

Лабораторная работа №3.

Таблицы идентификаторов

Теоретические сведения	1
Выбор хэш-функции	6
Мультипликативные хэш-функции	6
Универсальные хэш-функции	7
Устранение коллизий	7
Метод цепочек.....	7
Открытая адресация	8
Хранение информации об идентификаторах.....	9
Вложенные области видимости	9
Метод цепочек и вложенные области видимости.....	10
Открытая адресация и вложенные области видимости	10
Задание на лабораторную работу	13
Контрольные вопросы	13
Список литературы	13

Теоретические сведения

Таблица идентификаторов (символов) – это структура данных, которая используется компилятором для хранения информации о различных элементах программы, таких как переменные, константы, имена функций или процедур, т.е. идентификаторах.

Поиск в таблице идентификаторов происходит каждый раз, когда идентификатор встречается в программе. Если появляется новый идентификатор или новая информация об уже существующем в таблице идентификаторе, содержимое таблицы изменяется. Поэтому очень важно, чтобы в таблице идентификаторов использовались эффективные алгоритмы для добавления и изменения ее данных. Под эффективностью понимается минимальное время, затрачиваемое на доступ к таблице, как при добавлении новых идентификаторов, так и при изменении данных об уже существующих. А также способ выделения памяти для хранения информации в таблице. Таблица должна расти динамически по мере своего заполнения.

Каждая строка таблицы может быть реализована как запись, состоящая из нескольких полей. Количество этих полей зависит от информации, хранимой в таблице.

Каждый идентификатор в программе имеет свой набор атрибутов, который зависит от синтаксиса и семантики языка программирования и от использования идентификатора в конкретной программе. Наиболее распространенными атрибутами являются имя идентификатора, его тип, область видимости и размер. В табл. 1 приведены эти и другие атрибуты, а также тип данных, который может быть использован для представления атрибутов.

Таблица 1. Атрибуты идентификаторов

Имя атрибута	Возможный тип данных для представления в таблице	Значение атрибута
Имя	Строковый	Имя идентификатора
Класс	Перечисление	Класс памяти
Изменчивость (volatile)	Булев	Соответствует значению ключевого слова <i>volatile</i> ¹
Размер	Целый	Размер в байтах
Размер в битах	Целый	Размер в битах для размеров, не кратных байту
Выравнивание	Целый	Выравнивание в байтах
Выравнивание в битах	Целый	Выравнивание в битах
Тип	Перечисление или указатель на структуру, представляющую тип	Тип данных в исходном языке
Базовый тип	Перечисление или указатель на структуру, представляющую тип	Тип данных элементов для сложных типов в исходном языке
Машинный тип	Перечисление	Машинный тип, соответствующий типу в исходном языке
Количество элементов	Целый	Количество элементов для сложных типов данных
Регистровая переменная	Булев	Принимает значение «истина», если значение идентификатора хранится в регистре
Регистр	Строковый	Имя регистра
Базовый регистр	Строковый	Имя базового регистра для вычисления адреса идентификатора
Смещение	Целый	Смещение относительно значения в базовом регистре

Информация о типе может быть представлена перечисляемым типом (*enum*) для стандартных типов языка программирования (*int*, *char*, *double* и т.п.). Для пользовательских типов (массивы, структуры, объединения и т.п.) в таблице идентификаторов можно хранить только указатель на структуру, представляющую данный тип. Наличие отдельных полей для типа, базового типа и машинного типа позволяет определить, что, например, для идентификатора

¹ В языках программирования C и C++ есть ключевое слово *volatile*, которое указывает компилятору, что значение в соответствующей области памяти может быть изменено в произвольный момент и потому нельзя оптимизировать доступ к этой области.

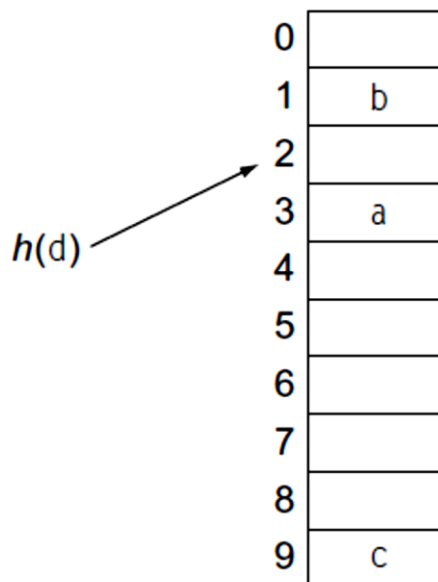
```
char A [3][5];
```

тип – array, базовый тип – char, машинный тип – byte, а количество элементов – $3 \times 5 = 15$.

