

**ВІДОКРЕМЛЕНИЙ СТРУКТУРНИЙ ПІДРОЗДІЛ
«ФАХОВИЙ КОЛЕДЖ РАКЕТНО-КОСМІЧНОГО МАШИНОБУДУВАННЯ
ДНІПРОВСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ імені Олеся Гончара»**

Циклова комісія програмної інженерії

КОНСПЕКТ ЛЕКЦІЙ НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

спеціальність 121 Інженерія програмного забезпечення

освітня програма Розробка програмного забезпечення
(назва освітньої програми)

відділення Комп'ютерної та програмної інженерії
(назва відділення)

**Дніпро
2024 рік**

Зміст

1. Поняття структури даних. Класифікація структур даних.	2
2. Структура даних зв'язаний список. Реалізація однозв'язного списку даних.	9
3. Реалізація основних функцій обробки однозв'язного списку.	14
4. Структури даних черга та стек. Способи реалізації та основні функції обробки черги та стеку.	23
5. Поняття графу. Види графів. Реалізація графів.	32
6. Структура даних дерево. Види дерев. Реалізація дерев.	37
7. Бінарні дерева. Види обходу бінарних дерев	42
8. Поняття рекурсії. Реалізація рекурсії. Рекурсивні математичні функції.	49
9. Рекурсивний перебір з поверненнями.	56
10. Методи сортування: вибором, вставками, злиттям, обмінами, шейкерне сортування	66
11. Сортування Шелла. Швидке сортування Хоара.	72
12. Методи пошуку в лінійних структурах даних.	80
13. Бінарні дерева пошуку.	86
14. Побудова збалансованих бінарних дерев пошуку	91
15. Алгоритми пошуку словесної інформації	98
16. Алгоритми обходу графів: обходи в ширину.	103
17. Алгоритми обходу графів: обходи в глибину.	109
18. Алгоритмічна стратегія «жадібні» алгоритми.	113
19. Алгоритми евристики	119
20. Метод гілок та границь. Задача комівояжера.	125
21. Порівняльні оцінки алгоритмів.	131
22. Асимптотичний аналіз функцій. Трудомісткість алгоритмів і часові оцінки	139
23. Методи аналізу рекурсивних алгоритмів.	151

Лекція № 1

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Поняття структури даних. Класифікація структур даних.
Мета заняття	Сформувати у студентів поняття концепції структури даних, дати класифікації структур даних, систематизувати стандартні прості та складені структури даних.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій, опорний конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Поняття структури даних.
- 2 Класифікація структур даних.
 - 2.1 Рівні структур даних.
 - 2.2 Категорії класифікації структур даних.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

1 Поняття структури даних

Під програмним забезпеченням ЕОМ розуміють сукупність програм, призначених як для підтримки належного функціонування ЕОМ, так і для виконання нею корисних функціональних завдань деякої прикладної області. Коли використовують термін «програма», мають на увазі не лише послідовність операторів деякої мови програмування, але і набір різних інформаційних об'єктів, над якими виконують ті або інші дії оператори програми. Такі інформаційні об'єкти програми називають даними.

З точки зору користувача та (або) програміста дані – це представлення фактів і ідей у формалізованому виді для передачі або обробки їх за допомогою деякого процесу (алгоритму).

З точки зору процесора, незалежно від змісту й складності будь-які дані в пам'яті ЕОМ представлені послідовністю двійкових розрядів, або бітів, а їхніми значеннями є відповідні двійкові числа.

Дані, розглянуті у вигляді послідовності бітів, мають дуже просту організацію або, інакше кажучи, слабо структуровані. Для людини описувати й досліджувати скільки-небудь складні дані в термінах послідовності бітів досить незручно. Більші за розміром і змістовніші, чим біт, "будівельні блоки" для організації довільних даних ґрунтуються на понятті "структури даних".

Під **структурою даних** у загальному випадку розуміють множину елементів даних і множину зв'язків між ними. Таке визначення охоплює всі можливі підходи до структуризації даних, але в кожному конкретному завданні використовуються ті або інші його аспекти. Тому вводиться додаткова класифікація структур даних, що відповідає різним аспектам їхнього розгляду. Дамо загальну класифікацію структур даних по декількох ознаках (див. рисунок 1.1).

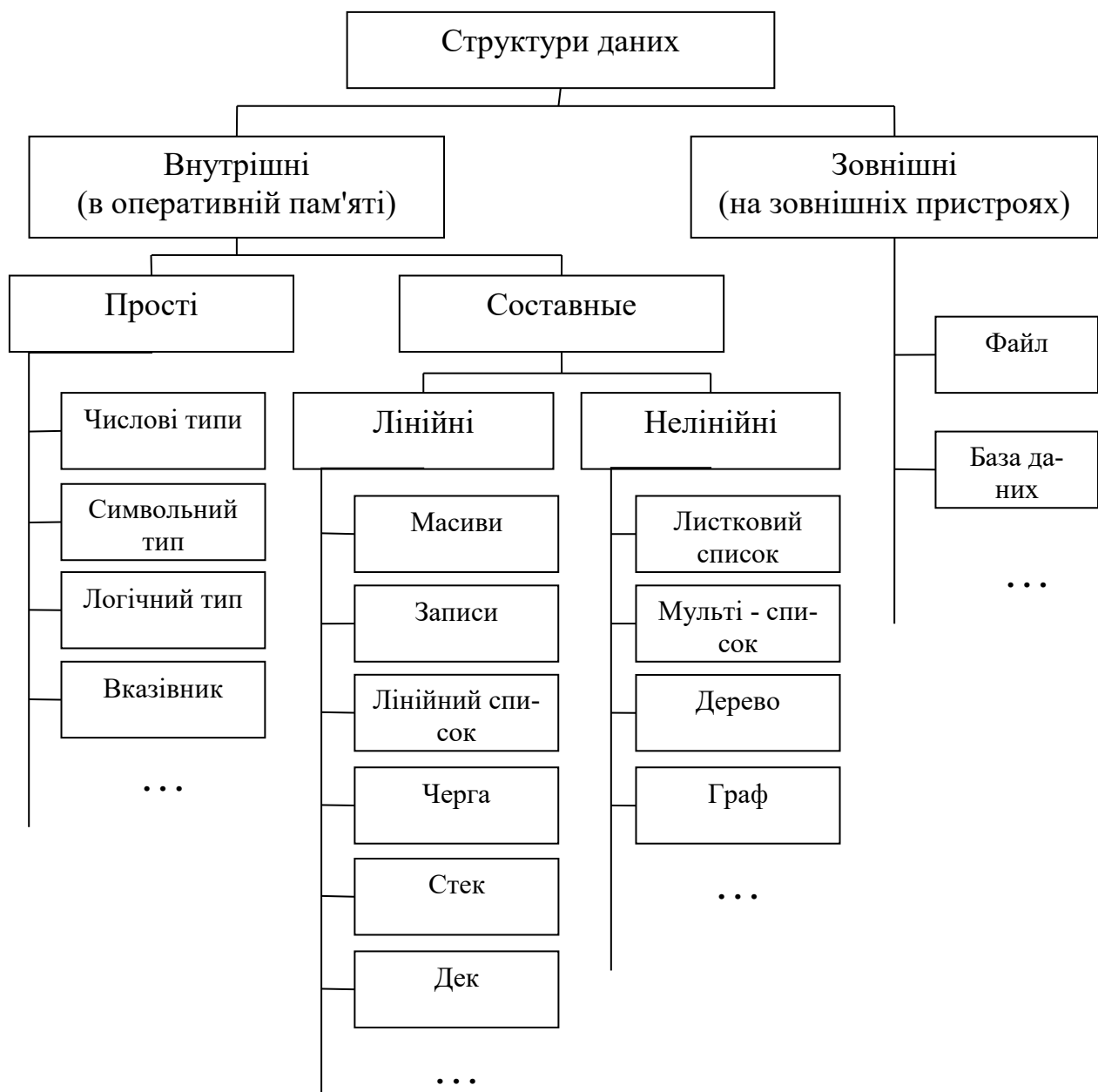


Рисунок 1.1. Загальна класифікація структур даних

Розрізняються прості (базові, примітивні) структури даних та складені (інтегровані, композитні, складні). *Простими* називаються такі структури даних, які не можуть бути розчленовані на складові частини, більше, ніж біти. З погляду фізичної структури важливим є та обставина, що в конкретній машинній архітектурі, у конкретної система програмування завжди можна заздалегідь сказати, яким буде розмір простого даного і яке його розміщення в пам'яті. З логічної точки зору прості дані є неподільними одиницями.

Складеними називаються такі структури даних, складовими частинами яких є інші структури даних - прості або у свою чергу складені. Складені структури даних конструюються програмістом з використанням засобів інтеграції даних, надаваних

мовами програмування. Важлива ознака складеної структури даних - характер упорядкованості її частин. За цією ознакою структури можна ділити на *лінійні й нелінійні*.

2 Класифікація структур даних

2.1 Рівні структур даних

Застосовуються три рівні структур даних :

1. змістовний
2. логічний
3. фізичний.

На змістовному рівні структур даних досліджуються конкретні об'єкти обробки, їх властивості і стосунки між об'єктами. На цьому рівні важливі не лише значення, але і сенс даних.

На логічному або абстрактному рівні структура даних вважається машинно-незалежним логічним поняттям, і виділяються наступні завдання: визначення множин даних як об'єктів дослідження, виділення складу множини даних, визначення структури даних по заданих вимогах, розробка кількісних методів оцінки ефективності різних видів структур даних.

Фізичний рівень (фізична структура). На цьому рівні розглядаються пам'ять ЕОМ і представлення в ній значень (осередки, розряди осередків, їх адреси і взаємне розташування значень.), тобто відображення даних в пам'яті ЕОМ. У загальному випадку між логічною і відповідною їй фізичною структурами існує відмінність, міра якої залежить від самої структури і особливостей того фізичного середовища, в якому вона має бути відбита.

2.2 Категорії класифікації структур даних

Структури даних можна класифікувати за декількома різними ознаками. Розглянемо два варіанти класифікації. По одному їх них структури даних розділяються на декілька категорій (до 7). Розглянемо ці категорії з прикладами, відповідними кожній категорії.

По рівню складності структури даних розділяються на

- прості - звичайні змінні або константи мов програмування стандартних типів, а також динамічні змінні цих же типів;
- набори однотипних даних - масиви, одно- і багатовимірні;

- інтегральні - записи і об'єкти класів і подібні структури;
- динамічні структури з внутрішніми зв'язками - списки, дерева, графи.

За способом створення структури даних можна розділити на

- звичайні -- змінні стандартних типів, звичайні (тобто не динамічні) масиви, записи і тому подібне;
- динамічні (створювані і руйновані за допомогою спеціальних операцій або процедур динамічного виділення і звільнення пам'яті) - динамічні масиви, динамічні змінні, зв'язні списки, дерева.

З точки зору архітектури можна виділити:

- лінійні структури -- одновимірні масиви, лінійні списки, лінійні черги, стеки;
- прямокутні структури - двовимірні і багатовимірні масиви;
- кільцеві структури - кільцеві списки, кільцеві черги, деякі реалізації графів;
- структури, що галузяться, - дерева різних видів, деякі реалізації графів;
- мережеві структури - графи.

Залежно від наявності або відсутності зв'язків розрізняють:

- незв'язні структури - вектори, масиви, рядки, стеки, черги;
- зв'язкові - списки, дерева, графи.

Залежно від постійності під час роботи програми розрізняють:

- статичні (що не змінюються) структури - змінні різних типів, записи, масиви, у тому числі динамічні;
- динамічні (що змінюються) - списки, дерева, черги, стеки, в загальному випадку графи. Тут слід звернути увагу на те, що термін "динамічні" вживається в різних сенсах, і структури, динамічні за способом створення, можуть бути статичними по своїй поведінці. Структури, динамічні по поведінці, як правило, виявляються динамічними і за способом створення.

Наступний критерій класифікації досить складно назвати одним або декількома словами, тому відразу приведемо його варіанти:

– дані, що створюються самим програмістом (незалежно від способу і часу створення), в англomовній літературі по програмуванню їх часто означають словом persistent - "стійкий" ("постійний", "постійно існуючий"). Таким є структури усіх даних, явно створених програмістом при написанні програми;

– створювані і руйновані безпосередньо під час роботи програми без участі програміста, зазвичай в допоміжних цілях, наприклад, при розбіжності типів даних в яких-небудь операціях. Для них використовується термін transient - "скороминущий", "тимчасовий". Їх дійсно можна коротко назвати тимчасовими, оскільки вони зазвичай знищуються відразу після їх використання. Як правило, такі дані, як об'єкти в програмі, є безіменними. У деяких мовах програмування, як наприклад, Сі++, такими даними можуть бути дані будь-яких типів. Про тимчасові структури часто говорять, що вони створюються не програмістом, а самим компілятором.

По місцю розміщення:

- існуючі в пам'яті ЕОМ (чи внутрішні) - ними можуть бути все раніше розглянуті приклади структур даних;
- що зберігаються на зовнішніх носіях (зовнішніх ЗУ) - файли і системи файлів.

Питання та завдання до контролю знань студентів

Питання для актуалізації опорних знань.

- 1 Дайте визначення поняття "тип даних".
- 2 Які прості типи даних ви знаєте?
- 3 Чим характеризуються прості типи даних?
- 4 До якого типу даних відноситься масив даних?
- 5 До якого типу даних відноситься запис?
- 6 Що таке файл даних?.
- 7 Які типи файлів ви знаєте?

Питання для узагальнення та перевірки засвоєного матеріалу на лекції

- 1 Сформулювати поняття даних.
 - 2 Сформулювати поняття структури даних.
 - 3 Які існують рівні структур даних?
 - 4 Що таке логічний рівень структур даних?
 - 5 Що таке фізичний рівень структур даних?
 - 6 Опишіть масив даних на логічному рівні та на фізичному рівні.
 - 7 Проведіть класифікацію даних за:
- місцем розміщення в пам'яті;
 - рівнем складності структури даних;

- за способом створення структури даних;
- з точки зору архітектури;
- залежно від постійності під час роботи програми.

Лекція № 2

з навчальної дисципліни «Алгоритми та структури даних»

Тема Структура даних зв'язаний список.
Реалізація однозв'язного списку даних.

Мета заняття Ознайомити студентів з поняттям динамічної структури даних
– однозв'язний список.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН
Конспект лекцій, опорний конспект лекцій.

Час – 2 години

План проведення лекції

1 Поняття динамічних структур даних

2 Однозв'язні списки даних

2.1 Структура однозв'язного списку

2.2 Реалізація однозв'язного списку.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Поняття динамічних структур даних

Існує великий клас завдань, які вимагають використання даних, які в процесі роботи програми змінюють не лише свої значення, але і свою структуру (наприклад, розмір). Такі дані називаються даними з динамічною структурою.

Динамічна структура характеризується такими рисами.

1. Непостійність розміру (числа елементів) структури в процесі її обробки. Число елементів може змінюватися від нуля до деякого значення, визначуваного специфікою завдання або доступним об'ємом машинної пам'яті.

2. Необхідність зберігання в оперативній пам'яті.

3. Відсутність фізичної суміжності елементів структури в пам'яті. Елементи динамічної структури можуть бути розкидані в пам'яті хаотичним чином. Наслідком цього є ускладнення процедури доступу до елементів динамічної структури в порівнянні із статичними або напівстатичними структурами.

Динамічна пам'ять - оперативна пам'ять персонального комп'ютера, що надається програмі при її роботі за виключенням сегменту стека даних і тіла програми. Її розмір визначається усією доступною пам'яттю комп'ютера. Часто використання динамічних структур даних - це єдина можливість обробки даних великої розмірності.

Пам'ять під динамічні структури виділяється під час виконання програми. Порядок роботи з динамічними структурами даних наступний:

- створити дані з динамічною структурою (відвести місце в динамічній пам'яті);
- працювати за допомогою покажчика;
- видалити дані (звільнити зайняте структурою місце).

З вище переліченого витікають наступні переваги динамічних змінних :

- дані, що розміщуються в динамічних змінних, можуть займати усе вільне місце оперативної пам'яті;
- після видалення динамічних змінних, що відпрацювали, динамічну пам'ять, що звільнилася, можна знову використовувати.

Самі динамічні величини не вимагають опису в програмі, оскільки під час компіляції пам'ять під них не виділяється. Під час компіляції пам'ять виділяється тільки під статичні величини. Робота з динамічними даними виконується за допомогою покажчиків. Покажчики - це статичні величини, тому при роботі з динамічними структурами даних тільки вони вимагають опису.

Слід виразно розуміти, що робота з динамічними даними уповільнює виконання програми, оскільки доступ до величини відбувається в два кроки: спочатку шукається покажчик, потім по ньому - величина. Як це часто буває, що діє "закон збереження прикростей" : виграш в пам'яті компенсується програшем в часі.

Оскільки елементи динамічної структури розташовуються по непередбачуваних адресах пам'яті, адреса елемента такої структури не може бути визначена з адреси початкового або попереднього елемента. Для встановлення зв'язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв'язки між елементами. Таке представлення даних в пам'яті називається зв'язковим.

Внаслідок цього динамічні структури часто фізично представляються у формі зв'язних списків.

2 Однозв'язні списки даних

2.1 Структура однозв'язного списку

Зв'язний список - така структура, елементами якої служать записи з одним і тим же форматом, пов'язані один з одним за допомогою покажчиків, що зберігаються в самих елементах. Таким чином, в зв'язаному списку елементи лінійно впорядковані, але порядок визначається не номерами, як в масиві, а покажчиками, що входять до складу елементів списку.

Простими зв'язними списками є лінійні зв'язні списки - однозв'язні і двозв'язні списки.

Розглянемо структуру однозв'язного списку. У однозв'язному списку кожен елемент або вузол списку це запис, що складається з двох різних за призначенням полів: поля інформації (чи поля даних) і поля покажчика (див. рис. 2.1).



Рисунок 2.1. Структура вузла списку даних.

У полі інформації зберігаються дані, заради яких і створюється список (наприклад, список дійсних чисел або список масивів). Це поле може бути будь якого допустимого типу, окрім файлового.

Поле покажчика зберігає адресу наступного елемента списку. За допомогою покажчика можна отримати доступ до наступного елемента списку, з наступного - до

чергового, і так далі, поки не буде досягнутий останній елемент списку. Поле показника останнього елементу списку повинне містити ознаку нульового, або порожнього показника, що свідчить про кінець списку.

Перш ніж рухатися по показниках, потрібно знати хоч би один елемент списку. Тому дуже важливою частиною логічної структури однозв'язного списку є показник на початок списку (показник на голову списку). Структура однозв'язного списку даних наведена на рисунку 2.2.

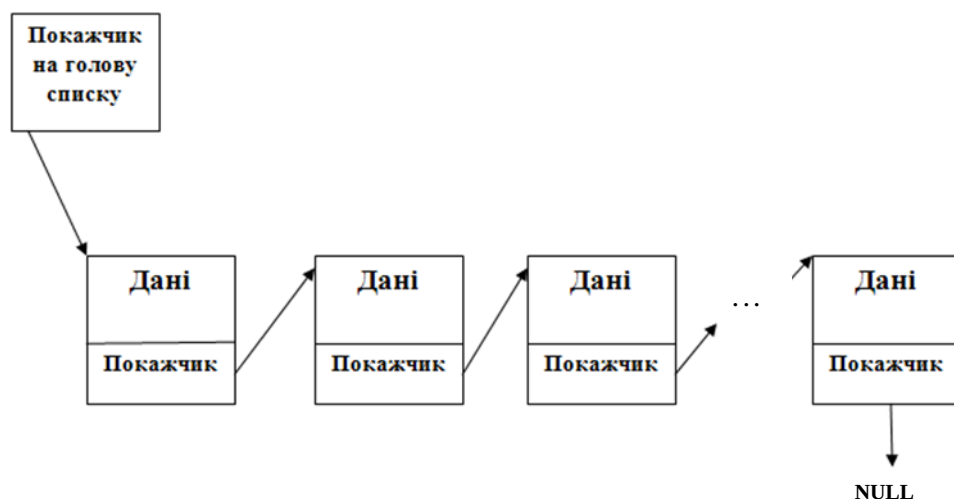


Рисунок 2.2. Структура однозв'язного списку даних

2.2 Реалізація однозв'язного списку. Операції над однозв'язним списком

Однозв'язний список даних, як сукупність зв'язаних вузлів, розміщується в динамічній пам'яті. Показник на початок списку (на перший елемент списку) повинен бути статичною змінною. Якщо показник на початок списку містить ознаку нульового, або порожнього показника, це свідчить про те, що список пустий.

Отже, для реалізації списку необхідно описати структуру вузла списку та сформувати показник на вузол списку (показник на структуру), який буде показником на перший елемент списку (на голову списку). Лістинг 2.1 містить фрагмент опису однозв'язного списку цілих чисел.

Лістинг 2.1.

```
// опис вузла списку
struct node
{
    int info;           // інформативна частина вузла
    node * next;       // показник на наступний вузол
};
```

```
void main()
{
    node * first=NULL; /* покажчик на голову списку зі стартовим значенням. Створено пустий список цілих чисел
    . . .
```

Питання та завдання до контролю знань студентів

Питання:

- 1 Що являють собою зв'язні списки?
- 2 До яких класифікаційних груп структур даних ставляться списки?
- 3 Які існують різновиду зв'язних списків?
- 4 У чому складається відмінність незв'язного списку від масиву?
- 5 У чому складається відмінність зв'язного списку від масиву?
- 6 У чому полягають недоліки однозв'язного списку?

Завдання:

Всі питання в даному пункті розглядаються для списку, елементами якого є наступні об'єкти:

```
struct zvn { int inf; zvn *nx;}
```

1. Як визначити об'єкт для списку, елементами якого є числа.
2. Як визначити об'єкт для списку, елементами якого є слова.
3. Як визначити об'єкт для списку, елементами якого є покажчики.
4. Визначено змінні: `zvn *fst = NULL`, `*en`, `*r`. Покажчики `fst`, `en` - відповідно адреси першого й останнього елемента списку; `r` - для значень адрес змінних типу `zvn`. Сформовано список з елементами типу `zvn`. Напишіть коди перебору елементів списку.

Лекція № 3

з навчальної дисципліни «Алгоритми та структури даних»

Тема Реалізація основних функцій обробки однозв'язного списку.

Мета заняття Ознайомити студентів з реалізацією основних функцій обробки однозв'язного списку.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

1 Операції над однозв'язним списком

1.1 Операції додавання елементу в список.

1.2 Операція перегляду вмісту списку

1.3 Операція очищення (видалення) списку

1.4 Операція видалення елементу з початку списку

1.5 Операція видалення елементу з довільного місця списку, відмінного від початку.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Операції над однозв'язним списком.

Виділимо типові операції над списками:

- додавання елементу в початок або в кінець списку;
- перегляд (друк) списку;
- додавання елементу в довільне місце списку, відмінне від початку (наприклад, після елементу, покажчик на який заданий);
- видалення елементу з початку списку;
- видалення елементу з довільного місця списку, відмінного від початку (наприклад, після елементу, покажчик на який заданий);
- перестановка елементів списку;
- злиття двох списків;
- очищення списку;

1.1 Операції додавання елементу в список.

Розглянемо операції додавання елементу в початок або в кінець списку. Ці операції є основою для алгоритму формування списку. Формується порожній список, і послідовно додаються елементи в його початок або в кінець.

На рисунку 1.1 схематично показано процес **додавання нового вузла в початок існуючого списку**, в якому: *first* – покажчик на голову списку а *work* – допоміжний покажчик на вузол, що буде вставлено в список.

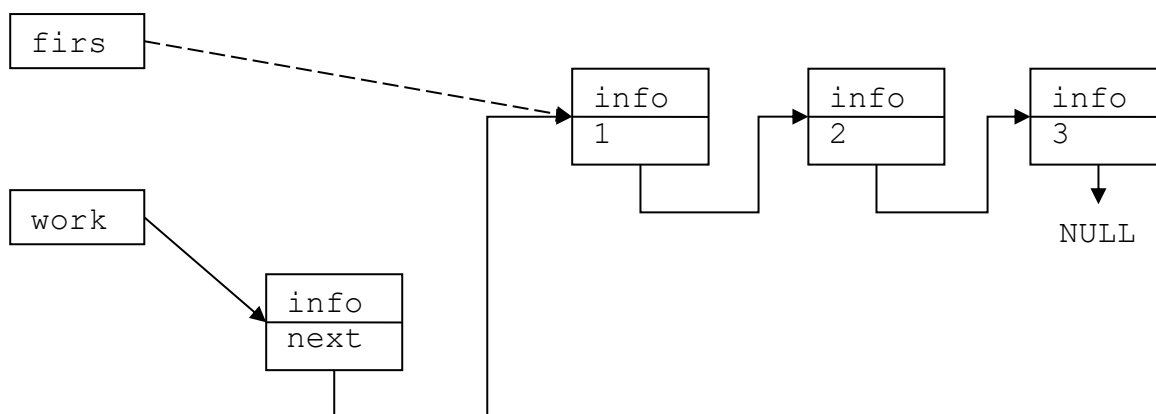


Рисунок 1.1. Формування списку додаванням нового елементу в початок списку.

В лістингу 1.1 представлено функцію формування списку додаванням нового елементу в початок списку.

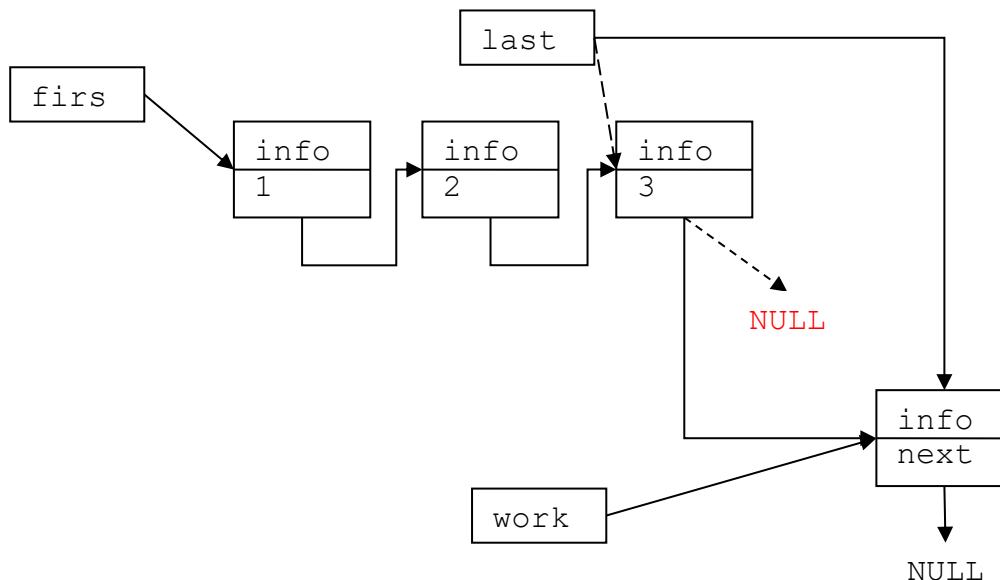
Лістинг 1.1.

```

node * creation_in_head (node * head)
{
    char c;
    node *work; // допоміжний покажчик
    c='y';
    while (c=='y')
    {
        cout<<endl<<"enter new element ";
        work=new node; // створюємо новий вузол
        work->next=head; // зв'язуємо його з початком списку
        head=work; // перенаправляємо початок списку на новий вузол
        cout<<endl<<"enter info ";
        cin>>head->info; // заповнюємо інформативну частину цілим числом
        cout<<"continue? y/n ";
        cin>>c;
    }
    return(head);
}

```

Додавання нового елемента в кінець існуючого списку відрізняється від попереднього прикладу тим, що, окрім first – покажчика на голову списку та work – допоміжного покажчика на вузол, що буде вставлено в список, необхідно мати ще один покажчик, назовемо його last, який завжди буде вказувати на останній елемент списку. На рисунку 1.2 схематично показано процес додавання нового вузла в кінець існуючого списку.



Ри-

суюнок 1.2. Формування списку додаванням нового елемента в кінець списку.

В лістингу 1.2 представлено функцію формування списку додаванням нового елемента в кінець списку.

Лістинг 1.2.

```

node * creation_in_tail (node * head)
{
    char c;

```

```

node *work, *last;
last=head;
if (head==NULL) // тобто список пустий
{
    // додаємо перший елемент в список
    work=new node;
    work->next=NULL;
    head=work; last=work;
    cout<<endl<<"enter first element ";
    cin>>head->info;
}
cout<<endl<<"continue? y/n ";
cin>>c;
while (c=='y')
{
    cout<<endl<<"enter new element ";
    work=new node;
    work->next=NULL;
    last->next=work; // last завжди фіксує останній елемент
    cout<<endl<<"enter info";
    cin>>work->info;
    last=work;
    cout<<endl<<"continue? y/n ";
    cin>>c;
}
return(head);
}

```

Вважається, що при виклику функції `creation_in_tail` список був пустим. Якщо нам потрібна функція додавання нового елементу в кінець вже існуючого непустого списку, то спочатку потрібно знайти останній елемент списку, зафіксувати його покажчиком `last`, і тільки потім вводити новий елемент. У лістингу 1.3 наведено функцію додавання нового елементу в кінець вже існуючого списку.

Лістинг 1.3.

```

node * enter_in_tail (node * head)
{
    node *last, *work;
    last=head;
    // переміщуємось до останнього елементу
    while (last->next != NULL) last=last->next;
    cout<<endl<<"enter new element ";
    work=new node; // створюємо новий вузол
    work->next=NULL;
    last->next=work; // прикріплюємо його до останнього елемента
    cout<<endl<<"enter info";
    cin>>work->info;
    last=work; //переміщуємо покажчик last на новий елемент
    return(head);
}

```

Операція додавання нового елементу в довільне місце списку, відмінне від початку.

На рисунку 1.3 схематично показано процес додавання нового вузла в довільне місце списку, відмінне від початку.

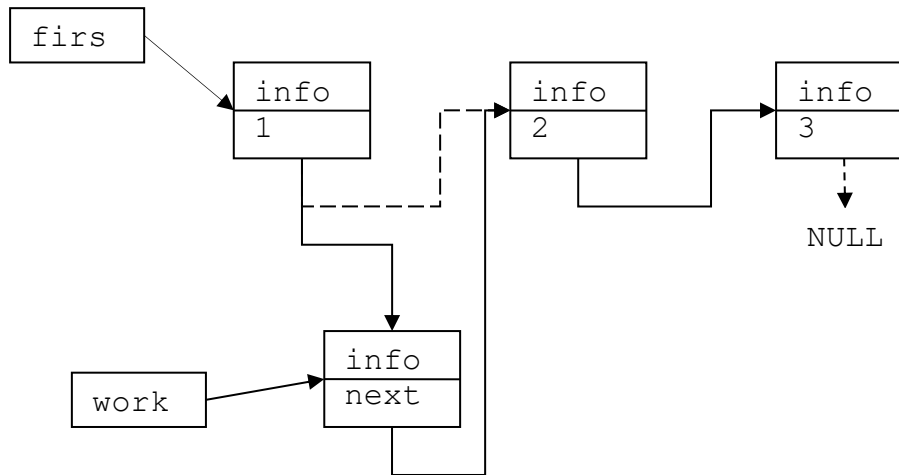


Рисунок 1.3. Додавання нового вузла в довільне місце списку, відмінне від початку.

Приклад.

Нехай створено список цілих чисел і в ньому знайдено максимальний елемент (можливо список містить декілька максимальних елементів). Розглянемо функцію, яка додає в список 1 після кожного максимального елемента(див. лістинг 1.4).

Лістинг 1.4.

```
node * enter_after_max (node *head, int el)
{
    node * work, * work1;
    work=head; // покажчик для переміщення по списку
    while (work!=NULL)
    {
        if (work->info==el) // якщо дивимось на потрібний вузол
        {
            work1=new node; // створюємо новий вузол
            work1->info=1;
            // прикріплюємо його до списку
            work1->next=work->next;
            work->next=work1;
        }
        work=work->next; // переміщуємось далі по списку
    }
    return head;
}
```

1.2 Операція перегляду вмісту списку

Операція перегляду вмісту списку здійснюється за допомогою додаткового покажчика (наприклад `work`), який переміщується від початку до кінця списку. Функція виведення вмісту списку на екран наведена в лістингу 1.5.

Лістинг 1.5.

```
void review_spisok (node * head)
{
    cout<<endl<<"spisok:"<<endl;
    node *work;
    // перевіряємо, чи не пустий список
    if (head==NULL) cout<<"spisok empty"<<endl;
    else
    {
        work=head;
        while (work!=NULL)
        {
            cout<<work->info<<endl;
            work=work->next; // переміщення по списку
        }
    }
}
```

1.3 Операція очищення (видалення) списку

Функція очищення (видалення) списку наведена в лістингу 1.6. Не забувайте використовувати цю функцію після закінчення роботи зі списком!

Лістинг 1.6.

```
node * moving_off_spisok (node * head)
{
    node *work;
    work=head;
    while (work!=NULL)
    {
        head=work->next;
        delete work;
        work=head;
    }
    head=work;
    return (head);
}
```

1.4 Операція видалення елемента з початку списку

На рисунку 1.4 схематично показано процес видалення елемента з початку списку.

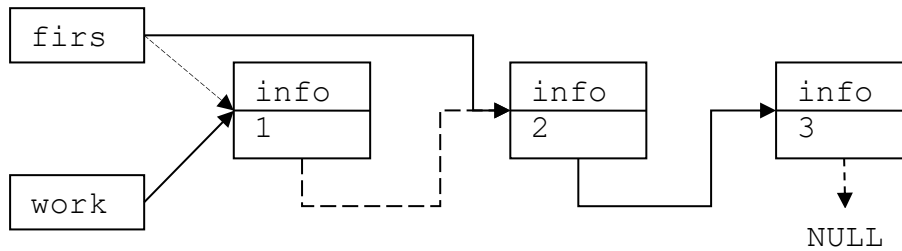
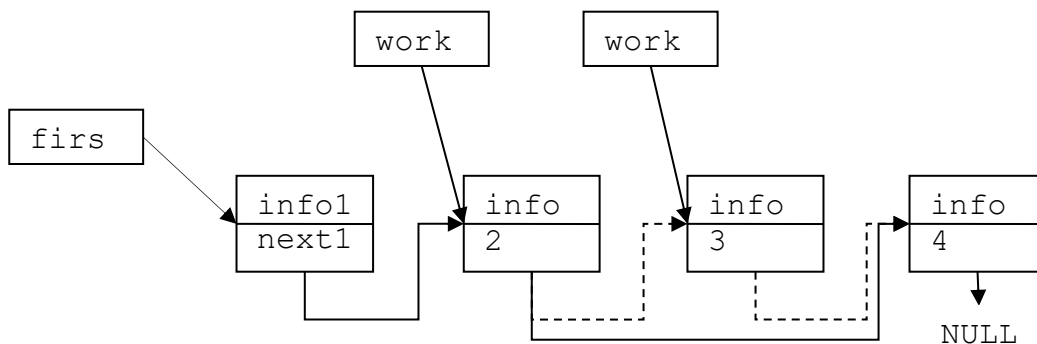


Рисунок 1.4.. Видалення елемента з початку списку.

1.5 Операція видалення елемента з довільного місця списку, відмінного від початку.

Для видалення вузла з середини списку потрібні два допоміжні покажчики. Покажчик `work` буде вказувати на попередній елемент до того, що потрібно видалити, а покажчик `work1` буде вказувати саме на той елемент, що потрібно видалити. На рисунку схематично показано процес видалення елемента з довільного місця списку, відмінного від початку.



