ЛАБОРАТОРНА РОБОТА № 1

Організація таблиць ідентифікаторів

Мета роботи

Мета роботи: вивчити основні методи організації таблиць ідентифікаторів, одержати уявлення про переваги і недоліки, властиві різним методам організації таблиць ідентифікаторів.

Для виконання лабораторної роботи потрібно написати програму, яка одержує на вході набір ідентифікаторів, організовує таблиці ідентифікаторів за допомогою заданих методів, дозволяє здійснити багатократний пошук довільного ідентифікатора в таблицях і порівняти ефективність методів організації таблиць. Список ідентифікаторів вважати заданим у вигляді текстового файлу. Довжина ідентифікаторів обмежена 32 символами.

Короткі теоретичні відомості. Призначення таблиць ідентифікаторів

При виконанні семантичного аналізу, генерації коду і оптимізації результуючої програми компілятор повинен оперувати характеристиками основних елементів початкової програми — змінних, констант, функцій і інших лексичних одиниць вхідної мови. Ці характеристики можуть бути одержані компілятором на етапі синтаксичного аналізу вхідної програми (найчастіше при аналізі структури блоків описів змінних і констант), а також доповнені на етапі підготовки до генерації коду (наприклад при розподілі пам'яті).

Набір характеристик, відповідний кожному елементу початкової програми, залежить від типу цього елементу, від його сенсу (семантики) і, відповідно, від тієї ролі, яку він виконує в початковій і результуючій програмах. У кожному конкретному випадку цей набір характеристик може бути свій залежно від синтаксису і семантики вхідної мови, від архітектури цільової

обчислювальної системи і від структури компілятора. Але є типові характеристики, які найчастіше властиві тим або іншим елементам початкової програми. Наприклад для змінної — це її тип і адреса елементу пам'яті, для константи — її значення, для функції - кількість і типи формальних аргументів, тип результату, що повертається, адреса виклику коду функції. Докладнішу інформацію про характеристики елементів початкової програми, їх аналіз і використання можна знайти в [1,3, 7].

Головною характеристикою будь-якого елементу початкової програми є його ім'я. Саме з іменами змінних, констант, функцій і інших елементів вхідної мови оперує розробник програми - тому і компілятор повинен уміти аналізувати ці елементи по їх іменах.

Ім'я кожного елементу повинне бути унікальним. Багато сучасних мов програмування допускають збіги (неунікальність) імен змінних і функцій залежно від їх області видимості і інших умов початкової програми. В цьому випадку унікальність імен повинен забезпечувати сам компілятор - про те, як вирішується ця проблема, можна дізнатися в [1 - 3, 7], тут же вважатимемо, що імена елементів початкової програми завжди є унікальними.

Таким чином, завдання компілятора полягає в тому, щоб зберігати деяку інформацію, пов'язану з кожним елементом початкової програми, і мати доступ до цієї інформації по імені елементу. Для вирішення цього завдання компілятор організовує спеціальні сховища даних, звані *таблицями ідентифікаторів,* або *таблицями символів.* Таблиця ідентифікаторів складається з набору полів даних (записів), кожне з яких може відповідати одному елементу початкової програми. Запис містить всю необхідну компілятору інформацію про даний елемент і може поповнюватися у міру роботи компілятора. Кількість записів залежить від способу організації таблиці ідентифікаторів, але у будь-якому випадку їх не може бути менше, ніж елементів в початковій програмі. В принципі, компілятор може працювати не з однією, а з декількома таблицями ідентифікаторів - їх кількість і структура залежать від реалізації компілятора [1,2].

Принципи організації таблиць ідентифікаторів

Компілятор поповнює записи в таблиці ідентифікаторів по мірі аналізу початкової програми і виявлення в ній нових елементів, що вимагають розміщення в таблиці. Пошук інформації в таблиці виконується кожен раз, коли компілятору необхідні відомості про той або інший елемент програми. Причому слід відмітити, що пошук елементу в таблиці виконуватиметься компілятором істотно частіше, ніж переміщення в неї нових елементів. Так відбувається тому, що описи нових елементів в початковій програмі, як правило, зустрічаються набагато рідше, ніж ці елементи використовуються. Крім того, кожному додаванню елементу в таблицю ідентифікаторів у будь-якому випадку передуватиме операція пошуку — щоб переконатися, що такого елементу в таблиці немає.

На кожну операцію пошуку елементу в таблиці компілятор витрачатиме час, і оскільки кількість елементів в початковій програмі велика (від одиниць до сотень тисяч залежно від об'єму програми), цей час істотно впливатиме на загальний час компіляції. Тому таблиці ідентифікаторів повинні бути організовані так, щоб компілятор мав можливість максимально швидко виконувати пошук потрібного йому запису в таблиці по імені елементу, з яким пов'язаний цей запис.

Можна виділити наступні способи організації таблиць ідентифікаторів:

- прості і впорядковані списки;
- бінарне дерево;
- хеш-адресація з рехешуванням;
- хеш-адресація по методу ланцюжків;
- комбінація хш-адресації із списком або бінарним деревом.

Далі буде дано короткий опис всіх вище перелічених способів організації таблиць ідентифікаторів. Докладнішу інформацію можна знайти в [3.7].

Прості методи побудови таблиць ідентифікаторів

У простому випадку таблиця ідентифікаторів є лінійним

неврегульованим списком, або масивом, кожен осередок якого містить дані про відповідний елемент таблиці. Розміщення нових елементів в такій таблиці виконується шляхом запису інформації в чергову чарунку масиву або списку по мірі виявлення нових елементів в початковій програмі. Пошук потрібного елементу в таблиці в цьому випадку виконуватиметься шляхом послідовного перебору всіх елементів і порівняння їх імені з ім'ям шуканого елементу, поки не буде знайдений елемент з таким же ім'ям. Тоді якщо за одиницю часу прийняти час, що витрачається компілятором на порівняння двох рядків (у сучасних обчислювальних системах таке порівняння найчастіше виконується однією командою), то для таблиці, що містить N елементів, в середньому буде виконано N/2 порівнянь.

Час, потрібний на додавання нового елементу в таблицю (ТД), не залежить від числа елементів в таблиці (N). Але якщо N велике, то пошук буде вимагати значних витрат часу. Час пошуку (Тп) в такій таблиці можна оцінити як Тп=O(N). Оскільки саме пошук в таблиці ідентифікаторів є найбільш часто виконуваною компілятором операцією, такий спосіб організації таблиць ідентифікаторов є неефективним. Він застосовується тільки для найпростіших компіляторів, що працюють з невеликими програмами.

Пошук може бути виконаний ефективніше, якщо елементи таблиці відсортовані (впорядковані) природним чином. Оскільки пошук здійснюється по імені, найбільш природним рішенням буде розташувати елементи таблиці в прямому або зворотному алфавітному порядку. Ефективним методом пошуку у впорядкованому списку з N елементів є *бінарний,* або *логарифмічний пошук.*

Алгоритм логарифмічного пошуку полягає в наступному: шуканий символ порівнюється з елементом (N+1)/2 в середині таблиці; якщо цей елемент не є шуканим, то ми повинні проглянути тільки блок елементів, пронумерованих від 1 до (N + 1) /2-1, або блок елементів від (N+1) /2+1 до N залежно від того, менше або більше шуканий елемент того, з яким його порівняли. Потім процес повторюється над потрібним блоком в два рази меншого розміру. Так продовжується до тих пір, поки або шуканий елемент не буде знайдений, або

алгоритм не дійде до чергового блоку, що містить один або два елементи (з якими можна виконати пряме порівняння шуканого елементу). Оскільки на кожному кроці число елементів, які можуть містити шуканий елемент, скорочується в два рази, максимальне число порівнянь рівне 1 + 1og2N. Тоді час пошуку елементу в таблиці ідентифікаторів можна оцінити як Тп=O(1og2N). Для порівняння: при N=128 бінарний пошук вимагає найбільше 8 порівнянь, а пошук в неврегульованій таблиці — в середньому 64 порівняння. Метод називають «бінарним пошуком», оскільки на кожному кроці об'єм даної інформації скорочується в два рази, а «логарифмічним» — оскільки час, що витрачається на пошук потрібного елементу в масиві, має логарифмічну залежність від загальної кількості елементів в ньому. Недоліком логарифмічного пошуку є вимога впорядковування таблиці ідентифікаторів. Оскільки масив інформації, в якому виконується пошук, повинен бути впорядкований, час його заповнення вже залежатиме від числа елементів в масиві. Таблиця ідентифікаторів часто є видимим компілятором ще до того, як вона наповнена, тому потрібно, щоб умова впорядкованості виконувалася на всіх етапах звернення до неї. Отже, для побудови такої таблиці можна користуватися тільки алгоритмом прямого впорядкованого включення елементів.

