МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Волинський національний університет імені Лесі Українки

Кафедра комп'ютерних наук та кібербезпеки

Т.О. Гришанович

КУРС ЛЕКЦІЙ ІЗ ДИСЦИПЛІНИ "АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ"

для студентів спеціальності 014.09 Середня освіта (Інформатика)
першого (бакалаврського) рівня

Луцьк 2021

УДК 004.421:004.422.63(075.8)
Г 85

Рекомендовано до видання науково-методичною радою
Волинського національного університету імені Лесі Українки
(протокол № 4 від 14 грудня 2021 р.)

Рецензенти:
Ройко Л. Л. – кандидат педагогічних наук, доцент, доцент кафедри загальної математики та методики навчання інформатики Волинського національного університету імені Лесі Українки;
Ліщина Н. М. – кандидат технічних наук, доцент, завідувач кафедри комп'ютерних наук Луцького національного технічного університету.

Курс лекцій з дисципліни «Алгоритми та структури даних» для студентів спеціальності 014 Середня освіта. Інформатика [Електронний ресурс] / Т.О. Гришанович; ВНУ імені Лесі Українки. Електронні текстові данні (1 файл: 1,33 МБ). Луцьк : ВНУ імені Лесі Українки, 2021. – 110 с.

Курс лекцій містить теоретичний матеріал, що стосується алгоритмів, їх проєктування, розробки та аналізу, а також структур даних, їх класифікації, особливостей обробки та використання при розробці алгоритмів. Матеріал структуровано за темами, до кожної із тем наведено питання для самоконтролю.

Видання призначене для студентів, що навчаються за спеціальністю 014 Середня освіта. Інформатика.

ЗМІСТ

Лекція 1. Поняття задачі. Аналіз процесу розв'язання задачі. Поняття алгоритму, даних та аналізу алгоритму. Основні етапи розробки алгоритмів

«Програма = Алгоритм + Структура даних».

Ця формула дуже точно відображає той факт, що для розробки ефективної програми важливим являється як вибір (або розробка) ефективного алгоритму, так і вибір структури даних, до яких такий алгоритм буде застосовано. Кожна із таких складових частин програми не може бути виключена, оскільки правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів, а вибір ефективних алгоритмів у свою чергу дозволяє оптимізувати використання машинного часу та пам'яті.

Таким чином одним із ключових понять програмування являється поняття алгоритмізації, оскільки, як було сказано вище, жодна програма не може бути написана без попередньої розробки алгоритму. Тому вивчення структур даних і алгоритмів, які є фундаментом сучасної методології розробки програм, оволодіння методами розробки алгоритмів та їх аналізу є одним із основних завдань фахівця з комп'ютерних наук та інформаційних технологій. Адже він повинен вміти орієнтуватись в існуючих алгоритмах, розуміти їх та аналізувати, видозмінювати відповідно до нових сфер використання. Ці знання дозволяють не лише створювати нові програми, але й є основою універсального мисленнєвого апарату, який необхідний у довільній сфері діяльності. Крім вищезазначеного, слід зазначити, що процеси розробки і аналізу алгоритмів сприяють розвитку аналітичного стилю мислення, яке є необхідним у довільній сфері діяльності людини.

Слово «алгоритм» походить від імені узбецького вченого аль Хорезмі, який у IX ст. розробив правила виконання чотирьох арифметичних дій над числами в десятковій системі числення. Сукупність цих правил у Європі почали називати «алгоризм». Пізніше цей термін перетворився у «алгоритм» і ним називали правила розв'язування різних задач.

Першим алгоритмом, що дійшов до нас на інтуїтивному розумінні як скінченна послідовність елементарних дій, які розв'язують поставлену задачу, вважають запропонований Евклідом у III ст. алгоритм знаходження найбільшого спільного дільника 2 чисел.

Аж до 30-х рр XX ст поняття алгоритму мало швидше методологічне значення. Під алгоритмом розуміли скінченну сукупність точно сформульованих правил, які дають змогу розв'язувати задачі певного класу. Таке визначення алгоритму не є строгим, а скоріше інтуїтивним, оскільки спирається на інтуїтивні поняття (правило, клас задач). У 20-х рр XX ст задача строгого визначення алгоритму стала однією із центральних математичних проблем.

Початковою точкою відліку сучасної теорії алгоритмів можна вважати теорему про неповноту символьних логік, доведену німецьким математиком К. Гьоделем у 1931 році У цій роботі з'ясовано, які математичні проблеми не можуть бути розв'язані алгоритмами певного класу. Загальність результатів К. Геделя пов'язана із питаннями про те, чи тотожний використовуваний ним клас алгоритмів з класом усіх алгоритмів в інтуїтивному розумінні цього терміну. Зазначена праця дала поштовх до пошуку й аналізу різних формалізацій поняття «алгоритм».

Перші фундаментальні праці з теорії алгоритмів опубліковані в середині 30-х років XX ст. Аланом Тьюрінгом, Алоїзом Черчем, Емілем Постом. Запропоновані ними машини Тьюрінга, Поста і клас рекурсивних функцій Черча стали першими формальними описами алгоритму, які використовувалися як строго визначені моделі обчислень. Сформульовані гіпотези Черча-Тьюрінга та Поста постулювали еквівалентність запропонованих ними моделей обчислень як формальних систем та інтуїтивного поняття алгоритму.

Алгоритм – це формально описана обчислювальна процедура, що отримує вхідні дані, які називають також входом алгоритму, або його аргументом, і вдає результат обчислень на вихід.

Надалі ми наведемо ще декілька означень алгоритму, але надалі у посібнику алгоритмом будемо вважати формально описану процедуру, що має вхідні та вихідні дані, як сформульовано в даному означенні.

Для того, аби надалі оперувати такими поняттями як «важко розв'язувані задачі», «еквівалентні задачі», «еквівалентні алгоритми розв'язання задачі», узгодимо деяку термінологію.

Розпочнемо із поняття задачі. Під масовою задачею (або просто задачею) розуміють деяке загальне питання, на яке слід дати відповідь. У загальному випадку задача містить декілька параметрів або вільних змінних, конкретні значення яких не є визначеними. Задача *N* визначається наступною інформацією:

1. Загальним списком усіх її параметрів.

2. Формулюванням тих властивостей, яким повинна задовольняти відповідь

або, іншими словами, розв'язання задачі.

Індивідуальна задачі *I* отримується із загальної задачі *N*, якщо усім параметрам задачі *N* присвоїти конкретні значення.

Як приклад, розглянемо класичну задачу про комівояжера. Параметри цієї масової задачі складаються із набору «міст» $C\ \{c_1, c_2 \ldots c_n\}$ та відстаней $d(c_i, c_j)$ між кожною парою міст c_i та c_j із C. Розв'язанням цієї задачі є такий впорядкований набір $c_{p1}, c_{p2} \ldots c_{pn}$ заданих міст, який мінімізує величину $\sum_{i=1}^{n-1} d\left(c_{pi}, c_{pi+1}\right) + d\left(c_{pn}, c_{p1}\right)$.

Цей вираз дає описує маршруту, який розпочинається у місті c_{p1}, проходить послідовно через усі міста та повертається у вихідний пункт із міста c_{pn}.

Індивідуальна задача комівояжера задається наступним чином (Рис. 1):

$C\ \{c_1, c_2 \ldots c_4\}$, $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$, $d(c_3, c_4) = 3$.

Послідовність $\{c_1, c_2, c_4, c_3\}$ являє собою розв'язання задачі, оскільки такий маршрут є мінімальним із можливих, його довжина становить 27.

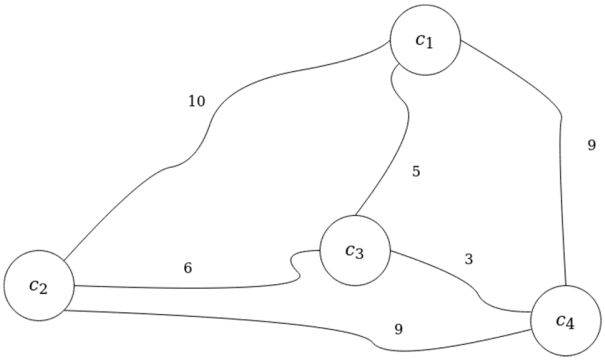


Рисунок 1 – Індивідуальна задача комівояжера

Під алгоритмом ми будемо розуміти загальну процедуру розв’язання задачі, що виконується послідовно, крок за кроком.

Говорять, що алгоритм розв’язує загальну задачу *T*, якщо він застосовний до довільної індивідуальної задачі *I* із *N* та обов’язково дає розв’язання задачі *I*. Слід зауважити, що термін «розв’язання» задачі у даному випадку вжито у строгій відповідності із сформульованим вище означенням. З цієї причини не можна сказати, що алгоритм розв’язує задачу про комівояжера, якщо він не повертає маршрут мінімальної довжини хоча б для однієї індивідуальної задачі.

Процес створення програми для розв’язання тієї чи іншої задачі можна описати за допомогою наступної схеми:

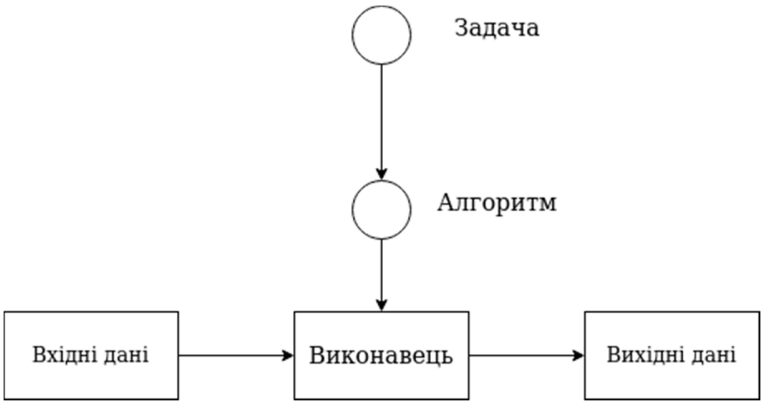


Рисунок 2 – Розв’язання задачі

Наведемо інші означення алгоритму. Томас Кормен означає алгоритм як формально описану обчислювальну процедуру, яка отримує вхідні дані (їх ще

називають входом алгоритму або його аргументом) та повертає на виході результат обчислень. Формулювання задачі описує, яким вимогам повинне задовольняти розв'язання, а алгоритм знаходить об'єкт, який цим вимогам задовольняє.

Під алгоритмом будемо розуміти скінченну чітко сформульовану сукупність вказівок, які визначають послідовність дій виконавця, спрямованих на досягнення мети або розв'язання задач певного класу за скінченний проміжок часу.

Алгоритм можна також розглядати як інструмент, який призначений для вирішення коректно поставленої обчислювальної задачі. У постановці задачі в загальних рисах визначаються відношення між входом та виходом. В алгоритмі описується конкретна обчислювальна процедура, за допомого якої можна досягнути виконання вказаних відношень.

Можна навести загальні риси алгоритму:

1. Дискретність інформації. Кожний алгоритм має справу з даними: вхідними, проміжними, вихідними. Ці дані представляються у вигляді скінченних слів деякого алфавіту.
2. Дискретність роботи алгоритму. Алгоритм виконується по кроках та при цьому на кожному кроці виконується тільки одна операція.
3. Детермінованість алгоритму. Система величин, які отримуються в кожний (не початковий) момент часу, однозначно визначається системою величини, які були отримані в попередні моменти часу.
4. Елементарність кроків алгоритму. Закон отримання наступної системи величин з попередньої повинен бути простим та локальним.
5. Виконуваність операцій. В алгоритмі не повинно бути невиконуваних операцій. Наприклад, не можна в алгоритмі призначити значення змінній «нескінченність», така операція була би невиконуваною. Кожна операція опрацьовує певну ділянку у слові, яке обробляється.
6. Скінченність алгоритму. Опис алгоритму повинен бути скінченним.

7. Спрямованість алгоритму. Якщо спосіб отримання наступної величини з деякої заданої величини не дає результату, то має бути вказано, що треба вважати результатом алгоритму.
8. Масовість алгоритму. Початкова система величин може обиратись з деякої потенційно нескінченної множини.

Природно, виникає наступне питання: «Для чого вивчати алгоритми?» По-перше, алгоритми є необхідними складовими для рішення будь-яких задач з різноманітних напрямків комп'ютерних наук. Алгоритми відіграють ключову роль у сучасному розвитку технологій. Тут можна згадати такі розповсюджені задачі, як:

- розв'язання математичних рівнянь різної складності
- знаходження оптимальних шляхів транспортування товарів та людей;
- знаходження оптимальних варіантів розподілення ресурсів між різними вузлами (виробниками, верстатами, працівниками, процесорами тощо);
- знаходження в геномі послідовностей, які співпадають;
- пошук інформації в глобальній мережі Інтернет;
- прийняття фінансових рішень в електронній комерції;
- обробка та аналіз аудіо та відео інформації.

Цей список можна продовжувати й продовжувати і, власне кажучи, майже неможливо знайти таку галузь комп'ютерних наук та інформатики, де б не використовувались ті або інші алгоритми.

По-друге, якісні та ефективні алгоритми можуть бути каталізаторами проривів у галузях, які є на перший погляд далекими від комп'ютерних наук (квантова механіка, економіка та фінанси, теорія еволюції).

І, по-третє, вивчення алгоритмів це також цікавий процес, який розвиває математичні здібності та логічне мислення.

Теоретичні дослідження та практика виконання алгоритмів підтверджують важливість наступних моментів:

Єдиного методу розробки алгоритмів для розв’язання задач не існує.

1. Існують задачі, для яких алгоритму розв’язання не існує. Це так звані алгоритмічно-нерозв’язні проблеми.
2. Існують важко розв’язувані задачі (алгоритм розв’язування задачі існує, але практично реалізований не може бути).
3. Для розв’язування однієї і тієї ж задачі може існувати декілька еквівалентних алгоритмів, тому потрібно мати методи деякого ефективного відбору алгоритмів.
4. В основу алгоритмів можуть бути покладені різні принципи, що може суттєво вплинути на час розв’язання задачі.
5. Один і той же алгоритм може бути поданий різними способами.
6. Ефективність алгоритму суттєво залежить від способу організації вхідних та вихідних даних.
7. Необхідно чітко вказувати діапазон вхідних та вихідних даних, які обробляються за допомогою алгоритму.
8. Кожен крок алгоритму повинен бути чітко та однозначно сформульований.

Розглянемо етапи повної побудови алгоритму.

1. Постановка задачі.
2. Побудова моделі.
3. Розробка алгоритму.
4. Перевірка правильності алгоритму.
5. Реалізація алгоритму.
6. Аналіз алгоритму та його складність.
7. Перевірка програми.
8. Створення документації.

Перш за все розглянемо призначення кожного етапу та з’ясуємо, як ці етапи об’єднуються в одне ціле.

1. Постановка задачі.

Для того, аби почати процес розв’язання задачі, необхідно попередньо її точно сформулювати.

Зазвичай процес формулювання задачі зводиться до постановки правильних питань. Перерахуємо деякі питання, які будуть корисними у процесі формулювання задачі:

- Чи є зрозумілою термінологія, що використовується у попередньому формулюванні?
- Що дано?
- Що потрібно знайти?
- Як визначити розв’язання?
- Яких даних не вистачає і чи усі дані є потрібними?
- Чи є дані, що не використовуються у задачі?
- Які зроблено припущення?

Є можливими й інші питання в залежності від конкретної задачі. Часто для отримання повних або часткових відповідей на деякі з питань необхідно ставити додаткові питання.

2. Побудова моделі.

Задача чітко поставлена, тепер потрібно побудувати для неї математичну модель. Це дуже важливий крок у процесі розв’язання задачі і його слід добре обдумати, оскільки вибір моделі суттєво впливатиме на подальші кроки розробки алгоритму.

Як ви можете здогадатись, неможливо запропонувати набір формальних правил, що автоматизують стадію моделювання. Більшість задач моделюються індивідуально. Але разом з тим існує ряд принципів, яких слід дотримуватись при моделюванні. Вибір моделі – у переважній більшості все ж мистецтво, ніж наука, тому вивчення вдалих моделей – найкращий спосіб набути досвіду у моделюванні.

Приступаючи до розробки моделі, слід поставити хоча б 2 наступні питання:

1. Які математичні структури найбільш підходять для задачі?

2. Чи існують розв’язані аналогічні задачі?

Друге питання, можливо, є найкориснішим у всій математиці. В контексті моделювання воно досить часто дає відповідь на перше питання. Дійсно, існують задачі, що є модифікаціями раніше уже розв’язаних задач. Перш за все слід розглянути перше питання. Ми повинні описати математично, що відомо і що потрібно знайти. На вибір відповідних структур будуть мати вплив наступні фактори:

1) обмеженість знань розробника відносно невеликою кількістю структур;
2) зручність представлення;
3) простота обчислень;
4) корисність операцій, пов’язаних із даною структурою.

Здійснивши пробний вибір математичної структури задачі, слід переформулювати в термінах відповідних математичних об’єктів. Це буде однією із можливих моделей, якщо вдасться дати відповіді на наступні питання:

Чи уся важлива інформація задачі добре описана математичними об’єктами?

Чи існує математична величина, яка пов’язана із результатом?

Чи було виявлено корисні співвідношення між об’єктами у моделі?

Чи можна працювати із моделлю? Чи зручно з нею працювати?

3. Розробка алгоритму.

Як тільки задача чітко сформульована та для неї побудована модель, слід розпочинати роботу над розробкою алгоритму її розв’язання. Вибір методу, що достатньо часто залежить від вибору моделі, може у значній мірі вплинути на ефективність роботи алгоритму. Два різних алгоритми можуть бути правильними, але дуже відрізнятись за ефективністю.

4. Правильність алгоритму.

Кажуть, що алгоритм є коректним, якщо для кожного входу результатом його роботи є коректний вивід. Тоді коректний алгоритм розв’язує дану обчислювальну

задачу. Якщо алгоритм некоректний, то для деяких входів він може взагалі не завершити свою роботу або видати відповідь, яка відрізняється від очікуваної.

Доведення правильності алгоритмів – це одна із найскладніших задач математичної науки вцілому. На даний час не існує єдиного ефективного методу доведення правильності алгоритмів. Є засоби, які дозволяють довести коректність деяких класів нескладних алгоритмів.

Ймовірно найбільш поширеною методикою доведення коректності алгоритмів є прогон їх на різних наборах тестів. Якщо вихідні дані алгоритму можна підтвердити наборами вхідних та вихідних даних, які обчислені вручну або є відомими, то виникає бажання сказати, що алгоритм працює. Але все ж існує ймовірність того, що алгоритм не запрацює.

Варто також зауважити, що правильність алгоритму ще нічого не говорить про його ефективність.

5. Реалізація алгоритму.

Як тільки описано алгоритм, наприклад, у вигляді послідовності кроків, та є підтвердження його коректності, то слід переходити до етапу його реалізації.

Цей крок може бути достатньо складним та громіздким. По-перше, складність полягає у тому, що досить часто окремо взятий крок алгоритму може бути виражений у формі, які важко перевести безпосередньо у конструкцію мови програмування. По-друге, реалізація виявитись складним процесом, тому що перед написанням програми слід побудувати цілу систему структур даних для представлення важливих аспектів моделі, що використовується у задачі.

Аби зробити це, слід дати відповіді на наступні питання:

Які основні змінні?

Яких вони типів?

Скільки потрібно, наприклад, масивів та якої вони розмірності?

Чи є потреба у використанні зв’язаних списків?

Які підпрограми можна використати?

Якою мовою програмування доцільно користуватись?

Конкретна реалізація буде суттєво впливати на вимоги до пам’яті та швидкості алгоритму.

Наступний аспект побудови програмної реалізації – це програмування зверху-вниз. Цей підхід до розробки та реалізації алгоритмів та програм буде розглядатись дещо пізніше.

Слід також зауважити, що важливо доводити як правильність алгоритмів, записаних у словесній формі, так і правильність програм, побудованих на їх основі. З цієї причини потрібно ретельно слідкувати за тим, аби процес перетворення алгоритму на програму «заслуговував на довіру».

6. Аналіз алгоритму та його складності.

Існує ряд важливих практичних причин для аналізу алгоритмів. Однією із них отримання оцінок чи границь для об’єму пам’яті чи часу роботи, які використовуються алгоритмом для обробки конкретних даних. Машинний час та машинна пам’ять – достатньо важливі ресурси, на які одночасно можуть претендувати багато користувачів. Тому якісний аналіз алгоритмів повинен виявити ті розділи програми чи алгоритму, на які затрачається більша кількість ресурсів.

