**Лабораторна робота № 4.**

***Тема:* “Алгоритми пошуку. Хешування”.**

***Мета:*** Розлянути алгоритми пошуку, дослідити доцільність застосування різних алгоритмів в конкретних випадках виходячи з їх ефективності. Розглянути поняття хеш-функції, суть та цілі процесу хешування.

***Теоретичні відомості:***

Кажучи про пошук, ми матимемо на увазі якийсь масив даних, а шукати будемо певний елемент в цьому масиві. Оптимальність пошуку для простоти визначимо дуже конкретно - це швидкість роботи алгоритму.

**Загальна класифікація алгоритмів пошуку**

* Всі методи можна розділити на **статичні і динамічні**. При статичному пошуку масив значень не міняється під час роботи алгоритму. Під час динамічного пошуку масив може перебудовуватися або змінювати розмірність.

Спробуємо вирішити просте завдання - знайти потрібне значення в масиві.

Якщо немає ніякої додаткової інформації про розшукувані дані, то очевидний підхід простий послідовний перегляд масиву із збільшенням крок за кроком тієї його частини, де бажаного елементу не виявлено. Такий метод називається лінійним пошуком. Умови закінчення пошуку такі:

Елемент знайдений, тобто аi = x.

Весь масив проглянутий і збігу не виявлено.

Це дає нам лінійний алгоритм:

*Алгоритм 1.*

*i:=0;*

*while (i<N) and (а[i]<>х) do i:=i+1*

Слід звернути увагу, що якщо елемент знайдений, то він знайдений разом з мінімально можливим індексом, тобто це перший з таких елементів. Рівність i=N свідчить, що збігу не існує.

Очевидно, що закінчення циклу гарантоване, оскільки на кожному кроці значення i збільшується, і, отже, воно досягне за кінцеве число кроків межі N; фактично ж, якщо збігу не було, це відбудеться після N кроків.

На кожному кроці алгоритму здійснюється збільшення індексу і обчислення логічного виразу. Можна спростити крок алгоритму, якщо спростити логічний вираз, який складається з двох членів. Це спрощення здійснюється шляхом формулювання логічного виразу з одного члена, але при цьому необхідно гарантувати, що збіг відбудеться завжди. Для цього достатньо в кінець масиву помістити додатковий елемент із значенням x. Такий допоміжний елемент називається барьером|. Тепер масив буде описаний так:

*а: array[0..N] of integer*

*і алгоритм лінійного пошуку з бар'єром виглядає таким чином:*

*Алгоритм 1-.*

*а[N]:=x; i:=0;*

*while а[i]<>x do i:=i+1*

Ясно, що рівність i=N свідчить про те, що збігу (якщо не рахувати збігу з бар'єром) не було.

**1. Алгоритм пошуку по бінарному дереву.**

Суть цього алгоритму достатньо проста. Уявимо собі, що у нас є набір карток з телефонними номерами і адресами людей. Картки відсортовані у порядку зростання телефонних номерів. Необхідно знайти адресу людини, якщо ми знаємо, що її номер телефону 222-22-22. Наші дії? Беремо картку з середини пачки, номер картки 444-44-44. Порівнюючи його з шуканим номером, ми бачимо, що наш менше і, отже, знаходиться точно в першій половині пачки. Сміливо відкладаємо другу частину пачки убік, вона нам не потрібна, масив пошуку ми звузили рівно в два рази. Тепер беремо картку з середини пачки, що залишилася, на ній номер 123-45-67. З чого виходить, що потрібна нам картка лежить в другій половині пачки, що залишилася... Таким чином, при кожному порівнянні, ми зменшуємо зону пошуку в два рази. Звідси і назва методу - половинного ділення або дихотомії.

Швидкість збіжності цього алгоритму пропорційна [Log(2)N](http://www.delphikingdom.com/article/alg_search.htm#ref). Це означає буквально те, що не більше, ніж через Log(2) N порівнянь, ми або знайдемо потрібне значення, або переконаємося в його відсутності.

Інша назва цього алгоритму – «метод бінарного дерева» походить з представлення "шляху" пошуку у вигляді дерева (у якого кожна наступна гілка розділяється на дві, по одній з яких ми і рухаємося надалі)..

Спосіб дуже поширений у наш час, можливо унаслідок його ефективності укупі з простотою програмування цього алгоритму. Саме бінарний пошук використовується при пошуку в індексах таблиць.

У цьому алгоритмі використовуються дві індексні змінні L і R, які відзначають відповідно лівий і правий кінець секції масиву а, де ще може бути виявлений необхідний елемент.

