Лабораторная работа № 1
Организация таблиц идентификаторов

Цель работы: изучить основные методы организации таблиц идентификаторов, получить представление о преимуществах и недостатках, присущих различным методам организации таблиц идентификаторов.
Для выполнения лабораторной работы требуется написать программу, которая получает на входе набор идентификаторов, организует таблицы идентификаторов с 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.

Краткие 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 исходной программы, зависит от типа этого элемента, от его смысла (семантики) и, соответственно, от той роли, которую он исполняет в исходной 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 есть типовые характеристики, которые чаще всего присущи тем или иным элементам исходной программы. Например для переменной – это ее тип и адрес ячейки памяти, для константы – ее значение, для функции – количество и типы формальных аргументов, тип возвращаемого результата, адрес вызова кода функции.
Главной характеристикой любого элемента исходной программы является его имя. Именно с именами переменных, констант, функций и других элементов входного языка оперирует разработчик программы – поэтому и компилятор должен уметь анализировать эти элементы по их именам.
Имя каждого элемента должно быть уникальным. Многие современные языки 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, здесь же будем считать, что имена элементов исходной программы всегда являются уникальными.
Таким образом, задача компилятора заключается в том, чтобы хранить 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 элементе и может пополняться по мере работы компилятора. Количество записей зависит от способа организации таблицы идентификаторов, но в любом случае их не может быть меньше, чем элементов в исходной программе. В принципе, компилятор может работать не с одной, а с несколькими таблицами идентификаторов.

Принципы организации таблиц идентификаторов
Компилятор пополняет записи в таблице идентификаторов по мере анализа исходной программы и обнаружения в ней новых элементов, требующих размещения в таблице. Поиск информации в таблице выполняется всякий раз, когда компилятору необходимы сведения о том или ином элементе программы. Причем следует заметить, что поиск элемента в таблице будет выполняться компилятором существенно чаще, чем помещение 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 будет предшествовать операция поиска – чтобы убедиться, что такого элемента в таблице нет.
На каждую операцию поиска элемента в таблице компилятор будет затрачивать время, и поскольку количество элементов в исходной программе велико (от единиц до сотен тысяч в зависимости от объема программы), это время будет существенно влиять на общее время компиляции. Поэтому таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстро выполнять поиск нужной ему записи таблицы по имени элемента, с которым связана эта запись.
Можно выделить следующие способы организации таблиц идентификаторов:
• простые и упорядоченные списки;
• бинарное дерево;
• хэш-адресация с рехэшированием;
• хэш-адресация по методу цепочек;
• комбинация хэш-адресации со списком или бинарным деревом.
Далее будет дано краткое описание всех вышеперечисленных способов организации таблиц идентификаторов.
Простейшие методы построения таблиц идентификаторов

В простейшем случае таблица идентификаторов представляет собой линейный неупорядоченный список, или массив, каждая ячейка которого содержит данные о соответствующем элементе таблицы. Размещение новых элементов в такой таблице выполняется путем записи информации в очередную ячейку массива или списка по мере обнаружения новых элементов в исходной программе.
Поиск нужного элемента в таблице будет в этом случае выполняться путем последовательного перебора всех элементов и сравнения 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, требуемое на добавление нового элемента в таблицу (Тд), не зависит от числа элементов в таблице (N). Но если N велико, то поиск потребует 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 является неэффективным. Он применим только для самых простых компиляторов, работающих с небольшими программами.
Поиск может быть выполнен более эффективно, если элементы таблицы отсортированы (упорядочены) 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, либо алгоритм не дойдет до очередного блока, содержащего один или два элемента (с которыми можно выполнить прямое сравнение искомого элемента).
Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается в два раза, максимальное число сравнений равно 1 + log2 N. Тогда время поиска элемента в таблице идентификаторов можно оценить как Тп = O(log2 N). Для сравнения: при N = 128 бинарный поиск требует самое большее 8 сравнений, а поиск в неупорядоченной таблице – в среднем 64 сравнения. Метод называют «бинарным поиском», поскольку на каждом шаге объем рассматриваемой 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 упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов зачастую просматривается компилятором еще до того, как она заполнена, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней. Следовательно, для построения такой таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.
Если пользоваться стандартными алгоритмами, применяемыми для организации упорядоченных массивов данных, то среднее время, необходимое на помещение всех элементов в таблицу, можно оценить следующим образом:

$$T_д = O(N \cdot \log_2 N) + k \cdot O(N^2).$$

Здесь k – некоторый коэффициент, отражающий соотношение между временами, затрачиваемыми компьютером на выполнение операции сравнения и операции переноса данных.
При организации логарифмического поиска в таблице идентификаторов обеспечивается существенное сокращение времени поиска нужного элемента за счет увеличения времени на помещение нового элемента в таблицу. Поскольку добавление новых элементов в таблицу идентификаторов происходит существенно реже, чем обращение к ним, этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы. Однако в реальных компиляторах этот метод непосредственно также не используется, поскольку существуют более 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 котором таблица имеет форму бинарного дерева. Каждый узел дерева представляет собой элемент таблицы, причем корневым узлом становится первый элемент, встреченный компилятором при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей. Для определенности будем называть две ветви «правая» и «левая».
Рассмотрим алгоритм заполнения бинарного дерева. Будем считать, что алгоритм работает с потоком входных данных, содержащим идентификаторы. 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 закончено.
2. Сделать текущим узлом дерева корневую вершину.
3. Сравнить имя очередного идентификатора с именем идентификатора, содержащегося 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 у текущего 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 идентификаторе, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.
7. Если у текущего узла существует правая 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 идентификаторе, 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.

