Лабораторная работа № 3 Работа с таблицей идентификаторов

Цель работы: изучить основные методы организации таблиц идентификаторов, получить представление о преимуществах и недостатках, присущих различным методам организации таблиц символов (идентификаторов).

Для выполнения лабораторной работы требуется написать программу, которая получает на входе набор идентификаторов, организует таблицу по заданному методу и позволяет осуществить поиск идентификатора в этой таблице.

Список идентификаторов задан в виде текстового файла. Длина идентификаторов ограничена 32 символами.

Назначение и особенности построения таблиц идентификаторов

Проверка правильности семантики и генерация кода требуют знания характеристик переменных, констант, функций и других элементов, встречающихся в программе на исходном языке. Все эти элементы в исходной программе, как правило, обозначаются идентификаторами. Выделение идентификаторов и других элементов исходной программы происходит на фазе лексического анализа.

Компилятор должен иметь возможность хранить все найденные идентификаторы и связанные с ними характеристики в течение всего процесса компиляции, чтобы иметь возможность использовать их на различных фазах компиляции. Для этой цели используются специальные хранилища данных, называемые *таблицами идентификаторов.*

Любая таблица идентификаторов состоит из набора полей, количество которых равно числу различных идентификаторов, найденных в исходной программе. Каждое поле содержит в себе полную информацию о данном элементе таблицы. Компилятор может работать с одной или несколькими таблицам идентификаторов — их количество зависит от реализации компилятора. Например, можно организовывать различные таблицы идентификаторов для различных модулей исходной программы или для различных типов элементов входного языка.

Состав информации, хранимой в таблице идентификаторов для каждого элемента исходной программы, зависит от семантики входного языка и типа элемента. Например, в таблицах идентификаторов может храниться следующая информация:

- *для переменных* - имя переменной; тип данных переменной; область памяти, связанная с переменной;
- *для констант* - название константы (если оно имеется); значение константы; тип данных константы (если требуется);
- *для функций* - имя функции; количество и типы формальных аргументов функции; тип возвращаемого результата; адрес кода функции.

Приведенный выше состав хранимой информации, конечно же, является только примерным. Конкретное наполнение таблиц идентификаторов зависит от реализации компилятора. Кроме того, не вся информация, хранимая в таблице идентификаторов, заполняется компилятором сразу — он может несколько раз выполнять обращение к данным в таблице идентификаторов на различных фазах компиляции. Например, имена переменных могут быть выделены на фазе лексического анализа, типы данных для переменных — на фазе синтаксического разбора, а область памяти связывается с переменной только на фазе подготовки к генерации кода.

Вне зависимости от реализации компилятора принцип его работы с таблицей идентификаторов остается одним и тем же — на различных фазах компиляции компилятор вынужден многократно обращаться к таблице для поиска информации и записи новых данных. Как правило, каждый элемент в исходной программе однозначно идентифицируется своим именем. Поэтому компилятору приходится часто выполнять поиск необходимого элемента в таблице идентификаторов по его имени, в то время как процесс заполнения таблицы

выполняется нечасто — новые идентификаторы описываются в программе гораздо реже, чем используются. Отсюда можно сделать вывод, что таблицы идентификаторов должны быть организованы таким образом, чтобы компилятор имел возможность максимально быстрого поиска нужного ему элемента.

Простейшие методы построения таблиц идентификаторов

Простейший способ организации таблицы состоит в том, чтобы добавлять элементы в порядке их поступления. Тогда таблица идентификаторов будет представлять собой неупорядоченный массив информации, каждая ячейка которого будет содержать данные о соответствующем элементе таблицы.

Поиск нужного элемента в таблице будет в этом случае заключаться в последовательном сравнении искомого элемента с каждым элементом таблицы, пока не будет найден подходящий. И если время, требуемое на добавление элемента ($T_З$), не будет зависеть от числа элементов в таблице N, то для поиска элемента (Тп) в неупорядоченной таблице из N элементов понадобится в среднем N/2 сравнений.

Поскольку поиск в таблице идентификаторов является чаще всего выполняемой компилятором операцией, а количество различных идентификаторов даже в реальной исходной программе достаточно велико (от нескольких сотен до нескольких тысяч элементов), то такой способ организации таблиц идентификаторов является неэффективным.