*Алгоритм 2.*

*L:=0; R:=N-1; Found:=false;*

*while(L<=R) and not Found do begin*

*m:=(L+R) div 2;*

*if а[m]=x then begin*

*Found:=true*

*end else begin*

*if а[m]<x then L:=m+1 else R:=m-1*

*end*

*end;*

Максимальне число порівнянь для цього алгоритму рівне log2n, закругленому до найближчого цілого. Таким чином, приведений алгоритм істотно виграє в порівнянні з лінійним пошуком, адже там очікуване число порівнянь v N/2.

Ефективність дещо поліпшується, якщо поміняти місцями заголовки умовних операторів. Перевірку на рівність можна виконувати в другу чергу, оскільки вона зустрічається лише одного разу і приводить до закінчення роботи. Але істотніший виграш дасть відмову від закінчення пошуку при фіксації збігу. На перший погляд це здається дивним, проте, при уважному розгляді виявляється, що виграш в ефективності на кожному кроці перевершує втрати від порівняння з декількома додатковими елементами (число кроків у гіршому разі рівне logN).

*Алгоритм 2-.*

*L:=0; R:=N;*

*while L<R do begin*

*m:=(L+R) div 2;*

*if а[m]<x then L:=m+1 else R:=m*

*end*

Закінчення циклу гарантоване. Це пояснюється наступним. На початку кожного кроку L<R. Для середнього арифметичного m справедливо умова L face="Symbol" =< m < R. Отже, різниця L-R дійсно убуває, адже або L збільшується при привласненні йому значення m+1, або R зменшується при привласненні значення m. При L face="Symbol" =< R повторення циклу закінчується.

Виконання умови L=R ще не свідчить про знаходження необхідного елементу. Тут потрібна додаткова перевірка. Також, необхідно враховувати, що елемент а[R] в порівняннях ніколи не бере участь. Отже, і тут необхідна додаткова перевірка на рівність а[R]=x. Слід зазначити, що ці перевірки виконуються одноразово.

Приведений алгоритм, як і у разі лінійного пошуку, знаходить співпадаючий елемент з найменшим індексом.

Ті, хто має нещастя часто працювати з текстовими редакторами, знають ціну функції знаходження потрібних слів в тексті, що істотно полегшує редагування документів і пошук потрібної інформації. Пани програмісти теж це знають, благо аж до самої компіляції програми їм доводиться працювати в текстовому редакторі.

Отже, хай у нас є текст, що складається з n символів, який надалі домовимося називати T, а T[i] його i-й символ. Рядок або просто слово, що складається з m символів, назвемо S, де S[i] -i-й символ рядка. Нам потрібно перевірити, чи входить даний рядок в даний текст, і якщо входить, то починаючи з якого символу тексту. розглянемо декілька відомих алгоритмів, вирішуючих поставлене завдання.

**Простий алгоритм**

Суть методу, про який піде мова нижче, полягає в наступному: ми перевіряємо, чи співпадають m символів тексту (починаючи з вибраного) з символами нашого рядка, намагаючись приміряти шаблон куди тільки можливо. Природно, реалізувати описаний алгоритм найпростіше

*Program SimpleSearch;*

*Var T : array[1..40000] of char; { виконує роль тексту}*

*S : array[1..10000] of char; { виконує роль рядка; як і текст, може бути достатньо велика}*

*i,j : longint;*

*m,n : longint;*

*Begin*

*{ Введення тексту і зразка}*

*for i:=1 to n-m+1 do*

*begin*

*j:=0;*

*while (j<m) and (S[j+1]=T[i+j]) do { По черзі порівнюємо всі символи починаючи з i-ого}*

*j:=j+1;*

*if j=m then { якщо всі символи співпадали}*

*writeln('Зразок входить в текст починаючи з ',i,'-ого символу'); { повідомлення про знаходження рядка в тексті}*

*end;*

*End.*

Просто і елегантно, ніби так і треба. Дійсно, це нескладно у виконанні, але і не дуже ефективно на практиці. Давайте оцінимо швидкість роботи цієї програмки. У ній присутні два цикли (один вкладений), час роботи зовнішнього більшою мірою залежить від n, а внутрішній у гіршому разі робить m операцій. Таким чином, час роботи всього алгоритму є O((n-m+1)m). Для маленьких рядків пошук пропрацює швидко, але якщо в якомусь многомегабайтном файлі ви шукатимете послідовність довгої 100 Кб, то, боюся, доведеться вам чекати ну дуже довго.