Операції видалення елемента з початку списку та видалення елемента з середини списку розглянемо на наступному прикладі:

Нехай створено список цілих чисел і в ньому знайдено максимальний елемент (можливо список містить декілька максимальних елементів). Розглянемо функцію, яка видаляє всі максимальні елементи списку (див. лістинг 1.7).

Лістинг 1.7.

```
node * away_elements (node * head, int el)
{
    node * work, * work1;
    // видалення з початку списку
    while ((head!=NULL) && (head->info==el))
    {
        work=head;
        head=work->next;
    }
}
```

```

        delete work;
    }

    // видалення з середини списку
    work=head;
    while (work->next!=NULL)
    {
        if (work->next->info==el)
        {
            work1=work->next;
            work->next=work1->next;
            delete work1;
        }
        else work=work->next;
    }
    return (head);
}

```

Питання та завдання до контролю знань студентів

Всі питання в даному пункті розглядаються для списку, елементами якого є наступні об'єкти:

```
struct zvn { int inf; zvn *nx;}
```

- 1 Перелічіть інформацію, що необхідна, для того щоб видалити зі списку зі значенням рівним деякому числу k.
- 2 Написати фрагмент програми, що видаляє необхідний елемент зі списку.
- 3 Перелічіть інформацію, що необхідна, для того щоб у список додати елемент зі значенням рівним деякому числу k.
- 4 Написати фрагмент програми, що додає новий елемент у список. Новий елемент треба за деяким даним елементом.
- 5 Визначено змінні: `zvn *fst = NULL`, `*en = NULL`, `*r`. Сформовано список. Нехай в покажчику `en` перебуває адреса останнього елемента списку. Напишіть коди приєднання нового елемента типу `zvn` до списку.
- 6 У покажчику `r` перебуває адреса деякого елемента списку. Напишіть код висновку елемента списку, що є п'ятим щодо даного. Цикли не використати. Відповідь повинен складатися з однієї директиви.
- 7 Визначено змінні: `zvn *fst = NULL`, `*en`, `*r`. Поясніть код `r -> nx`.
- 8 Визначено змінні: `zvn *fst = NULL`, `*en`, `*r`. Поясніть код `r -> inf`.

- 9 Визначено змінні: $zvn *fst = NULL$, $*en$, $*r$. Поясніть код $r = r \rightarrow nx$.
- 10 Визначено змінні: $zvn *fst = NULL$, $*en$, $*r$. Поясніть код $r \rightarrow nx \rightarrow nx \rightarrow nx \rightarrow nx \rightarrow inf$.
- 11 Перелічите інформацію, що необхідна для того, щоб видалити зі списку головний (перший) елемент списку.
- 12 Перелічите інформацію, що необхідна для того, щоб новий елемент став головним (першим) елементом списку.
- 13 Написати фрагмент програми, що новий елемент списку робить головним.

Лекція № 4

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Структури даних «черга» та «стек». Способи реалізації та основні функції обробки «черги» та «стеку».
Мета заняття	Сформувати у студентів поняття структур даних черга та стек. Розглянути основні методи реалізації цих структур
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

1 Структура даних – «черга»

- 1.1 Реалізація «черги» на основі однозв'язного списку
- 1.2 Реалізація черги на основі лінійного масиву масиву

2 Структура даних – «стек»

- 2.1 Реалізація «стеку» на основі однозв'язного списку
- 2.2 Стековий калькулятор і зворотний польський запис формули.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Структура даних – «черга»

«Черга» - це абстрактна лінійна структура даних для зберігання однотипних елементів, що підтримує тільки дві операції:

1. Помістити елемент в кінець «черги»;
2. Обробити й витягти елемент із початку «черги».

«Чергу» можна представляти як трубу із двома кінцями, усередині якої послідовно розташовані елементи (див. рисунок 1.1). Класти новий елемент можна тільки а правий кінець труби, а брати - тільки з лівого. Той елемент, що був поміщений у чергу першим, буде першим же витягнутий із «черги». «Чергу» також називають контейнером типу *FIFO* (*first in - first out*) – "перший зайшов - перший вийшов".

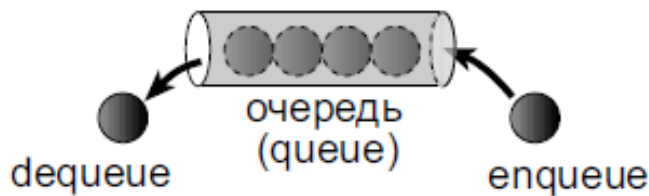


Рисунок 1.1. «Черга».

1.1 Реалізація «черги» на основі однозв'язного списку

Реалізувати «чергу» дуже зручно на основі однозв'язного списку. (див. рисунок 1.2). До цього списку допустимо застосовувати тільки дві операції:

- додавання нового елементу в кінець списку;
- видалення елементу з початку списку.

Отже, для роботи із «чергою» завжди необхідно мати дві змінні – вказівники: *first* - для збереження адреси першого елемента «черги» («голова «черги»») і *last* - для збереження адреси останнього елемента «черги» («хвіст «черги»»).

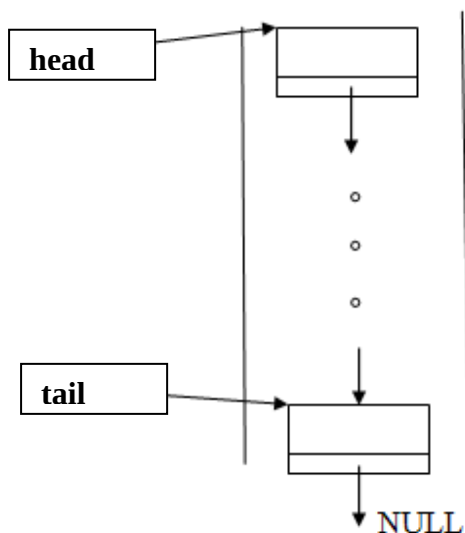


Рисунок 1.2. Реалізація «черги» на основі однозв'язного списку.

Перша операція – помістити елемент в кінець «черги».

Ця операція зводиться до операції додавання нового елементу в кінець списку. Але відмітимо, що випадок, коли `first==NULL` та `last==NULL`, є ознакою того, що черга не містить елементів, тобто є пустою. Цю ситуацію треба враховувати при додаванні елемента до черги і обробляти її окремо.

Розглянемо функцію `void addition (node ** head, node ** tail)`, яка має два формальні параметри: `node ** head` – в який ми будемо передавати вказівник на вказівник на «голову черги» та `node ** tail` – в який ми будемо передавати вказівник на вказівник на «хвіст черги».

```

void addition (node ** head, node ** tail)
{
    node *work;
    node * f=*head; //фіксуємо адресу початку списку
    node * l=*tail; //фіксуємо адресу кінця списку
    if ( f==NULL && l==NULL)
    {
        work=new node;
        work->next=NULL;
        f=work; l=work;
        cout<<endl<<"enter element ";
        cin>>work->info;
    }
    else
    {
        work=new node;
        work->next=NULL;
        l->next=work;
    }
}

```

```

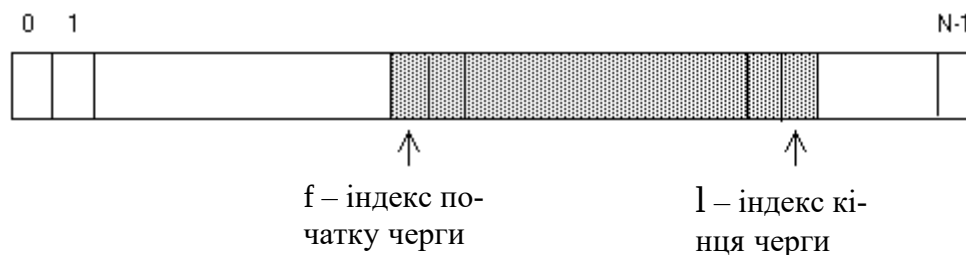
        cout<<endl<<"enter element  ";
        cin>>work->info;
        l=work;
    }
    *head=f; //повертаємо адресу початку списку
    *tail=l; //повертаємо адресу кінця списку
    return;
}

```

Друга операція – обробити й витягти елемент із початку «черги».
Ця операція зводиться до видалення елемента з початку списку.

1.2 Реалізація черги на базі масиву

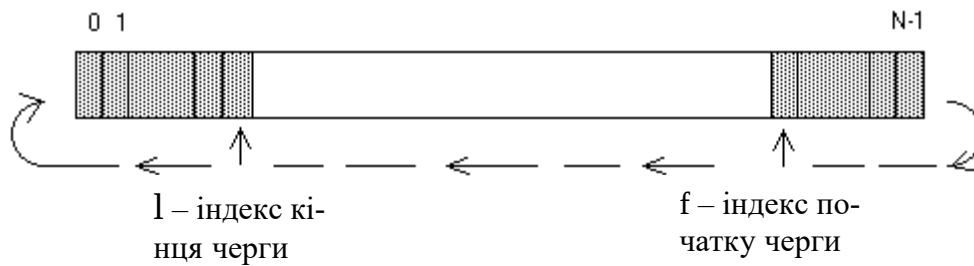
При реалізації черги на основі масиву фіксованої довжини N черга обмежена й не може містити більше N елементів. Індеси елементів масиву змінюються в межах від 0 до $N - 1$. Крім масиву, реалізація черги зберігає три прості змінні: індеси початку черги (f – стартове значення -1), індекс кінця черги (l – стартове значення теж -1), число елементів черги. Елементи черги втримуються у відрізку масиву від індексу початку до індексу кінця.



При додаванні нового елемента в кінець черги індекс l кінця спершу збільшується на одиницю (не забуваючи перевіряти, чи не вийшли за границю масиву), потім новий елемент записується в осередок масиву із цим індексом. Аналогічно, при витягу елемента з початку черги вміст осередку масиву з індексом f початку черги запам'ятовується як результат операції, потім індекс початку черги збільшується на одиницю.

Що відбувається, коли індекс кінця черги досягає кінця масиву, тобто $N - 1$?

Ключова ідея реалізації черги полягає в тому, що масив подумки як би зациклюється в кільце. Нагадаємо, що останній елемент має індекс $N - 1$, а перший - індекс 0. При зрушенні індексу кінця черги вправо у випадку, коли він указує на останній елемент масиву, він переходить на перший елемент. Таким чином, безперервний відрізок масиву, займаний елементами черги, може переходити через кінець масиву на його початок



2 Структура даних – «стек»

«Стек» - це абстрактна структура даних для зберігання однотипних елементів, що підтримує тільки дві операції:

1. Помістити елемент у стек;
2. Обробити та витягти (видалити) елемент зі стека.

"Стек" можна представити як стопку (див. рисунок 1.1) - класти можна тільки на верх стопки й брати теж тільки зверху. Той елемент, що покладений у стопку останнім, буде вилучений зі стопки першим.

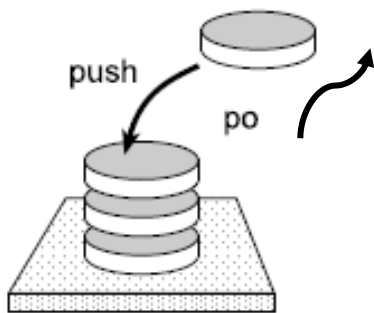


Рисунок 2.1. «Стек».

«Стек» також називають *контейнером типу LIFO (last in - first out)* – "останній зайшов - перший вийшов".

2.1 Реалізація «стеку» на основі однозв'язного списку

Реалізувати «стек» дуже зручно на основі однозв'язного списку. (див. рисунок 2.2). До цього списку допустимо застосовувати тільки дві операції:

- додавання нового елементу в початок списку;
- видалення елементу з початку списку.

Отже, для роботи із «стеком» завжди необхідно мати одну змінну – вказівник *first* - для збереження адреси першого елементу «стеку» («голова стеку»).

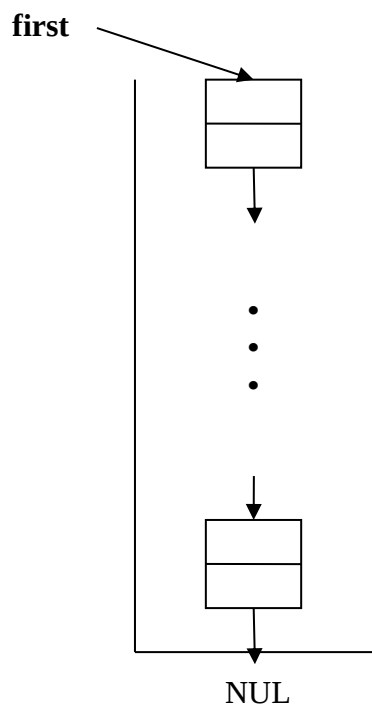


Рисунок 2.2. Реалізація «стеку» на основі однозв'язного списку.

2.2 Стековий калькулятор і зворотний польський запис формули.

В 1920 р. польський математик Ян Лукашевич запропонував спосіб запису арифметичних формул, що не використовує дужок. У звичній нам запису знак операції записується між аргументами, наприклад, сума чисел 2 й 3 записується як $2 + 3$. Ян Лукашевич запропонував дві інші форми запису: префіксна форма, у якій знак операції записується перед аргументами, і постфіксна форма, у якій знак операції записується після аргументів. У префіксній формі сума чисел 2 й 3 записується як $+ 2 3$, у постфіксній - як $2 3 +$. На честь Яна Лукашевича ці форми запису називають прямим й зворотним польським записом.

У польському записі дужки не потрібні. Наприклад, вираз

$$(2+3)*(15-7)$$

записується в прямому польському записі як

$$* + 2 3 - 15 7,$$

у зворотному польському записі - як

$$2 3 + 15 7 - *.$$

Якщо прямий польський запис не одержав великого поширення, то зворотна виявилася надзвичайно корисною. Неформальна перевага зворотного запису пе-

ред прямим польським записом або звичайним записом можна пояснити тим, що набагато зручніше виконувати деяку дію, коли об'єкти, над якими вона повинна бути зроблена, уже задані (відомі).

Зворотний польський запис формули дозволяє обчислювати вираження будь-якої складності, використовуючи стек як запам'ятовувальний пристрій для зберігання проміжних результатів. Такий стековий калькулятор був уперше випущений фірмою Hewlett Packard. Звичайні моделі калькуляторів не дозволяють обчислювати складні формули без використання паперу й ручки для запису проміжних результатів. У деяких моделях є дужки з одним або двома рівнями вкладеності, але більше складні вираження обчислювати неможливо. Також у звичайних калькуляторах важко зрозуміти, як результат й аргументи переміщуються в процесі введення й обчислення між регістрами калькулятора. Калькулятор звичайно має регістри X, Y і регістр пам'яті, проміжні результати якимсь образом переміщуються по регістрах, яким саме - запам'ятати неможливо.

На відміну від інших калькуляторів, пристрій стекового калькулятора цілком зрозуміло й легко запам'ятовується. Калькулятор має пам'ять у вигляді стека. При введенні числа воно просто додається в стек. При натисканні на клавішу операції, наприклад, на клавішу +, аргументи операції спочатку витягаються зі стека, потім з ними виконується операція, нарешті, результат операції міститься назад у стек. Таким чином, при виконанні операції із двома аргументами, наприклад, додавання, у стеці повинне бути не менш двох чисел. Аргументи видаляються зі стека й на їхнє місце поміщається результат, тобто, при виконанні додавання глибина стека зменшується на одиницю. Вершина стека завжди містить результат останньої операції й висвічується на дисплеї калькулятора.

Для обчислення виразу треба спочатку перетворити його у зворотний польський запис (при деякій навичці це легко зробити в розумі). У наведеному вище прикладі вираження $(2+3)*(15-7)$ перетвориться до

$2\ 3\ +\ 15\ 7\ -\ *$

Потім зворотний польський запис проглядається послідовно зліва на право. Якщо ми бачимо число, то просто вводимо його в калькулятор, тобто додаємо його в стек. Якщо ми бачимо знак операції, то з відповідну клавішу калькулятора, виконуючи в такий спосіб операцію із числами на вершині стека.

Зобразимо послідовні стани стека калькулятора при обчисленні по наведеній формулі. Скануємо ліворуч праворуч її зворотний польський запис:

$2\ 3\ +\ 15\ 7\ -\ *$

Стік спочатку порожній. Послідовно додаємо числа 2 й 3 у стек.

| | уводимо число 2 -> | 2 | уводимо число 3 -> | 3 |
| 2 |

Далі читаємо символ + і натискаємо на клавішу + калькулятора. Числа 2 й 3 витягаються зі стека, складаються, і результат міститься назад у стек.

| 3 | виконуємо додавання -> | 5 |
| 2 |

Далі, у стек додаються числа 15 й 7.

| 5 | уводимо число 15 -> | 15 | уводимо число 7 | 7 |
| 15 |
| 5 |

Читаємо символ - і натискаємо на клавішу - калькулятора. Зі стека при цьому знімаються два верхніх числа 7 й 15 і виконується операція вирахування. Причому зменшуваним є те число, що було уведено раніше, а від'ємником - число, уведене пізніше. Інакше кажучи, при виконанні некомутативних операцій, таких як вирахування або розподіл, правим аргументом є число на вершині стека, лівим - число, що перебуває під вершиною стека.

| 7 | виконуємо вирахування -> | 8 |
| 15 |
| 5 |

Нарешті, читаємо символ * і натискаємо на клавішу * калькулятора. Калькулятор виконує множення, зі стека знімаються два числа, перемножуються, результат міститься назад у стек.

| 8 | виконуємо множення -> | 40 |
| 5 |

Число 40 є результатом обчислення вираження. Воно перебуває на вершині стека й висвічується на дисплеї стекового калькулятора.

Питання та завдання до контролю знань студентів

- 1 Яка структура даних називається чергою?
- 2 Як структура даних «черга» відображається на пам'ять комп'ютера?
- 3 Який елемент називають «головою» черги?
- 4 Який елемент називають «хвостом» черги?
- 5 Як відбувається читання елементів черги? Запишіть алгоритм читання елемента з черги.
- 6 Як відбувається запис елементів у чергу? Запишіть алгоритм запису елемента в чергу.

- 7 Яким чином організовано ефективну обробку елементів черги?
- 8 Зобразіть схематично роботу з елементами черги.
- 9 Запишіть процедуру запису елемента в чергу.
- 10 Запишіть процедуру читання елемента з черги.
- 11 Запишіть процедуру перегляду елементів черги.
- 12 Який принцип доступу здійснюється до значень елементів черги при їх обробці? Обґрунтуйте свою відповідь.
- 13 Дайте означення структури даних «стек».
- 14 Як структура даних «стек» відображається на пам'ять комп'ютера?
- 15 Що називається вершиною стека?
- 16 Зобразіть схематично роботу з елементами стека.
- 17 Як відбувається операція запису елемента в стек?
- 18 Запишіть алгоритм запису елемента в стек.
- 19 Як відбувається операція читання елемента зі стека?
- 20 Запишіть алгоритм читання елемента зі стека.
- 21 Яким чином досягається швидкодія роботи зі стеком?
- 22 Запишіть процедуру запису в стек.
- 23 Запишіть процедуру читання зі стека.
- 24 Запишіть процедуру перегляду елементів стека.
- 25 Запишіть процедуру перегляду елементів масиву, на який відображається вміст стека.
- 26 Який принцип доступу здійснюється до значень елементів стека при їх обробці? Обґрунтуйте свою відповідь.

Лекція № 5

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Поняття графу. Види графів. Реалізація графів.
Мета заняття	Ввести поняття графа, як абстрактної структури даних. Показати методи реалізації графів.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Поняття графа
- 2 Абстрактне подання неорієнтованого графа
- 3 Реалізація графа

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Поняття графу

У математичної теорії графів і інформатики граф - це сукупність непорожньої множини вершин і множини пар вершин.

Об'єкти представляються як вершини або вузли графа, а зв'язки - як дуги або ребра. Для різних областей застосування види графів можуть відрізнятися спрямованістю, обмеженнями на кількість зв'язків і додатковими даними про вершини або ребрах.

Багато структур, які мають практичний інтерес в математики та інформатики, можуть бути представлені графами. Наприклад, будову Вікіпедії можна змоделювати за допомогою орієнтованого графа (орграф), в якому вершини - це статті, а дуги (орієнтовані ребра) – гіперпосилання.

Щоб дати визначення графа введемо кілька проміжних визначень.

Нехай задане деяка кінцева множина крапок (вершин). Будемо називати цю множину *вершинами графа* й позначати VG . Якщо вершини графа позначати натуральними числами, то $VG = \{1, 2, 3, \dots, n\}$.

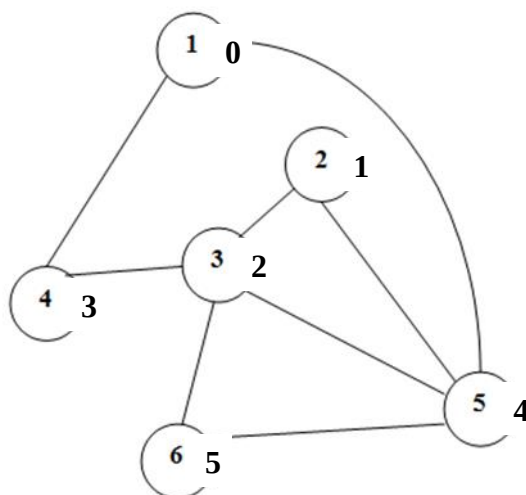
Зв'язані між собою вершини графа будемо позначати парою відповідних елементів з множини VG (наприклад, $(1,2)$ або $(4,6)$) і будемо називати ребрами графу. Позначимо всю множину ребер графа ім'ям RG . Наприклад, $RG = \{(1,2), (2,4), (2,6), (4,3)\}$

У такий спосіб: Граф G - це множина вершин VG і множина деяких пар вершин (ребер) RG .

Якщо всі зв'язані між собою вершини графа мають зв'язок як в одному, так й в іншому напрямку, то граф називається неорієнтованим. У цьому випадку зв'язані між собою вершини i й j графа задають у множині RG однією парою (i,j) .

2 Абстрактне подання неорієнтованого графа

На папері граф зручно представляти у вигляді малюнка, на якому кружком позначають вершину графа, а дуга, що з'єднує дві вершини графа позначає ребро графа. Нехай граф G представлений малюнком:



Граф G складається з шести вершин:

$VG = \{0, 1, 2, 3, 4, 5\}$

Та восьми ребер

$RG = \{(0, 3), (0, 4), (1, 2), (1, 4), (2, 3), (2, 4), (2, 5), (4, 0)\}$

Нехай G - неорієнтований граф. **Шляхом** в G називається така кінцева або нескінченна послідовність ребер і вершин $S = (\dots, a_0, E_0, a_1, E_1, \dots, E_{n-1}, a_n, \dots)$, що кожен два сусідніх ребра E_{i-1} та E_i мають загальну вершину a_i .

Таким чином, можна написати

$\dots, E_0 = (a_0, a_1), E_1 = (a_1, a_2), \dots, E_n = (a_n, a_{n+1}), \dots$

Відзначимо, що те саме ребро може зустрічатися в шляху кілька разів. Якщо немає ребер, що передують E_0 , то a_0 називається початковою вершиною S , а якщо немає ребер, що слідує за E_{n-1} , то a_n називається кінцевою вершиною S . Будь-яка вершина, що належить двом сусіднім ребрам, називається внутрішньою. Тому що ребра й вершини в шляху можуть повторюватися, внутрішня вершина може виявитися початковою або кінцевою вершиною.

Якщо початкова й кінцева вершини збігаються, шлях називається **циклічним**. Шлях називається **ланцюгом**, а циклічний шлях - **циклом**, якщо кожне його ребро зустрічається не більше одного разу (вершини можуть повторюватися). Нециклічний ланцюг називається **простим ланцюгом**, якщо в ній ніяка вершина не повторюється. Цикл із кінцем a_0 називається **простим циклом**, якщо a_0 не є в ньому проміжною вершиною й ніяк інш вершини не повторюються.

3 Реалізація графа

Існує кілька способів реалізації графа:

- матрицею суміжності;
- списком суміжності.

Розглянемо реалізацію графа G за допомогою матриці суміжності.

У даному графі визначено 6 вершин і тому для подання графа буде потрібно матриця розміром 6×6 (назвемо її MG). В комірках матриці $MG[i][j]$ та $MG[j][i]$ будемо ставити 1, якщо вершини графа i та j з'єднані ребром, у противному випадку заповнюємо комірки нулями.

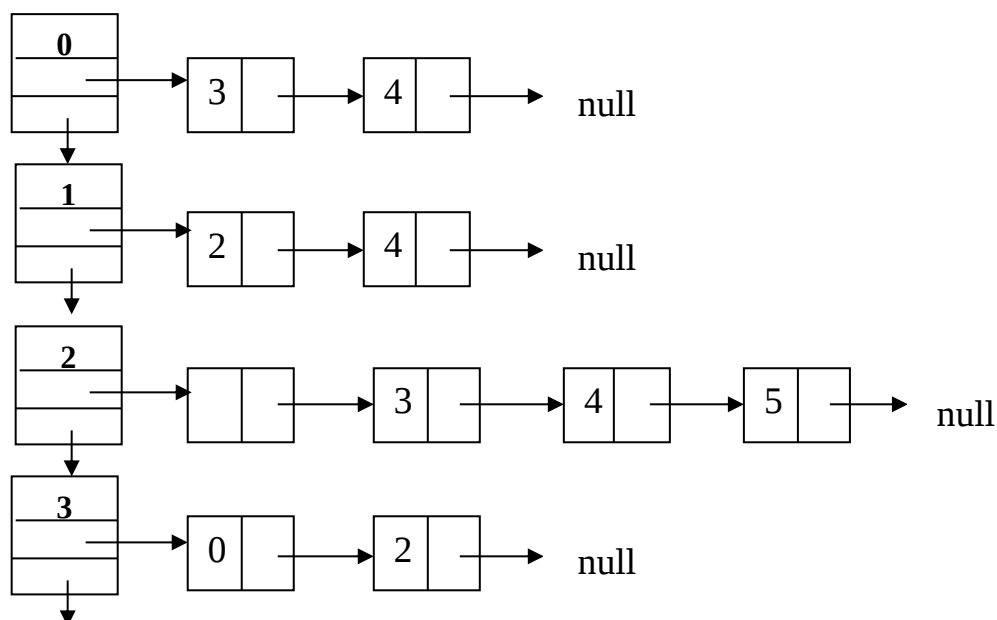
	0	1	2	3	4	5
0	0	0	0	1	1	0
1	0	0	1	0	1	0
MG: 2	0	1	0	1	1	1
3	1	0	1	0	0	0
4	1	1	1	0	0	1
5	0	0	1	0	1	0

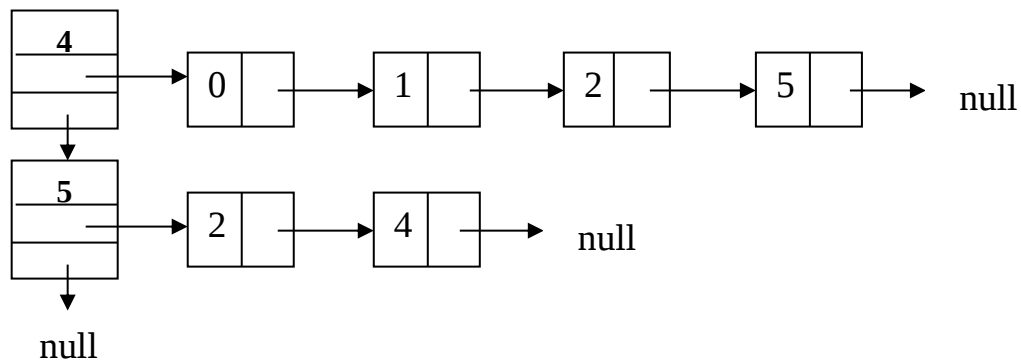
Матриця суміжності неорієнтованого графа є симетричною матрицею, на головній діагоналі якої розміщені нулі.

Завдання графа за допомогою матриці суміжності є дуже зручним: щоб довідатися, чи з'єднані вершини i й j ребром, досить тільки перевірити, чому дорівнює елемент $MG[i][j]$. Недоліком такого подання є те, що для зберігання графа з N вершин потрібно N^2 комірок пам'яті.

Цей недолік виправляється при завданні графа списком суміжності. При такому завданні вершині i графа ставиться у відповідність список пов'язаних з нею вершин, тобто тих, які в множині RG перебувають у парі з i .

Для нашого графа G список суміжності буде мати вигляд:





Питання та завдання до контролю знань студентів

- 1 Дати визначення графа
- 2 Що називається шляхом в графі?
- 3 Навести приклад абстрактного представлення графа.
- 4 Які способи реалізації графів ви знаєте?
- 5 Які особливості матриці суміжності для неорієнтованих графів?
- 6 Яка структура вузлів мультисписку, що використовується для реалізації графів?

Лекція № 6

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Структура даних дерево. Види дерев. Реалізація дерев.
Мета заняття	Ввести поняття дерева, як абстрактної структури даних. Показати методи реалізації графів.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Дерева. Загальні поняття
- 2 Реалізація дерев

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник. [Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Дерева. Загальні поняття

Дерево є одним з важливих і цікавих окремих випадків графа, тому воно розглядається окремо від графів інших видів. На рисунку 1.1 показано приклад багатогілкового дерева:

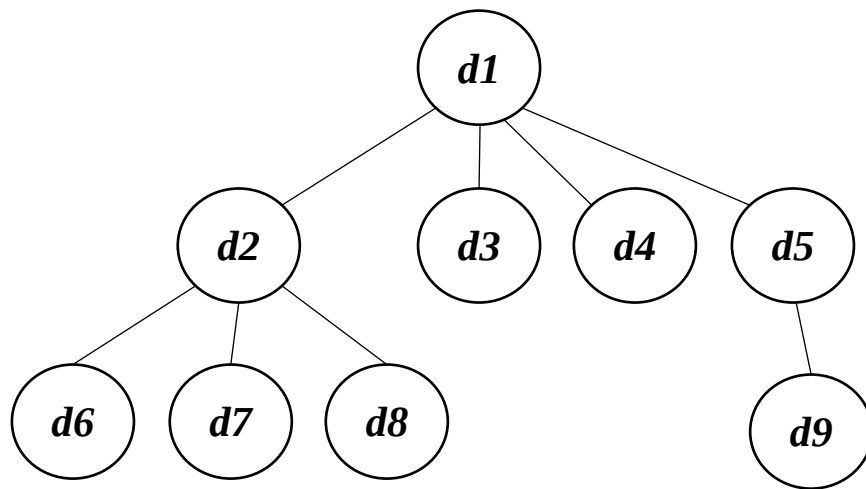


Рис.1.1 Багатогілкове дерево.

Деревом називається граф, для якого:

- 1) існує вузол, у який не входить ні одна дуга (корінь);
- 2) у кожен вузол, крім кореня, входить тільки одна дуга.

Вершини дерева підрозділяють на наступні три групи:

- корінь - вершина, у яку не входить ні одна дуга;
- вузли - вершини, у які входить одна дуга й виходить одна або більше дуг;
- листи - вершини, у які входить одна дуга й не виходить ні однієї дуги.

Всі вершини, у які входять дуги, що виходять із однієї вершини, називаються **нащадками** цієї вершини, а сама вершина - **предком**. Корінь не має предку, а листи не мають нащадків.

Виділяють рівні дерева. На першому рівні дерева може бути тільки одна вершина - корінь, на другому - нащадки кореня, на третьому - нащадки нащадків кореня й т.д. Піддеревом називається вершина з усіма її нащадками.

Висотою піддерева будемо вважати максимальну довжину ланцюга $u_1, \dots, u_n$ його вершин таку, що u_{i+1} - нащадок u_i для всіх i . Висота порожнього дерева дорівнює нулю, висота дерева з одного кореня - одиниці.

Ступенем вершини в дереві називається кількість дуг, що з неї виходить. Ступінь дерева дорівнює максимальному ступеню вершин. При цьому листами в дереві є

вершини, що мають ступінь нуль. По величині ступеня дерева розрізняють на два типи дерев:

- двійкові - ступінь дерева не більше двох;
- багатогілкові - ступінь дерева довільна.

2 Реалізація дерев

Дерево довільного ступеня (багатогілкове дерево) можна реалізувати як будь-який орієнтований граф (див. рисунок 2.1), а, отже, задати відповідною матрицею суміжності.

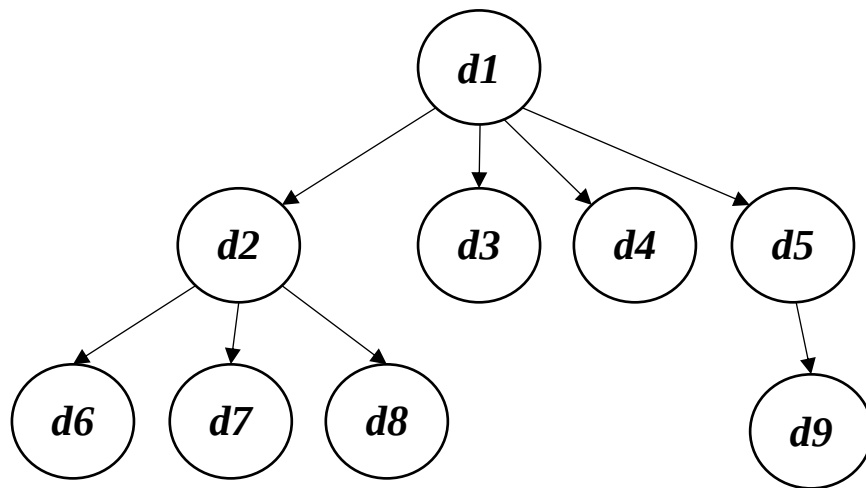


Рис.2.1. Багатогілкове дерево я орієнтований граф

Для графа, представленого на рис.2, матриця суміжності буде мати вид:

	d1	d2	d3	d4	d5	d6	d7	d8	d9
d1	0	1	1	1	1	0	0	0	0
d2	0	0	0	0	0	1	1	1	0
MD: d3	0	0	0	0	0	0	0	0	0
d4	0	0	0	0	0	0	0	0	0
d5	0	0	0	0	0	0	0	0	1
d6	0	0	0	0	0	0	0	0	0
d7	0	0	0	0	0	0	0	0	0
d8	0	0	0	0	0	0	0	0	0
d9	0	0	0	0	0	0	0	0	0

Однак, з огляду на специфіку дерева, як графа, можна розглянути окремий спосіб реалізації - як динамічну структуру у вигляді списку (рисунок 3). Облікове подання дерев довільного ступеня основане на елементах, що відповідають верши-

нам дерева. Кожен елемент (вершина) має поле даних і два поля покажчиків: показчик на початок списку нащадків показчик на наступний елемент у списку нащадків поточного рівня.

// опис вершини дерева

```
struct node
{
    float info; // інформативна частина вузла дерева
    node * childs; // показчик на початок списку нащадків
    node * next; // показчик на наступний елемент в списку нащадків
};
```

При такому способі подання дерева обов'язково варто зберігати показчик на вершину, що є коренем дерева. На рисунку 2.2 представлена схема динамічної реалізації нашого дерева.

