Дискретна математика - 2

Змістовий модуль 5. Дерева та їх застосування
Тема 2. Застосування дерев в інформаційних технологіях.

План лекції

- **Бінарне дерево пошуку**
  - **Алгоритм додавання об’єкта в дерево**
  - **Алгоритм пошуку об’єкта в дереві**
- **АВЛ-дерево**
- **Червоно-чорне дерево**
- **Дерево рішень**
- **Бектрекінг (пошук з поверненнями)**
  - **Побудова гамільтонових циклів**
  - **Розфарбування графа в *n* кольорів**
- **Каркаси. Теорема Келі**
- **Задача про мінімальний каркас. Алгоритм Краскала**

Бінарне дерево пошуку

Бінарне дерево забезпечує дуже зручний метод організації даних, у разі використання якого можна легко знайти будь-які конкретні дані чи виявити, що їх немає. Очевидно, що самий неефективний спосіб пошуку – послідовний перегляд усіх даних. Справді, якщо потрібних даних немає, то для виявлення цього потрібно переглянути весь список. Бінарне дерево пошуку дає змогу уникнути цього.

У *бінарному дереві пошуку* кожній вершині присвоєно значення, яке називають *ключем*. Ключ – це елемент якоїсь множини, на якій задано лінійний порядок. Ним може бути, наприклад, алфавітний або числовий порядок. Нагадаємо, що в лінійно впорядкованій множині будь-які два елементи можна порівняти. Лінійний порядок можуть мати теги, вказівники, файли чи інші ключі, які визначають дані. Але нас цікавитиме лише наявність якогось лінійного порядку.

Під час побудови бінарного дерева пошуку використовують його рекурсивну властивість, яку можна описати так. Кожна вершина розбиває дерево на два піддерева. Ліве піддерево містить лише ключі, менші від ключа цієї вершини, а праве – ключі, більші від ключа вершини. Ця властивість повторюється для кожної вершини (див. рис. 1 та 2).

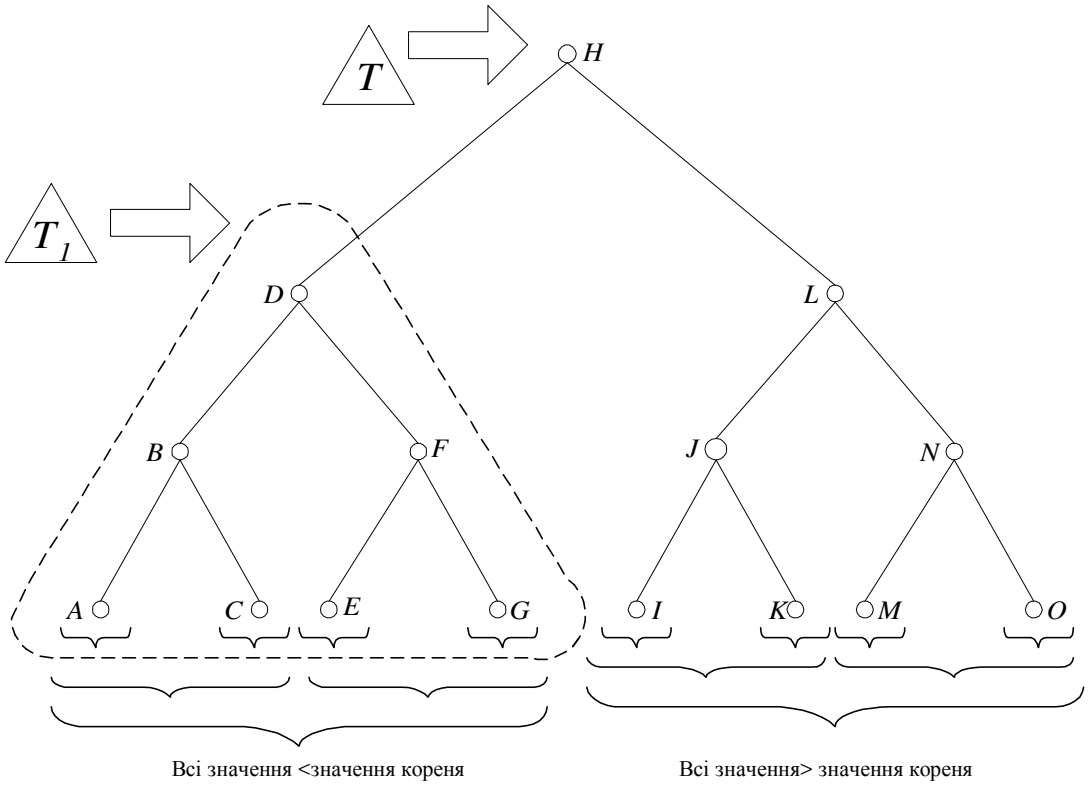

T
T1
H
D
L
B
F
J
N
A
C
E
G
I
K
M
O
Всі значення <значення кореня
Всі значення> значення кореня

Рис. 1

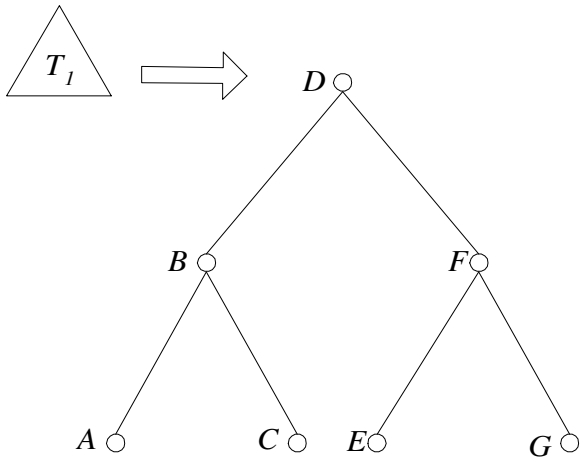

T1
D
B
F
A
C
E
G

Рис. 2

Розглянемо алгоритм додавання об’єкта до дерева пошуку. Цей алгоритм будує бінарне дерево пошуку. Почнемо з дерева, яке містить лише одну вершину, визначимо цю вершину як корінь. Перший об’єкт списку присвоюємо кореню, і це буде ключ кореня. Щоб додати новий об’єкт, виконуємо таку процедуру.

Алгоритм додавання об’єкта в дерево.

Крок 1. Почати з кореня.

Крок 2. Якщо об’єкт менший, ніж ключ у вершині, то перейти до лівого сина.

Крок 3. Якщо об’єкт більший, ніж ключ у вершині, то перейти до правого сина.

Крок 4. Повторювати кроки 2 та 3, доки не досягнемо вершини, яку не визначено (тобто її в дереві немає).

Крок 5. Якщо досягнуто невизначену вершину, то визначити (тобто додати) вершину з новим об’єктом як ключем.

Приклад. Побудуємо бінарне дерево пошуку для такого списку слів в українському алфавіті: математика, фізика, географія, зоологія, метеорологія, біологія, психологія, хімія. Процес побудови зображено на рис. 3.

