Алгоритми і структури даних

Опорний конспект лекцій

ВСТУП

Вони служать базовими елементами будь-якої машинної програми. В організації структур даних і процедур їх обробки закладена можливість перевірки правильності роботи програми.

Ніколас Вірт

Без розуміння структур даних і алгоритмів неможливо створити серйозний програмний продукт. І слова епіграфу є цьому підтвердженням. Тому головна мета даного навчального посібника полягає в наступному:

- показати всю різноманітність існуючих структур даних, представлення їх в пам’яті на фізичному рівні і на логічному рівні;
- виконувані над ними операції фізичного і логічного рівнів;
- показати значення структурного підходу до розробки алгоритмів.

У курсі лекцій, який не є самодостатнім, і рекомендується для спрямування студентів на вивчення дисципліни, приводиться класифікація структур даних, широка інформація про фізичне і логічне представлення структур даних усіх класів пам’яті комп’ютерів: простих, статичних, напівстатичних, динамічних; вичерпна інформація про операції над усіма перерахованими структурами. Приведено достатньо велика кількість алгоритмів виконання особливо важливих операцій.

Логічно курс складається з двох частин. У першій частині описується вся різноманітність структур даних та базові операції, які виконуються над ними. У другій частині описано методи створення алгоритмів та приводяться основні базові алгоритми над різними структурами даних.

1. АЛГОРИТМИ І ДАНІ

1.1. Структурування і абстракція програм

Будь-які достатньо великі програми проектуються шляхом декомпозиції задачі – виділення в задачі деяких структур і їхніх абстракцій. Абстрагування від проблеми передбачає ігнорування ряду деталей для того, щоб звести задачу до більш простої. Задачі абстрагування і наступна декомпозиція типові для процесу створення програм: декомпозиція використовується для поділу програм на компоненти; абстрагування ж передбачає продуманий вибір компонентів для цієї задачі.

Структурний підхід до даних і алгоритмів дає можливість структурування складної програми. Розробляти сучасну програмну методом „все відразу" неможливо, вона повинна бути представлена в вигляді деякої структури – складових частин і зв'язків між ними. Правильне структурування дає можливість на кожному етапі розробки зосередити увагу розробника на одній оглядовій її частині або доручити реалізацію різних її частин різним виконавцям.

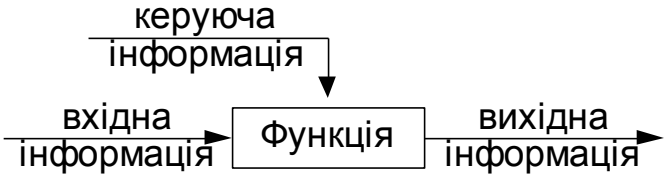
При структуруванні програм можна застосовувати підхід, який базується на **структуризації алгоритмів** і відомий, як „низхідне" проектування – „програмування з верху в низ", або підхід, який базується на **структуризації даних** і відомий, як „висхідне" проектування – „програмування з низу в верх".

У першому випадку – **структуризації алгоритмів** – структурують перш за все дії, які повинна виконувати програма. Велику і складну задачу представляють у вигляді декількох задач меншого обсягу. Таким чином, модуль самого верхнього рівня, який відповідає за вирішення задачі в цілому, отримується достатньо простим і він забезпечує тільки послідовність звертань до модулів, які вирішують задачі нижчого рівня. Потім кожна задача в свою чергу підлягає декомпозиції за такими ж правилами. Процес дроблення продовжується до тих пір, поки на черговому рівні декомпозиції не отримують задачу, реалізація якої буде досить простою. Для цього випадку будь-яку програму можна представити набором наступних функціональних

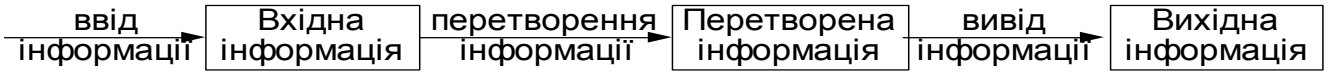
абстракцій.

Аналізуючи таке представлення програми, можна отримати узагальнену абстракцію функції.

Інший підхід базується на **структуризації даних** (програмування – це обробка даних). У програмах можна застосовувати різні алгоритми, але в реальної програми завжди є замовник, який маючи вхідні дані, хоче, щоб за ними можна було отримати вихідні дані, а якими засобами це забезпечується – його не цікавить. Таким чином, задачею будь-якої програми є перетворення вхідних даних у вихідні. Мови програмування представляють набір базових типів даних і операції над ними. Інтегруючи базові типи, програміст створює більш складні типи даних і визначає нові операції над складними типами. В ідеалі останній крок композиції дає типи даних, які відповідають вхідним і вихідним даним задачі, а

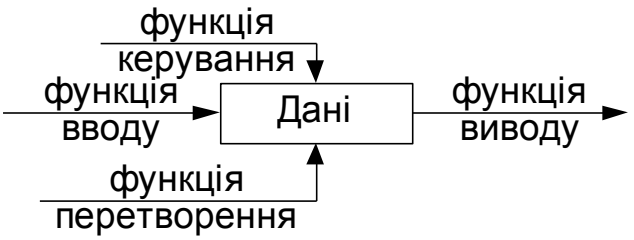


операції над цими типами реалізують в повному обсязі задачу проекту. Розглядаючи програму не як набір функцій, а перш за все, як деякий набір даних, кожен, із яких має дозволену групу функцій, отримують наступне абстрактне представлення програми.



Аналіз такого представлення програми дозволяє також отримати абстракцію даних.

Отримані абстракції алгоритмів і даних лежать в основі багатьох формалізованих методів розробки програмного забезпечення.



Не можна протиставляти низхідне проектування висхідному і дотримуватись одного вибраного підходу. Реалізація будь-якого реального проекту завжди ведеться зустрічними шляхами, причому, з постійною корекцією структур алгоритмів за результатами розробки структур даних і навпаки.

Ще одним досить продуктивним технологічним методом, який зв'язаний з структуризацією даних є **інкапсуляція**. Зміст її полягає у тому, що сконструйований новий тип даних – „будівельний блок" – оформляється таким чином, що його внутрішня структура стає недоступною для програміста – користувача цього типу. Програміст, який використовує цей тип даних у своїй програмі, може оперувати з даними цього типу тільки через виклик функцій, які визначені для цього типу. Новий тип даних представляється для нього у вигляді „чорного ящика" для якого відомі входи й виходи, але вміст – невідомий і недоступний.

Сучасні мови програмування блочного типу мають достатньо розвинуті можливості побудови програм з модульною структурою і управління доступом модулів до даних і функцій. Розширення ж мов додатковими можливостями конструювання типів і їх інкапсуляції робить мову об'єктно-орієнтованою. Сконструйовані й повністю закриті типи даних представляють собою класи, а функції, які працюють з їх внутрішньою структурою – методи роботи з класами. При цьому в значній мірі міняється й сама концепція програмування. Програміст, який оперує об'єктами, вказує в програмі „що" потрібно зробити з об'єктом, а не „як" це потрібно зробити.

1.2. Концепція структур даних

Структури даних і алгоритми є тими матеріалами, з яких будуються програми. Більше того, сам комп'ютер складається з структур даних і алгоритмів. Вбудовані структури даних представлені регістрами й словами пам'яті, де зберігаються бінарні величини. Закладені в конструкцію апаратури алгоритми – це реалізація в електронних логічних схемах жорстких правил, за якими поміщені в пам'ять дані інтерпретуються як команди, що підлягають виконанню. Тому в основі роботи будь-якого комп'ютера лежить вміння оперувати лише з одним видом даних – з окремими бітами. Працює ж з цими даними комп'ютер тільки у відповідності з

незмінним набором алгоритмів, які визначаються системою команд центрального процесора.

Задачі, які вирішуються за допомогою комп'ютера, рідко представляються мовою бітів. Як правило, дані мають форму чисел, літер, текстів, символів і більш складних структур типу послідовностей, списків і дерев.

Структура даних відноситься за своєю суттю до „просторових" понять: її можна звести до схеми організації інформації в пам'яті комп'ютера. Алгоритм ж є відповідним процедурним елементом в структурі програми – він служить рецептом розрахунку.

Структури даних, які застосовуються в алгоритмах, можуть бути досить складними. Вибір правильного представлення даних часто служить ключем до вдалого програмування і може в більшій мірі впливати на продуктивність програми, ніж деталі реалізації використовуваного алгоритму. Але, мабуть, ніколи не появиться загальна теорія вибору структур даних, у кожному конкретному випадку потрібно підходити до цього творчо.

Незалежно від змісту і складності будь-які дані в пам'яті комп'ютера представляються послідовністю бінарних розрядів, а їх значеннями є відповідні бінарні числа. Дані, які розглядаються у вигляді послідовності бітів, мають дуже просту організацію, тобто є слабо структурованими. Більш крупні й змістовніші, ніж біт, „будівельні блоки" для організації довільних даних отримуються на основі поняття „структури даних".

Поняття „фізична структура даних" відображає спосіб фізичного представлення даних в пам'яті машини і називається ще **структурою зберігання.**

Розгляд структури даних без врахування її представлення в машинній пам'яті називається абстрактною або **логічною структурою**. В загальному випадку між логічною і відповідною їй фізичною структурами існує відмінність, міра якої залежить від самої структури і особливостей того середовища, в якому вона повинна бути відображена.

Розгляд структур даних утруднений досить незручним змішуванням абстрактних властивостей структур даних і проблемами представлення, які зв'язані з комп'ютером і машинною мовою. Важливо чітко розрізняти ці проблеми за допомогою багаторівневого опису. Будь-яка структура даних може описуватися, таким чином, на трьох різних рівнях:

- **Функціональна специфікація** – вказує для деякого класу імен операції, які дозволені з цими іменами, і властивості цих операцій;
- **Логічний опис** – задає декомпозицію об'єктів на більш елементарні об'єкти і декомпозицію відповідних операцій на більш елементарні операції;
- **Фізичне представлення** – дає метод розміщення в пам'яті комп'ютера тих величин, які складають структуру, і відношення між ними, а також спосіб кодування операцій на мові програмування.

Одній і тій же функціональній специфікації можуть відповідати декілька логічних описів, які в свою чергу можуть реалізовуватися декількома фізичними представленнями. Проте, потрібно мати впевненість, що декомпозиція кожного нового рівня достатньо добре відображає декомпозицію безпосередньо вищого рівня.

1.3. Класифікація структур даних

Під структурою даних в загальному випадку розуміють множину елементів даних і множину зв'язків між ними.

Розрізняються **прості** (базові) **структури даних** і **інтегровані** (структуровані). **Простими** називаються такі структури даних, які не можуть бути розділені на складові частини більші, ніж біти. З точки зору фізичної структури важливою є та обставина, що в даній машинній архітектурі, в даній системі програмування завжди можна наперед сказати, яким буде розмір даного простого типу і яка структура його розміщення в пам'яті. З логічної точки зору прості дані є неподільними одиницями.

Інтегрованими називаються такі структури даних, складовими частинами яких є інші структури даних – прості але в свою чергу інтегровані. Інтегровані структури даних конструюються програмістом з використанням засобів інтеграції даних, які представляються мовами програмування.

У залежності від наявності чи відсутності явно заданих зв'язків між елементами даних потрібно розрізняти **незв'язні** структури і **зв'язні** структури.

Досить важлива ознака структури даних – її змінність – зміна кількості елементів і/або зв'язків між елементами структури. За ознакою змінності розрізняють структури **статичні, напівстатичні, динамічні.**

Ще одна важлива ознака структури даних – характер впорядкованості її елементів. За цією ознакою структури можна поділити на **лінійні** й **нелінійні** структури.

1.4. Операції над структурами даних

Над будь-якими структурами даних можуть виконуватися чотири загальні операції: створення, знищення, вибір (доступ), поновлення.

Операція **створення** полягає у виділенні пам'яті для зберігання структури даних. Пам'ять може виділятися в процесі виконання програми або на етапі компіляції. У деяких мовах для структурованих даних, які конструюються програмістом, операція створення включає в себе також встановлення початкових значень параметрів, створюваної структури (ініціалізація).

Незалежно від використовуваної мови програмування, наявні в програмі структури даних не появляються „з нічого", а явно або неявно оголошуються операторами створення.

Операція **знищення** структур даних протилежна до операції створення – вона звільнює пам'ять, яку займала структура, для подальшого використання. Операція знищення дозволяє ефективно використовувати пам'ять.

Операція **вибору** використовується програмістами для доступу до даних в самій структурі. Форма операції доступу залежить від типу структури даних, до якої здійснюється звертання. Метод доступу – одна з найбільш важливих властивостей структур, тому що вона має безпосереднє відношення до вибору конкретної структури даних.

Операція **поновлення** дозволяє змінити значення даних в структурі даних. Прикладом операції поновлення є операція присвоєння, або, більш складна форма – передача параметрів.

Вищевказані чотири операції обов’язкові для всіх структур і типів даних. Крім цих загальних операцій для кожної структури даних можуть бути визначені специфічні операції, які працюють тільки з даними конкретного типу чи структури.

2. ПРОСТІ СТРУКТУРИ ДАНИХ

2.1. Арифметичні типи

Стандартні типи даних часто називають арифметичними, оскільки їх можна використовувати в арифметичних операціях. Для опису основних типів визначено ключові слова: цілий, символьний, логічний, дійсний.

За допомогою цілих чисел можна представити кількість об’єктів, яка є дискретною за своєю природою (тобто кількість об’єктів можна перерахувати). Внутрішнє представлення величин цілого типу – ціле число в бінарному коді. Внутрішнє представлення дійсного типу складається з двох частин – мантиси і порядку.

Результати логічного типу отримуються при порівнянні даних будь-яких типів. Величини логічного типу можуть приймати тільки значення **true** і **false.** Внутрішня форма представлення значення **false** – 0 (нуль). Будь-яке інше значення інтерпретується як **true.**

Значенням символьного типу є символи з деякої наперед визначеної множини. В більшості сучасних персональних комп’ютерів цією множиною є ASCII. Ця множина складається з 256 різних символів, які впорядковані певним чином і містить символи великих і малих букв, цифр і інших символів, включаючи спеціальні керуючі символи. Значення символьного типу займає в пам’яті 1 байт. Іншою широко використовуваною множиною для представлення символьних даних є код Unicode. В Unicode кожний символ кодується двома байтами.

Над арифметичними типами, як і над всіма іншими, можливі перш за все чотири основних операції: створення, знищення, вибір, поновлення. Специфічні операції над числовими типами – додавання, віднімання, множення і ділення.

Ще одна група операцій над арифметичними типами – операції порівняння: „рівно”, „не рівно”, „більше”, „менше” і т.п. Говорячи про операції порівняння, потрібно звернути увагу на особливість виконання порівнянь на рівність/нерівність дійсних чисел. Оскільки ці числа представляються в пам’яті з деякою точністю, порівняння їх не завжди може бути абсолютно достовірним.

2.2. Перерахований тип

При написанні програм часто виникає потреба визначити декілька іменованих констант, для яких потрібно, щоб усі вони мали різні значення. Для цього зручно використовувати перерахований тип.

Перерахований тип представляє собою впорядкований тип даних, який визначається програмістом, тобто програміст перераховує всі значення, які може приймати змінна цього типу.

Діапазон значень перечислень визначається кількістю біт, які необхідні для представлення всіх його значень. Будь-яке значення цілого типу можна явно привести до перерахованого типу, але при виході за межі його діапазону результат буде невизначеним. При відсутності ініціалізації перша константа приймає нульове

значення, а кожній наступній присвоюється на одиницю більше значення від попереднього.

На фізичному рівні над змінними перерахованого типу визначені операції створення, знищення, вибору, поновлення. При цьому виконується визначення порядкового номера ідентифікатора за його значенням і, навпаки, за номером ідентифікатора його значення.

При виконанні арифметичних операцій перечислення перетворюються у ціле. Оскільки перечислення є типом, який визначається користувачем, для нього можна вводити власні операції.

2.3. Покажчики

Тип „покажчик” представляє собою адресу комірки пам’яті (в більшості сучасних обчислювальних систем розмір комірки – мінімальної адресованої одиниці пам’яті – складає один байт). При програмуванні на низькому рівні робота з адресами складає значну частину програм. При вирішенні прикладних задач з використанням мов високого рівня найбільш часті випадки, коли програміст може використовувати покажчики, наступні:

1. При необхідності представити одну й ту ж ділянку пам’яті, а значить, одні й ті ж фізичні дані, як дані різної логічної структури.
2. При роботі з динамічними структурами даних.

В C++ розрізняють три види покажчиків – покажчик на об’єкт, на функцію і на пустий тип, які відрізняються властивостями і набором допустимих операцій. Покажчик не є самостійним типом, він завжди зв’язаний з якимось іншим типом.

У програмах на мовах високого рівня покажчики можуть бути типізованими і нетипізованими. При оголошені типізованого покажчика визначається й тип об’єкту в пам’яті, який адресується цим покажчиком.

Хоча фізична структура адреси не залежить від типу й значення даних, які зберігаються за цією адресою, компілятор вважає покажчики на різні типи такими, що мають різний тип. Таким чином, коли йде мова про типізовані покажчики, правильніше говорити не про єдиний тип даних „покажчик”, а про цілу сім’ю типів: „покажчик на ціле”, „покажчик на символ”.

Нетипізований покажчик використовується для представлення адреси, за якою містяться дані невідомого типу. Робота з нетипізованими покажчиками суттєво обмежена, вони можуть використовуватися тільки для збереження адреси, звертатися за такою адресою не можна.

Основними операціями, в яких беруть участь покажчики є присвоєння, отримання адреси, вибір.

Перерахованих операцій достатньо для вирішення задач прикладного програмування тому набір операцій над покажчиками в більшості мов програмування цим й обмежується. Системне програмування потребує більш гнучкої роботи з адресами, тому в C/C++ доступні також операції адресної арифметики.

3. СТАТИЧНІ СТРУКТУРИ ДАНИХ

Статичні структури відносяться до класу структур, які представляють собою структуровану множину примітивних, базових, структур. Оскільки статичні

структури відрізняються відсутністю змінності, пам’ять для них виділяється автоматично – як правило, на етапі компіляції, або при виконанні – в момент активізації того програмного блоку, в якому вони описані. Ряд мов програмування допускають розміщення статичних структур в пам’яті на етапі виконання за явною вимогою програміста, але й у цьому випадку обсяг виділеної пам’яті залишається незмінним до знищення структури. Виділення пам’яті на етапі компіляції є такою зручною властивістю статичних структур, що у ряді задач програмісти використовують їх навіть для представлення об’єктів, які мають властивість змінності. Наприклад, коли розмір масиву невідомий наперед, для нього резервується максимально можливий розмір.

Статичні структури в мовах програмування зв’язані із структурованими типами. Структуровані типи в мовах програмування є тими засобами інтеграції, які дозволяють будувати структури даних будь-якої складності. До таких типів відносяться масиви, структури та їхні похідні типи.

3.1. Масиви

Логічно масив об’єднує елементи одного типу даних, тобто належить до однорідного типу даних. Більше формально його можна визначити як впорядковану сукупність елементів деякого типу, які адресуються за допомогою одного або декількох індексів.

Масиви можна класифікувати за кількістю розмірностей масиву масиви поділяються на одновимірні масиви (вектори), двохвимірні (матриці) і багатовимірні (трьох, чотирьох і більше).

Логічно масив – це така структура даних, яка характеризується:

- фіксованим набором елементів одного і того ж типу;
- кожний елемент має унікальний набір значень індексів;
- кількість індексів визначають мірність масиву;
- звернення до елемента масиву виконується за ім’ям масиву і значенням індексів для даного елемента.

Фізична структура масиву – це спосіб розміщення елементів масиву в пам’яті комп’ютера. Під елемент масиву виділяється кількість байт пам’яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам’яті для зберігання масиву.

Сама найважливіша операція фізичного рівня над масивом – доступ до заданого елемента. Як тільки реалізовано доступ до елемента, над ним може бути виконана будь-яка операція, що має сенс для того типу даних, якому відповідає елемент. Перетворення логічної структури масиву у фізичну називається процесом лінеаризації, в ході якого багатовимірна логічна структура масиву перетвориться в одновимірну фізичну структуру.

Адресою масиву є адреса першого байту початкового компоненту масиву. Індексація масивів в C/C++ обов’язково починається з нуля.

До операцій логічного рівня над масивами необхідно віднести такі як сортування масиву, пошук елемента за ключем.

3.2. Розріджені масиви

На практиці зустрічаються масиви, які через певні причини можуть займати пам’ять не повністю, а частково. Це особливо актуально для масивів великих розмірів, таких що для їхнього зберігання в повному об’ємі пам’яті може бути недостатньо. Розріджений масив – це масив, більшість елементів якого рівні між собою, так що зберігати в пам’яті достатньо лише невелику кількість значень відмінних від основного (фонового) значення інших елементів. При роботи з розрідженими масивами питання розташування їх в пам’яті реалізуються на логічному рівні з врахуванням їхнього типу.

