МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ
Кафедра інформаційних технологій

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Мірошник М.А., Григор'єва Т.І.

Конспект лекцій

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету

(протокол № 5 від 20.01.2025 р.)

Мірошник М. А., Григор'єва Т.І.
Алгоритми та структури даних: Конспект лекцій для здобувачів вищої освіти, які навчаються за спеціальностю «Середня освіта. Інформатика» [Електронне видання]. / Мірошник М.А., Григор'єва Т.І. Кафедра інформаційних технологій Міжнародного гуманітарного університету. Одеса, 2025. – 49 с.

Методичні рекомендації з курсу «**Алгоритми та структури даних**» розроблено відповідно до навчального плану, вони складаються з навчальної програми курсу, методичних рекомендацій з проведення практичних занять і завдань для самостійної роботи здобувачів, списку рекомендованої літератури. Матеріали призначено для студентів факультету кібербезпеки, програмної інженерії та комп'ютерних наук Міжнародного гуманітарного університету, які в бакалавраті вивчають спеціальність «Середня освіта (Інформатика та програмування)».

ПРОГРАМА НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Змістовий модуль 1. Базові структури даних: стеки, черги, зв'язані списки, кеш-таблиці, дерева, графи.

Тема 1. Базові структури даних: вектор, стек, черга.

Тема 2. Базові структури даних: зв'язані списки.

Тема 3. Базові структури даних: дерева.

Тема 4. Сбалансовані дерева.

Тема 5. Хеш-таблиці.

Змістовий модуль 2. Основні обчислювальні алгоритми: сортування, хеш-таблиці та алгоритми виключення колізій, двійкові дерева пошуку, представлення графів, обхід в глибину та в ширину.

Тема 6. Масиви, алгоритми сортування.

Тема 7. Алгоритми швидкого сортування.

Тема 8. Динамічне програмування.

Тема 9. «Жадібні» алгоритми.

Тема 10. Графи: подання, обходи, топологічне сортування.

Тема 11. Графи: сильно зв'язані компоненти, остовні дерева.

Тема 12. Графи: найкоротші шляхи.

Змістовий модуль 3. Рекурсія.

Тема 13. Рекурсивні алгоритми.

Змістовий модуль 4. Аналіз алгоритмів.

Тема 14. Складність алгоритму, асимптотичні функції.

Тема 15. Аналіз рекурсивних та не рекурсивних алгоритмів.

КОНСПЕКТ ЛЕКЦІЙ
дисципліни «Алгоритми та структури даних»

Тема 1. Базові структури даних: вектор, стек, черга

Під структурою даних в загальному випадку розуміють множина елементів даних і множина зв'язків між ними. Класифікація структур даних здійснюється за різними ознаками. Мінливість - зміна числа елементів і (або) зв'язків між елементами структури. У визначенні мінливості структури не відображений факт зміни значень елементів даних, оскільки в цьому випадку все структури даних мали б властивість мінливості. За ознакою мінливості розрізняють структури статичні, полустатичні, динамічні. Базові структури даних, статичні, полустатичні і динамічні характерні для оперативної пам'яті і часто називаються оперативними структурами. Файлові структури відповідають структурам даних для зовнішньої пам'яті.

Характер впорядкованості елементів. За цією ознакою структури можна ділити на лінійні і нелінійні структури.

Статичні структури даних.

Вектор (одновимірний масив) - структура даних з фіксованим числом елементів одного і того ж типу. Кожен елемент вектору має унікальний в рамках заданого вектору номер. Звернення до елементу вектору виконується по імені вектору і номером необхідного елемента.

Масиви. Масив - структура даних, яка характеризується: фіксованим набором елементів одного і того ж типу; кожен елемент має унікальний набір значень індексів; кількість індексів визначають рівномірність масиву; звернення до елементу масиву виконується по імені масиву і значенням індексів для даного елемента. Інше визначення: масив - це вектор, кожен елемент якого - вектор.

Множина.

Множина - структура, яка представляє собою набір неповторюваних даних одного і того ж типу.

Записи. Запис - кінцеве впорядкована множина полів, що характеризуються різним типом даних. Записи є надзвичайно зручним засобом для подання програмних моделей реальних об'єктів предметної області, бо, як правило, кожен такий об'єкт має набір властивостей, що характеризуються даними різних типів. Приклад запису - сукупність відомостей про деяке студента. Таблиці Полями записи можуть бути інтегровані структури даних - вектори, масиви, інші записи. Аналогічно і елементами векторів і масивів можуть бути також інтегровані структури. Одна з таких складних структур - таблиця. З фізичної точки зору таблиця являє собою вектор, елементами якого є записи. Характерною логічної особливістю таблиць, яка і визначила їх розгляд в окремому розділі, є те, що доступ до елементів таблиці проводиться не по номеру (індексу), а по ключу - за значенням одного з властивостей об'єкта, що описується записом-елементом таблиці. Ключ - це властивість, що ідентифікує даний запис в множинаі однотипних записів. Як правило, до ключу ставиться вимога унікальності в даній таблиці. Таблиця може мати один або кілька ключів. Наприклад, при інтеграції в таблицю записів про студентів вибірка може проводитися як за особистим номером студента, так і на прізвище.

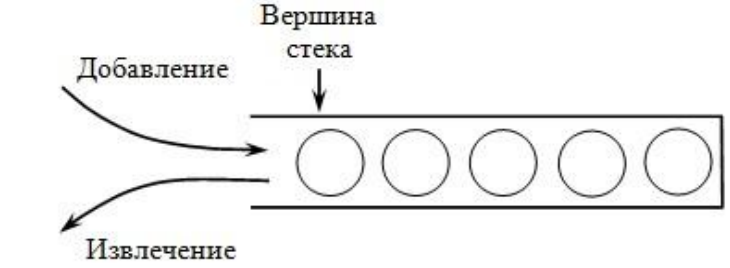
Операції логічного рівня над статичними структурами.

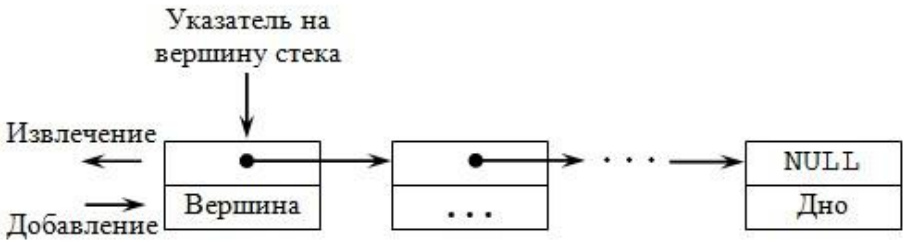
Пошук O(n), O(log n) Сортування, O(n), O(n*log n), O(n^2).

Стек - структура даних, що представляє собою список елементів, доступ до яких здійснюється за принципом LIFO, включення і виключення елементів списку виконуються тільки з одного боку списку, званого вершиною стека.

Функціонує за принципом LIFO (Last - In - First- Out - "останнім прийшов - першим вийшов"). Приклади стека: стопка книг.

Основні операції над стеком - включення нового елемента (англійська назва push - заштовхувати), виключення елемента зі стека (англ. pop - вискакувати).

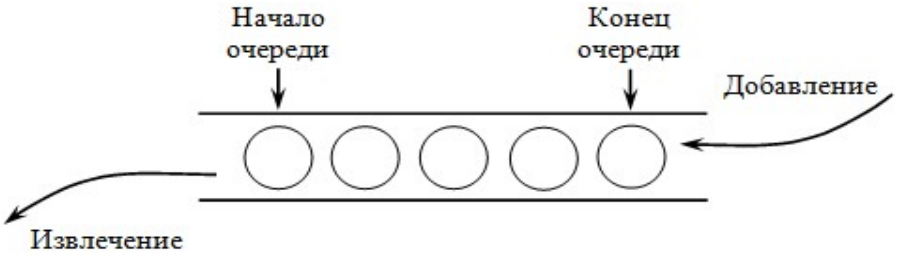


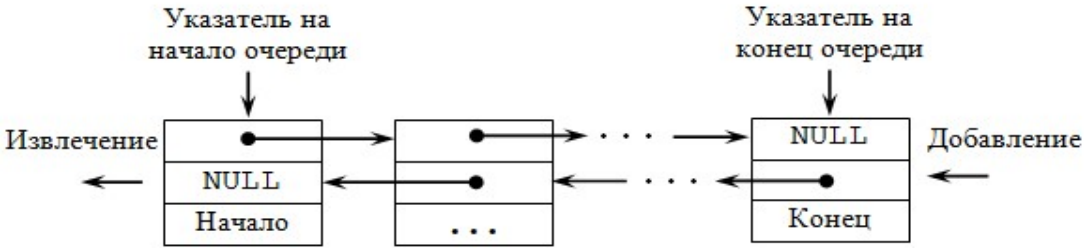



```
class Stack
    {
        Object[] s;
        int top;
int Max_count;
        public Stack(int m)
        {
            Max_count = m;
s = new Object[Max_count];
top = 0;

        }
        public void push(Object inf)
        {
            if (top<Max_count)s[top++] = inf;
        }
        public Object pop()
        {
            if (top > 0) { return
s[--top]; }             else
return(-1);
        }
```

Черга - це структура даних, що представляє собою послідовність елементів, утворена в порядку їх надходження. Кожен новий елемент розміщується в кінці черги; елемент, що стоїть на початку черги, вибирається з неї першим. У черзі використовується принцип доступу до елементів FIFO (First Input - First Output, "перший прийшов - першим вийшов"). У черзі доступні два елементи (дві

позиції): початок черги і кінець черги. Помістити елемент можна тільки в кінець черги, а взяти елемент тільки з її початку. Прикладом може служити звичайна чергу в магазині.





При поданні черги вектором в статичної пам'яті на додаток до звичайних для дескриптора вектора параметрів в ньому повинні знаходитися два покажчика: на початок черги (на перший елемент в черзі) і на її кінець (перший вільний елемент в черзі).

При включенні елемента в чергу елемент записується за адресою, що визначається покажчиком на кінець, після чого цей покажчик збільшується на одиницю. При виключення елемента з черги вибирається елемент, що адресується покажчиком на початок, після чого цей покажчик збільшується на одиницю.

Для того, щоб місця, займані виключеними елементами, могли бути повторно використані, чергу замикається в кільце: покажчики (на початок і на кінець), досягнувши кінця виділеної області пам'яті, переключаються на її початок. Така організація черги в пам'яті називається кільцевої чергою.

У початковому стані покажчики на початок і на кінець вказують на початок області пам'яті. Рівність цих двох покажчиків (при будь-якому їх значенні) є ознакою порожньої черги. Якщо в процесі роботи з кільцевої чергою число операцій включення перевищує число операцій виключення, то може виникнути ситуація, в якій покажчик кінця "наздожене" покажчик початку. Це ситуація заповненої черги, але якщо в цій ситуації покажчики зрівняються, ця ситуація буде відрізнити від ситуації порожній черзі. Для розрізнення цих двох ситуацій до кільцевої черзі ставиться вимога, щоб між покажчиком кінця і покажчиком початку залишався "зазор" з вільних елементів. Коли цей "зазор" скорочується до одного елемента, чергу вважається заповненою і подальші спроби записи в неї блокуються

class Queue

```
    {
int[] data;
        int first, last;
        public Queue(int c)
        {
            data = new
int[c];
first = last = 0;
```

```
        }
        public bool empty()
        {
            return first == last;
        }
       public int deque()
        {
           if
(empty())
return 0;
else
{
                int x=data[first];
                first=(first+1)%data.Length;
                return x;
            }
        }
```

Черги з пріоритетами

Порядок вибірки елементів з таких черг визначається пріоритетами елементів. Пріоритет в загальному випадку може бути представлений числовим значенням. При вибірці елемента щоразу вибирається елемент з найбільшим пріоритетом.

Черги з пріоритетами можуть бути реалізовані на лінійних спискові структурах - в суміжному або зв'язковому поданні. Можливі черзі з пріоритетним включенням - в яких послідовність елементів Черга підтримується впорядкованої, тобто кожен новий елемент включається на те місце в послідовності, яка визначається його пріоритетом, а при виключенні завжди вибирається елемент з початку. Можливі й черги з пріоритетним винятком - новий елемент включається завжди в кінець черги, а при виключенні в черзі шукається (цей пошук може бути тільки лінійним) елемент з максимальним пріоритетом і після вибірки видаляється з послідовності. І в тому, і в іншому варіанті потрібно шукати, а якщо чергу розміщується в статичної пам'яті - ще й переміщення елементів. Найбільш зручною формою для організації великих черг з пріоритетами є піраміда (бінарна купа).

Тема 2. Структури даних: зв'язані списки.

Динамічні структури даних - це структури даних, пам'ять під які виділяється і звільняється в міру необхідності. Пам'ять під динамічні структури даних виділяється і звільняється не на етапі компіляції, а в процесі роботи програми. Динамічні структури даних в процесі існування в пам'яті можуть змінювати не тільки число складових їх елементів, а й зв'язку між елементами.

Для вирішення проблеми адресації динамічних структур даних використовується метод, званий динамічним розподілом пам'яті, тобто пам'ять під окремі елементи виділяється в момент, коли вони "починають існувати" в процесі виконання програми, а не під час компіляції. Компілятор в цьому випадку виділяє фіксований обсяг пам'яті для зберігання адреси динамічно розміщується елемента, а не самого елемента.