Поиск может быть выполнен более эффективно, если элементы таблицы упорядочены (отсортированы) согласно некоторому естественному порядку.

Эффективным методом поиска в упорядоченном списке из N элементов является бинарный или логарифмический поиск. Идентификатор, который требуется найти, 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 элемент не будет найден, либо алгоритм не дойдет до очередного блока, содержащего один или два элемента (с которыми уже можно выполнить прямое сравнение искомого элемента).

Так как на каждом шаге число элементов, которые могут содержать искомый элемент, сокращается наполовину, то максимальное число сравнений равно $1+\log_2(N)$.

Тогда время поиска элемента в таблице идентификаторов можно оценить как $T_П = O(\log_2 N)$. 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 метода является требование упорядочивания таблицы идентификаторов. Так как массив информации, в котором выполняется поиск, должен быть упорядочен, то время его заполнения уже будет зависеть от числа элементов в массиве. Таблица идентификаторов зачастую просматривается еще до того, как она заполнена полностью, поэтому требуется, чтобы условие упорядоченности выполнялось на всех этапах обращения к ней.

Следовательно, для построения таблицы можно пользоваться только алгоритмом прямого упорядоченного включения элементов.

При добавлении каждого нового элемента в таблицу сначала надо определить место, куда поместить новый элемент, а потом выполнить перенос части информации в таблице, если элемент добавляется не в ее конец.

В итоге при организации логарифмического поиска в таблице идентификаторов мы добиваемся существенного сокращения 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, то этот метод следует признать более эффективным, чем метод организации неупорядоченной таблицы.

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 корневой узел является первым элементом, встреченным при заполнении таблицы. Дерево называется бинарным, так как каждая вершина в нем может иметь не более двух ветвей (и, следовательно, не более двух нижележащих вершин). Для определенности будем называть две ветви «правая» и «левая».

Будем считать, что алгоритм заполнения бинарного дерева работает с потоком входных данных, содержащим идентификаторы (в компиляторе этот поток данных порождается в процессе разбора текста исходной программы). Первый идентификатор, как уже было сказано, помещается в вершину дерева. Все дальнейшие идентификаторы попадают в дерево по следующему алгоритму.

Шаг 1. Выбрать очередной идентификатор из входного потока данных. Если очередного идентификатора нет, то построение дерева закончено.

Шаг 2. Сделать текущим узлом дерева корневую вершину.

Шаг 3. Сравнить очередной идентификатор с идентификатором, содержащемся в текущем узле дерева.

Шаг 4. Если очередной идентификатор меньше, то перейти к шагу 5, если равен — сообщить об ошибке и прекратить выполнение алгоритма (двух одинаковых идентификаторов быть не должно!), иначе — перейти к шагу 7.

Шаг 5. Если у текущего узла существует левая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе — перейти к шагу 6.

Шаг 6. Создать новую вершину, поместить в нее очередной идентификатор, сделать эту новую вершину левой вершиной текущего узла и вернуться к шагу 1.

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 3, иначе — перейти к шагу 8.

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

Рассмотрим в качестве примера последовательность идентификаторов GA, Dl, M22, E, A12, BC, F. На рис. 1 проиллюстрирован весь процесс построения бинарного дерева для этой последовательности идентификаторов.