Якщо користуватися стандартними алгоритмами, які використовуються для організації впорядкованих масивів даних, то середній час, необхідний на занесення всіх елементів в таблицю, можна оцінити таким чином:

Тд =O(N•log2N) + k•O(N2).

Тут k - деякий коефіцієнт, що відображає співвідношення між часом, що витрачається комп'ютером на виконання операції порівняння і операції занесення даних.

При організації логарифмічного пошуку в таблиці ідентифікаторів забезпечується істотне скорочення часу пошуку потрібного елементу за рахунок збільшення часу на додавання нового елементу в таблицю. Оскільки додавання нових елементів в таблицю ідентифікаторів відбувається істотно рідше, ніж

звернення до них, цей метод слід визнати ефективнішим, ніж метод організації неврегульованої таблиці. Проте в реальних компіляторах цей метод безпосередньо також не використовується, оскільки існують ефективніші методи.

Побудова таблиць ідентифікаторів по методу бінарного дерева

Можна скоротити час пошуку шуканого елементу в таблиці ідентифікаторів, не збільшуючи значно час, необхідний на її заповнення. Для цього треба відмовитися від організації таблиці у вигляді безперервного масиву даних. Існує метод побудови таблиць, при якій таблиця має форму бінарного дерева. Кожен вузол дерева є елементом таблиці, причому кореневим вузлом стає перший елемент, який зустрінеться компілятором при заповненні таблиці. Дерево називається бінарним, оскільки кожна вершина в ньому може мати не більше двох гілок. Для визначеності називатимемо дві гаілки: «права» і «ліва».

Розглянемо алгоритм заповнення бінарного дерева. Вважатимемо, що алгоритм працює з потоком вхідних даних, що містить ідентифікатор. Перший ідентифікатор, як вже було сказано, поміщається у вершину дерева. Всі подальші ідентифікатори потрапляють в дерево по наступному алгоритму:

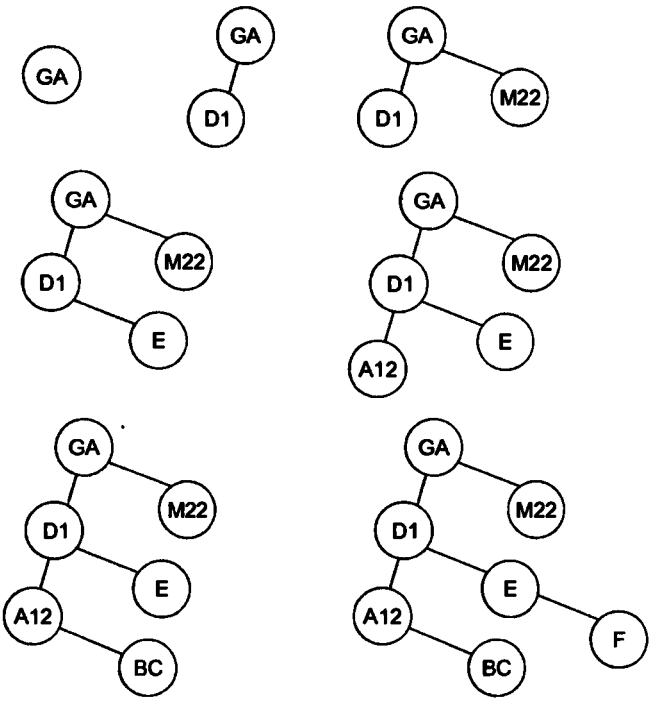
1. Вибрати черговий ідентифікатор з вхідного потоку даних. Якщо чергового ідентифікатора немає, то побудова дерева закінчена.
2. Зробити поточним вузлом дерева кореневу вершину.
3. Порівняти ім'я чергового ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.
4. Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, якщо рівне — припинити виконання алгоритму (двох однакових ідентифікаторів бути не повинно!), інакше — перейти до кроку 7.
5. Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше - перейти до кроку 6.
6. Створити нову вершину, помістити в неї інформацію про черговий

ідентифікатор, зробити цю нову вершину лівою вершиною поточного вузла і повернутися до кроку 1.

7. Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 3, інакше — перейти до кроку 8.

8. Створити нову вершину, помістити в неї інформацію про черговий ідентифікатор, зробити цю нову вершину правою вершиною поточного вузла і повернутися до кроку 1.

Розглянемо як приклад послідовність ідентифікаторів Ga, D1, M22, E, A12, BC, F. На мал. 1.1 проілюстрований весь процес побудови бінарного дерева для цієї послідовності ідентифікаторів.



Мал. 1.1. Заповнення бінарного дерева для послідовності ідентифікаторів Ga, D1, M22, E, A12, BC, F

Пошук елементу в дереві виконується по алгоритму, схожому з алгоритмом заповнення дерева:

1. Зробити поточним вузлом дерева кореневу вершину.

2. Порівняти ім'я шуканого ідентифікатора з ім'ям ідентифікатора, що міститься в поточному вузлі дерева.

3. Якщо імена співпадають, то шуканий ідентифікатор знайдений,

алгоритм завершується, інакше треба перейти до кроку 4.

4.Якщо ім'я чергового ідентифікатора менше, то перейти до кроку 5, інакше — перейти до кроку 6.

5.Якщо у поточного вузла існує ліва вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдений, алгоритм завершується.

6.Якщо у поточного вузла існує права вершина, то зробити її поточним вузлом і повернутися до кроку 2, інакше - шуканий ідентифікатор не знайдений, алгоритм завершується.

Для даного методу число необхідних порівнянь і форма дерева, що вийшло, залежать від того порядку, в якому надходять ідентифікатори. Наприклад, якщо в розглянутому вище прикладі замість послідовності ідентифікаторів Ga, D1, M22, E, A12, BC, F узяти послідовність A12, BC, D1, E, F, Ga, M22, то дерево виродиться у впорядкований однонаправлений зв'язний список. Ця особливість є недоліком даного методу організації таблиць ідентифікаторів. Іншими недоліками методу є: необхідність зберігати два додаткові посилання на ліву і праву гілку в кожному елементі дерева і робота з динамічним виділенням пам'яті при побудові дерева.

Якщо припустити, що послідовність ідентифікаторів в початковій програмі є статистично неврегульованою (що в цілому відповідає дійсності), то можна вважати, що побудоване бінарне дерево буде невиродженим. Тоді середній час на заповнення дерева (Тд) і на пошук елементу в нім (Тп) можна оцінити таким чином [3, 7]:

$$Тд=N•O(1og2N);$$

$$Тп=O(1og2N).$$

Не дивлячись на вказані недоліки, метод бінарного дерева є досить вдалим механізмом для організації таблиць ідентифікаторів. Він знайшов своє застосування у ряді компіляторів. Іноді компілятори будують декілька різних

дерев для ідентифікаторів різних типів і різної довжини [1,2,3,7].

Хеш-функції і хеш-адресація

У реальних початкових програмах кількість ідентифікаторів така велика, що навіть логарифмічну залежність часу пошуку від їх числа не можна визнати задовільною. Необхідні ефективніші методи пошуку інформації в таблиці ідентифікаторів. Кращих результатів можна досягти, якщо застосувати методи, зв'язані з використанням хеш-функцій і хеш-адресації.

Хеш-функцією F називається деяке відображення безлічі вхідних елементів R на безліч цілих ненегативних чисел Z: F(r)=n, r є R, N є Z. Сам термін «хеш-функція» походить від англійського терміну «hash function» (hash — «заважати», «змішувати», «плутати»).

Безліч допустимих вхідних елементів R називається областю визначення хеш-функції. Безліччю значень хеш-функції F називається підмножина M з безлічі цілих ненегативних чисел Z: M є Z, що містить всі можливі значення, які повертаються функцією F*:* ∀ rIR: F(r) є M і ∀ m є M: ∃ r є R: F(r)=m. Процес відображення області визначення хеш-функції на безліч значень називається *хешуванням.*

При роботі з таблицею ідентифікаторів хеш-функція повинна виконувати відображення імен ідентифікаторів на безліч цілих ненегативних чисел. Областю визначення хеш-функції буде безліч всіх можливих імен ідентифікаторів.