Динамічна структура даних характеризується тим що:

- їй виділяється пам'ять в процесі виконання програми;
- кількість елементів структури може не фіксуватися;

- розмірність структури може змінюватися в процесі виконання програми;
- в процесі виконання програми може змінюватися характер взаємозв'язку між елементами структури.

Кожній динамічної структурі даних зіставляється статична змінна типу покажчик (її значення - адреса цього об'єкта), за допомогою якої здійснюється доступ до динамічної структурі. Необхідність в динамічних структурах даних зазвичай виникає коли в процесі роботи програми потрібен масив, список або інша структура, розмір якої змінюється в широких межах і важко передбачуваний.

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

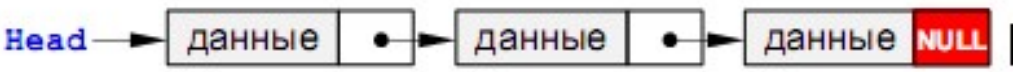
Для встановлення зв'язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв'язки між елементами. Таке уявлення даних в пам'яті називається зв'язковим. Переваги зв'язкового представлення даних - в можливості забезпечення значною мінливості структур:

- розмір структури обмежується тільки доступним об'ємом машинної пам'яті; • при зміні логічної послідовності елементів структури потрібно не переміщення даних в пам'яті, а тільки корекція покажчиків;
- велика гнучкість структури. Разом з тим, чіткий уявлення не позбавлене і недоліків, основними з яких є наступні:
- на поля, що містять покажчики для зв'язування елементів один з одним, збільшують споживання пам'ять;
- доступ до елементів зв'язної структури може бути менш ефективним за часом. Останній недолік є найбільш серйозним і саме їм обмежується можливість застосування зв'язкового представлення даних. Кожна компонента будьдинамічної структури є запис, що містить, принаймні, два поля: інформаційне поле (поля даних), в якому містяться ті дані, заради яких і створюється структура; в загальному випадку інформаційне поле саме є інтегрованою структурою - вектором, масивом, інший динамічною структурою і т.п .; адресне поле (поля зв'язок), в якому містяться один або кілька покажчиків, що зв'язує даний елемент з іншими елементами структури; Для звернення до динамічної структурі досить зберігати в пам'яті адреса першого елемента структури.

Оскільки кожен елемент динамічної структури зберігає адресу наступного за ним елемента, можна, рухаючись від початкового елемента за адресами, отримати доступ до будь-якого елементу даної структури. Списком називається впорядкована множина, що складається з змінного числа пов'язаних елементів, до яких застосовні операції включення, виключення. Списки є спосіб організації структури даних, при якій елементи деякого типу утворюють ланцюжок. Для зв'язування елементів в списку використовують систему покажчиків. У мінімальному випадку, будь-який елемент лінійного списку має один покажчик, який вказує на наступний елемент у списку або є порожнім покажчиком, що інтерпретується як кінець списку. Структура, елементами якої служать записи з одним і тим же форматом, пов'язані один з одним за допомогою покажчиків, що зберігаються в самих елементах, називають пов'язаним списком. Лінійні зв'язні списки є найпростішими динамічними структурами даних.

З усього різноманіття пов'язаних списків можна виділити наступні основні:

- односпрямовані (одинзв'язні) списки; • двонаправлені (двусвязного) списки;
- циклічні (кільцеві) списки.
- Односпрямований (однозв'язний) список - це структура даних, що представляє собою послідовність елементів, в кожному з яких зберігаються дані і покажчик на наступний елемент списку. В останньому елементі покажчик на наступний елемент дорівнює NULL.




```
public class Node
{     public
int Data;
    public Node Next;

    public Node (int i)
    {
        Data=i;
        Next=null;
    }
    public Node(int i, Node n)
    {
        Data = i;
        Next = n;
    }
}
class MyList
{
    public Node Head;
    public void AddF(int x)
    {
        Head = new Node(x,Head);
    }
    public void AddL(int x)
    {
        Node newNode = new
Node(x);          if (Head !=
null)
        {
            Node h = Head;
            while (h.Next != null) h =
h.Next;              h.Next = newNode;
        }
else
        {
            Head = newNode;
        }
    }
```

Видалення елемента з односпрямованого списку. З динамічних структур можна видаляти елементи, так як для цього достатньо змінити значення адресних полів. Операція видалення елемента односпрямованого списку здійснює видалення елемента, на який встановлено покажчик поточного елемента. Після

видалення покажчик поточного елемента встановлюється на попередній елемент списку або на новий початок списку, якщо видаляється перший. Алгоритми видалення першого і наступних елементів списку відрізняються один від одного. Тому у функції, що реалізує дану операцію, здійснюється перевірка, який об'єкт був видалений.

Двохнаправлений (двусвязний) список - це структура даних, що складається з послідовності елементів, кожен з яких містить інформаційну частину і два покажчика на сусідні елементи. При цьому два сусідні елементи повинні містити взаємні посилання один на одного. В такому списку кожен елемент (крім першого і останнього) пов'язаний з попереднім і наступним за ним елементами. Кожен елемент двонаправленого списку має два поля з покажчиками: одне поле містить посилання на наступний елемент, інше поле - посилання на попередній елемент і третє поле - інформаційне. Наявність посилань на наступну ланку і на попереднє дозволяє рухатися по списку від кожної ланки в будь-якому напрямку: від ланки до кінця списку або від ланки до початку списку, тому такий список називають двонаправленим

Видалення елемента з двонаправленого списку.

З динамічних структур можна видаляти елементи, так як для цього достатньо змінити значення адресних полів. Операція видалення елемента з двонаправленого списку здійснюється багато в чому аналогічно видалення з односпрямованого списку.

Тема 3. Базові структури даних: дерева

Дерево є одним з найважливіших і найцікавіших приватних випадків графа. Деревоподібна модель виявляється досить ефективною для подання динамічних даних з метою швидкого пошуку інформації. Дерева є одними з найбільш широко поширених структур даних в інформатиці та програмуванні, які представляють собою ієрархічні структури у вигляді набору пов'язаних вузлів. Дерево - це структура даних, що представляє собою сукупність елементів і відносин, що утворюють ієрархічну структуру цих елементів. Кожен елемент дерева називається вершиною (вузлом) дерева. Вершини дерева з'єднані спрямованими дугами, які називають гілками дерева. Початковий вузол дерева називають коренем дерева, йому відповідає нульовий рівень. Листям дерева називають

вершини, в які входить одна гілка і не виходить жодної гілки. Кожне дерево має такі властивості:

існує вузол, в який не входить ні однієї дуги (корінь);

в кожну вершину, крім кореня, входить одна дуга.

Дерева особливо часто використовують на практиці при зображенні різних ієрархій. Нащадки - це все вершини, в які входять гілки, що виходять з однієї загальної вершини. Предок - це вершина, з якої виходять гілки до вершин наступного рівня. Рівень нащадка на одиницю перевищує рівень його предка. Корінь дерева не має предка, а листя дерева не мають нащадків. Висота (глибина) дерева визначається кількістю рівнів, на яких розташовуються його вершини. Висота порожнього дерева рана нулю, висота дерева з одного кореня - одиниці. На першому рівні дерева може бути тільки одна вершина - корінь дерева, на другому - нащадки кореня дерева, на третьому - нащадки нащадків кореня дерева і т.д. Піддерево - частина древообразная структури даних, яка може бути представлена у вигляді окремого дерева. Ступенем вершини в дереві називається кількість дуг, яке з неї виходить. Ступінь дерева дорівнює максимальному ступені вершини, що входить в дерево. При цьому листям в дереві є вершини, що мають ступінь нуль.

Впорядковане дерево - це дерево, у якого гілки, що виходять з кожної вершини, впорядковані за певним критерієм. Дерева є рекурсивними структурами, так як кожне піддерево також є деревом. Таким чином, дерево можна визначити як рекурсивну структуру, в якій кожен елемент є:

або порожній структурою;

або елементом, з яким пов'язано кінцеве число піддерев.

Дії з рекурсивними структурами найзручніше описуються за допомогою рекурсивних алгоритмів. Для того, щоб виконати певну операцію над усіма вершинами дерева необхідно все його вершини переглянути. Таке завдання називається обходом дерева. Обхід дерева - це впорядкована послідовність вершин дерева, в якій кожна вершина зустрічається тільки один раз. При обході все вершини дерева повинен відвідувати у певному порядку. Існує кілька способів обходу всіх вершин дерева. Тип дерева визначається кількістю зв'язків у кожної вершини

Впорядковане дерево – довільне

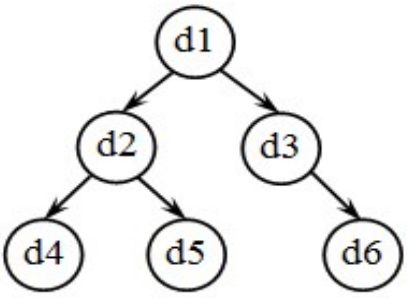
М-арное дерево – М

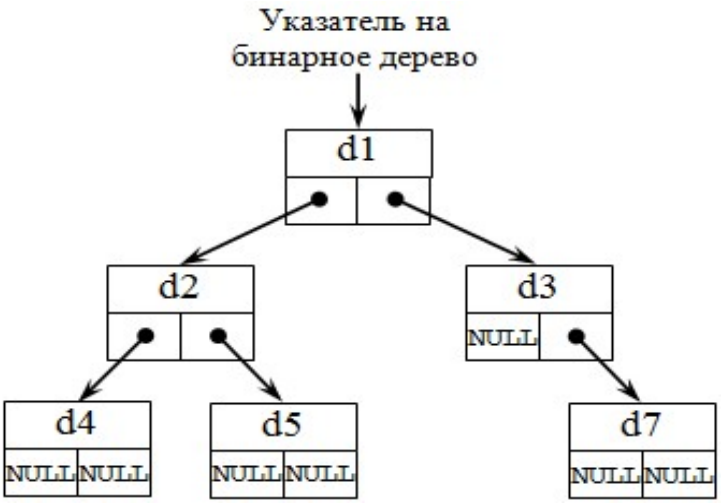
Бінарне дерево – 2

Шлях в дереві - це послідовність вершин, в якій такі друг за другом вершини з'єднуються ребрами дерева.

Між двома вершинами може існувати єдиний шлях

Бінарне (двійкове) дерево - це динамічна структура даних, що представляє собою дерево, в якому кожна вершина має не більше двох нащадків. Таким чином, бінарне дерево складається з елементів, кожен з яких містить інформаційне поле і не більше двох посилань на різні бінарні піддерева. На кожен елемент дерева є рівно одна посилання.





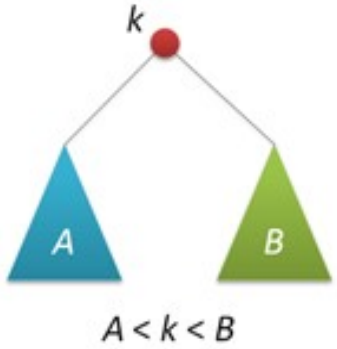
Основними операціями, здійснюваними з бінарними деревами, є:

- вставка елемента в бінарне дерево;
- друк бінарного дерева;
- пошук елемента в бінарному дереві;
- видалення елемента з бінарного дерева.

Двійкові дерева пошуку (binary search tree, BST) У двійковому дереві пошуку кожна вершина може мати (або не мати) ліву та праву дитини, кожна вершина, крім кореня має батька. При поданні з використанням покажчиків ми зберігаємо для кожної вершини дерева, крім значення ключа key також покажчики left, right і p (піддерева ліве, праве і батько). Якщо дитини (або батька - для кореня) немає, відповідне поле містить NULL. Ключі в двоїчному дереві пошуку зберігаються з дотриманням властивості впорядкованості. Нехай x довільна вершина двійкового дерева пошуку.

Якщо вершина y знаходиться в лівому піддереві вершини x, то key [y] <key [x].

Якщо y знаходиться в правому піддереві x, то key [y]> key [x].




```
class Tree
    {
int info;
Tree left;
        Tree
right;
Tree parent;
public Tree()
        {
info = -1;
```

```
left = null;
right = null;
parent = null;
        }
        public Tree(int value)
        {
info = value;
left = null;
right = null;
parent = null;
        }

public void Add(int value)
        {
if (info == -1)
info = value;
            else if (info > value)
            {
if (left == null)
                {
left = new Tree(value);
left.parent = this;
                }
else
left.Add(value);
            }
            else if (info < value)
            {
                if (right == null)
                {
                    right = new
Tree(value);
right.parent = this;
                }
else
right.Add(value);
            }        }
```

Тема 4. Сбалансовані дерева

Майже збалансоване дерево - це дерево, у якого довжини всіляких шляхів від кореня до зовнішніх вершин відрізняються не більше, ніж на одиницю. Збалансоване дерево - це дерево, у якого довжини всіх шляхів від кореня до зовнішніх вершин рівні між собою. Рівень вершини - це кількість дуг від кореня дерева до вершини. Збалансовані дерева пошуку - це дерева пошуку, структура яких підтримується наближеною до структури повного дерева, і висота таких дерев дорівнює O (log (n))

Методи підтримки балансу бінарних дерев пошуку:

рандомізовані дерева

АВЛ-дерево • Червоно-чорні дерева • та інші .

