внут}}$	$\Theta(Nf_{\text{внут}})$
	$f_{\text{внут}} = n_1 + (n_{\text{усл}} + n_2)$	
Лучший случай	$F_l(n) = 1 + n_1 + (n_{\text{усл}} + n_3)$	$\Theta(n)$
Средний случай	$F_c(n) = (F_x + F_l) / 2$?	

3.5. ПРИМЕРЫ АНАЛИЗА ПРОСТЫХ АЛГОРИТМОВ

Пример 1. Задача суммирования элементов квадратной матрицы

<>	
$Sum = 0$	1
$For\ i = 1\ to\ n$	n
$For\ j = 1\ to\ n$	n
$Sum = Sum + A(i, i)$	4
$Next\ j$	
$Next\ i$	
$Print\ (Sum)$	
<>	

Алгоритм выполняет одинаковое количество операций при фиксированном значении n , и, следовательно, является **количественно-зависимым**.

Применение методики анализа конструкции «Цикл 1» дает:

Внутренний: $f_1(n) = 1 + 3n + 4n$

Внешний: $f_2(n) = 1 + 3n + nf_1(n)$

Окончательно:

$$F(n) = 1 + 1 + 3n + n(1 + 3n + 4n) = 2 + 4n + 7n^2 = \Theta(n^2)$$

Под n понимается линейная размерность матрицы, в то время, как на вход алгоритма подается n^2 значений.

Пример 2. Задача поиска максимума в массиве

<>	
$Max = S(1)$	2
$For\ i = 2\ to\ n$	$n-1$
$If\ Max < S(i)$	$2 (< u\ S[i])$
$Max = S(i)$	$2 (= u\ S[i])$
$end\ if$	
$Next\ i$	
$Print\ (Max)$	
<>	

Данный алгоритм является **количественно-параметрическим**, поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая.

Худший случай

Максимальное количество переприсваиваний максимума (на

каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию.

Трудоёмкость алгоритма в этом случае равна

$$F(n) = 2 + 1 + 3(n - 1) + (n - 1)(2 + 2) = 7n - 4 \rightarrow \Theta(n).$$

Лучший случай

Минимальное количество переприсваиваний, если максимальный элемент расположен на первом месте в массиве. Трудоёмкость алгоритма в этом случае равна

$$F(n) = 2 + 1 + 3(n - 1) + (n - 1)(2) = 5n - 2 \rightarrow \Theta(n).$$

Средний случай

Элементарное усреднение даёт

$$F_c(n) = (F_x(n) + F_l(n)) / 2 = 6n - 3 \rightarrow \Theta(n).$$

3.6. ПЕРЕХОД К ВРЕМЕННЫМ ОЦЕНКАМ

Сравнение двух алгоритмов по их функции трудоёмкости вносит некоторую ошибку в получаемые результаты.

Основные причины этой ошибки:

- различная частотная встречаемость элементарных операций;
- различие во времени их выполнения на реальном процессоре.

Таким образом, возникает задача перехода от функции трудоёмкости к оценке времени работы алгоритма на конкретном процессоре.

Дано: $F(A)$ - трудоёмкость алгоритма. Требуется определить время работы программной реализации алгоритма $T(A)$.

Основные проблемы:

- неадекватность формальной системы записи алгоритма и реальной системы команд процессора;

- наличие архитектурных особенностей существенно влияющих на наблюдаемое время выполнения программы (конвейер; кэширование памяти; предвыборка команд, данных и т.д.);
- различные времена выполнения реальных машинных команд;
- различие во времени выполнения однородных (однотипных) команд в зависимости от значений операндов и типов данных;
- неоднозначности компиляции исходного текста, обусловленные как самим компилятором, так и его настройками.

Методики перехода к временным оценкам:

1. Пооперационный анализ
2. Метод Гиббсона
3. Метод прямого определения среднего времени

1. Пооперационный анализ

1-ый шаг. Получение пооперационной функции трудоемкости для каждой из элементарных операций с учетом типов данных $F_{A,оп,i}(n)$.

2-ой шаг. Экспериментальное определение среднего времени выполнения данной элементарной операции на конкретной вычислительной машинет $t_{оп,i,ср}$.

Ожидаемое время выполнения рассчитывается как сумма произведений пооперационной трудоемкости на средние времена операций:

$$T_A(n) = \sum_i F_{A, оп, i}(n) \cdot t_{оп, ср, i}$$

2. Метод Гиббсона

Метод предполагает переход к временным оценкам на основе принадлежности решаемой задачи к одному из следующих типов:

- задачи научно-технического характера с преобладанием операций с операндами действительного типа;
- задачи дискретной математики с преобладанием операций с операндами целого типа
- задачи баз данных с преобладанием операций с операндами строкового типа

Далее на основе анализа множества реальных программ для решения соответствующих типов задач определяется частотная встречаемость операций (рис. 11), создаются соответствующие тестовые программы, и определяется среднее время на операцию в данном типе задач – $t_{ср.тип.задач}$.

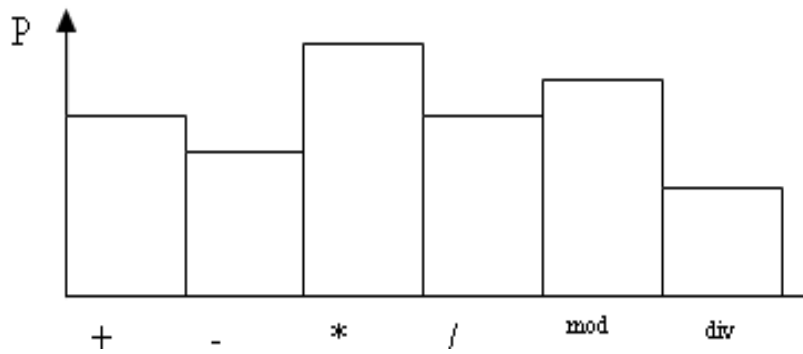


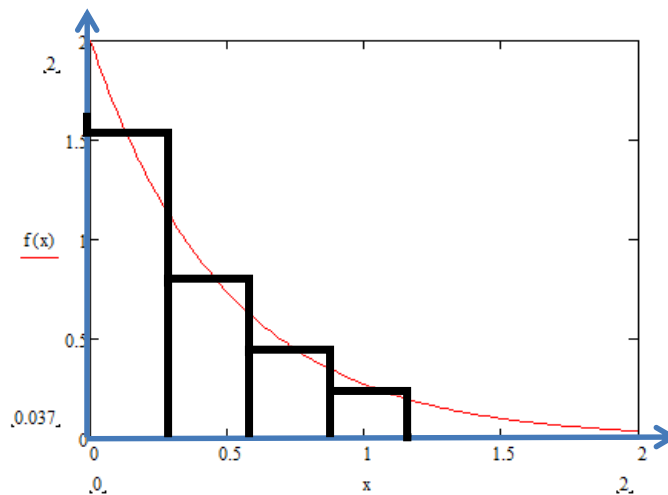
Рис. 11. Возможный вид частотной встречаемости операций

На основе полученной информации оценивается общее время работы алгоритма в виде:

$$t_{\text{оп, ср}} = \sum_i p_{\text{тип оп, i}} \cdot t_{\text{оп, i}}$$

$$T_A(n) = F_{A, \text{тип зад}}(n) \cdot t_{\text{оп, ср}}$$

Расчетные задачи



$$p(t) = a \exp(-at)$$

$$t_{\text{тип.ср}} = \int_0^{\infty} t p(t) dt = -\frac{(1+at)}{a} \exp(-at) \Big|_0^{\infty} = \frac{1}{a}$$

Смешанные задачи

$$p(t) = \frac{1}{b}$$

$$t_{\text{тип.ср}} = \int_0^b t p(t) dt = \frac{t^2}{2b} \Big|_0^b = \frac{b}{2}$$

Символьно-логические задачи

$$p(t) = \frac{1}{\sqrt{2\pi\sigma_t}} \exp\left(-\frac{(t-a)^2}{2\sigma_t^2}\right)$$

$$t_{\text{тип.ср}} = \int_0^{\infty} tp(t)dt = a$$

3. Метод прямого определения среднего времени

В этом методе проводится совокупный анализ по трудоемкости:

- определяется $F(n)$;
- определяется среднее время работы данной программы $T_{\text{ср}}$ на основе прямого эксперимента для различных значений размеров входных данных N ;
- рассчитывается среднее время на обобщенную элементарную операцию, порождаемое данным алгоритмом, компилятором и компьютером:

$$T_{\text{ср}} = \frac{\sum_i t_{N,i} N_{\text{оп},i}}{\sum_i N_{\text{оп},i}}$$

$$t_{A,\text{ср}} = \frac{T_{\text{ср}}}{n}$$

$$T(n) = F_A(n) t_{A,\text{ср}}$$

Эта формула может быть распространена на другие значения размерности задачи при условии устойчивости среднего времени по N .

Примеры пооперационного временного анализа

Пооперационный анализ позволяет выявить тонкие аспекты рационального применения того или иного алгоритма решения задачи.

Пример. Задача умножения двух комплексных чисел:

$$\mathbf{A1:} (a + b i) * (c + d i) = (ac - bd) + i(ad + bc) = e + i f$$

$$\mathbf{A2:} (a + b i) * (c + d i) = z_1 - z_2 + i(z_1 + z_3) = e + i f,$$

$$z_1 = c(a + b), \quad z_2 = b(d + c), \quad z_3 = a(d - c)$$

1. Алгоритм A1 (прямое вычисление e, f – 4 умножения)

MultiComplex1(a, b, c, d; e, f)

$$e \leftarrow a * c - b * d$$

$f_{A1} = 8$ операций

$$f \leftarrow a * d + b * c$$

$f_* = 4$ операций

Return(e, f)

$f_{+-} = 2$ операций

End.

$f_{\leftarrow} = 2$ операций

2. Алгоритм A2 (вычисление e, f за 3 умножения)

MultComplex2(a, b, c, d; e, f)

$z_1 \leftarrow c * (a + b)$

$z_2 \leftarrow b * (d + c)$

$z_3 \leftarrow a * (d - c)$

$f_{A2} = 13$ операций

$e \leftarrow z_1 - z_2$

$f_* = 3$ операций

$f \leftarrow z_1 + z_3$

$f_+ = 5$ операций

Return(e, f)

$f_- = 5$ операций

End.

По совокупному количеству элементарных операций алгоритм A2 уступает алгоритму A1, однако в реальных компьютерах операция умножения требует большего времени, чем операция сложения.

Введя параметры q и r , устанавливающие соотношения между временами выполнения операции умножения и сложения, получим временные оценки двух алгоритмов к времени выполнения операции сложения/вычитания (t_+):

$$t_* = q * t_+, q > 1;$$

$t_- = r * t_+, r < 1$, тогда приведенные к t_+ временные оценки имеют вид:

$$T_{A1} = 4 * q * t_+ + 2 * t_+ + 2 * r * t_+ = t_+ * (4 * q + 2 + 2 * r);$$

$$T_{A2} = 3 * q * t_+ + 5 * t_+ + 5 * r * t_+ = t_+ * (3 * q + 5 + 5 * r).$$

Равенство времен будет достигнуто при условии $4 * q + 2 + 2 * r = 3 * q + 5 + 5 * r$, откуда $q = 3 + 3r$ и, следовательно, при $q > 3 + 3r$ алгоритм A2 будет работать более эффективно.

$$\Delta T = (q - 3 - 3 * r) * t_+$$

При умножении в прямом коде:

n – число разрядов множителя,

p_i – вероятность появления 1 в множителе

$$t_* = \sum_{i=1}^n (t_- + p_i t_+) = n t_- + t_+ \sum_{i=1}^n p_i$$

Худший случай

$$\sum_{i=1}^n p_i = n, \quad t_- \ll t_+$$

$$t_{A1} = 4n(t_- + t_+) + (t_- + t_+) + 2t_- = (4n + 2)(t_- + t_+)$$

$$t_{A2} = 3n(t_- + t_+) + 2t_- + 3t_+ + 5t_- = (3n + 5)(t_- + t_+)$$

$$\text{A2 лучше если: } 4n + 2_+ > 3n + 5, \quad n > 3$$

Лучший случай $p = [1000 \dots 0]$

$$t_{A1} = 4(nt_{\leftarrow} + t_{+}) + t_{-} + t_{+} + 2t_{\leftarrow} = (4n + 2)t_{\leftarrow} + 6t_{+}$$

$$t_{A2} = 3(nt_{\leftarrow} + t_{+}) + 2t_{-} + 3t_{+} + 5t_{\leftarrow} = (3n + 5)t_{\leftarrow} + 8t_{+}$$

A2 лучше если: $(n - 3)t_{\leftarrow} - 2t_{+} > 0$

Пример. Анализ алгоритма сортировки вставками (рис. 12).

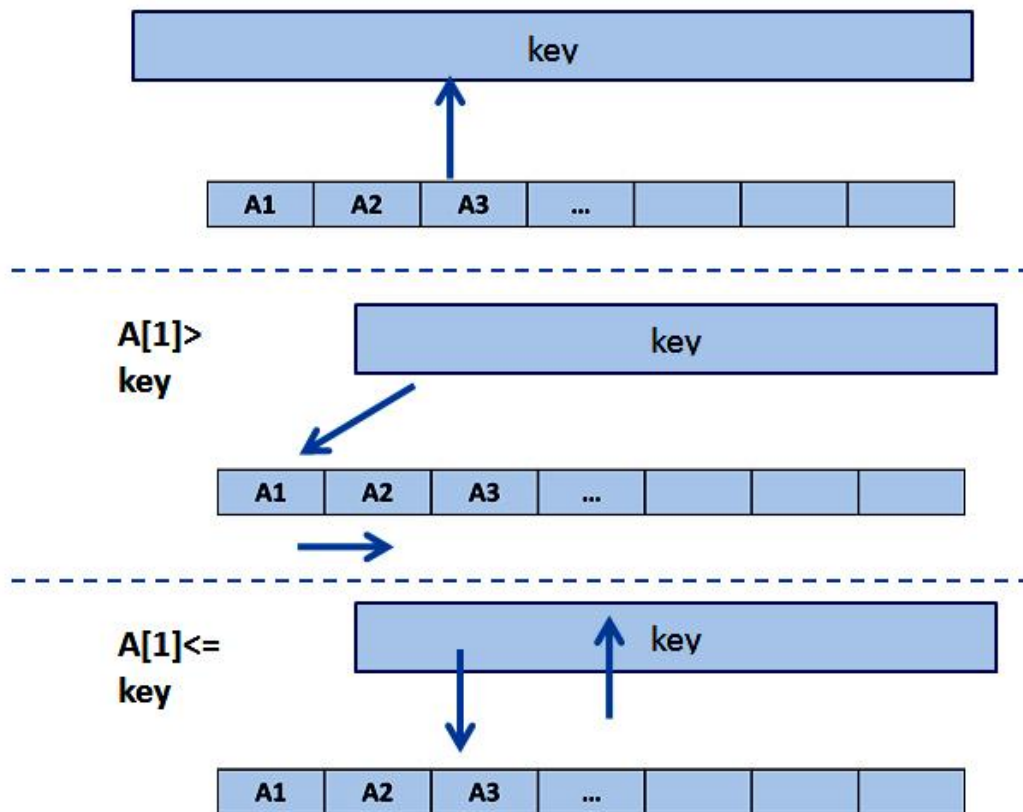


Рис. 12. Схема анализа алгоритма сортировки вставками

INSERTION_SORT(<i>A</i>)		время	количество раз
1	for $j \leftarrow 2$ to $\text{length}[A]$	c_1	n
2	do $\text{key} \leftarrow A[j]$	c_2	$n - 1$
3	▷ Вставка элемента $A[j]$ в отсортированную последовательность $A[1..j - 1]$.	0	$n - 1$
4	$i \leftarrow j - 1$	c_4	$n - 1$
5	while $i > 0$ and $A[i] > \text{key}$	c_5	$\sum_{j=2}^n t_j$
6	do $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7	$i \leftarrow i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8	$A[i + 1] \leftarrow \text{key}$	c_8	$n - 1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1).$$

Лучший случай: когда все элементы массива уже отсортированы (отсутствуют временные затраты c_6 и c_7):

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

$$T(n) = a(t)n + b(t).$$

Худший случай: когда элементы массива отсортированы в обратном порядке, $t_j = j$:

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1,$$

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2},$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

$$T(n) = a(t)n^2 + b(t)n + c(t)$$

В среднем случае необходимо совершить $j/2$ проверок, поэтому оценка та же.

3.7. ТЕОРЕТИЧЕСКИЙ ПРЕДЕЛ ТРУДОЕМКОСТИ АЛГОРИТМОВ

Дано:

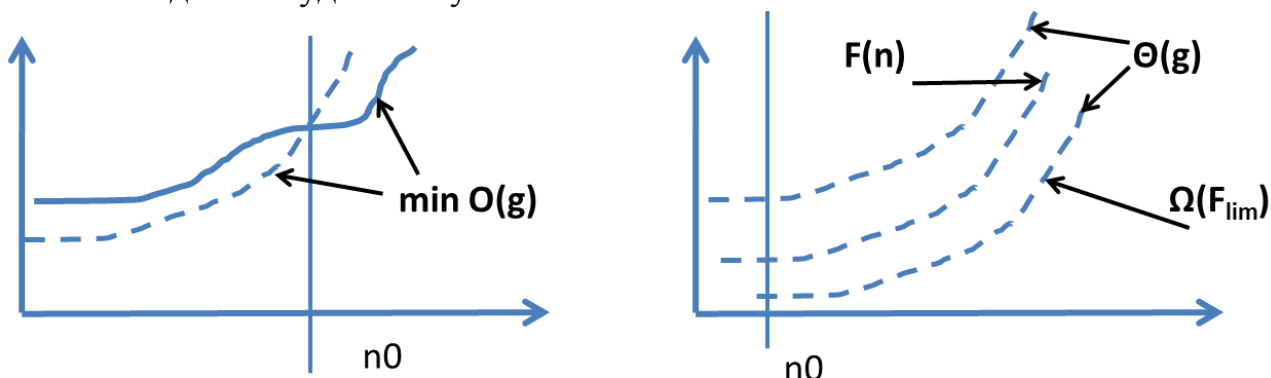
- алгоритмически разрешимая задача
- множество алгоритмов

Можно получить оценку трудоемкости этих алгоритмов в худшем случае: $f_i(D) = O(g(D_a))$, где D_a – множество конкретных проблем данной задачи, i – номер алгоритма.

Вопросы:

- существует ли функциональный нижний предел для $g(D_a)$?

- если «да», то существует ли алгоритм, решающий задачу с такой трудоемкостью в худшем случае?
- какова оценка сложности самого «быстрого» алгоритма решения задачи в худшем случае?



Ответ на третий вопрос - это оценка самой задачи, а не какого-либо алгоритма ее решения.

Функциональный теоретический нижний предел трудоемкости задачи в худшем случае:

$$F_{lim} = \min\{\Theta(F_a(D))\}.$$

Если можно на основе теоретических рассуждений доказать существование и получить оценивающую функцию F_{lim} , то можно утверждать, что

любой алгоритм, решающий данную задачу работает не быстрее, чем с оценкой в худшем случае: $F_a(D) = \Omega(F_{lim})$.

Примеры:

1. Просканировать поверхность требует время, линейно зависящее от его площади $\Theta(A)$.
2. Для задачи **поиска максимума в массиве** $A = (a_1, \dots, a_n)$ должны быть просмотрены все элементы, и $F_{lim} = \Theta(n)$.
3. Задача «**найти имя в телефонной книге**» требует время, логарифмически зависящее от количества записей: $\Theta(\log_2(n))$.
4. Для задачи **умножения матриц** можно сделать предположение, что необходимо выполнить некоторые арифметические операции со всеми исходными данными, что приводит к оценке $F_{lim} = \Theta(n^2)$.

Лучший алгоритм умножения матриц имеет оценку $\Theta(n^{2.376})$.

Расхождение между теоретическим пределом и оценкой лучшего известного алгоритма позволяет предположить, что

- либо существует, но еще не найден более быстрый алгоритм умножения матриц,
- либо оценка $\Theta(n^2)$ должна быть доказана, как теоретический предел.

Алгоритм Штрассена (1969)

$$O(n^{\log_2 7}) \approx O(n^{2.807})$$

Если размер умножаемых матриц n не является натуральной степенью двойки, то дополняют исходные матрицы дополнительными нулевыми строками и столбцами. При этом получают удобные для рекурсивного умножения размеры, но теряют в эффективности за счёт дополнительных ненужных умножений.

Матрицы A , B и C делятся на равные по размеру блочные матрицы:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$P_1 := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$P_2 := (A_{2,1} + A_{2,2})B_{1,1}$$

$$P_3 := A_{1,1}(B_{1,2} - B_{2,2})$$

$$P_4 := A_{2,2}(B_{2,1} - B_{1,1})$$

$$P_5 := (A_{1,1} + A_{1,2})B_{2,2}$$

$$P_6 := (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

$$P_7 := (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

$$C_{1,1} = P_1 + P_4 - P_5 + P_7$$

$$C_{1,2} = P_3 + P_5$$

$$C_{2,1} = P_2 + P_4$$

$$C_{2,2} = P_1 - P_2 + P_3 + P_6$$

Итерационный процесс продолжается n раз до тех пор, пока матрицы $C_{i,j}$ не вырождаются в числа.

На практике итерации останавливают при размере матриц от 32 до 128 и далее используют обычный метод умножения матриц, так как алгоритм Штрассена теряет эффективность по сравнению с обычным на малых матрицах из-за дополнительных копирований массивов.

Алгоритм Пана (1978): $\Theta(n^{2.78041})$.

Алгоритм Бини (1979): $\Theta(n^{2.7799})$.

Алгоритмы Шёнхаге (1981): $\Theta(n^{2.695})$, $\Theta(n^{2.5161})$.

Алгоритм Копперсмита-Винограда (1990): $\Theta(n^{2.376})$.

На сегодняшний день алгоритм Копперсмита-Винограда считается наиболее асимптотически быстрым, но он эффективен только на очень больших матрицах и поэтому применяется редко.

В 2003 Кох и др. рассмотрели в своих работах алгоритмы Штрассена и Копперсмита-Винограда в контексте теории групп.

Показана возможность существования алгоритмов умножения матриц со сложностью $\Theta(n^2)$.

3.8. РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ

Если алгоритм рекуррентно вызывает сам себя, время его работы можно описать с помощью рекуррентного соотношения.

***Рекуррентное соотношение** (recurrence) — это уравнение или неравенство, описывающее функцию с использованием ее самой, с меньшими аргументами.*

***Рекурсия** — приём программирования, который позволяет разбивать задачу на меньшие подзадачи, каждая из которых решается с помощью одного и того же алгоритма.*

Пример 1. Сортировка слиянием(mergesort) — алгоритм сортировки, который упорядочивает структуры данных, доступ к элементам которых можно получать только последовательно (*списки, потоки и т.п.*) (рис. 1).

Разделение: сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит $n/2$ элементов.

Покорение: сортировка каждой полученной последовательности методом слияния.

Комбинирование: объединение двух отсортированных последовательностей для получения окончательного результата.

Рекурсивное разбиение задачи на меньшие подзадачи происходит до тех пор, пока размер массива не достигнет единицы (*любой массив длины 1 можно считать упорядоченным*).

Нетривиальным этапом является **соединение двух упорядоченных массивов** в один.

Алгоритм слияния (на примере колоды карт):

1. Пусть имеем две стопки карт, лежащих рубашками вниз.
2. В каждой из этих стопок карты идут сверху вниз в неубывающем порядке.
3. На каждом шаге берём меньшую из двух верхних карт и кладём её (*рубашкой вверх*) в результирующую стопку.

4. Когда одна из оставшихся стопок становится пустой, мы добавляем все оставшиеся карты второй стопки к результирующей стопке.

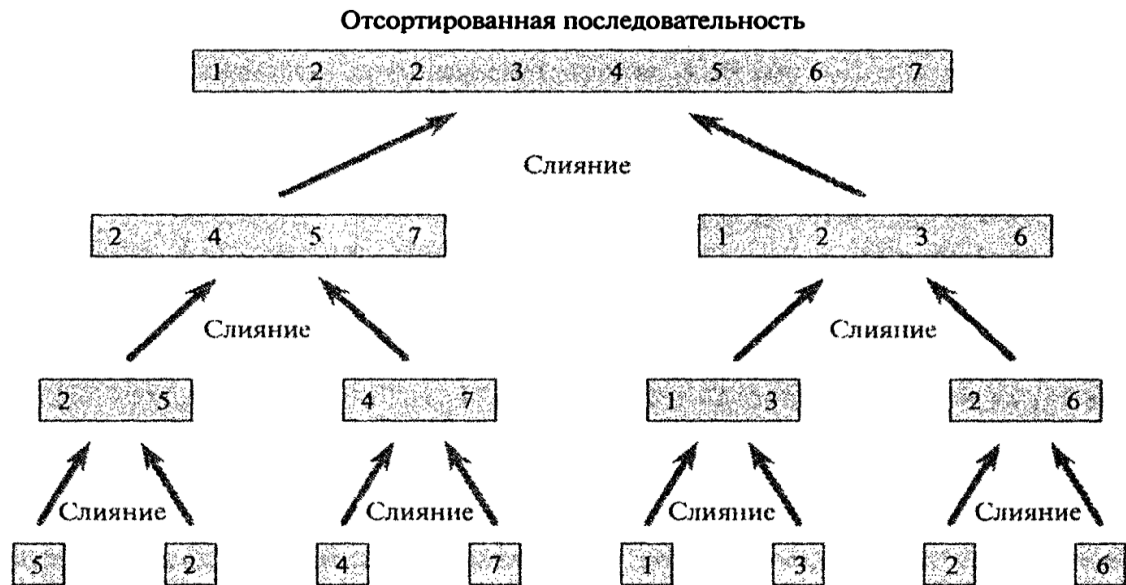


Рис.1. Сортировка слиянием

Время работы процедуры Merge_SortT(n) в самом неблагоприятном случае описывается с помощью следующего **рекуррентного соотношения**:

Разбиение. В ходе разбиения определяется, где находится середина подмассива. Эта операция длится фиксированное время поэтому $D(n) = \Theta(1)$ (рис. 2, а).

Покорение. Рекурсивно решаются две подзадачи, объем каждой из которых составляет $n/2$. Время решения этих подзадач равно $2 \cdot T(n/2)$ (рис. 2, б).