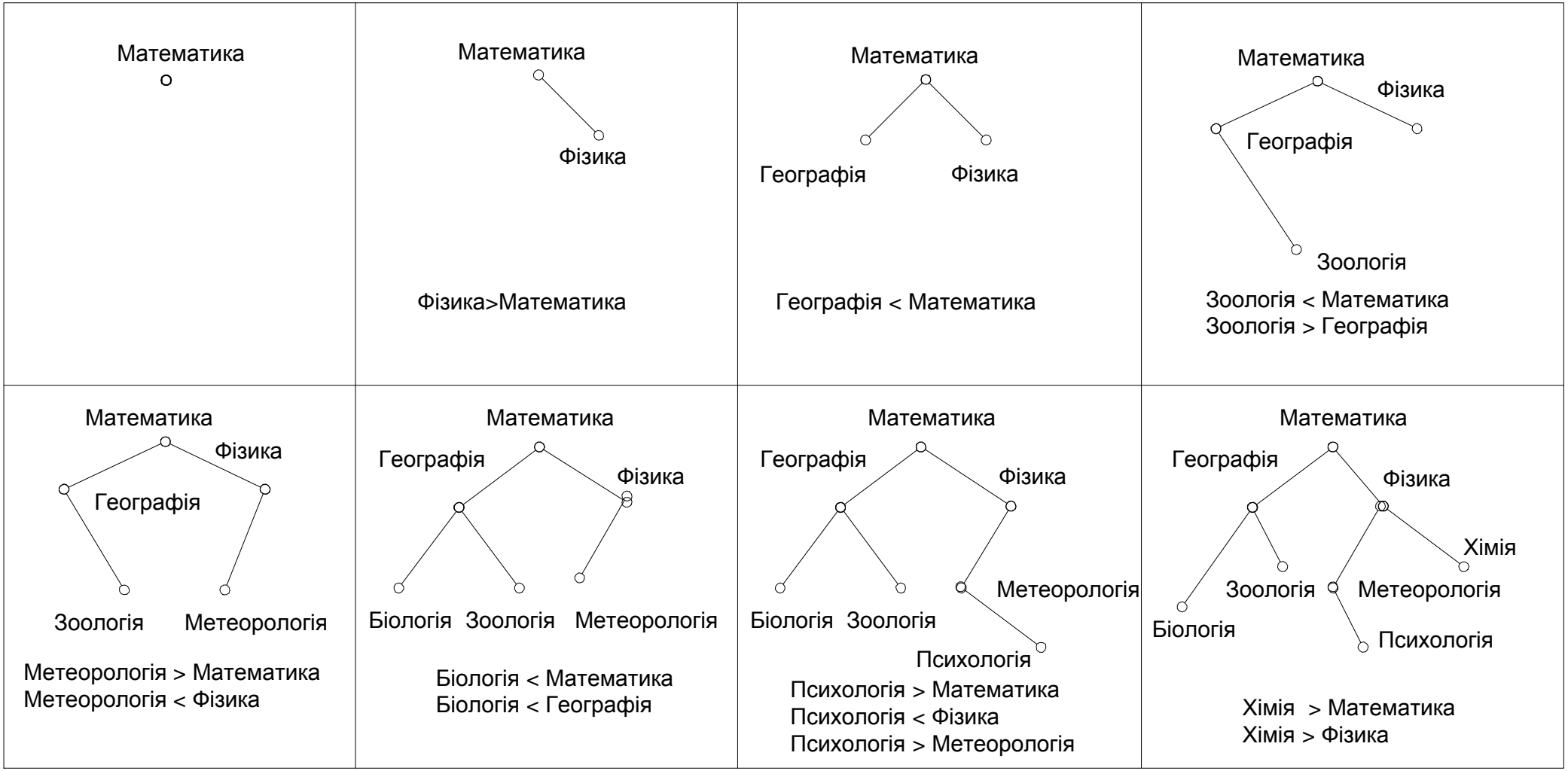


Рис. 3

Оскільки метод побудови дерева пошуку описано, легко зрозуміти, як відбувається пошук елемента в дереві. Застосовують той же підхід, тільки крім перевірки того, чи даний об’єкт більший або менший, ніж ключ у вершині, перевіряють також, чи збігається даний об’єкт із ключем у вершині. Якщо так, то процес пошуку завершено, якщо ні – описані дії повторюють. Якщо ж досягнуто вершину, яку не визначено, то це означає, що даний об’єкт не зберігається в дереві. Нижче наведено алгоритм пошуку об’єкта в бінарному дереві.

Алгоритм пошуку об’єкта в дереві.

Крок 1. Почати з кореня.

Крок 2. Якщо об’єкт менший, ніж ключ у вершині, то перейти до лівого сина.

Крок 3. Якщо об’єкт більший, ніж ключ у вершині, то перейти до правого сина.

Крок 4. Якщо об’єкт дорівнює ключу у вершині, то об’єкт знайдено; виконати потрібні дії й вийти.

Крок 5. Повторювати кроки 2, 3 та 4, доки не досягнемо вершини, яку не визначено.

Крок 6. Якщо досягнуто невизначену вершину, то даний об’єкт не зберігається в дереві; виконати потрібні дії й вийти.

Оцінимо обчислювальну складність алгоритмів включення та пошуку (локалізації) об'єкта в бінарному дереві пошуку. Припустимо, що є бінарне дерево пошуку T для списку з n об'єктів. Із дерева T утворимо повне бінарне дерево U, для чого додамо нові вершини (без ключів) так, щоб кожна вершина з ключем мала двох синів. Це показано на рис. 4, де вершини без ключів позначено подвійними кружечками.