Наличие атрибутов базовый регистр и смещение дает возможность указать, что для доступа, например, к массиву A необходимо сформировать адрес $[r7+8]$, где r7 – регистр, содержащий базовый адрес, а 8 – смещение относительно этого базового адреса.

Заполнение таблицы идентификаторов происходит на разных этапах компиляции. Лексический анализатор создает записи в таблице, как только встречается имя в исходной программе, а другие атрибуты заполняются по мере обработки объявления этого имени. Но часто одно и то же имя используется в программе для обозначения разных объектов, иногда даже внутри одного декларационного блока. Например, в языке C могут использоваться одинаковые имена для обозначения переменной и поля структуры в рамках одного блока. В таких случаях лексический анализатор возвращает синтаксическому анализатору только имя, а не запись в таблице идентификаторов. Потому что запись в таблице идентификаторов будет создана только тогда, когда будет понята синтаксическая роль данного имени в программе.

Как уже упоминалось ранее, компилятор часто обращается к таблице идентификаторов, поэтому вопрос минимизации времени поиска по таблице идентификаторов является важным. Поскольку хэш-таблицы могут обеспечить константное время поиска ($O(1)$), они и являются наиболее распространенным способом организации таблиц идентификаторов. Для каждого имени в программе с помощью некоторой выбранной хэш-функции h рассчитывается целое число n , которое и будет индексом записи в таблице идентификаторов для хранения этого имени и всей сопутствующей ему информации. На рис. 1 показана хэш-таблица с 10 записями. Переменные a , b и c уже вставлены в таблицу, для d посчитано значение хэш-функции, оно оказалось равно 2. Значит информация о переменной d будет добавлена в запись с этим номером.



0	
1	b
2	
3	a
4	
5	
6	
7	
8	
9	c

Рисунок 1. Хэш-таблица

Так как время размещения элемента в таблице и время его поиска определяются только временем, затрачиваемым на вычисление хэш-функции, ее вычисление не должно занимать много времени. Кроме того, она не должна приводить к частым *коллизиям*, ситуациям, когда двум и более идентификаторам соответствует одно и то же значение хэш-функции.

Для работы с таблицей идентификаторов должны быть реализованы две основные функции:

- 1) LookUp (name) , возвращающая запись, хранящуюся в таблице в ячейке h (name) , если такая существует. Иначе она возвращает значение, указывающее на то, что такой записи нет;
- 2) Insert (name) , сохраняющая идентификатор name в ячейке с номером h (name) .

Большинство языков программирования позволяют объявлять идентификаторы с одинаковыми именами в разных частях программы. Каждый такой идентификатор имеет свою *область видимости* и *время жизни*. В некоторых языках программирования области видимости могут быть вложены одна в другую. Для правильной интерпретации имен идентификаторов компилятору нужна таблица идентификаторов, учитывающая области видимости.

На рис. 2 показана программа на языке Си с пятью областями видимости.

```
static int w;      /* уровень 0 */
int x;

void example(int a, int b) {
    int c;          /* уровень 1 */
    {
        int b, z;    /* уровень 2a */
        ...
    }
    {
        int a, x;     /* уровень 2b */
        ...
        {
            int c, x; /* уровень 3 */
            b = a + b + c + w;
        }
    }
}
```

Рисунок 2. Программа на языке Си с пятью областями видимости

Самый верхний уровень 0 относится к глобальной области видимости, а уровень 3 соответствует самой внутренней области видимости.

Ниже в таблице приведены имена идентификаторов и уровни их объявления.

Имя идентификатора	Уровень объявления
w, x, example	0
a, b, c	1
b, z	2a
a, x	2b
c, x	3

Объявление переменной *b* на уровне 2a перекрывает объявление одноименной переменной на уровне 1. А на уровне 2b снова становится «видна» переменная *b* с уровня 1. Объявления переменных *a* и *x* на уровне 2b перекрывают объявления на уровнях 1 и 0

соответственно. В этом случае вычисление значения переменной b на уровне 3 с учетом областей видимости переменных можно представить так:

$$b_1 = a_{2b} + b_1 + c_3 + w_0 .$$

Для компиляции программы с вложенными областями видимости компилятору нужно выполнить *разрешение имен*, найти однозначное соответствие между используемым именем переменной и местом ее объявления. Для этого используются таблицы идентификаторов лексических областей видимости.

