Лабораторная работа № 1

Организация таблиц идентификаторов

цель работы

Цель работы: изучить основные методы организации таблиц идентификаторов, получить представление о преимуществах и недостатках, присущих разным методам организации таблиц идентификаторов.

Для выполнения лабораторной работы нужно написать программу, которая получает на входе набор идентификаторов, организует таблицы идентификаторов по помощью заданных методов, позволяет осуществить многократный поиск произвольного идентификатора в таблицах и сравнить эффективность методов организации таблиц. Список идентификаторов считать заданным в виде текстового файла. Длина идентификаторов ограничена 32 символами.

короткие теоретические сведения. назначение таблиц идентификаторов

При выполнении семантического анализа, генерации кода и оптимизации результирующей программы компилятор должен оперировать характеристиками основных элементов исходной программы - переменных, констант, функций и других лексических единиц входного языка. Эти характеристики могут быть получены компилятором на этапе синтаксического анализа входной программы (чаще всего при анализе структуры блоков описаний переменных и констант), а также дополнены на этапе подготовки к генерации кода (например, при распределении памяти).

Набор характеристик, соответствующий каждому элементу начальной программы, зависит от типа этого элемента, от его смысла (семантики) и, соответственно, от той роли, которую он выполняет в начальной и результирующей программах. В каждом конкретном случае этот набор характеристик может быть свой в зависимости от синтаксиса и семантики входного языка, от архитектуры целевой

вычислительной системы и от структуры компилятора. Но типичные характеристики, которые чаще всего присущи тем или иным элементам начальной программы. Например, для переменной - это ее тип и адрес ячейки памяти, для константы - ее значение для функции - количество и типы формальных аргументов, тип результата, возвращаемого адрес вызова кода функции. Более подробную информацию о характеристиках элементов исходной программы, их анализ и использования можно найти в [1,3, 7].

Главной характеристикой любого элемента исходной программы является его имя. Именно с именами переменных, констант, функций и других элементов входной языка оперирует разработчик программы - поэтому и компилятор должен уметь анализировать эти элементы по их именам.

Имя каждого элемента должно быть уникальным. Многие современные языков программирования допускают совпадения (неуникальность) имен переменных и функций в зависимости от их области видимости и других условий исходной программы. В этом случае уникальность имен должен обеспечивать сам компилятор - о том, как решается эта проблема, можно узнать в [1 - 3, 7], здесь же будем считать, что имена элементов исходной программы всегда уникальны.

Таким образом, задача компилятора состоит в том, чтобы сохранять некоторую информацию, связанную с каждым элементом исходной программы, и иметь доступ к этой информации по имени элемента. Для решения этой задачи компилятор организует специальные хранилища данных, называемые *таблицами идентификаторов,* или *таблицами символов.* Таблица идентификаторов состоит из набора полей данных (Записей), каждое из которых может соответствовать одному элементу исходной программы. Запись содержит всю необходимую компилятору информацию о данном элемент и может пополняться по мере работы компилятора. Количество записей зависит от способа организации таблицы идентификаторов, но в любом случае их может быть меньше, чем элементов в исходной программе. В принципе, компилятор может работать не с одной, а с несколькими таблицами идентификаторов - их количество и структура зависят от реализации компилятора [1,2].

Принципы организации таблиц идентификаторов

Компилятор пополняет записи в таблице идентификаторов по мере анализа начальной программы и выявления в ней новых элементов, требующих размещения в таблице. Поиск информации в таблице выполняется каждый раз, когда компилятор необходимые сведения о тот или иной элемент программы. причем следует отметить, что поиск элемента в таблице будет выполняться компилятором существенно чаще, чем перемещение в нее новых элементов. Так происходит потому, что описания новых элементов в исходной программе, как правило, встречаются гораздо реже, чем эти элементы используются. Кроме того, каждому добавлению элемента в таблицу идентификаторов в любом случае предшествовать операция поиска - убедиться, что такого элемента в таблицы нет.

На каждую операцию поиска элемента в таблице компилятор тратить время, и поскольку количество элементов в начальной программе велика (от единиц до сотен тысяч в зависимости от объема программы), это время существенно влиять на общее время компиляции. Поэтому таблицы идентификаторов должны быть организованы так, чтобы компилятор имел возможность максимально быстро выполнять поиск нужного ему записи в таблице по имени элемента, с которым связан эту запись.

Можно выделить следующие способы организации таблиц идентификаторов:

- простые и упорядоченные списки;
- бинарное дерево;
- хэш-адресация с рехешуванням;
- хэш-адресация по методу цепочек;
- комбинация ХШ-адресации со списком или бинарным деревом.

Далее будет дано краткое описание всех вышеперечисленных способов организации таблиц идентификаторов. Более подробную информацию можно найти в [3.7].

Простые методы построения таблиц идентификаторов

В простейшем случае таблица идентификаторов есть линейным

неурегулированным списку, или массивом, каждая ячейка которого содержит данные о соответствующий элемент таблицы. Размещение новых элементов в такой таблице выполняется путем записи информации в дежурную ячейку массива или списка по степени выявления новых элементов в исходной программе. Поиск нужного элемента в таблице в этом случае будет выполняться путем последовательного перебора всех элементов и сравнение их имени с именем искомого элемента, пока не будет найден элемент с таким же именем. Тогда если за единицу времени принять время, затрачиваемое компилятором на сравнение двух строк (в современных вычислительных системах такое сравнение чаще всего выполняется одной командой), то для таблицы, содержащей N элементов, в среднем будет выполнено N / 2 сравнений.

Время, необходимое на добавление нового элемента в таблицу (ТД), не зависит от числа элементов в таблице (N). Но если N велико, то поиск будет требовать значительных затрат времени. Время поиска (Тп) в такой таблице можно оценить как Тп = O (N). Поскольку именно поиск в таблице идентификаторов является наиболее часто выполняемой компилятором операцией, такой способ организации таблиц идентификаторов неэффективно. Он применяется только для простейших компиляторов, работающих с небольшими программами.

Поиск может быть выполнен эффективнее, если элементы таблицы отсортированы (упорядоченные) естественным образом. Поскольку поиск осуществляется по имени, наиболее естественным решением будет расположить элементы таблицы в прямом или обратном алфавитном порядке. Эффективным методом поиска в упорядоченном списке из N элементов является *бинарный,* или *логарифмический поиск.*

Алгоритм логарифмического поиска заключается в следующем: искомый символ сравнивается с элементом (N + 1) / 2 в середине таблицы; если этот элемент не является искомым, то мы должны просмотреть только блок элементов, пронумерованных от 1 до (N + 1) / 2-1, или блок элементов от (N + 1) / 2 + 1 до N в зависимости от того, меньше или больше искомый элемент того, с которым его сравнили. затем процесс повторяется над нужным блоком в два раза меньшего размера. так продолжается до тех пор, пока либо искомый элемент не будет найден, или

алгоритм не дойдет до очередного блока, содержащего один или два элемента (с которыми можно выполнить прямое сравнение искомого элемента). поскольку на каждом шагу число элементов, которые могут содержать искомый элемент, сокращается в два раза, максимальное число сравнений равно 1 + 1og2N. тогда время поиска элемента в таблице идентификаторов можно оценить как Тп = O (1og2N). для сравнения: при N = 128 бинарный поиск требует больше 8 сравнений, а поиск в неурегулированной таблицы - в среднем 64 сравнения. метод называют «Бинарным поиском», поскольку на каждом шагу объем данной информации сокращается в два раза, а «логарифмическим» - поскольку время, затрачиваемое на поиск нужного элемента в массиве, имеет логарифмическую зависимость от общего количества элементов в нем. Недостатком логарифмического поиска является требование упорядочивание таблицы идентификаторов. Поскольку массив информации, в котором выполняется поиск, должен быть упорядочен, время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов часто является видимым компилятором еще до того, как она наполнена, поэтому нужно, чтобы условие упорядоченности выполнялась на всех этапах обращения к ней. Итак, для построения такой таблицы можно пользоваться только методом прямого упорядоченного включения элементов.

