Курсовой проект: Архиватор (Arithmetic + LZ77)

Выполнил студент группы 08-208Б МАИ *Лисовский Олег Романович.*

Условие

Необходимо реализовать два известных метода сжатия данных для сжатия одного файла.

Формат запуска должен быть аналогичен формату запуска программы gzip, должны быть поддержаны следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Метод решения

Как и требуется в условии запуск программы аналогичен запуску утилиты gzip: ./main <ключи> <файлы> <ключи> <файлы> ...

Первоначально программа обрабатывает то, что получает на входе, отделяя имена файлов и директорий от введённых ключей. Ключи от файлов программа отделяет по первому символу слова: если этот символ является «-», то полученное слово является набором ключей. В процессе изучения поведения утилиты gzip я вывел следующее дерево приоритетов ключей:Т. е. если уже был введён ключ -t, то далее ключи -c, -k, -d, -1, -9 программа учитывать не будет, но если вы ввели ключ -l, то он делает ключи -c, -k, -d, -1, -9, -t недействительными как сейчас так и при их будущих вводах. В то же время сочетания -cd, -c1, -c9, -kd, -k1, -k9, а так же сочетания ключа -r со всеми остальными ключами не являются взаимоисключающими. Если же во время обработки ключей встречается неизвестный ключ, то программа прекращает свою работу с соответствующей ошибкой, как и утилита gzip.

Если же полученное слово не начинается с символа «-», то программа идентифицирует его как имя файла/директории и заносит в красно-чёрное дерево для последующей обработки.

После обработки всех полученных слов программа проверяет красно-чёрное дерево на пустоту: если оно пустое, то программа завершается с соответствующим сообщением. Если же оно не пустое, то программа начинает обрабатывать все строки в данном дереве по следующему алгоритму: проверяется принадлежит ли имя директории - если нет, то это имя файла, и в дальнейшем оно обрабатывается как файл; если же это имя принадлежит директории, то проверяется активность ключа -r: если он активен, то с данной директорией идёт работа, иначе она игнорируется, и выводится соответствующее сообщение.

Работа с директорией сводится к получению всех имен файлов и директорий из неё, кроме имён «.» «..» - они пропускаются. Работа с директориями повторяет вышеописанную.

Работа с файлами во многом зависит от введённых ключей.

Если введённые ключи говорят о необходимости компрессии, то при отсутствии ключа -c производится проверка на наличие у файла суффикса «.gz». Если он присутствует, то файл не обрабатывается, и выводится соответствующее сообщение. Если же у этого файла нет расширения «.gz», то при отсутствии ключа -c проверяется наличие файла, имя которого отличается от полученного только стоящим в конце суффиксом. Если такой файл не найден, то работа продолжается. Если же он найден, то пользователю предлагается сделать выбор - перезаписывать данный файл или нет. В случае отрицательного ответа работа с данным файлом прекращается. Далее в зависимости от наличия ключей -1 и -9 определяется степень сжатия. Далее поочерёдно происходит компрессия файла с помощью обоих алгоритмов компрессии (они будут описаны ниже вместе с алгоритмами декомпрессии). В случае провала любого алгоритма компрессии работа с файлом прекращается, и выводится соответствующая ошибка. Если же оба алгоритма сработали нормально, то при отсутствии ключа -c выбирается временный файл, созданный алгоритмами, с меньшим размером, и именно он получает расширение «.gz», а файл с большим размером удаляется. Так же проверяется наличие ключа -k. При его отсутствии изначальный файл удаляется.

Если же ключи указывают на необходимость декомпрессии, то в случае отсутствия ключа -c и наличии ключа -d проверяется наличие у файла суффикса «.gz». При выполнении всех этих условий работа с файлом прекращается, и выводится соответствующее сообщение. Далее при отсутствии ключей -t и -c проверяется наличие файла с таким же именем, но без расширения «.gz». Если такой файл есть, то пользователю предлагается сделать выбор о его перезаписи. В случае отказа, работа с данным файлом прекращается, и выводится соответствующее сообщение. Если ответ положительный или такого файла нет, то работа продолжается. Из поступившего архива считывается первый байт, который содержит указание на метод архивации. Если этот байт интерпретируется как символ «7» или «A», то декомпрессия производится по алгоритму LZ77 или Арифметика соответственно. Если первый байт интерпретируется иначе, то работа с файлом прекращается, и выводится соответствующее сообщение. В случае неудачного завершения алгоритма, выводится соответствующее сообщение и при отсутствии ключей -t и -c удаляется файл, в который записывались данные после декомпрессии, и работа с файлом прекращается. Далее при отсутствии ключей -t, -c и -k, удаляется изначальный архив, а при отсутствии ключей -t и -c временный файл для декомпрессии переименовывается и получает имя изначального архива без расширения «.gz».

В случае указания ключа -l производятся следующие действия. Читается первый байт, и в случае если он не совпадает с буквами, указывающими на метод архивации, то работа прекращается, и выводится соответствующее сообщение. Далее читается 8 байт, в которые помещается размер файла до компрессии. После программа считывает размер архива, и вычисляется процент сжатия. Далее выводятся размер сжатого файла, размер до компрессии, процент сжатия в полуинтервале $[-100\%; 100\%)$ и имя файла до архивации (если файл имеет расширение «.gz», то имя выводится без этого расширения, в противном случае выводится имя архива).

Описание программы

Весь код программы находится в единственном файле - lab1.c.

Основные типы данных

1. TInfo - отвечает за хранение пары «ключ-значение». Обладает полями number и key, которые хранят ключ типа unsigned long long int и значение типа char* соответственно.