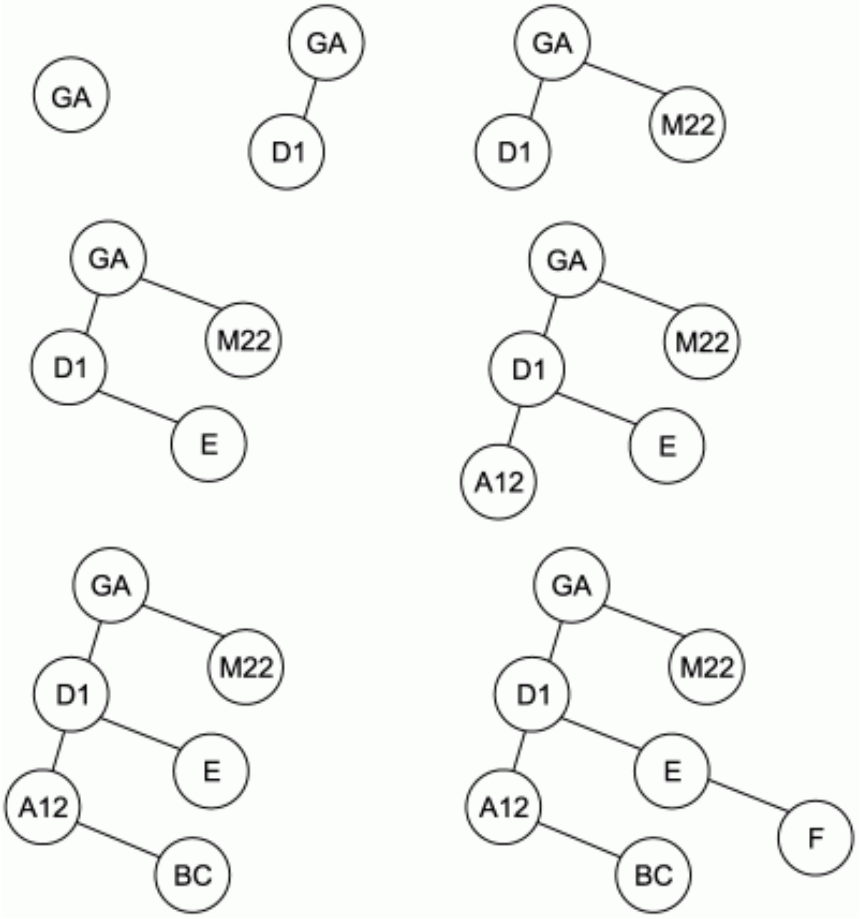

Рис. 1.1. Заполнение бинарного дерева для последовательности идентификаторов.

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, содержащимся 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 у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе – искомый идентификатор не найден, алгоритм завершается.
6. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе – искомый идентификатор не найден, алгоритм завершается.

Для данного метода число требуемых сравнений и форма получившегося дерева 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 две дополнительные ссылки на левую и правую ветви в каждом элементе дерева и работа с динамическим выделением памяти при построении дерева.

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

$$T_д = N \cdot O(\log_2 N);$$

$$T_п = O(\log_2 N).$$

Несмотря на указанные недостатки, метод бинарного дерева является довольно удачным механизмом для организации таблиц идентификаторов. Он нашел свое применение в ряде компиляторов. Иногда компиляторы строят несколько различных деревьев для идентификаторов разных типов и разной длины [1, 2, 3, 7].

Хэш-функции и хэш-адресация

В реальных исходных программах количество идентификаторов столь велико, что даже логарифмическую зависимость времени поиска от их числа нельзя признать удовлетворительной. Необходимы более 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:

$$F(r) = n, r \in \mathbf{R}, n \in \mathbf{Z}$$

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 из множества целых неотрицательных чисел Z:

$$\mathbf{M} \subseteq \mathbf{Z}$$

содержащее все возможные значения, возвращаемые функцией F:

$$\forall r \mathbf{R}: F(r) \in \mathbf{M} \text{ и } \forall m \in \mathbf{M}:$$
$$\exists r \in \mathbf{R}: F(\mathbf{r}) = m$$