Если пользоваться стандартными методами, которые используются для организации упорядоченных массивов данных, то среднее время, необходимое на занесения всех элементов в таблицу, можно оценить следующим образом:

$$Тд = O (N • log2N) + k • O (N2).$$

Здесь k - некоторый коэффициент, отражающий соотношение между тем, что расходуется компьютером на выполнение операции сравнения и операции занесения данных.

При организации логарифмического поиска в таблице идентификаторов обеспечивается существенное сокращение времени поиска нужного элемента за счет увеличение времени на добавление нового элемента в таблицу. поскольку добавления новых элементов в таблицу идентификаторов происходит существенно реже, чем

обращения к ним, этот метод следует признать эффективным, чем метод организации неурегулированной таблицы. Однако в реальных компиляторах этот метод непосредственно также не используется, поскольку существуют эффективные методы.

Построение таблиц идентификаторов по методу бинарного дерева

Можно сократить время поиска искомого элемента в таблице идентификаторов, не увеличивая значительно время, необходимое на ее заполнение. для этого надо отказаться от организации таблицы в виде непрерывного массива данных. Существует метод построения таблиц, при которой таблица имеет форму бинарного дерева. Каждый узел дерева является элементом таблицы, причем корневым узлом становится первый элемент, который встретится компилятором при заполнении таблицы. Дерево называется бинарным, поскольку каждая вершина в нем может иметь более двух ветвей. Для определенности будем называть две гаилкы «права» и «левая».

Рассмотрим алгоритм заполнения бинарного дерева. Будем считать, что алгоритм работает с потоком входных данных, содержащая идентификатор. первый идентификатор, как уже было сказано, помещается в вершину дерева. все дальнейшие идентификаторы попадают в дерево по следующему алгоритму:

1. Выбрать очередной идентификатор из входного потока данных. если очередного идентификатора нет, то построение дерева закончена.

2. Сделать текущим узлом дерева корневую вершину.

3. Сравнить имя очередного идентификатора с именем идентификатора, содержится в текущем узле дерева.

4. Если имя очередного идентификатора меньше, то перейти к шагу 5, если равно - прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе - перейти к шагу 7.

5. Если в текущем узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе - перейти к шагу 6.

6. Создать новую вершину, поместить в нее информацию об очередном

идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

7. Если в текущем узла существует права вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе - перейти к шагу 8.

8. Создать новую вершину, поместить в нее информацию об очередном идентификатор, сделать эту новую вершину правой вершиной текущего узла и вернуться к шагу 1.

Рассмотрим в качестве примера последовательность идентификаторов Ga, D1, M22, E, A12, BC, F. На рис. 1.1 проиллюстрирован весь процесс построения бинарного дерева для этой последовательности идентификаторов.

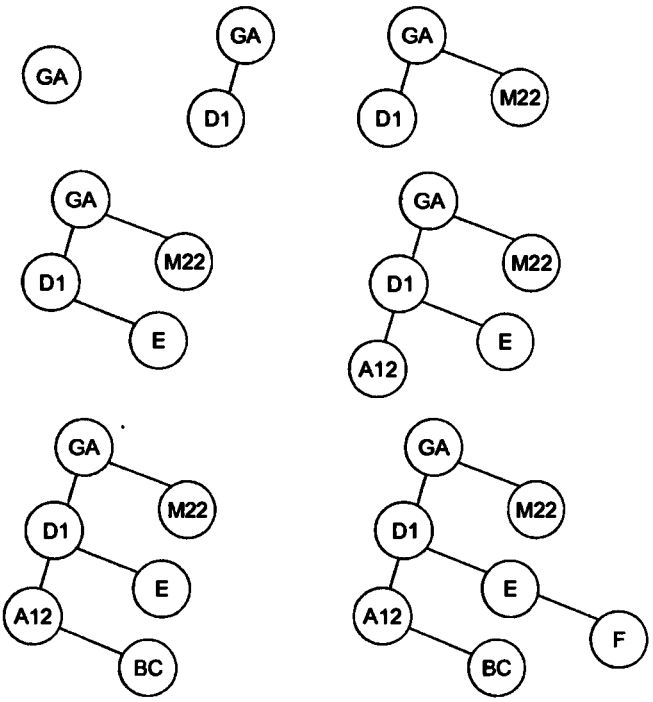


Рис. 1.1. Заполнение бинарного дерева для последовательности идентификаторов Ga, D1, M22, E, A12, BC, F

Поиск элемента в дереве выполняется по алгоритму, схожему с алгоритмом заполнения дерева:

1. Сделать текущим узлом дерева корневую вершину.

2. Сравнить имя искомого идентификатора с именем идентификатора, содержится в текущем узле дерева.

3. Если имена совпадают, то искомый идентификатор найден,

алгоритм завершается, иначе надо перейти к шагу 4.

4. Если имя очередного идентификатора меньше, то перейти к шагу 5, иначе - перейти к шагу 6.

5. Если в текущем узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе - искомый идентификатор не найден, алгоритм завершается.

6. Если в текущем узла существует права вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе - искомый идентификатор не найден, алгоритм завершается.

Для данного метода число необходимых сравнений и форма дерева, получилось, зависят от того порядка, в котором поступают идентификаторы. Например, если в рассмотренном выше примере вместо последовательности идентификаторов Ga, D1, M22, E, A12, BC, F взять последовательность A12, BC, D1, E, F, Ga, M22, то дерево выродится в упорядоченный однонаправленный связный список. эта особенность является недостатком данного метода организации таблиц идентификаторов. Другими недостатками метода являются: необходимость хранить два дополнительных ссылки на левую и правую ветвь в каждом элементе дерева и работа с динамическим выделением памяти при построении дерева.

Если предположить, что последовательность идентификаторов в начальной программе статистически неурегулированной (что в целом соответствует действительности), то можно считать, что построено бинарное дерево будет невырожденным. тогда среднее время на заполнение дерева (Тд) и на поиск элемента в нем (Тп) можно оценить следующим образом [3, 7]:

$$Тд = N \cdot O (1og2N)$$

$$Тп = O (1og2N).$$

Несмотря на указанные недостатки, метод бинарного дерева достаточно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных

деревьев для идентификаторов различных типов и разной длины [1,2,3,7].

Хэш-функции и хеш-адресация

В реальных начальных программах количество идентификаторов так велика, что даже логарифмическую зависимость времени поиска от их числа нельзя признать удовлетворительной. Необходимые эффективные методы поиска информации в таблице идентификаторов. Лучших результатов можно достичь, если применить методы, связанные с использованием хэш-функций и хэш-адресации.

Хэш-функцией F называется некоторое отображение множества входных элементов R на множество целых неотрицательных чисел Z: F (r) = n, r является R, N является Z. Сам термин «хэш-функция» происходит от английского термина «hash function» (hash - «Мешать», «смешивать», «путать»).

Множество допустимых входных элементов R называется областью определения хэш-функции. Множеством значений хэш-функции F называется подмножество M с множества целых неотрицательных чисел Z: M является Z, содержит все возможные значения, которые возвращаемые функцией F: • rIR: F (r) является M и • m является M: • r является R: F (r) = m. процесс отображение области определения хеш-функции на множество значений называется *хешированием.*

При работе с таблицей идентификаторов хэш-функция должна выполнять отображение имен идентификаторов на множество целых неотрицательных чисел. областью определение хэш-функции будет множество всех возможных имен идентификаторов.

Хэш-адресация заключается в использовании значения, которое возвращается хеш функцией, как адрес ячейки с некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хэш-функции.