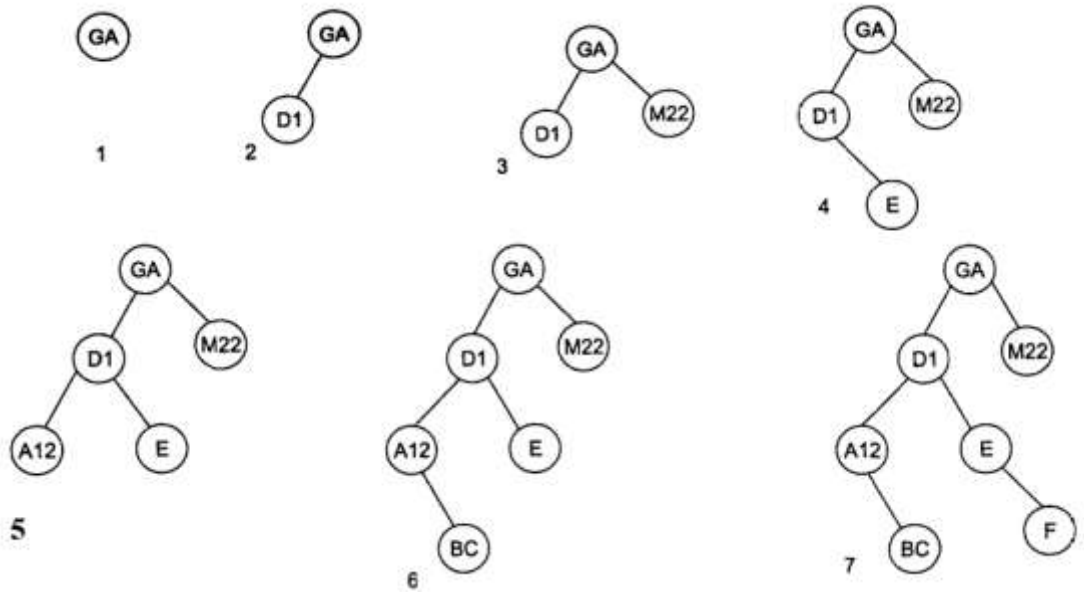


Рис 1. Пошаговое заполнение бинарного дерева для последовательности идентификаторов GA, D1, M22, E, A12, BC, F

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

Шаг 1. Сделать текущим узлом дерева корневую вершину.

Шаг 2. Сравнить искомый идентификатор с идентификатором, содержащемся в текущем узле дерева.

Шаг 4. Если идентификаторы совпадают, то искомый идентификатор найден, алгоритм завершается, иначе — надо перейти к шагу 5.

Шаг 5. Если очередной идентификатор меньше, то перейти к шагу 6, иначе — перейти к шагу 7.

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

Шаг 7. Если у текущего узла существует правая вершина, то сделать ее текущим узлом и вернуться к шагу 2, иначе искомый идентификатор не найден, алгоритм завершается.

Например, произведем поиск в дереве, изображенном на рис.1 (7), идентификатора A12. Берем корневую вершину (она становится текущим узлом), сравниваем идентификаторы GA и A12. Искомый идентификатор меньше - текущим узлом становится левая вершина D1. Опять сравниваем идентификаторы. Искомый идентификатор меньше - текущим узлом становится левая вершина A12. При следующем сравнении искомый идентификатор найден.

Если искать отсутствующий идентификатор - например, A11, - то поиск опять пойдет от корневой вершины. Сравниваем идентификаторы GA и AH. Искомый идентификатор меньше - текущим узлом становится левая вершина D1. Опять сравниваем идентификаторы. Искомый идентификатор меньше - текущим узлом становится левая вершина A12. Искомый идентификатор меньше, но левая вершина у узла A12 отсутствует, поэтому в данном случае искомый идентификатор не найден.

Для данного метода число требуемых сравнений и форма получившегося дерева во многом зависят от того порядка, в котором поступают идентификаторы. Например, если в

рассмотренном выше примере вместо последовательности идентификаторов GA, 01, M22, E, A12, BC, F взять последовательность A12, GA, D1, (22, E, BC, F, то полученное дерево будет иметь иной вид. А если в качестве примера взять последовательность идентификаторов A, B, C, D, E, F, то дерево выродится в упорядоченный однонаправленный связный список. Эта особенность является недостатком данного метода организации таблиц идентификаторов. Другим недостатком является необходимость работы 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 ($T_З$) и на поиск элемента в нем ($T_П$) можно оценить следующим образом:

$T_З = N*O(\log_2 N)$.

$T_П = O(\log_2 N)$.

В целом 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.

Хэш-функции и хэш-адресация. Принципы работы хэш-функций

Логарифмическая зависимость времени поиска и времени заполнения таблицы идентификаторов — это самый хороший результат, которого можно достичь за счет применения различных методов организации таблиц. Однако в реальных программах количество идентификаторов столь велико, что даже логарифмическую зависимость времени поиска от их числа нельзя считать удовлетворительной. 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»). Вместо термина «хэширование» иногда используются термины «рандомизация», «переупорядочивание».