Хеш-адресація полягає у використанні значення, яке повертається хеш-функцією, як адреса осередку з деякого масиву даних. Тоді розмір масиву даних повинен відповідати області значень використовуваної хеш-функції.

Отже, в реальному компіляторі область значень хеш-функції ніяк не повинна перевищувати розмір доступного адресного простору комп'ютера.

Метод організації таблиць ідентифікаторів, заснований на використанні хеш-адресації, полягає в додаванні кожного елементу таблиці в чарунку, адресу якої повертає хеш-функція, обчислена для цього елементу. Тоді в ідеальному випадку для додавання будь-якого елементу в таблицю ідентифікаторів досить

тільки обчислити його хеш-функцію і звернутися до потрібної чарунки масиву даних. Для пошуку елементу в таблиці також необхідно обчислити хеш-функцію для шуканого елементу і перевірити, чи не є задана нею чарунка масиву порожньою (якщо вона не порожня — елемент знайдений, якщо порожня — не знайдений). Спочатку таблиця ідентифікаторів повинна бути заповнена інформацією, яка дозволила б говорити про те, що всі її чарунки є порожніми. Цей метод вельми ефективний, оскільки як час розміщення елементу в таблиці, так і час його пошуку визначаються тільки часом, що витрачається на обчислення хеш-функції, яке в загальному випадку незіставно менше часу, необхідного для багатократних порівнянь елементів таблиці. Метод має два очевидні недоліки. Перший з них — неефективне використання об'єму пам'яті під таблицю ідентифікаторів: розмір масиву для її зберігання повинен відповідати всій області значень хеш-функції, тоді як ідентифікаторів, що реально зберігаються в таблиці, може бути істотно менше. Другий недолік — необхідність відповідного розумного вибору хеш-функції. Цей недолік є настільки істотним, що не дозволяє безпосередньо використовувати хеш-адресацію для організації таблиць ідентифікаторів.

Проблема вибору хеш-функції не має універсального рішення. Хешування зазвичай відбувається за рахунок виконання над ланцюжком символів деяких простих арифметичних і логічних операцій. Найпростішою хеш-функцією для символу є код внутрішнього уявлення в комп'ютері літери символу. Цю хеш-функцію можна використовувати і для ланцюжка символів, вибираючи перший символ в ланцюжку.

Очевидно, що така примітивна хеш-функція буде незадовільною: при її використанні виникне проблема - двом різним ідентифікаторам, що починаються з однієї і тієї ж букви, відповідатиме одне і те ж значення хеш-функції. Тоді при хеш-адресації в одну і ту ж чарунку таблиці ідентифікаторів повинні бути поміщені два різні ідентифікатори, що явно неможливо. Така ситуація, коли двом або більше ідентифікаторам відповідає одне і те ж значення хеш-функції, називається *колізією.*

Природно, що хеш-функція, що допускає колізії, не може бути використана для хеш-адресації в таблиці ідентифікаторів. Причому досить одержати хоча б один випадок колізії на всій множині ідентифікаторів, щоб такою хеш-функцією не можна було користуватися. Але чи можливо побудувати хеш-функцію, яка б повністю виключала виникнення колізій? Для повного виключення колізій хеш-функція повинна бути взаємно однозначною: кожному елементу з області визначення хеш-функції повинне відповідати одне значення з її множини значень, і навпаки — кожному значенню з множини значень цієї функції повинен відповідати тільки один елемент з її області визначення. Тоді будь-яким двом довільним елементам з області визначення хеш-функції завжди відповідатимуть два різних її значення. Теоретично для ідентифікаторів таку хеш-функцію побудувати можна, оскільки і область визначення хеш-функції (всі можливі імена ідентифікаторів), і область цих значень (цілі ненегативні числа) є нескінченними рахунковими множинами, тому можна організувати взаємнооднозначне відображення однієї множини на іншу.

Але на практиці існує обмеження, що робить створення взаємнооднозначної хеш-функції для ідентифікаторів неможливим. Річ у тому, що в реальності область значень будь-якої хеш-функції обмежена розміром доступного адресного простору комп'ютера. Множина адрес будь-якого комп'ютера з традиційною архітектурою може бути велика, але завжди скінченна. Організувати взаємно однозначне відображення нескінченної множини на кінцеве навіть теоретично неможливо. Можна, звичайно, врахувати, що довжина частини імені ідентифікатора, що приймається до уваги, в реальних компіляторах на практиці також обмежена — зазвичай вона лежить в межах від 32 до 128 символів (тобто і область визначення хеш-функції кінцева). Але і тоді кількість елементів в кінцевій множині, що становить область визначення хеш-функції, перевищуватиме їх кількість в кінцевій множині області її значень (кількість всіх можливих ідентифікаторів більше кількості допустимих адрес в сучасних комп'ютерах). Таким чином, створити взаємно однозначну хеш-

функцію на практиці неможливо. Отже, неможливо уникнути виникнення колізій.

Тому не можна організувати таблицю ідентифікаторів безпосередньо на основі однієї тільки хеш-адресації. Але існують методи, що дозволяють використовувати хеш-функції для організації таблиць ідентифікаторів навіть за наявності колізій.

Хеш-адресація з рехешуванням

Для вирішення проблеми колізії можна використовувати багато способів. Одним з них є метод *рехешування* (або розстановки). Згідно цього методу, якщо для елементу A адреса n0=h(A), обчислена за допомогою хеш-функції h, вказує на вже зайняту чарунку, то необхідно обчислити значення функції n1=h1(A) і перевірити зайнятість чарунки за адресою n1. Якщо і вона зайнята, то обчислюється значення h2(A), і так до тих пір, поки або не буде знайдена вільна чарунка, або чергове значення hi(A) не співпаде з h(A). У останньому випадку вважається, що таблиця ідентифікаторів заповнена і місця в ній більше немає — видається інформація про помилку розміщення ідентифікатора в таблиці. Тоді пошук елементу A в таблиці ідентифікаторів, організованій таким чином, виконуватиметься але наступному алгоритму:

1. Обчислити значення хеш-функції n=h(A) для шуканого елементу *A*.

2. Якщо чарунка за адресою n порожня, то елемент не знайдений, алгоритм завершений, інакше необхідно порівняти ім'я елементу в чарункі n з ім'ям шуканого елементу *A*. Якщо вони співпадають, то елемент знайдений і алгоритм завершений, інакше i:=1 і перейти до кроку 3.

3. Обчислити ni = hi(A). Якщо чарунка за адресою ni порожня або n=ni, то елемент не знайдений і алгоритм завершений, інакше - порівняти ім'я елементу в чарунці ni з ім'ям шуканого елементу *A*. Якщо вони співпадають, то елемент знайдений і алгоритм завершений, інакше i:=i + 1 і повторити крок 3.

Алгоритми розміщення і пошуку елементу схожі по виконуваних операціях. Тому вони матимуть однакові оцінки часу, необхідного для їх

виконання.

При такій організації таблиць ідентифікаторів у разі виникнення колізії алгоритм поміщає елементи в порожні елементи таблиці, вибираючи їх певним чином. При цьому елементи можуть потрапляти в чарунки з адресами, які потім співпадатимуть із значеннями хеш-функції, що приведе до виникнення нових, додаткових колізій. Таким чином, кількість операцій, необхідних для пошуку або розміщення в таблиці елементу, залежить від заповненої таблиці.

Для організації таблиці ідентифікаторів по методу рехешування необхідно визначити всі хеш-функції hi для всіх i. Найчастіше функції hi визначають як деякі модифікації хеш-функцій h. Наприклад, найпростішим методом обчислення функції hi(A) є її організація у вигляді hi(A)=(h(A)+pi)modNm, де pi — деяке обчислюване ціле число, а Nm — максимальне значення з області значень хеш-функції h. У свою чергу, найпростішим підходом тут буде покласти pi=i. Тоді одержуємо формулу hi(A)=(h(A)+i)modNm . В цьому випадку при збігу значень хеш-функції для яких-небудь елементів пошук вільної чарунки в таблиці починається послідовно від поточної позиції, заданою хеш-функцією h(A).

Цей спосіб не можна визнати особливо вдалим: при збігу хеш-адрес елементи в таблиці починають групуватися навколо них, що збільшує число необхідних порівнянь при пошуку і розміщенні. Але навіть такий примітивний метод рехешування є достатньо ефективним засобом організації таблиць ідентифікаторів при неповному заповненні таблиці. Середній час на додавання одного елементу в таблицю і на пошук елементу в таблиці можна понизити, якщо застосувати більш довершений метод рехешування. Одним з таких методів є використання як pi для функції hi(A)=(h(A)+pi)modNm послідовності псевдовипадкових цілих чисел p1, p2…p_k. При хорошому виборі генератора псевдовипадкових чисел довжина послідовності k=Nm .