Новая таблица идентификаторов может создаваться каждый раз, когда синтаксический анализатор входит в новую лексическую область видимости. В этом случае будет создан набор таблиц, связанных между собой в порядке лексических уровней. Если в текущей области видимости находятся объявления, информация об этих переменных вносится (Insert) в текущую активную таблицу. Когда компилятор встречает ссылку на переменную, функция LookUp должна вначале проверить таблицу для текущей области видимости. Если в текущей таблице нет такого имени, то проверяется таблица предыдущей области видимости. Проходя таким образом по всем таблицам идентификаторов предыдущих лексических уровней, компилятор либо находит объявление для данного имени, либо, дойдя до самого верхнего уровня, делает вывод о том, что данная переменная не была объявлена ни в одной из областей видимости.

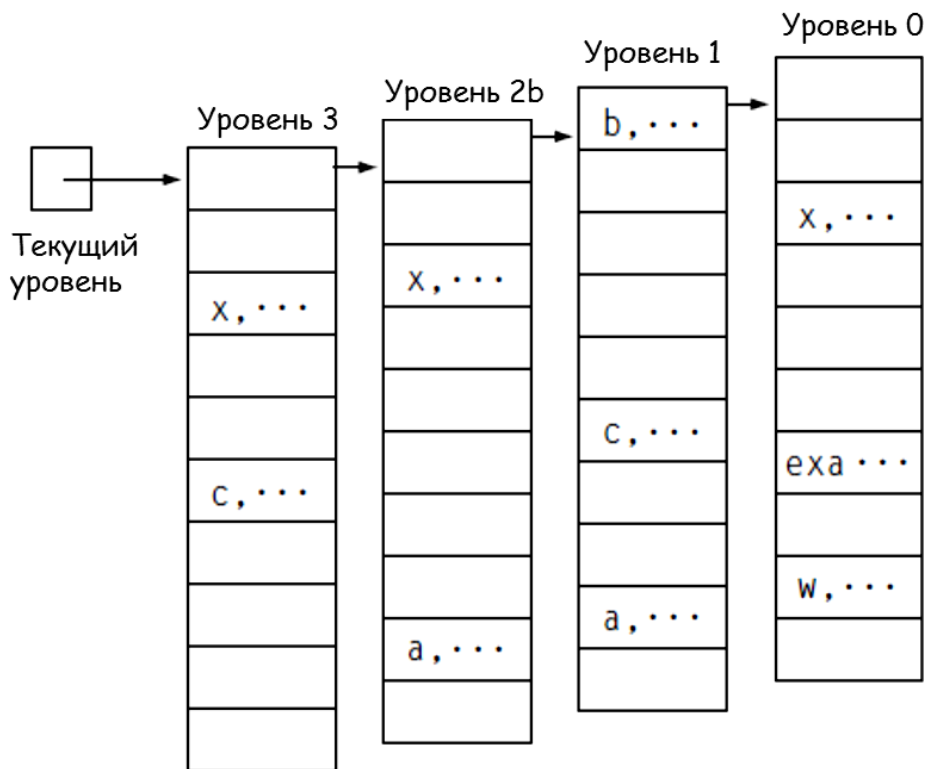


Рисунок 3. Хэш-таблицы для разных областей видимости

На рис. 3 показана таблица идентификаторов, построенная по такому принципу для программы-примера (см. рис. 2), в момент, когда синтаксический анализатор достиг выражения присваивания для b . Функции LookUp для имени b удастся найти информацию в таблице

уровня 1. И действительно, ближайшее объявление переменной *b* было в области видимости уровня 1. В наборе таблиц на рисунке отсутствует таблица, соответствующая уровню 2а, так как на момент обработки выражения присваивания эта область видимости уже закрыта.

Адрес каждого идентификатора может быть представлен *статической координатой* – парой *<level, offset>*. Здесь *level* – номер лексического уровня, соответствующий области видимости, *offset* – смещение от начала таблицы для данного уровня.

Для программной реализации поддержки областей видимости компилятору понадобятся еще две функции:

1) *InitializeScope()* для создания новой таблицы для данного лексического уровня. Эта функция связывает новую таблицу с таблицей предыдущего уровня и обновляет указатель текущего уровня, используемый функциями *LookUp* и *Insert*;

2) *FinalizeScope()* перенаправляет указатель текущего уровня на таблицу предыдущей области видимости. Если компилятору необходимо сохранить таблицы всех уровней для дальнейшего использования, эта функция может оставить таблицу в памяти или записать ее в файл на диск и освободить занимаемую ею память.