Існують також важливі і теоретичні причини для аналізу алгоритмів. Доцільно було б мати критерій, який дозволить порівняти два алгоритми, які розв’язують одну і ту ж задачу.

Питання для самоконтролю

1. Дайте означення задачі.
2. У чому полягає різниця між масовою та індивідуальною задачами?
3. Наведіть приклад індивідуальної та масової задач.
4. Що таке алгоритм?
5. Назвіть основні властивості алгоритму.
6. Назвіть основні етапи розробки алгоритму.

7. Як здійснюється перевірка правильності роботи алгоритму?
8. Використання яких ресурсів оцінюється при аналізі алгоритму?
9. Які методи використовуються при розробці моделі розв'язуваної задачі?
10. Що є важливішим при розв'язуванні задачі: алгоритм чи структура даних?

Лекція 2. Способи представлення алгоритмів

Для опису алгоритмів людина часто користується природною мовою, але для запису багатьох алгоритмів вона виявилась не зовсім зручною, тому виникла необхідність у створенні штучних мов, наприклад мови математичних формул для того, аби описати послідовність дій у алгоритмі. Найбільшого поширення для запису логічної структури алгоритмів отримали словесний спосіб, графічний (структурні схеми), псевдокод та мова програмування.

Словесна форма запису – це опис алгоритмів на природних мовах. Цей спосіб не так широко розповсюджений через його непрактичність, громіздкість.

Розглянемо опис алгоритму пошуку максимального числа серед трьох заданих (числа не рівні між собою) $\{a, b, c\}$.

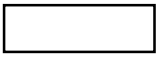
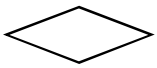
Крок 1. Якщо число a більше від числа b, то перейти до Крок 2. В іншому випадку перейти до Крок 3.

Крок 2. Якщо число a більше від числа c, то максимальне число a. В іншому випадку максимальне число c.

Крок 3. Якщо число b більше від числа c, то максимальне число b. В іншому випадку максимальне число c.

Графічна схема (блок-схема) алгоритму – це графічне зображення алгоритму у вигляді спеціальних блоків з необхідними словесними поясненнями. Кожний етап алгоритму представляється у вигляді геометричної фігури (блоку), що має певну форму в залежності від характеру операції. Блоки на схемі з'єднуються стрілками (лініями зв'язку), які визначають послідовність виконання операцій та

утворюють логічну структуру алгоритму. Основні блоки графічної схеми продемонстровані нижче:

Позначення	Найменування	Опис
	блок пуск-зупинка	визначає початок та кінець алгоритму (для блоку пуск (початок) - визначений тільки один вихід, для блоку зупинка (кінець) – тільки вхід);
	блок введення-виведення	визначає введення інформації в програму або виведення на пристрій
	блок процес	визначає зміну значення, форми уявлення або розташування даних
	блок перевірки умови	визначає подальші кроки виконання алгоритму в залежності від виконання умови

Важливою особливістю зображень базових структур алгоритмів є те, що вони мають один вхід і один вихід, що дозволяє при відносній незалежності конструювати окремі блоки алгоритмів, а потім окремо розроблені структури з’єднувати між собою (вихід однієї базової структури сполучається із входом іншої). Весь алгоритм представляє лінійну послідовність базових структур.

На будь-якій стадії існування алгоритми та програми представляють за допомогою конкретних графічних засобів, склад і правила вживання яких утворюють конкретні способи або форми запису.

Алгоритм пошуку максимального числа серед трьох заданих, представлений у вигляді блок-схеми, буде мати наступний вигляд:

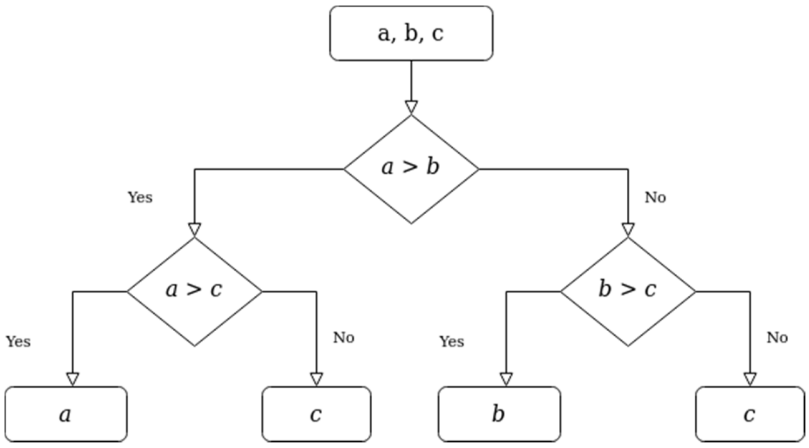


Рисунок 3 – Блок-схема алгоритму пошуку максимального числа серед трьох заданих

Поряд зі схемами для зображення алгоритмів широко використовується псевдокод. Псевдокод називається система правил запису алгоритму з використанням набору певних конструкцій для опису керуючих дій.

Псевдокод дозволяє формально зображати логіку алгоритму, використовуючи стандартизовані конструкції природної мови для зображення управління та зберігаючи можливості мови для опису дій по обробці інформації. Даний спосіб тісно пов’язаний зі структурним підходом до програмування. Псевдокод займає проміжне положення між природною мовою та мовою програмування. Його застосовують переважно для того, щоб детальніше пояснити роботу програми, що полегшує перевірку правильності програми. Крім того, псевдокод дає розробнику велику свободу в зображенні алгоритму. Потрібно тільки використовувати стандартні управляючі конструкції та правила запису.

Алгоритм пошуку максимального числа серед трьох заданих, представлений у вигляді псевдокоду, буде мати наступний вигляд.

```
begin
a, b, c, max
if a > b then
        if a > c then max := a else max := c
else
```

if b > c then max := b else max := c

Останнім способом запису алгоритмів є мова програмування. Розглянуті вище способи зручні для програміста, але не прийнятні для комп'ютера, оскільки вони не можуть бути однозначно зрозумілі.

Мова програмування – це знакова система, призначена для опису процесів вирішення завдань та їх реалізації із використанням обчислювальної техніки. Реалізація означає, що описи можуть бути задані комп'ютеру і однозначно ним зрозумілі. До мов програмування відносяться мови команд або машинні мови та мови високого рівня.

Перша група являє власну мову обчислювальної машини і виконання програми можливе лише в тому випадку, якщо вона записана на цій мові. Однак програмувати на машинному мові досить важко, що обумовлено надмірною деталізацією програми, необхідністю знати конкретну систему команд і детально представляти роботу. Представлення складної програми на машинній мові незручно для сприйняття людиною. Ці недоліки послужили стимулом для створення мов програмування високого рівня, які не збігаються з машинними. Ідея таких мов полягає в представленні програм у вигляді не тільки прийнятному для комп'ютера, а й зручному для користувача. Комфортність означає, що спосіб запису повинен відображати основні ідеї програмування і представляти програму в однозначною, природною, у формі, що легко сприймається. Це означає, що програма може бути введена задана комп'ютеру і однозначно зрозуміла, тобто однозначно переведена на машинну мову. Мови програмування високого рівня дають розробнику велику свободу в конструюванні програм, але не звільняють його від необхідності враховувати той факт, що саме комп'ютер буде виконувати його програму і що саме комп'ютер накладає на програму обмеження, обумовлені скінченністю її швидкості та пам'яті.

Алгоритм пошуку максимального числа серед трьох заданих, представлений у вигляді програми на мові програмування С++, буде мати наступний вигляд.

```
int a, b, c, maxNum;
if (a > b) {
        if (a > c) {maxNum = a;}
        else {maxNum = c;}
          }
else {
        if (b > c) {maxNum = b;}
        else {maxNum = c;}
}
cout<<maxNum;
```

Питання для самоконтролю

1. Назвіть основні способи представлення алгоритмів.
2. Назвіть переваги та недоліки словесного способу представлення.
3. Назвіть переваги та недоліки представлення алгоритмів у вигляді блок-схеми.
4. Назвіть переваги та недоліки представлення алгоритму у вигляді псевдокоду.
5. Назвіть переваги та недоліки представлення алгоритму у вигляді програмного коду.

Лекція 3. Поняття про базові алгоритмічні конструкції

Базові алгоритмічні конструкції або базові алгоритмічні структури – це основні структурні елементи, за допомогою яких створюють алгоритм для розв’язування деякої загальної задачі. Існують три основні (базові) алгоритмічні структури: слідування (або лінійна структура), розгалуження та повторення (або цикл), компонуючи які можна представити логічну структуру будь-якого алгоритму.

Основною особливістю базових алгоритмічних структур є їх повнота. Це означає, що цих структур достатньо для створення алгоритмів довільної складності, та наявність в них одного входу та одного виходу.

Структура слідування або лінійна алгоритмічна структура – це алгоритмічна структура, яка забезпечує отримання результату шляхом одноразового виконання дії, незалежно від вхідних даних та проміжних результатів. Дії у такій структурі виконуються послідовно, одна за одною, або лінійно.

На алгоритмічній мові структура слідування виглядає наступним чином:

Алгоритм:

Вхід

Дія 1

Дія 2

…

Дія N

Вихід

Алгоритми, які використовують лише структуру слідування, називаються лінійними. Варто зауважити, що усі алгоритми використовують структуру слідування, але не всі вони є лінійними.

Структура розгалуження (структура вибору або умова) – вибір дії або групи дій (блоку) у разі виконання або невиконання заданої умови. Ця алгоритмічна структура залежно від результату перевірки умови (так чи ні, істина або хибно) забезпечу виконання одного з альтернативних шляхів роботи алгоритму. Кожен із них веде до загального виходу, тому робота алгоритму триватиме незалежно від того, який із варіантів було обрано.

Розгалуження поділяються на два типи: повні і неповні.

Неповне розгалуження – це розгалуження, в якому хід алгоритму визначено лише в разі виконання умови. Якщо умова не виконується, то робота алгоритму визначається алгоритмічною конструкцією, яка розміщена після розгалуження.

На алгоритмічній мові структура неповного розгалуження виглядає наступним чином:

якщо … то …

Алгоритм:

Вхід

...

якщо умова то дія

…

Вихід

Повне розгалуження – це розгалуження, в якому певні дії визначено у двох випадках: виконання та невиконання умови.

На алгоритмічній мові структура повного розгалуження виглядає наступним чином:

якщо … то … інакше

Алгоритм:

Вхід

...

якщо умова то дія1 інакше дія2

…

Вихід

Поширеним випадком є розміщення однієї структури вибору в іншій. Такі структури називають вкладеними розгалуженнями.

Структура розгалуження існує в чотирьох основних варіантах:

- якщо то;
- якщо то-інакше;
- вибір;
- вибір-інакше.

Вибір – це вказівка багатовибірного розгалуження, яка дає змогу вибрати одну з множини альтернатив. Така структура використовується у тому випадку, коли кількість вкладень у розгалуженнях є досить значною, що у свою чергу погіршує читабельність коду.

На алгоритмічній мові структура повного розгалуження виглядає наступним чином:

вибір

при умова 1: дії 1

при умова 2: дії 2

. . . . . .

при умова N: дії N

У тому випадку, якщо вхідні дані не задовольняють жодну із умов, яка запропонована у структурі вибору, керування ходом алгоритму передається до команди, що розміщена після вибору.

Вибір-інакше – це структура вибору, для якої запропоновано команду для випадку, коли вхідні дані не задовольняють жодну із запропонованих альтернатив.

На алгоритмічній мові структура вибору виглядає наступним чином:

вибір

при умова 1: дії 1

при умова 2: дії 2

. . . . . .

при умова N-1: дії N-1

інакше дії N

Структура повторення (структура циклу або цикл) - це алгоритмічна структура, в якій передбачено повторення деякої вказівки або блоку вказівок. За допомогою цієї структури описують однотипні дії, що повторюються багатократно. Такі алгоритми забезпечують виконання значної послідовності дій, записаних порівняно короткою послідовністю команд.

Повторення забезпечує багаторазове виконання деякої сукупності команд, яку називають тілом циклу. Існують три види циклів.

Цикл із лічильником (цикл із параметром) – це алгоритмічна конструкція, в якій кількість повторень задано заздалегідь, керується лічильником (ітератором) і

логічними виразами, розташованими на його початку. Це єдиний вид циклу, для якого кількість повторень є наперед заданою.

На алгоритмічній мові структура циклу із параметром виглядає наступним чином:

початок циклу для і від і1 до і2
тіло циклу
кінець циклу

Цикл із передумовою – це алгоритмічна конструкція, що забезпечує виконання команди або групи команд доти, поки виконується (є істинною) логічна умова, сформульована на його початку. Для циклу із передумовою кількість повторень не є наперед заданою. Якщо вхідні дані забезпечують хибність логічної умови, яка задана на початку циклу, то його тіло може не виконатись жодного разу. У такому випадку керування ходом алгоритму переходить до команди (вказівки), яка розміщена після циклу.

На алгоритмічній мові структура циклу з передумовою виглядає наступним чином:

початок циклу поки умова
тіло циклу
кінець циклу

Цикл із післяумовою – це алгоритмічна конструкція, що забезпечує виконання команди або групи команд доти, поки виконується (є істинною) логічна умова, сформульована у його кінці. Для циклу із післяумовою кількість повторень не є наперед заданою. Якщо вхідні дані забезпечують хибність логічної умови, яка задана в кінці циклу, то його тіло виконається один раз. Після цього керування ходом алгоритму переходить до команди (вказівки), яка розміщена після циклу. Тобто якщо тіло циклу з передумовою може не виконатись жодного разу, то тіло циклу з післяумовою обов’язково виконається хоча б один раз.

На алгоритмічній мові структура циклу з післяумовою виглядає наступним чином:

початок циклу

тіло циклу

поки умова кінець циклу

Теоретичним обгрунтуванням того, що для побудови алгоритму довільної складності достатнім є використання лише трьох алгоритмічних структур (слідування, розгалуження та циклу), є теорема Бьома-Якопіні.

Теорема (Бьома-Якопіні): виконуваний алгоритм може бути втілено з використанням лише трьох конкретних керівних структур: послідовного виконання, розгалужень, повторень (циклів).

Цю теорему було сформульовано та доведено італійськими математиками Коррадо Бьомом і Джузеппе Якопіні в їхній статті у 1966 р. У статті було описано методи перетворення неструктурованих алгоритмів на структуровані на прикладі створеної Бьомом мови програмування P''. Структурна теорема Бьома-Якопіні була початком структурного програмування – парадигми програмування, яка виключає команди безумовного переходу (goto) й використовує виключно підпрограми, послідовне виконання, розгалуження (вибір) і цикли (ітерації). Ця теорема є науковим положенням, використаним Е. Дейкстрою для обґрунтування його ідеї про використання в програмах лише трьох керуючих конструкцій: послідовного виконання, розгалужень і циклів. Теорема Бьома-Якопіні не розв'язує питання про те, чи слід застосовувати структурне програмування для розробки програмного забезпечення. Деякі науковці використовували пуристській підхід до результату Бьома-Якопіні й стверджували, що навіть такі інструкції, як break і return в середини циклів, є поганим підходом до розробки алгоритмів і всі цикли повинні мати єдину точка виходу. Пряме застосування теореми Бьома-Якопіні може привести до додавання додаткових локальних змінних до структурованої програми, а також до деякого дублювання коду. Останнє питання в

цьому контексті називають «проблемою циклу з половиною». 1973 року С. Рао Косараджу довів, що можливо уникати додавання додаткових змінних в структурне програмування, якщо допускаються багаторівневі виходи довільної глибини з циклів.

Питання для самоконтролю

1. Дайте означення базової алгоритмічної конструкції.
2. Що таке структура слідування?
3. Які алгоритми називають лінійними?
4. Чи всі алгоритми, які містять структуру слідування лінійні? Обґрунтуйте.
5. Дайте означення структури розгалуження .Які бувають види розгалужень?
6. Що таке вкладене розгалуження?
7. Дайте означення структури вибору.
8. У чому полягає різниця між структурами “вибір” та “вибір-інакше”?
9. Дайте означення циклу. Назвіть види циклів.
10. Для яких видів циклів наперед відомою є кількість повторень?
11. Для яких видів циклів тіло циклу може не виконатись жодного разу?
12. Для яких видів циклів тіло циклу обов’язково виконається хоча б один раз?
13. Сформулюйте теорему Бьома-Якопіні.

Лекція 4. Рекурсія. Рекурсивні алгоритми

Метою будь-якого алгоритму є переробка вхідних даних на вихідні. Якщо окремому вихідному елементу відповідає окремий вхідний елемент, то основу алгоритму становить кострукція повторень. Якщо організація вихідного потоку відрізняється від організації вхідного, логічніше використовувати два незалежні алгоритми: один для отримання вихідних даних, інший – для їх організації.

Усі види обробки даних можна поділити на такі класи: послідовну, що використовує структуру повторення; структурну, що виконується паралельно за

допомогою незалежних програм; довільну обробку із застосуванням паралельних обчислень.

Повторення є основною керуючою конструкцією обробки даних, за винятком, коли вхідні дані складаються з одного елемента, який перетворюється на один вихідний елемент. Існують дві основні форми повторень: ітерація та рекурсія. Ітерація переважно використовується для тих видів обробки даних, які найкраще визначаються виразом типу «виконати для всіх *x*», а рекурсія – для отримання результуючих даних, які найлегше описати рекурсивно, тобто описати виразом «виконати те саме, що й востаннє». Поточна дія визначається за допомогою попередньої відповіді або попередніх стадій обчислень. Ітерація та рекурсія взаємозамінні.

Рекурсією називається такий спосіб організації обробки даних, при якому алгоритм чи програма (функція) викликає сама себе безпосередньо, або з інших програм (функцій). Функція називається рекурсивною, якщо під час її роботи відбувається або виникає її повторний виклик безпосередньо, або опосередковано через ланцюжок викликів інших функцій. Як правило, в основі рекурсивного алгоритму лежить рекурсивне визначення якогось поняття, тобто визначення, яке задає деякий обє'кт у термінах більш простого випадку цього ж об'єкту. Розрізняють пряму і непряму рекурсії.

Функція називається прямо рекурсивною, якщо містить у своєму тілі виклик самої себе.

Якщо ж функція викликає іншу функцію, що у свою чергу викликає першу, то така функція називається непрямо рекурсивною.

Рекурсивні функції найчастіше використовуються для компактної реалізації рекурсивних алгоритмів. Основою для розробки рекурсивних алгоритмів служать рекурентні співвідношення (формули), що встановлюють залежності між результатами яких-небудь дій (операцій) на *n*-ному кроці від результатів аналогічних дій (операцій), отриманих на попередньому (*n*-1)-ому кроці.

Класичним прикладом рекурсії може слугувати визнаення факторіалу в такому вигляді:

$$n! = 1 \; n = 0, \; n! = n(n - 1)!, \text{ де } n \text{ – натуральне число.}$$

Цей приклад носить тільки ілюстративний характер через свою зручність для пояснення поняття рекурсії. Однак алгоритми, що здійснюють розв'язання задачі із використанням рекурсії, практично не дають виграшу в програмній реалізації порівняно з ітераційним способом розв'язання цих задач.

Приклад використання рекурсивної процедури для організації обчислення n!:

```
int factor(int n){
if (n==0) D=D*1; else D=n*factor(n-1);
returnD;
}
```

Важливим моментом є той факт, що в наведеному прикладі рекурсивний виклик стоїть всередині оператора умови. Це є необхідною умовою того, щоби рекурсивний процес рано чи пізно завершився (рекурсія не продовжувалася до нескінченості). Важливим також є той факт, що функція викликає себе з іншим значенням параметра: не тим, із яким сама вона була викликана.

Схематично такий процес можна зобразити наступним чином: якщо випадок найпростіший, то розвязати його напряму, інакше виконувати рекурсивний виклик до настання найпростішого випадку.

Якщо, наприклад, викликати функцію factor зі значенням параметра n=3: factor(3), то схематично рекурсивний процес можна зобразити наступним чином:

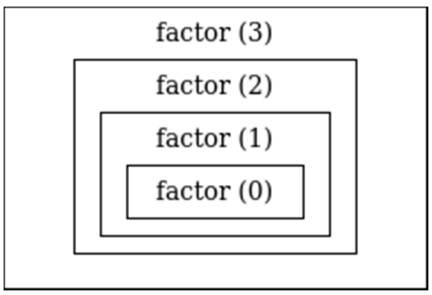


Рисунок 4 - Виконання рекурсивної функції factor (3)

Функція factor (3) викликається зі значенням параметра *n*=3. Вона, в свою чергу, містить виклик функції factor зі значенням параметра *n*=2. Фактично в оперативній памяті комп'ютера створюється ще одна функція factor і до завершення її роботи попередня функція роботу не закінчить. Процес рекурсивних викликів завершиться при значенні параметра *n*=0: в цей момент виконаються всі 4 функції. Цей процес можна зобразити у вигляді наступної діаграми, яка показує обидві фази функції обчислення факторіалу із використанням рекурсії.