Итак, в реальном компиляторе область значений хэш-функции никак должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хеш-адресации, заключается в добавлении каждого элемента таблицы в ячейку, адрес которой возвращает хеш-функция, вычисленная для этого элемента. Тогда в идеальном случае для добавления любого элемента в таблицу идентификаторов достаточно

только вычислить его хеш-функцию и обратиться к нужной ячейки массива данных. Для поиска элемента в таблице также необходимо вычислить хеш-функцию для искомого элемента и проверить, не является ли заданная ею ячейка массива пустой (если она не пуста - элемент найден, если пустые - не найден). Сначала таблица идентификаторов должна быть заполнена информации, которая позволила бы говорить о том, что все ее ячейки являются пустыми. Этот метод весьма эффективен, поскольку как время размещения элемента в таблице, так и время его поиска определяются только время, затрачиваемое на вычисления хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для многократных сравнений элементов таблицы. Метод имеет два очевидные недостатки. Первый из них - неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен отвечать всей области значений хэш-функции, тогда как идентификаторов, реально хранятся в таблице, может быть существенно меньше. Второй недостаток - необходимость соответствующего разумного выбора хеш-функции. Этот недостаток является настолько существенным, что не позволяет непосредственно использовать хеш адресацию для организации таблиц идентификаторов.

Проблема выбора хеш-функции не имеет универсального решения. Хеширования обычно происходит за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций. простой хэш-функцией для символа код внутреннего представления в компьютере буквы символа. Это хэш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке.

Очевидно, что такая примитивная хэш-функция будет неудовлетворительной: при ее использовании возникнет проблема - двум различным идентификаторам, что начинаются с одной и той же буквы, отвечать одно и то же значение хеш функции. Тогда при хэш-адресации в одну и ту же ячейку таблицы идентификаторов должны быть помещены два разных идентификаторы, явно невозможно. такая ситуация, когда двум или более идентификаторам соответствует одно и то же значение хэш-функции, называется *коллизией.*

Естественно, что хэш-функция, допускает коллизии, не может быть использована для хэш-адресации в таблице идентификаторов. причем достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов, чтобы такой хэш-функцией нельзя было пользоваться. Но возможно построить хеш-функцию, которая полностью исключала возникновение коллизий? для полного исключения коллизий хеш-функция должна быть взаимно однозначным: каждому элементу из области определения хэш-функции должно соответствовать друг значение с ее множества значений, и наоборот - каждому значению из множества значений этой функции должен соответствовать только один элемент из ее области определение. Тогда любым двум произвольным элементам из области определения хэш-функции всегда будут отвечать два различных ее значения. теоретически для идентификаторов такую хэш-функцию построить нельзя, поскольку и область определения хеш-функции (все возможные имена идентификаторов), и область этих значений (цели неотрицательные числа) бесконечны счетными множествами, поэтому можно организовать взаемнооднозначне отображения одного множества на другую.

Но на практике существует ограничение, что делает создание взаемнооднозначнои хэш-функции для идентификаторов невозможным. Дело в том, что в реальности область значений любой хеш-функции ограничена размером доступного адресного пространства компьютера. Множество адресов любого компьютера с традиционной архитектурой может быть большая, но всегда конечна. Организовать взаимно однозначное отображение бесконечной множества на конечное даже теоретически невозможно. Можно, конечно, учесть, что длина части имени идентификатора, принимается во внимание, в реальных компиляторах на практике также ограничено - обычно она лежит в пределах от 32 до 128 символов (то есть и область определения хеш-функции конечная). Но и тогда количество элементов в конечном множестве, что составляет область определения хеш функции, превышать их количество в конечном множестве области ее значений (Количество всех возможных идентификаторов больше количества допустимых адресов в современных компьютерах). Таким образом, создать взаимно однозначное хеш

функцию на практике невозможно. Следовательно, невозможно избежать возникновения коллизий.

Поэтому нельзя организовать таблицу идентификаторов непосредственно на основе одной только хеш-адресации. Но существуют методы, позволяющие использовать хэш-функции для организации таблиц идентификаторов даже при наличии коллизий.

Хэш-адресация с рехешуванням

Для решения проблемы коллизии можно использовать много способов. Одним из них является метод *рехешування (* или расстановки). Согласно этому методу, если для элемента A адрес n0 = h (A), исчисленная с помощью хэш-функции h, указывает на уже занятую ячейку, то необходимо вычислить значение функции n1 = h1 (A) и проверить занятость ячейки по адресу n1. Если и она занята, то вычисляется значение h2 (A), и так до тех пор, пока либо не будет найдена свободная ячейка, или очередное значение hi (A) не совпадет с h (A). В последнем случае считается, что таблица идентификаторов заполнена и места в ней больше нет - выдается информация об ошибке размещения идентификатора в таблице. тогда поиск элемента A в таблицы идентификаторов, организованной таким образом, выполняться но следующему алгоритму:

1. Вычислить значение хэш-функции n = h (A) для искомого элемента *A.*

2. Если ячейка по адресу n пустая, то элемент не найден, алгоритм завершен, иначе необходимо сравнить имя элемента в ячейки n с именем искомого элемента *A.* если они совпадают, то элемент найден и алгоритм завершен, иначе i: = 1 и перейти к шагу 3.

3. Вычислить ni = hi (A). Если ячейка по адресу N и пустая или n = ni, то элемент не найден и алгоритм завершен, иначе - сравнить имя элемента в ячейке ni с именем искомого элемента *A.* если они совпадают, то элемент найден и алгоритм завершен, иначе i = i + 1 и повторить шаг 3.

Алгоритмы размещения и поиска элемента похожи по выполняемым операциях. Поэтому они будут иметь одинаковые оценки времени, необходимого для их

выполнения.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм помещает элементы в пустые ячейки таблицы, выбирая их определенным образом. При этом элементы могут попадать в ячейки с адресами, которые затем совпадать со значениями хэш-функции, приведет к возникновению новых, дополнительных коллизий. Таким образом, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненной таблицы.

Для организации таблицы идентификаторов по методу рехешування необходимо определить все хэш-функции hi для всех i. Чаще всего функции hi определяют как некоторые модификации хеш-функций h. Например, самым простым методом вычисления функции hi (A) есть ее организация в виде hi (A) = (h (A) + p i) modNm, где p i - некоторое вычисляемое целое число, а Nm - максимальное значение из области значений хэш-функции h. В свою очередь, простым подходом здесь будет положить p i = i. Тогда получаем формулу hi (A) = (h (A) + i) modNm. В этом случае при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хеш-функцией h (A).

Этот способ нельзя признать особенно удачным: при совпадении хэш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Но даже такой примитивный метод рехешування является достаточно эффективным средством организации таблиц идентификаторов при неполном заполнении таблицы. Среднее время на добавление одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехешування. Одним из таких методов является использование в качестве pi для функции hi (A) = (h (A) + p i) modNm последовательности псевдослучайных чисел P1, P2 ... p k. При хорошем выборе генератора псевдослучайных чисел длина последовательности k = Nm .

Существуют и другие методы организации функций рехешування hi (A), основанные на квадратичных вычислениях или, например, на исчислении подъем по формуле: hi (A) = (h (A) N • i) mod N'm где N'm -найближче простое число, меньше

Nm. В целом рехешування позволяет добиться неплохих результатов для эффективного поиска элемента в таблице (лучших, чем бинарный поиск и бинарное дерево), но эффективность метода сильно зависит от заполненной таблицы идентификаторов и качества используемой хэш-функции - чем реже возникают коллизии, тем выше эффективность метода. Требование неполного заполнения таблицы ведет к неэффективному использованию объема доступной памяти.

Оценки времени размещения и поиска элемента в таблицах идентификаторов при использовании различных методов рехешування можно найти в [1,3,7].