Ідея побудови рандомізованого дерева: якщо заздалегідь перемішати все ключі і потім побудувати з них дерево (ключі вставляються за стандартною схемою в отриманому після перемішування порядку), то побудоване дерево виявиться

досить збалансованим - його висота буде близько 2log2n. В цьому випадку коренем може з однаковою ймовірністю опинитися будь-який з вихідних ключів. Будь-ключ (в тому числі і той, який зараз повинні вставити в дерево) може виявитися коренем з ймовірністю 1 / (n + 1) (n - розмір дерева до вставки). Виконуємо з вказаною ймовірністю вставку в корінь, а з ймовірністю 1-1 / (n+1) - рекурсивную вставку до правої чи лівої поддерево в залежності від значення ключа в корені: Кожен вузол двійкового дерева пошуку міститиме ключ key, і пару покажчиків left і right на ліве і праве піддерева. Якщо якийнебудь покажчик нульовий, то відповідне дерево є порожнім. Для реалізації рандомізованого дерева потрібно ще одне поле size, в якому буде зберігатися розмір дерева з корінням в даному вузлі (кількість вершин). Вузли будемо представляти структурами:

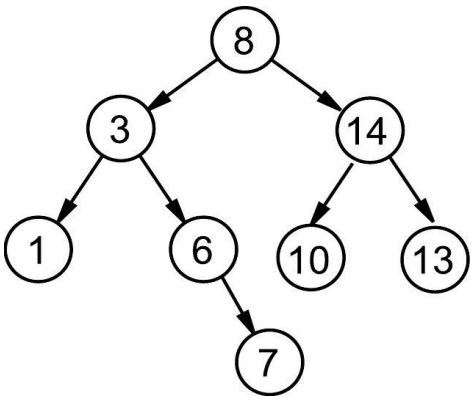
```
struct node // структура для представления узлов дерева
{
     int key;
int size;
node* left;
node* right;
     node(int k) { key = k; left = right = 0;
size = 1; }  };

node* insert(node* p, int k) // рандомизированная вставка нового
узла с ключом k в дерево p
{
     if( !p ) return new node(k);
if( rand()%(p->size+1)==0 )
return insertroot(p,k); // вставка в корень
if( p->key>k )
          p->left = insert(p-
>left,k);           else
          p->right = insert(p->right,k);
     fixsize(p);
    return p;
}
```

Видалення проводиться по ключу - шукаємо вузол з заданим ключем і видаляємо цей вузол з дерева. Пошук вершини з заданим ключем об'єднання лівого і правого піддерева знайденої вершини видалення вершини повертаємо корінь об'єднаного дерева

```
node* remove(node* p, int k) // удаление из дерева p первого
найденного узла с ключом k
{
     if( !p ) return p;
     if( p->key==k )
     {
      node* q = join(p->left,p->right);
          delete p;
          return q;
     }
 else if( k<p->key )
p->left = remove(p->left,k);
else
```

```
        p->right = remove(p->right,k);
    return p;
}
```

АВЛ-дерево - збалансоване по висоті двоичное дерево пошуку: для кожної його вершини висота її двох піддерев розрізняється не більше ніж на 1. Воно названо за першими літерами прізвищ його винахідників, Г. М. Адельсона-Бєльського і Е. М. Ландіса, які вперше запропонували використовувати АВЛ-дерева в 1962.

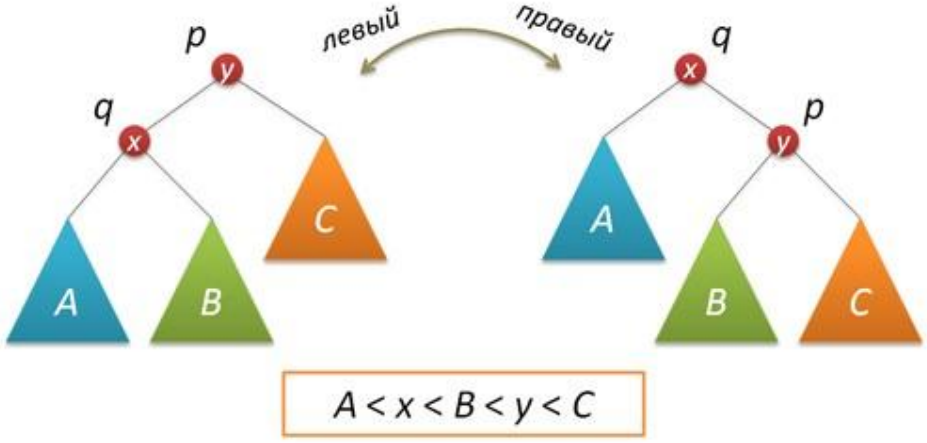


АВЛ-дерево - швидкість роботи. висота h АВЛ-дерева з n ключами лежить в діапазоні от $\log2(n + 1)$ до $1.44 \log2(n + 2) − 0.328$.

$h(n)=O(\log_2 n)$

Основні операції над двійковими деревами пошуку (пошук, вставка і видалення вузлів) лінійно залежать від його висоти, отримуємо гарантовану логарифмічну залежність часу роботи цих алгоритмів від числа ключів, що зберігаються в дереві.

Рандомізовані дерева пошуку забезпечують збалансованість тільки в імовірнісному сенсі: ймовірність отримання сильно незбалансованого дерева при великих n хоча і є пренебрежимо малої, але залишається не дорівнює нулю Механізм підтримка збалансованості АВЛ-дерева - зміна структури дерева при додаванні і видаленні елементів. повороти: лівий, правий, великий лівий, великий правий.



Тема 5. Хеш-таблиці.

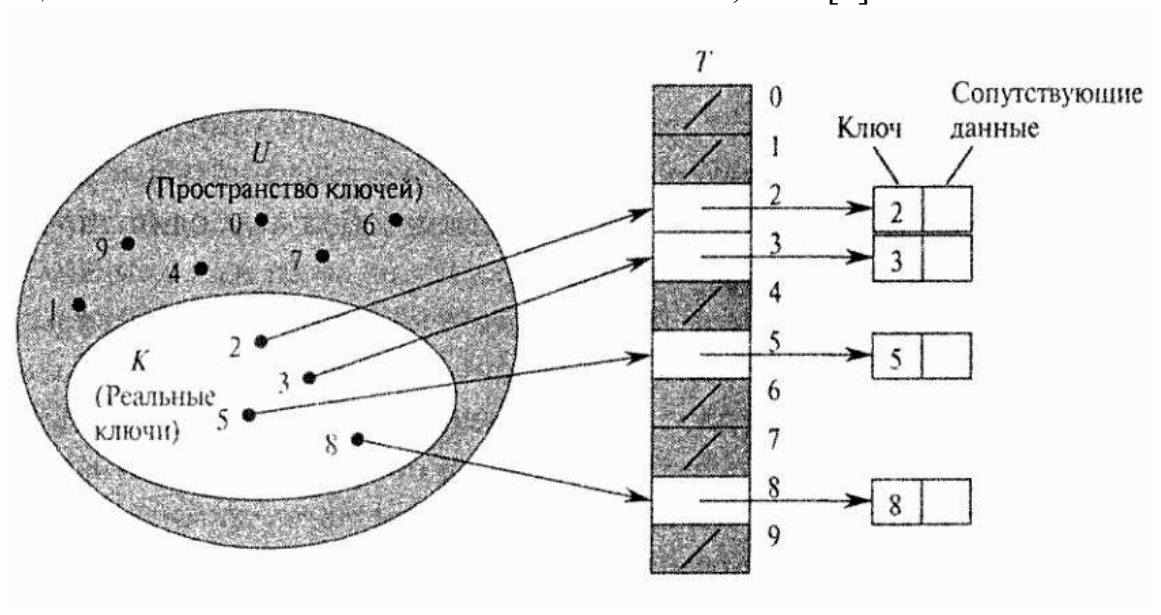
Для багатьох додатків необхідно і достатньо реалізовувати динамічні множини, що підтримують тільки стандартні операції: вставка, пошук, видалення, швидкість пошуку.

Лінійний пошук - Tprepare = O (n), Tfind = O (n)

Бінарний пошук і дерева пошуку - Tprepare = O (n • log (n)), Tfind = O (log (n))
А чи можна шукати за O (1)?

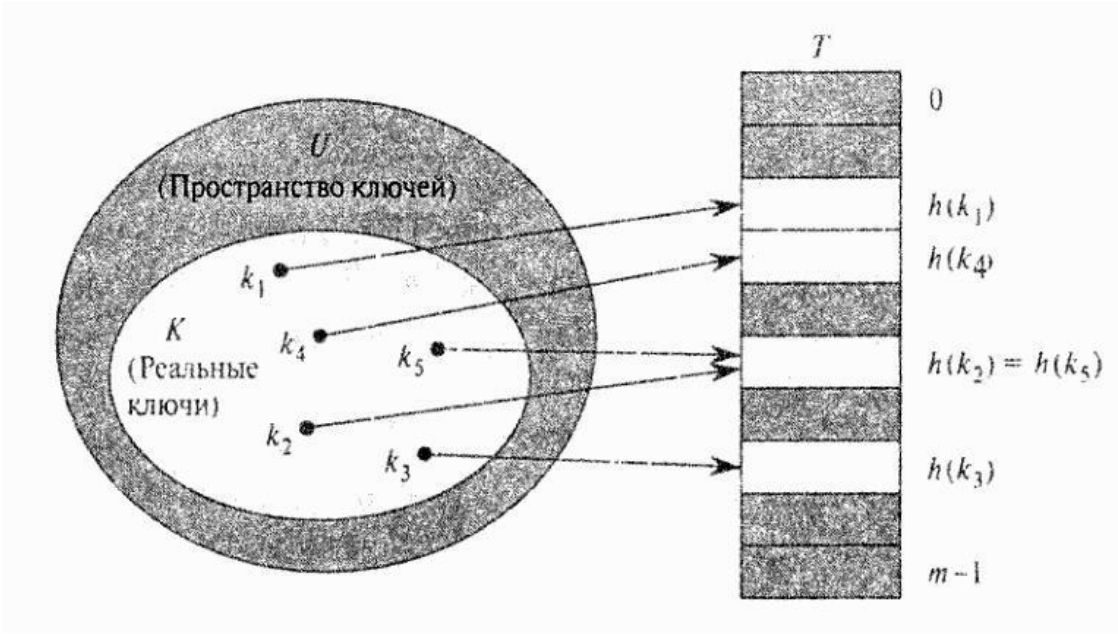
Пряма адресація застосовується для невеликих множин ключів. Потрібно поставити динамічне множина, кожен елемент якого має ключ з множинаі U = {0,1, ..., m - 1}, де m не надто велике, ніякі два елементи не мають однакових ключів.

Для подання динамічного множинаі використовується масив (таблицю з прямою адресацією), T [0..m - 1], кожна позиція, або осередок, якого відповідає ключу з простору ключів U. Осередок k вказує на елемент множинаі з ключем k. Якщо множина не містить елемента з ключем k, то T [k] = NULL.



Недоліки прямої адресації: якщо простір ключів U велике, зберігання таблиці T розміром | U | непрактично, а то і зовсім неможливо - в залежності від кількості доступної пам'яті і розміру простору ключів.

Множина K реально збережених ключів може бути малою в порівнянні з простором ключів U, а в цьому випадку пам'ять, виділена для таблиці T, в основному витрачається марно. Хеш-функція - така функція h, яка визначає розташування елементів множини U в таблиці T.

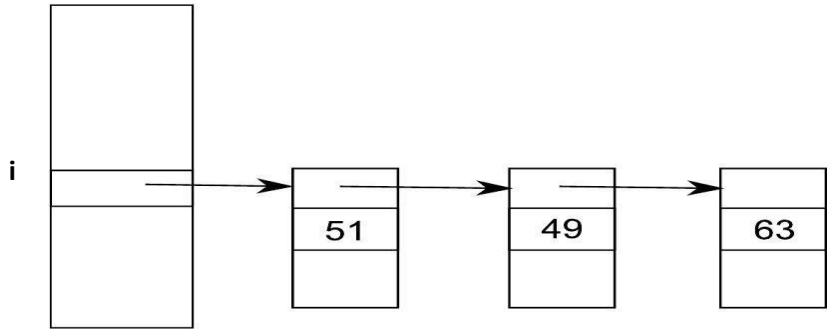


Колізія - ситуація, коли два ключа відображаються в одну і ту ж комірку. Наприклад, h (43) = h (89) = h (112) = k Вирішення колізій:

Метод ланцюжків

Відкрита адресація

Ідея: Зберігати елементи множини за однаковим значенням хеш-функції у вигляді списку.



h(51) = h(49) = h(63) = i

Вибір хеш-функції Ключі повинні рівномірно розподіляться по всіх осередків таблиці.

Закономірність розподілу ключів хеш-функції не повинна корелювати з закономірностями даних. (Наприклад, дані - це парні числа). методи:

Метод поділу • Метод множення метод поділу h (k) = k mod m.

Проблема маленького подільника m . Приклад №1. m = 2 і всі ключі парні непарні вічка не заповнені. Приклад №2. m = 2r хеш не залежить від бітів вище r.