Розрізняють два типи розріджених масивів:

- масиви, в яких розташування елементів із значеннями відмінними від фонового, можуть бути описані математично;
- масиви з випадковим розташуванням елементів.

До **масивів з математичним описом розташування елементів** відносяться масиви, в яких існує закономірності в розташуванні елементів із значеннями відмінними від фонового.

Елементи, значення яких є фоновими, називають нульовими; елементи, значення яких відмінні від фонового, – ненульовими. Фонове значення не завжди рівне нулю. Ненульові значення зберігаються, як правило, в одновимірному масиві, а зв’язок між розташуванням у розрідженому масиві і в новому, одновимірному, описується математично за допомогою формули, що перетворює індекси масиву в індекси вектора.

На практиці для роботи з розрідженим масивом розробляються функції:

- для перетворення індексів масиву в індекс вектора;
- для отримання значення елемента масиву з його упакованого представлення за індексами;
- для запису значення елемента масиву в її упаковане представлення за індексами.

До **масивів з випадковим розташуванням елементів** відносяться масиви, в яких не існує закономірностей у розташуванні елементів із значеннями відмінними від фонового.

Один з основних способів зберігання подібних розріджених матриць полягає в запам’ятовуванні ненульових елементів в одновимірному масиві і ідентифікації кожного елемента масиву індексами.

Дане представлення масиву скорочує вимоги до об’єму пам’яті більш ніж в 2 рази. Спосіб послідовного розподілу має також ту перевагу, що операції над матрицями можуть бути виконані швидше, ніж це можливо при представленні у вигляді послідовного масиву, особливо якщо розмір матриці великий.

Методи послідовного розміщення для представлення розріджених матриць звичайно дозволяють швидше виконувати операції над матрицями і більш ефективно використати пам’ять, ніж методи із зв’язаними структурами. Проте послідовне представлення матриць має певні недоліки. Так включення і виключення нових елементів матриці викликає необхідність переміщення великої кількості інших елементів. Якщо включення нових елементів і їхнє виключення здійснюється часто, то повинен бути вибраний метод зв’язаних структур.

Метод зв’язаних структур, проте, переводить структуру даних, що представляється, в інший розділ класифікації. При тому, що логічна структура даних залишається статичною, фізична структура стає динамічною.

3.3. Множини

Тип даних „множина” не реалізований як стандартний тип мови програмування C/C++, але дуже часто використовується в програмуванні і реалізовується засобами визначення типів користувача, тому дамо йому короткий опис.

Множина – така структура, яка є набором даних одного і того ж типу, що не повторюються (кожен елемент множини є унікальним). Порядок слідування елементів множини на має принципового значення.

До множин застосовується стандартний принцип виключення. Це означає, що конкретний елемент або є членом множини, або ні. Множина може бути пустою, таку множину називають нульовою.

Множина є підмножиною іншої множини, якщо в цій другій множині можна знайти усі елементи, які є в першій множині. Відповідно, множина вважається надмножиною іншої множини, якщо вона містить усі елементи цієї другої множини.

Кожен окремий елемент є членом множини, якщо він входить до складу елементів множини.

Над множинами визначені наступні специфічні операції:

1. Об’єднання множин. Результатом є множина, що містить елементи початкових множин.
2. Перетин множин. Результатом є множина, що містить спільні елементи початкових множин.
3. Різниця множин. Результатом є множина, яка містить елементи першої множини, які не входять в другу множину.
4. Симетрична різниця. Результатом є множина, яка містить елементи, які входять до складу однієї або другої множини (але не обох).
5. Перевірка на входження елемента в множину. Результатом цієї операції є значення логічного типу, що вказує чи входить елемент в множину.

3.4. Структури

На відміну від масивів чи множин, усі елементи яких однотипні, структура може містити елементи різних типів.

Елементи структури називаються полями структури і можуть мати довільний тип, крім типу цієї ж структури, але можуть бути покажчиками на неї. Якщо при описі структури відсутній тип структури, обов’язково повинен бути вказаний список змінних, покажчиків або масивів визначеної структури.

Звернення до окремих полів структури замінюються на їхні адреси ще на етапі компіляції.

Самою найважливішою операцією для структури є операція доступу до вибраного поля структури – операція кваліфікації.

Над вибраним полем структури можливі будь-які операції, які допустимі для типів цього поля.

Більшість мов програмування підтримує деякі операції, які працюють із структурою, як з єдиним цілим, а не з окремими її полями. Це операція присвоєння значення одного запису іншому однотипному запису, при цьому відбувається по елементне копіювання.

3.5. Об'єднання

Об'єднання представляють собою частковий випадок структури, усі поля якої розміщуються за однією ж і тою ж адресою. Формат опису такий же, як і в структури. Довжина об'єднання рівна найбільшій із довжин його полів. У кожен момент часу в змінній типу об'єднання зберігається тільки одне значення, і відповідальність за його правильне використання лягає на програміста.

Об'єднання застосовуються для економії пам'яті в тих випадках, коли відомо, що більше одного поля одночасно не потрібно, а також для різної інтерпретації одного і того ж бітового представлення.

Дуже часто деякі об'єкти програми відносяться до одного й того ж класу, відрізняючись лише деякими деталями. У цьому випадку застосовують комбінацію структурного типу і об'єднання. Об'єднання використовують як поля структури, при цьому в структурі включають поле, яке визначає, який саме елемент об'єднання використовується в кожний момент.

У загальному випадку змінна структура буде складатися з трьох частин: набір спільних компонентів, мітки активного компоненту і частини зі змінними компонентами.

3.6. Бітові типи

В ряді задач може стати в нагоді робота з окремими бінарними розрядами даних. Частіше всього такі задачі виникають в системному програмуванні, коли, наприклад, окремий розряд зв'язаний з станом окремого апаратного перемикача або окремої шини передачі даних. Дані такого типу представляються у вигляді набору бітів, які упаковані в байти або слова, і логічно не зв'язаних один з одним. Операції над такими даними забезпечують доступ до вибраного біта даного.

Бітові поля – це особливий вид полів структури. Вони використовуються для компактного розміщення даних, наприклад, прапорців типу „так/ні". Мінімально адресована комірка пам'яті – 1 байт, а для зберігання прапорця достатньо одного біта. Бітові поля описуються за допомогою структурного типу.

Бітові поля можуть бути довільного цілого типу. Ім'я поля може бути відсутнім, такі поля використовуються для вирівнювання на апаратну межу. Доступ до поля здійснюється звичайним способом – за іменем.

Над бітовими типами можливі три групи специфічних операцій: операції алгебри логіки, операції зсуву, операції порівняння.

Операції бульової алгебри – *НІ* („~"), *АБО* („|"), *І* („&"), *виключне АБО* („^"). Ці операції і за назвою, і за змістом подібні на операції над логічними аргументами, але відмінність у їх застосуванні до бітових аргументів полягає в тому, що операції виконуються над окремими розрядами. В мові С/С++ для побітових і загальних логічних операцій використовуються різні позначення.

Операції зсуву виконують зміщення бінарного коду на задану кількість розрядів ліворуч або праворуч. Із трьох можливих типів зсуву (арифметичний, логічний, циклічний) в мовах програмування частіше реалізується лише логічний.

3.7. Таблиці

Елементами векторів і масивів можуть бути інтегровані структури. Одна з таких складних структур – таблиця. З фізичної точки зору таблиця є вектором, елементами якого є структури. Характерною логічною особливістю таблиць є те, що доступ до елементів таблиці проводиться не за номером (індексом), а за ключем – значення однієї з властивостей об'єкту, який описується структурою-елементом таблиці. Ключ – це властивість, що ідентифікує дану структуру в множині однотипних структур і є, як правило, унікальним в даній таблиці. Ключ може включатися до складу структури і бути одним з його полів, але може і не включатися в структуру, а обчислюватися за деякими її властивостями. Таблиця може мати один або декілька ключів.

Основною операцією при роботі з таблицями є операція доступу до структури за ключем. Вона реалізовується процедурою пошуку. Оскільки пошук може бути значне більш ефективним в таблицях, впорядкованих за значеннями ключів, досить часто над таблицями необхідно виконувати операції сортування.

Іноді розрізняють таблиці з фіксованою і із змінною довжиною структури. Очевидно, що таблиці, які об'єднують структури ідентичних типів, будуть мати фіксовані довжини структур. Необхідність в змінній довжині може виникнути в задачах, подібних до тих, які розглядалися для об'єднань. Як правило таблиці для таких задач і складаються із структур до складу яких входять об'єднання, тобто зводяться до фіксованої (максимальній) довжини структури. Значно рідше зустрічаються таблиці з дійсно змінною довжиною структури. Хоча в таких таблицях і економиться пам'ять, але можливості роботи з такими таблицями обмежені, оскільки за номером структури неможливо визначити її адресу. Таблиці із структурами змінної довжини обробляються тільки послідовно – в порядку зростання номерів структур. Доступ до елемента такої таблиці звичайно здійснюється в два кроки. На першому кроці вибирається постійна частина структури, в якій міститься, – в явному чи неявному вигляді – довжина структури. На другому кроці вибирається змінна частина структури у відповідності з її довжиною. Додавши до адреси поточної структури її довжину, одержують адресу наступної структури.

4. НАПІВСТАТИЧНІ СТРУКТУРИ ДАНИХ

4.1. Характерні особливості напівстатичних структур

Напівстатичні структури даних характеризуються наступними ознаками:

- вони мають змінну довжину і прості процедури її зміни;
- зміна довжини структури відбувається в певних межах, не перевищуючи якогось максимального (граничного) значення.

Якщо напівстатичну структуру розглядати на логічному рівні, то це послідовність даних, зв'язана відносинами лінійного списку. Доступ до елемента може здійснюватися за його порядковим номером.

Фізичне представлення напівстатичних структур даних в пам’яті – це звичайно послідовність комірок в пам’яті, де кожний наступний елемент розташований в пам’яті в наступній комірці. Фізичне представлення може мати також вид одно-направленого зв’язного списку (ланцюжки), де кожний наступний елемент адресується покажчиком, який знаходиться в поточному елементі. У цьому випадку обмеження на довжину структури менш строгі.

4.2. Стеки

Стеком називається множина деякої змінної кількості даних, над якою виконуються наступні операції:

- Поповнення стеку новими даними;
- Перевірка, яка визначає чи стек пустий;
- Перегляд останніх добавлених даних;
- Знищення останніх добавлених даних.

На основі такого функціонального опису, можна сформувати логічний опис. Стек – це такий послідовний список із змінній довжиною, включення і виключення елементів з якого виконуються тільки з одного боку списку. Застосовуються і інші назви стеку – магазин, пам’ять що функціонує за принципом LIFO (Last – In – First – Out – „останнім прийшов – першим вийшов”).

Самий „верхній” елемент стеку, тобто останній добавлений і ще не знищений, відіграє особливу роль: саме його можна модифікувати й знищувати. Цей елемент називають вершиною стеку. Іншу частину стеку називають тілом стеку. Тіло стеку, само собою, є стеком: якщо виключити зі стеку його вершину, то тіло перетворюється в стек.

Основні операції над стеком – включення нового елемента (**push** – заштовхувати) і виключення елемента зі стеку (**pop** – вискакувати).

Корисними можуть бути також допоміжні операції:

- визначення поточної кількості елементів в стеку;
- очищення стеку;
- „неруйнуюче” читання елемента з вершини стека, яке може бути реалізоване, як комбінація основних операцій – виключити елемент зі стеку та включити його знову в стек.

При представленні стеку в статичній пам’яті для стеку виділяється пам’ять, як для вектора. В описі цього вектора окрім звичайних для вектора параметрів повинен знаходитися також покажчик стеку – адреса вершини стека. Обмеження даного представлення полягає в тому, що розмір стеку обмежений розмірами вектора.

Покажчик стеку може вказувати або на перший вільний елемент стеку, або на останній записаний в стек елемент. Однаково, який з цих двох варіантів вибрати, важливо надалі строго дотримуватися його при обробці стеку.

При занесенні елементу в стек елемент записується на місце, яке визначається покажчиком стеку, потім покажчик модифікується так, щоб він вказував на наступний вільний елемент (якщо покажчик вказує на останній записаний елемент, то спочатку модифікується покажчик, а потім проводиться запис елемента). Модифікація покажчика полягає в надбавці до нього або у відніманні від нього одиниці (стек росте у бік збільшення адреси).

Операція виключення елемента полягає в модифікації покажчика стеку (в напрямку, зворотному модифікації при включенні) і вибірці значення, на яке вказує покажчик стеку. Після вибірки комірка, в якій розміщувався вибраний елемент, вважається вільною.

Операція очищення стеку зводиться до запису в покажчик стеку початкового значення – адреси початку виділеної ділянки пам'яті.

Визначення розміру стека зводиться до обчислення різниці покажчиків: покажчика стеку й адреси початку ділянки.

При зв'язному представленні стеку кожен елемент стеку складається із значення і покажчика, який вказує на попередньо занесений у стек елемент. Зв'язне представлення викликає втрату пам'яті, що викликане наявністю покажчика в кожному елементі стеку, і представляє інтерес тільки у випадку, коли важко визначити максимальний розмір стеку.

Отже, для зв'язного представлення стеку потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Для виконання операцій над стеком потрібен один покажчик на вершину стеку. Створення пустого стеку полягатиме у присвоєнні покажчику на вершину нульового значення, що означатиме, що стек пустий.

Послідовність кроків для добавлення елементу в стек складається з декількох кроків:

1. Виділити пам'ять під новий елемент стеку;
2. Занесення значення в інформаційне поле;
3. Встановлення зв'язку між ним і „старою" вершиною стеку;
4. Переміщення вершини стеку на новий елемент:.

Вилучення елементу зі стеку також проводять за кілька кроків:

1. Зчитування інформації з інформаційного поля вершини стеку;
2. Встановлення на вершину стеку допоміжного покажчика;
3. Переміщення покажчика вершини стеку на наступний елемент;
4. Звільнення пам'яті, яку займає „стара" вершина стеку.

4.3. Черга

Чергою називається множина змінної кількості даних, над якою можна виконувати наступні операції:

- Поповнення черги новими даними;
- Перевірка, яка визначає чи пуста черга;
- Перегляд перших добавлених даних;
- Знищення самих перших добавлених даних.

На основі такого функціонального опису, можна сформувати логічний опис. Чергою FIFO (First – In – First – Out – „першим прийшов – першим вийшов"). називається такий послідовний список із змінній довжиною, в якому включення елементів виконується тільки з одного боку списку (хвіст черги), а виключення – з другого боку (голова черги).

Основні операції над чергою – ті ж, що і над стеком – включення, виключення, визначення розміру, очищення, „неруйнуюче" читання.

При представленні черги вектором в статичній пам’яті на додаток до звичайних для опису вектора параметрів в ньому повинні знаходитися два покажчики: на голову і на хвіст черги. При включенні елемента в чергу елемент записується за адресою, яка визначається покажчиком на хвіст, після чого цей покажчик збільшується на одиницю. При виключенні елемента з черги вибирається елемент, що адресується покажчиком на голову, після чого цей покажчик зменшується на одиницю.

Очевидно, що з часом покажчик на хвіст при черговому включенні елемента досягне верхньої межі тієї ділянки пам’яті, яка виділена для черги. Проте, якщо операції включення чергувати з операціями виключення елементів, то в початковій частині відведеної під чергу пам’яті є вільне місце. Для того, щоб місця, займані виключеними елементами, могли бути повторно використані, черга замикається в кільце: покажчики (на початок і на кінець), досягнувши кінця виділеної області пам’яті, перемикаються на її початок. Така організація черги в пам’яті називається кільцевою чергою. Можливі, звичайно, і інші варіанти організації: наприклад, всякий раз, коли покажчик кінця досягне верхньої межі пам’яті, зсовувати всі не порожні елементи черги до початку ділянки пам’яті, але як цей, так і інші варіанти вимагають переміщення в пам’яті елементів черги і менш ефективні, ніж кільцева черга.

У початковому стані покажчики на голову і хвіст вказують на початок ділянки пам’яті. Рівність цих двох покажчиків є ознакою порожньої черги. Якщо в процесі роботи з кільцевою чергою кількість операцій включення перевищує кількість операцій виключення, то може виникнути ситуація, в якій покажчик кінця „наздожене” покажчик початку. Це ситуація заповненої черги, але якщо в цій ситуації покажчики порівняються, ця ситуація буде така ж як при порожній черзі. Для розрізнення цих двох ситуацій до кільцевої черги пред’являється вимога, щоб між покажчиком кінця і покажчиком початку залишався „проміжок” з вільних елементів. Коли цей „проміжок” скорочується до одного елемента, черга вважається заповненою і подальші спроби запису в неї блокуються. Очищення черги зводиться до запису одного і того ж (не обов’язково початкового) значення в обидва покажчики. Визначення розміру полягає в обчисленні різниці покажчиків з урахуванням кільцевої природи черги.

При зв’язному представленні черги кожен елемент черги складається із значення і покажчика, який вказує на попередньо занесений у чергу елемент.

Зв’язне представлення викликає втрату пам’яті, що викликане наявністю покажчика в кожному елементі черги, і представляє інтерес тільки у випадку, коли важко визначити максимальний розмір черги. Для зв’язного представлення черги потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Для виконання операцій над чергою потрібно два покажчики: на голову і хвіст черги. Створення пустої черги полягатиме у присвоєнні покажчикам на голову і хвіст черги нульових значень, що означатиме, що черга пуста.

Послідовність кроків для добавлення елемента в кінець черги складається з декількох кроків:

1. Виділити пам’ять під новий елемент черги;

2. Занесення значення в інформаційне поле;
3. Занесення нульового значення в покажчик;
4. Встановлення зв’язку між ним і останнім елементом черги і новим, враховуючи випадок пустої черги;
5. Переміщення покажчика кінця черги на новий елемент.

Вилучення елементу з черги також проводять за кілька кроків:

1. Зчитування інформації з інформаційного поля голови черги;
2. Встановлення на голову черги допоміжного покажчика;
3. Переміщення покажчика початку черги на наступний елемент;
4. Звільнення пам’яті, яку займав перший елемент черги.

В реальних задачах іноді виникає необхідність у формуванні черг, відмінних від приведених структур. Порядок вибірки елементів з таких черг визначається пріоритетами елементів. Пріоритет в загальному випадку може бути представлений числовим значенням, яке обчислюється або на підставі значень яких-небудь полів елемента, або на підставі зовнішніх чинників. Так попередньо наведені структури стек і черги можна трактувати як пріоритетні черги, в яких пріоритет елемента залежить від часу його включення в структуру. При вибірці елемента всякий раз вибирається елемент з щонайбільшим пріоритетом.

Черги з пріоритетами можуть бути реалізовані на лінійних структурах – в суміжному або зв’язному представленні. Можливі черги з пріоритетним включенням – в яких послідовність елементів черги весь час підтримується впорядкованою, тобто кожний новий елемент включається на те місце в послідовності, яке визначається його пріоритетом, а при виключенні завжди вибирається елемент з голови. Можливі і черги з пріоритетним виключенням – новий елемент включається завжди в кінець черги, а при виключенні в черзі шукається (цей пошук може бути тільки лінійним) елемент з максимальним пріоритетом і після вибірки вилучається з послідовності. І в тому, і в іншому варіанті потрібний пошук, а якщо черга розміщується в статичній пам’яті – ще і переміщення елементів.

4.4. Деки

Дек – особливий вид черги. Дек (deq – double ended queue, тобто черга з двома кінцями) – це такий послідовний список, в якому як включення, так і виключення елементів, може здійснюватися з будь-якого з двох кінців списку. Так само можна сформулювати поняття деку, як стек, в якому включення і виключення елементів може здійснюватися з обох кінців.

Деки рідко зустрічаються у своєму первісному визначенні. Окремий випадок деку – дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури деку аналогічні логічній і фізичній структурі кільцевої черги. Проте, стосовно деку доцільно говорити не про голову і хвіст, а про лівий і правий кінець.

Над деком доступні наступні операції:

- включення елемента праворуч;
- включення елемента ліворуч;
- виключення елемента з права;
- виключення елемента з ліва;

- визначення розміру;
- очищення.

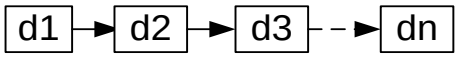
Фізична структура деку в статичній пам’яті ідентична структурі кільцевої черги.

4.5. Лінійні списки

Лінійні списки є узагальненням попередніх структур; вони дозволяють представити множину так, щоб кожний елемент був доступний і при цьому не потрібно було б зачіпати деякі інші.

Списки є досить гнучкою структурою даних, так як їх легко зробити більшими або меншими, і їх елементи доступні для вставки або вилучення в будь-якій позиції списку. Списки також можна об’єднувати або розділяти на менші списки.

Лінійний список – це скінчена послідовність однотипних елементів (вузлів), можливо, з повторенням. Кількість елементів у послідовності називається довжиною списку. Вона в процесі роботи програми може змінюватися. Лінійний список L, що складається з елементів $d_1, d_2, \ldots, d_n$, які мають однаковий тип, записують у вигляді $L = \langle d_1, d_2, \ldots, d_n \rangle$, або зображають графічно.