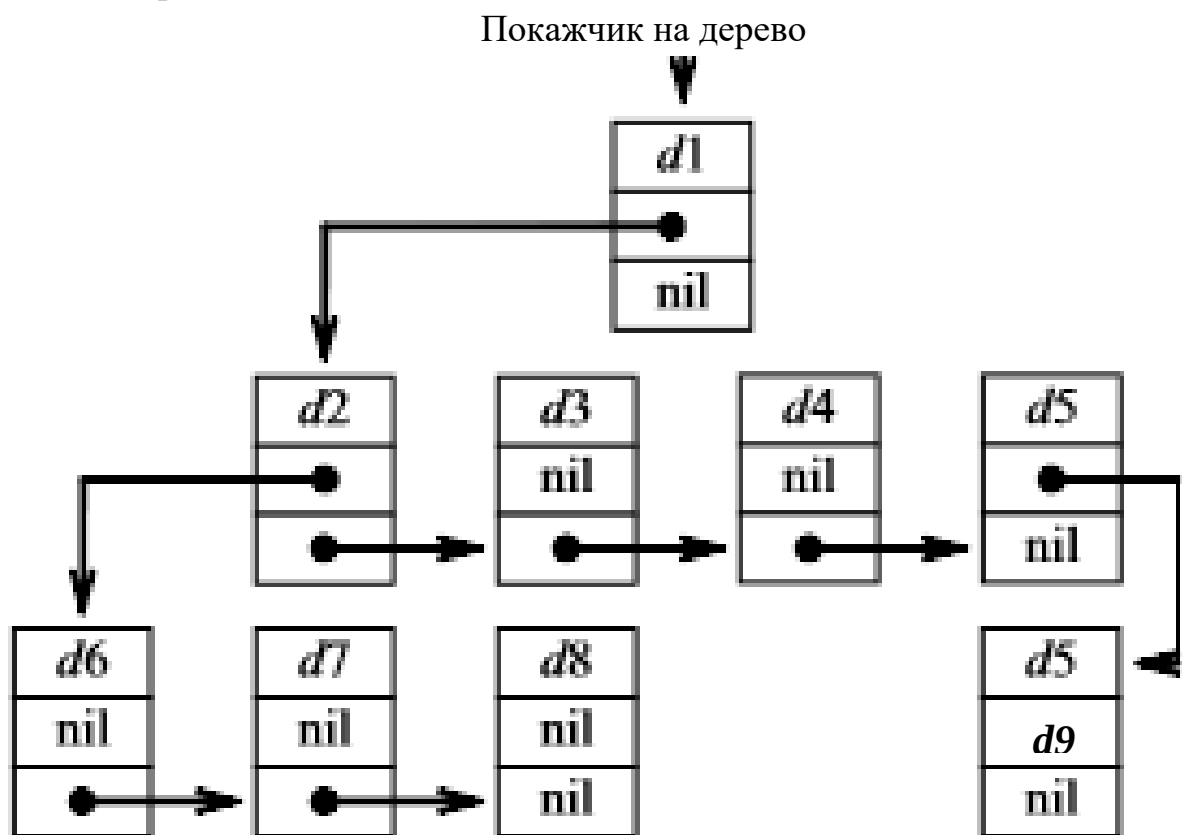


Рис. 2.2. Динамічна реалізація багатогілкового дерева.

Питання та завдання до контролю знань студентів

1. У чому складається основна відмінність деревоподібних структур від спискових?
2. Як рекурсивно визначається дерево?
3. Які типи вершин існують у деревах?
4. Які можна виділити типи дерев?

5. Які дерева називаються двійковими?
6. Які дерева називаються впорядкованими?
7. Які основні поняття зв'язуються з деревами?

Лекція № 7

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Бінарні дерева. Види обходу бінарних дерев
Мета заняття	Ввести поняття бінарного дерева. Пояснити метод реалізації бінарних дерев. Описати основні способи обходу бінарних дерев.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	
Конспект лекцій.	

Час – 2 години

План проведення лекції

- 1 Поняття бінарного дерева
- 2 Динамічна реалізація бінарного дерева
- 3 Основні операції над бінарними деревами:
 - 3.1 Створення бінарного дерева
 - 3.2 Способи обходу бінарних дерев
 - 3.3 Видалення бінарного дерева.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Поняття бінарного дерева

Дерево, у якому кожна вершина може мати не більше двох прямих нащадків, називається двійковим або бінарним деревом. Прямі нащадки кожної вершини називають лівим і правим нащадком (вузлом) відповідно (рисунок 1.1).

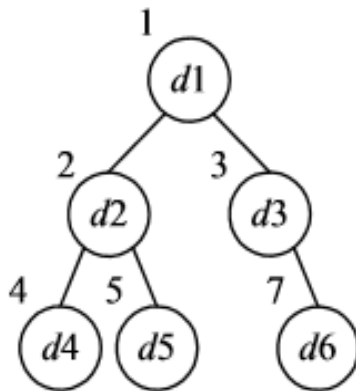
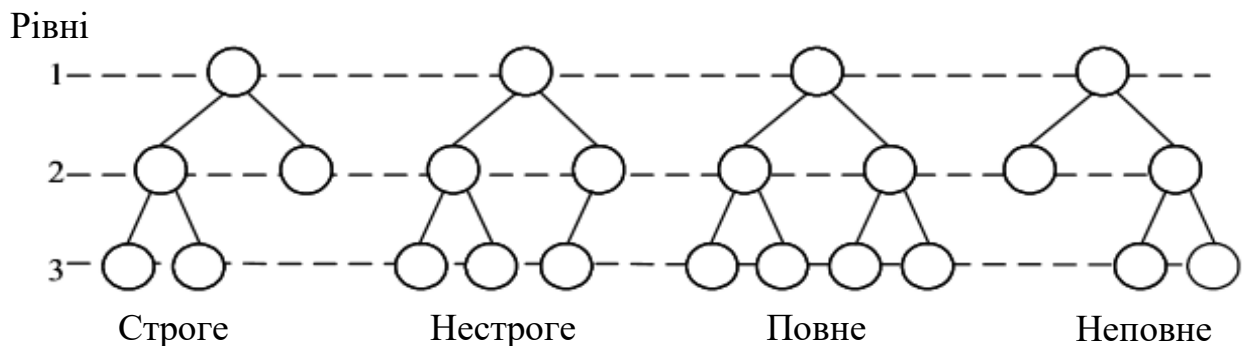


Рис.1.1. Приклад бінарного дерева

По ступені вершин бінарні дерева поділяються на повні, неповні, строгі та нестрогі:



2 Динамічна реалізація бінарного дерева

Реалізувати бінарне дерево можна за допомогою нелінійного списку, кожен вузол якого містить наступні поля: інформативна частина вузла, показчик на лівого прямого нащадку та показчик на правого прямого нащадку:

```

struct node
{
int info;
node * left;
node * right;
};

```

Бінарне дерево, представлене на рисунку 2.1, буде реалізоване наступним нелінійним списком (рисунок 2.2):

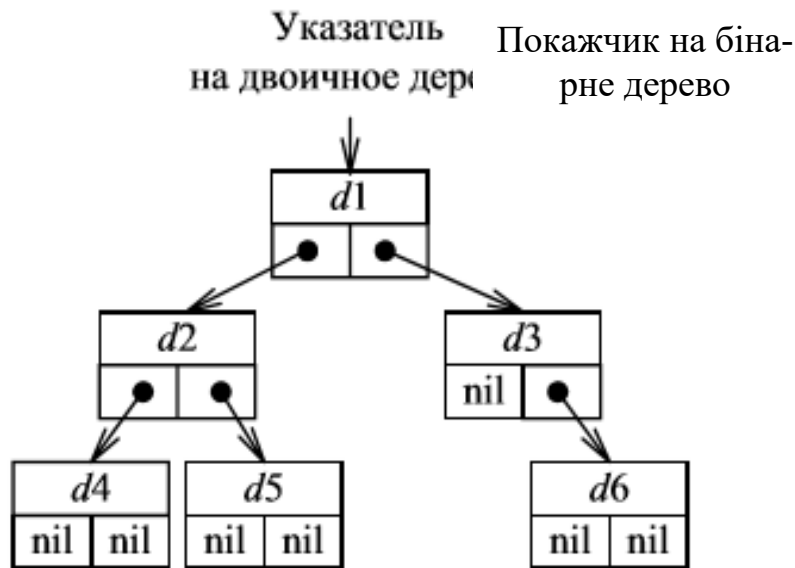


Рис.2.2. Схема реалізації бінарного дерева.

3 Основні операції над бінарними деревами:

3.1 Створення бінарного дерева

Рекурсивний варіант функції створення бінарного дерева наведено у лістингу 3.1:

Лістинг 3.1.

```

node * create_tree (char k)
{
node *work;
if (k!='y') return (NULL);
else
{
cout<<endl<<"enter info new element ";
work=new node;

```

```

cin>>work->info;
cout<<endl<<"continue left?  ";
cin>>k; work->left=create_tree(k);
cout<<endl<<"continue right?  ";
cin>>k; work->right=create_tree(k);
return (work);
}
}

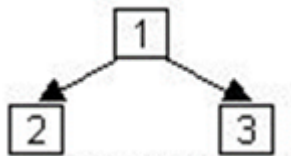
```

3.2 Способи обходу бінарних дерев

Обхід дерева - процес відвідування всіх його вершин у певному порядку. Існує кілька способів обходу (проходження) всіх вузлів дерева. Три найбільше часто використовувані способи обходу:

- обхід у прямому порядку,
- обхід у зворотному порядку,
- кінцевий (симетричний) обхід.

Прямий порядок проходження бінарного дерева ("униз", "у глибину"):



1. потрапити в корінь,
2. пройти в прямому порядку ліве піддерево,
3. пройти в прямому порядку праве піддерево.

На рисунку 3.1 подано схему прямого обходу бінарного дерева:

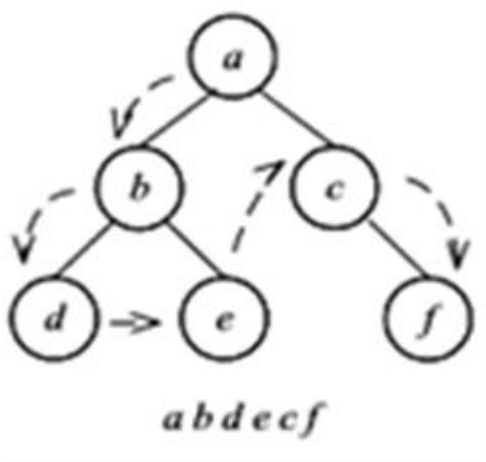


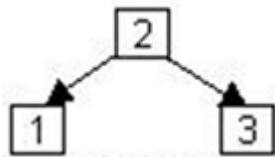
Рис. 3.1. Прямий обхід бінарного дерева.

Рекурсивна функція проходження дерева в прямому порядку наведена в лістингу 3.2:

Лістинг 3.2.

```
void tree_1 (node * bale)
{
node* work;
work=bale;
cout<<endl<<work->info;
if (work->left!=NULL) tree_1 (work->left);
if (work->right!=NULL) tree_1 (work->right);
}
```

Проходження бінарного дерева у *зворотному порядку* ("нагору"):



1. пройти у зворотному порядку ліве піддерево
2. потрапити в корінь
3. пройти у зворотному порядку праве піддерево.

На рисунку 3.2 подано схему зворотного обходу бінарного дерева:

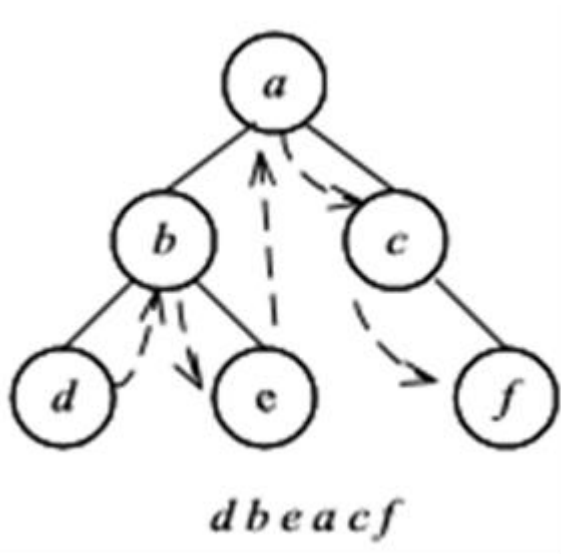


Рис. 3.3. Зворотний обхід бінарного дерева.

Рекурсивна функція проходження дерева в зворотному порядку наведена в лістингу 3.3:

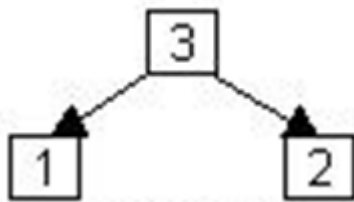
Лістинг 3.3.

```
void tree_2 (node * bale)
{
node* work;
work=bale;

if (work->left!=NULL) tree_2 (work->left);
```

```
cout<<endl<<work->info;
if (work->right!=NULL) tree_2 (work->right);
}
```

Кінцевий (симетричний) обхід дерева



1. пройти в симетричному порядку ліве піддерево,
2. пройти в симетричному порядку праве піддерево,
3. потрапити в корінь.

На рисунку 3.3 подано схему симетричного обходу бінарного дерева:

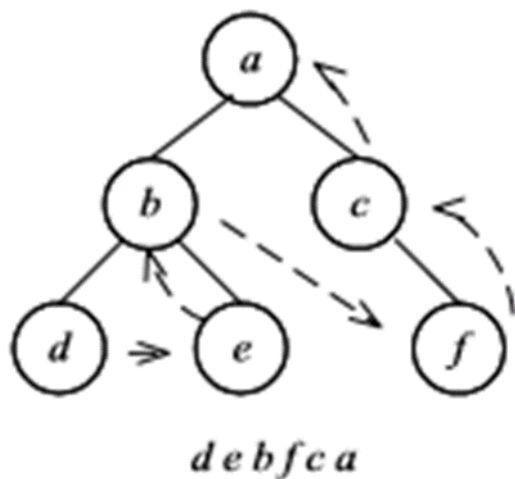


Рис. 3.3. Симетричний обхід бінарного дерева.

Рекурсивна функція проходження дерева в симетричному порядку наведена в лістингу 3.4:

Лістинг 3.4.

```
Void tree_3 (node * bale)
{
node* work;
work=bale;

if (work->left!=NULL) tree_3 (work->left);

if (work->right!=NULL) tree_3 (work->right);
cout<<endl<<work->info;
}
```

3.3 Видалення бінарного дерева.

Слід враховувати, що при видаленні вузла дерева спочатку повинні бути видалені його лівий та правий нащадок, а потім видається сам вузол. Отже для реалізації цієї операції слід використовувати кінцевий (симетричний) обхід дерева. Рекурсивна функція видалення бінарного дерева наведена в лістингу 3.5.

Лістинг 3.5.

```
node * delete_tree (node * bale)
{

    node *work;
    work=bale;

    if (work->left!=NULL) work->left=delete_tree(work->left);

    if (work->right!=NULL) work->right=delete_tree(work->right);

    cout<<endl<<work->info;
    getch();
    delete work; bale=NULL; return (bale);
}
```

Питання та завдання до контролю знань студентів

- 1 Яку структуру мають вершини двійкового дерева?
 - 2 Чому для дерев існує кілька правил обходу вершин?
 - 3 Які правила обходу вершин дерева є основними?
 - 4 Як виконується обхід дерева в прямому напрямку?
 - 5 Як виконується обхід дерева в симетричному напрямку?
 - 6 Як виконується обхід дерева у зворотному напрямку?
 - 7 Як виконується обхід дерева в назад-симетричному напрямку?
 - 8 Чому рекурсивна реалізація правил обходу є найбільш зручною?
 - 9 Що відбувається при рекурсивному виконанні обходу дерева?
 - 10 Як програмно реалізується обхід дерева в прямому напрямку?
 - 11 Як програмно реалізується обхід дерева в симетричному напрямку?
 - 12 Як програмно реалізується обхід дерева у зворотному напрямку?
 - 13 Який формальний параметр необхідний для рекурсивної реалізації правил обходу і як він використовується?
- Як правильно виконати знищення всієї деревоподібної структури?

Лекція № 8

з навчальної дисципліни «Алгоритми та структури даних»

Тема Поняття рекурсії. Реалізація рекурсії. Рекурсивні математичні функції

Мета заняття Сформулювати у студентів вміння реалізації рекурсивного звернення до функції. Розглянути приклади організації обчислення рекурсивних математичних функцій.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Поняття рекурсії
- 2 Приклади завдань рекурсивного рішення
- 3 Рекурсивні математичні функції
- 4 Непряма рекурсія

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

1 Поняття рекурсії

Рекурсія (від латинського *recursio* - повернення) - це такий спосіб організації обчислювального процесу, при якому процедура або функція в ході виконання складових її операторів звертається сама до себе.

Для того, щоб таке звернення не було нескінченним, в тексті підпрограми має бути умова, по досягненню якого подальшого звернення не відбувається. Таким чином, рекурсивне звернення може включатися тільки в одну з гілок підпрограми.

У таких мовах програмування, як Паскаль та C++, немає ніяких обмежень на рекурсивні виклики підпрограм, необхідно тільки розуміти, що кожен черговий рекурсивний виклик приводить до утворення нової копії локальних об'єктів підпрограми і усі ці копії, відповідні ланцюжку активізованих і не завершених рекурсивних викликів, існують незалежно один від одного

Рекурсія досить широко застосовується в програмуванні, що засновано на рекурсивній природі багатьох математичних алгоритмів. А також Ви повинні знати, що будь-який рекурсивний алгоритм можна перетворити в еквівалентний ітеративний (тобто що використовує циклічні конструкції).

У великих і складних програмах іноді доводиться замінювати рекурсію на ітерацію. Річ у тому, що рекурсія пов'язана з багатократними викликами процедур, а це декілька менш ефективно при виконанні в порівнянні з використанням циклів. Проте рекурсивні версії програм, як правило, набагато коротше і наочніше.

2. Приклади завдань рекурсивного рішення

Хорошою ілюстрацією механізму рекурсії є функція для обчислення факторіалу натурального числа. Згадаємо, що факторіалом числа називається добуток усіх натуральних чисел від 1 до цього числа включно:

$$N! = 1 * 2 * 3 * \dots * (N - 1) * N$$

$$1! = 1$$

$$0! = 1$$

Спочатку покажемо звичайну не рекурсивну функцію для обчислення факторіалу, яка реалізує ітеративний алгоритм обчислення :

```
long NonRecFact( int N)
{
    int i;           // змінна циклу
    long Res(1);      //результат
    for (i= 1; i<=N; i++)
        Res:= Res*i;
    return Res; }
```

Друга функція використовує рекурсивні звернення, що робить її набагато компактніше, і заснована вона на очевидному співвідношенні:

$$N! = (N - 1)! * N$$

Іншими словами, щоб набути значення факторіалу від числа N, досить помножити на N значення факторіалу від попереднього числа:

```
long  ResFact( int N)
{
    long Res;
    if (N <= 1)
        Res = 1
    else
        Res=N*ResFact(N - 1);
    return Res;
}
```

Повністю програма, що обчислює факторіал числа, виглядатиме так:

```
...
long  ResFact( int N)
{
    long Res;
    if (N <= 1)
        Res = 1
    else
        Res=N*ResFact(N - 1);
    return Res;
}
void main ()
{
    int N;
    long F;
    cout<<endl<<"Введіть число N";
    cin>>N;
    F:= ResFact(N);
    Cout<<endl<<"Для числа"<< N<<" значення факториала равно "<<
F;

    return;
}
```

Після запуску програми на екран виводиться запит "Введіть число N", потім з клавіатури прочитується введене значення і у виразі $F := \text{ResFact}(N)$ викликається функція ResFact з параметром-значенням N. У функції перевіряється умова $N \leq 1$. Якщо вона виконується, то функція ResFact повертає значення 1 і на цьому виконання функції завершується. Якщо умова $N \leq 1$ не вірна, то виконується обчислення добутку $N * \text{ResFact}(N - 1)$. Обчислення добутку носить рекурсивний характер, оскільки при цьому здійснюється виклик функції ResFact(N - 1), значення якої обчислюється, у свою чергу, через виклик функції ResFact, параметром якої також буде функція

ResFact, і так далі, до тих пір, поки значення формального параметра N не дорівнюватиме 1. Оскільки базова частина опису рекурсивної функції ResFact визначає значення ResFact для N=1, рівним одиниці, то рекурсивні виклики функції ResFact більше не виконуються, а навпаки виконується обчислення функції ResFact для чисел, що зростають від 1 до N, причому функція ResFact всякий раз повертає значення, рівне твору чергового числа на факторіал від попереднього числа. Останнє повернення результату обчислення функції ResFact присвоїть змінній F значення добутку усіх чисел від 1 до N, тобто факторіал числа N.

Отже, при виконанні рекурсивної підпрограми здійснюється багатократний перехід від деякого поточного рівня організації алгоритму до нижнього рівня послідовно до тих пір, поки не буде отримано тривіальне рішення поставленої задачі. У нашому прикладі рішення при N=1 тривіальне, тобто ResFact=1. Потім здійснюється повернення на верхній рівень з послідовним обчисленням значення функції ResFact.

При організації рекурентних викликів підпрограм (функцій) необхідно строго дотримання двох умов:

- 1 У тілі рекурсивної підпрограми обов'язково повинне бути умова, при виконанні якої припиняється рекурсивний виклик підпрограми - термінальна умова.
- 2 В основі рекурсивних рішень обчислювальних процесів лежать рекурентні співвідношення.

3 Рекурсивні математичні функції

Приклад.

Постановка задачі: Одержати m-й член послідовності: $\frac{\cos^{m-1}(3x)}{(x-5)^{m+2}}$.

Значення x й m вводяться користувачем.

Розробка рекурентного співвідношення й термінальної умови:

Термінальна умова: $y_1 = \frac{1}{(x-5)^2}$;

Рекурентне співвідношення: $y_m = y_{m-1} * \frac{\cos(3x)}{(x-5)^2}$;

Текст функції:

```
...
float P( float z, int n)
{   if (n==1) p=1/((z-5)*(z-5))
    Else   p=P(z, n-1)*(cos(3*z)/((z-5)*(z-5)));
}
void main ()
{
int m;
float a;
cout<<endl<<"Введіть аргумент послідовності"; cin>>a;
```

```

cout<<endl<<"Введіть номер члена послідовності";
cin>>m;
cout<<endl<<m<<"-й член послідовності дорівнює  "<<P(a,m);
return;
}

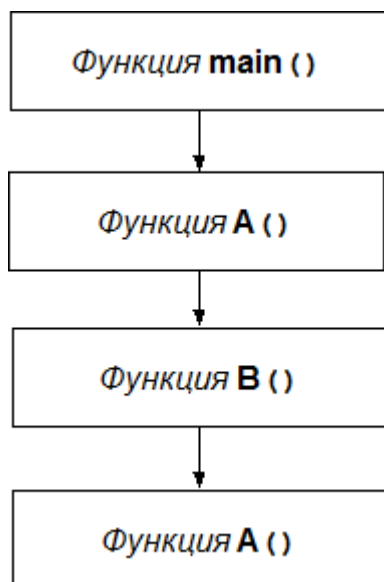
```

4 Непряма рекурсія

Розглянуті вище програми використовували так звану пряму рекурсію, коли в тілі деякої процедури містився безпосередній виклик самій себе. У мові Паскаль допускається також і непряма рекурсія, коли, наприклад, процедура, процедура А викликає процедуру В, а та, у свою чергу, - процедуру А. Довжина таких ланцюжків викликів може бути довільною, проте при розробці програми необхідно ретельно стежити за тим, щоб рекурсивний алгоритм сходився, тобто не призводив до нескінченних взаємних викликів підпрограм.

Образно непряму рекурсію можна описати так. Перед дзеркалом 1 розміщене дзеркало 2, в якому відбивається саме дзеркало 1. У останньому видно дзеркало 2 і так далі.

Непряма рекурсія створюється за рахунок виклику даної функції з якої-небудь іншої функції, що сама викликалася з даної функції. Наприклад, схема може бути такою:



Наведемо приклад програми, що ілюструє непрямі рекурсивні виклики процедур. У цій програмі процедури Rec1 і Rec2 рекурсивно викликають один одного, по черзі зменшуючи свої фактичні параметри. Легко бачити, що обидві процедури працюють з однією глобальною змінною А, яка передається в них по посиланню. Критерієм завершення роботи є звернення цієї змінної в нуль.

Зверніть увагу, що в програмі потрібне попереднє визначення другої процедури Rec2, оскільки її виклик зустрічається в процедурі Rec1, тобто перед її повним описом.

```
...
void Rec2 ( int Y );
void Rec1 (int X );
void main ()
{
    int A(15);
    Rec1(A);
return;
}
void Rec1 (int X )
{
    X:= X - 1;
    if (X>0) Rec2;
    cout<<endl<<X;
return;
}
void Rec2 ( int Y )
{
    Y:= Y / 2;
    if (Y>2) Rec1;
    cout<<endl<<Y;
return;
}
```

Короткі підсумки

Рекурсія характеризується визначенням об'єкта за допомогою посилання на себе.

Рекурсивні алгоритми містять у своєму тілі прямий або опосередкований обіг із самим собі.

Рекурсивні функції містять у своєму тілі звертання до самого себе зі зміненим набором параметрів у вигляді прямої рекурсії. При цьому звертання до себе може бути організоване за допомогою непрямої рекурсії - через ланцюжок взаємних обігів функцій, що замикаються в підсумку на первісну функцію.

Рішення завдань рекурсивними способами проводиться за допомогою розробки рекурсивної тріади.

Доцільність застосування рекурсії в програмуванні обумовлена специфікою завдань, у постановці яких явно або опосередковано вказується на можливість віднесення завдання до під завдань, аналогічних самому завданню.

Рекурсивні методи рішення завдань широко використовуються при моделюванні завдань із різних предметних областей.

Рекурсивні алгоритми ставляться до ресурсномістких алгоритмів. Для оцінки складності рекурсивних алгоритмів ураховується число вершин повного рекурсивного дерева, кількість переданих параметрів, тимчасові витрати на організацію стекових шарів.

Питання та завдання до контролю знань студентів

Питання для узагальнення та перевірки засвоєного матеріалу на лекції

- 1 Яка функція називається рекурсивною?
- 2 Які умови є необхідними для правильної організації рекурсивних викликів функцій?
- 3 Що таке термінальна умова?
- 4 Навіщо використовується рекурентне співвідношення?
- 5 Яка рекурсія називається непрямою?

Завдання для самоперевірки засвоєного матеріалу на лекції

Напишіть програми, що демонструють виконання рекурсивного алгоритму для завдань:

1 На друк виводиться казка "Про попа і його собаку" певне число разів. ("У попа був собака, він її любив. Вона з'їла шматок м'яса - він її убив. У землю закопав, написав ..")

2 Напишіть рекурсивний алгоритм знаходження:

- а) n -го члена геометричної прогресії
- б) знаходження суми n членів арифметичної прогресії
- в) знаходження суми n членів геометричної прогресії
- г) знаходження n -го члена ряду Фібоначчі.

3 Сформулюйте термінальну умову та рекурентне співвідношення для обчислення n -го члена наступної послідовності: 2, 4, 16, 265, ...

Лекція № 9

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Рекурсивний перебір з поверненнями.
Мета заняття	Дати характеристику методу перебору та описати обчислювальну схему перебору з поверненням. Розглянути метод рішення та реалізацію завдання про розставляння ферзів на шахівниці.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	
Конспект лекцій.	

Час – 2 години

План проведення лекції

- 1 Метод перебору
- 2 Обчислювальна схема перебору з поверненням
- 3 Завдання про розставляння ферзів на шахівниці.
- 4 Завдання про кількість розставлень ферзів на шахівниці.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Метод перебору

У багатьох практичних завданнях з різних предметних областей вимагається знайти загальну кількість варіантів рішення, число елементів в повному наборі рішень. Іноді, виходячи з постановки завдання, досить знайти один з варіантів, відповідних умові завдання. У деяких завданнях вивчається питання про існування рішення як такого.

Відповіді на поставлені питання, як правило, вимагають проведення вичерпного пошуку в деякій безлічі усіх можливих варіантів, серед яких знаходяться рішення конкретної задачі. Існують два загальні методи організації вичерпного пошуку: перебір з поверненням (backtracking) і його природне логічне доповнення - метод решета.

Рішення задачі методом перебору з поверненням будується конструктивно послідовним розширенням часткового рішення. Якщо на конкретному кроці таке розширення провести не вдається, то відбувається повернення до коротшого часткового рішення, і спроби його розширити тривають. Для прискорення перебору з поверненням обчислення завжди прагнуть організувати так, щоб була можливість відмовитися якомога раніше від як можна більшого числа свідомо невідходящих варіантів. Незначні модифікації методу перебору з поверненням, пов'язані з представленням даних або особливостями реалізації, мають і інші назви: метод гілок і меж (branchandbound), пошук в глибину (depthfirstsearch), метод проб і помилок і т. д. Перебір з поверненням практично одночасно і незалежно був винайдений багатьма дослідниками ще до його формального опису. Він знаходить застосування при рішенні різних комбінаторних завдань в області штучного інтелекту.

При використанні методу решета замість конструктивної побудови рішень задачі з безлічі можливих варіантів виключаються усі елементи, що не є рішеннями. Методи решета знайшли широке застосування в теоретико-числових завданнях.

Метод перебору з поверненням і метод решета, строго кажучи, не є ні методами, ні алгоритмами рішення завдань. Їх слід сприймати як на деякі загальні схеми, які застосовуються для вирішення того або іншого завдання. Реалізація цих схем у вигляді конкретних алгоритмів часто вимагає значних додаткових зусиль в представленні даних і описі залежностей між ними.

З'єднання методу перебору з поверненням і рекурсії визначає специфічний спосіб реалізації рекурсивних обчислень і називається поворотною рекурсією. Це з'єднання двох ефективних методів реалізації переборних алгоритмів. При використанні поворотної рекурсії відпадає необхідність безпосередньо організовувати повернення

і відстежувати правильність їх здійснення. Вони, як правило, стають вбудованою частиною механізму виконання рекурсивних викликів.

2 Обчислювальна схема перебору з поверненням

Опишемо загальну постановку класу завдань, до яких свідомо застосуємо алгоритм перебору з поверненням.

Нехай $M_0, M_1, \dots, M_{n-1}$ - n кінцевих лінійно впорядкованих великих кількостей і G - сукупність обмежень (умов), що ставлять у відповідність векторам виду, булеве значення $\{ \text{істина, брехня} \}$. Вектори $v=(v_0, v_1, \dots, v_k) T$, для яких $G(v) = \text{істина}$, назвемо частковими рішеннями. Нехай, далі, існує конкретне правило P , відповідно до якого деякі з часткових рішень можуть оголошуватися повними рішеннями. Тоді можлива постановка наступних пошукових завдань.

- Знайти усі повні рішення або встановити відсутність таких.
- Знайти хоч би одне повне рішення або встановити його відсутність.

Загальний метод рішення приведених завдань полягає в послідовному покомпонентному нарощуванні вектору v зліва направо, починаючи з v_0 , і наступних перевірок його обмеженнями G і правилом P .

У загальному випадку цей метод призводить до алгоритмів з експоненціальною тимчасовою складністю, а застосовується він в основному до класу так званих NP - повних завдань (завдання комівояжера, завдання про рюкзак і т. д.). Завдання цього класу еквівалентні один одному в тому сенсі, що усі вони вирішувані недетермінованими алгоритмами поліноміальної складності. Для них відомо, що або усі вони вирішувані, або жодна з них не вирішувана детермінованими алгоритмами поліноміальної складності. Іншими словами, якщо хоч би для одного з цих завдань не існує детермінованого алгоритму, що має у гіршому разі поліноміальну трудомісткість, то такі алгоритми не повинні існувати і для інших завдань цього класу. Навпаки, якщо хоч би для одного з цих завдань вдалося знайти детермінований алгоритм, що має у гіршому разі поліноміальну трудомісткість, то подібні алгоритми існували б і для інших завдань цього класу і, більше того, їх можна було б побудувати.

Опишемо схему виконання недетермінованого алгоритму. Нехай алгоритм виконується до тих пір, поки не доходить до місця, з якого має бути зроблений вибір з декількох альтернатив. Детермінований алгоритм однозначно здійснить вибір конкретної альтернативи і продовжить працювати відповідно до ці вибором. Недетермінований алгоритм досліджує усі можливості одночасно, як би копіюючи себе для реалізації обчислень по усіх альтернативах одночасно. Далі усі копії працюють незалежно один від одного і в міру необхідності продовжують створювати нові копії. Копія, що зробила неправильний або безрезультатний вибір припиняє свою роботу. Копія, що знайшла рішення задачі, оголошує про це, даючи тим самим сигнал іншим копіям про припинення обчислень. Недетерміновані алгоритми, будучи дуже корисною і

продуктивною абстракцією, рекурсивні по суті, бо при реалізації оператора вибору фактично звертаються самі до себе.

Можливо, що багато завдань, що вирішуються методом перебору з поверненням, можуть бути вирішені ефективніше іншими способами. Проте цінність методу перебору з поверненням в з'єднанні з рекурсією незаперечна. По-перше, програми рішення багатьох завдань будуються за єдиною схемою, а по-друге, вони компактні і тим самим прості для розуміння і засвоєння відповідних ідей.

3 Завдання про розставляння ферзів на шахівниці.

Складіть рекурсивну функцію, що знаходить можливе розставляння n ферзів на шахівниці розміром $n \times n$ так, щоб вони не били один одного (n - натуральне число).

Відповідно до загальної схеми алгоритму перебору з поверненням запропоновану задачу можна вирішувати по алгоритму "Усі розставляння", але завершити виконання алгоритму при знаходженні першого необхідного розставляння або при отриманні висновку про неможливість отримати потрібну комбінацію. Згідно з алгоритмом ферзі розставляються послідовно на вертикалях з номерами від нуля і далі. В процесі виконання приписи можливі зняття ферзів з дошки (повернення).

Алгоритм "Усі розставляння"

Крок 1. Вважаємо (D - безліч рішень, j - поточний стовпець для чергового ферзя).

Крок 2. Намагаємося в стовпці j просунути вниз по вертикалі або новий (якщо стовпець j порожній), або вже наявний там ферзь на найближчий допустимий рядок. Якщо це зробити не вдалося, то переходимо до кроку 4.

Крок 3. Збільшуємо j на 1, тобто переходимо до наступного стовпця. Якщо $j < n - 1$, то переходимо до кроку 2. Інакше $j = n - 1$, тобто усі вертикалі вже зайняті. Знайдене часткове рішення запам'ятовуємо в безлічі D і переходимо до кроку 2.

Крок 4. Зменшуємо j на 1, тобто знімаємо ферзь із стовпця j і переходимо до попереднього стовпця. Якщо $j > 0$, то виконуємо крок 2. Інакше обчислення припиняємо. Рішення задачі знаходяться в безлічі D , яка, може бути і порожнім.

Рішення задачі для невеликих розмірів дошки можна знайти "вручну", але для розробки алгоритму необхідно провести моделювання умови і зупинитися на якому-небудь представленні даних.

Проведемо параметризацію завдання. Введемо чотири допоміжні вектори: pos , ho , dd і du с довжинами n , n , $2n-1$ і $2n-1$ відповідно. Використовувати їх будемо таким чином:

$ho_i = 1$, якщо на горизонталі з номером i ($i = 0, 1, \dots, n - 1$) є ферзь, і $ho_i = 0$ - інакше;

$du_s = 1$, якщо на діагоналі з номером s ($s = 0, 1, \dots, 2n-2$), що йде зліва направо і від низу до верху, є ферзь, і $du_s = 0$ - інакше;

$dd_s=1$, якщо на діагоналі з номером s ($s=0,1,..2n-1$), що йде зліва направо і зверху вниз, є ферзь, і $dd_{s+1}=0$ - інакше;
 $pos_j=i$, якщо в позиції ($i, j=0,1,..n-1$) стоїть ферзь.