Розв’язання колізій: відкрита адресація

Не потрібно зберігати посилання

Будемо послідовно перевіряти осередки таблиць, поки не знайдемо порожню. • h: U x {0,1, ..., m - 1} → {0,1, ..., m - 1} • (h (k, 0), h (k, 1), ..., h (k, m - 1)) - це перестановка (0,1, ..., m - 1) n <m стратегії дослідження • Лінійна - h (k, i) = (h '(k) + i) mod m де h '(k) - звичайна хеш-функція

Дана стратегія легко реалізується, проте схильна до проблеми первинної кластеризації, пов'язаної зі створенням довгої послідовності зайнятих осередків, що збільшує середній час пошуку.

Квадратична h (k, i) = (h '(k) + з1 i + c2 i2) mod m де h '(k) - звичайна хешфункція

Подвійне хешування - h (k, i) = (h1 (k) + i • h2 (k)) mod m.

Тема 6. Масиви, алгоритми сортування

Масив

Масив - структура даних, для зберігання даних одного типу, ідентифікованих за допомогою одного або декількох індексів.

Дані розташовуються в пам'яті безпосередньо один за одним. При цьому доступ до окремих елементів масиву здійснюється за допомогою індексації, тобто посилання на масив із зазначенням номера потрібного елемента. • Розмірність масиву - це кількість індексів, необхідне для однозначного доступу до елементу масиву.

Сортування

Вибором

Вставками

Метод бульбашки

Сортування - процес перестановки об'єктів заданої сукупності в певному порядку (зростаючий або спадному). Метою сортування зазвичай є полегшення подальшого пошуку елементів в відсортованому безлічі. Формальне визначення завдання сортування Дано: N об'єктів a1, a2, ..., an Необхідно: упорядкувати задані об'єкти, тобто переставити їх у такій послідовності AP1, ..., aPN, щоб їх ключі розташувалися в неубутному порядку kP1 ≤ kP2 ≤. . . ≤ kPN Алгоритм сортування вибором: i-й (i = 0) елемент береться як мінімальний, його індекс записується в min, j = i + 1 A [min] порівнюється з A [j], якщо цей елемент менше A [min], його індекс зберігається в min і переходимо до наступного елементу, j ++ A [min] і A [i] міняємо місцями i ++ і алгоритм повторюється public static void selection_sort(int[] a)

```
             {
            int n =
a.Length;
int min, k;
            for (int i = 0; i < n - 1; i++)
            {
min = i;
```

```
                for (int j = i +
1; j <= n - 1; j++)
if (a[j] < a[min])
min = j;                    k =
a[i];                   a[i] =
a[min];                   a[min] =
k;
            }

}
```

Сортування простими вставками

Масив складається з двох частин - впорядкованої і невпорядкованою. Алгоритм вбудовує елементи невпорядкованою частини в відсортовану частину масиву. На кожному кроці алгоритму вибираємо один елемент з невпорядкованою частини і вставляємо його на потрібну позицію в уже відсортовану частину масиву.

Повторюємо до тих пір, поки весь набір вхідних даних не буде відсортований. Елементи обробляються по порядку їх появи у вхідному масиві.

```
public static void insert_sort(int[] a, int n)
        {
int x;
            int i, j;
            for (i = 1; i < n; i++)
            {
                x = a[i];
                for (j = i - 1; (j >= 0)&&(x<a[j]); j--)
                    a[j
+ 1] = a[j];
a[j + 1] = x;
             }

}
```

Сортування простим обміном. метод бульбашки.

Алгоритм складається в повторюваних проходах по сортованого масиву. На кожній ітерації послідовно порівнюються сусідні елементи, і, якщо порядок в парі невірний, то елементи міняють місцями.

За кожен прохід по масиву як мінімум один елемент стає на своє місце, тому необхідно зробити не більше n-1 проходів, де n розмір масиву, щоб впорядкувати масив.

```
public static void bubble_sort(int[] a, int n)
        {
            int temp;
            for (int i = 0; i < n - 1; i++)
            {
                for (int j = n - 1; j > i; j--)
                {
                    if (a[j] < a[j - 1])
                    {
```

```
temp = a[j];
a[j] = a[j - 1];
a[j - 1] = temp;
}
                }
            }

}
```

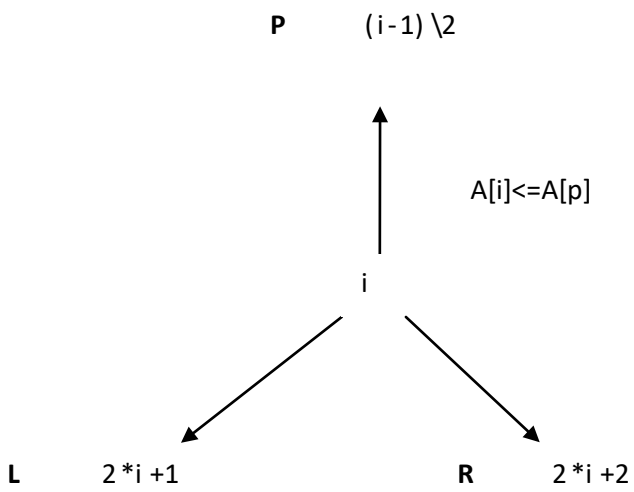
Двійковий пошук

```
public static int BinarySearch(int[] arr, int key, int last)
    {
      int c;
int first = 0;
while (last >= first)
         {
          c =
(first+last)/2;
if (arr[c] < key)
      first = c + 1;
     else if (arr[c] >
key)
last = c - 1;
else
               return c;
         }
     return -
1;
 }
```

Тема 7. Алгоритми швидкого сортування

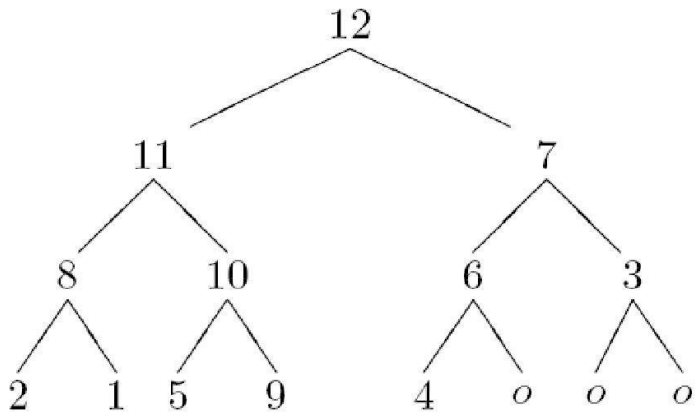
Пірамідальне сортування

Піраміда (binary heap) - це структура даних, що представляє собою об'єкт-масив, який можна розглядати як майже повне бінарне дерево. Кожен вузол цього дерева відповідає певному елементу масиву і для нього виконується наступна властивість:



-

Піраміда для 12 ти елементів



Алгоритм пірамідального сортування

Вхідні послідовність перетворюємо в піраміду.

Голову піраміди міняємо місцями з останнім елементом і зменшуємо розмір піраміди на 1. Потім відновлюємо піраміду. І повторюємо до тих пір поки в піраміді не залишиться один елемент

```
public static void Heap(int[] A, int i, int size)
{
int
l,r,largest,k;
l=2*i+1;
r=2*i+2;
largest=(l<size
&&A[l]>A[i])? l
: i; if
(r<size&&A[r]>A
[largest])
largest=r; if
(largest!=i) {
k=A[i];
```

```
        A[i]=A[largest];
        A[largest]=k;
        Heap(A, largest, size);
        }
}
public static void heap_sort (int[] A , int N)
{   int I, t;
 for (i = N/2-1; i >= 0; i--) // построение пирамиды
        Heap (A, i, N);
 for(i = N-1; i > 0; i--) { // сортировка
          t = A[0] ;
          A[0] = A[i]
;
         A[i] = t;
        Heap(A, 0, i);
     }
}
```

Аналіз пірамідального сортування. Час роботи Heap визначається відношенням T (N) ≤T (2 * N / 3) + θ (1) На підставі першого випадку теореми отримуємо оцінку T (N) = O (log (N)), (Висота піраміди з n елементів дорівнює O (log (N)) Час побудови піраміди: O (N • log (N)) Час сортування: O (N • log (N)) Загальний час: T (N) = O (N • log (N)).

Швидке сортування.

Алгоритм розроблений Чарльзом Хоаром в 1960 р. 1. Поділяємо масив на два подмассіва [1 ... m] і [m +1. ..N]

2. Рекурсивно сортуємо отримані два подмассіва.

Швидке сортування використовує стратегію Розділяй і володарюй.

Вибираємо в масиві певний елемент, який будемо називати опорним елементом. Відомі стратегії: вибирати постійно один і той же елемент, наприклад, перший, середній або останній по положенню; вибирати елемент зі випадково обраним індексом.

Операція поділу масиву: реорганізуємо масив таким чином, щоб всі елементи, менші або рівні опорного елементу, виявилися зліва від нього, а всі елементи, великі або рівні опорного - праворуч від нього.

Два індексу - l і r, прирівнюються до мінімального і максимального індексу масиву.

Обчислюється індекс опорного елемента. Встановлюються i = l; j = r.

Індекс і послідовно збільшується до тих пір, поки і-й елемент не перевищить опорний.

Індекс j послідовно зменшується до тих пір, поки j-й елемент не виявиться менше опорного.

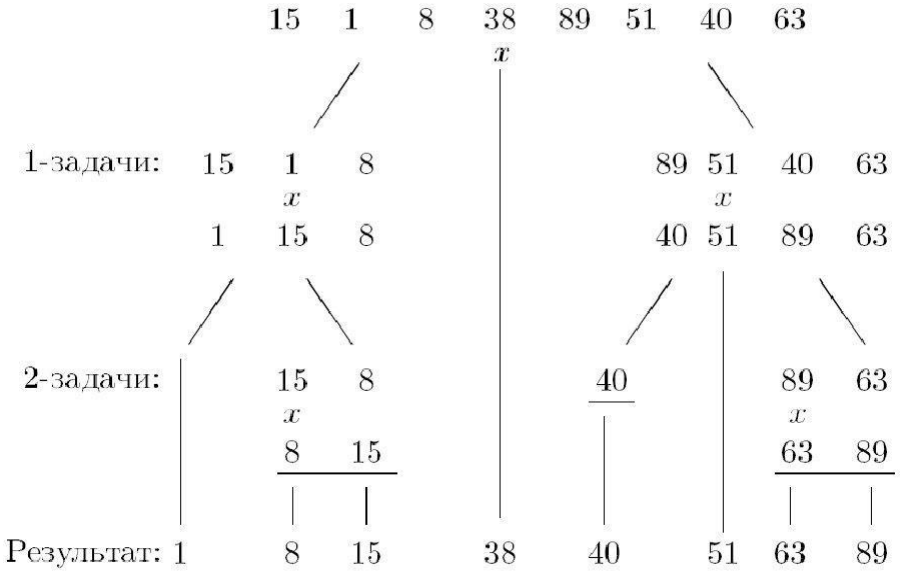
Якщо i <j - знайдену пару елементів потрібно обміняти місцями і продовжити операцію поділу з i ++, j-.

Якщо i = j - знайдена середина масиву - операція поділу закінчена, обидва індекси вказують на опорний елемент.

Рекурсивно упорядковуємо ліву і праву частину масиву.

Базою рекурсії є набори, що складаються з одного або двох елементів. Перший повертається в початковому вигляді, в другому, при необхідності, сортування зводиться до перестановки двох елементів. Всі такі відрізки вже впорядковані в процесі поділу.

Робота швидкого сортування.




```
static void quickSort(int[] a, int
l, int r)     {              int
temp;              int x = a[l + (r
- l) / 2];
            int
i = l;
int j = r;
while (i <= j)
            {
                while (a[i] <
x) i++;                   while
(a[j] > x) j--;
                if (i <= j)
                {

temp = a[i];
a[i] = a[j];
a[j] = temp;

i++;
j--;
                }
            }
            if (i < r)
quickSort(a, i, r);
if (l < j)  quickSort(a, l,
j);
        }
```

Тема 8. Динамічне програмування

Динамічне програмування - метод рішення класу задач, де загальне рішення задачі засноване на рішенні подзадач меншого розміру. Підзадачі, на які розбивається завдання, є пересічними. Динамічне програмування, як правило, застосовується до завдань оптимізації (optimization problems). У таких завданнях можлива наявність багатьох рішень. Кожному варіанту рішення можна зіставити якесь значення, і нам потрібно знайти серед них рішення з оптимальним (мінімальним або максимальним) значенням. Назвемо таке рішення одним з можливих оптимальних рішень. В силу того, що таких рішень з оптимальним значенням може бути кілька, слід відрізняти їх від єдиного оптимального рішення.

Ключова ідея в динамічному програмуванні досить проста. Щоб вирішити задачу, потрібно вирішити окремі частини завдання (підзадачі), після чого об'єднати рішення підзадач в одне загальне рішення. Часто багато хто з цих підзадач однакові. Підхід динамічного програмування полягає в тому, щоб вирішити кожну підзадачу тільки один раз, скоротивши тим самим кількість обчислень. Це особливо корисно у випадках, коли число повторюваних подзадач експоненціально велике. динамічне програмування користується такими властивостями завдання:

перекриваються підзадачі;

оптимальне рішення задачі засноване на оптимальному рішенні подзадач; • можливість запам'ятовування рішення отриманих оптимальних рішень

Динамічне програмування зазвичай дотримується двох підходів до вирішення завдань: спадний динамічне програмування: задача розбивається на підзадачі меншого розміру, вони вирішуються і потім комбінуються для вирішення вихідної задачі. Використовується запам'ятовування для рішень часто зустрічаються подзадач. висхідний динамічний програмування: всі підзадачі, які згодом знадобляться для вирішення вихідної задачі прораховуються заздалегідь і потім використовуються для побудови рішення вихідної задачі.