Наступний метод працює набагато швидше простого, але, на жаль, накладає деякі обмеження на текст і шуканий рядок.

**Алгоритм Рабіна-Карпа.**

Ідея, запропонована Рабіном і Карпом, має на увазі поставити у відповідність кожному рядку деяке унікальне число, і замість того щоб порівнювати самі рядки, порівнювати числа, що набагато швидше. Проблема в тому, що шуканий рядок може бути довгим, рядків в тексті теж вистачає. А оскільки кожному рядку потрібно зіставити унікальне число, то і чисел повинно бути багато, а отже, числа будуть великими (порядку Dm, де D - кількість різних символів), і працювати з ними буде так само незручно.

***Реалізація для тексту, що складається тільки з цифр, і рядка завдовжки до 8 символів.***

*Program RabinKarpSearch;*

*Var T : array[1..40000] of 0..9;*

*S : array[1..8] of 0..9;*

*i,j : longint;*

*n,m : longint;*

*v,w : longint; {v - число, що характеризує шуканий рядок, w характеризує рядок довгі m в тексті}*

*к : longint;*

*const D : longint = 10; { кількість різних символів (10 різних цифр)}*

*Begin*

*{ Введення тексту і зразка}*

*v:=0;*

*w:=0;*

*for i:=1 to m do*

*begin*

*v:=v\*D+S[i]; { обчислення v, рядок представляється як число}*

*w:=w\*D+T[i]; { обчислення початкового значення w}*

*end;*

*k:=1;*

*for i:=1 to m-1 do {к потрібне для багатократного обчислення w і має значення Dm-1}*

*k:=k\*D;*

*for i:=m+1 to n+1 do*

*begin*

*if w=v then { якщо числа рівні, то рядки співпадають, а значить, зразок знайдений в тексті}*

*writeln('Зразок входить в текст починаючи з ',i-m,'-ого символу');*

*if i<=n then*

*w:=d\*(w-k\*T[i-m])+T[i]; {обчислення нового значення w}*

*end;*

*End.*

Цей алгоритм виконує лінійний прохід по рядку (m кроків) і лінійний прохід по всьому тексту (n кроків), отже, загальний час роботи є O(n+m). Цей час лінійно залежить від розміру рядка і тексту, отже програма працює швидко. Але який інтерес працювати тільки з короткими рядками і цифрами? Розробники алгоритму придумали, як поліпшити цей алгоритм без особливих втрат в швидкості роботи. Як ви відмітили, ми ставили у відповідність рядку її числове уявлення, але виникала проблема великих чисел. Її можна уникнути, якщо проводити всі арифметичні дії по модулю якогось простого числа (постійно брати залишок від ділення на це число). Таким чином, знаходиться не саме число, що характеризують рядок, а його залишок від ділення на якесь просте число. Тепер ми ставимо число у відповідність не одному рядку, а цілому класу, але оскільки класів буде досить багато (стільки, скільки різних залишків від ділення на це просте число), то додаткову перевірку доведеться проводити рідко.

*Var T : array[1..40000] of char;*

*S : array[1..10000] of char;*

*i,j : longint;*

*n,m : longint;*

*v,w : longint;*

*до : longint;*

*const P : longint = 7919; {1000-е просте число}*

*D : longint = 256; { кількість різних символів (кількість всіх можливих значень символьного типа char)}*

*Begin*

*{ Введення тексту і зразка}*

*v:=0;*

*w:=0;*

*for i:=1 to m do { обчислення v і w}*

*begin*

*v:=(v\*D+ord(S[i])) mod P; {ord перетворить символ в число}*

*w:=(w\*D+ord(T[i])) mod P;*

*end;*

*k:=1;*

*for i:=1 to m-1 do*

*k:=k\*D mod P; {до має значення Dm-1 mod P}*

*for i:=m+1 to n+1 do*

*begin*

*if w=v then { якщо числа рівні, то рядки належать одному класу, і треба перевірити чи співпадають вони}*

*begin*

*j:=0;*

*while (j<m) and (S[j+1]=T[i-m+j]) do*

*j:=j+1;*

*if j=m then { остаточна перевірка}*

*writeln('Зразок входить в текст починаючи з ',i-m,'-ого символу');*

*end;*

*if i<=n then*

*w:=(d\*(w+P-(ord(T[i-m])\*k mod P))+ord(T[i])) mod P;*

*end;*

*End.*

Отже, нам все-таки доводиться проводити порівняння рядків посимвольно, але оскільки «холостих» спрацьовувань буде небагато (у 1/P випадках), то й очікуваний час роботи малий. Строго кажучи, час роботи є O(m+n+mn/P), mn/P достатньо невелике, так що складність роботи майже лінійна. Зрозуміло, що просте число слід вибирати більшим, чим більше це число, тим швидше працюватиме програма. Цей алгоритм значно швидше за попередній і цілком підходить для роботи з дуже довгими рядками.

