В современном стремительно развивающемся информационном обществе достаточно остро стоит вопрос хранения и передачи информации. Несмотря на непрерывно возрастающий накопительный объем информационных носителей, порой требуется сохранить большое количество данных на хранилище небольшой емкости (например, флэш-накопителе). Для уменьшения размеров применяются особые алгоритмы - так называемые алгоритмы сжатия. Сжатие сокращает объем пространства, требуемого для хранения файлов в ЭВМ, и количество времени, необходимого для передачи информации по каналу установленной ширины пропускания. Это есть форма кодирования. Другими целями кодирования являются поиск и исправление ошибок, а также шифрование. Процесс поиска и исправления ошибок противоположен сжатию - он увеличивает избыточность данных, когда их не нужно представлять в удобной для восприятия человеком форме. Удаляя из текста избыточность, сжатие способствует шифрованию, что затрудняет поиск шифра доступным для взломщика статистическим методом. Существует достаточно большое количество их всевозможных вариаций. В данной работе будут рассмотрены основные сведения об архивации и типах сжатия, а так же - приведены реализация алгоритма LZ77 на языке программирования C++. Понятие архивации Сжатие данных (также известное, как архивация) - алгоритмическое преобразование данных, производимое с целью уменьшения их объёма. Применяется для более рационального использования устройств хранения и передачи данных. Синонимы - упаковка данных, компрессия, сжимающее кодирование, кодирование источника. Обратная процедура называется восстановлением данных (распаковкой, декомпрессией). Сжатие основано на устранении избыточности, содержащейся в исходных данных. Простейшим примером избыточности является повторение в тексте фрагментов (например, слов естественного или машинного языка). Подобная избыточность обычно устраняется заменой повторяющейся последовательности ссылкой на уже закодированный фрагмент с указанием его длины. Другой вид избыточности связан с тем, что некоторые значения в сжимаемых данных встречаются чаще других. Сокращение объёма данных достигается за счёт замены часто встречающихся данных короткими кодовыми словами, а редких - длинными (энтропийное кодирование). Сжатие данных, не обладающих свойством избыточности (например, случайный сигнал или белый шум, зашифрованные сообщения), принципиально невозможно без потерь. В основе любого способа сжатия лежит модель источника данных, или,

точнее, модель избыточности. Иными словами, для сжатия данных используются некоторые априорные сведения о том, какого рода данные сжимаются. Не обладая такими сведениями об источнике, невозможно сделать никаких предположений о преобразовании, которое позволило бы уменьшить объём сообщения. Модель избыточности может быть статической, неизменной для всего сжимаемого сообщения, либо строиться или параметризоваться на этапе сжатия (и восстановления). Методы, позволяющие на основе входных данных изменять модель избыточности информации, называются адаптивными. Неадаптивными являются обычно узкоспециализированные алгоритмы, применяемые для работы с данными, обладающими хорошо определёнными и неизменными характеристиками. Подавляющая часть достаточно универсальных алгоритмов являются в той или иной мере адаптивными. Все методы сжатия данных делятся на два основных класса: . Сжатие без потерь . Сжатие с потерями При использовании сжатия без потерь возможно полное восстановление исходных данных, сжатие с потерями позволяет восстановить данные с искажениями, обычно несущественными с точки зрения дальнейшего использования восстановленных данных. Сжатие без потерь обычно используется для передачи и хранения текстовых данных, компьютерных программ, реже - для сокращения объёма аудио- и видеоданных, цифровых фотографий и т. п., в случаях, когда искажения недопустимы или нежелательны. Сжатие с потерями, обладающее значительно большей, чем сжатие без потерь, эффективностью, обычно применяется для сокращения объёма аудио- и видеоданных и цифровых фотографий в тех случаях, когда такое сокращение является приоритетным, а полное соответствие исходных и восстановленных данных не требуется. Основной характеристикой алгоритма сжатия является коэффициент сжатия. Он определяется как отношение объёма исходных несжатых данных к объёму сжатых, то есть: k=So/Sc , где k - коэффициент сжатия, So - объём исходных данных, а Sc - объём сжатых. Таким образом, чем выше коэффициент сжатия, тем алгоритм эффективнее. Следует отметить: · если k = 1, то алгоритм не производит сжатия, то есть выходное сообщение оказывается по объёму равным входному; · если k < 1, то алгоритм порождает сообщение большего размера, нежели несжатое, то есть, совершает «вредную» работу. Ситуация с k < 1 вполне возможна при сжатии. Принципиально невозможно получить алгоритм сжатия без потерь, который при любых данных образовывал бы на выходе данные меньшей или равной длины. Обоснование этого факта заключается в том, что поскольку число различных сообщений длиной n бит составляет ровно 2n, число различных сообщений с длиной меньшей или равной n (при наличии хотя бы одного сообщения меньшей длины) будет меньше 2n. Это значит, что невозможно