Основні властивості задач динамічного програмування:

Завдання має властивість оптимальності

Невелике число різних підзадач

Процес розробки алгоритмів динамічного програмування можна розбити на чотири етапи.

Опис структури оптимального рішення.

Рекурсивне визначення значення, що відповідає оптимальному вирішенню.

Обчислення значення, відповідного оптимального рішення, за допомогою методу висхідного аналізу.

Складання оптимального рішення на основі інформації, отриманої на попередніх етапах.

Послідовність Фібоначчі Fn задається формулами:

F0 = 0, F1 = 1, Fn = Fn – 1 + Fn – 2 при n > 1.

Необхідно найти Fn по номеру n. Рекурсивний метод рішення

```
int F(int n) {
     if (n == 0)
return 0;      if (n ==
1) return 1;
 else return F(n - 1) + F(n - 2);
```

```
}

Спадне динамічне програмування
int F(int n){
     if (A[n] != -1) return
A[n];     if (n == 0) return
0;
  if (n == 1) return 1;
    else
     {
          A[n] = F(n - 1) + F(n -
2);           return A[n];
     }
}
```

Висхідне динамічне програмування
F[0] =0; F[1] = 1;
for (i = 2; i < n; i++) F[i] = F[i - 1] + F[i - 2];

Завдання знаходження найбільшої спільної підпослідовності - завдання пошуку послідовності, яка є підпослідовність декількох послідовностей (зазвичай двох). Часто завдання визначається як пошук всіх найбільших підпослідовностей. Підпослідовність можна отримати з деякої кінцевої послідовності, якщо видалити з останньої деякий множина її елементів (можливо порожній). Наприклад, BCDB є підпослідовність послідовності ABCBDAB. Будемо говорити, що послідовність Z є спільною підпослідовність послідовностей X і Y, якщо Z є підпослідовність як X, так і Y. Потрібно для двох послідовностей X і Y знайти загальну підпослідовність найбільшої довжини. Зауважимо, що НОП може бути кілька.

Будова НОП.

Нехай є послідовності X = (x1, x2, ..., xm) і У = (y1, y2, ..., yn), а Z = (z1, z2, ..., zk) -одна з найбільших спільних підпослідовностей. тоді:

Якщо xm = yn, то zk = xm = yn і Zk-1 - НОП послідовностей Xm-1, У n-1

Якщо xm ≠ yn і zk ≠ xm, то Z - НОП для Xm-1 і У

Якщо xm ≠ yn і zk ≠ yn, то Z - НОП для Xm і Yn-1 рекурентна формула

c[i, j] - довжина НОП для послідовностей Xi і Yj, якщо i або j дорівнює 0, то одна з послідовностей порожня і з [i, j] = 0

$$\chi[\iota,\ \varphi] = \begin{cases} 0, & \iota = 0 \vee \varphi = 0 \\ \mu\alpha\xi(\chi\chi[[\iota\iota, -\varphi 1-,\ \varphi 1]-,1\chi][\iota+-11,,\xi_{\iota}\varphi])=,\psi\xi_{\iota\varphi},\neq\iota,\psi\varphi\ _{\varphi}>,\iota 0, & \varphi > 0 \end{cases}$$

Заповнення таблиці

```
LCS_LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                      then c[i, j] ← c[i − 1, j]
14                           b[i, j] ← "↑"
15                      else c[i, j] ← c[i, j − 1]
16                           b[i, j] ← "←"
17  return c и b
```

Тема 9. «Жадібні» алгоритми

Методи програмування.

Жадібні алгоритми - проводиться локально оптимальний вибір в надії, що він приведе до оптимального рішення всієї задачі.

Динамічне програмування - розбити задачу на підзадачі і не обчислювати підзадачі більше одного разу (використовувати пам'ять).

Жадібний алгоритм - алгоритм, полягає в прийнятті оптимальних рішень на кожному кроці, де загальне рішення задачі складається з оптимальних рішень на попередніх кроках.

Для завдань, що вирішуються жадібними алгоритмами, характерні дві особливості:

до них застосуємо Принцип жадібного вибору,

вони мають властивість Оптимальності для подзадач.

Принцип жадібного вибору.

Кажуть, що до оптимізаційної задачі застосуємо принцип жадібного вибору, якщо послідовність локально оптимальних виборів дає глобально оптимальне рішення.

У типовому випадку доказ оптимальності слід такою схемою:

Доводиться, що жадібний вибір на першому кроці не закриває шляхи до оптимального рішення: для будь-якого рішення є інше, узгоджене з жадібним вибором і не гірше першого.

Показується, що подзадача, що виникає після жодного вибору на першому кроці, аналогічна вихідної.

Міркування завершується по індукції.

Оптимальність для підзадач.

Кажуть, що завдання має властивість оптимальності для подзадач, якщо оптимальне рішення задачі містить в собі оптимальні рішення для всіх її підзадач.

Завдання про вибір процесів.

Дано: множина процесів S = {a1, a2, ..., an}, процесам потрібно деякий ресурс, який одночасно може використовуватися лише одним процесом.

Кожен процес ai характеризується початковим моментом si і кінцевим моментом fi, де 0 ≤ si <fi <∞. Будучи обраний, процес ai триває протягом [si, fi) (кінець одного процесу може збігатися з початком іншого процесу, це не вважається перетином), процеси ai і aj сумісні, якщо інтервали [si, fi) і [sj, fj) не перекриваються.

Знайти: підмножину взаємно сумісних процесів, що утворюють множину максимального розміру. Завдання про вибір процесів. Оптимальне вирішення. жадібний алгоритм .

Нехай S відсортовано за f в порядку зростання за час O (n log n).

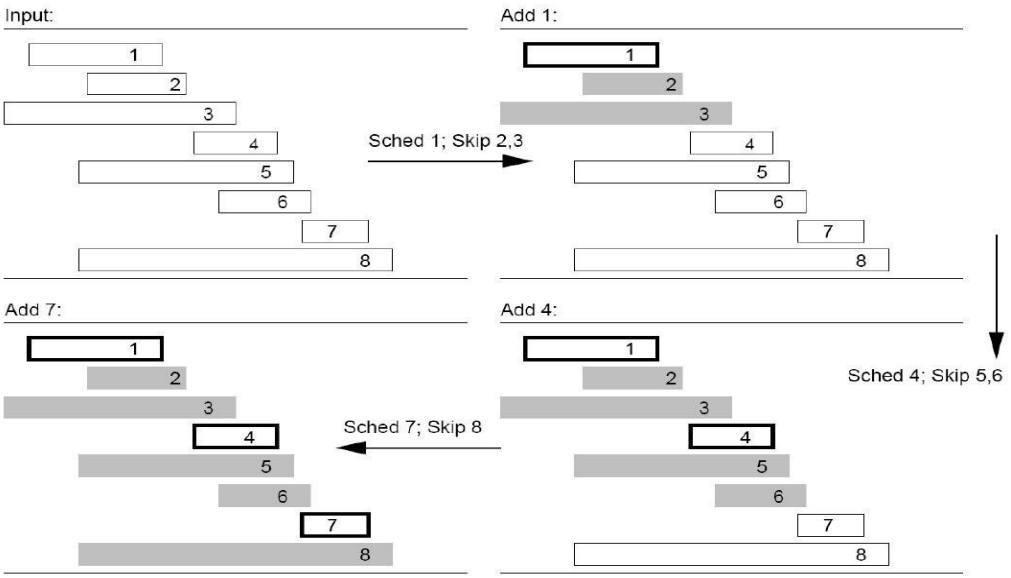
У результуючу множину процесів заносимо 1 процес (оскільки він закінчується раніше всіх інших)

З множини процесів прибираємо ті, які конфліктують з обраним процесом

Повторюємо алгоритм для меншої кількості процесів Складність алгоритму O

(n log n + n).

У доказі досліджується глобальне оптимальне рішення деякої підзадачі. Потім демонструється, що рішення можна перетворити так, щоб в ньому використовувався жадібний вибір, в результаті чого вийде аналогічна, але більш проста подзадача. Всі процеси відсортовані по неубиванія часу закінчення. Процес номер 1, очевидно, входить в оптимум (якщо немає, то замінимо найраніший процес в оптимумі на нього, що не зашкодить сумісності процесів). Прибравши всі процеси, що суперечать першому, отримаємо вихідну задачу з меншою кількістю процесів. Міркуючи по індукції, отримуємо, що роблячи на кожному кроці жадібний вибір, приходимо до оптимального рішення.



Тема 10. Графи: подання, обходи, топологічне сортування

Орієнтований граф G = (V, E) складається з кінцевого множинаі вершин V і множинаі упорядкованих пар вершин E, званих ребрами. Неорієнтовані граф G = (V, E) складається з кінцевого множинаі вершин V і множинаі невпорядкованих пар вершин E, званих ребрами. Вершина v є суміжною вершині u, якщо ∃ ребро (u, v). Ступінь вершини сума ребер виходять з даної вершини. Ступінь графа максимальний ступінь його вершин. Шлях в графі - послідовність (v0, v1, ..., vk), де (vi-1, vi) є ребром для i = 1,2, ..., k. Довжина шляху - кількість ребер, дорівнює k. Цикл - це шлях, довжина якого більше нуля і v0 = vk. Ациклічності граф - граф без циклів.

Вільне дерево - зв'язний ациклічності граф. Ліс дерев - незв'язних ациклічності граф.

DAG (directed acyclic graph) - орієнтований ациклічний граф

Подання графа.

Матриця суміжності.

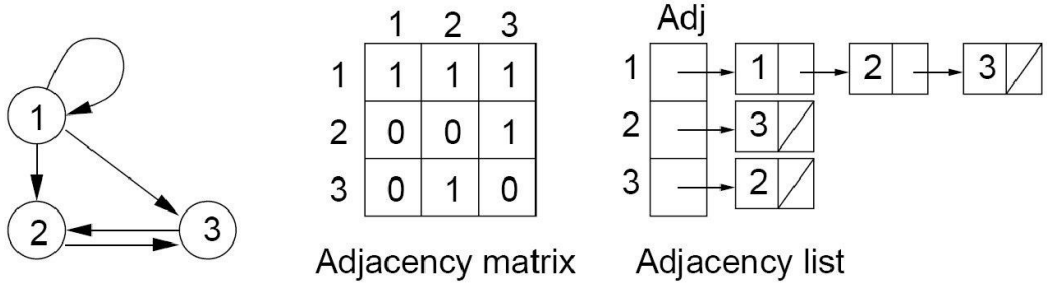
Список суміжних вершин

Нехай G = (V, E) - орграф з n = | V | вершин і e = | E | ребер. Нехай вершини пронумеровані {1,2, ..., n} Тоді n x n матриця є матрицею суміжності. Інший спосіб представлення - список суміжних вершин Якщо граф зважений, то якщо (v, w) □ E ⇒ A [v, w] = W (v, w).

Подання графа G у вигляді списку суміжності використовує масив Adj з | V | списків, по одному для кожної вершини з V. Для кожної вершини u з V список Adj [u] містить всі вершини v, такі що (u, v) належить E, тобто Adj [u] складається з усіх вершин, суміжних з u в графі G. Вершини в кожному списку зазвичай зберігаються в довільному порядку.

Якщо G - орієнтований граф, то сума довжин всіх списків суміжності дорівнює | E |, оскільки ребру (u, v) однозначно відповідає елемент v в списку Adj [u]. Якщо G - неорієнтований граф, то сума довжин всіх списків суміжності дорівнює 2 | E |, оскільки ребро (u, v), будучи неорієнтованим, з'являється в списку Adj [v] як u, і в списку Adj [u] - як v. Як для орієнтованих, так і для неорієнтованих графів подання до вигляді списків вимагає обсяг пам'яті, рівний θ (V + E).

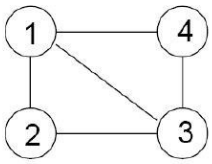
Матриця і список суміжності для орграфа



Кількість осередків у матриці: Θ (V2) - вигідно для щільних графів.

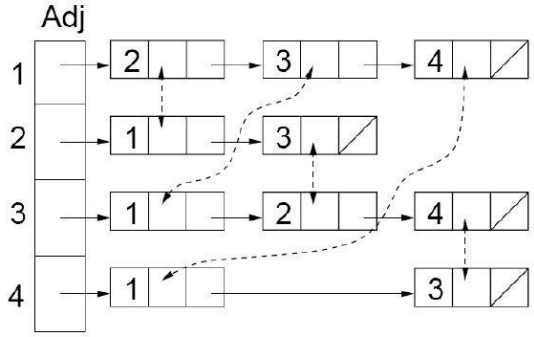
Кількість осередків у списку: Θ (V + E) - вигідно для розріджених графів.