Используя эти функции, мы получим следующий порядок вызовов функций для программы с рис. 2:

1. <i>InitializeScope</i>	10. <i>Insert(b)</i>	19. <i>LookUp(b)</i>
2. <i>Insert(w)</i>	11. <i>Insert(z)</i>	20. <i>LookUp(a)</i>
3. <i>Insert(×)</i>	12. <i>FinalizeScope</i>	21. <i>LookUp(b)</i>
4. <i>Insert(example)</i>	13. <i>InitializeScope</i>	22. <i>LookUp(c)</i>
5. <i>InitializeScope</i>	14. <i>Insert(a)</i>	23. <i>LookUp(w)</i>
6. <i>Insert(a)</i>	15. <i>Insert(×)</i>	24. <i>FinalizeScope</i>
7. <i>Insert(b)</i>	16. <i>InitializeScope</i>	25. <i>FinalizeScope</i>
8. <i>Insert(c)</i>	17. <i>Insert(c)</i>	26. <i>FinalizeScope</i>
9. <i>InitializeScope</i>	18. <i>Insert(×)</i>	27. <i>FinalizeScope</i>

Выбор хэш-функции

Важность выбора хорошей хэш-функции трудно переоценить. Хэш-функция, которая производит неравномерное распределение индексных значений, напрямую увеличивает временные затраты на вставку и поиск элементов в таблице. В литературе² описаны многие эффективные хэш-функции. Рассмотрим некоторые из них.

Мультипликативные хэш-функции

Мультипликативная хэш-функция вычисляется по формуле:

$$h(key) = [TableSize \cdot ((C \cdot key) \bmod 1)] ,$$

² Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1296 с. (глава 11).

Кнут Д.Э. Искусство программирования: В 3 т.: Пер. с англ. Т.3: Сортировка и поиск. – 2-е изд. – М.: Издат.дом «Вильямс», 2003. – 822 с. (раздел 6.4)

где C – константа, key – ключ в хэш-таблице, $TableSize$ – размер таблицы. Смысл этой формулы заключается в вычислении $C \cdot key$, получение дробной части этого произведения (с помощью $\text{mod } 1$) и умножение результата на размер таблицы. Для применения этой формулы необходимо выбрать значение C . Кнут предлагает использовать значение, равное золотому сечению:

$$0.6180339887 \approx \frac{\sqrt{5} - 1}{2}$$

Универсальные хэш-функции

Фиксированная хэш-функция, как в случае с мультипликативной, может стать уязвимой и привести к частому возникновению коллизий. Единственный эффективный выход из ситуации – случайный выбор хэш-функции, не зависящий от того, с какими именно ключами ей предстоит работать. Такой подход, который называется *универсальным хэшированием*, гарантирует хорошую производительность в среднем.

Изменяя хэш-функцию при каждом исполнении программы, универсальная хэш-функция производит различные распределения. Для того чтобы реализовать такой подход в случае с мультипликативной функцией, нужно случайным образом генерировать значение константы C .

Устранение коллизий

Метод цепочек

Метод цепочек исходит из того, что хэш-функция h производит коллизии. При этом h разделяет все множество входных ключей на фиксированное количество множеств или карманов. Каждый карман содержит линейный список идентификаторов, для которых вычисленное значение хэш-функции совпало, т.е. произошла коллизия.

Рис. 4 иллюстрирует этот подход. Значения хэш-функции для идентификаторов a и d совпали: $h(a) = h(d) = 3$. Поэтому a и d попадают в одну и ту же ячейку таблицы в виде связанного списка. Функция Insert вставит новый элемент в начало списка для сокращения времени, затрачиваемого на добавление элементов.