Важливою властивістю лінійного списку є те, що його елементи можна лінійно впорядкувати у відповідності з їх позицією в списку.

Для формування абстрактного типу даних на основі математичного визначення списку потрібно задати множину операторів, які виконуються над об’єктами типу список. Проте не існує однієї множини операторів, які виконуються над списками, які задовольняють відразу всі можливі застосування.

Найчастіше зі списками доводиться виконувати такі операції:

- Знайти елемент із заданою властивістю;
- Визначити і-й елемент у лінійному списку;
- Внести додатковий елемент до або після вказаного вузла;
- Вилучити певний елемент списку;
- Впорядкувати вузли лінійного списку в певному порядку.

У реальних мовах програмування не існує якої-небудь структури даних для зображення лінійного списку так, щоб усі операції над ним виконувалися в однаковій мірі ефективно. Тому при роботі з лінійними списками важливе значення має подання лінійних списків, які використовуються в програмі, таким чином, щоб була забезпечена максимальна ефективність і за часом виконання програми, і за обсягом потрібної їй пам’яті.

Лінійний список є послідовність об’єктів. Позиція елемента в списку має інший тип даних, відмінний від типу даних елемента списку, і цей тип залежить від конкретної фізичної реалізації.

Над лінійним списком допустимі наступні операції.

Операція вставки – вставляє елемент в конкретну позицію в списку, переміщуючи елементи від цієї позиції і далі в наступну, більш вищу позицію.

Операція локалізації – повертає позицію об’єкта в списку. Якщо в списку об’єкт зустрічається декілька разів, то повертається позиція першого від початку

списку об’єкта. Якщо об’єкта немає в списку, то повертається значення, яке рівне довжині списку, збільшене на одиницю.

Операція вибірки елемента з списку – повертає елемент, який знаходиться в конкретній позиції списку. Результат не визначений, якщо в списку немає такої позиції.

Операція вилучення – вилучає елемент в конкретній позиції зі списку. Результат невизначений, якщо в списку немає вказаної позиції.

Операції вибірки попереднього і наступного елемента – повертають відповідно наступній і попередній елемент списку відносно конкретної позиці в списку.

Функція очистки списку робить список пустим.

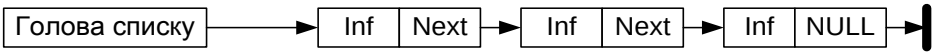
Основні методи зберігання лінійних списків поділяються на методи послідовного і зв’язного зберігання. При виборі способу зберігання в конкретній програмі слід враховувати, які операції і з якою частотою будуть виконуватися над лінійними списками, вартість їх виконання та обсяг потрібної пам’яті для зберігання списку.

Найпростіша форма представлення лінійного списку — це вектор. Визначивши таким чином список можна по чергово звертатися до них в циклі і виконувати необхідні дії. Однак при такому представленні лінійного списку не вдасться уникнути фізичного переміщення елементів, якщо потрібно добавляти нові елементи, або вилучати існуючі. Набагато швидше вилучати елементи можна за допомогою простої схеми чистки пам’яті. Замість вилучення елементів із списку, їх помічають як невикористані.

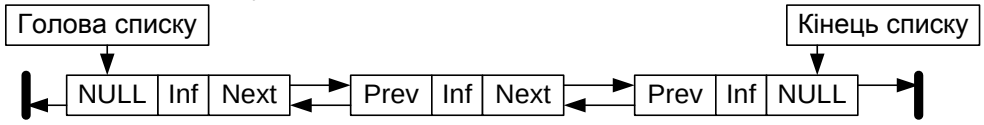
Більш складною організацією при роботі зі списками є розміщення в масиві декількох списків або розміщення списку без прив’язки його початку до першого елемента масиву.

При зв’язному представленні лінійного списку кожен його елемент складається із значення і покажчика, який вказує на наступний елемент у списку.

На наступному рисунку приведена структура однозв’язного списку. Кожний список повинен мати особливий елемент, який називається покажчиком на початок списку, або головою списку.



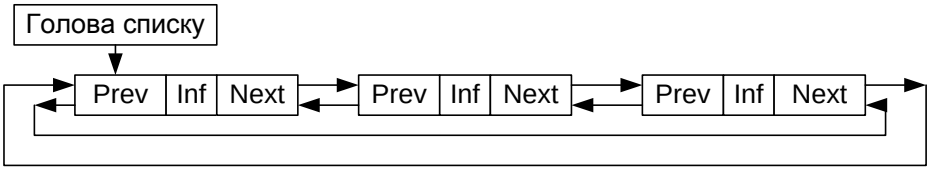
Проте, обробка однозв’язного списку не завжди зручна, оскільки відсутня можливість просування в протилежну сторону. Таку можливість забезпечує двох-зв’язний список, кожний елемент якого містить два покажчики: на наступний і попередній елементи списку.



Для зручності обробки списку додають ще один особливий елемент – покажчик кінця списку. Наявність двох покажчиків в кожному елементі ускладнює список і приводить до додаткових витрат пам’яті, але в той же час забезпечує більш ефективне виконання деяких операцій над списком.

Різновидом розглянутих видів лінійних списків є кільцевий список, який може бути організований на основі як однозв’язного, так і двох-зв’язного списків. При

цьому в однозв'язному списку покажчик останнього елемента повинен вказувати на перший елемент; в двох-зв'язному списку в першому і останньому елементах відповідні покажчики змінюються.

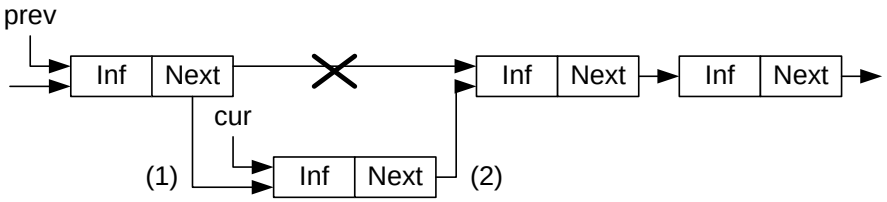


При роботі з такими списками дещо спрощуються деякі процедури, проте, при перегляді такого списку слід приймати деякі запобіжні засоби, щоб не потрапити в нескінченний цикл.

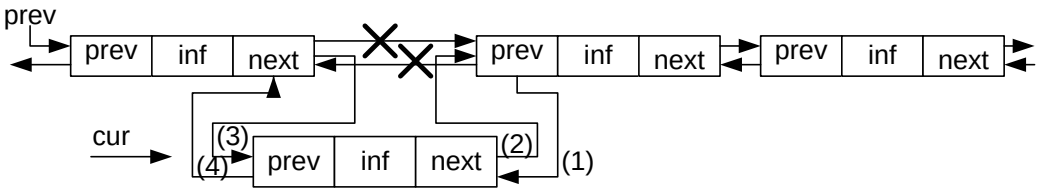
В пам'яті список є сукупністю опису однакових за розміром і форматом структур, які розміщені довільно в деякій ділянці пам'яті і пов'язані одна з одною в лінійно впорядкований ланцюжок за допомогою покажчиків. Структура містить інформаційні поля і поля покажчиків на сусідні елементи списку, причому деякими полями інформаційної частини можуть бути покажчики на блоки пам'яті з додатковою інформацією, що відноситься до елемента списку.

Розглянемо деякі прості операції над лінійними списками.

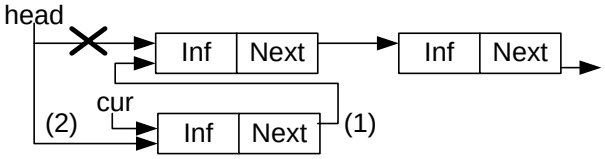
Вставка елемента в середину однозв'язного списку:



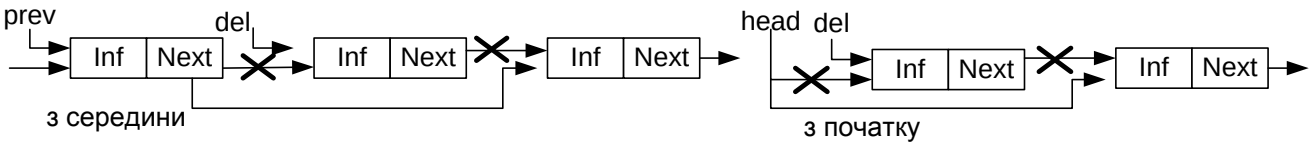
Вставка елемента в двох-зв'язний список:



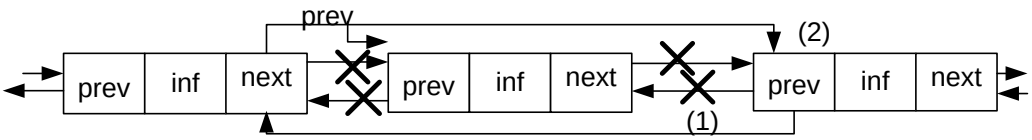
Наведені приклади забезпечують вставку в середину списку, але не можуть бути застосовані для вставки на початок списку. При такій операції повинен модифікуватися покажчик на початок списку:



Видалення елемента з однозв'язного списку для двох варіантів – з середини і з голови:

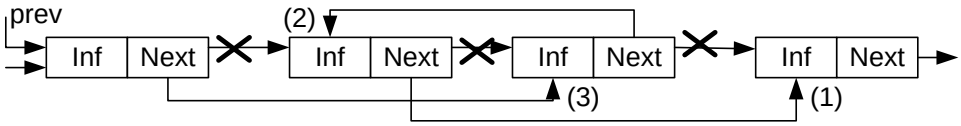


Видалення елемента з двох-зв'язного списку вимагає корекції більшої кількості покажчиків:



Процедура видалення елемента з двох-зв’язного списку виявиться навіть простішою, ніж для однозв’язного, оскільки в ній не потрібно шукати попередній елемент, він вибирається за покажчиком назад.

Змінність динамічних структур даних допускає не тільки зміни розміру структури, але і зміни зв’язків між елементами. Для зв’язних структур зміна зв’язків не вимагає пересилки даних в пам’яті, а тільки зміни покажчиків в елементах зв’язної структури. В якості прикладу приведена перестановка двох сусідніх елементів списку. В алгоритмі перестановки в однозв’язному списку виходили з того, що відома адреса елемента, який передує парі, в якій проводиться перестановка. В приведеному алгоритмі також не враховується випадок перестановки початкових елементів списку.



У процедурі перестановки для двох-зв’язного списку неважко врахувати і перестановку на початку списку.

4.6. Мультисписки

В програмних системах, які обробляють об’єкти складної структури, можуть вирішуватися різні підзадачі, кожна з яких вимагає обробки можливо не всієї множини об’єктів, а лише якоїсь його підмножини.

Для того, щоб при вибірці кожної підмножини не виконувати повний перегляд з відсіванням записів, які до необхідної підмножини не відносяться, в кожний запис включаються додаткові поля посилань, кожне з яких зв’язує в лінійний список елементи відповідної підмножини. В результаті виходить багато-зв’язковий список або мультисписок, кожний елемент якого може входити одночасно в декілька однозв’язних списків.

До переваг мультисписків крім економії пам’яті (при множині списків інформаційна частина існує в єдиному екземплярі) слід віднести також цілісність даних – в тому сенсі, що всі підзадачі працюють з однією і тією ж версією інформаційної частини і зміни в даних, зроблені одній підзадачей негайно стають доступними для іншої підзадачі.

Кожна підзадача працює з своєю підмножиною як з лінійним списком, використовуючи для цього певне поле зв’язку. Специфіка мультисписку виявляється тільки в операції виключення елемента із списку. Виключення елемента з якого-небудь одного списку ще не означає необхідності видалення елемента з пам’яті, оскільки елемент може залишатися у складі інших списків. Пам’ять повинна звільнятися тільки у тому випадку, коли елемент вже не входить ні в один з приватних списків мультисписку. Звичайно задача видалення спрощується тим, що один з приватних списків є головним – в нього обов’язково входять всі наявні елементи. Тоді виключення елемента з будь-якого неголовного списку полягає

тільки в зміні покажчиків, але не в звільненні пам’яті. Виключення ж з головного списку вимагає не тільки звільнення пам’яті, але і зміні покажчиків як в головному списку, так і у всіх неголовних списках, в які елемент, що видаляється, входив.

4.7. Стрічки

Стрічка – це лінійно впорядкована послідовність символів, які належать до скінченої множини символів, яка називається алфавітом.

Стрічки мають наступні важливі властивості:

- їхня довжина, як правило, змінна, хоч алфавіт фіксований;
- звичайне звернення до символів стрічки йде з будь-якого одного боку послідовності (важлива впорядкованість послідовності, а не її індексація);
- метою доступу до стрічки є на окремий її елемент, а ланцюжок символів.

Кажучи про стрічки, звичайно мають на увазі текстові стрічки – стрічки, що складаються з символів, які входять в алфавіт якої-небудь вибраної мови, цифр, розділових знаків і інших службових символів. Текстова стрічка є найбільш універсальною формою представлення будь-якої інформації.

Хоча стрічки й розглядаються в частині, яка присвячена напівстатичним структурам даних, в тих або інших конкретних задачах змінність стрічок може варіюватися від повної її відсутності до практично необмежених можливостей зміни. Орієнтація на ту чи іншу міру мінливості стрічок визначає і фізичне представлення їх в пам’яті і особливості виконання операцій над ними. В більшості мов програмування стрічки представляються саме як напівстатичні структури.

Базовими операціями над стрічками є:

- визначення довжини стрічки;
- присвоєння стрічки;
- конкатенація (зчеплення) стрічок;
- виділення підстрічки;
- пошук входження.

Операція визначення довжини стрічки має вид функції, яка повертає значення – ціле число – поточна кількість символів в стрічці. Операція присвоєння має такий же сенс, що і для інших типів даних.

Операція порівняння стрічок має такий же сенс, що і для інших типів даних. Порівняння стрічок проводиться за наступними правилами. Порівнюються перші символи двох стрічок. Якщо символи не рівні, то стрічка, що містить символ, місце якого в алфавіті ближче до початку, вважається меншою. Якщо символи рівні, порівнюються другі, треті і т.д. символи. При досягненні кінця в одній з стрічок стрічка меншої довжини вважається меншою. При рівності довжин стрічок і попарній рівності всіх символів в них стрічки вважаються рівними.

Результатом операції зчеплення двох стрічок є стрічка, довжина якої рівна сумарній довжині стрічок-операндів, а значення відповідає значенню першого операнда, за яким безпосередньо слідує значення другого операнда.

Операція виділення підстрічки виділяє з початкової стрічки послідовність символів, починаючи із заданої позиції, із заданою довжиною.

Операція пошуку входження знаходить місце першого входження підстрічки еталону в початкову стрічку. Результатом операції може бути номер позиції в

початковій стрічці, з яким починається входження еталону або покажчик на початок входження. У разі відсутності входження результатом операції повинне бути деяке спеціальне значення, наприклад, від'ємний номер позиції або порожній покажчик.

Найпростішим способом є представлення стрічки у вигляді вектора постійної довжини. При цьому в пам'яті відводиться фіксована кількість байт, в які записуються символи стрічки. Якщо стрічка менша відведеного під неї вектора, то зайві місця заповнюються пропусками, а якщо стрічка виходить за межі вектора, то зайві (праві) символи повинні бути відкинуті.

Можливе представлення стрічки вектором змінної довжини з ознакою завершення. Цей і всі подальші за ним методи враховують змінну довжину стрічок. Ознака завершення – це особливий символ, який належить до алфавіту (таким чином, корисний алфавіт виявляється меншим на один символ), і займає ту ж кількість розрядів, що і всі інші символи. Витрати пам'яті при цьому способі складають 1 символ на рядок.

Окрім ознаки завершення можна використати лічильник символів – це ціле число, і для нього відводиться достатня кількість бітів, щоб їх з надлишком вистачало для представлення довжини найдовшої стрічки, яку можна представити. При використовуванні лічильника символів можливий довільний доступ до символів в межах стрічки.

Представлення стрічок списком у пам'яті забезпечує гнучкість у виконанні різноманітних операцій над ними (зокрема, операцій включення і виключення окремих символів і цілих ланцюжків) і використовування системних засобів управління пам'яттю при виділенні необхідного об'єму пам'яті для стрічки. Проте, при цьому виникають додаткові затрати пам'яті. Іншим недоліком такого представлення стрічок є те, що логічно сусідні елементи стрічки не є фізично сусідніми в пам'яті. Це ускладнює доступ до груп елементів стрічки в порівнянні з доступом у векторному представленні.

При представленні стрічки однозв'язним лінійним списком кожний символ стрічки представляється у вигляді елемента зв'язного списку; елемент містить код символу і покажчик на наступний елемент. Одностороннє зчеплення представляє доступ тільки в одному напрямі уздовж стрічки.

При використанні двох-зв'язних лінійних списків у кожний елемент списку додається також покажчик на попередній елемент. Двостороннє зчеплення допускає двосторонній рух уздовж списку, що може значно підвищити ефективність виконання деяких стрічкових операцій.

Блочно-зв'язне представлення стрічок дозволяє в більшості операцій уникнути витрат, які пов'язані з управлінням динамічною пам'яттю, але в той же час забезпечує достатньо ефективне використовування пам'яті при роботі з стрічками змінної довжини.

5. ДИНАМІЧНІ СТРУКТУРИ ДАНИХ

5.1. Зв'язне представлення даних в пам'яті

Динамічні структури за визначенням характеризуються відсутністю фізичної суміжності елементів структури в пам'яті, непостійністю і непередбачуваністю розміру (кількість елементів) структури в процесі її обробки.

Оскільки елементи динамічної структури розташовуються за не передбачуваними адресами пам’яті, адресу елемента такої структури не можна обчислити за адресою початкового або попереднього елемента. Для встановлення зв’язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв’язки між елементами. Таке представлення даних в пам’яті називається зв’язним. Елемент динамічної структури складається з двох полів:

- інформаційного поля або поля даних, в якому містяться ті дані, заради яких і створюється структура;
- поле зв’язку, в якому міститься один або декілька покажчиків, які зв’язують даний елемент з іншими елементами структури.

Коли зв’язне представлення даних використовується для вирішення прикладної задачі, для кінцевого користувача „видимим” робиться тільки вміст інформаційного поля, а поле зв’язку використовується тільки програмістом-розробником.

Переваги зв’язного представлення даних:

- можливість забезпечення значної змінності структур;
- розмір структури обмежується тільки доступним об’ємом машинної пам’яті;
- при зміні логічної послідовності елементів структури потрібно виконати не переміщення даних в пам’яті, а тільки корекцію покажчиків.

Разом з тим зв’язне представлення не позбавлене й недоліків, основні з яких:

- робота з покажчиками вимагає більш високої кваліфікації від програміста;
- на поля зв’язку витрачається додаткова пам’ять;
- доступ до елементів зв’язної структури може бути менш ефективним за часом.

Останній недолік є найбільш серйозним і саме ним обмежується застосування зв’язного представлення даних. Якщо в суміжному представленні даних для обчислення адреси будь-якого елемента у всіх випадках достатньо номера елемента і інформації, яка міститься в описі структури, то для зв’язного представлення адреса елемента не може бути обчислена з початкових даних. Опис зв’язної структури містить один або декілька покажчиків, які дозволяють увійти до структури, далі пошук необхідного елемента виконується проходженням ланцюжком покажчиків від елемента до елемента. Тому зв’язне представлення практично ніколи не застосовується в задачах, де логічна структура даних має вигляд вектора або масиву – з доступом за номером елемента, але часто застосовується в задачах, де логічна структура вимагає іншої початкової інформації доступу (таблиці, списки, дерева і т.д.).

6. НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ

6.1. Графи

Граф – це складна нелінійна багато-зв’язна динамічна структура, що відображає властивості і зв’язки складного об’єкту.

Ця багато-зв’язна структура має наступні властивості:

- на кожний елемент (вузол, вершину) може бути довільна кількість посилань;
- кожний елемент може мати зв’язок з будь-якою кількістю інших елементів;

- кожний зв’язок (ребро, дуга) може мати напрям і вагу.

У вузлах графа міститься інформація про елементи об’єкту. Зв’язки між вузлами задаються ребрами графа. Ребра графа можуть мати спрямованість, тоді вони називаються орієнтованими, в іншому випадку – неорієнтовані. Граф, усі зв’язки якого орієнтовані, називається орієнтованим графом; граф зі всіма неорієнтованими зв’язками – неорієнтованим графом; граф із зв’язками обох типів – змішаним графом.

Існує два основні методи представлення графів в пам’яті комп’ютера: матричний і зв’язними нелінійними списками. Вибір методу представлення залежить від природи даних і операцій, що виконуються над ними. Якщо задача вимагає великої кількості включень і виключень вузлів, то доцільно представляти граф зв’язними списками; інакше можна застосувати і матричне представлення.