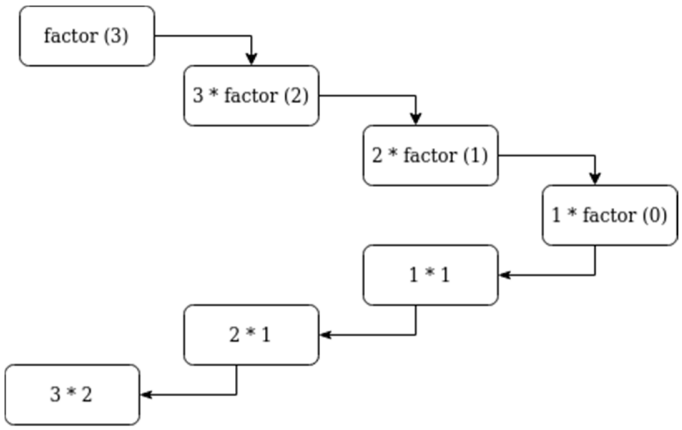


Рисунок 5 - Діаграма обчислення факторіалу із виконання рекурсивної функції factor (3)

У рекурсивних функціях можна виділити два процеси: рекурсивне занурення функції у себе, що відбувається доти, доки параметр не досягне деякого граничного значення, та рекурсивне повернення з підпрограми, що відбувається доти, доки параметр не сягне початкового значення. Після виклику функції з

параметром n виконується ще $n-1$ викликів, і загальна кількість незавершених викликів сягає n. Величина, що характеризує максимальну кількість незавершених рекурсивних викликів, називається глибиною рекурсії. Рекурсивний виклик може бути непрямим. В цьому випадку функція А звертається до себе опосередковано, шляхом виклику іншої В, в якій міститься звернення до першої, створючи рекурсивний ланцюжок. Умова завершення в такій рекурсії може бути в одній або всіх функціях. Важливим є те, що перша описана функція повинна викликати ще неописану функцію. Для цього необхідно використати опис функції В до її використання.

Питання для самоконтролю

1. Який процес обробки даних називають ітеративним?
2. Який процес обробки даних називають рекурсивним?
3. Дайте означення рекурсії.
4. Яка функція називається прямо рекурсивною?
5. Яка функція називається непрямо рекурсивною?
6. Якщо рекурсивна функція викликається із параметром p, скільки ще викликів цієї функції буде здійснено у програмі?
7. Який із способів обробки даних є більш виграшним з точки зору програмної реалізації: рекурсивний чи ітераційний?
8. Що таке глибина рекурсії? Якщо рекурсивна функція викликається із параметром p, яка у такому випадку глибина рекурсії?
9. Для чого призначена умова виходу з рекурсії?

Лекція 5. Оцінка ефективності алгоритмів

Аналізуючи алгоритм, можна отримати уявлення про те, який час займе розв’язання даної задачі за допомогою даного алгоритму. Для кожного розглянутого алгоритму ми оцінимо, наскільки швидко вирішується завдання на масиві вхідних даних довжини N. Наприклад, ми можемо оцінити, скільки

порівнянь вимагатиме алгоритм сортування при впорядкуванні списку з *N* величин за зростанням, або підрахувати, скільки арифметичних операцій потрібно для множення двох матриць розміром *N* x *N*.

Одну і ту ж задачу можна розв'язати за допомогою різних алгоритмів. Аналіз алгоритмів надає інструмент для вибору алгоритму. Розглянемо, наприклад, два алгоритми, в яких вибирається найбільше з чотирьох чисел:

```
largest = a
if b > largest then
   largest = b
end if
return a
if c > largest then
   largest = c
end if
if d > largest then
   largest = d
end if
return largest
```

```
if a > b then
    if a > c then
        if a > d then
            return a
        else
            return d
        end if
    else
            if c > d then
                return c
            else
                return d
            end if
    end if
else
    if b > c then
        if b > d then
            return b
        else
            return d
        end if
```

```
        else
            if c > d then
                return c
            else
                return d
            end if
        end if
    end if
```

Розглянувши ці алгоритми, можна помітити, що у кожному із них виконується 3 порівняння. Перший алгоритм легше прочитати та зрозуміти, але з точки зору виконання на комп'ютері у них однакова складність. З точки зору часу виконання ці два алгоритми однакові, але перший вимагатиме більше пам'яті у зв'язку із використанням змінної largest. Ця додаткова змінна не відіграє ролі, якщо працювати із простими типами даних, але при роботі із складними структурами, може суттєво впливати на кількість пам'яті. Різні характеристики алгоритмів призначені для порівняння ефективності різних алгоритмів, які розв'язують одну і ту ж задачу. Тому немає сенсу порівнювати алгоритми, наприклад, множення матриць та пошуку найбільшого елемента в наборі чисел.

Для обґрунтованого застосування алгоритмів необхідно, аби їх властивості були відомими. До однієї із категорій властивостей алгоритмів відносять: кількість операцій та операндів, кількість унікальних операцій та операндів, обсяг програми (міри Холстеда), цикломатична міра Мак-Кейба та інші. Ця категорія властивостей вирізняється тим, що вони можуть бути визначені за представленням алгоритмів та не потребують їх виконання.

Інша категорія властивостей алгоритмів – це властивості, які проявляються лише при виконанні алгоритмів або при їх багаторазовому виконанні. Такі властивості називають експлуатаційними характеристиками. До них відносять часову та ємнісну ефективність алгоритмів.

Алгоритм розв’язання обчислювальної задачі представляє собою деяку детерміновану процедуру, що виконується над набором вхідних даних. На різних наборах даних алгоритм може мати різний час роботи. З цієї причини під складністю алгоритму розуміють максимальний час роботи для всіх наборів даних із деякої фіксованої множини. Тобто часовою складністю алгоритму називають час *T*, затрачений алгоритмом на виконання, як функція розміру задачі *f(n)*. Поведінка цієї складності в певних межах при збільшенні розміру задачі називається асимптотичною часовою складністю. Аналогічно можна визначити ємнісну складність і асимптотичну ємнісну складність. Саме асимптотична складність алгоритму визначає розмір задач, які можна розв’язати за допомогою даного алгоритму.

Час, витрачений на обчислення при виконанні алгоритму, являє собою суму часів окремих виконаних операторів. Програму, написану на мові високого рівня, можна перетворити прямим (хоча і не простим) шляхом в програму на машинному коді заданого комп’ютера. Це дає метод оцінки часу виконання вказаної програми, але такий підхід орієнтований на конкретний комп’ютер і не дає загальної залежності часу обчислення від розмірів задачі. В області аналізу та побудови алгоритмів прийнято виражати час виконання, як і будь-яку іншу міру ефективності, з точністю до мультиплікативної константи. Це, зазвичай, робиться шляхом підрахунку лише певних ключових операцій, виконаних алгоритмом (що легко здійснити, аналізуючи версію цього алгоритму, записану на мові високого рівня).

Такий підхід абсолютно правомірний при визначені нижніх оцінок часу виконання, оскільки невраховані операції можуть лише збільшити їх. Однак при роботі з верхніми оцінками кількість вибраних ключових операцій в сумі відрізняється не більше, ніж в стале число разів від кількості усіх операцій, виконаних алгоритмом.

Означення: Час, який витрачається алгоритмом на його виконання, називають часовою складністю цього алгоритму. Граничну поведінку цієї складності при збільшені розміру задачі називають асимптотичною часовою складністю.

Аналогічно, можна виділити об'ємну (ємнісну) складність та асимптотичну об'ємну складність.

Ємнісна та часова складності, як функції від розмірів задачі, є двома фундаментальними оцінками ефективності при аналізі алгоритмів.

Д. Кнутом було запроваджено та популяризовано наступний апарат позначень, які відрізняють верхні та нижні оцінки.

Верхні оцінки: $\mathrm{O}\left(f\left(N\right)\right)-\left\{g\left(N\right)\exists C>0 i N_0>0:\left|g\left(N\right)\right|\le Cf\left(N\right),\forall N\ge N_0\right\}$.

Нижні оцінки: $\Omega\left(f\left(N\right)\right)-\left\{g\left(N\right)\exists C>0 i N_0>0:\left|g\left(N\right)\right|\ge Cf\left(N\right),\forall N\ge N_0\right\}$.

Ефективні оцінки:

$\Theta\left(f\left(N\right)\right)-\left\{g\left(N\right)\exists C_1,C_2,N_0>0:C_1\le\left|g\left(N\right)\right|\le C_2 f\left(N\right),\forall N\ge N_0\right\}$.

Таким чином, O(*f*(*N*)) використовується для позначення верхніх оцінок швидкості росту функцій або – для множини усіх функцій, які ростуть не швидше, ніж *f*(*N*).

$\Omega\left(N\right)$ використовується для позначення нижніх оцінок швидкості росту функцій або для множини усіх функцій, які ростуть не повільніше, ніж *f*(*N*).

Нарешті, $\Theta\left(N\right)$ використовується для позначення функцій того ж порядку, що і *f*(*N*). Цей спосіб потрібний для опису оптимальних алгоритмів.

Одним із основоположних понять теорії складності алгоритмів є поняття моделі обчислень

Означення: Модель обчислень визначає набір допустимих елементарних операцій і вартості цих операцій.

Означення: Елементарні операції – це такі операції, для кожної із яких призначається фіксована вартість, хоча ця вартість неоднакова для різних елементарних операцій.

До основних моделей обчислень відносять:

- РАМ-машина з довільним доступом до пам’яті (або рівнодоступна адресна машина);
- РАСП-машина з довільним доступом до пам’яті і програмою, яка зберігається (або рівно доступна адресна машина із програмою, яка зберігається);
- машина Тюрінга.

Детально опишемо першу із зазначених моделей РАМ (RAM – Random Access Machine) в кожній комірці пам’яті може зберігати єдине дійсне число та характеризується наступними елементарними операціями, що мають одиничну вартість: арифметичні операції, операції порівняння двох дійсних чисел, непряма адресація пам’яті лише з цілочисельними адресами. При необхідності використовуються логічні, алгебраїчні та тригонометричні операції.

Машина із довільним доступом до пам’яті моделює обчислювальну машину з одним суматором, в якій команди не можуть змінювати себе. Ця машина складається із вхідної стрічки, з якої вона може лише зчитувати дані, вихідної стрічки, на яку можна лише записувати, та пам’яті. Вхідна стрічка є послідовністю комірок, кожна з яких може містити ціле число. Кожного разу, коли символ зчитується із стрічки, читаюча головка зміщується на одну комірку вправо. Вихід являє собою стрічку, що розбита на комірки, які є порожніми від початку роботи. При виконанні команди здійснюється запис цілого числа у ту комірку, над якою на даний момент часу розміщена головка. Після запису головка зміщується вправо. Як тільки вихідний символ був записаний, його уже не можна змінити. Пам’ять РАМ-машини складається із регістрів r_1, r_2, ... r_i , кожен із яких може містити довільне ціле число.

Програма для РАМ-машини (або РАМ-програма) не записується у пам'ять. З цієї причини вважають, що програма не змінює сама себе. По суті, програма є послідовністю помічених команд. Точний тип команд не дуже важливий до того часу, поки вони не нагадують ті команди, які трапляються у реальних обчислювальних машинах. Усі обчислення виконуються у першому регістрі r_0, який називають сумматором. Кожна команда складається із двох частин – коду команди та адреси.

Тепер опишемо основні поняття, які стосуються складності РАМ-програм. Як уже було сказано вище, двома важливими мірами складності алгоритмів є часова і ємнісна складності, що розглядаються як функції від розміру входу. Якщо при даному розмірі в якості міри складності береться найбільша зі складностей (по всіх входах такого розміру), то її називають складністю в гіршому випадку. Якщо в якості міри складності береться середня складність по всіх входах даного розміру, то вона називається середньою (або усередненою) складністю. У переважній більшості випадків середню складність знайти важче, ніж складність в гіршому випадку, оскільки потрібно прийняти деяке припущення про розподіл входів. А реалістичні припущення часто буває важко сформулювати математично. При обчислення складності РАМ-програм основна увага приділяють складності у найгіршому випадку, оскільки її найлегше обчислити і вона має універсальну застосовність. Проте слід зазначити, що алгоритм із найменшою складністю в гіршому випадку не обов'язково має таку ж складність в середньому.

Часова складність в гіршому випадку РАМ програми-це функція *f(n)*, яка рівна найбільшій (за всіма входами розміру *n*) із сум часів, витрачених на кожну спрацьовану команду. Часова складність в середньому – це середнє, взяте за всіма входами розміру *n* тих же сум. Для того, щоб визначити часову складність, слід вказати час виконання для кожної із команд.

Існує два вагових критерії для обчислення складності РАМ-програми.

При рівномірному ваговому критерії кожна РАМ-команда витрачає одну одиницю часу і кожен регістр використовує одну одиницю пам’яті.

Наступний критерій, інколи більш реалістичний, приймає до уваги обмеженість розміру реальної комірки пам’яті та носить назву логарифмічного вагового критерію. Нехай *l(i)* – логарифмічна функція на цілих числах, яка задається наступними рівностями:

$$l(i)=\begin{cases}\log|i|, i\neq 0\\ 1, i=0\end{cases}$$

Логарифмічний критерій базується на припущенні, що ціна виконання команди (її вага) прямо пропорційна довжині її операнд.

Методика проведення аналізу часової та ємнісної складностей алгоритму буде продемонстрована нижче.

Питання для самоконтролю

1. За якими критеріями здійснюють оцінку ефективності алгоритму?
2. Які види складності алгоритмів виділяють?
3. Що впливає на складність алгоритму?
4. Що таке найкращий час виконання алгоритму?
5. Що таке найгірший час виконання алгоритму?
6. Що таке середній час виконання алгоритму?
7. Який час використовують для оцінки часу виконання алгоритму?
8. Що таке модель обчислень?
9. Перерахуйте приклади моделей обчислень.
10. Що таке ваговий критерій РАМ-програми?

Лекція 6. Поняття структури даних. Класифікація структур даних. Прості структури даних

Необхідною умовою зберігання інформації в пам’яті комп’ютера є можливість перетворення цієї інформації у прийнятну для комп’ютера форму. У

тому випадку, якщо ця умова виконується, слід визначити структуру, придатну саме для наявної інформації, ту, яка надасть необхідний набір можливостей роботи з нею. Тут і надалі під структурою даних розумітимемо спосіб представлення інформації, за допомогою якого сукупність окремо взятих елементів утворює щось єдине, обумовлене їх взаємозв'язком між собою. Скомпоновані за деякими правилами і логічно пов'язані між собою дані можуть ефективно оброблятися, так як загальна для них структура надає набір можливостей управління ними -одне з того за рахунок чого досягаються високі результати у вирішенні тих чи інших завдань. У програмуванні структури даних визначають способи організації даних у пам'яті комп'ютера.

Тобто, структура даних – це множина елементів даних і зв'язків між ними.

Незалежно від вмісту і складності будь-які дані в пам'яті комп'ютера представляються у вигляді послідовності двійкових розрядів (бітів), а їх значеннями є відповідні двійкові числа. Бітові послідовності слабо структуровані і незручні для практичного використання. На практиці зазвичай використовують більш складно організовані структури даних.

З поняттям структури даних тісно пов'язане поняття тип даних.

Розрізняють фізичну і логічну структури даних. Фізична структура на відміну від логічної відображає спосіб представлення даних в пам'яті комп'ютера і називається ще внутрішньою.

Розрізняються прості структури (типи) даних та інтегровані (складні). Прості структури не можуть бути розділені на складові частини, більші ніж біти. З точки зору фізичної структури для простого типу чітко визначений його розмір і спосіб розміщення в пам'яті комп'ютера. З точки зору логічної структури прості структури є неподільними одиницями. Інтегровані структури даних включають в себе інші структури даних – прості чи інтегровані.

Між окремими елементами структур можуть бути присутні чи неприсутні явно задані зв’язки. В залежності від цього слід розрізняти: незв’язні структури (вектори, масиви, рядки, стеки, черги) і зв’язні структури (зв’язні списки).

За ознакою мінливості (здатність змінювати кількість елементів, що входять до складу структури даних) розрізняють структури статистичні, напівстатистичні, динамічні. Під мінливістю розуміють зміна числа елементів структури чи зв’язків між цими елементами. Класифікація структур даних по признаку мінливості приведена на рисунку 6.

За ознакою впорядкованості елементів структури можна ділити на лінійні і нелінійні. Прикладом нелінійних структур – багатозв’язні списки, дерева, графи.

Лінійні структури, в свою чергу, діляться на структури з послідовним розподіленням (вектори, рядки, масиви, стеки, черги) і структури з довільним зв’язним розподіленням (однозв’язні, двозв’язні списки) за характером розподілення елементів в пам’яті.

На рисунку нижче продемонстровано класифікацію структур даних.

Рисунок 6 - Класифікація структур даних

При оголошенні типу даних чітко визначається:

- розмір пам’яті, відведеної під дану структуру і спосіб її розміщення в пам’яті;

- значення, допустимі для даного типу даних;
- операції, які можна над цими даними виконувати.

Прості структури даних служать основою для побудови більш складних структур. Їх також називають примітивними чи базовими структурами (типами даних). До них відносяться: числові, бітові, логічні, символьні, перелічувальні, інтервальні, вказівники. У різних мовах програмування набір і розміри простих типів відрізняються одне від одного. У таблицях нижче наведено класифікацію простих типів даних для мов програмування C++, Java та Phyton.

Базові типи C++

Тип	**Розмір (байт)**	**Опис**	**Множина значень**
char	1	Мале ціле	-128 до 127
unsigned char	1	Беззнакове мале ціле	0 до 255
short int	2	Коротке ціле	-32768 до 32767
Unsigned short int	2	Беззнакове Коротке ціле	0 до 65535
int	4	ціле	-2147483648 до -2147483647
Unsigned int	4	Беззнакове ціле	0 до 4294967295
bool	1	Логічне значення	True або false
float	8	Число із плаваючою комою	+-3.4 x 10 ^+-38 (7 цифр)
double	8	Подвійна плаваюча кома	+-1.7 x 10 ^+-308 (15 цифр)

Базові типи Java

Тип	Розмір (біт)	Множина значень
byte	8	-128 до 127
short	16	-32768 до 32767
int	32	-2147483648 до -2147483647
long	64	-9223372036854775808 до -9223372036854775807
char	16	0 до 65536
boolean		True або false
float	32	+3.4e+38
double	64	+1.7e+308

Базові типи Phyton

Тип	Назва	Елементи типу
str	рядок	Будь-які символи. Наприклад, “book 4”, “51 + 12”.
float	Дійсні числа	Множина всіх чисел. Наприклад, 0.99, -36.704, 0.
int	Цілі числа	Всі натуральні числа, їм обернені та число 0. Наприклад, 7, -12, 0

Статичні структури представляють собою структуровану множину примітивних структур. Наприклад, вектор може бути представлений впорядкованою множиною чисел. Мінливість не властива статичним структурам, тобто розмір пам’яті комп’ютера, яка відводиться для цих даних, сталий і виділяється на етапі компіляції чи виконання програми.

З логічної точки зору вектор (одновимірний масив) представляє собою структуру даних з фіксованим числом елементів одного і того ж типу. Кожен елемент вектора має свій унікальний номер (індекс). Звернення до елемента вектора виконується за іменем вектора і номером елемента.

З фізичної точки зору елементи вектора розміщуються в пам'яті у підряд розміщених комірках пам'яті. Під елемент вектора виділяється кількість байт пам'яті, яка визначається базовим типом елемента цього вектора. Тоді розмір пам'яті визначається співвідношенням: S = k * sizeof(num), де *k* – кількість елементів (довжина) вектора, а sizeof(num) – розмір пам'яті, яка необхідна для збереження одного елементу вектора.

Двовимірний масив (матриця) – це вектор, кожен елемент якого вектор. Тому те, що справедливо для вектора, справедливо і для матриці (аналогічно для *n*-вимірних масивів).

Множиною є структура, яка представляє собою набір даних, які не повторюються, одного і того ж типу. Множина може приймати всі значення базового типу. Множина в пам'яті зберігається як масив бітів, в якому кожен біт вказує, чи належить елемент оголошеній множині чи ні.