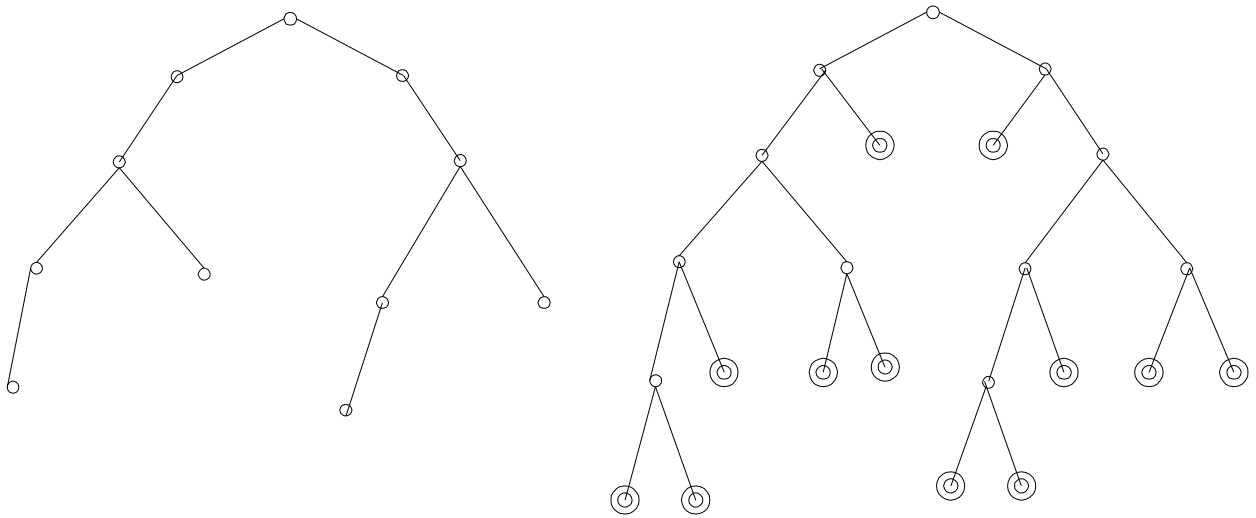


Рис. 4

Коли це зроблено, можна легко локалізувати об'єкт або додати новий об'єкт як ключ без додавання вершини. Найбільша кількість порівнянь, потрібних для додавання нового об'єкта, дорівнює довжині найдовшого шляху в U від кореня до листка, тобто дорівнює висоті дерева. Внутрішні вершини дерева U – це (всі) вершини дерева T, отже, U має n внутрішніх вершин. За теоремою одержимо, що кількість листків у дереві U дорівнює $2n+1-n=n+1$. Згідно з наслідком із теореми про зв'язок між висотою m-арного дерева і кількістю його листків, висота дерева U більша або дорівнює $\lceil \log(n+1) \rceil$. Отже, потрібно щонайменше $\lceil \log(n+1) \rceil$ порівнянь, щоб додати чи локалізувати довільний об'єкт. Якщо дерево збалансоване, то його висота дорівнює $\lceil \log(n+1) \rceil$. Отже, у цьому разі для додавання чи локалізації об'єкта потрібно не більше ніж $\lceil \log(n+1) \rceil$ порівнянь.

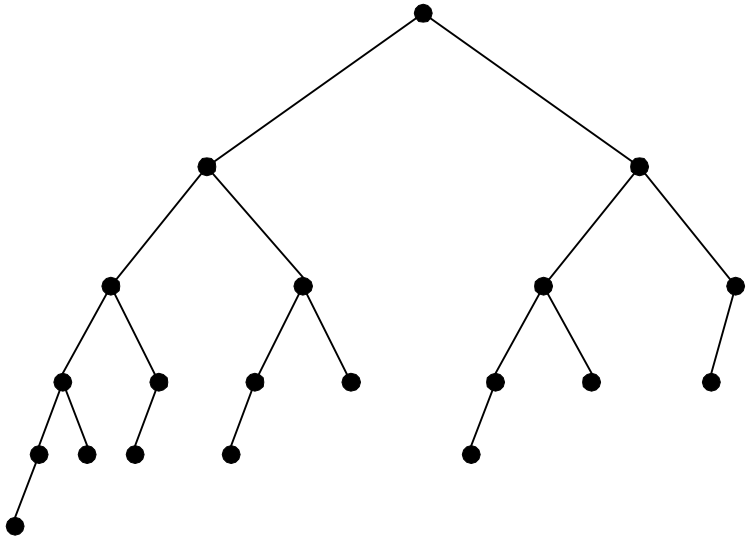


Рис. 5

Бінарне дерево пошуку може розбалансуватись унаслідок додавання нових об'єктів. Тому важливі алгоритми *балансування* (тобто відновлення збалансованості). Проте, процедура додавання об'єкта, яка відновлює збалансоване дерево, навряд чи завжди доцільна, бо відновлення збалансованості дерева після випадкового додавання – досить складна операція.

Тому розглядають збалансованість із дещо послабленими вимогами; зокрема, означають *АВЛ-дерево* – бінарне дерево, у якому висоти двох піддерев кожної з його вершин відрізняються не більше ніж на одиницю. Таке означення збалансованості ввели 1962 р. Г. Адельсон-Вельський та Є. Ландіс. Приклад АВЛ-дерева наведено на рис.5. Для АВЛ-дерев є дуже ефективна процедура балансування. Зазначимо, що висота АВЛ-дерева незалежно від кількості вершин ніколи не перевищує висоту збалансованого дерева більше, ніж на 45%. Якщо позначити висоту АВЛ-дерева з n вершинами як $h(n)$, то $\log(n+1) \le h(n) < 1.44 \cdot \log(n+2) - 0.328$.

Де використовують:

АВЛ-дерева використовують, коли потрібен швидкий доступ до елементів дерева. Можуть використовуватися для сортування даних.

Нині АВЛ-дерева рідко використовують на практиці через великі обчислювальні витрати при вставленні й видаленні елементів. У якості альтернативи даній структурі найчастіше використовують **червоно-чорні** дерева. Вони володіють хорошою швидкістю пошуку, як і АВЛ-дерева, але при цьому рідше виконують процедуру балансування.

Збалансованість даного типу дерева досягається введенням додаткових правил формування дерева, а також додаванням спеціального атрибута вершини – кольору. Цей атрибут може приймати одне з двох можливих значень: «чорний» або «червоний». Правила формування червоно-чорного дерева:

1. Кожна вершина пофарбована або в «червоний», або в «чорний» колір;
2. Будь-яка «червона» вершина має тільки «чорних» синів;
3. Всі шляхи від кореня до листків мають однакове число чорних вершин.

При цьому для зручності, листками червоно-чорного дерева вважаються фіктивні «нульові» вершини, що не містять даних. Приклад наведено на рис. 6.

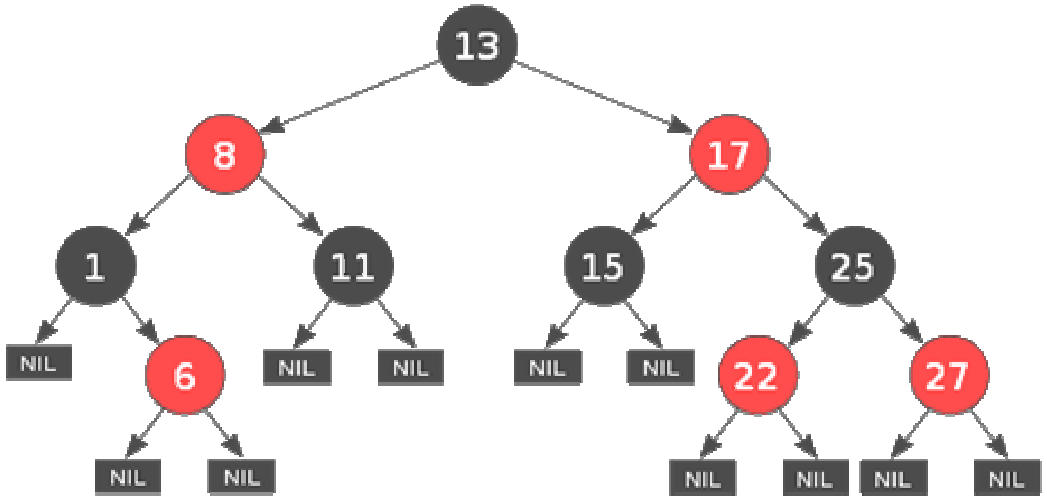


Рис. 6.

Де використовують:

- Контейнер map в реалізації бібліотеки STL мови C ++;
- Клас TreeMap мови Java;
- Багато інших реалізацій асоціативного масиву в різних бібліотеках;
- У ядрі Linux (для організації черг запитів, в ext3).

Порівняння червоно-чорних і АВЛ-дерев.