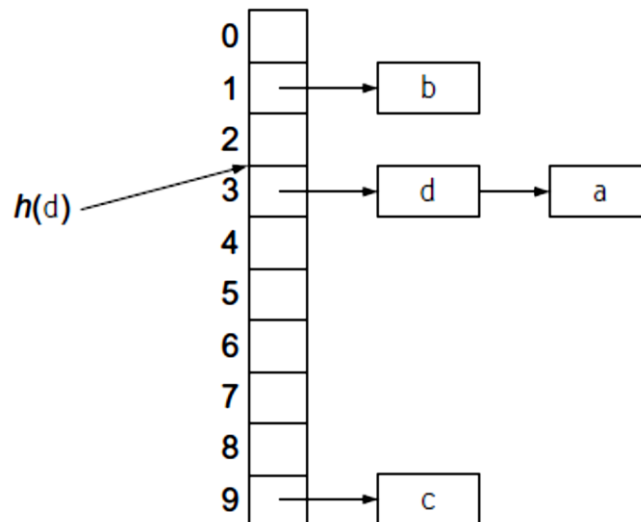


Рисунок 4. Метод цепочек

Открытая адресация

Открытая адресация, или рехэширование, в случае возникновения коллизии пересчитывает значение хэш-функции для идентификатора. Функция $\text{Insert}(\text{name})$ вычисляет $h(\text{name})$, ячейка таблицы с полученным номером пуста, происходит добавление идентификатора в эту ячейку. Если ячейка уже занята, Insert вычисляет значение $g(\text{name})$ хэш-функции, задающей инкремент для вставки: $(h(\text{name}) + g(\text{name})) \bmod S$, где S – размер таблицы. И так до тех пор, пока не будет найдена пустая ячейка. Если в результате рехэширования Insert вернется к значению $h(\text{name})$, это означает, что пустых ячеек в таблице больше нет. Аналогично происходит и работа функции LookUp .

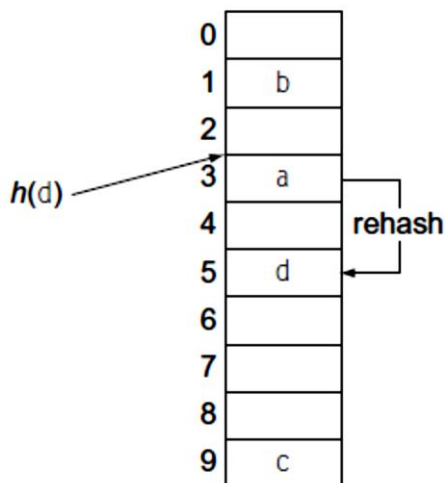


Рисунок 5. Открытая адресация

На рис. 5 показано, как происходит формирование таблицы идентификаторов для случая, когда $h(a) = h(d) = 3$. Коллизия при вставке d разрешается путем вычисления $g(d) = 2$, поэтому Insert вставляет d в ячейку $(h(d) + g(d)) \bmod S = (3 + 2) \bmod 10 = 5$ таблицы. Можно сказать, что при открытой адресации формируются цепочки идентификаторов, как и в методе цепочек, только хранятся они непосредственно в таблице. Ячейка таблицы может служить началом нескольких цепочек, формируемых с разным значением инкремента $g(\text{name})$.

При таком методе организации таблицы идентификаторов важно, чтобы h и g производили хорошие распределения, т.е. чтобы коллизии возникали нечасто. Тогда количество рехэширований будет небольшим, а значит время, затрачиваемое на вставку или поиск идентификатора в таблице будет низким.

К недостаткам данного метода можно отнести то, что размер S таблицы должен быть больше чем количество N идентификаторов. А это значит, что при больших значениях N таблица идентификаторов будет занимать значительный объем памяти.

Реализацию таблицы идентификаторов с помощью открытой адресации можно упростить, если использовать константную функцию для g . Это уменьшает время рехэширования, однако в этом случае формируется одна цепочка рехэширования для каждого значения h , что приводит к слиянию цепочек рехэширования всякий раз, когда второй индекс указывает на уже занятую ячейку таблицы.

Размер S таблицы идентификаторов играет важную роль при открытой адресации. Функция LookUp должна уметь распознавать ситуацию, когда она попадает в ячейку, в которой уже была, иначе поиск будет происходить бесконечно. Это возможно, если функция будет гарантировать, что в конечном итоге произойдет возврат к значению $h(n)$, с которого начинался поиск. Если S – простое число, тогда при любом $0 < g(n) < S$ будет сгенерирована

последовательность p_1, p_2, \dots, p_s , обладающая свойствами $p_1 = p_s = h(n)$ и $p_i \neq h(n), \forall i: 1 < i < s$. Таким образом, LookUp пройдет по каждой ячейке таблицы идентификаторов прежде, чем возвратится к $h(n)$.

Хранение информации об идентификаторах

Ни открытая адресация, ни метод цепочек не дают ответа на вопрос, как выделять память для хранения информации, связанной с элементами хэш-таблицы. При использовании метода цепочек предпочтение отдается хранению всей, связанной с идентификатором информации, непосредственно в элементах цепочки. При открытой адресации данные, относящиеся к идентификатору, хранятся в самой хэш-таблице. Однако такие подходы не лишены недостатков. Лучших результатов можно достичь, если использовать отдельно выделенный стек, как показано на рис. 6.