Хэш-адресация с использованием метода цепочек

Неполное заполнение таблицы идентификаторов при применении рехешування ведет к неэффективному использованию всего объема памяти, доступного компилятору. Причем объем неиспользуемой памяти будет тем выше, чем больше информации хранится для каждого идентификатора. этого недостатка можно избежать, если дополнить таблицу идентификаторов некоторой промежуточной хеш-таблицей.

В ячейках хэш-таблицы может храниться или пустое значение, или значение указателя на некоторую область памяти с основной таблицы идентификаторов. Тогда хеш-функция вычисляет адрес, по которому происходит обращение сначала к хеш-таблицы, а затем уже через нее с найденной адресу - к самой таблицы идентификаторов. Если соответствующий элемент таблицы идентификаторов пустой, то ячейка хэш-таблицы будет содержать пустое значение. Тогда совсем не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции - таблицу можно сделать динамичной, так чтобы ее объем рос по мере заполнения (сначала таблица идентификаторов не содержит ячейки, а все ячейки хеш-таблицы имеют пустое значение).

Такой подход позволяет добиться двух положительных результатов: во-первых, нет необходимости заполнять пустыми значениями таблицу идентификаторов - это можно сделать только для хеш-таблицы; во-вторых, каждому идентификатору отвечать строго одна ячейка в таблице идентификаторов.

Пустые ячейки в таком случае будут только в хэш-таблице, и объем неиспользуемой памяти не будет зависеть от объема информации, хранящейся для каждого идентификатора - для каждого значения хэш-функции будет расходоваться только память, необходимая для хранения одного указателя на основную таблицу идентификаторов.

На основе этой схемы можно реализовать еще один способ организации таблиц идентификаторов с помощью хэш-функции, который называется *методом цепочек.* В этом случае в таблицу идентификаторов для каждого элемента добавляется еще одно поле, в котором может содержаться ссылки на любой ячейку таблицы. Сначала это поле всегда пустое (никуда не указывает). также необходимо иметь одну специальную переменную, которая всегда указывает на первый свободный элемент основной таблицы идентификаторов (сначала она указывает на начало таблицы). Метод цепочек работает по следующему алгоритму:

1. Во все ячейки хэш-таблицы поместить пустое значение, таблица идентификаторов пуста, переменная FreePtr (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов.

2. Вычислить значение хэш-функции n для нового элемента A. Если ячейка хэш-таблицы по адресу n пустая, то поместить в нее значение переменной *FreePtr* и перейти к шагу 5; иначе перейти к шагу 3.

3. Выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m и перейти к шагу 4.

4. Для элемента таблицы идентификаторов по адресу m проверить значение поля ссылки. Если оно пустое, то записать в него адрес переменной *FreePtr* и перейти к шагу 5; иначе выбрать из поля ссылки новую адрес m и повторить шаг 4.

5.Додаты в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента *A (* поле ссылка должна быть пустым), в переменную *FreePtr* поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо поместить в таблицу, то выполнение алгоритма закончено, иначе перейти к шагу 2.

Поиск элемента в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму:

1. Вычислить значение хэш-функции n для искомого элемента *A*. если ячейка хэш-таблицы по адресу n пустая, то элемент не найден и алгоритм завершен, иначе выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m.

2. Сравнить имя элемента в ячейке таблицы идентификаторов по адресу m с именем искомого элемента A. Если они совпадают, то искомый элемент найден и алгоритм завершен, иначе перейти к шагу 3.

3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе выбрать из поля ссылки адрес m и перейти к шагу 2.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм помещает элементы в элементы таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые затем будут совпадать со значениями хэш-функции. Таким образом, дополнительные коллизии не возникают. В результате в таблице возникают своеобразные цепочки связанных элементов, откуда и происходит название данного метода - «Метод цепочек».

На рис. 1.2 проиллюстрировано заполнения хеш-таблицы и таблицы идентификаторов для ряда идентификаторов: A1, A2, A3, A4, A5 если только h (A1) = h (A2) = h (A5) = n1; h (A3) = n2; h (A4) = n4. После размещения в таблице для поиска идентификатора A1 потребуется одно сравнение, для A2 - два сравнения, для A3 -одно сравнения, для A4 - одно сравнение и для A5 - три сравнения (попробуйте сравнить эти данные с результатами, полученными с использованием простого рехешування для тех же идентификаторов).

Метод цепочек является очень эффективным средством организации таблиц идентификаторов. Среднее время на размещение одного элемента и на поиск элемента в таблице для него зависит только от среднего числа коллизий,

возникающие при исчислении хэш-функции. Накладные расходы памяти, связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый ее элемент, можно признать вполне оправданными, поскольку возникает экономия используемой памяти за счет промежуточной хеш-таблицы. этот метод позволяет экономнее использовать память, но требует организации работы с динамическими массивами данных.

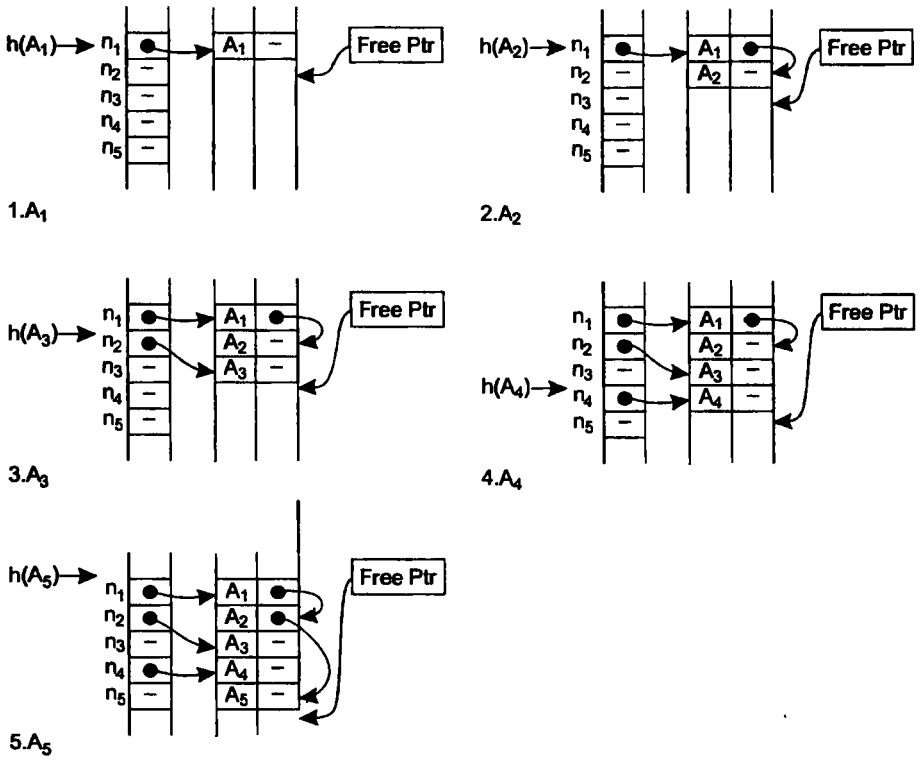


Рис. 1.2. Заполнение таблицы идентификаторов при использовании метода цепочек

Комбинированные способы построения таблиц идентификаторов

Кроме рехешування и метода цепочек можно использовать комбинированные методы для организации таблиц идентификаторов с помощью хеш адресации. В этом случае для исключения коллизий хеш-адресация сочетается с одним из ранее рассмотренных методов - простым списком, упорядоченным списком или бинарным деревом, который используется как дополнительный метод упорядочивания идентификаторов, для которых возникают коллизии. причем, поскольку при качественном выборе хеш-функции количество коллизий обычно невелика (Единицы или десяти случаев), даже простой список может быть вполне

удовлетворительным решением при использовании комбинированного метода.