однозначно сопоставить все исходные сообщения сжатым: либо некоторые исходные сообщения не будут иметь сжатого представления, либо нескольким исходным сообщениям будет соответствовать одно и то же сжатое, а значит их нельзя отличить. Но даже когда алгоритм сжатия увеличивает размер исходных данных, легко добиться того, чтобы их объём гарантировано не мог увеличиться более, чем на 1 бит. То есть сделать так, чтобы даже в самом худшем случае имело место неравенство: =So/(So+1) Делается это следующим образом: если объём сжатых данных меньше объёма исходных, возвращаются сжатые данные путем добавления к ним «1», иначе возвращаем исходные данные, добавив к ним «0». Коэффициент сжатия может быть как постоянным (некоторые алгоритмы сжатия звука, изображения и т. п., например A-закон, μ-закон, ADPCM, усечённое блочное кодирование), так и переменным. Во втором случае он может быть определён либо для каждого конкретного сообщения, либо оценён по некоторым критериям: · средний (обычно по некоторому тестовому набору данных); · максимальный (случай наилучшего сжатия); · минимальный (случай наихудшего сжатия); или каким-либо другим. Коэффициент сжатия с потерями при этом сильно зависит от допустимой погрешности сжатия или качества, которое обычно выступает как параметр алгоритма. В общем случае постоянный коэффициент сжатия способны обеспечить только методы сжатия данных с потерями. Основным критерием различия между алгоритмами сжатия является описанное выше наличие или отсутствие потерь. В общем случае алгоритмы сжатия без потерь универсальны в том смысле, что их применение безусловно возможно для данных любого типа, в то время как возможность применения сжатия с потерями должна быть обоснована. Для некоторых типов данных искажения не допустимы в принципе. В их числе символические данные, изменение которых неминуемо приводит к изменению их семантики: программы и их исходные тексты, двоичные массивы и т. п.; жизненно важные данные, изменения в которых могут привести к критическим ошибкам: например, получаемые с медицинской измерительной аппаратуры или контрольных приборов летательных, космических аппаратов и т. п.; многократно подвергаемые сжатию и восстановлению промежуточные данные при многоэтапной обработке графических, звуковых и видеоданных. Различные алгоритмы могут требовать различного количества ресурсов вычислительной системы, на которых они реализованы: · оперативной памяти (под промежуточные данные); · постоянной памяти (под код программы и константы); · процессорного времени. В целом, эти требования зависят от сложности и «интеллектуальности» алгоритма. Общая тенденция такова: чем эффективнее и универсальнее алгоритм, тем большие требования к вычислительным ресурсам он предъявляет. Тем не менее, в специфических случаях

простые и компактные алгоритмы могут работать не хуже сложных и универсальных. Системные требования определяют их потребительские качества: чем менее требователен алгоритм, тем на более простой, а следовательно, компактной, надёжной и дешёвой системе он может быть реализован. Так как алгоритмы сжатия и восстановления работают в паре, имеет значение соотношение системных требований к ним. Нередко можно усложнив один алгоритм значительно упростить другой. Таким образом, возможны три варианта: Алгоритм сжатия требует больших вычислительных ресурсов, нежели алгоритм восстановления. Это наиболее распространённое соотношение, характерное для случаев, когда однократно сжатые данные будут использоваться многократно. В качестве примера можно привести цифровые аудио- и видеопроигрыватели. Алгоритмы сжатия и восстановления требуют приблизительно равных вычислительных ресурсов. Наиболее приемлемый вариант для линий связи, когда сжатие и восстановление происходит однократно на двух её концах (например, в цифровой телефонии). Алгоритм сжатия существенно менее требователен, чем алгоритм восстановления. Такая ситуация характерна для случаев, когда процедура сжатия реализуется простым, часто портативным устройством, для которого объём доступных ресурсов весьма