В этом случае цепочки реализуются на стеке, что снижает затраты на выделение памяти для индивидуальных записей. Кроме того, когда данные сохраняются в стеке, они тем самым формируют плотную таблицу, что делает более эффективными операции чтения и записи.

И, наконец, такая схема существенно упрощает задачу увеличения множества индексов. Старое множество индексов удаляется, выделяется память для множества большего размера, а затем записи вставляются в новую таблицу, начиная со дна стека до его вершины. При этом не надо хранить в памяти одновременно старую и новую таблицы. И не происходит попаданий в пустые ячейки таблицы, что возможно, когда при открытой адресации расширяется индексное множество для обеспечения коротких последовательностей рехэширования.

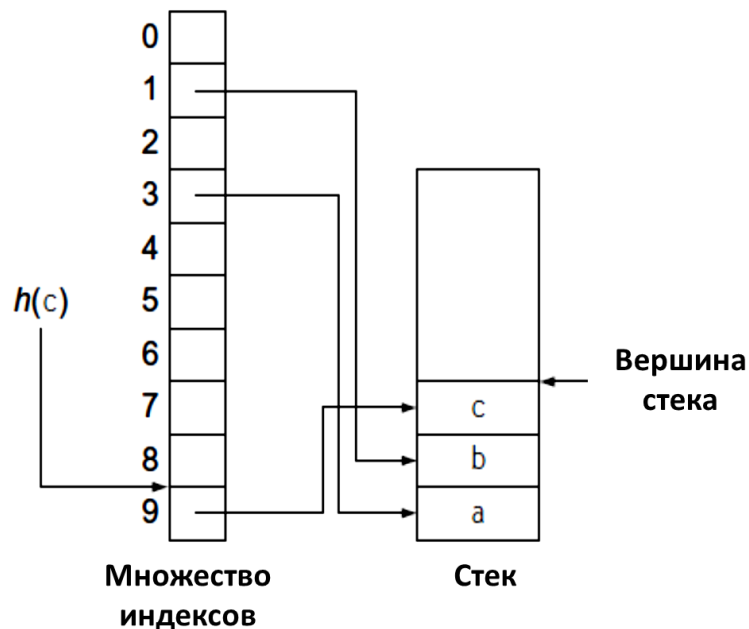


Рисунок 6. Стек для хранения данных об идентификаторах

Память не должна сразу выделяться для всего стека в целом. Стек реализуется как последовательность узлов, каждый из которых содержит k записей. Когда все записи узла заполнены данными, создается и добавляется к последовательности новый узел.

Вложенные области видимости

Ранее был рассмотрен вопрос создания таблиц идентификаторов с учетом вложенных областей видимостей переменных. Предложенный метод прост в реализации, но недостаточно

эффективен, так как основные затраты от вложенности несет в себе функция `LookUp`. Поскольку эта функция вызывается компилятором чаще, чем остальные, имеет смысл рассмотреть другие реализации.

Код, представленный на рис. 2, выполняет следующие действия:

```
↑ <w,0> <x,0> <example,0> ↑ <a,1> <b,1> <c,1>  
↑ <b,2> <z,2> ↓ ↑ <a,2> <x,2> ↑ <c,3>, <x,3> ↓ ↓ ↓ ↓
```

где \uparrow представляет собой вызов функции `InitializeScope`, \downarrow – вызов `FinalizeScope` и $\langle \text{name}, n \rangle$ – вызов функции `Insert`, которая добавляет идентификатор с именем `name` на уровень `n`.

Метод цепочек и вложенные области видимости

Если в методе цепочек будут учитываться уровни вложенности областей видимости, то функция `Insert` сможет проверять наличие дублирующихся имен, сравнивая и уровни областей видимости. `LookUp` будет возвращать первую найденную запись в таблице для данного идентификатора. `InitializeScope` просто будет увеличивать счетчик текущего лексического уровня. При такой схеме основные сложности реализуются в функции `FinalizeScope`, которая должна не только уменьшать счетчик текущего лексического уровня, но и удалять записи для идентификаторов данной области видимости, для которых происходит освобождение памяти.

Если память для элементов цепочек выделяется индивидуально, как показано на рис. 7, тогда функция `FinalizeScope` должна обнаружить все подлежащие удалению записи в данной области видимости и удалить их из соответствующих цепочек. Если эти записи в дальнейшем не будут использоваться компилятором, `FinalizeScope` должна их удалить, иначе они должны быть собраны в одну цепочку и сохранены. На рисунке 7 показана таблица, реализующая этот подход для программного кода из примера.