При використанні матриць суміжності їхні елементи представляються в пам’яті комп’ютера елементами масиву. При цьому, для простого графа матриця складається з нулів і одиниць, для мультиграфа – з нулів і цілих чисел, які вказують кратність відповідних ребер, для зваженого графа – з нулів і дійсних чисел, які задають вагу кожного ребра.

Орієнтований граф представляється зв’язним нелінійним списком, якщо він часто змінюється або якщо півміри входу і виходу його вузлів великі.

Багато-зв’язна структура – граф – знаходить широке застосування при організації банків даних, управлінні базами даних, в системах програмного імітаційного моделювання складних комплексів, в системах штучного інтелекту, в задачах планування і в інших сферах.

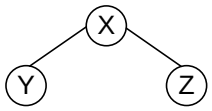
6.2. Дерева

Дерево – це граф, який характеризується наступними властивостями:

- Існує єдиний елемент (вузол або вершина), на який не посилається ніякий інший елемент, – він називається коренем.
- Починаючи з кореня і слідуючи по певному ланцюжку покажчиків, що містяться в елементах, можна здійснити доступ до будь-якого елемента структури.
- На кожний елемент, крім кореня, є єдине посилання, тобто кожний елемент адресується єдиним покажчиком.

Назва „дерево” виникла з логічної еквівалентності дерево-видної структури абстрактному дереву з теорії графів. Лінія зв’язку між парою вузлів дерева називається гілкою. Ті вузли, які не посилаються ні на які інші вузли дерева, називаються листям. Вузол, що не є листком або коренем, вважається проміжним або вузлом галуження.

В багатьох застосування відносний порядок проходження вершин на кожному окремому ярусі має певне значення. При представленні дерева в пам’яті комп’ютера такий порядок вводиться автоматично, навіть якщо він сам по собі довільний. Порядок проходження вершин на деякому ярусі можна легко ввести, позначаючи одну вершину як першу, іншу – як другу і т.д. Замість впорядковування вершин можна задавати порядок на ребрах. Якщо в орієнтованому дереві на кожному ярусі заданий порядок проходження вершин, то таке дерево називається впорядкованим деревом.



Введемо ще деякі поняття, пов'язані з деревами. Вузол X називається предком, або батьком, а вузли Y і Z називаються нащадками, або синами, їх, відповідно, між собою називають братами. Причому, лівий син є старшим сином, а правий – молодшим. Кількість піддерев даної вершини називається мірою цієї вершини.

Якщо з дерева прибрати коріння і ребра, що сполучають коріння з вершинами першого ярусу, то вийде деяка множина незв'язаних дерев. Множина незв'язаних дерев називається лісом.

Є ряд способів графічного зображення дерев. Перший спосіб полягає у використовуванні для зображення піддерев відомого методу діаграм Венна, другий – методу дужок, що вкладаються одна в одну, третій спосіб – це спосіб, який використовується при складанні змісту книг. Останній спосіб, що базується на форматі з нумерацією рівнів, схожий з методами, які використовуються в мовах програмування. При застосуванні цього формату кожній вершині приписується числовий номер, який повинен бути менший номерів, приписаних кореневим вершинам приєднаних до неї піддерев.

Повне дерево містить максимально можливу кількість вузлів на кожному рівні, крім нижнього. Повні дерева мають ряд важливих властивостей.

По-перше, це найкоротші дерева, які можуть містити задану кількість вузлів. Досить корисна властивість повних дерев полягає в тому, що вони можуть бути дуже компактно записані в масивах. Якщо пронумерувати вузли в „природному" порядку, зверху вниз і зліва направо, то можна помістити елементи дерева в масив у цьому ж порядку.

Існують *m*-арні дерева, тобто такі дерева, в яких півміра виходу кожної вершини менша або рівна *m*. Якщо півміра виходу кожної вершини в точності рівна або *m*, або нулю, то таке дерево називається повним m-арним деревом. При *m=2* такі дерева називаються відповідно бінарними, або повними бінарними.

Представити *m*-арне дерево в пам'яті комп'ютера складно, так як кожен елемент дерева повинен містити стільки покажчиків, скільки ребер, виходить з вузла. Це приведе до підвищеної витрати пам'яті, різноманітності початкових елементів і ускладнить алгоритми обробки дерева. Тому *m*-арні дерева, ліс необхідно привести до бінарних для економії пам'яті і спрощенню алгоритмів. Усі вузли бінарного дерева представляються в пам'яті однотипними елементами з двома покажчиками, крім того, операції над бінарними деревами виконуються просто і ефективно.

Правило побудови бінарного дерева з будь-якого дерева:

1. В кожному вузлі залишити тільки гілку до старшого сина;

2. З'єднати горизонтальними ребрами всіх братів одного батька;

3. Таким чином перебудувати дерево за правилом:

лівий син – вершина, розташована під даною;

правий син – вершина, розташована праворуч від даної (тобто на одному ярусі з нею).

4. Розвернути дерево так, щоб усі вертикальні гілки відображали лівих синів, а горизонтальні – правих.

У результаті перетворення будь-якого дерева, в бінарне, виходить дерево у вигляді лівого піддерева, підвішеного до кореня.

У процесі перетворення правий покажчик кожного вузла бінарного дерева буде вказувати на сусіда по рівню. Якщо такого немає, то правий покажчик – **NULL**. Лівий покажчик буде вказувати на вершину наступного рівня. Якщо такої немає, то покажчик встановлюється на **NULL**.

Описаний вище метод представлення довільних впорядкованих дерев за допомогою бінарних дерев можна узагальнити на представлення довільного впорядкованого лісу.

Правило побудови бінарного дерева з лісу: корені всіх піддерев лісу з'єднати горизонтальними зв'язками. В отриманому дереві вузли в даному прикладі будуть розташовуватися на трьох рівнях. Далі перебудовувати по раніше розглянутому плану. В результаті перетворення впорядкованого лісу в бінарне дерево виходить повне бінарне дерево з лівим і правим піддеревом.

Дерева можна представляти за допомогою зв'язних списків і масивів (або послідовних списків).

Частіше всього використовується зв'язне представлення дерев, так як воно дуже сильно нагадує логічне. Зв'язне зберігання полягає в тому, що задається зв'язок від батька до синів. В бінарному дереві є два покажчики, тому зручно вузол представити у вигляді структури в якій *left* – покажчик на ліве піддерево, *right* – покажчик на праве піддерево, *inf* – містить інформацію, яка зв'язана з вершиною і має наперед визначений тип – *data*.

Над деревами визначені наступні основні операції:

1) Пошук вузла із заданим ключем.

2) Додавання нового вузла.

3) Видалення вузла (піддерева).

4) Обхід дерева в певному порядку:

Низхідний обхід;

Змішаний обхід;

Висхідний обхід.

Потрібна вершина в дереві шукається за ключем. Пошук в бінарному дереві здійснюється таким чином.

Нехай побудовано деяке дерево і вимагається знайти вузол з ключем X. Спочатку порівнюємо з X ключ, що знаходиться в корені дерева. У разі рівності пошук закінчений і потрібно повернути покажчик на корінь в якості результату пошуку. Інакше переходимо до розгляду вершини, яка знаходиться зліва внизу, якщо ключ X менший тільки що розглянутого, або справа внизу, якщо ключ X більший тільки що розглянутого. Порівнюємо ключ X з ключем, що міститься в цій вершині, і т.д. Процес завершується в одному з двох випадків:

1) знайдена вершина, що містить ключ, рівний ключу X;

2) в дереві відсутня вершина, до якої потрібно перейти для виконання чергового кроку пошуку.

В першому випадку повертається покажчик на знайдену вершину. В другому – покажчик на вузол, де зупинився пошук (що зручне для побудови дерева).

Для включення запису в дерево перш за все потрібно знайти в дереві ту вершину, до якої можна приєднати нову вершину, відповідну запису, що включається. При цьому впорядкованість ключів повинна зберігатися.

Алгоритм пошуку потрібної вершини, взагалі кажучи, той же самий, що і при пошуку вершини із заданим ключем. Ця вершина буде знайдена в той момент, коли в якості чергового покажчика, який визначає гілку дерева, в якій треба продовжити пошук, виявиться покажчик **NULL.**

В багатьох задачах, пов'язаних з деревами, вимагається здійснити систематичний перегляд всіх його вузлів в певному порядку. Такий перегляд називається проходженням або обходом дерева.

Бінарне дерево можна обходити трьома основними способами: низхідним, змішаним і висхідним (можливі також зворотний низхідний, зворотний змішаний і зворотний висхідний обходи). Прийняті назви методів обходу зв'язані з часом обробки кореневої вершини: До того як оброблено обидва його піддерева, після того, як оброблено ліве піддерево, але до того як оброблено праве, після того, як оброблено обидва піддерева. Використовувані назви методів відображають напрям обходу в дереві: від кореневої вершини вниз до листя – низхідний обхід; від листя вгору до кореня – висхідний обхід, і змішаний обхід – від найлівішого листка дерева через корінь до найправішого листка.

Схемно алгоритм обходу бінарного дерева відповідно до низхідного способу може виглядати таким чином:

1. В якості чергової вершини взяти корінь дерева. Перейти до пункту 2.

2. Провести обробку чергової вершини відповідно до вимог задачі. Перейти до пункту 3.

3.а) Якщо чергова вершина має обидві гілки, то в якості нової вершини вибрати ту вершину, на яку посилається ліва гілка, а вершину, на яку посилається права гілка, занести в стек; перейти до пункту 2;

3.б) якщо чергова вершина є кінцевою, то вибрати в якості нової чергової вершини вершину із стека, якщо він не порожній, і перейти до пункту 2; якщо ж стек порожній, то це означає, що обхід всього дерева закінчений, перейти до пункту 4;

3.в) якщо чергова вершина має тільки одну гілку, то в якості чергової вершини вибрати ту вершину, на яку ця гілка вказує, перейти до пункту 2.

4. Кінець алгоритму.

Алгоритм істотно спрощується при використовуванні рекурсії. Так, низхідний обхід можна описати таким чином:

1). Обробка кореневої вершини;

2). Низхідний обхід лівого піддерева;

3). Низхідний обхід правого піддерева.

Змішаний обхід можна описати таким чином:

1) Спуститися по лівій гілці із запам'ятовуванням вершин в стеку;

2) Якщо стек порожній те перейти до п.5;

3) Вибрати вершину із стеку і обробити дані вершини;

4) Якщо вершина має правого сина, то перейти до нього; перейти до п.1.

5) Кінець алгоритму.

Рекурсивний змішаний обхід описується таким чином:

1) Змішаний обхід лівого піддерева;

2) Обробка кореневої вершини;

3) Змішаний обхід правого піддерева.

Трудність реалізації висхідного обходу полягає в тому, що на відміну від попереднього методу в цьому алгоритмі кожна вершина запам'ятовується в стеку двічі: вперше – коли обходиться ліве піддерево, і другий раз – коли обходиться праве піддерево. Таким чином, в алгоритмі необхідно розрізняти два види стекових записів: 1-й означає, що в даний момент обходиться ліве піддерево; 2-й – що обходиться праве, тому в стеку запам'ятовується покажчик на вузол і ознаку (код-1 і код-2 відповідно).

Алгоритм висхідного обходу можна представити таким чином:

1) Спуститися по лівій гілці із запам'ятовуванням вершини в стеку як 1-й вид стекових записів;

2) Якщо стек порожній, то перейти до п.5;

3) Вибрати вершину із стека, якщо це перший вид стекових записів, то повернути його в стек як 2-й вид стекових записів; перейти до правого сина; перейти до п.1, інакше перейти до п.4;

4) Обробити дані вершини і перейти до п.2;

5) Кінець алгоритму.

Рекурсивний змішаний обхід описується таким чином:

1). Висхідний обхід лівого піддерева;

2). Висхідний обхід правого піддерева;

3). Обробка кореневої вершини.

Якщо в розглянутих вище алгоритмах поміняти місцями поля покажчики на лівого і правого сина, то отримують процедури зворотного низхідного, зворотного змішаного і зворотного висхідного обходів.

7. ПОБУДОВА І АНАЛІЗ АЛГОРИТМІВ

7.1. Формалізація алгоритмів.

Процес створення комп'ютерної програми для вирішення будь-якої практичної задачі складається з декількох етапів:

- формалізація і створення технічного завдання на вихідну задачу;
- розробка алгоритму вирішення задачі;
- написання, тестування, наладка і документування програми;
- отримання розв'язку вихідної задачі шляхом виконання програми.

Половина справи зроблена, якщо знати, що поставлена задача має вирішення. В першому наближенні більшість задач, які зустрічаються на практиці, не мають чіткого й однозначного опису. Певні задачі взагалі неможливо сформулювати в термінах, які допускають комп'ютерне вирішення. Навіть якщо допустити, що задача може бути вирішена на комп'ютері, часто для її формального опису потрібна велика кількість різноманітних параметрів. І лише в ході додаткових експериментів можна знайти інтервали зміни цих параметрів.

Якщо певні аспекти вирішуваної задачі можна виразити в термінах якої-небудь формальної моделі, то це, безумовно, необхідно зробити, так як в цьому

випадку в рамках цієї моделі можна взнати, чи існують методи й алгоритми вирішення задачі. Навіть якщо такі методи й алгоритми не існують на сьогоднішній день, то застосування засобів і властивостей формальної моделі допоможе в побудові вирішення вихідної задачі.

Практично будь-яку галузь математики або інших наук можна застосувати до побудови моделі певного класу задач. Для задач, числових за своєю природою, можна побудувати моделі на основі загальних математичних конструкцій, таких як системи лінійних рівнянь, диференціальні рівняння. Для задач з символьними або текстовими даними можна застосувати моделі символьних послідовностей або формальних граматик. Вирішення таких задач містить етапи компіляції і інформаційного пошуку.

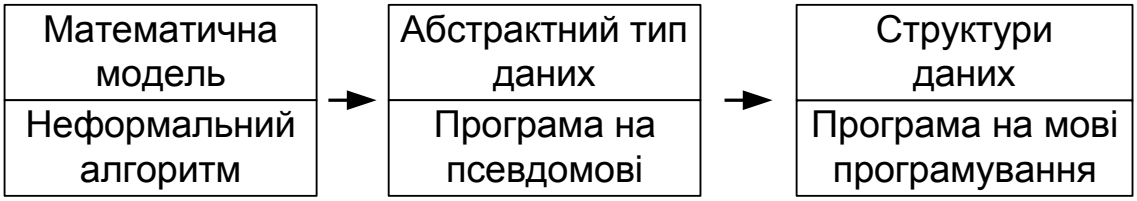
Коли побудована чи підібрана потрібна модель вихідної задачі, то природно шукати її вирішення в термінах цієї моделі. На цьому етапі основна мета полягає в побудові розв'язку в формі алгоритму, який складається з скінченої послідовності інструкцій, кожна з яких має чіткий зміст і може бути виконана з скінченими обчислювальними затратами за скінчений час. Інструкції можуть виконуватися в алгоритмі будь-яку кількість раз, при цьому вони самі визначають цю кількість повторень. Проте вимагається, щоб при будь-яких вхідних даних алгоритм завершився після виконання скінченої кількості інструкцій. Таким чином, програма, яка написана на основі розробленого алгоритму, при будь-яких початкових даних ніколи не повинна приводити до нескінченних циклічних обчислень.

Є ще один аспект у визначення алгоритмів. Алгоритмічні інструкції повинні мати „чіткий зміст" і виконуватися з „скінченими обчислювальними затратами". Природно, те, що зрозуміло одній людині і має для неї „чіткий зміст", може зовсім інакше представлятися іншій. Те ж саме можна сказати про поняття „скінчених затрат": на практиці часто важко довести, що при будь-яких вихідних даних виконання послідовності інструкцій завершиться, навіть якщо чітко розуміти зміст кожної інструкції. У цій ситуації, враховуючи всі аргументи за і проти, було б корисним спробувати досягнути узгодження про „скінченні затрати" у відношенні до послідовності інструкцій, які складають алгоритм.

7.2. Покрокове проектування алгоритмів.

Оскільки для вирішення вихідної задачі застосовують деяку математичну модель, то тим самим можна формалізувати алгоритм вирішення в термінах цієї моделі. У початкових версіях алгоритму часто застосовуються узагальнені оператори, які потім перевизначаться у вигляді більш дрібних, чітко визначених інструкцій. Але для перетворення неформальних алгоритмів у комп'ютерні програми необхідно пройти через декілька етапів формалізації (цей процес називають покроковою деталізацією), поки не отримають програму, яка повністю складається з формальних операторів мови програмування.

Схематично узагальнений процес програмування можна представити наступною схемою.



На першому етапі створюється модель вихідної задачі, для чого застосовуються відповідні математичні моделі. На цьому етапі для знаходження рішення також будується неформальний алгоритм.

На наступному етапі алгоритм записується на псевдомові – композиції конструкцій мови програмування і менш формальних і узагальнених операторів на простій мові. Продовженням цього етапу є заміна неформальних операторів послідовністю більш детальних і формальних операторів. З цієї точки зору програма на псевдомові повинна бути достатньо детальною, так як в ній фіксуються (визначаються) різні типи даних, над якими виконуються оператори. Потім створюються абстрактні типи даних для кожного зафіксованого типу даних (за винятком елементарних типів даних, таких як цілі й дійсні числа або символьні стрічки) шляхом завдання імен функцій для кожного оператора, який працює з даними абстрактного типа, і заміни їх (операторів) викликом відповідних функцій.

Третій етап процесу – програмування – забезпечує реалізацію кожного абстрактного типа даних і створення функцій для виконання різних операторів над даними цих типів. На цьому етапі також замінюються всі неформальні оператори псевдомови на код мови програмування. Результатом цього етапу повинна бути виконувана програма. Після її наладки отримують працюючу програму.

7.3. Характеристики алгоритму

Охарактеризуємо поняття алгоритму не формально, а дескриптивно за допомогою таблиці:

		Вирішення гарантоване	
		Так	Ні
Чи обов'язкове оптимальне вирішення	Так	Алгоритм	Імовірнісний алгоритм
	Ні	Приблизний алгоритм	Евристичний алгоритм

Як бачимо, алгоритм – це механізм, який не тільки повинен гарантувати те, що вирішення колись буде знайдене, але й те, що буде знайдене саме оптимальне, тобто найкраще вирішення. Крім того, алгоритм повинен мати наступні п'ять якостей:

1. Обмеженість в часі – робота алгоритму обов'язково повинна завершитись через деякий розумний період часу.
2. Правильність – алгоритм повинен знаходити правильне рішення.
3. Детермінованість – скільки б разів не виконувався алгоритм з однаковими вхідними даними, результат повинен бути однаковим.
4. Скінченність – опис алгоритму повинен мати скінчену кількість кроків.
5. Однозначність – кожний крок алгоритму повинен інтерпретуватися однозначно.

Як видно з таблиці, евристика – це пряма протилежність класичному алгоритму, так як вона не дає ніяких гарантій на те, що рішення буде знайдене, так само, як і на те, що воно буде оптимальним. Між ними є два перехідні стани – приблизні алгоритми (рішення гарантоване, але його оптимальність – ні) і

імовірнісні алгоритми (рішення не гарантоване, але якщо воно буде знайдене, то обов'язково буде оптимальним).

7.4. Складність алгоритму

У процесі вирішення прикладних задач вибір потрібного алгоритму викликає певні труднощі. І справді, на чому базувати свій вибір, якщо алгоритм повинен задовольняти наступні протиріччя.

1. Бути простим для розуміння, перекладу в програмний код і наладки.
2. Ефективно використовувати комп'ютерні ресурси і виконуватися швидко.

Якщо написана програма повинна виконуватися лише декілька разів, то перша вимога найбільш важлива. Вартість робочого часу програміста, звичайно, значно перевищує вартість машинного часу виконання програми, тому вартість програми оптимізується за вартістю написання (а не виконання) програми. Якщо мати справу з задачею, вирішення якої потребує значних обчислювальних затрат, то вартість виконання програми може перевищити вартість написання програми, особливо якщо програма повинна виконуватися багаторазово. Тому, з економічної точки зору, перевагу буде мати складний комплексний алгоритм (в надії, що результуюча програма буде виконуватися суттєво швидше, ніж більш проста програма). Але і в цій ситуації розумніше спочатку реалізувати простий алгоритм, щоб визначити, як повинна себе вести більш складна програма. При побудові складної програмної системи бажано реалізувати її простий прототип, на якому можна провести необхідні виміри й змоделювати її поведінку в цілому, перш ніж приступати до розробки кінцевого варіанту. Таким чином, програмісти повинні бути обізнані не тільки з методами побудови швидких алгоритмів, але й знати, коли їх потрібно застосувати.

Існує декілька способів оцінки складності алгоритмів. Програмісти, звичайно, зосереджують увагу на швидкості алгоритму, але важливі й інші вимоги, наприклад, до розмірів пам'яті, вільного місця на диску або інших ресурсів. Від швидкого алгоритму може бути мало толку, якщо під нього буде потрібно більше пам'яті, ніж встановлено на комп'ютері.

Важливо розрізняти практичну складність, яка є точною мірою часу обчислення і об'єму пам'яті для конкретної моделі обчислювальної машини, і теоретичну складність, яка більш незалежна від практичних умов виконання алгоритму і дає порядок величини вартості.