критичен, например, космический аппарат или большая распределённая сеть датчиков. Это могут быть также данные, распаковка которых требуется в очень малом проценте случаев, например запись камер видеонаблюдения. . Описание и особенности некоторых алгоритмов архивации Рассмотрим следующие алгоритмы сжатия: . RLE . коды Хаффмана . метод «стопки книг» (MTF - Move to Font) . арифметическое кодирование . вероятностное сжатие . LZ77 1) RLE Алгоритм RLE (Run Length Encoding, упаковка, кодирование длин серий), является самым быстрым, простым и понятным алгоритмом сжатия данных и при этом иногда оказывается весьма эффективным. Именно подобный алгоритм используется для сжатия изображений в файлах PCX. Он заключается в следующем: любой последовательности повторяющихся входных символов ставится в соответствие набор из трех выходных символов: первый-байт префикса, говорящий о том, что встретилась входная повторяющаяся последовательность, второй-байт, определяющий длину входной последовательности, третий-сам входной символ - . Лучше всего работу алгоритма пояснить на конкретном примере. Например: пусть имеется (шестнадцатиричный) текст вида 05 05 05 05 05 01 01 03 03 03 03 03 03 05 03 FF FF FF FF из 20 байт. Выберем в качестве префикса байт FF. Тогда на выходе архиватора мы получим последовательность06 05 FF 02 01 FF 06 03 FF 01 05 FF 01 03 FF 04 FF Ее длина-18 байт, то есть достигнуто некоторое сжатие. Однако, нетрудно заметить, что при кодировании некоторых символов размер выходного

кода возрастает (например, вместо 01 01 - FF 02 01). Очевидно, одиночные или дважды (трижды) повторяющиеся символы кодировать не имеет смысла - их надо записывать в явном виде. Получим новую последовательность:06 05 01 01 FF 06 03 05 03 FF 04 FF длиной 13 байт. Достигнутая степень сжатия: 13/20 = 65%. Нетрудно заметить, что префикс может совпасть с одним из входных символов. В этом случае входной символ может быть заменен своим “префиксным” представлением, например:то же самое, что и FF 01 FF (три байта вместо одного). Поэтому, от правильного выбора префикса зависит качество самого алгоритма сжатия, так как, если бы в нашем исходном тексте часто встречались одиночные символы FF, размер выходного текста мог бы даже превысить входной. В общем, случае в качестве префикса следует выбирать самый редкий символ входного алфавита. Можно сделать следующий шаг, повышающий степень сжатия, если объединить префикс и длину в одном байте. Пусть префикс-число F0...FF, где вторая цифра определяет длину length от 0 до 15. В этом случае выходной код будет двухбайтным, но мы сужаем диапазон представления длины с 255 до 15 символов и еще более ограничиваем себя в выборе префикса. Выходной же текст для нашего примера будет иметь вид:05 F2 01 F6 03 05 03 F4 FF Длина-10 байт, степень сжатия-50%. 1 Далее, так как последовательность длиной от 0 до 3 мы условились не кодировать, код length удобно использовать со смещением на три, то есть 00=3, 0F=18, FF=258, упаковывая за один раз более длинные цепочки. 2 Если одиночные символы встречаются достаточно редко, хорошей может оказаться модификация алгоритма RLE вообще без префикса, только . В этом случае одиночные символы также обязательно должны кодироваться в префиксном виде, чтобы декодировщик мог различить их в выходном потоке: 06 05 02 01 06 03 01 05 01 03 04 FF Длина-12 байт, степень сжатия-60%. 3 Возможен вариант алгоритма, когда вместо длины length кодируется позиция относительно начала текста distance первого символа, отличающегося от предыдущего. Для нашего примера это будет выходная строка вида: Выводы Алгоритм Степень сжатия Ско-рость Память Сжатие без потерь Про-ходы РО ВИ RLE 2-3 10 10 Да 1 Нет Возм. Здесь и далее приняты следующие обозначения: РО - распространение ошибки, ВИ - возрастание избыточности. Степень сжатия, скорость и используемая оперативная память оцениваются по десятибалльной системе с точки зрения автора. Чем больше величина, тем лучше указанный параметр (выше скорость работы, выше степень сжатия и меньшая потребляемая память). Работа: в реальном масштабе времени и в потоке. Основное применение: PCX, сжатие изображений. 2) Коды Хаффмана. Наиболее эффективным кодом переменной длины, в котором ни одно слово не совпадает с началом другого (т.е. префиксный код) является код Хаффмана. Пусть l1,...,lk-положительные целые числа (k>0). Для того, чтобы существовал префиксный код,