При таком подходе возможны два варианта: в первом случае, как и для метода цепочек, в таблице идентификаторов организуется специальное дополнительное поле ссылки. Но в отличие от метода цепочек оно имеет несколько иное значение: при отсутствии коллизий для выборки информации из таблицы используется хэш-функция, поле ссылки остается пустым. если же возникает коллизия, то через поле ссылки организуется поиск идентификаторов, для которых значения хэш-функции совпадают, - это поле должно указывать на структуру данных для дополнительного метода: начало списка, первый элемент динамического массива или корневой элемент дерева.

Во втором случае используется хеш-таблица, аналогичная хеш таблицы для метода цепочек. Если по данному адресу хэш-функции идентификатор отсутствует, то ячейка хэш-таблицы пустой. когда появляется идентификатор с данным значением хеш-функции, то создается соответствующая структура для дополнительного метода, в хэш-таблицу записывается ссылка на эту структуру, а идентификатор помещается в созданную структуру по правилам выбранного дополнительного метода.

В первом варианте при отсутствии коллизии поиск выполняется быстрее, но второй вариант предпочтительнее, так как за счет использования промежуточной хеш-таблицы обеспечивается более эффективное использование памяти.

Как и для метода цепочек, для комбинированных методов размещении и время поиска элемента в таблице идентификаторов зависит только от среднего числа коллизий, возникающих при исчислении хэш-функции. накладные расходы памяти при использовании промежуточной хеш-таблицы минимальны. Очевидно, что если как дополнительный метод использовать простой список, то получится алгоритм, полностью аналогичен методу цепочек. Если же использовать упорядоченный список или бинарное дерево, то метод цепочек и комбинированные методы будут примерно равную эффективность при незначительном числе коллизий (Единичные случаи), но с ростом количества коллизий эффективность комбинированных методов по сравнению с методом цепочек расти.

Недостатком комбинированных методов является сложная организация алгоритмов поиска и размещения идентификаторов, необходимость работы с динамически распределяемыми областями памяти, а также большие затраты времени на размещение нового элемента в таблице идентификаторов по сравнению с методом цепочек.

То, какой конкретно метод применяется в компиляторе для организации таблиц идентификаторов, зависит от реализации компилятора. Один и тот же компилятор может иметь даже несколько разных таблиц идентификаторов, организованных на основе различных методов. Как правило, применяются комбинированные методы. Создание эффективной хэш-функции - это отдельная задача разработчиков компиляторов, и полученные результаты, как правило, держатся в секрете. хорошая хэш-функция распределяет идентификаторы, что на ее вход, равномерно по адресам, чтобы свести к минимуму количество коллизий. В настоящее время существует множество хеш-функций, но, как было показано выше, идеального хеширования достичь невозможно.

Хэш-адресация - это метод, который применяется не только для организации таблиц идентификаторов в компиляторах. Данный метод нашел свое применение и в операционных системах, и в системах управления базами данных [5,6,11].

Требования к выполнению работы

Порядок выполнения работы

Во всех вариантах задачи необходимо разработать программу, которая может обеспечить сравнение двух способов организации таблицы идентификаторов по помощью хеш-адресации. Для сравнения предлагаются способы, основанные на использовании рехеширования или комбинированных методов. программа должна считывать идентификаторы из входного файла, размещать их в таблицах по помощью заданных методов и выполнять поиск указанных идентификаторов на требованию пользователя. В процессе размещения и поиска идентификаторов в таблицах программа должна подсчитывать среднее число выполненных операций сравнения для сопоставления эффективности используемых методов. для организации таблиц предлагается использовать простую хеш-функцию, которую разработчик

программы должен выбрать самостоятельно. Хэш-функция должна обеспечивать работу не менее чем из 200 идентификаторами, допустимая длина идентификатора должна быть не менее 32 символов. запрещается использовать в работе хеш-функции, взятые на примере выполнения работы.

Лабораторная работа должна выполняться в следующем порядке:

1. Получить вариант задания у преподавателя.

2. Выбрать и описать хеш-функцию.

3. Описать структуры данных, используемые для заданных методов организации таблиц идентификаторов.

4. Подготовить и защитить отчет.

5. Написать и отладить программу на ЭВМ.

6. Сдать работающую программу преподавателю.

Требования к оформлению отчета

Отчет по лабораторной работе должен содержать следующие разделы:

Об задачи по лабораторной работе;

1) описание выбранной хэш-функции;

2) схемы организации таблиц идентификаторов (согласно варианту Задание);

3) описание алгоритмов поиска в таблицах идентификаторов (в соответствии с варианта задания);

4) текст программы (оформляется после выполнения программы на ЭВМ);

5) результаты обработки заданного набора идентификаторов (входного файла) с помощью методов организации таблиц идентификаторов, указанных в варианте Задание;

6) анализ эффективности используемых методов организации таблиц идентификаторов и выводы по проделанной работе.

Основные контрольные вопросы

- Что такое таблица символов и для чего она предназначена? которая информация может храниться в таблице символов?
- Какие цели преследуются при организации таблицы символов?
- Какими характеристиками могут обладать лексические элементы начальной программы? Какие характеристики являются обязательными?
- Какие существуют способы организации таблиц символов?
- В чем заключается метод логарифмического поиска? какие преимущества он дает по сравнению с простым перебором и которые он имеет недостатки?
- Расскажите о древовидную организацию таблиц идентификаторов. В чем ее преимущества и недостатки?
- Что такое хэш-функции и для чего они используются? В чем суть хеш-адресации?
- Что такое коллизия? Почему она происходит? Можно ли полностью избежать коллизии?
- Что такое рехешування? Методы рехешування существуют?
- Расскажите о преимуществах и недостатки организации таблиц идентификаторов с помощью хэш-адресации и рехешування.
- В чем заключается метод цепочек?
- Расскажите о преимуществах и недостатки организации таблиц идентификаторов с помощью хэш-адресации и метода цепочек.
- Как могут быть скомбинированы различные методы организации хеш-таблиц?
- Расскажите о преимуществах и недостатки организации таблиц идентификаторов с помощью комбинированных методов.

варианты заданий

В табл. 1.1 перечислены методы организации таблиц идентификаторов, используемые в задачах.

Таблица 1.1. методы организации таблиц идентификаторов

№ метода Способ решения коллизий

1	простое рехешування
2	Рехешування с использованием псевдослучайных чисел
3	Рехешування с помощью произведения
4	метод цепочек
5	простой список
6	упорядоченный список
7	бинарное дерево

В табл. 1.2 данные варианты заданий на основе методов организации таблиц идентификаторов, перечисленных в табл. 1.1.

Таблица 1-2-Вариантов задач

№ варианта	первый метод организации таблицы	второй метод организации таблицы
1	1	5
2	1	6
3	1	7
4	2	1
5	2	5
6	2	6
7	3	5
8	3	6
9	3	7
10	7	5
11	4	6
12	4	7
13	1	4
14	2	4
15	3	4
16	2	3

Пример выполнения работы

Задача например

В качестве примера выполнения лабораторной работы возьмем сопоставления двух методов: хэш-адресации с рехешуванням на основе псевдослучайных чисел и комбинации хеш-адресации с бинарным деревом. Если обратиться к приведенной выше таб. 1.1. то такой вариант задания отвечать комбинации методов 2 и 7 (в табл. 1.2 среди вариантов задач такая комбинация отсутствует).

Выбор и описание хэш-функции

Для хэш-адресации с рехешуванням как хэш-функция возьмем функцию, которая будет получать на входе строку, а в результате выдавать сумму кодов первого, среднего и последнего элементов строки. Причем если строка содержит меньше трех символов, то один и тот же символ будет взят и первого, и как средний, и как последний. Будем считать, что прописные и строчные буквы в идентификаторах разные. Как коды символов возьмем коды таблицы ASCII, которая используется в вычислительных системах на базе ОС типа Microsoft Windows Тогда, если положить, что строка из области определения хэш-функции содержит только цифры и буквы английского алфавита, то минимальным значением хэш-функции будет сумма трех кодов цифры «0», а максимальным значением -сумма трех кодов буквы «Z».