Процесс отображения области определения хэш-функции на множество значений называется хэшированием.
При работе с таблицей идентификаторов хэш-функция должна выполнять отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения хэш-функции будет множество всех возможных имен идентификаторов.
Хэш-адресация заключается в использовании значения, возвращаемого хэш-функцией, в качестве адреса ячейки из некоторого массива данных. Тогда размер массива данных должен соответствовать области значений используемой хэш-функции. Следовательно, в реальном компиляторе область значений хэш-функции никак не должна превышать размер доступного адресного пространства компьютера.
Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в помещении каждого элемента таблицы в ячейку, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Тогда в идеальном случае для помещения любого элемента в таблицу идентификаторов достаточно только вычислить его хэш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице также необходимо вычислить хэш-функцию для искомого элемента и проверить, не является ли заданная ею ячейка массива пустой (если она не пуста – элемент найден, если пуста – не найден). Первоначально таблица идентификаторов должна быть заполнена информацией, 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 временем, затрачиваемым на вычисление хэш-функции, которое в общем случае несопоставимо меньше времени, необходимого для многократных сравнений элементов таблицы.
Метод имеет два очевидных недостатка. Первый из них – неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать всей области значений хэш-функции, в то время как реально хранимых в таблице идентификаторов может быть существенно меньше. Второй недостаток – необходимость соответствующего разумного выбора хэш-функции. Этот недостаток является настолько существенным, что не позволяет непосредственно использовать хэш-адресацию для организации таблиц идентификаторов.
Проблема выбора хэш-функции не имеет универсального решения. Хэширование обычно происходит за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций. Самой простой хэш-функцией для символа является код внутреннего представления в компьютере литеры символа. Эту 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, начинающимся с одной и той же буквы, будет соответствовать одно и то же значение хэш-функции. Тогда при хэш-адресации в одну и ту же ячейку таблицы идентификаторов должны быть помещены два различных идентификатора, что явно невозможно. Такая ситуация, когда двум или более идентификаторам соответствует одно и то же значение хэш-функции, называется коллизией.
Естественно, что хэш-функция, допускающая 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 ли построить хэш-функцию, которая бы полностью исключала возникновение коллизий?
Для полного исключения коллизий хэш-функция должна быть взаимно однозначной: каждому элементу из области определения хэш-функции должно соответствовать одно значение из ее множества значений, и наоборот – каждому значению из множества значений этой функции должен соответствовать только один элемент из ее области определения. Тогда любым двум произвольным элементам из области определения хэш-функции будут всегда соответствовать два различных ее значения. Теоретически для идентификаторов такую хэш-функцию построить можно, так как и область определения хэш-функции (все возможные имена идентификаторов), и область ее значений (целые неотрицательные числа) являются бесконечными счетными множествами, поэтому можно организовать взаимно однозначное отображение одного множества на другое.

Но на практике существует ограничение, делающее создание взаимно однозначной хэш-функции для идентификаторов невозможным. Дело в том, что в реальности область значений любой хэш-функции ограничена размером доступного адресного пространства компьютера. Множество адресов любого компьютера с традиционной архитектурой может быть велико, но всегда конечно, то есть ограничено. Организовать взаимно однозначное отображение бесконечного множества на конечное даже теоретически невозможно. Можно, конечно, учесть, что длина принимаемой во внимание части имени идентификатора в реальных компиляторах на практике также ограничена – обычно она лежит в пределах от 32 до 128 символов (то есть и область определения хэш-функции конечна). Но и тогда количество элементов в конечном множестве, составляющем область определения хэш-функции, будет превышать их количество в конечном множестве области ее значений (количество всех возможных идентификаторов больше количества допустимых адресов в современных компьютерах). Таким образом, создать взаимно однозначную хэш-функцию на практике невозможно. Следовательно, невозможно избежать возникновения коллизий.
Поэтому нельзя организовать таблицу идентификаторов непосредственно на основе одной только хэш-адресации. Но существуют методы, позволяющие использовать хэш-функции для организации таблиц идентификаторов даже при наличии коллизий.

Хэш-адресация с рехэшированием
Для решения проблемы коллизии можно использовать много способов. Одним из них является метод рехэширования (или расстановки). Согласно этому методу, если для элемента A адрес n0 = h(A), вычисленный 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, либо 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 таблице идентификаторов, организованной таким образом, будет выполняться по 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 ячейке 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 ni 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 схожи по выполняемым операциям. 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 потом будут совпадать со значениями хэш-функции, что приведет к возникновению новых, дополнительных коллизий. Таким образом, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненности таблицы.
Для организации таблицы идентификаторов по методу рехэширования необходимо определить все хэш-функции hi для всех i. Чаще всего функции hi определяют как некоторые модификации хэш-функции h. Например, самым простым методом вычисления функции hi(A) является ее организация в виде hi(A) = (h(A) + pi) mod Nm, где pi – некоторое вычисляемое целое число, а Nm – максимальное значение из области значений хэш-функции h. В свою очередь, самым простым подходом здесь будет положить pi = i. Тогда получаем формулу hi(A) = (h(A) + i) mod Nm. В этом случае при совпадении значений хэш-функции для каких-либо элементов поиск свободной ячейки в таблице начинается последовательно от текущей позиции, заданной хэш-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 рехэширования является достаточно эффективным средством организации таблиц идентификаторов при неполном заполнении таблицы.
Среднее время на помещение одного элемента в таблицу и на поиск элемента в таблице можно снизить, если применить более совершенный метод рехэширования. Одним из таких методов является использование в качестве pi для функции hi(A) = (h(A) + pi) mod Nm последовательности псевдослучайных целых чисел p1, p2, …, pk. При хорошем выборе генератора псевдослучайных чисел длина последовательности k = Nm.
Существуют и другие методы организации функций рехэширования hi(A), основанные на квадратичных вычислениях или, например, на вычислении произведения по формуле: hi(A) = (h(A)N·i) mod N'm, где N'm – ближайшее простое число, меньшее Nm. В целом рехэширование позволяет добиться неплохих результатов для эффективного поиска элемента в таблице (лучших, чем бинарный поиск и бинарное дерево), но эффективность метода сильно зависит от заполненности 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 рехэширования можно найти в [1, 3, 7].