АВЛ-дерева рекомендується використовувати, коли потрібний швидкий пошук елемента в фіксованих даних. Якщо ж дані динамічні, тобто багато операцій вставки і видалення, то краще використовувати червоно-чорні дерева.

Дерево рішень

Дерево рішень використовують як математичну модель у задачах класифікації та прогнозування. До них відносяться проблеми медичного діагностування, оцінювання кредитного ризику, визначення тенденцій на фінансових ринках тощо.

Дерева рішень використовують техніку „поділяй і володарюй” для послідовного поділу простору пошуку (тут *простір пошуку* – це множина об’єктів, серед яких шукають потрібний).

Дерево рішень – це кореневе дерево, у якому кожну внутрішню вершину позначено запитанням. Ребра, які виходять з кожної такої вершини, подають можливі відповіді на запитання, асоційоване з цією вершиною. Кожний листок являє собою прогноз розв’язку розглядуваної проблеми.

Бінарне дерево пошуку – частковий випадок дерева рішень. Зазначимо, що дерево рішень – НЕ обов’язково бінарне.

Приклад. Серед восьми монет є одна фальшива, вона має меншу вагу. Потрібно знайти цю монету послідовністю зважувань на балансових терезах. Під час кожного зважування на балансових терезах є три можливі відповіді на запитання „Яка чаша легша?” А саме, чаші мають однакову вагу, перша чаша легша або друга чаша легша. Отже, дерево рішень для послідовності зважувань 3-арне. Це дерево має щонайменше вісім листків, бо є вісім можливих варіантів розв’язку (кожна з восьми наявних монет може бути фальшивою). Найменша кількість зважувань, необхідних для виявлення фальшивої монети, дорівнює висоті дерева рішень. Із наслідку з теореми випливає, що висота дерева рішень складає щонайменше $\lceil \log_3 8 \rceil = 2$. Отже, потрібно не менше двох зважувань. У цьому прикладі фальшиву монету можна виявити, використавши два зважування. Розв’язок подано на рис. 7. Занумеруємо монети числами 1, 2, ..., 8. Розділимо монети на три групи: першу групу складуть монети з номерами 1, 2, 3, другу – монети з номерами 4, 5, 6, а третю – монети з номерами 7 та 8. Порівнюємо вагу монет першої й другої груп. Якщо одна із цих груп виявиться легшою, то це означає, що фальшива монета є в цій групі, і її можна виявити другим зважуванням. У разі балансу терезів фальшива одна з монет третьої групи, що також можна виявити другим зважуванням. Кількість зважувань дорівнює висоті дерева, тобто двом.

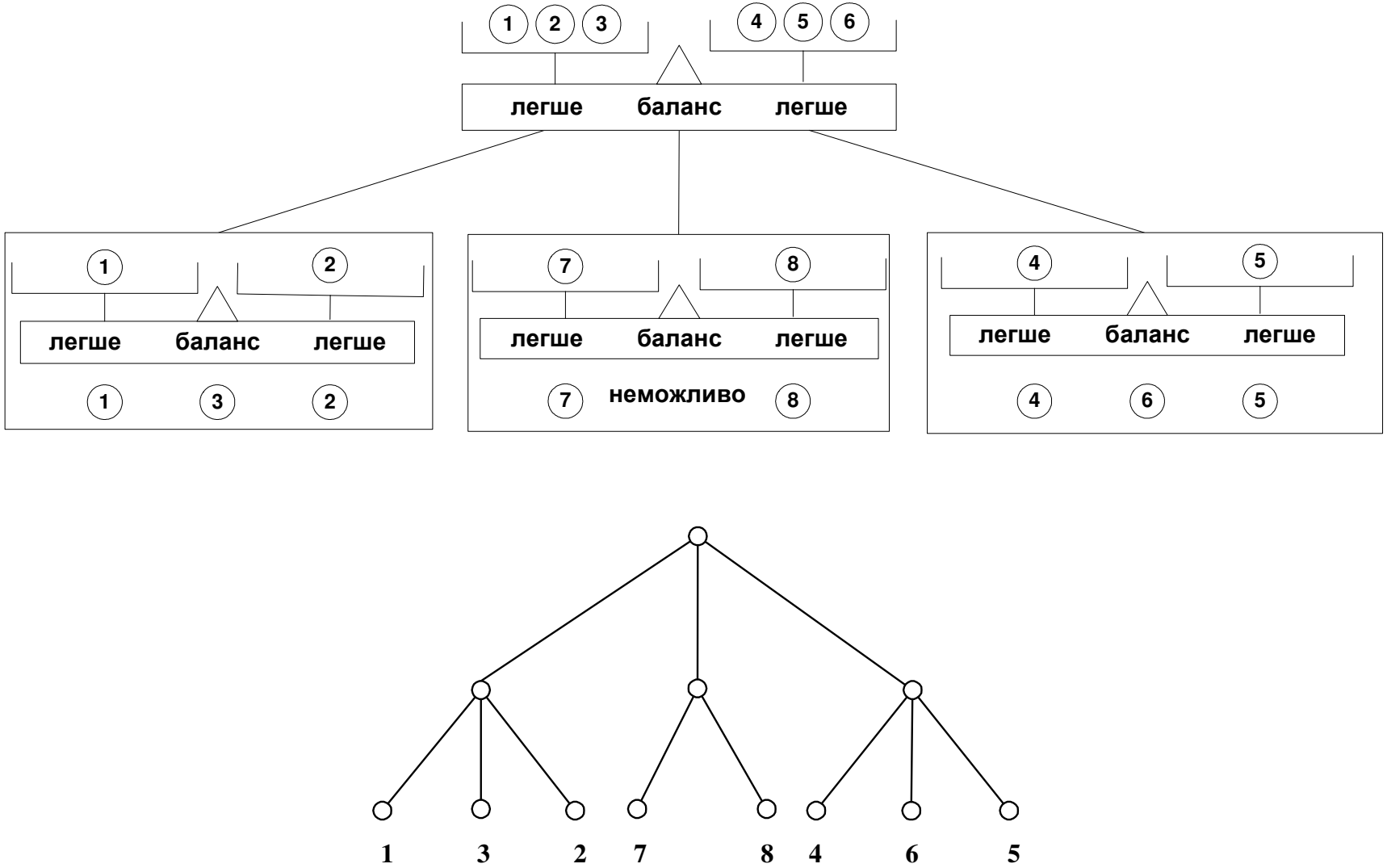


Рис. 7

Підхід, який ґрунтується на деревах рішень, особливо корисний для задач класифікації. За цією технікою дерево рішень будують, щоб змоделювати процес класифікації.

Оцінка складності алгоритму сортування. Нині досліджено багато алгоритмів сортування. Для вирішення питання, чи є алгоритм сортування ефективним, потрібно визначити його обчислювальну складність. Для знаходження нижньої оцінки алгоритмів сортування , заснованих на попарному порівнянні, як модель можна використати дерево рішень. Зазначимо, що за наявності n елементів існує $n!$ можливих упорядкувань, бо кожна із $n!$ перестановок може виявитись коректним упорядкуванням. Алгоритм сортування можна подати бінарним деревом рішень, у якому кожна внутрішня вершина відповідає порівнянню двох елементів. Кожний листок цього дерева подає одну з $n!$ перестановок n елементів.