Існують і інші методи організації функцій рехешування hi(A), засновані на квадратичних обчисленнях або, наприклад, на обчисленні піднесення за формулою: hi(A)=(h(A)N•i)mod N’m де N’m —найближче просте число, менше

Nm. В цілому рехешування дозволяє добитися непоганих результатів для ефективного пошуку елементу в таблиці (кращих, ніж бінарний пошук і бінарне дерево), Але ефективність методу сильно залежить від заповненої таблиці ідентифікаторів і якості використовуваної хеш-функції — чим рідше виникають колізії, тим вище ефективність методу. Вимога неповного заповнення таблиці веде до неефективного використання об'єму доступної пам'яті.

Оцінки часу розміщення і пошуку елементу в таблицях ідентифікаторів при використанні різних методів рехешування можна знайти в [1,3,7].

Хеш-адресація з використанням методу ланцюжків

Неповне заповнення таблиці ідентифікаторів при застосуванні рехешування веде до неефективного використання всього об'єму пам'яті, доступного компілятору. Причому об'єм невживаної пам'яті буде тим вище, чим більше інформації зберігається для кожного ідентифікатора. Цього недоліку можна уникнути, якщо доповнити таблицю ідентифікаторів деякою проміжною хеш-таблицею.

В чарунках хеш-таблиці може зберігатися або порожнє значення, або значення вказівника на деяку область пам'яті з основної таблиці ідентифікаторів. Тоді хеш-функція обчислює адресу, по якій відбувається звернення спочатку до хеш-таблиці, а потім вже через неї за знайденою адресою — до самої таблиці ідентифікаторів. Якщо відповідний елемент таблиці ідентифікаторів порожній, то чарунка хеш-таблиці міститиме порожнє значення. Тоді зовсім не обов'язково мати в самій таблиці ідентифікаторів чарунку для кожного можливого значення хеш-функції - таблицю можна зробити динамічною, так щоб її об'єм ріс по мірі заповнення (спочатку таблиця ідентифікаторів не містить жодної чарунки, а всі чарунки хеш-таблиці мають порожнє значення).

Такий підхід дозволяє добитися двох позитивних результатів: по-перше, немає необхідності заповнювати порожніми значеннями таблицю ідентифікаторів - це можна зробити тільки для хеш-таблиці; по-друге, кожному ідентифікатору відповідатиме строго одна чарунка в таблиці ідентифікаторів.

Порожні чарунки у такому разі будуть тільки в хеш-таблиці, і об'єм невживаної пам'яті не залежатиме від об'єму інформації, що зберігається для кожного ідентифікатора, — для кожного значення хеш-функції витрачатиметься тільки пам'ять, необхідна для зберігання одного вказівника на основну таблицю ідентифікаторів.

На основі цієї схеми можна реалізувати ще один спосіб організації таблиць ідентифікаторів за допомогою хеш-функції, який називається *методом ланцюжків.* В цьому випадку в таблицю ідентифікаторів для кожного елементу додається ще одне поле, в якому може міститися посилання на будь-який елемент таблиці. Спочатку це поле завжди порожнє (нікуди не вказує). Також необхідно мати одну спеціальну змінну, яка завжди вказує на перший вільний елемент основної таблиці ідентифікаторів (спочатку вона вказує на початок таблиці). Метод ланцюжків працює по наступному алгоритму:

1. У всі чарунки хеш-таблиці помістити порожнє значення, таблиця ідентифікаторів порожня, змінна FreePtr (вказівник першої вільної чарунки) вказує на початок таблиці ідентифікаторів.

2.Обчислити значення хеш-функції n для нового елементу A. Якщо чарунка хеш-таблиці за адресою n порожня, то помістити в неї значення змінної *FreePtr* і перейти до кроку 5; інакше перейти до кроку 3.

3.Вибрати з хеш-таблиці адресу елементу таблиці ідентифікаторів m і перейти до кроку 4.

4.Для елементу таблиці ідентифікаторів за адресою m перевірити значення поля посилання. Якщо воно порожнє, то записати в нього адресу змінної *FreePtr* і перейти до кроку 5; інакше вибрати з поля посилання нову адресу m і повторити крок 4.

5.Додати в таблицю ідентифікаторів нову чарунку, записати в неї інформацію для елементу *A* (поле посилання повинне бути порожнім), в змінну *FreePtr* помістити адресу за кінцем доданої чарунки. Якщо більше немає ідентифікаторів, які треба помістити в таблицю, то виконання алгоритму закінчене, інакше перейти до кроку 2.

Пошук елементу в таблиці ідентифікаторів, організованій таким чином, виконуватиметься по наступному алгоритму:

1. Обчислити значення хеш-функції n для шуканого елементу *A*. Якщо чарунка хеш-таблиці за адресою n порожня, то елемент не знайдений і алгоритм завершений, інакше вибрати з хеш-таблиці адресу елементу таблиці ідентифікаторів m.

2. Порівняти ім'я елементу в елементі таблиці ідентифікаторів за адресою m з ім'ям шуканого елементу A. Якщо вони співпадають, то шуканий елемент знайдений і алгоритм завершений, інакше перейти до кроку 3.

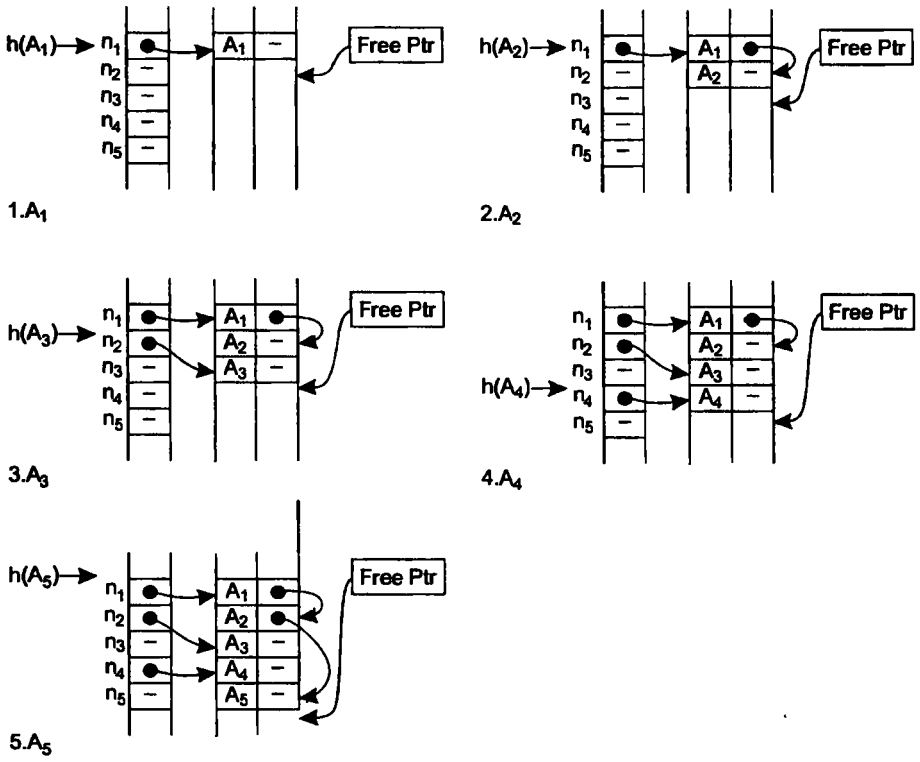
3. Перевірити значення поля посилання в чарунці таблиці ідентифікаторів за адресою m. Якщо воно порожнє, то шуканий елемент не знайдений і алгоритм завершений; інакше вибрати з поля посилання адресу m і перейти до кроку 2.

При такій організації таблиць ідентифікаторів у разі виникнення колізії алгоритм поміщає елементи в елементи таблиці, пов'язуючи їх один з одним послідовно через поле посилання. При цьому елементи не можуть потрапляти в чарунки з адресами, які потім співпадатимуть із значеннями хеш-функції. Таким чином, додаткові колізії не виникають. У результаті в таблиці виникають своєрідні ланцюжки зв'язаних елементів, звідки і походить назва даного методу – «метод ланцюжків».

