.

IV. АЛГОРИТМІЧНІ СТРАТЕГІЇ

4.1 Поняття та види стратегій

Поняття алгоритмічної стратегії дозволяє класифікувати базові алгоритми обробки даних по групах їх використання. Сучасніше розуміння терміну відноситься до програмних реалізацій алгоритмів.

Стратегія, Strategy - поведінковий шаблон проектування, призначений для визначення сімейства алгоритмів, інкапсуляції кожного з них і забезпечення їх взаємозамінюваності. Це дозволяє вибирати алгоритм шляхом визначення відповідного класу. Шаблон Strategy дозволяє міняти вибраний алгоритм незалежно від об'єктів- клієнтів, які його використовують.

Базові алгоритми обробки даних є результатом досліджень і розробок, що проводилися упродовж десятків років. Але вони, як і раніше, продовжують відігравати важливу роль в застосуванні обчислювальних процесів, що все розширюється.

До базових алгоритмів програмування можна віднести:

- Алгоритми роботи із структурами даних. Вони визначають базові принципи і методологію, використовувані для реалізації, аналізу і порівняння алгоритмів. Дозволяють отримати уявлення про методи представлення даних. До таких структур відносяться зв'язні списки і рядки, дерева, абстрактні типи даних, такі як стеки і черги.
- Алгоритми сортування, призначені для впорядкування масивів і файлів, мають особливу важливість. З алгоритмами сортування пов'язані, зокрема, черги по пріоритету, завдання вибору і злиття.
- Алгоритми пошуку, призначені для пошуку конкретних елементів у великих колекціях елементів. До них відносяться основні і розширені методи пошуку з використанням дерев і перетворень цифрових ключів, у тому числі дерева цифрового пошуку, збалансовані дерева, хешування, а також методи, які підходять для роботи з дуже великими файлами.
- Алгоритми на графах корисні при рішенні ряду складних і важливих завдань. Загальна стратегія пошуку на графах розробляється і застосовується до фундаментальних завдань зв'язності, у тому числі до завдання відшукування найкоротшого шляху, побудови мінімального остовного дерева, до завдання про потоки в мережах і завданні про паросочетаниях. Уніфікований підхід до цих алгоритмів показує, що в їх основі лежить одна і та ж процедура, і що ця процедура базується на основному абстрактному типі даних черги по пріоритету.
- Алгоритми обробки рядків включають ряд методів обробки (довгих) послідовників символів. Пошук в рядку призводить до зіставлення з еталоном, що у свою чергу веде до синтаксичного аналізу. До цього ж класу завдань можна віднести і технології стискування файлів.

- Геометричні алгоритми - це методи рішення завдань з використанням точок і ліній (і інших простих геометричних об'єктів), які увійшли до вживання досить нещодавно. До них відносяться алгоритми побудови опуклих оболонок, заданих набором точок, визначення перетинів геометричних об'єктів, рішення завдань відшукування найближчих точок і алгоритму багатовимірного пошуку. Багато хто з цих методів доповнює прості методи сортування і пошуку.

Виділимо алгоритмічні стратегії, які використовуються в алгоритмах :

- алгоритми грубої сили;
- жадібні алгоритми;
- "Розділяй і володарюй";
- алгоритми з поверненням;
- евристичні алгоритми;
- зіставлення із зразком і алгоритми обробки рядків/текстів;
- алгоритми чисельної апроксимації;
- online- і offline-алгоритми;
- динамічне програмування; і інші.

Розглянути реалізацію усіх вище приведених стратегій у рамках даного курсу представляється неможливо складним, та і потреби більшості завдань обмежуються лише вузьким кругом алгоритмічних стратегій, які, так або інакше, зустрінуться нам нижче.

Представимо основні парадигми.

Рекурсія - фундаментальне поняття в математиці і комп'ютерних науках. У мовах програмування рекурсивною програмою називається програма, яка звертається сама до себе. Рекурсивна програма не може викликати себе до безкінечності, отже, друга важлива особливість рекурсивної програми - наявність умови завершення, що дозволяє програмі припинити викликати себе.