Комбинирование. Процедура Merge в n -элементном подмассиве выполняется в течение времени $\Theta(n)$, поэтому $C(n) = \Theta(n)$ (рис. 2, в).

$$T(n) = \begin{cases} \Theta(1) & \text{при } n = 1, \\ 2T(n/2) + \Theta(n) & \text{при } n > 1. \end{cases}$$

$$T(n) = \begin{cases} c & \text{при } n = 1, \\ 2T(n/2) + cn & \text{при } n > 1, \end{cases}$$

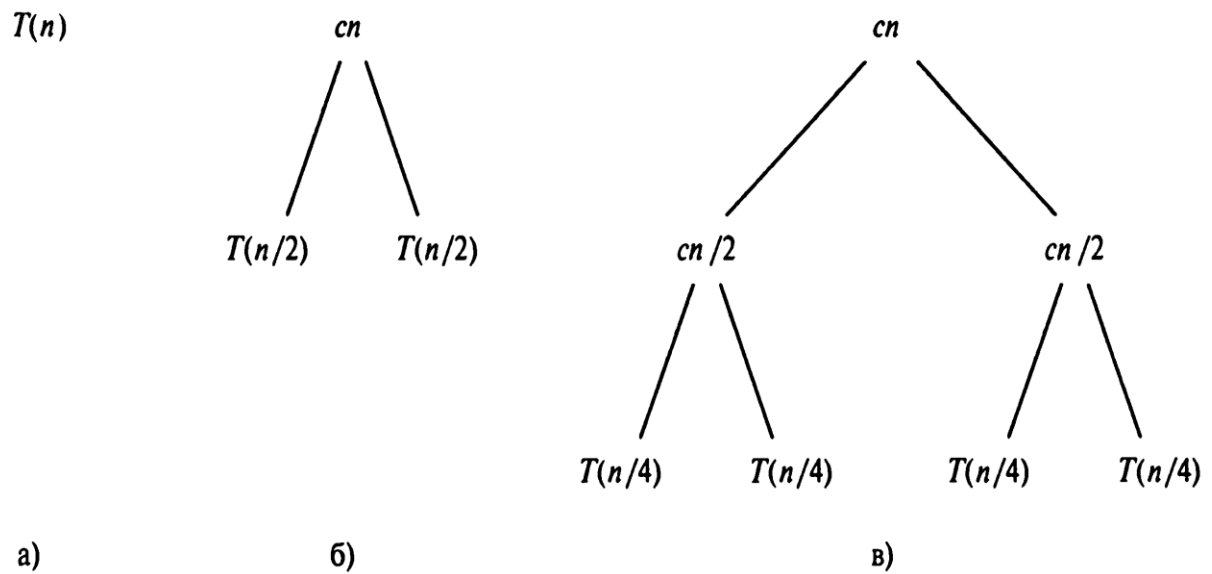


Рис. 2. а, б, в – Методы разбиения, покорения и комбинирования
Асимптотическая функция временных трудозатрат алгоритма сортировки
слиянием (рис. 3):

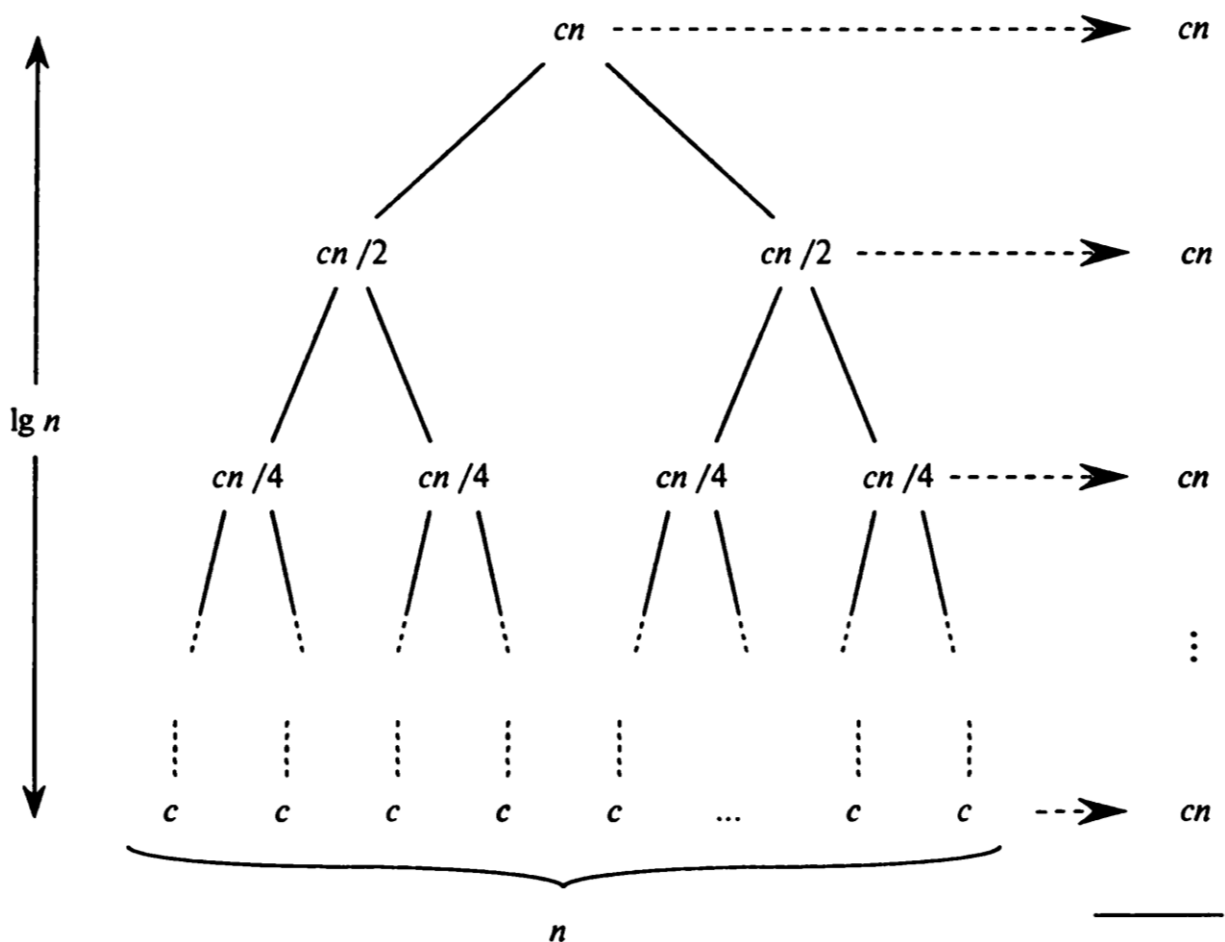


Рис. 3. Асимптотическая функция временных трудозатрат алгоритма

$$2 \cdot \log_2 n = n.$$

Всего уровней $(\log_2 n + 1)$, $T(n) = cn(\log_2 n + 1) = O(n \cdot \log_2 n)$

Такой подход получил название **построения дерева рекурсии**.

Пример 2. $T(n) = 3T(n/4) + cn^2$ (рис. 4).

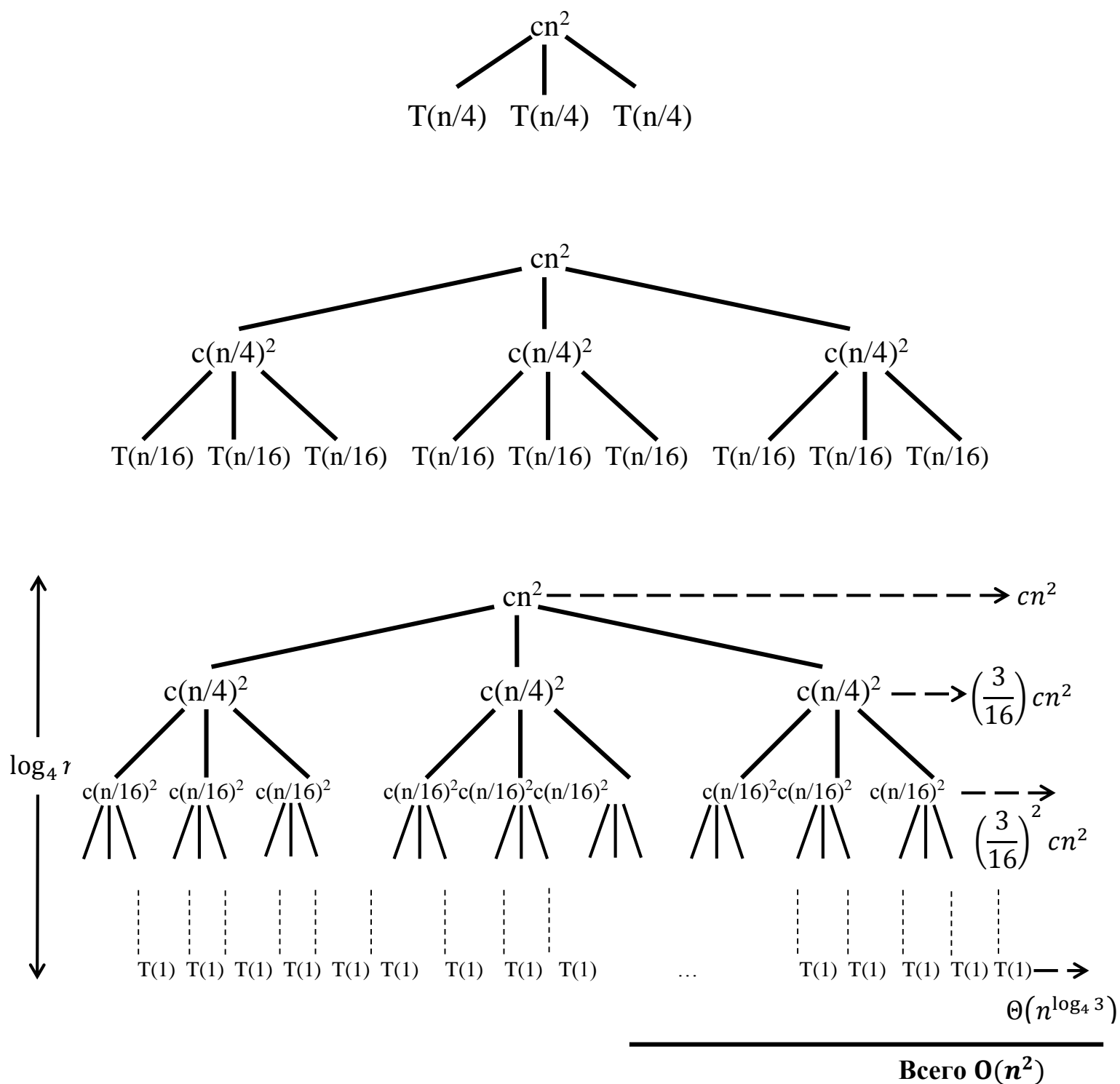


Рис. 4. Построение дерева рекурсии

Теорема. Пусть $a > 1$ и $b > 1$ – константы, $f(n)$ – произвольная функция, а $T(n)$ – функция, определенная на множестве неотрицательных целых чисел с помощью рекуррентного соотношения $(n/b == \lfloor n/b \rfloor)$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Тогда асимптотическое поведение функции $T(n)$ можно выразить следующим образом:

1. Если $F(n) = O(n^{\log_b a - \varepsilon})$ для некоторой константы $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.

2. Если $F(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$.

3. Если $F(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторой константы $\varepsilon > 0$ и, если $af(n/b) \leq cf(n)$ для некоторой константы $c < 1$ и всех достаточно больших n , то $T(n) = \Theta(f(n))$.

Чтобы использовать основной метод, достаточно определить, какой из частных случаев основной теоремы (если такой есть) применим в конкретной задаче, а затем записать ответ.

Пример 3.

$$T(n) = 9T(n/3) + n, \quad f(n) = n, \quad a = 9, \quad b = 3.$$

$f(n) = n = O(n^{\log_b a - \varepsilon}) = O(n^{\log_3 9 - 1}) = O(n)$, то приходим к случаю 1.

$$T(n) = \Theta(n^2).$$

Пример 4.

$$T(n) = T(2n/3) + 1, \quad f(n) = 1, \quad a = 1, \quad b = 3/2.$$

$f(n) = 1 = n^{\log_{3/2} 1} = n^0$, то приходим к случаю 2.

$$T(n) = \Theta(\log_2 n).$$

Пример 5.

$$T(n) = 3T(n/4) + n \cdot \lg n.$$

$$f(n) = n^{\log_4 3}.$$

$$T(n) = n^{0.972} \log_2 n = \Theta(n \log_2 n).$$

Более общий случай разбиения задачи на существенно неравные подзадачи:

$$T(n) = \sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) + f(n)$$

$$T(n) = \Theta(n^p) + \Theta\left(n^p \int_{n'}^n \frac{f(x)}{x^{p+1}} dx\right)$$

$$a_i > 0, \quad b_i > 1, \quad \sum_{i=1}^k a_i b_i^{-p} = 1.$$

Пример 6. Вычисление факториала (рис. 5).

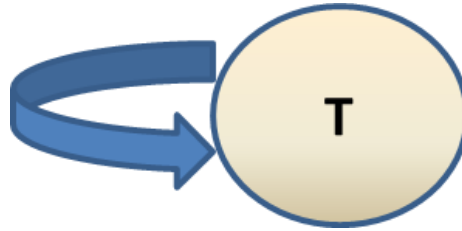


Рис. 5. Прямая рекурсия

$$F(0) = 1$$

$$F(n) = n * f(n - 1)$$

$$T(n) = T(n - 1) + \Theta(1) = cn$$

Пример 7. Наибольший общий делитель

FunctionGCD(A, B)

If A Mod B = 0

' Делится ли B на A нацело?

GCD = B

' Да. Процедура завершена.

Else

GCD = GCD(B, A Mod B) ' Нет. Рекурсия.

$T_0 = T(mod) + 1 + 1 = F(mod) + 2$ – трудоемкость до рекурсии

$T(n) = T(n - 1) + T(mod) + 1 = n [T(mod) + 1] + 1$ – трудоемкость при достижении рекурсии.

Косвенная рекурсия (рекурсивная процедура вызывает другую процедуру, которая, в свою очередь, вызывает первую) (рис. 6).

$$T1(n) = T2(n - 1) + \Theta(1)$$

$$T2(n) = T1(n - 1) + \Theta(1)$$

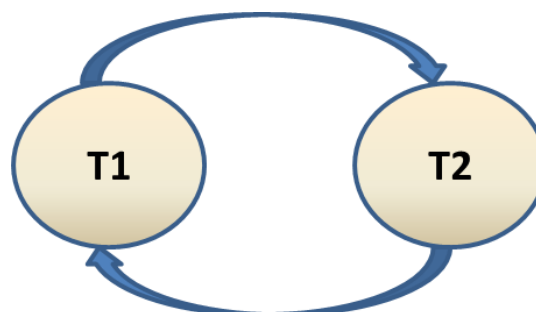


Рис. 6. Косвенная рекурсия

Пример 8. Вычисление чисел Фибоначчи (рис. 7).

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$

Function Fib(n)

If n <= 1

Fib = 1

Else

Fib(n) = Fib(n - 1) + Fib(n - 2)

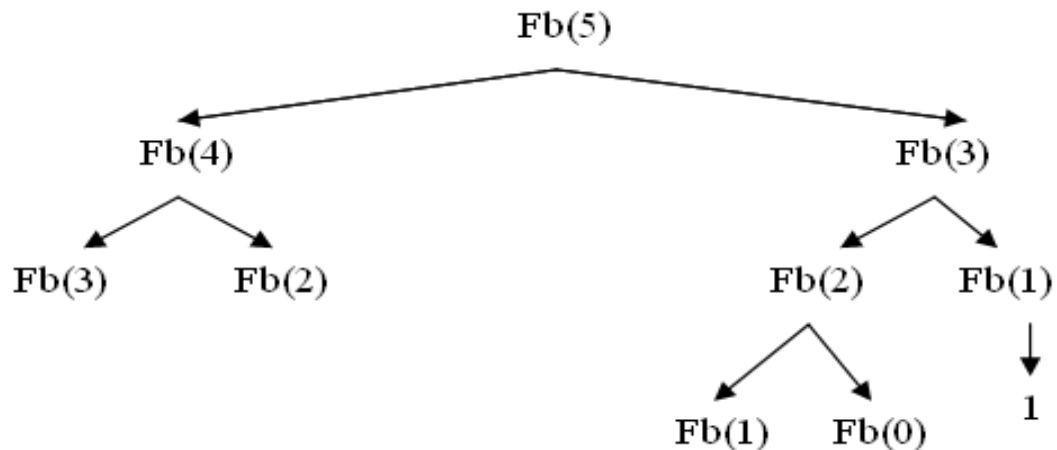


Рис. 7. Вычисление чисел Фибоначчи

Анализ времени выполнения программы

Сколько раз выполняется одно из условий остановки $n \leq 1$

G(0) = 1

G(1) = 1

G(n) = G(n - 1) + G(n - 2) для n > 1

Сколько раз алгоритм достигает рекурсивного шага

H(0) = 0

H(1) = 0

H(n) = 1 + H(n - 1) + H(n - 2) для n > 1.

Тогда **H(n) = Fib(n) - 1** и **G(n) = Fib(n)**

Окончательно: T(n) = H(n) + G(n) = 2Fib(n) - 1.

Общее решение уравнения Фибоначчи имеет вид:

$$F(n) = c_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n$$

Начальными условиями являются значения $F(0) = 0$ и $F(1) = 1$. В соответствии с этим получаем систему уравнений

$$\begin{cases} c_1 + c_2 = 0, \\ \frac{\sqrt{5}}{2} (c_1 - c_2) = 1. \end{cases}$$

Решая эту систему уравнений, находим:

$$c_1 = -c_2 = \frac{1}{\sqrt{5}}$$

$$F(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

Опасности рекурсии:

1. Бесконечная рекурсия.

2. Потери памяти.

Существует несколько способов уменьшения этих накладных расходов:

- Не следует использовать большого количества ненужных переменных.

Даже если подпрограмма их не использует, но память под эти переменные будет отводиться.

- Можно уменьшить использование стека за счет применения глобальных переменных.
- При использовании статических переменных системе не нужно отводить память под новые копии переменных при каждом вызове подпрограммы.

3. Необоснованное применение рекурсии.

Приведенные выше функции факториала, наибольшего общего делителя, чисел Фибоначчи не обязательно должны быть рекурсивными.

Контрольные вопросы

1. С какой целью применяется система сравнительных оценок алгоритмов?
2. Приведите общее определение трудоемкости алгоритма решения задачи с учетом его комплексного анализа.
3. Приведите примеры количественно-зависимых по трудоемкости алгоритмов.
4. Приведите примеры параметрически-зависимых по трудоемкости алгоритмов.
5. Приведите примеры количественно-параметрических по трудоемкости алгоритмов.
6. В чем заключается цель асимптотического анализа функций трудоемкости алгоритмов?
7. Приведите примеры θ -оценок роста функций трудоемкости.
8. Приведите примеры O -оценок роста функций трудоемкости.
9. Приведите примеры Ω -оценок роста функций трудоемкости.

10. Напишите формулу определяющую трудоемкость последовательной конструкции «ветвление» и «цикл».
11. Напишите формулу определяющую трудоемкость конструкции «цикл» со вложенным циклом.
12. Напишите формулу определяющую трудоемкость конструкции «цикл» с m вложенными циклами.
13. В чем заключаются основные проблемы при переходе от операционных к временным оценкам трудоемкости алгоритмов
14. В чем состоит отличие подходов при переходе к временным оценкам?
15. Проведите анализ трудоемкости алгоритма сортировки вставками.
16. Дайте определение теоретического предела трудоемкости алгоритмов.
17. Можно ли оценить теоретический предел трудоемкости алгоритмов нахождения простых чисел?
18. Дайте определение рекуррентных соотношений, приведите примеры.
19. Приведите примеры оценки трудоемкости рекуррентных алгоритмов с использованием основной теоремы.

4. ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ

4.1. МНОЖЕСТВА

Множество — это одно из фундаментальных понятий как в математике, так и в работе с программными средствами. Множества, которые в ходе выполнения алгоритмов могут расширяться, уменьшаться или подвергаться другим изменениям, называются **динамическими**.

В алгоритмах обработки множеств, требуется выполнять операции:

- **Вставить/удалить элементы** в множество,
- **Проверить принадлежность множеству** (*поиск*).

Динамическое множество, поддерживающее перечисленные операции, называется **словарем**.

Более сложные операции:

- **Вставка/извлечение минимального/максимального элемента** в неубывающих очередях с приоритетами.

Оптимальный способ реализации динамического множества зависит от того, какие операции должны им поддерживаться.

В объектно-ориентированных языках программирования объекты могут быть созданы:

- во время компиляции — **статические объекты**,
- во время выполнения программы, путем вызова функций из стандартной библиотеки — **динамические объекты**.

Основная разница этих методов – в их эффективности и гибкости.

Использование статических объектов более эффективно, но менее гибко – выделение памяти происходит до выполнения программы, но при этом необходимо заранее указать тип и размер создаваемого объекта (**искусство программирования**). Задачи, в которых нужно хранить и обрабатывать неизвестное заранее число элементов, требуют динамического выделения памяти.

Отличия между статическим и динамическим выделением памяти:

- статические объекты обозначаются именованными переменными, действия над объектами производятся напрямую, с использованием их имен;
- динамические объекты не имеют собственных имен, действия над ними производятся косвенно, с помощью указателей;
- выделение и освобождение памяти под статические объекты производится компилятором автоматически, под динамические объекты – программистом.

В реализациях динамического множества каждый его элемент представляется некоторым объектом.

Указатель – это переменная, **хранящая адрес** другой переменной или объекта. Если имеется указатель на объект, то можно проверять и изменять значения его полей.

Одно из полей объекта – ключевое (*key field*). Если все ключи различны, то динамическое множество определено набором ключевых значений.

- Объекты могут содержать сопутствующие данные (*satellite data*), которые находятся в других его полях, но не используются.
- Объект может содержать поля, доступные для манипуляции во время выполнения операций над множеством (*иногда в этих полях хранятся данные или указатели на другие объекты множества*).

Ключи могут являться членами полностью упорядоченного множества:

- множества действительных чисел
- множества всех слов, которые могут быть расположены в алфавитном порядке.

ОПЕРАЦИИ ДЛЯ ДИНАМИЧЕСКИХ МНОЖЕСТВ

- **Запросы** (*accessor*) – возвращают информацию о множестве,
- **Модифицирующие операции** (*mutator*) – изменяющие множество.

Список типичных операций:

- ***Search(S, k)*** –запрос (возвращает указатель на элемент множества S с ключом k).
- ***Insert(S, x)*** – модифицирующая операция
- ***Delete(S, x)*** – модифицирующая операция
- ***Minimum(S), Maximum(S)*** – запрос (возвращает указатель на элемент множества S с наименьшим (наибольшим) ключом).
- ***Successor(S, x), Predecessor(S, x)*** –запросы (возвращают указатели на элементы множества S , которые являются соответственно «следующим» и «предшествующим» элементом для элемента x).

Время, необходимое для выполнения операций с множеством измеряется в единицах, связанных с размером множества. Например, для двоичного поиска это время $O(\log_2 n)$.

4.2. СТЕКИ И ОЧЕРЕДИ

Первым из стека (*stack*) удаляется элемент, который был помещен туда последним. В стеке реализуется стратегия "последним вошел – первым вышел" (*last in, first out – LIFO*). Стеки применяются, например, для отслеживания точек возврата из подпрограмм. Языки программирования высокого уровня также используют стек вызовов для передачи параметров при вызове процедур.

Псевдокод операций:

1. Проверка стэка на «пустоту» – **Stack_Empty(S)**;
2. Затолкнуть (добавить) элемент в стэк – **Push(S, x)**;
3. Вытолкнуть (извлечь) элемент из стэка – **Pop(S)**.

STACK_EMPTY(S)

```

1  if  $top[S] = 0$ 
2      then return TRUE
3      else return FALSE

```

PUSH(S, x)

```

1   $top[S] \leftarrow top[S] + 1$ 
2   $S[top[S]] \leftarrow x$ 

```

POP(S)

```

1  if STACK_EMPTY( $S$ )
2      then error "underflow"
3      else  $top[S] \leftarrow top[S] - 1$ 
4          return  $S[top[S] + 1]$ 

```

Каждая операция выполняется в течение времени $O(1)$.

Пример использования стэка – алгоритм обращения порядка элементов массива:

1. все элементы последовательно проталкиваются в стек;
2. все элементы выталкиваются из стека в обратном порядке и записываются обратно в массив.

В очереди (*queue*) удаляется элемент, который был помещен туда первым. В очереди реализуется стратегия "первым вошел первым вышел" (*first in, first out* — *FIFO*). Очередь имеет «голову» (*head*) и «хвост» (*tail*). Когда элемент добавляется в очередь, он занимает место в ее «хвосте». Из очереди всегда выводится элемент, который находится в ее «голове».

Псевдокод операций:

1. Добавить элемент в очередь – **Enqueue(Q, x)**;
2. Извлечь элемент из очереди – **Dequeue(Q, x)**.

```

ENQUEUE( $Q, x$ )
1   $Q[tail[Q]] \leftarrow x$ 
2  if  $tail[Q] = length[Q]$ 
3      then  $tail[Q] \leftarrow 1$ 
4      else  $tail[Q] \leftarrow tail[Q] + 1$ 

DEQUEUE( $Q$ )
1   $x \leftarrow Q[head[Q]]$ 
2  if  $head[Q] = length[Q]$ 
3      then  $head[Q] \leftarrow 1$ 
4      else  $head[Q] \leftarrow head[Q] + 1$ 
5  return  $x$ 

```

4.3. СПИСКИ

Простые списки

- Если в программе необходим список постоянного размера, его можно создать, используя массив.
- В этом случае можно при необходимости опрашивать его элементы в цикле For.
- Многие программы используют списки, которые растут или уменьшаются со временем.
- Можно создать массив, соответствующий максимально возможному размеру списка, но такое решение не всегда будет оптимальным.

Неупорядоченные списки

Порядок расположения элементов не важен.

Поддерживаются следующие операции:

- добавление элемента к списку;
- удаление элемента из списка;
- определение наличия элемента в списке;
- операции для всех элементов списка (*например, вывод списка на дисплей, принтер или в файл*).

СВЯЗАННЫЕ СПИСКИ

Связный список (*linked list*) – базовая динамическая структура данных, состоящая из **узлов**, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.

Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связанного списка может не совпадать с порядком

расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями. Связные списки обеспечивают простое и гибкое представление динамических множеств и поддерживают все операции.

Линейные связные списки

1. Односвязный список (Однонаправленный связный список).

В односвязном списке можно передвигаться только в одну сторону – от начала в конец. Узнать адрес предыдущего элемента, опираясь на содержимое текущего элемента невозможно (рис.3).



Рис.3. Односвязный список

2. Двусвязный список (Двунаправленный связный список)

В этом списке можно передвигаться в обоих направлениях, так как каждый узел хранит адреса как следующего, так и предыдущего узла.. Это свойство делает его более гибкой структурой, чем односвязный список (рис.4).

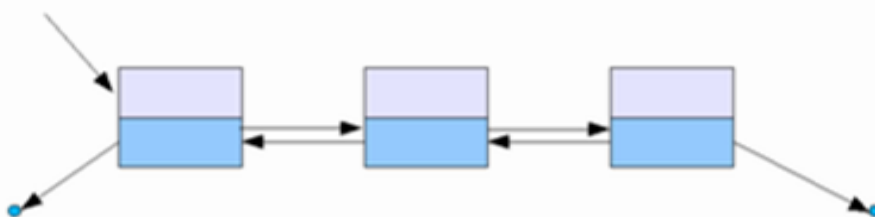


Рис.4. Двусвязный список

3. **XOR-связный список** – структура, похожая на обычный двусвязный список, однако в каждом элементе хранится только один адрес – результат выполнения операции XOR над адресами предыдущего и следующего элементов списка.

$$A[n]=f(A[n+1]\oplus A[n-1])$$

Чтобы перемещаться по списку, необходимо выполнить операцию

$$A[n]=f(A[n-1]\oplus A[n-2])$$

В сравнении с обычным двусвязным списком, XOR список расходует в два раза меньше памяти для хранения связей между элементами. Используется довольно редко, так как существуют хорошие альтернативы, например, развёрнутый связный список.

Кольцевой связный список

Может быть односвязным или двусвязным. Ключевая особенность этих списков в том, что его последний элемент содержит указатель на первый, а первый (в случае двусвязного списка) – на последний.



Рис. 5. Кольцевой связный список

Развёрнутый связный список

Каждый физический элемент содержит несколько логических (*обычно в виде массива, что позволяет ускорить доступ к отдельным элементам*).

- Позволяет значительно уменьшить расход памяти и увеличить производительность по сравнению с обычным списком.
- Особенно большая экономия памяти достигается при малом (*оптимальном*) размере логических элементов и большом их количестве.

Например, располагая 10 тысяч 4-байтных целых чисел в виде *односвязного списка* с 4-байтной адресацией, получим расход памяти

$$10\,000 * 4 \text{ (числа)} + 4 * 10\,000 \text{ (адресация)} = 80\,000 \text{ байт.}$$

Объединяя числа в 100 массивов по 100 элементов, получим *развёрнутый связный список*, в котором расход памяти на адреса упадёт до 400 байт, и суммарный расход памяти составит 40400 байт.