Матриця і список суміжності для неорієнтованого графа



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Adjacency matrix



Adjacency list (with crosslinks)

Обхід в ширину

Нехай заданий граф G = (V, E) і виділена вихідна (source) вершина s. Алгоритм пошуку в ширину систематично обходить всі ребра G для "відкриття" всіх вершин, досяжних з s, обчислюючи при цьому відстань (мінімальна кількість ребер) до кожної досяжною з s вершини. Крім того, в процесі обходу будується "дерево пошуку в ширину" з коренем s, що містить всі досяжні вершини. Для кожної досяжною з s вершини v шлях в дереві пошуку в ширину відповідає найкоротшому (тобто містить найменшу кількість ребер) шляху від s до v в G. Алгоритм працює як для орієнтованих, так і для неорієнтованих графів.

Пошук в ширину будує дерево пошуку в ширину, яке спочатку складається з одного кореня, яким є вихідна вершина s. Якщо в процесі сканування списку суміжності вже відкритої вершини u відкривається біла вершина v, то вершина v і ребро (u, v) додаються в дерево. Ми говоримо, що u є попередником, або батьком, v в дереві пошуку в ширину. Оскільки вершина може бути відкрита не більше одного разу, вона має не більше одного з батьків. Взаємовідносини предків і нащадків визначаються в дереві пошуку в ширину як зазвичай - якщо u знаходиться на шляху від кореня s до вершини v, то u є предком v, а v - нащадком u. Наведена нижче процедура пошуку в ширину BFS передбачає, що вхідний граф G = (V, E) представлений за допомогою списків суміжності. Крім того, підтримуються додаткові структури даних в кожній вершині графа. Колір кожної вершини u зберігається в змінної color [u], а попередник - у змінній pred [u]. Якщо попередника у u немає, то pred [u] = null. Відстань від s до вершини u, що обчислюється алгоритмом, зберігається в поле d [u]. Алгоритм використовує чергу Q для роботи з великою кількістю сірих вершин.

Обхід графа в ширину.

```
BFS(G,s) {
     for each u in V  {                  //
initialization           color [u]   = white
     d [u] = infinity         pred [u]     =
null
     }
     color [s]   = gray // initialize
source s       d[s]   = 0

    Q =  {s}                             // put s in the queue
  while   (Q is nonempty) {
```

```
        u = Q.Dequeue()                 // u is the next to visit
      for each v in Adj[u]
                              {
           if   (color[v]   == white)              //if neighbor v
{                                       undiscovered
                color[v]   = gray       //   ...mark it discovered
                 d[v] = d[u]+1          //   ...set its distance
                pred[v]    = u         //   ...and its predecessor
                  Q.Enqueue(v)         //   ...put it in the queue
              }
         }
      color [u]   = black       // we are done with u
    }
}
```

Обхід графа в глибину

Стратегія пошуку в глибину, як випливає з її назви, полягає в тому, щоб йти "вглиб" графа, наскільки це можливо. При виконанні пошуку в глибину досліджуються всі ребра, що виходять з вершини, відкритої останньої, і покидается вершина, тільки коли не залишається недосліджених ребер - при цьому відбувається повернення в вершину, з якої була відкрита вершина v. Цей процес триває до тих пір, поки не будуть відкриті всі вершини, досяжні з вихідної. Якщо при цьому залишаються невідкриті вершини, то одна з них вибирається в якості нової вихідної вершини і пошук повторюється вже з неї. Цей процес повторюється до тих пір, поки не будуть відкриті всі вершини. Як і в разі пошуку в ширину, коли вершина v відкривається в процесі сканування списку суміжності вже відкритої вершини u, процедура пошуку записує цю подію, встановлюючи поле попередника v рівним u. На відміну від пошуку в ширину, де підграф передування утворює дерево, при пошуку в глибину підграф передування може складатися з декількох дерев, так як пошук може виконуватися з декількох вихідних вершин.

Подграф передування (predecessor subgraph) пошуку в глибину, складається з ребер, що з'єднують предка вершини v і саму вершину v E pred = {(pred [v], v): v Є V і pred [v] ≠ NULL}. Подграф передування пошуку в глибин утворює ліс пошуку в глибину (depth-first forest), який складається з декількох дерев пошуку в глибину. Кожна вершина має дві мітки часу-першу d [u], в якій вказується, коли вершина u відкривається (і забарвлюється в сірий колір), і друга - f [u], яка фіксує момент, коли пошук завершує сканування списку суміжності вершини u і вона стає чорною. Ці мітки використовуються багатьма алгоритмами і корисні при розгляді поведінки пошуку в глибину. Для кожної вершини u d [u] <f [u]. До моменту часу d [u] вершина має колір white, між d [u] і f [u] - колір gray, а після f [u] - колір black.

Обхід графа в глибину.

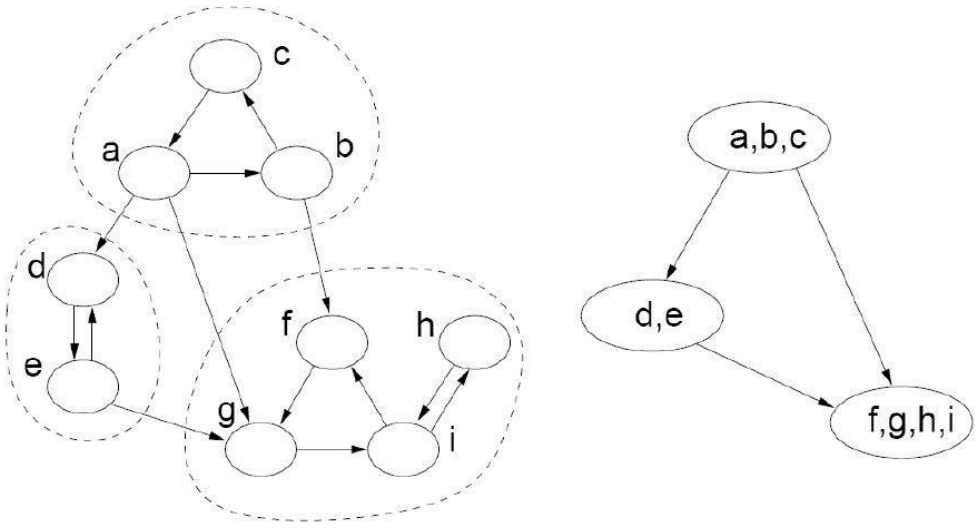
Алгоритм

```
DFS(G)   {                      // main program
   for each u in V {                          //
                                initialization
       color[u]   = white;
       pred[u]    = null;
    }
```

```
     time = 0;      for each u in V          if   (color[u]
== white)      // found an undiscovered vertex
              DFSVisit(u);      // start a new search here
}
DFSVisit(u)   {                          // start a search
at u      color[u]   = gray;                 // mark u
visited
 d[u]   = ++time;  for each v
in Adj(u) do   if   (color[v]
== white)   {
             pred[v]   = u;    //  ...set predecessor pointer
              DFSVisit(v);      //   ...visit v
          }
     color[u]   = black;            // we're done with u
     f[u]  = ++time;
}
```

Тема 11. Графи: сильно зв'язані компоненти, остовні дерева

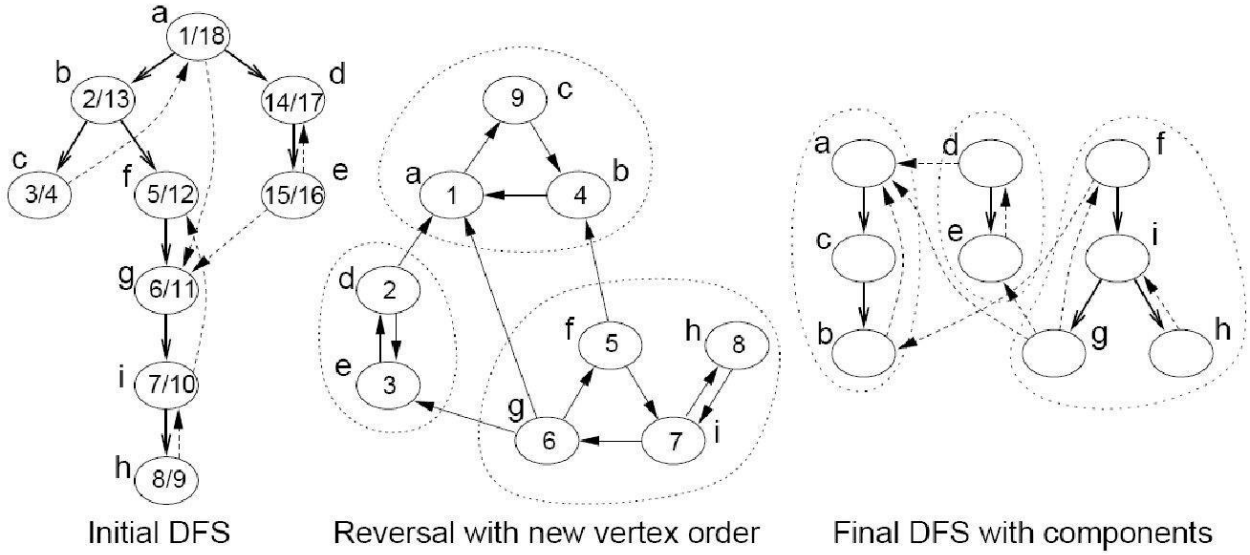
Проблема зв'язності У комунікаційних і транспортних мережах важливо знати, що будь-який вузол мережі доступний з будь-якого іншого вузла. Орграф називається сильно зв'язковим, якщо для кожної пари вершин u, v ∈ V існує шлях з v в і й назад. Сильно пов'язаної компонентою орієнтованого графа G називається максимальне множина вершин U з такою властивістю: будь-які дві вершини u і v з U досяжні одна з одної.



Digraph and Strong Components Component DAG

Пошук сильно зв'язкових компонентів
FindStrongComp(G)
DFS (G) ► вычислим f[u] для каждой вершины и
R ← Reverse(G) ► инвертируем все ребра в G
SortByF(R) ► сортируем вершины в R по f[u] по убыванию
DFS(R)

Результат: кожне DFS-дерево в R є сильно пов'язаним компонентом. Кожне ребро в GT, що з'єднує два сильно зв'язкових компонента, йде від компонента з більш раннім часом завершення (при пошуку в глибину) до компоненту з більш пізнім часом завершення.



Initial DFS Reversal with new vertex order Final DFS with components

Мінімальний кістяк

Дан зважений неорієнтовані граф G з n вершинами і m ребрами. Потрібно знайти таке поддерево цього графа, яке б об'єднувало всі його вершини, і при цьому мало найменшим можливим вагою (тобто сумою ваг ребер).

Піддерево - це набір ребер, що з'єднують всі вершини, причому з будь-якої вершини можна дістатися до будь-якої іншої рівно одним простим шляхом.

Таке поддерево називається мінімальним остовне деревом або просто мінімальним кістяком. Будь-остов буде містити n-1 ребро. кістяк - ациклічності підграф G1 = (V1, E1) графа G = (V, E), де E1 □ E, V1 = V.

Мінімальна кістяк (МОД) - кістяк з мінімальною вагою. У природному постановці ця задача звучить наступним чином: є n міст, і для кожної пари відома вартість з'єднання їх дорогою (або відомо, що з'єднати їх не можна). Потрібно з'єднати всі міста так, щоб можна було доїхати з будь-якого міста в інший, а при цьому вартість була б мінімальною. Проблема побудови мережі Області виникнення: комунікаційні та дорожні мережі. Дано: множина вузлів мережі (хости, міста). Необхідно: побудова мережі з найменшим загальною вагою ребер (Довжина мережевих кабелів, довжина доріг). Графова модель: вузли мережі є вузлами графа, E = V2, нам відомі ваги всіх ребер. Результат:

вільний дерево.

Підхід до пошуку МОД.

Будуємо дерево A шляхом додавання по одному ребру, і перед кожною итерацией поточний дерево є підмножиною деякого МОД. На кожному кроці алгоритму ми визначаємо ребро (u, v), яке можна додати до A без порушення цієї властивості. Ми назвемо таке ребро безпечним GenericMST(G, w)

A ← 0

while A не является МОД

do Найти безопасное для A ребро (u, v)

A ← A U {(u, v)}
return A

Алгоритм Круськала.

Алгоритм Круськала спочатку поміщає кожну вершину в своє дерево, а потім поступово об'єднує ці дерева, об'єднуючи на кожній ітерації два деяких дерева ребром мінімальної ваги, яка не призведе до утворення циклу.

Перед початком виконання алгоритму, всі ребра упорядковано відповідно до вазі (в порядку неспадання). Потім починається процес об'єднання: перебираються всі ребра від першого до останнього (в порядку сортування), і якщо у поточного ребра його кінці належать різним піддерев, то ці піддерева об'єднуються, а ребро додається до відповіді.

Після закінчення перебору всіх ребер всі вершини опиняться належать одному піддерево, відповідь знайдений.

Алгоритм Прима.

Шуканий мінімальний остов будується поступово, додаванням в нього ребер по одному. Спочатку остов покладається складається з єдиною вершини (її можна вибрати довільно). Потім вибирається ребро мінімальної ваги, що виходить з цієї вершини, і додається в мінімальний остов. Після цього остов містить вже дві вершини, і тепер шукається і додається ребро мінімальної ваги, має один кінець в одній з двох обраних вершин, а інший - навпаки, у всіх інших, крім цих двох. І так далі, тобто всякий раз шукається мінімальне по вазі ребро, один кінець якого - вже взята в остов вершина, а інший кінець - ще не взята, і це ребро додається в остов (якщо таких ребер кілька, можна взяти будь-який). Цей процес повторюється до тих пір, поки остов не стане утримувати все вершини (або, що те ж саме, n-1 ребро). В результаті буде побудований кістяк, який є мінімальним. Якщо граф був з самого початку не зв'язний, то остов знайдений не буде (кількість обраних ребер залишиться менше n-1).

