МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
МИРЭА – РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

М.Л. РЫСИН, М.В. САРТАКОВ, М.Б. ТУМАНОВА

ВВЕДЕНИЕ В СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

ЧАСТЬ 2.
ПОИСК В ТЕКСТЕ
НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ
КОДИРОВАНИЕ ИНФОРМАЦИИ
АЛГОРИТМИЧЕСКИЕ СТРАТЕГИИ

УЧЕБНОЕ ПОСОБИЕ

Москва — 2022

УДК 004.43
ББК 32.973.26-018.1я73
Р 95

Рысин М.Л. Введение в структуры и алгоритмы обработки данных. Часть 2. Поиск в тексте. Нелинейные структуры данных. Кодирование информации. Алгоритмические стратегии. [Электронный ресурс]: Учебное пособие / Рысин М.Л., Сартаков М.В., Туманова М.Б. – М.: МИРЭА – Российский технологический университет, 2022. – 109 с. – 1 электрон. опт. диск (CD-ROM).

Учебное пособие посвящено основам разработки эффективных алгоритмов в парадигме структурного программирования. Показаны приёмы разработки и анализа эффективности алгоритмов средствами языка высокого уровня C++ в инструментальной среде Visual Studio. Приводятся необходимые теоретические сведения и практические задания для самостоятельного освоения материала и организации практических занятий.

Материал пособия необходим для освоения дисциплины «Структуры и алгоритмы обработки данных», может быть полезен при изучении других дисциплин, связанных с алгоритмизацией и структурным программированием.

Предназначено для студентов вузов, обучающихся по направлениям подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.03 «Прикладная информатика», 09.03.04 «Программная инженерия», 01.03.04 «Прикладная математика». Пособие может быть полезным для любых иных категорий читателей, интересующихся программированием.

Учебное пособие издается в авторской редакции.

Авторский коллектив:

Рысин Михаил Леонидович, Сартаков Михаил Валерьевич, Туманова Марина Борисовна.

Рецензенты:

Ахмедова Хамида Гаджиалиевна, к.физ.-мат.н., доцент, доцент кафедры ИБМ-6 ФГБОУ ВО «Московский государственный технический университет им. Н.Э. Баумана».

Николаева Светлана Владимировна, д.т.н., профессор, профессор кафедры физики им. В.А. Фабриканта, ФГБОУ ВО «Национальный исследовательский университет "МЭИ"».

Системные требования:
Наличие операционной системы Windows, поддерживаемой производителем.
Наличие свободного места в оперативной памяти не менее 128 Мб.
Наличие свободного места в памяти хранения (на жестком диске) не менее 30 Мб.
Наличие интерфейса ввода информации.
Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).
Подписано к использованию по решению Редакционно-издательского совета
МИРЭА – Российского технологического университета.
Объем __ Мб
Тираж 10

ISBN ____

© Рысин М.Л., Сартаков М.В.,
Туманова М.Б., 2022
© МИРЭА – Российский
технологический университет, 2022

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ 3
ВВЕДЕНИЕ 5
1. ПОИСК В ТЕКСТЕ 6
1.1. Постановка задачи 6
1.2. Линейный (последовательный, прямой) поиск 6
1.3. Алгоритм Кнута-Морриса-Пратта (КМП) 8
1.4. Алгоритм Бойера-Мура (БМ) 12
1.5. Алгоритм Рабина-Карпа (РК) 18
Контрольные вопросы 21
2. ДЕРЕВЬЯ 22
2.1. Понятие структуры данных 22
2.2. Понятие иерархической структуры данных 24
2.3. Бинарное дерево поиска 30
2.4. Алгоритмы обхода бинарного дерева 33
2.5. AVL-дерево 35
2.6. Красно-чёрное дерево 39
2.7. Дерево синтаксического анализа 42
Контрольные вопросы 43
3. ГРАФЫ 44
3.1. Понятие графовой структуры 44
3.2. Представление графа в программе 46
3.3. Обход графа в программе 49
3.4. Транзитивное замыкание в орграфе. Алгоритм Уоршелла 53
3.5. Поиск кратчайшего пути на графе. Алгоритм Дейкстры 56
3.6. Алгоритм Флойда-Уоршелла 60
3.7. Минимальное остовное дерево в графе. Алгоритм Прима 63
3.8. Алгоритм Краскала 66
Контрольные вопросы 68
4. КОДИРОВАНИЕ ИНФОРМАЦИИ 69
4.1. Понятие кодирования данных 69
4.2. Статистическое (энтропийное) кодирование 72
4.3. Арифметический код 75
4.4. Алгоритм Шеннона-Фано 76
4.5. Код Хаффмана 77
4.6. Словарное кодирование. Алгоритм RLE 79
4.7. Алгоритмы семейства LZ 80

Контрольные вопросы 83
5. ПРАКТИЧЕСКИЕ ЗАДАНИЯ 85
5.1. Практическая работа №2.1 85
5.2. Практическая работа №2.2 87
5.3. Практическая работа №2.3 90
5.4. Практическая работа №2.4 92
5.5. Практическая работа №2.5 95
5.6. Практическая работа №2.6 99
5.7. Практическая работа №2.7 103
5.8. Практическая работа №2.8 108
ЛИТЕРАТУРА 111

ВВЕДЕНИЕ

Знание основных структур и алгоритмов обработки данных лежит в основе базовой подготовки специалиста любого ИТ-профиля. Особенно актуально это для тех студентов и выпускников, кто предполагает связать свою трудовую деятельность с разработкой программного обеспечения. Собеседования на подобные вакансии в ИТ-компаниях не обходятся без задач по предметной области алгоритмов и структур данных.

Настоящее учебное пособие позволит освоить этот материал в объёме, достаточном как для успешной сдачи экзамена, так и для работы начинающим (junior) программистом.

Основной материал пособия соответствует содержанию одноимённой дисциплины, преподаваемой студентам ИТ-профиля (второй из двух частей двухсеместрового курса).

Содержание пособия сформировано темами, в каждой имеется необходимая теоретическая часть, вопросы для самоконтроля, листинги с кодом для воспроизведения, изучения и доработки, а также задания для самостоятельной работы. На освоение каждой темы требуются аудиторные занятия (лекции и практические/лабораторные) и часы для самостоятельной работы.

Материал курса, представленный в данном пособии, изложен средствами псевдокода и языка высокого уровня (ЯВУ) C++. Это объясняется и его универсальностью и возможностями по работе с динамической памятью.

Задания предполагают написание текстов программ в популярной инструментальной среде Visual Studio Community, для личного использования она доступна для свободного скачивания на сайте производителя (компании Microsoft). Листинги и задачи можно реализовывать в любой иной среде или редакторе кода, поддерживающем C++, например, Visual Studio Code.

Проверку выполнения заданий рекомендуется проводить в виде очной защиты результата студентом.

Система заданий в настоящем пособии вполне встраивается в балльно-рейтинговую систему контроля успеваемости студентов, используемую в настоящее время во многих вузах России. Так, реализация студентом готовых листингов и умение их объяснить могут быть оценены на «удовлетворительно»; выполнение заданий на доработку (изменение) данного в листингах кода – «хорошо»; выполнение заданий на самостоятельную разработку программ – «отлично».

1. ПОИСК В ТЕКСТЕ

1.1. Постановка задачи

Пусть даны:

• Некоторый текст T (haystack) и **образец** или **шаблон** W (needle) – тоже текст (подстрока). Или формально: Пусть заданы массивы T и W из n и m символов соответственно, причем $0 < m \leq n$.

• Элементы массивов T и W – это символы некоторого конечного алфавита – например: {0, 1}, или {a, …, z}, или {а, …, я}.

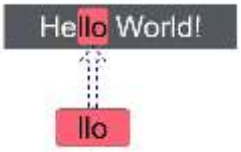


Рисунок 1. Иллюстрация задачи поиска

В результате необходимо:

• Найти первое слева вхождение этого образца в указанный текст, т.е. сообщить о нахождении и, возможно, вернуть индекс, начиная с которого образец присутствует в тексте (рис. 1).

1.2. Линейный (последовательный, прямой) поиск

Линейный поиск часто называют примитивным или наивным алгоритмом (англ. brute force algorithm). Он прост и понятен, но в плане вычислительной сложности он наименее эффективен.

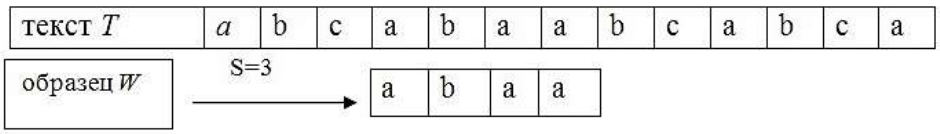


Рисунок 2. Иллюстрация работы алгоритма линейного поиска

Пример (рис. 2):

• Пусть требуется найти образец W = abaa в тексте T = abcabaabcabca

Тогда:

• Со сдвигом S от 0 до успеха или до $|T| - |W|$ посимвольно сравниваются T и W.

Результат:

• Образец входит в текст со сдвигом S=3, индекс i=4.[1]

[1] Здесь и далее сдвиг образца – не физическое перемещение данных в памяти, а лишь приращение значения индексной переменной.

Листинг 1.1. Псевдокод алгоритма прямого поиска

```
justFindStr (char *T, char *W) do
//i – индекс в строке T, j – индекс в подстроке W, lT – длина T, lW – длина W
//f – индекс начала возможного вхождения
  i ← 1, j ← 1, f, lT ← length(T), lW ← length(W);
  if (lT < lW) возврат -1;
  while (i ≤ lT-lW+1) //сдвиг по строке T не дальше, чем lT - lW
  do
    f ← i; //начальный индекс возможного вхождения
    while ((j≤lW) and (T[i]==W[j])) do //сравнение до конца образца
      i ← i+1; j ← j+1;
    od
    if (j = lW+1) возврат f; // удачный поиск
    j ← 1; i ← f+1; //сдвиг++ и возврат к началу подстроки
  od
  возврат -1; // неудачный поиск
od
```

По представленному в лист. 1.1 псевдокоду не составит труда написать программу на любом языке высокого уровня, например, на языке C++.

Временная сложность алгоритма прямого поиска подстроки:

• здесь худший случай – обнаружение образца W→{A….AB} длиной ⎪W⎪=m в конце текста T→{AAA….AAAB} длиной ⎪T⎪=n;

• для обнаружения совпадения в конце строки потребуется произвести порядка n·m сравнений, то есть O(n·m).

Недостатки наивного алгоритма:

1. Как алгоритм «грубой силы», он отличается высокой сложностью: O(n·m) в худшем, O(n-m+1) в лучшем, O(2·n) в среднем случае.

2. После несовпадения просмотр всегда начинается с первого символа W. Если образец читается из внешней памяти, то такие возвраты занимают недопустимо много времени.

3. Информация о тексте T, получаемая при проверке с очередным сдвигом, никак не учитывается при проверке последующих сдвигов.

Улучшение эффективности поиска подстроки:

• Цель улучшений – по возможности при поиске сдвигать образец больше, чем на одну позицию, что, в конечном счёте, потребует меньшего числа посимвольных сравнений.

• Механизм – **препроцессинг** – это предварительная обработка образца, позволяющая учесть частичные совпадения с текстом.

Некоторые реализации улучшенных алгоритмов поиска в тексте: алгоритм Кнута-Морриса-Пратта, алгоритм Бойера-Мура, алгоритм конечного автомата.

1.3. Алгоритм Кнута-Морриса-Пратта (КМП)

Идея алгоритма предложена в 1970-е годы Д. Кнутом и В. Праттом и, независимо от них, Д. Моррисом.

Как уже говорилось выше, в ходе поиска в тексте на очередном сдвиге образца при несовпадении пары символов важно:

• обеспечить ***максимально допустимый*** следующий сдвиг образца;

• гарантированно избежать ***пропуска вхождений*** – величина сдвига должна каждый раз вычисляться заново (рис. 3).

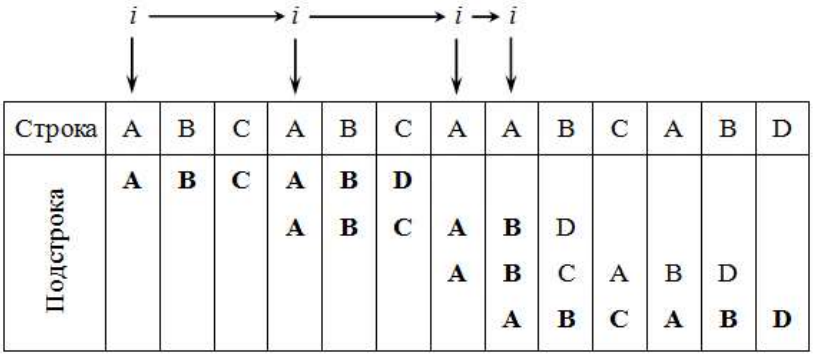


Рисунок 3. Иллюстрация величины сдвига при поиске в тексте

В алгоритме КМП используются понятия префикса и суффикса.

Префикс – это начальные символы последовательности.

Суффикс – окончание последовательности символов.

Собственные префикс и суффикс – не могут быть всей строкой, т.е. собственный префикс – с начального по максимум предпоследнего символа строки, а собственный суффикс – с максимум второго до последнего символа строки (рис. 4).

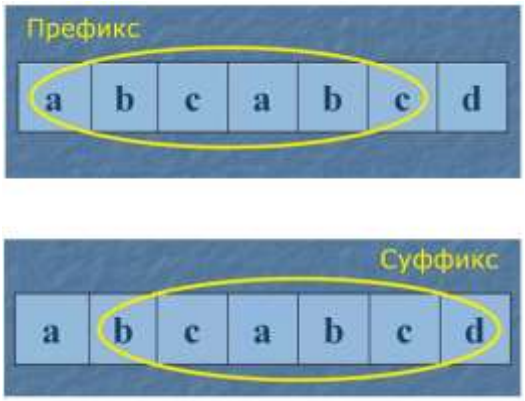


Рисунок 4. Допустимые собственные префикс и суффикс строки

Идея расчёта очередного сдвига:

• образец располагается по левому краю исходного текста (рис. 5);

• организуется посимвольное сравнение текста и образца вплоть до первого несовпадения;

• суффикс образца (подчёркнут красным) без последнего несовпавшего символа (выделено синим) – ***зона частичного совпадения*** образца с текстом – важен при расчёте следующего сдвига;

• если такой суффикс образца совпадает с его префиксом (подчёркнут жёлтым), тогда образец можно сдвинуть вправо на i позиций так, чтобы его префикс расположился под зоной частичного совпадения в основном тексте;

• т.о. сдвиг образца может быть больше, чем на один символ.

Строка	A	B	C	A	B	C	A	A	B	C	A	B	D
Подстрока	A	B	C	A	B	D							
				A	B	C	A	B	D				

Рисунок 5. Очередной сдвиг в алгоритме КМП на основе зоны частичного совпадения образца с текстом

Посимвольное сравнение можно продолжить сразу с точки несовпадения в основном тексте (т.е. без возврата в начало образца).

Индекс последнего совпавшего с текстом символа в образце должен стать ключом к ***длине суффикса***, равного префиксу совпавшей части образца (рис. 6).

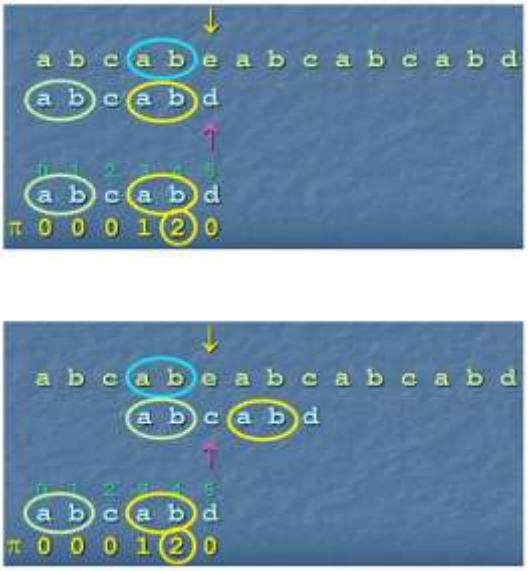


Рисунок 6. Иллюстрация расчёта сдвига в алгоритме КМП

Тогда этап, предваряющий сам поиск (препроцессинг) – создание массива (вектора) длиной, равной длине образца, в котором каждое i-е значение есть длина наибольшего собственного префикса части образца, одновременно являющегося его суффиксом.

Такой вектор (π) называется **префикс-функция**.

В примере на рис. 6 четвёртый индекс последнего совпавшего символа образца ("b") в соответствии с префикс-функцией показывает длину совпадения с

префиксом 2, что позволит сдвинуть образец не на одну, а сразу на 3 позиции вправо.

Пусть дана строка (образец) s[1..m]. Требуется вычислить для неё префикс-функцию, т.е. линейный массив (вектор) чисел π[1..m], где π[i] определяется как наибольшая длина собственного суффикса подстроки s[1..i], совпадающего с её префиксом (рис. 7). Значение π[1] полагается равным 0.

s	a	b	b	a	a	b	b	a	b
π	0	0	0	1	1	2	3	4	2

Рисунок 7. Иллюстрация расчёта префикс-функции

1 этап – Препроцессинг (листинг 1.2) – формирование массива префиксов. Пусть j = π[s, i-1]. Шаги вычисления префикс-функции для i:

1. При s[i] == s[j] определить π[s, i] = j+1.
2. Иначе при j==0 определить π[s, i] = j (т.е. 0)
3. Иначе установить j = π[s, i-1] и перейти к п.1.

Алгоритм требует не более 2·|s| итераций, т.о. для шаблона поиска длиной m сложность составит O(m).

Листинг 1.2. Псевдокод алгоритма префикс-функции

```
Prefix_f(char *s) do
//i – текущий индекс в векторе, j – длина суффикса в предыдущем элементе вектора,
//ls – длина образца s, π[ls] – искомый вектор
  i, j, ls ← length(s), π[ls]; //сначала вектор заполнен нулями
  for i←2 to ls do
    j ← π[i-1];
    while ((j>0) and (s[i]<>s[j])) do //текущий суффикс нельзя расширить
      j ← π[j-1]; //берём меньшие длины префикса и суффикса
    od
    if (s[i] == s[j]) j++; // расширяем длину префикса-суффикса
    π[i] ← j;
  od
  возврат π;
od
```

2 этап – сам КМП-поиск (листинг 1.3).

Листинг 1.2. Псевдокод алгоритма префикс-функции

```
KMPsearch(char *T, char *W, int *π) do
//i – индекс в строке T, j – кол-во совпавш. симв-в = индекс сравниваемого символа в образце,
//lT – длина T, lW – длина W
  i ← 1, j ← 1, lT ← length(T), lW ← length(W);
  for i←1 to lT do
```

```
    while ((j>0) and (T[i]<>W[j])) do //пока несовпадение – сдвигаем образец
      j ← π[j-1]; //берём меньшие длины префикса и суффикса
    od
    if (T[i] == W[j]) j++; // увеличение длины совпавшего фрагмента на +1
    if (j == lW) возврат i-lW;
  od
  возврат -1;
od
```

Шаги основной программы поиска КМП:

Задать текст и образец поиска.

Сформировать префикс-функцию для образца

С помощью префикс-функции осуществить саму процедуру поиска.Есть интересный приём в реализации алгоритма КМП-поиска:

• Склеим образец «aabaa» и основной текст «aabaabaaaabaabaaab» через символ-разделитель, который не может встречаться ни в тексте, ни в образце: «aabaa@aabaabaaaabaabaaab».

• Вызовем для всей склейки префикс-функцию (рис. 8).

a	a	b	a	a	@	a	a	b	a	a	b	a	a	a	a	b	a	a
0	1	0	1	2	0	1	2	3	4	5	3	4	5	2	2	3	4	5

Рисунок 8. Иллюстрация расчёта префикс-функции для склейки текста и образца

• Тогда достаточно последовательным просмотром части этого массива ***после разделителя*** найти позиции со значениями, равными длине образца (в нашем примере 5).

• Это и будут признаки вхождений образца в основной текст.

Важные особенности алгоритма КМП:

• Движение по основному тексту осуществляется только вперёд (без возврата при нахождении несоответствия) – это важное преимущество при работе с данными во внешней памяти.

• Сложность O(n+m) – ***сублинейна*** – даже с учётом префикс-функции (O(n+m) меньше, чем O(n·m) для простого поиска)

• Затраты времени на префикс-функцию окупаются при поиске, если неудаче предшествовало некоторое количество совпадений.

• Однако, вероятность совпадения ниже несовпадения, поэтому на практике выигрыш при использовании КМП-поиска невелик. Пример[2] из рис. 9: получаем 17 смещений вместо 23 в простом поиске.

[2] Пример взят из [2].

H	o	o	l	a	-	H	o	o	l	a		g	i	r	l	s		l	i	k	e		H	o	o	l	i	g	a	n	s	.
H	o	o	l	i	g	a	n																								
				H	o	o	l	i	g	a	n																				
					H	o	o	l	i	g	a	n																			
						H	o	o	l	i	g	a	n																		
										H	o	o	l	i	g	a	n														
											H	o	o	l	i	g	a	n													
												H	o	o	l	i	g	a	n												
													H	o	o	l	i	g	a	n											

Рисунок 9. Пример работы алгоритма поиска КМП

• КМП-поиск – это основа для алгоритма Ахо-Корасика.

1.4. Алгоритм Бойера-Мура (БМ)

Здесь, как и в КМП-поиске, ценой предварительной работы над образцом (препроцессинга) пропускается часть проверок на этапе поиска.

Идея алгоритма БМ-поиска:

1. Прикладывание образца к тексту и его сдвиг ***слева вправо*** до успеха или до достижения конца строки (неуспеха).

2. Сравнение образца ***справа налево***.

3. Сдвиг образца на наибольшее значение из двух, определяемых:

   3.1. Правило сдвига по **стоп-символу** («плохому», несовпавшему).

   3.2. Правило сдвига по **совпавшему** («хорошему», безопасному) **суффиксу** образца.

Рисунок 10. Плохой символ и хороший суффикс

Совместный эффект пунктов 2-3 даст меньше, чем n+m сравнений.

С целью сдвинуть шаблон, по возможности, больше, чем на 1 позицию, необходимо осуществить подготовку – препроцессинг – создание **таблицы смещений** (сдвигов, стоп-символов) для образца.

Сравнение производят с конца образца (справа налево). При несоответствии – сдвиг образца P вправо до совмещения не совпавшего символа в T с последним его вхождением в P по таблице смещений (рис. 11).

После сдвига сравнение начинается заново с правого конца образца.

Таблица смещений (сдвигов, стоп-символов) для образца – вспомогательная структура, создаваемая перед поиском, хранящая информацию о вхождении в него каждого символа. При поиске эта структура позволит быстро определить

величину сдвига образца до ближайшего слева от точки несовпадения в нём стоп-символа.

```
Строка:          * * * * * * к * * * * * *
Шаблон:          к о л о к о л
Следующий шаг:       к о л о к о л

Строка:          * * * * * а л * * * * * * * *
Шаблон:          к о л о к о л
Следующий шаг:               к о л о к о л

Строка:         * * * * к о л * * * * *
Шаблон:           к о л о к о л
Следующий шаг:  к о л о к о л      ?????
```

Рисунок 11. Иллюстрация сдвигов образца в алгоритме БМ

Подходы к реализации таблицы смещений:

• Двумерный массив распределения символов алфавита в образце размером m*|A|, где |A| – длина алфавита – нерационально ни по времени, ни по памяти.

• Линейный массив длиной |A| списков индексов вхождений каждого символа алфавита в образец.

• Таблица стоп-символов длиной |A| – вектор индексов крайне правых вхождений каждого символа алфавита в образец.

В разных реализациях алгоритма БМ возможны вариации таблицы смещений (рис. 12):

• Нумерация может быть с 0 или с 1.

• Индексирование в образце может вестись с его левого края или с правого.

• Признаком отсутствия стоп-символа в образце может быть либо начальный индекс - 1 (будет -1 или 0), либо конечный индекс + 1 (т.е. заведомо несуществующий индекс).

Символ	a	b	d	[остальные]
Позиция	4	3	5	0

« abbad »

Символ	a	b	c	d	[все остальные]
Последняя позиция	4	1	6	5	-1

abcdadcd

Рисунок 12. Примеры таблиц смещений (стоп-символов)

Но обязательно не учитывается последнее (не единственное) вхождение символа, приходящееся на крайне правую позицию в образце – тогда учитывается предпоследнее.

Правило сдвига по стоп-символу на этапе поиска рассмотрим на примере 1 (рис. 13).

Пусть есть алфавит A (|A|=5): a, b, c, d, e.

И необходимо найти вхождение образца "abbad" в строку "abeccacbadbabbad".

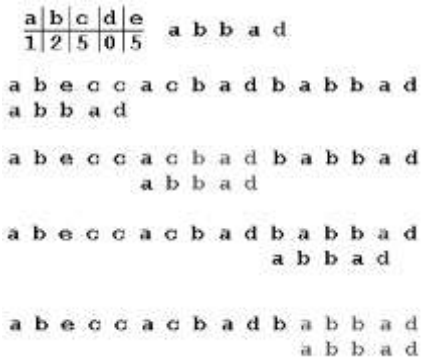


Рисунок 13. Иллюстрация работы алгоритма БМ (пример 1)

В результате препроцессинга создаётся таблица стоп-символов (в примерах нумерация с 0 справа).

Результат: 3 смещения образца до успеха.

Пример 2 (рис. 14):

• Таблица стоп-символов: H – 7; o – 5; l – 4; i – 3; g – 2; a – 1; n – 0

Рисунок 14. Иллюстрация работы алгоритма БМ (пример 2)

• Результат: 4 смещения (вместо 17 и 23).

Пример реализации кода на языке C++ для создания таблицы стоп-символов приведён в листинге 1.3.

Листинг 1.3. Код для создания таблицы стоп-символов

```
//Построение таблицы стоп-символов для шаблона s
vector <size_t> badCharHeuristic(string s) {
    size_t size = s.length();
    vector <size_t> badchar(NO_OF_CHARS); // таблица стоп-символов
    // Инициализация таблицы стоп-символов
    // -1 - признак отсутствия символа в шаблоне
    for (size_t i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;
    // Заполнение индексами крайних справа вхождений в шаблон,
    for (size_t i = 0; i < size - 1; i++) // исключая последний символ
        badchar[(int)s[i]] = i;
    return badchar; //возврат таблицы стоп-символов
}
```

Анализ правила сдвига по стоп-символу:

• Наибольший эффект при несовпадениях, близких к правому концу шаблона поиска.

• Дополнительная память O(|A|) и время на подготовку шаблона O(|A|), где |A| – длина алфавита.

• Нет эффекта, если стоп-символ в шаблоне справа от точки несовпадения – сдвиг не может быть отрицательным, т.е. влево (см. нижнее изображение на рис. 11). Возможные решения:

• Расширенное правило стоп-символа (см. Д.Гасфилд).

• Правило хорошего суффикса.

Рассмотрим правило (эвристику) сдвига по **совпавшему суффиксу**.

Рисунок 15. Иллюстрация правила сдвига по совпавшему суффиксу

Пусть P приложена к T и подстрока t из T совпадает с суффиксом P, но следующий слева символ (x) уже не совпадает (y).