Розмір пам'яті (в байтах), яка виділяється під множину обчислюється за формулою: S = (max div 8) - (min div 8) + 1, де max та min – верхня і нижня границі базового типу даної множини, а div– цілочисельне ділення.

Запис – це комбінований тип, значення якого представляють собою нетривіальну структуру даних. Вони складаються з декількох полів різного типу, доступ до яких здійснюється за їхніми іменам. Записи представляють собою засіб для представлення програмних моделей реальних об'єктів предметної області, так як кожен такий об'єкт володіє декількома властивостями, які можуть описуватись даними різних типів.

Приклад запису – набір даних про співробітника.

Об'єкт «співробітник» може володіти наступними властивостями:

- табельний номер – ціле додатне число;
- прізвище, ім’я, по-батькові – рядок символів і т.д.;
- стать – символ;
- посада – рядок символів;
- заробітна плата – дійсне число;

В пам’яті ця структура може бути представлена в одному з двох видів:

- у вигляді послідовності полів, які займають неперервну область пам’яті. Щоб отримати доступ до будь якого елемента запису, потрібно знати адресу початку запису і зміщення відносно початку. При цьому досягається економія пам’яті комп’ютера, але затрачається зайвий час на обчислення адрес полів;
- у вигляді зв’язного списку з вказівниками на значення полів запису. При такій організації має місце швидкий доступ до елементів, але неекономна трата пам’яті для збереження.

Властивості напівстатичних структур даних:

- вони мають змінну довжину і прості способи її зміни;
- зміна довжини структури відбувається у визначених межах, не перевищуючи деякого максимального (граничного) значення.

З логічної точки зору напістатична структура представляє собою послідовність даних, зв’язану відношеннями лінійного списку. Доступ до елемента може здійснюватись по його порядковому номеру.

Фізично напістатичні структури представляються чи у вигляді вектора, тобто розміщуються в неперервній області пам’яті, чи у вигляді однозв’язного списку, де кожен наступний елемент адресується вказівником, який знаходиться в поточному елементі.

До напівстатичних структур належать стеки, черги, деки, рядки.

Динамічні структури не мають постійного розміру, тому пам’ять під окремі елементи таких структур виділяється в момент, коли вони створюються в процесі

виконання програми, а не під час трансляції. Коли в елементі структури більше немає необхідності, пам’ять звільняється (елемент «руйнується»).

Оскільки елементи динамічної структури розміщуються в пам’яті не по порядку і навіть не в одній області, адреса елементу такої структури не може бути обчислена з адреси початкового чи попереднього елемента. Зв’язок між елементами динамічної структури встановлюється через вказівники, які містять адреси елементів пам’яті. Таке представлення даних в пам’яті називається зв’язним.

Таким чином, крім інформаційних полів, заради який створюється структура і які є видимими для кінцевого користувача програмного забезпечення, динамічні структури містять поля для зв’язку з іншими елементами, видимі для розробника програмного забезпечення.

З допомогою зв’язного представлення даних забезпечується висока мінливість структури. Переваги динамічних структур:

- розмір структури обмежується тільки об’ємом пам’яті комп’ютера;
- при зміні логічної послідовності елементів структури (видалення, додавання елемента, зміна порядку розміщення елементів) вимагається тільки корекція вказівників.

З іншого боку, такі структури володіють ряд недоліків:

- робота з вказівниками вимагає високої кваліфікації програміста;
- на вказівники витрачається додаткова пам’ять;
- додаткова трата часу на доступ до елементів зв’язної структури.

Лінійні зв’язні списки є найпростішими динамічними структурами даних. Вони є впорядкованими множинами, які містять змінне число елементів, на які не накладаються обмеження по довжині.

На рисунку 7 наведена структура однозв’язного списку. Тут поле Inf – інформаційне поле, яке містить дані, NEXT – вказівник на наступний елемент списку. Голова списку – вказівник на початок списку. Вказівник на наступний

елемент останнього елемента містить значення NULL, це є ознакою останнього елемента.

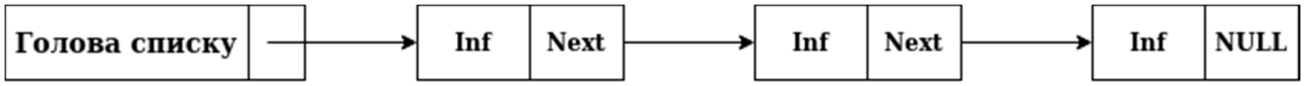


Рисунок 7 - Структура однозв’язного списку

Обробка однозв’язного списку не завжди зручна, так як неможливо рухатись в протилежну сторону. Таку можливість забезпечує двозв’язний список, кожен елемент якого містить два вказівника: на наступний і попередній елементи. Для зручності обробки списку додають ще один особливий елемент – вказівник кінця списку. Наявність двох вказівників в кожному елементі ускладнює список і призводить до додаткових затрат пам’яті, але в той же час забезпечує більш ефективне виконання деяких операцій над списком.

Питання для самоконтролю

1. Що таке структура даних?
2. Що таке прості типи даних? Наведіть приклади простих типів даних.
3. Які типи даних відносяться до базових?
4. Що таке структуровані типи даних? Наведіть приклади структурованих типів даних.
5. Що таке динамічні структури даних? З чого вони складаються?
6. Які види зберігання лінійних списків ви знаєте?
7. Яка різниця між логічною та фізичною організацією структур даних?
8. Опишіть структур даних типу “запис”.
9. Як визначити розмірність вектора, якщо відомими є його загальний об’єм та базовий тип?
10. Опишіть структур даних “багатовимірний масив”.
11. Що задається при оголошенні структури даних?

Лекція 7. Масиви

У програмуванні масив – одна з найпростіших структур даних, сукупність елементів переважно одного типу даних, впорядкованих за індексами, які зазвичай репрезентовані натуральними числами, що визначають положення елемента в масиві.

Масив може бути одновимірним (вектором), та багатовимірним (наприклад, двовимірною таблицею), тобто таким, де індексом є не одне число, а кортеж (сукупність) з декількох чисел, кількість яких співпадає з розмірністю масиву.

В переважній більшості мов програмування масив є стандартною вбудованою структурою даних.

Масиви ефективні при звертанні до довільного елементу, яке відбувається за постійний час (O(1)), однак такі операції як додавання та видалення елементу, потребують часу O(n), де n – розмір масиву. Тому масиви переважно використовуються для зберігання даних, до елементів яких відбувається довільний доступ без додавання або видалення нових елементів, тоді як для алгоритмів с інтенсивними операціями додавання та видалення, ефективнішими є зв'язані списки.

Інша перевага масивів, яка є досить важливою – це можливість компактного збереження послідовності їх елементів в локальній області пам'яті (що не завжди вдається, наприклад, для зв'язаних списків), що дозволяє ефективно виконувати операції з послідовного обходу елементів таких масивів.

Масиви є дуже економною щодо пам'яті структурою даних. Для збереження 100 цілих чисел в масиві необхідно рівно в 100 разів більше пам'яті, ніж для збереження одного числа (плюс, можливо, ще декілька байтів). В той же час, усі структури даних, які базуються на вказівниках, потребують додаткової пам'яті для збереження самих вказівників разом з даними. Однак, операції з фіксованими масивами ускладнюються тоді, коли виникає необхідність додавання нових елементів у вже заповнений масив. Тоді його необхідно розширювати, що не

завжди можливо і для таких задач слід використовувати зв’язані списки, або динамічні масиви.

У випадках, коли розмір масиву є досить великий та використання звичайного звертання за індексом стає проблематичним, або великий відсоток його комірок не використовується, слід звертатись до асоціативних масивів, де проблема індексування великих об’ємів інформації вирішується більш оптимально.

З тої причини, що масиви мають фіксовану довжину, слід дуже обережно ставитись до процедури звертання до елементів за їхнім індексом, тому що намагання звернутись до елементу, індекс якого перевищує розмір такого масива (наприклад, до елементу з індексом 6 в масиві з 5 елементів), може призвести до непередбачуваних наслідків.

Слід також бути уважним щодо принципів нумерації елементів масиву, яка в одних мовах програмування може починатись з 0, а в інших – з 1.

Таким чином масив – це структура даних, яка характеризується:

- фіксованим набором елементів одного і того ж типу;
- кожен елемент має унікальний набір значень індексів;
- кількість індексів визначають розмірність масиву;
- звернення до елементу масиву виконується по імені масиву і значенням індексів для даного елементу.

Збереження одновимірного масиву в пам’яті є тривіальним, тому що сама пам’ять комп’ютера є одновимірним масивом. Елементи розміщують послідовно,

один за одним. Також немає потреби запам'ятовувати адресу всіх елементів масиву. Запам'ятовується лише адреса першого елементу і до цієї адреси додається кількість байт, яку займає кожен елемент, помножена на порядковий номер кожного елемента.

Для збереження багатовимірного масиву ситуація ускладнюється. Припустимо, що ми хочемо зберігати двовимірний масив наступного виду:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

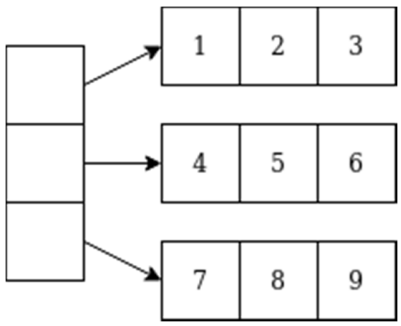
Найпоширеніші способи його організації в пам'яті такі:

- Розташування «рядок за рядком». Це найбільш уживаний на сьогодні спосіб, який зустрічається у більшості мов програмування.

  1 2 3 4 5 6 7 8 9

  У такому випадку адреса кожного елемента масиву обчислюється за наступною формулою: <адреса a_{ij}> = <адреса a_{11}> + ((i – 1)*m + j – 1) * кількість байт вказаного типу.
- Розташування «стовпчик за стовпчиком». Такий метод розташування масивів використовується, зокрема, в мові програмування Fortran.

  1 4 7 2 5 8 3 6
- Масив з масивів. Багатовимірні масиви репрезентуються одновимірними масивами вказівників на одновимірні масиви. Розташування може бути як «рядок за рядком» так і «стовпчик за стовпчиком».



Перші два способи дозволяють розміщувати дані компактніше (мають більшу локальність), однак це одночасно і обмеження: такі масиви мають бути «прямокутними», тобто кожний рядок повинен містити однакову кількість елементів. Розташування «масив з масивів», з іншого боку, не дуже ефективне щодо використання пам'яті (необхідно зберігати додатково інформацію про вказівники), але знімає обмеження на «прямокутність» масиву.

Питання для самоконтролю

1. Дайте означення масиву.
2. Чим характеризується масив?
3. Як відбувається звертання до елементів масиву?
4. Як зберігається у пам'яті комп'ютера одновимірний масив?
5. Як зберігається у пам'яті комп'ютера двовимірний масив?
6. У чому полягає ефективність використання масивів?

Лекція 8. Динамічні структури даних: список, стек, дек, черга

Для статичних типів даних характерним є те, що для їх збереження область в оперативній пам'яті виділяється на етапі компіляції програми та не змінюється у процесі її виконання. Але при розв'язуванні прикладних задач трапляються випадки, коли обсяг оперативної пам'яті, для зберігання даних, неможливо визначити заздалегідь. У таких випадках використовують типи даних, які можуть бути створені та видалені в процесі виконання програми.

Динамічні структури даних характеризуються наступними ознаками:

1. Змінна довжина і прості процедури її зміни.

2. Зміна довжини структури відбувається в певних межах, не перевищуючи певного (граничного) значення.

Якщо динамічну структуру розглядати на логічному рівні, то це послідовність даних, пов'язаних за відношенням лінійного списку. Доступ до елемента може здійснюватися за його порядковим номером. Фізичне представлення

динамічної структури даних в пам’яті – це послідовність комірок в пам’яті, де кожний наступний елемент розташований в наступній комірці пам’яті.

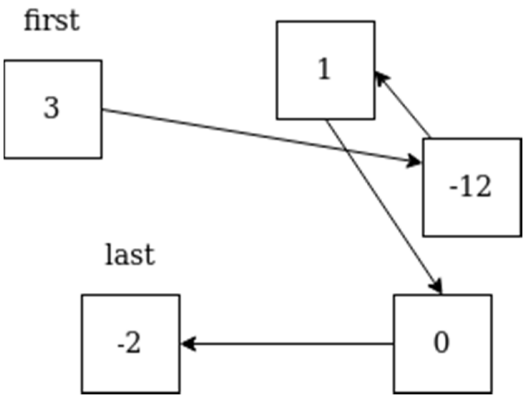
Фізичне представлення може мати також вид однонаправленого зв’язного списку (ланцюжки), де кожний наступний елемент адресується покажчиком, який знаходиться в поточному елементі. У цьому випадку обмеження на довжину структури менш строгі.

Список це послідовність з $n \geq 0$ елементів $x[0], x[1], \ldots x[n-1]$, для якої виконується наступна умова: якщо $n > 0$ та $x[0]$ перший елемент у списку, а $x[n-1]$ останній, то k-тий елемент розташований між $x[k-1]$ та $x[k+1]$ елементами для усіх $1< k < n$.

В оперативній пам’яті одновимірний масив з 5-ти елементів, наприклад, цілих чисел 3, -12, 1, 0, -2 розміщується наступним чином:

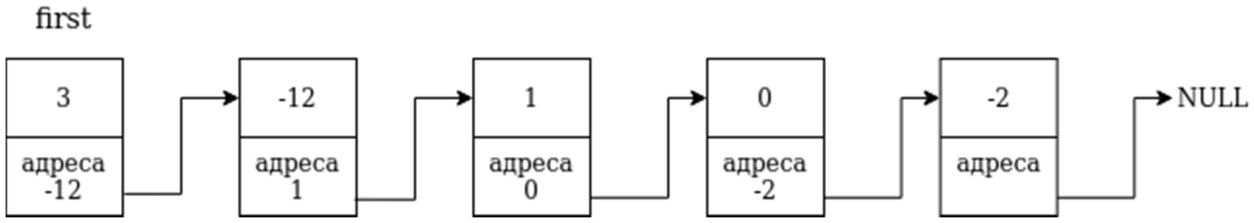
3 -12 1 0 -2

Це означає, що елементи розташовані в пам’яті один за одним, що дає можливість їх нумерувати, але ускладнює хід роботи програми. Зокрема, це стосується операцій додавання нових елементів, чи їх вилучення. На відміну від масиву, елементи списку розташовані в оперативній пам’яті випадковим чином, що дозволяє без проблем додавати чи видаляти елементи з довільної позиції. Наприклад, список елементів, аналогічний попередньому масиву, може бути розташований у пам’яті наступним чином:



На рисунку стрілками задано порядок проходження елементів списку. Для того, аби працювати з елементами як зі зв’язною структурою, необхідно задати, де в пам’яті розташовано кожний з елементів, тобто фактично знати їхні адреси. Цілком достатньо зберігати тільки адресу першого елемента (покажчик на елемент First). Кожний наступний елемент списку повинен містити інформацію про розташування наступного за ним елементу. При цьому немає необхідності в даному елементі зберігати інформацію про розташування решти елементів. До будь-якого *i*-того елементу можна дістатись, послідовно пройшовши по ланцюжку від першого до другого, від другого до третього і т. д аж до і-того елемента.

Для організації списку кожен елемент, окрім даних, повинен також містити ще одне значення: адресу наступного елемента. Зручно подати елементи списку у вигляді структури, яка містить два поля: поле даних (значення елемента) та поле адреси (вказівник на наступний елемент). Після останнього елемента більше елементів немає, тому вказівник на наступний елемент по винен залишатися порожнім (NULL). Верхня частина кожного елементу задає його числове значення (поле даних), а нижня – задає адресу наступного елементу (поле next).



Лінійний зв’язаний список – це набір однотипних компонентів, які послідовно пов’язані між собою за допомогою покажчиків. Кожен компонент списку може складатися із кількох інформаційних полів та покажчика на наступний елемент. Інформаційні поля елементу списку можуть бути змінними будь-яких типів Зі такими структурами типу список можуть бути реалізовані наступні операції:

- отримання k-того елемента списку для читання чи запису в нього нового значення:
- додавання нового елемента в будь-яку позицію в списку;
- видалення елемента списку;
- об’єднання в одному списку двох або більше лінійних списків;
- розбиття списку на два або більше фрагментів;
- створення копії списку;
- визначення кількості елементів в списку;
- сортування елементів списку;
- обмін елементів списку місцями;
- пошук елемента, що задовільняють певним критеріям.

Усі можливі варіанти застосування операцій вставки та видалення елементів із списку:

- створення списку, тобто внесення першого елемента до списку;
- додавання елемента в кінець списку;
- додавання елемента на початої: списку;
- вставка елемента в середину списку;
- видалення елемента з початку списку;
- видалення елемента з кінця списку;
- видалення елемента з середини списку.

У загальному випадку для роботи з однозв’язним лінійним списком потрібні такі покажчики:

- покажчик head на початок списку;
- покажчик current на поточний елемент списку;
- покажчик previous на елемент, розташований перед поточним;
- покажчик newptr на елемент, що додається до списку;
- покажчик last на кінець списку.

Розглянем алгоритм вставки елемента всередину списку.

Вважаємо, що новий елемент має бути вставлений між елементами previous^ і current^.

1. Новий елемент вважати наступним для previous^:

previous^.next:= newptr.

2. Елемент current^ вважати наступним для нового елемента:

newptr^. next: = current.

На рисунку нижче продемонстровано алгоритм вставки елемента всередину списку.

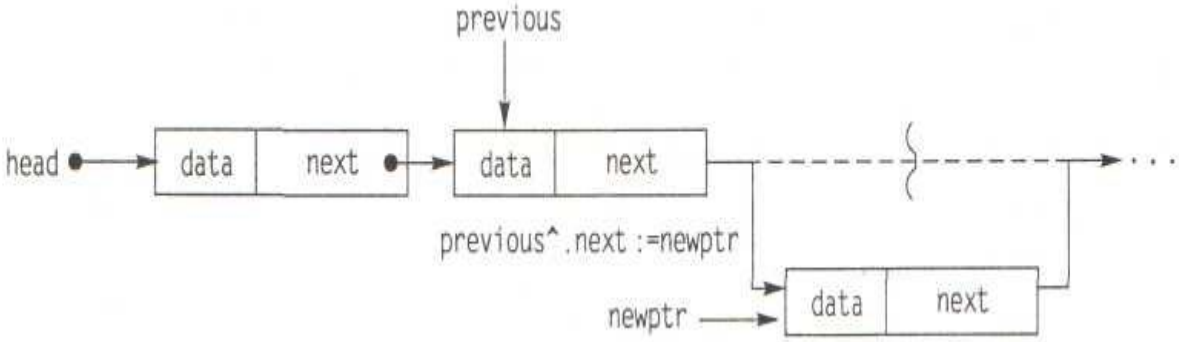


Рисунок 8 – Вставки елемента всередину однозв’язного списку

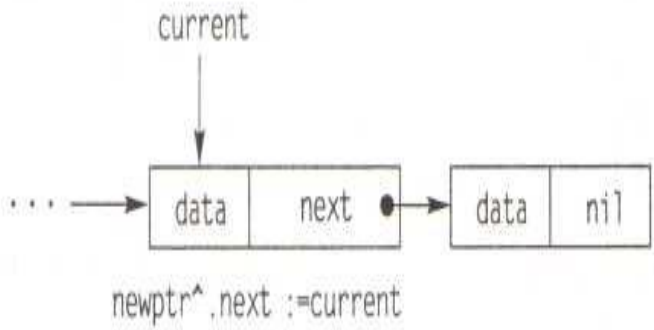


Рисунок 9 – Зміна вказівників при виконанні операції вставки елемента всередину однозв’язного списку

Розглянемо алгоритм видалення елемента з середини списку. Вважаємо, що видаляється елемент current^, розташований безпосередньо за елементом previous^.

1. Вважати, що за елементом previous^ буде розташований той елемент, що раніше знаходився за елементом
   current^: previous^.next:=current^.next;
   previous.next:-current.next
2. Звільнити пам’ять із-під елемента current^:
   Dispose (current);

Алгоритм вставки елемента в кінець однозв’язного списку має наступний вигляд. Вважаємо, що на передостанній елемент посилається покажчик previous.

1. Записати до передостаннього елемента ознаку кінця списку:
   previous^. next:= nil.
2. Звільнити пам’ять із-під колишнього останнього елемента:
   Dispose (last).
3. Вважати останнім колишній передостанній елемент:
   last:=previous.