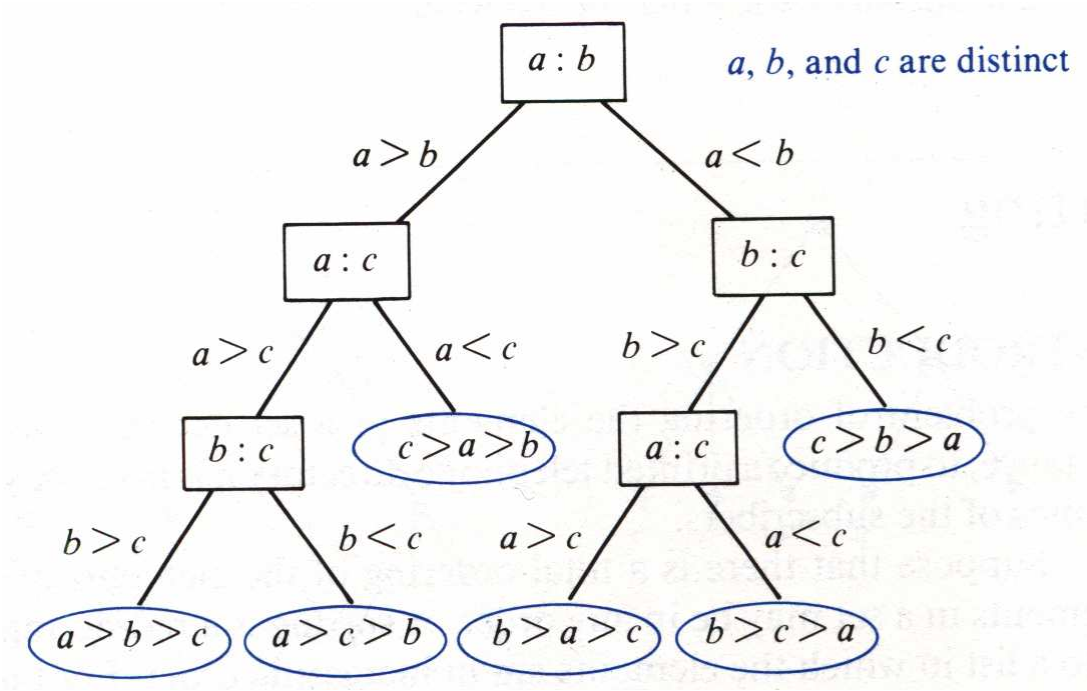


Рис. 8

На рис. 8 подано дерево, яке упорядковує елементи a, b, c. Складність сортування вимірюється кількістю операцій порівняння. Найбільша кількість операцій порівняння, яка завжди потрібна для сортування n елементів, дає характеристику складності алгоритму. Найбільша кількість використаних порівнянь відповідає найдовшому шляху в дереві рішень, яке репрезентує процедуру сортування. Цей шлях дорівнює висоті дерева. Оскільки висота бінарного дерева з $n!$ листками оцінюється знизу як $\lceil \log n! \rceil$, то щонайменше $\lceil \log n! \rceil$ порівнянь потрібно для сортування n елементів. Зазначимо, що $\lceil \log n! \rceil$ є величина $O(\lceil n \log n \rceil)$. (Це пропонується довести як вправу.)

Бектрекінг (пошук із поверненнями)

Опишемо загальний метод, який дає змогу значно зменшити обсяг обчислень в алгоритмах типу „повного перебору всіх можливостей". Щоб застосувати цей метод, розв'язок задачі повинен мати вигляд послідовності $(x_1,\ldots,x_n)$. Головна ідея методу полягає в тому, що розв'язок будують поступово, починаючи з порожньої послідовності λ (довжиною 0). Загалом, якщо є частковий (неповний) розв'язок $(x_1,\ldots,x_i)$, де $i<n$, то намагаємося знайти таке допустиме значення x_{i+1}, що не заперечується можливість продовження $(x_1,\ldots,x_i,x_{i+1})$ до повного розв'язку. Якщо таке допустиме, але ще не використане значення x_{i+1} існує, то долучаємо цю нову компоненту до часткового розв'язку та продовжуємо процес для послідовності $(x_1,\ldots,x_i,x_{i+1})$. Якщо такого значення x_{i+1} немає, то повертаємося до попередньої послідовності $(x_1,\ldots, x_{i-1})$ і продовжуємо процес, шукаючи нове, іще не використане значення x_i', – звідси назва „*бектрекінг*" (англ. backtracking – пошук із поверненнями).

Роботу цього алгоритму можна інтерпретувати як процес обходу певного дерева. Кожна вершина цього дерева природно відповідає якійсь послідовності $(x_1,\ldots,x_i)$, причому вершини, які відповідають послідовностям вигляду $(x_1,\ldots,x_i,y)$, – сини цієї вершини. Корінь дерева відповідає порожній послідовності.

В алгоритмі бектрекінг виконується обхід цього дерева пошуком углиб. Окрім того, задають предикат P, означений на всіх вершинах дерева. Якщо $P(v)$=F процес обходу відкидає розгляд вершин піддерева з коренем у вершині v, зменшуючи тим самим обсяг перебору. Сам предикат $P(v)$ набуває значення F тоді, коли стає зрозумілим, що послідовність $(x_1,\ldots,x_i)$, яка відповідає вершині v, ніяким способом не можна добудувати до повного розв'язку.

Проілюструємо застосування алгоритму бектрекінг на конкретних прикладах.

Побудова гамільтонових циклів у графі. Починаємо з довільної вершини. Будуємо шлях без повторення вершин, доки це можливо. Якщо вдалося пройти всі вершини, то перевіряємо, чи існує ребро, що з'єднує останню й початкову вершини цього шляху. Якщо описаний процес у певний момент неможливо продовжити, то повертаємося на одну вершину назад, і намагаємося продовжити побудову шляху (без повторення вершин) іншим способом.