длины слов которого равны l1,...,lk, необходимо и достаточно выполнение неравенства Крафта : Все префиксные коды являются кодами со свойством однозначного декодирования, но не наоборот (например, однозначно декодируемый код 0, 01, 011, 0111, ... не является префиксным). Избыточность дешифрируемого кодирования неотрицательна. Для кода Хаффмана (Шеннона) избыточность не превышает 1, т.е. 0 £ R £ 1. Длина кода Шеннона равна |ai| = é-log(p(ai))ù, а длина кода Хаффмана не превышает величины |ai|. Отсюда, в частности следует вывод, что чем больше длина T символов входного алфавита (для которого строится код Хаффмана), тем меньше избыточность выходного текста и тем выше степень сжатия. Однако, как будет видно в дальнейшем, при этом значительно возрастают требования к памяти данных и к быстродействию программы, поскольку количество кодов равно количеству символов исходных букв. Избыточность кода Хаффмана в значительной мере зависит, как следует из приведенной формулы, от того, на сколько вероятности появления символов близки к отрицательным степеням числа 2. Для двухсимвольного алфавита, например, код Хаффмана никогда не сможет дать сокращения избыточности, пусть даже вероятности различаются на несколько порядков. Рассмотрим построение кода Хаффмана на простом примере: пусть есть текст ABACCADAA. При кодировке двоичным кодом постоянной длины вида: - 00, B - 01, C - 10, D - 11 мы получим сообщение 0001001010001100000 длиной 18 бит. Теперь построим код Хаффмана, используя дерево Хаффмана. Для этого первоначально требуется просмотреть весь исходный текст и получить вероятность (или, что проще - частоту) каждого входящего в него символа. Далее, возьмем два самых редких кода и объединим их в единый узел, частота появления которого равна сумме частот появления входящих в него символов. Рассматривая этот узел, как новый символ, объединяющий два предыдущих, будем повторять операцию слияния символов до тех пор, пока не останется ни одной буквы из входного алфавита. Получим дерево, где листьями являются символы входного алфавита, а узлами - соединение символов из листов-потомков данного узла. Рассматривая каждую левую ветвь как 0, а правую как 1 и спускаясь по дереву вниз до листьев, получим код любого символа: - 0, B - 110, C - 10, D - 111 Исходное сообщение после кодирования станет 011001010011100 (15 бит). Степень сжатия: 15/18 = 83%. Легко убедиться, что имея коды Хаффмана, можно однозначно восстановить первоначальный текст Другой алгоритм построения оптимальных префиксных кодов, дающий аналогичный результат, был предложен Шенноном и Фано . 1 Недостатком такого кодирования является тот факт, что вместе с закодированным сообщением необходимо передавать также и построенную таблицу кодов (дерево), что снижает величину сжатия. Однако существует динамический алгоритм построения дерева

Хаффмана , при котором кодовая таблица обновляется самим кодировщиком и (синхронно и аналогично) декодировщиком в процессе работы после получения каждого очередного символа. Получаемые при этом коды оказываются квазиоптимальными, поэтому текст сжимается несколько хуже. Более того, возрастают сложность алгоритма и время работы программы (хотя она и становится однопроходной), поэтому, в настоящее время динамический алгоритм почти полностью уступил место статическому. 2 Следующим вариантом рассматриваемого алгоритма является фиксированный алгоритм Хаффмена, сочетающий в себе достоинства двух предыдущих - высокую скорость работы, простоту и отсутствие дополнительного словаря, создающего излишнюю избыточность. Идея его в том, чтобы пользоваться некоторым усредненным по многим текстам деревом, одинаковым для кодировщика и декодировщика. Такое дерево не надо строить и передавать вместе с текстом, а значит-отпадает необходимость первого прохода. Но, с другой стороны, усредненное фиксированное дерево оказывается еще более неоптимальным, чем динамическое. Поэтому, иногда может быть удобно иметь несколько фиксированных деревьев для разных видов информации. · Вариантами дерева Хаффмана следует также считать деревья, полученные для монотонных источников. Эти источники имеют два важных для наших целей свойства: · нет необходимости вычислять частоты появления символов входного текста - как следствие, однопроходный алгоритм; · дерево Хаффмана для таких источников может быть представлено в виде вектора из нескольких байтов, каждый из которых обозначает количество кодов дерева определенной длины (см. Задачи), например: запись 1,1,0,2,4 говорит о том, что в дереве имеется по одному коду длин 1 и 2 бита, 0 кодов длиной 3 бита, 2 кода длиной 4 бита и 4 кода длиной 5 бит. Сумма всех чисел в записи даст количество символов в алфавите. Основное применение: универсальный, сжатие текстов и бинарной информации. Динамический алгоритм построения кода Хаффмана используется в архиваторе ICE, статический - в LHA, ARJ. Статический алгоритм Шеннона-Фано используется архиватором PKZIP. Применяется для сжатия изображений (формат JPEG). алгоритм архивация код хаффман Алгоритм Степень сжатия Ско-рость Память Сжатие без потерь Про-ходы РО ВИ Хаффмен статич. 