Преимущество состоит в том, что в развёрнутый список легко добавлять новые элементы – нет необходимости переписывать весь массив. Однако, при *слишком большом размере блока* список начинает страдать от тех же проблем, что и обыкновенный массив:

- ✓ долгая вставка элементов в начало или середину, долгое удаление элементов оттуда же, и т.п.,

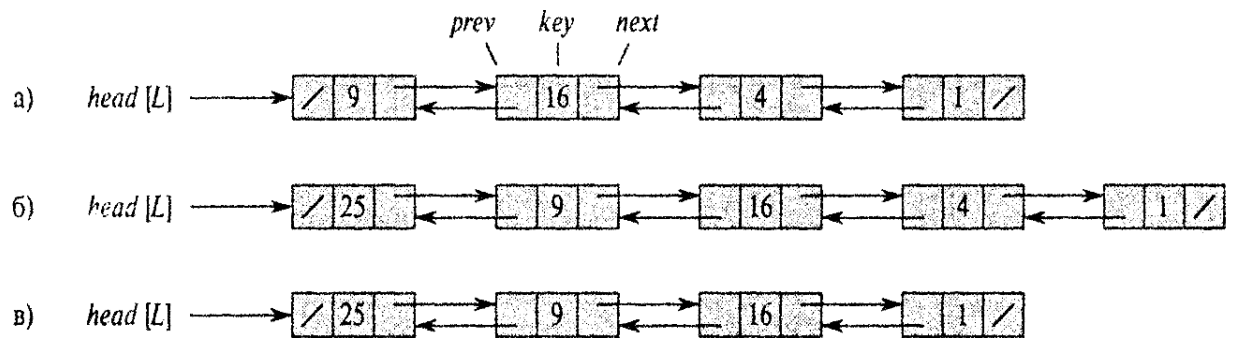
при слишком маленьком

- ✓ увеличивается расход памяти.

ОПЕРАЦИИ С ДВУСВЯЗНЫМ СПИСКОМ

Элемент (*узел*) списка это объект с полями:

- **key**(ключ) – хранимые данные;
- **next**(следующий) – указатель;
- **prev**(предыдущий) – указатель.



Если **prev[x] = NULL**, то узел **x** является первым в списке.

Если **next[x] = NULL**, то узел **x** является последним в списке.

Указатель **head[L]** указывает на первый элемент списка.

Если **head[L] = NULL**, то список пуст.

- 1) Процедура **List_Search(L, k)** позволяет найти в списке **L** первый элемент с ключом **k** путем линейного поиска, и возвращает указатель на найденный элемент. Если элемент с ключом **k** в списке отсутствует, возвращается значение **NULL**. Асимптотическая трудоемкость $\Theta(n)$.

Псевдокод процедуры:

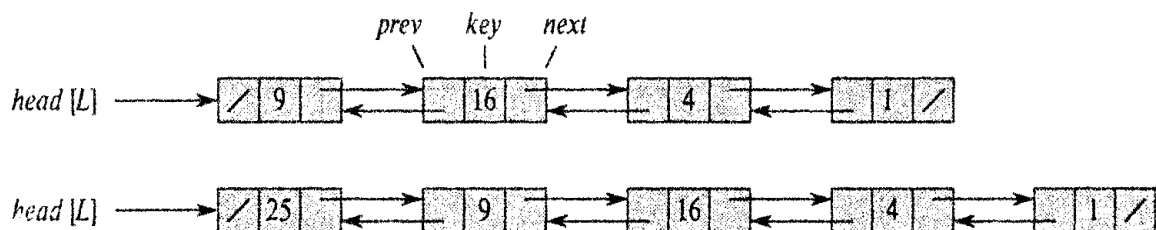
LIST_SEARCH(L, k)

```

1   $x \leftarrow head[L]$ 
2  while  $x \neq NIL$  и  $key[x] \neq k$ 
3      do  $x \leftarrow next[x]$ 
4  return  $x$ 

```

- 2) Процедура **List_Insert(L, x)** добавляет элемент **x** в начало списка. Асимптотическая трудоемкость $O(1)$.



Псевдокод процедуры:

LIST_INSERT(L, x)

```

1   $next[x] \leftarrow head[L]$ 
2  if  $head[L] \neq NIL$ 
3      then  $prev[head[L]] \leftarrow x$ 
4   $head[L] \leftarrow x$ 
5   $prev[x] \leftarrow NIL$ 

```

- 3) Процедура **List_Delete(L, x)** удаляет элемент x из связанного списка L . В процедуру необходимо передать указатель на элемент x , после чего он удаляет x из списка путем перенаправления указателей. Асимптотическая трудоемкость удаления элемент с заданным ключом $\Theta(n)$. Чтобы удалить элемент с заданным ключом, необходимо сначала вызвать процедуру **List_Search(L, k)** для получения указателя на элемент.

Псевдокод процедуры:

```

LIST_DELETE( $L, x$ )
1  if  $prev[x] \neq \text{NIL}$ 
2      then  $next[prev[x]] \leftarrow next[x]$ 
3      else  $head[L] \leftarrow next[x]$ 
4  if  $next[x] \neq \text{NIL}$ 
5      then  $prev[next[x]] \leftarrow prev[x]$ 

```

4.4. ДЕРЕВЬЯ

Дерево – структура данных, представляющая собой связный, ациклический граф.

Связный граф – граф между любой парой вершин которого существует по крайней мере один путь.

Ацикличность означает, что между любой парой вершин в дереве существует только один путь.

Ориентированное (направленное) дерево – ациклический оргграф, в котором только одна вершина имеет нулевую степень захода, а все остальные вершины имеют степень захода 1.

Корень дерева — вершина с нулевой степенью захода.

Концевая вершина (лист) – вершина с нулевой степенью исхода.

Дерево – конечное множество T со свойствами:

- существует *единственный корень* дерева T
- остальные узлы распределены среди T_i непересекающихся множеств и каждое из множеств является деревом.

$$T = \bigcup_{i=1}^m T_i, \quad \bigcap_{i=1}^m T_i = \emptyset.$$

Деревья T_i называются *поддеревьями*.

Основные свойства структуры данных ДЕРЕВО

- Дерево не имеет кратных рёбер и петель.
- Любое дерево с n вершинами содержит $n - 1$ ребро.

- Граф является деревом тогда и только тогда, когда любые две различные его вершины можно соединить единственным элементарным путём.
- Любое дерево однозначно определяется расстояниями (*длиной наименьшей цепи*) между его концевыми вершинами.
- Для любых трех вершин дерева, пути между парами этих вершин имеют ровно одну общую вершину.

Неориентированное дерево – степени вершин не превосходят 3.

Ориентированное дерево – исходящие степени вершин не превосходят 2.

Бинарное дерево – абстрактная структура данных, на которой основаны:

- бинарное дерево поиска
- бинарная куча
- красно-чёрное дерево
- АВЛ-дерево, фибоначчиева куча и др.

N-арные деревья — деревья с произвольным количеством дочерних элементов. В неориентированном – степени вершин не превосходят $N+1$. В ориентированном исходящие степени вершин не превосходят N . Каждый узел дерева представляет собой отдельный объект. Как и в связных списках, предполагается, что каждый узел содержит ключевое поле **key**. Остальные поля – это указатели на другие узлы, и их вид зависит от типа дерева.

4.5. КУЧА

Куча – структура данных (*типа дерева*), которая удовлетворяет основному свойству кучи (рис.9):

если B является узлом-потомком узла A , то $\text{key}(A) \geq \text{key}(B)$.

Элемент с наибольшим ключом всегда является корневым узлом кучи.

Бинарная куча, пирамида, или сортирующее дерево – такое двоичное дерево, для которого выполнены три условия:

1. *Значение в любой вершине не меньше, чем значения её потомков.*
2. *Каждый лист имеет глубину (расстояние до корня) либо h либо $h-1$. Все слои, кроме, быть может, последнего, заполнены полностью.*
3. *Последний слой заполняется слева направо.*

Над кучами обычно проводятся следующие операции:

- **найти** максимум или найти минимум
- **удалить** максимум или удалить минимум
- **увеличить** ключ или уменьшить ключ
- **слияние**: соединение двух куч с целью создания новой кучи, содержащей все элементы обеих исходных
- **добавить**: добавление нового ключа в кучу

Двоичная куча позволяет за время $O(\log_2 n)$ добавлять, изменять, извлекать элемент с максимальным приоритетом.

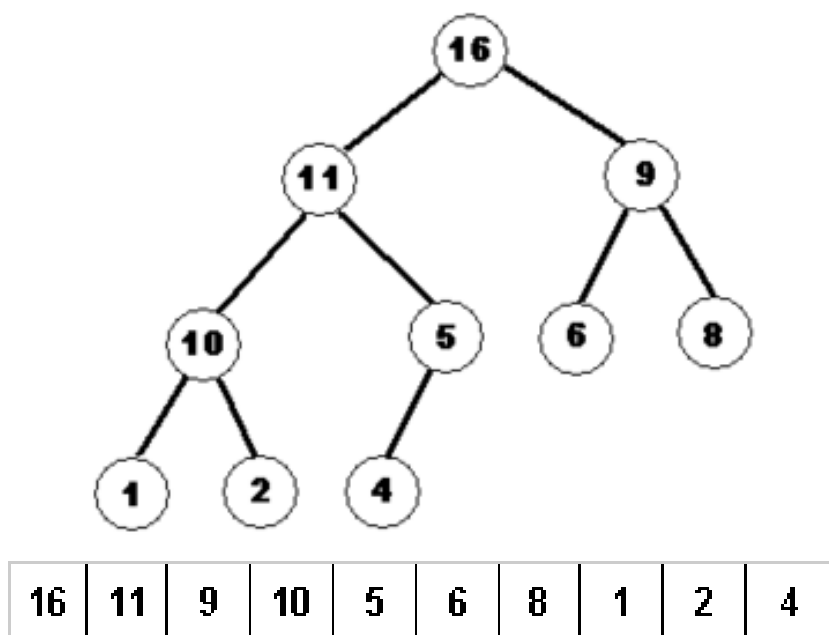


Рис.9. Куча

Сравнение теоретических оценок сложности вычислений

Операция	Двоичная	Биномиальная	Фибоначчиева	Спаренная	Бродал
Найти минимум	$\theta(1)$	$\theta(\log n)$ or $\theta(1)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
Удалить минимум	$\theta(\log n)$	$\theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$
Добавить	$\theta(\log n)$	$O(\log n)$	$\theta(1)$	$O(1)^*$	$\theta(1)$
Уменьшить ключ	$\theta(\log n)$	$\theta(\log n)$	$\theta(1)^*$	$O(\log n)^*$	$\theta(1)$
Слияние	$\theta(n)$	$O(\log n)^*$	$\theta(1)$	$O(1)^*$	$\theta(1)$

Таб.1. Сравнение теоретических оценок сложности вычислений

Контрольные вопросы

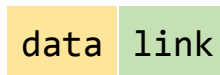
1. Поясните в чем заключается отличие динамического множества от статического
2. Какие стандартные операции могут совершаться с динамическим множеством?
3. Приведите примеры использования стека.
4. Опишите и поясните основные операции над стеком.
5. Приведите примеры использования очереди.
6. Опишите и поясните основные операции над очередью.
7. Дайте развернутую классификацию списков.
8. Поясните принципы организации связанных списков и приведите их классификацию.
9. Опишите организацию операции поиска элемента в связном списке.
10. Опишите организацию операции вставки элемента в связном списке.
11. Опишите организацию операции удаления элемента в связном списке.
12. Опишите основные свойства структуры данных дерево.
13. Опишите основные свойства структуры данных куча.

Лабораторная работа №1

Цель работы: Изучение принципов организации и работы с абстрактной структурой данных **ОДНОСВЯЗНЫЙ ЛИНЕЙНЫЙ СПИСОК**.

Односвязный линейный список — структура данных, где каждый элемент содержит указатель **только на следующий** элемент.

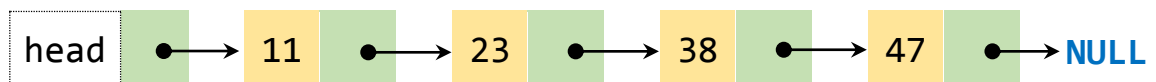
Общий вид элемента (узла) списка:



data — информационная часть (**public**)

link — указатель (**private**)

Общий вид односвязного линейного списка:



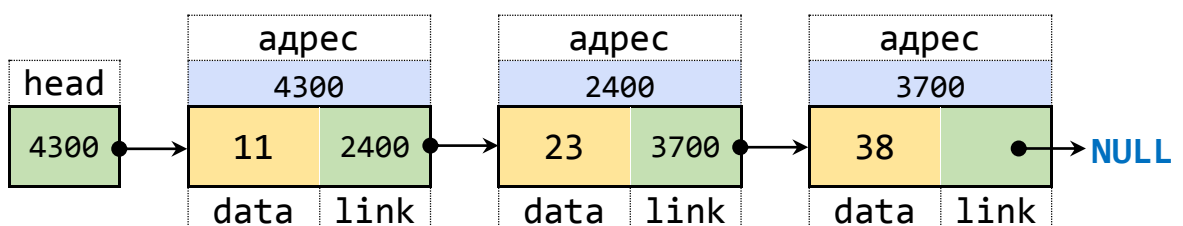
Каждый элемент списка может быть описан как **класс (class)** или **структура (struct)**. Информационная часть может меняться в зависимости от того, какие данные обрабатываются. Однако, присутствие указателя на следующий элемент обязательно.

В простейшем случае, элемент списка можно представить в виде структуры:

```
struct nodeType
{
    int data;
    nodeType *link;
};

nodeType *head, *current;
```

Рассмотрим следующий список:



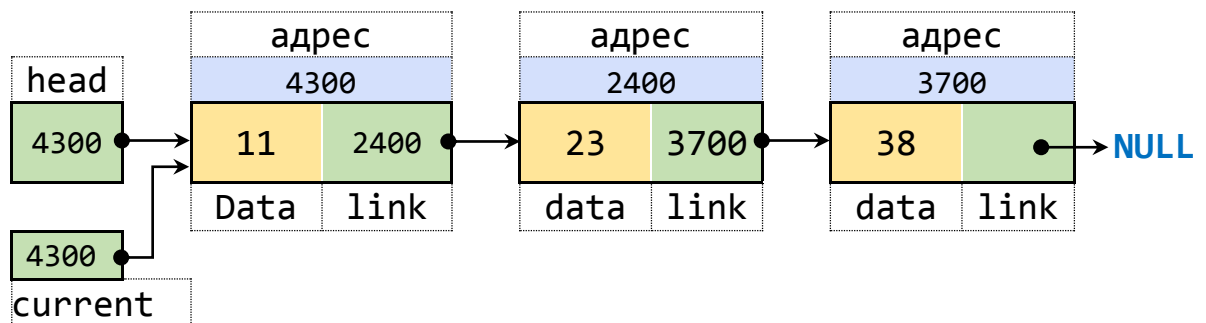
Тогда имеет место таблица:

Переменная	Значение
head	4300
head->data	11
head->link	2400
head->link->data	23

Пусть указатель **current** будет копией указателя **head**:

current = head;

В памяти компьютера это будет выглядеть следующим образом:



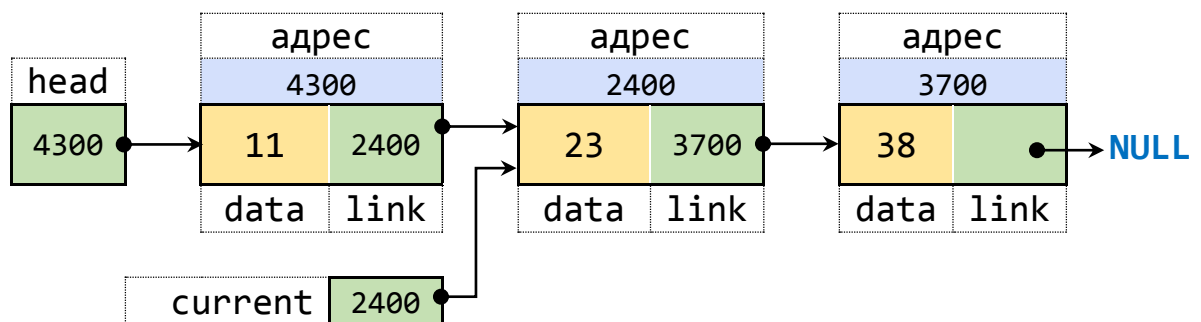
Для указателя **current** имеет место та же самая таблица:

Переменная	Значение
current	4300
current->data	11
current->link	2400
current->link->data	23

Если теперь

current = current->link;

тов памяти компьютера произойдёт следующее:



В этом случае таблица для указателя **current** будет выглядеть так:

Переменная	Значение
current	2400
current->data	23
current->link	3700
current->link->data	38

При этом указатель **head** остаётся без изменения и «указывает», как и прежде, на первый элемент списка. Доступ к любому элементу списка можно получить с помощью обоих указателей, — **head** или **current**, но перемещаться по списку можно лишь **в одном направлении** — от текущего элемента к следующему слева направо:

Переменная	Значение
head->link->link	3700
head->link->link->data	38
head->link->link->link	NULL
current->link->link	NULL
current->link->data	38
current->data	23
current->link	3700

Имея в своём распоряжении указатель **current**, можно легко «пройти» по списку от начала до конца. Например, вывести информационную часть элементов списка в консоль.

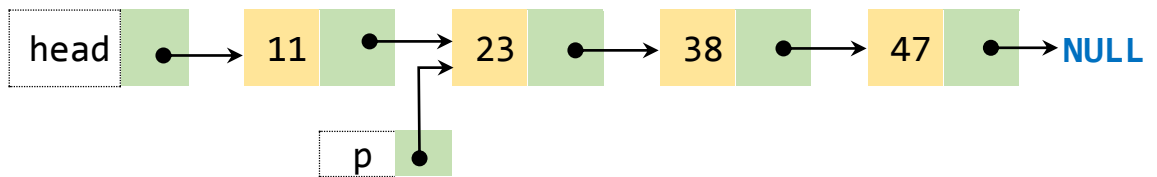
```
current = head;
while(current != NULL)
{
    cout << current->data << " ";
    current = current->link;
}
```

Самые естественные операции со списком это

1. создание списка;
2. добавление элементов;
3. удаление элементов;
4. поиск и извлечение элементов.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ОДНОСВЯЗНЫЙ СПИСОК.

Пусть список имеет следующий вид:



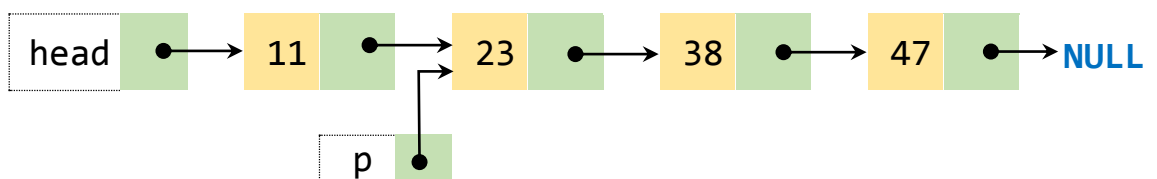
Нужно добавить в список элемент сразу после элемента, на который указывает **p**. Эта задача решается следующим образом:

1. `newNode = new nodeType;`
//создание нового эл-та списка
2. `newNode->data = 50;`
//запись информационной части
3. `newNode->link = p->link;`
//инициализация указателя нового элемента
4. `p->link = newNode;`
//перенаправление указателя p

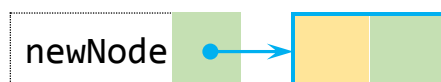
Работа с указателями должна осуществляться именно в такой последовательности — существующий указатель **p** сначала используется, а только потом перенаправляется.

Графическая иллюстрация:

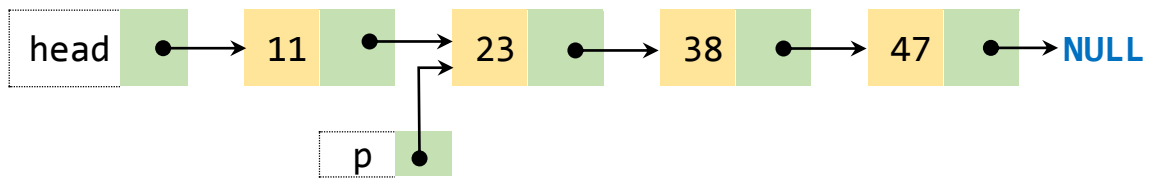
1. `newNode = new nodeType;`



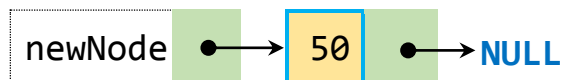
//создание нового эл-та списка



```
2. newNode->data = 50;
   newNode->link = NULL;
```

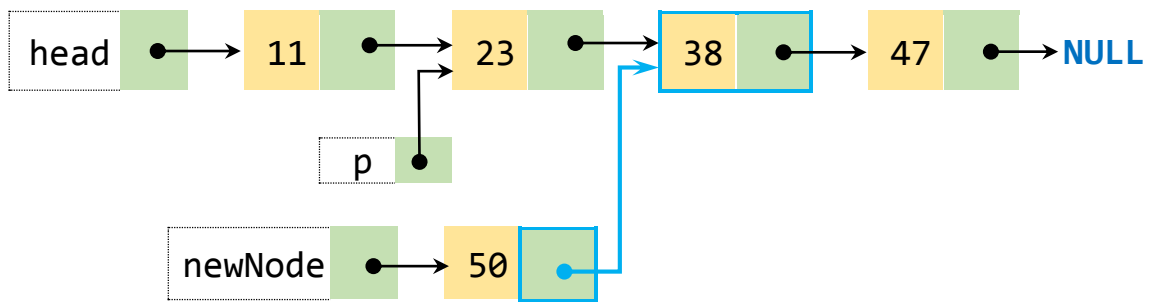


//запись информационной части указателя



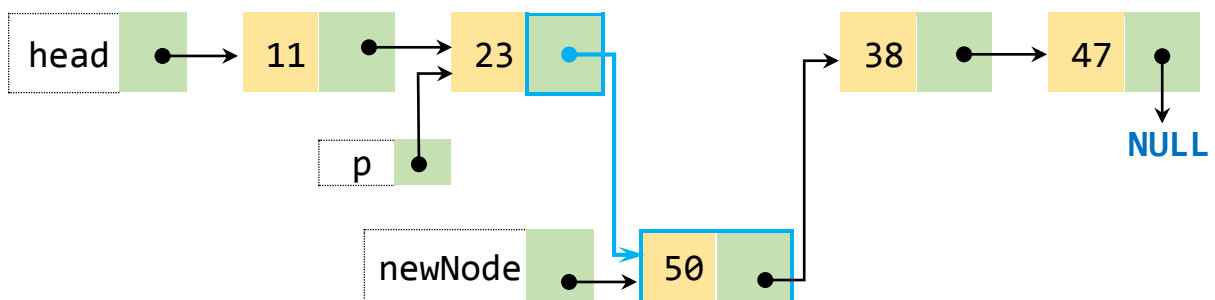
```
3. newNode->link = p->link;
```

//инициализация указателя нового элемента



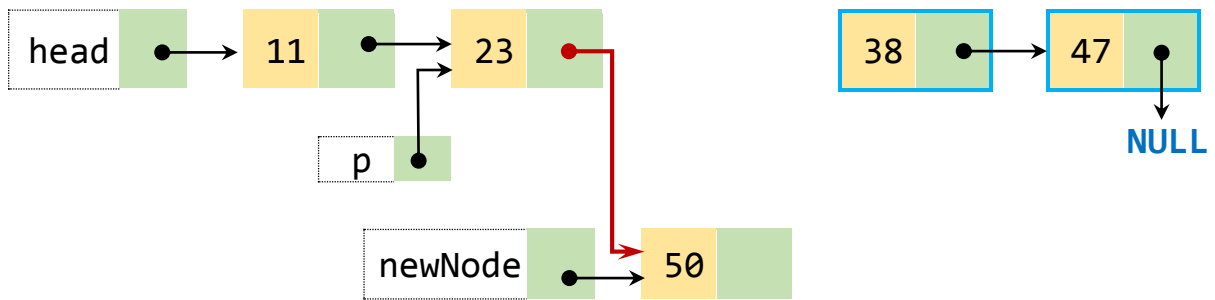
```
4. p->link = newNode;
```

//перенаправление указателя p

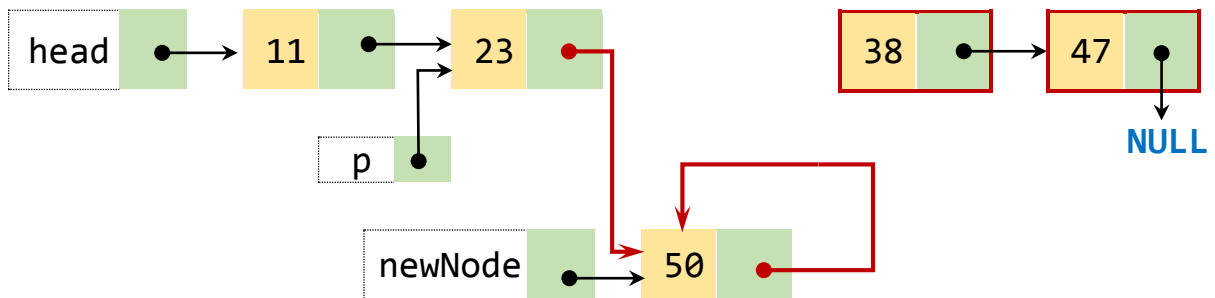


Внимание! Последовательность перенаправления указателей существенно важна! Если порядок перенаправления указателей изменить, произойдёт потеря данных! А именно:

3. `p->link = newNode;`



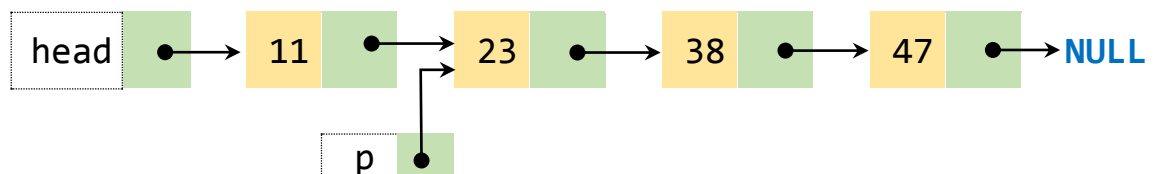
4. `newNode->link = p->link;`



В этом случае элементы списка со значениями 38 и 47 удалены из списка и потеряны.

УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ОДНОСВЯЗНОГО СПИСКА.

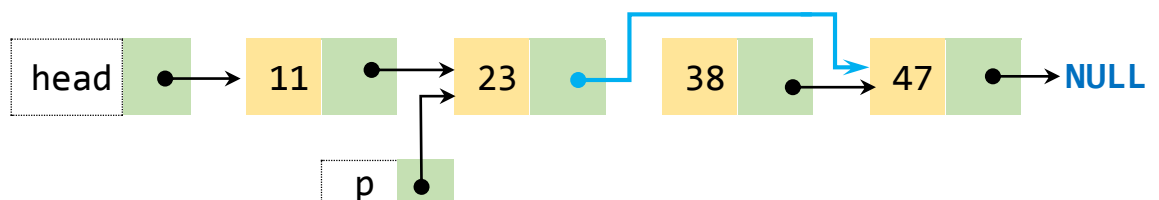
Пусть список имеет такой же вид, как в предыдущем примере:



Нужно удалить из списка элемент, следующий сразу же после того элемента, на который указывает **p**. Удалить элемент из списка можно следующим образом:

`p->link = p->link->link;`

Выглядит это так:



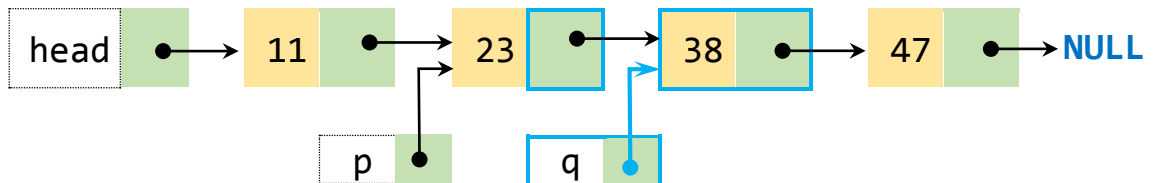
Однако, удалённый из списка элемент продолжает занимать выделенную ему память и она недоступна для использования. Чтобы освободить память нужно иметь указатель на этот элемент. Поэтому полное решение задачи будет таким:

```
1. nodeType *q;  
2. q = p->link;  
3. p->link = q->link;  
4. delete q;
```

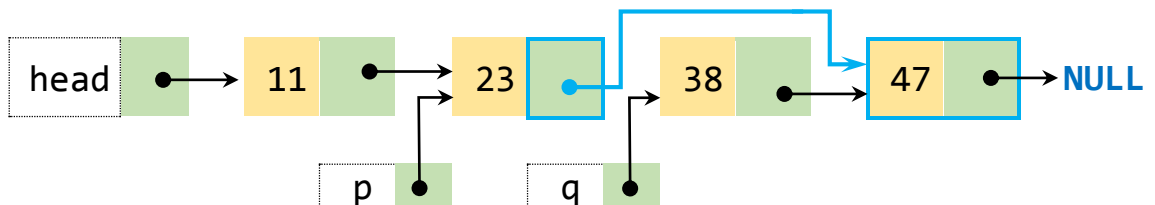
В этом случае элемент удаляется из списка и освобождается память.