Хэш-адресация с использованием метода цепочек

Неполное заполнение таблицы идентификаторов при применении рехэширования 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 хэш-таблицей.

В ячейках хэш-таблицы может храниться либо пустое значение, либо значение указателя на некоторую область памяти из основной таблицы идентификаторов. Тогда хэш-функция вычисляет адрес, по которому происходит обращение сначала к хэш-таблице, а потом уже через нее по найденному адресу – к самой таблице идентификаторов. Если соответствующая ячейка таблицы идентификаторов пуста, то ячейка хэш-таблицы будет содержать пустое значение. Тогда вовсе не обязательно иметь в самой таблице идентификаторов ячейку для каждого возможного значения хэш-функции – таблицу можно сделать динамической, так чтобы ее объем рос по мере заполнения (первоначально таблица идентификаторов не содержит ни одной ячейки, а все ячейки хэш-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 хэш-таблицы; во-вторых, каждому идентификатору будет соответствовать 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, хранимой 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, называемый методом цепочек. В этом случае в таблицу идентификаторов для каждого элемента добавляется еще одно поле, в котором может содержаться ссылка на любой элемент таблицы. Первоначально это поле всегда пустое (никуда не указывает). Также необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально 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 ячейки таблицы идентификаторов по адресу m проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной FreePtr и перейти к шагу 5; иначе выбрать из поля ссылки новый адрес m и повторить шаг 4.
5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A (поле ссылки должно 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 ячейки таблицы, связывая их друг с другом последовательно через поле ссылки. При этом элементы не могут попадать в ячейки с адресами, которые потом будут совпадать со значениями хэш-функции. Таким образом, дополнительные коллизии не возникают. В итоге в таблице возникают своеобразные цепочки связанных элементов, откуда и происходит название данного метода – «метод цепочек».
На рис. 1.2 проиллюстрировано заполнение хэш-таблицы и таблицы идентификаторов для ряда идентификаторов: A1, A2, A3, A4, A5 при условии, что 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 сравнение, для A4 – одно сравнение и для A5 – три сравнения (попробуйте сравнить эти данные с результатами, полученными с использованием простого рехэширования 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, возникающих при вычислении хэш-функции. Накладные расходы памяти,

связанные с необходимостью иметь одно дополнительное поле указателя в таблице идентификаторов на каждый ее элемент, можно признать вполне оправданными, так как возникает экономия используемой памяти за счет промежуточной хэш-таблицы. Этот метод позволяет более экономно использовать память, но требует организации работы с динамическими массивами данных.