5-6 8 8 Да 2 Да Редко Хаффмен динамич. 4-5 6 7 Да 1 Да Редко Хаффмен фиксир. 3-6 9 Да 1 Да Возм. Монотон. 3-4 8 9 Да 1 Да Возм. 3) Метод «стопки книг» (MTF). Сравнительно недавно появилась еще одна разновидность адаптивного алгоритма Хаффмана, описанная Рябко, а затем Бентли и названная, соответственно, алгоритмом стопки книг или “двигай вверх” (“MTF-Move To Front”). Каждому символу (букве) присваивается код в зависимости от его положения в алфавите - чем ближе символ к началу алфавита - тем короче код (в качестве кода можно

использовать, например, код дерева для монотонных источников). После кодирования очередного символа мы помещаем его в начало алфавита, сдвигая все остальные буквы на одну позицию вглубь. Через некоторое время наиболее часто встречаемые символы сгруппируются в начале, что и требуется для успешного кодирования. Работа алгоритма, действительно, напоминает перекладывание наиболее нужных книг из глубин библиотеки ближе к верху, вследствие чего потом их будет легче найти (аналогия с обыденной жизнью). Алгоритм Степень сжатия Ско-рость Память Сжатие без потерь Про-ходы РО ВИ Стопка книг 3-5 7 8 Да 1 Да Возм. 4) Арифметическое кодирование. Арифметическое кодирование является методом, позволяющим упаковывать символы входного алфавита без потерь при условии, что известно распределение частот этих символов. Арифметическое кодирование является оптимальным, достигая теоретической границы степени сжатия H0. Однако, коды Хаффмана в предыдущем разделе мы тоже называли оптимальными. Как объяснить этот парадокс? Ответ заключается в следующем: коды Хаффмана являются неблочными, т.е каждой букве входного алфавита ставится в соответствие определенный выходной код. Арифметическое кодирование-блочное и выходной код является уникальным для каждого из возможных входных сообщений; его нельзя разбить на коды отдельных символов. Текст, сжатый арифметическим кодером, рассматривается как некоторая двоичная дробь из интервала [0, 1). Результат сжатия можно представить как последовательность двоичных цифр из записи этой дроби. Каждый символ исходного текста представляется отрезком на числовой оси с длиной, равной вероятности его появления и началом, совпадающим с концом отрезка символа, предшествующего ему в алфавите. Сумма всех отрезков, очевидно должна равняться единице. Если рассматривать на каждом шаге текущий интервал как целое, то каждый вновь поступивший входной символ “вырезает” из него подинтервал пропорционально своей длине и положению. Поясним работу метода на примере. Пусть алфавит состоит из двух символов: “a” и “b” с вероятностями соответственно 3/4 и 1/4. Рассмотрим (открытый справа) интервал [0, 1). Разобьем его на части, длина которых пропорциональна вероятностям символов. В нашем случае это [0, 3/4) и [3/4, 1). Суть алгоритма в следующем: каждому слову во входном алфавите соответствует некоторый подинтервал из [0, 1). Пустому слову соответствует весь интервал [0, 1). После получения каждого очередного символа арифметический кодер уменьшает интервал, выбирая ту его часть, которая соответствует вновь поступившему символу. Кодом сообщения является интервал, выделенный после обработки всех его символов, точнее, число минимальной длины, входящее в этот интервал. Длина полученного интервала пропорциональна вероятности появления кодируемого текста. Построение интервала для сообщения

“АВГ...”. Выполним алгоритм для цепочки “aaba” Построение арифметического кода. Шаг Просмотренная цепочка Интервал 0. “” [0, 1) = [0, 1) 1. “a” [0, 3/4) = [0, 0.11) 2. “aa” [0, 9/16) = [0, 0.1001) 3. “aab” [27/64, 36/64) = [0.011011, 0.100100) 4. “aaba” [108/256, 135/256) = [0.01101100, 0.10000111) На первом шаге мы берем первые 3/4 интервала, соответствующие символу “a”, затем оставляем от него еще только 3/4. После третьего шага от предыдущего интервала останется его правая четверть в соответствии с положением и вероятностью символа ”b”. И, наконец, на четвертом шаге мы оставляем лишь первые 3/4 от результата. Это и есть интервал, которому принадлежит исходное сообщение. В качестве кода можно взять любое число из диапазона, полученного на шаге 4. В качестве кода мы можем выбрать, например, самый короткий код из интервала, равный 0.1 и получить четырехкратное сокращение объема текста. Для сравнения, код Хаффмана не смог бы сжать подобное сообщение, однако, на практике выигрыш обычно невелик и предпочтение отдается более простому