Таблиця 4.1. Основні алгоритмічні стратегії

Тип алгоритму	**Ідея алгоритму і "при родящего ефективності**	**Діапазон трудаем-кісток**
Рекурсивний або звичайний розподіл	"Розділяй і володарюй": завдання розбивається на ідентичні підзадачі, результати яких об'єднуються в загальне рішення	$N \ldots N \log N \ldots N^2$
Повний перебір	"Гірше не буває"(без коментарів)	$2^N \ldots N^N \ldots N!$
Динамічне програмування	"Де жа ию": запам'ятовування результатів підзадач, що повторюються, збільшення продуктивності за рахунок додаткової пам'яті.	
Жадібний яскраво-червоний горитм	"Лицар на роздоріжжі": локальний вибір єдиної з підзадач на кожному кроці дає глобальне оптимальне рішення	$\log N \ldots N$

Рекурсивне рішення задачі полягає в декомпозиції великого завдання на дрібніші підзадачі того ж виду, рішенні цих підзадач і в наступному об'єднанні отриманих рішень для формування рішення початкової задачі. Підзадачі вирішуються рекурсивно тим же самим алгоритмом. В результаті декомпозиції

мають бути, кінець кінцем, отримані підзадачі такі прості (малій розмірності), що їх рішення може бути отримане безпосередньо.

Така алгоритмічна парадигма називається рекурсивною декомпозицією. Алгоритми, засновані на рекурсивній декомпозиції, аналізуються за допомогою рекурентних стосунків. Приклад рекурентного відношення визначення функції факторіалу :

$$n!=n*(n-1)! \quad \text{При } n>1$$
$$n!=1 \quad \text{при } n=0.$$

Рекурсивну програму завжди можна перетворити в нерекурсивну (ітеративну, використовуючу цикли), яка виконує ті ж обчислення. І навпаки, використовуючи рекурсію, будь-яке обчислення, що припускає використання циклів, можна реалізувати, не прибігаючи до циклів.

Рекурентне співвідношення це рекурсивна функція з цілочисельними значеннями. Значення будь-якої такої функції можна визначити, обчислюючи усі її значення починаючи з найменшого, використовуючи на кожному кроці раніше вичислені значення для підрахунку поточного значення. Рекурентні вирази використовуються, зокрема, для визначення складності рекурсивних обчислень.

Наприклад, при спробі вичислити числа Фібоначчі за рекурсивною схемою F(i) = F(i - 1) + F(i - 2), при N >= 1; F(0) = 0; F(1) = 1;

Текст програми буде приблизно таким:

```
Function F( n : integer ) : longint;
 begin
  if n < 2 then F := n
         else F := F(n-1) + F(n-2)
 end;
```

Кількість рекурсивних викликів при обчисленні значення F(N) за такою схемою може бути отримана з рішення рекурентного виразу $T_N = T_{N-1} + T_{N-2}$, при $N>=1$; $T_0 = 1$; $T_1 = 1$, де

T_N приблизно рівний $Ф^N$, де Ф~1.618 - золота пропорція, тобто приведена вище програма зажадає експоненціальних тимчасових витрат на обчислення.

Рекурсивне програмування дає загальний підхід до рішення завдань - розбиття їх на аналогічні підзадачі меншої розмірності, або на завдання, що є кроками в можливих напрямах її рішення. Так або інакше, рекурсивні виклики утворюють деревовидну структуру, кількість вершин в якій визначає ефективність алгоритму (виразиму зазвичай через трудомісткість). Різниця між різними типами алгоритмів полягає в способі отримання підзадач, їх розмірності, способі з'єднання отриманих результатів.

Ідея рекурсивного або звичайного розподілу сходить до технологічного прийому - модульного програмування. У своєму початковому варіанті вона припускає розбиття на завдання різної природи. Не рекурсивний розподіл дозволяє досягти певного ефекту за рахунок розбиття завдання на безліч ідентичних завдань меншої розмірності з наступним об'єднанням результату. Застосування того ж самого алгоритму рекурсивне до отриманих підзадач дає

наступний клас алгоритмів - рекурсивний розподіл. Як правило, незалежність отриманих підзадач має на увазі відповідний розподіл початкових даних завдання на підмножини, що не перетинаються, - цьому і відповідає сам термін розподіл. Ефективність (і трудомісткість) таких алгоритмів, залежить від витрат на само розподіл і від пропорцій частин, що розділяються : кращому випадку відповідає ділення на рівні частини (логарифмічні залежність), гіршому - виділення єдиного елементу (лінійні залежності).