Списки мають ряд переваг над масивами. Вони досить ефективні щодо операцій додавання або видалення елементу в довільному місці списку, виконуючи їх за постійний час, тоді як масиви для цього потребують часу O(n), тобто час зростає з ростом кількості елементів масиву. В списках також не існує проблеми «розширення яка рано чи пізно виникає в масивах фіксованого розміру, коли виникає необхідність включити в нього додаткові елементи. Точно так, фіксований масив, з якого було видалено багато елементів (або вони просто не використовуються) є досить неефективним з точки зору використання пам’яті. Функціонування списків можливо в ситуації, коли пам’ять комп’ютера фрагментована, тоді як масиви переважно потребують неперервної області для зберігання.

З іншого боку, масиви дозволяють безпосередній доступ до будь-якого елементу. Однобічно зв’язані списки, натомість, потребують проходження усіх попередніх елементів. Це призводить до складнощів застосування списків в

задачах, де необхідно швидко знаходити елемент за його індексом, наприклад, в алгоритмах сортування. Кешування списків в таких випадках майже не дає ефекту. Іншим очевидним недоліком списків є необхідність разом з корисною інформацією додаткового збереження інформації про вказівники, що позначається на ефективності використання пам'яті цими структурами. Списки використовуються замість масивів для зберігання й опрацювання однотипних даних, кількість яких заздалегідь є невідома й може змінюватися у процесі роботи. У списках просто організувати процеси видалення елементу чи його вставку на довільне місце. Принципова перевага списку перед масивом полягає у їх структурній гнучкості: порядок слідування елементів списку може не співпадати із порядком розміщення елементів у пам'яті комп'ютера, а порядок обходу списку завжди задається явним чином за допомогою його внутрішніх зв'язків. Лінійний зв'язаний список – це набір однотипних компонентів, які послідовно пов'язані між собою за допомогою покажчиків. Кожен компонент списку може складатися із кількох інформаційних полів та покажчика на наступний елемент. Інформаційні поля елементу списку можуть бути змінними будь-яких типів.

Стек, черга і дек – це однотипні лінійні структури. Їх називають динамічними лінійними структурами даних у зв'язку з тим ,що вибірка елемента із цих послідовностей зводиться до операції вилучення, тобто ліквідації його у послідовностях. Відрізняються ці структури одна від одних порядком виконання операцій введення і вилучення елементів.

До основних операцій над цими структурами належать:

a) створення нової структур даних;

b) запис або введення елемента до вже наявної структури даних;

c) вилучення елемента зі структури;

d) перевірка умови існування структури.

Стек – це впорядкована лінійна послідовність елементів, яка динамічно змінюється і у якій виконуються наступні умови:

- новий елемент приєднується завжди до того самого боку послідовності;
- доступ до елемента здійснюється завжди з отого боку послідовності, до якого приєднується елемент;
- елемент зберігається в послідовності до моменту його виклику.

Прикладом стеку є стос тарілок або магазин патронів у рушниці. З аналогією з останнім прикладом стек ще часто називають магазином з пам'яттю. Операцію введення елемента до стека називають «проштовхуванням», а операцію вилучення із послідовності – «виштовхуванням». Найбільш і найменш доступні елементи стеку називають, відповідно, верхом і низом стека. Операції ведення вилучення із стека. виконують з одного і того ж боку послідовності, але вилучаються елементи зі стека у послідовності, зворотній до тієї, в якій вони потрапили до послідовності. У кожний момент часу зі стека можна забрати лише один елемент. Дисципліну обслуговування стека називають дисципліною LIFO: «last in first out».

Незалежно від того, як реалізовано стек, процедури, що працюють із ним, повинні вміти вмішувати елементи в стек, вибирати елемент зі стеку й перевіряти, чи не порожній стек. Для того, щоб можна було працювати з новим тлом даних, необхідні спеціальні операції, наприклад:

NEW (стек) – створення порожнього стека;

PUSH (стек, елемент даних) – розміщення елемента в стек;

POP (стек) – вибірка елемента з вершини стека;

EMPTY (стек) – перевірка, чи порожній стек; результатом є значення «істина», якщо стек порожній, і «хибно» у протилежному випадку.

NEW() і PUSH() базові операції, використовувані піл час роботи зі стеком. Перша створює стек, а друга вмішує в нього елементи. POP() процедура, характерна тільки для роботи зі стеком. Її можна визначити за допомогою двох базових проміжних операцій TOP() і REMOVE(), які дають змогу виділити дві основні функції, реалізовані POP(), але не визначають способів організації доступу до структури з інших модулів. Відповідно до наведених вище визначень, POP()

вибирає вершину стека й видаляє її зі стека. Вершина стека описується так: якщо стек не порожній, вершина являє собою останній доданий до стека елемент; а якщо ні, то стек не має вершини. Операція REMOVE() дає в результаті стек без елемента, розміщеного в нього останнім. До тільки-но створеного стека операцію REMOVE() застосовувати не можна. Для визначення порожнього стека використовується поняття нового (або тільки-но створеного) стека: порожній стек не містить жодного елемента. Будь-яка реалізація стека повинна задовольняти ці визначення. Важливо, що стек визначається як спосіб зберігання даних, які підпорядковуються певним правилам, що діють при звертанні до даних, а не як масив зі вказівниками, які пересуваються за певними правилами. Основною перевагою стека перед іншими організаціями даних є те, що у ньому не потрібна адресація елементів. Для обслуговування стека потрібні лише дві команди: PUSH() – «проштовхнути» і POP() – «виштовхнути».

Зі стеком пов’язаний завжди один вказівник, який вказує на верхній елемент у стеку. На початку такий вказівник дорівнює нулю. Найпростішим прикладом вдалого застосування стека може бути задача передавання вектора у зворотній послідовності. На практиці стекова структура даних найчастіше застосовується в рекурсивних алгоритмах, під час трансляції, а також обробки переривань програм. Стек дає змогу організувати рекурсію. тобто звернення підпрограми до самої себе або безпосередньо, або через ланцюжок інших викликів. Нехай, наприклад, підпрограма А виконує алгоритм, що залежить від вхідного параметра Х і, можливо, від стану і глобальних даних. Для найпростіших значень Х алгоритм реалізується безпосередньо у випадку складніших значень Х алгоритм реалізовується як звернення до застосування того самого алгоритму для простіших значень Х. При цьому підпрограма А звертаються сама до себе, передаючи як параметр простіше значення Х. Під час такого попереднє значення параметра Х, а також усі локальні змінні підпрограми А зберігаються в стеці. Потім створюють новий набір локальних змінних і змінну, що містить нове (простіше) значення

параметра X. Викликана підпрограма A працює з новим набором змінних, не руйнуючи попереднього набору. Після закінчення виклику старий набір локальних змінних і старий стан вхідного параметра X відновлюються зі стека, і підпрограма продовжує роботу з того місця, де її було перервано.

Реалізація стека на базі масиву є класикою програмування. Іноді навіть саме поняття стека не зовсім коректно ототожнюється із цією реалізацією. Базою реалізації є масив розміром *N*. Так реалізується стек обмеженого розміру, максимальна глибина якого не може перевищувати *N*. Індекси елементів масиву змінюються від 0 до *N*-1. Елементи стека зберігаються в масиві так: елемент на дні стека розташовується на початку масиву, тобто в елементі з індексом 0. Елемент, розташований над найнижчим елементом стека, зберігається в елементі з індексом 1, і так далі. Вершина стека зберігається десь у середині масиву. Індекс елемента на вершині стека зберігається в спеціальній змінній, яку зазвичай називають вказівником стека (англійською Slack Pointer, або просто SP).

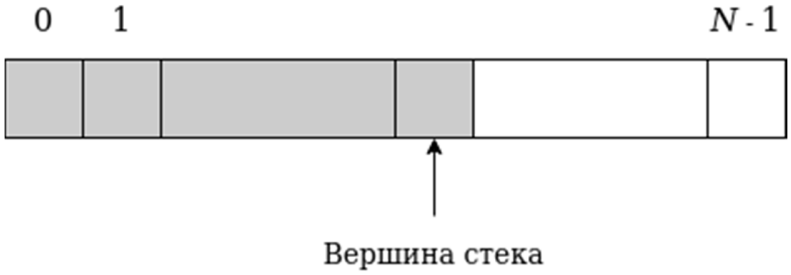


Рисунок 10 – Реалізація стеку на базі масиву

Коли стек порожній, вказівник стека містить значення -1. Під час введення елемента вказівник стека спочатку збільшується на одиницю, потім у елемент масиву з індексом, що зберігається у вказівнику стека, записується елемент, який вводять .У разі видалення елемента зі стека спершу вміст елемента масиву з індексом, що зберігається у вказівнику стека, запам’ятовується в тимчасовій змінній як результат операції, потім вказівник стека зменшується на одиницю. У наведеній реалізації стек зростає у бік збільшення індексу елементів масиву. Часто

використовується інший варіант реалізації стека: на базі вектора, коли дно стека міститься в останньому елементі масиву, тобто в елементі з індексом *N*-1. Елементи стека займають неперервний відрізок масиву, починаючи із елемента, індекс якого зберігається у вказівнику стека, і закінчуючи останнім елементом масиву. За цим варіантом стек росте в бік зменшення індексів. Якщо стек порожній, то вказівник стека містить значення *N*, яке на одиницю більше, ніж індекс останнього елемента вектора.

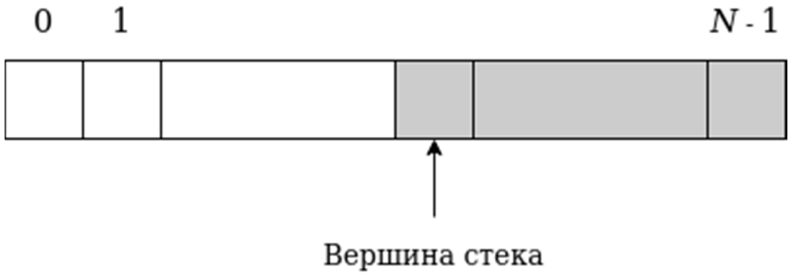


Рисунок 11 – Реалізація стеку на базі масиву зі зменшенням індексу

Черга – це лінійна послідовність елементів, що змінюється динамічно та у якій виконуються такі умови:

- новий елемент приєднується завжди з одного і того самого боку послідовності;
- доступ до елементів або їх вилучення завжди здійснюється з іншого боку послідовності.

Наприклад, нехай послідовність $Q = \{Q_1,.....Q_n\}$ позначає чергу. Тоді елемент Q_1 називають «головою» черги, який вказує на місце для вилучення елемента; елемент Q_n називають «хвостом» черги, який вказує на місце для введення елемента до черги. Отже, в структурі черги потрібні два вказівники: один для посилання на вершину послідовності (для введення елемента), другий – для посилання на основу послідовності (для вилучення елемента). За кожного вилучення елемента із такої структури вилучається завжди найстаріший елемент.

Черги обслуговуються за дисципліною FIFO - «first in first out». Типовими операціями для черги є:

- SIZE.(черга) – визначення кількості елементів у черзі;
- PUSH (черга, елемент даних) – додавання елемента в кінець черги;
- POP (черга) – вибірка елемента з початку черги;
- EMPTY (черга) – перевірка чи порожня черга; результатом і значення «істина», якщо черга порожня, і «хибно» у протилежному випадку.

Розрізняють такі типи черг:

- лінійні черги;
- черги з пріоритетом;
- циклічні черги.

Звичайну лінійну чергу можна зобразити масивом із двома вказівниками: перший вказує на елемент для вибірки з черги, другий на останній елемент, записаний у чергу. Багаторазове звертання до елементів лінійної черги призводить до того, що пам’ять для зберігання такої черги використовується неефективно. У лінійній черзі після вибірки елемента пам’ять, зайнята чергою, звільняється і повторно не використовується, оскільки нові елементи приписуються і іншого краю послідовності. Чергу, в якій є можливість додавати або вилучати елементи з певної позиції залежно від деяких її характеристик, називають пріоритетною чергою. Прикладом пріоритетної черги може бути порядок розв’язуваній потоку задач у деяких операційних системах. Така черга зводиться до послідовності лінійних черг, якщо відомі пріоритети її елементів. Кожна черга з послідовності обслуговується за дисципліно FIFO, але елементи з другої черги обслуговуються тільки тоді, коли порожня попередня черга, а з третьої – тоді, коли порожні перша і друга черги. Під час введення елементи приєднуються до боку однієї з черг згідно з їхнім пріоритетом.

Черги, в яких елементи Q_1, Q_2... Q_n розмішуються так, що за елементом Q_n розміщується елемент Q_1, називаються циклічними. Циклічну чергу також можна

зобразити лінійною послідовністю, але при записі нового елемента на відміну від лінійної черги вся послідовність зсувається на одне поле і новий елемент записується знову на її початку. Вибірка елемента буде також виконуватися за вказівником на кінець послідовності. Прикладом циклічної черги може бути робота обчислювальної системи з розподілом часу, з якою одночасно працює багато користувачів. Оскільки така система має переважно один блок обробки, який називають процесором, і одну пам'ять, то в кожний момент часу ці ресурси належать одному користувачеві. Для кожного користувача виділяються певний інтервал часу. Тільки на відміну від пріоритетної черги одна задача до кінця не розв'язується, а робить це «порціями» протягом виділеною їй інтервалу часу. Програми, що чекають на виконання, утворюють циклічну чергу.

Використання черги в програмуванні майже відповідає її ролі у звичайному житті. Черга практично завжди пов'язана з обслуговуванням запитів у тих випадках, коли вони не можуть бути виконані миттєво. Черга підтримує також порядок обслуговування запитів.

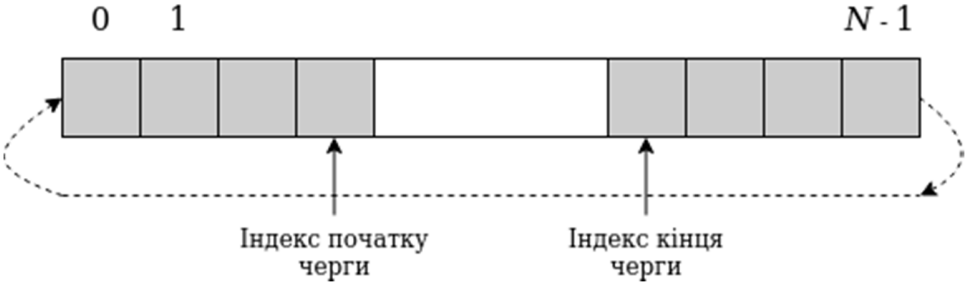
Усі описані вище структури даних можна реалізовувати на основі масиву. Зазвичай, така реалізація може бути багатоетапною і не завжди масив є безпосередньою базою реалізації. У випадку черги найпопулярніші дві реалізації: безперервна на основі масиву, яку називають також реалізацією на основі кільцевого буфера, і реалізація з посиланнями, або реалізація на основі списку.

При безперервній реалізації черги базою є масив фіксованої довжини *N*. Тобто черга обмежена й не може містити понад *N* елементів (рис. 12). Індекси елементів масиву змінюються в межах від 0 до *N*-1. Крім масиву, реалізація черга зберігає три прості змінні: індекс початку черги, індекс кінця черги, число елементів черги. Елементи черга зберігаються у відрізку масиву від індексу початку до індексу кінця.

Рисунок 12 - Реалізація черги на основі масиву

При додаванні нового елемента в кінець черга індекс кінця спочатку збільшується на одиницю, потім новий елемент записується в елемент масиву із цим індексом. Аналогічно, в разі видалення елемента з початку черга вміст елемента масиву з індексом початку черги запам’ятовується як результат операції, потім індекс початку черги збільшується на одиницю. Як індекс початку черги, так і індекс кінця під час роботи рухаються зліва направо. Що відбувається, коли індекс кінця черги досягає кінця масиву, тобто *N*-1?

Ключова ідея реалізації черги полягає в тому, що масив ніби зациклюється в кільце (рис. 13). Вважають, що за останнім елементом масиву розміщений його перший елемент (останній елемент мас індекс *N*-1, а перший – індекс 0). При зрушенні індексу кінця черга праворуч. коли він вказує на останній елемент масиву, він переходить на перший елемент. Тобто, безперервний відрізок масиву, зайнятий елементами черг, може перехопити через кінець масиву на його початок.



Ричунок 13 - Просування черги масивом

Дек – це впорядкована лінійна динамічно змінювана послідовність елементів, у якій виконуються такі умови:

- новий елемент може приєднуватися з обох боків послідовності;
- вибірка елементів можлива також з обох боків послідовності.

Наприклад, якщо послідовність $D = D_1 ... D_n$ зображує дек, елементи D_1 та D_n, вказують одночасно на місце введенні і вилучення елементів. Деки називають також реверсивними, або двосторонніми чергами, в яких введення і вилучення елемента здійснюють з двох боків послідовності.

Типовими операціями для дека є:

- PUSHBACK (дек, елемент даних) – додавання в кінець дека;
- PUSHFRONT (дек, елемент даних) – додавання на початок дека;
- POPBACK (дек) – вибірка з кінця дека;
- POPFRONT (дек) – вибірка з початку дека;
- EMPTY (дек) – перевірка, чи порожній дек.

Деки бувають з обмеженнями з одною боку на виконання однієї з операцій (наприклад, один бік послідовності може бути закритий для вибірки елементів і відкритий тільки для приєднування, і навпаки). Тоді вони стають звичайними чергами з дисципліною обслуговування FIFO. Як і у випадку звичайних черг, для обслуговування деків також потрібні два вказівники - по одному за кожного боку послідовності.

Деки є найзагальнішою лінійною динамічною структурою даних. Задачі, що вимагають структури дека, трапляються в обчислювальній техніці і програмуванні набагато рідше, ніж задачі, що реалізовуються на структурі стека або черги. Прикладом дека може бути певний термінал, до якого вводяться команди, кожна з яких виконується заданий час. Якщо ввести наступну команду і не дочекатись закінчення виконаній попередньої, то вона встане у чергу і почне виконуватися, як тільки звільниться термінал. Це черга з дисципліною обслуговування FIFO. Якщо ж додатково ввести операцію відміни останньої введеної команди, то виходить дек.

Питання для самоконтролю

1. Що таке динамічна структура даних?
2. Які структури даних відносять до динамічних?
3. Дайте означення списку.
4. Як розміщуються елементи у списку?
5. Які існують види списків?
6. Назвіть основні операції, які виконуються над списками.
7. Опишіть алгоритм видалення елементу із довільної позиції у списку.
8. Опишіть алгоритм додавання елементу у довільну позицію у списку.
9. Назвіть переваги використання списків у порівнянні із масивами.
10. Що таке стек?
11. Яка структура даних обслуговується за дисципліною LIFO?
12. Які операції можна виконати над стеком?
13. Які переваги стека над іншими способами організації даних?
14. Опишіть прикладне застосування структури даних «стек».

    Що таке черга?
15. Яка структура даних обслуговується за дисципліною FIFO?
16. Які операції можна виконати над чергою?
17. Які переваги черги над іншими способами організації даних?
18. Опишіть прикладне застосування структури даних «черга».
19. Дайте ви значення лінійної черги.
20. Дайте визначення пріоритетної черги.
21. Дайте визначення циклічної черги.
22. Що таке дек?
23. Які операції можна виконати над деком?
24. Опишіть прикладне застосування структури даних «дек».

Лекція 9. Графи. Деякі алгоритми на графах

Зважаючи на те, що значну частину складних задач вдається сформулювати в термінах теорії графів, а конструювання структур даних для подання у програмі об'єктів математичної моделі є однією із основоположних задач практичного програмування, природним є питання про способи подання вихідних даних про самі графи. Для графів існує декілька основних способів подання їх у машинній пам'яті. Вибір найкращого із них визначається вимогами тієї чи іншої задачі. Крім того, при розв'язанні конкретних прикладних задач використовуються, як правило, деякі комбінації або модифікації вказаних способів, але незважаючи на це усі вони ґрунтуються на способах представлення графів, які описані нижче.