Тогда найдём (если существует) t' – крайнюю правую копию t в P, для которой z ≠ y. И сдвинем P вправо, чтобы t' совпало с t в T (рис. 16).

```
Строка:          * * * * * * р к а * * * * *
Шаблон:          с к а л к а л к а
Следующий шаг:                с к а л к а л к а
```

Рисунок 16. Пример сдвига по совпавшему суффиксу до совпадения t' с t

При отсутствии t' в P образец смещается на сдвиг, чтобы префикс образца P совпал с суффиксом подстроки t в T (рис. 17). Если такого сдвига нет, то на |P|.

При отсутствии t образец сдвигается на 1 позицию.

```
Строка:          * * т о к о л * * * * *
Шаблон:          к о л о к о л
Следующий шаг:           к о л о к о л
```

Рисунок 17. Пример сдвига по совпавшему суффиксу при отсутствии t'

Условие y ≠ z необходимо, без него алгоритм становится неприемлемо медленным – O(n·m).

В примере на рис. 16 правило хорошего суффикса смещает образец на 6, а правило стоп-символа сместило бы только на 3.

Также как и в правиле плохого символа здесь необходим препроцессинг – создание **таблицы суффиксов**, т.е. вектора размером с длину шаблона |P|=m.

Таблица суффиксов строится на основе следующей идеи: для каждого возможного t в образце P, совпадающего с собственным суффиксом P в таблицу записывается наименьший размер сдвига, по которому t' в P (если он существует) совпадёт с вхождением t в T.

Суффикс	[пустой]	d	cd	dcd	...	abcdadcd
Сдвиг	1	2	4	8	...	8
Иллюстрация						
было	?	?d	?cd	?dcd	...	abcdadcd
стало	abcdadcd	abcdadcd	abcdadcd	abcdadcd	...	abcdadcd

Рисунок 18. Пример сдвигов по таблице суффиксов для шаблона "abcdadcd"

На рис. 18 и 19 представлены примеры работы правила хорошего суффикса для шаблонов "abcdadcd" и "колокол" (в последнем его начало совпадает с окончанием).

Суффикс	[пустой]	л	ол	...	олокол	колокол
Сдвиг	1	4	4	...	4	4
Иллюстрация						
было	?	?л	?ол	...	?олокол	колокол
стало	колокол	колокол	колокол	...	колокол	колокол

Рисунок 19. Пример сдвигов по таблице суффиксов для шаблона "колокол"

Пример реализации кода на языке C++ для создания таблицы суффиксов с помощью **z-функции** приведён в листинге 1.4.

Листинг 1.4. Код для создания таблицы суффиксов

```
45  //Построение таблицы суффиксов на основе Z-функции
46  vector <size_t> goodSuffHeuristic(string s) {
47      size_t m = s.length();
48      vector<size_t> suffshift(m + 1, m);
49      //Вычисление Z-функции
50      vector<size_t> z(m, 0);
51      for (int j = 1, maxZidx = 0, maxZ = 0; j < m; ++j) {
52          if (j <= maxZ) z[j] = Min(maxZ - j + 1, z[j - maxZidx]);
53          while (j + z[j] < m && s[m - 1 - z[j]] == s[m - 1 - (j + z[j])]) z[j]++;
54          if (j + z[j] - 1 > maxZ) {
55              maxZidx = j;
56              maxZ = j + z[j] - 1;
57          }
58      }
59      //Формирование таблицы суффиксов
60      for (int j = m - 1; j > 0; j--) suffshift[m - z[j]] = j; //цикл №1
61      for (int j = 1, r = 0; j <= m - 1; j++) //цикл №2
62          if (j + z[j] == m)
63              for (; r <= j; r++)
64                  if (suffshift[r] == m) suffshift[r] = j;
65      return suffshift;
66  }
```

Реализация процедуры БМ-поиска на основе ранее созданной таблицы суффиксов представлена в листинге 1.5.

Листинг 1.5. Код основной программы БМ-поиска

```
68  //БМ-поиск
69  void BM_string_serch(string t, string s, vector<size_t> badChar, vector <size_t> suffshift) {
70      for (int shift = 0; shift <= t.length() - s.length();) {
71          int pos = s.length() - 1;
72          while (s[pos] == t[pos + shift]) {
73              //успех
74              if (pos == 0) {
75                  cout << "Индекс подстроки (с нуля): " << shift << "\n"; return;
76              }
77              --pos;
78          }
79          // Неудача - выбор величины сдвига из двух возможных
80          shift += Max(suffshift[pos + 1], pos - badChar[t[pos + shift]]);
81      }
82      cout << "Подстроки нет!" << "\n"; return;
83  }
```

Тогда в основной программе необходимо предусмотреть ввод основного текста и шаблона поиска, для которого, в свою очередь, вызвать функцию построения таблицы суффиксов и запустить сам поиск.

Анализ алгоритма поиска БМ:

• Использование только правила («плохого») стоп-символа даст наихудшее время O(n·m), в среднем случае (в т.ч. в текстах на естественных языках) время сублинейно – O(n+m).

• Использование только правила совпавшего («хорошего») суффикса даст наихудшее время:

  • без модификаций – сублинейное O(n+m);

  • с модификациями – линейное O(m), по Р. Коулу – 3·m сравнений.

• В целом, использование полного алгоритма БМ в наихудшем случае даст:

  • без модификаций – сублинейное время O(n+m);

  • с модификациями – линейное O(m), например, до 2·m сравнений в алгоритме Апостолико-Джанкарло.

• Ёмкостная сложность: дополнительная память оценивается как O(|A|+m).

• Эффективен для естественных языков, снижение замечено только на специально подобранных примерах.

• На больших алфавитах (ASCII) растёт ёмкостная сложность БМ-поиска.

• На малых алфавитах (ДНК) ухудшается время БМ-поиска.

Существует несколько модификаций алгоритма БМ-поиска, например **алгоритм Бойера-Мура-Хорспула**. Это упрощённый вариант алгоритма БМ с одной изменённой эвристикой стоп-символов[3].

На случайных текстах (т.е. в среднем случае) этот вариант показывает время лучше, чем «классический» БМ. При этом алгоритм проще в реализации.

[3] Не составит труда найти в литературе описание этого алгоритма.

Использует стандартную функцию сравнения участков памяти, оптимизированную на ассемблерном уровне под конкретный процессор.

Алгоритм показывает себя стабильнее на больших алфавитах.

Другие модификации БМ-поиска:

• Алгоритм «турбо-Бойера-Мура» (БМ-поиск с турбо-сдвигом).

• Алгоритм Апостолико-Джанкарло – аналог БМ-поиска с 2·m сравнениями в худшем случае.

• Алгоритм Чжу-Такаоки.

• Алгоритм Райты (Райта).

1.5. Алгоритм Рабина-Карпа (РК)

Рассмотрим подробнее другой подход в реализации поиска в тексте – **алгоритм Рабина-Карпа**, который был предложен, соответственно, М. Рабином и Р. Карпом в 1987 году.

Это алгоритм поиска подстроки с использованием ***хеширования***.

В худшем случае он показывает сложность O(n·m), т.е. как у линейного поиска, но в лучшем и среднем – до O(n).

Главная особенность РК-поиска – способность в среднем за время O(n) найти любой из k образцов одинаковой длины независимо от величины k.

Поэтому алгоритм используется для поиска не одиночного шаблона, а ***множественных шаблонов*** одинаковой длины. Пример применения – поиск плагиата.

Ранее рассмотренные алгоритмы настроены на оптимизацию ***величины сдвига*** образца. А РК-алгоритм пытается ускорить процесс ***сравнения*** текста с образцом.

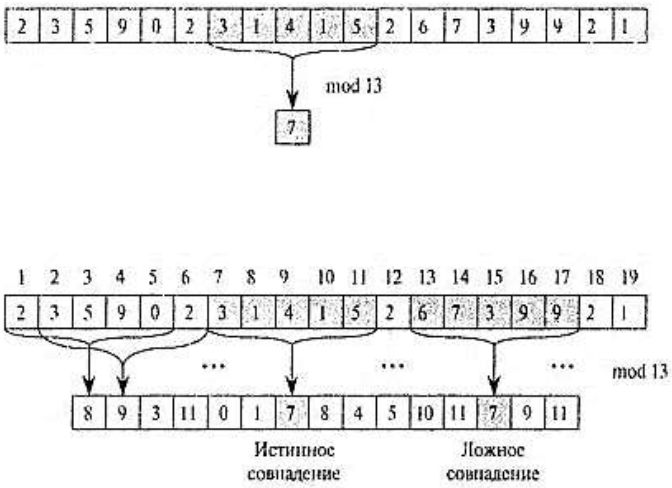


Рисунок 20. Иллюстрация работы алгоритма РК-поиска

Идея РК-алгоритма (рис. 20):

• Специализированная ***хеш-функция*** преобразует символьную последовательность (строку) в числовое значение – хеш-значение (хеш).

• Далее в хеш шаблона поиска сравнивается с хешами частей основного текста.

• Неравенство хешей гарантирует отсутствие шаблона в этой части текста, значит, можно посимвольно не сравнивать.

• Равенство хешей указывает на возможность присутствия (ложные совпадения), следовательно, необходимо сравнить соответствующую часть текста и шаблон посимвольно.

Псевдокод алгоритма РК-поиска в двух вариантах (для поиска одиночного образца и множества образцов) представлен, соответственно, в листингах 1.6 и 1.7.

Листинг 1.6. Псевдокод РК-поиска одиночного образца

```
function RabinKarp(string s[1..n], string sub[1..m])
    hsub := hash(sub[1..m])
    hs := hash(s[1..m])
    for i from 1 to (n-m+1)
        if hs = hsub
            if s[i..i+m-1] = sub
                return i
        hs := hash(s[i+1..i+m])
    return not found
```

Листинг 1.7. Псевдокод РК-поиска множества образцов

```
function RabinKarpSet(string s[1..n], set of string subs, m) {
    set hsubs := ∅
    for each sub in subs
        hsubs := hsubs ∪ {hash(sub[1..m])}
    hs := hash(s[1..m])
    for i from 1 to (n-m+1)
        if hs ∈ hsubs
            if s[i..i+m-1] = string из subs с хешем hs
                return i
        hs := hash(s[i+1..i+m])
    return не найдено
}
```

Проблема в работе РК-поиска – это ложное совпадение хешей. Между двумя строками (частью текста и образцом) может произойти коллизия – совпадение их хешей.

Тогда необходимо посимвольно проверять совпадение самих строк много времени, если строки имеют большую длину.

Проверка не нужна, если ложные срабатывания допустимы, тогда в среднем время поиска O(n).

В противном случае важно выбрать хорошую хеш-функции. Как говорится,

«алгоритм хорош настолько, насколько хороша его хеш-функция».

При использовании достаточно хороших хеш-функций ***коллизии*** случаются крайне редко (в среднем до O(n)).

Низкую вероятность коллизий и эффективное вычисление сочетает в себе ***полиномиальный хеш***:

$$H = c_1 \times b^{m-1} + c_2 \times b^{m-2} + c_3 \times b^{m-3} ... + c_m \times b^0$$

c = символы в строке, m = длина строки, b = константа

Символы здесь представлены своими числовыми кодами.

Во избежание переполнения целочисленной разрядной сетки в памяти при больших H:

$$H = (c_1 \times b^{m-1} + c_2 \times b^{m-2} ... + c_m \times b^0) \; mod \; Q$$

Выбор b и Q:

• Метод авторов алгоритма: фиксированное b=2, Q – простое случайное число из диапазона [2..n^3], тогда вероятность коллизии не превосходит 1/n.

• Модификация (Дитзфелбингер и др.): фиксированное простое Q (например, простые числа Мерсенна, 2^{31}-1, 2^{61}-1, 2^{32}-5, 2^{64}-59), b – случайное число из диапазона [0..Q-1], тогда при Q>n^c для какого-то c>2 вероятность коллизии не превосходит $1/n^{c-2}$.

Проблема вторая – многократный пересчёт хеша подстрок в тексте при сдвиге образца.

При наивном пересчёте хеша сложность РК-поиска сравняется с алгоритмом линейного поиска в худшем случае – O(n·m).

Приём решения – ***кольцевой (скользящий) хеш***, когда при сдвиге на одну позицию в тексте новый хеш вычисляется на основе старого, убрав из подсчёта первый символ предыдущей подстроки и добавив в подсчёт новый символ справа.

Например, в случае полиномиального хеша:

$$H = ((H_p - C_p \times b^{m-1}) \times b + C_n \,) \; mod \; Q$$

Hp = предыдущий хеш, Cp = предыдущий символ, Cn – новый символ, m = размер подстроки, b = константа

Контрольные вопросы

1. Что такое строка, её префикс и суффикс?
2. Объясните идею алгоритма последовательного (наивного) поиска шаблона в строке.
3. Какая асимптотическая сложность наивного поиска подстроки в строке?
4. Назовите два основных направления оптимизации наивного алгоритма.
5. В чём идея поиска образца алгоритмом Бойера–Мура?
6. Назовите асимптотическую сложность алгоритма Бойера–Мура поиска подстроки в строке по времени и памяти.
7. Как влияет размер таблицы кодов в алгоритме Бойера-Мура на скорость поиска?
8. За счет чего в алгоритме Бойера-Мура поиск оптимален в большинстве случаев?
9. Объясните идею алгоритма Бойера-Мура-Хорспула.
10. Объясните влияние префикс-функции в алгоритме Кнута-Морриса-Пратта (КМП) на поиск подстроки в строке. Приведите пример префикс-функции.
11. Приведите пример строки, для которой поиск подстроки "aaabaaa" более эффективен методом КМП, чем методом Бойера-Мура (и наоборот).
12. В чём идея поиска образца алгоритмом Рабина-Карпа?
13. Приведите асимптотическую сложность алгоритма Рабина-Карпа поиска подстроки в строке.
14. Объясните идею алгоритма Ахо–Корасика. Приведите его вычислительную и ёмкостную сложность.

2. ДЕРЕВЬЯ

2.1. Понятие структуры данных

Структура данных (data structure) – это именованная совокупность логически связанных значений в памяти ЭВМ в соответствии с определённым ***макетом***.

Макет задаёт:

• логические (причинно-следственные) связи между элементами;

• способ доступа к значениям (прямой или последовательный).

Один и тот же макет может быть эффективен для одних задач и неэффективен для других, т.е. нет универсальной структуры.

Изображение структур часто происходит в виде графа (рис. 21).

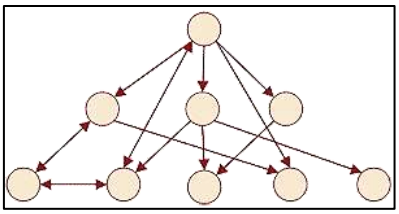


Рисунок 21. Иллюстрация структуры данных в виде графа

Граф – это эффективное средство моделирования в реальных задачах.

Виды структур данных по характеру связи:

• линейные – элементы образуют последовательность, цепочку, список – обход узлов линеен (пример – список игроков команды).

• нелинейные – совокупность элементов без позиционного упорядочения (пример – структура доменов в пространстве имён DNS).

Логическая (абстрактная) структура данных, изображаемая графом, располагается в линейной цепочке адресуемых ячеек ОЗУ[4].

Структура хранения данных – это представление логической структуры данных в памяти ЭВМ (машинный образ абстрактной структуры): статические (определены в коде на этапе компиляции) и динамические (создаются на этапе исполнения кода).

Одну и ту же логическую структуру данных можно представить разными структурами хранения (матрица, вектор линейных списков, нелинейные списки).

Отражение структуры данных в структуру хранения обеспечивается средствами того или иного языка высокого уровня и компилятора.

Структуры хранения данных:

- ***линейные***: векторные (вектор, массив) и списочные (связанные линейные

[4] Адрес ячейки памяти – это номер байта, начиная с которого ячейка размещена в памяти.

списки): одно- и двунаправленный списки, стек, очередь, дек;

- ***нелинейные***: графовые (деревья и лес, граф, сеть) и теоретико-множественные (реляционные).

Нелинейные структуры позволяют выражать более сложные отношения между элементами – классами и объектами предметной области (рис. 22).

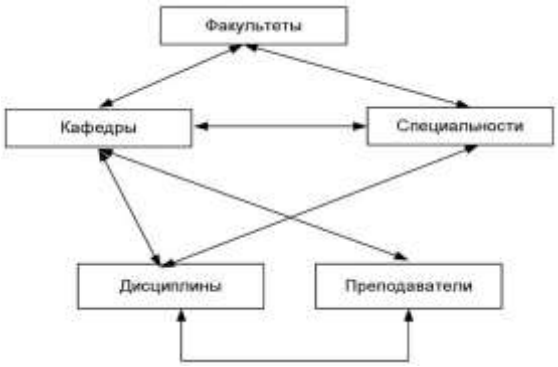


Рисунок 22. Пример нелинейной структуры образовательной организации

Множество предметных областей описываются нелинейно. Примеры:

• оглавление книги;

• файловая система;

• пространство имён DNS;

• логистические схемы (сеть автодорог);

• компьютерная сеть (её топология);

• организационная структура предприятия;

• топология микросхем;

• химические структуры.

Способы представления нелинейных структур в памяти ЭВМ: встроенные типы данных и пользовательские типы (классы, списки на основе указателей).

Базовые операции над структурами данных:

• найти нужный элемент в структуре;

• получить доступ к полям элемента в структуре;

• вставить (добавить) элемент в структуру;

• удалить элемент из структуры.

2.2. Понятие иерархической структуры данных

Дерево (иерархия) – это связный граф без циклов (ациклический), в котором потомок может иметь не более одного предка (рис. 23).

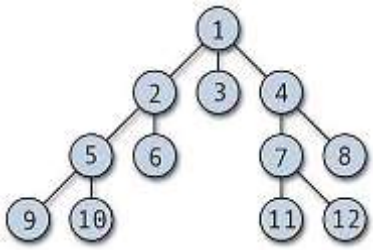


Рисунок 23. Пример древовидной (иерархической) структуры

Следствия из определения:

• число рёбер всегда на одно меньше числа вершин;

• между любой парой вершин всегда есть только один путь.

Лес – это множество деревьев (домены Windows NT в лесу).

Иерархическая структура – основная терминология:

• **вершины** (узлы, сегменты, записи) – это элементы данных [+ ссылка(-и)];

• **рёбра** – отражают связь «предок-потомок»;

• корневой узел – не имеет предка

• уровни;

• братья (близнецы) – узлы, имеющие одного непосредственного предка;

• «дядя-племянник» и «дед-внук»;

• **лист** – это узел без потомков;

• высота дерева – это количество рёбер между корнем и максимальным уровнем;

• степень узла – количество непосредственных потомков;

• степень дерева – максимальная степень его узлов;

• путь в дереве – последовательность узлов от корня до узла;

• помеченное дерево – образовано узлами с метками;

• длина пути до узла (номер уровня) – количество рёбер от корня до узла;

• длина пути в дереве – сумма длин всех его рёбер;

• сбалансированное дерево – когда высота поддеревьев отличается не более, чем на 1;

• упорядоченное дерево – если существует определённый порядок расположения узлов («левый сын» → «правый брат»);

• неупорядоченное дерево – если нет определённого порядка расположения узлов.

Преимущества иерархической организации данных:

• простота восприятия человеком;

• высокое быстродействие при транзакционной обработке.

Недостатки:

• медленный доступ к данным нижних уровней;

• склонность к избыточности (если узел должен иметь >1 предка → дублирование дерева, т.е. лес);

• на больших объёмах данных требуется индексация элементов.

Дерево типа Т:

Пусть существуют деревья T_1, T_2, …, T_k с корнями n_1, n_2, … n_k (рис. 24).

Тогда можно построить дерево Т с корнем n_0, в котором n_1, n_2, … n_k – его поддеревья.

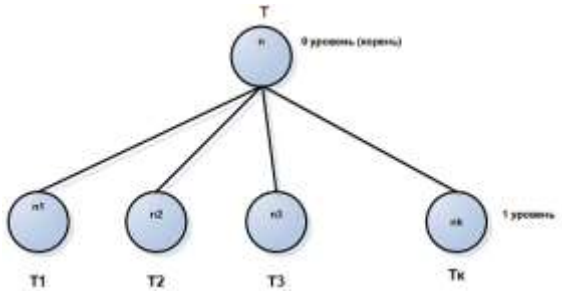


Рисунок 24. К определению дерева типа Т

Рекурсивное определение дерева типа Т:

• пустое или

• состоит из одного узла n_0 типа Т

• состоит из узла n_0 типа Т, с которым связано конечное число поддеревьев базового типа Т (т.е. T_1, T_2, …, T_k с корнями n_1, n_2, … n_k).

Бинарное (двоичное) дерево: k=2 – т.е. каждый узел имеет не более двух потомков (рис. 25).

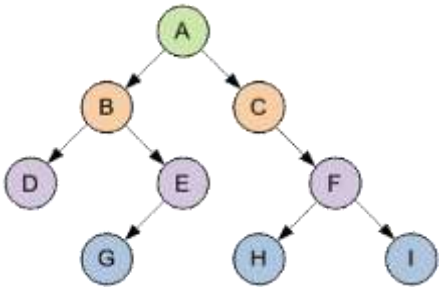


Рисунок 25. Бинарное дерево

Для N-арного дерева k=N.

Для реализации деревьев в памяти ЭВМ можно использовать:

• структуры: массив, линейный список;

• иерархии на основе указателей.

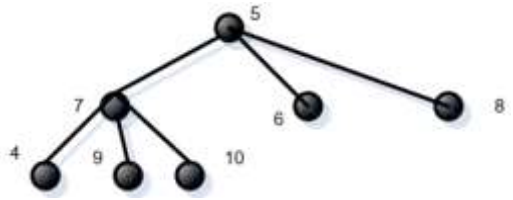
Способы:

1. Массив родителей.
2. Линейный список.
3. Массивы «левых потомков» и «правых братьев»:
   A. На курсоре.
   B. На таблице.
4. Иерархическая структура с указателями.

Рассмотрим эти способы более подробно.

1. Массив родителей.

Пусть дано ***помеченное*** дерево, в котором метка – неотрицательное целое число (рис. 26).



1	2	3	4	5	6	7	8	9	10	*Метки узлов*
-1	-1	-1	7	0	5	5	5	7	7	*Ссылки на родителей (их метки)*

Рисунок 26. Бинарное дерево и его массив родителей

Тогда массив родителей этого дерева строится по принципу, что в позицию с индексом, равным метке узла дерева заносится метка его родителя (в примере -1 – есть признак отсутствия узла с такой меткой). Такой массив будет хранить информацию, достаточную для обхода всего дерева.

В листингах 2.1 и 2.2 предложены два варианта реализации этого способа средствами языка C++:

• вариант 1 – отдельные массивы для связей и данных;

• вариант 2 – всё в одной структуре.

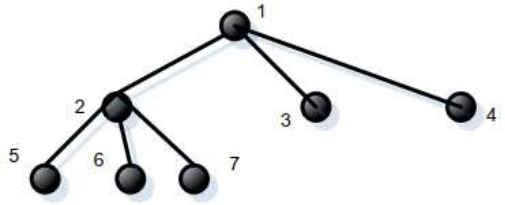
Листинг 2.1. Код реализации дерева в памяти массивом родителей с отдельными массивами для связей и данных

```
12    typedef int TDataNode;
13
14    const int MaxLen = 100;
15    using Tree = unsigned int[MaxLen];
16    using infoNodeTree = TDataNode[MaxLen];
17    Tree TreeParents; //массив родителей
18    infoNodeTree InfoTree; //данные
19    //...
```

Пример использования 1 – поиск «левого сына» i-го узла (рис. 27). Для решения этой задачи нужна последовательная нумерация узлов в порядке возрастания.

Листинг 2.2. Код реализации бинарного дерева в памяти массивом родителей с размещением всего в одной структуре

```
12      typedef int TDataNode;
13
14      const int MaxLen = 100; //макс. количество узлов
15      struct TTree {
16          unsigned int Tree [MaxLen]; //массив родителей
17          TDataNode infoNodeTree [MaxLen];//массив значений
18          byte n; //количество узлов
19      };
20      TTree T;
21      //...
```



метки узлов:	1	2	3	4	5	6	7
метки родителей:	0	1	1	1	2	2	2

Рисунок 27. Иллюстрация к примеру использования 1

Тогда «левый сын» узла 2 – это первый элемент массива со значением 2. Реализация поиска «левого сына» i-го узла приведена в листингах 2.3 и 2.4.

Листинг 2.3. Массив родителей для организации дерева в памяти для примера использования 1

```
12      typedef int TDataNode;
13
14      const int MaxLen = 100; //макс. количество узлов
15      struct TTree {
16          unsigned int Tree [MaxLen]; //массив родителей
17          TDataNode infoNodeTree [MaxLen];//массив значений
18          byte n; //количество узлов
19      };
20      TTree T;
21      //...
```

Листинг 2.4. Код реализации поиска «левого сына» i-го узла в дереве (пример использования 1)

```
18  //функция возвращает номер элемента массива
19  //первый узел со значением i
20  int leftSon(TTree T, int i) {
21      int j = 1;  //текущий индекс
22      while (j <= T.n && T.Tree[j] != i) {
23          j++;
24          if (j <= T.n) return j;
25      }
26      return -1;
27  }
```

Пример использования 2 – поиск «правого брата» i-го узла. Шаги решения:

А. Найти родителя заданного узла.

Б. Найти следующий элемент.

Реализация примера использования 2 на C++ приведена в листинге 2.5.

Листинг 2.5. Код реализации поиска «правого брата» i-го узла в бинарном дереве (пример использования 2)

```
29  //Поиск родителя заданного узла
30  int Parent(TTree T, int i) {
31      return T.Tree[i];
32  }
33  //Поиск правого брата
34  int rightBrother(TTree T, int i) {
35      int j, p;
36      p = Parent(T, i); //Метка родителя заданного узла
37      //Поиск правого брата узла, следующего за исходным:
38      if (T.Tree[i + 1] == p) return (i + 1);
39      else return -1;
40  }
```

Продолжим рассмотрение других способов организации дерева в памяти.

2. Дерево линейными списками потомков.

Пусть требуется представить в памяти дерево (рис. 28).