Жадібні алгоритми. Ідеальним випадком можна вважати алгоритм, здатний "вибрати з декількох зол" єдино правильне. У основі його так само лежить принцип розподілу, але в кожній точці він має основу вибрати одну з підзадач. Зазвичай це робиться на підставі особливостей організації оброблюваних даних або їх надмірності. Основою жадібних алгоритмів є завжди досить спірне твердження: рух "по лінії найменшого опору" в кожній точці приведе до бажаного результату.

Повний перебір (вичерпний, комбінаторний перебір). Перелічені вище підходи засновані на всіляких "хитрощах", заснованих на особливостях предметної області алгоритму. Якщо ж нічого не допомагає, то залишається повний перебір усіх можливих варіантів рішення задачі.

Динамічне програмування. В процесі породження дерева рекурсивних викликів можливе повторення підзадач з одними і тими ж даними. Якщо запам'ятовувати результат їх виконання, то ефективність алгоритму може бути значно збільшена. Вивчення рекурсії нерозривно пов'язане з вивченням рекурсивно визначуваних структур даних, званих деревами (trees). Дерева використовуються як для спрощення розуміння і аналізу рекурсивних програм, так і в якості явних структур даних. У свою чергу, рекурсивні програми використовуються для побудови дерев. Глобальний зв'язок між ними (і рекурентними стосунками) використовується при аналізі алгоритмів. Багато алгоритмів використовують два рекурсивні виклики, кожен з яких працює приблизно з половиною вхідних даних. Така рекурсивна схема, по- видимому є найбільш важливим випадком добре відомого методу "розділяй і володарюй" (divide and conquer) розробки алгоритмів. В якості прикладу розглянемо завдання відшукування максимального з N елементів, збережених в масиві a[1], . . ., a[N] з елементами типу Item. Це завдання легко може бути вирішене за один прохід масиву

```
Max:=a[1];
For i:=1 to N do
  if a[i] > Max then Max:=a[i];
```

Рекурсивне рішення типу "розділяй і володарюй ще один простий (хоча абсолютно інший) спосіб рішення тієї ж задачі :

```
Function Max (a array of Item; l, r : integer) : Item;
var u, v : Item; m : integer;
begin
  m := (l+r) / 2;
  if (l = r)
    then Max := a[l]
    else begin
        u := Max (a, l, m);
        v := Max (a, m+1, r);
        if (u > v) then Max := u else Max := v
    end
end;
```

Більшість сучасних мов високого рівня підтримують механізм рекурсивного виклику, коли функція, як елемент структури мови програмування, повертає вичислене значення по своєму імені, може викликати сама себе з іншим аргументом. Ця можливість дозволяє безпосередньо реалізовувати обчислення рекурсивно певних функцій. Відмітимо, що через тезу Черча - Тюринга апарат рекурсивних функцій Черча равний за потужністю машині Т'юринга, і, отже, будь-який рекурсивний алгоритм може бути реалізований ітераційно.

Найчастіше підхід "розділяй і володарюй" використовують через те, що він забезпечує швидші рішення, ніж ітераційні алгоритми. Основним недоліком алгоритмів типу "розділяй і володарюй" являється те, що ділять завдання на незалежні підзадачі. Коли підзадачі незалежні, це часто призводить до неприпустимо великих витрат часу, оскільки одні і ті ж підзадачі починають вирішуватися багато разів.

Аналіз трудомісткості рекурсивних реалізацій алгоритмів, очевидно, пов'язаний як з кількістю операцій, що виконуються при одному виклику функції, так і з кількістю таких викликів. Графічне представлення породжуваного цим алгоритмом ланцюжка рекурсивних викликів називається деревом рекурсивних викликів. Детальніший розгляд призводить до необхідності обліку витрат як на організацію виклику функції і передачі параметрів, так і на повернення вичислених значень і передачу управління в точку виклику. Можна помітити, що деяка гілка дерева рекурсивних викликів обривається досягши такого значення передаваного параметра, при якому функція може бути вичислена безпосередньо. Таким чином, рекурсія еквівалентна конструкції циклу, в якому кожен прохід є виконання рекурсивної функції із заданим параметром. Дерево рекурсивних викликів може мати і складнішу структуру, якщо на кожному виклику породжується декілька звернень - фрагмент дерева рекурсій для чисел Фібоначчі представлений на рис. 4.1.