Більшість алгоритмів надає вибір між швидкістю виконання і ресурсами. Задача може виконуватися швидше, використовуючи більше пам'яті, або навпаки – повільніше з меншим обсягом пам'яті. Із цього зв'язку випливає ідея просторово-часової складності алгоритмів. При цьому підході складність алгоритму оцінюється в термінах часу і простору, і знаходиться компроміс між ними.

7.5. Ефективність алгоритмів

Одним із способів визначення часової ефективності алгоритмів полягає в наступному: на основі даного алгоритму потрібно написати програму і виміряти час її виконання на певному комп'ютері для вибраної множини вхідних даних. Хоча такий спосіб популярний і, безумовно, корисний, він породжує певні проблеми. Визначений час виконання програми залежить не тільки від використаного

алгоритму, але й від архітектури і набору внутрішніх команд даного комп'ютера, від якості компілятора, і від рівня програміста, який реалізував даний алгоритм. Час виконання також може суттєво залежати від вибраної множини тестових вхідних даних. Ця залежність стає очевидною при реалізації одного й того ж алгоритму з використанням різних комп'ютерів, різних компіляторів, при залучені програмістів різного рівня і при використанні різних тестових даних. Щоб підвищити об'єктивність оцінки алгоритмів прийняли **асимптотичну часову складність** як основну міру ефективності виконання алгоритму.

Говорять, що час виконання алгоритму має порядок *T(N)* від вхідних даних розміру *N*. Одиниця вимірювання *T(N)* не визначена, але під нею розуміють кількість інструкцій, які виконуються на ідеалізованому комп'ютері.

Для багатьох програм час виконання дійсно є функцією вхідних даних, а не їх розміру. У цьому випадку визначають *T(N)* як час виконання в найгіршому випадку, тобто, як максимум часів виконання за всіма вхідними даними розміру *N*. Поряд з тим розглядають $T_{ср}(N)$ як середній (в статистичному розумінні) час виконання за всіма вхідними даними розміру *N*. Хоча $T_{ср}(N)$ є достатньо об'єктивною мірою виконання, але часто неможливо передбачити, або обґрунтувати, рівнозначність усіх вхідних даних. На практиці середній час виконання знайти складніше, ніж найгірший час виконання, так як математично це зробити важко і, крім цього, часто не буває простого визначення поняття „середніх" вхідних даних. Тому, в основному, користуються найгіршим часом виконання як міра часової складності алгоритмів.

Продуктивність алгоритму оцінюють за порядком величини. Говорять, що алгоритм має складність порядку $O\left(f\left(N\right)\right)$, якщо час виконання алгоритму росте пропорційно функції $f\left(N\right)$ із збільшенням розмірності початкових даних N. O – позначає „величина порядку".

Приведемо деякі функції, які часто зустрічаються при оцінці складності алгоритмів. Ефективність степеневих алгоритмів звичайно вважається поганою, лінійних – задовільній, логарифмічних – хорошою.

Функція	Примітка
f(N)=C	C – константа
f(N)=log(log(N))	
f(N)=log(N)	
f(N)=NC	C – константа від нуля до одиниці
f(N)=N	
f(N)=N*log(N)	
$f(N)=N^C$	C – константа більша одиниці
$f(N)=C^N$	C – константа більша одиниці
f(N)=N!	тобто 1*2* … N

Оцінка з точністю до порядку дає верхню межу складності алгоритму. Те, що програма має певний порядок складності, не означає, що алгоритм буде дійсно виконуватися так довго. При певних вхідних даних, багато алгоритмів виконується набагато швидше, ніж можна припустити на підставі їхнього порядку складності.

У числових алгоритмах точність і стійкість алгоритмів не менш важлива, ніж їх часова ефективність.

7.6. Правила аналізу складності алгоритмів.

У загальному випадку час виконання оператора або групи операторів можна розглядати як функцію з параметрами – розміром вхідних даних і/або одної чи декількох змінних. Але для часу виконання програми в цілому допустимим параметром може бути лише розмір вхідних даних.

Час виконання операторів присвоєння, читання і запису має порядок $O(1)$.

Час виконання послідовності операторів визначається за правилом сум. Тому міра росту часу виконання послідовності операторів без визначення констант пропорційності співпадає з найбільшим часом виконання оператора в даній послідовності.

Час виконання умовних операторів складається з часу виконання умовно виконуваних операторів і часу обчислення самого логічного виразу. Час обчислення логічного виразу часто має порядок $O(1)$. Час для всієї умовної конструкції складається з часу обчислення логічного виразу і найбільшого з часів, який необхідний для виконання операторів, що виконуються при різних значеннях логічного виразу.

Час виконання циклу є сумою часів усіх часів виконуваних конструкцій циклу, які в свою чергу складаються з часів виконання операторів тіла циклу і часу обчислення умови завершення циклу. Часто час виконання циклу обчислюється, нехтуючи визначенням констант пропорційності, як добуток кількості виконуваних операцій циклу на найбільший можливий час виконання тіла циклу. Час виконання кожного циклу, якщо в програмі їх декілька, повинен визначатися окремо.

8. АЛГОРИТМИ СОРТУВАННЯ

8.1. Задача сортування

Для самого загального випадку задачу сортування формулюється: є деяка неврегульована вхідна множина ключів і потрібно отримати множину цих же ключів, впорядкованих за збільшенням або зменшенням.

Зі всіх задач програмування сортування, можливо, має найбагатший вибір алгоритмів розв’язку. Назвемо деякі чинники, які впливають на вибір алгоритму.

1. Наявний ресурс пам’яті: повинні вхідна й вихід множини розташовуватися в різних ділянках пам’яті, чи вихідна множина може бути сформована на місці вхідної. В останньому випадку наявна ділянка пам’яті повинна в ході сортування динамічно перерозподілятися між вхідною і вихідною множинами.
2. Початкова впорядкованість вхідної множини: у вхідній множині можуть попадатися впорядковані ділянки. В граничному випадку вхідна множина може виявитися вже впорядкованою. Одні алгоритми не враховують початкової впорядкованості і вимагають одного і того ж часу для сортування будь-якої множини даного обсягу, інші виконуються тим швидше, чим краще впорядкованість на вході.
3. Часові характеристики операцій: при визначенні порядку алгоритму час виконання вважається звичайно пропорційним кількості порівнянь ключів. Ясно, проте, що порівняння числових ключів виконується швидше, ніж стрічкових, операції пересилки, характерні для деяких алгоритмів,

виконуються тим швидше, ніж менший об’єм записів. Залежно від характеристик запису таблиці може бути вибраний алгоритм, що забезпечує мінімізацію числа тих чи інших операцій.

4. Складність алгоритму. Простий алгоритм вимагає меншого часу для його реалізації і вірогідність помилки в реалізації його менше. При програмуванні вимоги дотримання термінів розробки і надійності продукту можуть навіть превалювати над вимогами ефективності функціонування.

Алгоритм сортування називається усталеним, якщо у відсортованому масиві він не змінює порядку розташування елементів.

Ефективність методів сортування визначається двома параметрами:

- кількістю порівнянь;
- кількістю пересилань елементів.

Різноманітність алгоритмів сортування вимагає деякої їхньої класифікації. Вибраний один з вживаних для класифікації підходів, орієнтований перш за все на логічні характеристики використовуваних алгоритмів. Згідно цьому підходу будь-який алгоритм сортування використовує одну з наступних чотирьох стратегій (або їхню комбінацію).

1. Стратегія вибірки. З вхідної множини вибирається наступний за критерієм впорядкованості елемент і включається в вихідну множину на наступне з місце.
2. Стратегія включення. З вхідної множини вибирається наступний за номером елемент і включається в вихідну множину на те місце, яке він повинен займати відповідно до критерію.
3. Стратегія розподілу. Вхідна множина розбивається на ряд підмножин і сортування ведеться у середині кожної такої підмножини.
4. Стратегія злиття. Вихідна множина отримується шляхом злиття маленьких впорядкованих підмножин.

8.2. Сортування вибіркою

Даний метод реалізує практично „дослівно” стратегію вибірки. При програмній реалізації алгоритму виникає проблема значення ключа „порожньо”. Досить часто програмісти використовують в якості такого деяке явно відсутнє у вхідній послідовності значення ключа. Інший підхід – створення окремого вектора, кожний елемент якого має логічний тип і відображає стан відповідного елемента вхідної множини.

Алгоритм сортування простою вибіркою рідко застосовується. Набагато частіше застосовується його обмінний варіант. При обмінному сортуванні вибіркою вхідна і вихід множини розташовуються в одній і тій же ділянці пам’яті; вихідна – на початку ділянки, вхідна – в тій частині, що залишилася. У початковому стані вхідна множина займає всю ділянку, а вихідна множина – порожня. У міру виконання сортування вхідна множина звужується, а вихідна – розширяється.

Принцип методу полягає в наступному. Знаходять і вибирають в масиві елементів елемент з мінімальним значенням на інтервалі від першого до останнього елемента і міняють його місцями з першим елементом. На другому кроці знаходять

елемент з мінімальним значенням на інтервалі від другого до останнього елемента і міняють місцями його з другим елементом. І так далі для всіх елементів.

Очевидно, що обмінний варіант забезпечує економію пам'яті та при його реалізації не виникає проблема „порожнього" значення. Загальна кількість порівнянь зменшується удвічі – *N*(N-1)/2*, але порядок алгоритму залишається степеневим. Кількість перестановок *N-1*, але перестановка удвічі більше потребує часу, ніж пересилка в попередньому алгоритмі.

Досить проста модифікація алгоритму обмінного сортування вибіркою передбачає пошук в одному циклі перегляду вхідної множини відразу і мінімуму, і максимуму, і обмін їх з першим і з останнім елементами множини відповідно. Хоча сумарна кількість порівнянь і пересилок в цій модифікації не зменшується, досягається економія на кількості ітерацій зовнішнього циклу.

Приведені вище алгоритми сортування вибіркою практично нечутливі до початкової впорядкованості. В будь-якому випадку пошук мінімуму вимагає повного перегляду вхідної множини. В обмінному варіанті початкова впорядкованість може дати деяку економію на перестановках для випадків, коли мінімальний елемент знайдений на першому місці у вхідній множині.

Ще один варіант такого сортування – сортування бульбашкою. При перегляді вхідної множини попарно порівнюються сусідні елементи множини. Якщо порядок їхнього проходження не відповідає заданому критерію впорядкованості, то елементи міняються місцями. В результаті одного такого перегляду при сортуванні за збільшенням елементів елемент з найбільшим значенням ключа переміститься („спливе") на останнє місце в множині. При наступному проході на своє місце „спливе" другий за величиною ключа елемент і т.д. Вихідна множина, таким чином, формується в кінці сортованої послідовності, при кожному наступному проході його об'єм збільшується на *1*, а об'єм вхідної множини зменшується на *1*.

Порядок сортування бульбашкою – $O\left(N^2\right)$. Середнє число порівнянь – *N*(N-1)/2* і таке ж середня кількість перестановок, що значно гірше, ніж для обмінного сортування простим вибором. Проте, та обставина, що тут завжди порівнюються і переміщаються тільки сусідні елементи, робить сортування бульбашкою зручним для обробки зв'язних списків.

Ще одна перевага сортування бульбашкою полягає в тому, що при незначних модифікаціях її можна зробити чутливою до початкової впорядкованості вхідної множини.

Ще одна модифікація сортування бульбашкою носить назву шейкер-сортування. Суть її полягає в тому, що напрями переглядів чергують: за проходом до кінця множини слідує прохід від кінця до початку вхідної множини. При перегляді в прямому напрямку запис з найбільшим ключем ставиться на своє місце в послідовності, при перегляді у зворотному напрямі – запис з самим меншим. Цей алгоритм досить ефективний для задач відновлення впорядкованості, коли початкова послідовність вже була впорядкована, але піддалася не дуже значним змінам. Впорядкованість в послідовності з одиночною зміною буде гарантовано відновлена усього за два проходи.

Сортування Шелла – ще одна модифікація сортування бульбашкою. Суть її полягає в тому, що тут виконується порівняння ключів, віддалених один від одного

на деяку відстань *d*. Початковий розмір *d* звичайно вибирається рівним половині загального розміру сортованої послідовності. Виконується сортування бульбашкою з інтервалом порівняння *d*. Потім величина *d* зменшується удвічі і знов виконується сортування бульбашкою, далі *d* зменшується ще удвічі і т.д. Останнє сортування бульбашкою виконується при *d=1*. Якісний порядок сортування Шелла залишається $O\left(N^2\right)$, середнє ж число порівнянь, визначене емпіричним шляхом, – *N*log2(N)^2*. Прискорення досягається за рахунок того, що виявленні „не на місці" елементи при *d>1*, швидше „спливають" на свої місця.

8.3. Сортування включенням

Цей метод – „дослівна" реалізації стратегії включення. Порядок алгоритму сортування простим включенням – $O\left(N^2\right)$, якщо враховувати тільки операції порівняння. Але сортування вимагає ще й в середньому $N^2/4$ переміщень, що робить її в такому варіанті значне менш ефективною, ніж сортування вибіркою.

Ефективність алгоритму може бути дещо поліпшена при застосуванні не лінійного, а дихотомічного пошуку. Проте, слід мати на увазі, що таке збільшення ефективності може бути досягнуте лише на значній кількості елементів. Так як алгоритм вимагає великої кількості пересилок, при значному обсязі одного запису ефективність може визначатися не кількістю операцій порівняння, а кількістю пересилок.

Реалізація алгоритму обмінного сортування простими вставками відрізняється від базового алгоритму тільки тим, що вхідна і вихідна множина розміщені в одній ділянці пам'яті.

Бульбашкове сортування включенням – це модифікація обмінного варіанту сортування. В цьому методі вхідна і вихід множини знаходяться в одній послідовності, причому вихід – в початковій її частині. В початковому стані можна вважати, що перший елемент послідовності вже належить впорядкованій вихідній множині, інша частина послідовності – невпорядкована. Перший елемент вхідної множини примикає до кінця вихідної множини. На кожному кроці сортування відбувається перерозподіл послідовності: вихідна множина збільшується на один елемент, а вхідна – зменшується. Це відбувається за рахунок того, що перший елемент вхідної множини тепер вважається останнім елементом вихідної. Потім виконується перегляд вихідної множини від кінця до початку з перестановкою сусідніх елементів, які не відповідають критерію впорядкованості. Перегляд припиняється, коли припиняються перестановки. Це приводить до того, що останній елемент вихідної множини „випливає" на своє місце в множині. Оскільки при цьому перестановка приводить до зсуву нового в вихідній множині елемента на одну позицію ліворуч, немає сенсу кожен раз проводити повний обмін між сусідніми елементами – достатньо зсовувати старий елемент праворуч, а новий елемент записати в вихідну множину, коли його місце буде встановлено.

Хоча обмінні алгоритми стратегії включення і дозволяють скоротити число порівнянь за наявності деякої початкової впорядкованості вхідної множини, значна кількість пересилок істотно знижує ефективність цих алгоритмів. Тому алгоритми включення доцільно застосовувати до зв'язних структур даних, коли операція

перестановки елементів структури вимагає не пересилки даних в пам'яті, а виконується способом корекції покажчиків.

Турнірний метод сортування отримав свою назву через схожість з кубковою системою проведення спортивних змагань: учасники змагань розбиваються на пари, в яких розігрується перший тур; з переможців першого туру складаються пари для розиграшу другого туру і т.д. Алгоритм сортування складається з двох етапів. На першому етапі будується дерево: аналогічне схемі розиграшу кубка.

Алгоритм сортування впорядкованим бінарним деревом складається з побудови впорядкованого бінарного дерева і подальшого його обходу. Якщо немає необхідності в побудові всього лінійного впорядкованого списку значень, то немає необхідності і в обході дерева, в цьому випадку застосовується пошук у впорядкованому бінарному дереві. Відзначимо, що порядок алгоритму – *O(N*log2(N))*, але в конкретних випадках все залежить від впорядкованості початкової послідовності, який впливає на ступінь збалансованості дерева і нарешті – на ефективність пошуку.

Заслуговує на увагу модифікація цього алгоритму запропонована Р.Флойдом. Метод сортування за допомогою прямої вибірки базується на повторних пошуках найменшого ключа серед *N* елементів, серед тих що залишилися *N-1* елементів і так далі. Удосконалити такий метод сортування можна залишаючи після кожного проходу більше інформації, ніж просто ідентифікація єдиного мінімального елемента. Наприклад, виконавши *n/2* порівнянь, можна визначити в кожній парі ключів менший. За допомогою *n/4* порівнянь – менший із пари вже вибраних менших і так далі. Провівши *n-1* порівнянь, можна побудувати дерево вибору і ідентифікувати його корінь як потрібний найменший ключ.

Другий етап сортування – спуск вздовж шляху, відміченого найменшим елементом, і виключення його з дерева шляхом заміни або на пустий елемент (дірку) в самому низу, або на елемент із сусідньої гілки в проміжних вершинах. Елемент, який перемістився в корінь дерева, знову буде найменшим (тепер вже другим) ключем, і його можна виключити. Після *n* таких кроків дерево стане пустим і процес сортування завершується.

Звичайно, хотілося б позбавитися дірок, якими в кінцевому рахунку буде заповнене все дерево і які породжують багато непотрібних порівнянь. Крім того, потрібно знайти б таке представлення дерева з *n* елементів, яке потребує лише *n* одиниць пам'яті.

Р. Флойдом був запропонований деякий „лаконічний" спосіб побудови піраміди „на тому ж місці", який використовує функцію зсуву елементів початкового вектора.

Сортування частково впорядкованим бінарним деревом також належить до цієї групи сортування. У бінарному дереві, яке будується при цьому для кожного вузла справедливе наступне твердження: значення ключа, записане у вузлі, менше, ніж ключі його нащадків. Для повністю впорядкованого дерева є вимоги до співвідношення між ключами нащадків. Для даного дерева таких вимог немає, тому таке дерево і називається частково впорядкованим. Крім того, таке дерево повинно бути абсолютно збалансованим. Це означає не тільки те, що довжини шляхів до будь-якого двох листків розрізняються не більш, ніж на *1*, але і те, що при додаванні

нового елемента в дерево перевага завжди віддається лівій гілці, поки це не порушує збалансованість.

Для сортування цим методом потрібно визначити дві операції: вставка в дерево нового елемента і вибірка з дерева мінімального елемента; причому виконання будь-якій з цих операцій не повинне порушувати ні сформульованої вище часткової впорядкованості дерева, ні його збалансованості.

Якщо застосовувати сортування частково впорядкованим деревом для впорядкування вже готової послідовності розміром *N*, то необхідно *N* раз виконати вставку, а потім *N* раз – вибірку. Порядок алгоритму – *O(N*log2(N))*, але середнє значення кількості порівнянь приблизно в 3 рази більше, ніж для турнірного сортування. Але сортування частково впорядкованим деревом має одну істотну перевагу перед всіма іншими алгоритмами – це найзручніший алгоритм для „сортування on-line", коли сортована послідовність не зафіксована до початку сортування, а міняється в процесі роботи і вставки чергують з вибірками. Кожна зміна (додавання елемента) сортованої послідовності вимагає тут не більш, ніж *2*log2(N)* порівнянь і перестановок, в той час, як інші алгоритми вимагають при одиничній зміні нового впорядковування всієї послідовності „за повною програмою".

8.4. Сортування розподілом

Алгоритм порозрядного сортування вимагає представлення ключів сортованої послідовності у вигляді чисел в деякій системі числення *P*. Число проходів сортування рівно максимальному числу значущих цифр в числі – *D*. При кожному проході аналізується значуща цифра в черговому розряді ключа, починаючи з молодшого розряду. Всі ключі з однаковим значенням цієї цифри об'єднуються в одну групу. Ключі в групі розташовуються в порядку їхнього надходження. Після того, як вся початкова послідовність розподілена по групах, групи розташовуються в порядку зростання пов'язаних з групами цифр. Процес повторюється для другої цифри і т.д., поки не будуть вичерпані значущі цифри в ключі. Основа системи числення *P* може бути будь-якою при цьому потрібно *P* груп.

Порядок алгоритму якісно лінійний – *O(N)*, для сортування потрібно *D*N* операцій аналізу цифри. Проте, в такій оцінці порядку не враховується ряд обставин.

По-перше, операція виділення значущої цифри буде простою і швидкою тільки при *P=2*, для інших систем числення ця операція може вимагати значно більше часу, ніж операція порівняння.

По-друге, при оцінці алгоритму не враховуються затрати часу і пам'яті на створення і ведення груп. Розміщення груп в статичній робочій пам'яті вимагає пам'яті для *P*N* елементів, оскільки в граничному випадку всі елементи можуть потрапити в якусь одну групу. Якщо ж формувати групи усередині тієї ж послідовності за принципом обмінних алгоритмів, то виникає необхідність перерозподілу послідовності між групами і всі проблеми і недоліки, властиві алгоритмам включення. Найбільш раціональним є формування груп у вигляді зв'язних списків з динамічним виділенням пам'яті.