При работе с таблицей идентификаторов хэш-функция должна выполнять отображение имен идентификаторов на множество целых неотрицательных чисел. Областью определения хэш-функций 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-функций. Следовательно, в реальном компиляторе область значений хэш-функций никак не должна превышать размер доступного адресного пространства компьютера.

Метод организации таблиц идентификаторов, основанный на использовании хэш-адресации, заключается в размещении каждого элемента таблицы в ячейке, адрес которой возвращает хэш-функция, вычисленная для этого элемента. Тогда в идеальном случае для размещения любого элемента в таблице идентификаторов достаточно только вычислить его хэш-функцию и обратиться к нужной ячейке массива данных. Для поиска элемента в таблице 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).

На рис.2 проиллюстрирован метод организации таблиц идентификаторов с использованием хэш-адресации. Трем различным идентификаторам A1, A2, A3 соответствуют на рисунке три значения хэш-функций n1, n2, n3. В ячейки, адресуемые n1, n2, n3, помещается информация об идентификаторах a1, A2, A3. При поиске идентификатора Аз вычисляется значение адреса n3 и выбираются данные из соответствующей ячейки таблицы.

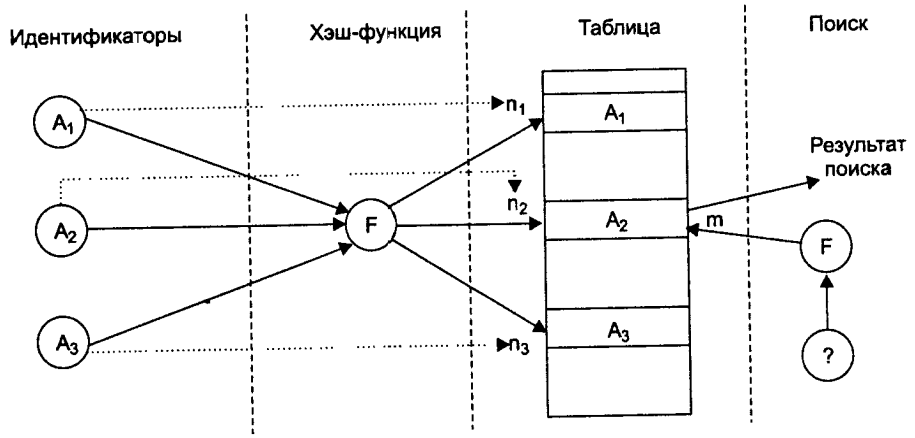


Рис.2. Организация таблицы идентификаторов с использованием хэш-адресации

Этот метод весьма эффективен — как время размещения элемента в таблице, так и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функций, которое в общем случае несопоставимо меньше времени, необходимого на многократные сравнения элементов таблицы.

Метод имеет два очевидных недостатка. Первый из них — неэффективное использование объема памяти под таблицу идентификаторов: размер массива для ее хранения должен соответствовать 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. Этому существенному вопросу посвящены следующие два подраздела.

Построение таблиц идентификаторов на основе хэш-функции

Существуют различные варианты хэш-функции. Получение результата хэш-функции — «хэширование» — обычно достигается за счет выполнения над цепочкой символов некоторых простых арифметических и логических операций. Самой простой хэш-функцией для символа является код внутреннего представления в ЭВМ литеры символа. Эту хэш-функцию можно использовать и для цепочки символов, выбирая первый символ в цепочке. Так, если двоичное ASCII представление символа A есть 00100001, то результатом хэширования идентификатора ATable будет код 00100001.

Хэш-функция, предложенная выше, очевидно не удовлетворительна: при использовании такой хэш-функции возникнет новая 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 ячейку таблицы идентификаторов по одному и тому же адресу должны быть помещены два различных идентификатора, что явно невозможно. Такая ситуация, когда двум или более идентификаторам соответствует одно и то же значение функции, называется *коллизией.* Возникновение коллизии проиллюстрировано на рис.3 — двум различным идентификаторам A_1 и A_2 соответствуют два совпадающих значения хэш-функции $n_1 = n_2$.