2. TBucket - отвечает за хранение ключей в определённом промежутке. Обладает полями data и inBucket хранящие пары «ключ-значений» типа TInfo** и количество элементов в бакете типа int соответственно.

Основные функции

1. void PrepareData - инициализирует бакеты.

2. void FullBuckets - заполняет бакеты массивом типа TInfo одновременно совершая сортировку вставками «ключ-значений» в бакетах.

3. void Print - печатает в порядке возрастания все элементы бакетов, после чего освобождает выделенную под них память.

4. int main - считывает «ключ-значения» и помещает их в динамический массив типа TInfo.

Дневник отладки

Первой ошибкой было выведение сообщения о том что файл пуст, из-за чего чекер выдавал ошибку. Сообщение было устранено. Далее было обнаружено, что программа работает медленнее чем предполагалось. Была исправлена функция заполнения бакетов. Следующей ошибкой было добавление элемента с максимальным ключом в самый первый бакет. После исправления, при получении максимально допустимого ключа он помещался в дополнительный бакет. Заключительная ошибка заключалась в формуле вычисления бакетов: целочисленное значение «ключ» делилось на заведомо большее значение максимального «ключа» из за чего получалось нулевое значение. Исправлено путём явного приведения выражения к типу long double.

Тест производительности

Было проведено 4 теста:

1. 10000 элементов - выполнен за 0.06849 секунд.

2. 100000 элементов - выполнен за 0.61145 секунд.

3. 1000000 элементов - выполнен за 6.3306 секунд.

4. 10000000 элементов - выполнен за $64,2903$ секунд.

На основании проведённых тестов можно увидеть, что время выполнения программы с увеличением количества элементов в 10 раз так же увеличивается в 10 раз. Из этого можно сделать вывод, что программа работает за линейное время - $O(n)$. Это возможно благодаря использованию такого же количества карманов, что и количество поступивших «ключ-значений». Да, из-за этого многие карманы могут быть не использованы, но в тоже время только благодаря этому можно добиться линейного времени исполнения.

Выводы

Данный алгоритм может быть применим в любой программе, имеющей сортировку числовых элементов. Однако некоторые вещи могут довольно сильно снизить производительность данного алгоритма:

1. Неправильное понимание работы алгоритма. Эта проблема заключается в том что программист может неправильно понять как и создавать и заполнять бакеты. К примеру вместо указанной выше формулы определения номера бакета, программист может начать задавать границы для каждого бакета вручную из-за чего будет тратиться лишняя память и потребуется затратить дополнительное время на присваивания новых значений. После чего при заполнении бакетов программист будет ориентироваться по этим границам из-за чего потребуется дополнительный цикл для того чтобы определить номер бакета по его границам. Вдобавок из-за работы с очень большими числами можно не заметить того как возникнет переполнение из-за чего номер бакета может определиться неправильно.

2. Количество поступивших значений для сортировки, а так же верхний и нижний пределы поступивших значений. Если заранее известно сколько всего поступит элементов, то можно будет заранее задать размеры бакетов, а не переопределять их после каждого считанного значения, что сэкономит время, а если ещё и знать хотя бы максимальное значение из тех что поступит, то поступающие элементы можно будет сразу отправлять на хранение - сортировку в бакеты. Если же количество поступающих элементов заранее не известно, но известны границы области из которой поступают элементы, то можно будет не проводить постоянного сравнения поступающих элементов с поставленными программистом минимумом и максимумом, тем самым затратив меньше времени на работу алгоритма.

3. Равномерность распределения элементов. Элементы считаются равномерно распределёнными на участке если при помещении их на числовую прямую каждые два соседних числа будут находиться друг от друга на одинаковом расстоянии (пример: 2 4 6 8 ... 1000000). Если элементы распределены таким образом то для

каждого элемента будет инициализирован отдельный бакет и по завершении считывания элементов можно будет сразу приступить к следующим действиям (в нашем случае к печати). Если элементы распределены не так равномерно, но хотя бы какой то средний интервал соблюдается (пример 1 3 4 5 7 9 10), то в каждом бакете придётся заводить массив и проводить в нем сортировку сложности $O(n^2)$, так как на небольшом количестве элементов сортировки квадратной сложности по времени работы эквивалентны сортировкам за линейное время, а при такой равномерности элементов в каждый бакет должно будет попасть небольшое количество элементов. Если же элементы распределены неравномерно (1 2 3 ... 200000 18446744073709551615) то тогда в один бакет может попасть слишком много элементов из-за чего его их сортировка за $O(n^2)$ даст о себе знать и алгоритм карманной сортировки тоже станет работать за $O(n^2)$, что полностью убьет все его плюсы и вдобавок заставит использовать больше памяти чем необходимо. Возможно эту проблему можно решить сортируя числа в бакетах с помощью карманной сортировки, но это значительно усложнит написание программы. Именно этот момент является самым критически важным для использования алгоритма в той или иной программе. Если заведомо известно, что числа которые будут упорядочиваться алгоритмом сортировки будут неравномерно распределены, то лучше подобрать другую сортировку. Так же если известно что сортироваться будет не большое количество чисел, то гораздо проще использовать сортировку за $O(n^2)$, так как она реализуется гораздо проще и по времени работы будет практически идентична карманной сортировке.

Классической задачей для применения данного алгоритма можно назвать такую: мы являемся создателем популярной онлайн игры в крестики-нолики, и мы каждый год проводим по ней международный чемпионат, где награда сопоставима с количеством побед и подсчёт мест всех участников турнира будет проводиться после завершения мероприятия. Очевидно, что количества побед будут распределены более-менее равномерно благодаря чему можно будет применить карманную сортировку и вознаградить каждого игрока в соответствии занятым им местом, и это удастся сделать достаточно быстро.