и быстрому алгоритму из предыдущего раздела. Арифметический декодер работает синхронно с кодером: начав с интервала [0, 1), он последовательно определяет символы входной цепочки. В частности, в нашем случае он вначале разделит (пропорционально частотам символов) интервал [0, 1) на [0, 0.11) и [0.11, 1). Поскольку число 0.1 (переданный кодером код цепочки “aaba”) находится в первом из них, можно получить первый символ: “a”. Затем делим первый подинтервал [0, 0.11) на [0, 0.1001) и [0.1001, 0.1100) (пропорционально частотам символов). Опять выбираем первый, так как 0 < 0.1 < 0.1001. Продолжая этот процесс, мы однозначно декодируем все четыре символа. При рассмотрении этого метода возникают две проблемы: во-первых, необходима вещественная арифметика, вообще говоря, неограниченной точности, и во-вторых, результат кодирования становится известен лишь при окончании входного потока. Дальнейшие исследования, однако, показали, что можно практически без потерь обойтись целочисленной арифметикой небольшой точности (16-32 разряда), а также добиться инкрементальной работы алгоритма: цифры кода могут выдаваться последовательно по мере чтения входного потока . Подобно алгоритму Хаффмана, арифметический кодер также является двупроходным и требует передачи вместе с закодированным текстом еще и таблицы частот символов. Вообще, эти алгоритмы очень похожи и могут легко взаимозаменяться. Следовательно, существует адаптивный алгоритм арифметического кодирования со всеми вытекающими достоинствами и недостатками. Его основное отличие от статического в том, что новые интервалы вероятности перерассчитываются после получения каждого следующего символа из входного потока. Алгоритм Степень сжатия Ско-рость Память Сжатие без потерь Про-ходы РО Арифмет. статич. 5-6 8 8 Да 2 Да Редко Арифмет. динамич. 4-5 6 7 Да 1

Да Редко Работа: в реальном масштабе времени и в потоке (кроме статического). Основное применение: универсальный, сжатие текстов. Используется в архиваторе LZARI, для сжатия изображений (JPEG). 5) Вероятностное сжатие. Чрезвычайно интересный и самый быстрый из известных (наряду с RLE) методов сжатия информации, дающий, к тому же неплохие результаты. Вероятностное сжатие во многом напоминает гадание на кофейной гуще или предсказание погоды, только более эффективно. Работает алгоритм следующим образом: имеется достаточно большая, динамически обновляющаяся таблица предсказаний, в которой для каждой возможной пары последовательных входных символов указывается предсказываемый следующий (третий) символ. Если символ предсказан правильно - генерируется код в виде однобитного префикса, равного 1. Если же мы не угадали - выдается код в виде префикса, равного 0 и неугаданного символа. При этом неугаданный символ замещает в таблице бывший там до этого, обеспечивая постоянное обновление статистической информации. Алгоритм является адаптивным и поэтому он однопроходный и не требует хранения таблицы совместно с сжатым текстом. Декомпрессор работает по аналогичной программе, поддерживая и обновляя таблицу синхронно с компрессором. Если поступил префикс 1, очередная выходная буква извлекается из таблицы по индексу, полученному из двух предыдущих букв, иначе-просто копируется код из входного потока в выходной. Для небольшого повышения степени сжатия рекомендуется инициализировать таблицу перед началом работы какими-нибудь часто встречающимися буквосочетаниями. Похожий алгоритм, основанный на модели Маркова 1 порядка, использовался в архиваторе PKZIP Ver.1.5 под именем Reducing . Он является почти по всем параметрам хуже, чем Fin (двупроходный, требует передачи таблицы вместе с текстом), но тем не менее достоин упоминания. Таблица программы Reducing содержит массив символов V[256][32], т.е. для каждой входной буквы на первом проходе находятся не более, чем 32 (но может быть и меньше) наиболее часто встречающихся следующих буквы. На втором проходе дело обстоит сходно с Fin-если очередной символ b, следующий за символом a находится среди 32 ожидаемых, генерируется код , prefix=1, position=положению символа b в массиве V[a][ ] из 32 букв. Иначе выдается префикс 0 и сам символ b. Алгоритм статический, поэтому таблица не обновляется в процессе работы. Выводы Алгоритм Степень сжатия Ско-рость Память Сжатие без потерь Проходы РО ВИ Reducing 4-5 8 7 Да 2 Да Редко. Вероятн. 5-6 9 6 Да 1 Да Возм. Работа: в реальном масштабе времени и в потоке. Основное применение: универсальный. 6) LZ77 Самый эффективный алгоритм сжатия информации, являющийся одновременно и очень простым для понимания и очень сложным в реализации, был предложен Лемпелом и Зивом (Lempel, Ziv) в 1977 г. (LZ77).