Ще один важливий метод - алгоритм Кнута-Морріса-Пратта, один з кращих на нинішній момент, працює за лінійний час для будь-якого тексту і будь-якого рядка.

**Алгоритм Кнута-Морріса-Пратта**

Метод використовує передобробку шуканого рядка, а саме: на її основі створюється так звана *префікс-функція*. Суть цієї функції в знаходженні для кожного підрядка S[1..i] рядка S найбільшого підрядка S[1..j] (j<i), присутнього одночасно і на початку, і в кінці підрядка (як префікс і як суфікс). Наприклад, для підрядка abcHelloabc таким підрядком є abc (одночасно і префіксом, і суфіксом). Сенс функції префікса в тому, що ми можемо відкинути свідомо невірні варіанти, тобто якщо при пошуку співпав підрядок abcHelloabc (наступний символ не співпав), то нам має сенс продовжувати перевірку продовжити пошук вже з четвертого символу (перші три і так співпадуть). От як можна обчислити цю функцію для всіх значень параметра від 1 до m:

*var S : array[1..10000] of char;*

*P : array[1..10000] of word; { масив, в якому зберігаються значення функції префікса}*

*i,k : longint;*

*m : longint;*

*Procedure Prefix; { процедура, що обчислює префікс-функцію}*

*Begin*

*P[1]:=0; { префікс рядка з одного символу має нульову довжину}*

*k:=0;*

*for i:=2 to m do { обчислюється для префіксів рядка довгої від 2 до m символів}*

*begin*

*while (k>0) and (S[k+1]<>S[i]) do*

*k:=P[k]; { значення функції може бути набуте з раніше зроблених обчислень}*

*if S[k+1]=S[i] then*

*k:=k+1;*

*P[i]:=k; { привласнення функції префікса}*

*end;*

*End;*

Тепер розберемося, чому ж дана процедура обчислює префікс-функцію правильно. Використовується наступна ідея: якщо префікс (він же суфікс) рядка довгої i довше за один символ, то він одночасно і префікс підрядка довгої i-1. Таким чином, ми перевіряємо префікс попереднього підрядка, якщо ж той не підходить, то префікс її префікса, і т.д. Діючи так, знаходимо найбільший шуканий префікс. Наступне питання, на яке варто відповісти: чому час роботи процедури лінійно, адже в ній присутній вкладений цикл? Ну, по-перше, привласнення функції префікса відбувається чітко m раз, решта часу міняється змінна до. Оскільки в циклі while вона зменшується (P[k]<k), але не стає менше 0, то зменшуватися вона може не частіше, ніж зростати. Змінна до зростає на 1 не більше m раз. Означає, змінна до міняється всього не більше 2m разу. Виходить, що час роботи всієї процедури є O(m). А зараз ми переходимо до самої програми, що шукає рядок в тексті.

*Program KnutMorrisPrattSearch;*

*{ Опис процедури Prefix і пов'язаних з нею змінних}*

*var n : longint;*

*T : array[1..40000] of char;*

*Begin*

*{ Введення тексту і зразка}*

*Prefix; { Обчислення функції префікса}*

*k:=0; { кількість символів, що співпали на даний момент}*

*for i:=1 to n do*

*begin*

*while (k>0) and (S[k+1]<>T[i]) do*

*k:=P[k];*

*if S[k+1]=T[i] then*

*k:=k+1;*

*if k=m then { якщо співпали всі символи}*

*begin*

*writeln('Зразок входить в текст починаючи з ',i-m+1,'-ого символу');*

*k:=P[k];*

*end;*

*end;*

*End.*

## Довести що ця програма працює за лінійний час, можна точно так, як і для процедури Prefix. Отже, загальний час роботи програми є O(n+m), тобто лінійний час.

Наступна програма демонструє КМП-алгоритм.