На мал. 1.2 проілюстровано заповнення хеш-таблиці і таблиці ідентифікаторів для ряду ідентифікаторів: A1, A2, A3, A4, A5 за умови що h(A1)=h(A2)=h(A5)=n1; h(A3)=n2; h(A4)=n4. Після розміщення в таблиці для пошуку ідентифікатора A1 буде потрібно одне порівняння, для A2 — два порівняння, для A3 —одне порівняння, для A4 - одне порівняння і для A5 — три порівняння (спробуйте порівняти ці дані з результатами, одержаними з використанням простого рехешування для тих же ідентифікаторів).

Метод ланцюжків є дуже ефективним засобом організації таблиць ідентифікаторів. Середній час на розміщення одного елементу і на пошук елементу в таблиці для нього залежить тільки від середнього числа колізій, що

виникають при обчисленні хеш-функції. Накладні витрати пам'яті, пов'язані з необхідністю мати одне додаткове поле вказівника в таблиці ідентифікаторів на кожен її елемент, можна визнати цілком виправданими, оскільки виникає економія використовуваної пам'яті за рахунок проміжної хеш-таблиці. Цей метод дозволяє економніше використовувати пам'ять, але вимагає організації роботи з динамічними масивами даних.



Мал. 1.2. Заповнення таблиці ідентифікаторів при використанні методу ланцюжків

Комбіновані способи побудови таблиць ідентифікаторів

Окрім рехешування і методу ланцюжків можна використовувати комбіновані методи для організації таблиць ідентифікаторів за допомогою хеш-адресації. В цьому випадку для виключення колізій хеш-адресація поєднується з одним з раніше розглянутих методів — простим списком, впорядкованим списком або бінарним деревом, який використовується як додатковий метод впорядковування ідентифікаторів, для яких виникають колізії. Причому, оскільки при якісному виборі хеш-функції кількість колізій зазвичай невелика (одиниці або десяти випадків), навіть простій список може бути цілком

задовільним рішенням при використанні комбінованого методу.

При такому підході можливі два варіанти: у першому випадку, як і для методу ланцюжків, в таблиці ідентифікаторів організовується спеціальне додаткове поле посилання. Але на відміну від методу ланцюжків воно має декілька інше значення: за відсутності колізій для вибірки інформації з таблиці використовується хеш-функція, поле посилання залишається порожнім. Якщо ж виникає колізія, то через поле посилання організовується пошук ідентифікаторів, для яких значення хеш-функції співпадають, - це поле повинне указувати на структуру даних для додаткового методу: початок списку, перший елемент динамічного масиву або кореневий елемент дерева.

У другому випадку використовується хеш-таблиця, аналогічна хеш-таблиці для методу ланцюжків. Якщо за даною адресою хеш-функції ідентифікатор відсутній, то осередок хеш-таблиці порожній. Коли з'являється ідентифікатор з даним значенням хеш-функції, то створюється відповідна структура для додаткового методу, в хеш-таблицю записується посилання на цю структуру, а ідентифікатор поміщається в створену структуру за правилами вибраного додаткового методу.

У першому варіанті за відсутності колізії пошук виконується швидше, але другий варіант переважно, оскільки за рахунок використання проміжної хеш-таблиці забезпечується ефективніше використання пам'яті.

Як і для методу ланцюжків, для комбінованих методів час розміщення і час пошуку елементу в таблиці ідентифікаторів залежить тільки від середнього числа колізій, що виникають при обчисленні хеш-функції. Накладні витрати пам'яті при використанні проміжної хеш-таблиці мінімальні. Очевидно, що якщо як додатковий метод використовувати простій список, то вийде алгоритм, повністю аналогічний методу ланцюжків. Якщо ж використовувати впорядкований список або бінарне дерево, то метод ланцюжків і комбіновані методи матимуть приблизно рівну ефективність при незначному числі колізій (одиничні випадки), але із зростанням кількості колізій ефективність комбінованих методів в порівнянні з методом ланцюжків зростатиме.

Недоліком комбінованих методів є складніша організація алгоритмів пошуку і розміщення ідентифікаторів, необхідність роботи з динамічно розподілюваними областями пам'яті, а також великі витрати часу на розміщення нового елементу в таблиці ідентифікаторів в порівнянні з методом ланцюжків.

То, який конкретно метод застосовується в компіляторі для організації таблиць ідентифікаторів, залежить від реалізації компілятора. Один і той же компілятор може мати навіть декілька різних таблиць ідентифікаторів, організованих на основі різних методів. Як правило, застосовуються комбіновані методи. Створення ефективної хеш-функції - це окреме завдання розробників компіляторів, і отримані результати, як правило, тримаються у секреті. Хороша хеш-функція розподіляє ідентифікатори, що на її вхід, рівномірно на адреси, щоб звести до мінімуму кількість колізій. В даний час існує безліч хеш-функцій, але, як було показано вище, ідеального хешування досягти неможливо.

Хеш-адресація — це метод, який застосовується не тільки для організації таблиць ідентифікаторів в компіляторах. Даний метод знайшов своє застосування і в операційних системах, і в системах управління базами даних [5,6,11].

Вимоги до виконання роботи

Порядок виконання роботи

У всіх варіантах завдання потрібно розробити програму, яка може забезпечити порівняння двох способів організації таблиці ідентифікаторів за допомогою хеш-адресації. Для порівняння пропонуються способи, засновані на використанні рехеширования або комбінованих методів. Програма повинна прочитувати ідентифікатори з вхідного файлу, розміщувати їх в таблицях за допомогою заданих методів і виконувати пошук вказаних ідентифікаторів на вимогу користувача. В процесі розміщення і пошуку ідентифікаторів в таблицях програма повинна підраховувати середнє число виконаних операцій порівняння для зіставлення ефективності використовуваних методів. Для організації таблиць пропонується використовувати просту хеш-функцію, яку розробник

програми повинен вибрати самостійно. Хеш-функція повинна забезпечувати роботу не менше ніж з 200 ідентифікаторами, допустима довжина ідентифікатора повинна бути не меншого 32 символів. Забороняється використовувати в роботі хеш-функції, узяті з прикладу виконання роботи.

Лабораторна робота повинна виконуватися в наступному порядку:

1. Одержати варіант завдання у викладача.
2. Вибрати і описати хеш-функцію.
3. Описати структури даних, використовувані для заданих методів організації таблиць ідентифікаторів.
4. Підготувати і захистити звіт.
5. Написати і відладити програму на ЕОМ.
6. Здати працюючу програму викладачеві.

Вимоги до оформлення звіту

Звіт по лабораторній роботі повинен містити наступні розділи:

Об завдання по лабораторній роботі;

1) опис вибраної хеш-функції;

2) схеми організації таблиць ідентифікаторів (відповідно до варіанту завдання);

3) опис алгоритмів пошуку в таблицях ідентифікаторів (відповідно до варіанту завдання);

4) текст програми (оформляється після виконання програми на ЕОМ**);**

5) результати обробки заданого набору ідентифікаторів (вхідного файлу) за допомогою методів організації таблиць ідентифікаторів, вказаних у варіанті завдання;

6) аналіз ефективності використовуваних методів організації таблиць ідентифікаторів і висновки по виконаній роботі.

Основні контрольні питання

- Що таке таблиця символів і для чого вона призначена? Яка інформація може зберігатися в таблиці символів?
- Які цілі переслідуються при організації таблиці символів?
- Якими характеристиками можуть володіти лексичні елементи початкової програми? Які характеристики є обов'язковими?
- Які існують способи організації таблиць символів?
- В чому полягає алгоритм логарифмічного пошуку? Які переваги він дає в порівнянні з простим перебором і які він має недоліки?
- Розкажіть про деревовидну організацію таблиць ідентифікаторів. У чому її переваги і недоліки?
- Що таке хеш-функції і для чого вони використовуються? У чому суть хеш-адресації?
- Що таке колізія? Чому вона відбувається? Чи можна повністю уникнути колізії?
- Що таке рехешування? Які методи рехешування існують?
- Розкажіть про переваги і недоліки організації таблиць ідентифікаторів за допомогою хеш-адресації і рехешування.
- В чому полягає метод ланцюжків?
- Розкажіть про переваги і недоліки організації таблиць ідентифікаторів за допомогою хеш-адресації і методу ланцюжків.
- Як можуть бути скомбіновані різні методи організації хеш-таблиц?
- Розкажіть про переваги і недоліки організації таблиць ідентифікаторів за допомогою комбінованих методів.

Варіанти завдань

У табл. 1.1 перераховані методи організації таблиць ідентифікаторів, використовувані в завданнях.