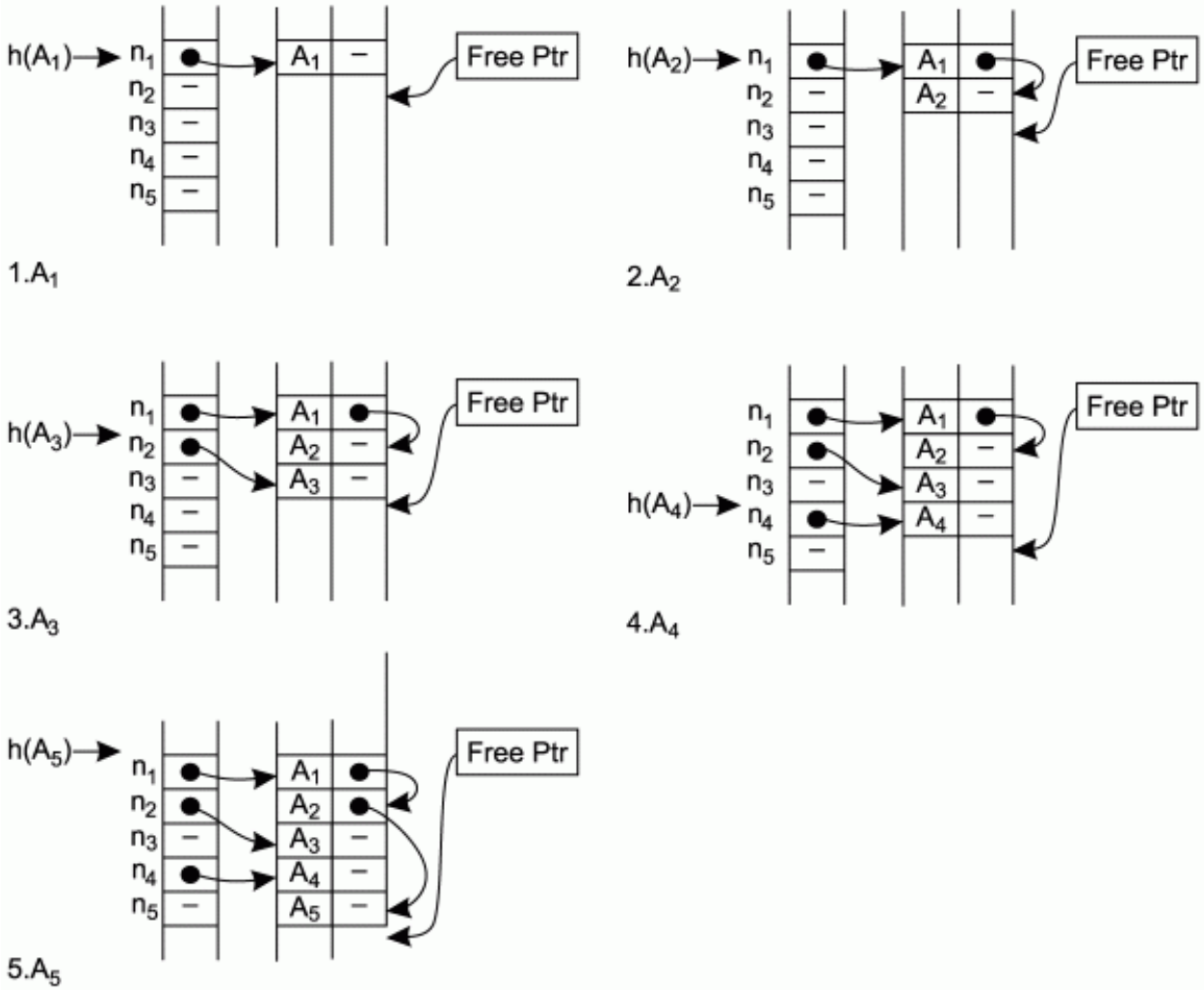

Рис. 1.2. Заполнение таблицы идентификаторов при использовании метода цепочек.

Комбинированные способы построения таблиц идентификаторов
Кроме рехэширования и метода цепочек можно использовать комбинированные методы для организации таблиц идентификаторов с помощью хэш-адресации. В этом случае для исключения коллизий хэш-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 хэш-функции количество коллизий обычно невелико (единицы или десятки 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 хэш-таблица, аналогичная хэш-таблице для метода цепочек. Если по данному адресу хэш-функции идентификатор отсутствует, то ячейка хэш-таблицы пустая. Когда появляется идентификатор с данным значением хэш-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 коллизий поиск выполняется быстрее, но второй вариант предпочтительнее, так как за счет использования промежуточной хэш-таблицы обеспечивается более эффективное использование памяти.

Как и для метода цепочек, для комбинированных методов время размещения и время поиска элемента в таблице идентификаторов зависит только от среднего числа коллизий, возникающих при вычислении хэш-функции. Накладные расходы памяти при использовании промежуточной хэш-таблицы минимальны.

Очевидно, что если в качестве дополнительного метода использовать простой список, то получится алгоритм, полностью аналогичный методу цепочек. Если же использовать упорядоченный список или бинарное дерево, то метод цепочек и комбинированные методы будут иметь примерно равную эффективность при незначительном числе коллизий (единичные случаи), но с ростом количества коллизий эффективность комбинированных методов по сравнению с методом цепочек будет возрастать.

Недостатком комбинированных методов является более сложная организация алгоритмов поиска и размещения идентификаторов, необходимость работы с динамически распределяемыми областями памяти, а также бóльшие 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 поступающие на ее вход идентификаторы равномерно на все имеющиеся в распоряжении адреса, чтобы свести к минимуму количество коллизий. В настоящее время существует множество хэш-функций, но, как было показано выше, идеального хэширования 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 задания требуется разработать программу, которая может обеспечить сравнение двух способов организации таблицы идентификаторов с помощью хэш-адресации. Для сравнения предлагаются способы, основанные на использовании рехэширования или комбинированных методов. Программа должна считывать идентификаторы из входного файла, размещать их в таблицах с помощью заданных методов и выполнять поиск указанных идентификаторов по 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 простейшую хэш-функцию, которую разработчик программы должен выбрать самостоятельно. Хэш-функция должна обеспечивать работу не менее чем с 200 идентификаторами, допустимая длина идентификатора должна быть не менее 32 символов. Запрещается использовать в работе хэш-функции, взятые из примера выполнения работы.
Лабораторная работа должна выполняться в следующем порядке:
1. Получить вариант задания у преподавателя.
2. Выбрать и описать хэш-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:
• задание по лабораторной работе;
• описание выбранной хэш-функции;
• схемы организации таблиц идентификаторов (в соответствии с вариантом задания);

• описание алгоритмов поиска в таблицах идентификаторов (в соответствии с вариантом задания);
• текст программы (оформляется после выполнения программы на ЭВМ);
• результаты обработки заданного набора идентификаторов (входного файла) с помощью методов организации таблиц идентификаторов, указанных в варианте задания;
• анализ эффективности используемых методов организации таблиц идентификаторов и выводы по проделанной работе.