Графическая иллюстрация

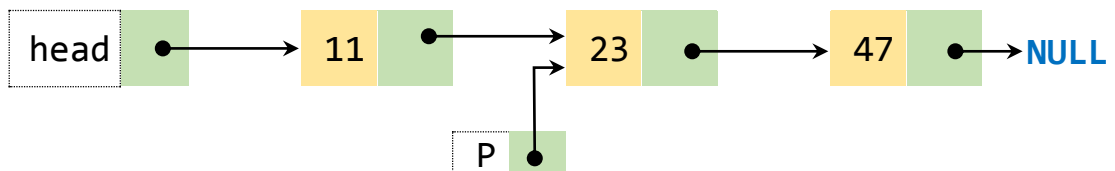
```
1. nodeType *q;  
2. q = p->link;  
   //дополнительный указатель q для освобождения  
   памяти
```



```
3. p->link = q->link;  
   //перенаправление указателя p – удаление элемента  
   из списка
```



```
4. delete q;  
   //освобождение памяти
```



СОЗДАНИЕ ОДНОСВЯЗНОГО СПИСКА.

До сих пор речь шла о уже существующем, непустом списке. Чтобы создать самый простой список используем три указателя. Один из них будет указывать на первый элемент списка, второй на последний и третий на новый.

```
nodeType *first, *last, *newNode;
```

```
int num;
```

```
//поскольку списка ещё нет ...
```

```
1. first = NULL;
```

```
2. last = NULL;
```

A diagram showing a box labeled 'first' with a green square next to it. An arrow points from the green square to the word 'NULL'.

A diagram showing a box labeled 'last' with a green square next to it. An arrow points from the green square to the word 'NULL'.

```
//ввод числа с консоли и сохранение в переменной num
```

```
3. cin >> num;
```

```
//выделение памяти под новый элемент списка
```

```
4. newNode = new nodeType;
```

```
//инициализация инф. части элемента и установка в  
NULL поля-указателя link элемента
```

```
newNode(т.к. список пока пуст).
```

```
5. newNode->data = num;
```

```
6. newNode->link = NULL;
```

A diagram showing a box labeled 'newNode' with a green square next to it. An arrow points from the green square to a yellow box labeled 'num'. To the right of 'num' is another green square, with an arrow pointing from it to the word 'NULL'.

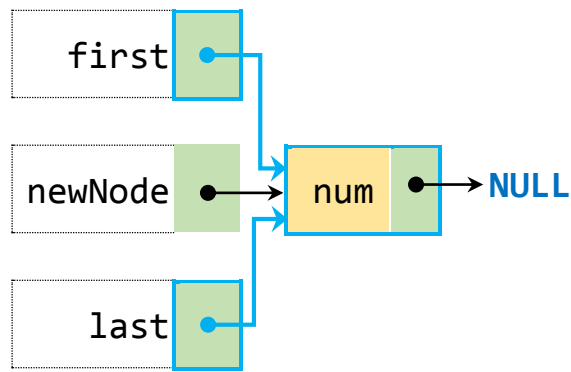
```
7. if (first == NULL) //если в списке нет ни одного  
элемента ...
```

```
{
```

```
8. first = newNode;
```

```
9. last = newNode;
```

```
} //указатели first и last направлены на newNode –  
первый, и пока единственный элемент нового списка
```

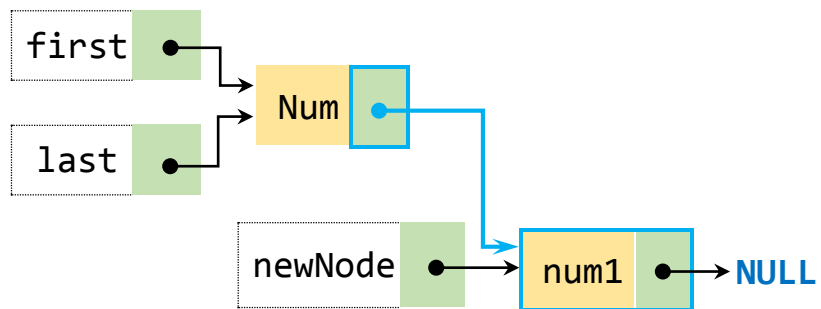


```
10.     else //список не пуст...
```

```
{
```

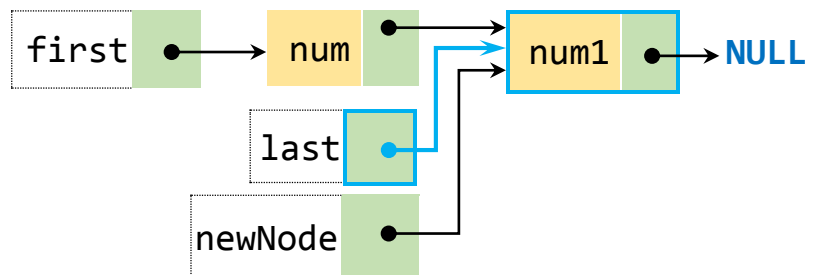
```
11.   last->link = newNode;
```

```
      //новый узел newNode добавляется  
      в конец списка
```



```
12. last = newNode;
```

```
      //указатель last перенаправляется на последний  
      элемент списка – newNode
```



```
}.
```

ОДНОСВЯЗНЫЙ ЛИНЕЙНЫЙ СПИСОК КАК АБСТРАКТНЫЙ ТИП ДАННЫХ (ADT).

Поскольку в форме ОЛН (односвязный линейный список) можно представить различные объекты и типы данных, естественно сделать независимой реализацию методов списка от природы объектов, образующих список. Это достигается с помощью шаблона класса. Для реализации ОЛН потребуется как минимум два шаблона – один для узлов списка, второй для самого списка. Узлы списка можно описывать как структуру(**struct**) или класс (**class**). Например, в простейшем случае

1. Структура

```
template<class Type>
struct Node
{
    Type data;
    Node<Type>* next;
    Node(const Type& d = Type(),
         Node<Type>* p = nullptr):data{d},
        next{p} {}
};
```

2. Класс

```
template<typename T>
class Node
{
public:
    .....
private:
    T data;
    Node<T>* next;
};
```

Поскольку класс определяющий структуру данных содержит члены-указатели, в него **обязательно** должны быть включены:

1. Деструктор, освобождающий память от указателей;
2. Перегруженный оператор присваивания ‘=’;
3. Копирующий конструктор.

Поддерживаемые операции

1. вставить элемент в начало списка;
2. вставить элемент в конец списка;
3. получить указатель на первый элемент списка.
4. получить указатель на последний элемент списка.
5. получить информационную часть последнего элемента списка.
6. получить информационную часть первого элемента списка.
7. получить указатель на следующий, относительно текущего, элемент;
8. найти элемент с указанной информационной частью и вернуть указатель на него.
9. вставить элемент после текущего элемента и вернуть указатель на него;
10. удалить выбранный элемент из списка;
11. очистить список;
12. копировать список;
13. вывести в консоль все элементы списка.

Пример интерфейса класса `LinkedList`

```
template<class T>
class LinkedList
{
public:
    LinkedList(); // Конструктор по умолчанию. Создаёт
                  пустой список.
    ~LinkedList(); // Деструктор
    const LinkedList<T>& operator=
                      (const LinkedList<T>&);
    // Перегрузка оператора присваивания.
    LinkedList(const LinkedList<T>& otherList);
    // Копирующий конструктор
    bool isEmpty() const;
    // Отвечает на вопрос пуст список или нет.
    // Возвращает true, если список не пуст и false в
    // противном случае.
    void print() const;
    // Выводит данные из каждого элемента списка.
    int length() const;
    // Возвращает количество элементов в списке.
    void empty();
    // Удаляет все узлы списка. В результате
```

```

        список пуст: first = NULL, last = NULL,
        count = 0;
T front() const;
// Возвращает информационную часть первого
  элемент списка.
T end() const; // Возвращает информационную часть
  последнего элемента списка.
void insertFirst(const T& newItem);
// Добавляет элемент в начало списка.
void insertLast(const T& newItem);
// Добавляет элемент в конец списка.
Node<T>* findNode(const T& item);
// Ищет в списке элемент item и возвращает
  указатель на этот элемент.
Node<T>* nextNode(Node<T>* curr = nullptr);
// Возвращает указатель на следующий элемент
  списка.
void deleteNode(const T& item);
  // Удаляет элемент item из списка.
private:
void copyList(const LinkedList<T>
  &otherList);
  // Функция, создающая копию списка otherList.
int count; // кол-во элементов списка.
Node<T>* first; // Указатель на первый элемент списка.
Node<T>* last; // Указатель на последний элемент
  списка.
};

```

Контрольные вопросы

1. Что такое абстрактная структура данных?
2. Основные свойства структуры данных односвязный линейный список?
3. Напишите алгоритм основных операций с односвязным линейным списком.
4. Какие способы реализации структуры данных односвязный линейный список вы знаете?

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1

Разработать консольное приложение, которое с помощью абстрактной структуры данных **ОДНОСВЯЗНЫЙ ЛИНЕЙНЫЙ СПИСОК** формирует список из:

- строк определённой тематики (*тематику выбрать самостоятельно*);
- массивов чисел;
- массивов строк;
- структур, содержащих не менее 2-х числовых и 2-х строковых полей.

Приложение должно:

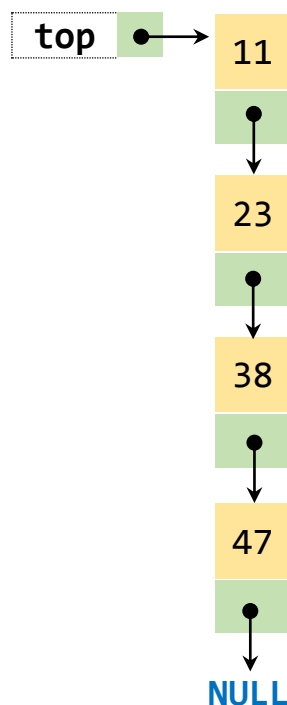
1. делать запрос на ввод данных пользователем при создании списка;
2. выводить по запросу длину (*к-во элементов*) списка;
3. осуществлять поиск данных по введённому значению и сообщать номер найденного элемента или соответствующее сообщение, в случае, если данные не найдены.

Лабораторная работа №2

Цель работы: Изучение принципов организации и работы с абстрактной структурой данных СТЭК в форме односвязного линейного списка.

Стэк — структура данных, в которую можно добавлять элементы и извлекать их, причём элемент, который был добавлен последним, извлекается первым (**LIFO—Last In First Out**, последний пришёл первый ушёл). Указатель **top** всегда указывает на **последний** добавленный элемент.

Графическая иллюстрация:

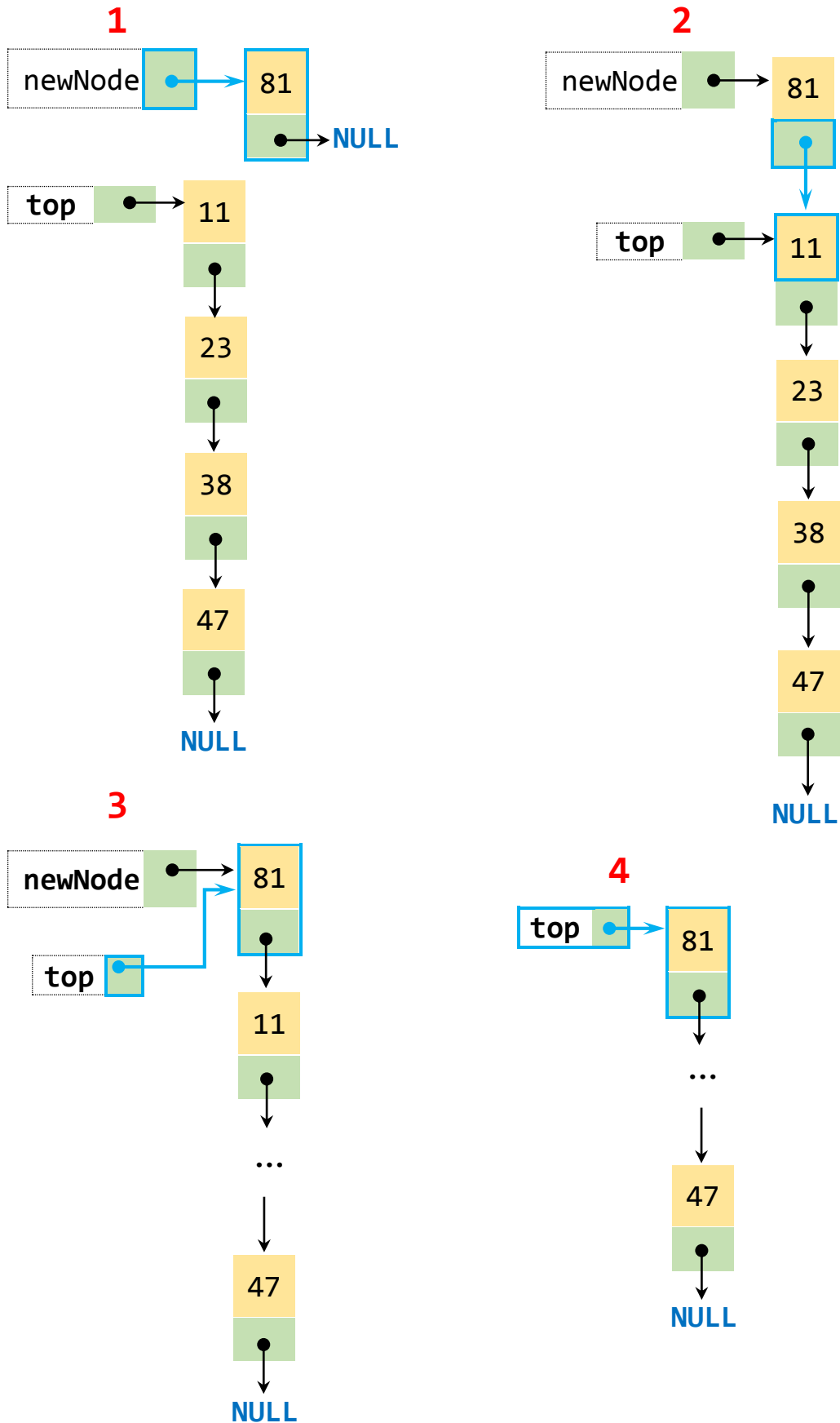


Поддерживаемые операции

1. добавить элемент в стэк;
2. извлечь информационную часть «верхнего» элемента;
3. удалить «верхний» элемент;
4. проверить на пустоту;
5. очистить стэк;
6. копировать стэк.

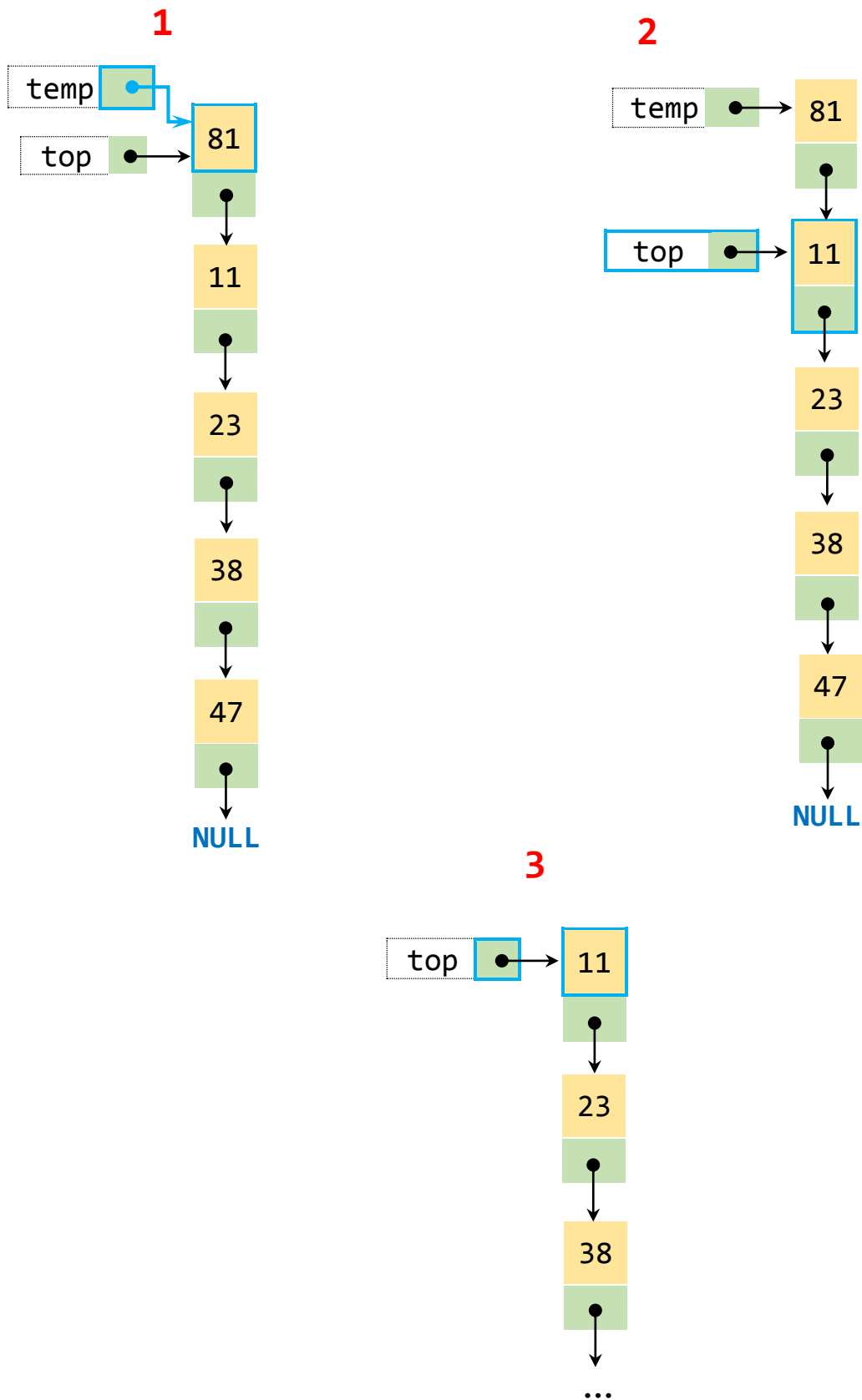
ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В СТЭК (метод `push()`)

Графическая иллюстрация:



УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СТЭКА (метод `pop()`)

Графическая иллюстрация:



Пример интерфейса класса **LinkedStack** и структуры **Node**

```
template<class Type>
struct Node
{
    Type info;
    Node<Type>* link;
};

template<class Type>
class LinkedStack
{
public:
    linkedStack(); // Конструктор по умолчанию
    linkedStack(const linkedStack<Type>
                &otherStack);

    // Копирующий конструктор
    ~linkedStack(); // Деструктор
    const linkedStack<Type>& operator=(const
                                     linkedStack<Type>& otherStack);

    // Перегрузка оператора присваивания
    bool isEmpty() const;
    // Отвечает на вопрос пуст ли стэк?
    void initialize();
    // Опустошает стэк. (top = NULL)
    void push(const Type& newItem);
    // Добавляет newItem в стэк
    Type top() const;
    // Возвращает данные top элемента стэка
    void pop(); // Удаляет top элемент стэка
```

private:

```
Node<Type>* top;// указатель на последний
добавленный элемент стэка
void copy(const linkedStack<Type>
&otherStack) ;

// Функция копирования. Создает копию стэка
otherStack и присваивает её this стэку.
```

Контрольные вопросы

5. Что такое абстрактная структура данных?
6. Основное свойство структуры данных стэк?
7. Напишите алгоритм основных операций со стэком.
8. Какие способы реализации структуры данных стэк вы знаете?
9. Каков будет результат применения операций PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,6), PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), POP(S) к пустому стэку, хранящемуся в массиве S[1..6]? К тому же стэку, уже хранящему 2 элемента?

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1

Разработать консольное приложение, которое с помощью абстрактной структуры данных СТЭК располагает элементы данного массива (*вектора*) в обратном порядке (*инвертирует*). Тип данных элементов массива выбрать произвольно. Приложение должно:

1. делать запрос на ввод размера и элементов создаваемого массива;
2. выводить в консоль исходный массив после его создания;
3. делать запросы на:
 - 3.1.инвертирование исходного массива:
 - ✓ в случае «Да» выводить в консоль инвертированный массив;
 - 3.2.изменение исходного массива:
 - ✓ в случае «Да» создавать новый массив и выводить его в консоль;
 - 3.3.выход из приложения
 - ✓ в случае «Да» выходить из приложения;

Задание 2

Разработать консольное приложение, которое с помощью абстрактной структуры данных СТЭК проверяет соответствие открывающих и

закрывающих HTML-тэгов во фрагменте HTML кода, введённого с клавиатуры. Приложение должно:

1. делать запрос на ввод HTML кода;
2. выводить в консоль:
 - ✓ в случае соответствия – сообщение об этом;
 - ✓ в случае несоответствия – сообщение об этом и тэг(и), для которого нет пары;
3. делать запрос на выход из приложения.

Задание 3

Разработать консольное приложение, которое с помощью абстрактной структуры данных СТЭК вычисляет арифметическое выражение, записанное в постфиксной форме (*postfix notation*). Выражение может содержать 4 основных действия, скобки и числа. Для распознавания чисел их можно вводить сразу после определённого символа, например, #. Приложение должно:

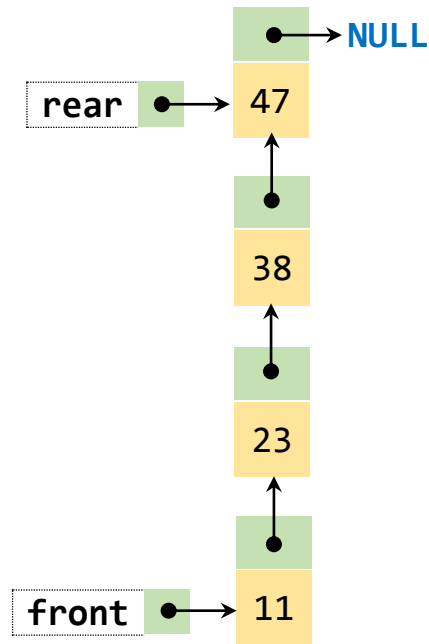
1. делать запрос на ввод выражения в постфиксной форме;
2. выводить в консоль:
 - ✓ введённое выражение и результат вычислений в случае корректного ввода;
 - ✓ сообщение об ошибке и введённое выражение в случае некорректного ввода;
3. делать запрос на выход из приложения.

Лабораторная работа №3

Цель работы: Изучение принципов организации и работы с абстрактной структурой данных ОЧЕРЕДЬ в форме односвязного линейного списка.

Очередь — структура данных, в которую можно добавлять элементы и извлекать их, причём элемент, который был добавлен первым, извлекается также первым (**FIFO–First In First Out**, первый пришёл первый ушёл). Указатель **front** указывает на **первый** добавленный элемент, а **rear** – на **последний**.

Графическая иллюстрация:

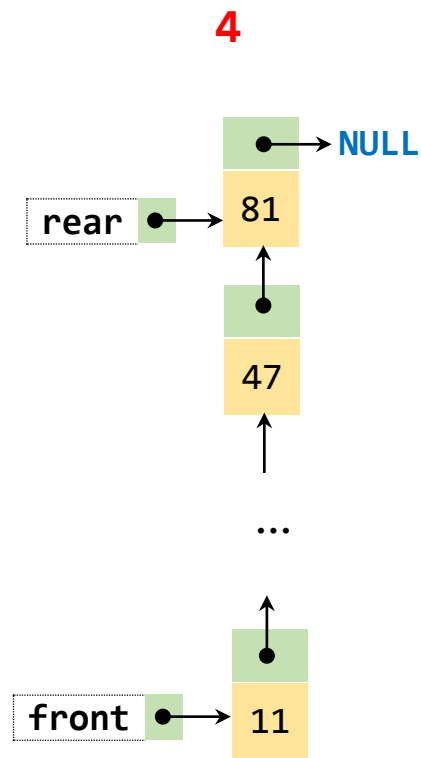
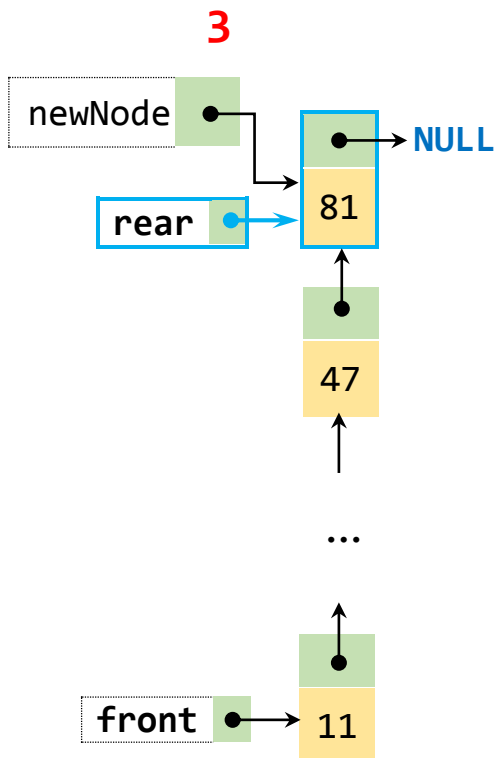
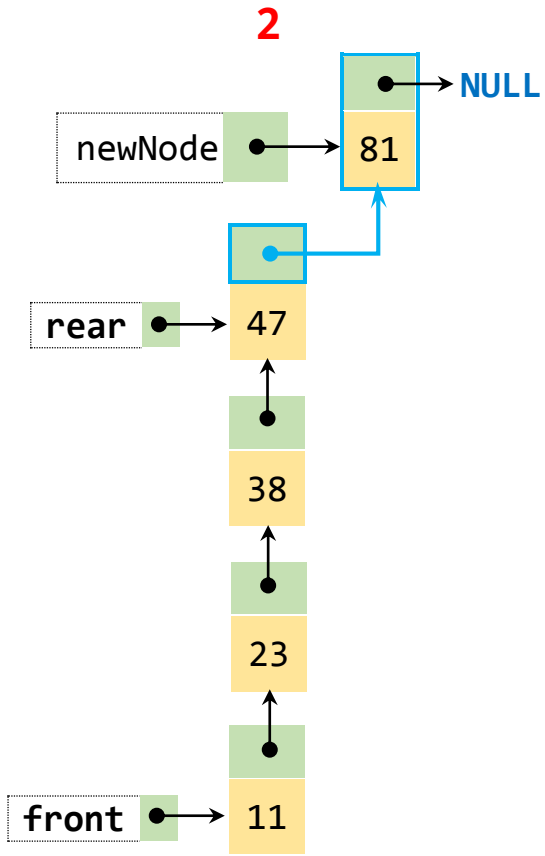
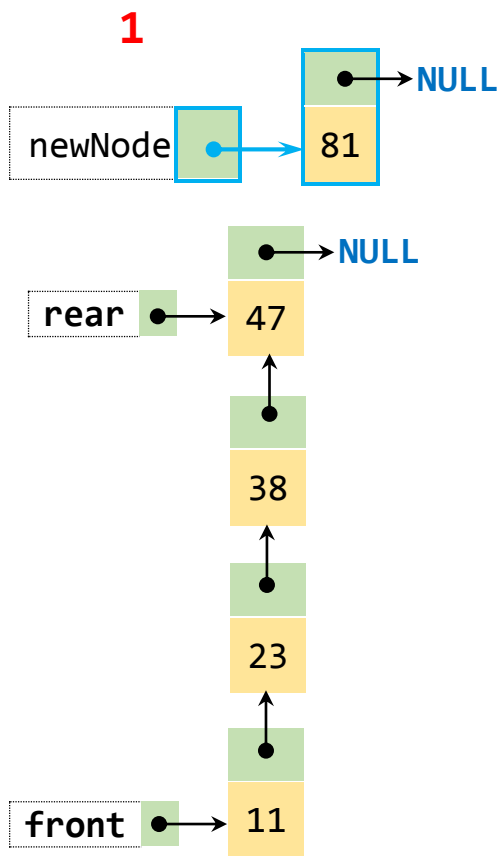


Поддерживаемые операции

1. добавить элемент в очередь;
2. извлечь информационную часть «первого» элемента;
3. удалить «первого» элемент;
4. проверить на пустоту;
5. очистить очередь;
6. копировать очередь.

ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ОЧЕРЕДЬ (метод `enqueue()`)

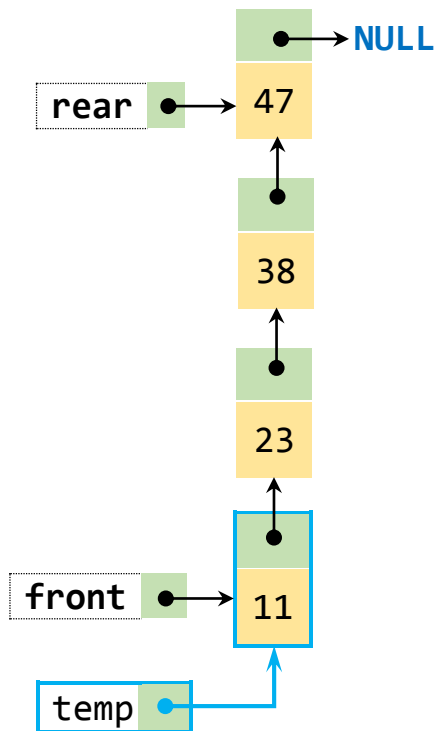
Графическая иллюстрация:



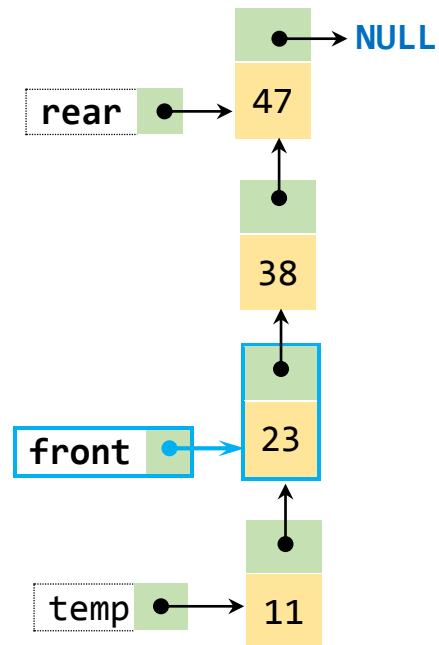
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ОЧЕРЕДИ (метод `deQueue()`)

Графическая иллюстрация:

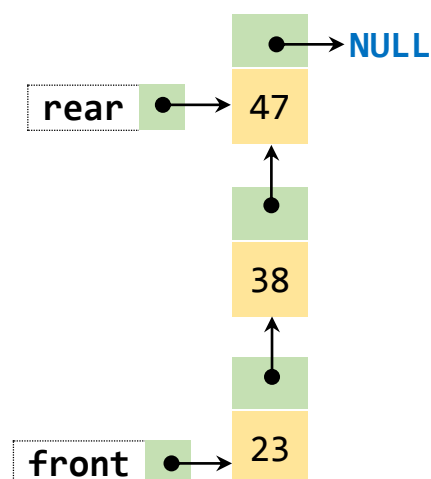
1



2



3



Контрольные вопросы

1. Что такое абстрактная структура данных?
2. Основное свойство структуры данных очередь?
3. Напишите алгоритм основных операций с очередью.
4. Какие способы реализации структуры данных очередь вы знаете?
5. Каков будет результат последовательности операций
ENQUEUE(Q ,4), DEQUEUE(Q), DEQUEUE(Q), ENQUEUE(Q ,1),
ENQUEUE(Q ,3), ENQUEUE(Q ,6), DEQUEUE(Q), ENQUEUE(Q ,8),
ENQUEUE(Q ,11), ENQUEUE(Q ,2), DEQUEUE(Q), DEQUEUE(Q) к
пустой очереди, хранящейся в массиве $Q[1..6]$? К той же очереди,
уже хранящей 2 элемента?

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1

Разработать консольное приложение, которое с помощью абстрактной структуры данных ОЧЕРЕДЬ моделирует работу аэропорта (*или другой системы массового обслуживания*) с одной взлётно-посадочной полосой, которую в каждый момент времени (*цикл for*) может использовать только один самолёт – для взлёта или посадки. Приложение должно:

1. сделать запрос на ввод данных для моделирования:
 - ✓ максимальное кол-во самолётов в очереди на посадку и взлёт;
 - ✓ интервал времени для моделирования;
 - ✓ предполагаемое (*ожидаемое*) число прилетающих и взлетающих самолётов в единицу времени;
2. используя псевдослучайные числа, удовлетворяющие закону распределения Пуассона, смоделировать кол-во запросов на взлёт и посадку самолётов в каждый момент времени;
3. в случае, если очередь на посадку не пуста, то использовать взлётную полосу для посадки самолёта, в противном случае использовать её для взлёта;
4. если какая-либо очередь заполнена, «отправить» самолёт на другой аэропорт (*вывести сообщение в консоль*) в случае посадки или «попросить» подождать некоторое время в случае взлёта;
5. все события (*запрос на взлёт/посадку, взлёт/посадка, отказ во взлёте/посадке*) сопровождаются соответствующими сообщениями в консоли;
6. вести статистику:
 - ✓ сколько получено запросов на взлёт/посадку;

- ✓ сколько принято запросов на взлёт/посадку;
 - ✓ сколько самолётов взлетело/приземлилось;
 - ✓ сколько было отказов на взлёт/посадку;
 - ✓ общее время ожидания взлетевших/приземлившихся самолётов;
 - ✓ время простоя взлётной полосы (*обе очереди пусты*).
7. по окончании интервала моделирования (*цикл **for***) вывести в консоль (*файл*) отчёт, который содержит:
1. общее кол-во обработанных запросов;
 2. кол-во запросов на взлёт/посадку;
 3. кол-во принятых запросов на взлёт/посадку;
 4. кол-во отклонённых запросов на взлёт/посадку;
 5. кол-во взлетевших/приземлившихся самолётов;
 6. кол-во самолётов, оставшихся в очереди на взлёт/посадку;
 7. время простоя взлётной полосы в процентах;
 8. среднее время ожидания для взлёта/приземления;
 9. среднее кол-во поступивших запросов на взлёт/приземление.

Для реализации создать классы **Runway** и **Plane**, которые описывают аэропорт (*взлётную полосу*) и самолёт соответственно.

Для генерации псевдослучайного числа с законом распределения Пуассона рекомендуется использовать

```
...
#include<random>
...
...
int main(){
    ...
    ...
    std::random_device rd; //генератор случайных чисел.
    std::mt19937 gen(rd()); //генератор псевдослучайных
        чисел, который инициализируется случайным числом
        rd().
    std::poisson_distribution<> distr(mean); //создание
        объекта для генерации случайных чисел с
        распределением Пуассона при заданном среднем
        (ожидаемом) значении mean.
    distr(gen); //генерирование случайного числа с
        распределением Пуассона с заданным параметром
        mean.
```

Задание 2

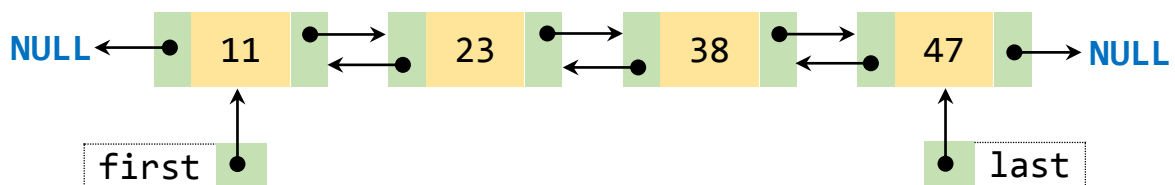
Разработать консольное приложение, которое с помощью абстрактных структур данных **СТЭК** и **ОЧЕРЕДЬ** проверяет введенную с клавиатуры строку текста на палиндром. Приложение должно:

1. делать запрос на ввод строки текста;
2. выводить в консоль:
 - ✓ в случае палиндрома – сообщение об этом;
 - ✓ в противном случае – сообщение об этом и символ, для которого не оказалось равного ему симметричного;
3. делать запрос на выход из приложения.

Лабораторная работа №4

Цель работы: Изучение принципов организации и работы с абстрактной структурой данных **ДВУСВЯЗНЫЙ ЛИНЕЙНЫЙ СПИСОК**.

Двусвязный линейный список — структура данных, где каждый элемент содержит указатель на следующий и предыдущий элементы.



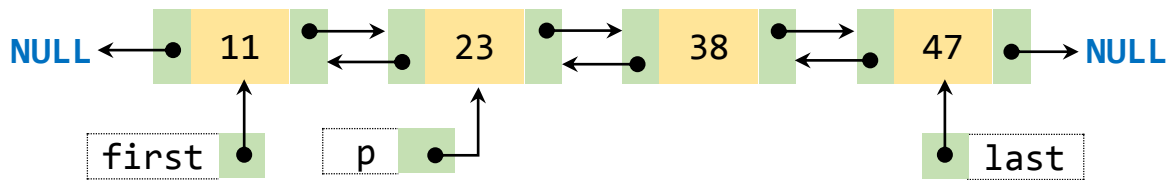
Поддерживаемые операции

1. вставить элемент перед текущим элементом;
2. вставить элемент после текущего элемента;
3. получить указатель на первый элемент списка.
4. получить указатель на последний элемент списка.
5. получить данные первого элемента списка.
6. получить данные последнего элемента списка.
7. получить указатель на следующий, относительно данного, элемент списка;
8. получить указатель на предыдущий, относительно данного, элемент списка;
9. удалить выбранный элемент из списка;
10. найти элемент с указанной информационной частью и вернуть указатель на него;
11. очистить список;
12. копировать список;
13. вывести в консоль все элементы списка.

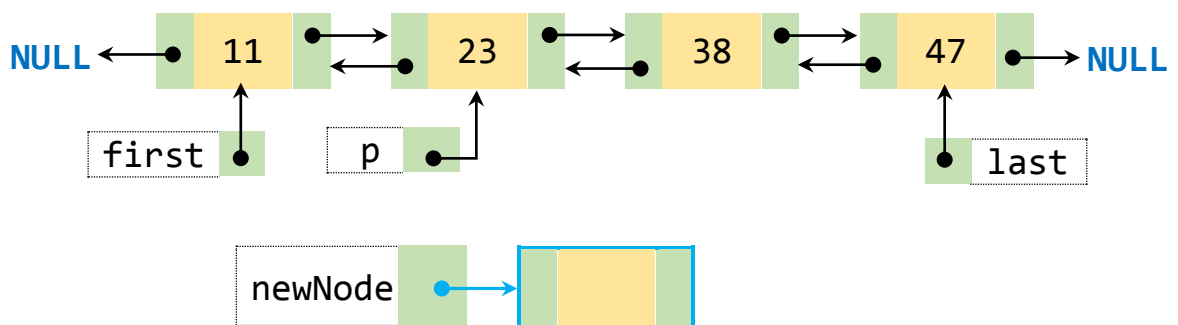
ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ДВУСВЯЗНЫЙ СПИСОК

Графическая иллюстрация:

Пусть список имеет следующий вид:

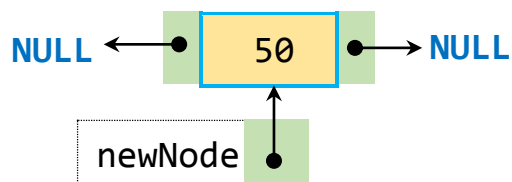


1. `newNode = new Node;`



2. `newNode->data = 50;`

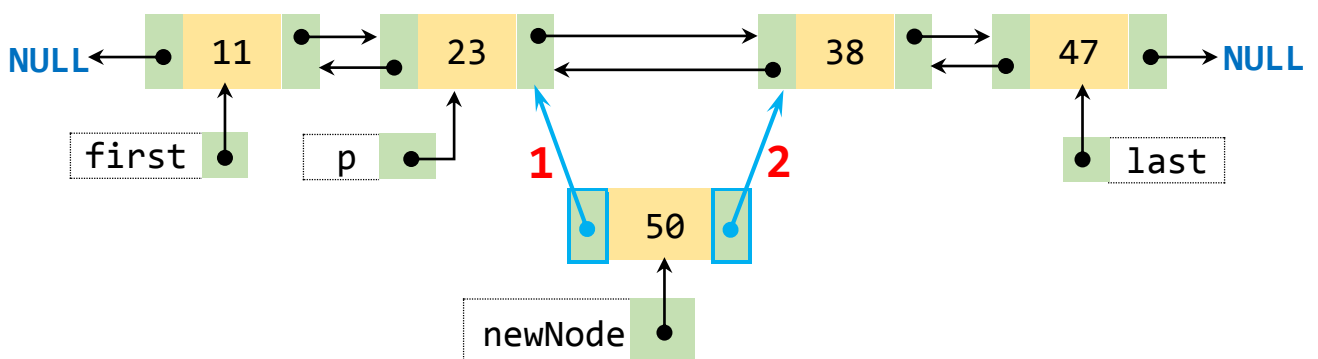
//запись информационной части и указателей



3. `newNode->prev = p;`

4. `newNode->next = p->next;`

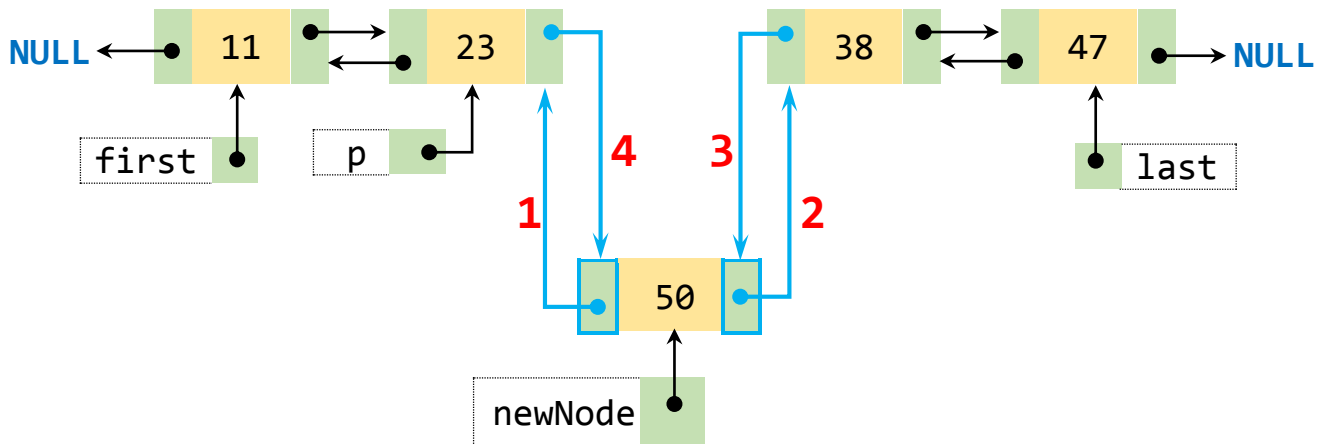
//инициализация указателей нового элемента



5. `p->next->prev = newNode;`

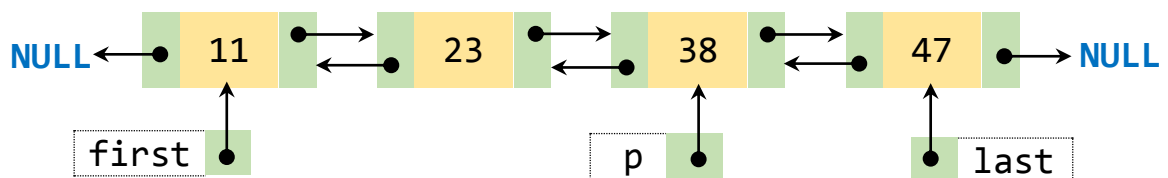
6. `p->next = newNode;`

// перенаправление указателей соседних элементов списка



УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ДВУСВЯЗНОГО СПИСКА

Графическая иллюстрация:



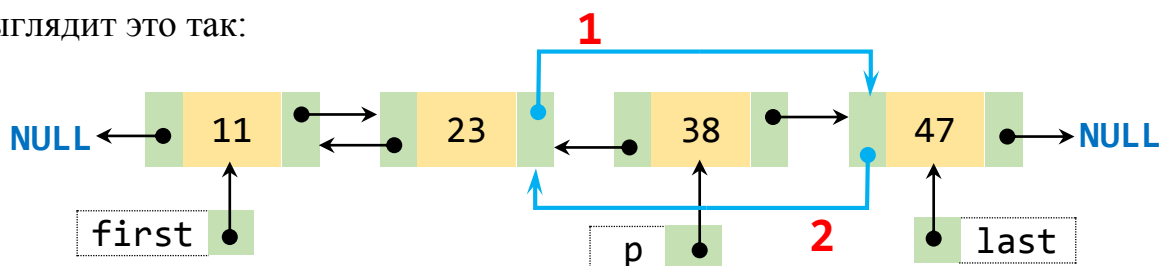
Нужно удалить из списка элемент, на который указывает **p**. Перенаправление указателей и освобождение памяти:

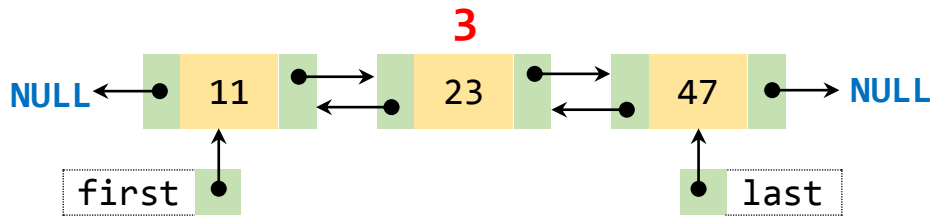
1. `p->prev->next = p->next;`

2. `p->next->prev = p->prev;`

3. `delete p;`

Выглядит это так:





Важно! Как при добавлении, так и при удалении элементов необходимо:

1. Специально рассматривать случай когда элемент добавляется или удаляется в качестве первого или последнего элемента списка. В этом случае перенаправляются указатели **first** и **last**.
2. Проверять, является ли список пустым или нет.

При создании двусвязного списка нужно помнить, что **каждый** элемент списка имеет **2 (два)** указателя, — на предыдущий и на следующий элементы. Первый элемент списка имеет указатель на предыдущий элемент, равный **NULL (nullptr)**. Последний элемент списка имеет указатель на следующий элемент, равный **NULL (nullptr)**.

Контрольные вопросы

1. Что такое абстрактная структура данных?
2. Основное свойство структуры данных двусвязный линейный список?
3. Напишите алгоритм основных операций с двусвязным линейным списком?
4. Перечислите операции двусвязного списка, которые «недоступны» для стека и очереди?
5. Перечислите операции двусвязного списка, которые также «доступны» для стека? Очереди?

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1

Используя абстрактную структуру данных **ДВУСВЯЗНЫЙ ЛИНЕЙНЫЙ СПИСОК**, разработать консольное приложение в соответствии со своим вариантом. Приложение должно обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 1

Динамическая информация о наличии автобусов в автобусном парке и на маршрутах включают в себя следующие сведения о каждом автобусе:

- ✓ номер автобуса;
- ✓ фамилию и инициалы водителя;

- ✓ номер маршрута.

Программа должна обеспечивать:

- ✓ начальное формирование данных о всех автобусах в виде списка;
- ✓ по номеру автобуса:
 - «переводить» автобус из парка на маршрут – удалять данные об этом автобусе из списка автобусов, находящихся в парке, и добавлять эти данные в список автобусов, находящихся на маршруте;
 - «переводить» автобус с маршрута в парк – удалять данных об этом автобусе из списка автобусов, находящихся на маршруте, и добавлять эти данные в список автобусов, находящихся в парке;
- ✓ выдавать по запросу сведения об автобусах, находящихся в парке, или об автобусах, находящихся на маршруте;
- ✓ выдавать по запросу сведения о всех автобусах.

Вариант 2

Список содержит текущую информацию о заявках на авиабилеты.

Каждая заявка включает в себя:

- ✓ пункт назначения;
- ✓ номер рейса;
- ✓ фамилию и инициалы пассажира;
- ✓ желаемую дату вылета.

Программа должна обеспечивать:

- ✓ хранение всех заявок в виде списка;
- ✓ добавление заявок в список;
- ✓ удаление заявок из списка;
- ✓ вывод заявок по заданному номеру рейса и дате вылета;
- ✓ вывод всех заявок.

Вариант 3

Список содержит текущую информацию о книгах в библиотеке.

Сведения о книгах включают в себя:

- ✓ номер УДК;
- ✓ фамилию и инициалы автора;
- ✓ название;
- ✓ год издания;
- ✓ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ✓ начальное формирование данных о всех книгах в библиотеке в виде вписка;
- ✓ возможность получить книгу при вводе номера УДК и в зависимости от состояния списка:
 - выдавать книгу «на руки» и уменьшать кол-во экземпляров данной книги в библиотеке на единицу;

- выдавать сообщение о том, что требуемой книги в библиотеке нет;
- требуемая книга находится на руках.
- ✓ возможность вернуть книгу при вводе номера УДК (*кол-во экземпляров книги увеличивается на единицу*);
- ✓ выдавать сведения о наличии книг в библиотеке по запросу.

Вариант 4

Для каждого файла в некотором каталоге содержатся следующие сведения:

- ✓ имя файла;
- ✓ дата создания;
- ✓ количество обращений к файлу.

Программа должна обеспечивать:

- ✓ начальное формирование каталога файлов;
- ✓ вывод каталога файлов;
- ✓ удаление файлов, дата создания которых меньше заданной;
- ✓ выборку файлов с наибольшим количеством обращений.

Вариант 5

Каждая компонента предметного указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти.

Программа должна обеспечивать:

- ✓ начальное формирование предметного указателя;
- ✓ вывод предметного указателя;
- ✓ вывод номера страниц для заданного слова.

Вариант 6

Каждая компонента текста помощи для программы содержит термин (*слово*) и текст, содержащий пояснения к этому термину. Количество строк текста, относящихся к одному термину, от одной до пяти.

Программа должна обеспечивать:

- ✓ начальное формирование текста помощи;
- ✓ вывод текста помощи;
- ✓ вывод поясняющего текста для заданного термина.

Вариант 7

Картотека в бюро обмена квартир содержит следующие сведения о каждой квартире:

- ✓ количество комнат;
- ✓ этаж;
- ✓ площадь;
- ✓ адрес.

Программа должна обеспечивать:

- ✓ начальное формирование картотеки;
- ✓ ввод заявки на обмен;
- ✓ поиск в картотеке подходящего варианта:
 - при равенстве кол-ва комнат и этажа и различии площадей в пределах 10м² выводится соответствующая карточка и удаляется из списка;
 - в противном случае поступившая заявка включается в список;
- ✓ вывод всего списка.

Вариант 8

Анкета для опроса населения содержит две группы вопросов. Первая группа содержит сведения о респонденте:

- ✓ возраст;
- ✓ пол;
- ✓ образование (*начальное, среднее, высшее*).

Вторая группа содержит собственно вопрос анкеты, ответ на который может быть ДА или НЕТ.

Программа должна обеспечивать:

- ✓ начальный ввод анкет и формирование из них линейного списка;
- ✓ ответы на следующие вопросы на основе анализа анкет:
 - а). сколько мужчин старше 40 лет, имеющих высшее образование, ответили ДА на вопрос анкеты;
 - б). сколько женщин моложе 30 лет, имеющих среднее образование, ответили НЕТ на вопрос анкеты;
 - в). сколько мужчин моложе 25 лет, имеющих начальное образование, ответили ДА на вопрос анкеты;
 - г). вывод всех анкет и ответов на вопросы в консоль.

Вариант 9

На междугородной телефонной станции картотека абонентов, содержит сведения о телефонах и их владельцах.

Программа должна обеспечивать:

- ✓ начальное формирование картотеки в виде линейного списка;
- ✓ вывод всей картотеки в консоль;
- ✓ ввод номера телефона и вывод времени разговора;
- ✓ ввод данных абонента и вывод извещения на оплату телефонного разговора.

Вариант 10

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается:

- ✓ номер поезда;

- ✓ станция назначения;
- ✓ время отправления.

Данные в информационной системе организованы в виде линейного списка.

Программа должна обеспечивать:

- ✓ первоначальный ввод данных в информационную систему и формирование линейного списка;
- ✓ вывод всего списка в консоль;
- ✓ ввод номера поезда и вывод всех данных об этом поезде;
- ✓ ввод названия станции назначения и вывод данных обо всех поездах, следующих до этой станции.

Лабораторная работа №5

Цель работы: Изучение принципов организации и работы с абстрактной структурой данных БИНАРНОЕ ДЕРЕВО.

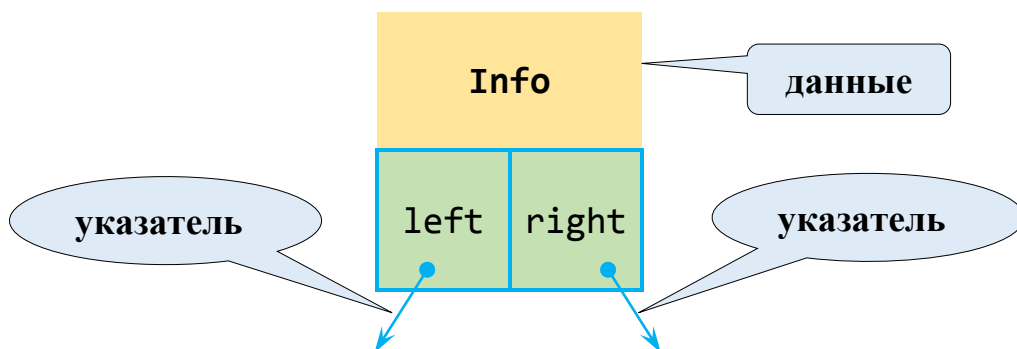
Бинарное дерево — структура данных, состоящая из множества узлов, объединённых связями типа **родитель →ребёнок (parent→child)**. Эти связи удовлетворяют следующим условиям:

1. Существует **единственный** элемент (*узел*), у которого нет «родителя». Этот элемент называется **корнем** (*всего*) дерева.
2. Всякий элемент (*узел*), не являющийся корнем дерева, имеет **ровно одного** родителя.
3. Всякий элемент (*узел*) может иметь одного, двух или ни одного «ребёнка».


Свойство: *Всякий элемент (узел) является корнем дерева, состоящего из его левого и правого поддеревьев.*

Кроме того, элементы дерева определённым образом **упорядочены**. Отношения порядка определяются в зависимости от типа элементов, составляющих дерево. Для чисел это отношения больше, меньше.