Таким образом, область значений выбранной хэш-функции в терминах языка Object Pascal может быть описана как:

(Ord ('0') + Ord ('0') + Ord ('0')). (Ord ('z') + Ord ('z') + Ord ('z'))

Диапазон области значений составляет 223 элементы, удовлетворяет требованиям задача (не менее 200 элементов). Длина входных идентификаторов в данном случае ничем не ограничена. Для удобства пользования опишем две константы, задающие пределы области значений хэш-функции;

HASH_MIN = Ord ('0') + Ord ('0') + Ord ('0').

HASH_MAX = Ord ('z') + Ord ('z') + Ord ('z').

Сама хэш-функция без учета рехеширования вычислять

следующее выражение:

Ord (sName [1]) + Ord (sName [(Length (sName) + 1div2]) + Ord (sName [Len

gth (sName)]

здесь sName - это входная строка (аргумент хэш-функции).

для рехешування возьмем простой генератор последовательности псевдослучайных чисел, построенный на основе формулы F = i • H1 mod H2, где H1 и H2- простые числа, выбранные так, чтобы H1 было в диапазоне от H2 / 2 до H2. Причем, чтобы этот генератор выдавал максимально длинную последовательность во всем диапазоне от HASH_MIN к HASH_MAX, H2 должно быть максимально близко до величины HASH_MAX-HASH_MIN + 1 В данном случае диапазон содержит 223 элементы, и поскольку 223 - простое число, то возьмем H2 = 223 (если бы размер диапазона не был простым числом, то как H2 нужно было бы взять ближе всего к него меньше простое число). Как H1 возьмем 127: H1- = 127. опишем соответствующие константы:

REHASH = 127;

REHASH = 223;

Тогда хэш-функция с учетом рехеширования иметь следующий вид:

Function VarHash (const sName: string; iNum: integer): longint.

Begin

Result: = (Ord (sName [1]) + Ord (sName [(Length (sName) 1) div2]) + Ord (sName [Length (sName)]) - HASH_MIN + iNum * REHASH1modREHAsH2)
mod (HASH_MAX-HASH_MIN + 1) + HASH_MIN; end;

Входные параметры этой функции: sName - имя хешируемого идентификатора. iNum - индекс рехешуванняя (если iNum = 0, то рехьширование отсутствует). Строка проверки величины результата (Result <HASH_MIN) добавлен, чтобы исключить ошибки в тех случаях, когда на вход функции подается строка, содержащего символы вне диапазона '0' .. 'z' (поскольку контроль входных идентификаторов отсутствует, это имеет смысл).

Для комбинации хеш-адресации и бинарного дерева можно использовать простую хеш-функцию - сумму кодов первого и среднего символов входного строки. Диапазон значений такой хэш-функции в терминах языка Object Pascal будет выглядеть так:

(Ord ('0') + Ord ('0')) .. (Ord ('z') + Ord ('z'))

Этот диапазон содержит менее 200 элементов, однако функция удовлетворять требованиям задания, поскольку в сочетании с бинарным деревом она будет обеспечивать обработку неограниченного количества идентификаторов (Максимальное количество идентификаторов будет ограничена только объемом доступной оперативной памяти компьютера).

Без применения рехешнрования эта хэш-функция будет выглядеть значительно проще, чем описанная выше хэш-функция с учетом рехешнрования:

```
function VarHash (const sName: string): longint. begin
Result: = (Ord (sName [1]) + Ord (sName [(Length (sName) + 1div2]) -
HASH_MIN) mod (HASH_MAX-HASH_MIN + 1) + HASH_MIN;
If Result <HASH_MIN then Result: = HASH_MIN;
end.
```

Описание структур данных таблиц идентификаторов

В первую очередь необходимо описать структуру данных, которая будет использована для хранения информации об идентификаторах в таблицах идентификаторов. для обеих таблиц (с рехешуванням на основе генератора псевдослучайных чисел и в комбинации с бинарным деревом) будем использовать одну и ту же структуру. В этом случае в таблицах будут храниться неиспользуемые данные, но программный код будет проще. В качестве учебного примера такой подход оправдан.

Структура данных таблицы идентификаторов (назовем ее TVarInfo) должна содержать в обязательном порядке поле имени идентификатора (поле sName: string), а также поля дополнительной информации о идентификатор на усмотрение разработчиков компилятора. В лабораторной работе не предусмотрено хранение какой-либо

дополнительной информации об идентификаторах, потому как иллюстрация информационного поля включим в структуру TVarInfo дополнительную информационную структуру TAddVarInfo (поле pInfo: TAddVarInfo). Поскольку в языке Object Pascal для полей и переменных, описанных как class, хранятся только ссылки на соответствующую структуру, такой подход не приведет к значительным затратам памяти, но позволит в будущем сохранять любую информацию, связанную с идентификатором, в отдельной структуре данных (поскольку предполагается использовать создаваемые программные модули в последующих лабораторных работах). В данном случае другой подход невозможен, поскольку заранее не известно, какие данные необходимо будет хранить в таблицах идентификаторов. Но разработчик реального компилятора, как правило, знает, какую информацию нужно хранить, и может использовать другой подход - непосредственно включить все необходимые поля в структуру данных таблицы идентификаторов (в данном случае - в структуру TVarInfo) без использования промежуточных структур данных и ссылок.

Первый подход, реализованный в данном примере, обеспечивает экономнее использование оперативной памяти, но более сложный и требует работы с динамическими структурами, второй подход проще в реализации, но менее экономно использует память. Какой из двух подходов выбрать, решает разработчик компилятора в каждом конкретном случае (второй подход будет проиллюстрирован позже в примере к лабораторной работе №4). Для работы с структурой данных TVarInfo потребуются следующие функции:

- функция создания структуры данных и освобождения занимаемой памяти - реализованы как constructor Create и destructor Destroy;
- функции доступа к дополнительной информации - в данной реализации это procedure SetInfo и procedure ClearInfo.

Эти функции будут общими для таблицы идентификаторов с рехешуванням и для комбинированной таблице идентификаторов.

Однако для комбинированной таблицы идентификаторов в структуру данных TVarInfo нужно будет также включить дополнительные поля данных и функции, обеспечивают организацию бинарного дерева:

- ссылки на левую («меньше») и правую («большую») ветку дерева - реализованы как поля данных minEl, maxE1: TVarInfo;

- функцию добавления элемента в дерево - function AddElCnt и function AddElem;

- функцию поиска элемента в дереве - function FindElCnt и function FindElem;

- функция очистки информационных полей во всем дереве - procedure ClearAllInfo;

- функция вывода содержимого бинарного дерева в одну строку (для получения списка всех идентификаторов) - function GetElList.

Функции поиска и размещения элемента в дереве реализованы в двух экземплярах, поскольку одна из них выполняет подсчет количества сравнений, а другая - нет.

Поскольку на функции и процедуры не расходуется оперативная память, в результате получилось, что при использовании одной и той же структуры данных для различных таблиц идентификаторов в таблице с рехешуванням расходоваться неиспользуемая память только на хранение двух лишних ссылок (minE1 и maxEl).

Полностью вся структура данных TVarInfo и связанные с ней процедуры и функции описаны в программном модуле TblElem. Полный текст этого программного модуля приведен в листинге П3.1 в приложении 3.