Означення: Нехай задано граф *G(V, E)*: $|V| = n$, $|E| = m$. Матрицею інцидентності графу *G(V, E)* називають матрицю $B = [b_{ij}]$ розмірності $n \times m$, елементи якої обчислюються за правилом:

$$b_{ij} = \begin{cases} 1, \text{вершина } \mathrm{v_i} \text{ інцидентна ребру } e_j \in E \\ 0, \text{ вершина } \mathrm{v_i} \text{ не інцидентна ребру } e_j \in E \end{cases}$$

Означення: Нехай дано граф *G (V, E)*: $|V| = n$, $|E| = m$. Матрицею суміжності графу *G(V, E)* називають матрицю $A = [a_{ij}]$ розмірності $n \times n$, елементи якої обчислюються за правилом:

$$a_{\mathrm{ij}} = \{1, (v_i, \mathrm{v}) \in E.$$

Матриця є формальним математичним поняттям, при чому з алгебраїчної точки зору – одним із найбільш ефективних. Відповідна матриця графу (матриця суміжності, суміжності ребер, інцидентності) визначається єдиним способом з точністю до перестановок рядків та (або) стовпців. Але неоднозначність матричного представлення не впливає на складність операцій, які виконуються над матрицями, оскільки перестановка стовпців та рядків (це по суті найпростіші операції над матрицями) не збільшує ні ємнісної, ні часової складності операцій, що виконуються над матрицями.

Використовуючи матрицю суміжності графу, можна представити його у вигляді двійкового коду. Очевидно, якщо запам'ятати послідовність 0 та 1 матриці суміжності, то за нею можна відновити уесь граф. Так як матриця *A* симетрична, то достатньо запам'ятати лише ті елементи, які розміщені над головною діагоналлю, тобто послідовність a_{12}, a_{13}, a_{14}, ...a_{n-1n}. Довжина цієї послідовності становить $C_n^2 = \frac{n(n-1)}{2}$.

Число, яке отримується після виконання операції $a_{1,2} \cdot 2^0 + a_{1,3} \cdot 2^1 + a_{2,3} \cdot 2^2 + a_{1,4} \cdot 2^3 + \ldots + a_{n-1,n} \cdot 2^{c_n^2-1}$, називають двійковим *кодом* матриці. Оскільки різним нумераціям вершин графу відповідають різні матриці суміжності, то існує *n!* таких кодів. Найменший із цих кодів називають міні-кодом μ (G), а найбільший – максі-кодом μ (G).

Таким чином, за умови подання графу за допомогою двійкового коду після переведення максі-коду μ (G) у двійкове число ми отримаємо матрицю суміжності графу.

Якщо граф подано у вигляді двійкового коду кількість вершин можна не вказувати при умові, що граф не має ізольованих вершин. Якщо задавати граф за допомогою максі-коду, стає очевидним, що він завжди міститиме одиницю старшого розряду 2^k. Таким чином, максі-код μ (G) відповідає такій нумерації вершин графу, при якій вершини з номерами *n – 1* та *n* суміжні. В силу цього, число вершин графу однозначно визначається із умови: $n = \frac{1+\sqrt{1+8k}}{2}$. Виняток становить лише той випадок, коли μ (G) = 0, тобто коли G – безреберний граф (нуль-граф). У такому випадку кількість його вершин може бути іншою.

Ще одним способом подання графів у машинній пам'яті є списки суміжності. Подання графу *G(V, E)* як списками вершин, так і списками суміжності ребер, мають одну і ту ж структуру. Тому природно визначити стандартну форму цих представлень єдиним чином. Уніфіковане представлення має наступний вигляд.

Означення: Стандартною формою представлення графу *G(V, E)*, $\left(m,n \in N, 0 \le m \le \left(\frac{n}{2}\right)\right)$списками суміжності називають таку *k*-елементну послідовність списків LIST: $LIST_1, LIST_2 \ldots LIST_k$, де

$$k = \begin{cases} n, для\ списків\ суміжності\ вершин \\ m, для\ списків\ суміжності\ ребер \end{cases},$$

що елементи кожного списку $LIST_i$ (*i=1, 2, ...k*) розміщені в порядку зростання. Тобто вершини графу *G*, суміжні з вершиною *i*, у випадку списків суміжності вершин, або ребра графу *G*, суміжні з ребром *i*, у випадку списків суміжності ребер.

В означенні зафіксовано порядок, в якому елементи, що належать до лінійно-впорядкованої множини S=({*1, 2, ...k* }, <), розміщені у кожному із списків $LIST_i$ (*i=1, 2, ...k*). А саме вибрано порядок «», що відповідає зростанню елементів у відповідності до їх лінійного порядку «<« у вихідній множині, тобто прийнято узгодження про те, що порядки «» та «<« співпадають. Конкретний вид порядку, у якому розміщені елементи у списках, в загальному випадку значення не має. Але для ефективної роботи із такою структурою даних необхідно, щоб відношення порядку «» можна було визначити у термінах вихідного відношення «<«. Той факт, що при використанні списків суміжності важливою є фіксація порядку слідування елементів, обумовлений наступним: представлення підмножин лінійно-впорядкованої множини списком з фіксованим порядком слідування елементів є ефективною структурою даних, призначеною для виконання теоретико-множинних операцій.

Спосіб представлення графу із *n* вершинами та *m* ребрами за допомогою списків суміжності вимагає *(n + m)* одиниць пам'яті. Списками суміжності доцільно користуватись у тих випадках, коли *m* значно менше від n^2, тобто коли граф розріджений.

Але існує ряд задач, в яких важливо тільки визначити суміжність двох довільних вершин графу, що розглядається. Тому виникає необхідність отримати про граф інформацію такого виду за допомогою обчислень певного роду, не загромаджуючи пам'ять машини вихідними даними про структуру самого графу.

В останній час теорія графів стала інтенсивно використовуватись у різноманітних керуючих системах та інформаційних моделях, найважливішою властивістю яких є структура або сукупність бінарних відношень на наборах елементарних одиниць та дій. Залежно від конкретної задачі можна використовувати той чи інших спосіб подання графу. Зокрема, можна визначити граф як бінарне відношення на множині його вершин.

Очевидним є те, що будь – яке означення задовольняє універсальному визначенню графу, яке наводиться нижче.

Означення: Графом називається трійка $G = (X, U, F)$, де $X = \{x_1, x_2, \ldots x_n\}$ – множина вершин, $U = \{u_1, u_2, \ldots u_p\}$ – множина твірних, F – деяка однозначна функція від двох аргументів (породжуючи функція) така, що:

$$\forall u \exists i \exists j (u \in U) \wedge (x_i, x_j \in X) \wedge [F(x_i, x_j) \in U]$$

Відносно функції F вважають, що вона має явний аналітичний вигляд.

Такі графи, що задаються аналітичним способом, називають аналітичними. Аналітичний граф скінченний, якщо множина X – скінченна. Елементами множини X можуть бути довільні об'єкти (числа, множини, матриці) та найбільший інтерес представляє випадок, коли елементами множини X є числа (цілі, раціональні, дійсні, комплексні). Такі графи називають числовими.

Означення: Числовим графом $G=(X, U, F, g)$ називають n-вершинний граф, що представлений двома множинами: $X=\{1, 2, \ldots n\}=N_n$ – множиною вершин; U N – множиною твірних; функцією суміжності $F(x_i, x_j)$ та функцією виключення $g(x)$.

У таких графах вершина $x_k X$, якщо $g(x_k)=0$, а вершини $x_i, x_j X$ є суміжними, якщо $F(x_i, x_j)$ U.

Якщо $F(x_i, x_j) = x_i + x_j$, то такий числовий граф називають арифметичним (*A*-графом). Якщо $X=\{1, 2, \ldots n\}$, то такі графи називають натуральними арифметичними графами (*NA*-графами). Якщо функція суміжності має наступний вигляд: $F(x_i, x_j) = |x_i - x_j|$, то граф називають натуральним модульним графом (*NM*-графом). Відносно функції $g(x)$, то для неї ніяких особливих властивостей не передбачено, вона може просто перераховувати множину вершин, що не належать множині X вершин графу.

Враховуючи той факт, що розв'язання прикладних задач передбачає використання графів великої розмірності на практиці матрицями суміжності користуються досить рідко. Списками суміжності зручно користуватись тоді, коли у графі визначаються різні маршрути або графи використовуються для представлення мереж великого об'єму. У випадку наявності у графі великої розмірності фрагментів, що повторюються або декількох осей симетрії варто використовувати числові графи.

Значна частина практичних задач, що виникають при формуванні розкладів, транспортуванні, плануванні перевезень, часто можуть бути представлені як задачі теорії графів, тісно пов'язані із так званою «задачею розфарбування» або «задачею розкладання».

При розкладанні графів звертаються до двох основних задач: розфарбування (розкладання) вершин та ребер. Розкладання ребер графу полягає у приписуванні цим ребрами таких кольорів, що жодних два суміжних ребра не мають однакового кольору. Такий розподіл кольорів називають власним розфарбуванням ребер. Аналогічно означають і розкладання вершин графу.

Означення**:** Розфарбуванням вершин графу $G=(V, U)$ у k кольорів (чи k – *розфарбуванням*) називають таке розбиття множини V, при якому кожна підмножина $V_i(\bigcup_{i=1}^{k} V_i = V, V_{ia} \cap V_{ib} = \varnothing, i_a, i_b = 1,2 \ldots k)$ не містить жодної пари

суміжних вершин. Кожній підмножині ставиться у відповідність колір, у який зафарбовуються всі елементи.

Хроматичним числом *h(G)* графу *G* називається мінімальне число *k*, для якого граф має *k*-розфарбування. Граф, хроматичне число якого рівне *k*, називають *k-хроматичним*.

Тобто задача розфарбування (розкладання, декомпозиції) графу за його вершинами полягає у відшуканні його хроматичного числа.

Зважаючи на те, що задача розкладання графу за його вершинами є достатньо відомою, для її розв'язання розроблено ряд методів, а на їх основі – алгоритмів, які дозволяють знайти розфарбування графу. Такі алгоритми поділяться на два класи:

- точні, які гарантують відшукання оптимального розфарбування вершин та істинного значення хроматичного числа довільного графу.
- наближені, які не завжди знаходять точний розв'язок, але є більш ефективними.

Зокрема, до наближених алгоритмів відносяться алгоритм послідовного розфарбування та покращений алгоритм послідовного розфарбування, до точних – точний алгоритм розкладання, алгоритм розкладання, що базується на використанні максимальних *r* – підграфів. Нижче наведено опис кожного із цих алгоритмів.

Алгоритм послідовного розфарбування. Довільній вершині v_1 графу *G* приписується колір 1. Якщо вершини $v_1, v_2, \ldots v_i$ розфарбовані *k* кольорами, $k \le i$, то новій, довільним чином вибраній, вершині v_{i+1} приписується мінімальний колір, що не був використаний при побудові розфарбування суміжних вершин.

Вхід: граф *G*.

Вихід: розфарбування графу – масив C: **array** [1..p] **of** 1..p.

for vV **do**

C[v]:=0 {*усі вершини не пофарбовані*}

end for

for vV **do**

A:={1, …p} {*усі кольори*}

for uΓ^+ **do**

A:=A\{C[u]} {*зайняті усі кольори*}

end for

C[v]:=min A {*мінімальний вільний колір*}

end for

Покращений алгоритм послідовного розфарбування. Алгоритм будує допустиме розфарбування способом, аналогічним до попереднього випадку, але на відміну від попереднього, розфарбування розпочинається із вершини з найбільшим степенем. Причина вибору вершини із максимальним степенем наступна: якщо розфарбовувати таку вершину в останню чергу, то може виявитись, що для неї немає вільного кольору.

Вхід: граф *G*.

Вихід: розфарбування графу – масив C: **array** [1..p] **of** 1..p.

Sort(v) {*впорядкувати вершини за спаданням степенів*}

c:=1

for v V **do**

C[v]:=0 {*усі вершини не зафарбовані*}

end for

while V≠ø **do**

for v V **do**

for uΓ^{+}(v) **do**

if C[u]=c **then**

next for v {*вершину не можна пофарбувати у колір c*}

end if

end for

c[v]:=c {*зафарбовуємо вершину у колір c*}

V:=V\{v} {*видаляємо вершину із переліку тих, які будуть розглядатись*}

end for

c:=c+1

end while

Точний алгоритм розфарбування. Даний алгоритм базується на використанні незалежних множин вершин графу.

Означення: Незалежною називається підмножина вершин графу, у якій жодні дві вершини не є суміжними

Схема рекурсивної процедури P, що будує розфарбування графу, має наступний вигляд:

1. Вибрати у графі *G* деяку максимальну незалежну підмножину *S*.
2. Зафарбувати вершини множини *S* черговим кольором.
3. Застосувати процедуру P до графу *G* – *S*.

Вхід: граф *G*, номер вільного кольору *i*.

Вихід: розфарбування графу, задане масивом C[v] - номери кольорів приписані вершинам.

if V=ø **then**

return {*розфарбування закінчено*}

end if

s:=selectmax (G) {*S – максимальна незалежна множина*}

C[S]:=i {*зафарбовуємо вершини множини у колір i*}

P(G-S, i+1) {*рекурсивний виклик*}

Побудова максимальної незалежної множини вершин графу (selectmax) розпочинається із порожньої множини і в ході виконання алгоритму поповнюється вершинами із зберіганням незалежності цієї множини.

Вхід: граф *G(V, E)*.

Вихід: послідовність максимально незалежних множин.

k:=0 {*кількість елементів у поточній незалежній множині*}

S[k]:=ø {*незалежна множина із k вершин*}

Q^-[k]:=ø {*множина вершин, що уже використані для розширення S[k]*}

Q^+[k]:=V {*множина вершин, які можна використати для розширення S[k]*}

M1: {*крок вперед*}

select *v* Q^+[k] {*розширяюча вершина*}

S[k+1]:=S[k] *v* {*розширена множина*}

Q^-[k+1]:= Q^-[k] \ Γ[*v*] {*вершина v використана для розширення*}

Q^+[k+1]:= Q^+[k] \ (Γ[*v*] {*v*}) {*усі вершини, суміжні з v не можуть бути використані для розширення*}

k:=k+1

M2:

for u Q^-[k] **do**

if Γ[u] Q^+[k]= ø **then**

goto M3 {*можна повертатись*}

end if

end for

if $Q^+[k]:= ø$ **then**

if $Q^-[k]:= ø$ **then**

yield S[k] {*множина S[k] максимальна*}

end if

goto M3 {*можна повертатись*}

else goto M1 {*можна іти вперед*}

end if

M3: {*крок назад*}

v:=last(S[k]) {*останній доданий елемент*}

k:=k-1

S[k]:=S[k+1] – {*v*}

$Q^-[k]:= Q^-[k]$ {*v*}

$Q^+[k]:= Q^+[k] \setminus$ {*v*} {*вершина v уже додавалась*}

if k=0 **and** $Q^+[k]= ø$ **then**

stop {*перебір завершено*}

else goto M2 {*перехід на перевірку*}

end if

Алгоритм, що базується на використанні максимальних *r*-підграфів.

Означення**:** Породжений підграф $<S_r[G]>$ графу $G=(X, Г)$, де $S_r[G]X$, називають *r*-підграфом, якщо він *r*-хроматичний. Якщо не існує такої множини *H*,

що $HS_r[G]$ і підграф H є r-хроматичним, то $<S_r[G]>$ називають максимальним r-підграфом графу G. [34]

Хроматичне число графу є найменшим значенням r, при якому $S_r[G]$ – множина вершин деякого максимального r – підграфу – співпадає із множиною вершин V графу. Тому процедуру послідовної побудови r – підграфів можна використовувати для знаходження хроматичного індексу графу у тих випадках, коли на кожному кроці процедури перевіряти, чи не міститься множина вершин графу у котромусь із знайдених r – підграфів.

Вхід: граф $G(V, E)$.

Вихід: послідовність множин, що утворює розбиття графу. Вершини кожної із множин пофарбовані різними кольорами.

```
r:=1
Q:={S^j_r[G] | j=1, … q_r} {знаходимо максимальну множину r – підграфів}
label
k:=Selectmax(X - S^j_r[G]); {знаходимо максимальну незалежну множину}
if k≠ø then
        S:= S^j_r[G]S_1[G]
          if S=X then stop {(r+1) – хроматичне число, розфарбування знайдено}
  else
        case S of
                SS′ then goto label { S′  Q}
                S S′ then Q:=Q – {S}  { S′  Q}
                        else Q:=Q  {S} goto label
        end case
        end if
```

else

if $j<q_r$ **then** j:=l+1 **goto** label

end if

if $j=q_r$ **then** j:=l+1

r:=r+1

q_r:=|Q| **goto** label

end if

Не зважаючи на те, що розглянутий алгоритм відноситься до точних алгоритмів розкладання графів за їх вершинами, він не дає повного переліку усіх можливих розфарбувань у *r+1* колір, а тільки породжує оптимально незалежні розфарбування. Такі розфарбування можуть бути лише невеликою частиною усіх можливих розфарбувань у *r+1* колір.

Задача відшукання гамільтонових циклів у графі є однією із найстаріших та модельною задачею теорії графів, а також має численні застосування як у теорії графів, так і у її прикладних галузях. [90]

Означення: Гамільтоновим циклом у графі *G (V, E)* називають цикл, що проходить через кожну із вершин графу рівно по одному разу.

Означення: Граф, що має гамільтонів цикл, називають гамільтоновим.

На даний час немає достатньо простого критерію чи алгебраїчного методу, який би дозволив відповісти на питання: чи існує у графі гамільтонів цикл? Критерії гамільтоновості графів, подані у роботах Дж.Ф. Неша, Ф. Уіл'ямса та О. Оре, представляють теоретичний інтерес, але являються досить загальними результатами та не є придатними для використання на практиці.

Алгебраїчні критерії гамільтоновості не можуть бути застосовані до графів, які мають більш, ніж декілька десятків вершин, оскільки потребують значних витрат часу та великої кількості пам'яті. Спосіб С.М. Робертса та Б. Флореса не

вимагає значних витрат памяті, але час його роботи залежить експоненціально від кількості вершин графу.

Основна частина доведених теорем стверджує, що при виконанні деяких умов граф містить гамільтонів цикл. Як правило, метод доведення таких теорем дає і ефективний метод побудови самих гамільтонових циклів.

Очевидно, якщо граф містить велику кількість ребер і ці ребра достатньо рівномірно розподілені, то ймовірність існування гамільтонового циклу у ньому достатньо висока.

Алгоритм із поверненням. Для того, аби перерахувати усі гамільтонові цикли графу, даний алгоритм багатократно виконує наступну операцію: маючи простий поточний ланцюг

$P: v_0 = u_0, u_1, \ldots, u_{n-1},$

він по черзі додає до нього нові вершини, продовжуючи ланцюг до усіх можливих простих ланцюгів із початком у вершині v_0. Для відшукання гамільтонового циклу використовується наступна рекурсивна процедура:

```
1.  Procedure Hamilton_Cycle(k);
2.  begin
3.  for yS_k do
4.    if k=n-1 and connected(y, v_0) then
5.    write (v_0, x[1], x[2], …x[n-2], v_0) else
6.    begin
7.     status[y]:=1, S_k:= S_k\{y}; x[k]:=y;
8.     S_{k+1}:={v | v  list[y] and status[v]=0};
9.     Hamilton_Cycle(k);
10.    Status[x[k]]:=0;
11.   end;
12. end;
```

Таким чином описана вище процедура перебирає усі прості ланцюги, які є продовженням ланцюга $v_0, x[1], \ldots x[n-1]$, додаючи по-черзі нові вершини. Якщо знайдено максимальний простий ланцюг, то відбувається повернення: одна чи декілька вершин відкидається (у тому випадку, коли знайдений максимальний простий ланцюг міститься у гамільтоновім циклі, перед поверненням цей цикл виводиться на друк). Реалізований алгоритм відноситься до класу алгоритмів з поверненням. Алгоритм, що перераховує всі гамільтонові цикли у графі має наступний вигляд:

begin

status[v_0]:=1; S_k:=*list*[v_0];

for v $V \setminus \{v_0\}$ **do** *status*[v]:=0;

Hamilton_Cycle(1);

end

Такий алгоритм, розглядаючи вершину v_0 як початкову, повертає усі гамільтонові цикли графу.

Approx-TSP(G). Даний алгоритм було розроблено спеціально для розв'язання задачі комівояжера (дана задача заключається у знаходженні гамільтонового циклу мінімальної довжини [105]) у метричному просторі. Задача комівояжера у метричному просторі є частковим випадком задачі комівояжера, де відстані вихідного графу задовольняють нерівності трикутника:

$$d(x_1, x_3) \le d(x_1, x_2) + d(x_2, x_3),$$

де x_1, x_2, x_3 – вершини графу, $d(x_i, x_j)$ – відстань між заданими вершинами).