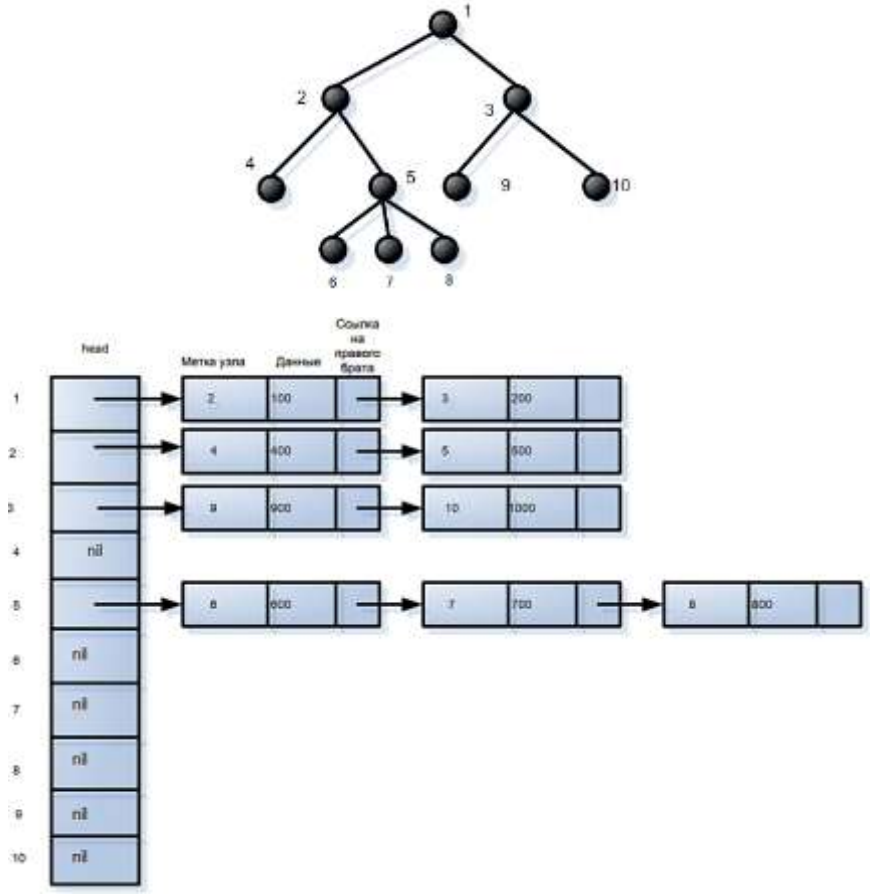


Рисунок 28. Пример дерева и его реализация в памяти списками потомков

Здесь head – массив родителей, содержит ссылки на списки сыновей; индексы – это метки узлов; nil – признак отсутствия потомков. Первый узел в списке потомков – это «левый сын».

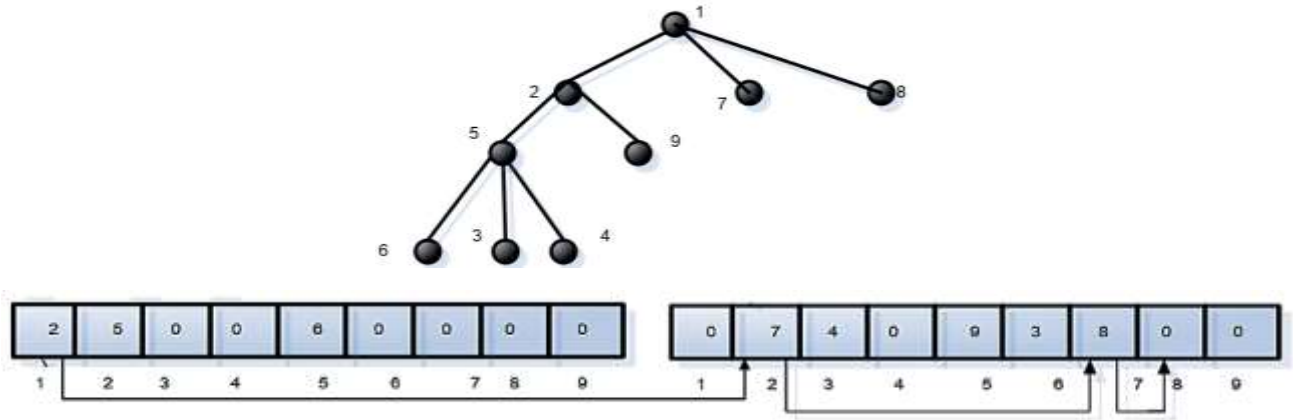
Реализация этого способа средствами языка C++ представлена в листинге 2.6.

Листинг 2.6. Код реализации дерева линейными списками потомков

```
42  //описание узла
43  struct Tnode {
44      int info;
45      Tnode *next;
46  };
47  //массив указателей на потомков
48  struct Tree {
49      Tnode *L[MaxLen];
50      int n;
51  };
52  Tree T;
```

3. Массивы «левых потомков» и «правых братьев».

А. Реализация на ***курсоре*** – структуре на основе массива индексов (ссылок) на другие элементы этого или другого массива (рис. 29).



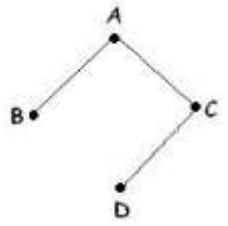
а) родители с указателями на своих «правых братьев»

б) «левые потомки» с указателями на своих «правых братьев»

Рисунок 29. Пример дерева и его реализация в памяти массивами «левых потомков» и «правых братьев» на курсоре

Б. Реализация дерева на ***таблице*** «левых потомков» и «правых братьев» (рис. 30).

Код на языке C++ для реализации этого способа представлен в листинге 2.7.



Индекс элемента	Метка	Индекс левого сына	Индекс правого брата
1	D	0	0
2	B	0	4
3	A	2	0
4	C	1	0

Рисунок 30. Пример дерева и его реализация в памяти массивами «левых потомков» и «правых братьев» на таблице

Листинг 2.7. Код реализации дерева на таблице «левых потомков» и «правых братьев»

```
12  //узел
13  struct TNode{
14      int Left; //левый сын
15      char Nod; //метка
16      int Right; //правый брат
17  };
18  //дерево на таблице
19  struct Tree {
20          TNode tabl[MaxLen]; //таблица
21          int info[MaxLen]; //данные
22          int Root; //корень
23      };
```

Выстраивание иерархии в памяти с помощью указателей рассмотрим в последующих пунктах.

2.3. Бинарное дерево поиска

Рекурсивное определение **двоичного (бинарного) дерева** (рис. 25):

• пустое или

• состоит из одного узла

• состоит из узла, с которым связано не более 2-х бинарных поддеревьев.

Полное бинарное дерево – на всех уровнях, кроме последнего, строго 2 потомка. **Сбалансированное дерево** – высота поддеревьев отличается не более, чем на 1.

В составе узла бинарного дерева: данные, ссылка на левое поддерево, ссылка на правое поддерево.

Двоичное дерево поиска – это бинарное дерево с дополнительными условиями (рис. 31):

Листинг 2.8. Определение узла бинарного дерева

```
8   //Узел дерева:
9   struct node
10  {
11      int info;      //Информационное поле
12      node *l, *r;   //Левая и Правая часть дерева
13  };
```

• оба поддерева – двоичные деревья поиска;

• у всех узлов левого поддерева любого узла X значения ключей данных (числовых меток) не больше значения ключа самого узла X;

• у всех узлов правого поддерева – соответственно, не меньше значения ключа узла X.

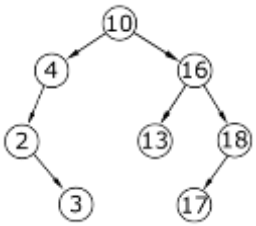


Рисунок 31. Пример двоичного дерева поиска

Примечание: для ключей должна быть определена операция "<".

Операции над двоичным деревом поиска:

• определение;

• поиск узла с заданным значением ключа;

• доступ к полям узла;

• вставка нового узла;

• удаление узла из дерева: листа, с одним правым поддеревом, с одним левым поддеревом, с двумя поддеревьями.

В листингах 2.9 и 2.10 предложены два варианта реализации бинарного дерева поиска средствами языка C++:

• вариант 1: с отдельным полем для ключа;

• вариант 1: ключ – это само значение.

Листинг 2.9. Определение узла бинарного дерева поиска (с отдельным полем для ключа)

```
8   //Узел дерева:
9   struct node
10  {
11      int info;      //Информационное поле
12      int key;       //ключ данных
13      node *l, *r;   //Левая и Правая часть дерева
14  };
```

Листинг 2.10. Определение узла бинарного дерева поиска (ключ – само значение)

```
8   //Узел дерева:
9   struct node
10  {
11      int info;      //Информационное поле
12      node *l, *r;   //Левая и Правая часть дерева
13  };
14
15  node *T = NULL;    //указатель на дерево
```

В следующих листингах 2.11 и 2.12 представлена реализация операций вставки нового узла и вывода на экран содержимого бинарного дерева поиска средствами языка C++. В листинге 2.13 – пример кода основной программы.

Листинг 2.11. Вставка нового узла в бинарное дерево поиска

```
17  //Запись элемента в бинарное дерево
18  void push (int a, node **t) {
19      //Если дерева нет, то формируем корень:
20      if ((*t) == NULL) {
21          (*t) = new node;                //Выделяем память
22          (*t)->info = a;                 //Вносим значение узла
23          (*t)->l = (*t)->r = NULL;       //Очищаем память для следующего роста
24          return;
25      }
26      //Дерево есть:
27      if (a > (*t)->info)
28          push(a, &(*t)->r); //Если а больше текущего элемента, кладем его вправо
29      else
30          push(a, &(*t)->l); //Иначе - влево
31  }
```

Листинг 2.12. Вывод содержимого бинарного дерева поиска

```
33  //Вывод на экран
34  void print(node *t, int u)
35  {
36      if (t == NULL) return;                  //Если дерево пустое - выходим
37      else {
38          print(t->l, ++u);                   //Рекурсивно идём в левое поддерево
39          for (int i = 0; i<u; ++i) cout << "|";
40          cout << t->info << endl;            //Показываем элемент
41          u--;
42      }
43      print(t->r, ++u);                       //Рекурсивно идём в правое поддерево
44  }
```

Пример работы кода из листинга 2.12 представлен на рис. 32.

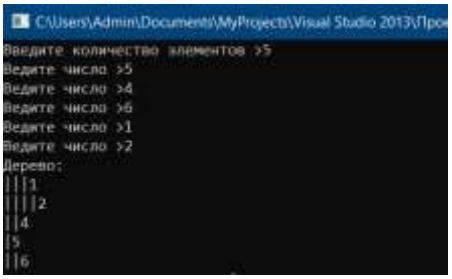


Рисунок 32. Пример вывода содержимого двоичного дерева поиска в консоль

Листинг 2.13. Пример кода основной программы для работы с бинарным деревом поиска

```
46 int main() {
47     SetConsoleCP(1251);
48     SetConsoleOutputCP(1251);
49
50     int n;                              //Количество узлов в дереве
51     int s;                              //Число, передаваемое в дерево
52     cout << "Введите количество элементов >"; cin >> n;
53     for (int i = 0; i < n; ++i) {
54         cout << "Ведите число >";
55         cin >> s;                       //Считываем элемент за элементом
56         push(s, &T);                 //И каждый сохраняем в дерево
57     }
58     cout << "Дерево:\n";
59     print(T, 0);
60     //cin.ignore().get();
61
62     system("PAUSE");
63     return 0;
64 }
```

2.4. Алгоритмы обхода бинарного дерева

Обход дерева – это алгоритм, обеспечивающий доступ к каждому узлу дерева для выполнения операций с данными узла.

Виды обходов дерева:

• в глубину – по поддеревьям (ветвям);

• в ширину – по уровням (потомкам).

Обходы в глубину (depth-first search, DFS): прямой, симметричный (центрированный), обратный.

Алгоритм **прямого обхода** (NLR) непустого бинарного дерева (рис. 33):

A. Посещается корень (выводится поле данных).

B. Прямой обход левого поддерева (рекурсивно)

C. Прямой обход правого поддерева (рекурсивно).

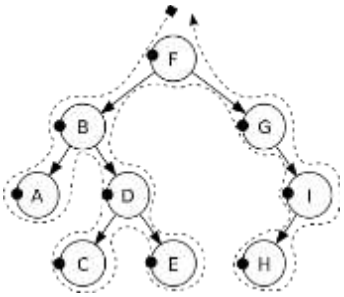


Рисунок 33. Иллюстрация прямого обхода бинарного дерева

Результат прямого обхода: F, B, A, D, C, E, G, I, H.

Алгоритм **симметричного обхода** (LNR) непустого бинарного дерева (рис. 34):

A. Обход левого поддерева рекурсивно алгоритмом симметричного обхода.

B. Посещается текущий узел (или корень).

C. Обход правого поддерева (рекурсивно) алгоритмом симметричного обхода.

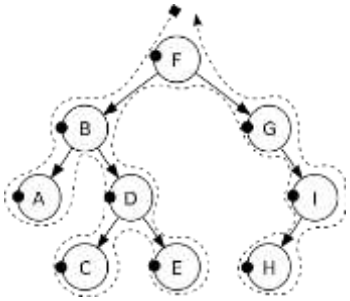


Рисунок 34. Иллюстрация симметричного обхода бинарного дерева

Результат симметричного обхода: A, B, C, D, E, F, G, H, I.

В двоичном дереве поиска центрированный обход извлекает данные в ***отсортированном порядке***.

Алгоритм **обратного обхода** (LRN) непустого бинарного дерева (рис. 35):

A. Обход левого поддерева (рекурсивно) алгоритмом обратного обхода.

B. Обход правого поддерева рекурсивно алгоритмом обратного обхода.

C. Посещается текущий узел (или корень).

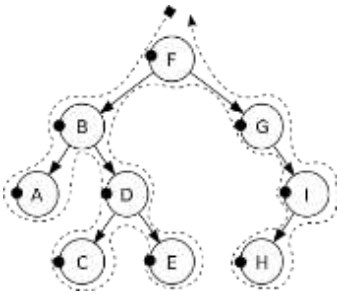


Рисунок 35. Иллюстрация обратного обхода бинарного дерева

Результат обратного обхода: A, C, E, D, B, H, I, G, F.

Алгоритм обхода **в ширину** (breadth-first search, BFS): посещаем каждый узел на уровне прежде, чем перейти на следующий уровень.

Результат обхода в ширину: F, B, G, A, D, I, C, E, H (рис. 36).

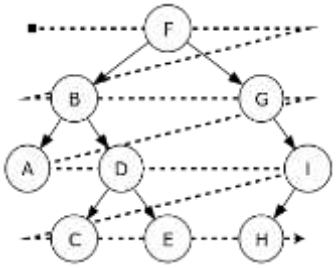


Рисунок 36. Иллюстрация обхода в ширину

В реализации обхода в ширину необходимы:

• абстрактный тип данных – дерево, в котором определён узел с полями: метка, ссылки;

• корневой узел;

• функции:

  • создание дерева;

  • поиск «левого потомка»;

  • поиск «правого брата»;

  • поиск родителя;

  • переход в корень;

  • проверка не пусто ли дерево.

На практике возможна сортировка массива с использованием двоичного дерева поиска. Шаги:

• Построение двоичного дерева поиска по ключам исходного массива (например, чтением из потока – консоли, файла).

• Сборка результирующего массива путём обхода узлов построенного дерева ключей в симметричном (центрированном) порядке следования ключей.

Здесь средняя алгоритмическая сложность построения дерева – $O(n \cdot \log(n))$. В случае разбалансированного дерева сложность может достигать $O(n^2)$.

При этом следует помнить, что древовидная структура в памяти требует не менее 4n служебных значений (ссылки – на элемент исходного массива, на родителя, на поддеревья).

2.5. AVL-дерево

Напомним, что у ***сбалансированного дерева*** высота поддеревьев отличается не более, чем на 1.

AVL-дерево – это сбалансированное двоичное дерево поиска (Г.М. Адельсон-Вельский и Е.М. Ландис, 1962 г.).

Высота AVL-дерева логарифмически зависит от числа узлов: от $\log2(n + 1)$ до $1.44 \cdot \log2(n + 2) - 0.328$.

В свою очередь, операции над бинарным деревом поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, т.о. получаем гарантированно логарифмическую зависимость времени работы этих алгоритмов от числа ключей в дереве.

Количество возможных высот на практике сильно ограничено (при 32-битной адресации – 44, что означает $n=10^9$ узлов, 10+ GB памяти).

Минимальное количество узлов n для каждой высоты h можно подсчитать

рекуррентной формулой Фибоначчи:

$$n_0 = 0, n_1 = 1, n_0 = 0, n_h = n_{h-2} + n_{h-1} + 1.$$

Для каждой вершины в AVL-дереве определено дополнительное поле – показатель ***balance factor*** – разность между высотами левого и правого поддерева с допустимыми значениями -1, 0 и 1 (рис. 37).

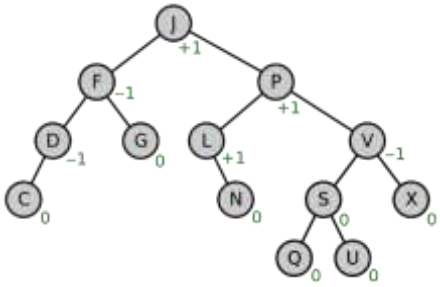


Рисунок 37. Значения balance factor в AVL-дереве

Если при добавлении/удалении узла balance factor становится -2 или 2, то требуется **балансировка** – изменение связи предок-потомок в поддереве данного узла, так, что разница становится не больше 1.

4 метода балансировки:

1. Малый поворот (вращение)

• Левый (рис. 38.а): когда h_b-h_L=2 и h_c<=h_R.

• Правый (рис. 38.б): когда h_b-h_R=2 и h_c<=h_L.

а) левый поворот *б) правый поворот*

Рисунок 38. Малые повороты в AVL-дереве

2. Большой поворот (вращение)

• Левый (рис. 39.а): когда h_b-h_L=2 и h_c>h_R.

• Правый (рис. 39.б): когда h_b-h_R=2 и h_c>h_L.

а) левый поворот *б) правый поворот*

Рисунок 39. Большие повороты в AVL-дереве

Пример определения структуры узла AVL-дерева средствами языка C++ представлен в листинге 2.14.

Листинг 2.14. Пример определения структуры узла AVL-дерева

```
struct node // структура для представления узлов дерева
{
        int key;
        unsigned char height;
        node* left;
        node* right;
        node(int k) { key = k; left = right = 0; height = 1; }
};
```

Здесь key – это ключ, height – высота поддерева с корнем в данном узле, поля left и right – указатели на левое и правое поддеревья. Конструктор создаёт новый узел.

Вспомогательные функции для AVL-дерева:

• возврат высоты;

• вычисление balance factor заданного узла;

• восстановление корректного значения поля height заданного узла.

Пример реализации этих функций на языке C++ представлен в листинге 2.15.

Листинг 2.15. Вспомогательные функции для AVL-дерева

```
//возврат высоты:

unsigned char height(node* p)
{
        return p?p->height:0;
}

// вычисление balance factor заданного узла:

int bfactor(node* p)
{
        return height(p->right)-height(p->left);
}

// восстановление корректного значения поля height заданного узла:

void fixheight(node* p)
{
        unsigned char hl = height(p->left);
        unsigned char hr = height(p->right);
        p->height = (hl>hr?hl:hr)+1;
}
```

Пример реализации левого и правого малых поворотов на языке C++ представлен в листинге 2.16 (см. рис. 40).

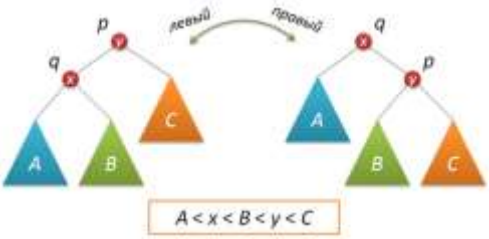


Рисунок 40. Иллюстрация к листингу 2.16

Листинг 2.16. Малые повороты в AVL-дереве

```
node* rotateright(node* p) // правый поворот вокруг p
{
        node* q = p->left;
        p->left = q->right;
        q->right = p;
        fixheight(p);
        fixheight(q);
        return q;
}

 node* rotateleft(node* q) // левый поворот вокруг q
 {
         node* p = q->right;
         q->right = p->left;
         p->left = q;
         fixheight(q);
         fixheight(p);
         return p;
 }
```

Алгоритм добавления вершины в AVL-дерево:

• Проход по пути поиска, пока не убедимся, что ключа в дереве нет.

• Включение новой вершины в дерево и определение результирующих показателей балансировки.

• Возврат по пути поиска и проверка в каждой вершине показателя сбалансированности (если необходимо — балансировка).

Включение вершины в левое поддерево может привести к:

a) $h_l < h_r \rightarrow h_l = h_r$, ничего делать не нужно.

b) $h_l = h_r \rightarrow h_l = h_r+1$, т.е. левое поддерево будет больше, но балансировка не требуется.

c) $h_l > h_r \rightarrow h_l = h_r+2$ – требуется балансировка.

В третьей ситуации требуется определить балансировку левого поддерева: если левое поддерево этой вершины выше правого, то требуется большое правое вращение, иначе хватит малого правого.

Аналогичные (симметричные) рассуждения можно привести и для включения в правое поддерево.

Алгоритм удаления вершины AVL-дерева (рекурсивный):

• Находим узел для удаления p с заданным ключом k.

• В правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

• Если у узла p нет правого поддерева:

  • по свойству AVL-дерева слева у этого узла может быть только один потомок, либо узел p – лист;

  • в обоих случаях удаляют p и в качестве результата возвращают указатель на левого потомка узла p.

• Если правое поддерево у p есть, то находим минимальный ключ в этом поддереве (по свойству бинарного дерева поиска этот ключ находится в конце левой ветки от корня.

2.6. Красно-чёрное дерево

Красно-чёрное дерево (КЧД) также относится к сбалансированным бинарным деревьям поиска (самобалансирующимся).

Свойства КЧД, обеспечивающие его сбалансированность (рис. 41):

1. В структуре узла такого дерева появляется новое поле – бит цвета (красный/чёрный).

2. Корень (как правило) и все листья – чёрные.

3. Листья не содержат данные (это признаки окончания дерева).

4. Каждый красный узел должен иметь 2 чёрных потомка.

5. Пути от узла к его листьям должны содержать одинаковое количество чёрных узлов (**чёрная высота**).

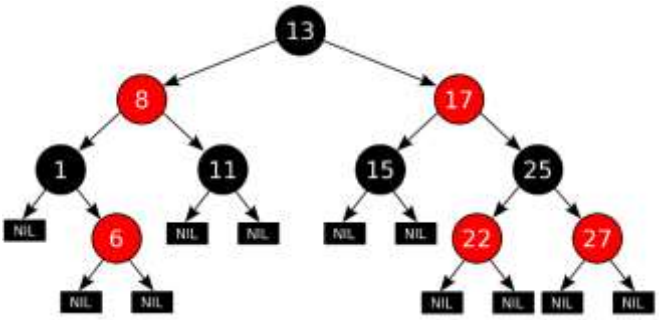


Рисунок 41. Пример красно-чёрного дерева

Высота КЧД из N узлов, лежит в диапазоне от log2(N+1) до 2·log2 (N+1).

Путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего и дерево примерно сбалансировано.

Операции вставки, удаления и поиска требуют (в худшем случае) время,

пропорциональное высоте дерева, т.е. КЧД более эффективны, чем обычные бинарные деревья поиска.

Алгоритм ***вставки*** узла в КЧД:

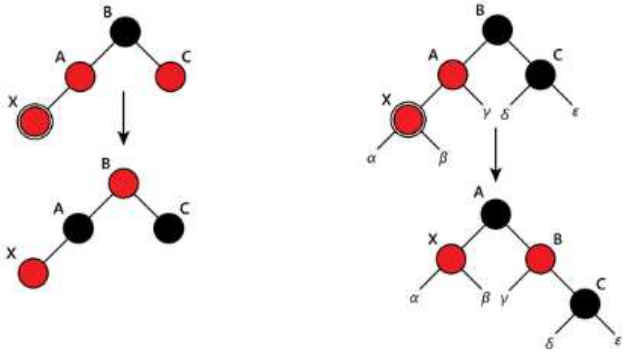
• Новый узел добавляется на место одного из листьев, окрашивается красным, прикрепляется 2 листа.

• Далее идет процедура проверки сохранения свойств КЧД (свойства 4 и 5).

• В случае нарушения («отец» - красный) происходит балансировка дерева:

• если «дядя» тоже красный, то инвертируем их обоих и «деда» (рис. 42.а);

• если «дядя» черный – инвертируем «отца» с «дедом» с правым поворотом (рис. 42.б).



а) *б)*

Рисунок 42. Иллюстрация вставки нового узла X в КЧД

При ***удалении*** узла из КЧД возможны следующие случаи:

1. У вершины нет потомков – изменяем указатель на неё у родителя на nil.

2. Есть один потомок, то делаем у родителя ссылку на него вместо удаляемой вершины.

3. Есть оба потомка:

1. Находим вершину со следующим значением ключа.

2. Делаем у родителя ссылку на него вместо удаляемой вершины.

Проверка балансировки при удалении узла из КЧД:

• Удаление красной вершины свойства КЧД не нарушает.

• Восстановление балансировки требуется только при удалении чёрной вершины.

• Если «брат» текущей вершины x (потомка удалённого узла) ***красный*** (рис. 43.а):

• Поворот вокруг ребра между отцом и братом.

• Красим брата в чёрный, а отца – в красный.

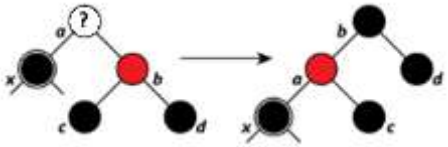
• Если «брат» текущей вершины x (потомка удалённого узла) ***чёрный*** и:

А) Оба потомка у «брата» чёрные (рис. 43.б), то красим брата в красный цвет и делаем отца чёрным.

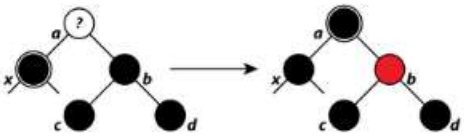
Б) У «брата» правый потомок чёрный, а левый красный (рис. 43.в):

• Перекрашиваем «брата» и его «левого сына».

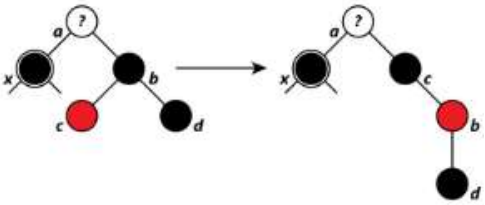
• Делаем вращение.



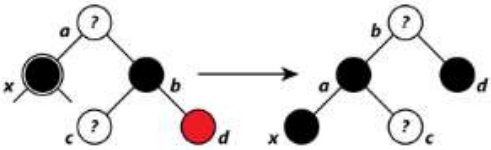
а)



б)



в)



г)

Рисунок 43. Балансировка КЧД при удалении узла

• В) У «брата» правый потомок красный (рис. 43.г):

• Перекрашиваем «брата» в цвет «отца».

• Перекрашиваем его ребёнка и отца в чёрный.

• Делаем вращение.

• Продолжаем тот же алгоритм, пока текущая вершина чёрная и не дошли до корня дерева.

При удалении выполняется не более трёх вращений.

2.7. Дерево синтаксического анализа

Парсер (синтаксический анализатор) – это программа или её часть, выполняющая **синтаксический анализ** – процесс сопоставления линейной последовательности ***лексем языка*** (естественного или формального) с его ***формальной грамматикой***.