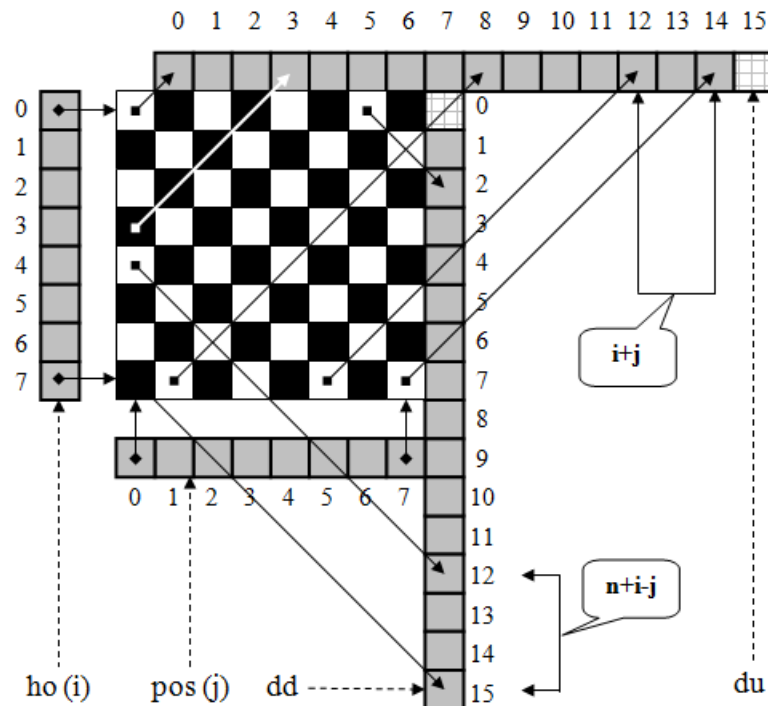


Схема використання допоміжних масивів в завданні

Використання цих угод дозволяє отримати такі твердження:

У позицію (i, j) можна поставити ферзь, якщо $ho_i + du_{i+j} + dd_{n+i-j} = 0$.

Поставити ферзь в позицію (i, j) рівносильно привласненню: $ho_i=1, du_{i+j}=1, dd_{n+i-j}=1$.

Прибрати ферзь з позиції (i, j) рівносильно привласненню: $ho_i=0, du_{i+j}=0, dd_{n+i-j}=0$.

Цей опис алгоритму є моделлю рішення загальної задачі про знаходження усіх варіантів розставлянь. Рекурсія тут здійснюється за не зовсім стандартною схемою. У кожному рекурсивному виклику глибини j робиться спроба помістити ферзь в деяку позицію i стовпця j ($i, j=0,1,..n-1$), а сам виклик відповідає переходу від роботи з поточним стовпцем до роботи з наступним стовпцем. При цьому на початку обчислень і при переходах до будь-якого наступного рекурсивного виклику параметр i міняється від нуля і далі з кроком, рівним одиниці, намагаючись набути значення найменшого номера поля, допустимого для установки ферзя. При переходах до будь-якого попереднього рекурсивного виклику параметр i продовжує змінюватися від свого поточного на цьому рівні значення з кроком, рівним одиниці, також намагаючись набути значення найменшого номера поля, допустимого для установки ферзя. Якщо в поточному стовпці ферзь встановити вже не вдається, то ситуацію, що ство-

рилася, назовемо безвихіддю. Попадання в безвихідь призводить до завершення поточного рекурсивного виклику, тобто до повернення до попереднього стовпця і продовження роботи з ним. Інших випадків завершення рекурсивних викликів не існує. Тому базою рекурсії ми повинні рахувати сукупність усієї безвиході. Помітимо, що в даному випадку елементи бази заздалегідь до обчислень невідомі.

Після установки ферзя в один з рядків і останнього стовпця $i=n-1$ формується одне з рішень задачі - при пошуку одного варіанту розставляння на цьому етапі слід завершити виконання алгоритму. При пошуку усіх розставлянь обчислення припиняються, коли ми потрапляємо в безвихідь при роботі із стовпцем 0. Отримані рішення задачі, якщо вони є, повертаються у вигляді стовпців матриці otv , починаючи від першого і далі.

Розглянемо, як реалізується декомпозиція. Для цього замість початкового завдання зручно вирішувати її наступне узагальнення.

На дошці розміру $n \times m$ ($m=n$, $n - 1, \dots, 0$) вимагається встановити m ферзів так, щоб вони не били один одного. При цьому є деякі клітини дошки, на які ферзь свідомо ставити не можна. Безліч цих "заборонених" клітин позначимо через *

Початкове завдання є . Проведемо її декомпозицію. Представимо дошку у вигляді двох частин: нульового стовпця (А) і частини (В), що залишилася. Відповідно до цього розбиття вирішуватимемо завдання i . Кожне з n можливих рішень i (ферзь встановлений в рядку $i=0, 1, \dots, n - 1$) першого завдання однозначно визначає безліч заборонених клітин для другого завдання. При цьому в потрапляють ті клітини В, які в "об'єднаній" дошці б'є ферзь, встановлений в рядку i дошки А. Нехай i зафіксоване і знайдене безліч $d(i)$ рішень другої задачі для дошки В. Тоді розташування ферзя в рядку i дошки А і будь-яке з отриманих рішень на В в сукупності дають різні рішення $otv(i)$ початкового завдання . Для отримання усіх рішень цієї задачі залишається лише узяти об'єднання і безлічі $otv(i)$.

При аналізі трудомісткості алгоритму отримуємо, що глибина рекурсії рівна n^2 - при кожному рекурсивному виклику по j ($j=0, 1, \dots, n - 1$) відбувається n рекурсивних викликів по i ($i=0, 1, \dots, n - 1$).

```
//знаходження одного варіанту розставляння
#include "stdafx.h"
#include <iostream>
using namespace std;
void Initialization(int n, int ***x);
void Destruction(int n, int **x);
bool Queen(int n, int **x);
void Placement(int n, int **a, int *p, int *h, int *du,
int *dd, inti=0, int j=0);

int _tmain(intargc, _TCHAR* argv[]){
int n, i, j;
boolotv;
```

```

int **mas;
printf("Введите размер шахматной доски n : ");
scanf ("%d",&n);
Initialization(n,&mas);
otv = Queen(n,mas);
if ( otv ) printf("Правильное размещение найдено\n");
else printf("Правильного размещения не найдено\n");
Destruction(n,mas);
system("pause");
return 0;
}

//ініціалізація поля шахівниці
void Initialization(int n, int ***x){
    *x = new int*[n];
    for (inti = 0 ; i< n; i++ ){
        (*x)[i] = new int[n];
        for (int j = 0 ; j < n ; j++ )
            (*x)[i][j] = 0;
    }
}

//виведення знайденого розставляння у файл
void Destruction(int n, int **x){
    FILE *f;
    if( ( f = fopen("out.txt","w") ) == NULL ){
        printf("Файл out.txt не может быть открыт для записи");
    }
    else{
        fprintf(f,"%d\n",n);
        for (inti = 0 ; i< n; i++ ){
            for (int j = 0 ; j < n ; j++ )
                fprintf(f,"%2d",x[i][j]);
            fprintf(f,"\n");
        }
        for (inti = 0 ; i< n; i++ )
            delete [] x[i];
        delete [] x;
    }
}

// перевірка можливості постановки ферзя
bool Queen(int n, int **x){
    bool rez = false;
    int *p, *h, *du, *dd;
    p = new int[n];
    h = new int[n];
    du = new int[2 * n - 1];

```

```

dd = new int[2 * n - 1];
for ( inti = 0 ; i< n ; i++ )
p[i] = h[i] = 0;
for ( inti = 0 ; i< (2 * n - 1) ; i++ )
du[i] = dd[i] = 0;
Placement(n,x,p,h,du,dd);
if ( h[0] != 0 ) rez = true;
delete [] dd, du, h, p;
return rez;
}

// опис функції розставляння ферзів
void Placement(int n, int **a, int *p, int *h, int *du,
int *dd, inti, int j){
if ( j >= 0 && j < n )
if ( i< n )
if (h[i]==0 && du[i+j] == 0 && dd[n+i-j-1] == 0 ) {
h[i] = 1;
du[i+j] = 1;
dd[n+i-j-1] = 1;
p[j] = i;
a[i][j] = 1;
Placement(n,a,p,h,du,dd,0,j+1);
}
else
Placement(n,a,p,h,du,dd,i+1,j);
else
if ( j > 0 ) {
h[p[j-1]] = 0;
du[p[j-1]+j-1] = 0;
dd[n+p[j-1]-(j-1)-1] = 0;
a[p[j-1]][j-1] = 0;
Placement(n,a,p,h,du,dd,p[j-1]+1,j-1);
}
}
}

```

4 Завдання про кількість розставлянь ферзів на шахівниці.

Скласти рекурсивну функцію, що знаходить кількість можливих розставлянь п ферзів на шахівниці розміром $n \times n$ так, щоб вони не били один одного. Запропоновану задачу можна вирішувати по приведених в Прикладі 1 функціям, спростивши їх таким чином. Замість запам'ятовування знайдених рішень підраховуватимемо в змінній *otv* їх кількість.

Завдання про основні розставляння.

Для формулювання наступного завдання введемо поняття. Серед усіх розставлянь n ферзів на дошці $n \times n$ виділимо окремі класи H_s ($s=0,1,\dots, q$) розставлянь, що не перетинаються, так, що усі елементи цього класу можна отримати з будь-якого його представника якими-небудь елементарними перетвореннями типу :

- поворот дошки в її площині навколо центру на 90, 180 і 270 градусів ;
- перетворення симетрії відносно діагоналей;
- перетворення симетрії відносно прямих, що проходять через центр дошки по межах клітин.

Узявши по одному представникові з кожного класу H_s ($s=0,1,\dots, q$), отримаємо деяку множину, звану основними розставляннями. Скласти рекурсивну функцію, що знаходить яку-небудь безліч основних розставлянь n ферзів на шахівниці розміру $n \times n$. Це завдання вирішується за допомогою рекурсивних функцій практично аналогічно завданню з Прикладу 1. Відмінності тут такі. Рекурсивна функція послідовно формує кожне з можливих розставлянь ферзів на дошці, але не усі з них запам'ятовуються в матриці відповіді otv . Чергове отримане розставляння піддається перевірці - включати або не включати її в матрицю otv . Робиться це таким чином. З вектору pos описаними вище елементарними перетвореннями формуються ще сім розставлянь (деякі з них можуть виявитися співпадаючими). Якщо жодна з них не входить в поточну матрицю відповідей otv , то otv доповнюється новим рішенням і так далі. Отже, після закінчення обчислень otv міститиме деяку безліч основних розставлянь.

Ключові терміни

Поворотна рекурсія - це з'єднання методу перебору з поверненням і рекурсії.

Детермінований алгоритм - це алгоритм, який однозначно здійснить вибір конкретної альтернативи і продовжить працювати відповідно до ці вибором.

Вичерпний пошук - це процес знаходження в деякій безлічі усіх можливих варіантів, серед яких є рішення конкретної задачі.

Метод решета - це один з методів організації вичерпного пошуку, при якому з безлічі можливих варіантів виключаються усі елементи, що не є рішеннями.

Недетермінований алгоритм - це алгоритм, який досліджує усі можливості одночасно, як би копіюючи себе для реалізації обчислень по усіх альтернативах одночасно.

Перебір з поверненням (backtracking) - це один з методів організації вичерпного пошуку, який будується конструктивно послідовним розширенням часткового рішення.

Повне рішення - це набір варіантів, що утворює хоч би одне рішення в цілому.

Часткове рішення - це неповний набір варіантів, який входить в одне або декілька повних рішень.

Короткі підсумки

Рішення переборних завдань зводиться до визначення наявності рішення, усіх різних варіантів рішення або до знаходження одного рішення.

Залежно від постановки завдання переборные алгоритми реалізуються методами перебору з поверненням або методом решета.

Поворотна рекурсія - це з'єднання методу перебору з поверненням і рекурсії.

Детермінований алгоритм однозначно здійснить вибір конкретної альтернативи і продовжить працювати відповідно до цієї вибором. Недетермінований алгоритм досліджує усі можливості одночасно, як би копіюючи себе для реалізації обчислень по усіх альтернативах одночасно.

Для вирішення завдання про розставляння ферзів на шахівниці як альтернативний використовується метод поворотної рекурсії.

При розробці тріади для вирішення завдання про розставляння ферзів вводяться додаткові параметри при описі процесу розставляння.

Базою в цьому завданні вважається сукупність усіх тупикових ситуацій.

Декомпозиція проводиться від часткового до повного рішення або до зняття ферзя з вертикалі (повернення).

Питання та завдання до контролю знань студентів

1. У чому проявляється рекурсивність методу перебору з поверненням?
2. Чому повний метод перебору з поверненням гарантує відшукування усіх рішень задачі?
3. Як формується рекурсивна база методу поворотної рекурсії?
4. Які класи завдань зводяться до розробки детермінованих алгоритмів, а які - до недетермінованих? Поясніть прикладами.
5. Поясніть, чому ці описи характеризують опис дій над ферзем в контексті моделі шахівниці :

У позицію (i, j) можна поставити ферзя, якщо $hoi + dui + j + ddn + i - j = 0$.

Поставити ферзя в позицію (i, j) рівносильно привласненню: $hoi = 1$, $dui + j = 1$, $ddn + i - j = 1$.

Прибрати ферзя з позиції (i, j) рівносильно привласненню: $hoi = 0$, $dui + j = 0$, $ddn + i - j = 0$.

Лекція № 10

з навчальної дисципліни «Алгоритми та структури даних»

Тема Методи сортування: вибором, вставками, злиттям, обмінами, шейкерне сортування

Мета заняття Ознайомити студентів з основними методами сортування в лінійних структурах даних.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

1 Алгоритми сортування лінійних масивів

1.1 Сортування обмінами

1.2 Сортування простими вставками

1.3 Сортування методом вибору.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Вступ

Упорядкування елементів множини даних у зростаючому або спадаючому порядку називається сортуванням.

З упорядкованими елементами працювати простіше, ніж з довільно розташованими: легше знайти елементи, виключити, вставити нові. Сортування застосовується при трансляції програми, при організації наборів даних на зовнішніх носіях, при створенні бібліотек, каталогів, баз даних і т.д.

Алгоритми сортування можна розбити на наступні групи (рисунок 1).

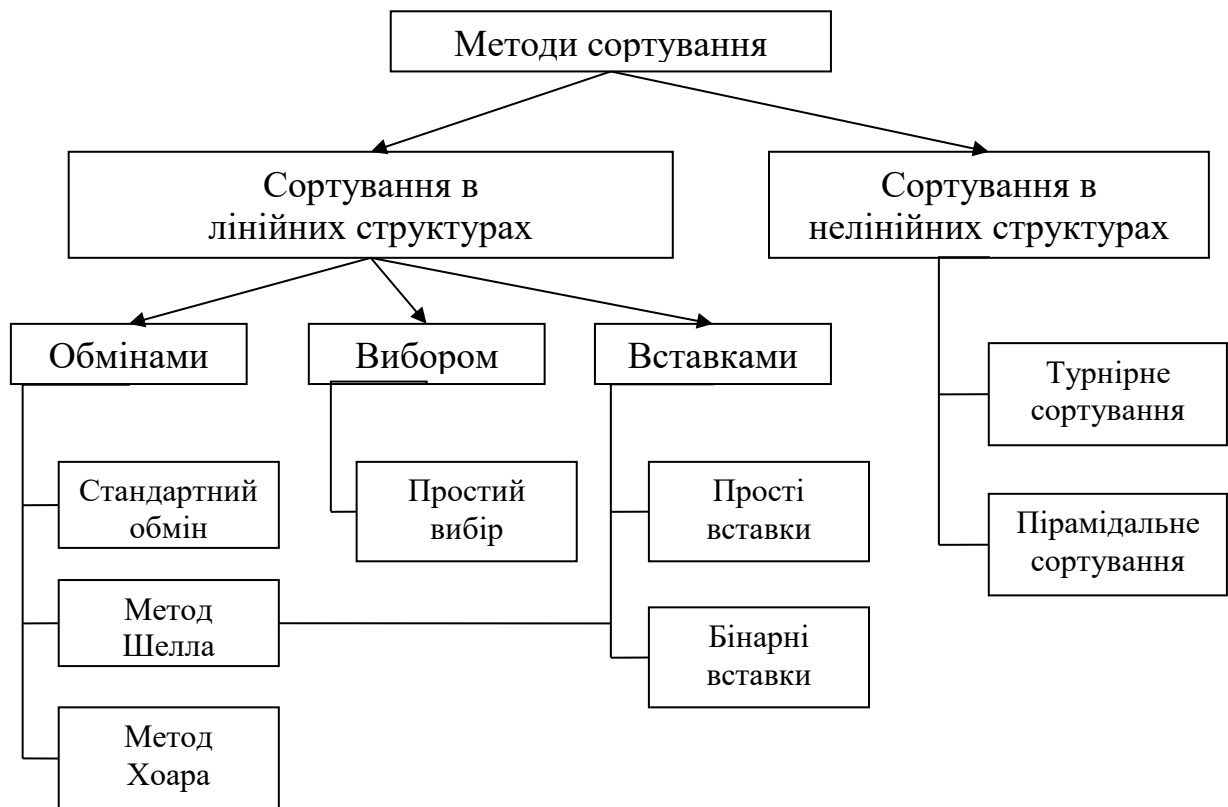


Рисунок 1. Класифікація методів сортування.

1 Алгоритми сортування лінійних масивів

1.2 Сортування обмінами

Стандартний обмін («бульбашковий» метод сортування)

Одномірний масив із n цілих чисел відсортуємо за зростанням значень елементів. Алгоритм полягає в тому, що при перегляді вхідного масиву попарно порівнюються сусідні елементи. Якщо порядок їхнього проходження не відповідає заданому критерію впорядкованості, то елементи міняються місцями. В результаті одного та-

кого перегляду, при сортуванні за збільшенням елементів, елемент з найбільшим значенням переміститься на останнє місце в масиві. При наступному проході на своє місце переміститься другий за величиною елемент і т.д. Для постановки на свої місця n елементів слід зробити $n-1$ проходів. При кожному черговому проході кількість елементів, що порівнюються, треба зменшувати на одиницю.

Нехай $b(n)$ – масив із n цілих чисел (n вводиться користувачем). Організуємо псевдо динамічний масив та відсортуємо його за зростанням елементів.

Фрагмент програми сортування за зростанням одномірного масиву стандартним обміном представлено в лістингу 1.

Лістинг 1.

```
. . .
//Цикл проходів по масиву до  $i$ -того елементу (до границі перег-
ляду)
for (i=n-1; i>1; i--)
{ // цикл попарного порівняння сусідніх елементів
  for (j =0; j<=i-1; j++)
    if (b[j]>b[j+1])
    { // обмін елементів з номерами  $j$  та  $j+1$ 
      int a = b[j];
      b[j] = b[j+1];
      b[j+1] = a;
    }
}
```

Способи модифікації сортування стандартним обміном

1) *Дотримання умови Айверсона.*

Якщо в ході сортування при порівнянні елементів не було зроблено ні однієї перестановки, то множина елементів даних вважається впорядкованою.

Фрагмент програми сортування за зростанням стандартним обміном з дотриманням умови Айверсона представлено в лістингу 2

Лістинг 2.

```
. . .
i=n-1;
while ((k!=0) && (i>1))
{
  k=0; // обмінів не було

  for (j =0; j<=i-1; j++)
    if (b[j]>b[j+1])
```

```

        {
            int a = b[j];
            b[j] = b[j+1];
            b[j+1] = a;
            k=1; // обмін відбувся
        }
        i--; // зменшення нижньої границі перегляду
    }
    . . .

```

2) Шейкерне сортування

Очевидний прийом поліпшення алгоритму стандартного обміну - запам'ятовувати, були або не були перестановки в процесі чергового проходу. Якщо в останньому проході перестановок не було, то сортування можна закінчувати. Це поліпшення можна знову ж поліпшити, якщо запам'ятовувати не тільки сам факт, що обмін мав місце, але й індекс останнього обміну. Ясно, що всі пари сусідніх елементів нижче цього індексу вже впорядковані. Тому перегляди можна закінчувати на цьому індексі, а не йти до заздалегідь певної нижньої границі.

Фрагмент програми шейкерного сортування за зростанням представлено в лістингу 3

Лістинг 3.

```

    . . .
    p=n-1;
    while ((k!=0) && (i>1))
    {
        k=0; i=p;
        for (j =0; j<=i-1; j++)
            if (b[j]>b[j+1])
            {
                int a = b[j];
                b[j] = b[j+1];
                b[j+1] = a;
                k=1; p=j+1;
            }
    }
    . . .

```

1.2 Сортування простими вставками

Одномірний масив $b[]$ із n цілих чисел відсортуємо за зростанням значень елементів методом вибору. Ідея алгоритму: для кожного елемента масиву $b[i]$ вважаємо,

що попередні елементи вже відсортовані. Елемент $b[i]$ потрібно вставити на відповідне місце в вже відсортовану послідовність $b[0] \dots b[i-1]$. Це місце визначається послідовним порівнянням $b[i]$ з вже впорядкованими елементами i , якщо необхідно - елементи міняються місцями.

Нехай $b(n)$ – масив із n цілих чисел (n вводиться користувачем). Організуємо псевдо динамічний масив та відсортуємо його за зростанням елементів.

Фрагмент програми сортування за зростанням одномірного масиву вставками представлено в лістингу 4.

Лістинг 4.

```
// цикл порівняння b[i]-го елемента з усіма попередніми
for (i=1; i<n; i++)
{
    for (j =0; j<=i-1; j++)
        if (b[i]<b[j])
        {
            // обмін місцями елемента b[j]-го з b[i]-м
            int a = b[j];
            b[j] = b[i];
            b[i] = a;
        }
}
```

1.3 Сортування методом вибору.

Одномірний масив із n цілих чисел відсортуємо за зростанням значень елементів методом вибору. Алгоритм полягає в тому, що вибирається найменший елемент і міняється місцями з першим елементом масиву, потім розглядаються елементи масиву, починаючи із другого, і найменший з них міняється місцями із другим елементом, і так далі $n-1$ раз (при останньому проході циклу при необхідності міняються місцями передостанній й останній елементи масиву).

Нехай $b(n)$ – масив із n цілих чисел (n вводиться користувачем). Організуємо псевдо динамічний масив та відсортуємо його за зростанням елементів.

Фрагмент програми сортування за зростанням одномірного масиву методом вибору представлено в лістингу 5.

Лістинг 1.6

```
. . .
for (i=0; i<n-1; i++) // n-1 раз шукаємо найменший елемент
{
    // приймаємо за найменший перший з розглянутих елементів
    min = b[i]; jmin=i; // та запам'ятовуємо його номер
```

```

// пошук мінімального елемента з неупорядкованих:
for (j = i + 1; j < n; j++)
    if (b[j] < min)
        // якщо знайшли менший елемент, запам'ятовуємо його та його
номер
        { min=b[j]; jmin = j; }
// обмін елементів з номерами i та jmin
int a = b[i];
b[i] = b[jmin];
b[jmin] = a;
}

```

Питання та завдання до контролю знань студентів

Питання для узагальнення та перевірки засвоєного матеріалу на лекції

- 1 Яка ідея полягає в основі алгоритму сортування обмінами?
- 2 Опишіть алгоритм сортування методом вибору?
- 3 Опишіть алгоритм сортування вставками?
- 4 Скільки проходів по масиву відбувається при сортуванні обмінами?
вибором? вставками?
- 5 На вхід подається відсортований масив. Як вдосконалити метод об-
мінами так, щоб сортування припинялось вже після першого проходу по ма-
сиву?

Завдання для самоперевірки засвоєного матеріалу на лекції

- 1 В заданому одномірному масиві дійсних чисел відсортувати за зро-
станням тільки непарні елементи використовуючи метод вставками.
- 2 Написати програму, що методом прямого вибору сортує за спадан-
ням введений із клавіатури одномірний масив.
- 3 Написати програму, що методом обміну («бульбашковим» мето-
дом) сортує за спаданням введений із клавіатури одномірний масив.
- 4 Введені з клавіатури два масиви дійсних чисел впорядкувати за зро-
станням. Об'єднати два впорядковані масиви в один, теж впорядкований масив.

Лекція № 11

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Сортування Шелла. Швидке сортування Хоара.
Мета заняття	Ознайомити студентів з основними методами швидкого сортування в лінійних структурах даних.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Сортування Шелла.
- 2 Швидке сортування (quick sort) Хоара

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Сортування Шелла.

Сортування Шелла називається так на ім'я свого автора, Дональда Л. Шелла (відкрите в 1959 р.). Проте ця назва закріпилася тому, що дія цього методу часто ілюструється рядами морських раковин, які перекривають одна одну (по-англійськи «shell» - «раковина»). Загальна ідея запозичена з сортування вставками і ґрунтується на зменшенні кроків (крок – це відстань між сортованими елементами на конкретному етапі сортування).

Ідея методу полягає в порівнянні розділених на групи елементів послідовності, що перебувають друг від друга на деякій відстані. Ця відстань дорівнює $d = N/2$, де N - загальне число елементів. На першому кроці кожна група містить у собі два елементи розташованих друг від друга на відстані $N/2$; вони рівняються між собою, і, якщо буде потреба, міняються місцями. На наступних кроках також відбуваються перевірка й обмін, але відстань d скорочується на $d/2$, і кількість груп, відповідно, зменшується. Поступова відстань між елементами зменшується, і на $d=1$ прохід по масиву відбувається востаннє.

Розглянемо приклад (рисунок 1). Спочатку сортуються всі елементи, віддалені один від одного на три позиції. Потім сортуються елементи, розташовані на відстані двох позицій. Нарешті, сортуються всі сусідні елементи.

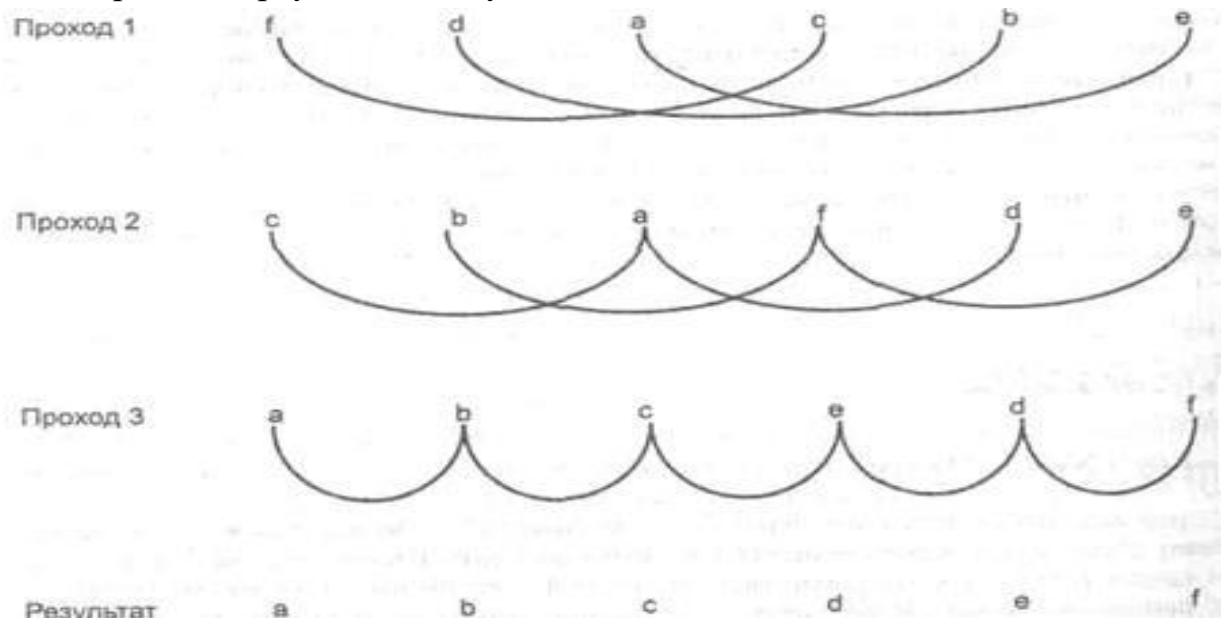


Рис.1 Сортування Шелла

Те, що цей метод дає гарні результати, або навіть те, що він взагалі сортує масив, побачити не так просто. Проте, це вірно. Кожен прохід сортування розповсюджується на відносно невелику кількість елементів або на елементи, розташовані вже у відносному порядку. Тому сортування Шелла ефективне, а кожен прохід підвищує впорядкованість, тобто зменшує кількість безладів (інверсій).

Конкретна послідовність кроків може бути і інша. Єдине правило полягає в тому, щоб останній крок був рівний 1. Наприклад, така послідовність: 9, 5, 3, 2, 1 дає ефективні результати і застосовується у показаній тут реалізації сортування Шелла. Слід уникати послідовностей, які є ступенями числа 2 – по математично складних міркуваннях вони зменшують ефективність сортування (але сортування буде працювати).

Код програми на C++

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int i, j, n, d, count;
void Shell(int A[], int n) //сортування Шелла
{
    d=n;
    d=d/2;
    while (d>0)
    {
        for (i=0; i<n-d; i++)
        {
            j=i;
            while (j>=0 && A[j]>A[j+d])
            {
                count=A[j];
                A[j]=A[j+d];
                A[j+d]=count;
                j--;
            }
        }
        d=d/2;
    }
    for (i=0; i<n; i++) cout<<A[i]<<" "; //виведення масиву
}
//головна функція
void main()
{
    setlocale(LC_ALL, "Rus");
    cout<<"Размер массива > "; cin>>n;
    int *A= new int[n]; //створення динамічного масиву
    for (i=0; i<n; i++) //введення масиву
    { cout<<i+1<<" элемент > "; cin>>A[i]; }
    cout<<"\nРезультирующий массив: ";
```

```
Shell(A, n);  
delete [] A; //визволення пам'яті  
system("pause>>void");  
}
```

Аналіз сортування Шелла пов'язаний з дуже складними математичними задачами. Час сортування пропорційний $n^1,2$ при сортуванні n елементів. А це вже істотне поліпшення в порівнянні з сортуваннями порядку n^2 . Проте, алгоритм швидкого сортування ще ефективніший.

2 Швидке сортування (quick sort) Хоара

Швидке сортування, придумане Ч. А. Р. Хоаром (Charles Antony Richard Hoare) і названо його ім'ям, вважається кращим з існуючих в даний час алгоритмів сортування загального призначення. У його основі лежить сортування обмінами.

Відмінною рисою швидкого сортування є операція розбивки масиву на дві частини щодо опорного елемента. Наприклад, якщо послідовність потрібно впорядкувати по зростанню, то в ліву частину будуть поміщені всі елементи, значення яких менше значення опорного елемента, а в праву елементи, чиї значення більше або рівні опорному.

Поза залежністю від того, який елемент обраний у якості опорного, масив буде відсортований, але все-таки найбільш удалим вважається ситуація, коли по обидва боки від опорного елемента виявляється приблизно рівна кількість елементів. Якщо довжина якоїсь із частин, що вийшли в результаті розбивки, перевищує один елемент, то для неї потрібно рекурсивно виконати впорядкування, тобто повторно запустити алгоритм на кожному з відрізків.

Таким чином, алгоритм швидкого сортування містить у собі два основних етапи:

- розбивка масиву щодо опорного елемента;
- рекурсивне сортування кожної частини масиву.

Розбивка масиву.

Ще раз про опорний елемент. Його вибір не впливає на результат, і тому може впасти на довільний елемент. Проте, як було відмічено вище, найбільша ефективність алгоритму досягається при виборі опорного елемента, що ділить послідовність на рівні або приблизно рівні частини. Але, як правило, через недостачу інформації не представляється можливості напевно визначити такий елемент, тому найчастіше доводиться вибирати опорний елемент випадковим образом.

У наступних п'ятих пунктах описана загальна схема розбивки масиву (сортування по зростанню):

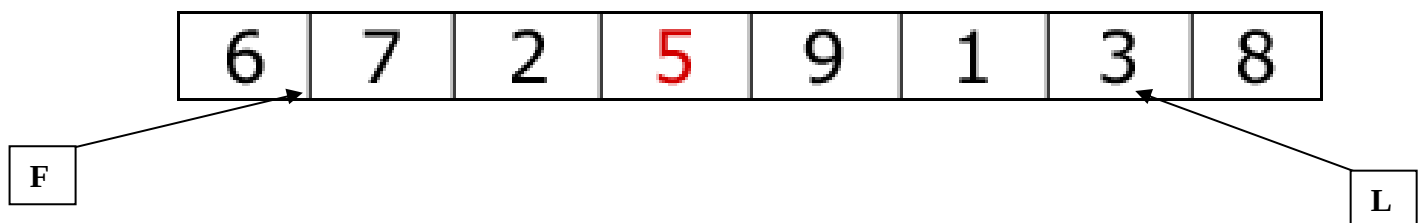
- 1 уводяться покажчики first й last для позначення початкового й кінцевого елементів послідовності, а також опорний елемент mid;
- 2 обчислюється значення опорного елемента $(first+last)/2$, і заноситися в змінну mid;
- 3 покажчик first зміщується із кроком в 1 елемент до кінця масиву доти, поки $Mas[first]>mid$. А покажчик last зміщується від кінця масиву до його початку, поки $Mas[last]<mid$;
- 4 кожні два знайдені елементи міняються місцями;
- 5 пункти 3 й 4 виконуються доти, поки $first<last$.

Після розбивки послідовності варто перевірити умова на необхідність подальшого продовження сортування його частин. Цей етап буде розглянутий пізніше, а зараз на конкретному прикладі виконаємо розбивку масиву.

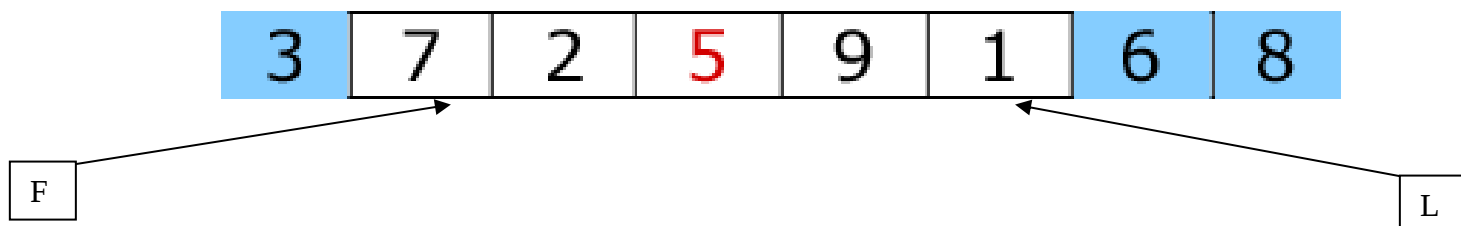
Є масив цілих чисел Mas, що складає з 8 елементів (рисунок 2): Mas[8]. Початковим значенням first буде 0, а last - 7. Пройдена частина зафарбовується блакитним кольором.

6	7	2	5	9	1	3	8
---	---	---	---	---	---	---	---

Як опорний елемент візьмемо елемент зі значенням 5, і індексом 4. Його ми обчислили, використовуючи вираження $(first+last)/2$, відкинувши дробову частину. Тепер $mid=5$.



Перший елемент лівої частини рівняється з mid. $Mas[0]>mid$, отже first залишається рівним 1. Далі, елементи правої частини рівняються з mid. Перевіряється елемент із індексом 7 і значенням 8. $Mas[7]>mid$, отже last зміщується на одну позицію вліво. $Mas[6]<mid$, отже last залишається рівним 6. На даний момент $first=0$, а $last=6$. Нульовий і шостий елементи міняються місцями. Обидва покажчики зміщуються на одну позицію кожний у своєму напрямку.



Алгоритм знову переходить до порівняння елементів. Другий елемент рівняється з опорним: $Mas[1]>mid$, отже first залишається рівним 1. Далі, елементи правої частини рівняються з mid. Перевіряється елемент із індексом 5 і значенням 1:

$Mas[5] < mid$, отже $last$ не змінює своєї позиції. На даний момент $first=1$, а $last=5$. Перший і п'ятий елементи міняються місцями. Обидва покажчики зміщаються на одну позицію кожний у своєму напрямку.

3	1	2	5	9	7	6	8
---	---	---	---	---	---	---	---

Алгоритм знову переходить до порівняння елементів. Другий елемент рівняється з опорним: $Mas[2] < mid$, отже $first$ зміщається на одну позицію вправо. Далі, елементи правої частини рівняються з mid . Перевіряється елемент із індексом 4 і значенням 9: $Mas[4] > mid$, отже $last$ зміщається на одну позицію вліво. Тепер $first=last=3$, а виходить, умова $first < last$ не виконується, етап розбивки завершується.

3	1	2	5	9	7	6	8
---	---	---	---	---	---	---	---

На цьому етап розбивки закінчений. Масив розділений на дві частини щодо опорного елемента. Залишилося зробити рекурсивне впорядкування його частин.

Рекурсивне впорядкування

Якщо в якийсь із частин, що залишилися в результаті розбивки масиву перебуває більше одного елемента, то варто зробити рекурсивне впорядкування цієї частини, тобто виконати над нею операцію розбивки, описану вище. Для перевірки умови "кількість елементів > 1 ", потрібно діяти приблизно за наступною схемою:

Є масив $Mas[L..R]$, де L й R - індекси крайніх елементів цього масиву. По закінченню розбивки, покажчики $first$ й $last$ виявилися приблизно в середині послідовності, тим самим образуя два відрізки: лівий від L до $last$ і правий від $first$ до R . Виконати рекурсивне упорядочивання лівої частини потрібно в тому випадку, якщо виконується умова $L < last$. Для правої частини умова аналогічно: $first < R$.

Реалізації алгоритму швидкого сортування:

Код програми на C++:

```
#include "stdafx.h"
#include <iostream>
#include <ctime>
using namespace std;
const int n=7;
int first, last;
//функція сортування
void quicksort(int *mas, int first, int last)
{
```

```

int mid, count;
int f=first, l=last;
mid=mas[(f+l) / 2]; //обчислення опорного елементу
do
{
while (mas[f]<mid) f++;
while (mas[l]>mid) l--;
if (f<=l) //перестановка елементів
{
count=mas[f];
mas[f]=mas[l];
mas[l]=count;
f++;
l--;
}
} while (f<l);
if (first<l) quicksort(mas, first, l);
if (f<last) quicksort(mas, f, last);
}
//головна функція
void main()
{
setlocale(LC_ALL, "Rus");
int *A=new int[n];
srand(time(NULL));
cout<<"Исходный массив: ";
for (int i=0; i<n; i++)
{
A[i]=rand()%10;
cout<<A[i]<<" ";
}
first=0; last=n-1;
quicksort(A, first, last);
cout<<endl<<"Результирующий массив: ";
for (int i=0; i<n; i++) cout<<A[i]<<" ";
delete []A;
system("pause>>void");
}

```

Варто відзначити, що швидке сортування може виявитися малоефективною на масивах, що складаються з невеликого числа елементів, тому при роботі з ними розумніше відмовитися від даного методу. У цілому алгоритм нестійкий, а також використання рекурсії в невірному складеному коді може привести до переповнення стека. Але, незважаючи на ці й деякі інші мінуси, швидке сортування все-таки є одним з найефективніших і часто використовуваних методів.

Питання та завдання до контролю знань студентів

Питання для узагальнення та перевірки засвоєного матеріалу на лекції

1. Які методи сортування масивів ви знаєте?
2. Розкрийте сутність бульбашкового методу сортування. Наведіть приклад.
3. Розкрийте сутність методу сортування за допомогою вибору. Наведіть приклад.
4. Розкрийте сутність методу сортування вставками. Наведіть приклад.
5. Розкрийте сутність методу сортування Шелла. Наведіть приклад.
6. Розкрийте сутність методу швидкого сортування. Наведіть приклад.

Завдання для самоперевірки засвоєного матеріалу на лекції

- 5 В заданому одномірному масиві дійсних чисел відсортувати за зростанням тільки непарні елементи використовуючи метод вставками.
- 6 Написати програму, що методом прямого вибору сортує за спаданням введений із клавіатури одномірний масив.
- 7 Написати програму, що методом обміну («бульбашковим» методом) сортує за спаданням введений із клавіатури одномірний масив.
- 8 Введені з клавіатури два масиви дійсних чисел впорядкувати за зростанням. Об'єднати два впорядковані масиви в один, теж впорядкований масив.

Лекція № 12

з навчальної дисципліни «Алгоритми та структури даних»

Тема Методи пошуку в лінійних структурах даних.

Мета заняття Сформулювати постановку задачі пошуку. Дати опис основних алгоритмів пошуку в лінійних структурах даних.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Постановка задачі пошуку
- 2 Методи пошуку
 - 2.1 Послідовний пошук
 - 2.2 Бінарний пошук
 - 2.3 Пошук Фібоначчі
 - 2.4 Інтерполяційний пошук

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник. [Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Постановка задачі пошуку

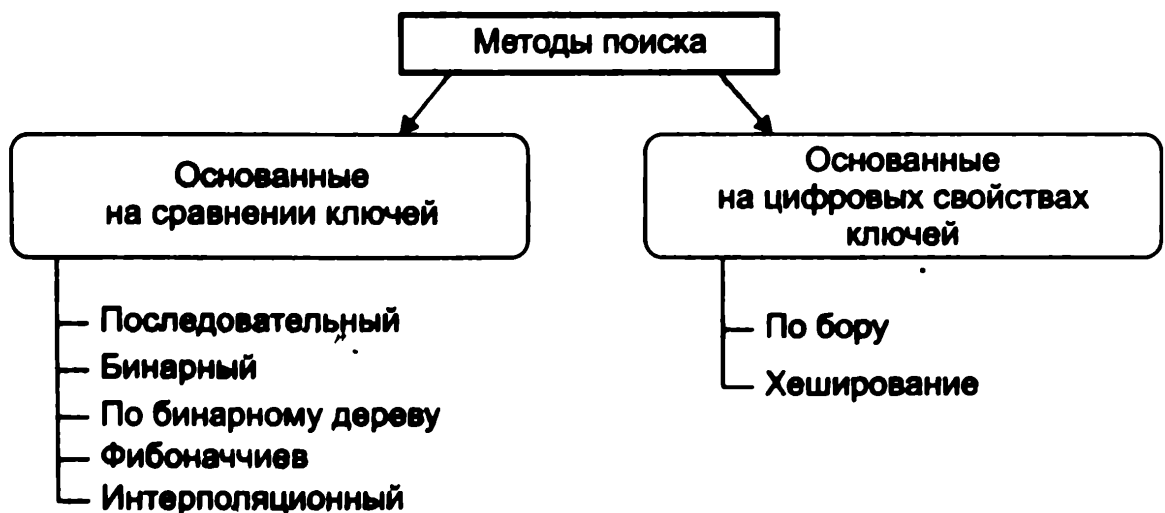
Предмети (об'єкти), що становлять множину, називаються його елементами. Елемент множини називатиметься ключем і позначатися латинською буквою k_i з індексом, що вказує номер елемента.

Задача пошуку полягає в наступному: Нехай задана множина ключів $\{k_1, k_2, \dots, k_n\}$. Необхідно відшукати в множині ключ k . Пошук може бути завершений в двох випадках:

- ключ в множині відсутній;
- ключ знайдений в множині.

2 Методи пошуку

Алгоритми пошуку можна розбити на наступні групи:



2.1 Послідовний пошук

У послідовному пошуку початкова множина не впорядкована, тобто є довільний набір ключів $\{k_1, k_2, \dots, k_n\}$. Метод полягає в тому, що відшукуваний ключ k_i послідовно порівнюється з усіма елементами множини. При цьому пошук закінчується достроково, якщо ключ знайдений.

Алгоритм пошуку зводиться до послідовності кроків :

Крок 1. [Початкова установка.] Встановити $i := 1$.

Крок 2. [Порівняння.] Якщо $k = k_i$ алгоритм закінчується вдало.

Крок 3. [Просування.] Збільшити i на 1.

Крок 54. [Кінець файлу?] Якщо $i < N$, то повернутися до кроку 2. Інакше алгоритм закінчується невдало.

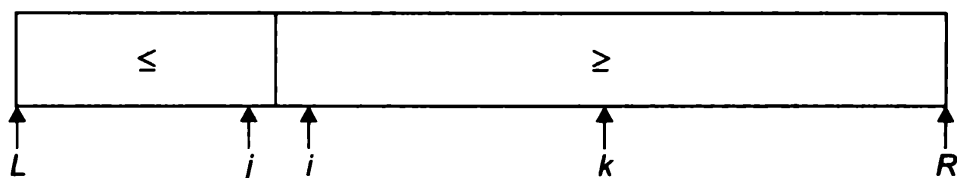
2.2 Бінарний пошук

У бінарному пошуку висхідна множина має бути впорядкована за збільшенням, іншими словами, кожен наступний ключ більший за попередній, тобто $k_1 \leq k_2 \leq \dots \leq k_{n-1} \leq k_n$.

Відшукуваний ключ порівнюється з центральним елементом множини, якщо він менший за центральне, то пошук триває в лівій підмножині, інакше - в правому. Медіаною послідовності з n елементів вважається елемент, значення якого менше (чи рівно) половини n елементів і більше (чи рівно) іншої половини. Завдання пошуку медіани прийнято зв'язувати з сортуванням, оскільки медіану завжди можна знайти наступним способом: відсортувати n елементів і потім вибрати середній елемент.

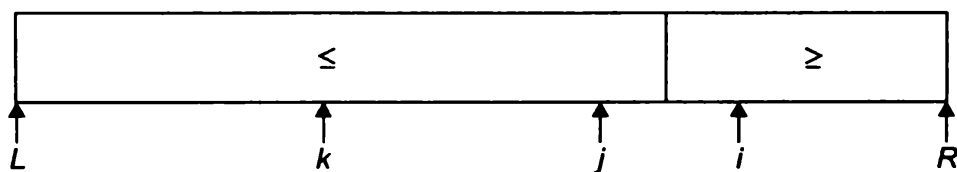
У алгоритмі Хоара для знаходження медіани використовується операція розподілу, вживана при швидкому сортуванні, з $L = 1$, $R = n$ і з $a[k]$, вибраними як розділяюче значення x . Виходять значення індексів i та j :

1) розділяюче значення x було занадто мало; в результаті межа між двома частинами нижче за шукане значення k . Процес розподілу слід повторити для елементів $a[i], \dots, a[R]$;



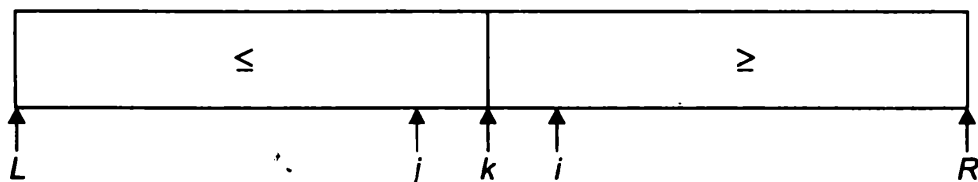
Зміщення межі вліво

2) вибрана межа x була занадто велика. Операцію розбиття слід повторити на підмасиві $a[L], \dots, a[j]$;



Зміщення межі управо

3) значення k : лежить в інтервалі $j < k < i$: елемент $a[k]$ розділяє масив в заданій пропорції і, отже, є шуканим.



Межа визначена точно

Процес розбиття повторюється до появи останнього випадку.

Центральний елемент знаходиться по формулі Нел-та $= \lfloor n/2 \rfloor + 1$, де квадратні дужки означають, що від ділення береться тільки ціла частина, дробова частина відкидається. У методі бінарного пошуку аналізуються тільки центральні елементи підмножин.

Приклад 1. В множині елементів відшукати ключ, рівний 653. У квадратних дужках виділені множини аналізованих елементів. Центральні елементи підмножин підкреслені.

Пошук ключа $K = 653$ здійснюється за чотири кроки:
 [061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]
 0061 087 154 170 275 426 503 509 [512 612 653 677 703 765 897 908]
 0061 087 154 170 275 426 503 509 [512 612 653] 677 703 765 897 908
 0061 087 154 170 275 426 503 509 512 612 [653] 677 703 765 897 908

Приклад 2. Дана впорядкована множина елементів

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69,
 75, 79, 80, 81, 95, 101, 123, 198}.

Знайти в множині ключ $K = 61$.

Крок 1. Нал-та = $\lfloor n/2 \rfloor + 1 = \lfloor 22/2 \rfloor + 1 = 12$;

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69,
 75, 79, 80, 81, 95, 101, 123, 198};

$K \vee K_{12}$ (значок "V" означає порівняння елементів: чисел, значень). Оскільки $61 > 60$, подальший пошук виконуємо в правій підмножині.

Крок 2. Нал-та = $\lfloor n/2 \rfloor + 1 = \lfloor 10/2 \rfloor + 1 = 6$;

{61, 69, 75, 79, 80, 81, 95, 101, 123, 198};

$K \vee K_{18}$. Оскільки $61 < 81$, подальший пошук виконуємо в лівій підмножині.

Крок 3. Нал-та = $\lfloor n/2 \rfloor + 1 = \lfloor 5/2 \rfloor + 1 = 3$;

{61, 69, 75, 79, 80};

$K \vee K_{15}$. Оскільки $61 < 75$, подальший пошук виконуємо в лівій підмножині.

Крок 4. Нал-та = $\lfloor n/2 \rfloor + 1 = \lfloor 2/2 \rfloor + 1 = 2$;

{61, 69};

$K \vee K_{14}$. Оскільки $61 < 69$, подальший пошук виконуємо в лівій підмножині.

Крок 5. {61}. $K \vee K_{13}$. Оскільки $61 = 61$, робимо висновок: шуканий ключ знайдений під номером 13

2.3 Пошук Фібоначчі

У цьому пошуку аналізуються елементи, що знаходяться в позиціях, рівних числам Фібоначчі. Числа Фібоначчі виходять за наступним правилом: кожне наступне число дорівнює сумі двох попередніх чисел, наприклад:

{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..}.

Пошук триває до тих пір, поки не буде знайдений інтервал між двома ключами, де може розташовуватися відшукуваний ключ. Фібоначчів пошук призначається для пошуку аргументу K серед розташованих в порядку зростання ключів $K_1 < K_2 < \dots < K_n$

Для зручності опису передбачається, що $n + 1$ є число Фібоначчі F_{n+1} . Відповідною початковою установкою цей метод можна зробити придатним для будь-якого n .

F₁. [Початкова установка.] Встановити $i := F_k$, $r := F_{k-1}$, $q := F_{k-2}$ - (В алгоритмі r і q означають послідовні числа Фібоначчі.)

F₂. [Порівняння.] Якщо $K < K_i$, то на F₃; якщо $K > K_i$, то перейти на F₄, якщо $K = K_i$, алгоритм закінчується вдало.

F₃. [Зменшення і.] Якщо $q = 0$, алгоритм закінчується невдало. Якщо $q \neq 0$, то встановити $i := i - q$, замінити (r, q) на $(q, r - q)$ і повернутися на F₂.

F₄. [Збільшення і.] Якщо $r = 1$, алгоритм закінчується невдало. Якщо $r \neq 1$, встановити $i := i + q$, $r := r - q$, $q := q - r$ і повернутися на F₂.

Приклад. Дана початкова множина ключів

{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}.

Нехай відшукуваний ключ дорівнює 42 ($K = 42$). Послідовне порівняння відшукуваного ключа проводитиметься з елементами початкової множини, розташованими в позиціях, рівних числам Фібоначчі, : 1, 2, 3, 5, 8, 13, 21,...

Крок 1. $K \sim k_1$, $42 > 3$, відшукуваний ключ порівнюється з ключем, що стоїть в позиції, рівній числу Фібоначчі.

Крок 2. $K \sim k_2$, $42 > 5$, порівняння триває з ключем, що стоїть в позиції, рівній наступному числу Фібоначчі.

Крок 3. $K \sim k_3$, $42 > 8$, порівняння триває.

Крок 4. $K \sim k_5$, $42 > 11$, порівняння триває.

Шаг 5. $K \sim k_8$, $42 > 19$, порівняння триває.

Крок 6. $K \sim k_{13}$, $42 > 35$, порівняння триває.

Крок 7. $K \sim k_{18}$, $42 < 52$, знайдений інтервал, в якому знаходиться відшукуваний ключ, тобто відшукуваний ключ може знаходитися в початковій множині між позиціями 13 і 18, тобто {35, 37, 42, 45, 48, 52}.

У знайденому інтервалі пошук знову ведеться в позиціях, рівних числам Фібоначчі.

2.4 Інтерполяційний пошук

Початкова множина ключів має бути впорядкована за збільшенням значень. Первинне порівняння здійснюється на відстані кроку d , який визначається по формулі:

$$d = \left\lfloor \frac{(j - i)(K - K_i)}{K_j - K_i} \right\rfloor,$$

Де i - номер першого елемента;

j - номер останнього елемента;

K - відшукуваний ключ;

K_i, K_j - значення ключів в позиціях i та j ;

$\lfloor \rfloor$ - ціла частина числа.

Ідея методу полягає в наступному: крок d міняється після кожного етапу по формулі, приведеній вище.

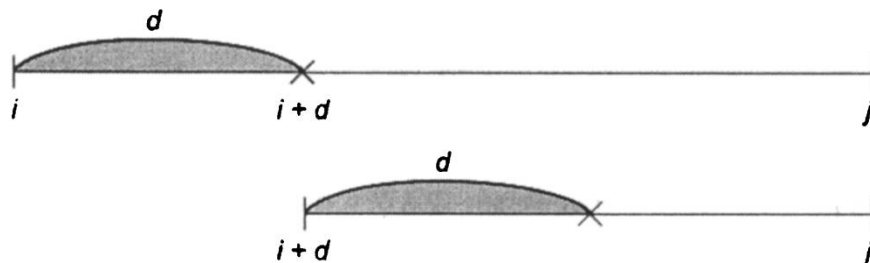


Схема інтерполяційного пошуку

Алгоритм закінчує роботу при $d = 0$, при цьому аналізуються сусідні елементи, після чого приймається остаточне рішення про результати пошуку. Цей метод прекрасно працює, якщо початкова множина є арифметичною прогресією або множиною, наближеною до неї.

Приклад 1. Дана множина ключів :

{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76}.

Нехай шуканий ключ $K = 70$.

Визначимо крок d для початкової безлічі ключів ($i = 1; j = 18$) :

$$d = \left\lceil \frac{(18 - 1)(70 - 2)}{76 - 2} \right\rceil = 15.$$

Порівнюємо ключ, що стоїть під шістнадцятим порядковим номером в цій множині з шуканим ключем: $k_{16} \vee K$; $70 = 70$; ключ знайдений.

Приклад 2. Дана безліч ключів :

{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95, 98, 102}.

Вимагається відшукати ключ $K = 90$.

Шаг 1. Визначимо крок d для початкової безлічі ключів ($i = 1; j = 17$):

$$d = \left\lceil \frac{(17 - 1)(90 - 4)}{102 - 4} \right\rceil = 14.$$

Порівнюємо ключ, що стоїть під п'ятнадцятим порядковим номером, з відшуканим ключем: $k_{15} \vee K$, $95 > 90$, отже, звужується зона пошуку :

{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95}.

Шаг 2. Визначимо крок d для безлічі ключів ($i = 1; j = 15$) :

$$d = \left\lceil \frac{(15 - 1)(90 - 4)}{95 - 4} \right\rceil = 13.$$

Порівнюємо ключ, що стоїть під чотирнадцятим порядковим номером, з відшуканим ключем: $k_{14} \vee K$; $k_{14} = K$; $90 = 90$; ключ знайдений.

Питання та завдання до контролю знань студентів

- 1 Що розуміється під пошуком?
- 2 Які особливості послідовного й бінарного пошуку?
- 3 Які особливості послідовного пошуку?
- 4 Які особливості інтерполяційного пошуку?
- 5 Які особливості пошуку Фібоначчі?

Лекція № 13

з навчальної дисципліни «Алгоритми та структури даних»

Тема Бінарні дерева пошуку

Мета заняття Ввести поняття бінарного дерева пошуку. Описати алгоритми побудови бінарного дерева та пошуку ключового елементу по бінарному дереву.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Поняття бінарного дерева пошуку.
- 2 Алгоритм пошуку по бінарному дереву.

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник. [Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Поняття бінарного дерева пошуку

Бінарне дерево пошуку - це бінарне дерево, що володіє додатковими властивостями: значення лівого нащадку менше значення батька, а значення правого нащадку більше значення батька для кожного вузла дерева. Тобто, дані в бінарному дереві пошуку зберігаються у відсортованому виді. При кожній операції вставки нового або видалення існуючого вузла відсортований порядок дерева зберігається.

Рекурсивна функція створення бінарного дерева пошуку наведена в лістингу 1:

Лістинг 1: Функція створення незбалансованого бінарного дерева пошуку

```
node* enter_bale(node * bale, int y)
{
    node *work;
    int x;
    if (bale==NULL)
    {
        bale=new node;
        bale->info=y;
        bale->left=NULL;
        bale->right=NULL;
    }
    else
    {
        if ( y<bale->info) bale->left=enter_bale(bale->left,y);
        if (y>bale->info) bale->right=enter_bale( bale->right,y);
    }
    return bale;
}
```

Наприклад, якщо для неупорядкованого набору чисел

{7, 3, 5, 2, 8, 1, 6, 10, 9, 4, 11}

застосувати цю функцію, вийде бінарне дерево, зображене на рисунку 1. Для того, щоб правильно врахувати повторення чисел, можна ввести додаткове поле, що буде зберігати кількість входжень для кожного числа.

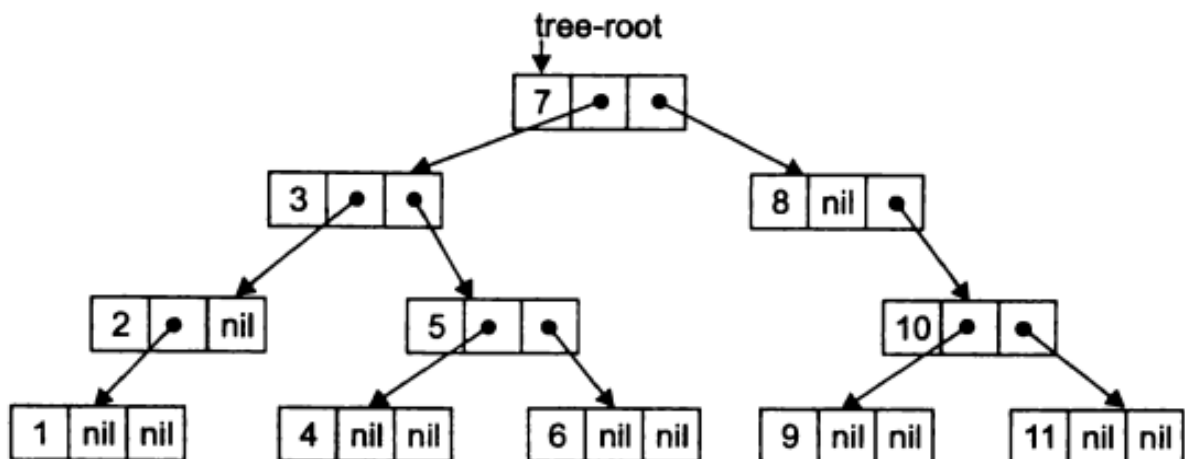


Рис.1. Бінарне дерево пошуку.

Скріншот застосування функції `enter_bale()` до заданої послідовності чисел наведено на рисунку 2.

```
D:\BC\BIN\BC.EXE

enter info new element 6
enter info new element 10
enter info new element 9
enter info new element 4
enter info new element 11

      11
     10
    9
   8
  7
 6
5
4
3
2
1
```

Рис. 2. Результат роботи функції `enter_bale()`

Відзначимо, що побудова бінарного дерева пошуку «впорядковує» за зростанням вхідний набір чисел. Якщо, застосовуючи прямий обхід дерева, вивести на екран вміст вузлів, отримаємо впорядковану за зростанням послідовність чисел. Скріншот застосування такої функції до побудованого бінарного дерева пошуку наведено на рисунку 3.

```
D:\BC\BIN\BC.EXE

      11
     10
    9
   8
  7
 6
5
4
3
2
1

1
2
3
4
5
6
7
8
9
10
11
```

Рис. 3. Прямий обхід дерева

2 Алгоритм пошуку по бінарному дереву

Спочатку аргумент(ключ пошуку) пошуку порівнюється із значенням, що перебуває в корені. Якщо ключ збігається із цим значенням, пошук закінчений, якщо не збігається, то у випадку, коли ключ виявляється менше значення вузла, пошук

триває в лівому піддереві, а у випадку, коли більше значення, - у правому піддереві. Збільшивши рівень на 1, повторюють порівняння, уважаючи поточний вузол коренем.

Рекурсивна функція пошука ключа в бінарному дереві наведена в лістингу 2.

Лістинг 2. Функція пошуку в бінарному дереві

```
void search(node *bale, int y)
{
    if (bale==NULL)
    {
        cout<<" Елемент відсутній "<<endl;return;
    }
    else
    {
        if (bale->info==y)
        {
            cout<<" Елемент знайдено "<<endl;return;
        }
        if (y<bale->info) search(bale->left,y);
        if (y>bale->info) search(bale->right,y);
    }
}
```

Приклад.

Нехай задана вхідна послідовність чисел

{21, 7, 32, 5, 14, 27, 37, 4, 6, 25, 30, 12, 18, 34, 39, 2, 9, 24, 33,}.

Від лінійного списку переходимо до побудови бінарного дерева пошуку (рисунок 4).

Нехай шуканий ключ $K = 24$.

Пошук ключа $K = 24$ по бінарному дереву від кореня до листів показано на рисунку 4.

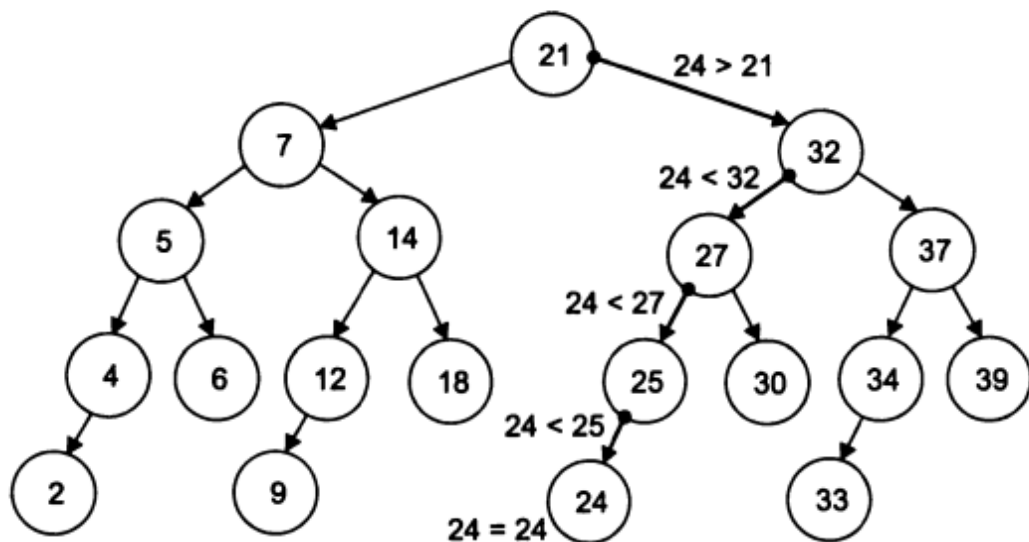


Рис.4. Бінарне дерево пошуку

Питання та завдання до контролю знань студентів

Питання для актуалізації опорних знань.

- 1 У чому складається основна відмінність деревоподібних структур від спискових?
- 2 Як рекурсивно визначається дерево?
- 3 Які типи вершин існують у деревах?
- 4 Які можна виділити типи дерев?
- 5 Які дерева називаються двійковими?
- 6 Які основні поняття зв'язуються з деревами?
- 7 Які основні операції характерні при використанні дерев?
- 8 Яку структуру мають вершини двійкового дерева?
- 9 Які правила обходу вершин дерева є основними?
- 10 Як виконується обхід дерева в прямому напрямку?
- 11 Як виконується обхід дерева в симетричному напрямку?
- 12 Як виконується обхід дерева у зворотному напрямку?

Питання для узагальнення та перевірки засвоєного матеріалу на лекції

- 1 Яке дерево називається деревом пошуку?
- 2 У чому складається практична важливість використання дерев пошуку?
- 3 Які переваги має використання дерев пошуку для зберігання впорядкованих даних у порівнянні з масивами й списками?
- 4 Приведіть алгоритм пошуку в дереві пошуку.
- 5 Як програмно реалізується пошук у дереві пошуку?

Лекція № 14

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Побудова збалансованих бінарних дерев пошуку
Мета заняття	Ввести поняття збалансованого бінарного дерева пошуку. Описати алгоритм побудови збалансованого бінарного дерева пошуку на основі операцій лівого та правого поворотів навколо вузла.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	
Конспект лекцій.	

Час – 2 години

План проведення лекції

- 1 Поняття збалансованого бінарного дерева пошуку (АВЛ-дерева).
- 2 Балансування вузлів
- 3 Вставка ключів

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Поняття збалансованого бінарного дерева пошуку(АВЛ-дерева)

АВЛ-дерево - це насамперед бінарне дерево пошуку, ключі якого задовольняють стандартній властивості: ключ будь-якого вузла дерева не менше будь-якого ключа в лівому піддереві даного вузла й не більше будь-якого ключа в правому піддереві цього вузла. Це значить, що для пошуку потрібного ключа в АВЛ-дереві можна використати стандартний алгоритм. Для простоти подальшого викладу будемо вважати, що всі ключі в дереві цілочисельні й не повторюються.

Бінарне дерево називається збалансованим(АВЛ-деревом), якщо висота лівого піддерева кожного вузла відрізняється від висоти правого не більше ніж на одиницю (рисунок 1).

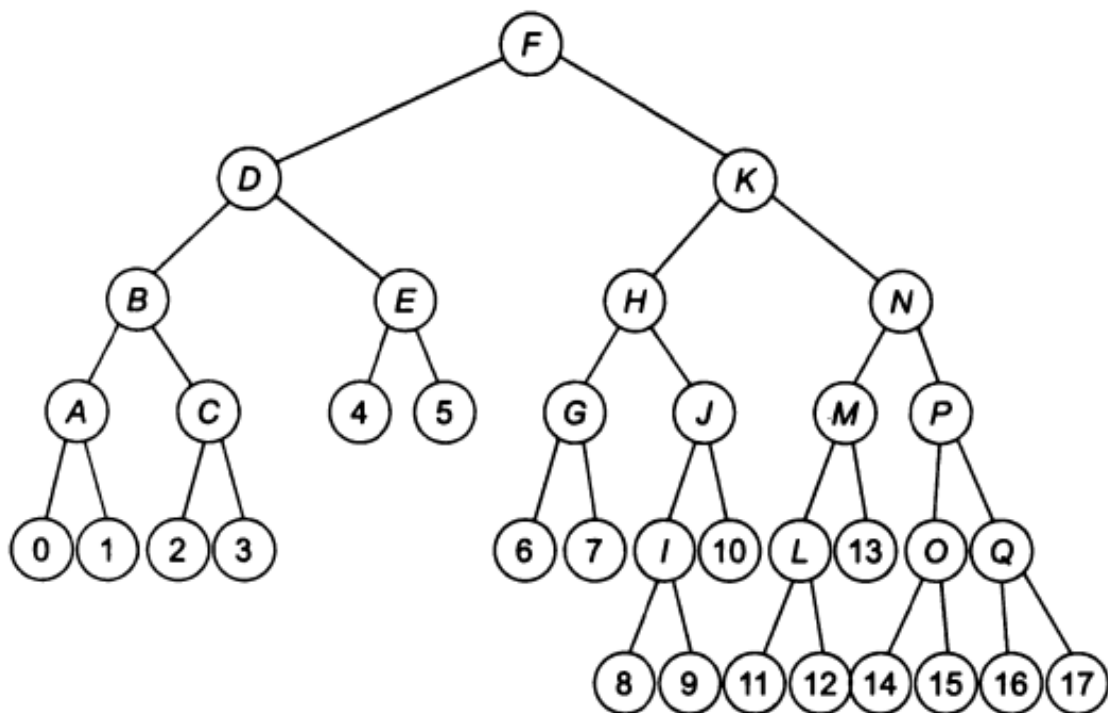


Рис.1. Приклад збалансованого бінарного дерева пошуку

Будемо представляти вузли АВЛ-дерева наступною структурою:

```
struct node
{
    int info;
    int h;
    node * left;
    node * right;
};
```

Поле `info` зберігає ключ вузла, поле `h` - висоту піддерева з коренем у даному вузлі, полючки `left` й `right` - покажчики на ліве й праве піддерева.

Традиційно, вузли АВЛ-дерева зберігають не висоту, а різницю висот правого й лівого піддерев (так званий `balance factor`), що може приймати тільки три значення -1, 0 й 1.

Визначимо три допоміжні функції, пов'язані з висотою.

Перша є обгорткою для поля h , вона може працювати й з нульовими показниками (з порожніми деревами). Вона повертає висоту дерева(піддерева):

```
int height(node* p)
{ return p->h; }
```

Друга обчислює balance factor заданого вузла (і працює тільки з ненульовими показниками):

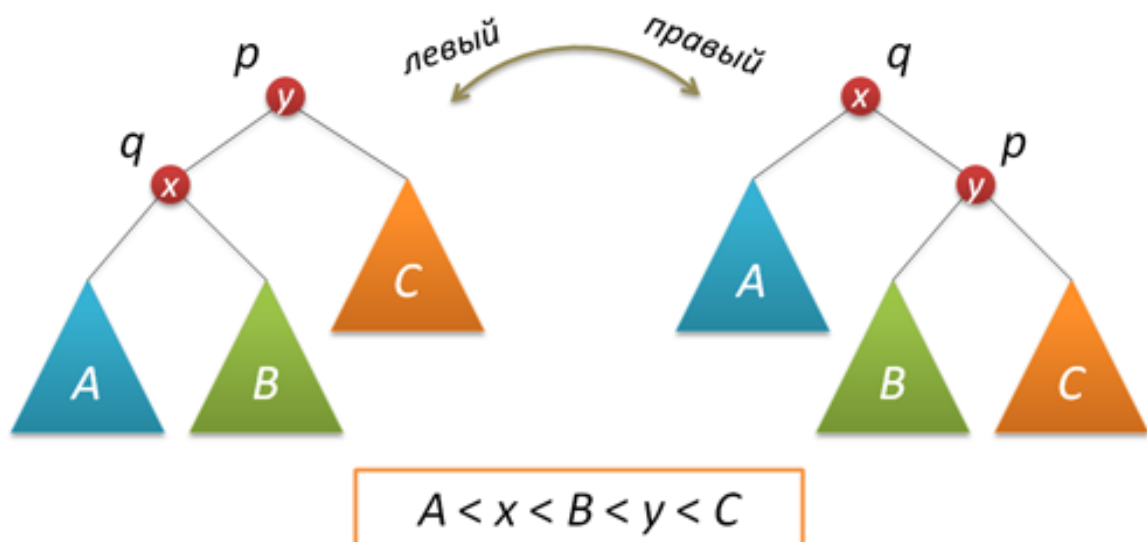
```
int bfactor (node* p)
{ return height(p->right)-height(p->left); }
```

Третя функція відновлює коректне значення поля h заданого вузла (за умови, що значення цього поля в правому і лівому дочірніх вузлах є коректними):

```
void fixheight (node* p)
{ int h1=height(p->left);
  int h2=height(p->right);
  if (h1>h2) p->h=h1+1;
  else p->h=h2+1; }
```

2 Балансування вузлів

У процесі додавання або видалення вузлів в AVL-дереві можливе виникнення ситуації, коли balance factor деяких вузлів виявляється рівними 2 або -2, тобто виникає розбалансування піддерева. Для виправлення ситуації застосовуються повороти навколо тих або інших вузлів дерева. **Простий поворот** вправо (уліво) робить наступну трансформацію дерева (рисунок 2):



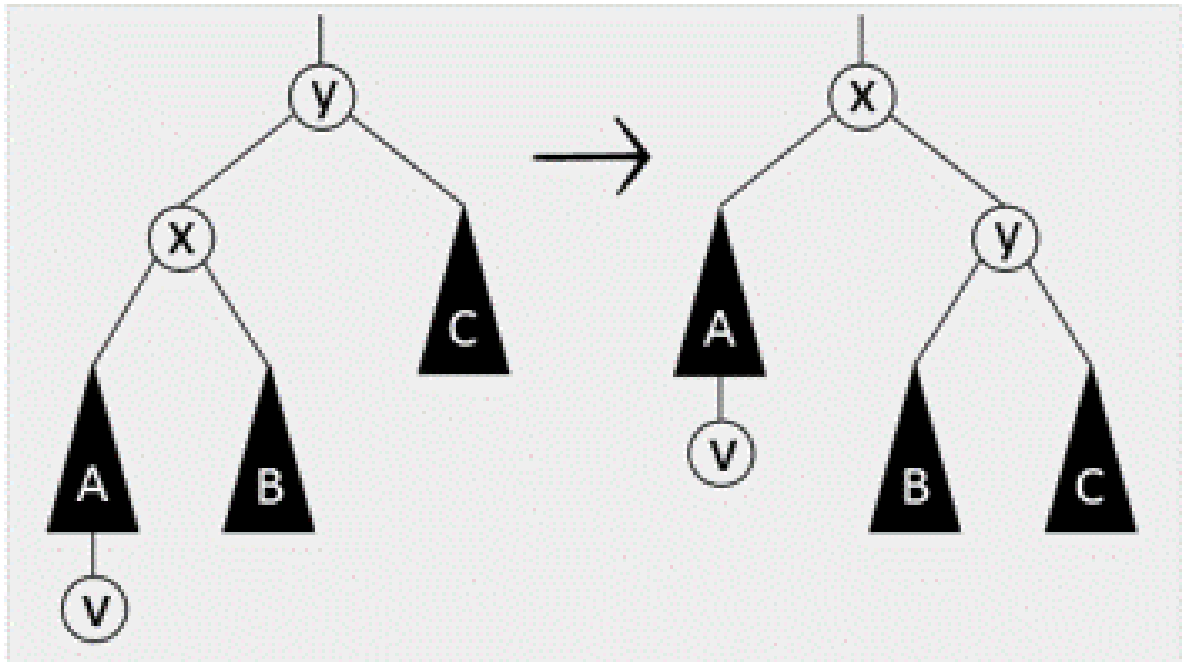


Рис. 2. Правий та лівий прості повороти.

Код, що реалізує правий поворот, виглядає в такий спосіб(лістинг 1). Як звичайно, кожна функція, що змінює дерево, повертає новий корінь отриманого дерева:

Лістинг 1. Реалізація правого простого повороту.