Таблиця 1.1. Методи організації таблиць ідентифікаторів

№ методу	Спосіб вирішення колізій
1	Просте рехешування
2	Рехешування з використанням псевдовипадкових чисел
3	Рехешування за допомогою твору
4	Метод ланцюжків
5	Простий список
6	Впорядкований список
7	Бінарне дерево

У табл. 1.2 дані варіанти завдань на основі методів організації таблиць ідентифікаторів, перерахованих в табл. 1.1.

Таблиця 1-2- Варіантів завдань

№ варіанту	Перший метод організації таблиці	Другий метод організації таблиці
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

Приклад виконання роботи

Завдання для прикладу

Як приклад виконання лабораторної роботи візьмемо зіставлення двох методів: хеш-адресації з рехешуванням на основі псевдовипадкових чисел і комбінації хеш-адресації з бінарним деревом. Якщо звернутися до приведеної вище за табли. 1.1. те такий варіант завдання відповідатиме комбінації методів 2 і 7 (у табл. 1.2 серед варіантів завдань така комбінація відсутня).

Вибір і опис хеш-функції

Для хеш-адресації з рехешуванням як хеш-функція візьмемо функцію, яка одержуватиме на вході рядок, а в результаті видавати суму кодів першого, середнього і останнього елементів рядка. Причому якщо рядок містить менше трьох символів, то один і той же символ буде взятий і за перший, і як середній, і як останній. Вважатимемо, що прописні і рядкові букви в ідентифікаторах різні. Як коди символів візьмемо коди таблиці ASCII, яка використовується в обчислювальних системах на базі ОС типу Microsoft Windows Тоді, якщо покласти, що рядок з області визначення хеш-функції містить тільки цифри і букви англійського алфавіту, то мінімальним значенням хеш-функції буде сума трьох кодів цифри «0», а максимальним значенням —сумма трьох кодів літери «z».

Таким чином, область значень вибраної хеш-функції в термінах мови Object Pascal може бути описана як:

(Ord(‘0’)+Ord(‘0’)+Ord(‘0’)) .(Ord(‘z’)+Ord(‘z’)+Ord(‘z’))

Діапазон області значень складає 223 елементи, що задовольняє вимогам завдання (не меншого 200 елементів). Довжина вхідних ідентифікаторів в даному випадку нічим не обмежена. Для зручності користування опишемо дві константи, задаючі межі області значень хеш-функції;

HASH_MIN= Ord(‘0’)+Ord(‘0’)+Ord(‘0’).

HASH_MAX= Ord(‘z’)+Ord(‘z’)+Ord(‘z’).

Сама хеш-функція без урахування рехеширования обчислюватиме

наступний вираз:

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

тут sName - це вхідний рядок (аргумент хеш-функції).

Для рехешування візьмемо простий генератор послідовності псевдовипадкових чисел, побудований на основі формули F=i•H1 mod H2, де H1 і H2— прості числа, вибрані так, щоб H1 було в діапазоні від H2/2 до H2. Причому, щоб цей генератор видавав максимально довгу послідовність у всьому діапазоні від HASH_MIN до HASH_MAX, H2 повинне бути максимальне близько до величини HASH_MAX-HASH_MIN + 1. В даному випадку діапазон містить 223 елементи, і оскільки 223 - просте число, то візьмемо H2 = 223 (якби розмір діапазону не був простим числом, то як H2 потрібно було б узяти найближче до нього менше просте число). Як H1 візьмемо 127: H1-=127. Опишемо відповідні константи:

REHASH=127;

REHASH=223;

Тоді хеш-функція з урахуванням рехеширования матиме наступний вигляд:

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

Begin

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

Вхідні параметри цієї функції: sName — ім'я хешируемого ідентифікатора. iNum — індекс рехешуванняя (якщо iNum=0, то рех’ширование відсутній). Рядок перевірки величини результату (Result < HASH_MIN) доданий, щоб виключити помилки в тих випадках, коли на вхід функції подається рядок, що містить символи поза діапазоном '0'..'z’ (оскільки контроль вхідних ідентифікаторів відсутній, це має сенс).

Для комбінації хеш-адресації і бінарного дерева можна використовувати простішу хеш-функцію — суму кодів першого і середнього символів вхідного рядка. Діапазон значень такої хеш-функції в термінах мови Object Pascal виглядатиме так:

(Ord(‘0’)+Ord(‘0’))..(Ord(‘z’)+Ord(‘z’))

Цей діапазон містить менше 200 елементів, проте функція задовольнятиме вимогам завдання, оскільки в комбінації з бінарним деревом вона забезпечуватиме обробку необмеженої кількості ідентифікаторів (максимальна кількість ідентифікаторів буде обмежена тільки об'ємом доступної оперативної пам'яті комп'ютера).

Без застосування рехешнрования ця хеш-функція виглядатиме значно простіше, ніж описана вище хеш-функція з урахуванням рехешнрования:

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

Опис структур даних таблиць ідентифікаторів

В першу чергу необхідно описати структуру даних, яка буде використана для зберігання інформації про ідентифікатори в таблицях ідентифікаторів. Для обох таблиць (з рехешуванням на основі генератора псевдовипадкових чисел і в комбінації з бінарним деревом) будемо використовувати одну і ту ж структуру. В цьому випадку в таблицях будуть зберігатися невикористовувані дані, але програмний код буде простішим. В якості навчального прикладу такий підхід виправданий.

Структура даних таблиці ідентифікаторів (назвемо її TVarInfo) повинна містити в обов'язковому порядку поле імені ідентифікатора (поле sName: string), а також поля додаткової інформації про ідентифікатор на розсуд розробників компілятора. У лабораторній роботі не передбачено зберігання якої-небудь

додаткової інформації про ідентифікатори, тому як ілюстрація інформаційного поля включимо в структуру TVarInfo додаткову інформаційну структуру TAddVarInfo (поле pInfo: TAddVarInfo). Оскільки в мові Object Pascal для полів і змінних, описаних як class, зберігаються тільки посилання на відповідну структуру, такий підхід не приведе до значних витрат пам'яті, але дозволить в майбутньому зберігати будь-яку інформацію, пов'язану з ідентифікатором, в окремій структурі даних (оскільки передбачається використовувати створювані програмні модулі в подальших лабораторних роботах). В даному випадку інший підхід неможливий, оскільки наперед не відомо, які дані необхідно буде зберігати в таблицях ідентифікаторів. Але розробник реального компілятора, як правило, знає, яку інформацію потрібно зберігати, і може використовувати інший підхід — безпосередньо включити всі необхідні поля в структуру даних таблиці ідентифікаторів (в даному випадку — в структуру TVarInfo) без використання проміжних структур даних і посилань.

Перший підхід, реалізований в даному прикладі, забезпечує економніше використання оперативної пам'яті, але є складнішим і вимагає роботи з динамічними структурами, другий підхід простіший в реалізації, але менш економно використовує пам'ять. Який з двох підходів вибрати, вирішує розробник компілятора у кожному конкретному випадку (другий підхід буде проілюстрований пізніше в прикладі до лабораторної роботи №4). Для роботи із структурою даних TVarInfo будуть потрібні наступні функції:

- функція створення структури даних і звільнення займаної пам'яті — реалізовані як constructor Create і destructor Destroy;

- функції доступу до додаткової інформації - в даній реалізації це procedure SetInfo і procedure ClearInfo.

Ці функції будуть загальними для таблиці ідентифікаторів з рехешуванням і для комбінованої таблиці ідентифікаторів.

Проте для комбінованої таблиці ідентифікаторів в структуру даних TVarInfo потрібно буде також включити додаткові поля даних і функції, що забезпечують організацію бінарного дерева:

- посилання на ліву («меншу») і праву («велику») гілку дерева — реалізовані як поля даних minEl, maxE1: TVarInfo;

- функцію додавання елементу в дерево - function AddElCnt і function AddElem;

- функцію пошуку елементу в дереві — function FindElCnt і function FindElem;

- функція очищення інформаційних полів у всьому дереві — procedure ClearAllInfo;

- функція виведення вмісту бінарного дерева в один рядок (для отримання списку всіх ідентифікаторів) — function GetElList.

Функції пошуку і розміщення елементу в дереві реалізовані в двох екземплярах, оскільки одна з них виконує підрахунок кількості порівнянь, а інша - ні.