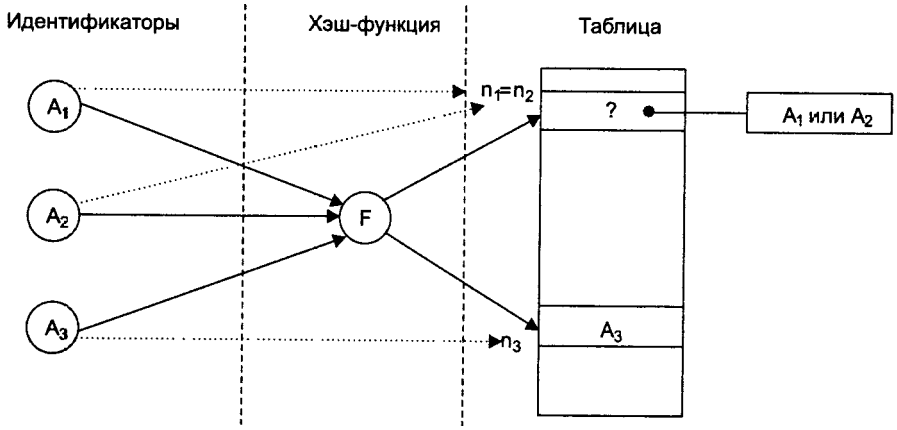


Рис. 3. Возникновение коллизии при использовании хэш-адресации

Естественно, что хэш-функция, допускающая коллизии, не может быть напрямую использована для хэш-адресации в таблице идентификаторов. Достаточно получить хотя бы один случай коллизии на всем множестве идентификаторов, чтобы такой хэш-функцией нельзя было пользоваться непосредственно. Но в примере взята самая элементарная хэш-функция.

Очевидно, что для полного исключения коллизий хэш-функция должна быть *взаимно однозначной*. 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.

Но практически это сделать исключительно сложно, т.к. в реальности область значений любой хэш-функции ограничена размером доступного адресного пространства в данной архитектуре компьютера. Организовать же взаимно однозначное отображение бесконечного множества на конечное даже теоретически невозможно. Если даже учесть, что длина принимаемой во внимание строки идентификатора в реальных компиляторах также практически ограничена — обычно она лежит в пределах от 32 до 128 символов (то есть и область определения хэш-функции конечна), то и тогда количество всех возможных идентификаторов все равно больше количества допустимых адресов в современных компьютерах. Таким образом, создать взаимно однозначную хэш-функцию практически ни в каком варианте невозможно. Следовательно, **невозможно избежать возникновения коллизий**.

Одним из **способов решения проблемы коллизий** является метод «рехэширования» (или «расстановки»). Согласно этому методу, если для элемента A адрес h(A), вычисленный с помощью хэш-функции, указывает на уже занятую ячейку, то необходимо вычислить значение функции $n_1 = h_1(A)$ и проверить занятость ячейки по адресу n_1. Если и она занята, то вычисляется значение $h_2(A)$, и так до тех пор, пока либо не будет найдена свободная ячейка, либо очередное значение $h_i(A)$ совпадет с h(A). В последнем случае считается, что таблица идентификаторов заполнена и места в ней больше нет — дается информация об ошибке размещения идентификатора в таблице. Особенностью метода является то, что первоначально таблица идентификаторов должна быть заполнена информацией, которая позволила бы говорить о том, что, ее ячейки являются пустыми (не содержат данных). Например, если используются указатели для хранения имен идентификаторов, то таблицу надо предварительно заполнить пустыми указателями.

Такую таблицу идентификаторов можно организовать по следующему алгоритму размещения элемента.

Шаг 1. Вычислить значение хэш-функции $n = h(A)$ для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, то поместить в нее элемент A и завершить алгоритм, иначе i:=1 и перейти к шагу 3.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая, то поместить в нее элемент A и завершить алгоритм, иначе перейти к шагу 4.

Шаг 4. Если $n = n_i$, то сообщить об ошибке и завершить алгоритм, иначе i:=i+1 и вернуться к шагу 3.