```
node* rotateright(node* p) // правий поворот навколо p
{
    node* q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
}
```

Лівий поворот є симетричною копією правого (лістинг 2):

Лістинг 2. Реалізація лівого просторо повороту.

```
node* rotateleft(node* q) // левый поворот вокруг q
{
    node* p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
}
```

Розглянемо тепер ситуацію дисбалансу, коли висота правого піддерева вузла p на 2 більше висоти лівого піддерева (зворотний випадок є симетричним і реалізується

аналогічно). Нехай q - правий дочірній вузол вузла p , а s - лівий дочірній вузол вузла q (рисунок 3).

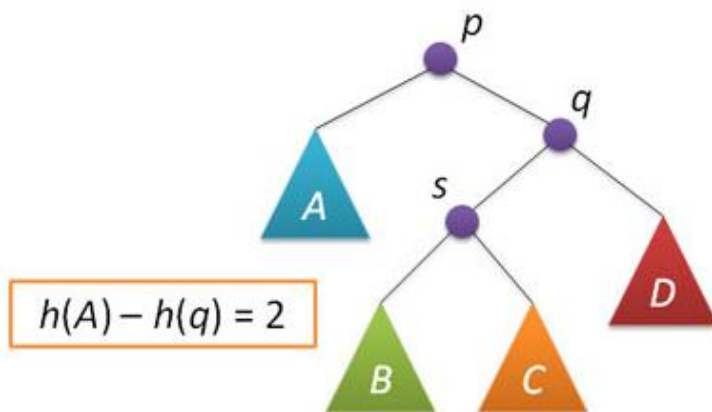


Рис. 3. Розбалансоване бінарне дерево пошуку.

Аналіз можливих випадків у рамках даної ситуації показує, що для виправлення розбалансування у вузлу p досить виконати або простий поворот уліво навколо p , або так званий великий поворот уліво навколо того ж p . Простий поворот виконується за умови, що висота лівого піддерева вузла q більше висоти його правого піддерева: $h(s) \leq h(D)$ (рисунок 4).

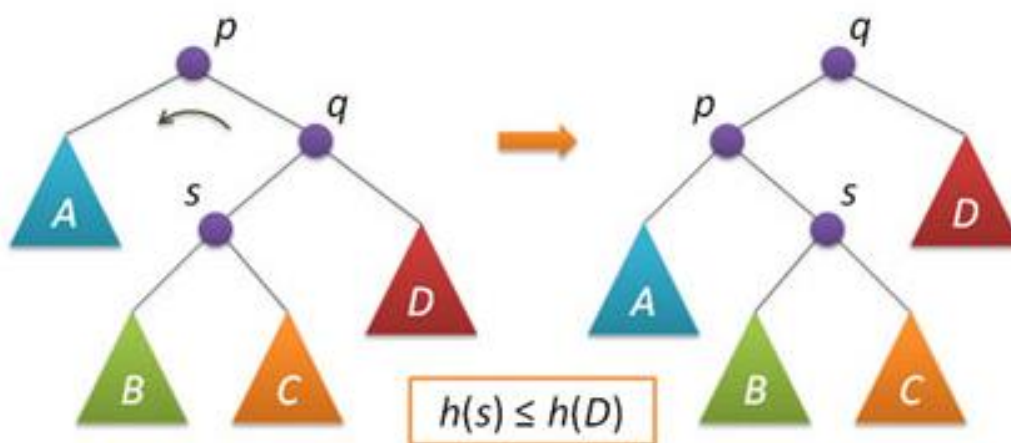


Рис. 4. Простий лівий поворот навколо вузла P .

Великий поворот застосовується за умови $h(s) > h(D)$ і зводиться в цьому випадку до двох простих поворотів - спочатку правий поворот навколо q і потім лівий навколо p (рисунок 5).

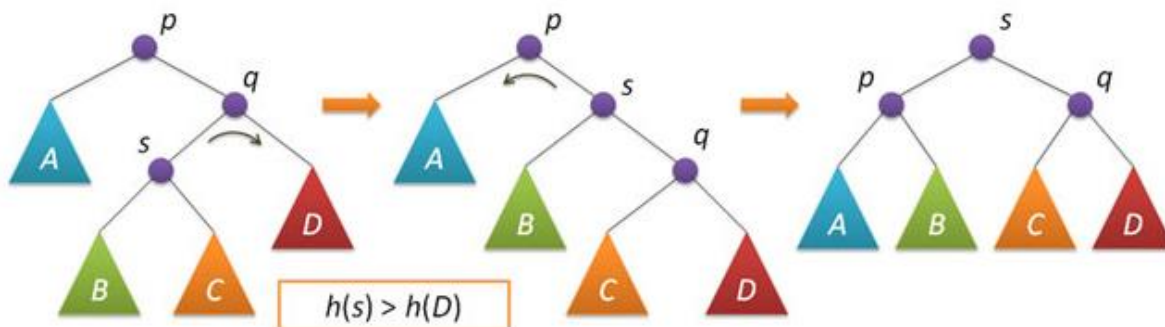


Рис. 5. Великий поворот навколо P .

Код, що виконує балансування, зводиться до перевірки умов і виконанню поворотів наведено у лістингу 3.

Лістинг 3. Функція балансування вузлу бінарного дерева пошуку.

```
node* balance(node* p) // балансування вузла p
{
    fixheight(p);
    if( bfactor(p)==2 )
    {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if( bfactor(p)==-2 )
    {
        if( bfactor(p->left) > 0 )
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p; // балансування не нужна
}
```

Описані функції поворотів і балансування також не містять ні циклів, ні рекурсії, а значить виконуються за постійний час, що не залежить від розміру AVL-дерева.

3 Вставка ключів

Вставка нового ключа в AVL-дерево виконується, по великому рахунку, так само, як це робиться в простих деревах пошуку: спускаємося вниз по дереву, вибираючи правий або лівий напрямок руху залежно від результату порівняння ключа в поточному вузлі й ключа, що вставляється. Єдина відмінність полягає в тім, що при поверненні з рекурсії (тобто після того, як ключ вставлений або в праве, або в ліве поддерево, і це дерево збалансоване) виконується балансування поточного вузла. Строго доводиться, що виникаючий при такій вставці дисбаланс у будь-якому вузлі по шляху руху не перевищує двох, а значить застосування вищеописаної функції балансування є коректним.

Код функції вставки нового вузла в збалансоване дерево пошуку наведено в лістингу 4.

Лістинг 4. Функція вставки нового вузла.

```
node* insert(node* p, int k) // вставка ключа k в дерево з корнем p
{
    if( p==NULL )
    {
        p=new node;
        p->info=k;
    }
}
```

```

        p->h=0;
        p->left=NULL;
        p->right=NULL;
    }
    else

    if( k<p->info )
        p->left = insert(p->left,k);
    else
        p->right = insert(p->right,k);
    return balance(p);
}

```

Питання та завдання до контролю знань студентів

- 1 Дайте визначення AVL-дерева.
- 2 Чи є AVL-дерево деревом пошуку?
- 3 Яка висота AVL-дерева?
- 4 Для чого потрібні повороти AVL-дерева?
- 5 Яка трудомісткість включення й виключення вершини AVL-дерева?

Лекція № 15

з навчальної дисципліни «Алгоритми та структури даних»

Тема Алгоритми пошуку словесної інформації.

Мета заняття Ознайоми студентів з основними алгоритмічними стратегіями пошуку текстової інформації.

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Алгоритм Кнута, Моріса, Пратта
- 2 Алгоритм Боуєра та Мура
- 3 Алгоритм Рабіна

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник. [Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1. Алгоритм Кнута, Моріса, Пратта

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 пересунути текст. Приведений приклад пошуку слова ABCABD показує принцип роботи такого алгоритму. Символи, які пройшли порівняння, – підкреслені. Зверніть увагу: при кожному незбігу пари символів слово зсувається на всю пройдену відстань, оскільки менші зсуви не можуть привести до повного збігу.

```
ABCABCABAABCABD
ABCABD
  ABCABD
    ABCABD
      ABCABD
        ABCABD
```

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

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

Так як величини d_j залежать лише від слова, то перед початком фактичного пошуку можна обчислити допоміжну таблицю d ; ці обчислення зводяться до деякої попередньої трансляції слова. Відповідні зусилля будуть оправдані, якщо розмір тексту значно перевищує розмір слова ($M \ll N$). Якщо потрібно шукати багатократні

входження одного й того ж слова, то можна користуватися одними й тими ж d .

Наведені приклади пояснюють функцію d .

Текст	A A A A C	$j=5; d[5]=4; \text{max_shift}=j-d[5]=1$
Слово	A A A A B	
Зсунуте слово	A A A A B	
Текст	A B C A B D	$j=5; d[5]=2; \text{max_shift}=j-d[5]=3$
Слово	A B C A B C	
Зсунуте слово	A B C A B C	
Текст	A B C D E A	$j=5; d[5]=0; \text{max_shift}=j-d[5]=5$
Слово	A B C D E F	
Зсунуте слово	A B C D	

Розглянемо програмну реалізацію цього методу.

```

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)
    ; // Успіх!

```

2 Алгоритм Боуєра та Мура

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

БМ-пошук базується на незвичних міркуваннях – порівняння символів починається з кінця слова, а не з початку. Як і у випадку КМП-пошуку, слово перед фактичним пошуком трансформується в деяку таблицю. Нехай для кожного символу x із алфавіту величина dx – відстань від самого правого в слові входження x до правого кінця слова. Уявимо, що виявлена розбіжність між словом і текстом. У цьому випадку слово відразу ж можна зсунути праворуч на $dpM-1$ позицій, тобто на кількість позицій, швидше за все більше одиниці. Якщо символ, який не збігся, тексту в слові взагалі не зустрічається, то зсув стає навіть більшим, а саме зсовувати можна на довжину всього слова. Ось приклад, який ілюструє цей процес:

```

ABCSABCSABFABCSABD
ABCSABD_
  ABCSABD_
    ABCSABD_
      ABCSABD_

```

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.

Програму можна записати таким чином.

```

for (ch=0; ch<256; ch++)
    d[ch] = M; // заповнення

```

```

for (i=0; i<M-1; i++)
    d[s[i]] = M-i-1; // уточнення
// Поиск слова p в тексте s
i = M;
do
{
    j = M;
    k = i;
    do // Цикл порівняння символів
    {
        k--;
        j---; // слова, починаючи з правого
        while ( (j<0) || (a[j]!=s[k]) ); //Вихід, при порівн. все
слово або незбіг
        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.

3 Алгоритм Рабіна.

Цей алгоритм заснований на простій ідеї. Уявімо собі, що в слові довжиною m шукається зразок довжиною n . Виріжемо віконце розміром n і будемо рухати його по вхідному слову. При цьому перевіряємо, не співпадає слово у віконці з заданим зразком. Порівнювати по буквах довго. Натомість фіксуємо деяку функцію, визначену на словах довжиною n . Якщо значення цієї функції на слові в віконечку і на зразку різні, то збігу немає.

Виграш при такому підході полягає в наступному. Щоб обчислити значення

функції на слові Виграш при такому підході полягає в наступному. Щоб обчислити значення функції на слові в віконечку, потрібно прочитати всі букви цього слова. Так вже краще їх відразу порівняти зі зразком. Проте виграш можливий, тому що при зсуві віконця слово не змінюється повністю, а лише додається буква в кінці і забирається на початку. Замінімо всі букви в слові і зразку їх номерами, котрі представляють собою цілі числа. Тоді зручною функцією є сума цифр. (При зсуві віконця потрібно додати нове число і відняти зникле)

Питання та завдання до контролю знань студентів

- 1 Що розуміється під пошуком?
- 2 Які особливості пошуку словесної інформації?
- 3 Сформулювати ідею алгоритму Кнута, Моріса, Пратта.
- 4 Сформулювати ідею алгоритму Боуєра та Мура.
- 3 Сформулювати ідею алгоритму Рабіна.

Лекція № 16

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Алгоритми обходу графів: обходи в ширину.
Мета заняття	Дати опис алгоритму обходу графу завширшки та пояснити метод реалізації алгоритму.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Алгоритм обходу в ширину неорієнтованого графа
- 2 Реалізація алгоритму обходу (пошуку) неорієнтованого графа завширшки

Література

- 1 Гагарина, Л.Г. Алгоритмы и структуры данных. [Текст]: учеб. пособие/ Л.Г. Гагарина, В.Д. Колдаев. – М.: Финансы и статистика, ИНФРА-М, 2009. – 304 с.: ил.
- 2 Пышкин, Е.В. Структуры данных и алгоритмы: реализация на C/C++. [Текст] / Е.В. Пышкин. – СПб.: ФТК СПбГПУ, 2009.- 200 с., ил.

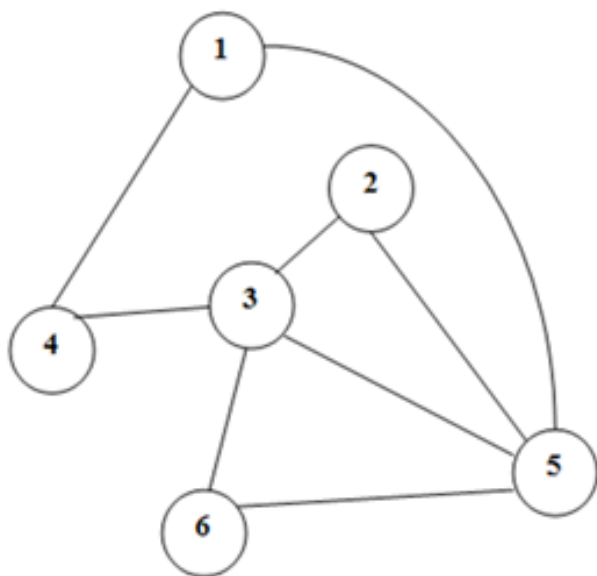
Навчальні матеріали лекції

1 Алгоритм обходу в ширину неорієнтованого графа

Обхід (пошук) завширшки є алгоритмом, що необхідний при рішенні багатьох завдань. Пошук завширшки ефективно переглядає всі вершини й ребра графа, задаючи їм деякі мітки. У результаті обходу графа завширшки проглядаються всі вершини (починаючи зі стартової) і визначаються найкоротші (по кількості пройдених ребер) шляху зі стартової вершини в усі інші вершини графа.

Розглянемо дію алгоритму на прикладі графа G:

1. Вибираємо стартову вершину обходу графа (нехай це буде вершина 0) і



привласнюємо їй мітку 0. Всі інші вершини вважаються непоміченими.

2. Кожній вершині з оточення стартової вершини (тобто кожній суміжній з нею вершині) привласнюємо мітку 1.

Увага: присвоювання відбувається тільки в тому випадку, якщо вершина не помічена.

Таким чином, вершини 3 й 4 одержали мітку 1.

3. Розглянемо по черзі оточення вершин з міткою 1 і кожній непоміченій вершині із цього оточення привласнимо мітку 2. Таким чином, вершини 1, 2 й 5 одержать мітки 2.

4. Далі, розглядаючи по черзі оточення вершин 1, 2, і 5 зауважуємо, що непомічених вершин не залишилося. Це означає, що пошук завширшки закінчений.

Після того, як всі вершини одержали свої мітки, можна визначити відстань від стартової вершини до будь-якої вершини графа. Воно буде дорівнює мітці цієї вершини.

Процес розміщення міток можна зрівняти із процесом поширення хвилі. Тому часто пошук завширшки називають **хвильовим алгоритмом**.

2 Реалізація алгоритму обходу (пошуку) неорієнтованого графа завширшки

У даному графі визначено 6 вершин і тому для реалізації графа буде потрібна матриця розміром 6*6 (назвемо її MG). В комірках MG[i][j] й MG[j][i] будемо ставити 1, якщо вершини графа i й j з'єднані ребром, у противному випадку заповнюємо комірки нулями.

	0	1	2	3	4	5
0	0	0	0	1	1	0
1	0	0	1	0	1	0
MG: 2	0	1	0	1	1	1
3	1	0	1	0	0	0
4	1	1	1	0	0	1
5	0	0	1	0	1	0

Крім уже створеної нами матриці суміжності (матриці MG) будемо використати одномірний масив міток MET, довжина якого дорівнює кількості вершин у графі:

	0	1	2	3	4	5
MET:						

i-й елемент масиву MET буде дорівнює величині мітки, що у процесі роботи програми буде позначатися i-я вершина графа.

Нам потрібна буде черга, у яку будемо послідовно заносити вершини, суміжні з тією, котра перебувати в "голові" черги. Організуємо чергу на основі одномірного масиву OCH, довжина якого дорівнює кількості вершин у графі:

	0	1	2	3	4	5
ОСН:						

Для фіксації "голови" черги будемо використати цілочисельну змінну **first**. Для фіксації "хвоста" черги будемо використати цілочисельну змінну **last**. Для фіксації кінця чергової "хвилі" будемо використати цілочисельну змінну **last1**.

Будемо використати цілочисельну змінну **M**, значення якої буде дорівнює величині мітки вершин чергової хвилі.

Опис алгоритму:

1. Уведенням із клавіатури заповнюємо матрицю суміжності **MG**.
2. Масив міток **МЕТ** заповнюємо -1 (ознака того, що вершини не помічені).
3. Запитуємо номер *i* - стартової вершини.
4. **M=0;**
5. У відповідній стартовій вершині елемент масиву **МЕТ[i]** заносимо **M**.
6. Номер стартової вершини заносимо в **ОСН[1]** і встановлюємо **first** й **last** й **last1** на **ОСН[0]**, тобто **first=0, last=0** й **last1=0**

Нехай, наприклад, стартової буде вершина 1. Тоді: **M=0**

	0	1	2	3	4	5
МЕТ:	-1	0	-1	-1	-1	-1

	0	1	2	3	4	5
ОСН:	1					

$\swarrow$ $\nwarrow$ $\nearrow$
first=0 **last=0** **last1=0**

7. Організуємо **цикл 1** (поки черга не порожня, тобто поки **first <= last**)
 - 7.1. Збільшуємо **M** на 1 **M=M+1;**
 - 7.2. Організуємо **цикл2** (цикл обробки вершин чергової хвилі (тобто вершин, що перебувають між **first** й **last1**.)

7.2.1. Ідемо на рядок матриці суміжності, номер якої відповідає вершині, що перебуває в "голові" черги", тобто в **ОСН[first]**. Якщо деякий елемент **MG[first][j]** дорівнює 1 а **МЕТ[j]=-1**(вершина не помічена), заносимо **j** у чергу (*зрушуємо last. !!! last1 не торкаємо*) а елементу **МЕТ[j]** привласнюємо значення **M** т.е. **МЕТ[j]:=M**.

7.2.2. Вийшовши з рядка матриці суміжності в масиві **ОСН** пересуваємо **first** на наступний елемент масиву (на наступну вершину "хвилі"), тобто **first=first+1**.

Кінець цикл 2 (коли **first** стане більше **last1**)

- 7.3. Зрушуємо **last1** на останній, занесений у чергу елемент (якщо такі були). Т.е. **last1=last**

Кінець цикл 1

Питання та завдання до контролю знань студентів

1. Як зв'язані між собою різні способи подання графів?
2. Як від виду або подання графа залежить часова складність алгоритмів пошуку в глибину й завширшки ?
3. Як при реалізації в коді виконується повернення з тупикових вершин при обході графа?
4. Як виконується обхід у незв'язному графі?
5. Чи поширюються поняття "пошук завширшки" на незв'язний граф? Відповідь обґрунтуйте.
6. Як в алгоритмах обходу графа завширшки фіксується, що переглянута вершина не нова?
7. Як називається алгоритм обходу графа, що використовує для своєї роботи черга?
8. Що зберігається в черзі алгоритму обходу графа, що використовує для своєї роботи черга?

Лекція № 17

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Алгоритми обходу графів: обходи в глибину.
Мета заняття	Дати опис алгоритму обходу графу в глибину та пояснити метод реалізації алгоритму.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	
Конспект лекцій.	

Час – 2 години

План проведення лекції

- 1 Алгоритм обходу в глибину неорієнтованого графа.
- 2 Опис алгоритму пошуку в глибину.
 - 2.1 Рекурсивний алгоритм пошуку у глибину
 - 2.1 Нерекурсивний алгоритм пошуку у глибину

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

1 Алгоритм обходу в глибину неорієнтованого графа

Пошук (обхід) у глибину (англ. depth-first search, DFS) - це рекурсивний алгоритм обходу вершин графа. Якщо метод пошуку завширшки проводився симетрично (вершини графа проглядалися по рівнях), то даний метод припускає просування вглиб доти, поки це можливо. Неможливість подальшого просування, означає, що наступним кроком буде перехід на останній, що має кілька варіантів руху (один із яких досліджений повністю), раніше відвіданий вузол (вершина). Відсутність останнього свідчить про одній із двох можливих ситуацій: або всі вершини графа уже переглянуті, або переглянуті всі ті, що доступно з вершини, узятої в якості початкової, але не всі (незв'язні й орієнтовані графи допускають останній варіант).

При обході неорієнтованого графу в глибину ми відвідуємо перший вузол, а потім ідемо уздовж ребер графа, поки не потрапимо в тупик. Вузол неорієнтованого графа є тупиком, якщо ми вже відвідали всі вузли, що примикають до нього. В орієнтованому графі тупиком також виявляється вузол, з якого немає вихідних ребер.

Після потрапляння в тупик ми повертаємося назад уздовж пройденого шляху поки не виявимо вершину, у якої є ще не відвіданий сусід, а потім рухаємося в цьому новому напрямку. Процес виявляється завершеним, коли ми повернулися у відповідну вершину, а всі вершини, що примикають до неї, уже виявилися відвіданими.

Ілюструючи цей й інші алгоритми обходу, при виборі однієї із двох вершин ми завжди будемо вибирати вершину з меншим (у числовому або лексикографічному порядку) номером. При реалізації алгоритму вибір буде визначатися способом зберігання ребер графа.

Розглянемо граф на рисунку 1. Почавши обхід у глибину у вершині 1, ми потім відвідаємо послідовно вершини 2, 3, 4, 7, 5 й 6 й уперемося в тупик. Потім нам доведеться повернутися у вершину 7 і виявити, що вершина 8 залишилася невідвіданою. Однак, перейшовши в цю вершину, ми негайно знову виявляємося в тупику. Повернувшись у вершину 4, ми виявляємо, що невідвіданою залишилася вершина 9; її відвідування знову заводить нас у тупик. Ми знову повертаємося назад у вихідну вершину 1, і оскільки всі сусідні з нею вершини виявилися відвіданими, обхід закінчений.

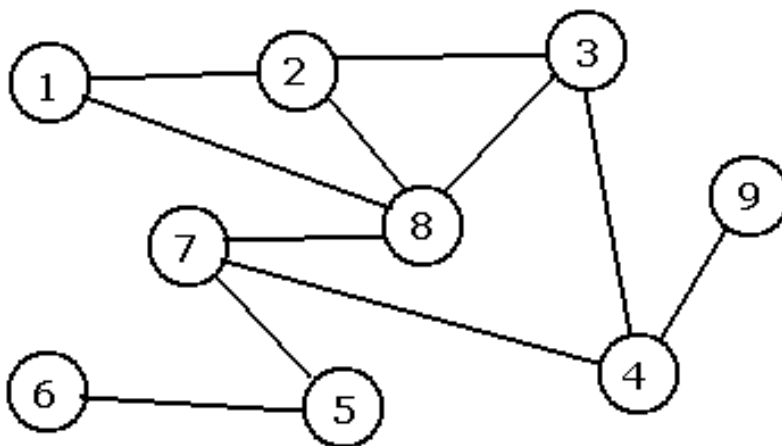


Рис. 1. Приклад графа

2 Опис алгоритму пошуку в глибину

Крок 1. Всім вершинам графа привласнюється значення (мітка) не відвідана. Вибирається перша (стартова) вершина й позначається як відвідана.

Крок 2. Для останньої позначеної як відвідана вершини вибирається суміжна вершина, що є першою позначеною як не відвідана, і їй привласнюється значення відвідана. Якщо таких вершин немає, то береться попередня позначена вершина.

Крок 3. Повторити крок 2 доти, поки всі вершини не будуть позначені як відвідані.

Робота цього алгоритму показана на рисунку 2.

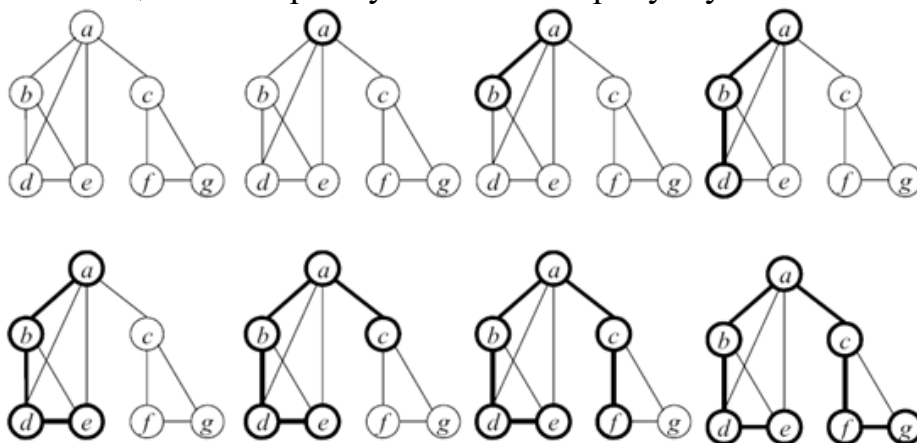


Рис. 2. Демонстрація алгоритму пошуку в глибину

2.1 Рекурсивний алгоритм пошуку у глибину

Рекурсивний алгоритм пошуку в глибину базується на функція обходу, яка по мірі виконання викликає сама себе, що робить код у цілому досить компактним. Псевдокод алгоритму:

Заголовок функції DFS(st)

Вивести (st)

visited[st] = відвідана;

Для r=1 до n виконувати

Якщо (graph[st, r] $\neq$ 0) і (visited[r] не відвідана) те DFS(r)

Тут DFS (depth-firstsearch) - назва функції. Єдиний її параметр st - стартовий вузол, переданий з головної частини програми як аргумент. Кожному з елементів логічного масиву visited заздалегідь привласнюється значення false, тобто кожна з вершин стартово буде значитися як не відвідана. Двовимірний масив graph - це матриця суміжності графа. Більшу частину уваги варто сконцентрувати на останньому рядку. Якщо елемент матриці суміжності, на якомусь кроці дорівнює 1 (а не 0) і вершина з тим же номером стовпця не була відвідана, то функція рекурсивно повторюється. У протилежному випадку функція повертається на попередню стадію рекурсії.

Цей рекурсивний алгоритм використовує системний стек для відстеження поточної вершини графа, що дозволяє правильно здійснити повернення, наткнувшись на тупик. Можна побудувати аналогічний нерекурсивний алгоритм, скориставшись стеком для занесення туди й видалення пройдених вершин графа.

2.1 Нерекурсивний алгоритм пошуку у глибину

У наведеній нижче формулюванні нерекурсивного алгоритму пошуку в глибину на графі передбачається: по-перше, що зафіксовано деякий лінійний порядок на множині всіх вершин графа, і, по-друге, що множина вершин, суміжних із усякою вершиною графа, також лінійно впорядкована.

```
while (Є хоча б одна не відвідана вершина)
{
    Нехай p - перша з не відвіданих вершин.
    Відвідати вершину p і помістити її в порожній стек S;
    while (Стік S непорожній )
    {
        Нехай p - вершина, що перебуває на верхівці стека S;
        if (У вершини p є не відвідані суміжні вершини)
        { Нехай q - перша не відвідана вершина з вершин, суміжних вершині p. Пройти по ребру (p,q), відвідати вершину q і помістити її в стек S }
        else Видалити вершину p зі стека S
    }
}
```

Питання та завдання до контролю знань студентів

1. Поняття обходу графу.
2. Суть алгоритму обходу в глибину.
3. Рекурсивний алгоритм обходу в глибину.
4. Ітеративний алгоритм обходу в глибину (з використанням стеку).

Лекція № 18

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Жадібні алгоритми
Мета заняття	Ввести поняття алгоритмічної стратегії жадібні алгоритми. Розглянути приклади типових оптимізаційних завдань.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	
Конспект лекцій.	

Час – 2 години

План проведення лекції

- 1 Алгоритмічна стратегія – жадібні алгоритми
- 2 Приклади оптимізаційних завдань
 - 2.1 Завдання про вибір заявок
 - 2.2 Завдання про число аудиторій
 - 2.3 Дискретне завдання про рюкзак

Література

- 1 Гагарина, Л.Г. Алгоритмы и структуры данных. [Текст]: учеб. пособие/ Л.Г. Гагарина, В.Д. Колдаев. – М.: Финансы и статистика, ИНФРА-М, 2009. – 304 с.: ил.
- 2 Пышкин, Е.В. Структуры данных и алгоритмы: реализация на C/C++. [Текст] / Е.В. Пышкин. – СПб.: ФТК СПбГПУ, 2009.- 200 с., ил.

Навчальні матеріали лекції

1 Алгоритмічна стратегія – жадібні алгоритми

У теорії алгоритмів відіграють важливу роль жадібні алгоритми. Вони прості для розуміння й реалізації, працюють порівняно швидко, відомо багато різноманітних завдань, які можна вирішити за допомогою таких алгоритмів. Однак не завжди можна довести можливість застосовності жадібного алгоритму для знаходження точного рішення багатьох завдань.

Жадібний алгоритм(greedy algorithm) - метод рішення оптимізаційних завдань, заснований на тім, що процес ухвалення рішення можна розбити на елементарні кроки, на кожному з яких приймається окреме рішення. Рішення прийняте на кожному кроці повинне бути оптимальним тільки на поточному кроці й повинне прийматися без обліку попередніх або наступних рішень.

У жадібному алгоритмі завжди робиться вибір, що здається найкращим у цей момент - тобто робиться локально оптимальний вибір у надії, що він приведе до оптимального рішення глобального завдання. Пошук оптимальної конфігурації в жадібних алгоритмах відбувається покроково, і на кожному кроці виконуються дії, які поліпшують конфігурацію.

Не для кожного оптимізаційного завдання існує жадібний алгоритм, що приводить до оптимальної конфігурації. Так для завдань про максимальне число заявок і мінімальне число аудиторій оптимальне рішення можна одержати, успадковуючи деяку "жадібну" стратегію. А для завдань, подібним до завдання про рюкзак не існує жадібної стратегії, що для будь-яких вхідних даних гарантує знаходження оптимальної конфігурації.

Завжди працюючих жадібних стратегій немає для більшості оптимізаційних завдань, які виникають на практиці, більш того, для них часто взагалі немає алгоритмів рішення, які працюють поліноміальний час. У таких випадках, жадібні стратегії використовують для одержання наближених рішень оптимізаційних завдань. Жадібні стратегії часто здатні за короткий час знаходити нехай не оптимальні, але прийнятні рішення складних завдань.