Заключается он в следующем: имеется “скользящий” (sliding) словарь V объемом |V|=2...32 кБ. Если очередная входная строка s текста совпадает со строкой из словаря, то она заменяется на указатель ptr на эту строку вида ptr=, для которой prefix=1, после чего текущая позиция в исходном тексте pos сдвигается на length символов дальше. Понятие словаря и строки несколько отличается от обыденного: под строкой подразумевается любая непрерывная последовательность символов (букв), начинающаяся с определенной позиции в словаре или в тексте, длиной |s|£smax. smax обычно выбирается из диапазона 16...256 байт; в качестве словаря используются |V| байт сжимаемого текста, предшествующие текущему кодируемому символу в позиции pos (отсюда и название “скользящий”-словарь как бы скользит вдоль по тексту от его начала к концу, поддерживая самую свежую информацию о его содержании). Поскольку, перед началом процесса компрессии перед первым символом ничего нет, первоначально словарь пуст (или заполнен каким-нибудь определенным символом). Словами в словаре выступают любые строки длины не более smax, начинающиеся в любой позиции словаря. Таким образом, в словаре всегда присутствует |V|×smax слов. Если же строки s в словаре не оказалось, генерируется код chr вида chr=, где prefix=0, a symbol-текущий символ исходного текста (в позиции pos). Префикс, как видно, нужен для того, чтобы отличить код ptr от кода chr. Именно префикс вносит в алгоритм кодирования LZ77 некоторую дополнительную информацию, из-за которой возможно возрастание избыточности. Вид словаря и исходного текста для алгоритма LZ77 перед началом работы а) и на 23-ем шаге б). На рисунке б) показан вид словаря и кодируемого текста на 23-ем символе, когда алгоритм обнаружил совпадение слова “где “ с таким же словом в словаре длины length=4 байта, на расстоянии distance=23 байта от pos. Будет сгенерирован код ptr=<1,23,4>. Длина ptr-кода |ptr| = 1 + élog(|V|)ù + élog (|smax|)ù (бит),кода |chr| = 1 + élog(|symbol|)ù (бит). Отсюда следует вывод, что, если длина ptr-кода в байтах превышает length, то такое кодирование лишь увеличит избыточность, поэтому в алгоритм вводят порог (threshold), равный обычно 2-3 байтам и, если совпадающая строка короче этого порога, она не кодируется (вернее, кодируется явно, посимвольно, посредством chr-кодов). Неплохой альтернативой последовательному поиску может служить использование множественных двусвязных списков - несложной структуры данных, которая позволяет ускорить поиск более чем в 10 раз. Она, конечно, существенно уступает в скорости бинарным деревьям, но значительно проще в реализации. Идея метода заключается в том, что создается 256 (или более) двусвязных списков, в каждый из которых включаются слова, начинающиеся на одинаковую букву (или 1.5-2 буквы). Поиск очередного слова осуществляется, очевидно, только в том списке, слова которого начинаются на ту же букву, что и

*первая буква текущего слова symbol. Обратное восстановление текста осуществляется значительно быстрее и проще. Поскольку к моменту декодирования очередного символа весь предыдущий текст уже известен, декодировщику не требуется передавать словарь вместе с упакованными данными. Словарь строится декомпрессором по тому же алгоритму, что и компрессором, на основе уже полученных символов. Если декодировщик обнаруживает (по префиксу), что поступил chr-код, он просто копирует symbol в выходной поток и в словарь, передвигая затем словарь на одну позицию вслед за текстом. Если же поступил ptr-код, декодировщик копирует length символов, начиная с позиции distance в словаре. Напомним, что distance отсчитывается от конца словаря к началу. Все характеристики “классического” варианта алгоритма почти целиком определяются размером словаря |V| и максимальной длиной строк в словаре smax (см. Задачи). Высокоэффективным оказывается способ двухступенчатого кодирования информации, когда выходной поток кодировщика LZ77 поступает на вход блока арифметического кодера (LZARI) или кодера Хаффмена (LZHUF). Такое возможно, т.