*Program KMP;*

*const*

*Mmax = 100; Nmax = 10000;*

*var*

*i, j, до, M, N: integer;*

*p: array[0..Mmax-1] of char; {слово}*

*s: array[0..Mmax-1] of char; {текст}*

*d: array[0..Mmax-1] of integer;*

*begin*

*{ Введення тексту s і слова p}*

*Write('N:'); Readln(N);*

*Write('s:'); Readln(s);*

*Write('M:'); Readln(M);*

*Write('p:'); Readln(p);*

*{ Заповнення масиву d}*

*j:=0; k:=-1; d[0]:=-1;*

*while j<(M-1) do begin*

*while(k>=0) and (p[j]<>p[k]) do k:=d[k];*

*j:=j+1; k:=k+1;*

*if p[j]=p[k] then*

*d[j]:=d[k]*

*else*

*d[j]:=k;*

*end;*

*{ Пошук слова p в тексті s}*

*i:=0; j:=0;*

*while (j<M) and (i<N) do begin*

*while (j>=0) and (s[i]<>p[j]) do j:=d[j]; {Зрушення слова}*

*i:=i+1; j:=j+1;*

*end;*

*{ Виведення результату пошуку}*

*if j=M then Writeln('Yes') {знайдений }*

*else Writeln('No'); {не знайдений}*

*Readln;*

*end.*

**Алгоритм Боуєра і Мура**

БМ-пошук грунтується на незвичайному міркуванні порівняння символів починається з кінця слова, а не з початку. Як і у разі КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Хай для кожного символу x з алфавіту величина dx v відстань від найправішого в слові входження x до правого кінця слова. Уявимо собі, що виявлено розбіжність між словом і текстом. В цьому випадку слово відразу ж можна зрушити управо на dpM-1 позицій, тобто на число позицій, швидше за все більше одиниці. Якщо неспівпадаючий символ тексту в слові взагалі не зустрічається, то зрушення стає навіть більшим, а саме зрушувати можна на довжину всього слова.

Нижче приводиться програма із спрощеною стратегією Боуера-мура, побудована так само, як і попередня програма з КМП-алгоритмом. Зверніть увагу на таку деталь: у внутрішньому циклі використовується цикл з repeat, де перед порівнянням s і p збільшуються значення до і j. Це дозволяє виключити в індексних виразах складову -1.

*Program BM;*

*const*

*Mmax = 100; Nmax = 10000;*

*var*

*i, j, до, M, N: integer;*

*ch: char;*

*p: array[0..Mmax-1] of char; {слово}*

*s: array[0..Nmax-1] of char; {текст}*

*d: array[' '..'z'] of integer;*

*begin*

*{ Введення тексту s і слова p}*

*Write('N:'); Readln(N);*

*Write('s:'); Readln(s);*

*Write('M:'); Readln(M);*

*Write('p:'); Readln(p);*

*{Заповнення масиву d}*

*for ch:=' ' to 'z' do d[ch]:=M;*

*for j:=0 to M-2 do d[p[j]]:=M-j-1;*

*{ Пошук слова p в тексті s}*

*i:=M;*

*repeat*

*j:=M; k:=i;*

*repeat { Цикл порівняння символів }*

*k:=k-1; j:=j-1; { слова, починаючи з правого.}*

*until (j<0) or (p[j]<>s[k]); { Вихід, якщо порівняли все}*

*{ слово або неспівпадання. }*

*i:=i+d[s[i-1]]; { Зрушення слова управо }*

*until (j<0) or (i>N);*

*{Виведення результату пошуку}*

*if j<0 then Writeln('Yes') {знайдений }*

*else Writeln('No'); {не знайдений}*

*Readln;*

*end.*

Майже завжди, окрім спеціально побудованих прикладів, даний алгоритм вимагає значно менше N порівнянь. У найсприятливіших же обставинах, коли останній символ слова завжди потрапляє на неспівпадаючий символ тексту, число порівнянь рівне N/M.