Алгоритм Approx-TSP(G) складається із наступних кроків:

1. Побудувати мінімальний кістяк графу. Оскільки граф, що розглядається є ненавантаженим, то для виконання цього кроку алгоритму достатньо побудувати звичайний кістяк графу. Для цього скористаємось *алгоритмом пошуку в ширину*. Алгоритм має наступний вигляді. Вибирають та відвідують початкову (довільну) вершину графу x_0. Після цього вибирають ребро (x_0, x_i) інцидентне x_0, та

відвідують вершину x_i. Нехай x – остання відвідана вершина. Виберемо довільне ребро (x,y), інцидентне x, яке ще не розглядалось. Якщо вершина y уже відвідувалась, то шукаємо нове ребро, яке інцидентне x. Якщо y – не відвідана вершина, ідемо до неї та розпочинаємо пошук з цієї вершини. У тому випадку, коли переміщення вперед є неможливим, повертаємось до вершини x (до тієї вершини, по якій ми прийшли до y) та розпочинаємо пошук з неї. В результаті ми повернемось до вершини x_0 та виявимо неможливість продовжувати пошук. Після цього шукаємо нову вершину x_1, яка не відвідувалась раніше, та продовжуємо з неї пошук. Пошук закінчено у тому випадку, коли усі вершини графу були відвідані.

Загальний вигляд алгоритму наступний:

for $v \in V$ **do**
  $x[v]:=0$ {усі вершини не відмічені}
end for
select $v \in V$ {початок обходу – довільна вершина}
$v \rightarrow T$ {помістити v у структуру даних T}
$x[v]:=1$ {помітити вершину v}
repeat
  $u \leftarrow T$ {вилучити вершину із структури даних T}
  yield u {повернути її в якості наступної пройденої вершини}
  for $w \in \Gamma(u)$ **do**
    if $x[w]=0$ **then**
      $w \rightarrow T$ {помістити w у структуру даних T}
      $x[w]:=1$ {помітити вершину w}
    end if
  end for
until $T = \varnothing$

Якщо T – стек (LIFO – last in first out), то обхід називається пошуком в ширину. Якщо T – черга (FIFO – first in first out), то обхід називається пошуком в глибину.

2. Продублювати кожне із ребер знайденого кістяка та в отриманому графі відшукати ейлерів цикл. (Ейлеровим називають цикл, який містить усі ребра та вершини графу. [67])

Для відшукання ейлеревого циклу слід скористатись наступним алгоритмом [14]:

1. **begin**
2. *SWork:=nil, SRes:=nil,*
3. $v_0 \rightarrow SWork$
4. **for** $v \in V$ **do** $listW[v]:=list[v]$
5. **while** $SWork \neq nil$ **do**
6. **begin**
7. *v:=top(SWork)*
8. **if** $listW(v) \neq \emptyset$ **then**
9. **begin**
10. *u:=перша вершина listW*[*v*]
11. $u \rightarrow SWork$
12. $listW(v):=listW(v) \setminus \{u\}$
13. $listW(u):=listW(u) \setminus \{v\}$
14. **end**
15. **else**
16. **begin** $v \leftarrow SWork$; $SRes \leftarrow v$ **end**
17. **end**
18. **end**

SWork та SRes – стеки, елементами якових є вершини графу. Принци роботи описаного алгоритму полягає в наступному. Алгоритм розпочинає роботу із деякої вершини v_0 та переходить по ребрах графу, при чому кожне ребро і графу

видаляється. Зрозуміло, що виконання групи операторів рядків 10-13 дозволить виділити у графі деякий замкнений ланцюг. Потім розпочинається виконання групи операторів із рядка 16. Ці оператори «виштовхують» чергову вершину із стеку SWork до стеку SRes до того часу, поки не виконається одна із умов:

1) стек SWork порожній (алгоритм закінчує свою роботу);

2) для вершини v=top(SWork) існує не пройдене ребро uv. У такому випадку алгоритм розпочинає роботу із вершини u.

3. У знайденому ейлеревому циклі видалити усі вершини, які повторюються. Отриманий цикл буде гамільтоновим циклом вихідного графу. Алгоритм Approx-TSP(G) відноситься до наближених алгоритмів.

Алгоритм із поліноміальним часом. Такий алгоритм передбачає виконання наступних кроків:

1. Знайти цикл, який довільним чином містить усі вершини графу. У такому циклі дві сусідні вершини можуть бути не суміжними у вихідному графі. Такі точки будемо називати «break point».

2. Для кожної такої точки, крім одної (її називають «main beak point»), додати нове ребро (слід запам’ятати додані ребра, оскільки кожного разу видаляючи ребро, ми будемо розпочинати роботу алгоритму спочатку).

3. Виправити «main beak point»: вирізати сегмент (частину) із «main beak point» та вставити цей сегмент у деяке довільне місце у циклі.

4. Виправити наступну «main beak point».

5. Виправлення (пункти 2-3) здійснювати до того часу, поки не буде знайдено гамільтонів цикл на графу.

Виникає питання: яким чином здійснювати таку вставку та вирізання? Правило наступне: зробити кількість нових «break point» якнайменшим і кожна нова «break point» повинна відрізнятись від усіх попередніх «main break point». При визначенні наступної «main break point» може трапитись так, що більш ніж, одна «break point» підходить для цього. У такому випадку слід порівняти наступний, але лише один,

крок для кожної із цих точок. Також слід уникати вставки одного і того ж сегменту у різні частини циклу послідовно.

Питання для самоконтролю

1. У чому полягає задача розкладання або розфарбування графу?
2. Що таке хроматичне число графу?
3. Опишіть алгоритм розкладання графу за допомогою кістяків.
4. Який цикл називають гамільтоновим?
5. Опишіть алгоритм з поверненням відшукання гамільтонового циклу на графі.
6. Опишіть поліноміальний алгоритм відшукання гамільтонового циклу на графі.
7. Опишіть алгоритм Approx-TSP(G) відшукання гамільтонового циклу на графі.
8. На які класи поділяються алгоритми відшукання гамільтонових циклів?
9. Назвіть способи представлення графів у машинній пам’яті.
10. Які переваги та недоліки представлення графів у вигляді матриці суміжності?
11. Які переваги та недоліки представлення графів у вигляді матриці інцидентності?
12. Які переваги та недоліки представлення графів у вигляді двійкового коду?
13. Які переваги та недоліки представлення графів у вигляді матриці числових графів?

Лекція 10. Алгоритми пошуку числових даних

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.

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*).

Програмна ілюстрація лінійного пошуку в неврегульованому масиві наведена в наступному прикладі, де a – початковий масив, key – ключ, який шукається; функція повертає індекс знайденого елемента.

```
int LinSearch(int *a, int key) {
        int i = 0;
                while ( (i<N) && (a[i] != key) )
                i++;
return i;
}
```

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".

```
int LinSearch(int *a, int key) {
a[N] = key;
i = 0;
while (a[i] != key)
i++;
return i; // i<N – повернення номера елемента
```

}

Очевидно, що інших способів пришвидшення пошуку не існує, якщо, звичайно, немає ще ядеякої інформації про дані, серед яких ведеться 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. Записи в таблицю заносяться в лексикографічному (символьні ключі) або чисельно (числові ключі) зростаючому порядку. Для досягнення впорядкованості може бути використаний котрийсь з методів сортування, які розглянемо пізніше.

Оскільки шуканий елемент швидше за все знаходиться «десь в середині», перевіримо 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 $log_2(N)$ порівнянь. Це значно краще, ніж при послідовному пошуку.

Приведемо ілюстрація бінарного пошуку на прикладі.

```
int BinSearch(int *a, int key){
int b, e, i;
b = 0; e = N-1; // початкові значення меж
bool Found = false; // прапорець
while ( (b < e) && !Found) // цикл, поки інтервал пошуку не звузиться до 0
{
i = ( b + e ) / 2; // середина інтервалу
if ( a[i] == key )
Found = true; // ключ знайдений
else
if ( a[i] < key )
```

```
b = i + 1; // пошук в правому підінтервалі
else
e = i - 1; // пошук в лівому підінтервалі
}
return i;
}
```

Максимальна кількість порівнянь для цього алгоритму рівна $log_2(N)$. Таким чином, приведений алгоритм суттєво виграє у порівнянні з лінійним пошуком. Ефективність дещо покращиться, якщо поміняти місцями заголовки умовних операторів. Перевірку на рівність можна виконувати в другу чергу, так як вона зустрічається лише одноразово і приводить до завершення роботи. Але більш суттєвий виграш дасть відмова від завершення пошуку при фіксації знаходження елемента.

```
int BinSearch(int *a, int key){
int b, e, i;
b = 0; e = N-1;
while (b<e){
i = (b + e) / 2;
if (a[m] < x)
b = i + 1;
else
e = i - 1;
}
return i
}
```

Завершення циклу гарантовано. Це пояснюється наступним. На початку кожного кроку b<e. Для середнього арифметичного і справедлива умова b<=i<e. Значить, різниця e-b дійсно спадає, тому що або b збільшується при присвоєнні

йому значення i+1, або e зменшується при присвоєнні йому значення i-1. При b<=i повторення циклу закінчується. Виконання умови b=e ще не засвідчує знаходження потрібного елемента. Тут потрібна додаткова перевірка. Також, необхідно враховувати, що елемент a[e] у порівняннях ніколи не бере участі. Значить, і тут необхідна додаткова перевірка на рівність a[e]=key. Але ці перевірки виконуються однократно.

Алгоритм бінарного пошуку можна представити і трохи інакше, використовуючи рекурсивний опис. В цьому випадку граничні індекси інтервалу b і e є параметрами алгоритму. Рекурсивна процедура бінарного пошуку представлена в наступній програмі. Для виконання пошуку необхідно при виклику процедури задати значення її формальних параметрів b і e – 0 і N-1 відповідно, де b, e – граничні індекси області пошуку.

```
int BinSearch(int *a, int key, int & b, int & e){
int i;
if ( b > e )
return -1; // перевірка ширини інтервалу
else
{
i = ( b + e ) / 2; // середина інтервалу
if ( a[i] == key )
return i; // ключ знайдений, повернення індексу
else
if ( a[i] < key ) // пошук в правому підінтервалі
return BinSearch(a, key, i+1, e);
else // пошук в лівому підінтервалі
return BinSearch(a, key, b, i-1);
}
}
```

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

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», тобто такий, номер якого пропорційний значенню key:

Програмна реалізація такого варіанту пошуку матиме вигляд:

```
int BinSearch(int *a, int key){
int b, e, i;
b = 0; e = N-1; // початкові значення меж
while ( b < e ) // цикл, поки інтервал пошуку не звузиться до 0
{
i = b + (key – a[b])*(e-b) / (a[e] – a[b]);
if ( a[i] == key )
return i; // ключ знайдений - повернення індексу
else
if ( a[i] < key )
b = i + 1; // пошук в правому підінтервалі
else
e = i - 1; // пошук в лівому підінтервалі
}
return -1; // ключ не знайдений
}
```

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.

Питання для самоконтролю

1. Сформулюйте задачу пошуку.
2. Для пошуку якої інформації використовується метод послідовного пошуку?
3. В якому випадку використовують двійковий пошук?
4. Поясніть процедуру двійкового пошуку на власному прикладі.
5. Чому дорівнює мінімальна і максимальна кількість порівнянь при двійковому пошуку?

Лекція 11. Алгоритми пошуку підрядка в рядку

Даний клас задача відноситься до задачі пошуку слів у тексті. Нехай масив *a[N]* вважається масивом символів останній елемент якого – 0: char a[N]; у якому слід знайти заданий рядок символів: S = $s_0s_1s_2\ldots s_m$ довжиною m.

Одним з найпростіших методів пошуку є послідовне порівняння першого символу *s* з символами масиву *a*. 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 *a* та першого символу рядку *s*. Це визначається елементарною програмою:

```
i = 0;          // номер символу масиву a
while (i < N - lenghts)
{
   j = 0;       // номер символу рядка s
   while ((s[j] == a[i+j]) && (j<lenghts)
      j++;
   if (j==lenghts)
```

```
    return i;       // успіх
}
```

Якщо збіги відбуватимуться досить часто, то час роботи програми може бути досить значним.

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 *s* немає фрагментів, що повторюються.

```
j = 0; // j - номер символа b a
found = false;
while (!found)
{
   i = 0;       // i - номер символа b s
   while ((s[i] == a[j]) && (s[i] != '/0')
   {
      i++;
      j++;
   };
   if (s[i] == '/0'
      found = true;
   else
      j -= i-1;
};
```

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 пересунутися текстом.

Основною відмінністю алгоритму Кнута-Моріса-Пратта (КМП) 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*. Наведені приклади пояснюють функцію *d*.

Розглянемо опис цього методу у вигляді псевдокоду.

```
int d[M]; j = 0;  k = -1; d[0] = -1;
while (i < M-1) { // попереднє заповнення масиву d зсувів
   while ((k>=0) && (s[i] != s[k]))
      k = d[k];
   i++;
   k++;
   if (s[i] == s[k])
      d[i] = d[k]
   else
```

```
        d[i] = k;
}
i = 0;
j = 0;
k = 0;
while ((i < M) && (j < N)){
    while (k <= j)  {
        cout << a[k];
        k++;
    }
    while ((i>=0) && (a[j] != s[i]){
        i = d[i];
    j++;
    i++;
}
if (i == M)
```

Алгоритм Кнута-Моріса-Пратта 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.

Алгоритм Боуєра-Мура (БМ) 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.

На початку роботи слід завести масив, який зберігав би для кожного символу, що може зустрітися у масиві *a*, значення зсуву. Для символів, що взагалі не зустрічаються у образі *s*, зсув дорівнює *M* – довжині образу. Для символів, що зустрічаються у *s*, зсув буде меншим, щоби не пропустити можливих попадань.

Алгоритм у вигляді псевдокоду можна записати таким чином.

```
for (ch=0; ch<256; ch++)
   d[ch] = M; // замовчування
for (i=0; i<M-1; i++)
   d[s[i]] = M-i-1; // уточнення
i = M;
do
{
   j = M;
   k = i;
   do // Цикл порівняння символів
   {
      k--;
      j---; // слова, начинаючи з правого
   while ( (j<0) || (a[j]!=s[k]) ); //Вихід
   i += d[s[i-1]]; // Зсув слова вправо
while ( (j<0) || (i>N));
```

У випадку постійних незбігів цей алгоритм робить одне порівняння на *M* символів.

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.

Питання для самоконтролю:

1. Сфомулюйте задачу пошуку підрядка у рядку.
2. Опишіть алгоритм прямого пошуку. Продемонструйте на прикладі.
3. Опишіть алгоритм Кнута-Моріса-Пратта. Продемонструйте на прикладі.
4. Опишіть алгоритм Боуєра-Мура. Продемонструйте на прикладі.

Лекція 12. Алгоритми сортування. Детальний аналіз алгоритму швидкого сортування

Основне питання сортування — це економія пам'яті, тобто переставлення на тому ж самому місці.

Методи сортування поділяються на

1. сортування за допомогою включення (insersion sort);
2. сортування за допомогою вибору (selection sort);
3. сортування за допомогою обмінів (exchange sort).

За кількістю виконуваних операцій при виконанні сортування методи поділяються на

1. прямі;
2. покращені;
3. удосконалені.

Прямі методи сортування.

Сортування за допомогою прямого включення

Нехай задана послідовність a[1], a[2], ..., a[n]. Відомо, що на i-му кроці a[1], a[2], ..., a[i-1] - відсортована частина послідовності. Ідея метода полягає в тому, що

для кожного i=2,3,...,n береться елемент a[i] і ставиться у відсортовану частину послідовності на своє місце.

```
for (i=2; i<=n; i++){
    x:=a[i]; a[0]:=x; j:=i;
    while (x<a[j-1]){
        a[j]:=a[j-1];
        j:=j-1
      }
    a[j]:=x
  }
```

Визначення ефективності можна поділити на два моменти — кількість порівнянь для визначення місця знаходження елемента a[i] в послідовності a[1], a[2], ..., a[i-1] та кількість зсувів для його вставки на своє місце.

Метод з двійковим включенням (binary insertion). Оскільки a[1], a[2], ..., a[i-1] вже впорядкована частина послідовності, то для знаходження його місцеположення можна використати бінарний пошук.

Сортування за допомогою прямого вибору

Алгоритм:

- вибирається елемент з найменшим значенням;
- він міняється місцями з першим елементом a[1];
- цей процес повторюється з (n-1) елементами, що залишились, потім з (n-2) і т.д.

Другий метод ефективніший, якщо масив не впорядкований. У випадку частково упорядкованого або майже упорядкованого масиву перший метод має значні переваги по кількості порівнянь.

Сортування за допомогою прямого обміну

Хоча два попередні методи використовують моменти обміну, у даному методі обмін є найхарактернішою особливістю.

Кожний раз розглядаємо з кінця в початок два сусідніх елемента і міняємо їх місцями, якщо правий елемент менший за лівий. При цьому відбувається “виштовхування” найлегшого (найменшого) елемента в початок.

Такий метод ще носить назву “бульбашкового” сортування.

```
for (i=2; i<=n; i++)
  for (j=n; j>=i; j--)
    if (a[j-1]>a[j]){
       x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x
       }
```

Запам’ятати, чи в останньому перегляді були виконані переставлення. Якщо ні, то сортування можна припиняти. Ще одне удосконалення — запам’ятовувати на кожному кроці, де був виконаний останній обмін. Це означатиме, що всі елементи вище (або нижче) цього місця вже упорядковані. Тому наступний перегляд можна виконувати лише до цього елемента.

Покращені методи сортування

«*Шейкерне» сортування*

Розглянемо два приклади масивів і сортування методом обміну.

12 18 42 44 55 67 94 06.

В цьому прикладі поганий елемент знаходиться на «важкому кінці». Він “спливе” на своє місце за один прохід.

94 06 12 18 42 44 56 67.

В цьому прикладі поганий елемент на «легкому кінці». Він «потоне» за сім проходів.

Метод прямого обміну можна удосконалити таким чином: чередувати направлення послідовних переглядів масиву. Цей метод носить назву «шейкерного» сортування.

Розглянемо даний метод на прикладі.

repeat

```
  for j:=R downto L do
    if a[j-1]>a[j] then
       begin
         x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x; k:=j
       end;
  L:=k+1;
  for j:=L to R do
    if a[j-1]>a[j] then
       begin
         x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x; k:=j
       end;
  R:=k-1;
until L>R;
```

Удосконалені методи сортування

Сортування за допомогою дерева (пірамідальне сортування)

Сортування за допомогою прямого вибору побудоване на відшуканні на кожному кроці найменшого (або найбільшого) значення.

```
    i:=L; j:=2*L; x:=a[L];
    if (j<R) and (a[j+1]<a[j]) then j:=j+1;
    while (j<=R) and (a[j]<x) do
       begin
         a[i]:=a[j]; i:=j;j:=2*j;
         if (j<R) and (a[j+1])<a[j]) then j:=j+1
       end;
    a[i]:=x
  end;
```

Для того, щоб побудувати піраміду з усіх елементів, необхідно виконати процедуру для віcх елементів, що залишились, тобто (n-(n div 2) +1)-раз:

```
L:=(n div 2)+1;
while L>1 do
  begin
    L:=l-1;
    sift(L,n)
  end;
```

Розглянемо запропонований метод на прикладі:

44 55 12 42 | 94 18 06 67

44 55 12 | 42 94 18 06 67

44 55 | 06 42 94 18 12 67

44 | 42 06 55 94 18 12 67

06 42 12 55 94 18 44 67

Одержали на першому місці найменший елемент. Щоб відкинути вже одержаний найменший елемент, його міняють місцями з останнім і більше не розглядають. З останніми (n-1) елементами знову виконують вже описану процедуру.

```
R:=n;
while R>1 do
  begin
    x:=a[1]; a[1]:=a[R]; a[R]:=x;
    R:=R-1; sift(1,R);
```

Наш приклад буде виглдядати таким чином:

06 42 12 55 94 18 44 67

12 42 18 55 94 67 44 | 06

18 42 44 55 94 67 | 12 06

42 55 44 67 94 | 18 12 06

44 55 94 67 | 42 18 12 06

55 67 94 | 44 42 18 12 06

67 94 | 55 44 42 18 12 06

94 | 67 55 44 42 18 12 06

Залишилося записати сам алгоритм.

```
L,R: integer; x: real;
       a: array[1..100] of real;
  procedure sift(L,R: integer);
    L:=(n div 2)+1; R:=n;
    while R>1 do
      begin
        L:=L-1;
        while L>1 do
          begin
            L:=L-1;
            sift(L,R)
          end;
          x:=a[1]; a[1]:=a[R]; a[R]:=x;
          R:=R-1;
      end;
  end;
```

Сортування за допомогою поділу (швидке сортування)

```
procedure MergeSort;
i,j,k,l: integer;
        up: boolean; p: integer;
   begin
     up:=true; p:=1;
     repeat
       {визначення індексів}
       if up
```

```
        then
          begin
            i:=1; j:=n;k:=n+1; l:=2*n
          end
        else
          begin
            k:=1; l:=n; i:=n+1; j:=2*n
          end;
      {злиття p-наборів з i та j-входів у k та l-виходи}
        up:=not (up);p:=2*p;
      until p=n;
```

Запишемо уточнений алгоритм, який працює для будь-якого n, ввівши ще такі змінні:

m — кількість елементів, які ще залишилось обробити;

q та r — довжини підпослідовностей, що зливаються.