Алгоритм швидкого сортування Хоара відноситься до розподільних і забезпечує показники ефективності *O(N*log2(N))* навіть при якнайгіршому початковому розподілі.

Використовується два індекси з початковими значеннями початку і кінця множини відповідно. Ключ початку порівнюється з ключем кінця. Якщо ключі задовольняють критерію впорядкованості, то індекс кінця зменшується на *1* і проводиться наступне порівняння. Якщо ключі не задовольняють критерію, то записи міняються місцями. При цьому індекс кінця фіксується і починає мінятися індекс початку (збільшуватися на *1* після кожного порівняння). Після наступної перестановки фіксується початок і починає змінюватися кінець і т.д. Прохід закінчується, коли індекси стають рівними. Запис, що знаходиться на позиції зустрічі індексів, стоїть на своєму місці в послідовності. Цей запис ділить послідовність на дві підмножини. Всі записи, розташовані ліворуч від неї мають ключі, менші ніж ключ цього запису, всі записи праворуч – більші. Той же самий алгоритм застосовується до лівої підмножини, а потім до правої. Записи підмножини розподіляються на дві менші підмножини і так далі. Розподіл закінчується, коли отримана підмножина буде складатися з єдиного елемента – така підмножина вже є впорядкованою.

8.5. Сортування злиттям

Алгоритми сортування злиттям, як правило, мають порядок *O(N*log2(N))*, але відрізняються від інших алгоритмів більшою складністю і вимагають великої кількості пересилок. Алгоритми злиття застосовуються в основному, як складова частина зовнішнього сортування.

При сортуванні попарним злиттям вхідна множина розглядається, як послідовність підмножин, кожна з яких складається з єдиного елемента і, отже, є вже впорядкованим. На першому проході кожні дві сусідні одноелементних множини зливаються в одну двоелементну впорядковану множину. На другому проході двоелементні множини зливаються в 4-елементні впорядковані множини і т.д. Врешті-решт отримують одну велику впорядковану множину.

Самою найважливішою частиною алгоритму є злиття двох впорядкованих множин. Цю частину алгоритму опишемо більш детально.

1. **Початкові установки**. Визначити довжини першої і другої початкових множин – *l1* і *l2* відповідно. Встановити індекси поточних елементів в початковій множині *i1* і *i2* в 0. Встановити індекс в вихідній множині *j=1*.

2. **Цикл злиття**. Виконувати крок 3 до тих пір, поки *i1<=l1* і *i2<=l2*.

3. **Порівняння**. Порівняти ключ *i1*-го елемента з першої початкової множини з ключем *i2*-го елемента з другої початкової множини. Якщо ключ елемента з 1-ої множини менший, то записати *i1*-тий елемент з 1-ої множини на *j*-те місце в вихідній множині і збільшити *i1* на 1. Інакше – записати *i2*-тий елемент з 2-ої множини на *j*-те місце в вихідній множині і збільшити *i2* на 1. Збільшити *j* на 1.

4. **Виведення залишків**. Якщо *i1<=l1*, то переписати частину 1-ої початкової множини від *i1* до *l1* включно в вихідну множину. Інакше – переписати частину 2-ої початкової множини від *i2* до *l2* включно в вихідну множину.

8.6. Рандомізація

В деяких програмах потрібно виконання операцій, протилежних сортуванню. Отримавши множину елементів, програма повинна розмістити їх у випадковому порядку. Рандомізацію нескладно виконати, використовуючи алгоритм, подібний на сортування вибіркою.

Для кожного розміщення в множині, алгоритм випадковим чином вибирає елемент, який повинен його зайняти з тих, які ще не були розміщені на своєму місці. Потім цей елемент міняється місцями з елементом, який, знаходиться на цій позиції. Так як алгоритм заповнює кожну позицію лише один раз, його складність – *O(N)*.

Нескладно показати, що імовірність того, що елемент виявиться на якій-небудь позиції, рівна *1/N*. Оскільки елемент може виявитися на будь-якій позиції з однаковою імовірністю, цей алгоритм дійсно приводить до випадкового розміщення елементів.

Результат рандомізації залежить від того, наскільки ефективним є генератор випадкових чисел. Для даного алгоритми не важливий початковий порядок розміщення елементів. Якщо необхідно неодноразово рандомізувати множину елементів, немає необхідності її попередньо сортувати.

9. АЛГОРИТМИ ПОШУКУ

Одна з тих дій, які найбільш часто зустрічаються в програмуванні – пошук. Існує декілька основних варіантів пошуку, і для них створено багато різноманітних алгоритмів.

Задача пошуку – відшукати елемент, ключ якого рівний заданому „аргументу пошуку”. Отриманий в результаті цього індекс забезпечує доступ до усіх полів виявленого елемента.

9.1. Послідовний (лінійний) пошук

Найпростішим методом пошуку елемента, який знаходиться в неврегульованому наборі даних, за значенням його ключа є послідовний перегляд кожного елемента набору, який продовжується до тих пір, поки не буде знайдений потрібний елемент. Якщо переглянуто весь набір, і елемент не знайдений – значить, шуканий ключ відсутній в наборі. Цей метод ще називають методом повного перебору.

Для послідовного пошуку в середньому потрібно *N/2* порівнянь. Таким чином, порядок алгоритму – лінійний – *O(N)*. Якщо елемент знайдено, то він знайдений разом з мінімально можливим індексом, тобто це перший з таких елементів. Рівність $i=N$ засвідчує, що елемент відсутній.

Єдина модифікація цього алгоритму, яку можна зробити, – позбавитися перевірки номера елементу масиву в заголовку циклу ($i<N$) за рахунок збільшення масиву на один елемент у кінці, значення якого перед пошуком встановлюють рівним шуканому ключу – *key* – так званий „бар’єр”.

9.2. Бінарний пошук

Алгоритм пошук може бути значно ефективнішим, якщо дані будуть впорядковані.

Іншим, відносно простим, методом доступу до елемента є метод бінарного (дихотомічного) пошуку, який виконується в явно впорядкованій послідовності елементів.

Оскільки шуканий елемент швидше за все знаходиться „десь в середині”, перевіряють саме середній елемент: *a[N/2]==key*? Якщо це так, то знайдено те, що потрібно. Якщо *a[N/2]<key*, то значення *i=N/2* є замалим і шуканий елемент знаходиться „праворуч”, а якщо *a[N/2]>key*, то „ліворуч”, тобто на позиціях *0...i*.

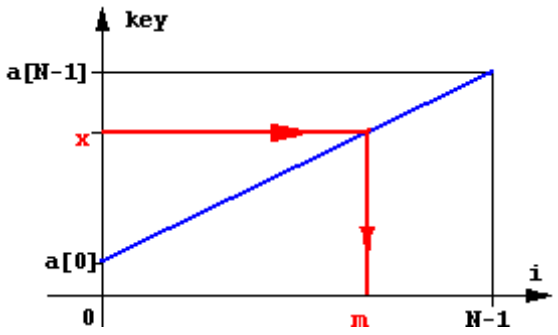
Для того, щоб знайти потрібний запис в таблиці, у гіршому випадку потрібно *log2(N)* порівнянь. Це значно краще, ніж при послідовному пошуку.

Максимальна кількість порівнянь для цього алгоритму рівна *log2(N)*. Таким чином, приведений алгоритм суттєво виграє у порівнянні з лінійним пошуком.

Відомо декілька модифікацій алгоритму бінарного пошуку, які виконуються на деревах.

9.3. Метод інтерполяції

Якщо немає ніякої додаткової інформації про значення ключів, крім факту їхнього впорядкування, то можна припустити, що значення *key* збільшуються від *a[0]* до *a[N-1]* більш-менш „рівномірно”. Це означає, що значення середнього елементу *a[N/2]* буде близьким до середнього арифметичного між найбільшим та найменшим значенням. Але, якщо шукане значення *key* відрізняється від вказаного, то є деякий сенс для перевірки брати не середній елемент, а „середньо-пропорційний”.



Вираз для поточного значення *i* одержано з пропорційності відрізків:

$$\frac{a[e]-key}{key-a[b]}=\frac{e-i}{i-b}$$

В середньому цей алгоритм має працювати швидше за бінарний пошук, але у найгіршому випадку буде працювати набагато довше.

9.4. Метод „золотого перерізу”

Деякий ефект дає використання так званого „золотого перерізу”. Це число φ, що має властивість:

$$\varphi-1=\frac{1}{\varphi};\quad \varphi^2-\varphi-1=0;\quad \varphi_{1,2}=\frac{1\pm\sqrt{5}}{2}.$$

Доданий корінь $\varphi=\frac{1+\sqrt{5}}{2}=1{,}61803398875\ldots$ і є золотим перерізом.

Згідно цього алгоритму відрізок слід ділити не навпіл, як у бінарному алгоритмі, а на відрізки, пропорційні φ та 1, в залежності від того, до якого краю ближче *key*.

9.5. Алгоритми пошуку послідовностей

Даний клас задача відноситься до задачі пошуку слів у тексті. Одним з найпростіших методів пошуку є послідовне порівняння першого символу з символами масиву. Якщо наявний збіг, тоді порівнюються другі, треті,... символи аж до повного збігу рядка *s* з частиною вектору такої ж довжини, або до незбігу у деякому символі. Тоді пошук продовжується з наступного символу масиву та першого символу рядку.

Існує варіант удосконалення цього алгоритму – це починати пошук після часткового збігу не з наступного елементу масиву, а з символу, наступного після тих, що переглядалися, якщо у рядку немає фрагментів, що повторюються.

Д. Кнут, Д. Моріс і В. Пратт винайшли алгоритм, який фактично потребує лише *N* порівнянь навіть в самому поганому випадку. Новий алгоритм базується на тому, що після часткового збігу початкової частини слова з відповідними символами тексту фактично відома пройдена частина тексту і можна „обчислити" деякі відомості (на основі самого слова), за допомогою яких потім можна швидко переміститися текстом.

Основною відмінністю КМП-алгоритму від алгоритму прямого пошуку є здійснення зсуву слова не на один символ на кожному кроці алгоритму, а на деяку змінну кількість символів. Таким чином, перед тим як виконувати черговий зсув, потрібно визначити величину зсуву. Для підвищення ефективності алгоритму необхідно, щоб зсув на кожному кроці був би якомога більшим.

Якщо *j* визначає позицію в слові, в якій міститься перший символ, який не збігається (як в алгоритмі прямого пошуку), то величина зсуву визначається як *j-D*. Значення *D* визначається як розмір самої довшої послідовності символів слова, які безпосередньо передують позиції *j*, яка повністю збігається з початком слова. *D* залежить тільки від слова і не залежить від тексту. Для кожного *j* буде своя величина *D*, яку позначимо d_j.

Так як величини d_j залежать лише від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю *d*; ці обчислення зводяться до деякої попередньої трансляції слова. Відповідні зусилля будуть оправдані, якщо розмір тексту значно перевищує розмір слова (*M*<<*N*). Якщо потрібно шукати багатократні входження одного й того ж слова, то можна користуватися одними й тими ж *d*.

КМП-пошук дає справжній виграш тільки тоді, коли невдачі передувала деяка кількість збігів. Лише у цьому випадку слово зсовується більше ніж на одиницю. На жаль, це швидше виняток, ніж правило: збіги зустрічаються значно рідше, ніж незбіги. Тому виграш від практичного використання КМП-стратегії в більшості випадків пошуку в звичайних текстах досить незначний. Метод, який запропонували Р. Боуєр і Д. Мур в 1975 р., не тільки покращує обробку самого поганого випадку, але й дає виграш в проміжних ситуаціях.

БМ-пошук базується на незвичних міркуваннях – порівняння символів починається з кінця слова, а не з початку. Як і у випадку КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Нехай для кожного символу *x* із алфавіту величина *dx* – відстань від самого правого в слові входження *x* до правого кінця слова. Уявимо, що виявлена розбіжність між словом і текстом. У цьому випадку слово відразу ж можна зсунути праворуч на *dpM-1* позицій, тобто на

кількість позицій, швидше за все більше одиниці. Якщо символ, який не збігся, тексту в слові взагалі не зустрічається, то зсув стає навіть більшим, а саме зсовувати можна на довжину всього слова.

Варто сказати, що запропоновані методи пошуку послідовностей можна модифікувати таким чином, щоб у кожному рядку пошук йшов не до кінця кожного рядка, а на кількість шуканих символів менше, бо слово *s* не може бути розташоване у кінці одного рядка та на початку наступного.

10. МЕТОДИ ШВИДКОГО ДОСТУПУ ДО ДАНИХ

10.1. Хешування даних

Для прискорення доступу до даних можна використовувати попереднє їх впорядкування у відповідності зі значеннями ключів. При цьому можуть використовуватися методи пошуку в упорядкованих структурах даних, наприклад, метод дихотомічного пошуку, що суттєво скорочує час пошуку даних за значенням ключа. Проте при добавленні нового запису потрібно дані знову впорядкувати. Втрати часу на повторне впорядкування можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних використовується так зване випадкове впорядкування або хешування. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції *h*, яка використовується для „обчислення” адреси за значенням ключа.

Ідеальною хеш-функцією є така хеш-функція, яка для будь-яких двох неоднакових ключів дає неоднакові адреси. Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів відомі наперед. Така організація даних носить назву „досконале хешування”.

Наприклад, потрібно запам’ятати декілька записів, кожен з яких має унікальний ключ зі значенням від 1 до 100. Для цього можна створити масив з 100 комірками і присвоїти кожній комірці нульовий ключ. Щоб добавити в масив новий запис, дані з нього просто копіюються у відповідну комірку масиву. Щоб добавити запис з ключем 37, дані з нього копіюються в 37 позицію в масиві. Щоб знайти запис з певним ключем – вибирається відповідна комірка масиву. Для вилучення запису ключу відповідної комірки масиву просто присвоюється нульове значення. Використовуючи цю схему, можна добавити, знайти і вилучити елемент із масиву за один крок.

У випадку наперед невизначеної множини значень ключів і обмеженні розміру таблиці підбір досконалої функції складний. Тому часто використовують хеш-функції, які не гарантують виконанні умови.

Очевидно, що оскільки існує більше можливих значень ключа, ніж комірок в таблиці, то деякі значення ключів можуть відповідати одним і тим коміркам таблиці. Даний випадок носить назву „колізія”, а такі ключі називаються „ключі-синоніми”.

Щоб уникнути цієї потенційної проблеми, схема хешування повинна включати в себе алгоритм вирішення конфліктів, який визначає послідовність дій у випадку, якщо ключ відповідає позиції в таблиці, яка вже зайнята іншим записом.

Усі методи використовують для вирішення конфліктів приблизно однаковий підхід. Вони спочатку встановлюють відповідність між ключем запису і розміщенням в хеш-таблиці. Якщо ця комірка вже зайнята, вони відображають ключ

на іншу комірку таблиці. Якщо вона також вже зайнята, то процес повторюється знову до тих пір, поки нарешті алгоритм не знайде пусту комірку в таблиці. Послідовність позицій, які перевіряються при пошуку або вставці елемента в хеш-таблицю, називається тестовою послідовністю.

В результаті, для реалізації хешування необхідні три речі:

- Структура даних (хеш-таблиця) для зберігання даних;
- Функція хешування, яка встановлює відповідність між значеннями ключа і розміщенням в таблиці;
- Алгоритм вирішення конфліктів, який визначає послідовність дій, якщо декілька ключів відповідають одній комірці таблиці.

10.2. Методи розв'язання колізій

Для розв'язання колізій використовуються різноманітні методи, які в основному зводяться до методів „ланцюжків" і „відкритої адресації".

Методом ланцюжків називається метод, в якому для розв'язання колізій у всі записи вводяться покажчики, які використовуються для організації списків – „ланцюжків переповнення". У випадку виникнення колізій при заповненні таблиці в список для потрібної адреси хеш-таблиці добавляється ще один елемент.

Пошук в хеш-таблиці з ланцюжками переповнення здійснюється наступним чином. Спочатку обчислюється адреса за значенням ключа. Потім здійснюється послідовний пошук в списку, який зв'язаний з обчисленим адресом.

Процедура вилучення з таблиці зводиться до пошуку елемента в його вилучення з ланцюжка переповнення.

Якщо сегменти приблизно однакові за розміром, то у цьому випадку списки усіх сегментів повинні бути найбільш короткими при даній кількості сегментів. Якщо вихідна множина складається з N елементів, тоді середня довжина списків буде рівна N/B елементів. Якщо можна оцінити N і вибрати B якомога ближчим до цієї величини, то в списку буде один-два елементи. Тоді час доступу до елемента множини буде малою постійною величиною, яка залежить від N.

Одна з переваг цього методу хешування полягає в тому, що при його використанні хеш-таблиці ніколи не переповнюються. При цьому вставка і пошук елементів завжди виконується дуже просто, навіть якщо елементів в таблиці дуже багато. Із хеш-таблиці, яка використовує зв'язування, також просто вилучати елементи, при цьому елемент просто вилучається з відповідного зв'язного списку.

Один із недоліків зв'язування полягає в тому, що якщо кількість зв'язних списків недостатньо велика, то розмір списків може стати великим, при цьому для вставки чи пошуку елемента необхідно буде перевірити велику кількість елементів списку.

Метод відкритої адресації полягає в тому, щоб, користуючись якимось алгоритмом, який забезпечує перебір елементів таблиці, переглядати їх в пошуках вільного місця для нового запису.

Лінійне випробування зводиться до послідовного перебору елементів таблиці з деяким фіксованим кроком. При кроці рівному одиниці відбувається послідовний перебір усіх елементів після поточного.

Квадратичне випробування відрізняється від лінійного тим, що крок перебору елементів нелінійно залежить від номеру спроби знайти вільний елемент. Завдяки нелінійності такої адресації зменшується кількість спроб при великій кількості ключів-синонімів. Проте навіть відносно невелика кількість спроб може швидко привести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номеру спроби.

Ще один різновид методу відкритої адресації, яка називається подвійним хешуванням, базується на нелінійній адресації, яка досягається за рахунок сумування значень основної і додаткової хеш-функцій.

Очевидно, що в міру заповнення хеш-таблиці будуть відбуватися колізії і в результаті їх розв'язання методами відкритої адресації чергова адреса може вийти за межі адресного простору таблиці. Щоб це явище відбувалося рідше, можна піти на збільшення розмірів таблиці у порівнянні з діапазоном адрес, які обчислюються хеш-функцією.

З однієї сторони це приведе до скорочення кількості колізій і прискоренню роботи з хеш-таблицею, а з іншої – до нераціональних витрат адресного простору. Навіть при збільшенні таблиці в два рази у порівнянні з областю значень хеш-функції нема гарантій того, що в результаті колізій адреса не перевищить розмір таблиці. При цьому в початковій частині таблиця може залишатися достатньо вільних елементів. Тому на практиці використовують циклічний перехід на початок таблиці.

Розглядаючи можливість виходу за межі адресного простору таблиці, ми не враховували фактори наповненості таблиці й вдалого вибору хеш-функції. При великій наповненості таблиці виникають часті колізії і циклічні переходи на початок таблиці. При невдалому виборі хеш-функції відбуваються аналогічні явища. В найгіршому випадку при повному заповнені таблиці алгоритми циклічного пошуку вільного місця приведуть до зациклювання. Тому при використанні хеш-таблиць необхідно старатися уникати дуже густого заповнення таблиць. Звичайно довжину таблиці вибирають із розрахунку дворазового перевищення передбачуваної максимальної кількості записів. Не завжди при організації хешування можна правильно оцінити потрібну довжину таблиці, тому у випадку великої наповненості таблиці може знадобитися рехешування. У цьому випадку збільшують довжину таблиці, змінюють хеш-функцію і впорядковують дані.

Проводити окрему оцінку густини заповнення таблиці після кожної операції вставки недоцільно, тому можна проводити таку оцінку непрямим способом – за кількістю колізій під час однієї вставки. Достатньо визначити деякий поріг кількості колізій, при перевищенні якого потрібно провести рехешування. Крім того, така перевірка гарантує неможливість зациклювання алгоритму у випадку повторного перегляду елементів таблиці.

Як вже було відмічено, дуже важливий правильний вибір хеш-функції. При вдалій побудові хеш-функції таблиця заповнюється більш рівномірно, зменшується кількість колізій і зменшується час виконання операцій пошуку, вставки і вилучення. Для того щоб попередньо оцінити якість хеш-функції можна провести імітаційне моделювання.

Моделювання проводиться наступним чином. Формується вектор цілих чисел, довжина якого співпадає з довжиною хеш-таблиці. Випадково генерується достатньо велика кількість ключів, для кожного ключа обчислюється хеш-функція. В елементах вектора підраховується кількість генерацій даної адреси. За результатами такого моделювання можна побудувати графік розподілу значень хеш-функції. Для отримання коректних оцінок кількість генерованих ключів повинна в декілька разів перевищувати довжину таблиці.