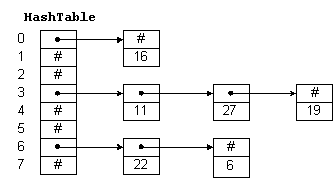
### Хешування

З хешуванням ми стикаємося чи не на кожному кроці: при роботі з браузером (список Web-посилань), текстовим редактором і перекладачем (словник), мовами скриптів (Perl, Python, PHP і ін.), компілятором (таблиця символів). За словами Брайана Кернігана, це «один з найбільших винаходів інформатики». Заглядаючи в адресну книгу, енциклопедію, алфавітний покажчик, ми навіть не замислюємося, що впорядкування за абеткою є не чим іншим, як хешуванням. Хешування є розбиття безлічі ключів (що однозначно характеризують елементи зберігання і представлених, як правило, у вигляді текстових рядків або чисел) на непересічні підмножини (набори елементів), що володіють певною властивістю. Ця властивість описується функцією хешування, або хеш-функцією, і називається хеш-адресою. Рішення зворотної задачі покладене на хеш-структури (хеш-таблиці): по хеш-адресі вони забезпечують швидкий доступ до потрібного елементу. У ідеалі для завдань пошуку хеш-адреса повинен бути унікальним, щоб за одне звернення дістати доступ до елементу, що характеризується заданим ключем (ідеальна хеш-функція). Проте, на практиці ідеал доводиться замінювати компромісом і виходити з того, що набори, що виходять, з однаковою хеш-адресою містять більше одного елементу. Термін «хешування» (hashing) в роботах по програмуванню з'явився порівняно недавно (1967 р.), хоча сам механізм був відомий і раніше. Дієслово «hash» в англійській мові означає «рубати, кришити».

**Хеш-функції**

Хеш-функція – це деяка функція h(K), яка бере якийсь ключ K і повертає адресу, по якій проводиться пошук в хеш-таблиці, щоб одержати інформацію, пов'язану з K. Наприклад, K – це номер телефону абонента, а шукана інформація – його ім'я. Функція в даному випадку нам точно скаже, за якою адресою знайти шукане.

Наприклад, на **hashTable** мал. 1 - це масив з 8 елементів. Кожен елемент є покажчиком на лінійний список, що зберігає числа. Хеш-функція в даному прикладі просто ділить ключ на 8 і використовує залишок як індекс в таблиці. Це дає нам числа від 0 до 7. Оскільки для адресації в hashTable нам і потрібні числа від 0 до 7, алгоритм гарантує допустимі значення індексів.



**Мал. 1:** Хеш-таблиця

Щоб вставити в таблицю новий елемент, ми хешируем ключ, щоб визначити список, в який його потрібно додати, потім вставляємо елемент в початок цього списку. Наприклад, щоб додати 11, ми ділимо 11 на 8 і одержуємо залишок 3. Таким чином, 11 слід розмістити в списку, на початок якого указує **hashTable[3]**. Щоб знайти число, ми його хешируем і проходимо за відповідним списком. Щоб видалити число, ми знаходимо його і видаляємо елемент списку, його що містить.

Розглянемо приклад реалізації недосконалої хеш-функції на мові TurboPascal. Припустимо, що ключ складається з чотирьох символів. При цьому таблиця має діапазон адрес від 0 до 10000.

*function hash (key : string[4]): integer;*

*var*

*f: longint;*

*begin*

*f:=ord (key[1]) - ord (key[2]) + ord (key[3]) -ord (key[4]);*

*{ обчислення функції по значенню ключа}*

*f:=f+255\*2;*

*{ поєднання початку області значень функції з початковою*

*адресою хеш-таблиці (a=1)}*

*f:=(f\*10000) div (255\*4);*

*{ поєднання кінця області значень функції з кінцевою адресою*

*хеш-таблиці (a=10 000)}*

*hash:=f*

*end;*

*Колізія* – це ситуація, коли h(K1)= h(K2), тоді як K1 <> K2. В цьому випадку, очевидно, необхідно знайти нове місце для зберігання даних. Очевидно, що кількість колізій необхідно мінімізувати. Хороша хеш-функція повинна задовольняти двом вимогам:

- її обчислення повинне виконуватися дуже швидко;

- вона повинна мінімізувати число колізій.

**Оцінка якості хеш-функції**

Як вже було відмічено, дуже важливий правильний вибір хеш-функції. При вдалій побудові хеш-функції таблиця заповнюється більш рівномірно, зменшується число колізій і зменшується час виконання операцій пошуку, вставки і видалення. Для того, щоб заздалегідь оцінити якість хеш-функції можна провести імітаційне моделювання. Моделювання проводиться таким чином. Формується цілочисельний масив, довжина якого співпадає із завдовжки хеш-таблиці. Випадково генерується достатньо велике число ключів, для кожного ключа обчислюється хеш-функція. У елементах масиву прораховується число генерацій даної адреси. За наслідками такого моделювання можна побудувати графік розподілу значень хеш-функції. Для отримання коректних оцінок число ключів, що генеруються, повинне у декілька разів перевищувати довжину таблиці.

Якщо число елементів таблиці достатньо велике, то графік будується не для окремих адрес, а для груп адрес. Наприклад, весь адресний простір розбивається на 100 фрагментів і підраховується число попадань адреси для кожного фрагмента. Великі нерівномірності свідчать про високу вірогідність колізій в окремих місцях таблиці. Зрозуміло, така оцінка є наближеною, але вона дозволяє заздалегідь оцінити якість хеш-функції і уникнути грубих помилок при її побудові.