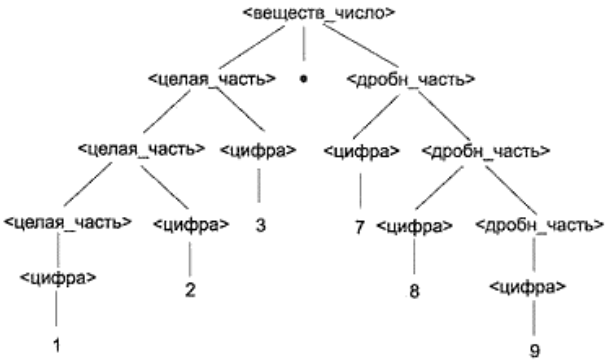


Рисунок 44. Дерево синтаксического анализа вещественного числа 123,789

Исходный текст преобразуется в древовидную структуру, удобную для последующей обработки – дерево разбора (рис. 44).

Рассмотрим это на примере лексемы «вещественное число»:

<веществ_число> ::= <целая_часть> . <дробн_часть>
<целая_часть> ::= <цифра> | <целая_часть> <цифра>
<дробн_часть> ::= <цифра> | <цифра> <дробн_часть>
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8| 9

Дерево разбора (синтаксического анализа), соответствующее числу 123.789, представлено на рис. 44.

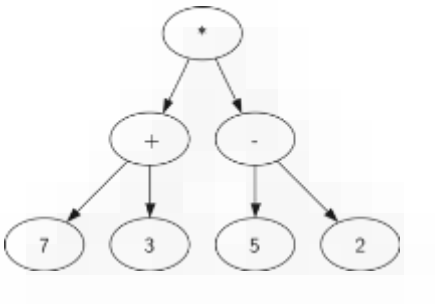


Рисунок 45. Дерево синтаксического анализа арифметического выражения

Области применения:

• математические выражения (рис. 45);

• регулярные выражения (могут использоваться для автоматизации лексического анализа);

• формальные грамматики;

• естественные языки (машинный перевод и другие генераторы текстов).

• языки программирования – разбор исходного кода в процессе трансляции;

• структурированные данные (XML, HTML, CSS, JSON, ini-файлы и т. п.);

• построение индекса в поисковой системе;

• SQL-запросы (DSL-язык);

• извлечение данных веб-страниц – веб-скрейпинг (частным случай парсинга).

Контрольные вопросы

1. Что такое нелинейная структура данных? Приведите примеры.
2. Что такое структура хранения данных?
3. Перечислите базовые операции над структурами данных.
4. Что такое дерево? Что такое лист?
5. Как вычислить высоту дерева?
6. Что называется сбалансированным деревом?
7. Дайте рекурсивное определение двоичного дерева как дерева типа T.
8. Перечислите способы реализации деревьев в памяти.
9. Что такое бинарное дерево поиска?
10. Какие существуют способы обхода дерева?
11. В чём особенность симметричного обхода бинарного дерева поиска?
12. Дайте определение AVL-дереву. Почему поиск в нём эффективнее?
13. Может ли balance factor в AVL-дереве превысить значение 2?
14. Перечислите способы балансировки AVL-дерева.
15. Что такое красно-чёрное дерево?

3. ГРАФЫ

3.1. Понятие графовой структуры

Как уже было сказано выше, нелинейные структуры позволяют выражать более сложные отношения между элементами предметной области.

Виды нелинейных структур:

• теоретико-графовые: деревья и лес, граф, сеть;

• теоретико-множественные: реляционные

Граф моделирует отношения (связи) между вершинами в реальных задачах (рис. 46).

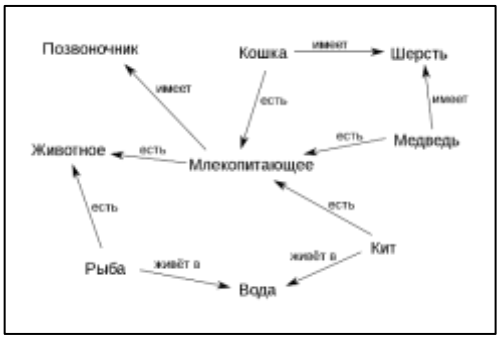


Рисунок 46. Пример нелинейного характера связей в предметной области

Множество предметных областей описываются нелинейно. Примеры уже были перечислены на стр. 23.

Граф G [R,A] – это совокупность двух множеств – вершин R и рёбер (в неориентированном) или дуг (в ориентированном) A.

Граф **конечный** – если множества R и A конечны; **взвешенный** – если рёбрам присваиваются числовые веса (отражают расстояние, время перехода, затраты и пр.).

Дерево – это частный случай графа (связный ациклический).

Ориентированный (направленный) граф (**орграф**):

• Пусть вершины $p_i, p_j \in [R]$ – концевые точки ребра a.

• Тогда если порядок обхода концевых точек существенен (есть начальная вершина и конечная), то ребро a ориентированное (дуга).

• Иначе ребро a неориентированное.

• Граф ориентированный (направленный), если ориентированы все его рёбра.

• Т.о. граф ориентирован, когда всем рёбрам присвоено направление.

Важные понятия:

• ***Помеченный*** граф, если вершины имеют метки (числовые или текстовые).

• Две вершины ***смежные***, если они соединены одним ребром.

• Соседи – это вершины, смежные с некоторой вершиной.

• **Путь** – последовательность вершин от одной вершины к другой. Между двумя вершинами может быть более одного пути.

• ***Связный*** граф – между любой парой вершин есть хотя бы один путь (маршрут).

Важные задачи теории графов:

• Семь мостов Кёнигсберга (1736, Л. Эйлер).

• Задача коммивояжёра – поиск самого выгодного маршрута, проходящего через указанные вершины.

• Проблема четырёх красок (1976, К. Аппель, В. Хакен).

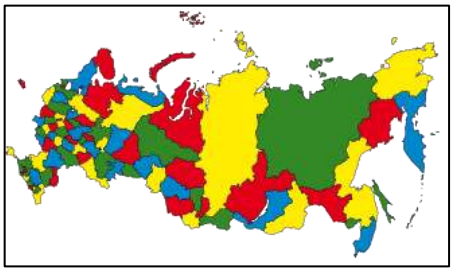


Рисунок 47. Иллюстрация задачи четырёх красок

• Задача о клике (клика – полный подграф) – существует ли в графе клика размера k; найти в заданном графе клику максимального размера.

• Нахождение минимального стягивающего (остовного) дерева в связном взвешенном неориентированном графе (рис. 48).

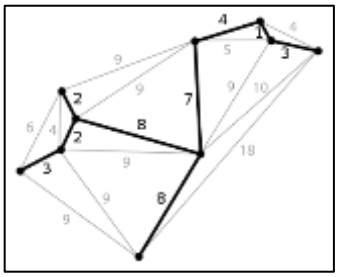


Рисунок 48. Иллюстрация задачи поиска минимального остовного дерева

• Изоморфизм графов – можно ли путём перенумерации вершин одного графа получить другой.

• Планарность графа – можно ли изобразить граф на плоскости без пересечений рёбер (или с минимальным числом слоёв).

Сеть (транспортная сеть, flow network) – это ориентированный граф G = (V,E), в котором каждое ребро (u,v) ∈ E имеет неотрицательную ***пропускную способность*** c(u,v) ≥ 0 и ***поток*** f (u,v) (рис. 49).

Если (u,v) ∉ E, то c(u,v) = 0.

Выделяются две вершины – **исток** (источник) s и **сток** t такие, что любая другая вершина сети лежит на пути из s в t.

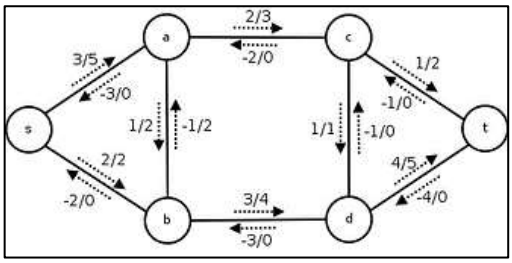


Рисунок 49. Пример транспортной сети

Сети используют для моделирования, например, трафика.

Потоком (англ. flow) f в сети G является действительная функция f:V×V→R, удовлетворяющая условиям:

1) f(u,v) = −f(v,u) (антисимметричность или кососимметричность);

2) |f(u,v)| ⩽ c(u,v) (ограничение пропускной способности), если ребра нет, то f(u,v)=0;

3) $\sum_v f(u,v) = 0$ для всех вершин u, кроме истока s и стока t (закон сохранения потока).

Величина потока – это сумма потоков из истока, определяется как |f|=$\sum_{v \in V} f(s,v)$, она равна сумме всех потоков в сток.

3.2. Представление графа в программе

Вершины в графе – экземпляры сущностей, описываемых набором полей с данными (аналогично деревьям), следовательно в программе для них выбирают тип – структура или класс.

Рёбра у узла в графе – их может быть >2, т.е. имеем нежёсткую структуру.

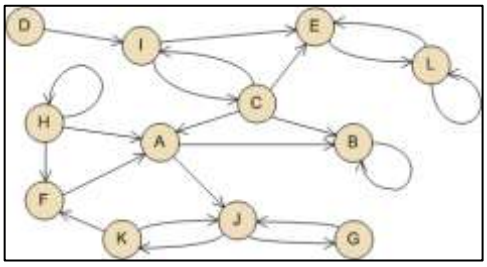


Рисунок 50. Пример невзвешенного орграфа

Способы представления графа в программе:

1.Матрица смежности.

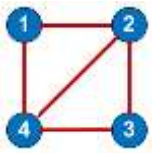
2.Матрица инцидентности.

3.Список смежности.

4.Список рёбер.

Далее разберём эти способы подробнее.

1. **Матрица смежности** – это двумерный массив (квадратная матрица) для представления неориентированных графов (рис. 51). Здесь 0 в ячейке (false) – это отсутствие ребра.



	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	1
4	1	1	1	0

Рисунок 51. Пример невзвешенного неориентированного графа и его матрица смежности

Нули на главной диагонали, если нет ***петель***.

Треугольная часть матрицы над диагональю – есть зеркальное отражение части под ней, что означает избыточность – при добавлении ребра необходимо изменить 2 элемента матрицы.

Недостаток матрицы смежности как способа представления графа в памяти – требования к памяти (n^2).

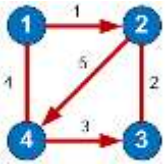
Матрицу смежности лучше использовать для плотных графов.

2. **Матрица инцидентности** (инциденции).

Инцидентность – отношение между парой вершин и ребром.

Инциденция – тройка вида (a, u, b), указывающая, какую пару (a, b) элементов множества вершин графа соединяет тот или иной элемент u множества рёбер.

Количество строк в матрице равно числу вершин, количество столбцов – числу рёбер (рис. 52).



	1	2	3	4	5
1	1	0	0	1	0
2	-1	1	0	0	1
3	0	1	-1	0	0
4	0	0	1	1	-1

Рисунок 52. Пример взвешенного орграфа и его матрица инцидентности

В ориентированном графе если ребро выходит из инцидентной вершины, то 1, если входит в вершину, то -1, отсутствует – 0.

Способ требователен к памяти, используется редко (для нахождения циклов)

3. **Список смежности** – это набор списков, i-й из которых содержит номера вершин, в которые идут рёбра из вершины i (рис. 53).

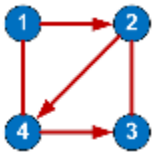
Не является таблицей, это «список списков».

Преимущества:

• Наиболее удобный способ для представления разреженных графов, при реализации алгоритмов обхода графа.

• Рациональное использование памяти O(|V|+|E|).

• Позволяет проверять наличие ребра и удалять его.



1	2, 4
2	3, 4
3	2
4	1, 3

Рисунок 53. Пример орграфа и его список смежности

Недостатки:

• Низкая скорость в плотных графах.

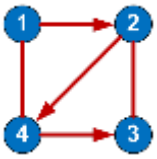
• Сложно проверить, существует ли ребро между двумя вершинами.

• Количество вершин графа должно быть известно заранее.

• Для взвешенных графов нужно хранить список элементов с двумя полями, что усложняет код.

4. **Список рёбер** (рис. 54).

В каждой строке записываются две смежные вершины, инцидентные ребру (для взвешенного графа и вес ребра).



	Начало	Конец
1	1	2
2	1	4
3	2	3
4	2	4
5	3	2
6	4	1
7	4	3

Рисунок 54. Пример орграфа и его список рёбер

Количество строк в списке ребер равно результату сложения ориентированных рёбер с удвоенным количеством неориентированных рёбер.

Размер занимаемой памяти: O(|E|). Это наиболее компактный способ представления графов, часто применяется для внешнего хранения или обмена данными.

3.3. Обход графа в программе

Обход – это процесс систематического просмотра всех рёбер или вершин графа с целью отыскания рёбер или вершин, удовлетворяющих некоторому условию.

Способы обхода: в ширину и в глубину.

Алгоритмы обхода лежат в основе решения различных задач обработки графов – построения остовного дерева, проверки связности, ацикличности, вычисления расстояний между вершинами и др.

1. **Поиск в ширину** (breadth-first search, BFS) подразумевает поуровневое исследование графа (рис. 55):

• сначала посещается узел-источник – произвольно выбранный узел,

• затем – все его потомки,

• после этого потомки потомков и т.д.

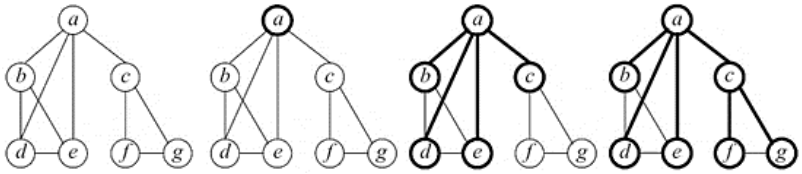


Рисунок 55. Иллюстрация способа обхода графа в ширину

Вершины просматриваются в порядке возрастания их расстояния от корня.

Алгоритм прекращает свою работу после обхода всех вершин графа, либо в случае выполнения требуемого условия (нахождения целевого узла).

Особенности алгоритма поиска в ширину:

• При обнаружении заданной вершины (целевого узла) построенный путь является кратчайшим.

• Для реализации алгоритма удобно использовать очередь.

• Сложность поиска при списочном (нематричном) представлении графа O(n+m), т.к. рассматриваются все n вершин и m ребер.

Использование матрицы смежности приводит к оценке $O(n^2)$.

Применения алгоритма поиска в ширину:

- Поиск кратчайшего пути в невзвешенном графе (ориентированном или неориентированном).
- Поиск компонент связности.
- Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов.
- Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин.
- Найти все вершины, лежащие на каком-либо кратчайшем пути

между заданной парой вершин.

Реализация обхода графа в ширину на языке C++ (с использованием очереди STL) представлена в листинге 3.1. Граф в примере задан матрицей смежности (рис. 56).

Листинг 3.1. Обход графа в ширину

```
1   #include <iostream>
2   #include <queue> // очередь
3   using namespace std;
4   int main()
5   {
6     queue<int> Queue;
7     int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 }, // матрица смежности
8     { 1, 0, 1, 1, 0, 0, 0 },
9     { 1, 1, 0, 0, 0, 0, 0 },
10    { 0, 1, 0, 0, 1, 0, 0 },
11    { 0, 0, 0, 1, 0, 1, 0 },
12    { 0, 0, 0, 0, 1, 0, 1 },
13    { 1, 0, 0, 0, 0, 1, 0 } };
14    int nodes[7]; // вершины графа
15    for (int i = 0; i < 7; i++)
16      nodes[i] = 0; // исходно все вершины равны 0
17    Queue.push(0); // помещаем в очередь первую вершину
18    while (!Queue.empty())
19    { // пока очередь не пуста
20      int node = Queue.front(); // извлекаем вершину
21      Queue.pop();
22      nodes[node] = 2; // отмечаем ее как посещенную
23      for (int j = 0; j < 7; j++)
24      { // проверяем для нее все смежные вершины
25        if (mas[node][j] == 1 && nodes[j] == 0)
26        { // если вершина смежная и не обнаружена
27          Queue.push(j); // добавляем ее в очередь
28          nodes[j] = 1; // отмечаем вершину как обнаруженную
29        }
30      }
31      cout << node + 1 << endl; // выводим номер вершины
32    }
33    cin.get();
34    return 0;
35  }
```

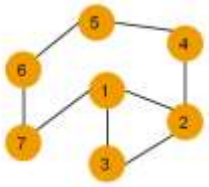


Рисунок 56. Граф из примера в листингах 3.1, 3.2 и 3.3

Тогда в консоли будет выведена последовательность: 1,2,3,7,4,6,5.

2. **Поиск в глубину** (Depth-first search, DFS) предполагает продвижение

вглубь до тех пор, пока это возможно – до попадания в тупик (рис. 57).

Тупик – вершина графа, у которой все смежные вершины посещены.

Из тупика возвращаемся вдоль пройденного пути до вершины, у которой есть ещё не посещенный сосед. И далее двигаться в этом направлении.

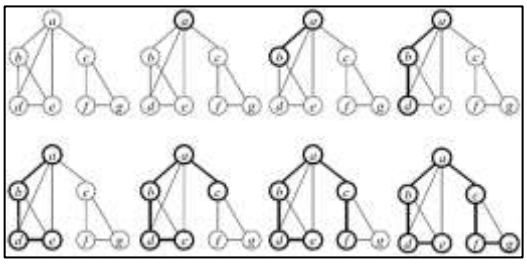


Рисунок 57. Иллюстрация способа обхода графа в глубину

Процесс считается завершенным при попадании в начальную вершину при том, что все смежные с ней должны быть посещены.

Особенности алгоритма поиска в глубину:

• Алгоритм работает как на ориентированных, так и на неориентированных графах.

• Применимость алгоритма зависит от конкретной задачи.

• Для реализации алгоритма удобно использовать стек (нерекурсивный алгоритм) или рекурсию.

• Временная сложность зависит от представления графа:

  • матрицей смежности – $O(n^2)$,

  • нематричное представление – $O(n+m)$, т.к. рассматриваются все вершины и все рёбра.

Применения алгоритма поиска в глубину:

- Поиск любого пути в графе.
- Поиск лексикографически первого пути в графе (лексикографический порядок).
- Проверка, является ли одна вершина графа предком другой.
- Поиск наименьшего общего предка.
- Топологическая сортировка.
- Поиск компонент связности.

Реализация обхода графа в глубину на C++ (с использованием очереди STL) стеком и рекурсией представлены, соответственно в листингах 3.2 и 3.3. Внешний вид графа представлен на рис. 56., граф задан матрицей смежности.

Результатом выполнения этих примеров будет последовательный вывод: 1,2,3,4,5,6,7.

Обход в глубину может быть использован при программировании игр.

В типичной игре существует постоянно расширяющийся граф вариантов

(например, «крестики-нолики»). Последовательность ходов, в свою очередь, хорошо представляется в виде дерева (в котором узлы – это отдельные ходы) – ***дерево игры***.

Листинг 3.2. Обход графа в глубину (стеком)

```
1   #include <iostream>
2   #include <stack> // стек
3   using namespace std;
4   int main()
5   {
6     stack<int> Stack;
7     int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 }, // матрица смежности
8     { 1, 0, 1, 1, 0, 0, 0 },
9     { 1, 1, 0, 0, 0, 0, 0 },
10    { 0, 1, 0, 0, 1, 0, 0 },
11    { 0, 0, 0, 1, 0, 1, 0 },
12    { 0, 0, 0, 0, 1, 0, 1 },
13    { 1, 0, 0, 0, 0, 1, 0 } };
14    int nodes[7]; // вершины графа
15    for (int i = 0; i < 7; i++) // исходно все вершины равны 0
16      nodes[i] = 0;
17    Stack.push(0); // помещаем в очередь первую вершину
18    while (!Stack.empty())
19    { // пока стек не пуст
20      int node = Stack.top(); // извлекаем вершину
21      Stack.pop();
22      if (nodes[node] == 2) continue;
23      nodes[node] = 2; // отмечаем ее как посещенную
24      for (int j = 6; j >= 0; j--)
25      { // проверяем для нее все смежные вершины
26        if (mas[node][j] == 1 && nodes[j] != 2)
27        { // если вершина смежная и не обнаружена
28          Stack.push(j); // добавляем ее в стек
29          nodes[j] = 1; // отмечаем вершину как обнаруженную
30        }
31      }
32      cout << node + 1 << endl; // выводим номер вершины
33    }
34    cin.get();
35    return 0;
36  }
```

Выбор хода – на основе анализа всех путей до конца игры (в примере 9!= 362 880 путей).

В более сложных играх проходят путь до определённой глубины – даже суперЭВМ не способна видеть позиции до конца игры.

Эффективный способ анализа здесь – обход в глубину.

Лишь часть путей дерева игры ведет к выигрышу (другие пути ведут к победе противника – неудаче). При достижении неудачи алгоритм отступает к предыдущему узлу и проверяет другой путь.

Анализ дерева продолжается, пока не будет найден путь к выигрышу (останется лишь сделать первый ход на этом пути).

Листинг 3.3. Обход графа в глубину (рекурсивный способ)

```
1   #include <iostream>
2   using namespace std;
3   int mas[7][7] = { { 0, 1, 1, 0, 0, 0, 1 }, // матрица смежности
4   { 1, 0, 1, 1, 0, 0, 0 },
5   { 1, 1, 0, 0, 0, 0, 0 },
6   { 0, 1, 0, 0, 1, 0, 0 },
7   { 0, 0, 0, 1, 0, 1, 0 },
8   { 0, 0, 0, 0, 1, 0, 1 },
9   { 1, 0, 0, 0, 0, 1, 0 } };
10  int nodes[7]; // вершины графа
11  void search(int st, int n)
12  {
13    int r;
14    cout << st + 1 << " ";
15    nodes[st] = 1;
16    for (r = 0; r < n; r++)
17      if ((mas[st][r] != 0) && (nodes[r] == 0))
18        search(r, n);
19  }
20  int main()
21  {
22    for (int i = 0; i < 7; i++) // исходно все вершины равны 0
23      nodes[i] = 0;
24    search(0, 7);
25    cin.get();
26    return 0;
27  }
```

3.4. Транзитивное замыкание в орграфе. Алгоритм Уоршелла

Бинарное отношение R на множестве X называется транзитивным, если для любых трёх элементов a, b, c этого множества выполнение отношений aRb и bRc влечёт выполнение отношения aRc (рис. 58).

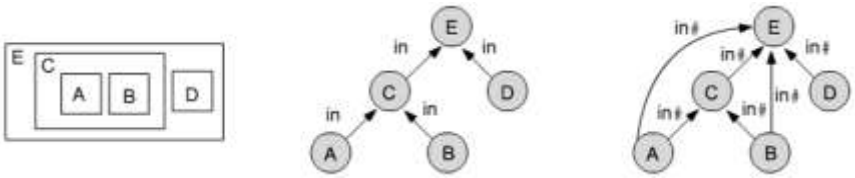


Рисунок 58. Примеры транзитивных отношений

Связь вершин в графе транзитивна – т.е. если есть ориентированный путь из A в B и из B в C, то это значит, что есть путь из A в C.

Транзитивное замыкание бинарного отношения – это пополнение отношения минимальным количеством новых пар так, чтобы отношение стало транзитивным.

Транзитивное замыкание в орграфе – добавление дуг между парами вершин, если в исходном орграфе между ними существуют пути (рис. 59). Т.е. рёбра в транзитивном замыкании соединяют каждую вершину со всеми вершинами, достижимыми из этой вершины в исходном орграфе.

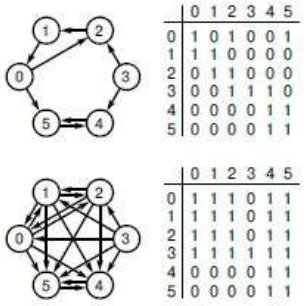


Рисунок 59. Пример орграфа, заданного матрицей смежности, и его транзитивное замыкание

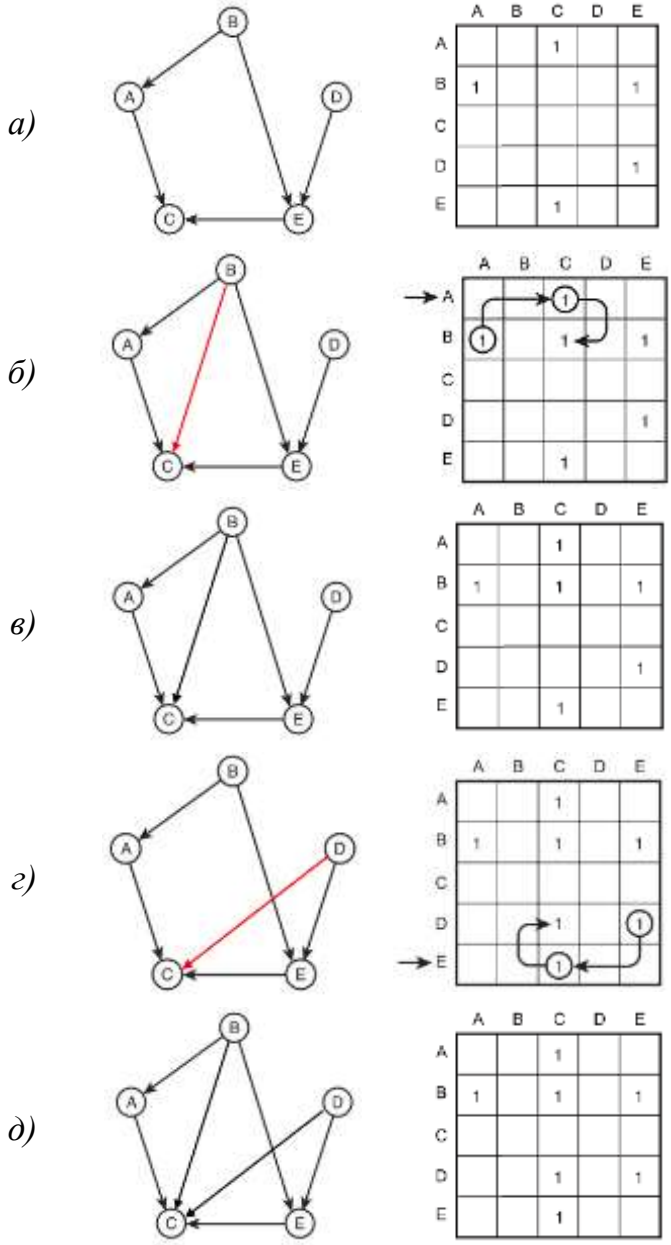


Рисунок 60. Иллюстрация работы алгоритма Уоршелла

Рассмотрим **алгоритм Уоршелла** построения транзитивного замыкания в графе. Пусть орграф задан матрицей смежности (рис. 60.а).

Матрица смежности как способ представления графа содержит все допустимые одношаговые пути – это основа алгоритма.

Идея алгоритма Уоршелла: если есть ориентированный путь из s в i и из i в t, то тогда есть путь и из s в t (за 2 шага).

На основании найденных ранее путей можно строить пути произвольной длины.

Первая строка A (рис. 60.б):

• В ячейке C – единица, т.е. существует путь от A к C.

• Если бы существовал путь от другой вершины X к A, то существовал бы и путь от X к C.

• Какие рёбра графа заканчиваются в вершине A (и есть ли они?)? – см. столбец A – есть одна единица в строке B, т.е. существует ребро от B к A.

• Значит, существуют пути от B к A и от A к C. Тогда от B к C можно перейти за 2 шага.

• Сохранение результата: 1 на пересечении строки B и столбца C.