Надо обратить внимание на один важный момент в реализации функции поиска идентификатора в дереве (function TVarInfo.FindElCnt). если выполнять сравнение двух строк (в данном случае - имени искомого идентификатора sN и имени идентификатора в текущем узле дерева sName) с помощью стандартных методов сравнения строк языка Object Pascal, то фрагмент программного кода выглядел бы примерно так:

```
If sN <sName then begin
...
end
```

```
else
if sH> sName then
begin
....
end else ...
```

В этом фрагменте сравнения строк выполняется дважды: сначала проверяется отношение «меньше» (sN <sName), а затем - «больше» (sN> SNAM). И хотя в программном коде явно это не указано, для каждого из этих операторов будет вызвана библиотечная функция сравнения строк (т.е. операция сравнения может исполниться дважды!). Чтобы этого избежать, в реализации предложенной в примере, выполняется явный вызов функции сравнения строк, а затем обрабатывается полученный результат:

```
i = SlrComp (PChar (sH), PChar (sName)):
If i <0 tnen
Begin
....
end
else
if i> 0 then
begin
... ..
end
else ...
```

В таком варианте дважды может быть выполнено только сравнения целого числа с нулем, а сравнение строк всегда выполняется только один раз, что существенно увеличивает эффективность процедуры поиска.

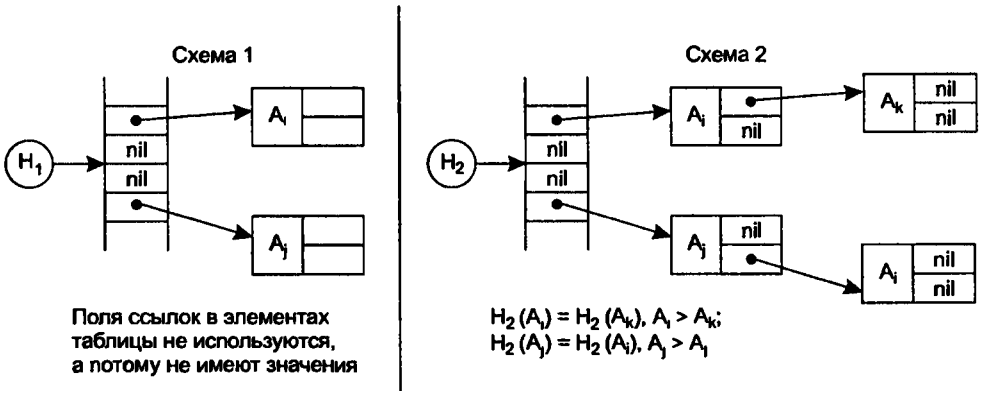
Организация таблиц идентификаторов

Таблицы идентификаторов реализованы в виде статических массивов

размером HASH_MIN .. HASH_MAX, элементами которых являются структуры данных типа TVarInfo. В языке Object Pascal, как было сказано выше, для структур таких типов хранятся ссылки. Поэтому для обозначения пустых ячеек в таблицах идентификаторов использоваться пустое ссылки - nil.

Поскольку в памяти хранятся ссылки, описаны массивы играть роль хэш-таблиц, ссылки с которых указывают непосредственно на информацию в таблицах идентификаторов.

На рис. 1.3 показаны условные схемы, наглядно иллюстрируют организацию таблиц идентификаторов. Схема 1 иллюстрирует таблицу идентификаторов с рехешурованням на основе генератора псевдослучайных чисел, схема 2 - таблицу идентификаторов на основе комбинации хеш-адресации с бинарным деревом. Ячейки с надписью «nil» соответствуют пустым ячейки хеш таблицы.



I

H1 и H2 - соответствующие хэш-функции

Рис. 1.3. Схемы организации таблиц идентификаторов

Для каждой таблицы идентификаторов реализованы следующие функции:

- функцию начальной инициализации хеш-таблицы - InitTreeVar и InitHashVar;

- функции освобождения памяти хеш-таблицы - ClearTreeVar и ClearHashVar;

- функцию удаления дополнительной информации в таблице - ClearTreeInfo и ClearHashInfo;

- функции добавления элемента в таблицу идентификаторов - AddTreeVar и AddHashVar;

- функции поиска элемента в таблице идентификаторов - GetTreeVar и GetHashVar;

- функции, возвращающие количество выполненных операций сравнения при размещении или поиска элемента в таблице - GetTreeCount и GetHashCount.

Алгоритмы поиска и размещения идентификаторов для двух данных методов организации таблиц были описаны выше в разделе «Краткие теоретические сведения», поэтому приводить их здесь повторно нет смысла. Они реализованы в виде четырех перечисленных выше функциями (AddTreeVar и AddHashVar - для размещения элемента; GetTreeVar и GetHashVar - для поиска элемента). функции поиска и размещения элементов в таблице как результат возвращают ссылки на элемент таблицы (структура которого описана в модуле TblElem) в случае успешного исполнения и нулевое ссылки - иначе.

Надо отметить, что функции размещения идентификатора в таблице организованы таким образом, что если на момент добавления нового идентификатора в таблице уже есть идентификатор с таким же именем, то функция не добавляет новый идентификатор в таблицу, а возвращает как результат ссылки на ранее помещен в таблицу идентификатор. Таким образом, в таблице не может быть два и больше идентификатора с одинаковым именем. При этом наличие одинаковых идентификаторов во входном файле не воспринимается как ошибка - это допустимо, поскольку в задании не предусмотрено ограничение на наличие совпадающих имен идентификаторов.

Все перечисленные функции описаны в двух программных модулях: FncHash - для таблицы идентификаторов, построенной на основе рехешування с использованием генератора псевдослучайных чисел, и FncTree - для таблицы идентификаторов, построенной на основе комбинации хеш-адресации и бинарного дерева. Кроме массивов данных для организации таблиц идентификаторов и функций работы с ними эти модули содержат описание переменных, используемых для подсчета количества выполненных операций сравнения при размещении и поиске

идентификатора в таблицах.

Полные тексты обоих модулей (FncHash и FncTree) можно найти на сайте издательства, в файлах FncHash.pas и FncTree.pas. Кроме того, текст модули FncTree приведен в листинге П3.2 в приложении 3.

Хочется обратить внимание на то, что в разделах инициализации (initialisation) обоих модулей вызывается функция начального заполнения таблицы идентификаторов, а в разделах завершения (flna1ization) обох модулей - функция освобождения памяти. Это гарантирует корректную работу модулей при любом порядка вызова остальных функций, поскольку Object Pascal сам обеспечивает своевременный вызов программного кода в разделах инициализации и завершения модулей.

текст программы

Кроме перечисленных выше модулей необходим еще модуль обеспечивает интерфейс с пользователем. Этот модуль (FormLab1) реализует графическое окно TLab1Form на основе класса TForm библиотеки VCL. он обеспечивает интерфейс

- функции поиска элемента в таблице идентификаторов - GetTreeVar и GetHashVar;
- функции, возвращающие количество выполненных операций сравнения при размещении или поиска элемента в таблице - GetTreeCount и GetHashCount.

Алгоритмы поиска и размещения идентификаторов для двух данных методов организации таблиц были описаны выше в разделе «Краткие теоретические сведения», поэтому приводить их здесь еще раз нет смысла. Они реализованы в виде четырех вышеперечисленных функций (AddTreeVar и AddHashVar - для размещения элемента; GetTreeVar и GetHashVar - для поиска элемента). Функции поиска и размещения элементов в таблице как результат возвращают ссылки на элемент таблицы (структура которого описана в модуле TblElem) в случае успешного выполнения и нулевое ссылки - в противном случае.

Надо отметить, что функции размещения идентификатора в таблице организованы таким образом, что если на момент внесения нового идентификатора в таблицы уже есть идентификатор с таким же именем, то функция не добавляет новый идентификатор в таблицу, а возвращает как результат ссылки на ранее внесенный в таблицу идентификатор. Таким образом, в таблице не может быть двух и более идентификаторов с одинаковым именем. При этом наличие одинаковых идентификаторов во входном файле не воспринимается как ошибка - это допустимо, поскольку в задании не предусмотрено ограничение на наличие совпадающих имен идентификаторов.