Якщо кількість елементів таблиці достатньо велика, то графік будується не для окремих адрес, а для груп адрес. Великі нерівномірності засвідчують високу імовірність колізій в окремих місцях таблиці. Зрозуміло, така оцінка є наближеною, але вона дозволяє попередньо оцінити якість хеш-функції і уникнути грубих помилок при її побудові.

Оцінка буде більше точною, якщо генеровані ключі будуть більш близькими до реальних ключів, які використовуються при заповненні хеш-таблиці. Для символьних ключів дуже важливо добитися відповідності генерованих кодів символів тим кодам символів, які є в реальному ключі. Для цього потрібно проаналізувати, які символи можуть бути використані в ключі.

Наприклад, якщо ключ представляє собою прізвище українською мовою, то будуть використані українські букви. Причому перший символ може бути великою буквою, а інші – малими. Якщо ключ представляє собою номерний знак автомобіля, то також нескладно визначити допустимі коди символів в певних позиціях ключа.

10.3. Організація даних для прискорення пошуку за вторинними ключами

До тепер розглядалися способи пошуку в таблиці за ключами, які дозволяють однозначно ідентифікувати запис. Такі ключі називають первинними ключами. Можливий варіант організації таблиці, при якому окремий ключ не дозволяє однозначно ідентифікувати запис. Така ситуація часто зустрічається в базах даних. Ідентифікація запису здійснюється за деякою сукупністю ключів. Ключі, які не дозволяють однозначно ідентифікувати запис в таблиці, називаються вторинними ключами.

Навіть при наявності первинного ключа, для пошуку запису можуть використовуватися вторинні. Наприклад, пошукові системи Internet часто організовані як набори записів, які відповідають Web-сторінкам. В якості вторинних ключів для пошуку виступають ключові слова, а сама задача пошуку зводиться до вибірки з таблиці деякої множини записів, які містять потрібні вторинні ключі.

Розглянемо метод організації таблиці з інвертованими індексами. Для таблиці будується окремий набір даних, який містить так звані інвертовані індекси. Допоміжний набір містить для кожного значення вторинного ключа відсортований список адрес записів таблиці, які містять даний ключ.

Пошук здійснюється у допоміжній структурі достатньо швидко, так як фактично відсутня необхідність звернення до основної структури даних. Ділянка пам’яті, яка використовується для індексів, є відносно невеликою у порівнянні з іншими методами організації таблиць.

Недоліками даної системи є великі затрати часу на складання допоміжної структури даних і її поновлення. Причому ці затрати зростають зі збільшенням об'єму бази даних.

Система інвертованих індексів є досить зручною й ефективною при організації пошуку в великих таблицях.

Для таблиць невеликого об'єму використовують організацію допоміжної структури даних у вигляді бітових карт. Для кожного значення вторинного ключа записів основного набору даних записується послідовність бітів. Довжина послідовності бітів рівна кількості записів. Кожен біт в бітовій карті відповідає одному значенню вторинного ключа і одному запису. Одиниця означає наявність ключа в запису, а нуль – відсутність.

Основною перевагою такої організації є дуже проста й ефективна організація обробки складних запитів, які можуть об'єднувати значення ключів різними логічними предикатами. У цьому випадку пошук зводиться до виконання логічних операцій запиту безпосередньо над бітовими стрічками й інтерпретації результуючої бітової стрічки. Іншою перевагою є простота поновлення карти при добавленні записів.

До недоліків бітових карт варто віднести збільшення довжини стрічки пропорційно довжині файлу. При цьому наповненість карти одиницями зменшується зі збільшенням довжини файлу. Для великої довжини таблиці і ключів, які рідко зустрічаються, бітова карта перетворюється в велику розріджену матрицю, яка складається, в основному, з одних нулів.

11. МЕТОДИ РОЗРОБКИ АЛГОРИТМІВ

Цей розділ присвячений основним методами розв'язку на комп'ютері задач, які вважалися традиційно задачами, що під силу розв'язати тільки людині. Це задачі, у яких серед великої (часто нескінченої) кількості варіантів слід визначити один-єдиний – найкращий, або кілька найкращих. Людина, розв'язуючи такі задачі, звичайно не перебирає усі варіанти – життєвий досвід дозволяє їй відразу відкинути явно дурні варіанти, зменшуючи область пошуку. Що більше у людини досвіду у розв'язку аналогічних задач, то швидше та цілеспрямовано вона веде пошук потрібного варіанту. Такі алгоритми часто називають алгоритмами штучного інтелекту.

Алгоритми відшукання найкращого варіанту серед багатьох можливих називаються звичайно алгоритмами перебору. Задача програміста-розробника алгоритму ввести у алгоритм якомога більше „досвіду" – умов, що дозволяють зменшити кількість варіантів, які перебираються. Чим більш цілеспрямовано буде проводитись пошук, тим більше програма набуває інтелектуальних рис, тим швидше буде знайдено найкращий результат. Фактично більшість з методів є саме цілеспрямованим перебором варіантів і до „інтелектуальної" групи відноситься лише традиційно.

11.1. Метод часткових цілей

Цей метод полягає у тому, що глобальна велика задача ділиться (якщо це можливо) на окремі задачі. Якщо велику задачу, можливо, і не можна осягнути,

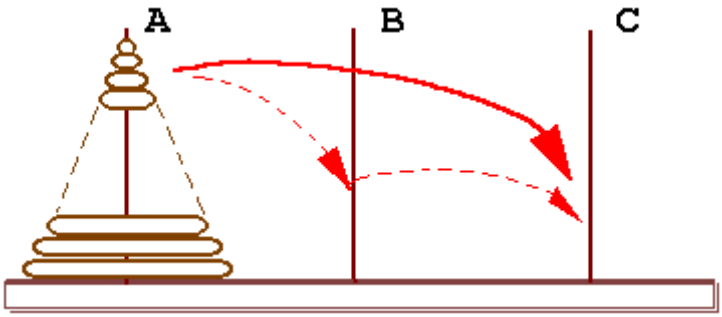
зрозуміти, як її розв'язувати, то для кожної з поділених задач може існувати давно відомий алгоритм розв'язку, або пошук такого алгоритму є значно легшою задачею.

Наприклад задача сортування масиву прямим обміном. Алгоритм сортування зводиться до відшукання мінімуму у несортованій частині масиву та дописування його до сортованої частини.

Класичний приклад методу частинних цілей – „Ханойська вежа". Принц Шак'я-Муні, якого ще називали Буддою, що означає „просвітлений", під час одної з своїх подорожей заснував у Ханої монастир. У головному храмі монастиря стоїть три стержні. На одну з них Будда надягнув 64 дерев'яні диски, усі різного діаметру, причому найширший поклав униз, а решту впорядкував за зменшенням розміру.

Слід переставити піраміду з осі *A* на вісь *C* у тому ж порядку, користуючись віссю *B*, як допоміжною, та додержуючись наступних правил:

- за один хід переставляти лише один диск з одної осі на іншу, а не декілька;
- забороняється класти більший диск на менший, тобто впорядкованість на кожній осі має зберігатися.



Ченці монастиря перекладають, не зупиняючись ні на мить, щосекунди по одному диску й досі. Коли піраміду буде складено на осі *C*, наступить кінець світу.

Рекурсивний підхід до програмування можна застосувати, якщо розуміти вежу з *n* дисків, як вежу з *n-1* диску, що стоїть ще на одному. Щоби переставити всю піраміду з *A* на *C*, слід *n-1* - піраміду переставити з *A* на *B*, перекласти один – найбільший – диск з *A* на *C*, а потім *n-1* - піраміду з *B* на *C*. Таким чином, якщо за дрібнішу задачу править та ж задача, тільки з меншим значенням параметрів, то ця форма алгоритму частинних цілей є рекурсивним алгоритмом.

Як переставити вежу з двох дисків з *A* на *C*?

- переставити диск з *A* на *B*;
- переставити диск з *A* на *C*;
- переставити диск з *B* на *C*.

Повністю аналогічно, анітрохи не складніше, програмується повний алгоритм.

Щоби переставити вежу з *n* дисків (позначимо її *B(n)*) з *A* на *C*, слід:

- *B(n-1)* переставити з *A* на *B*;
- *B(1)* перекласти з *A* на *C*;
- *B(n-1)* переставити з *B* на *C*;

Усі рекурсивні алгоритми є реалізацією методу частинних цілей.

11.2. Динамічне програмування

Нерідко не вдається поділити задачу на невелику кількість задач меншого розміру, об'єднання розв'язків яких дозволить отримати рішення початкової задачі. У таких випадках пробують поділити задачу на стільки задач, скільки необхідно, потім кожну поділену задачу ділять на ще декілька менших і так далі. Якщо б весь

алгоритм зводився саме до такої послідовності дій, то в результаті отримали б алгоритм з експоненціальним часом виконання.

Але часто вдається отримати лише поліноміальну кількість задач меншого розміру і тому ту чи іншу задачу доводиться вирішувати багаторазово. Якщо б замість того відслідковувати рішення кожної вирішеної задачі і просто шукати у випадку необхідності відповідний розв'язок, то отримують алгоритм з поліноміальним часом виконання.

З точки зору реалізації, іноді, буває простіше створити таблицю розв'язків усіх задач меншого розміру, які доведеться вирішувати. Заповнюють таку таблицю незалежно від того, чи потрібна в реальному випадку конкретна задача для отримання загального розв'язку. Заповнення таблиці складових задач для отримання розв'язку певної задачі отримало назву динамічне програмування.

Динамічним програмуванням (в найбільш загальній формі) називають процес покрокового розв'язку задачі, коли на кожному кроці вибирається один розв'язок з множини допустимих на цьому кроці, причому такий, який оптимізує задану цільову функцію або функцію критерію. В основі теорії динамічного програмування лежить принцип оптимальності Белмана.

Форма алгоритму динамічного програмування може бути різною – спільною їх темою є лише заповнення таблиці і порядок заповнення її елементами.

11.3. Метод сходження

Даний метод полягає у тому, щоби протягом пошуку найкращого розв'язку алгоритм відшукував все кращі та кращі варіанти розв'язку. Якщо ввести деяку кількісну оцінку якості розв'язку, який шукається, то такий метод подібний на здолання все нової та нової висоти при сходженні на вершину.

Розглянемо як приклад задачу про мандрівного крамаря. Крамар має проїхати *n* міст по циклу, використавши для цього цикл найменшої довжини та побувавши у кожному місті по одному разу.

Розв'язок можна шукати різними шляхами. Але самим очевидним є відшукати спочатку просто який-небудь цикл, що може вважатися „хорошим". Для цього, наприклад, можна з першого міста їхати у найближче, звідти – у найближче з тих, що залишилися і так далі. З останнього міста треба буде повернутися назад. Це – перша вершина. Далі можна спробувати поліпшити цей варіант. Якщо це вийде, то це буде нова висота. Найкоротший цикл шукається за допомогою досить складних алгоритмів.

Ще один приклад використання методу сходження – ітерації. Даний тип алгоритмів має ще одну назву – „скупі" алгоритми. На кожній окремій стадії „скупий" алгоритм вибирає той варіант, який є локально оптимальним в тому чи іншому змісті. Раніше вже були розглянуті різні „скупі" алгоритми – алгоритм побудови найкоротшого шляху Дейкестри і алгоритм побудови стовбурного дерева мінімальної вартості Крускала. Алгоритм найкоротшого шляху Дейкестри є „скупим" у тому розумінні, що він вибирає вершину, найближчу до джерела, серед тих, найкоротший шлях яких ще невідомий. Алгоритм Крускала також „скупий" – він вибирає із ребер, які залишилися і не створюють цикл, ребро з мінімальною вартістю.

Потрібно зауважити, що не кожний „скупий” алгоритм дозволяє отримати оптимальний результат в цілому. Як це буває у житті, „скупа стратегія” у більшості випадків забезпечує локальний оптимум, у той же час як в цілому результат буде неоптимальним.

Узагальнюючи вище сказане, можна зробити висновок – стратегія методу полягає в тому, щоб почати з випадкового рішення і потім робити послідовні наближення. Почавши з випадково вибраного рішення, алгоритм робить випадковий вибір. Якщо нове рішення краще попереднього, програма закріплює зміни і продовжує перевірку інших випадкових змін. Якщо зміна не покращує рішення, програма відкидає його і робить нову спробу.

11.4. Дерева розв’язків

Багато складних реальних задач можна змоделювати за допомогою дерев розв’язків. Кожний вузол дерева представляє один крок вирішення задачі. Кожна гілка в дереві представляє розв’язок, який веде до більш повного рішення. Листки є остаточним розв’язком. Мета полягає в тому, щоб знайти „найкращий шлях” від кореня дерева до листка при виконанні певних умов. Ці умови і значення поняття „найкращий” для шляху залежить від конкретної задачі.

Стратегію настільних ігор, таких як шахи, шашки, або „хрестики-нулики” можна змоделювати за допомогою дерев гри. Якщо в який-небудь момент гри існує 30 можливих ходів, то відповідний вузол у дереві гри матиме 30 гілок. Наприклад, для гри в „хрестики-нулики” кореневий вузол відповідає початковій позиції, при якій дошка порожня. Перший гравець може помістити хрестик в будь-яку з дев’яти кліток дошки. Кожному з цих дев’яти можливих ходів відповідає гілка дерева, яка виходить з кореня. Дев’ять вузлів на кінці цих гілок відповідають дев’яти різним позиціям після першого ходу гравця.

Після того, як перший гравець зробив хід, другий може поставити нулик в будь-яку з восьми кліток, що залишилися. Кожному з цих ходів відповідає гілка, що виходить з вузла, який відповідає поточній позиції.

Як можна здогадатися, дерево гри навіть для такої простої гри росте дуже швидко. Якщо воно буде продовжувати рости таким чином, що кожний наступний вузол в дереві матиме на одну гілку менше попереднього, то повне дерево гри матиме 9! = 362880 листки. Тобто, в дереві буде 362880 можливих шляхів розв’язку, які відповідають можливим варіантам гри.

Насправді багато з вузлів дерева в реальній грі будуть відсутніми, оскільки відповідні їм ходи заборонені правилами гри. Якщо гравець, що ходив першим, за три свої ходи поставить хрестики у верхній лівій, верхній середній і верхній правій клітках, то він виграє і гра закінчиться. Вузол, який відповідає цій позиції, не матиме нащадків, оскільки гра завершується на цьому кроці.

Після вилучення всіх неможливих вузлів в дереві залишається біля четверті мільйона листків. Це все ще дуже велике дерево, і пошук у ньому оптимального розв’язку методом повного перебору займає достатньо багато часу. Можна ще скоротити розмір цього дерева, враховуючи симетричність деяких позиція, але це підходить лише для конкретної гри. Для складніших ігор, таких як шашки, шахи або го, дерева мають величезний розмір. Якби під час кожного ходу в шахах гравець мав

би 16 можливих варіантів, то дерево гри мало б більше трильйона вузлів після п’яти ходів кожного з гравців.

Для виконання пошуку в дереві гри, потрібно мати можливість визначити вагу позиції на дошці. Для гри в „хрестики-нулики”, для першого гравця більшу вагу мають позиції, в яких три хрестики розташовано в ряд, оскільки при цьому він виграє. Вага тих же позицій для другого гравця мала, тому що в цьому випадку він програє.

Для кожного гравця, можна присвоїти кожній позиції один з чотирьох вагових коефіцієнтів. Якщо коефіцієнт рівний 4, то це значить, що гравець в цій позиції виграє. Якщо – 3, то з поточного положення на дошці неясно, хто з гравців виграє врешті-решт. 2 – означає, що позиція приводить до нічиєї. І, нарешті, – 1, означає, що виграє супротивник.

Для пошуку розв’язку методом повного перебору можна використовувати мінімаксну стратегію, при якій робиться спроба мінімізувати максимальну вагу, яку може мати позиція для супротивника після наступного ходу. Це можна зробити, визначивши максимально можливу вагу позиції для супротивника після кожного з своїх можливих ходів, і потім вибрати хід, який дає позицію з мінімальною вагою для супротивника.

Тобто алгоритм повинен обчислювати вагу позиції на дошці, перевіряючи всі можливі ходи. Для кожного ходу він повинен рекурсивно викликати себе, щоб знайти вагу, яку матиме нова позиція для супротивника. Потім вибирається хід, при якому вага отриманої позиції для супротивника буде якнайменшою.

Програмно для визначення ваги позиції на дошці потрібна функція, яка рекурсивно викликає себе до тих пір, поки не відбудеться одна з трьох подій. По-перше, вона може дійти до позиції, в якій гравець виграє. В цьому випадку, вона присвоює позиції ваговий коефіцієнт 4, що вказує на виграш гравця, що виконав останній хід. По-друге, може знайти позицію, в якій жоден з гравців не може зробити наступний хід. Гра при цьому закінчується нічиєю, тому функція присвоює цій позиції – 2. І нарешті, функція може досягти заданої максимальної глибини рекурсії. В цьому випадку, процедура присвоює позиції – 3, що вказує на неможливість визначити переможця. Функція при цьому вибирає хід, який має якнайменшу вагу для супротивника.

Завдання максимальної глибини рекурсії обмежує час пошуку в деревах розв’язку. Це особливо важливо для складніших ігор, таких як шахи, в яких пошук в дереві гри може продовжуватися практично вічно. Максимальна глибина пошуку також може задавати рівень майстерності програми. Чим далі вперед програма зможе аналізувати ходи, тим краще вона гратиме.

Описаний алгоритм має цікавий побічний ефект. Якщо він знаходить два однаково хороших ходи, то вибирає з них той, який знайдений першим. Іноді це приводить до дивної поведінки програми. Наприклад, якщо програма визначає, що при будь-якому своєму ходу вона програє, то вона вибирає перший з них. Може створитися враження, що програма вибрала випадковий хід і здається. В якійсь мірі це дійсно так.

Один із способів запобігання такої поведінки полягає в тому, щоб задати більше можливих вагових коефіцієнтів позицій. В попередньому варіанті всі

програшні позиції мають однакову вагу. Можна присвоїти позиції, в якій програш відбувається за два ходи, більшу вагу, ніж позиції, в якій програш наступає на наступному ходу. Тоді програма зможе вибирати ходи, які приведуть до затягування гри. Також можна присвоювати більшу вагу позиції, в якій є два можливі виграшні ходи, ніж позиції, в якій є тільки один виграшний хід. У такому разі програма спробує заблокувати один з можливих виграшних ходів.

Якби для пошуку в дереві гри була б лише мінімаксна стратегія, то виконувати пошук у великих деревах було б дуже складно. Такі ігри, як шахи, настільки складні, що програма може провести пошук всього лише на декількох рівнях дерева. На щастя, існують декілька методів, які можна використати для пошуку у великих деревах.

По-перше, в програмі можуть бути записані початкові ходи, які вибрані експертами. Можна вирішити, що програма гри в „хрестики-нулики" повинна робити перший хід в центральну клітку. Це визначає першу гілку дерева гри, тому програма може ігнорувати всі шляхи, які не проходять через першу гілку. Це зменшує дерево гри в 9 разів.

Фактично, програмі не потрібно виконувати пошук в дереві до того, поки супротивник не зробить свій хід. У цей момент і комп'ютер, і супротивник вибрали кожний свою гілку, дерево, яке залишилося, стане набагато меншим, і міститиме менше ніж 7!=5040 шляхів. Прорахувавши наперед всього один хід, можна зменшити розмір дерева гри від четверті мільйона до менше ніж 5040 шляхів.

Аналогічно, можна записати відповіді на перші ходи, якщо супротивник ходить першим. Є дев'ять варіантів першого ходу, отже, потрібно записати дев'ять відповідних ходів. При цьому програмі не потрібно поводити пошук деревом, поки супротивник не зробить два ходи, а комп'ютер – один. Тоді дерево гри міститиме менше ніж 6! = 720 шляхів. Записано всього дев'ять ходів, а розмір дерева при цьому зменшується дуже сильно. Це ще один приклад просторово-часового компромісу. Використовування більшої кількості пам'яті зменшує час, необхідний для пошуку в дереві гри.

Комерційні програми гри в шахи також починають із записаних ходів і відповідей, рекомендованих гросмейстерами. Такі програми можуть робити перші ходи дуже швидко. Після того, як програма вичерпає всі записані наперед ходи, вона почне робити ходи набагато повільніше.

Інший спосіб покращання пошуку в дереві гри полягає в тому, щоб визначати важливі позиції. Якщо програма розпізнає одну з цих позицій, вона може виконати певні дії або змінити спосіб пошуку в дереві гри.

Під час гри в шахи гравці часто розташовують фігура так, щоб вони захищали інші фігури. Якщо супротивник бере фігуру, то гравець бере фігуру супротивника замість. Часто таке узяття дозволяє супротивнику у свою чергу узяти іншу фігуру, що приводить до серії обмінів.

Деякі програми знаходять можливі послідовностей обмінів. Якщо програма розпізнає можливість обміну, вона на якийсь час змінює максимальну глибину, на яку вона проглядає дерево, щоб прослідити до кінця ланцюжок обмінів. Це дозволяє програмі вирішити, чи варто йти на обмін. Після обміну фігур їх кількість також зменшується, тому пошук в дереві гри стає в майбутньому більш простим.