Оскільки на функції і процедури не витрачається оперативна пам'ять, в результаті вийшло, що при використанні однієї і тієї ж структури даних для різних таблиць ідентифікаторів в таблиці з рехешуванням витрачатиметься невживана пам'ять тільки на зберігання двох зайвих посилань (minE1 і maxEl).

Повністю вся структура даних TVarInfo і пов'язані з нею процедури і функції описані в програмному модулі TblElem. Повний текст цього програмного модуля приведений в лістингу П3.1 в додатку 3.

Треба звернути увагу на один важливий момент в реалізації функції пошуку ідентифікатора в дереві (function TVarInfo.FindElCnt). Якщо виконувати порівняння двох рядків (в даному випадку - імені шуканого ідентифікатора sN і імені ідентифікатора в поточному вузлі дерева sName) за допомогою стандартних методів порівняння рядків мови Object Pascal, то фрагмент програмного коду виглядав би приблизно так:

```
If  sN<sName then begin
…
end
```

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

У цьому фрагменті порівняння рядків виконується двічі: спочатку перевіряється відношення «менше» (sN < sName), а потім — «більше» (sN > sName). І хоча в програмному коді явно це не вказано, для кожного з цих операторів буде викликана бібліотечна функція порівняння рядків (тобто операція порівняння може виконатися двічі!). Щоб цього уникнути, в реалізації запропонованою в прикладі, виконується явний виклик функції порівняння рядків, а потім обробляється отриманий результат:

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

У такому варіанті двічі може бути виконано тільки порівняння цілого числа з нулем, а порівняння рядків завжди виконується тільки один раз, що істотно збільшує ефективність процедури пошуку.

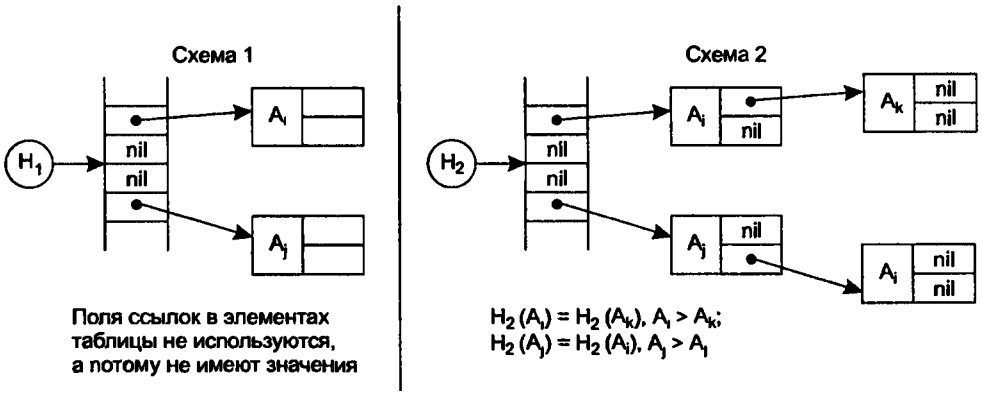
Організація таблиць ідентифікаторів

Таблиці ідентифікаторів реалізовані у вигляді статичних масивів

розміром HASH_MIN.. HASH_MAX, елементами яких є структури даних типу TVarInfo. У мові Object Pascal, як було сказано вище, для структур таких типів зберігаються посилання. Тому для позначення порожніх чарунок в таблицях ідентифікаторів використовуватиметься порожнє посилання — nil.

Оскільки в пам'яті зберігаються посилання, описані масиви гратимуть роль хеш-таблиць, посилання з яких указують безпосередньо на інформацію в таблицях ідентифікаторів.

На мал. 1.3 показані умовні схеми, що наочно ілюструють організацію таблиць ідентифікаторів. Схема 1 ілюструє таблицю ідентифікаторів з рехешуванням на основі генератора псевдовипадкових чисел, схема 2 - таблицю ідентифікаторів на основі комбінації хеш-адресації з бінарним деревом. Чарунки з написом «nil» відповідають незаповненим чарункам хеш-таблиці.



I

H1 і H2 - відповідні хеш-функції

Мал. 1.3. Схеми організації таблиць ідентифікаторів

Для кожної таблиці ідентифікаторів реалізовані наступні функції:

- функцію початкової ініціалізації хеш-таблиці — InitTreeVar і InitHashVar;

- функції звільнення пам'яті хеш-таблиці - ClearTreeVar і ClearHashVar;

- функцію видалення додаткової інформації в таблиці — ClearTreeInfo і ClearHashInfo;

- функції додавання елементу в таблицю ідентифікаторів — AddTreeVar і AddHashVar;

- функції пошуку елементу в таблиці ідентифікаторів — GetTreeVar і GetHashVar;

- функції, що повертають кількість виконаних операцій порівняння при розміщенні або пошуку елементу в таблиці — GetTreeCount і GetHashCount.

Алгоритми пошуку і розміщення ідентифікаторів для двох даних методів організації таблиць були описані вище в розділі «Короткі теоретичні відомості», тому приводити їх тут повторно немає сенсу. Вони реалізовані у вигляді чотирьох перерахованих вище функціями (AddTreeVar і AddHashVar - для розміщення елементу; GetTreeVar і GetHashVar - для пошуку елементу). Функції пошуку і розміщення елементів в таблиці як результат повертають посилання на елемент таблиці (структура якого описана в модулі TblElem) у разі успішного виконання і нульове посилання - інакше.

Треба відзначити, що функції розміщення ідентифікатора в таблиці організовані таким чином, що якщо на момент додавання нового ідентифікатора в таблиці вже є ідентифікатор з таким же ім'ям, то функція не додає новий ідентифікатор в таблицю, а повертає як результат посилання на раніше поміщений в таблицю ідентифікатор. Таким чином, в таблиці не може бути два і більше ідентифікатора з однаковим ім'ям. При цьому наявність однакових ідентифікаторів у вхідному файлі не сприймається як помилка — це допустимо, оскільки в завданні не передбачено обмеження на наявність співпадаючих імен ідентифікаторів.

Всі перераховані функції описані в двох програмних модулях: FncHash - для таблиці ідентифікаторів, побудованої на основі рехешування з використанням генератора псевдовипадкових чисел, і FncTree — для таблиці ідентифікаторів, побудованої на основі комбінації хеш-адресації і бінарного дерева. Окрім масивів даних для організації таблиць ідентифікаторів і функцій роботи з ними ці модулі містять також опис змінних, які використовуються для підрахунку кількості виконаних операцій порівняння при розміщенні і пошуку

ідентифікатора в таблицях.

Повні тексти обох модулів (FncHash і FncTree) можна знайти на веб-сайті видавництва, у файлах FncHash.pas і FncTree.pas. Крім того, текст модулі FncTree приведений в лістингу П3.2 в додатку 3.

Хочеться звернути увагу на те, що в розділах ініціалізації (initialisation) обох модулів викликається функція початкового заповнення таблиці ідентифікаторів, а в розділах завершення (flna1ization) обох модулів - функція звільнення пам'яті. Це гарантує коректну роботу модулів при будь-якому порядку виклику решти функцій, оскільки Object Pascal сам забезпечує своєчасний виклик програмного коду в розділах ініціалізації і завершення модулів.

Текст програми

Окрім перерахованих вище модулів необхідний ще модуль, що забезпечує інтерфейс з користувачем. Цей модуль (FormLab1) реалізує графічне вікно TLab1Form на основі класу TForm бібліотеки VCL. Він забезпечує інтерфейс

- функції пошуку елементу в таблиці ідентифікаторів - GetTreeVar і GetHashVar;
- функції, що повертають кількість виконаних операцій порівняння при розміщенні або пошуку елементу в таблиці - GetTreeCount і GetHashCount.

Алгоритми пошуку і розміщення ідентифікаторів для двох даних методів організації таблиць були описані вище в розділі «Короткі теоретичні відомості», тому приводити їх тут ще раз немає сенсу. Вони реалізовані у вигляді чотирьох перерахованих вище функцій (AddTreeVar і AddHashVar - для розміщення елементу; GetTreeVar і GetHashVar - для пошуку елементу). Функції пошуку і розміщення елементів в таблиці як результат повертають посилання на елемент таблиці (структура якого описана в модулі TblElem) у разі успішного виконання і нульове посилання – у протилежному випадку.

Треба відзначити, що функції розміщення ідентифікатора в таблиці організовані таким чином, що якщо на момент внесення нового ідентифікатора в таблиці вже є ідентифікатор з таким же ім'ям, то функція не додає новий ідентифікатор в таблицю, а повертає як результат посилання на раніше внесений в таблицю ідентифікатор. Таким чином, в таблиці не може бути двох і більш ідентифікаторів з однаковим ім'ям. При цьому наявність однакових ідентифікаторів у вхідному файлі не сприймається як помилка - це допустимо, оскільки в завданні не передбачено обмеження на наявність співпадаючих імен ідентифікаторів.