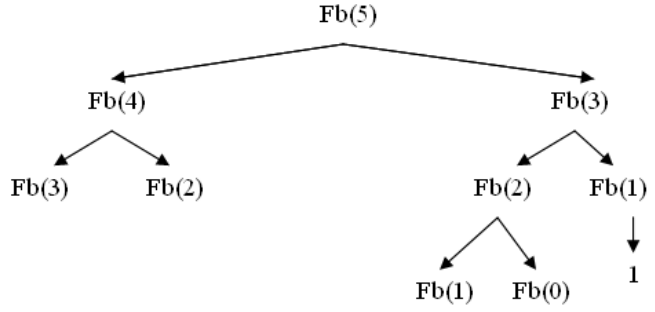

Рис. 4.1. Фрагмент дерева рекурсії при обчисленні чисел Фібоначчі

4.2 Методи розробки алгоритмів

Розкладання завдання в послідовність різнорідних підзадач

Цей метод називають іноді методом **"розділяй і володарюй"**.

У цьому методі зазвичай виділяється відносно невелике число підзадач. Наприклад: завдання - виконати програму на ЕОМ; підзадачі - ввести початковий текст програми; транслювати програму в машинні команди; приєднати до машинного коду стандартні процедури з бібліотеки; завантажити програму в оперативну пам'ять; запустити про-цесс виконання; завершити процес виконання програми.

Результати рішення першої підзадачі стають початковими даними для другої підзадачі і т. д. Таким чином, тут використаний другий підхід в чистому вигляді - декомпозиція функції, завдання її суперпозицією простіших. Помітимо також, що така суперпозиція може бути задана послідовним з'єднанням машин Т'юринга.

На алгоритмічній мові цей метод може бути виражений записом процедур, що послідовно викликаються.

Розкладання завдання в послідовність однорідних підзадач (ітерація)

Важливий окремий випадок попереднього методу, що придбаває, однак, нова якість за рахунок того, що завдання P зводиться до n екземплярів більш простого завдання R і до простого завдання Q, що об'єднує n рішень.

Дуже простий приклад: обчислення скалярного твору двох векторів A і B :

```
S:=0; {Завдання Q - підготовка місця для підсумовування.}
for i:= 1 to n do_
S:=S+A[i]*B[i];  {Завдання R - перемножування компонент і сумування.}
```

Цей алгоритм використовує розбиття початкових даних на частини - окремі компоненти векторів.

Однорідність підзадач дозволяє значно скоротити довжину тексту алгоритму за рахунок застосування операторів повторення. Ітерація на рівні великих підзадач або окремих невеликих операторів зустрічається у більшості реальних алгоритмів і служить основним джерелом ефективного використання комп'ютера в порівнянні з іншими обчислювальними засобами (наприклад, непрограмованим калькулятором).

Зведення завдання до самої собі (рекурсія)

Завдання також, як і в попередньому методі зводиться до більше за просту. Але це простіше завдання має те ж формулювання, що і початкова, з тією лише різницею, що вирішуватися вона повинна для простіших початкових даних. Це чистий варіант спрощення початкових даних.

Метод послідовних наближень

Спочатку яким-небудь чином вгадується значення x0, близьке до рішення. Завдання P знаходження рішення зводиться до багатократного рішення завдання R поліпшення рішення. Метод припускає, що якимсь чином може бути оцінена "якість" рішення (зазвичай - точність). Найчастіше абсолютна точність недосяжна, тому процес потенційно нескінченний, т. е. не виконується властивість скінченності алгоритму. Для того, щоб цього уникнути, дещо змінюють первинне формулювання завдання: вимагають відшукати не точне

рішення Y, а будь-яке рішення, отли-чающееся від Y не більше ніж на деяку величину Δ - тобто наближене рішення. Характерний приклад - завдання відшукування кореня рівняння або завдання пошуку кореня *p* -ої міри з *x*.

Основною проблемою є побудова завдання R по початковій задачі P, доказ факту збіжності процесу до шуканого рішення і забезпечення досить високої швидкості збіжності.

Цей варіант методу ітерацій використовується зазвичай для завдань, в яких шуканий результат Y виражається за допомогою речових або комплексних чисел. В усякому разі, на безлічі рішень повинна існувати метрика і повинно бути гарантовано, що задовільні наближені рішення існують. Передбачається, що початкові дані не розбиваються на частини, не спрощуються, а використовуються на кожному кроці ітерації. Причому, якщо в загальному методі ітерацій з розбиттям вихідних даних зазвичай не важливо, в якому порядку проводяться окремі кроки ітерації (вони можуть бути проведені одночасно, якщо дозволяє апаратура ЕОМ), то в методі послідовних наближень цей порядок дуже важливий.