Оскільки n не завжди є степенню 2, то умовою виходу буде p>=n. З цієї причини підпослідовності з q та r елементів не завжди зійдуться в центрі. Тому залшки копіюються в хвіст вихідної послідовності.

```
procedure StraightMerge;
  var i,j,k,l,t: integer;
      h,m,p,q,r: integer;
      up: boolean;
  begin
    up:=true;
    repeat
      h:=1; m:=m;
      if up
        then
```

```
      begin
        i:=1; j:=n; k:=n+1; l:=2*n
      end
    else
      begin
        k:=1; l:=n; i:n+1; j:=2*n
      end;
repeat {злиття серій з i та j-входів у k-вихід}
    if m>=p then q:=p else q:=m;
    m:=m-q;
    if m>=r then r:=p else r:=m;
    m:=m-r;
    while (q<>0) and (r<>0) do {злиття q та r елементів}
          if a[i]<a[j]
            then
              begin
                  a[k]:=a[i]; k:=k+h;
                  i:=i+1; q:=q-1
               end
            else
              begin
                 a[k]:=a[j]; k:=k+h;
                 j:=j-1; r:=r-1
              end;
    while r>0 do
        begin
          a[k]:=a[j]; k:=k+h;
          j:=j-1; r:=r-1
```

```
            end ;
          while q>0 do
            begin
              a[k]:=a[i]; k:=k+h;
              i:=i+1; q:=q-1
            end ;
          h:=-h; t:=k; k:=l; l:=t; {переключення на кінець вихідного файла }
        until m=0;                {для рівномірного розподілу}
        up:=not up; p:=2*p;
      until p>=n;
      if not up then
        for i:=1 to n do
          a[i]:=a[i+n];
    end;
```

Швидке сортування – це алгоритм сортування, час роботи якого для вхідного масиву з *n* чисел в найгіршому випадку дорівнює *Θ(n2)*. Не дивлячись на таку повільну роботу в найгіршому випадку, цей алгоритм на практиці часто виявляється оптимальним завдяки тому, що в середньому час його роботи набагато кращий: Θ(*nlogn*). Окрім того, сталі множники, які не враховуються у виразі Θ(*nlogn*), достатньо малі за величиною (наприклад, у порівнянні з алгоритмом сортування методом злиття). Алгоритм також має суттєву перевагу в тому, що працює без використання додаткової пам'яті (на противагу, наприклад, тому самому алгоритму сортування методом злиття).

Швидке сортування, аналогічно до сортування злиттям, засноване на парадигмі «розділяй та володарюй». Нижче описаний процес сортування підмасиву *A[p...r],* який складається з трьох етапів, як і всі алгоритми цієї парадигми.

- *Розділення:* Масив *A[p…r]* розбивається на два (можливо порожніх) підмасиви *A*[*p...q-1*] та *A[q-1...r]* шляхом переупорядкування його елементів. Кожний елемент масиву *A*[*p...q-1*] не є більшим за елемент *A*[*q*], а кожний елемент підмасиву *A[q-1...r]* є більшим елементу *A*[*q*]. Індекс *ц* обраховується під час процедури розбиття.
- *Рекурсивний розв'язок:* Підмасиви *A[p...q*-1] та *A[q-1...r]* відсортовуються шляхом рекурсивного виклику процедури швидкого сортування.
- *Комбінування:* Оскільки підмасиви відсортовуються на місці без застосування додаткової пам'яті, для їх об'єднання не потрібні жодні додаткові дії: увесь масив *A[p... r]* виявляється відсортованим.

Алгоритм швидкого сортування представлений наступною процедурою.

В процесі роботи масив *A[p…r]* складається з чотирьох частин (деякі з них можуть бути порожніми) (рис. 1), які утворюють інваріант циклу:

1. Якщо $p \le k \le i$, то $A[k] \le x$;
2. якщо $i+1 \le к \le j-1$, то $A[к] > x$;
3. якщо $k=r$, то $A[к] = x$;
4. якщо $j \le k < r-1$, то елементи можуть мати довільні значення.

Тепер покажемо, що вказаний вище інваріант циклу справедливий перед початком циклу, перед кожною операцією та вкінці циклу.

1. *Ініціалізація.* Перед першою ітерацією циклу for $i = p-1$ та $j = p$. Між елементами з індексами *p* та *i* немає жодних елементів, як їх немає між елементами з індексами *i+1* та *j-1*, тому перші дві умови інваріанту виконуються. Виконання алгоритму призводить до виконання і третьої умови (четверта умова виконується за визначенням).
2. *Збереження.* Необхідно розглянути два випадки, вибір кожного з яких визначається перевіркою у алгоритмі.

3. *Завершення.* По завершенню роботи алгоритму $j = r$. Тому кожний елемент масиву є членом однієї з трьох множин (четверта множина стає порожньою).

Таким чином, всі елементи масиву розбиті на три підмножини: величина яких не більше *x,* більша за *x,* та одноелементна множина, яка складається з *x*

Час роботи алгоритму швидкого сортування залежить від степеня збалансування, яким характеризується розбиття. Збалансування, в свою чергу, залежить від того, який елемент був обраний в якості опорного. Якщо розбиття збалансоване, асимптотично алгоритм працює так само швидко, як сортування злиттям. В протилежному випадку асимптотична поведінка цього алгоритму така сама повільна, як й в алгоритму сортування включенням.

Найгірша поведінка алгоритму швидкого сортування має місце в тому випадку, коли процедура розбиття породжує одну підзадачу зn-1 елементами, а другу - з 0 елементами. Припустимо, що таке незбалансоване розбиття виникає при кожному рекурсивному виклику. Для виконання розбиття необхідний час Θ(n). Оскільки рекурсивний виклик процедури розбиття, на вхід якої подається масив розміру 0, нічого не робить, то T(0) = =Θ(1). Отже, рекурентне співвідношення, яке описує час роботи цієї процедури, записується наступним чином:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n).$$

Інтуїтивно зрозуміло, що при підсумовуванні проміжків часу, який витрачається на кожний рівень рекурсії, отримується арифметична прогресія, що дає в результаті оцінку Θ(л2). Це легко показати за допомогою методу підстановки для наведеного вище рекурентного співвідношення.

Таким чином, якщо на кожному рівні рекурсії алгоритму розбиття максимально незбалансоване, то час роботи алгоритму дорівнює Θ(n2). Відповідно, ефективність такого методу не буде кращою за ефективність сортування методом включення. Більше того, за такий саме час алгоритм

швидкого сортування опрацьовує масив, який вже повністю відсортований, - ситуація, яка зустрічається часто і в якій час роботи алгоритму сортування включенням дорівнює Θ(n).

У найбільш сприятливому випадку процедура Partition розбиває задачу розміром *n* на дві підзадачі, розмір кожної з яких не перевищує *n/2*. В такому випадку швидке сортування працює набагато більш ефективно, і час її роботи описується наступним рекурентним співвідношенням:

$$T(n) = 2T(n/2) + \Theta(n).$$

Це рекурентне співвідношення підпадає під випадок 2 основного методу (власне, воно ідентичне до рекурентного співвідношення для сортування методом злиття), так що його розв'язок - $T(n) = \Theta(n\lg n)$. Отже, розбиття на рівні частини призводить до асимптотично більш швидкого алгоритму.

Тепер розглянемо випадок, коли розбиття далеке від збалансованого (але не таке погане, як у наведеному вище найгіршому випадку). Припустимо, наприклад, що розбиття проводиться у співвідношенні один до дев'яти. В цьому випадку для часу роботи алгоритму швидкого сортування отримується наступне рекурентне співвідношення:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) \text{ або } T(n) \le T(9n/10) + T(n/10) + cn.$$

Алгоритм швидкого сортування найкраще працює при збалансованому розбитті вхідного масиву на підмасиви. Для досягнути цього на практиці часто використовується підхід, коли в якості опорного елементу обирається не останній елемент вхідного масиву *A[p…r],* а випадково обраний елемент з цього масиву. Це дозволяє асимптотично наблизитись до збалансованого розбиття.

Нижче наводиться аналіз оцінки часу роботи алгоритму швидкого сортування над зафіксованим випадковим масивом *A*.

Час роботи процедури QuickSort визначається здебільшого часом роботи, який витрачається на виконання процедури Partition. При кожному виконанні

останньої відбувається вибір опорного елементу, який потім не приймає участі в жодному рекурсивному виклику процедур QuickSort та Partition. Таким чином, протягом всього часу виконання алгоритму швидкого сортування процедура Partition викликається не більше *n* разів. Робота цієї процедури здебільшого зосереджена у циклі for в рядках 3-6. В кожній ітерації циклу в рядку 4 опорний елемент порівнюється з іншими елементами масиву *A*. Тому, якщо відомо скільки разів виконувався рядок 4, то можна оцінити повний час, який витрачається на виконання циклу for в процесі роботи процедури QuickSort.

Позначимо через *X* - кількість порівнянь, які виконуються в рядку 4 процедури Partition протягом повної обробки n-елементного масиву процедурою QuickSort. Як зазначалось вище, процедура Partition викликається *n* разів, і при цьому виконується певний фіксований об'єм роботи. Далі певну кількість разів виконується цикл for, при кожній ітерації якого виконується рядок 4. Отже, час роботи процедури QuickSort становитиме *O(n + X)*.

Тож, щоб оцінити час роботи алгоритму швидкого сортування, необхідно обрахувати величину *X*. Для цього потрібно зрозуміти, в яких випадках в алгоритмі відбувається порівняння двох елементів, а в яких - ні. Для спрощення аналізу перейменуємо елементи масиву ***A*** як *z*1, *z2, …, zn,* де *n - i-й* найменший елемент масиву. Окрім того, визначимо множину *Zij = {z1,z1+1,...,z}*, яка містить елементи, що розташовані між елементами *zi* та *zj* включно.

В яких випадках в алгоритмі відбувається порівняння елементів *zi* та *zj?* Відмітимо, що порівняння кожної пари елементів відбувається не більше одного разу. Дійсно, всі елементи порівнюються з опорним елементом, який ніколи не використовується в двох різних викликах процедури Partition. Таким чином, після конкретного виклику цієї процедури, елемент, який використовується в якості опорного, більше не буде порівнюватись з іншими.

Визначимо випадкову величину *Xij,* яка дорівнює кількості порівнянь елементів *zi* та *zj.* З попередніх міркувань зрозуміло, що *Xij* може дорівнювати або 0, або 1. Тепер повна кількість порівнянь, які виконуються протягом роботи алгоритму, можна виразити наступним чином: $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$.

Застосувавши до обох частин цього виразу операцію обчислення математичного сподівання і використовуючи властивість лінійності математичного сподівання, отримаємо:

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P\{z_i \text{ порівнюється з } z_j\}.$$

Адже,

$$E[X_{ij}] = 0 \cdot P[X_{ij} = 0] + 1 \cdot P[X_{ij} = 1] = P[X_{ij} = 1] .$$

Таким чином, все звелось до визначення $P\{z_i \text{ порівнюється з } z_j\}$. При чому вважається, що опорні елементи обираються випадковим чином, незалежно один від одного.

Спробуємо тепер розібратись, коли два елементи не порівнюються один з одним. Розглянемо, наприклад, в якості вхідних даних масив, який складається з чисел від 1 до 10 у довільному порядку, і припустимо, що в якості першого опорного елементу обрано число 7. Тоді в результаті першого виклику процедури Partition всі числа розпадуться на дві множини: {1, 2, 3, 4, 5, 6} та {8, 9, 10}. При чому елемент 7 порівнюється з усіма іншими елементами. Зрозуміло, що жодне з чисел, які потрапили в першу множину (наприклад, 2), більше не буде порівнюватись з жодним елементом другої підмножини (наприклад, 9).

Оскільки передбачається, що значення всіх елементів різні, то при виборі *x* в якості опорного елементу далі не будуть порівнюватись жодні елементи *zi* та *zj* для яких *zi* <*k*< *zj*. З іншого боку, якщо в якості опорного елементу обраний елемент *zi*,

то він буде порівнюватись з кожним елементом множини *Zij*, окрім самого себе. Те саме стосується елементу *zj*. Таким чином, елементи *zi* та *zj* будуть порівнюватись тоді й тільки тоді, коли першим в ролі опорного в множині *2ц* обраний один з них.

Тепер обчислимо ймовірність цієї події. Перед тим як в множині *Zij* буде обраний опорний елемент, вся ця множина не є розділеною, і будь-який її елемент може бути обраний в якості опорного. Оскільки всього в множині *j-i+1* елементів, а опорні елементи обираються випадково та незалежно один від одного, то ймовірність того, що конкретний елемент буде обраний першим в якості опорного, дорівнює 1/(*j-i+1*). Таким чином, виконується наступне співвідношення:

$P\{z_i$ порівнюється з $z_j\}$ = P\{Першим опорним було обрано z_i або $z_j\}$ = P\{Першим опорним було обрано $z_i\}$ + P\{Першим опорним було обрано $z_j\}$ =

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

Із цих співвідношень отримаємо:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Цю суму можна оцінити, скориставшись заміною змінних (*k=j– i).*

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} =$$

Внутрішня сума є сумою $\sum_{k=1}^{n} \frac{1}{k}$ гармонічного $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$ ряду і яку можна оцінити як lg*n* + O(1).

Таким чином, вираз спрощується до:

$$E[X] =< 2\sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} = 2\sum_{i=1}^{n-1} \mathrm{O}(\lg n) = \mathrm{O}(n \lg n).$$

Отже, можна зробити висновок, що при використанні процедури RandomizePartition математичне сподівання роботи алгоритму швидкого

сортування (при різних елементах в масиві) становить $O(n\lg n)$. Використовуючи оцінку для найкращого випадку, приходимо до того, що математичне сподівання часу роботи алгоритму становить $\Theta(n\lg n)$.

Питання для самоконтролю

1. Сформулюйте задачу сортування.
2. Які є підходи до розв'язування задачі сортування.
3. Сформулюйте алгоритм сортування вибором. Продемонструйте на прикладі.
4. Сформулюйте алгоритм сортування вставкою. Продемонструйте на прикладі.
5. Сформулюйте алгоритм сортування злиттям. Продемонструйте на прикладі.
6. Від чого залежить ефективність алгоритму сортування?
7. У чому полягає ідея алгоритму швидкого сортування?
8. До якого із класів алгоитмів сортування відноситься цей алгоритм?
9. Від чого залежить вибір алгоритму сортування?
10. Від чого залежить час виконання алгоритму сортування?
11. Для яких наборів вхідних даних ефективним є застосування алгоритму

Список використаних джерел

1. Ільман В. М., Іванов О. П., Панік Л. О. Алгоритми, дані і структури : навч. посіб. Дніпро : Дніпропет. нац. ун-т залізн. трансп.ім. акад. В. Лазаряна, 2019. 134 с.
2. Алгоритми та структури даних: конспект лекцій. Частина 1. Структури даних / упоряд.: О. Д. Воробйов, Л. В. Глазунов. Одеса : ОНАЗ ім.О.С. Попова, 2017. 48 с.
3. Алгоритми та структури даних: конспект лекцій. Частина 2. Алгоритми пошуку,стиснення даних, внутріщнього та зовнішнього сортування, алгоритми на графах / упоряд.: О. Д. Воробйов, Л. В. Глазунов. Одеса : ОНАЗ ім. О.С. Попова, 2017. 52 с.
4. Богач І. В., Довгалець С. М., Дубовой В, М. Алгоритми розв’язання задач з програмування. Вінниця : ВНТУ, 2017. 119 с.
5. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. СПб. : Питер, 2017. 288 с.
6. Власій О. О. Алгоритми та структури даних: Лабораторний практикум. Івано-Франківськ : ДВНЗ «Прикарпатський національний університет імені Василя Стефаника», 2015. 68 с.
7. Дудзяний І. М. Програмування мовою С++. Частина 1 : Парадигма процедурного програмування : навчальний посібник. Львів : ЛНУ імені Івана Франка, 2013. 468 с.
8. Клакович Л. С., Левицька С. М., Костів О, В. Теорія алгоритмів. Львів : Видавничий центр ЛНУ імені Івана Франка, 2008. 138 с.
9. Кормен Т. Алгоритмы: вводный курс. Москва : ООО "И.Д. Вильяме", 2014. 208 с.
10. Коротєєва Т. О. Алгоритми та структри даних : навчальний посібник. Львів : Видавництво Львівської політехніки, 2014. 80 с.
11. Макконнелл Дж. Анализ алгоритмов. Вводный курс. Москва : Техносфера, 2002. 304 с.
12. Мелешко Є. В., Якименко М. С., Поліщук Л. І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. Кропивницький: Видавець Лисенко В. Ф., 2019. 156 с.
13. Махровська Н.А., Погромська Г. С. Алгоритми і структури даних: навчально-методичний посібник. Миколаїв : МНУ ім. В.О. Сухомлинського, 2019. 279 с.
14. Льовкін В. М. Методичні вказівки до виконання самостійної роботи студентів та розрахунково-графічних завдань з дисципліни “Алгоритмізація та програмування” для студентів спеціальності 122 “Комп’ютерні науки” (всіх форм навчання). Запоріжжя : ЗНТУ, 2017. 54 с.
15. Онищенко В. В., Коник Р. С. Алгоритми та структури даних. К : 2017 - 66 с.

16. Прийма С.М. Теорія алгоритмів: навчальний посібник. Мелітополь: ФОП Однорог Т. В., 2018. 116 с.
17. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 2 : навчальний посібник. Львів : Видавництво «Новий Світ-2000», 2020. 320 с.
18. Сергієнко А. М., Марченко О. І. Конспект лекцій по курсу "Алгоритми і структури даних" для напряму підготовки 123 Комп'ютерна інженерія. К : Національний технічний Університет України «Київський Політехнічний Інститут імені Ігоря Сікорського», 2017. 74 с.
19. Спірінцева О. В., Спірінцева О. В., Литвинов О. А., Герасимов В. В. Java-технології та мобільні пристрої. Алгоритми і структури даних: навчальний посібник. Дніпропетровськ : Вид-во ДНУ ім. О. Гончара, 2016. 140 с.
20. Марченко О. І. Структури даних та алгоритми – 1. Основи алгоритмізації: завдання до виконання лабораторних робіт з дисципліни «Структури даних та алгоритми» для студентів напрямку підготовки 6.050102«Комп'ютерна інженерія» [Електронне видання]. К : НТУУ «КПІ», 2013. 57 с.
21. Структури даних та алгоритми – 2. Складні структури даних та алгоритми: завдання до виконання лабораторних робіт з дисципліни «Структури даних та алгоритми» для студентів напрямку підготовки 6.050102 «Комп'ютерна інженерія» [Електронне видання]. К : НТУУ «КПІ», 2013. 106 с.
22. Ткачук В. М. Алгоритми та структура даних : навчальний посібник. Івано-Франківськ : Видавництво Прикарпатського національного університету імені Василя Стефаника, 2016. 286 с.
23. Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. C++. Алгоритмізація та програмування: підручник. Одеса : Фенікс, 2019. 477 с.
24. Хайнеман Дж., Пояяис Г., Сеяков С. Алгоритмы. Справочник с примерами на C, C++, Java и Python. СпБ : ООО "Альфа-книга", 2017. 432 с .
25. Шаховська Н. Б., Голощук Р. О. Алгоритми і структури даних. Навчальний посібник. Львів : Магнолія, 2018. 216 с.
26. Шевчук І.Б. Конспект лекцій з навчальної дисципліни "Алгоритмізація та програмування". Львів : Львівський національний університет ім. Івана Франка, 2018. 30с.

УДК 004.421:004.422.63(075.8)

Електронне мережне навчальне видання

Т. О. Гришанович

КУРС ЛЕКЦІЙ ІЗ ДИСЦИПЛІНИ “АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ”

для студентів спеціальності 014.09 Середня освіта (Інформатика)
першого (бакалаврського) рівня

Друкується в авторській редакції