Оцінка буде точнішою, якщо ключі, що генеруються, будуть ближчі до реальних ключів, використовуваних при заповненні хеш-таблиці. Для символьних ключів дуже важливо добитися відповідності кодів символів, що генеруються, тим кодам символів, які є в реальному ключі. Для цього варто проаналізувати, які символи можуть бути використані в ключі.

Наприклад, якщо ключ є прізвищем російській мовою, то будуть використані російські букви. Причому перший символ може бути великою буквою, а решта v малими. Якщо ключ є номерним знаком автомобіля, то також нескладно визначити допустимі коди символів в певних позиціях ключа.

Розглянемо приклад генерації ключа з десяти латинських букв, перша з яких є великою, а решта малими.

Приклад

: ключ 10 символів, 1-й велика латинська буква

2-10 малі латинські букви

var i:integer; s:string[10];

begin

s[1]:=chr(random(90-65)+65);

for i:=2 to 10 do s[i]:=chr(random(122-97)+97);

end

У даному фрагменті використовується той факт, що допустимі коди символів маються в своєму розпорядженні послідовними безперервними ділянками в кодовій таблиці. Розглянемо більш загальний випадок. Припустимо, необхідно згенерувати ключ з m символів з кодами в діапазоні від n1 до n2.

Генерація ключа з m символів з кодами в діапазоні від n1 до n2

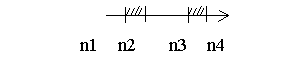
(діапазон безперервний)

for i:=1 to m do str[i]:=chr(random(n2-n1)+n1);

На практиці можливі варіанти, коли символи в одних позиціях ключа можуть належати до різних діапазонів кодів, причому між цими діапазонами може існувати розрив.

Генерація ключа з m символів з кодами

у діапазоні від n1 до n4 (діапазон має розрив від n2 до n3)



for i:=1 to m do

begin

x:=random((n4 - n3) + (n2 v n1));

if x<=(n2 - n1) then str[i]:=chr(x + n1)

else str[i]:=chr(x + n1 + n3 v n2)

end;

Розглянемо ще один конкретний приклад. Допустимий відомо, що ключ складається з 7 символів. З них три перші символи v великі латинські букви, далі йдуть дві цифри, решта v мала латинська.

Приклад: довжина ключа 7 символів

1. 3 великі латинські (коди 65-90)
2. 2 цифри (коди 48-57)
3. 2 малі латинські (коди 97-122)

var

key: string[7];

begin

for i:=1 to 3 do key[i]:=chr(random(90-65)+65);

for i:=4 to 5 do key[i]:=chr(random(57-48)+57);

for i:=6 to 7 do key[i]:=chr(random(122-97)+97);

end;

У даних прикладах ми виходили з припущення, що хешування буде реалізовано на мові Turbo Pascal, а коди символів відповідають альтернативному кодуванню.

*Застосування хешування*

Одне з побічних застосувань хешування полягає в тому, що воно створює свого роду зліпок, «відбиток пальця» для повідомлення, текстового рядка, області пам'яті і т.п. Такий «відбиток пальця» може прагнути як до «унікальності», так і до «схожості» (яскравий приклад зліпка — контрольна сума CRC). У цій якості однією з найважливіших областей застосування є криптографія. Тут вимоги до хеш-функцій мають свої особливості. Крім швидкості обчислення хеш-функції потрібно значно ускладнити відновлення повідомлення (ключа) по хеш-адресі. Відповідно необхідно утруднити знаходження іншого повідомлення з тією ж хеш-адресою. При побудові хеш-функції однонаправленого характеру звичайно використовують функцію стиснення (видає значення довжини n при вхідних даних більше довжини m і працює з декількома вхідними блоками). При хешуванні враховується довжина повідомлення, щоб виключити проблему появи однакових хеш-адрес для повідомлень різної довжини. Найбільшу популярність мають наступні хеш-функції [17]: MD4, MD5, RIPEMD-128 (128 біт), RIPEMD-160, SHA (160 біт). У російському стандарті цифрового підпису використовується розроблена вітчизняними криптографами хеш-функція (256 біт) стандарту ГОСТ Р 34.11—94.