Строки B, C, D (рис. 60.в):

• Столбец A содержит 1, указывая на существование ребра от B к A.

• Но существуют ли другие рёбра, заканчивающиеся в B?

• Столбец B пуст (ни одно ребро не заканчивается в B), значит ни одна из единиц в строке B не приведет к обнаружению более длинного пути.

• В строке C нет единиц.

• В строке D есть ребро от D к E, но столбец D пуст, значит рёбер в D не существует.

Строка E (рис. 60.г):

• В строке E есть ребро от E к C.

• В столбце E есть ребро от B к E, т.е. существует путь от B к C.

• Однако этот путь уже был обнаружен ранее (есть 1 в соответствующей ячейке таблицы).

• Пути от D к E и от E к C образуют путь от D к C, в эту ячейку ставится 1.

Результат (рис. 60.д):

• В матрицу смежности были добавлены две единицы.

• Измененная матрица показывает, к каким узлам можно перейти от другого узла за любое количество шагов.

• Полученный граф на основе новой матрицы – это и есть транзитивное

замыкание исходного графа.

Программная реализация алгоритма Уоршелла основана на ***дизъюнктивном объединении*** строк, причём в текущей строке:

- Игнорируются диагональные и нулевые ячейки.
- Второй индекс ненулевой ячейки – это номер строки, с которой нужно осуществить дизъюнктивное объединение текущей строки (сравните матрицы смежности на рис. 60.а и 60.д).

Псевдокод основной части алгоритма Уоршелла представлен в листинге 3.4. Внешним циклом перебираем вершины, которые могут быть промежуточными (транзитными). Средним циклом перебираем вершины, которые могут быть начальными в 2-шагового пути. Внутренним циклом перебираем вершины, которые могут быть конечными в 2-шагового пути.

Листинг 3.4. Псевдокод основной части алгоритма Уоршелла

```
for (i = 0; i < V; i++)
  for (s = 0; s < V; s++)
    for (t = 0; t < V; t++)
      if (A[s][i] && A[i][t]) A[s][t] = 1;
```

Наличие ребра между начальной и транзитной вершинами и ребра между транзитной и конечной вершинами позволяет сделать вывод о наличии 2-шагового пути между начальной и конечной вершиной.

3.5. Поиск кратчайшего пути на графе. Алгоритм Дейкстры

Задача поиска кратчайшего пути на графе – это наиболее распространённая задача на взвешенных графах.

Она находит практическое применение во множестве предметных областей. Пример – железная дорога: какой маршрут будет самым экономичным для заданных начальной и конечной станций (например, из A в E)?

«Кратчайший» путь – самый оптимальный по какой-либо характеристике маршрут. Возможные решения задачи поиска:

- алгоритм Дейкстры;
- алгоритм Флойда-Уоршелла;
- алгоритм Беллмана-Форда и др.

Рассмотрим подробнее первые два из перечисленных алгоритмов.

Алгоритм Дейкстры.

Предложен в 1959 г. известным нидерландским учёным в области информатики Э. Дейкстрой.

Алгоритм предназначен для поиска кратчайшего пути в ориентированном

взвешенном графе, при условии, что все рёбра в графе имеют неотрицательные веса.

Позволяет находить кратчайший путь не только от одной заданной вершины к другой, но и ко всем остальным вершинам.

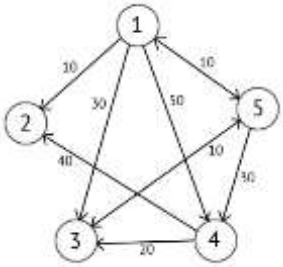
Алгоритм широко применяется в программировании и технологиях, например, в протоколах маршрутизации.

Возможные варианты постановки задачи поиска:

• Найти кратчайшие пути из текущего города до других городов.

• Найти маршрут минимальной стоимости.

Пусть граф представлен в виде небинарной матрицы смежности (рис. 61).



	1	2	3	4	5
1		10	30	50	10
2					
3					10
4		40	20		
5	10		10	30	

Рисунок 61. Орграф, заданный матрицей смежности

Формальное определение задачи поиска:

• Дан взвешенный ориентированный граф G(V,E) без дуг отрицательного веса.

• Найти кратчайшие пути от некоторой вершины графа G до всех остальных вершин этого графа.

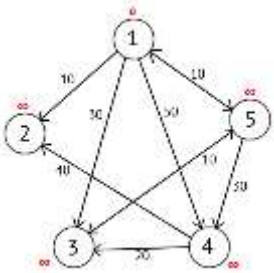


Рисунок 62. Начальная инициализация в алгоритме Дейкстры

Этап 0 – инициализация (рис. 62):

• Каждой вершине из V сопоставим временную метку — минимальное известное расстояние от этой вершины до a.

• Метка самой вершины a (вершина 1) полагается равной 0 (источник),

метки остальных вершин — бесконечности (т.е. расстояния от а до других вершин пока неизвестны).

• Все вершины графа помечаются как непосещённые.

• Алгоритм работает пошагово – на каждом шаге он посещает одну вершину и пытается уменьшать метки.

• Работа алгоритма завершается, когда все вершины посещены (иначе из ещё не посещённых вершин выбирается вершина, имеющая минимальную метку).

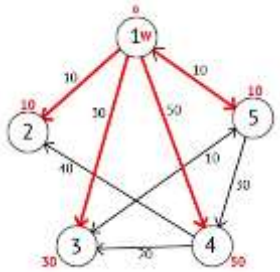


Рисунок 63. Иллюстрация шага 1 алгоритма Дейкстры

Этап 1 (рис. 63):

• Выберем вершину W, которая имеет минимальную метку (сейчас это вершина 1).

• Рассмотрим все вершины, в которые из W есть путь без посредников (2, 3, 4, 5).

• Каждой из них назначим метку, равную сумме метки W и длины пути из W в рассматриваемую вершину, но только в том случае, если полученная сумма будет меньше предыдущего значения метки.

• Если сумма не будет меньше, то оставляем предыдущую метку без изменений.

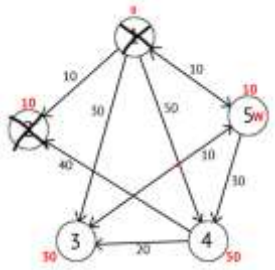


Рисунок 64. Иллюстрация шага 2 алгоритма Дейкстры

Этап 2 (рис. 64):

• Перебрав все вершины, в которые есть прямой путь из W, саму W отмечаем как посещённую.

• Выбираем из ещё не посещенных такую, которая имеет минимальное значение метки – она будет следующей вершиной W (вершины 2 или 5).

• Если есть несколько вершин с одинаковыми метками, то не имеет значения, какую из них выбрать как W – выберем вершину 2.

• Из 2 нет исходящих путей, сразу отмечаем её как посещенную и переходим к следующей вершине с минимальной меткой (вершина 5).

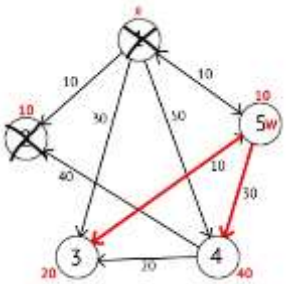


Рисунок 65. Иллюстрация шага 3 алгоритма Дейкстры

Этап 3 (рис. 65):

• Рассмотрим все непосещённые вершины, в которые есть прямые пути из 5.

• Снова находим сумму метки вершины W и веса ребра из W в текущую вершину, и если эта сумма будет меньше предыдущей метки, то заменяем значение метки на полученную сумму.

• Метки 3 и 4-ой вершин стали меньше, т.е. был найден более короткий маршрут в них из источника.

• Далее отмечаем 5-ю вершину как посещенную и выбираем следующую вершину, которая имеет минимальную метку (вершина 3).

• Повторяем все перечисленные действия, пока есть непосещённые вершины.

В результате метки будут равны минимальному расстоянию от источника (рис. 66).

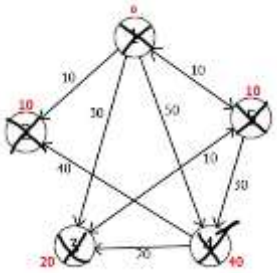


Рисунок 66. Результат работы алгоритма Дейкстры

Время работы алгоритма зависит от используемого типа данных (простая очередь с приоритетом, бинарная или фибоначчиева Куча).

Соответственно, время работы будет варьироваться от O(V3) до O(V*E*log(V)), где V – количество вершин, а E – рёбер.

Пример реализации на языке C++ алгоритма Дейкстры представлен в листинге 3.5.

Листинг 3.5. Код алгоритма Дейкстры

```
1   #include <iostream>
2   using namespace std;
3   const int V = 6;  //число вершин
4   //алгоритм Дейкстры
5   void Dijkstra(int GR[V][V], int st) {
6       int distance[V], count, index, i, u, m = st + 1;
7       bool visited[V];
8       for (i = 0; i < V; i++) {
9           distance[i] = INT_MAX; visited[i] = false;  }
10      distance[st] = 0;
11      for (count = 0; count < V - 1; count++) {
12          int min = INT_MAX;
13          for (i = 0; i < V; i++)
14              if (!visited[i] && distance[i] <= min) {
15                  min = distance[i]; index = i; }
16          u = index;
17          visited[u] = true;
18          for (i = 0; i < V; i++)
19              if (!visited[i] && GR[u][i] && distance[u] != INT_MAX &&
20                  distance[u] + GR[u][i] < distance[i])
21                  distance[i] = distance[u] + GR[u][i];
22      }
23      cout << "Стоимость пути из начальной вершины до остальных:\t\n";
24      for (i = 0; i < V; i++)
25          if (distance[i] != INT_MAX)
26              cout << m << " > " << i + 1 << " = " << distance[i] << endl;
27          else cout << m << " > " << i + 1 << " = " << "маршрут недоступен" << endl;
28  }
29
30  int main() {
31      setlocale(LC_ALL, "Rus");
32      int start, GR[V][V] = {
33      {0, 1, 4, 0, 2, 0},
34      {0, 0, 0, 9, 0, 0},
35      {4, 0, 0, 7, 0, 0},
36      {0, 9, 7, 0, 0, 2},
37      {0, 0, 0, 0, 0, 8},
38      {0, 0, 0, 0, 0, 0} };
39      cout << "Начальная вершина >> "; cin >> start;
40      Dijkstra(GR, start - 1);
41      system("pause");
42      return 0;
43  }
```

3.6. Алгоритм Флойда-Уоршелла

Алгоритм Флойда-Уоршелла – это динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа без циклов. Предложен Р. Флойдом и С. Уоршеллом 1962 г. и, независимо от них, Б. Ройем в 1959 г.

Возможны отрицательные веса.

Алгоритм представляет собой простой перебор всех путей по матрице смежности и выбор из них наименьшего. Эффективен в плотных графах.

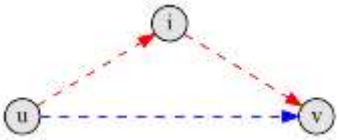


Рисунок 67. Постановка задачи в алгоритме Флойда-Уоршелла

Постановка задачи (рис. 67):

• Дан взвешенный ориентированный граф G(V,E), в котором вершины пронумерованы от 1 до n.

• Требуется найти матрицу кратчайших расстояний d, в которой элемент d_{ij} либо равен длине кратчайшего пути из i в j, либо равен +∞, если вершина j не достижима из i.

• d^i_{uv} – длина кратчайшего пути между вершинами u и v, содержащего, помимо u и v, только вершины из множества {1..i}.

Описание алгоритма:

• Перед началом алгоритма матрица d заполняется длинами рёбер графа (или признаками их отсутствия): $d^0_{uv} = \omega_{uv}$ – длина ребра, если оно существует.

• На каждом шаге алгоритма берётся очередная i-я вершина, для которой справедливо:

• Кратчайший путь между u, v не проходит через вершину i, тогда

$$d^i_{uv} = d^{i-1}_{uv}.$$

• Существует более короткий путь между u, v, проходящий через i, тогда он сначала идёт от u до i, потом от i до v: $d^i_{uv} = d^{i-1}_{ui} + d^{i-1}_{iv}$.

• Тогда кратчайший путь между u, v – это минимум из двух значений:

$$d^i_{uv} = \min(d^{i-1}_{uv}, d^{i-1}_{ui} + d^{i-1}_{iv}) \text{ – рекуррентная формула.}$$

Пошаговый пример работы алгоритма Флойда-Уоршелла для взвешенного орграфа представлен на рис. 68.

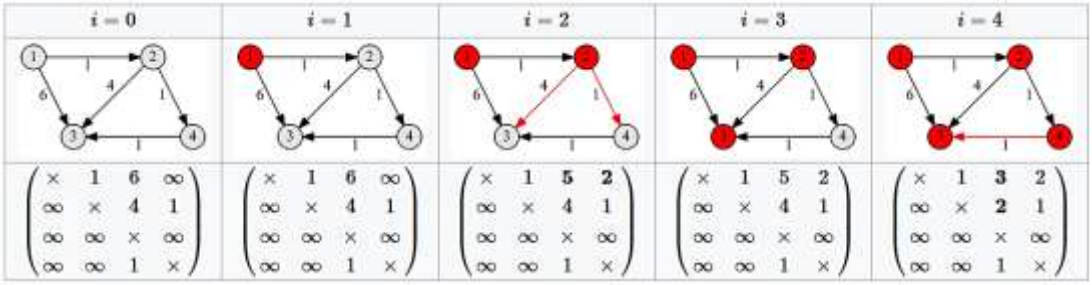


Рисунок 68. Иллюстрация работы алгоритма Флойда-Уоршелла

Псевдокод основной части алгоритма представлен в листинге 3.6.

В итоге матрица d^n является искомой матрицей кратчайших путей, по-

скольку содержит в себе длины кратчайших путей между всеми парами вершин, имеющих в качестве промежуточных вершины из множества {1..n}, т.е. все вершины графа.

Листинг 3.6. Псевдокод основной части алгоритма Флойда-Уоршелла

$d^0_{uv} = w;$
for $i \in V$
  for $u \in V$
    for $v \in V$
      $d^i_{uv} = min(d^{i-1}_{uv}, d^{i-1}_{ui} + d^{i-1}_{iv})$

Такая реализация работает за $\Theta(n^3)$ времени и использует $\Theta(n^3)$ памяти.

Пример реализации алгоритма средствами языка C++ представлен в листинге 3.7.

Листинг 3.7. Пример реализации алгоритма Флойда-Уоршелла

```
1   #include <iostream>
2   #include <vector>
3   #include <iomanip>
4   using namespace std;
5
6   void FloydWarshall(vector<vector<int>>& myG) {
7
8       int inf = 1000000000;
9       size_t k, i, j, w = myG.size();
10      vector<vector<int>> minP = myG;
11
12      // Устанавливаем в вершины где нет путей inf
13      for (i = 0; i < w; i++)
14          for (j = 0; j < w; j++)
15              if (minP[i][j] == 0 && i != j)
16                  minP[i][j] = inf;
17      // Собственно реализация алгоритма Флойда-Уоршелла
18      for (k = 0; k < w; k++)
19          for (i = 0; i < w; i++)
20              for (j = 0; j < w; j++)
21                  if (minP[i][k] + minP[k][j] < minP[i][j])
22                      myG[i][j] = minP[i][k] + minP[k][j];
23  }
24
```

```
25  int main() {
26      // Матрица смежности
27      vector<vector<int>> G{
28          {0, 4, 0, 2, 2, 0, 0, 0, 0},
29          {0, 0, 7, 0, 0, 3, 0, 0, 0},
30          {0, 0, 0, 0, 0, 2, 0, 0, 0},
31          {0, 0, 0, 0, 0, 0, 1, 1, 0},
32          {0, 2, 0, 3, 0, 2, 0, 6, 1},
33          {0, 0, 0, 0, 0, 0, 0, 0, 2},
34          {0, 0, 0, 0, 0, 0, 0, 5, 0},
35          {0, 0, 0, 0, 0, 0, 0, 0, 5},
36          {0, 0, 0, 0, 0, 0, 0, 0, 0}
37      };
38
39      FloydWarshall(G);
40      for (auto& i : G) {
41          for (auto& j : i)
42              cout << j << setw(3);
43          cout << "\n";
44      }
45      system("pause");
46      return 0;
47  }
```

Результат работы кода из листинга 3.7 см. на рис. 69.

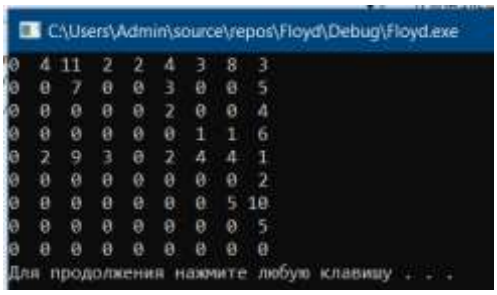


Рисунок 69. Вывод программы из листинга 3.7

3.7. Минимальное остовное дерево в графе. Алгоритм Прима

Остовное (стягивающее, покрывающее) дерево в графе можно получить из него удалением некоторых рёбер. Может существовать несколько остовных деревьев.

Во взвешенном графе вес остовного дерева – это сумма весов всех его ребёр.

Минимальное остовное дерево (МОД) – с минимальным возможным весом.

Важная в практическом плане задача – нахождение минимального остовного дерева в связном взвешенном неориентированном графе (рис. 48).

Пример:

• Есть n городов, их необходимо соединить дорогами, так, чтобы был путь из любого города в любой другой (напрямую или через другие города).

• Разрешается строить дороги между заданными парами городов и известна стоимость строительства каждой такой дороги.

• Требуется решить, какие именно дороги нужно строить, чтобы минимизировать общую стоимость строительства.

В терминах теории графов это задача о нахождении минимального остовного дерева в графе, вершины которого – города, а рёбра – пары городов, между которыми можно проложить дорогу; вес ребра – это стоимость строительства соответствующей дороги.

Основные решения этой задачи:

• алгоритм Прима;

• алгоритм Краскала;

• алгоритм Борувки;

• алгоритм обратного удаления.

Рассмотрим подробнее первые два алгоритма.

Алгоритм Прима – это алгоритм построения МОД взвешенного связного неориентированного графа, предложен В. Ярником в 1930 г. и, независимо, Р. Примом в 1957 г. и Э. Дейкстрой в 1959 г.

В своей идее алгоритм Прима похож на алгоритм Дейкстры.

Вычислительная сложность зависит от способа поиска очередного минимального ребра – разные реализации: от $O(N^3)$ в худшем случае до (при оптимизации) $O(M \cdot \log N)$ в разреженных графах и $O(N^2)$ в плотных.

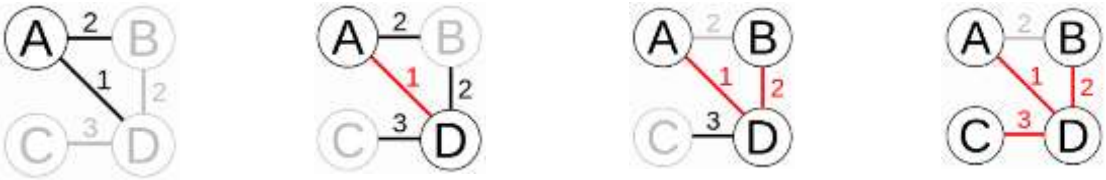


Рисунок 70. Последовательность шагов в алгоритме Прима

Идея алгоритма Прима (рис. 70):

• Дан связный неориентированный граф.

• Если разделить вершины графа на два множества (обработанные и необработанные), первое из которых составляет связную часть МОД, то ребро минимальной длины, связывающее эти два множества гарантированно будет входить в МОД.

• Тогда для нахождения МОД можно начать с произвольной вершины и постепенно добавлять ближайшие (через инцидентное ребро минимальной стоимости) к уже имеющимся.

• Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Реализация алгоритма Прима на языке C++ представлена в листинге 3.8.

Листинг 3.8. Пример реализации алгоритма Прима

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4
5  const int V = 5; //число вершин
6  //матрица смежности графа
7  int G[V][V] = {
8      {0, 9, 75, 0, 0},
9      {9, 0, 95, 19, 42},
10     {75, 95, 0, 51, 66},
11     {0, 19, 51, 0, 31},
12     {0, 42, 66, 31, 0}
13 };
14
15 int main() {
16     int no_edge = 0;  // количество рёбер
17     int selected[V]; // массив обработанных вершин
18     // изначально все необработаны:
19     memset(selected, false, sizeof(selected));
20     selected[0] = true; // посещаем начальную вершину
21     int x, y;
22     cout << "Edge" << " : " << "Weight" << endl;
23     // Для каждой вершины в наборе найти все смежные вершины,
24     // вычислить расстояние от вершины, выбранной на шаге 1.
25     // Если вершина уже обработана, отбросить её,
26     // иначе выбрать другую вершину, ближайшую к выбранной
27     while (no_edge < V - 1) {
28         int min = INT_MAX;
29         x = 0; y = 0;
30         for (int i = 0; i < V; i++) {
31             if (selected[i]) {
32                 for (int j = 0; j < V; j++) {
33                     if (!selected[j] && G[i][j]) {
34                         if (min > G[i][j]) {
35                             min = G[i][j];
36                             x = i; y = j;
37                         }
38                     }
39                 }
40             }
41         }
42         cout << x << " - " << y << " :  " << G[x][y] << endl;
43         selected[y] = true;
44         no_edge++;
45     }
46     system("pause");
47     return 0;
48 }
```

Вывод программы из листинга 3.8:

Edge : Weight

0 – 1 : 9

1 – 3 : 19

3 – 4 : 31

3.8. Алгоритм Краскала

Алгоритм Краскала (Крускала) – это эффективный алгоритм построения МОД взвешенного связного неориентированного графа.

Он предполагает сортировку всех рёбер в порядке возрастания длины и поочередное добавление их в МОД, если они соединяют различные компоненты связности.

Общее время работы алгоритма можно принять за O(M*log(M)).

Идея алгоритма Краскала:

• Пусть есть некоторые рёбра, входящие в МОД.

• Тогда среди всех рёбер, соединяющих различные компоненты связности, в МОД будет входить ребро с минимальной длиной.

• Для реализации алгоритма необходимо сортировать рёбра по возрастанию длины и проверять, соединяет ли ребро две различных компоненты связности.

Поддерживать текущие компоненты связности можно с помощью структуры данных DSU (система непересекающихся множеств).

Шаги алгоритма Краскала (рис. 71):

• Сортировать все рёбра от малого веса до высокого.

• Взять ребро с наименьшим весом и добавить его в МОД.

• Если добавление ребра создало цикл, то отклонить это ребро.

• Добавлять рёбра, пока не будут достигнуты все вершины.

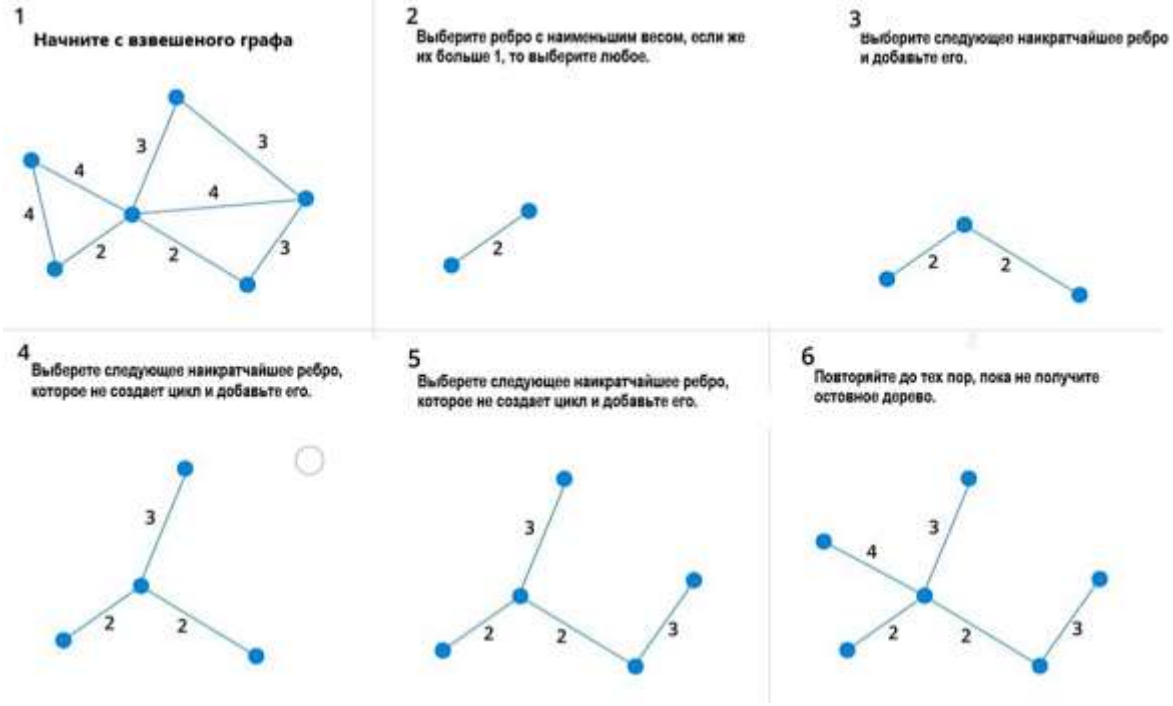


Рисунок 71. Иллюстрация работы алгоритма Краскала

Реализация алгоритма Краскала на языке C++ представлена в листинге 3.9.

Листинг 3.9. Пример реализации алгоритма Краскала

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int p[100000];
7  int rk[100000];
8
9  void init_dsu() {
10     for (int i = 0; i < 100000; i++) { p[i] = i; rk[i] = 1; }
11 }
12
13 int get_root(int v) {
14     if (p[v] == v) return v;
15     //На выходе из рекурсии переподвешиваем v:
16     else return p[v] = get_root(p[v]);
17 }
18
19 bool merge(int a, int b) {
20     int ra = get_root(a), rb = get_root(b);
21     if (ra == rb) return false;
22     else {
23         if (rk[ra] < rk[rb]) p[ra] = rb;
24         else if (rk[rb] < rk[ra])
25             p[rb] = ra;
26         else {p[ra] = rb; rk[rb]++;}
27         return true;
28     }
29 }
31 //Тип для представления рёбер.
32 struct edge {
33     int a, b, len;
34     bool operator<(const edge& other) {
35         return len < other.len;
36     }
37 };
38
39 int main() {
40     vector<edge> edges;
41     //Ввод edges...
42
43     sort(edges.begin(), edges.end());
44     int mst_weight = 0;
45     init_dsu();
46     for (edge e : edges) {
47         //Если a и b находятся в разных компонентах:
48         if (merge(e.a, e.b)) {
49             //Добавить ребро в минимальный остов:
50             mst_weight += e.len;
51         }
52     }
53     cout << "Minimum spanning tree weight: " << mst_weight << endl;
54     system("pause");
55 }
```

Контрольные вопросы

1. Что такое орграф? Что такое связный граф?
2. В чём заключается задача четырёх красок?
3. Что такое сеть? Поток в сети?
4. Перечислите основные способы представления графа в программе.
5. Что такое инциденция?
6. Перечислите способы обхода графа. Что такое тупик в графе?
7. Что такое транзитивное замыкание в орграфе?
8. Опишите основную идею алгоритма Уоршелла.
9. Опишите идею задачи поиска кратчайшего пути на графе.
10. Опишите суть алгоритма Дейкстры. Каковы его ограничения?
11. Опишите шаги алгоритма Флойда-Уоршелла.
12. Что такое минимальное остовное дерево в графе?
13. Опишите суть алгоритма Прима.
14. Опишите шаги алгоритма Краскала.