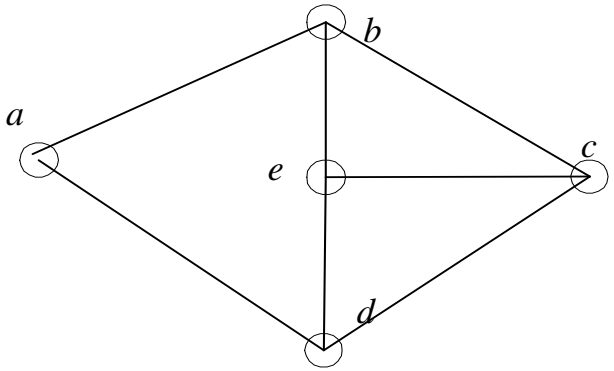


Рис. 9

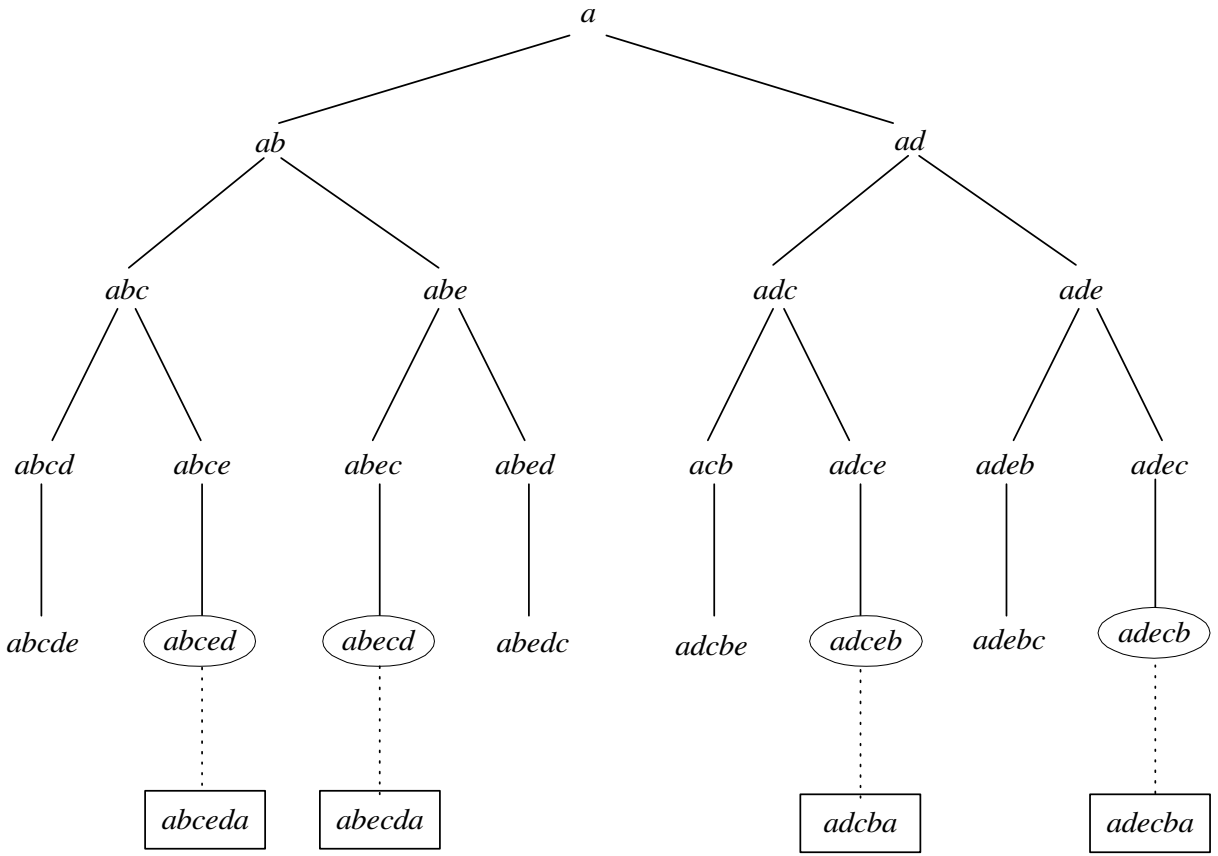


Рис. 10

Отже, замість побудови та аналізу 5!=120 послідовностей довжиною 5, елементами котрих є вершини графа на рис. 9, ми розглянули всього 23 послідовності довжиною від 1 до 5 (рис. 10).

Розфарбування графа в *n* кольорів. Нехай вершини графа позначено як *a*, *b*, *c*,… Спочатку розфарбуємо вершину *a* в колір 1. Після цього розфарбуємо вершину *b* в колір 1, якщо *b* не суміжна з вершиною *a*, а ні, то розфарбуємо вершину *b* в колір 2. Перейдемо до третьої вершини *c*. Використаємо для вершини *c* колір 1, якщо це можливо, а ні, то колір 2, якщо це можливо. Тільки якщо жоден із кольорів 1 і 2 не можна використати, розфарбуємо вершину *c* в колір 3. Продовжуємо цей процес, доки це можливо, використовуючи один з *n* кольорів для кожної нової вершини, причому завжди братимемо перший можливий колір зі списку кольорів. Досягнувши вершини, яку не можна розфарбувати в жоден з *n* кольорів, повертаємося до останньої розфарбованої вершини, відміняємо її колір і розфарбовуємо в наступний можливий колір зі списку. Якщо й це неможливо, то повертаємось іще до попередньої вершини, відміняємо її колір і намагаємося розфарбувати в новий колір, наступний можливий зі списку. Цей процес продовжуємо аналогічно. Якщо розфарбування в *n* кольорів існує, то така процедура дає змогу знайти його. На рис. 11 зображено граф і процес розфарбування в три кольори його вершин із використанням алгоритму бектрекінг.

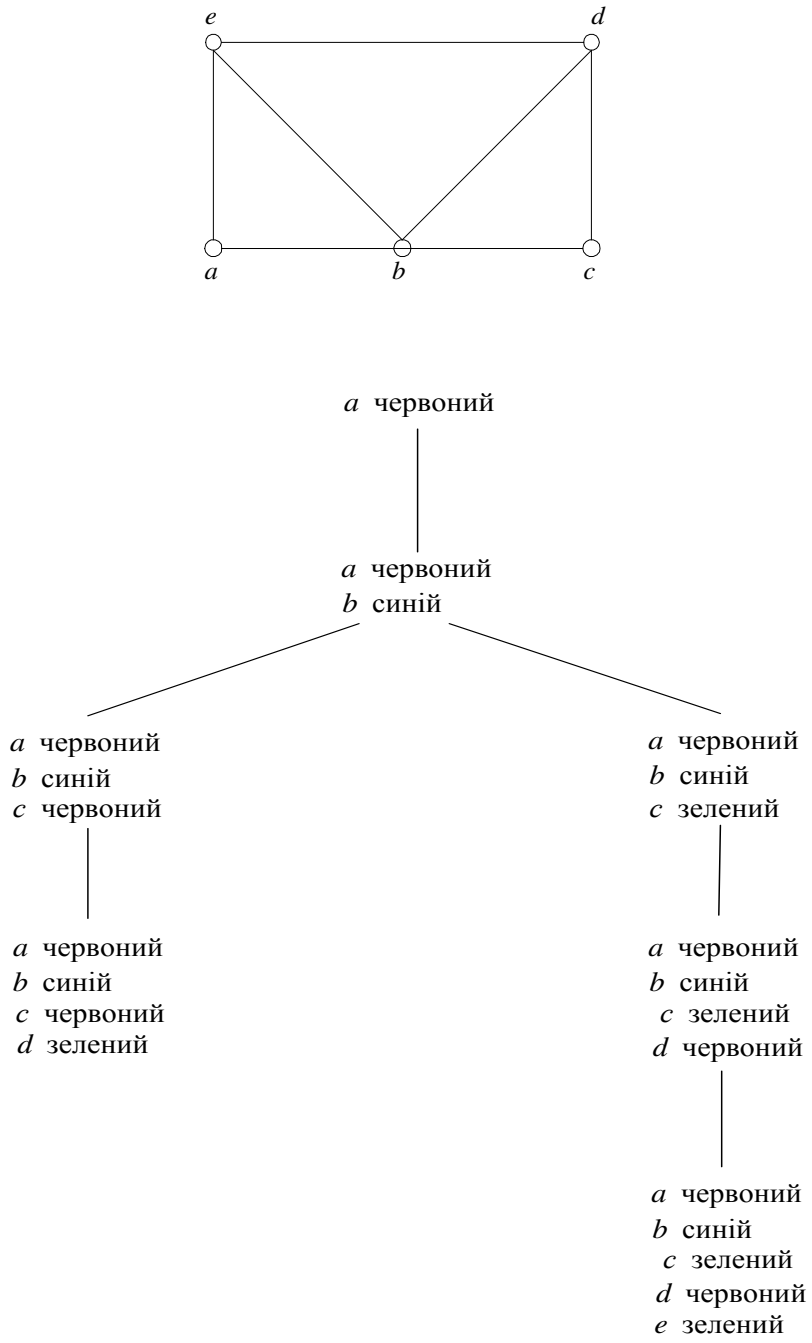

e
d
a
b
c
a червоний
a червоний
b синій
a червоний
b синій
c червоний
a червоний
b синій
c зелений
a червоний
b синій
c червоний
d зелений
a червоний
b синій
c зелений
d червоний
a червоний
b синій
c зелений
d червоний
e зелений