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

Варианты заданий
В табл. 1.1 перечислены методы организации таблиц идентификаторов, используемые в заданиях.
Таблица 1.1. Методы организации таблиц идентификаторов

№ метода	Способ разрешения коллизий
1	Простое рехэширование
2	Рехэширование с использованием псевдослучайных чисел

№ метода	Способ разрешения коллизий
3	Рехэширование с помощью произведения
4	Метод цепочек
5	Простой список
6	Упорядоченный список
7	Бинарное дерево

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

Таблица 1.2. Варианты заданий

№ 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

Задание для примера

В качестве примера выполнения лабораторной работы возьмем сопоставление двух методов: хэш-адресации с рехэшированием на основе псевдослучайных чисел и комбинации хэш-адресации с бинарным деревом. Если обратиться к приведенной выше табл. 1.1, то такой вариант задания будет соответствовать комбинации методов 2 и 7 (в табл. 1.2 среди вариантов заданий такая комбинация отсутствует).

Выбор и описание хэш-функции

Для хэш-адресации с рехэшированием в качестве хэш-функции возьмем функцию, которая будет получать на входе строку, а в результате выдавать сумму кодов первого, среднего и последнего элементов строки. Причем если

строка содержит менее трех символов, то один и тот же символ будет взят и в качестве первого, и в качестве среднего, и в качестве последнего.
Будем считать, что прописные и строчные буквы в идентификаторах различны. В качестве кодов 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 литеры «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 элемента, что удовлетворяет требованиям задания (не менее 200 элементов). Длина входных идентификаторов в данном случае ничем не ограничена. Для удобства пользования опишем две константы, задающие границы 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 рехэширования будет вычислять следующее выражение:
Ord(sName[1]*Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. – СПб.: Питер, 2003. – 396 с.*
) + Ord(sName[(Length(sName)+1) div 2]) + Ord(sName[Length(sName);
здесь sName – это входная строка (аргумент хэш-функции).
Для рехэширования возьмем простейший генератор последовательности псевдослучайных чисел, построенный на основе формулы F = i-H1 mod H2, где H1 и H2 – простые числа, выбранные таким образом, чтобы H1 было в диапазоне от H2/2 до H2. Причем, чтобы этот генератор выдавал максимально длинную последовательность во всем диапазоне от HASH_MIN до HASH_MAX, H2 должно быть максимально близко к величине HASH_MAX – HASH_MIN + 1. В данном случае диапазон содержит 223 элемента, и поскольку 223 – простое число, то возьмем H2 = 223 (если бы размер диапазона не был простым числом, то в качестве H2 нужно было бы взять ближайшее к нему меньшее простое число). В качестве H1 возьмем 127: H1 = 127.
Опишем соответствующие константы:
REHASH1 = 127;
REHASH2 = 223;
Тогда хэш-функция с учетом рехэширования будет иметь следующий вид:
function VarHash(const sName: string; iNum: integer):longint;
begin
Result:=(Ord(sName[1]*Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. – СПб.: Питер, 2003. – 396 с.*
)+Ord(sName[(Length(sName)+1) div 2])

```
+ Ord(sName[Length(sName)]) – HASH_MIN
+ iNum*REHASH1 mod REHASH2)
mod (HASH_MAX-HASH_MIN+1) + HASH_MIN;
if Result < HASH_MIN then Result:= HASH_MIN;
end;
```

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

Для комбинации хэш-адресации и бинарного дерева можно использовать более простую хэш-функцию – сумму кодов первого и среднего символов входной 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 будет удовлетворять требованиям задания, так как в комбинации с бинарным деревом она будет обеспечивать обработку неограниченного количества идентификаторов (максимальное количество идентификаторов будет ограничено только объемом доступной оперативной памяти компьютера).

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

```
function VarHash(const sName: string): longint;
begin
Result:=(Ord(sName[1]*Молчанов А. Ю. Системное программное обеспечение: Учебник для вузов. – СПб.: Питер, 2003. – 396 с.*
)+Ord(sName[(Length(sName)+1) div 2])
– 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 рехэшированием 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 об идентификаторе по

усмотрению разработчиков компилятора. В лабораторной работе не предусмотрено хранение какой-либо дополнительной информации об идентификаторах, поэтому в качестве иллюстрации 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 расходам памяти, но позволит в будущем хранить 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, так как заранее не известно, какие данные необходимо будет хранить в таблицах идентификаторов. Но разработчик реального компилятора, как правило, знает, какую информацию требуется 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 более экономное использование оперативной памяти, но является более сложным и требует работы с динамическими структурами, второй подход более прост 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 со структурой данных TVarInfo потребуются следующие функции:

• функции 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 рехэшированием и для комбинированной таблицы идентификаторов.

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

• ссылки на левую («меньшую») и правую («большую») ветвь дерева – реализованы как поля данных minEl, maxEl: TVarInfo;

• функции добавления элемента в дерево – function AddElCnt и function AddElem;

• функции 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, так как 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 разных таблиц идентификаторов в таблице с рехэшированием будет расходоваться неиспользуемая память только на хранение двух лишних ссылок (minEl 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 sN > sName then
begin
…
end
else…
```

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

```
i:= StrComp(PChar(sN), PChar(sName));
if i < 0 then
begin
…
end
else
if i > 0 then
begin
…
end
else…
```

В таком варианте дважды может быть выполнено только сравнение 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 будет использоваться пустая ссылка – nil.
Поскольку в памяти хранятся ссылки, описанные массивы будут играть роль хэш-таблиц, ссылки из которых указывают непосредственно на информацию в таблицах идентификаторов.
На рис. 1.3 показаны условные схемы, наглядно иллюстрирующие организацию таблиц идентификаторов. Схема 1 иллюстрирует таблицу идентификаторов с рехэшированием на основе генератора псевдослучайных чисел, схема 2 – таблицу идентификаторов на основе комбинации хэш-адресации с бинарным деревом. Ячейки с надписью «nil» соответствуют незаполненным ячейкам хэш-таблицы.