В случае, если память для записей выделяется в стеке, `FinalizeScope` может пройти по стеку сверху вниз, пока не достигнет первой записи, находящейся ниже лексического уровня, который нужно удалить. Для каждой записи обновляется запись в индексном множестве, в которую записывается указатель на следующий элемент цепочки. Если записи удаляются, `FinalizeScope` устанавливает значение указателя на следующий доступный блок, иначе помеченные к удалению записи сохраняются вместе в стеке. На рисунке 8 показана реализованная таким образом таблица идентификаторов для кода из примера.

Открытая адресация и вложенные области видимости

Если таблица идентификаторов реализована с помощью открытой адресации, добавление информации об уровнях вложенности реализуется сложнее. Записи в таблице являются критическим ресурсом. Когда таблица полностью заполнена, она должна быть расширена, прежде чем произойдет добавление новой записи. Удаление данных из таблицы, использующей хеширование, выполнить трудно. Ведь удаляемая запись может быть частью цепочки хеширования.

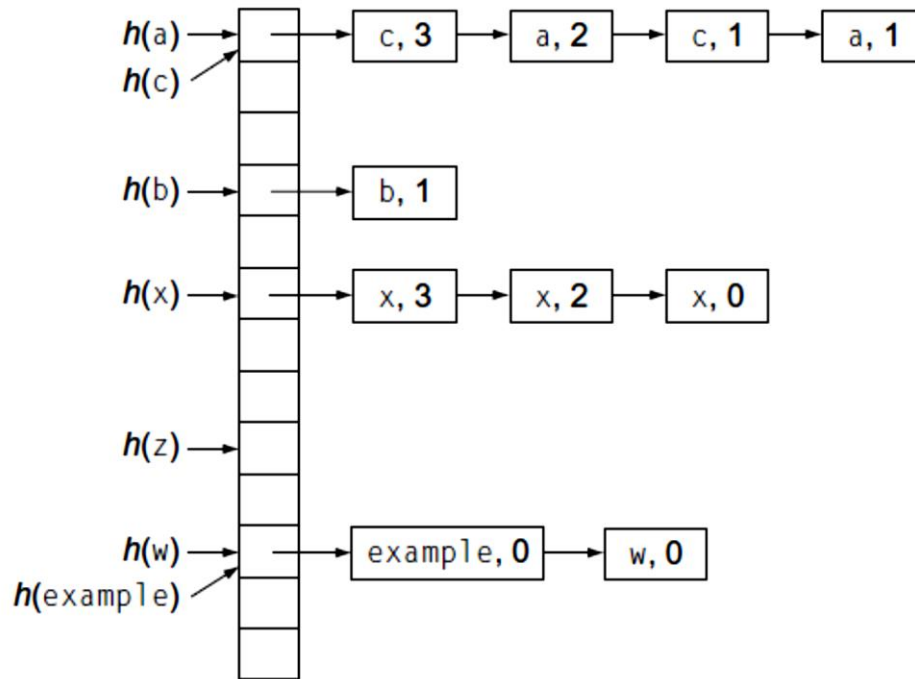


Рисунок 7. Метод цепочек и вложенные области видимости при индивидуальном выделении памяти для элементов цепочек

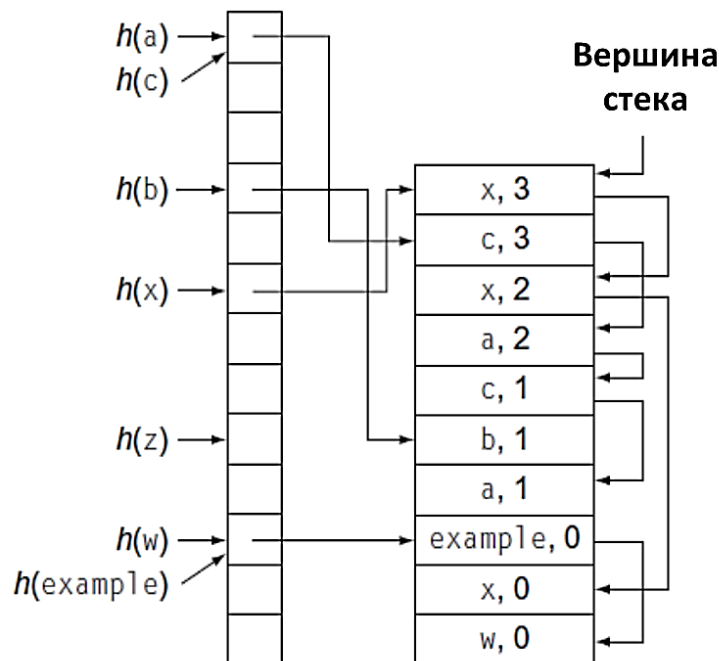


Рисунок 8. Метод цепочек и вложенные области видимости при выделении памяти в стеке