Рис. 11

Каркаси

Нехай G – простий граф. *Каркасом* графа G називають його підграф, який являє собою дерево та містить усі вершини графа G. Інакше кажучи, каркас – це каркасний підграф, який являє собою дерево.

Теорема. Простий граф зв'язний тоді й лише тоді, коли він має каркас.

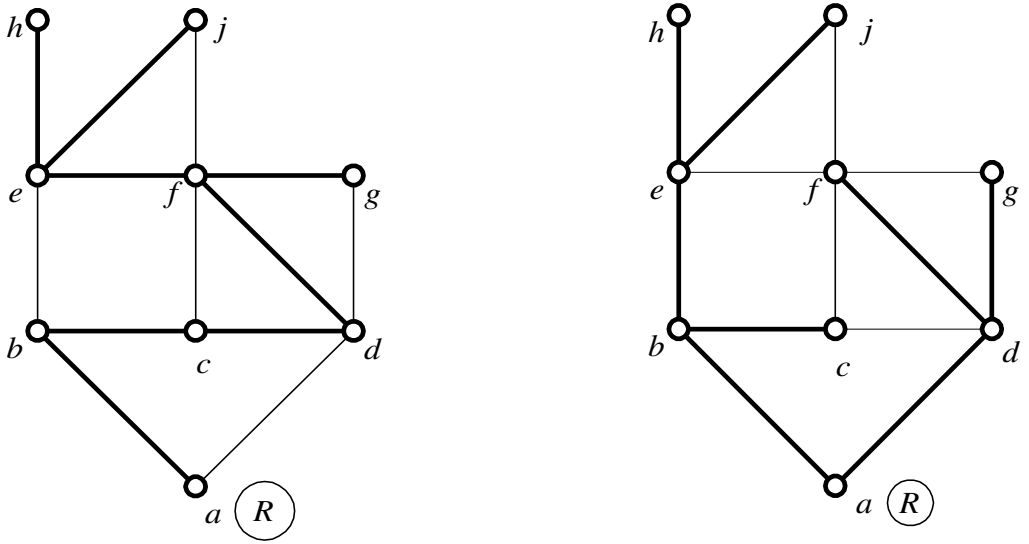
Доведення. Необхідність. Нехай граф G зв'язний. Якщо G – не дерево, то він має простий цикл. Вилучимо ребро з одного з простих циклів. Одержаний підграф має на одне ребро менше, але містить усі вершини графа G і зв'язний. Якщо цей підграф не дерево, то він має простий цикл, отже, як і перед цим, вилучимо ребро з простого циклу. Повторюватимемо ці дії доти, доки залишатимуться прості цикли. Це можливо, бо граф має скінченну кількість ребер. Процес закінчимо, коли не залишиться жодного простого циклу. Одержимо дерево, бо після вилучення ребер граф залишався зв'язним. Це дерево – каркас, бо містить усі вершини графа G.

Достатність. Тепер припустимо, що граф G має каркас T. Дерево T містить усі вершини графа G. Більше того, в T є шлях між будь-якими двома його вершинами. Оскільки T – підграф графа G , то й у графі G існує шлях між будь-якою парою його вершин. Теорему доведено.

Доведення останньої теореми дає алгоритм для знаходження каркаса процедурою вилучення ребер із простих циклів. Нехай G має n вершин та m ребер. У такому разі, очевидно, вилучається $\gamma(G) = m - (n - 1) = m - n + 1$ ребер.

Число $\gamma(G)$ називають *цикломатичним числом* графа G. З теореми про оцінку кількості ребер простого графа випливає, що $\gamma(G) \geq 0$. Цикломатичне число певною мірою є числовою характеристикою зв'язності графа; цикломатичне число дерева дорівнює 0.

Побудова каркаса є поширеною задачею. Описаний вище алгоритм, який полягає у вилученні ребер із простих циклів, неефективний для комп'ютерної реалізації. Для його виконання потрібно ідентифікувати всі прості цикли. Ефективний із погляду комп'ютерної реалізації алгоритм побудови каркаса – це послідовний добір ребер у каркас. Це можна зробити за допомогою обходу графа G як пошуком углиб, так і пошуком ушир. Під час виконання цих алгоритмів природним чином будують каркас (ребра каркаса позначено потовщеними лініями). Якщо до протоколу обходу графа додати четвертий стовпчик, то там можна накопичувати ребра, які під час роботи алгоритму позначають потовщеною лінією, і, отже, відбирають у каркас.



а *б*

Рис. 12

Приклад. На рис. 12, *а* зображено каркас, який було одержано пошуком углиб, а на рис. 12, *б* – пошуком ушир. Початкова вершина в обох випадках – *a*.

Теорема. Нехай T – каркас графа G, побудований пошуком ушир, починаючи з вершини a. Тоді шлях з a до довільної вершини v в T – найкоротший шлях від a до v в графі G.

Доведення. Доведення ґрунтується на аналізі алгоритму пошуку вшир у графі G. Пропонуємо виконати його як вправу.

Нагадаємо, що граф з n вершинами називають позначеним, якщо його вершинам приписано мітки, наприклад, числа 1, 2, …, n. Якщо розглянути повний позначений граф K_n, то скільки каркасів можна побудувати? Іншими словами, скільки є позначених дерев з n вершинами? Відповідь на це запитання дає теорема Келі.

Теорема 4.8 (Келі, 1897 р.). Кількість позначених дерев з n вершинами дорівнює n^{n-2}.

Теорема Келі дає кількість позначених дерев з n вершинами, або, що те саме, кількість каркасів у повному позначеному графі K_n.

Задача про мінімальний каркас.

Тепер розглянемо одну важливу задачу, пов'язану з ідеєю оптимізації.

У реальних задачах на графах часто потрібно брати до уваги додаткову інформацію – фактичну віддаль між окремими пунктами, вартість проїзду, час проїзду тощо. Для цього використовують поняття зваженого графа.

Зваженим називають граф, кожному ребру e якого приписано дійсне число $w(e)$. Це число називають *вагою* ребра e. Аналогічно означають *зважений орієнтований* граф: це такий орієнтований граф, кожній дузі e якого приписано дійсне число $w(e)$, називане *вагою* дуги.