Ознаки того, що завдання можливо вирішити за допомогою жадібного алгоритму:

- 1.Завдання можна розбити на підзадачі;
2. Величини, розглянуті в завданні, можна дробити так само на підзадачі;
- 3.Сума оптимальних рішень для двох підзадач дасть оптимальне рішення для всього завдання.

2 Приклади оптимізаційних завдань

2.1 Завдання про вибір заявок

Нехай дані n заявок на проведення занять в одній і тій же аудиторії. У кожній заявці зазначений час початку й кінця заняття (s_i й f_i для i -ї заявки). Заявки i -я й j -я сумісні, якщо тимчасові інтервали (s_i, f_i) і (s_j, f_j) не перетинаються. Потрібно вибрати максимальну кількість попарно сумісних заявок.

Це завдання може бути сформульоване геометричною мовою:

На прямій дана множина інтервалів. Необхідно знайти максимальну підмножину непересічних інтервалів.

Пропонується наступна жадібна стратегія: упорядкуємо заявки по якому-небудь принципу. Потім будемо послідовно розглядати заявки. Для кожної заявки будемо перевіряти, чи не перекривається вона з однією з обраних (удоволених) заявок. Якщо перекривається, то її слід відкинути, інакше - додаємо цю заявку в множину обраних заявок (рисунок 1).

"Жадібність" тут полягає в наступному - кожну нову розглянуту заявку ми "не роздумуючи" вибираємо, якщо її можна вибрати (якщо вона сумісна із уже обраними заявками).

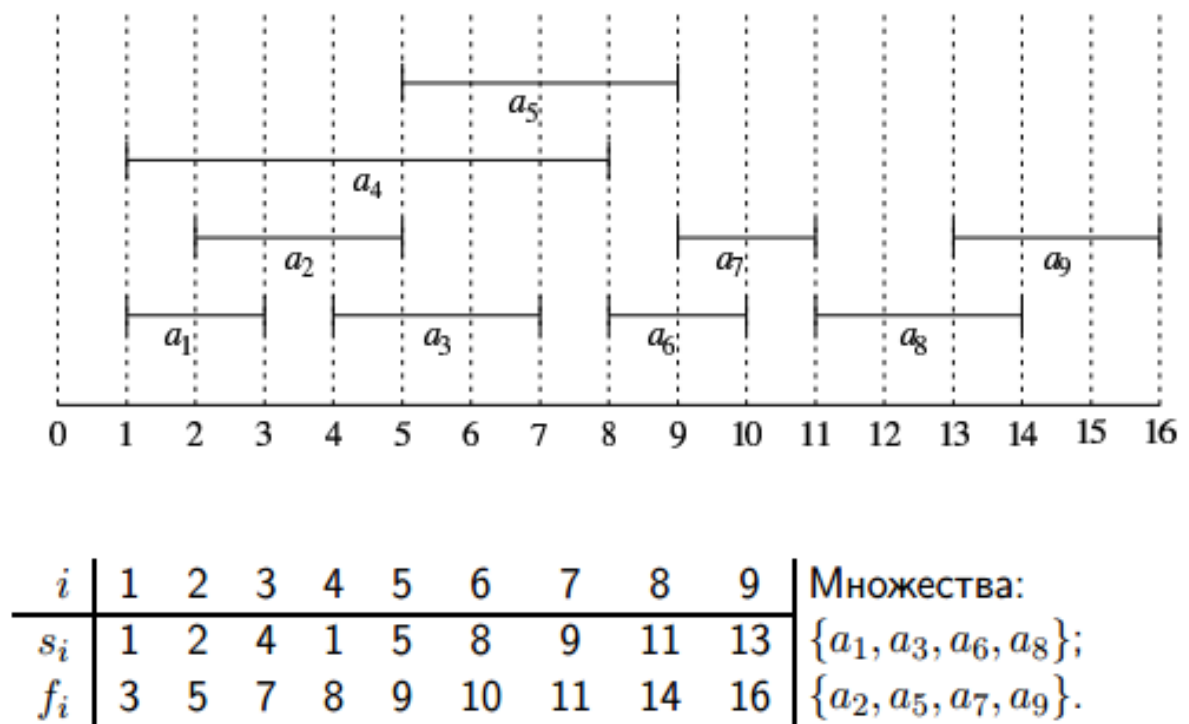


Рис. 1. «Геометричне» подання задачі про вибір заявок.

Найбільше часто пропонуються наступні принципи впорядкування заявок:

- упорядкувати заявки по тривалості (можливо, має сенс спочатку задовольнити заявки маленької тривалості);
- упорядкувати заявки за часом початку;
- упорядкувати заявки за часом закінчення.

Для третього способу впорядкування жадібна стратегія завжди дає оптимальне рішення.

Отже, сформулюємо алгоритм точного рішення завдання.

Рішення. Відсортуємо заявки за часом закінчення. На сортування піде час $O(n \log n)$. Заново перенумеруємо заявки, так що

$$f_1 \leq f_2 \leq \dots \leq f_n.$$

Заявки з однаковим часом кінця розташуємо в довільному порядку.

Виберемо першу заявку. Потім виберемо другу заявку, якщо вона не перекривається з першою. На третьому кроці виберемо третю заявку, якщо вона не перекривається з обраними заявками. Аналогічно, на i -м кроці ми вибираємо i -ю заявку, якщо вона не перекривається із уже обраними. Помітимо, що на кожному кроці досить перевіряти, що чергова заявка не перетинається з останньою обраною заявкою, а точніше, що час початку чергової заявки більше часу закінчення останньої обраної заявки.

2.2 Завдання про число аудиторій

Розглянемо інше завдання про заявки, у якій потрібно задовольнити всі заявки, але при цьому використати якнайменше аудиторій.

Завдання. Нехай дані n заявок на проведення занять. У кожній заявці зазначений час початку й кінця заняття (s_i й f_i для i -ї заявки). Потрібно розподілити заявки по мінімальному числу аудиторій.

Для рішення цього завдання також застосовна жадібна стратегія. Упорядковані по зростанню часу початку (s_i) заявки можна послідовно розподілити по аудиторіях у такий спосіб.

1. Розглядаємо чергову заявку
2. Пробуємо неї помістити в першу аудиторію. Якщо вільного часу для заявки в цій аудиторії ні, пробуємо помістити заявку в наступну аудиторію, і так далі.
3. Якщо в жодну аудиторію помістити заявку не вдалося, відкриваємо нову аудиторію

2.3 Дискретне завдання про рюкзак

Злодій під час пограбування магазину виявив n предметів. Предмет під номером i має вартість v_i й вага w_i , де v_i й w_i - цілі числа.

Потрібно винести речі, сумарна вартість яких була б як можна більшою, однак вантажопідйомність рюкзака обмежується W кілограмами. Які предмети варто взяти із собою?

Алгоритм рішення задачі:

1 Відсортуємо речі за «питомою цінністю» v_i / w_i .

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

2 Будемо послідовно обирати речі з максимальною питомою цінністю та поміщати їх в рюкзак.

Таке рішення працює для неперервної задачі про рюкзак, але не працює для дискретної. Так «жадібне» рішення дає результат: речі 1 і 2, сума 160, вага 30. Тоді як оптимальне рішення: речі 2 і 3, сума 220, вага 50.

Питання та завдання до контролю знань студентів

1. Що таке жадібний алгоритм?
2. Коли застосовні жадібні алгоритми?
3. Указати дві відмінні риси жадібних алгоритмів.

Лекція № 19

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Алгоритми евристики.
Мета заняття	Ввести поняття та дати характеристики евристичних алгоритмів.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Евристичні алгоритми
- 2 Застосування
- 3 Приклад оцінки евристичного рішення

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротиєєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Евристичні алгоритми

Евристичний алгоритм (евристика) -- це алгоритм розв'язку задачі, який не має точного обґрунтування, але в більшості практичних випадків забезпечує прийнятну відповідь (за якістю та часом роботи).

Це не значить, що рішення, отримане евристичним шляхом завжди є приблизним. Під неточністю евристики розуміють, що навіть, якщо знайдені таким чином відповіді є точними, немає гарантій, що вони так само будуть точними для будь-яких інших вхідних даних.

Евристичні методи передбачають застосування творчого підходу та досвіду для пошуку рішення. В загальному випадку вони застосовуються не лише в програмуванні, але для вирішення будь-яких задач. Наприклад, проектування складних систем, розв'язування логічних головоломок, будь-яка творча діяльність.

В програмуванні це реалізується в спробі відтворити дії людини з розв'язку задачі і формалізувати їх у вигляді алгоритму. Якщо розглядати задачу як лабіринт (розповсюджена метафора з психології), то пошук рішення є пошуком шляху в цьому лабіринті, а евристика -- це набір прийомів, вироблених вами, які дозволяють відкинути частину шляхів, зменшуючи таким чином простір можливих рішень.

Таким чином, будь-які ваші алгоритми, розроблені на інтуїтивному рівні, та будь-які оптимізації алгоритмів, які впроваджуються для збільшення швидкості їх роботи, можна вважати евристичними

Простіше кажучи, евристика - це не повністю математично обґрунтований (або навіть "не зовсім коректний"), але при цьому практично корисний алгоритм.

Важливо розуміти, що евристика, на відміну від коректного алгоритму розв'язання задачі, володіє наступними особливостями:

- 1 Вона не гарантує знаходження кращого рішення.
- 2 Вона не гарантує знаходження рішення, навіть якщо воно явно існує (можливий "пропуск цілі").
- 3 Вона може дати невірне рішення в деяких випадках.

Отже, це алгоритми, що мають такі властивості:

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

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

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

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

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

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

Цю ситуацію образно показано на малюнку

	1	2
a)	Слід задовольнити	?
b)	?	Варто відмовитися

Приклад 1. Задача про мандрівного крамаря.

Якщо міст, про які йдеться - багато, то пошук точно мінімального за довжиною циклу може вимагати дуже багато часу. З іншої сторони, об'їзд слід зробити лише один раз. Якщо витрати на пошук оптимального маршруту можуть виявитися

більшими або еквівалентними до витрат на пальне під час подорожі, то, можливо, варто просто рушати в путь, прямучи до найближчого міста кожного разу.

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

Приклад 2. Сорткування масиву.

Якщо Вам треба відсортувати один раз масив на 100 записів, то можна використовувати будь-який з простих методів сортування - на весь процес буде витрачено щонайбільше 2-3 секунди машинного часу.

Якщо ж ви розроблюєте пакет, що буде регулярно сортувати значні масиви інформації, то варто написати модернову програму.

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

2 Застосування

Евристичні алгоритми широко застосовуються для вирішення завдань високої обчислювальної складності (завдання, що належать класу NP), тобто замість повного перебору варіантів, що займає істотне час, а іноді технічно неможливого, застосовується значно швидший, але недостатньо обґрунтований теоретично, алгоритм. В областях штучного інтелекту, таких, як розпізнавання образів, евристичні алгоритми широко застосовуються також й через відсутність спільного рішення поставленої задачі. Різні евристичні підходи застосовуються в антивірусних програмах, комп'ютерних іграх і т. д. Наприклад, програми, що грають в шахи, проводять середину гри, ґрунтуючись, переважно, на евристичних алгоритмах (в дебюті може використовуватися база даних, в ендшпілі - таблиці Налимова, але в міттельшпілі часто кількість можливих ходів виключає повний перебір, а точних алгоритмів правильної гри не існує).

За твердженням Judea Pearl, евристичні методи засновані на інтелектуальному пошуку стратегій комп'ютерного вирішення проблеми з використанням декількох альтернативних підходів.

Можливість (допустимість) використання евристик для розв'язання кожної конкретної задачі визначається співвідношенням витрати на вирішення завдання точним і евристичним методом, ціною помилки та статистичними параметрами евристики. Крім того, важливим є наявність або відсутність на виході "фільтра здорового глузду" - оцінки результату людиною.

3 Приклад оцінки евристичного рішення

Розглянемо уможливленний приклад. Припустимо, що є відомий, але надзвичайно складний точний алгоритм вирішення задачі, і евристика, яка вимагає в 1000 разів менше витрат і найчастіше дає прийнятне рішення (нехай в 95% випадків). Для простоти приймемо, що ціна точного рішення постійна, як і ціна помилки.

Тоді в середньому рішення евристичним методом буде коштувати $(T / 1000 + 0.05 * E)$, де T - ціна точного рішення, а E - ціна помилки. Середня різниця в ціні рішення точним і евристичним методом $(T - T / 1000 - 0,05 * E) = (19,98 * T - E) / 20 = 0,999 * T - E / 20$, Тобто евристика в середньому виявляється вигідніше точного рішення, якщо тільки ціна помилки не перевищує двадцятикратному (!) Ціну точного рішення.

Якщо ж на виході результат рішення критично оцінюється людиною, то ситуація стає ще краще: коли помилка, видана евристикою, виявляється досить мала, щоб людина її не помітив, ціна цієї помилки зазвичай набагато нижча, а серйозні помилки будуть відсіяні "фільтром здорового глузду", отже, не завдадуть істотної шкоди.

Питання та завдання до контролю знань студентів

Теоретичні питання:

- 1 Сформулюйте основну ідею методу евристик.
- 2 Які вимоги до розв'язку аналізуються у евристичних підходах?
- 3 У яких ситуаціях бажане застосування евристичних підходів?

Практичні завдання:

- 1 Для задачі про мандрівного крамаря порівняйте час пошуку розв'язку та довжини відшуканих циклів для методу, який вимагає на кожному кроці їхати у найближче місто (евристика), та такого, що шукає оптимальний маршрут з усіх можливих (повним перебором варіантів). Матрицю відстаней формуйте за допомогою генератора випадкових чисел. Кількість міст $n < 7$.
- 2 Скільки різних циклів можна побудувати у повному графі з n вершинами?

Лекція № 20

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Метод гілок та границь. Задача комівояжера.
Мета заняття	Познайомити студентів з постановкою завдання дискретного програмування, основними класами завдань. Вивчити один з основних методів рішення даного класу завдань - метод гілок і границь.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	
Конспект лекцій.	

Час – 2 години

План проведення лекції

- 1 Загальна постановка задачі
- 2 Загальний план рішення задачі комівояжера
- 3 Докладна методика рішення задачі комівояжера

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп. ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Загальна постановка задачі

Одна з найвідоміших і важливих завдань транспортної логістики (і класу завдань оптимізації в цілому) - завдання комівояжера (англ. "Travelling salesman problem", TSP). Також зустрічається назва "завдання про бродячого торговця". Суть завдання зводиться до пошуку оптимального, тобто найкоротшого шляху, що проходить через якісь пункти по одному разу. Наприклад, завдання комівояжера може застосовуватися для знаходження самого вигідного маршруту, що дозволяє комівояже-рові об'їхати певні міста зі своїм товаром по одному разі й повернутися у вихідну точку. Мірою вигідності маршруту буде мінімальний час, проведене в дорозі, мінімальні витрати на дорогу або, у найпростішому випадку, мінімальна довжина шляху. Хто й коли вперше почав досліджувати завдання комівояжера невідомо, але одним з перших запропонував рішення подібної проблеми видатний математик ХІ в. - Вільям Гамільтон. Тут ми розглянемо замкнутий варіант завдання (тобто такий, коли в підсумку ми повертаємося у вихідну точку) і її рішення методом гілок і границь.

2 Загальний план рішення задачі комівояжера

Для рішення завдання комівояжера методом гілок і границь необхідно виконати наступний алгоритм (послідовність дій):

1. Побудова матриці з вихідними даними.
2. Знаходження мінімуму по рядках.
3. Редукція рядків.
4. Знаходження мінімуму по стовпцях.
5. Редукція стовпців.
6. Обчислення оцінок нульових кліток.
7. Редукція матриці.
8. Якщо повний шлях ще не знайдений, переходимо до пункту 2, якщо знайдено до пункту 9.
9. Обчислення підсумкової довжини шляхи й побудова маршруту.

Більш докладно ці етапи рішення завдання розкриті нижче.

3 Докладна методика рішення задачі комівояжера

З метою кращого розуміння завдання будемо оперувати не поняттями графа, його вершин і т.д., а поняттями простими й максимально наближеними до реальності: вершини графа будуть називатися "міста", ребра їх з'єднуючі - "дороги".

Отже, методика рішення завдання комівояжера:

1. Побудова матриці з вихідними даними

Спочатку необхідно довжини доріг з'єднуючі міста представити у вигляді наступної таблиці (матриці сумісності для зваженого графу):

Вершина (місто)	1	2	3	4
1	∞	5	11	9
2	10	∞	8	7
3	7	14	∞	8
4	12	6	15	∞

Відстань від міста до цього ж міста позначено знаком нескінченності. Це зроблено для того, щоб даний відрізок шлях був умовно прийнятий за нескінченно довгий. Тоді не буде змісту вибрати рух від 1-ого міста до 1-му, від 2-ого до 2-му, і т.п. як відрізок маршруту.

2. Знаходження мінімуму по рядках

Знаходимо мінімальне значення в кожному рядку (min i) і виписуємо його в окремий стовпець.

Вершина (місто)	1	2	3	4	min i
1	∞	5	11	9	5
2	10	∞	8	7	7
3	7	14	∞	8	7
4	12	6	15	∞	6

3. Редукція рядків

Робимо редукцію рядків - з кожного елемента в рядку віднімаємо відповідне значення знайденого мінімуму (min i).

Вершина (місто)	1	2	3	4	min i
1	∞	0	6	4	5
2	3	∞	1	0	7
3	0	7	∞	1	7
4	6	0	9	∞	6

У підсумку в кожному рядку буде хоча б одна нульова клітка.

4. Знаходження мінімуму по стовпцях

Далі знаходимо мінімальні значення в кожному стовпці ($\min j$). Ці мінімуми виписуємо в окремий рядок.

Вершина (місто)	1	2	3	4	$\min i$
1	∞	0	6	4	5
2	3	∞	1	0	7
3	0	7	∞	1	7
4	6	0	9	∞	6
$\min j$	0	0	1	0	

5. Редуція стовпців

Віднімаємо з кожного елемента матриці відповідне йому d_j

Вершина (місто)	1	2	3	4	$\min i$
1	∞	0	5	4	5
2	3	∞	0	0	7
3	0	7	∞	1	7
4	6	0	8	∞	6
$\min j$	0	0	1	0	

У підсумку в кожному стовпці буде хоча б одна нульова клітка.

6. Обчислення оцінок нульових кліток

Для кожної нульової клітки отриманої перетвореної матриці знаходимо "оцінку". Нею буде сума мінімального елемента по рядку й мінімальному елементу по стовпці, у яких розміщена дана нульова клітка. Сама вона при цьому не враховується. Знайдені раніше $\min i$ й $\min j$ не враховуються. Отриману оцінку записуємо поруч із нулем, у дужках. І так по всіх нульових клітках:

Вершина (місто)	1	2	3	4
1	∞	0 (4)	5	4
2	3	∞	0 (5)	0 (1)
3	0 (4)	7	∞	1
4	6	0 (6)	8	∞

7. Редуція матриці

Вибираємо нульову клітку з найбільшою оцінкою. Заміняємо її на ∞ . Ми знайшли один з відрізків шляху. Виписуємо його (від якого міста до якого рухаємося, у нашому прикладі від 4-ого до 2-му).

Вершина (місто)	1	2	3	4
1	∞	0 (4)	5	4
2	3	∞	0 (5)	0 (1)
3	0 (4)	7	∞	1
4	6	0 (6)	8	∞

Той рядок і той стовпець, де утворилося дві ∞ повністю викреслюємо. У клітку, що відповідає дорозі назад, ставимо ще одну ∞ (тому що ми вже не будемо повертатися назад).

Вершина (місто)	1	2	3	4
1	∞	0 (4)	5	4
2	3	∞	0 (5)	∞
3	0 (4)	7	∞	1
4	6	∞	8	∞

8. Якщо повний шлях ще не знайдений, переходимо до пункту 2, якщо знайдено до пункту 9

Якщо ми ще не знайшли всі відрізки шляху, то повертаємося до 2-му пункту й знову шукаємо мінімуми по рядках і стовпцям, проводимо їхню редукцію, враховуючи оцінки нульових кліток і т.д.

Якщо всі відрізки шляху знайдені (або знайдені ще не всі відрізки, але частина, що залишилася, шляху очевидна) - переходимо до пункту 9.

9. Обчислення підсумкової довжини шляхи й побудова маршруту

Знайшовши всі відрізки шляху, залишається тільки з'єднати їх між собою й розрахувати загальну довжину шляху (вартість поїздки по цьому маршруті, витрачене час і т.д.). Довжини доріг з'єднуючі міста беремо з найпершої таблиці з вихідними даними.

У нашому прикладі маршрут вийшов наступний: $4 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 4$.

Загальна довжина шляху: $L = 30$.

Питання та завдання до контролю знань студентів

1. Сформулюйте визначення завдання дискретного програмування.
2. Яка основна ідея методу гілок і границь?
3. У чому складаються особливості реалізації методу гілок і границь при рішенні конкретних класів завдань?

4. Що таке нижня оцінка множини?
5. У чому полягає процес розгалуження множини?
6. Які множини називаються кінцевими?
7. Сформулюйте ознаку оптимальності в завданні цілочисельного лінійного програмування.
8. Сформулюйте постановку завдання про комівояжера.
9. У чому полягає процедура приведення матриці в завданні про комівояжера?
10. Сформулюйте критерій оптимальності циклу для завдання про комівояжера.

Лекція № 21

з навчальної дисципліни «Алгоритми та структури даних»

Тема	Порівняльні оцінки алгоритмів.
Мета заняття	Ввести поняття трудомісткості алгоритмів. Дати визначення стандартних класів складності.
Матеріально-технічне забезпечення та дидактичні засоби, ТЗН	Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Порівняльні оцінки алгоритмів
- 2 Система позначень в аналізі алгоритмів
- 3 Теорія складності обчислень і класи завдань за складністю
 - 3.1 Теоретична межа трудомісткості завдання
 - 3.2 Стандартні класи складності

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Порівняльні оцінки алгоритмів

Для розв'язання більшості задач можна застосувати не один, а кілька різних алгоритмів. Якщо ми маємо справу з невеликими об'ємами даних, оптимізація не є суттєвою, так як на малих розмірностях задач різниця в часі виконання різних алгоритмів буде непомітною. Для великих об'ємів даних використання неефективного алгоритму може призвести до надто довгого виконання програми. Наприклад, той же підбір паролю користувача. Задача може бути розв'язана перебором всіх можливих паролів, але час роботи повного перебору для достатньо довгих паролів на одній машині може становити роки – отримання відповіді в такий термін є неприйнятним.

При використанні алгоритмів для вирішення практичних завдань ми стикаємося з проблемою раціонального вибору алгоритму рішення задачі. Вирішення проблеми вибору пов'язане з побудовою системи порівняльних оцінок, яка у свою чергу істотно спирається на формальну модель алгоритму.

Розглядатимемо надалі, дотримуючись визначень Поста, застосовані до загальної проблеми, правильні і фінітні алгоритми, тобто алгоритми, що дають рішення загальної проблеми. Як формальна система, розглядатимемо абстрактну машину, що включає процесор з фон-Нейманівською архітектурою, що підтримує адресну пам'ять і набір "елементарних" операцій співвіднесених з мовою високого рівня.

В цілях подальшого аналізу приймемо наступні допущення:

- кожна команда виконується не більше ніж за фіксований час;
- початкові дані алгоритму представляються машинними словами по (8 бітів кожне).

Конкретна проблема задається N словами пам'яті, таким чином, на вході алгоритму :

$$N_{\beta} = N * \beta \text{ біт інформації.}$$

Відмітимо, що у ряді випадків, особливо при розгляді матричних завдань N є мірою довжини входу алгоритму, що відбиває лінійну розмірність.

Програма, що реалізовує алгоритм для вирішення загальної проблеми складається з M машинних інструкцій по β_m бітів – $M_\beta = M * \beta_m$ біт інформації.

Крім того, алгоритм може вимагати наступних додаткових ресурсів абстрактної машини :

- S_d - пам'ять для зберігання проміжних результатів;
- S_r - пам'ять для організації обчислювального процесу (пам'ять, необхідна для реалізації рекурсивних викликів і повернень).

При вирішенні конкретної проблеми, заданою N словами пам'яті алгоритм виконує не більше, ніж кінцева кількість "елементарних" операцій абстрактної машини через умову розгляду тільки фінітних алгоритмів. У зв'язку з цим введемо наступне визначення:

Під *трудомісткістю алгоритму* для цього конкретного входу - $F_a(N)$, розумітимемо кількість "елементарних" операцій здійснюваних алгоритмом для вирішення конкретної проблеми в цій формальній системі.

Комплексний аналіз алгоритму може бути виконаний на основі комплексної оцінки ресурсів формальної системи, потрібних алгоритмом для вирішення конкретних проблем. Очевидно, що для різних сфер застосування ваги ресурсів будуть різні, що призводить до наступної комплексної оцінки алгоритму :

$$\psi_A = c_1 * F_a(N) + c_2 * M + c_3 * S_d + c_4 * S_r,$$

де c_i - ваги ресурсів.

2 Система позначень в аналізі алгоритмів

При детальнішому аналізі трудомісткості алгоритму виявляється, що не завжди кількість елементарних операцій, що виконуються алгоритмом на одному вході довжини N , співпадає з кількістю операцій на іншому вході такої ж довжини. Це призводить до необхідності введення спеціальних позначень, що відбивають поведінку функції трудомісткості цього алгоритму на вхідних даних фіксованої довжини.

Нехай DA - множина конкретних проблем цього завдання, задана у формальній системі. Нехай $D \in DA$ - завдання конкретної проблеми і $|D| = N$. У загальному випадку існує власна підмножина безлічі DA , що включає усі конкретні проблеми, що мають потужність N , :

позначимо цю підмножину через DN :

$$D_N = \{D \in DA, : |D| = N\};$$

позначимо потужність безлічі DN через

$$M_{DN} \rightarrow M_{DN} = |D_N|.$$

Тоді змістовно цей алгоритм, вирішуючи різні завдання розмірності N , виконуватиме в якомусь випадку найбільшу кількість операцій, а в якомусь випадку найменша кількість операцій. Ведемо наступні позначення:

1. $F_a^{\wedge}(N)$ - гірший випадок - найбільша кількість операцій, що здійснюються алгоритмом A для вирішення конкретних проблем розмірністю N , :

$$F_a^{\wedge}(N) = \max \{F_a(D)\} - \text{гірший випадок на } DN$$

2. $F_a^{\vee}(N)$ - кращий випадок - найменша кількість операцій, що здійснюються алгоритмом A для вирішення конкретних проблем розмірністю N , :

$$F_a^{\vee}(N) = \min \{F_a(D)\} = \min \{F_a(D)\} - \text{кращий випадок на } DN$$

3. $\bar{F}_a(N)$ - середній випадок - середня кількість операцій, що здійснюються алгоритмом A для вирішення конкретних проблем розмірністю N :

$$\bar{F}_a(N) = (1 / M_{DN}) * \sum_{D \in D_N} \{F_a(D)\} - \text{середній випадок на } DN$$

3 Теорія складності обчислень і класи завдань за складністю

3.1 Теоретична межа трудомісткості завдання

Розглядаючи деяке алгоритмічно вирішуване завдання, і аналізуючи один з алгоритмів її рішення, ми можемо отримати оцінку трудомісткості цього алгоритму у гіршому разі - $f_A(D_A) = O(g(D_A))$. Такі ж оцінки ми можемо отримати і для інших ві-

домих алгоритмів рішення цієї задачі. Розглядаючи завдання з цієї точки зору, виникає резонне питання - а чи існує функціональна нижня межа для $g(D_A)$ і якщо "так", то чи існує алгоритм, що вирішує задачу з такою трудомісткістю у гіршому разі.

Інша, точніше формулювання, має наступний вигляд: яка оцінка складності "найшвидшого" алгоритму рішення цієї задачі у гіршому разі? Очевидно, що це оцінка самого завдання, а не якого або алгоритму її рішення. Таким чином, ми приходимо до визначення поняття функціональної теоретичної нижньої межі трудомісткості завдання у гіршому разі:

$$F_{thlim} = \min \{ \Theta(F_a^{\wedge}(D)) \}$$

Якщо ми можемо на основі теоретичних міркувань довести існування і отримати оцінюючу функцію, то ми можемо стверджувати, що будь-який алгоритм, що вирішує цю задачу працює не швидше, ніж з оцінкою F_{thlim} у гіршому разі, :

$$F_a^{\wedge}(D) = \Omega(F_{thlim})$$

Приведем ряд прикладів:

1) Задача пошуку максимуму в масиві $A=(a_1, \dots, a_n)$ – для цієї задачі необхідно переглянути всі елементи і $F_{thlim} = \Theta(n)$.

2) Завдання множення матриць - для цього завдання можна зробити припущення, що необхідно виконати деякі арифметичні операції з усіма початковими даними, що приводить нас до оцінки $F_{thlim} = \Theta(n^2)$. Розбіжність між теоретичною межею і оцінкою кращого відомого алгоритму дозволяє припустити, що або існує, але ще не знайдений швидший алгоритм множення матриць, або оцінка $\Theta(n^{2.34})$ має бути доведена, як теоретична межа.

3.2 Стандартні класи складності

На початку 1960-х років, у зв'язку з початком широкого використання обчислювальної техніки для вирішення практичних завдань, виникло питання про межі практичної застосовності цього алгоритму рішення задачі в сенсі обмежень на її розмірність. Які завдання можуть бути вирішені на ЕОМ за реальний час?

Відповідь на це питання була дана в роботах Кобмена (Alan Cobham, 1964), і Едмондса (Jack Edmonds, 1965), де були введені класи завдань за складністю

1) Клас P (задачі з поліноміальною складністю)

Завдання називається поліноміальним, тобто відноситься до класу P , якщо існує константа k і алгоритм, що вирішує задачу з $F_d(n) = O(n^k)$, де n - довжина входу алгоритму в бітах $n = |D|$.

Завдання класу P - це інтуїтивно, завдання, що вирішуються за реальний час. Відмітимо наступні переваги алгоритмів з цього класу:

- для більшості завдань з класу P константа k менше 6;
- клас P інваріантний по моделі обчислень (для широкого класу моделей);
- клас P має властивість природної замкнутості (сума або твір поліномів є поліном).

Таким чином, завдання класу P є уточнення визначення "практично вирішувального" завдання.

2) Клас NP (поліноміальні завдання, що перевіряються)

Уявимо собі, що деякий алгоритм отримує рішення деякої задачі - чи відповідає отримана відповідь поставленому завданню, і наскільки швидко ми можемо перевірити його правильність?

Розглянемо, наприклад завдання про суму:

Дано N чисел – $A = (a_1, \dots, a_n)$ і число V .

Задача: Знайти вектор (масив) $X = (x_1, \dots, x_n)$, $x_i \in \{0, 1\}$, такий, що $\sum a_i x_i = V$.

Змістовно: чи може бути представлене число V у вигляді суми яких або елементів масиву A .

Якщо якийсь алгоритм видає результат - масив X , то перевірка правильності цього результату може бути виконана з поліноміальною складністю: перевірка $\sum a_i x_i = V$ вимагає не більше $\Theta(N)$ операцій.

Формально: $\forall D \in D_A$, $|D|=n$ поставимо у відповідність сертифікат $S \in S_A$, такий що $|S| = O(n^l)$ і алгоритм $A_s = A_s(D, S)$, такий, що він видає "1", якщо рішення правильне, і "0", якщо рішення невірне. Тоді завдання належить класу NP , якщо $F(A_s) = O(n^m)$.

Змістовно завдання відноситься до класу NP , якщо її рішення деяким алгоритмом може бути швидко (поліноміальний) перевірене.

Проблема $P = NP$

Після введення в теорію алгоритмів понять класів за складністю Едмондсом (Edmonds, 1965) була поставлена основна проблема теорії складності - $P = NP$? Словесне формулювання проблеми має вигляд: чи можна усі завдання, рішення яких перевіряється з поліноміальною складністю, вирішити за поліноміальний час?

Очевидно, що будь-яке завдання, що належить класу P , належить і класу NP , оскільки вона може бути поліноміальний перевірений - завдання перевірки рішення може полягати просто в повторному рішенні задачі.

На сьогодні відсутні теоретичні докази як збігу цих класів ($P=NP$), так й їхньої розбіжності. Припущення полягає в тому, що клас P є власною підмножиною класу NP , тобто $NP \setminus P$ не порожньо

Клас NPC (NP - повні завдання)

Поняття NP - повноту було введено незалежно Куком (Stephen Cook, 1971) і Левиним (журнал "Проблеми передачі інформації", 1973, т.9, вып. 3) і ґрунтується на понятті сводимості одного завдання до іншої.

Сводимість може бути представлена таким чином: якщо ми маємо завдання 1 і алгоритм, що вирішує цю задачу, видає правильну відповідь для усіх конкретних проблем, що становлять завдання, а для завдання 2 алгоритм рішення невідомий, то якщо ми можемо переформулювати (звести) завдання 2 в термінах завдання 1, то ми вирішуємо задачу 2.

Таким чином, якщо задача 1 задана множиною певних проблем D_{A1} , а задача 2 – множиною, і існує функція f_s (алгоритм), що зводить постановку задачі 2 (d_{A2}) до постановки задачі 1 (d_{A1}): $f_s(d_{(2)} \in D_{A2}) = d_{(1)} \in D_{A1}$, то задача 2 зводиться до задачі 1.

Визначення класу NPC (NP -complete) або класу NP -повних задач потребує виконання наступних двох умов: по-перше, задача повинна належати класу NP ($L \in NP$), і, по-друге, до неї поліноміально повинні зводитись всі задачі з класу ($L_x \leq_P L$, для кожного $L_x \in NP$), (дивись рис. 1).

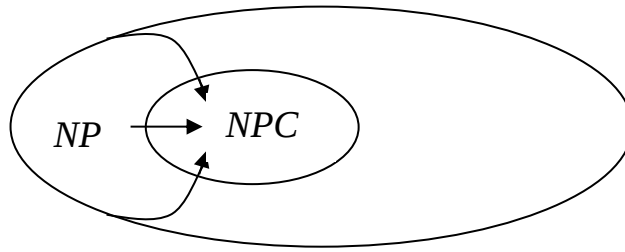


Рис 1. Сводимість NPі класу NPC

Питання та завдання до контролю знань студентів

1. Узагальнений критерій оцінки якості алгоритму,
2. Система позначень в аналізі алгоритмів - гірший, кращий і середній випадки;
3. Поняття класів завдань за складністю, клас P;
4. Проблема $P=NP$, і її сучасний стан;
5. Сводимість мов і визначення класу NPC;
6. Приклади NP - повних завдань;

Лекція № 22

з навчальної дисципліни «Алгоритми та структури даних»

Тема Асимптотичний аналіз функцій.
Трудомісткість алгоритмів і часові оцінки

Мета заняття Ввести поняття функцій трудомісткості та асимптотичного аналізу функцій. Навести приклади оцінки трудомісткості алгоритмів та часові оцінки

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Класифікація алгоритмів за видом функції трудомісткості
- 2 Асимптотичний аналіз функцій
- 3 Трудомісткість алгоритмів і часові оцінки
 - 3.1 Елементарні операції в мові запису алгоритмів
 - 3.2 Приклади аналізу простих алгоритмів
 - 3.3 Перехід до часових оцінок

Література

1. . Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Класифікація алгоритмів за видом функції трудомісткості

Залежно від впливу початкових даних на функцію трудомісткості алгоритму може бути запропонована наступна класифікація, що має практичне значення для аналізу алгоритмів, :

1. Кількісно-залежні по трудомісткості алгоритми. Це алгоритми, функція трудомісткості яких залежить тільки від розмірності конкретного входу, і не залежить від конкретних значень:

$$F_a(D) = F_a(|D|) = F_a(N)$$

Прикладами алгоритмів з кількісно-залежною функцією трудомісткості можуть служити алгоритми для стандартних операцій з масивами і матрицями - множення матриць, множення матриці на вектор і так далі

2. Параметрично-залежні по трудомісткості алгоритми. Це алгоритми, трудомісткість яких визначається не розмірністю входу (як правило, для цієї групи розмірність входу зазвичай фіксована), а конкретними значеннями оброблюваних слів пам'яті :

$$F_a(D) = F_a(d_1, \dots, d_n) = F_a(P_1, \dots, P_m), m \leq n$$

Прикладами алгоритмів з параметрично-залежною трудомісткістю є алгоритми обчислення стандартних функцій із заданою точністю шляхом обчислення відповідних статечних рядів. Очевидно, що такі алгоритми, маючи на вході два числові значення - аргумент функції і точність виконують істотно залежну від значень кількість операцій.