Рішення зворотної задачі

Іноді зворотне завдання, т. е. завдання, що відповідає функції f, - 1(Y)=X, вирішується значно простіше, ніж початкове завдання. Тоді наявний алгоритм рішення зворотної задачі R іноді можна використовувати для побудови алгоритму рішення прямої задачі P._

Метод повного перебору

Коли говорили про рішення задачі обчислення f(X) шляхом декомпозиції функції f або розбиття початкових даних X на частини, то малося на увазі, що існують математичні результати або інші підстави для таких перетворень. Інакше кажучи, завдання аддитивне - її рішення може бути отримане об'єднанням рішень приватних завдань.

У ряді випадків це не так. Розглянемо, наприклад, завдання про рюкзак.

Дані ціле ненегативне число N і k чисел {n1, n2, n3, ..., nk}; знайти підмножину в {n1, n2, n3, ..., nk}, сума чисел якого рівна N, якщо така підмножина існує.

Немає ніяких підстав для того, щоб розділити початкові дані на частини і вирішувати завдання для спрощених початкових даних. Саме формулювання завдання також не вказує ніякої можливості декомпозиції.

Вихід виявляється одночасно і простим і складним. Можна узяти деяку підмножину і безпосередньою перевіркою (сумуванням чисел з цієї підмножини і порівнянням суми з N) дізнатися, чи задовольняє цю підмножину поставленій умові. Оскільки різних підмножин є кінцева кількість 2k, то потенційно можна перебрати усі підмножини і знайти рішення.

Складність полягає в тому, що зі збільшенням кількості вихідних даних (переході від k до k+1) швидко збільшується необхідне число перевірок. При k = 10 їх буде 1024, а при k = 40 вже більше 1012.

Метод повного перебору застосовний в тих випадках, коли шукане рішення Y =f(X) належить деякій кінцевій області і може бути знайдена проста функція quality(Y) для перевірки правильності (чи якості) вибраного рішення. Тоді завдання P обчислення функції f замінюється на багатократне рішення задачі R обчислення функції quality (стільки разів, скільки елементів є в області

рішень). Причому, в загальному випадку, проглянути треба усю область і порядок, в якому проглядаються елементи, не важливий.

Евристичні методи розробки алгоритмів

Під евристичними розуміються методи, правильність яких не доведена. Вони виглядають правдоподібними, здається, що у більшості випадків вони повинні давати вірне рішення. Іноді не вдається побудувати контрприклад, що демонструє помилковість або неуніверсальність метода. Але не вдається довести математичними засобами і правильність методу. Проте, практика використання евристичних методів дає позитивні результати.

Евристичні методи різноманітні, тому не можна описати загальну схему розробки таких методів. Найчастіше евристичні методи застосовують спільно з методами перебору для скорочення кількості варіантів, що перевіряються : деякі підмножини варіантів згідно з вибраною евристикою вважаються свідомо неприйнятними і не перевіряються. Таким чином, алгоритм перебору з евристикою виконується значно швидше, ніж алгоритм повного перебору. Платою за це є відсутність гарантії правильності рішення або гарантії того, що з усіх можливих вибрано найкраще рішення.

В якості прикладу можна привести завдання розфарбовування вершин графа: розфарбувати вершини графа так, щоб суміжні вершини були розфарбовані в різні кольори, а кількість використаних в графі фарб була мінімальною.

Точне рішення задачі можливе методом повного перебору, але воно виходитиме за занадто великий час для графів, декількох десятків вершин, що містять. Проте можна помітити, що у формулюванні завдання містяться дві умови - перше - обов'язкове і друге (мінімальності), яке частенько можна ослабити. Наприклад, якщо мінімальне число фарб рівне десяти, а алгоритм швидко знайде розфарбовування, що використовує одинадцять фарб, то часто таке рішення можна розглядати як прийнятне.

Динамічне програмування

Найбільш загальною формою називають процес покрокового рішення задач, коли на кожному кроці вибирається одно значення з безлічі допустимих на цьому кроці, причому таке, яке оптимізує задану мету. У основі програмування лежить принцип оптимальності Беномена. Суть методу :

Часто не вдається розбити завдання на невелику кількість підзадач, об'єднане рішення яких дозволяє отримати рішення шуканих задач.