к. коды length и distance распределены далеко не случайно и неплохо описываются моделью для монотонных источников. Действительно, более длинные совпадения встречаются явно реже коротких, а вероятность совпадения слов на небольшом расстоянии выше, чем вероятность встретить похожее слово в “глубинах” словаря. Оставшиеся после первого прохода коды symbol по-прежнему подчиняются модели Бернулли. Подобное решение, используемое в архиваторах LHA, ARJ, PKZIP, RAR делает их одними из самых эффективных и популярных в настоящий момент. Выводы Алгоритм Степень сжатия Ско-рость Сжатие без потерь Про-ходы РО ВИ LZ77 LZSS 9 3 2 Да 1 Да Редко LZHUF LZARI 10 2 2 Да 2 Да Редко Работа: в реальном масштабе времени и в потоке (кроме LZARI, LZHUF). Основное применение: универсальный, сжатие текстов (LHA, ICE, ARJ, PKZIP, PAK, LZEXE). Практическая реализация алгоритма LZ77 Примечание: т.к. алгоритм был подробно рассмотрен в предыдущем разделе, все описание этого раздела находится внутри программы в виде комментариев. Листинг 1. Реализация алгоритма LZ77 на языке C++ #define DICBITS 12 // Log(DICSIZE) #define DICSIZE (1<< #include #include #include FILE *first_file, *second_file; long srcleng=0, fileleng;char *srcbuf, *srcstart; /*================================ ========Функция записи========= ================================*/putbits(int data, int nbits) {int bitcounter=0;int outdata=0;bit, error;<<=(16-nbits);(; nbits > 0 ; nbits--) {(bitcounter == 8) {=0;=putc(outdata,second_file);(error == EOF) {("Error writing to Second file.");-5; } }<<=1;=(data & 0x8000) ? 1:0;+=bit;++;<<=1; } } /*================================ ======Функция*

Архивирования=====
==========/compress_stud() {stat buf;char *position, *pointer; //байт в файле , байт в буферei, dist, offset=0, last=NO, cnt, maxleng;("Compress started."); // Записываем размер исходного файла fstat(fileno(first_file),&buf); fileleng=buf.st_size;(fileno(second_file), &fileleng, sizeof(long)); // Читаем файл в буфер по частям размера TEXTSIZE((srcleng=fread(srcstart+offset, 1, TEXTSIZE, first_file))>0) {(srcleng < TEXTSIZE) // Последняя часть текста {=YES; }=srcstart;=srcbuf;+=offset;(srcleng>0) {("\n\nStep - %d\n",srcleng);=0;((last == NO) && (srcleng < STRMAX)) // Если в буфере текста осталось мало символов, сдвигаем словарь и оставшийся текст в начало буфера и дочитываем следующую часть из файла {(srcbuf,pointer,DICSIZE+(int)srcleng);=(int)srcleng;; }(i=DICSIZE-1; i >= 0; i--)// Ищем самую длинную совпадающую строку в словаре {(cnt=0; cnt < STRMAX; cnt++)(*(position+cnt) != *(pointer+i+cnt));(cnt <= THRESHOLD)// Если длина меньше порога, отбрасываем;(cnt == STRMAX)// Если максимальная строка, дальше не ищем {=DICSIZE-1-i; //позиция=STRMAX; //длина; }(cnt > maxleng) // Если очередная строка длиннее уже найденных, сохраняеМ ее длину и позицию {=DICSIZE-1-i; // позиция=cnt; //длина } {=(int)srcleng; //обрезаем длину по границу буфера }(maxleng > THRESHOLD)//Если строка достаточно длинная, формируем pointer-код {("link!\n");(1,1); //помечаем как ссылку(dist,DICBITS); //записываем позицию(maxleng-THRESHOLD-1,STRBITS); //записываем длину+=maxleng;=maxleng;+=maxleng; }// Иначе - chr-код {("Char!\n");(0,1); //помечаем как chr-код(*position,8); //записываем чар код++;-;++; } } } // Записываем оставшиеся биты(0,8); printf("\nCompress completed!\n",fileleng); } //Разархивирование Алгоритм LZ77 /*==========*/
======Функция считывания========
==========/getbits(int nbits) {int bitcounter=8;int indata=0;bit, data=0;(; nbits > 0 ; nbits--) {(bitcounter == 8) {=0;=getc(first_file); }(indata == EOF) {("Error writing to First file.");-6; } =(indata & 0x80) ? 1:0;<<=1;+=bit;++;<<=1; }data; } /*==========*/
=======Функция Извлечения=======
==========/decompress_stud() {stat buf;char *pos;i, dist, ch, maxleng;("Decompress started.\n"); // Получаем длину исходного файла(fileno(first_file),&fileleng,sizeof(long));=srcstart;(fileleng > 0) {((ch=getbits(1)) == 0) // Если chr-код, копируем в буфер текста символ {=getbits(8);(ch,second_file); *pos=ch;++;-; }// Иначе - копируем maxleng символов из словаря, начиная с позиции dist {=getbits(DICBITS)+1;=getbits(STRBITS)+THRESHOLD+1;(i=0; i < maxleng; i++) { *(pos+i)=*(pos+i-dist);(*(pos+i-dist),second_file); }+=maxleng;=maxleng; }(pos >

srcstart+TEXTSIZE) // Если буфер заполнен, записываем его на диск и сдвигаем словарь в начало буфера {(srcbuf,pos-DICSIZE,DICSIZE);=srcstart; } }("\nDecompress completed."); }. Сравнительная таблица результатов сжатия. Имя файла Исходный размер Размер после сжатия Test1.txt 240 КБ 27 КБ Test2.docx 450 КБ 50 КБ Test3.docx 370 КБ 40 КБ