*Хешування паролів*

Нижче передбачається, що для шифрування використовується 128-бітовий ключ. Зрозуміло, це не більше, ніж конкретний приклад. Хешування паролів – метод, що дозволяє користувачам запам'ятовувати не 128 байт, тобто 256 шістнадцяткових цифр ключа, а деякий осмислений вираз, слово або послідовність символів, що називається паролем. Дійсно, при розробці будь-якого криптоалгоритма слід враховувати, що в половині випадків кінцевим користувачем системи є людина, а не автоматична система. Це ставить питання про те, зручно, і чи взагалі реально людині запам'ятати 128-бітовий ключ (32 шістнадцяткових цифри). Насправді межа запам'ятовуваності лежить на межі 8-12 подібних символів, а, отже, якщо ми примушуватимемо користувача оперувати саме ключем, тим самим ми практично змусимо його до запису ключа на якому-небудь листку паперу або електронному носії, наприклад, в текстовому файлі. Це, природно, різко знижує захищеність системи. Для вирішення цієї проблеми були розроблені методи, що перетворюють вимовний, осмислений рядок довільної довжини – пароль, у вказаний ключ наперед заданої довжини. У переважній більшості випадків для цієї операції використовуються так звані хеш-функції. Хеш-функцією в даному випадку називається таке математичне або алгоритмічне перетворення заданого блоку даних, яке володіє наступними властивостями:

1. хеш-функція має нескінченну область визначення

2. хеш-функція має кінцеву область значень

3. вона необратима

4. зміна вхідного потоку інформації на один біт міняє близько половини всіх біт вихідного потоку, тобто результату хеш-функції.

Ці властивості дозволяють подавати на вхід хеш-функції паролі, тобто текстові рядки довільної довжини на будь-якій національній мові і, обмеживши область значень функції діапазоном 0..2N-1, де N – довжина ключа в бітах, одержувати на виході достатньо рівномірно розподілені по області значення блоки інформації – ключі.

Хешування, яке народилося ще у середині минулого століття, активно використовується в наші дні скрізь, де потрібно провести швидку вибірку даних. З'явилися нові методи хешування, нові модифікації алгоритмів, написаних раніше. На думку Дональда Кнута, найбільш важливим відкриттям у області хешування з часів 70 років, ймовірно, є лінійне хешування Вітольда Литвина . Лінійне хешування, яке не має нічого спільного з класичною технологією лінійної адресації, дозволяє багатьом хеш-адресам рости и/или виступати в поли елементів, що вставляються і видаляються. Лінійне хешування може також використовуватися для величезних баз даних, розподілених між різними вузлами в мережі. Зрозуміло, методи і сфери застосування хешування не обмежуються тим, що представлене в цій роботі. Не вдаючись в строгий аналіз ефективності, були розглянуті тільки базові, найбільш відомі методи. Крім них можна відзначити поліноміальне хешування (М. Ханан і ін., 1963), впорядковане хешування (О. Амбль, 1973), лінійне хешування (У. Литвин, 1980).

**Завдання.**

*Варіант 1.*

Реалізувати лінійний пошук елемента в масиві.

*Варіант 2.*

Реалізувати двійковий пошук елемента в масиві.

*Варіант 3.*

Реалізувати пошук елемента в бінарному дереві пошуку.

*Варіант 4.*

Реалізувати простий алгоритм пошуку слова в тексті.

*Варіант 5.*

Реалізувати алгоритм Рабіна-Карпа пошуку слова в тексті для коротких рядків.

*Варіант 6.*

Реалізувати поліпшений алгоритм Рабіна-Карпа пошуку слова в тексті.

*Варіант 7.*

Реалізувати алгоритм Кнута-Моріса-Пратта пошуку слова в тексті.

*Варіант 8.*

Реалізувати алгоритм Боуєра-Мура пошуку слова в тексті.

*Варіант 9.*

Написати програму, що організовує дані у вигляді хеш-таблиці, здійснює додавання нового елемента в таблицю, вилучення елемента з таблиці та пошук заданого елемента.

*Варіант 10.*

Реалізувати лінійний пошук елемента в масиві.

*Варіант 11.*

Реалізувати двійковий пошук елемента в масиві.

*Варіант 12.*

Реалізувати пошук елемента в бінарному дереві пошуку.

*Варіант 13.*

Реалізувати простий алгоритм пошуку слова в тексті.

*Варіант 14.*

Реалізувати алгоритм Рабіна-Карпа пошуку слова в тексті для коротких рядків.

*Варіант 15.*

Реалізувати поліпшений алгоритм Рабіна-Карпа пошуку слова в тексті.

***Контрольні питання:***

1. Які алорими пошуку ви знаєте?
2. Що таке хеш-функція?
3. В чому полягає алгоритм хешування?
4. Що таке колізія?
5. Які існують методи розв'язання колізій?