Тема 12. Графи: найкоротші шляхи

Завдання пошуку найкоротшого шляху
з заданої вершини в усі інші
між усіма парами вершин

Властивість найкоротшого шляху

Нехай p = (v1, v2 ..... vk) - найкоротший шлях з вершини v1 до вершини vk в заданому зваженому орієнтованому графі G = (V, E) з ваговою функцією w: E → R а pij = (vi, vi +1 ..... vj) - частковий шлях шляху p, який проходить з вершини vi до вершини vj для довільних i і j ($1 \le i < j \le k$). Тоді pij - найкоротший шлях з вершини vi до vj

Дерево найкоротших шляхів

Дерево найкоротших шляхів з коренем у вершині S - це орієнтований підграф G '= (V', E '), в якому множинаі V' V і E 'E визначаються такими умовами:

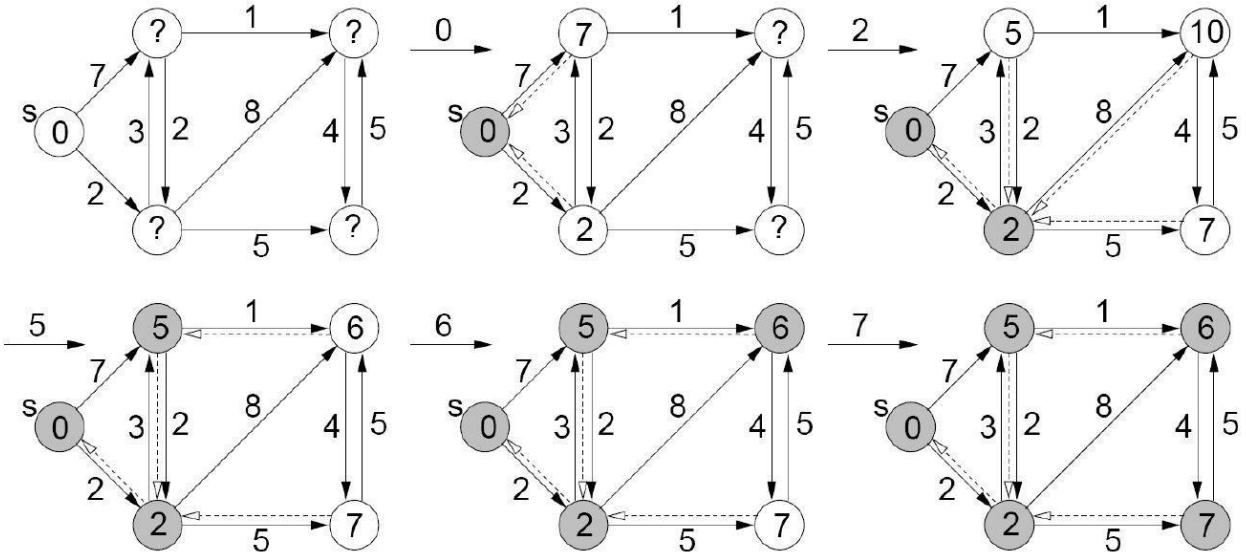
V - множина вершин, досяжних з витоку s графа G:

граф G 'утворює кореневе дерево з коренем у вершині s

для всіх v V 'однозначно певний простий шлях з вершини s у вершину v у графі G' збігається з найкоротшим шляхом з вершини s у вершину v у графі G.

Пошук шляхів з заданої вершини: Алгоритм Дейкстры (Dijkstra)

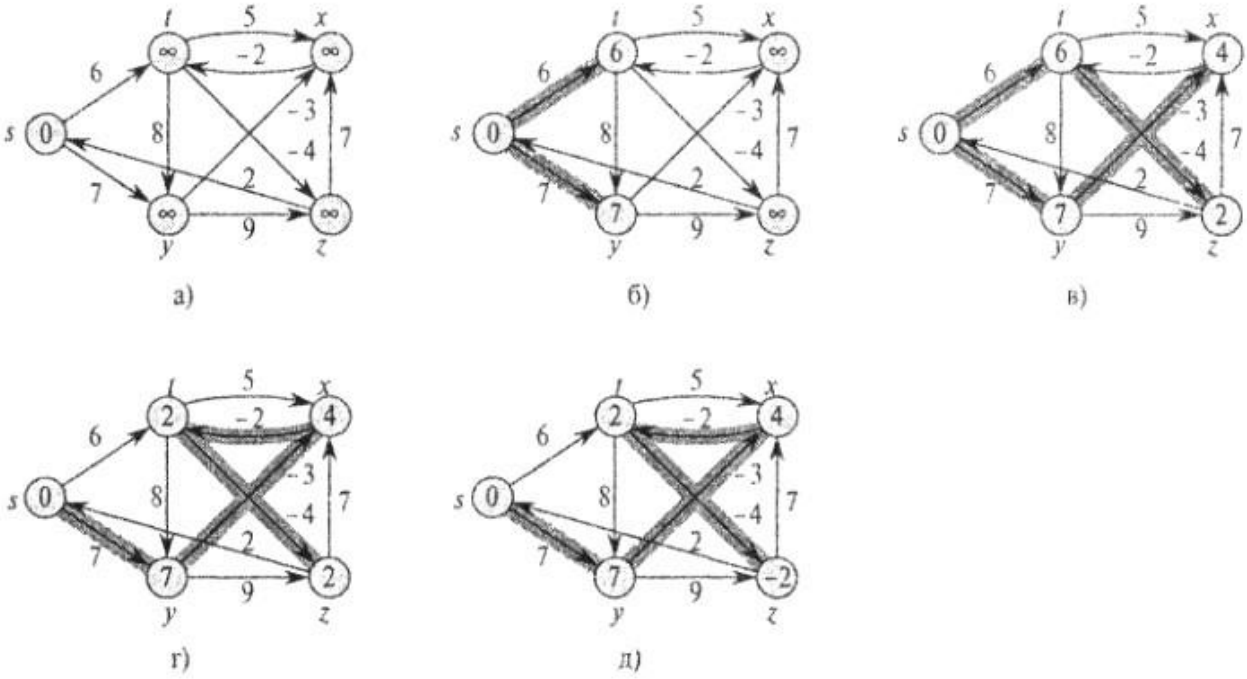
```
Dijkstra(G, w, s) {
     for each  (u in V)            // initialization
{              d [u] =
+infinity           color
[u] = white
        pred[u]   = null
     }
    d[s]   =0                     // dist to source is 0
    Q = new PriQueue(V)            // put all vertices in Q
  while   (Q.nonEmpty())   {             // until all vertices
                                  processed
         u = Q.extractMin()}        // select u closest to s
       for each  (v in Adj[u]) {
   if   (d[u]   + w(u,v)   < d[v])   {  //
Relax(u,v)     d [v]   = d [u]   + w(u,v)
Q.decreaseKey(v,  d[v])
                  pred [v]   = u
               }
          }
       color [u]   = black
     }
}
```



Алгоритм Беллмана-Форда .

Алгоритм Беллмана-Форда (Bellman-Ford algorithm) дозволяє вирішити задачу про найкоротший шлях з однієї вершини в загальному випадку, коли вага кожного з ребер може бути негативним. Для заданого зваженого орієнтованого графа G = (V, E) з витоком s і ваговою функцією w: E -> R алгоритм БеллманаФорда повертає логічне значення, яке вказує на те, чи міститься в графі цикл з негативним вагою, досяжний з витоку. Якщо такий цикл існує, в алгоритмі вказується, що рішення не існує. Якщо ж таких циклів немає, алгоритм видає найкоротші шляхи і їх вага.

```
Bellman_Ford (G, w, s)
// w - весовая функция графа
//  s  - исходная вершина
Initialize_Single_Source(
G, s)   for i =1 to
|V[G]| - 1       do for
each (u, v)    E[G]
do Relax (u, v, w)   for
each (u, v)    E[G]
do if d[v]>d[u] + w(u, v)
then return FALSE  return
TRUE.
```



Тема 13. Рекурсивні алгоритми

Рекурсія - фундаментальне математичне поняття. Вона передбачає покрокову організацію обчислювального процесу, де обчислення проводяться по одному і тому ж алгоритму, але кожен раз з меншим обсягом даних. Попередній етап не може завершитися, тому що його результат обчислюється через результат цього ж алгоритму, але з меншим об'ємом даних. Така організація обчислювального процесу передбачає отримання результату на останньому кроці. Потім проводиться повернення до недовичісленним кроків до отримання остаточного результату. Розглянемо приклад рекурсії для знаходження суми двох натуральних чисел z = x + y. Запишемо цю функцію в термальному вигляді z = f + (x, y). Значення функції для y: x + (y + 1) = (x + y) + 1 = z + 1, або f (x, y + 1) = f + (x, y) +1. Якщо вважати, що z + 1 - функція, яка обчислює значення наступного натурального числа через попереднє, то маємо рекурсивне визначення для функції складання. Якщо y = 0, то f + (x, 0) = x. Таким чином маємо рекурсивне визначення для функції додавання:

$$\begin{cases} f+(x,0)=x. \text{ – базис рекурсии.} \\ f+(x,y+1)=f+(x,y)+1 \end{cases}$$

Базис рекурсії - це той останній крок рекурсії, де виходить результат кроку. Продемонструємо функціонування рекурсивної схеми на прикладі обчислення f + (4, 3).

Відповідно до другим рядком рекурсивного визначення маємо: f + (4,3) = f + (4,2 + 1) = f + (4,2) +1 вихід для обчислення f + (4,2), f + (4,2) = f + (4,1 + 1) = f + (4,1) +1 вихід для обчислення f + (4,1), f + (4,1) = f + (4,0 + 1) = f + (4,0) + 1 = 4 + 1 = 5.

Тепер проводиться повернення до недовичісленним двом крокам: f + (4,1) + 1 = 5 + 1 = 6, f + (4,2) + 1 = 6 + 1 = 7. Отже f + (4,3) = 7.

У рекурсивних алгоритмах передбачається механізм виходу з попереднього кроку зі збереженням станів залишеного алгоритму і механізм повернення для здійснення "довичісленій". При складанні рекурсивного алгоритму ці міркування залишаються "за кадром" для становить алгоритм, але про них потрібно пам'ятати. Крім того, при використанні рекурсії потрібно завжди пам'ятати про базис рекурсії, який задає результат для останнього кроку рекурсивного обчислення.

Розглянемо приклад складання рекурсивного алгоритму для обчислення функції n !. Спочатку складемо алгоритм покроковим методом (без рекурсії): FACT_it(n).

```
F←1
for i←1 to n
do F←F*i
return F
```

Тут використовується формула для обчислення n! = 1 * 2 * 3 * ... * n. Оцінимо тимчасову складність T (n) цього алгоритму. Для рядків 1,4 час оцінюється як константа Q (1). Для рядків 2, 3 час оцінюється як Q (n). T (n = Q (n) + Q (1) = Q (n).

Складемо тепер рекурсивний алгоритм обчислення функції n !, враховуючи, що n! = (N-1)! * N. Базис рекурсії маємо при n = 0, т. Е. O! = 1. У рекурсивних алгоритмах перевірка базису є однією з перших команд (рядків).

```
   FACT_rek(n)                оценка
if n=0                        Q(1)
then return 1                 Q(1)
return FACT_rek(n-1)*n        T(n-1)
```

Методика оцінки рекурсивних алгоритмів аналогічна оцінці покрокових алгоритмів, але якщо рядок містить рекурсивне звернення, то вона оцінюється тією ж функцією. Так в алгоритмі FACT_rek, якщо FACT_rek (n) оцінюється як T (n), то FACT_rek (n-1) оцінюється як T (n-1). Загальна оцінка алгоритму FACT_rek, виходить як сума оцінок його рядків: T (n) = Q (1) + Q (1) + T (n-1) = T (n-1) +1. Рівність T (n) = T (n-1) +1 називається рекурсивним співвідношенням, де роль невідомого грає функція T (n). Вирішуючи рекурсивне співвідношення можна отримати вид функції T (n). Вирішити T (n) = T (n-1) +1 можна методом ітерацій:

T(n-1)= T(n-2)+1

T(n-2)= T(n-3)+1

…

T(2)= T(1)+1.