Виникають 2 можливості:

1. Можна спробувати розділити завдання на стільки завдань, скільки необхідно, потім кожну на ще дрібніші і так далі. Якщо алгоритм зводиться до саме такої послідовності, то отримаємо завдання з експоненціальним часом рішенням.
2. Іноді вдається отримати алгоритм з поліноміальним числом підзадач, і ту або іншу доводитися вирішувати багаторазово. Якщо би відслідковували кожну підзадачу і просто відшукували їх рішення, то можна отримали поліноміальне рішення.

Іноді простіше створити таблицю рішення усіх підзадач незалежно від того, потрібна вона або ні.

Заповнення таблиць - рішення підзадач для отримання рішення певної задачі - в теорії алгоритмів дістало назву динамічного програмування. Форми алгоритмів динамічного програмування можуть бути різними, загальними можуть бути лише заповнені таблиці і порядок заповнення її елементів.

Метод балансування

При проектуванні деяких алгоритмів доводитися йти на раз-особисті компроміси, тобто по можливості збалансувати обчислювальні витрати на використання різних частин алгоритми. Метод балансування розглядається як балансування дерев. Дерево - важлива структура даних, вживана для зберігання, обробки і представлення інформації. Дерево складається з елементів^ вершин і зв'язків між ними (дуг). Серед вершин виділяється одна, яка називається коренем. Вона є батьківською по відношенню до інших пов'язаних з нею вершин. Усі вершини, пов'язані з коренем дугами, називаються нащадками. Кожна вершина в дереві, окрім кореня, має точно одну батьківську вершину і більше нащадків. Дерево має дві властивості:

- зв’язність: з кореня треба пройти по дугах до кожної вершини;
- воно не містить циклів, тобто замкнутих послідовностей вершин і дуг.

У комп'ютерних науках часто використовуються дерева, в яких усі вершини містять обмежене число нащадків (не більше двох).

Якщо це число 2 (0,1,2), то такі дерева називаються бінарними. Якщо у вершини два нащадки, те дерево ділиться на ліве піддерево і праве. Розрізняють ідеальні і вироджені дерева.

Ідеальне дерево - дерево, в якому усі вершини розташовуються на k рівнях : корінь - на 1-му, дві вершини на 2-му, чотири - на 3-му. Нова вершина може бути поміщена тільки на k+1 рівень.

Вироджене дерево - дерево, яке є лінійним списком елементів, впорядкованих за збільшенням або зменшенням інформаційних полів.

Критерій якості дерева - це довжина найдовшої гілки. На одиницю від цього значення відрізняється кількість рівнянь дерева, тобто її висота (на одиницю менше). Дерева мінімальної висоти мають максимальну кількість вершин на одному рівні. Таким чином хороші з точки зору пошуку і вставки вершини - це дерева мінімальної висоти. Вони мають мінімальну кількість вершин на кожному рівні.

Метод балансування - метод що дозволяє не допустити занадто поганих дерев.

Суть: дерево називається збалансованим, якщо висота hl лівого піддерева і висота hr правого піддерева для кожної вершини відрізняються не більше ніж на одиницю. Кращі зі збалансованих дерев є ідеальними. При включенні нової вершини в дерево може (але не обов’язково) збільшуватися висота одного з дерев.

Можливі 3 ситуації.

1. Вершина включається в піддерево меншої висоти, його висота збільшується на одиницю, дерево ставає збалансованим (ідеальним).
2. Піддерева мають однакову висоту. При включенні нової вершини, висота однієї з них збільшується на одиницю, дерево залишається збалансованим.

3. Вершина включається в піддерево більшої висоти, різниця висот ставає рівною двом. Дерево стає незбалансованим. Варіанти рішення : треба гілці одного з піддерев підняти на одну одиницю і одночасно на одиницю опустити інше. Ця модифікація призводить до зміни форми дерева. Піднімаючи ліве піддерево, включаючи його корінь, тим самим, робиться рівень кореня лівого піддерева вищим, ніж корінь самого дерева. Отже, корінь лівого піддерева стає коренем усього дерева. В цьому випадку доведеться змінити деякі зв'язки між вершинами, оскільки якщо корені міняються ролями, то на зворотні міняються і відношення пращур-нащадок.