4. КОДИРОВАНИЕ ИНФОРМАЦИИ

4.1. Понятие кодирования данных

Информация – это первичное, неопределяемое понятие в информатике. Информация – свойство материи, она уменьшает неопределенность состояния системы.

В информации заключен определённый смысл, который может быть воспринят получателем (***семантика***). Но, как показал ещё К. Шеннон, для технических систем важна форма представления информации (***синтаксис***).

Данные – это формализованная информация.

Количество информации – мера уменьшения этой неопределенности. Так, количество информации в 1 бит уменьшает неопределённость состояния системы в 2 раза.

Кодирование информации (в широком смысле) – это преобразование формы информации, удобной для её непосредственного использования, в форму, удобную для её передачи, хранения или автоматической обработки.

Код – взаимно однозначное отображение конечного упорядоченного множества символов из конечного алфавита на иное множество символов для передачи, хранения или модификации информации (рис. 72).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рисунок 72. Пример кодирования – кодировка символов ASCII

Примеры: код Морзе, двоичное кодирование, кодировка символов ASCII.

Теория кодирования – это наука о свойствах кодов и их пригодности для достижения поставленных целей.

Основные проблемы, решаемые в теории кодирования – вопросы взаимной однозначности кодирования и сложности реализации канала связи при заданных условиях.

Направления в рамках этой области науки:

- сжатие данных;
- криптология (криптография и криптоанализ);
- контроль целостности данных при модификации и передаче;
- физическое кодирование.

Исторически явно проявляет себя тенденция на увеличение объёмов обрабатываемой информации, что, в свою очередь, предъявляет всё более высокие требования к ёмкости и скорости доступа к данным накопителей, а также к пропускной способности каналов связи.

Но есть физические ограничения оборудования и предел экономической целесообразности. Поэтому востребовано направление **сжатия данных** – алгоритмического преобразования для уменьшения их объёма.

Сжатие основано на устранении **избыточности**.

Обычное представление информации почти всегда избыточно (***естественная избыточность***):

• повторяющиеся фрагменты в текстах;

• невоспринимаемые или несущественные для восприятия частоты в звуковой информации;

• области одинакового цвета в графических изображениях.

Устраняя избыточность, можно избежать расходов на дополнительные ресурсы.

Методы (алгоритмы) сжатия:

• Неискажающие (loseless, сжатие без потерь) – гарантируют, что декодированные данные будут в точности совпадать с исходными. Сжатие достигается за счет более экономичного представления данных.

• Искажающие (lossy, сжатие с потерями) – полное восстановление невозможно. Сжатие достигается за счет потери части информации.

Методы (алгоритмы) сжатия:

• Общего назначения (general-purpose) – не учитывают физическую природу, специфику входных данных, ориентированы на сжатие любых видов данных, которые хранятся в ЭВМ.

• Специальные (special) методы сжатия – ориентированы на сжатие данных определённой природы, например, звука, изображений и пр. Учётом специфики достигается лучшая степень сжатия.

Методы сжатия общего назначения – неискажающие; искажающими могут быть только специальные методы.

Критерии методов сжатия:

1. Качество (коэффициент или степень) сжатия, т. е. отношение длины (в битах) исходного представления к длине сжатого представления данных.

2. Скорость кодирования и декодирования, определяемые временем, затрачиваемым на кодирование и декодирование данных.

3. Объём требуемой для реализации алгоритма сжатия памяти.

Алгоритмы, использующие больше ресурсов (времени и памяти), обычно достигают лучшего качества сжатия, и наоборот: менее ресурсоемкие алгоритмы по качеству сжатия, как правило, уступают более ресурсоемким.

Построение оптимального алгоритма сжатия данных – нетривиальная задача, т.к. необходимо добиться высокого качества сжатия при небольшом объёме используемых ресурсов.

Критерии оценки методов сжатия на практике сильно зависят от области применения, например:

• в ***системах реального времени*** необходимо обеспечить высокую скорость кодирования и декодирования;

• для ***встроенных систем*** критический параметр – объем требуемой памяти;

• для систем ***долговременного хранения*** данных – качество сжатия и/или скорость декодирования и пр.

Надёжность алгоритма сжатия обеспечивается как безошибочностью программирования и дизайна, так и характеристиками выбранных алгоритмов.

Для обеспечения конечного и заранее известного времени сжатия (в наихудшем случае), необходимо, чтобы алгоритм обладал:

• хорошо детерминированным временем работы (желательно, мало зависящим от кодируемых данных) и

• заранее известным объёмом требуемой памяти.

Выполнение этих требований необходимо при разработке встроенных систем, систем реального времени, файловых систем со сжатием данных и других систем с жесткими ограничениями на разделяемые ресурсы.

Сложность алгоритма сжатия:

• Алгоритмы, обладающие полиномиальной или экспоненциальной сложностью, считаются в теории хорошим решением проблемы.

• Но на практике приемлемы только алгоритмы с линейной или линейно-логарифмической (квазилинейной) временной сложностью.

• Крайне желательно, чтобы среднее время работы (на типичных данных) было линейным.

Эффективные методы сжатия без потерь (рис. 73):

1. Статистические (энтропийные) алгоритмы.
2. Словарные алгоритмы.
3. Алгоритмы сортировкой блоков (семейство BWT/BS).
4. Прочие.

Рисунок 73. Алгоритмы сжатия без потерь

4.2. Статистическое (энтропийное) кодирование

Статистическое (энтропийное) **моделирование** – это обратимое кодирование с целью сжатия с помощью усреднения вероятностей (частот) появления элементов в закодированной последовательности.

Так, до кодирования у элементов последовательности вероятность появления (частота) различна. В результирующей последовательности вероятности появления отдельных символов почти одинаковы.

Вероятность появления следующего символа предсказывается на основе анализа частоты появления различных последовательностей символов в ранее закодированной части сообщения (рис. 74).

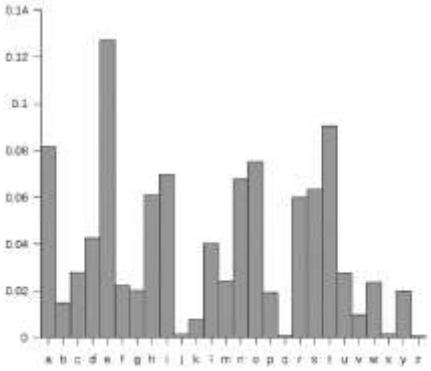


Рисунок 74. Пример распределения вероятностей символов английского алфавита

Вероятность события появления в тексте того или иного символа алфавита (**вероятность символа**) – случайная величина.

Вероятности символов различны, функция распределения – нормальное распределение Гаусса, однозначно характеризуется двумя величинами:

1. **Математическое ожидание** – среднее ожидаемое значение случайной величины с учетом распределения. Так, средняя длина кода – среднее число битов (элементарных символов) на один символ первичного алфавита с учётом частот символов:

$$l_{\text{ср}} = \sum_{s} p_s \cdot l_s,$$

где s — множество символов алфавита; p_s — вероятность появления символа; l_s — количество бит в коде символа.

2. **Дисперсия** характеризует меру рассеивания возможных значений случайной величины около её математического ожидания (меньший разброс считается лучшим):

$$\delta = \sum_s p_s (l_s - l_{\text{ср}})^2.$$

Алфавит	и	л	п	к	« »	в	у
Кол. вх.	6	3	2	2	2	2	1
Вероятн.	0.24	0.12	0.08	0.08	0.08	0.08	0.04

Алфавит	н	а	с	й	р	о	ч
Кол. вх.	1	1	1	1	1	1	1
Вероятн.	0.04	0.04	0.04	0.04	0.04	0.04	0.04

Рисунок 75. Пример числовых характеристик кодирования

На рис. 75 представлен алфавит фразы, отсортированный по вероятности (частоте вхождений) всех её (фразы) символов. Тогда эту фразу можно закодировать следующим образом:

```
 п     у     п     к     и     н    « »    в     а     с     и     л     и     й
1010 11000 1010 1011 00 11001 010 011 11010 11011 00 100 00 11100
« »   к     и     р     и     л     л     о     в     и     ч
010 1011 00 11101 00 100 100 11110 011 00 11111
```

Для полученного кода средняя длина будет равна

$$l_{\text{ср}} = 0.24 \cdot 2 + 0.12 \cdot 3 + 2 \cdot 0.08 \cdot 3 + 2 \cdot 0.08 \cdot 4 + 8 \cdot 0.01 \cdot 5 = 2.36 \text{ бит/символ.}$$

Для полученного кода дисперсия будет равна

$$\delta = 0.24 \cdot (2 - 2.36)^2 + 0.12 \cdot (3 - 2.36)^2 + 2 \cdot 0.08 \cdot (3 - 2.36)^2 + \\ + 2 \cdot 0.08 \cdot (4 - 2.36)^2 + 8 \cdot 0.01 \cdot (5 - 2.36)^2 = 1.1337$$

Неоднородность распределения частот символов алфавита определяет **избыточность** сообщения. **Коэффициент сжатия** – отношение длины исходного сообщения к выходному коду.

Рассчитаем коэффициент сжатия относительно использования кодировки ASCII (8 бит/символ).

$L_{\text{ASCII}} = 8 \cdot 25 = 200$ бит.

$L_{\text{Huff}} = 6 \cdot 2 + 3 \cdot 3 + 2 \cdot 2 \cdot 3 + 2 \cdot 2 \cdot 4 + 8 \cdot 5 = 89$ бит.

Следовательно, коэффициент сжатия будет равен

$$K_{\text{сж}} = \frac{L_{\text{ASCII}}}{L_{\text{Huff}}} \approx 2.247$$

Коэффициент сжатия относительно равномерного кода (5 бит/символ, т. к. у нас всего 25 символов) будет равен

$$K_{\text{сж}} = \frac{5 \cdot 25}{L_{\text{Huff}}} = \frac{125}{89} \approx 1.404$$

Варианты кодов энтропийного моделирования:

• Различным элементам исходной последовательности сопоставляются результирующие коды переменной длины (код Шеннона-Фано, код Хаффмана). При этом чем больше вероятность появления исходного элемента, тем короче соответствующая ему результирующая последовательность.

• Различным элементам исходной последовательности сопоставляется фиксированное число элементов конечной последовательности (код Танстола – отображение исходных символов строки на коды с фиксированным количеством битов).

• Иные коды с операциями над последовательностями символов (например, кодирование длин серий: kkkkk→(k,5)).

• Если заранее известны примерные характеристики энтропии потока данных – унарное кодирование, гамма-коды Элиаса, Фибоначчи, Голомба, Райса.

Применимость статистических алгоритмов:

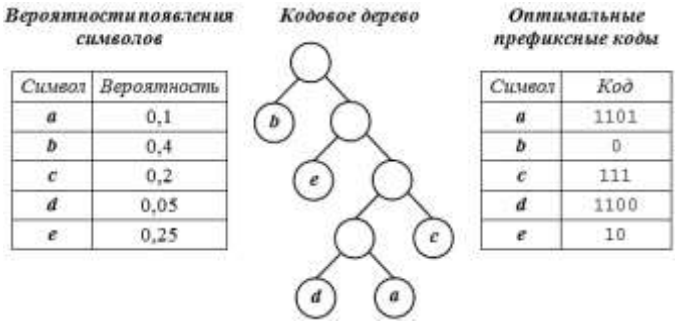
• Чем более предсказуемы входные данные (энтропия меньше), тем лучше их можно сжать. Значит, случайная независимая равновероятная последовательность сжатию без потерь не поддаётся.

• Статистические алгоритмы обычно обеспечивают высокое качество сжатия, но обладают низкой скоростью сжатия/декодирования и требуют большого объёма ОЗУ.

• Многие способы реализации для получения оценок вероятностей появления символов используют команды умножения и/или деления, вычисления с плавающей точкой.

• В целом, область их применения ограничена.

В **префиксных кодах** ни одно кодовое слово не является полным началом (префиксом) никакого другого слова – гарантия однозначности декодирования (рис. 76).



Символ	Вероятность
a	0,1
b	0,4
c	0,2
d	0,05
e	0,25

Символ	Код
a	1101
b	0
c	111
d	1100
e	10

Рисунок 76. Префиксный код

Способы построения префиксных кодов: коды Шеннона, Шеннона-Фано, Хаффмана.

Длина каждого кодового слова – целое число битов, следовательно, префиксные коды неэффективны на алфавитах малой мощности (2-8 символов)

или при наличии символов с большой вероятностью появления (более 30-50%).

По качеству сжатия могут уступать **арифметическим**.

4.3. Арифметический код

В арифметических кодах нет явного соответствия между символами и кодовыми словами, отсюда большая гибкость алгоритма в представлении дробных частот.

Качество близко к теоретическому минимуму (и при малой мощности алфавита, и при неравномерном распределении вероятностей).

При большой мощности кодируемого алфавита арифметический код медленнее метода префиксных кодов (при небольшой разнице в качестве).

В большинстве случаев префиксное кодирование более предпочтительно для практического использования.

Пусть имеется алфавит, а также данные о частотности использования символов (рис. 77).

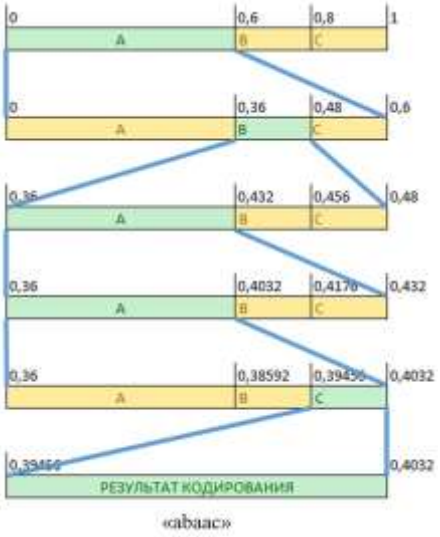


Рисунок 77. Иллюстрация работы арифметического кодирования

• Рассмотрим на координатной прямой отрезок от 0 до 1 – рабочий отрезок.

• Разобьём его на отрезки с длинами, равными частотам использования символов, каждый такой отрезок будет соответствовать одному символу.

• Возьмём первый символ из потока и найдём для него отрезок среди только что сформированных – теперь он рабочий.

• Разобьём его таким же образом пропорционально частотам.

• Выполним эту операцию аналогично для некоторого числа последовательных символов.

• Кодом будет последний рабочий отрезок или любое число из него

(например, 0.4).

4.4. Алгоритм Шеннона-Фано

Алгоритм Шеннона-Фано – это алгоритм префиксного неоднородного кодирования, предложен в 1948 г. К. Шенноном, позже Р. Фано.

Избыточность – в неоднородности распределения частот символов его (первичного) алфавита.

Коды более частых символов заменяются короткими двоичными последовательностями, коды более редких — более длинными.

Коды префиксные – никакое кодовое слово не является префиксом любого другого, что позволяет однозначно декодировать любую последовательность кодовых слов.

Символ	Вероятность	Разбиение на подгруппы				
a_1	0.5	1				
a_2	0.2		01			
a_3	0.1			001		
a_4	0.1	0			0001	
a_5	0.05		00	000	0000	00001
a_6	0.05					00000

Рисунок 78. Иллюстрация работы алгоритма кодирования Шеннона-Фано

Этапы построения кодов Шеннона-Фано (рис. 78):

1. Расположить все символы алфавита в порядке невозрастания вероятностей.

2. Разделить весь алфавит на две группы так, чтобы суммарные вероятности в каждой из групп были примерно равны.

Первой или верхней группе в качестве первого символа кодовой комбинации присвоить 1, второй или нижней – 0.

3. Продолжаем процесс разделения: каждую из групп вновь делим на две подгруппы с примерно одинаковыми вероятностями.

В качестве вторых символов кодовых комбинаций для верхних подгрупп выбираем 0, а для нижних – 1.

4. Весь процесс повторяется до тех пор, пока в каждой из подгрупп не останется по одному символу (листы дерева).

Особенности кодирования Шеннона-Фано:

• В большинстве случаев длина кода равна длине похожего по принципу кода Хаффмана.

• Но алгоритм не гарантирует оптимального кодирования, поэтому более

эффективным (более устойчивым) методом считается код Хаффмана.

Кодирование Шеннона-Фано на сегодняшний день не представляет практического интереса.

4.5. Код Хаффмана

Алгоритм предложен в 1952 г. Д. Хаффманом. Остаётся оптимальным (в отличие от кода Шеннона-Фано).

Обладая высокой эффективностью, **код Хаффмана** и его многочисленные адаптивные версии лежат в основе многих методов, используемых в алгоритмах кодирования.

Этапы кодирования (рис. 79):

• Построение оптимального кодового Н-дерева.

• Построение отображения код-символ на основе построенного Н-дерева.

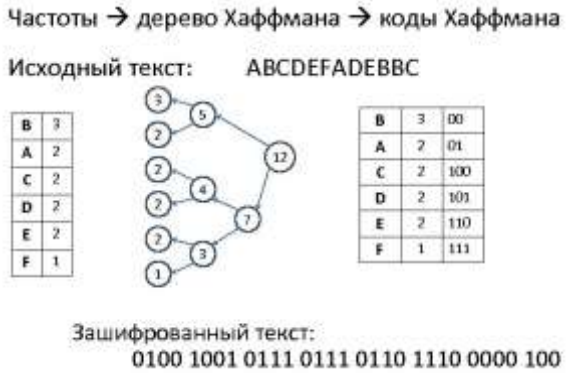


Рисунок 79. Иллюстрация работы алгоритма кодирования Хаффмана

Идея кодирования Хаффмана:

1. Расположить все символы алфавита в порядке невозрастания вероятностей.

2. Два наименее вероятных символа объединяются в одного родителя с суммарной вероятностью, отсюда возникает новый алфавит, который также подлежит упорядочиванию.

3. Процесс объединения продолжается до тех пор, пока в новом алфавите не останется всего два символа, порождающих единственного родителя (корень Н-дерева).

4. Процедура кодирования: двигаясь в обратном направлении, верхнему из объединённых символов в качестве очередного бита кодовой комбинации добавляется 1, нижнему – 0.

Процесс создания кодовых комбинаций продолжается, пока не дойдёт до исходного алфавита.

Особенности кода:

• Префиксный код – ни один из кодов не является префиксом другого, они

могут быть однозначно декодированы.

• Наиболее частый символ сообщения кодируется наименьшим количеством бит, наиболее редкий символ – наибольшим.

Недостатки:

• Для восстановления содержимого сообщения декодер должен знать таблицу частот кодера, следовательно, длина сжатого сообщения увеличивается на длину таблицы частот.

• Необходимость наличия полной частотной статистики перед началом кодирования требует 2 проходов по сообщению: для построения модели сообщения (таблицы частот и дерева), другого для кодирования.

На практике частоты символов алфавита бывают известны кодеру крайне редко. **Адаптивный алгоритм** Хаффмана позволяет не передавать таблицу кодов и ограничиться одним проходом по сообщению (рис. 80).

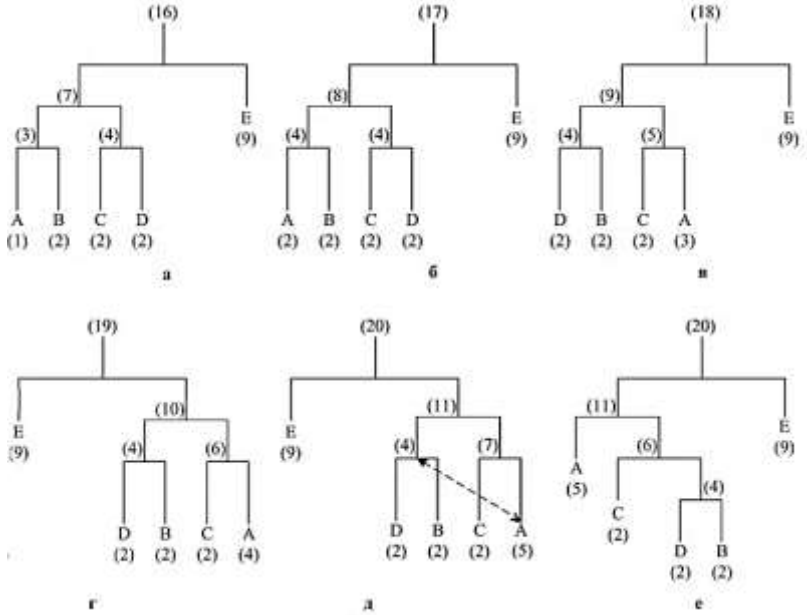


Рисунок 80. Иллюстрация работы адаптивного кода Хаффмана

Кодер и декодер согласованно динамически строят и модифицируют Н-дерево с каждым новым символом:

• Первый символ записывается в выходной поток несжатым, помещается в пустое дерево, ему присваивается код.

• Если он встретится еще раз, в выходной поступит код, а вес листа в дереве инкрементирован (а, значит, и веса родителей до корня).

• Если изменение веса нарушает упорядоченность Н-дерева, то перестраивается дерево.

Алгоритмы перестроения: Фоллера-Галлагера-Кнута (FGK) и алгоритм Виттера.

Кодирование Хаффмана широко применяется при сжатии данных, в том

числе:

• при сжатии фото- и видеоизображений (JPEG, MPEG),

• в популярных архиваторах (PKZIP, LZH и др.),

• в протоколах передачи данных HTTP (Deflate), MNP5 и MNP7 и др.

В 2013 г. была предложена модификация алгоритма Хаффмана, позволяющая кодировать символы дробным количеством бит – ANS. На базе данной модификации реализованы алгоритмы сжатия Zstandard (Zstd, Facebook, 2015-2016) и LZFSE (Apple, OS X 10.11, iOS 9, 2016).

4.6. Словарное кодирование. Алгоритм RLE

Словарное кодирование основано на разбиении данных на подстроки (слова) и замене их на их индексы в словаре (рис. 81).

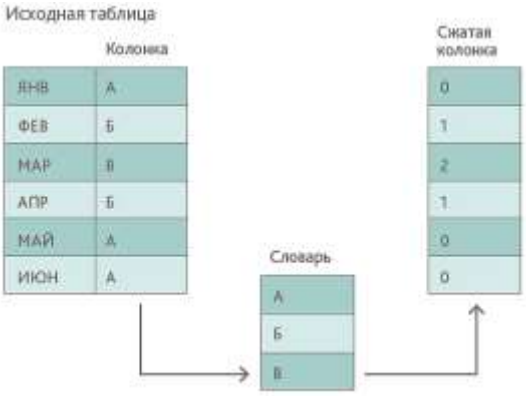


Рисунок 81. Иллюстрация работы словарного кодирования

К данной группе относятся алгоритмы семейств RLE (run-length encoding, кодирование длин серий); LZ (Лемпеля-Зива) и все его модификации и аналоги.

В настоящее время наиболее распространённый подход. Его преимущество – оперативный доступ к данным – простая и быстрая процедура распаковки.

Словарь чаще всего постепенно пополняется словами из исходного блока данных в процессе сжатия (рис. 82).

ВХОДНАЯ ФРАЗА (В СЛОВАРЬ)	КОД	ПОЗИЦИЯ СЛОВАРЯ
		0
"К"	<0,'К'>	1
"Р"	<0,'Р'>	2
"А"	<0,'А'>	3
"С"	<0,'С'>	4
"Н"	<0,'Н'>	5
"АЯ"	<3,'Я'>	6
	<0,' '>	7
"КР"	<1,'Р'>	8
"АС"	<3,'С'>	9
"КА"	<1,'А'>	10

Рисунок 82. Пример словаря

Чем больше размер словаря, тем выше эффективность кода.

При неоднородных данных (с резким изменением типа) словарь может заполняться неактуальными словами.

Сжатие требует много памяти – на порядок больше размера словаря.

Словарь может быть статическим (слова только добавляются) и динамическим (адаптивным, слова могут удаляться по мере чтения входных данных).

Алгоритм группового кодирования RLE (run-length encoding) – кодирование длин серий (повторов). Серия (последовательность одинаковых символов) заменяется на один символ и число повторов (рис. 83).

Здесь отрицательное значение повтора – это длина последовательности неповторяющихся символов.

Алгоритм RLE используется в форматах PCX, TIFF, BMP.

Рисунок 83. Примеры работы алгоритма группового кодирования RLE

Области применения:

- простая графика (без плавных переходов – не фото);
- двоичные данные;
- низкокачественные звуковые семплы (после дельта-кодирования).

4.7. Алгоритмы семейства LZ

В 1977-78 гг. Я. Зив, А. Лемпел, М. Махони предложили алгоритмы LZ77 и LZ78.

Идея алгоритмов LZ проста: второе и последующие вхождения некоторой строки в сообщении заменяются ссылкой на её первое появление (рис. 84).

Рисунок 84. Иллюстрация идеи алгоритмов LZ

Семейство LZ включает в себя также LZW, LZSS, LZMA и др. алгоритмы.

Почти все современные программы-архиваторы (PKZip, LHA, ARJ и др.) используют ту или иную модификацию алгоритма LZ.

Алгоритм Лемпеля-Зива **LZ77** использует словарный метод сжатия. Словарь – уже просмотренная часть сообщения.

Очередной фрагмент сообщения заменяется на указатель в содержимое словаря:

• смещение назад от текущей позиции;

• длина совпадающей подстроки;

• первый символ после совпадения.

Составляющие алгоритма:

• принцип скользящего окна,

• механизм кодирования совпадений.

Рисунок 85. Иллюстрация работы скользящего окна в алгоритме LZ77

Принцип **скользящего окна** (рис. 85):

• В основе – динамический словарь – скользящее по сообщению окно = словарь + буфер (общей длиной 2, 4 или 32KB).

• Совпадения ищутся не на всём обработанном префиксе, а в его части – словаре, состоящем из последних уже обработанных символов.

• Буфер – небольшая порция ещё не закодированных символов входного потока (ограничивает длину кодируемой подстроки).

• При больших объёмах ввода алгоритм тратит меньше времени – просматривается не вся исходная строка.

• Слишком маленькая длина словаря приводит к совпадениям на большем расстоянии, чем длина словаря, и они не будут учтены, а, значит, кодирование менее оптимально.

кот_от_окон_отошел

Шаг	Скользящее окно		Совпадающая фраза	Закодированные данные		
	Словарь	Буфер		*i*	*j*	*s*
1	-	кот_от	-	0	0	"к"
2	к	от_от_о	-	0	0	"о"
3	ко	т_от_ок	-	0	0	"т"
4	кот	_от_око	-	0	0	"_"
5	кот_	от_окон	"от_"	3	3	"о"
6	кот_от_о	кон_ото	"ко"	8	2	"н"
7	кот_от_окон	_отошел	"_от"	7	3	"о"
8	кот_от_окон_ото	шел	-	0	0	"ш"
9	кот_от_окон_отош	ел	-	0	0	"е"
10	кот_от_окон_отоше	л	-	0	0	"л"
11	кот_от_окон_отошел	-	-	-1	-	-

<0,0,к> <0,0,о> <0,0,т> <0,0,_> <3,3,о> <8,2,н> <7,3,о> <0,0,ш> <0,0,е> <0,0,л> -1

Рисунок 86. Шаги кодирования совпадений в алгоритме LZ77

Механизм кодирования совпадений (рис. 86):

• Поиск самого длинного совпадения между строкой из буфера и всеми фразами словаря.