Следующий рисунок иллюстрирует работу с вложенными областями видимости переменных при открытой адресации. Функция `Insert` создает новую запись на вершине стека. Если она обнаруживает более раннее объявление того же имени во множестве индексов, то она замещает эту ссылку ссылкой на новую запись и связывает между собой старую запись и новую. Функция `FinalizeScope` выполняет перебор верхних значений в стеке, как и в случае с методом цепочек. При удалении записи об идентификаторе, имеющем другие варианты

объявлений, происходит изменение указателя во множестве индексов на более старый вариант объявления. Если удаляется единственная запись об идентификаторе, то должна произойти вставка ссылки на специальную запись, обозначающую удаленную ссылку. Функция `LookUp` должна уметь распознавать удаленные ссылки как части цепочек рехэширования. А `Insert` должна знать, что удаленные ссылки можно заменять вновь вставленными записями.

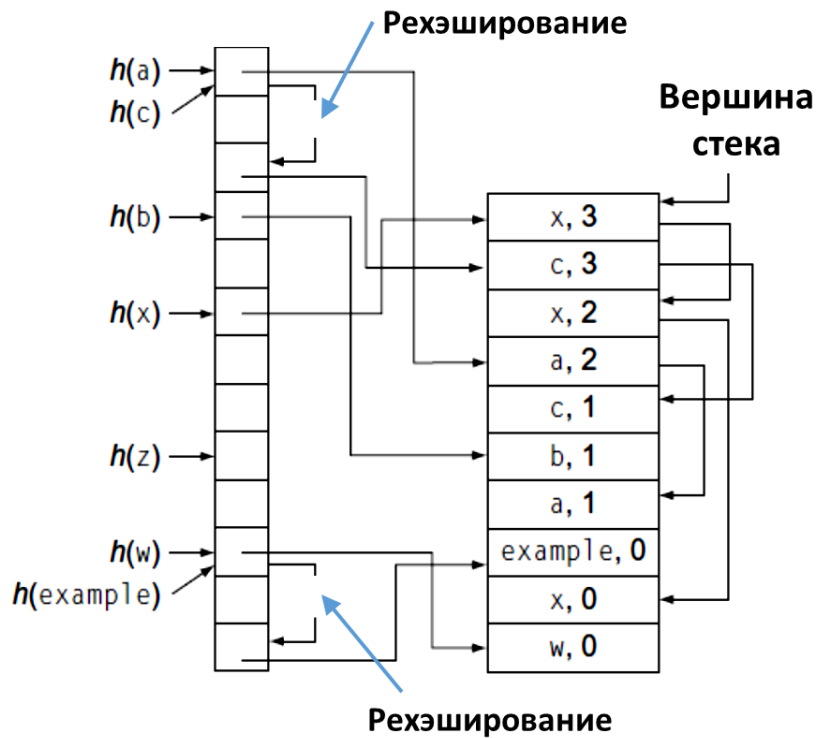


Рисунок 9. Открытая адресация и вложенные области видимости

Задание на лабораторную работу

Добавить в программу, разработанную в лабораторной работе №2, функции формирования таблицы идентификаторов и поиска в ней по заданному методу. Для студентов с четными номерами по журналу – метод цепочек, с нечетными – открытая адресация.

Контрольные вопросы

1. Что такое таблица идентификаторов, и для чего она предназначена?
2. Какая информация может храниться в таблице идентификаторов?
3. Какие цели преследуются при организации таблицы идентификаторов?
4. Какими характеристиками могут обладать константы, переменные?
5. Какие существуют способы организации таблиц идентификаторов?
6. Что такое коллизия? Почему она происходит?
7. Что такое хэш-функции и для чего они используются?
8. Что такое рехэширование? В чём заключается метод открытой адресации?
9. В чём заключается метод цепочек?

Список литературы

1. Cooper K.D., Torczon L. Engineering a Compiler, 2nd ed. – Elsevier, Inc., 2012. – 825 p.
2. Kakde O.G. Algorithms for Compiler Design. – Charles River Media, 2002. – 334 p.