Цей метод полягає у тому, щоб, почавши не з початку, а з цільової точки, якщо це можливо, визначити, як і звідки у неї можна потрапити. Далі слід шукати шляхи не до мети безпосередньо, а до „проміжних пунктів”.

Наприклад – пошук найкоротшого шляху. Буквально застосовуємо цей підхід у названій задачі. Дається граф, вершини якого – населені пункти, а кожній дузі приписано відстань між вершинами. Слід визначити найкоротший маршрут між двома заданими вершинами – *a* та *b*.

Визначивши відстані від *b* до найближчих, а вірніше, до усіх суміжних з нею вершин, шукаємо шляхи до цих вершин з *a*. Залишимо з усіх шляхів той, для якого сумарний шлях є найменшим.

Наступний приклад – сітковий графік, або бізнес-план. Для того, щоб у термін, визначений у контракті, здати замовнику роботу, слід вже за 3 дні підготувати зразки та надрукувати звіт. Щоб підготувати зразки, слід ще за день до того зробити ще деякі роботи. Щоб звіт було надруковано за три дні до терміну, слід почати друкувати щонайменше за три тижні двома принтерами. І так далі, аж до самого початку.

11.5. Програмування з поверненнями назад

Іноді доводиться мати справу з задачами пошуку оптимального розв’язку, коли неможливо застосувати жоден з відомих алгоритмів, які здатні допомогти відшукати оптимальний варіант розв’язку, і залишається застосувати останній засіб – повний перебір.

Багато задач не допускають аналітичного розв’язку, а тому їх доводиться вирішувати методом спроб та помилок, тобто перебираючи усі можливі варіанти та відкидаючи їх у випадку невдачі. У разі, якщо побудова розв’язку є складною процедурою, то фактично під час роботи будується дерево можливих кроків алгоритму, а потім – у випадку невдачі – відрізаються відповідні гілки дерева, доки не буде побудовано той шлях, який веде до успіху. Проходження вздовж гілок дерева та відхід у разі невдач і є алгоритм з поверненнями.

Розглянемо гру, в яку грають два гравці, наприклад, шахи, шашки чи „хрестики-нулики”. Гравці почергово роблять ходи, і стан гри відображається відповідним станом на дошці. Будемо вважати, що є скінчена кількість позицій на дошці і в грі передбачене певне „правило завершення”. З кожною такою грою можна асоціювати дерево, яке називається деревом гри. Кожний вузол такого дерева представляє певну позицію на дошці. Початкова позиція відповідає кореню дерева. Якщо позиція *x* асоціюється з вузлом *n*, тоді нащадки вузла *n* відповідають сукупності допустимих ходів із позиції *x*, і з кожним нащадком асоціюється відповідна результуюча позиція на дошці.

Листки цього дерева відповідають таким позиціям на дошці, з яких неможливо виконати хід, – або тому, що хтось з гравців вже отримав перемогу, або тому, що усі можливі ходи вже вичерпані. Кожному вузлу дерева відповідає певна ціна. На початку назначають ціни листкам. Нехай мова йде про гру „хрестики-нулики”. В такому випадку листкам назначається ціна -1, 0 і 1 в залежності від того, чи відповідає даній позиції програш, нічия або виграш.

Ці ціни розповсюджуються вверх по дереві у відповідності з наступним правилом. Якщо вузол відповідає такій позиції, з якої повинен виконати хід гравець 1, тоді відповідна ціна є максимальним значенням цін нащадків даного вузла. При цьому передбачається, що цей гравець зробить самий вигідний для себе хід, тобто такий, який принесе йому самий цінний результат. Якщо ж вузол відповідає ходу гравця 2, тоді відповідна ціна є мінімальним значенням цін нащадків. При цьому вважається, що гравець два виконає самий вигідний для себе хід, тобто такий, який при самих сприятливих умовах приведе до програшу гравця 1, або до нічиї.

Цей алгоритм покажемо на прикладі задачі, яка має назву „хід коня”.

На шахівниці *n*x*n* стоїть на полі (x, y) шаховий кінь. Слід знайти такий маршрут коня (який ходить згідно шахових правил), коли він обходить усю шахівницю, побувавши у кожній клітинці рівно один раз.

Загальна структура алгоритму буде така: на кожному кроці буде аналізуватися, чи можна ще зробити хід куди-небудь. Якщо так, то робимо хід, якщо ні, то повертаємося на один хід назад. Так робимо до тих пір, поки довжина ланцюга ходів не стане рівною *n-1*. Тоді це повний „хід коня”.

Характерним для цього прикладу є те, що під час пошуку розв’язку поступово то збільшують номер ходу, записуючи свій маршрут, то зменшують, витираючи ті гілки дерева можливих маршрутів, які не ведуть до успіху, після їх повного обстеження. Ця дія називається поверненням.

Приведемо алгоритм побудови загального алгоритму пошуку з поверненням. Нехай задані правила деякої гри, тобто допустимі ходи і правила її завершення. Потрібно побудувати дерево цієї гри і оцінити його корінь. Це дерево можна побудувати звичним чином, а потім виконати обхід його вузлів у зворотному порядку. Такий обхід у зворотному порядку гарантує, що алгоритм попадає у внутрішній вузол лише після обходу всіх його нащадків, в результаті чого можна оцінити цей вузол, знайшовши максимум або мінімум значень його усіх нащадків.

Об’єм пам’яті для зберігання такого дерева може виявитися достатньо великим, але, якщо дотримуватися мір безпеки, можна обійтися зберіганням в пам’яті в будь-який заданий момент часу лише одного шляху – від кореня до того чи іншого вузла. Приведемо алгоритм пошуку, який виконує обхід дерева за допомогою послідовності рекурсивних викликів. Цей алгоритм передбачає виконання наступних умов:

1. Виграші є дійсними числами з обмеженого інтервалу, наприклад [-1, 1].
2. Константа ∞ більша, ніж будь-який додатній виграш, а -∞ менша, ніж будь-який від’ємний виграш.
3. Дані типу *modetype* (тип режиму) можуть приймати два фіксовані значення – MIN, або MAX.
4. Передбачений тип даних *boardtype* (тип ігрової дошки), яка визначається способом представлення позицій на дошці.
5. Передбачена функція *payoff* (виграш), яка обчислює виграш для будь-якої позиції, яка є листком дерева.

Одна дуже проста умова дозволяє позбавитися від розгляду значної частини типового дерева гри.

Загальне правило відсікання вузлів зв'язане з поняттям кінцевих і приблизних значень вузлів. Кінцеве значення – це то, що називають виграшем. Приблизне значення – це верхня межа значення вузла в режимі MIN, або нижня межа значення вузла в режимі MAX. Приведемо правила обчислення цих значень.

1. Якщо вже розглянуто або відсічено всіх нащадків вузла, то його приблизне значення стає кінцевим.
2. Якщо орієнтовне значення вузла в режимі MAX рівне *v1*, а кінцеве значення одного з його нащадків рівне *v2*, тоді встановити приблизне значення вузла рівним *max(v1,v2)*. Якщо вузол знаходиться в режимі MIN – *min(v1,v2)*.
3. Якщо *p* є вузлом в режимі MAX, і має батька *q*, а приблизні значення вузлів рівні *v1* і *v2*, відповідно, причому *v1<v2*, тоді можна відсікти всіх нерозглянутих нащадків вузла *p*, якщо *p* є вузлом в режимі MAX, а *q* є, таким чином в режимі MIN, і *v2<v1*.

Метод гілок і границь є ще одним з методів відсікання гілок в дереві рішень, щоб не було необхідно розглядати всі гілки дерева. Загальний підхід при цьому полягає в тому, щоб відстежувати межі вже знайдених і можливих рішень. Якщо в якійсь точці найкраще з вже знайдених рішень краще, ніж краще можливе рішення в нижніх гілках, то можна ігнорувати всі шляхи вниз від вузла.

Наприклад, припустимо, що маємо 100 мільйонів доларів, які потрібно вкласти в можливі інвестиції. Кожне з вкладень має різну вартість і дає різний прибуток. Необхідно вирішити, як вкласти гроші найкращим чином, щоб сумарний прибуток був максимальним.

Задачі такого типу називаються задачею формування портфеля. Є декілька позицій (інвестицій), які повинні поміститися в портфель фіксованого розміру (100 мільйонів доларів). Кожна з позицій має вартість (гроші) і ціну (теж гроші). Необхідно знайти набір позицій, який поміщається в портфель і має максимально можливу ціну.

Цю задачу можна змоделювати за допомогою дерева рішень. Кожний вузол дерева відповідає певній комбінації позицій в портфелі. Кожна гілка відповідає ухваленню рішення про те, щоб видалити позицію з портфеля або додати її в нього. Ліва гілка першого вузла відповідає першому вкладенню.

Дерево рішень для цієї задачі є повним бінарним деревом, глибина якого рівна кількості інвестицій. Кожний листок відповідає повному набору інвестицій.

Щоб використати метод гілок і границь, створюють масив, який міститиме позиції з якнайкращого знайденого дотепер рішення. При ініціалізації масив повинен бути порожній. Можна також використати змінну для стеження за ціною цього рішення. Спочатку ця змінна може мати невелике значення, щоб перше ж знайдене реальне рішення було краще початкового.

При пошуку в дереві рішень, якщо в якійсь точці аналізоване рішення не може бути краще, ніж існуюче, то можна припинити подальший пошук цим шляхом. Також, якщо в якійсь точці вибрані позиції коштують більше 100 мільйонів, то можна також припинити пошук.

У міру просування деревом алгоритму не потрібно постійно перевіряти, чи буде часткове рішення, яке вона розглядає, краще, ніж найкраще знайдене дотепер рішення. Якщо часткове рішення краще, то буде краще і вузол, який знаходиться

праворуч внизу від цього часткового рішення. Цей вузол представляє той же самий набір позицій, як і часткове рішення, оскільки вся решта позицій при цьому була виключена. Це означає, що алгоритмові необхідно шукати краще рішення тільки тоді, коли він досягає листка дерева.

Фактично, будь-який листок, до якого доходить програма завжди є більш кращим рішенням. Якби це було не так, то гілка, на якому знаходиться цей листок, була б відсічена, коли програма розглядала батьківський вузол. У цій точці переміщення до листка зменшить ціну невибраних позицій до нуля. Якщо ціна рішення не більша, ніж найкраще знайдене дотепер рішення, то перевірка нижньої межі зупинить просування до листка. Використовуючи цей факт, алгоритм може поновляти найкраще рішення досягнувши листка.

При пошуку методом гілок і границь кількість вузлів, які перевіряються, набагато менша ніж при повному переборі. Дерево рішень для задачі портфеля з 20 позиціями містить більше 2 мільйони вузлів. При повному переборі доведеться перевірити їх усіх, при пошуку методом гілок і границь знадобиться перевірити тільки приблизно 1500 з них.

Кількість вузлів, які перевіряє алгоритм при використанні методу гілок і границь, залежить від точних значень даних. Якщо ціна позицій висока, то в правильне рішення входитиме небагато елементів. З другого боку, якщо елементи мають низьку вартість, то до правильного рішення увійде велика їх кількість, тому алгоритмові доведеться досліджувати безліч комбінацій.

11.6. Евристичні алгоритми

У складніших іграх практично неможливо провести пошук навіть в невеликому фрагменту дерева. У цих випадках, можна використовувати різні евристики. Евристикою називається алгоритм або емпіричне правило, яке ймовірно, але не обов’язково дасть добрий результат.

Наприклад, в шахах звичайною евристикою є „посилення переваги”. Якщо у супротивника менше сильних фігур і однакова кількість інших, то слід йти на розмін при кожній нагоді. Зменшення кількості фігур робить дерево рішень коротшим і може збільшити відносну перевагу. Ця стратегія не гарантує виграшу, але підвищує його вірогідність.

Інша евристика, що часто використовується, полягає в присвоєнні різних ваг різним клітинкам. У шахах вага кліток в центрі дошки вища, оскільки фігури, що знаходяться на цих позиціях, можуть атакувати більшу частину дошки. Коли обчислюється вага поточної позиції на дошці, вона може присвоюватися більшою фігурам, які займають клітки в центрі дошки.

Якщо якість рішення не так важлива, то прийнятним може бути результат, отриманий за допомогою евристики. В деяких випадках точність вхідних даних може бути недостатньою. Тоді хороше евристичне рішення може бути таким же правильним, як і теоретично „якнайкраще рішення”.

У попередньому прикладі метод гілок і границь використовувався для вибору інвестиційних можливостей. Проте, вкладення можуть бути ризикованими, і точні результати часто наперед невідомі. Можливо, що наперед буде невідомий точний дохід або навіть вартість деяких інвестицій. В цьому випадку, ефективне евристичне

рішення може бути таким же надійним, як і якнайкраще рішення, яке ви може обчислити точно.

Отже, евристичні алгоритми – це алгоритми, які мають такі властивості:

- Вони дозволяють знайти добрі, хоча і не завжди найкращі розв'язки з усіх, що існують.
- Метод пошуку або побудови розв'язку звичайно значно простіший, ніж той що гарантує оптимальність розв'язку.

Поняття „добрий розв'язок" змінюється від задачі до задачі, тому його важко визначити точно. Припустимість використання евристики залежить від співвідношення часу та складності пошуку розв'язку обома способами та співвідношення якості обох розв'язків.

У цьому розумінні усі умови, що висуваються до розв'язку, звичайно ділять на дві групи щодо витрат праці:

- ті, які легко задовольнити;
- ті, що вимагають великої роботи.

З іншої сторони, вони поділяються на такі групи щодо їхньої важливості для кінцевої якості:

- ті, які обов'язково слід задовольнити;
- ті, що можуть бути послаблені або змінені.

Цю ситуацію образно показано в таблиці

	1	2
a	Слід задовольнити	?
b	?	Варто відмовитися

Повернемося до нашого прикладу – задачі про мандрівного крамаря. Якщо міст, про які йдеться – багато, то пошук точно мінімального за довжиною циклу може вимагати дуже багато часу. З іншої сторони, об'їзд слід зробити лише один раз. Якщо витрати на пошук оптимального маршруту можуть виявитися більшими або еквівалентними до витрат на пальне під час подорожі, то, можливо, варто просто рушати в путь, прямуючи до найближчого міста кожного разу.

Якщо ж слід відшукати найкращий маршрут для регулярного об'їзду (наприклад, вибирання пошти зі скриньок, розвезення хлібу з заводу по магазинах, маршрут для руки-маніпулятора робота під час монтування друкованих плат) , то все ж варто відшукати оптимальний маршрут, бо витрати на програмування є разовими, а витрати на зайвий рух є регулярними.

Візьмемо за приклад сортування масиву. Якщо потрібно відсортувати один раз масив зі 100 записів, то можна використати будь-який з простих методів сортування – на весь процес буде витрачено щонайбільше 2-3 секунди машинного часу. Якщо ж розробляють пакет, що буде регулярно сортувати значні масиви інформації, то варто написати сучасну програму.

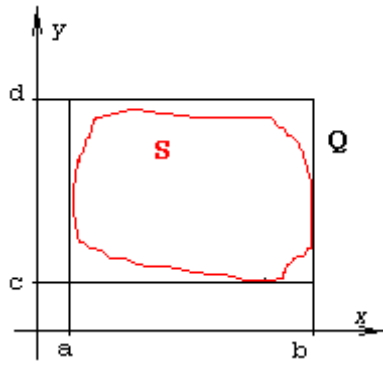
Частіше за все евристичні алгоритми базуються на методі сходження або на методі частинних цілей.

11.7. Імовірнісні алгоритми

До цієї групи відносяться всі алгоритми, де у деякій мірі використовуються результати теорії ймовірності, математичної статистики, генератори випадкових чисел тощо.

Серед усієї гами цих алгоритмів розглянемо лише один для прикладу - обчислення площі фігури методом Монте-Карло.

Нехай слід обчислити площу криволінійної області S, рівняння межі якої таке складне, що виключає застосування аналітичних методів обчислення.



Якщо область S повністю можна включити у прямокутну область

$$Q = \{ (x, y): x \in [a;b]; y \in [c;d] \},$$

то S ділить Q на долі пропорційно

$$\frac{\|S\|}{\|Q\| - \|S\|}$$

Якщо за допомогою датчика випадкових чисел генерувати точки $(x, y) \in Q$, які б рівномірно заповнювали прямокутник Q, то кількість тих точок, що потраплять усередину області S, та тих, що потраплять за її межі, буде відноситися приблизно так само

$$\frac{N_S}{N_Q} = \frac{\|S\|}{\|Q\|} \text{ тоді } \|S\| = \|Q\| \cdot \frac{N_S}{N_Q} = (b-a) \cdot (d-c) \cdot \frac{N_S}{N_Q}$$

Кількість N_S та N_Q підраховуються під час експерименту.

Аналогічно працюють алгоритми випадкового пошуку, де пошук виконується відповідно до своєї назви. Для задачі формування портфеля – на кожному кроці алгоритм додає випадкову позицію, яка задовольняє верхньому обмеженню на сумарну вартість позицій в портфелі. Цей метод пошуку також називається методом Монте-Карло.

Оскільки малоймовірно, що випадково вибране рішення виявиться якнайкращим, необхідно багато разів повторювати цей пошук, щоб отримати прийнятний результат. Хоча може видатися, що вірогідність знаходження хорошого рішення при цьому мала, цей метод іноді дає хороші результати. Залежно від значень даних і числа перевірених випадкових рішень результат, отриманий за допомогою цього методу, часто виявляється краще, ніж у випадку використання інших методів.

Перевага випадкового пошуку полягає також і в тому, що цей метод легкий в розумінні і реалізації. Іноді складно уявити, як реалізувати рішення задачі за допомогою складних алгоритмів пошуку, але завжди просто вибирати рішення випадковим чином. Навіть для дуже складних проблем, випадковий пошук є простим евристичним методом.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Структурування алгоритмів.
2. Структурування даних.
3. Інкапсуляція, як засіб структуризації.
4. Концепція структур даних.
5. Класифікація структур даних.
6. Базові операції над структурами даних.
7. Дані арифметичного типу.
8. Дані перерахованого типу.
9. Властивості статичних структур даних.
10. Масив, як структура даних
11. Розріджені масиви.
12. Множина, як структура даних.
13. Структурний тип даних.
14. Об’єднання, як структура даних.
15. Бітовий тип даних.
16. Таблиця, як структура даних.
17. Особливості напівстатичних структур даних.
18. Стек, як структура даних.
19. Черга, як структура даних.
20. Дек, як структура даних.
21. Лінійні списки.
22. Однонаправлений лінійний список.
23. Двонаправлений лінійний список.
24. Основні поняття мультисписків.
25. Стрічка, як структура даних.
26. Зв’язне представлення даних.
27. Представлення графа, як структури даних.
28. Дерево, як структура даних.
29. Алгоритм перетворення дерева в бінарне.
30. Представлення дерев в пам’яті.
31. Операції над деревами.
32. Алгоритми обходу дерева.
33. Принципи формалізації алгоритмів.
34. Покрокове проектування алгоритмів.
35. Основні характеристики алгоритмів.
36. Поняття складності алгоритму.
37. Ефективність алгоритмів.
38. Правила аналізу складності алгоритмів.
39. Постановка задачі сортування.
40. Класифікація алгоритмів сортування.
41. Сортування вибіркою.
42. Сортування включенням.
43. Сортування розподілом.
44. Сортування злиттям.
45. Принципи рандомізації.
46. Постановка задачі пошуку.
47. Послідовний пошук.
48. Бінарний пошук.
49. Пошук методом інтерполяції.
50. Пошук методом ,,золотого перерізу”.
51. Алгоритми пошуку послідовностей.
52. Хешування даних.
53. Алгоритми розв’язання колізій при хешуванні.
54. Організація даних для пошуку.
55. Метод часткових цілей.
56. Метод динамічного програмування.
57. Метод сходження.
58. Дерева розв’язків.
59. Програмування з поверненням назад.
60. Евристичні алгоритми.
61. Імовірнісні алгоритми.
62.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы – М.: Изд. Дом «Вильямс», 2001. – 384с.
2. Браунси К. Основные концепции структур данных и реализация в С++. – М.: Изд. Дом «Вильямс», 2002. – 320с.
3. Проценко В.С. Техніка програмування мовою Сі: Навчальний посібник – К.: Либідь, 1993. – 224 с.
4. Шилдт Г. Теория и практика С++ – СПб.: BHV – Санкт-Питербург, 1996. – 416 с.
5. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: «Мир», 1979. - 536с.
6. Вирт Н. Алгоритмы + структуры данных = программы. - М.: "Мир", 1985. - 544 с.
7. Гудман С. Хидетниеми С. Введение в разработку и анализ алгоритмов. - М.: "Мир", 1981. - 366 с.
8. Кнут Д. Искусство программирования для ЭВМ. т.3. Сортировка и поиск. М.:Мир, 1976. - 678 с.
9. Мейер Б., Бодуэн К. Методы программирования: В 2-х томах – М.: Мир, 1982. – 356+368с.

ЗМІСТ