• Кодовая пара трактуется как команда копирования символов из словаря с определенной позиции.

Алгоритм LZ78 ориентируется на данные, которые только будут получены. Не использует «скользящее» окно, хранит словарь из уже просмотренных фраз (рис. 87).

ACAGAATAGAGA

Словарь Словарь	Считываемое содержимое	Код Код
	A	<0, A>
A = 1	C	<0, C>
A = 1 C = 2	AG	<1, G>
A = 1, C = 2 AG = 3	AA	<1, A>
A = 1, C = 2 AG = 3, AA = 4	T	<0, T>
A = 1, C = 2, AG = 3 AA = 4, T = 5	AGA	<3, A>
A = 1, C = 2, AG = 3 AA = 4, T = 5, AGA = 6	G	<0, G>
A = 1, C = 2, AG = 3, AA = 4 T = 5, AGA = 6, G = 7		<1, EOF>

<0, A><0, C><1, G><1, A><0, T><3, A><0, G><1, EOF>

Рисунок 87. Шаги кодирования совпадений в алгоритме LZ78

1. Считываются символы из кодируемого текста, пока накапливаемая подстрока входит целиком в одну из фраз словаря.

2. Как только считанная подстрока перестаёт соответствовать хотя бы одной фразе словаря, алгоритм генерирует код:

• индекс строки в словаре, которая до последнего введённого символа содержала подстроку,

• символ, нарушивший совпадение.

3. Затем в словарь добавляется введенная подстрока.

4. Если словарь уже заполнен, то из него предварительно удаляют менее

всех используемую в сравнениях фразу.

Сжатие битовой последовательности методом Лемпеля-Зива (рис. 88):

• Если в тексте появляется последовательность из двух ранее уже встречавшихся символов, то эта последовательсность объявляется новым символом, для нее назначается код, который (при определенных условиях) может быть короче исходной последовательности.

• Коды – это порядковые номера, начиная с 0 (очередной номер, равный числу уже использованных кодов).

Так, если в алфавите 8 символов, то их двоичные коды – от 0 до 111; тогда первая 2-символьная комбинация получит код 1000, следующая – 1001 и т.д.

• В сжатый текст на место исходной последовательности записывается её код.

• При декодировании повторяются аналогичные действия, поэтому становятся известными последовательности символов для каждого кода.

Исходный текст	00000001111111111111000000000011011110 0.00.000. 01. 11. 111. 1111. 110. 0000. 00000. 1101. 1110.
LZ-код	0.00.100.001. 011. 1011.1101.1010.00110.10010.10001.01100
R	2 3 4
Вводимые коды	- 10 11 100 101 110 111 1000 1001 1010 1011 1100

Рисунок 88. Иллюстрация кодирования битовой последовательности методом Лемпеля-Зива

Алгоритм кодирования битовой последовательности:

1. Выбирается первый символ сообщения и заменяется на его код (0).

2. Выбираются следующие 2 символа и заменяются своими кодами. Одновременно комбинации этих 2-х символов присваивается свой код.

3. Из исходного текста выбираются очередные 2,3,...,N символов, пока не образуется ещё не встречавшаяся комбинация.

Тогда этой комбинации присваивается очередной код, и поскольку совокупность из первых N-1 символов уже встречалась, то она имеет свой код, который и записывается вместо этих N-1 символов. Каждый акт введения нового кода – шаг кодирования.

4. Процесс продолжается до исчерпания исходного текста.

Задача – в определении R – требуемого количества двоичных разрядов для кодирования. R – количество разрядов в наиболее длинном коде.

В общем случае R=K после шага кодирования 2^{K-1}-1.

Контрольные вопросы

1. Что такое информация, данные, информационное сообщение, сигнал?

2. На чём основана идея сжатия данных в сообщении?
3. Сформулируйте основные отличия кодов Шеннона-Фано и Хаффмана.
4. В чём идея адаптированного кодирования Хаффмана?
5. Опишите идею словарного кодирования?
6. Что такое динамический словарь?
7. Что означает -5 в кодовой последовательности RLE?
8. На чём основана идея кодирования в алгоритмах Лемпеля-Зива?
9. Опишите шаги кодирования битовой последовательности методом Лемпеля-Зива.
10. Дайте характеристику формату сжатия данных Zip.

5. ПРАКТИЧЕСКИЕ ЗАДАНИЯ

5.1. Практическая работа №2.1

Тема: «Сортировка числового файла с помощью битового массива».

Цель. Поучить практический опыт по применению битовых операций.

Задание 1. Битовые операции в С++

Пример – как установить 5-й бит произвольного целого числа в 0 и что получится в результате:

unsigned char x=255; *//8-разрядное двоичное число 11111111*
unsigned char maska = 1; *//1=00000001 – 8-разрядная маска*
x = x & (~ (maska<<4)); *//результат x=239*

1.а. Реализуйте вышеприведённый пример, проверьте правильность результата в том числе и на других значениях x.

1.б. Реализуйте по аналогии с предыдущим примером установку 7-го бита числа в единицу.

1.в. Реализуйте код листинга 5.1, объясните выводимый программой результат.

Листинг 5.1.

```
1   //Битовые операции
2   #include <cstdlib>
3   #include <iostream>
4   #include <Windows.h>
5   #include <bitset>
6   using namespace std;
7
8   int main()
9   {
10      SetConsoleCP(1251);
11      SetConsoleOutputCP(1251);
12
13      unsigned int x = 25;
14      const int n = sizeof(int)*8; //=32 - количество разрядов в числе типа int
15      unsigned maska = (1 << n - 1);  //1 в старшем бите 32-разрядной сетки
16      cout << "Начальный вид маски: " << bitset<n> (maska) << endl;
17      cout << "Результат: ";
18      for (int i = 1; i <= n; i++)  //32 раза - по количеству разрядов:
19      {
20          cout << ((x & maska) >> (n - i));
21          maska = maska >> 1;   //смещение 1 в маске на разряд вправо
22      }
23      cout << endl;
24      system("pause");
25      return 0;
26  }
```

Задание 2. Сортировка числового множества битовым массивом.

Пусть даны не более 8 чисел со значениями от 0 до 7, например, {1, 0, 5, 7, 2, 4}.

Подобный набор чисел удобно отразить в виде 8-разрядной битовой по-

следовательности 11101101. В ней единичные биты показывают наличие в исходном наборе числа, равного номеру этого бита в последовательности (нумерация с 0 слева). Т.о. индексы единичных битов в битовом массиве – это и есть числа исходной последовательности.

Последовательное считывание бит этой последовательности и вывод индексов единичных битов позволит естественным образом получить исходный набор чисел в отсортированном виде – {0, 1, 2, 4, 5, 7}.

2.а. Реализуйте вышеописанный пример с вводом произвольного набора до 8-ми чисел (со значениями от 0 до 7) и его сортировкой битовым массивом в виде числа типа unsigned char. Проверьте работу программы.

2.б. Адаптируйте вышеприведённый пример для набора из 64-х чисел (со значениями от 0 до 63) с битовым массивом в виде числа типа unsigned long long.

2.в. Исправьте программу задания 2.б, чтобы для сортировки набора из 64-х чисел использовалось не одно число типа unsigned long long, а линейный массив чисел типа unsigned char.

Задание 3. Сортировка числового файла битовым массивом.

Постановка задачи:

Входные данные: файл, содержащий не более n=107 неотрицательных целых чисел[5], среди них нет повторяющихся.

Результат: упорядоченная по возрастанию последовательность исходных чисел в выходном файле.

Время работы программы: ~10 с (до 1 мин. для систем малой вычислительной мощности).

Максимально допустимый объём ОЗУ для хранения данных: 1 МБ.

Очевидно, что размер входных данных гарантированно превысит 1МБ (это, к примеру, максимально допустимый объём стека вызовов, используемого для статических массивов).

Требование по времени накладывает ограничение на количество чтений исходного файла.

3.а. Реализуйте задачу сортировки числового файла с заданными условиями. Добавьте в код возможность определения времени работы программы.

Примечание: содержимое входного файла должно быть сформировано неповторяющимися значениями заранее, это время не должно учитываться

[5] Для упрощения кода работы с файлами в текстовом режиме можно ограничиться только семизначными числами в диапазоне [1000000…9999999]; для работы с файлами в бинарном режиме это не актуально.

при замере времени сортировки.

В отчёт внесите результаты тестирования для наибольшего количества входных чисел, соответствующего битовому массиву длиной 1МБ.

3.б. Определите программно объём оперативной памяти, занимаемый битовым массивом.

Содержание отчёта:[6]

1. Титульный лист.
2. Цель работы.
3. Ход работы (по каждому заданию):
    a. Формулировка задачи.
    b. Математическая модель решения (описание алгоритма).
    c. Код программы с комментариями.
    d. Результаты тестирования.
4. Вывод (решены ли задачи, достигнута ли цель).

5.2. Практическая работа №2.2

Тема: «Алгоритмы поиска в таблице (массиве)».

Цель. Поучить практический опыт по применению алгоритмов поиска в таблицах данных.

Задание. Разработать программу поиска записей с заданным ключом в двоичном файле с применением различных алгоритмов.

Задание 1. Создать двоичный файл из записей (структура записи определена вариантом). Поле ключа записи в задании варианта подчеркнуто. Заполнить файл данными, используя для поля ключа датчик случайных чисел. Ключи записей в файле уникальны. Рекомендация: создайте сначала текстовый файл, а затем преобразуйте его в двоичный.

Задание 2. Поиск в файле с применением линейного поиска

Разработать программу поиска записи по ключу в бинарном файле с применением алгоритма линейного поиска.

Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

Составить таблицу с указанием результатов замера времени

Задание 3. Поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти.

---

[6] Указанную структуру отчёта рекомендуется соблюдать во всех последующих практических работах.

Для оптимизации поиска в файле создать в оперативной памяти структур данных – таблицу, содержащую ключ и ссылку (смещение) на запись в файле.

Разработать функцию, которая принимает на вход ключ и ищет в таблице элемент, содержащий ключ поиска, а возвращает ссылку на запись в файле. Алгоритм поиска определен в варианте.

Разработать функцию, которая принимает ссылку на запись в файле, считывает ее, применяя механизм прямого доступа к записям файла. Возвращает прочитанную запись как результат.

Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

Составить таблицу с указанием результатов замера времени.

Форма отчета:

Отчет по заданию 1

Постановка задачи

Описание подхода к решению

Определить структуру записи файла.

Определить размер записи в байтах.

Описать, как организуется прямой доступ к записям в бинарном (двоичном) файле.

Перечислить алгоритмы, которые реализуются в форме функций. Привести прототипы функций.

Код программы. Для функций указать предусловие и постусловие.

Выполните тестирование программы для 100 записей.

Отчет по заданию 2

Постановка задачи

Алгоритм

Приведите алгоритм линейного поиска записи с ключом в файле на псевдокоде.

Код функции поиска. Указать предусловие и постусловие. Включить код функции в код программы задания 1.

Код программы линейного поиска записи по ключу.

Результат тестирования программы для 100 записей.

Таблица с замерами времени поиска записи по заданному ключу для файла из 100 и 100 записей.

Отчет по заданию 3

Постановка задачи

Описание алгоритма доступа к записи в файле посредством таблицы. Что

определяет ссылка в таблице? Средства C++, которые используются для организации доступа к записи в файле по ссылке.

Алгоритм

Приведите алгоритм поиска, определенный вариантом, записи с ключом в файле на псевдокоде.

Код функции поиска. Указать предусловие и постусловие. Включить код функции в код программы задания 1.

Код программы линейного поиска записи по ключу.

Результат тестирования программы для 100 записей.

Таблица с замерами времени поиска записи по заданному ключу для файла из 100 и 100 записей.

Представить анализ эффективности рассмотренных алгоритмов поиска в файле.

Выводы.

Таблица 1. Варианты индивидуальных заданий

№ варианта		Задача
№	Алгоритм поиска	Структура записи файла (ключ – подчеркнутое поле)
1	Бинарный однородный без использования дополнительной таблицы	Читательский абонемент: номер читательского билета - целое пятизначное число, ФИО, Адрес
2	Бинарный поиск	Счет в банке: номер счета - 7 разрядное число, ФИО, Адрес
3	Бинарный однородный с использование таблицы смещений	Владелец телефона: номер телефона – последовательность символов, адрес
4	Фибоначчи поиск	Владельцев автомобилей. номер машины, марка, сведения о владельце.
5	Интерполяционный поиск	Пациент поликлиники: номер карточки, код хронического заболевания, Фамилия лечащего врача
6	Бинарный однородный без использования дополнительной таблицы	Товар: название, код – шестиразрядное число
7	Бинарный поиск	Специализация вуза: код специальности, название вуза
8	Бинарный однородный с использование таблицы смещений	Книга: ISBN – двенадцатизначное число, Автор, Название
9	Фибоначчи поиск	Страховой полис: номер полиса, компания, фамилия владельца
10	Интерполяционный поиск	Страхование автосредства: регистрационный номер – шестизначное число, название страховой компании

11	Бинарный однородный без использования дополнительной таблицы	Железнодорожная справка: номер поезда, пункт отправления, пункт назначения, время отправления
12	Бинарный поиск	Регистрация малого предприятия: номер лицензии, название, учредитель
13	Бинарный однородный с использование таблицы смещений	Студент: номер зачетной книжки, номер группы, ФИО
14	Фибоначчи поиск	Справочная межгорода: код города, название города
15	Интерполяционный поиск	Учет налогоплательщиков ИНН -10 - значное число, Фамилия, телефон
16	Фибоначчи поиск	Регистрация земельного участка в СНТ: кадастровый номер – семизначное число, адрес СНТ

5.3. Практическая работа №2.3

Тема: «Хеширование – прямой доступ к данным».

Цель: освоить приёмы хеширования и эффективного поиска элементов множества.

Задание:

Разработайте приложение, которое использует хеш-таблицу (пары «ключ – хеш») для организации прямого доступа к элементам динамического множества полезных данных. Множество реализуйте на массиве, структура элементов (перечень полей) которого приведена в индивидуальном варианте.

Приложение должно содержать класс с базовыми операциями: вставки, удаления, поиска по ключу, вывода. Включите в класс массив полезных данных и хеш-таблицу. Хеш-функцию подберите самостоятельно, используя правила выбора функции.

Реализуйте расширение размера таблицы и рехеширование, когда это требуется, в соответствии с типом разрешения коллизий.

Предусмотрите автоматическое заполнение таблицы 5-7 записями.

Реализуйте текстовый командный интерфейс пользователя для возможности вызова методов в любой произвольной последовательности, сопроводите вывод достаточными для понимания происходящего сторонним пользователем подсказками.

Проведите полное тестирование программы (все базовые операции, изменение размера и рехеширование), тест-примеры определите самостоятельно. Результаты тестирования включите в отчет по выполненной работе.

Примечание: тесты должны включать в себя случаи коллизий, проверке подлежит правильность вставки, поиска и удаления записей, вызвавших коллизию.

Оформите отчёт с подробным описанием созданного массива и хеш-таблицы, подходов к программной реализации базовых операций, описанием текста исходного кода и проведенного тестирования программы.

В отчёте сделайте вывод о проделанной работе, основанный на полученных результатах.

Содержание отчёта:

Титульный лист.

Цель работы.

Ход работы (по каждому заданию):

Формулировка задачи.

Математическая модель решения (описание алгоритма).

Код программы с комментариями.

Результаты тестирования.

Вывод (решены ли задачи, достигнута ли цель).

Таблица 2. Варианты индивидуальных заданий

Вариант	Метод хеширования (тип последовательностей проб)	Структура элемента множества. Ключи записей подчеркнуты
1	Цепное хеширование	Читательский абонемент: номер читательского - целое пятизначное число, ФИО, адрес
2	Цепное хеширование	Счет в банке: номер счета целое 7-зн.число, ФИО,адрес
3	Открытая адресация (двойное хеширование)	Студент: номер зачетной книжки, номер группы, ФИО
4	Открытая адресация (квадратичное пробирование)	Регистрация малого предприятия: номер лицензии, название, учредитель
5	Открытая адресация (двойное хеширование)	Товар: код – шестиразрядное число, название, цена
6	Открытая адресация (линейное пробирование)	Специализация вуза: код специальности – (прим.: "09.03.01"), название вуза
7	Открытая адресация (линейное пробирование)	Студент: номер зачетной книжки, номер группы, ФИО
8	Открытая адресация (квадратичное пробирование)	Читательский абонемент: номер читательского - целое пятизначное число, ФИО, адрес
9	Открытая адресация (двойное хеширование)	Читательский абонемент: номер читательского - целое пятизначное число, ФИО, адрес
10	Цепное хеширование	Регистрация малого предприятия: номер лицензии, название, учредитель
11	Открытая адресация (двойное хеширование)	Владелец телефона: номер телефона – последовательность 10 **символов**, адрес
12	Открытая адресация (линейное пробирование)	Страховой полис: номер, компания, фамилия владельца
13	Открытая адресация (двойное хеширование)	Счет в банке: номер счета целое семизначное число, ФИО, адрес

Продолжение табл. 2.

14	Открытая адресация (двойное хеширование)	Регистрация малого предприятия: номер лицензии, название, учредитель
15	Открытая адресация (линейное пробирование)	Счет в банке: номер счета целое семизначное число, ФИО, адрес
16	Открытая адресация (двойное хеширование)	Страховой полис: номер, компания, фамилия владельца
17	Цепное хеширование	Специализация вуза: код специальности – (прим.: "09.03.01"), название вуза
18	Открытая адресация (линейное пробирование)	Товар: код – шестиразрядное число, название, цена
19	Цепное хеширование	Книга: ISBN – 12-значное число, автор, название
20	Открытая адресация (линейное пробирование)	Книга: ISBN – двенадцатизначное число, автор, название
21	Открытая адресация (квадратичное пробирование)	Книга: ISBN – двенадцатизначное число, автор, название
22	Открытая адресация (квадратичное пробирование)	Счет в банке: номер счета целое семизначное число, ФИО, адрес
23	Открытая адресация (квадратичное пробирование)	Страховой полис: номер, компания, фамилия владельца
24	Цепное хеширование	Товар: код – шестиразрядное число, название, цена
25	Открытая адресация (линейное пробирование)	Владелец телефона: номер телефона – последовательность 10 **символов**, адрес
26	Открытая адресация (двойное хеширование)	Специализация вуза: код специальности – (прим.: "09.03.01"), название вуза
27	Цепное хеширование	Студент: номер зачетной книжки, номер группы, ФИО
28	Открытая адресация (квадратичное пробирование)	Владелец телефона: номер телефона – последовательность 10 **символов**, адрес
29	Открытая адресация (линейное пробирование)	Читательский абонемент: номер читательского - целое пятизначное число, ФИО, адрес
30	Открытая адресация (квадратичное пробирование)	Товар: код – шестиразрядное число, название, цена

5.4. Практическая работа №2.4

Тема: «Поиск образца в тексте».

Цель: освоить приёмы поиска образца в тексте.

Задание: Разработайте приложения в соответствии с заданиями в индивидуальном варианте.

В отчёте в разделе «Математическая модель решения (описание алгоритма)» разобрать алгоритм поиска на примере. Подсчитать количество срав-

нений для успешного поиска первого вхождения образца в текст и безуспешного поиска.

Определить функцию (или несколько функций) для реализации алгоритма поиска. Определить предусловие и постусловие.

Сформировать таблицу тестов с указанием успешного и неуспешного поиска, используя большие и небольшие по объему текст и образец, провести на её основе этап тестирования.

Оценить практическую сложность алгоритма в зависимости от длины текста и длины образца и отобразить результаты в таблицу (для отчета).

В отчёте сделайте вывод о проделанной работе, основанный на полученных результатах.

Содержание отчёта:

Титульный лист.

Цель работы.

Ход работы (по каждому заданию):

Формулировка задачи.

Математическая модель решения (описание алгоритма).

Код программы с комментариями.

Результаты тестирования.

Вывод (решены ли задачи, достигнута ли цель).

Таблица 3. Варианты индивидуальных заданий

Вариант	**Задачи варианта**
1	1. Линейный поиск первого вхождения подстроки в строку. 2. Используя алгоритм Бойера-Мура-Хорспула, найти последнее вхождение подстроки в строку.
2	1. Дано предложение, состоящее из слов. Сформировать массив слов – целых чисел. Словом считаем подстроку, ограниченную с двух сторон пробелами. 2. Найти все вхождения подстроки в строку, используя алгоритм Бойера-Мура (только эвристика хорошего суффикса).
3	1. Дано предложение, состоящее из слов. Найти самое длинное слово предложения, первая и последняя буквы которого одинаковы. 2. Используя алгоритм Кнута-Мориса-Пратта, найти индекс последнего вхождения образца в текст.
4	1. Дано предложение, состоящее из слов, разделенных знаками препинания. Определить, сколько раз в предложение входит первое слово. 2. Проверка на плагиат. Используя алгоритм Рабина-Карпа, проверить, входит ли подстрока проверяемого текста в другой текст.
5	1. Дано предложение, состоящее из слов, разделенных одним пробелом, удалить из него слова, встретившиеся более одного раза. 2. Дано предложение, состоящее из слов, разделенных одним пробелом. Удалить из предложения все вхождения заданного слова, применяя для поиска слова в тексте метод Кнута-Мориса-Пратта.

Продолжение табл. 3.

6	1. Дан произвольный текст, состоящий из слов, разделенных знаками препинания. Отредактировать его, оставив между словами по одному пробелу, а между предложениями по два. 2. Дана непустая строка S, длина которой N не превышает 10^6. Считать, что элементы строки нумеруются от 1 до N. Требуется для всех i от 1 до N вычислить π[i] – префикс функцию.
7	1. Дано предложение, состоящее из слов, разделенных знаками препинания. Определить количество слов равных последнему слову, больших последнего слова. 2. Строка S была записана много раз подряд, после чего из получившейся строки взяли произвольную часть строки - подстроку и передали как входные данные. Необходимо определить минимально возможную длину исходной строки S. Реализация алгоритмом Кнута-Мориса-Пратта.
8	1. Дано предложение, слова в котором разделены пробелами и запятыми. Распечатать те слова, которые являются обращениями других слов в этом предложении. 2. Даны две строки a и b. Требуется найти максимальную длину префикса строки a, который входит как подстрока в строку b. При этом считать, что пустая строка является подстрокой любой строки. Реализация алгоритмом Кнута-Мориса-Пратта.
9	1. Дано предложение, слова в котором разделены пробелами и запятыми. Распечатать те пары слов, расстояние между которыми наименьшее. Расстояние – это количество позиций, в которых слова различаются. Например, МАМА и ПАПА, МЫШКА и КОШКА расстояние этих пар равно двум. 2. Найти все вхождения подстроки в строку, используя алгоритм Бойера-Мура с турбосдвигом.
10	1. Дано предложение из слов, разделенных знаками препинания. Удалить из предложения все слова, равные заданному слову. 2. Назовем строку палиндромом, если она одинаково читается слева направо и справа налево. Примеры палиндромов: "abcba", "55", "q", "xyzzyx". Требуется для заданной строки найти максимальную по длине ее подстроку, являющуюся палиндромом. Реализация алгоритмом Кнута-Мориса-Пратта.
11	1. Дан текст, состоящий из слов, разделенных знаками препинания. Сформировать массив из слов, которые содержат заданную подстроку. 2. Назовем строку палиндромом, если она одинаково читается слева направо и справа налево. Примеры палиндромов: "abcba", "55", "q", "xyzzyx". Требуется для заданной строки найти максимальную по длине ее подстроку, являющуюся палиндромом. Реализация алгоритмом Бойера-Мура-Хорспула.
12	1. Дан текст, разделенных знаками препинания. Сформировать массив из слов, в которых заданная подстрока размещается с первой позиции. 2. В текстовом файле хранятся входные данные: на первой сроке – подстрока (образец)длиной не более 17 символов для поиска в тексте; со второй строки – текст (строка), в котором осуществляется поиск образца. **Строка**, в которой надо искать, **не ограниченна по длине.** Применяя алгоритм Бойера-Мура с турбосдвигом вывести индексы строки, на которые смещается алгоритм при поиске вхождения образца.

Продолжение табл. 3.

13	1. Дан текст, состоящий из слов, разделенных знаками препинания. Сформировать массив из слов, в которых заданная подстрока размещается в конце слова. 2. В текстовом файле хранятся входные данные: на первой сроке – подстрока (образец)длиной не более 17 символов для поиска в тексте; со второй строки – текст (строка), в котором осуществляется поиск образца. **Строка**, в которой надо искать, **не ограниченна по длине.** Применяя алгоритм Рабина-Карпа определить количество вхождений в текст заданного образца.
14	1. Дан текст, состоящий из слов, разделенных знаками препинания. Переставить первое и последнее слово в тексте. 2. Дан текст и множество подстрок образцов. Определить сколько раз каждый из образцов входит в исходный текст. Реализовать на алгоритме Рабина-Карпа. Примечание: для всех образцов создать хеш-таблицу.
15	1. Дан массив ключевых слов языка C++. Упорядочить их, располагая слова в алфавитном порядке, используя обменную сортировку. 2. Дан текст и множество подстрок образцов. Определить сколько раз каждый из образцов входит в исходный текст. Реализовать алгоритм Бойера-Мура-Хорспула. Примечание. Для всех образцов создать хеш-таблицу.

5.5. Практическая работа №2.5

Тема: «Бинарное дерево поиска. AVL-дерево. Красно-чёрное дерево».

Цель: освоить приёмы работы с нелинейными списками.

Задача:

Составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту.

Процедуры оформить в виде самостоятельных режимов работы созданного дерева. Выбор режимов производить с помощью пользовательского (иерархического ниспадающего) меню.

Провести полное тестирование программы на дереве размером n=10 элементов, сформированном вводом с клавиатуры. Тест-примеры определить самостоятельно. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе.

Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием созданного дерева, принципов программной реализации алгоритмов работы с деревом, описанием текста исходного кода и проведенного тестирования программы.

Таблица 4. Варианты индивидуальных заданий

Вариант	Тип значения узла	Тип дерева	Реализовать алгоритмы								
			Вставка элемента	Прямой обход	Обратный обход	Симметричный обход	Обход в ширину	Найти сумму значений листьев	Найти среднее арифметическое всех узлов	Найти длину пути от корня до заданного значения	Найти высоту дерева
1	Строка – имя	Красно-чёрное дерево	+ (и балансировка)			+	+	+			+
2	Целое	АВЛ-дерево	+ (и балансировка)			+				+	+
3	Символ	Красно-чёрное дерево	+ (и балансировка)	+		+			+	+	
4	Символ	Красно-чёрное дерево	+ (и балансировка)			+		+	+		
5	Символ	АВЛ-дерево	+ (и балансировка)			+	+	+			+
6	Вещественное	АВЛ-дерево	+ (и балансировка)		+	+		+	+		
7	Строка – имя	Бинарное дерево поиска	+		+	+				+	+
8	Символ	Бинарное дерево поиска	+			+		+			+
9	Целое	АВЛ-дерево	+ (и балансировка)		+	+			+	+	

Продолжение табл. 4.