Графическая иллюстрация одного узла бинарного дерева:



Каждый элемент бинарного дерева содержит данные (**Info**) и два указателя. Левый (**left**) указатель направлен на «левого» ребёнка (**child**),

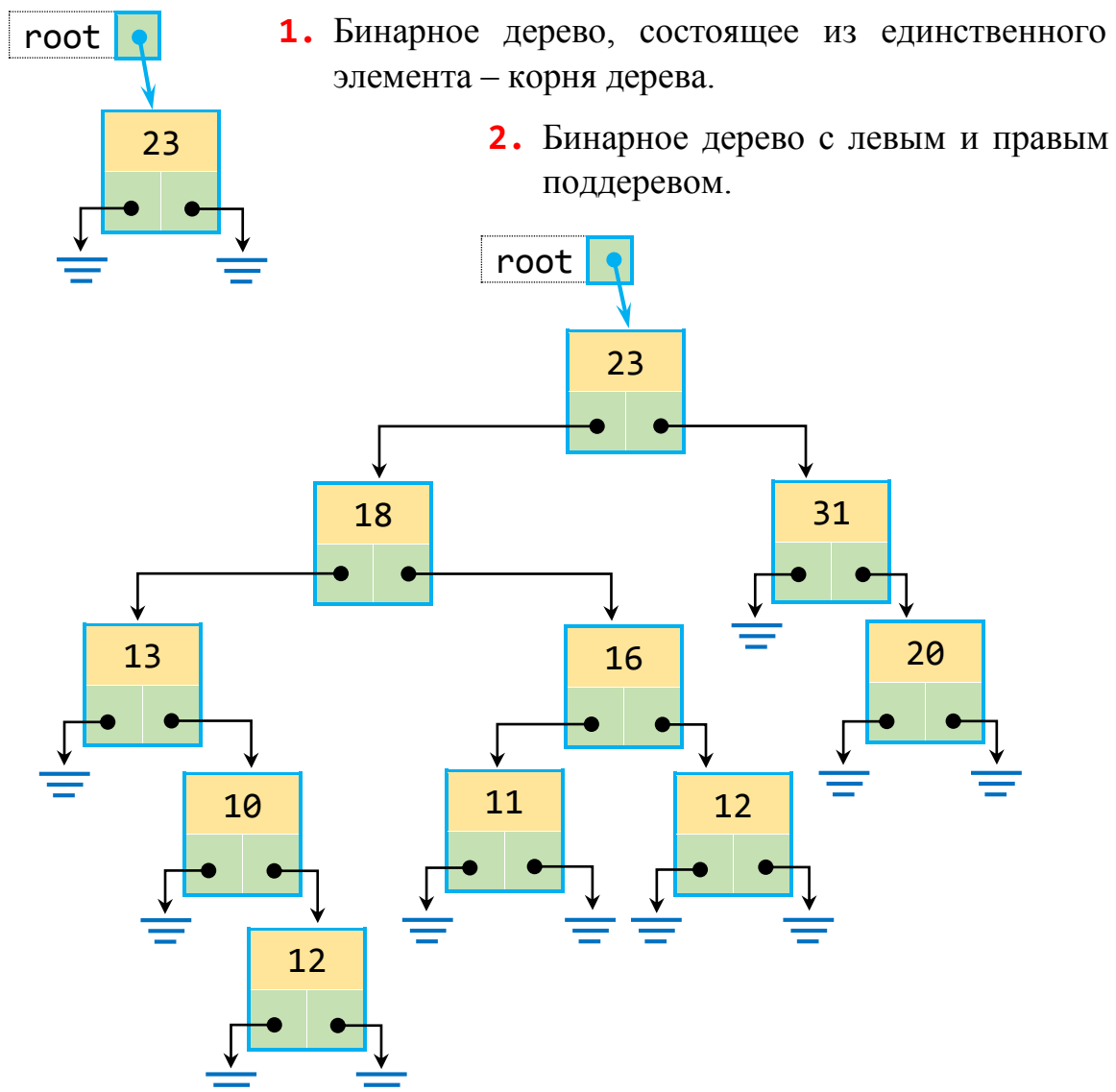
правый (right) — на «правого». Если какого-то (или *обоих*) «детей» нет, то соответствующие указатели равны **NULL**().

Упорядочивание элементов бинарного дерева осуществляется по следующему правилу:

Слева — элемент, **предшествующий** текущему (в случае чисел — *меньшее*), **справа** — **следующий по порядку** (в случае чисел — *большее*). Отношение порядка устанавливается в зависимости от природы самих элементов и (или) логики приложения.

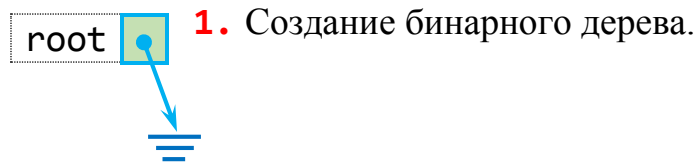
В левом поддереве расположены только *предшествующие* корневому элементы, **в правом** — *следующие* по порядку.

Графическая иллюстрация бинарных деревьев



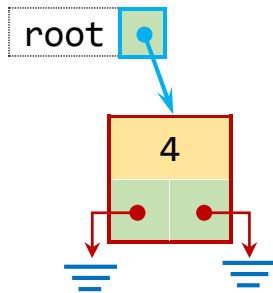
СОЗДАНИЕ БИНАРНОГО ДЕРЕВА И ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ

Графическая иллюстрация:

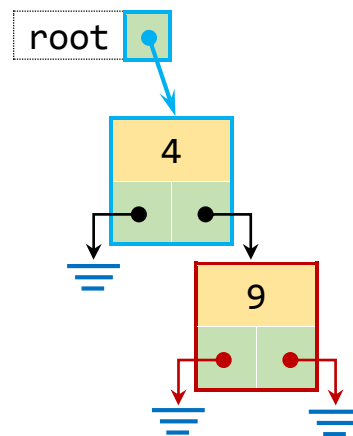


2. Добавление элементов (**Insert**)

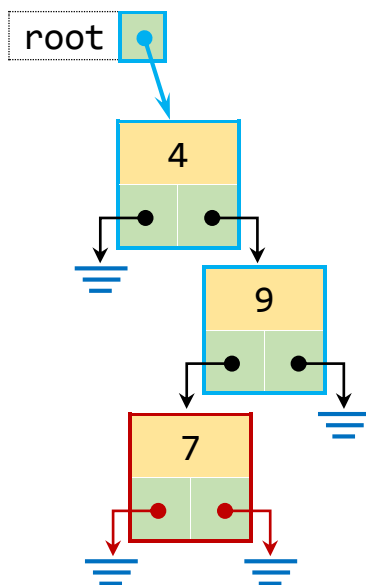
Insert 4:



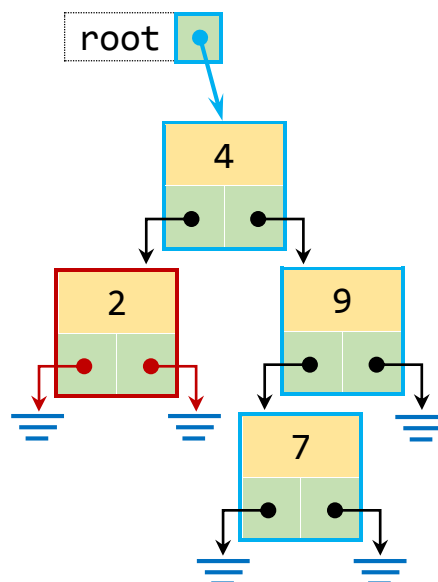
Insert 9(9>4):



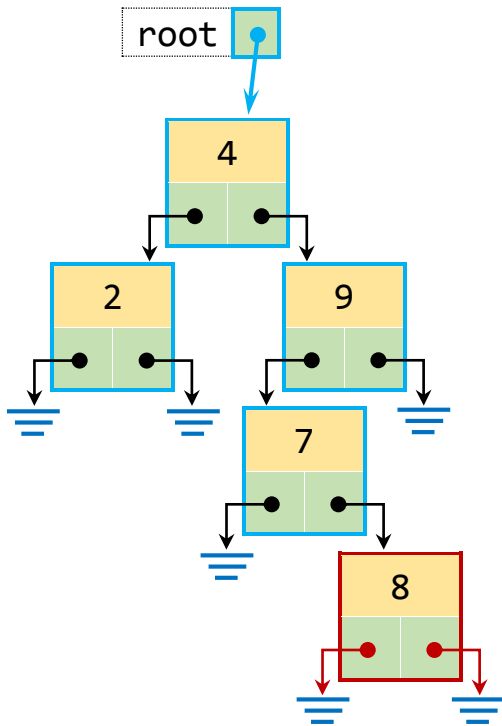
Insert 7(7>4, 7<9):



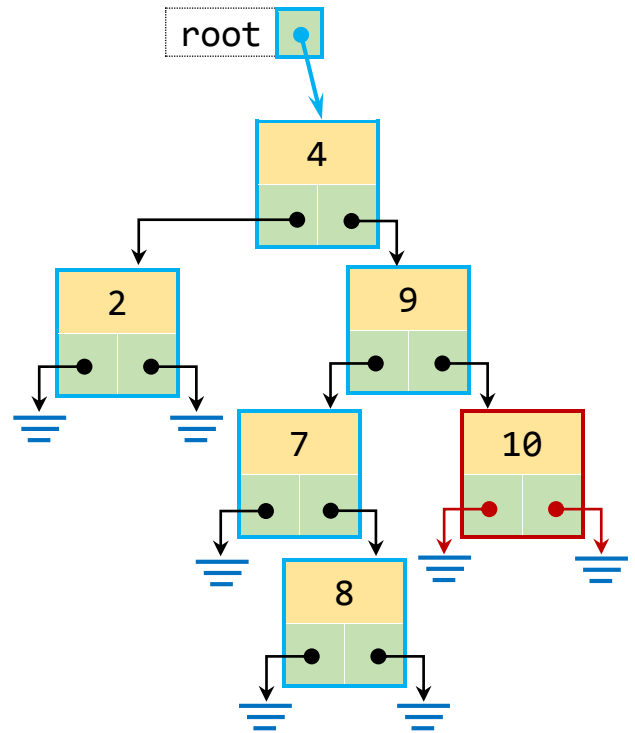
Insert 2(2<4):



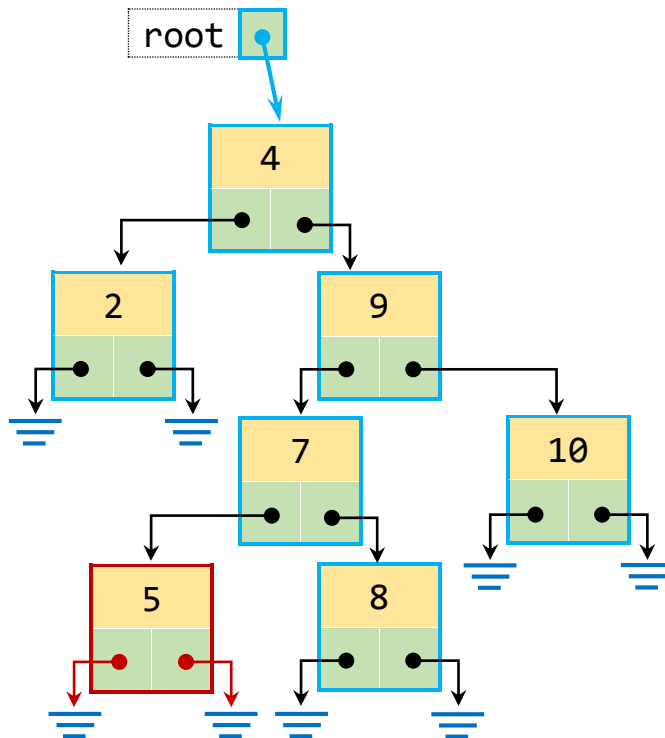
Insert 8($8 > 4$, $8 < 9$, $8 > 7$):



Insert 10($10 > 4$, $10 > 9$):



Insert 5($5 > 4$, $5 < 9$, $5 < 7$):

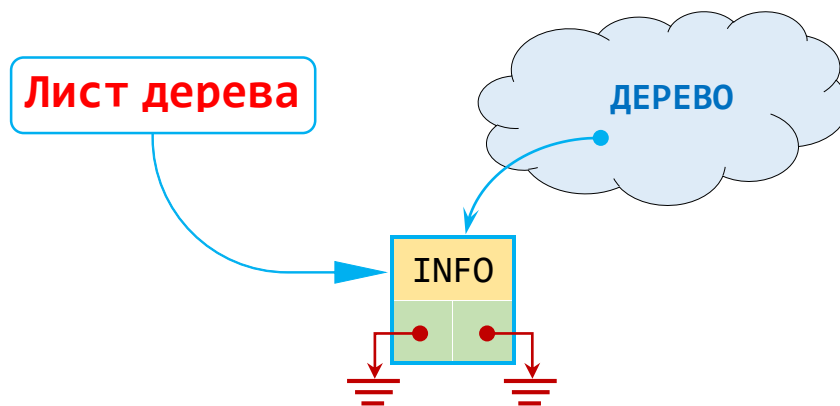


Правило добавления элементов в бинарное дерево:

- Элемент добавляется на соответствующую ему в бинарном дереве позицию в качестве **ЛИСТА**.
- Новый элемент *замещает* какой-либо указатель **NULL** в существующем (или пустом) дереве.

Листом называется элемент дерева, у которого нет «детей». Другими словами, указатели **left** и **right** такого элемента равны **NULL**.

Общий вид элемента типа «ЛИСТ»:



Для добавления элементов в бинарное дерево может быть применена рекурсия, так как каждый раз необходимо проделать одни и те же действия, вплоть до основного случая (*base case*), когда нужное место для добавления элемента найдено.

Каждый раз нужно:

1. «Сравнить» добавляемый элемент с текущим.
2. Если добавляемый элемент **предшествует** текущему (*меньше*), то перейти к левому (указатель **left**) «ребёнку» текущего элемента.
3. Если добавляемый элемент **следует за** текущим (*больше*), то перейти к правому (указатель **right**) «ребёнку» текущего элемента.

Основной случай (*base case*):

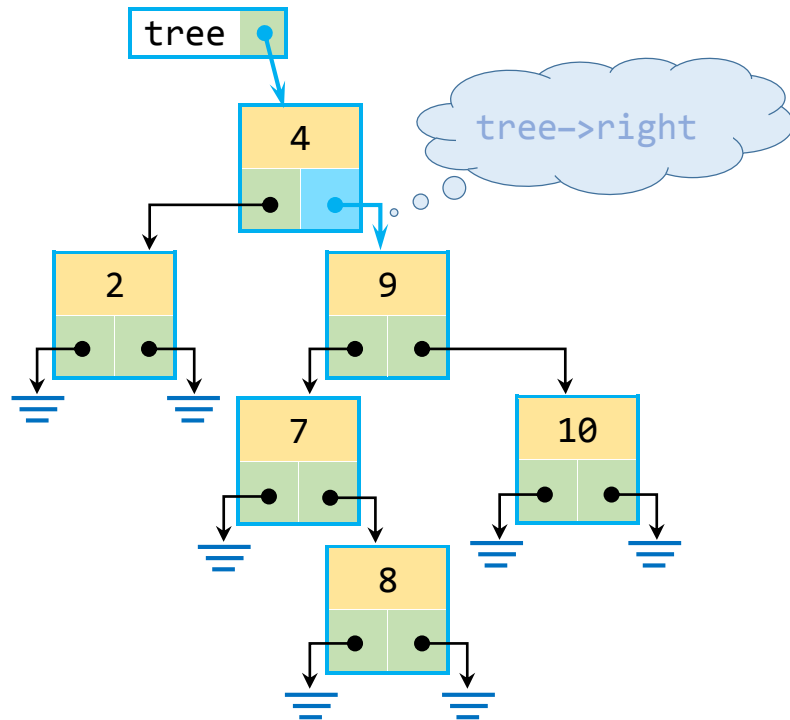
*Если «ребёнок» (левый или правый) равен **NULL**, то место для добавления нового элемента **найден**.*

Теперь

1. Создаётся новый элемент.
2. Записывается информационная часть.
3. Указатели **left** и **right** устанавливаются в **NULL**.

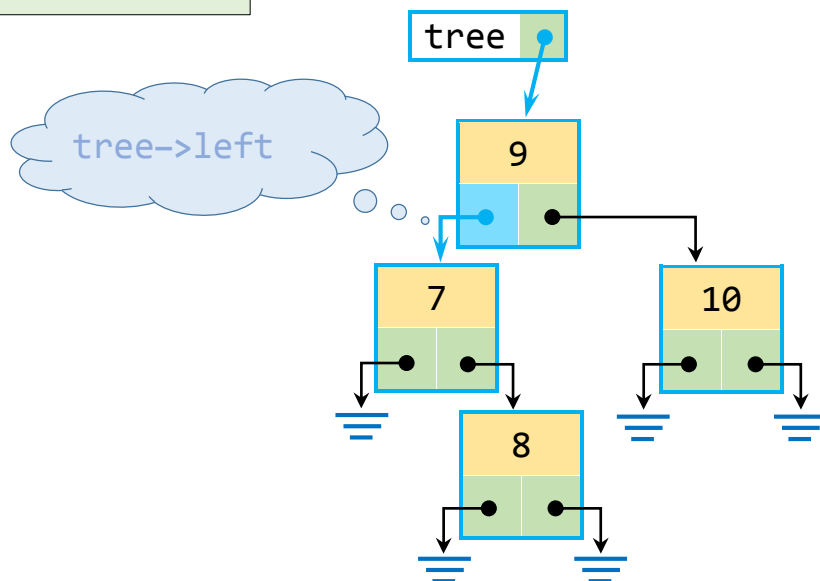
Графическая иллюстрация:

Для определённости рассмотрим случай **Insert 5** ($5 > 4$, $5 < 9$, $5 < 7$).
Исходное бинарное дерево:



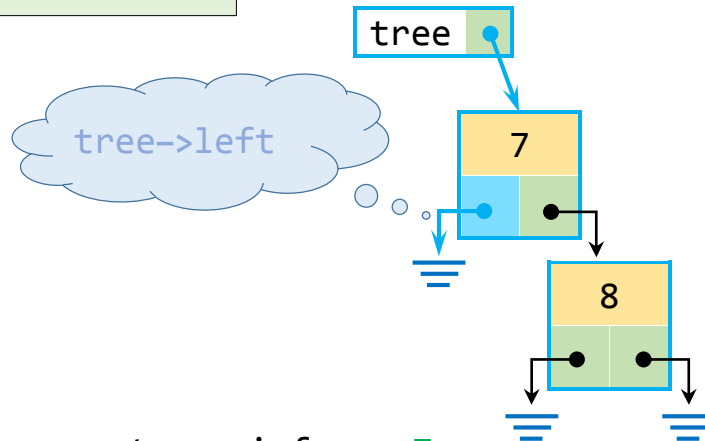
1. Указатель **tree** на текущий элемент, значение **tree->info == 4**.
2. Поскольку $5 > \text{tree->info}$ ($=4$), переходим для сравнения к корню правого поддерева (**info=9**):

```
tree=tree->right;
```



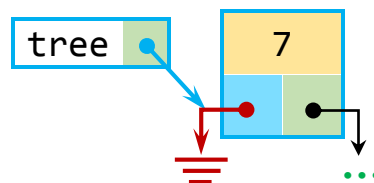
3. Текущее значение `tree->info == 9`.
4. Поскольку `5 < tree->info (=9)`, переходим для сравнения к корню левого поддерева (`info=7`):

```
tree=tree->left;
```



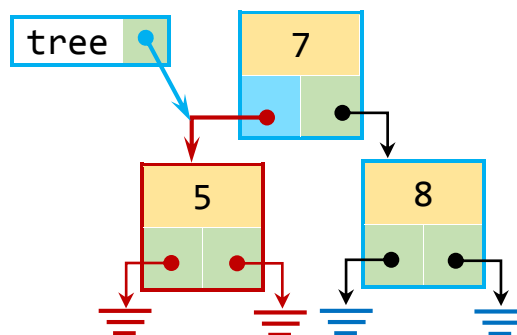
5. Текущее значение `tree->info == 7`.
6. Поскольку `5 < tree->info (=7)`, переходим для сравнения к корню левого поддерева, но указатель `tree->left == NULL`:

```
tree=tree->left (=NULL);
```



7. Поэтому наступает **основной случай** (*base case*) – создание и добавление в дерево нового элемента:

```
tree = new Node(5, nullptr, nullptr);
```



Пример **private**-функции, реализующей рекурсивный алгоритм добавления элементов в бинарное дерево:

```
template<class Type>
void BinaryTree<Type>::insertNode(
    const Type& data, Node<Type>*& tree)
{
    if(tree == nullptr )// Основной случай
        tree = new Node<Type>{item, nullptr, nullptr};
    else if(data < tree->info )
        insert(data, tree->left );
    else
        insert(data, tree->right );
}
```

Пример **public**-функции, члена класса **BinaryTree**:

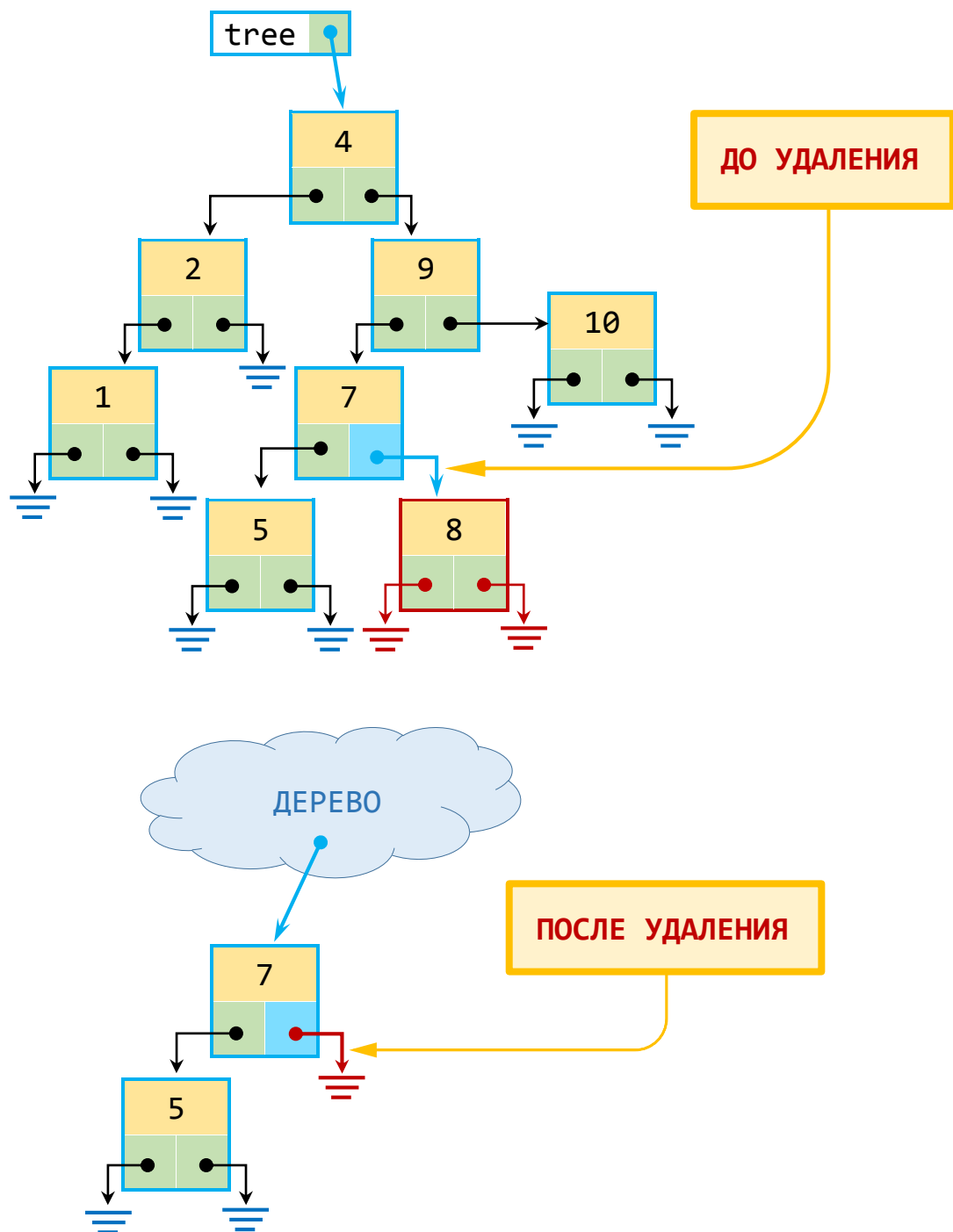
```
template<class Type>
void BinaryTree<Type>::insert(const Type& data)
{
    insert(data, root);
}
```

Важно! Для правильной работы алгоритма, значение **info** добавляемого элемента должно иметь **уникальное** значение!

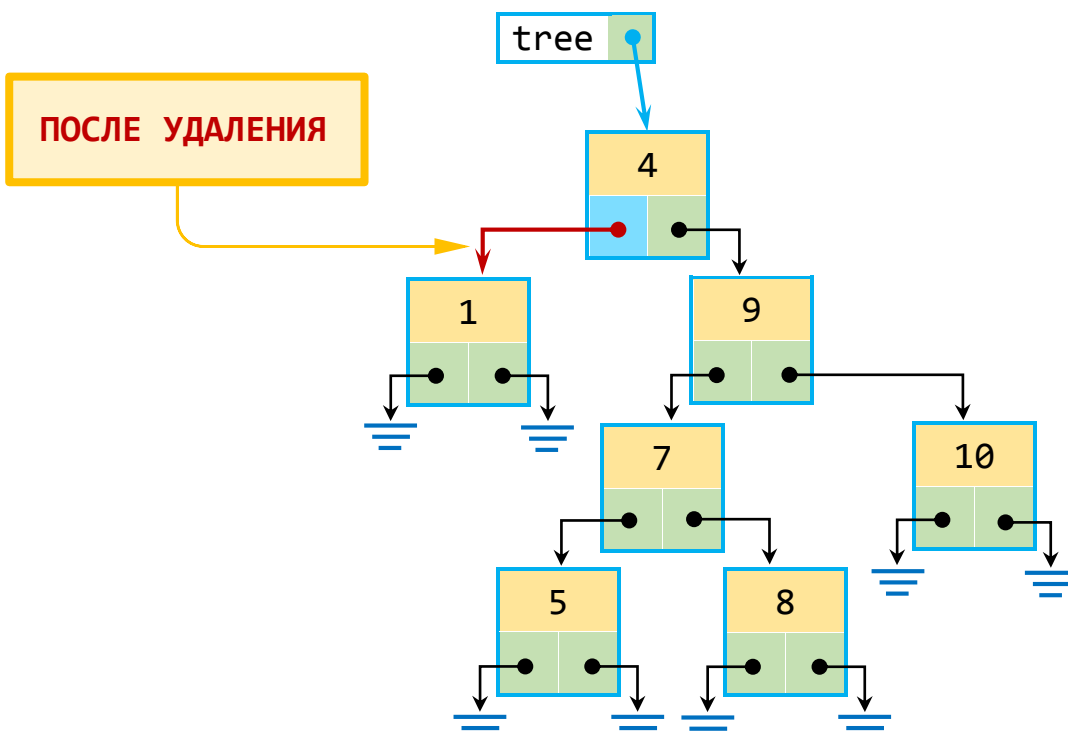
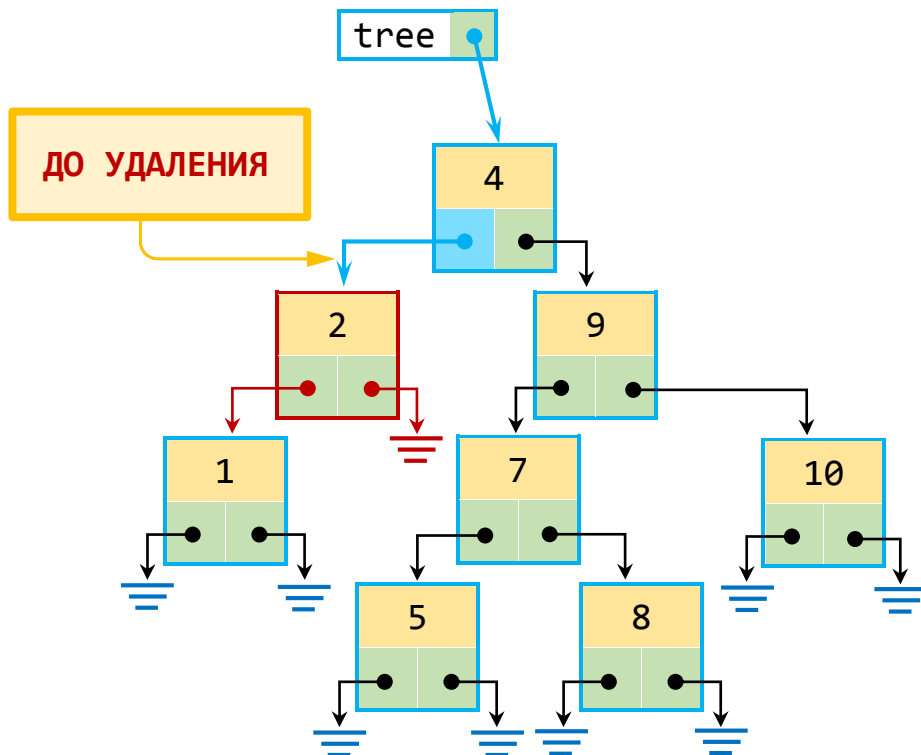
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ БИНАРНОГО ДЕРЕВА

При удалении элементов из бинарного дерева рассматривается 3 (три) различных случая:

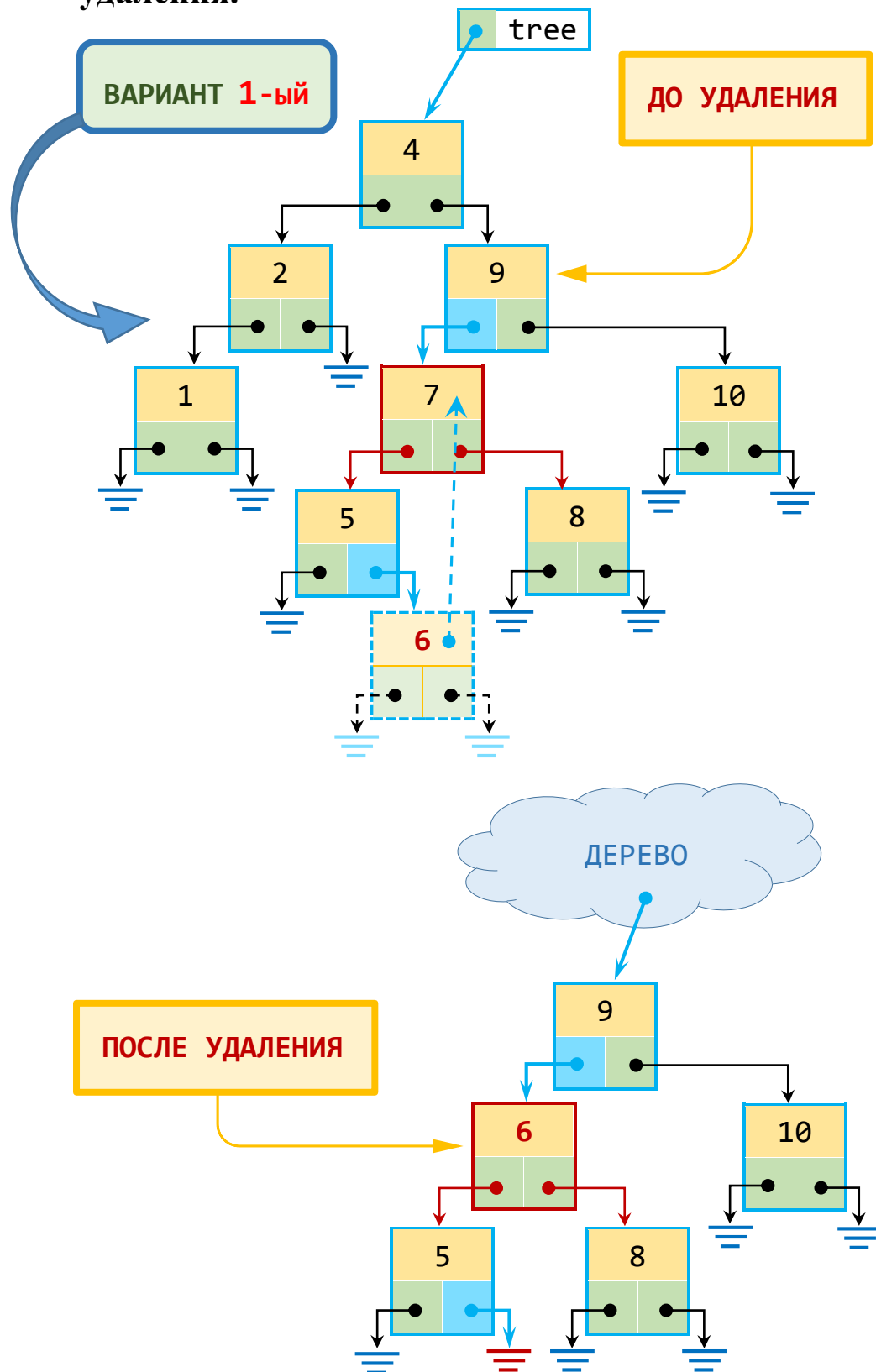
1. Удаляется «ЛИСТ». Оба указателя **left** и **right** равны **NULL**.