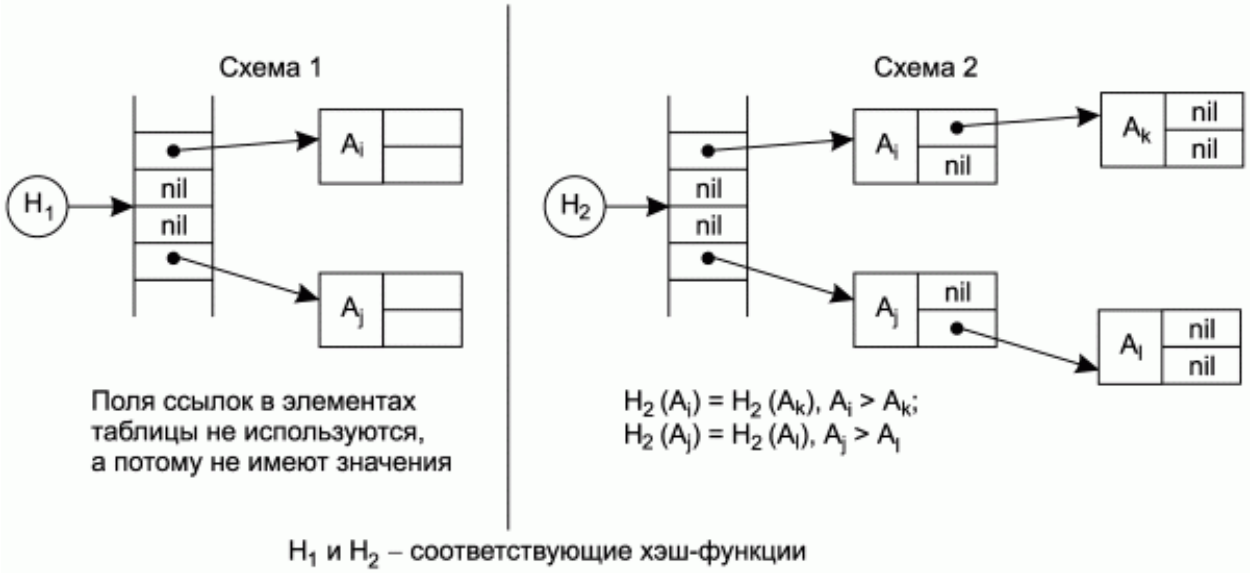

Рис. 1.3. Схемы организации таблиц идентификаторов.
Для каждой таблицы идентификаторов реализованы следующие функции:
• функции начальной инициализации хэш-таблицы – InitTreeVar и InitHashVar;
• функции освобождения памяти хэш-таблицы – ClearTreeVar и ClearHashVar;
• функции удаления дополнительной информации 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 поиске 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 функций (AddTreeVar и AddHashVar – для размещения элемента; GetTreeVar и GetHashVar – для поиска элемента). Функции поиска и размещения элементов в таблице в качестве результата возвращают ссылку на элемент таблицы (структура которого описана в модуле TblElem) в случае успешного выполнения и нулевую ссылку – в противном случае.

Надо отметить, что функции размещения идентификатора в таблице организованы таким образом, что если на момент помещения нового идентификатора в таблице уже есть идентификатор с таким же именем, то функция не добавляет новый идентификатор в таблицу, а возвращает в качестве результата ссылку на ранее помещенный 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, так как 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 рехэширования с использованием генератора псевдослучайных чисел, и FncTree – для таблицы идентификаторов, построенной на основе комбинации хэш-адресации и бинарного дерева. Кроме массивов данных для организации таблиц идентификаторов и функций работы с ними эти модули содержат также 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 веб-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 (initialization) 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 порядке 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, обеспечивающий интерфейс с пользователем. Этот модуль (FormLab1)

реализует графическое окно TLab1Form на основе 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 FormLab1.pas) модуль включает в себя описание ресурсов пользовательского интерфейса (файл FormLab1.dfm). Более подробно принципы организации пользовательского интерфейса на основе GUI и работа систем программирования с ресурсами интерфейса описаны в [3, 5, 6, 7].
Кроме описания интерфейсной формы и ее органов управления модуль FormLab1 содержит три переменные (iCountNum, iCountHash, iCountTree), служащие для накопления 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, описанная в модуле, содержит следующие основные органы управления:
• поле ввода имени файла (EditFile), кнопка выбора имени файла из каталогов файловой системы (BtnFile), кнопка чтения файла (BtnLoad);
• многострочное поле для отображения прочитанного файла (Listldents);
• поле ввода имени искомого идентификатора (EditSearch);
• кнопка для поиска введенного идентификатора (BtnSearch) – этой кнопкой однократно вызывается процедура поиска (procedure SearchStr);
• кнопка автоматического поиска всех идентификаторов (BtnAllSearch) – этой кнопкой процедура поиска идентификатора (procedure SearchStr) вызывается циклически для всех считанных из файла идентификаторов (для всех, перечисленных в поле Listldents);
• кнопка сброса накопленной статистической информации (BtnReset);
• поля для отображения статистической информации;
• кнопка завершения работы с программой (BtnExit).
Внешний вид этой формы приведен на рис. 1.4.