Все перечисленные функции описаны в двух программных модулях: FncHash - для таблицы идентификаторов, построенной на основе рехешування с использованием генератора псевдослучайных чисел, и FncTree - для таблицы идентификаторов, построенной на основе комбинаций хеш-адресации и бинарного дерева. Кроме массивов данных для организации таблиц идентификаторов и функций работы с ними эти модули содержат описание переменных, используемых для подсчета количества выполненных операций сравнения при размещении и поиске идентификатора в таблицах.

Полные тексты обоих модулей (FncHash и FncTree) можно найти на сайте издательства, в файлах FncHash.pas и FncTree.pas. кроме того, текст модуля FncTree приведен в листинге п3.2 в приложении 3.

Хочется обратить внимание на то, что в разделах инициализации (initia1ization) обоих модулей вызывается функция начального заполнения таблицы идентификаторов, а в разделах завершения (finalization) обоих модулей - функция освобождения памяти. Это гарантирует корректную работу модулей при любом порядка вызова остальных функций, поскольку ObjectPascal сам обеспечивает своевременный вызов программного кода в разделах инициализации и завершения модулей.

текст программы

Кроме перечисленных выше модулей необходимо еще модуль, обеспечивающий интерфейс с пользователем. Этот модуль (FormLabl) реализует графическое окно

TlablForm на базе класса TForm библиотеки VCL. он обеспечивает интерфейс средствами Graphical User Interface (GUI) в ОС типа Windows на основе стандартных органов управления из системных библиотек данной ОС. кроме программного кода (файл FormLabl.pas) модуль включает описание ресурсов пользовательского интерфейса (файл FотLabl.dfll). Подробнее принципы организации пользовательского интерфейса на основе GUI и работы систем программирования с ресурсами интерфейса описаны [3 5. 7 июня].

Кроме описания интерфейсной формы и ее средств управления модуль FотLabl содержит три переменные (iCountNum, iCountHash, iCountTree), служащих для накопление статистических результатов по мере выполнения размещения и поиска идентификаторов в таблицах, а также функцию (Procedure ViewStatistic) для отображения накопленной статистической информации на экране.

Интерфейсная форма описана в модуле, содержит следующие основные органы управления:

- поле ввода имени файла (EditFi1e), кнопка выбора имени файла с каталогов файловой системы (BtnFile), кнопка чтения файла (BtnLoad)
- многострочное поле для отображения прочитанного файла (ListIdents)
- поле ввода имени идентификатора, который надо найти (EditSearch)
- кнопка для поиска введенного идентификатора (BtnSearch) - этой кнопкой однократно вызывается процедура поиска (procedure SearchStr)
- кнопка автоматического поиска всех идентификаторов (BtnA.11Search) - этой кнопкой процедура поиска идентификатора (procedure SearchStr) вызывается циклически для всех считанных из файла идентификаторов (для всех, перечисленных в поле ListIdents)
- кнопка сброса накопленной статистической информации (BtnReset)
- поля для отображения статистической информации;
- кнопка завершения работы с программой (BtnExit).

Внешний вид такой формы приведен на рис. 1.4

Функция чтения содержимого файла с идентификаторами (procedure TLablForm.BtnLoadClick) вызывается щелчком по кнопке BtnLoad. она организована таким образом, что сначала содержимое файла читается в многострочное поле ListIdents, а потом все прочитаны идентификаторы записываются в две таблицы идентификаторов Каждая строка файла считается отдельным идентификатором пробелов начале и в конце строки игнорируются. При ошибке размещения идентификатора в одной из таблиц выдается предупреждение (например, если считан более 223 различных идентификаторов, то рехешування станет невозможным и будет выдано сообщение об ошибке).

функция поиска идентификатора (Procedure TlablForm.SearchStr) вызывается однократно щелчком по кнопке BtnSearch (процедура procedue TlablForm.BtnSearchClick) или многократно щелчком по кнопке BtnA11Search (Процедура procedure TlablForm.SearchStr. BtnA11SearchClick). Поиск идет сразу в двух таблицах, результаты поиска и накопленная статистическая информация отражаются в соответствующих полях.

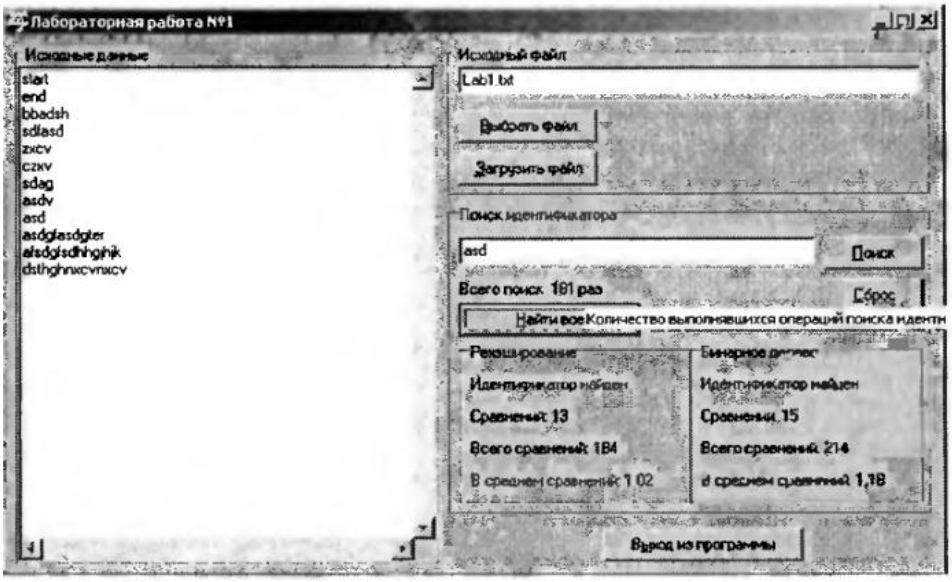


Рис. 1.4. Внешний вид интерфейсной формы для лабораторной работы №1

Полный текст программного кода модуля интерфейса с пользователем и описание ресурсов пользовательского интерфейса находятся в архиве,

расположенном на сайте издательства, в файлах FormLab1.pas и FormLab1.dfm в соответствии.

Полный текст всех программных модулей, реализующих рассмотрен например для лабораторной работы •1, можно найти в архиве, расположенном на сайте, в подкаталогах LABS и COMMON (в подкаталог COMMON вынесены те программные модули, исходный текст которых не зависит от входного языка и задачи по лабораторной работе). главным г. проекта является файn LAB1.PDR в подкаталоге LABS. Кроме того, текст модуля FncTree приведен в листиигу п3.1 в приложении 3.

Выводы по проделанной работе

В результате выполнения написанного программного кода для ряда текстовых файлов было установлено, что при заполнении таблицы идентификаторов до 20% (в 45 идентификаторов) для поиска и размещения идентификатора с использованием рехеширования на основе генератора псевдослучайных чисел в среднем нужно меньшее число сравнений, чем при использовании хэш-адресации в комбинации с бинарным деревом. При заполнении таблицы от 20% то 40% (Примерно 45-90 идентификаторов) оба метода имеют примерно одинаковые показатели, но при заполнении таблицы более чем на 40% (90-223 идентификаторов), эффективность комбинированного метода по сравнению с методом рехеширования резко возрастает. Если на входе есть более 223 идентификаторов, рехешування полностью перестанет работать.

Таким образом, установлено, что комбинированный метод работоспособен даже при наличии простой хэш-функции и дает неплохие результаты (в среднем 3-5 сравнений на входных файлах, 500-700 идентификаторов, содержащих), тогда как метод на основе рехеширонания для реальной работы требует сложной хеш функции с диапазоном значений в несколько тысяч или десятков тысяч.