Когда поиск элемента A в таблице идентификаторов, организованной таким образом, будет выполняться по следующему алгоритму.

Шаг 1. Вычислить значение хэш-функции $n = h(A)$ для нового элемента A.

Шаг 2. Если ячейка по адресу n пустая, то элемент не найден, алгоритм завершен, иначе 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.

Шаг 3. Вычислить $n_i = h_i(A)$. Если ячейка по адресу n_i пустая или $n = n_i$, то элемент не найден и алгоритм завершен, иначе сравнить имя элемента в ячейке n_i с именем искомого элемента A. Если они совпадают, то элемент найден и алгоритм завершен, иначе i:=i+1 и повторить к шаг 3.

Итак, количество операций, необходимых для поиска или размещения в таблице элемента, зависит от заполненности таблицы. Естественно, функции $h_i(A)$ должны вычисляться единообразно на этапах размещения и поиска элемента. Вопрос только в том, как организовать вычисление функций $h_i(A)$.

Самым простым методом вычисления функции $h_i(A)$ является ее организация в виде $h_i(A) = (h(A) + p_i) \bmod N_m$, где p_i — некоторое вычисляемое целое число, а N_m — максимальное число элементов в таблице идентификаторов. В свою очередь, самым простым подходом здесь будет положить $p_i = i$. Тогда получаем формулу $h_i(A) = (h(A)+i) \bmod N_m$. 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).

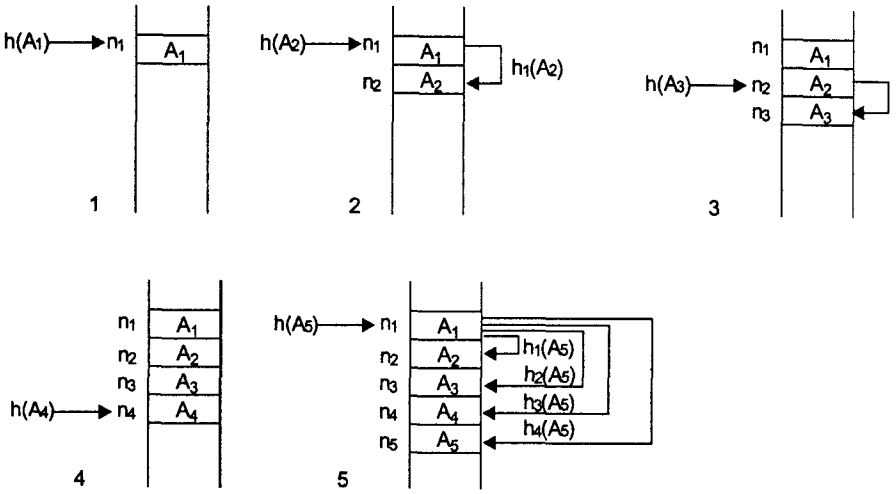
Этот способ нельзя признать особенно удачным — при совпадении хэш-адресов элементы в таблице начинают группироваться вокруг них, что увеличивает число необходимых сравнений при поиске и размещении. Среднее время поиска элемента в такой таблице в зависимости от числа операций сравнения можно оценить следующим образом:

$T_П = O((1-Lf/2)/(1- Lf))$.

Здесь Lf — (load factor) степень заполненности таблицы идентификаторов — отношение числа занятых ячеек таблицы к максимально допустимому числу элементов в ней: $Lf = N/N_m$.

Рассмотрим в качестве примера ряд последовательных ячеек таблицы n_1, n_2, n_3, n_4, n_5 и ряд идентификаторов, которые надо разместить в ней: A_1, A_2, A_3, A_4, A_5 при условии, что $h(A_1) = h(A_2) = h(A_3) = h(A_4) = h(A_5)$. Последовательность размещения идентификаторов в таблице при использовании простейшего метода рехэширования показана на рис. 4. В итоге после размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 2 сравнения, для A_4 — 1 сравнение и для A_5 — 5 сравнений.