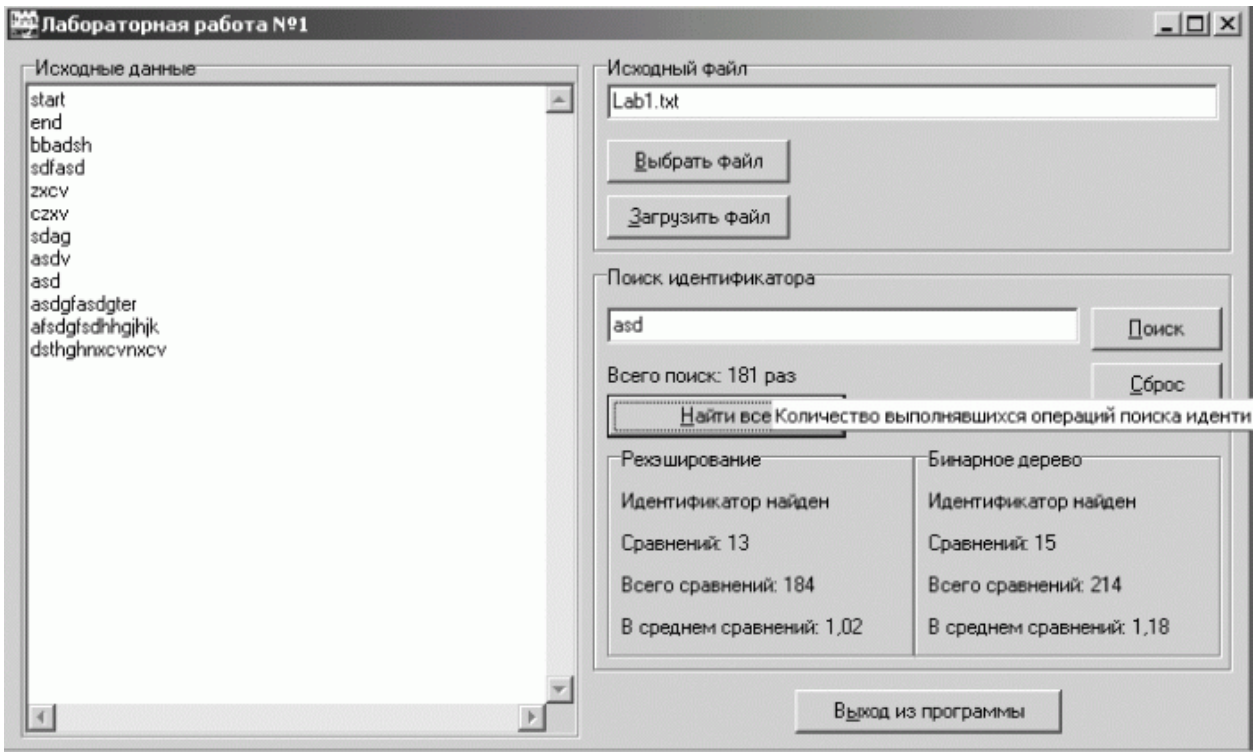

Рис. 1.4. Внешний вид интерфейсной формы для лабораторной работы № 1.

Функция чтения содержимого файла с идентификаторами (procedure TLab1Form. BtnLoadClick) вызывается щелчком по кнопке BtnLoad. Она организована таким образом, что сначала содержимое файла читается в многострочное поле Listldents, а затем все прочитанные идентификаторы записываются в две таблицы идентификаторов. Каждая строка файла считается отдельным идентификатором, пробелы в начале и в конце строки игнорируются. При ошибке размещения идентификатора в одной из таблиц выдается предупреждающее сообщение (например, если будет считано более 223 различных идентификаторов, то рехэширование станет невозможным и будет выдано сообщение об ошибке).

Функция поиска идентификатора (procedure TLab1Form.SearchStr) вызывается однократно щелчком по кнопке BtnSearch (процедура procedure TLab1Form.BtnSearchClick) или многократно щелчком по кнопке BtnAllSearch (процедура procedure TLab1Form. BtnAllSearchClick). Поиск идет сразу в двух таблицах, результаты поиска и накопленная статистическая информация отображаются в соответствующих полях.

Полный текст программного кода модуля интерфейса с пользователем и описание ресурсов пользовательского интерфейса находятся в архиве, располагающемся на веб-сайте издательства, в файлах FormLab1.pas и FormLab1.dfm соответственно.

Полный текст всех программных модулей, реализующих рассмотренный пример для лабораторной работы № 1, можно найти в архиве, располагающемся на вебсайте, в подкаталогах LABS и COMMON (в подкаталог COMMON вынесены те программные модули, исходный текст которых не зависит от входного языка и задания по лабораторной работе).

Главным файлом проекта является файл LAB1.DPR в подкаталоге LABS. Кроме того, текст модуля FncTree приведен в листинге П3.1 в приложении 3.

Выводы по проделанной работе

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

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