10	Строка – имя	Красно-чёрное дерево	+ (и балан-сировка)		+	+				+	+
11	Веще-ственное	Красно-чёрное дерево	+ (и балан-сировка)			+		+	+		
12	Строка – город	Бинарное дерево поиска	+			+	+			+	+
13	Строка – город	Бинарное дерево поиска	+			+	+		+	+	
14	Целое	АВЛ-де-рево	+ (и балан-сировка)		+	+		+			+
15	Целое	Красно-чёрное дерево	+ (и балан-сировка)			+	+	+	+		
16	Веще-ственное	Бинарное дерево поиска	+			+		+	+		
17	Строка – город	Красно-чёрное дерево	+ (и балан-сировка)	+		+		+			+
18	Веще-ственное	АВЛ-де-рево	+ (и балан-сировка)			+		+	+		
19	Строка – город	АВЛ-де-рево	+ (и балан-сировка)		+	+				+	+
20	Символ	Бинарное дерево поиска	+	+		+			+	+	

Продолжение табл. 4.

21	Строка – имя	АВЛ-дерево	+ (и балансировка)			+	+	+			+
22	Строка – имя	Бинарное дерево поиска	+	+		+				+	+
23	Вещественное	Красно-чёрное дерево	+ (и балансировка)	+		+		+	+		
24	Строка – город	Бинарное дерево поиска	+	+		+			+	+	
25	Вещественное	Бинарное дерево поиска	+			+			+	+	
26	Строка – город	Красно-чёрное дерево	+ (и балансировка)	+		+				+	+
27	Символ	Бинарное дерево поиска	+		+	+		+			+
28	Целое	Красно-чёрное дерево	+ (и балансировка)			+	+		+	+	
29	Целое	АВЛ-дерево	+ (и балансировка)	+		+			+	+	
30	Строка – имя	АВЛ-дерево	+ (и балансировка)		+	+		+	+		

5.6. Практическая работа №2.6

Тема: «Основные алгоритмы работы с графами».

Цель: освоить приёмы работы с графовыми структурами.

Задача:

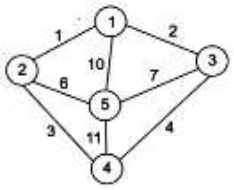
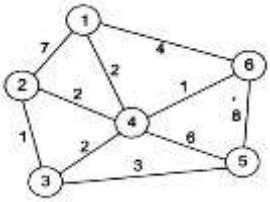
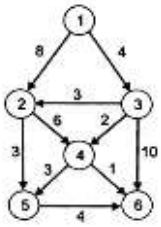
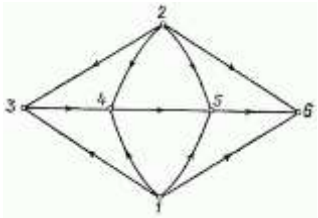
Составить программу создания графа и реализовать процедуру для работы с графом, определенную индивидуальным вариантом задания. Самостоятельно выбрать и реализовать способ представления графа в памяти.

В программе предусмотреть ввод с клавиатуры произвольного графа. В вариантах построения остовного дерева также разработать доступный способ (форму) вывода результирующего дерева на экран монитора.

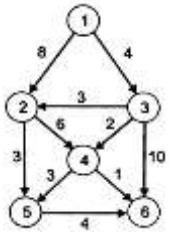
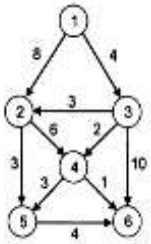
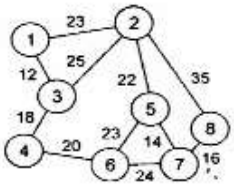
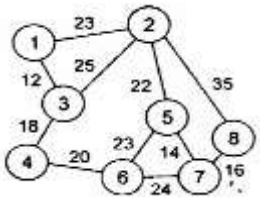
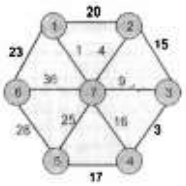
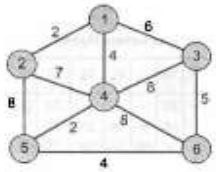
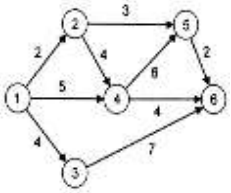
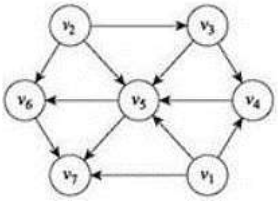
Провести тестовый прогон программы на предложенном в индивидуальном варианте задания графе. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе. Сделать выводы о проделанной работе, основанные на полученных результатах.

Оформить отчет с подробным описанием рассматриваемого графа, принципов программной реализации алгоритмов работы с графом, описанием текста исходного кода и проведенного тестирования программы.

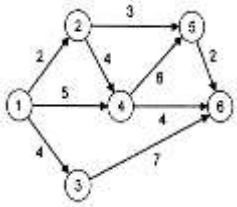
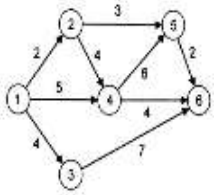
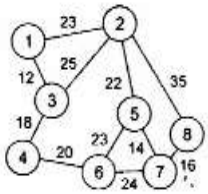
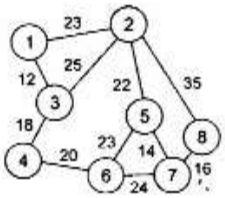
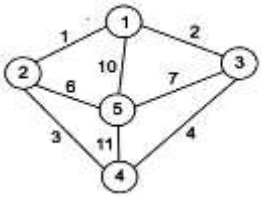
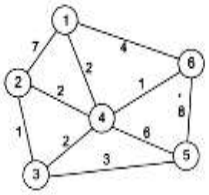
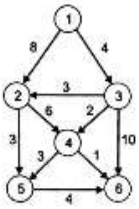
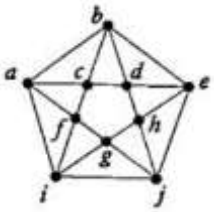
Таблица 5. Варианты индивидуальных заданий

Вариант	Алгоритм	Предложенный граф
1	Построение остовного дерева алгоритмом Крускала	
2	Построение остовного дерева алгоритмом Прима	
3	Нахождение кратчайшего пути методом построения дерева решений	
4	Определить, является ли граф связным, является ли граф ациклическим.	

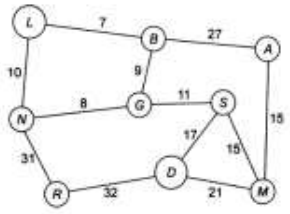
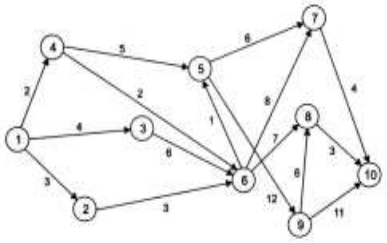
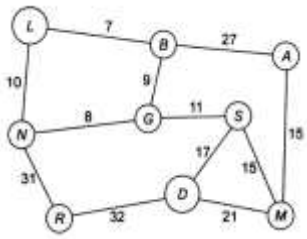
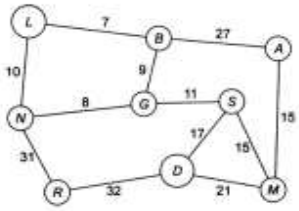
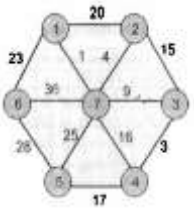
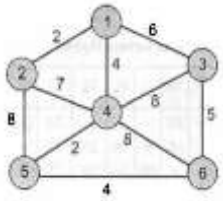
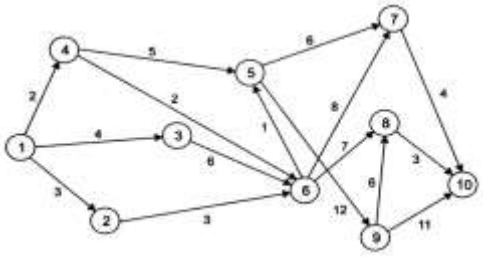
Продолжение табл. 5.

5	Нахождение кратчайшего пути методом Дейкстра	
6	Нахождение кратчайшего пути методом Флойда	
7	Нахождение кратчайшего пути методом Йена	
8	Нахождение кратчайшего пути методом Беллмана-Форда	
9	Построение остовного дерева алгоритмом Крускала	
10	Построение остовного дерева алгоритмом Прима	
11	Нахождение кратчайшего пути методом построения дерева решений	
12	Определить, является ли граф связным, является ли граф ациклическим.	

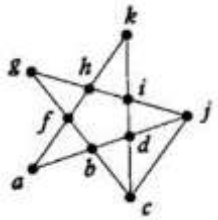
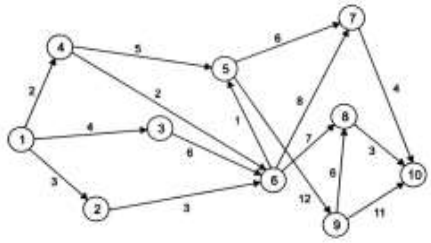
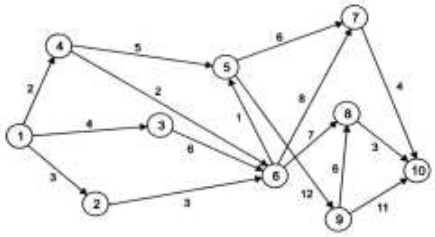
Продолжение табл. 5.

13	Нахождение кратчайшего пути методом Дейкстра	
14	Нахождение кратчайшего пути методом Флойда	
15	Нахождение кратчайшего пути методом Йена	
16	Нахождение кратчайшего пути методом Беллмана-Форда	
17	Построение остовного дерева алгоритмом Крускала	
18	Построение остовного дерева алгоритмом Прима	
19	Нахождение кратчайшего пути методом построения дерева решений	
20	Найти и вывести эйлеров цикл в графе. Реализовать обход графа в ширину. ~~Нахождение кратчайшего пути методом естественного слияния~~	

Продолжение табл. 5.

21	Нахождение кратчайшего пути методом Дейкстра	
22	Нахождение кратчайшего пути методом Флойда	
23	Нахождение кратчайшего пути методом Йена	
24	Нахождение кратчайшего пути методом Беллмана-Форда	
25	Построение остовного дерева алгоритмом Крускала	
26	Построение остовного дерева алгоритмом Прима	
27	Нахождение кратчайшего пути методом построения дерева решений	

Продолжение табл. 5.

28	Найти и вывести эйлеров путь в графе. Реализовать обход графа в ширину.	
29	Нахождение кратчайшего пути методом Дейкстра	
30	Нахождение кратчайшего пути методом Флойда	

5.7. Практическая работа №2.7

Тема: «Кодирование и сжатие данных без потерь».

Цель: освоить алгоритмы кодирования и сжатия данных без потерь.

Задание 1: Исследование алгоритмов сжатия на примерах

Выполнить каждую задачу варианта, представив алгоритм решения в виде таблицы и указав результат сжатия.

Описать процесс восстановления сжатого текста.

Сформировать отчет, включив задание, вариант задания, результаты выполнения задания варианта.

Задание 2: Разработать программы сжатия и восстановления текста методами Хаффмана и Шеннона – Фано.

Реализовать и отладить программы.

Сформировать отчет по разработке каждой программы в соответствии с требованиями.

По методу Шеннона-Фано привести: постановку задачи, описать алгоритм формирования префиксного дерева и алгоритм кодирования, декодирования, код и результаты тестирования. Рассчитать коэффициент сжатия. Сравнить с результат сжатия вашим алгоритмом с результатом людого архиватора.

По методу Хаффмана выполнить и отобразить результаты выполнения всех

требований, предъявленных в задании и оформить разработку программы: постановка, подход к решению, код, результаты тестирования.

Таблица 6. Варианты индивидуальных заданий

Вариант	Закодировать фразу методами Шеннона–Фано	Сжатие данных по методу Лемпеля– Зива LZ77 Используя двухсимвольный алфавит (0, 1) закодировать следующую фразу:	Закодировать следующую фразу, используя код LZ78
1	Ана, дэус, рики, паки, Дормы кормы констунтаки, Дэус дэус канадэус – бац!	0001010010101001101	кукуркукурекурекун
2	One, two, Freddy's coming for you Three, four, better lock your door Five, six, grab a cruci-fix Seven, eight, gonna stay up late.	0100100010010000101	упупапекапекаупуп
3	Эне-бене, рики-таки, Буль-буль-буль, Караки-шмаки Эус-деус-краснодеус бац	0100101010010000101	лорлоралоранранлоран
4	Кони-кони, коникони, Мы сидели на балконе, Чай пили, воду пили, По-турецки говорили.	0100001000000100001	пропронепронепрнепрона с
5	Прибавь к ослиной голове Еще одну, получишь две. Но сколько б ни было ослов, Они и двух не свяжут слов.	101000100101010001011	какатанекатанекатата
6	По-турецки говорили. Чяби, чяряби Чяряби, чяби-чяби. Мы набрали в рот воды.	0001011101101001 11	менменаменаменатеп
7	Тише, мыши, кот на крыше, А котята ещё выше. Кот пошёл за молоком, А котята кувырком.	11010101100110000100 1	долделдолдилделдил

Продолжение табл. 6.

8	Мой котёнок очень странный, Он не хочет есть сметану, К молоку не прикасался И от рыбки отказался.	01011011011010001000 1	sarsalsarsanlasanl 33
9	Эни-бени рити-Фати. Дорба, дорба сентибрати. Дэл. Дэл. Кошка. Дэл. Фати!	00010010110010001000 1	kloklonkolonklonkl
10	Самолёт-вертолёт! Посади меня в полёт! А в полёте пусто – Выросла капуста.	1110100110110001101	tertrektekertektrek
11	Кот пошёл за молоком, А котята кувырком. Кот пришёл без молока, А котята ха-ха-ха.	10101001101100111010	bigbonebigborebigbo
12	Цветом мой зайчишка – белый, А ещё, он очень смелый! Не боится он лисицы, Льва он тоже не боится.	0001001010101001101	commercommecommerce
13	Эне, бене, лики, паки, Цуль, буль-буль, Калики-цваки, Эус-беус, кликмадеус, бокс...	01011011001010101011	webwerbweberweberweb
14	Ана-дэус-рики-паки, Дормы-кормыконсту-таки, Энус-дэус-кана-дэусБАЦ!	0010100110010000001	porpoterpoterporter
15	Раз, два – упала гора; три, четыре – прицепило; пять, шесть – бьют шерсть; семь, восемь – сено косим.	10110111100110001101	mantopmentopomantomen
16	Зуба зуба, зуба зуба, Зуба дони дони мэ, А шарли буба раз два три, А ми раз два три замри.	0100101010010000101	roporopoterropoterter
17	Плыл по морю чемодан, В чемодане был диван, На диване ехал слон. Кто не верит – выйди вон!	0001000010101001101	webwerbweberweberweb
18	Дрынцы-брынцыбубен-цы, Раз-звонилисьудальцы, Диги-диги-диги-дон, Выхо-ди-скорее-вон!	1110100110111001101	sionsinossionsinos

Продолжение табл. 6.

19	Перводан, другодан, На колоде барабан; Свистель, коростель, Пятерка, шестерка, утюг.	0001000010101001101	comconcomconacom
20	Эни бэни рики паки Турбаурбасентибряки. Может – выйдет, может – нет, В общем – полный Интернет	0100101010010000101	mantopmentopomantomen

Требования к выполнению задания 2

Разработать алгоритм и реализовать программу сжатия текста алгоритмом Шеннона – Фано. Разработать алгоритм и программу восстановления сжатого текста. Выполнить тестирование программы на текстовом файле. Определить процент сжатия.

Провести кодирование(сжатие) исходной строки символов «Фамилия Имя Отчество» с использованием алгоритма Хаффмана. Исходная строка символов, таким образом, определяет индивидуальный вариант задания для каждого студента.

Для выполнения работы необходимо выполнить следующие действия:

Построить таблицу частот встречаемости символов в исходной строке символов для чего сформировать алфавит исходной строки и посчитать количество вхождений (частот) символов и их вероятности появления, например, для строки "пупкин василий кириллович" такая таблица будет иметь вид: Таблица частот:

Алфавит	п	у	к	и	н	« »	в
Кол. вх.	2	1	2	6	1	2	2
Вероятн.	0.08	0.04	0.08	0.24	0.04	0.08	0.08
Алфавит	а	с	л	й	р	о	ч
Кол. вх.	1	1	3	1	1	1	1
Вероятн.	0.04	0.04	0.12	0.04	0.04	0.04	0.04

(скобки < > обозначают пробел в исходной строке)

Отсортировать алфавит в порядке убывания частот появления символов по аналогии как показано ниже.

Таблица отсортированных частот:

Алфавит	и	л	п	к	« »	в	у
Кол. вх.	6	3	2	2	2	2	1
Вероятн.	0.24	0.12	0.08	0.08	0.08	0.08	0.04
Алфавит	н	а	с	й	р	о	ч
Кол. вх.	1	1	1	1	1	1	1
Вероятн.	0.04	0.04	0.04	0.04	0.04	0.04	0.04

Построить дерево кодирования Хаффмана, в данном примере оно имеет вид:

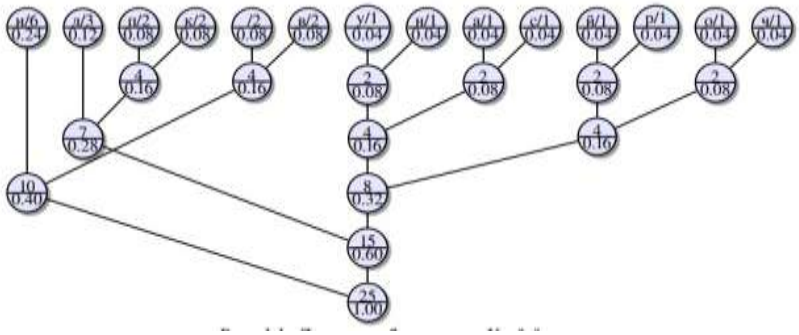


Рисунок 89. Дерево кодирования Хаффмана

Упорядочить построенное дерево слева-направо (при необходимости).

Присвоить ветвям коды.

Определить коды символов:

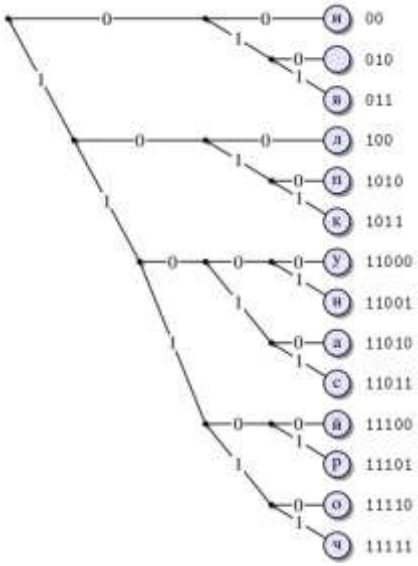


Рисунок 90. Упорядоченное дерево кодирования Хаффмана

Провести кодирование исходной строки по аналогии с примером:

```
 п     у     п    к   и    н    « »   в    а     с    и   л   и    й
1010 11000 1010 1011 00 11001 010 011 11010 11011 00 100 00 11100
« »   к    и    р    и   л    л    о    в    и    ч
010 1011 00 11101 00 100 100 11110 011 00 11111
```

Рассчитать коэффициенты сжатия относительно кодировки ASCII и относительно равномерного кода.

Рассчитать среднюю длину полученного кода и его дисперсию.

По результатам выполненной работы сделать выводы и сформировать отчет.

Применить алгоритм Хаффмана для архивации данных текстового файла. Выполнить практическую оценку сложности алгоритма Хаффмана. Провести архивацию этого же файла любым архиватором. Сравнить коэффициенты сжатия разработанного алгоритма и архиватора.

5.8. Практическая работа №2.8

Тема: «Алгоритмические стратегии».

Цель: освоить приёмы сокращения числа переборов.

Задание:

Разработать алгоритм решения задачи с применением метода, указанного в варианте и реализовать программу.

Оценить количество переборов при решении задачи стратегией «в лоб» - грубой силы. Сравнить с числом переборов при применении метода.

Оформить отчет в соответствии с требованиями документирования разработки ПО: Постановка задачи, Описание алгоритмов и подхода к решению, Код, результаты тестирования, Вывод.

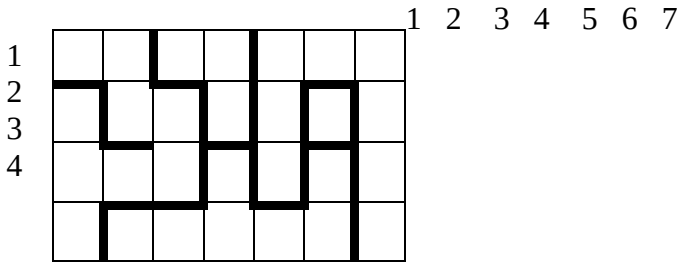
Таблица 7. Варианты индивидуальных заданий

№_	**Задача**	**Метод**
1	Посчитать число последовательностей нулей и единиц длины *n*, в которых не встречаются две идущие подряд единицы.	Динамическое программирование
2	Дана последовательность целых чисел. Необходимо найти ее самую длинную строго возрастающую подпоследовательность.	Динамическое программирование
3	Дана строка из заглавных букв латинского алфавита. Найти длину наибольшего палиндрома, который можно получить вычеркиванием некоторых букв из данной строки.	Динамическое программирование
4	Имеется рюкзак с ограниченной вместимостью по массе; также имеется набор вещей с определенным весом и ценностью. Необходимо подобрать такой набор вещей, чтобы он помещался в рюкзаке и имел максимальную ценность (стоимость).	Динамическое программирование
5	Дано прямоугольное поле размером *n***m* клеток. Можно совершать шаги длиной в одну клетку вправо или вниз. Посчитать, сколькими способами можно попасть из левой верхней клетки в правую нижнюю.	Динамическое программирование
6	Дано прямоугольное поле размером *n***m* клеток. Можно совершать шаги длиной в одну клетку вправо, вниз или по диагонали вправо-вниз. В каждой клетке записано некоторое натуральное число. Необходимо попасть из верхней левой клетки в правую нижнюю. Вес маршрута – это сумма чисел всех посещенных клеток. Найти маршрут с минимальным весом.	Динамическое программирование

Продолжение табл. 7.

7	Вычисление значения определенного интеграла с применением численных методов. «Вычислить значение определенного интеграла с заданной точностью определенным методом трапеции. Реализовать следующие подзадачи в виде функций: вычисление значения подинтегральной функции в заданной точке x; вычисление значения интеграла установленным методом на заданном отрезке интегрирования при n разбиениях; вычисление интеграла установленным методом с заданной точностью.	Динамическое программирование
8	Черепашке нужно попасть из пункта А в пункт В. Поле движения разбито на квадраты. Известно время движения вверх и вправо в каждой клетке (улицы). На каждом углу она может поворачивать только на север или только на восток. Найти минимальное время, за которое черепашка может попасть из А в В.	Динамическое программирование
9	Треугольник имеет вид, представленный на рисунке. Напишите программу, которая вычисляет наибольшую сумму чисел, расположенных на пути от верхней точки треугольника до его основания. 7 3 8 8 1 0 2 7 4 4 1 5 2 6 5	Динамическое программирование
10	Из листа клетчатой бумаги вырезали фигуру точно по границам клеток. Разработать программу вычисления площади вырезанной фигуры.	метод ветвей и границ
11	Разработать программу расстановки на 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого».	метод ветвей и границ
12	Разработать программу поиска и вывода всех гамильтоновых циклов в произвольном графе.	метод ветвей и границ
13	Пронумеровать позиции в матрице размером 5*5 следующим образом: если номер i ($1 \le i \le 25$) соответствует позиции (x,y), то номер i+1 может соответствовать позиции с координатами (z,w), вычисляемыми по одному из следующих правил: 1) (z,w)=(x±3,y) 2) (z,w)=(x,y±3) 3) (z,w)=(x±2,y±2) 1) Написать программу, которая последовательно нумерует позиции матрицы при заданных координатах позиции, в которой содержится номер 1. 2) Вычислить число всех возможных расстановок номеров для всех начальных позиций, расположенных под главной диагональю.	метод ветвей и границ

Продолжение табл. 7.

14	Замок имеет прямоугольную форму и разделен на M*N клеток (M<=50; N>=50). Каждая клетка может иметь от 0 до 4 стен, отделяющих комнаты. Определить: количество комнат в замке; площадь наибольшей комнаты; какую стену следует удалить, чтобы получить комнату наибольшей площади. Пример плана замка: 	метод ветвей и границ
15	Автозаправка. Вдоль кольцевой дороги расположено M городов. В каждом городе есть автозаправка. Известна стоимость Z[i] заправки горючим в городе с номером i b стоимость C[i] проезда по дороге, соединяющей i-ый и (i+1)-й города и стоимость проезда между первым и M-ым городами. Города пронумерованы по часовой стрелке. Определить для жителей каждого города тот город в котором им выгодно заправляться, и направление «по часовой стрелке» или «против часовой стрелки»	метод ветвей и границ
16	В массиве размером M×N, заполненном нулями и единицами найти квадратный блок, состоящий из одних нулей.	метод ветвей и границ
17	Монетная система некоторого государства состоит из монет достоинством $a_1 = 1 < a_2 < ... < a_n$. Требуется выдать сумму наименьшим возможным количеством монет.	Жадный алгоритм
18	Разработать процедуру оптимального способа расстановки скобок в произведении последовательности матриц, размеры которых равны (5,10,3,12,5,50,6), чтобы количество скалярных умножений стало минимальным (максимальным).	Жадный алгоритм
19	Решить задачу о раскраске вершин графа. Применить к задаче управления светофорами на сложном перекрестке. (См. Ахо А., Хопкрофт Д., Ульман Дж. Структуры данных и алгоритмы).	Жадный алгоритм
20	Задача о коммивояжере	метод ветвей и границ

ЛИТЕРАТУРА

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.
2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.
3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.
4. Кораблин Ю.П. Структуры и алгоритмы обработки данных : учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.
5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1328 с.
6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.
7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.
8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.
9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.
10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).
11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).
12. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных [Электронный ресурс]. URL: https://intuit.ru/studies/courses/3496/738/info (дата обращения 15.03.2022).

Сведения об авторах:

1. **Рысин Михаил Леонидович**, к.п.н., доцент, преподаёт на кафедре математического обеспечения и стандартизации информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет», автор более 60 научных и учебно-методических работ;
2. **Сартаков Михаил Валерьевич**, к.т.н., доцент кафедры математического обеспечения и стандартизации информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет», автор более 40 научных и учебно-методических работ;
3. **Туманова Марина Борисовна**, к.п.н., доцент, преподаёт на кафедре математического обеспечения и стандартизации информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет», автор более 60 научных и учебно-методических работ.