Рис. 4. Заполнение таблицы идентификаторов при использовании простейшего рехэширования



Даже такой примитивный метод рехэширования является достаточно эффективным средством организации таблиц идентификаторов при неполном заполнении таблицы. Имея, например, даже заполненную на 90 % таблицу для 1024 идентификаторов, в среднем необходимо выполнить 5,5 сравнений для поиска одного идентификатора, в то время как даже логарифмический поиск дает в среднем от 9 до 10 сравнений. Сравнительная эффективность метода будет еще выше при росте числа идентификаторов и снижении заполненности 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 p_i для функции $h_i(A) = (h(A)+p_i) \bmod N_m$ последовательности псевдослучайных целых чисел $p_1, p_2, ..., p_k$. При хорошем выборе генератора псевдослучайных чисел длина последовательности k будет $k=N_m$. Тогда среднее время поиска одного элемента в таблице можно оценить следующим образом:

$$E_П = O((1/Lf)*\log_2(1- Lf)).$$

Существуют и другие методы организации функций рехэширования $h_i(A)$, основанные на квадратичных вычислениях или, например, на вычислении по формуле:
$h_i(A) = (h(A)*i) \bmod N_m$, если N_m - простое число. 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**.

Построение таблиц идентификаторов по методу цепочек

Неполное заполнение таблицы идентификаторов при применении хэш-функции 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 дина-

мической 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 (в ней не будет пустых неиспользуемых ячеек). 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 «метод цепочек». Для метода цепочек 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 для этого метода необходимо иметь одну специальную переменную, которая всегда указывает на первую свободную ячейку основной таблицы идентификаторов (первоначально — указывает на начало таблицы).

Метод цепочек работает следующим образом по следующему алгоритму.

Шаг 1. Во все ячейки хэш-таблицы поместить пустое значение, таблица идентификаторов не должна содержать ни одной ячейки, переменная *FreePtr* (указатель первой свободной ячейки) указывает на начало таблицы идентификаторов; i:=1.

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

Шаг 3. Положить j:=1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j и перейти к шагу 4.

Шаг 4. Для ячейки таблицы идентификаторов по адресу m_j проверить значение поля ссылки. Если оно пустое, то записать в него адрес из переменной *FreePtr* и перейти к шагу 5; иначе j:=j+1, выбрать из поля ссылки адрес m_j и повторить шаг 4.

Шаг 5. Добавить в таблицу идентификаторов новую ячейку, записать в нее информацию для элемента A_i (поле ссылки должно быть пустым), в переменную *FreePtr* поместить адрес за концом добавленной ячейки. Если больше нет идентификаторов, которые надо разместить в таблице, то выполнение алгоритма закончено, иначе i:=i+1 и перейти к шагу 2.

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

Шаг 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 положить j:=1, выбрать из хэш-таблицы адрес ячейки таблицы идентификаторов m_j=n.

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

Шаг 3. Проверить значение поля ссылки в ячейке таблицы идентификаторов по адресу m_j. Если оно пустое, то искомый элемент не найден и алгоритм завершен; иначе j:=j+1, выбрать из поля ссылки адрес m_j и перейти к шагу 2.

При такой организации таблиц идентификаторов в случае возникновения коллизии алгоритм размещает элементы в ячейках 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 происходит и название данного метода — «метод цепочек».

На рис.5 проиллюстрировано заполнение хэш-таблицы и таблицы идентификаторов для примера, который ранее был рассмотрен на рис. 4 для метода простейшего рехэширования. После размещения в таблице для поиска идентификатора A_1 потребуется 1 сравнение, для A_2 — 2 сравнения, для A_3 — 1 сравнение, для A_4 — 1 сравнение и для A_5 — 3 сравнения (сравните с результатами простого рехэширования).