а) Обчислення x^k послідовним множенням $F_a(x, k) = F_a(k)$.

б) Обчислення $e^x = \sum (x^n/n!)$, з точністю до $\xi \Rightarrow F_a = F_a(x, \xi)$

3. Кількісно-параметричні по трудомісткості алгоритми. Проте в більшості практичних випадків функція трудомісткості залежить як від кількості даних на вході, так і від значень вхідних даних, в цьому випадку:

$$F_a(D) = F_a(|D|, P_1, \dots, P_m) = F_a(N, P_1, \dots, P_m)$$

Як приклад можна привести алгоритми чисельних методів, в яких параметрично-залежний зовнішній цикл по точності включає кількісно-залежний фрагмент по розмірності.

4. *Порядково-залежні по трудомісткості алгоритми.* Серед різноманітності параметрично-залежних алгоритмів виділимо ще оду групу, для якої кількість операцій залежить від порядку розташування початкових об'єктів.

Нехай безліч D складається з елементів $(d_1, \dots, d_n)$, і $\|D\|=N$, визначимо $D_p = \{(d_1, \dots, d_n)\}$ -множина усіх впорядкованих N -ок з $d_1, \dots, d_n$, відмітимо, що $|D_p|=n!$. Якщо

$$F_a(iD_p) \neq F_a(jD_p),$$

де $iD_p, jD_p \in D_p$, то алгоритм називатимемо порядково-залежним по трудомісткості.

Прикладами таких алгоритмів можуть служити ряд алгоритмів сортування, алгоритми пошуку мінімуму і максимуму в масиві. Розглянемо детальніше алгоритм пошуку максимуму в масиві S , що містить n елементів, :

$MaxS(S, n; Max)$

$Max \leftarrow S_1$

For $i \leftarrow 2$ to n

if $Max < S_i$ then $Max \leftarrow S_i$

(кількість виконаних операцій привласнення залежить від порядку дотримання елементів масиву)

2 Асимптотичний аналіз функцій

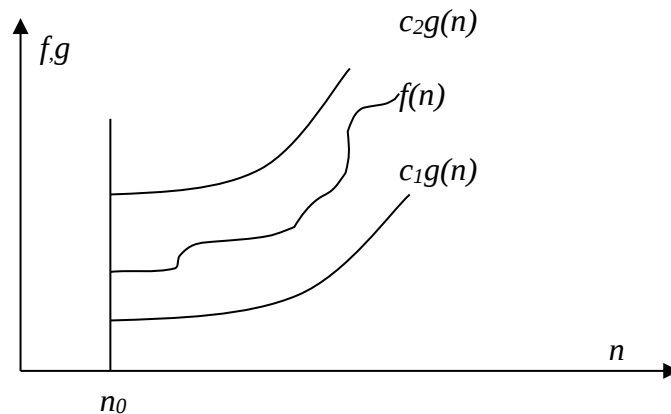
При аналізі поведінки функції трудомісткості алгоритму часто використовують прийняті в математиці асимптотичні позначення, що дозволяють показати швидкість росту функції, маскуючи при цьому конкретні коефіцієнти. Така оцінка функції трудомісткості алгоритму називається складністю алгоритму і дозволяє визначити переваги у використанні того або іншого алгоритму для великих значень розмірності початкових даних. У асимптотичному аналізі прийняті наступні позначення:

Оцінка Θ (тетта). Нехай $f(n)$ і $g(n)$ - позитивні функції позитивного аргументу, $n \geq 1$ (кількість об'єктів на вході і кількість операцій - позитивні числа), тоді:

$$f(n) = \Theta(g(n))$$

якщо існують позитивні c_1, c_2, n_0 , такі, що:

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n), \text{ при } n > n_0$$



Зазвичай говорять, що при цьому функція $g(n)$ є асимптотично точною оцінкою функції $f(n)$, оскільки за визначенням функція $f(n)$ не відрізняється від функції $g(n)$ з точністю до постійного множника. Відмітимо, що з $f(n) = \Theta(g(n))$ виходить, що $g(n) = \Theta(f(n))$.

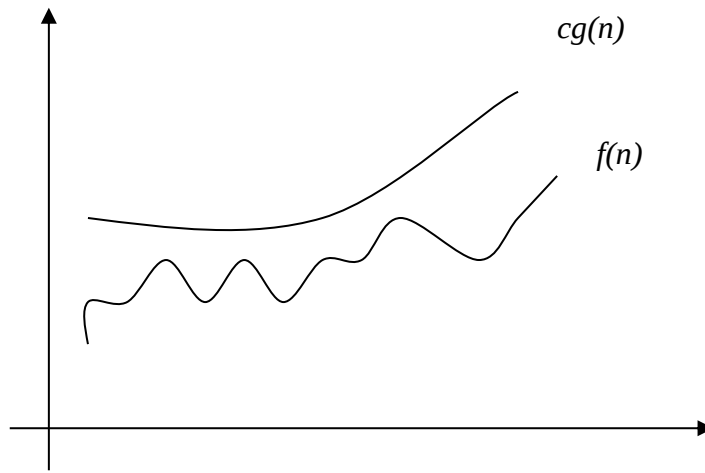
Приклади:

1) $f(n) = 4n^2 + n \ln n + 174 - f(n) = \Theta(n^2)$;

2) $f(n) = \Theta(1)$ - запис означає, що $f(n)$ або дорівнює константі, не рівній нулю, або $f(n)$ обмежена константою на ∞ : $f(n) = 7 + 1/n = \Theta(1)$.

.

Оцінка O (O велике). На відміну від оцінки Θ , оцінка O вимагає тільки, щоб функція $f(n)$ не перевищувала $g(n)$ починаючи з $n > n_0$, з точністю до постійного множника:



$$\exists c > 0, n_0 > 0 :$$

$$0 \leq f(n) \leq c * g(n), \forall n > n_0$$

Взагалі, запис $O(g(n))$ означає клас функцій, таких, що усі вони ростуть не швидше, ніж функція $g(n)$ з точністю до постійного множника, тому іноді говорять, що $g(n)$ мажорує функцію $f(n)$.

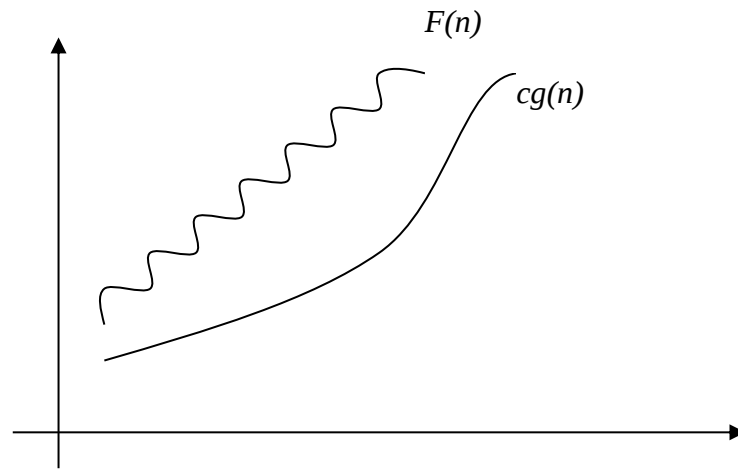
Наприклад, для усіх функцій:

$$f(n)=1/n, f(n)=12, f(n)=3*n+17, f(n)=n*Ln(n), f(n)=6*n^2 + 24*n + 77$$

буде справедлива оцінка $O(n^2)$

Вказуючи оцінку Про є сенс вказувати найбільш "близьку" мажоруючу функцію, оскільки наприклад для $f(n)=n^2$ справедлива оцінка $O(2^n)$, проте вона не має практичного сенсу.

Оцінка Ω (Омега). На відміну від оцінки O , оцінка Ω є оцінкою знизу - тобто визначає клас функцій, які ростуть не повільніше, ніж $g(n)$ з точністю до постійного множника, :



$$\exists c > 0, n_0 > 0 :$$

$$0 \leq c * g(n) \leq f(n)$$

Наприклад, запис $\Omega(n * \ln(n))$ означає клас функцій, які ростуть не повільніше, ніж $g(n) = n * \ln(n)$, в цей клас потрапляють усі поліноми з мірою більшої одиниці, так само як і усі статичні функції з основою великим одиниці.

Відмітимо, що не завжди для пари функцій справедливе одне з асимптотичних співвідношень, наприклад для $f(n) = n^{1+\sin(n)}$ і $g(n) = n$ не виконується жодне з асимптотичних співвідношень.

У асимптотичному аналізі алгоритмів розроблені спеціальні методи отримання асимптотичних оцінок, особливо для класу рекурсивних алгоритмів. Очевидно, що Θ оцінка являється більш кращою, ніж оцінка O . Знання асимптотики поведінки функції трудомісткості алгоритму - його складності, дає можливість робити прогнози по вибору раціональнішого з точки зору трудомісткості алгоритму для великої розмірності початкових даних.

3 Трудомісткість алгоритмів і часові оцінки

3.1 Елементарні операції в мові запису алгоритмів

Для отримання функції трудомісткості алгоритму, представленого у формальній системі введеної абстрактної машини необхідно уточнити поняття "елементарних" операцій, співвіднесених з мовою високого рівня.

Як такі "елементарні" операції пропонується використовувати наступні:

- 1) Просте привласнення: $a \leftarrow b$;
- 2) Одномірна індексація $a[i]$: (адрес $(a) + i * \text{довжина елемента}$);
- 3) Арифметичні операції: $(*, /, -, +)$;
- 4) Операції порівняння: $a < b$;
- 5) Логічні операції: $(l1) \{or, and, not\} (l2)$;

Спираючись на ідеї структурного програмування, виключимо команду переходу за адресою, вважаючи її пов'язаною з операцією порівняння в конструкції галузнення.

Після введення елементарних операцій аналіз трудомісткості основних алгоритмічних конструкцій в загальному вигляді зводиться до наступних положень:

A) Конструкція "Послідовність"

Трудомісткість конструкції є сума трудоемкостей блоків, наступних один за одним.

$$F_{\text{«следование»}} = f_1 + \dots + f_k \text{ где } k - \text{кількість блоків.}$$

B) Конструкція «Розгалуження»

if (l) then

f_{then} з верогідністю p

else

f_{else} з верогідністю $(1-p)$

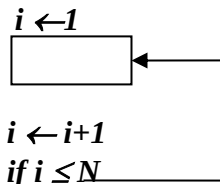
Загальна трудомісткість конструкції "Розгалуження" вимагає аналізу вірогідності виконання переходів на блоки "Then" і "Else" і визначається як:

$$F_{\text{«ветвление»}} = f_{\text{then}} * p + f_{\text{else}} * (1-p).$$

C) Конструкція «Цикл»

for i ← 1 to N

$\Leftrightarrow$
end



Після зведення конструкції до елементарних операцій її трудомісткість визначається як:

$$F_{\text{«цикл»}} = 1 + 3 * N + N * f_{\text{«тела цикла»}}$$

3.2 Приклади аналізу простих алгоритмів

Приклад 1 Завдання підсумовування елементів квадратної матриці

SumM (A, n; Sum)

Sum $\leftarrow$ 0

For *i* $\leftarrow$ 1 to *n*

For *j* $\leftarrow$ 1 to *n*

Sum $\leftarrow$ *Sum* + *A*[*i*,*j*]

end for

Return (*Sum*)

End

Алгоритм виконує однакову кількість операцій при фіксованому значенні *n*, і отже є кількісно-залежним. Застосування методики аналізу конструкції "Цикл " дає:

$FA(n) = 1 + 1 + n * (3 + 1 + n * (3 + 4)) = 7n^2 + 4n + 2 = ((n^2))$, помітимо, що під *n* розуміється лінійна розмірність матриці, тоді як на вхід алгоритму подається *n*² значень.

Приклад 2 Завдання пошуку максимуму в масиві

MaxS (S,n; Max)

Max $\leftarrow$ *S*[1]

For *i* $\leftarrow$ 2 to *n*

if *Max* < *S*[*i*]

then *Max* $\leftarrow$ *S*[*i*]

end for

return *Max*

End

Цей алгоритм є кількісно-параметричним, тому для фіксованої розмірності початкових даних необхідно проводити аналіз для гіршого, кращого і середнього випадку.

А). Гірший випадок

Максимальна кількість переприсвоювань максимуму (на кожному проході циклу) буде у тому випадку, якщо елементи масиву відсортовані за збільшенням. Трудомісткість алгоритму в цьому випадку рівна:

$$F_A^{\wedge}(n)=1+1+1+(n-1)(3+2+2)=7n-4=\Theta(n).$$

Б) Кращий випадок

Мінімальна кількість переприсвоювання максимуму (жодного на кожному проході циклу) буде у тому випадку, якщо максимальний елемент розташований на першому місці в масиві. Трудомісткість алгоритму в цьому випадку рівна:

$$F_A^{\vee}(n)=1+1+1+(n-1)(3+2)=5n-2=\Theta(n).$$

В) Середній випадок

Алгоритм пошуку максимуму послідовно перебирає елементи масиву, порівнюючи поточний елемент масиву з поточним значенням максимуму. На черговому кроці, коли є видимим k -ий елемент масиву, переприсвоювання максимуму станеться, якщо в підмасиві з перших до елементів максимальним елементом є останній. Очевидно, що у разі рівномірного розподілу початкових даних, вірогідність того, що максимальний з k елементів розташований в певній (останньої) позиції рівна $1/k$. Тоді в масиві з n елементів загальна кількість операцій переприсвоювання максимуму визначається як:

$$\sum_{i=1}^N 1/i = Hn \approx \ln(N) + \gamma, \quad \gamma \approx 0,57$$

Величина Hn називається n -им гармонійним числом. Таким чином, точне значення (математичне очікування) середньої кількості операцій привласнення в алгоритмі пошуку максимуму в масиві з n елементів визначається величиною Hn (на нескінченності кількості випробувань), тоді:

$$\bar{F}_A(n)=1+(n-1)(3+2)+2(\ln(n)+\gamma)=5n+2\ln(n)-4+2\gamma=\Theta(n).$$

3.3 Перехід до часових оцінок

Порівняння двох алгоритмів по їх функції трудомісткості вносить деяку помилку в отримувані результати. Головною причиною цієї помилки є різниця частоти зустрічання елементарних операцій, що породжується різними алгоритмами і

відмінність в часах виконання елементарних операцій на реальному процесорі. Таким чином, виникає завдання переходу від функції трудомісткості до оцінки часу роботи алгоритму на конкретному процесорі:

Дано: $F_A(D_A)$ – трудомісткість алгоритму. Необхідно визначити час роботи програмної реалізації алгоритму – $T_A(D_A)$.

На шляху побудови тимчасових оцінок ми стикаємося з цілим набором різних проблем, пофакторний облік яких викликає істотні труднощі. Вкажемо основні з цих проблем:

- неадекватність формальної системи запису алгоритму і реальної системи команд процесора;
- наявність архітектурних особливостей істотна що впливають на спостережуваний час виконання програми, таких як конвеєр, хешування пам'яті, передвибірка команд і даних, і так далі;
- різні часи виконання реальних машинних команд;
- відмінність в часі виконання однієї команди, залежно від значень операндів
- різні часи реального виконання однорідних команд залежно від типів даних;
- неоднозначності компіляції початкового тексту, обумовлені як самим компілятором, так і його налаштуваннями.

Спроби різного підходу до обліку цих чинників привели до появи різноманітних методик переходу до тимчасових оцінок.

1) Післяопераційний аналіз

Ідея післяопераційного аналізу полягає в отриманні післяопераційної функції трудомісткості для кожної з використовуваних алгоритмом елементарних операцій з урахуванням типів даних. Наступним кроком є експериментальне визначення середнього часу виконання цієї елементарної операції на конкретній обчислювальній машині. Очікуваний час виконання розраховується як сума творів післяопераційної трудомісткості на середні часи операцій :

$$T_A(N) = \sum F_{aoni}(N) * \bar{t}_{oni}$$

2) Метод Гиббсона

Метод припускає проведення сукупного аналізу по трудомісткості і перехід до тимчасових оцінок на основі приналежності вирішуваної задачі до одного з наступних типів :

- завдання науково-технічного характеру з переважанням операцій з операндами дійсного типу;
- завдання дискретної математики з переважанням операцій з операндами цілого типу
- завдання баз даних з переважанням операцій з операндами строкового типу

На основі отриманої інформації оцінюється загальний час роботи алгоритму у виді:

$$T_A(N) = F_A(N) * \bar{t}_{тип\ задачі}$$

3) Метод прямого визначення середнього години

У цьому методі так само проводиться сукупний аналіз по трудомісткості - визначається $F_A(N)$, після чого на основі прямого експерименту для різних значень N визначається середній час роботи цієї програми T і на основі відомої функції трудомісткості розраховується середній час на узагальнену елементарну операцію, що породжується цим алгоритмом, компілятором і комп'ютером, - t_a . Ці дані можуть бути (в припущенні про стійкість середнього часу по N) інтерпольовані або екстрапольовані на інші значення розмірності завдання таким чином:

$$\bar{t}_a = T_9(N_9) / F_A(N_9), T(N) = \bar{t}_a * F_A(N).$$

Питання та завдання до контролю знань студентів

1. Класифікація алгоритмів по виду функції трудомісткості;
2. Приклади кількісних і параметрично-залежних алгоритмів;
3. Позначення в асимптотичному аналізі функцій;
4. Приклади функцій, не пов'язаних асимптотичними позначеннями;
5. Елементарні операції в псевдомові високого рівня;

6. Аналіз трудомісткості основних алгоритмічних конструкцій;
7. Побудова функції трудомісткості для підсумовування матриці;
8. Побудова функції трудомісткості для завдання пошуку максимуму ;
9. Проблеми при переході від трудомісткості до тимчасових оцінок;
10. Методики переходу від функції трудомісткості до тимчасових оцінок;
11. Можливості післяопераційного аналізу алгоритмів на прикладі завдання множення комплексних чисел;
12. Теоретична межа трудомісткості завдання;

Лекція № 23

з навчальної дисципліни «Алгоритми та структури даних»

Тема Методи аналізу рекурсивних алгоритмів.

Мета заняття Показати методи обчислення функції трудомісткості рекурсивних алгоритмів

Матеріально-технічне забезпечення та дидактичні засоби, ТЗН

Конспект лекцій.

Час – 2 години

План проведення лекції

- 1 Механізм рекурсивного виклику процедур та функцій
- 2 Аналіз трудомісткості механізму виклику процедури
- 3 Методи рішення рекурсивних співвідношень
- 4 Оцінка трудомісткості рекурсивних алгоритмів

Література

1. Крєневич А.П. Алгоритми і структури даних. Підручник.[Текст]: – А.П. Крєневич. – К.: ВПЦ "Київський Університет", 2021. – 200 с.
2. Коротисєва Т.О. Алгоритми та структури даних: навч. посібник. [Текст]: - Львів : Видавництво Львівська політехніка, 2014. – 280 с.
3. Алгоритми, дані і структури. [Текст], навч. посіб. / В.М. Ільман, О.П. Іванов, Л.О. Панік. Дніпровський. нац.. ун-т залізн. трансп .ім. акад. В. Лазаряна. – Дніпро, 2019. – 134 с

Навчальні матеріали лекції

1 Механізм рекурсивного виклику процедур та функцій

Більшість сучасних мов високого рівня підтримують механізм рекурсивного виклику, коли функція, як елемент структури мови програмування, повертає вичислене значення по своєму імені, може викликати сама себе з іншим аргументом. Ця можливість дозволяє безпосередньо реалізовувати рекурсивне обчислення певних функцій.

Розглянемо класичний приклад рекурсивної функції, що обчислює факторіал натурального числа: $F(n)$;

If $n=0$ or $n=1$ (перевірка можливості прямого обчислення)

Then $F \leftarrow 1$ Else

*$F (n * F(n - 1));$ (рекурсивний виклик функції)*

Return (F);

End;

Аналіз трудомісткості рекурсивних реалізацій алгоритмів, очевидно, пов'язаний як з кількістю операцій, що виконуються при одному виклику функції, так і з кількістю таких викликів. Графічне представлення породжуваного цим алгоритмом ланцюжка рекурсивних викликів називається деревом рекурсивних викликів. Детальніший розгляд призводить до необхідності обліку витрат як на організацію виклику функції і передачі параметрів, так і на повернення вичислених значень і передачу управління в точку виклику.

Можна помітити, що деяка гілка дерева рекурсивних викликів обривається досягнувши такого значення параметра, що передається, при якому функція може бути обчислена безпосередньо. Таким чином, рекурсія еквівалентна конструкції циклу, в якому кожен прохід є виконання рекурсивної функції із заданим параметром.

Розглянемо приклад для функції обчислення факторіалу (рисунок 1) :

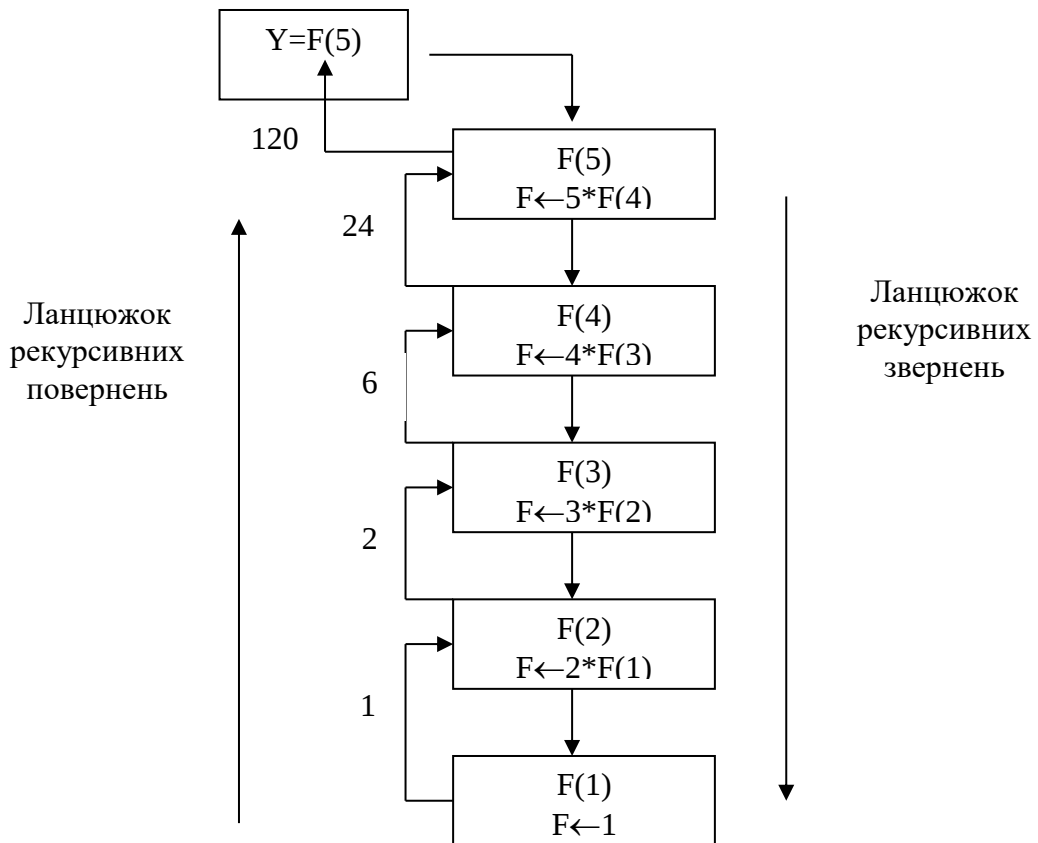


Рис.1. Дерево рекурсії при обчисленні факторіалу - $F(5)$

Дерево рекурсивних викликів може мати і складнішу структуру, якщо на кожному виклику породжується декілька звернень - фрагмент дерева рекурсій для чисел Фібоначчі представлений на рисунку 2:

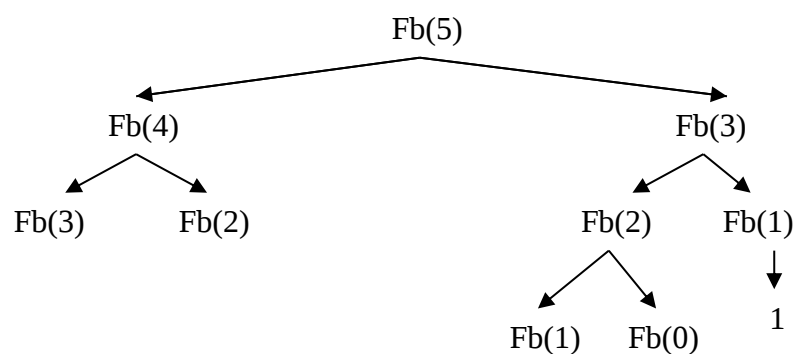


Рис. 2. Фрагмент дерева рекурсії при обчисленні чисел Фібоначчі - $F(5)$

2 Аналіз трудомісткості механізму виклику процедури

Механізм виклику функції або процедури в мові високого рівня істотно залежить від архітектури комп'ютера і операційної системи. У рамках IBM PC сумісних комп'ютерів цей механізм реалізований через програмний стек. Як фактичні параметри, що передаються в процедуру або функцію, так і параметри, в яких повертаються обчислені процедурою або функцією значення, поміщаються в програмний стек спеціальними командами процесора. Додатково зберігаються значення необхідних регістрів і адреса повернення в зухвалу процедуру. Схематично цей механізм ілюстрований на рисунку 3.

Для підрахунку трудомісткості виклику рахуватимемо операції приміщення слова в стек і виштовхування із стека елементарними операціями у формальній системі. Тоді при виклику процедури або функції в стек поміщається адреса повернення, стан необхідних регістрів процесора, адреси повернутих значень і передавані параметри.

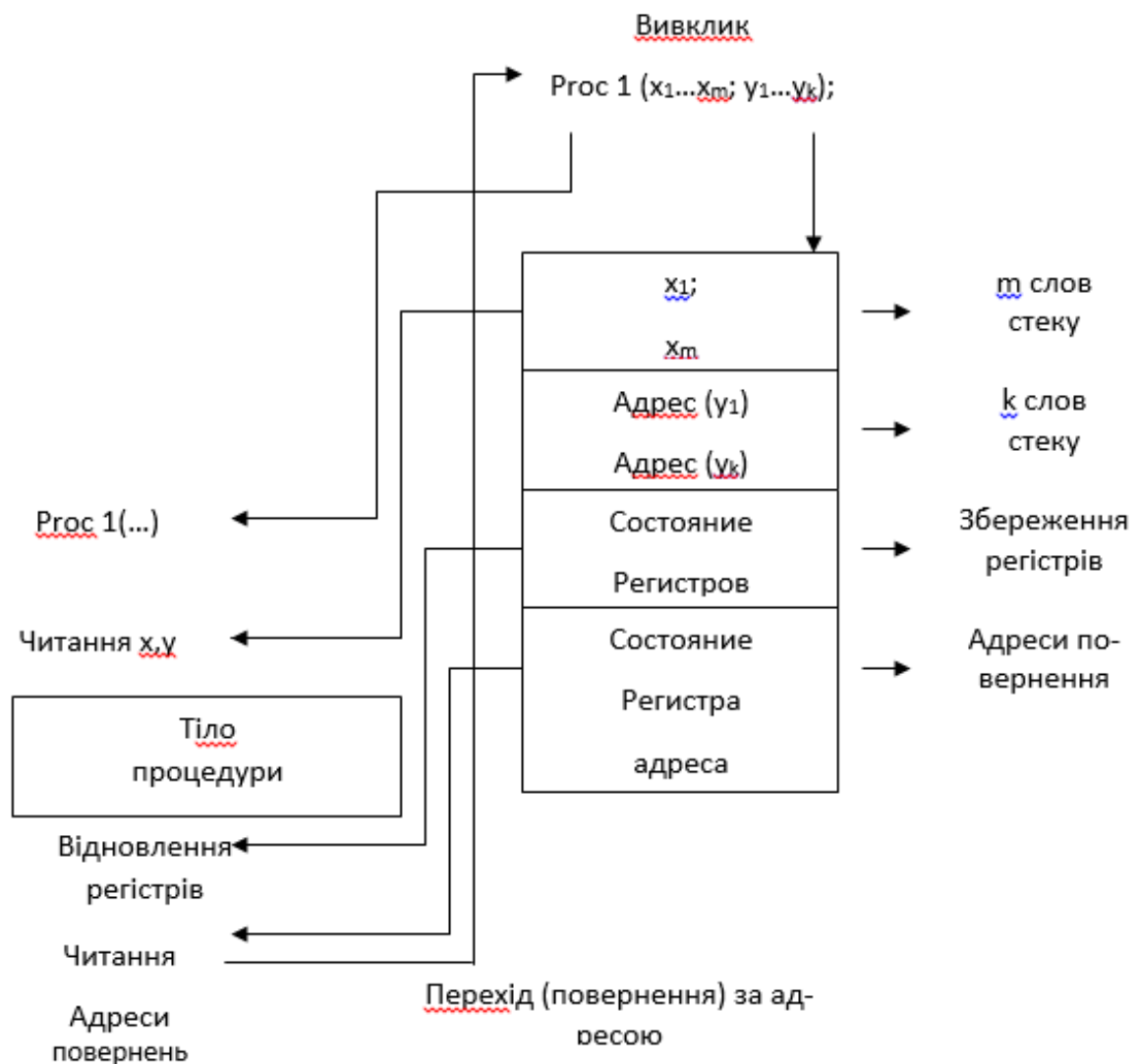


Рис. 3. Механізм виклику процедури з використанням програмного стеку.

Після цього виконується перехід за адресою на процедуру, що викликається, яка витягає передані фактичні параметри, виконує обчислення, поміщає їх по вказаних в стеку адресах, і при завершенні роботи відновлює регістри, виштовхує із стека адресу повернення і здійснює перехід за цією адресою. Позначивши через:

m - кількість передаваних фактичних параметрів

k - кількість повернутих процедурою значень

r - кількість регістрів, що зберігаються в стеку, маємо:

$f_{\text{виклику}} = m+k+r+1+m+k+r+1 = 2*(m+k+r+1)$ елементарних операцій на один виклик і повернення.

Аналіз трудомісткості рекурсивних алгоритмів в частині трудомісткості самого рекурсивного виклику можна виконувати різними способами залежно від того, як формується підсумкова сума елементарних операцій - розглядом окремо ланцюжка рекурсивних викликів і повернень, або сукупно по вершинах дерева рекурсивних викликів.

3 Методи рішення рекурсивних співвідношень

У математиці розроблений ряд методів, за допомогою яких можна отримати явний вид рекурсивно заданої функції, - метод індукції, формальні статечні ряди, метод ітерацій і так далі. Розглянемо деякі з них:

а) Метод індукції. Метод полягає в тому, що б спочатку вгадати рішення, а потім довести його правильність по індукції. Приклад:

$$\begin{cases} f(0)=1 \\ f(n+1)=2*f(n) \end{cases}$$

Припущення: $f(n)=2^n$

Базис: якщо $f(n)=2^n$, то $f(0)=1$, що виконується за визначенням.

Індукція: Нехай $f(n)=2^n$, тоді для $n+1 \Rightarrow f(n+1)=2 * 2^n = 2^{n+1}$

Відмітимо, що базис істотно впливає на рішення, так наприклад:

Якщо $f(0)=0$, то $f(n)=0$;

якщо $f(0)=1/7$, то $f(n)=(1/7)*2^n$;

якщо $f(0)=1/64$, то $f(n)=(2)^{n-6}$

б) Метод ітерації (підстановки). Суть методу полягає в послідовній підстановці - ітерації рекурсивного визначення, з наступним виявленням загальних закономірностей.

Нехай $f(n)=3*f(n/4)+n$, тоді:

$$f(n)=n+3*f(n/4)=n+3*[n/4+3*f(n/16)]=n+3*[n/4+3\{n/16+3*f(n/64)\}],$$

І, розкриваючи дужки, отримаємо:

$$f(n)=n+3*n/4+9*n/16+27*n/64+\dots+3^i*n/4^i$$

Зупинка рекурсії відбудеться при $(n / 4^i) \leq 1 \Rightarrow i \geq \log_4 n$, в цьому випадку останній доданок не більше, чим $3^{\log_4 n} * \Theta(1) = n^{\log_4 3} * \Theta(1)$.

$f(n) = n * \sum (3/4)^k + n^{\log_4 3} * \Theta(1)$, т.к. $\sum (3/4)^k = 4$, то остаточно:

$$f(n) = 4 * n + n^{\log_4 3} * \Theta(1) = \Theta(n)$$

4 Оцінка трудомісткості рекурсивних алгоритмів

Основний метод побудови рекурсивних алгоритмів - це метод декомпозиції. Ідея методу полягає в розподілі завдання на частини меншої розмірності, отримання рішення для отриманих частин і об'єднання рішень.

У загальному вигляді, якщо відбувається розподіл завдання на b підзадач, яке призводить до необхідності рішення a підзадач розмірністю n/b , то загальний вигляд функції трудомісткості має вигляд:

$$f_A(n) = a * f_A(n/b) + d(n) + U(n)$$

де $d(n)$ – трудомісткість алгоритму ділення задачі на підзадачі,

$U(n)$ – трудомісткість алгоритму об'єднання отриманих рішень.

Розглянемо, наприклад, відомий алгоритм сортування злиттям:

На кожному рекурсивному виклику переданий масив ділиться навпіл, що дає оцінку для $d(n) = (1)$, далі рекурсивно викликаємо сортування отриманих масивів половинної довжини (до тих пір, поки довжина масиву не стане рівній одиниці), і зливаємо повернені відсортовані масиви за $\Theta(n)$.

Тоді очікувана трудомісткість на сортування складе:

$$f_A(n) = 2 * f_A(n/2) + \Theta(1) + \Theta(n)$$

Тим самим виникає завдання про отримання оцінки складності функції трудомісткості, для довільних значень a і b .

Ця теорема є потужним засобом аналізу асимптотичної складності рекурсивних алгоритмів, на жаль, вона не дає можливості отримати повний вид функції трудомісткості.

Питання та завдання до контролю знань студентів

1. В чому полягає механізм рекурсивного виклику процедур та функцій?
2. Як проводиться аналіз трудомісткості механізму виклику рекурсивної процедури?
3. В чому полягає метод індукції рішення рекурсивних співвідношень?
4. В чому полягає метод ітерації (підстановки) рішення рекурсивних співвідношень?
5. Як проводиться аналіз трудомісткості алгоритму при використанні методу декомпозиції?