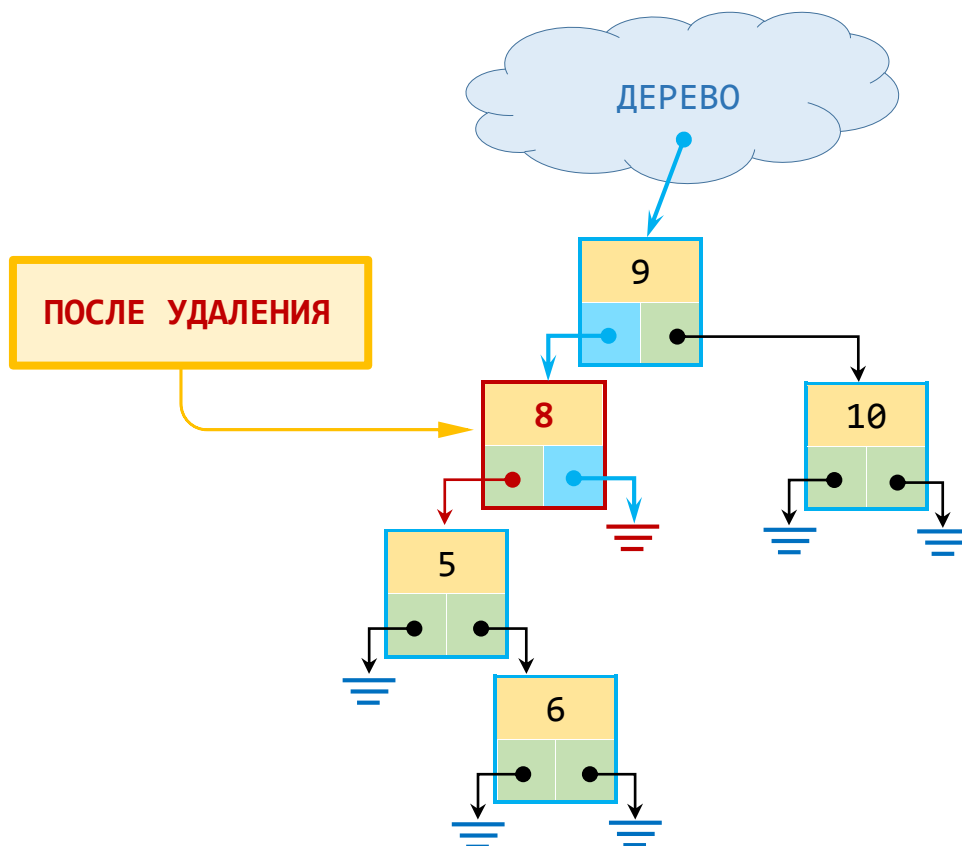
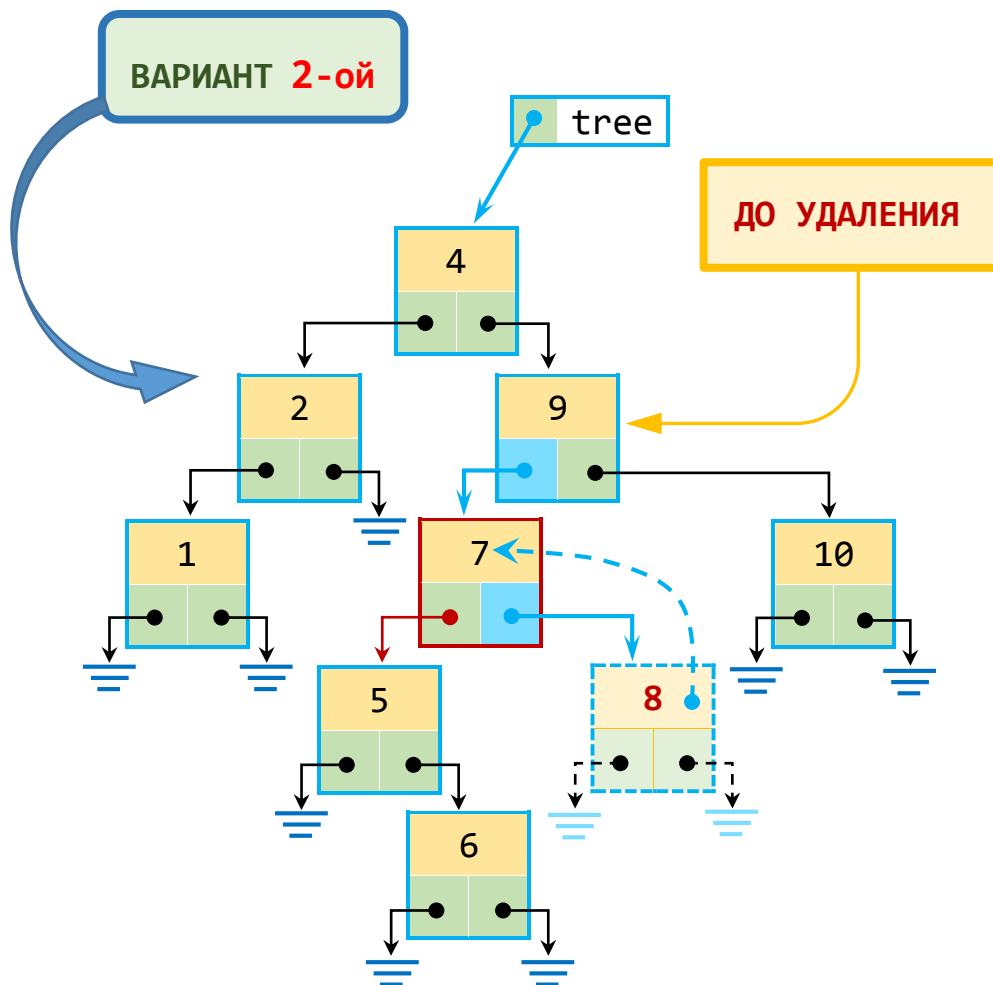


2. Удаляется элемент с **ОДНИМ** «ребёнком». Один из указателей **left** или **right** элемента **равен NULL**.



3. Удаляется элемент с **ДВУМЯ** «детьми». Оба указателя **left** и **right** не равны **NULL**. Возможно **2** варианта удаления.

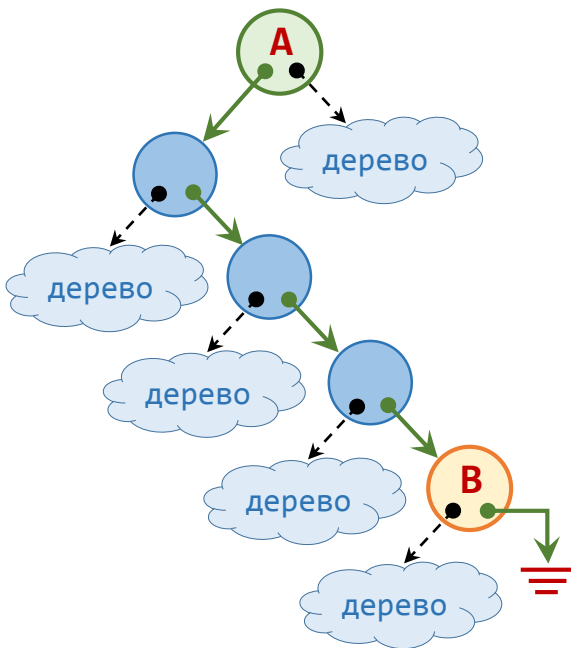




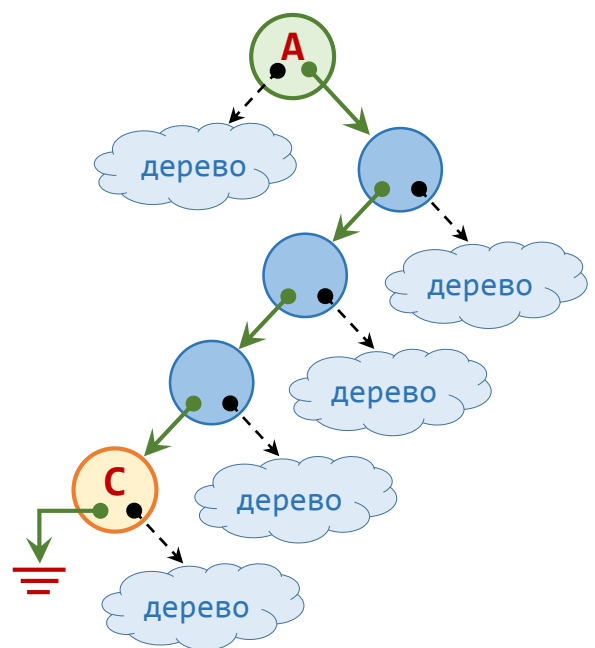
Важно! При удалении из дерева элемента, у которого оба указателя **left** и **right** не равны **NULL**, фактически происходит перезапись информационной части этого элемента, а не его удаление из структуры данных. Удаляется при этом **другой** элемент. Для этого

- 1.** ищется **предыдущий** или **следующий** относительно «удаляемого» элемент, обозначим его **A**;
- 2.** информационная часть элемента **A** записывается **вместо** информационной части «удаляемого» элемента;
- 3.** после этого элемент **A** удаляется из структуры дерева.

В силу самой структуры дерева, **предыдущий** или **следующий** относительно данного элемент **ВСЕГДА** будет либо «листом», либо узлом с одним «ребёнком». Поэтому удаление такого узла сводится к случаю **1** или **2**.



Элемент В – «наибольший» среди «меньших», то есть предшественник.



Элемент С – «наименьший» среди «больших», то есть следующий.

Правила удаления элементов из дерева в случаях 1 или 2:

- *при удалении «листа» соответствующий указатель родительского элемента перенаправляется **на любой** из указателей удаляемого элемента (**NULL**);*

- *при удалении узла с одним дочерним элементом, соответствующий указатель родительского элемента перенаправляется на тот из указателей удаляемого узла, который **не равен NULL**.*

Фактически, при удалении узла из структуры дерева, перенаправляется указатель **родительского** элемента.

Для реализации указанного алгоритма можно применить **2** функции:

- 1. первая** принимает в качестве аргумента указатель на элемент дерева и «удаляет» его в зависимости от количества дочерних элементов («лист», один дочерний элемент, два дочерних элемента) соответствующим образом;
- 2. вторая** принимает в качестве аргумента данные (значение поля **info**), которые требуется удалить, и, если находит их в дереве, передаёт указатель на соответствующий узел дерева первой функции.

Вариант функции №1, реализующей удаление данного узла дерева в зависимости от количества дочерних элементов.

```
template<class Type>
void deleteNode (Node<Type>*& tree)
{
    Node<Type>* temp;

    temp = tree;
    if (tree->left == nullptr )
        // Один или ни одного дочернего узла
    {
        tree = tree->right;
        delete temp;
    }
    elseif ( tree->right == nullptr )
        // Один или ни одного дочернего узла
    {
        tree = tree->left;
        delete temp;
    }
    else // Ровно два дочерних узла
    {
        Node<Type>* pred= GetPredecessor (tree) ;
```

```

        // Находит «предыдущий» элемент
tree->info = pred->info;
        // Записывает данные в «удаляемый» узел
deleteNode(pred); //Удаляет «предыдущий» элемент
}

```

Вариант функции №2 (рекурсия), которая ищет узел с данными, которые требуется удалить, и передаёт указатель на этот узел функции №1.

```

template<class Type>
void delete(Node<Type>*& tree, const Type& data)
{
    if(data < tree->info ) // Если данные «меньше», чем
        у текущего узла ...
        delete(tree->left, data); // Идём налево
    elseif( data > tree->info ) // Если данные «больше»,
        чем у текущего узла ...
        delete(tree->right, data); // Идём направо
    else // Узел найден, вызываем ф-цию №1
        deleteNode(tree);
}

```

Функция GetPredecessor() возвращает указатель на «предыдущий» элемент.

```

template<class Type>
Node<Type>* GetPredecessor(Node<Type>* tree)
{
    if(tree == nullptr ) {...} // Проверка...
    Node<Type>* curr = tree->left;
    // Идём направо, пока есть узлы...
    while(curr->right != nullptr )
        curr = curr->right;
    return curr; //Получаем указатель
}

```

ОБХОД БИНАРНОГО ДЕРЕВА(*tree traversal*)

Существует три распространённых способа обхода или, другими словами, перемещения между узлами бинарного дерева.

1. Симметричный обход (*inorder traversal*).

- 1.1. Обход левого поддерева данного узла.
- 1.2. Посещение данного узла.
- 1.3. Обход правого поддерева данного узла.

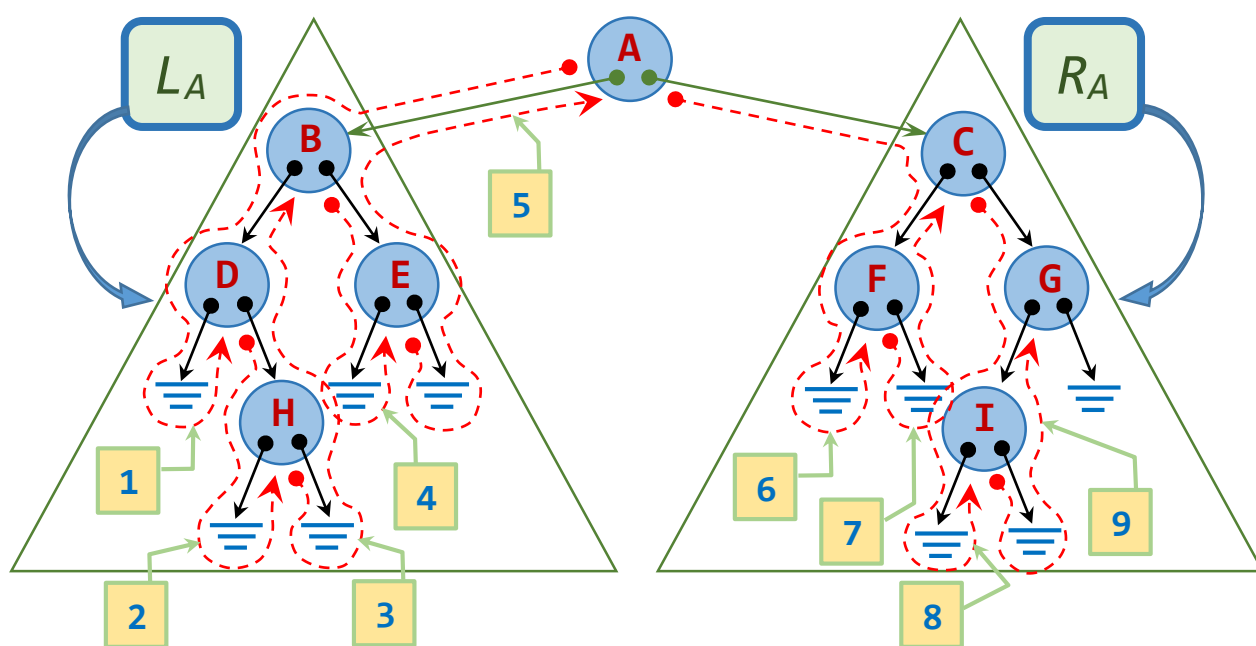
2. Обход в прямом порядке или обход в ширину (*preorder traversal*).

- 2.1. Посещение данного узла.
- 2.2. Обход левого поддерева данного узла.
- 2.3. Обход правого поддерева данного узла.

3. Обход в обратном порядке или обход в глубину (*postorder traversal*).

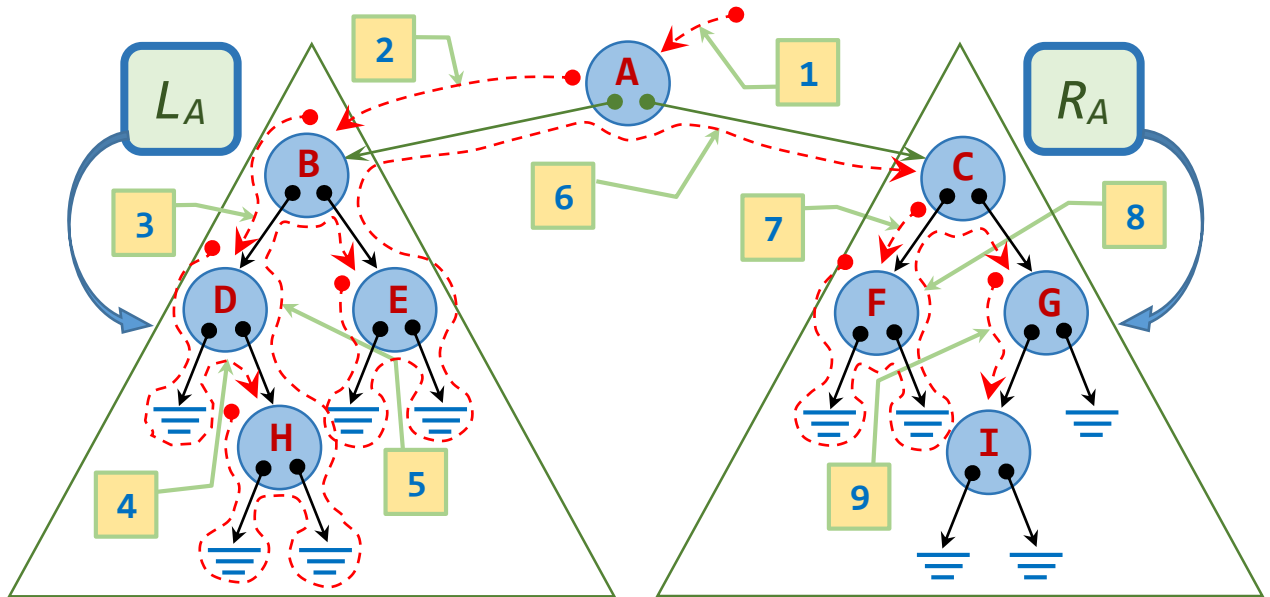
- 3.1. Обход левого поддерева данного узла.
- 3.2. Обход правого поддерева данного узла.
- 3.3. Посещение данного узла.

СИММЕТРИЧНЫЙ ОБХОД(*inorder traversal*)



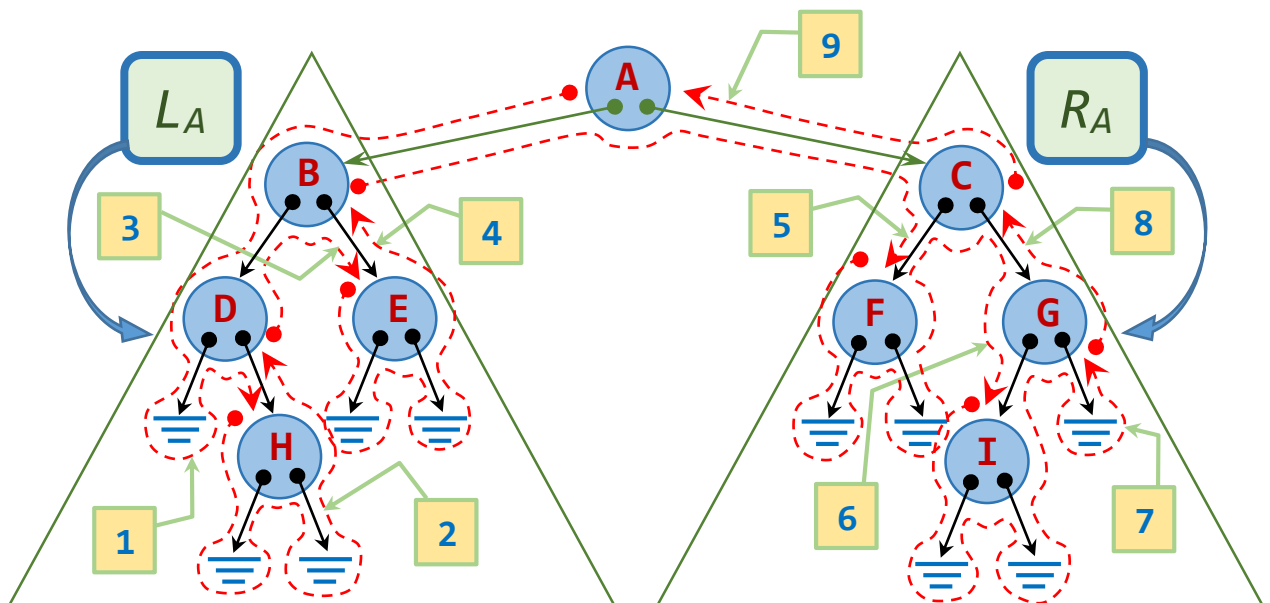
Маршрут симметричного обхода: ***DHBEAFCIG***.

ОБХОД В ШИРИНУ (*preorder traversal*)



Маршрут обхода в ширину: **ABDHECFGI**.

ОБХОД В ГЛУБИНУ (*postorder traversal*)



Маршрут обхода в глубину: **HDEBFIGCA**.

При выборе того или иного маршрута обхода дерева, информационные части узлов могут, например, быть записаны в соответствующую очередь или стек, которые затем и будут использоваться для вывода данных.

Рекурсивный алгоритм обхода бинарного дерева

Пример **private**-функции, реализующей рекурсивный алгоритм обхода бинарного дерева в симметричном (*inorder*) порядке:

```
template<class Type>
void BinaryTree<Type>::inorder (Node<Type>*
                                tree) const
{
    if (tree != nullptr ) //Только если дерево не пусто
    {
        inorder (tree->left ) ; //Рекурсивный вызов
        cout << tree->info << " ";
        //Или добавление в очередь или стек...
        inorder (tree->right ) ; //Рекурсивный вызов
    } ;
}
```

Пример **public**-функции, члена класса **BinaryTree**:

```
template<class Type>
void BinaryTree<Type>::inorderTree () const
{
    inorder (root) ;
}
```

Аналогично реализовываются другие маршруты обхода бинарного дерева.

Поддерживаемые операции

1. добавить элемент;
2. удалить элемент;
3. удалить дерево;
4. проверить, пусто ли дерево;
5. найти и извлечь данные;
6. копировать дерево;
7. обойти дерево по трём различным маршрутам;
8. вывести в консоль (или файл) все элементы дерева.

Пример интерфейса классов `BinaryTree` и `Node`

```
template<class T>
class Node
{
public:
    // Конструктор по умолчанию.
    Node() { left = right = nullptr; }
    // Конструктор с параметрами.
    Node(const T& d, Node<T>* l = nullptr,
        Node<T>* r = nullptr)
    {
        info = d;
        left = l;
        right = r;
    }
    T info; // Информационная часть.
    Node<T> *left, *right; // Указатели на дочерние
    узлы.
};

template<class T>
class BinaryTree
{
public:
    BinaryTree(); // Конструктор по умолчанию. Создаёт
    Пустое дерево.
    ~BinaryTree(); // Деструктор
    const BinaryTree<T>& operator=(const
        BinaryTree<T>& otherTree);
        // Перегрузка оператора присваивания.
    BinaryTree(const BinaryTree<T>& otherTree);
    // Копирующий конструктор
    bool isEmpty() const; // Отвечает на вопрос содержит
    дерево элементы или нет. Возвращает true, если дерево не
    пусто и false в противном случае.
    void print() const; // Выводит данные из дерева.
    int nodeCount() const; // Возвращает количество
```

```

    узлов в дереве.
void destroy() {
    clear(root);
}; // Удаляет все узлы дерева. В результате дерево
    пусто и root == NULL;
void inorderTree() const{
    inorder(root);
}; // Симметричный обход дерева.
void preorderTree() const{
    preorder(root);
}; // Обход дерева в ширину.
void postorderTree() const{
    postorder(root);
}; // Обход дерева в глубину.
T* retrieveItem(const T& item) const{
    return search(root, item);
}; // Возвращает указатель на item, если найдено
void deleteItem(const T& item){
    return delete(root, item);
};
// Удаляет узел с данными item из дерева.
void insertItem(const T& item){
    return insert(root, item);
}; // Добавляет узел с данными item в дерево.
private:
Node<T>*
    copyTree(BinaryTree<T>* otherTree) const;
// Функция, создающая копию дерева otherTree.
Node<T>* root; // Указатель на корень дерева.
T* search(Node<T>*, const T&) const;
// Ищет и возвращает данные.
void inorder(Node<T>*) const; // Обход дерева
void preorder(Node<T>*) const; // Обход дерева
void postorder(Node<T>*) const; // Обход дерева
void clear(Node<T>* &); // Удаляет все узлы дерева
    и устанавливает указатель root в NULL.
void delete(Node<T>* &, const T&);

```

```
void deleteNode (Node<T>* &) ;  
void insert (Node<T>* &, const T&) ;  
};
```

Контрольные вопросы

1. Что такое абстрактная структура данных?
2. Что такое *бинарное поисковое* дерево?
3. Основные свойства и способы построения структуры данных бинарное поисковое дерево?
4. Напишите алгоритм основных операций для структуры данных бинарное поисковое дерево?
5. В каком порядке посещаются узлы бинарного дерева в случае:
 - ✓ симметричного обхода – *inorder traversal*;
 - ✓ обхода в прямом порядке (*в ширину*) – *preorder traversal*;
 - ✓ обхода в обратном порядке (*в глубину*) – *postorder traversal*.