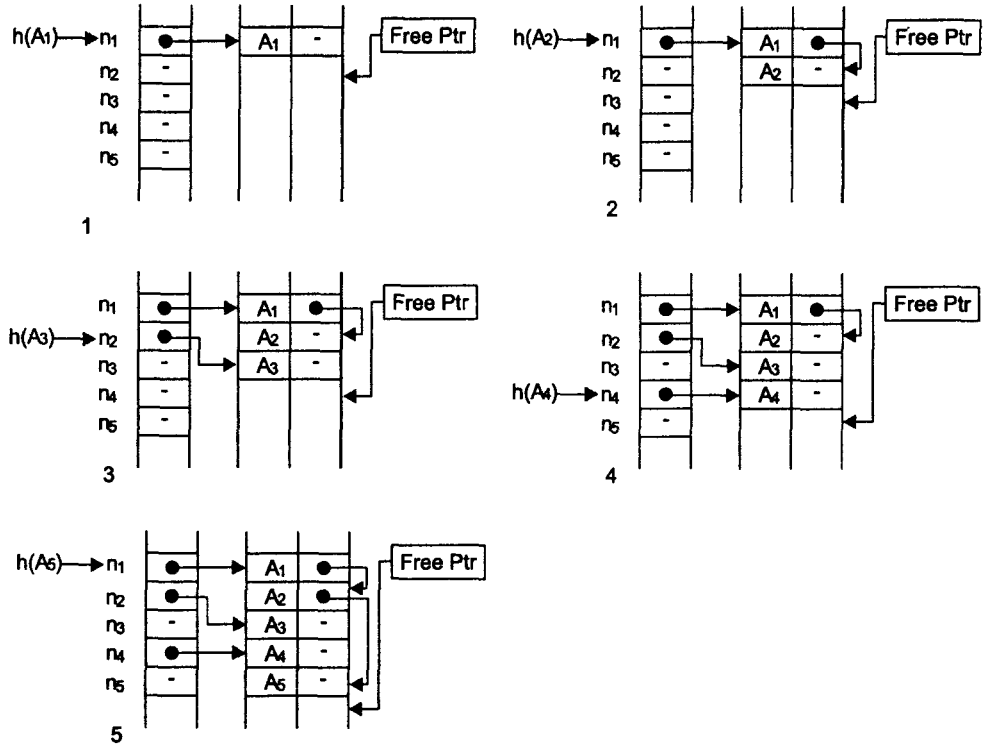


Рис. 5. Заполнение хэш-таблицы и таблицы идентификаторов при использовании метода цепочек

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. Этот метод позволяет более экономно использовать память, но требует организации работы с динамическими массивами данных.

Комбинированные способы построения таблиц идентификаторов

Выше была рассмотрена весьма примитивная хэш-функция, которую никак нельзя назвать удовлетворительной. Хорошая хэш-функция распределяет поступающие на ее вход идентификаторы равномерно на все имеющиеся в распоряжении адреса, так что коллизии возникают не столь часто.

В реальных компиляторах практически всегда так или иначе используется хэш-адресация. Алгоритм применяемой хэш-функции обычно составляет «ноу-хау» разработчиков компилятора. Обычно при разработке хэш-функции создатели компилятора стремятся свести к минимуму количество возникающих коллизий не на всем множестве возможных идентификаторов, а на тех их вариантах, которые наиболее часто встречаются во входных программах. Конечно, принять во внимание все допустимые исходные программы невозможно. Чаще всего выполняется статистическая обработка встречающихся имен идентификаторов на некотором множестве типичных исходных программ, а также принимаются во внимание соглашения о выборе имен идентификаторов, общепринятые для входного языка. Хорошая хэш-функция — это шаг к значительному ускорению работы компилятора,

поскольку обращения к таблицам идентификаторов выполняются многократно на различных фазах компиляции.

Как правило, применяются комбинированные методы. В этом 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 по одному из рассмотренных выше методов:

- неупорядоченный список,
- упорядоченный список или же
- бинарное дерево.

При хорошо построенной хэш-функции коллизии будут возникать редко, поэтому количество идентификаторов, для которых значения хэш-функции совпали, будет не столь велико. Тогда и время поиска одного среди них будет незначительным (в принципе при высоком качестве хеш-функции подойдет даже перебор по неупорядоченному списку).

Такой подход имеет преимущество по сравнению с методом цепочек: для хранения идентификаторов с совпадающими значениями хэш-функции используются области памяти, не пересекающиеся с основной таблицей идентификаторов, а значит, их размещение не приведет к возникновению дополнительных коллизий. Недостатком метода является необходимость работы с динамически распределяемыми областями памяти. Эффективность такого метода, очевидно, в первую очередь зависит от качества применяемой хэш-функции, а во вторую — от метода организации дополнительных хранилищ данных.