Розглянемо три способи зберігання простого зваженого графа $G=(V, E)$ в пам'яті комп'ютера. Нехай $|V| = n, |E| = m$.

Перший – подання графа *матрицею ваг* W, яка являє собою аналог матриці суміжності. Її елемент $w_{ij} = w(v_i, v_j)$, якщо ребро $\{v_i, v_j\} \in E$ (у разі орієнтованого графа – дуга $(v_i, v_j) \in E$). Якщо ж ребро $\{v_i, v_j\} \notin E$ (або дуга $(v_i, v_j) \notin E$), то $w_{ij} = 0$ чи $w_{ij} = \infty$ залежно від розв'язуваної задачі.

Другий спосіб – подання графа списком ребер. Для зваженого графа під кожний елемент списку E можна відвести три комірки – дві для ребра й одну для його ваги, тобто всього потрібно $3m$ комірок.

Третій спосіб – подання графа списками суміжності. Для зваженого графа кожний список $Adj[u]$ містить окрім вказівників на всі вершини v з множини $\Gamma(u)$ ще й числа $w(u, v)$.

Нехай G – зв'язний зважений граф.

Потрібно описати алгоритм побудови каркаса T, у якого сума ваг ребер $M(T)=\Sigma w(e)$ якнайменша (суму Σ беруть за всіма ребрами дерева T). Каркас T називають *мінімальним*. Розв'язати цю задачу дає змогу *алгоритм Краскала* (J. Kruskal).

Алгоритм Краскала

Нехай G – зв'язний зважений граф з n вершинами та m ребрами. Виконати такі дії.

1. Вибирати ребро e_1, яке має в графі G найменшу вагу.
2. Визначити послідовність ребер $e_2, e_3, \ldots, e_{n-1}$; на кожному кроці вибирати відмінне від попередніх ребро з найменшою вагою й таке, що не утворює простих циклів з уже вибраними ребрами. Одержане дерево T із множиною ребер $ET=\{e_1, e_2, e_3, \ldots, e_{n-1}\}$ – мінімальний каркас графа G.

Розглянемо одну з можливих реалізацій алгоритму Краскала.

Крок 1. Упорядкувати множину ребер у порядку зростання ваг: e_1, e_2, e_3, …, e_m.

Крок 2. Утворити розбиття множини вершин на одноелементні підмножини: $\{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$.

Крок 3. Вибирати таке чергове ребро з упорядкованої послідовності ребер, що його кінці містяться в різних множинах розбиття (це забезпечить відсутність простих циклів). Якщо вибрано ребро $e_i = \{v, w\}$, то множини розбиття, які містять вершини v та w, об'єднують в одну множину.

Крок 4. Якщо вже вибрано (n–1) ребро (у такому разі всі підмножини розбиття об'єднаються в одну), то зупинитись, бо вибрані ребра утворюють мінімальний каркас. Інакше перейти до кроку 3.

Алгоритм Краскала належить до *жадібних* алгоритмів. Так називають алгоритми оптимізації, які на кожному кроці вибирають найкращий із можливих варіантів.

Приклад. На рис. 13 зображено граф, ребра якого пронумеровано за зростанням ваг. Мінімальний каркас виділено потовщеними лініями. Процес відбору ребер наведено в таблиці. Мінімальний каркас утворюють ребра e_1, e_2, e_3, e_5, e_8, e_{11}.

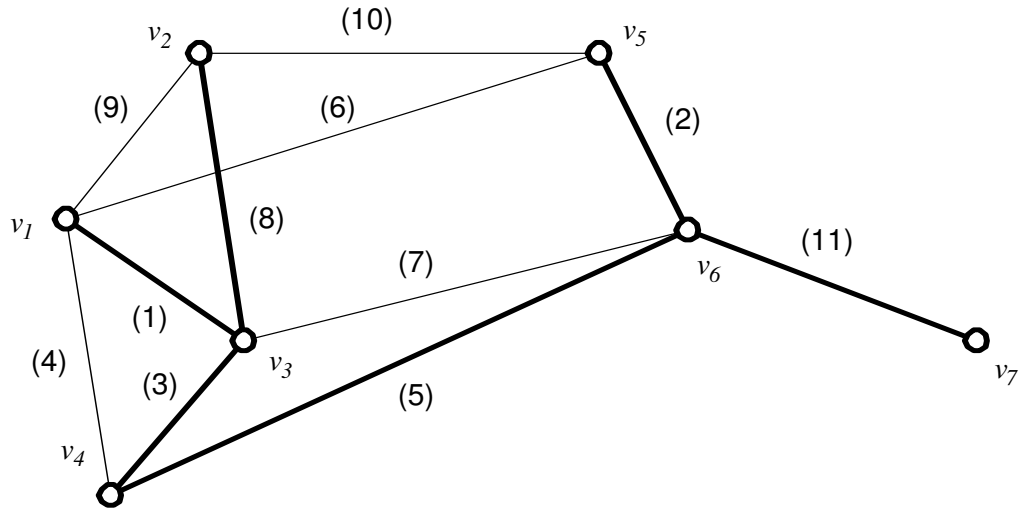


Рис. 13

Таблиця відбору ребер у мінімальний каркас

Ребро	Розбиття множини вершин	Відбір ребра у мінімальний каркас
$e_1=\{v_1,v_3\}$	$\{\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_7\}\}$	e_1
$e_2=\{v_5,v_6\}$	$\{\{v_1,v_3\}, \{v_2\}, \{v_4\}, \{v_5\}, \{v_6\}, \{v_7\}\}$	e_2
$e_3=\{v_3,v_4\}$	$\{\{v_1,v_3\}, \{v_2\}, \{v_4\}, \{v_5,v_6\}, \{v_7\}\}$	e_3
$e_4=\{v_1,v_4\}$	$\{\{v_1,v_3,v_4\}, \{v_2\}, \{v_5,v_6\}, \{v_7\}\}$	–
$e_5=\{v_4,v_6\}$	$\{\{v_1,v_3,v_4\}, \{v_2\}, \{v_5,v_6\}, \{v_7\}\}$	e_5
$e_6=\{v_1,v_5\}$	$\{\{v_1,v_3,v_4,v_5,v_6\}, \{v_2\}, \{v_7\}\}$	–
$e_7=\{v_3,v_6\}$	$\{\{v_1,v_3,v_4,v_5,v_6\}, \{v_2\}, \{v_7\}\}$	–
$e_8=\{v_2,v_3\}$	$\{\{v_1,v_3,v_4,v_5,v_6\}, \{v_2\}, \{v_7\}\}$	e_8
$e_9=\{v_1,v_2\}$	$\{\{v_1,v_2,v_3,v_4,v_5,v_6\}, \{v_7\}\}$	–
$e_{10}=\{v_2,v_5\}$	$\{\{v_1,v_2,v_3,v_4,v_5,v_6\}, \{v_7\}\}$	–
$e_{11}=\{v_6,v_7\}$	$\{\{v_1,v_2,v_3,v_4,v_5,v_6\}, \{v_7\}\}$	e_{11}
	$\{\{v_1,v_2,v_3,v_4,v_5,v_6,v_7\}\}$	