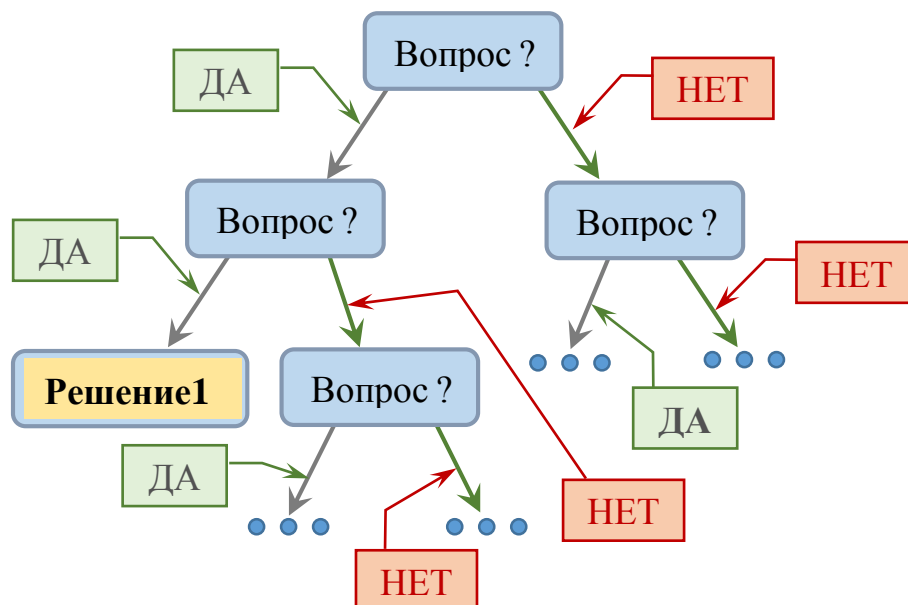
ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Задание 1

Используя абстрактную структуру данных БИНАРНОЕ ДЕРЕВО, реализовать свой вариант Задания №1 из Лабораторной работы №3. Приложение должно обеспечивать диалог с помощью меню и контроль ошибок при вводе. Определить *ключевое поле*, по которому будет происходить упорядочивание бинарного дерева. Обеспечить возможность ПОИСКА данных по ключевому полю.

Задание 2

Используя абстрактную структуру данных БИНАРНОЕ ДЕРЕВО, разработать и реализовать ДЕРЕВО РЕШЕНИЙ для какой-либо задачи из повседневной жизни (*выдача кредита, страхование имущества, диагностика заболевания и т.п.*) по следующему принципу:



Требования:

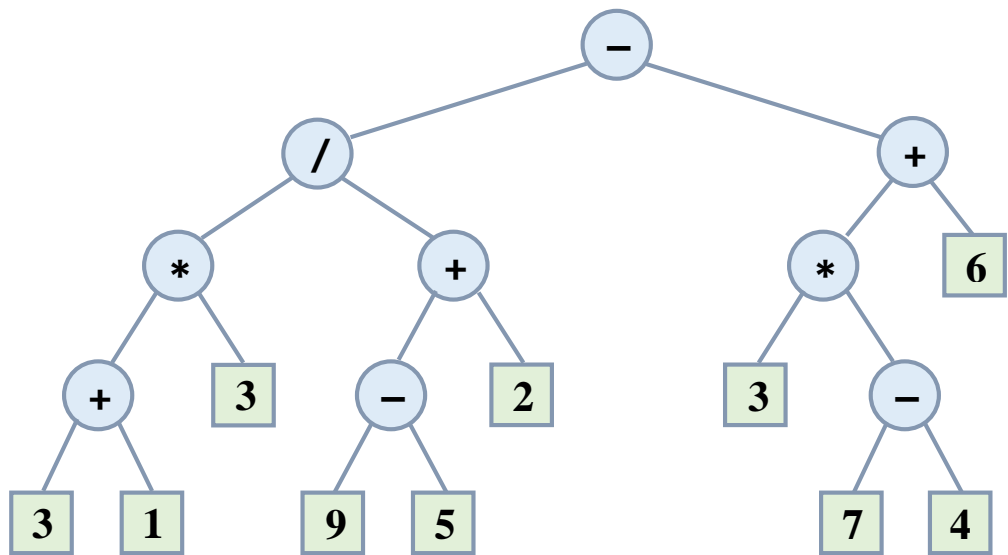
1. дерево должно иметь не менее 4-х уровней;
2. программа должна обеспечить диалог, с помощью которого пользователь может отвечать на вопросы;
3. после окончания опроса (достижения какого-либо решения), программа выводит в консоль решение и ответы, данные пользователем во время опроса.

Задание 3

Используя абстрактную структуру данных БИНАРНОЕ ДЕРЕВО, разработать калькулятор, вычисляющий арифметические выражения из 4-х основных действий и скобок, записанные в постфиксной форме (*postfix notation*), используя подходящий порядок обхода и следующие правила:

1. дерево состоит только из узлов, у которых ровно 2 ребёнка и листьев;
2. листьям дерева соответствуют операнды – числа;
3. остальным узлам соответствуют бинарные операции – действия над 2-мя числами.

Например, выражению $((3+1) \times 3) / ((9-5)+2) - (3 \times (7-4) + 6)$ соответствует дерево



Белорусский государственный университет

УТВЕРЖДАЮ

Декан гуманитарного факультета

_____ В.Е. Гурский
(подпись)

(дата утверждения)

Регистрационный № УД-_____/р.

Алгоритмы и структуры данных
Учебная программа учреждения высшего образования
по учебной дисциплине: 1-31 03 07-03
для специальности: Прикладная информатика

Факультет Гуманитарный

Кафедра Информационных технологий

Курс (курсы) 1

Семестр (семестры) 2

Лекции 34

Экзамен 2

Практические (семинарские)
занятия

Зачет

Лабораторные
занятия 34

Курсовая работа (проект)

Аудиторных часов по
учебной дисциплине 68

Всего часов по
учебной дисциплине 158

Форма получения
высшего образования очное

Составил(а) А.В. Овсянников, кандидат технических наук, доцент

2013 г.

Учебная программа составлена на основе типовой учебной программы «Алгоритмы и структуры данных» Регистрационный номер №ТД – G 278/тип, дата утверждения 16.06.2010

Рассмотрена и рекомендована к утверждению кафедрой
Информационных технологий

(дата, номер протокола)

Заведующий кафедрой

(подпись) В.А. Нифагин

Одобрена и рекомендована к утверждению Научно-методическим советом¹
Гуманитарного факультета БГУ

28.06.2013 г. №9

Председатель

(подпись) О.В. Немкович

¹ Учебная программа может быть рекомендована к утверждению Советом факультета или методической комиссией факультета, или общеуниверситетской (общакадемической) кафедрой.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Учебная программа «Алгоритмы и структуры данных» разработана для специальности 1-31 03 07-03 «Прикладная информатика» высших учебных заведений.

Дисциплина «Алгоритмы и структуры данных» знакомит студентов с фундаментальными понятиями, используемыми при разработке алгоритмов и оценке их трудоемкости.

Цель дисциплины - изучение подходов к разработке эффективных алгоритмов для разнообразных задач дискретной и комбинаторной оптимизации.

Задачи дисциплины - выработать навыки по оценке трудоемкости алгоритмов и по применению современных структур данных для эффективной реализации различных базовых операций.

В курсе рассматриваются такие фундаментальные понятия как информация, размерность задачи и трудоемкость алгоритмов. Особое внимание уделено способам определения трудоемкости алгоритмов с помощью таких методов, как составление и решение рекуррентных уравнений. Наряду с классическим подходом оценки трудоемкости рассматриваются также способы определения усредненной оценки трудоемкости алгоритма для группы операций.

Основой для дисциплины «Алгоритмы и структуры данных» являются следующие дисциплины: «Дискретная математика и математическая логика», «Теория вероятностей и математическая статистика», «Программирование». Изучение курса позволяет дать студентам базу, необходимую для успешного усвоения материала, а также получить знания, необходимые им в дальнейшем для успешной работы при разработке эффективных алгоритмов,

В результате изучения дисциплины студент должен *знать*:

- понятие размерности задачи и трудоемкости алгоритма;
- основные приемы разработки эффективных алгоритмов: динамическое программирование и метод «разделяй и властвуй»;
- основные структуры данных и трудоемкость базовых операций для них;
- виды поисковых деревьев;
- основные алгоритмы поиска на графах и их трудоемкость; уметь:
- определять трудоемкость основных алгоритмов поиска и внутренней сортировки, используя технику рекуррентных соотношений;
- осуществлять выбор структуры данных для разработки эффективного алгоритма решения задачи;
- реализовывать поисковые деревья;
- реализовывать основные алгоритмы поиска на графах.

уметь:

- определять трудоемкость основных алгоритмов поиска и внутренней сортировки, используя технику рекуррентных соотношений;

- осуществлять выбор структуры данных для разработки эффективного алгоритма решения задачи;
- реализовывать поисковые деревья;
- реализовывать основные алгоритмы поиска на графах.

Учебная программа предусматривает для изучения дисциплины 158 часов, в том числе 68 аудиторных часов: лекции - 34 часа, лабораторные занятия - 34 часа.

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

Тема 1

Понятие структуры данных и алгоритмов.

Определение алгоритма. Формальные свойства алгоритмов. Понятие структуры данных. Классификация структур данных. Операции над структурами данных. Структурность данных и технология программирования.

Тема 2

Абстрактные вычислительные машины.

Машина Поста. Машина Тьюринга, детерминированная и недетерминированная машина Тьюринга. Вероятностная машина Тьюринга.

Тема 3

Анализ алгоритмов.

Сравнительные оценки алгоритмов. Классификация алгоритмов по виду функции трудоёмкости. Асимптотический анализ функций. Оценка Θ , O , Ω и др. Элементарные операции в языке записи алгоритмов (следование, ветвление, цикл)

Трудоёмкость алгоритмов и временные оценки.

Примеры анализа простых алгоритмов. Переход к временным оценкам. Пооперационный анализ. Метод Гиббсона. Метод прямого определения среднего времени. Пример пооперационного временного анализа.

Теория сложности вычислений, классы сложности задач.

Теоретический предел трудоёмкости задачи. Задача умножения матриц. Классы P и NP , NP – полные задачи. Примеры.

Рекуррентные функции и алгоритмы.

Рекуррентные соотношения. Понятие рекуррентного соотношения. Решение рекуррентных уравнений. Примеры рекуррентных уравнений.

Тема 4

Структуры данных.

Элементарные структуры данных (массивы, списки, стеки, очереди). Связанные списки. Множества. Представление корневых деревьев. Понятие сложных структур данных.

Тема 5

Базовые алгоритмы поиска и сортировки.

Поиск методом полного перебора. Поиск в упорядоченных списках. Поиск в связных списках. Двоичный поиск. Следящий поиск. Сортировка выбором. Сортировка пузырьком. Алгоритм простыми вставками.

Тема 6

Алгоритмы на графах.

Графы. Основные определения. Поиск в глубину и ширину в графе. Пути в графах. Кратчайшие пути. Алгоритм Дейкстры. Минимальные остовные деревья. Алгоритм Борувки. Алгоритм Крускала. Алгоритм Прима.

Тема 7

Организация поиска.

Деревья. Основные определения. Бинарные поисковые деревья. Сбалансированные деревья. Хеш-таблицы.

Тема 8

Методы разработки алгоритмов.

Алгоритмы "разделяй и властвуй". Динамическое программирование. "Жадные" алгоритмы и оптимизационные задачи.

УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Номер раздела, темы, занятия	Название раздела, темы, занятия; перечень изучаемых вопросов	Количество аудиторных часов				Материальное обеспечение занятия	Литература	Формы контроля знаний
		Лекции	Практические (семинарские) занятия	Лабораторные занятия	Управляемая самостоятельная работа			
1	2	3	4	5	6	7	8	9
1	Понятие структуры данных и алгоритмов Определение алгоритма. Формальные свойства алгоритмов. Понятие структуры данных. Классификация структур данных. Операции над структурами данных. Структурность данных и технология программирования.	4		2	2	Эл. през.	[1-4]	Опрос, КСР
2	Абстрактные вычислительные машины Машина Поста. Машина Тьюринга, детерминированная и недетерминированная машина Тьюринга. Вероятностная машина Тьюринга.	2		2	2	Эл. през.	[1-4]	Опрос, КСР
3	Анализ алгоритмов Сравнительные оценки алгоритмов. Классификация алгоритмов по виду функции трудоёмкости. Асимптотический анализ функций. Оценка Θ , O , Ω и др. Элементарные операции в языке записи алгоритмов (следование, ветвление, цикл)	2		2	2	Эл. през.	[1-4]	Опрос, КСР
4	Трудоёмкость алгоритмов и временные оценки Примеры анализа простых алгоритмов. Переход к временным оценкам. Пооперационный анализ. Метод Гиббсона. Метод прямого определения среднего времени. Пример пооперационного временного анализа.	2		2	2	Эл. през.	[1-6]	Опрос, КСР

Продолжение Таблицы «Учебно-методическая карта учебной дисциплины»

1	2	3	4	5	6	7	8	9
5	Теория сложности вычислений, классы сложности задач Теоретический предел трудоемкости задачи. Задача умножения матриц. Классы P и NP, NP – полные задачи. Примеры.	2		2	2	Эл. през.	[1-6]	Опрос, КСР
6	Рекуррентные функции и алгоритмы Рекуррентные соотношения. Понятие рекуррентного соотношения. Решение рекуррентных уравнений. Примеры рекуррентных уравнений	2		2	2	Эл. през.	[1-6]	Опрос, КСР
7	Структуры данных Элементарные структуры данных (массивы, списки, стеки, очереди). Связанные списки. Множества. Представление корневых деревьев. Понятие сложных структур данных	4		4	4	Эл. през.	[1-6]	Опрос, КСР
8	Базовые алгоритмы поиска и сортировки Поиск методом полного перебора. Поиск в упорядоченных списках. Поиск в связных списках. Двоичный поиск. Следящий поиск. Сортировка выбором. Сортировка пузырьком. Алгоритм простыми вставками	4		4	4	Эл. през.	[1-6]	Опрос, КСР
9	Алгоритмы на графах Поиск в глубину и ширину в графе. Пути в графах. Кратчайшие пути. Алгоритм Дейкстры. Минимальные остовные деревья. Алгоритм Борувки. Алгоритм Крускала. Алгоритм Прима.	4		4	4	Эл. през. и	[1-6]	Опрос, КСР
10	Организация поиска Деревья. Основные определения. Бинарные поисковые деревья. Сбалансированные деревья. Хеш-таблицы	4		4	4	Эл. през.	[1-6]	Опрос, КСР
11	Методы разработки алгоритмов Алгоритмы "разделяй и властвуй". Динамическое программирование. "Жадные" алгоритмы и оптимизационные задачи.	4		6	6	Эл. през.	[1-6]	Опрос, КСР

ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

ОСНОВНАЯ ЛИТЕРАТУРА

1. Ахо, А. В. Структуры данных и алгоритмы / А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман. : Учеб. пособие/ пер. с англ. М. : Вильямс, 2000. 384 с.
2. Алгоритмы : построение и анализ / Т. Кормен, и др. М. : Вильямс, 2005. 1296 с.
3. Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ. / Р. Седжвик. - К.: Издательство «ДиаСофт», 2001.- 688 с.
4. Фундаментальные алгоритмы на С++. Алгоритмы на графах: Пер. с англ. / Р. Седжвик. - К.: Издательство «ДиаСофт», 2002.- 688 с.
5. Котов В. М. Разработка и анализ алгоритмов : теория и практика: пособие для студентов мат. и физ. специальностей / В. М. Котов, Е. П. Соболевская. - Минск : БГУ, 2009. - 251 с.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы и структуры данных. / Н. Вирт, М.: Мир, 1989.-360с.
2. Вирт Н. Алгоритмы и структуры данных. / Н. Вирт, СПб.:Невский диалект, 2001.-352с
3. Котов, В. М. Структуры данных и алгоритмы : теория и практика :/ В.М. Котов, Е- П. Соболевская. : учеб. пособие. Минск : БГУ, 2004. 252 с.
4. Головешкин В. А. Теория рекурсии для программистов. / В.А. Головешкин М. В.: Ульянов М.: ФИЗМАТЛИТ, 2006. 296 с.

ПЕРЕЧЕНЬ ИСПОЛЬЗУЕМЫХ СРЕДСТВ ДИАГНОСТИКИ РЕЗУЛЬТАТОВ УЧЕБНОЙ ДЕЯТЕЛЬНОСТИ

Оценка промежуточных учебных достижений студента также осуществляется по десятибалльной шкале.

Для оценки достижений студента используется следующий диагностический инструментарий:

- защита выполненных на лабораторных занятиях индивидуальных заданий;
- проведение текущих контрольных вопросов по отдельным темам;
- выступление студента на конференции по подготовленному реферату;
- сдача зачета по дисциплине;
- сдача экзамена.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ РАБОТ

1. Работа со структурами данных. Сложные структуры: стеки, очереди, деревья, кучи, контейнеры и т.д.
2. Алгоритмы поиска.
3. Алгоритмы сортировки.
4. Алгоритмы на графах.
5. Поисковые деревья.

ПРИЛОЖЕНИЕ 4

ПРОТОКОЛ СОГЛАСОВАНИЯ УЧЕБНОЙ ПРОГРАММЫ (примерная форма)

Название учебной дисциплины, с которой требуется согласование	Название кафедры	Предложения об изменениях в содержании учебной программы учреждения высшего образования по учебной дисциплине	Решение, принятое кафедрой, разработавшей учебную программу (с указанием даты и номера протокола) ²
1.			

²При наличии предложений об изменениях в содержании учебной программы учреждения высшего образования по учебной дисциплине.

ДОПОЛНЕНИЯ И ИЗМЕНЕНИЯ К УЧЕБНОЙ ПРОГРАММЕ
на ____/____ учебный год

№№ ПП	Дополнения и изменения	Основание

Учебная программа пересмотрена и одобрена на заседании кафедры
_____ (название кафедры) (протокол № ____ от _____ 201_ г.)

Заведующий кафедрой

_____ (ученая степень, ученое звание) _____ (подпись) _____ (И.О.Фамилия)

УТВЕРЖДАЮ
Декан факультета

_____ (ученая степень, ученое звание) _____ (подпись) _____ (И.О.Фамилия)

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Кнут, Д. Искусство программирования. Том 1. Основные алгоритмы / Д. Кнут. – М.: Вильямс, 2010. – 720 с.
2. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман. – М.: Вильямс, 2010. – 400 с.
3. Сэджвик, Р. Алгоритмы на C++ / Р. Сэджвик. – М.: Вильямс, 2014. – 1056 с.
4. Алгоритмы. Построение и анализ / Т. Кормен, и др. – М.: Вильямс, 2013. – 1328 с.
5. Скиена, С. Алгоритмы. Руководство по разработке / С. Скиена. – СПб.: БХВ-Петербург, 2011. – 720 с.
6. Котов, В. М. Алгоритмы и структуры данных: учеб. пособие / В. М. Котов, Е. П. Соболевская, А. А. Толстиков. – Минск: БГУ, 2011. – 267 с.

Дополнительная литература

1. Кормен, Т. Алгоритмы. Вводный курс / Т. Кормен. – М.: Вильямс, 2015. – 208 с.
2. Кнут, Д. Искусство программирования. Том 2. Получисленные алгоритмы / Д. Кнут. – М.: Вильямс, 2011. – 832 с.
3. Кнут, Д. Искусство программирования. Том 3. Сортировка и поиск / Д. Кнут. – М.: Вильямс, 2012. – 824 с.
4. Shaffer, C. Data Structures and Algorithm Analysis, Third Edition / C. Shaffer. – Dover Publications, 2013. – 624 p.
5. Weiss, M. Data Structures and Algorithm Analysis in C++, Fourth Edition / M. Weiss. – Addison-Wesley, 2014. – 656 p.
6. Googrich, M. Data Structures and Algorithms in C++, Second Edition / M. Googrich, R. Tamassia, D. Mount. – John Wiley & Sons, 2011. – 738 p.
7. Carrano, F. Data Abstraction & Problem Solving with C++. Walls and Mirrors, 6th Edition / F. Carrano, T. Henry. – Pearson, 2013. – 840 p.

СОДЕРЖАНИЕ

1. МЕТОДОЛГИЯ ТЕОРИИ АЛГОРИТМОВ И СТРУКТУР ДАННЫХ	4
ИСТОРИЯ ВОЗНИКНОВЕНИЯ ПОНЯТИЯ «АЛГОРИТМ»	4
ЦЕЛИ И ЗАДАЧИ ТЕОРИИ АЛГОРИТМОВ	5
ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ РЕЗУЛЬТАТОВ ТЕОРИИ АЛГОРИТМОВ	5
ФОРМАЛИЗАЦИЯ ПОНЯТИЯ АЛГОРИТМА	6
ФОРМАЛЬНЫЕ СВОЙСТВА АЛГОРИТМОВ	7
ПОНЯТИЕ СТРУКТУРЫ ДАННЫХ	8
КЛАССИФИКАЦИЯ СТРУКТУР ДАННЫХ	9
ОПЕРАЦИИ НАД СТРУКТУРАМИ ДАННЫХ	10
СТРУКТУРНОСТЬ ДАННЫХ И ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ	10
ПРИНЦИП МОДУЛЬНОГО ПРОГРАММИРОВАНИЯ	12
КОНТРОЛЬНЫЕ ВОПРОСЫ	13
2. АБСТРАКТНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ	15
МАШИНА ПОСТА	15
МАШИНА ТЬЮРИНГА И АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ПРОБЛЕМЫ	16
АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ПРОБЛЕМЫ	18
ПРИЧИНЫ (СВОДИМЫЕ К ПРОБЛЕМЕ ОСТАНОВА) ВЕДУЩИЕ К АЛГОРИТМИЧЕСКОЙ НЕРАЗРЕШИМОСТИ	19
КОНТРОЛЬНЫЕ ВОПРОСЫ	20
3. АНАЛИЗ АЛГОРИТМОВ	20
3.1. СРАВНИТЕЛЬНЫЕ ОЦЕНКИ АЛГОРИТМОВ	20
3.2. КЛАССИФИКАЦИЯ АЛГОРИТМОВ ПО ВИДУ ФУНКЦИИ ТРУДОЁМКОСТИ	21
3.3. АСИМПТОТИЧЕСКИЙ АНАЛИЗ ФУНКЦИЙ	22
3.4. ТРУДОЕМКОСТЬ АЛГОРИТМОВ И ВРЕМЕННЫЕ ОЦЕНКИ	27
3.5. ПРИМЕРЫ АНАЛИЗА ПРОСТЫХ АЛГОРИТМОВ	28
3.6. ПЕРЕХОД К ВРЕМЕННЫМ ОЦЕНКАМ	29
1. Пооперационный анализ	30
2. Метод Гиббсона	30
3. Метод прямого определения среднего времени	32
3.7. ТЕОРЕТИЧЕСКИЙ ПРЕДЕЛ ТРУДОЕМКОСТИ АЛГОРИТМОВ	35
3.8. РЕКУРРЕНТНЫЕ СООТНОШЕНИЯ	38
КОНТРОЛЬНЫЕ ВОПРОСЫ	45
4. ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ	47
4.1. МНОЖЕСТВА	47
ОПЕРАЦИИ ДЛЯ ДИНАМИЧЕСКИХ МНОЖЕСТВ	48
4.2. СТЕКИ И ОЧЕРЕДИ	48
4.3. СПИСКИ	50
4.4. ДЕРЕВЬЯ	54
4.5. КУЧА	55
КОНТРОЛЬНЫЕ ВОПРОСЫ	57
ЛАБОРАТОРНАЯ РАБОТА №1	58
ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ОДНОСВЯЗНЫЙ СПИСОК.	61
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ОДНОСВЯЗНОГО СПИСКА.	63
Графическая иллюстрация	64
СОЗДАНИЕ ОДНОСВЯЗНОГО СПИСКА.	65
ОДНОСВЯЗНЫЙ ЛИНЕЙНЫЙ СПИСОК КАК АБСТРАКТНЫЙ ТИП ДАННЫХ (ADT).	67
Поддерживаемые операции	68
Пример интерфейса класса <i>LinkedList</i>	68
КОНТРОЛЬНЫЕ ВОПРОСЫ	70
ПРАКТИЧЕСКИЕ ЗАДАНИЯ	70
Задание 1	70

ЛАБОРАТОРНАЯ РАБОТА №2	71
ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ	71
ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В СТЭК (МЕТОД <code>PUSH()</code>)	72
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ СТЭКА (МЕТОД <code>POP()</code>)	73
ПРИМЕР ИНТЕРФЕЙСА КЛАССА <code>LINKEDSTACK</code> И СТРУКТУРЫ <code>NODE</code>	74
КОНТРОЛЬНЫЕ ВОПРОСЫ	75
ПРАКТИЧЕСКИЕ ЗАДАНИЯ	75
<i>Задание 1</i>	75
<i>Задание 2</i>	75
<i>Задание 3</i>	76
ЛАБОРАТОРНАЯ РАБОТА №3	76
ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ	77
ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ОЧЕРЕДЬ (МЕТОД <code>ENQUEUE()</code>)	78
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ОЧЕРЕДИ (МЕТОД <code>DEQUEUE()</code>)	79
КОНТРОЛЬНЫЕ ВОПРОСЫ	80
ПРАКТИЧЕСКИЕ ЗАДАНИЯ	80
<i>Задание 1</i>	80
<i>Задание 2</i>	82
ЛАБОРАТОРНАЯ РАБОТА №4	82
ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ	82
ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ В ДВУСВЯЗНЫЙ СПИСОК	83
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ ДВУСВЯЗНОГО СПИСКА	84
КОНТРОЛЬНЫЕ ВОПРОСЫ	85
ПРАКТИЧЕСКИЕ ЗАДАНИЯ	85
<i>Задание 1</i>	85
ЛАБОРАТОРНАЯ РАБОТА №5	89
ГРАФИЧЕСКАЯ ИЛЛЮСТРАЦИЯ БИНАРНЫХ ДЕРЕВЬЕВ	90
СОЗДАНИЕ БИНАРНОГО ДЕРЕВА И ДОБАВЛЕНИЕ ЭЛЕМЕНТОВ	91
УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ БИНАРНОГО ДЕРЕВА	97
ОБХОД БИНАРНОГО ДЕРЕВА(<i>TREE TRAVERSAL</i>)	104
<i>СИММЕТРИЧНЫЙ ОБХОД(inorder traversal)</i>	104
<i>ОБХОД В ШИРИНУ(preorder traversal)</i>	105
<i>ОБХОД В ГЛУБИНУ(postorder traversal)</i>	105
РЕКУРСИВНЫЙ АЛГОРИТМ ОБХОДА БИНАРНОГО ДЕРЕВА	106
ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ	106
ПРИМЕР ИНТЕРФЕЙСА КЛАССОВ <code>BINARYTREE</code> И <code>NODE</code>	107
КОНТРОЛЬНЫЕ ВОПРОСЫ	109
ПРАКТИЧЕСКИЕ ЗАДАНИЯ	109
<i>Задание 1</i>	109
<i>Задание 2</i>	109
<i>Задание 3</i>	110
УЧЕБНАЯ ПРОГРАММА УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ	112
СПИСОК ЛИТЕРАТУРЫ	122