Всі перераховані функції описані в двох програмних модулях: FncHash - для таблиці ідентифікаторів, побудованої на основі рехешування з використанням генератора псевдовипадкових чисел, і FncTree - для таблиці ідентифікаторів, побудованої на основі комбінацій хеш-адресації і бінарного дерева. Окрім масивів даних для організації таблиць ідентифікаторів і функцій роботи з ними ці модулі містять також опис змінних, використовуваних для підрахунку кількості виконаних операцій порівняння при розміщенні і пошуку ідентифікатора в таблицях.

Повні тексти обох модулів (FncHash і FncTree) можна знайти на веб-сайті видавництва, у файлах FncHash.pas і FncTree.pas. крім того, текст модуля FncTree приведений в лістингу п3.2 в додатку 3.

Хочеться звернути увагу на те, що в розділах ініціалізації (initia1ization) обох модулів викликається функція початкового заповнення таблиці ідентифікаторів, а в розділах завершення (finalization) обоих модулів - функція звільнення пам'яті. Це гарантує коректну роботу модулів при будь-якому порядку виклику решти функцій, оскільки ObjectPascal сам забезпечує своєчасний виклик програмного коду в розділах ініціалізації і завершення модулів.

Текст програми

Окрім перерахованих вище модулів необхідно ще модуль, що забезпечує інтерфейс з користувачем. Цей модуль (FormLabl) реалізує графічне вікно

TlablForm на базі класу TForm бібліотеки VCL. він забезпечує інтерфейс засобами Graphical User Interface (GUI) в ОС типу Windows на основі стандартних органів керування з системних бібліотек даної ОС. Окрім програмного коду (файл FormLabl.pas) модуль включає опис ресурсів призначеного для користувача інтерфейсу (файл FotLabl.dfll). Детальніше принципи організації призначеного для користувача інтерфейсу на основі GUI і роботи систем програмування з ресурсами інтерфейсу описані [3 5. 6 7].

Окрім опису інтерфейсної форми і її засобів управління модуль FotLabl містить три змінні (iCountNum, iCountHash, iCountTree), що служать для накопичення статистичних результатів у міру виконання розміщення і пошуку ідентифікаторів в таблицях, а також функцію (Procedure ViewStatistic) для відображення накопиченої статистичної інформації на екрані.

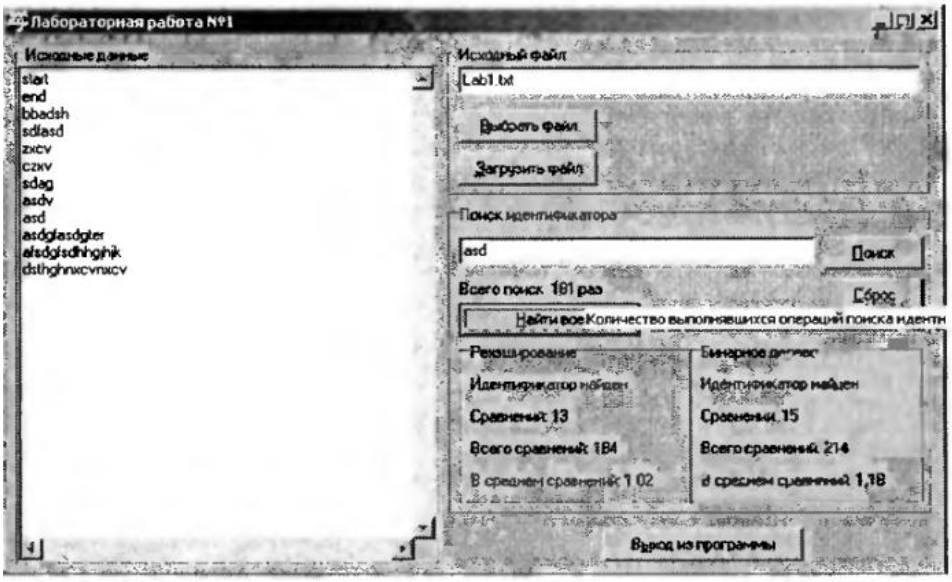
Інтерфейсна форма описана в модулі, містить наступні основні органи управління:

- поле введення імені файлу (EditFi1e), кнопка вибору імені файлу з каталогів файлової системи (BtnFile), кнопка читання файлу (BtnLoad);
- багаторядкове поле для відображення прочитаного файлу (ListIdents);
- поле введення імені ідентифікатора, який треба знайти (EditSearch);
- кнопка для пошуку введеного ідентифікатора (BtnSearch) - цією кнопкою одноразово викликається процедура пошуку (procedure SearchStr);
- кнопка автоматичного пошуку всіх ідентифікаторів (BtnA.11Search) - цією кнопкою процедура пошуку ідентифікатора (procedure SearchStr) викликається циклічно для всіх зчитаних з файлу ідентифікаторів (для всіх, перерахованих в полі ListIdents);
- кнопка скидання накопиченої статистичної інформації (BtnReset);
- поля для відображення статистичної інформації;
- кнопка завершення роботи з програмою (BtnExit).

Зовнішній вигляд такої форми приведений на мал. 1.4

Функція читання вмісту файлу з ідентифікаторами (procedure TLablForm.BtnLoadClick) викликається клацанням по кнопці BtnLoad. Вона організована таким чином, що спочатку вміст файлу читається в багаторядкове поле ListIdents, а потім всі прочитані ідентифікатори записуються в дві таблиці ідентифікаторів Кожен рядок файлу вважається окремим ідентифікатором пробілів початку і в кінці рядка ігноруються. При помилці розміщення ідентифікатора в одній з таблиць видається попередження (наприклад, якщо буде зчитано більше 223 різних ідентифікаторів, то рехешування стане неможливим і буде видане повідомлення про помилку).

Функція пошуку ідентифікатора (procedure TlablForm.SearchStr) викликається одноразово клацанням по кнопці BtnSearch (процедура procedue TlablForm.BtnSearchClick) або багато разів клацанням по кнопці BtnA11Search (процедура procedure TlablForm.SearchStr. BtnA11SearchClick). Пошук йде відразу в двох таблицях, результати пошуку і накопичена статистична інформація відображаються у відповідних полях.



Мал. 1.4. Зовнішній вигляд інтерфейсної форми для лабораторної роботи №1

Повний текст програмного коду модуля інтерфейсу з користувачем і опис ресурсів призначеного для користувача інтерфейсу знаходяться в архіві,

розташованому на веб-сайті видавництва, у файлах FormLab1.pas і FormLab1.dfm відповідно.

Повний текст всіх програмних модулів, що реалізовують розглянутий наприклад для лабораторної роботи È1, можна знайти в архіві, розташованому на веб-сайті, в підкаталогах LABS і COMMON (у підкаталог COMMON винесені ті програмні модулі, початковий текст яких не залежить від вхідної мови і завдання по лабораторній роботі). головним м проекту є файn LAB1.PDR у підкаталозі LABS. Крім того, текст модуля FncTree приведений в листиигу п3.1 в додатку 3.

Висновки по виконаній роботі

Врезультаті виконання написаного програмного коду для ряду текстових файлів було встановлено, що при заповненні таблиці ідентифікаторів до 20% (до 45 ідентифікаторів) для пошуку і розміщення ідентифікатора з використанням рехешированія на основі генератора псевдовипадкових чисел в середньому потрібне менше число порівнянь, чим при використанні хеш-адресації в комбінації з бінарним деревом. При заповненні таблиці від 20% те 40% (приблизно 45-90 ідентифікаторів) обидва методи мають приблизно однакові показники, але при заповненні таблиці більш, чим на 40% (90-223 ідентифікаторів), ефективність комбінованого методу в порівнянні з методом рехешированія різко зростає. Якщо на вході є більше 223 ідентифікаторів, рехешування повністю перестане працювати.

Таким чином, встановлено, що комбинированный метод працездатний навіть за наявності простої хеш-функції і дає непогані результати (в середньому 3-5 порівнянь на вхідних файлах, 500-700 ідентифікаторів, що містять), тоді як метод на основі рехеширонанія для реальної роботи вимагає складнішої хеш-функції з діапазоном значень в декілька тисяч або десятків тисяч.