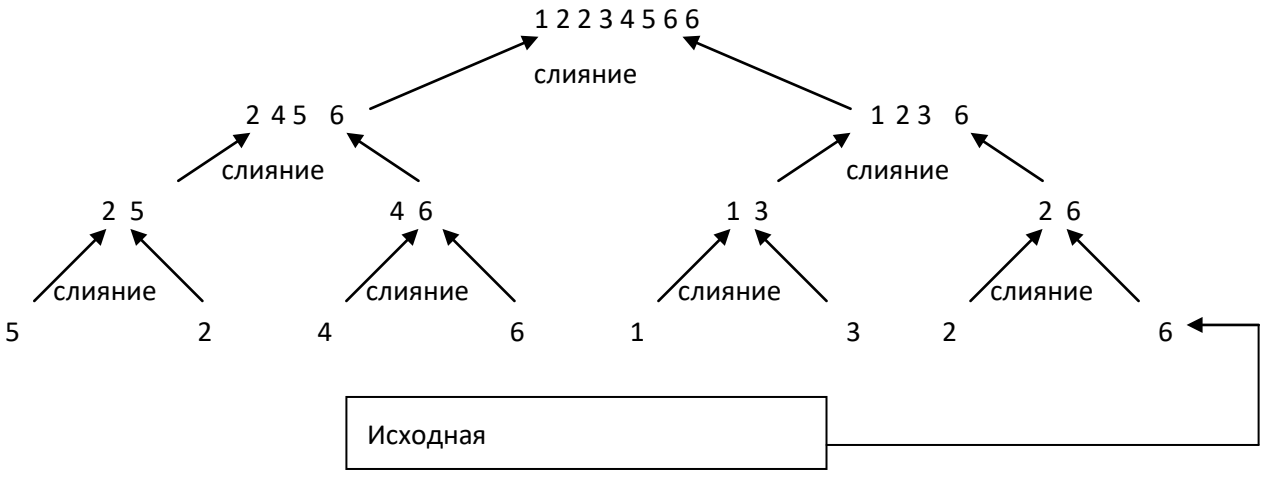
З огляду на, що T (1) = Q (1) = 1 і підставляючи ітерації в основну формулу отримаємо: T (n) = T (1) + 1 + ... + 1 = n = Q (n), так як кількість одиниць справа (поки дійдемо в ітераціях до T (1), і з огляду на T (1) = 1) є n.

Порівнюючи покроковий алгоритм з рекурсивним, відзначаємо їх однакову складність Q (n). Але, враховуючи, рекурсивний алгоритм вимагає для реалізації додаткові ресурси на здійснення рекурсії, слід віддати перевагу покрокового алгоритму.

Розглянемо тепер рекурсивний алгоритм для сортування чисел. Для складання рекурсивного алгоритму ми використовуємо метод "розділяй і володарюй", який наказує, вирішуючи деяку задачу, розбити її на підзадачі і кожну підзадачу вирішити тим же алгоритмом, що і основне завдання. Для сортування вихідний масив розбивається на дві частини меншого розміру (наприклад, на навпіл). Потім обидві половинки упорядковано окремо. Після сортування половинок їх слід з'єднати (злити) разом, виробляючи сортування при злитті. Рекурсивне розбиття завдання на меншу відбувається до тих пір, поки розмір масиву не дійде до одиниці (будь-який масив довжини 1 впорядкований).

Будемо вважати, що поєднання двох упорядкованих масивів в один виробляється за допомогою алгоритму Slianie (A, p, q, r). Параметрами цього алгоритму є ім'я масиву A, і числа p, q, r, яке вказує кордону зливаються ділянок (суміжних в масиві A). Алгоритм передбачає, що p <= q <r і що ділянки A [p..q] і A [q + 1, r] вже відсортовані, і зливайте ихв один A [p..r].

Сортування злиттям для масиву A = <5, 2, 4, 6, 1, 3, 2, 6>:



Нижче наводиться алгоритм сортування злиттям Sort_Slijan (A, p, r), яка сортує ділянку A [p..r] масиву A, не змінюючи решту масиву. При p = r ділянку містить максимум один елемент, і тим самим вже відсортований. В іншому випадку ми відшукаємо число q, яке ділить ділянку на дві приблизно рівні частини A [p..q] і A [q + 1..r], кожна з яких містить □n / 2□ і □n / 2□ елементів. Тут через □x□

позначена ціла частина з недоліком, а через □n / 2□-з надлишком. Sort_Slijan(A, p, r)

		Оценка
1.	if p<r	Q(1)
2.	then q□□(p+r)/2□	Q(1)
3.	Sort_Slijan(A, p, q)	T(n/2)
4.	Sort_Slijan(A, q+1, r)	T(n/2)
5.	Sort_Slijan(A, p, q, r)	f(n)

Увесь масив A тепер можна відсортувати за допомогою виклику Sort_Slijan (A, 1, length [A]), де n = length [A] - розмір масиву. Якщо і є ступінь двійка, т. Е. N = 2k, то в процесі сортування відбудеться злиття пар елементів в відсортовані ділянки довжини 2, 4, 8и т. Д (на останньому кроці зливаються два відсортованих ділянки довгі n / 2)

Для оцінки алгоритму Sort_Slijan спочатку покладемо, що n = 2k. У записі алгоритму Sort_Slijan записані оцінки кожного рядка. Функція T (n) виразиться як:

T (n) = Q (1) + Q (1) + 2T (n / 2) + f (n) або, враховуючи поглинання Q (1)

функцією f (n), T (n) = 2T (n / 2) + f (n). (4.1)

Для вирішення рекурентного співвідношення (4.1) необхідно визначити функцію f (n), для чого потрібно скласти алгоритм Slijanie. Нехай при з'єднанні двох відсортованих масивів ми використовуємо такі дії: порівнюємо поточні (ліві) елементи масивів, менший елемент записуємо в проміжний масив, і зрушуємося вправо на один елемент з масиві з меншим елементом, цю процедуру виробляємо до вичерпання одного з масивів, що залишилися елементи переносимо в проміжний масив, а з нього всі елементи - в масив A.

Замінюючи 2k на n провівши перетворення отримаємо T(n)=n log n.

Можна показати, що в цьому випадку T(n)=Q(n log n).

Тема 14. Складність алгоритму, асимптотичні функції

Аналіз алгоритмів - теоретичне дослідження продуктивності комп'ютерних програм і споживаних ними ресурсів.

Швидкодія - час роботи алгоритму, в залежності від обсягу вхідних даних.

Визначається функцією T (n), де n - обсяг вхідних даних Покроковий метод аналізу алгоритму □ Записується алгоритм на псевдокоді.

□ Кожному кроку алгоритму присвоюється ваговий коефіцієнт □ Підраховується кількість виконань кожного кроку

Функція T (n) є сумою множення вагового коефіцієнта кроку на кількість виконань.

Алгоритм сортування вибором:

i-й (i = 0) елемент береться як мінімальний, його індекс записується в min, j = i +

1

A [min] порівнюється з A [j], якщо цей елемент менше A [min], його індекс зберігається в min і переходимо до наступного елементу, j ++, A [min] і A [i] міняємо місцями ,i ++ , алгоритм повторюється

for(i=0; i < n-1; i++) {	c1	n
min = i ;	c2	n-1
for (j = i+1; j <= n-1; j++)	c3	
if (a[j] < a [min])	c4	
min = j;	c5	
k = a[i];	c6	n-1
a[i] = a[min];	c7	n-1
a[min] = k; }	c8	n-1

$$T(\nu)= \chi1*\nu+\chi2*(\nu-1)+\chi3*\sum_{\iota}^{\nu=-0^{1}}\tau_{\iota}+(\chi4+\chi5)*$$
$$\sum\iota^{\nu=-0^{1}}(\tau_{\iota}-1)+(\chi6+\chi7+\chi8)*(\nu-1)=$$

$$\chi1*\nu+\chi2*(\nu-1)+\chi3*\left(\frac{\nu(1+\nu)}{2}-1\right)+(\chi4+\chi5)*\left(\frac{\nu(\nu-1)}{2}\right)$$
$$+(\chi6+\chi7+\chi8)*(\nu-1)=$$
$$\chi1*\nu+\chi2*(\nu-1)+\chi3*\left(\left(\frac{\nu+2\nu2-1}{\chi3}\right)\right)+(\chi4+\chi5)*\left(\left(\frac{\nu22-\nu}{\chi4\ \chi5}\right)\right)+$$
$$(\chi6+\chi7+\chi8)*(\nu-1)=(\chi1+\chi2+\frac{(c3+c4+c5)}{2}*\nu^{2}-(\chi2+\chi3+\chi6+\chi7+\chi8)\approx \alpha*\nu^{2}+\beta*\nu-\chi$$

Види аналізу.

Найгірший випадок: T (n) - максимальний час для будь-яких вхідних даний розміру n.

Середній випадок: T (n) - очікуваний час для будь-яких вхідних даних розміру n.

Найкращий випадок - мінімальний час роботи.

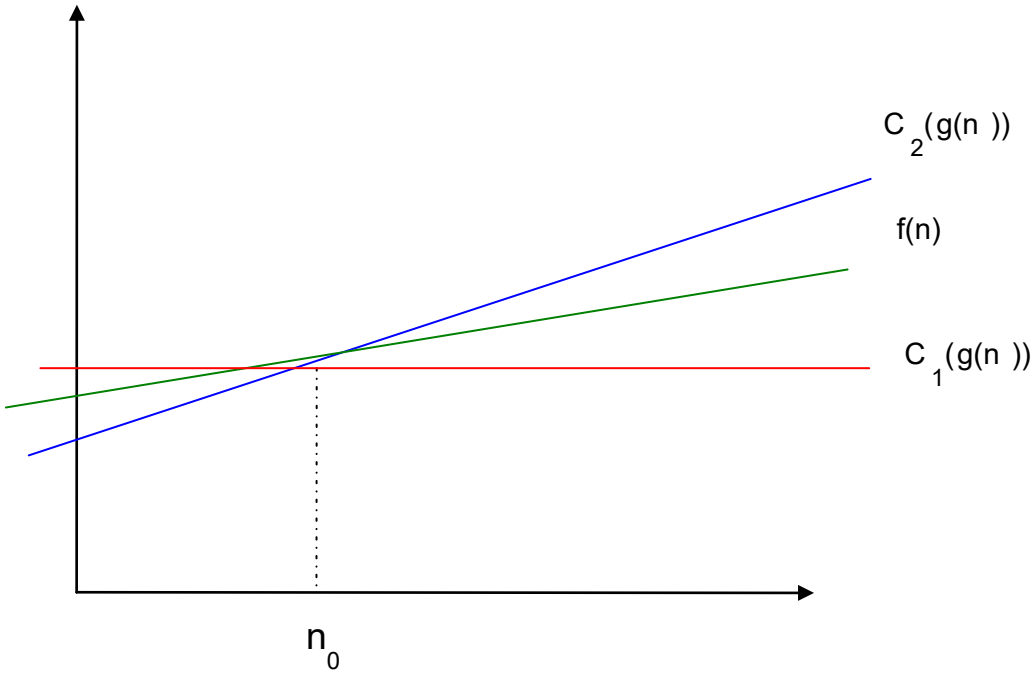
Асимптотична оцінка

f(n)= Θ (g(n)), если найдутся такие c1,c2>0 и n0>0, что

$0 \le c1 \cdot g(n) \le f(n) \le c2 \cdot g(n),\ n \ge n0$

Если f(n)=Θ (g(n)), то g(n) – асимптотически точная оценка для функции f(n)

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$



Тема 15. Аналіз рекурсивних та не рекурсивних алгоритмів

Рекурентне співвідношення – це рівняння або нерівність, яка описує функцію із використанням самої себе, але тільки з меншими аргументами. Наприклад, час роботи процедури MergeSort T(n) в найгіршому випадку описується за допомогою наступного рекурентного співвідношення:

$$T(n) = \begin{cases} \Theta(1) & \text{якщо } n = 1, \\ 2T(n/2) + \Theta(n) & \text{якщо } n > 1. \end{cases}$$

Рішенням цього співвідношення є функція T(n) = Θ(nlgn).

При розгляді рекурентних співвідношень зазвичай вводяться два спрощення. По-перше, приймається, що час роботи алгоритму T(n) визначається лише для цілочислових n, оскільки в більшості алгоритмів кількість вхідних елементів виражається цілим числом.

По-друге, оскільки час роботи алгоритму з вхідними даними фіксованого розміру виражається константою, то в рекурентних співвідношеннях для достатньо малих n зазвичай справедлива тотожність T(n) = Θ(1). Тому для зручності граничні умови рекурентних співвідношень, як правило, відкидаються та приймається, що для малих n час роботи алгоритму є T(n) константою. Тому попереднє рекурентне співвідношення зазвичай буде записуватись як:

$T(\nu) = 2T(\nu/2) + \Theta(\nu)$, ⟨∑| □®⎨ ⎩©⎩ | ◊| ⎨◊ |∑⎨⎨□

$T(\nu)$ ™⎣□ ⎡◊⎣⎦ ν.

Метод підстановки.

Метод підстановки, який застосовується для рішення рекурентних співві дношень, складається з двох етапів:

- робиться припущення про вигляд розв’ язку;
- за допомогою методу математичної індукції визначаються константи та доводиться, що рішення вірне.

Назва методу пояснює, що припущене рішення підставляється у рекурентне рівняння. Цей метод достатньо потужний, але може застосовуватись лише коли легко можна висунути припущення щодо вигляду розв’ язку.

Метод підстановки можна використовувати для визначення або верхньої, або нижньої межі рекурентного співвідношення. В якості прикладу розглянемо верхню границю рекурентного співвідношення:

$$T(n) = 2T(n/2) + n,$$

яке подібне до (4.2). Ми припускаємо, що розв’ язок має вигляд T(n) = O(nlgn). Метод полягає в доведенні того, що при придатному виборі константи c>0 виконується нерівність T(n) ≤ cn lgn. Почнемо з того, що припустимо справедливість цієї нерівності для n / 2 , тобто що виконується співвідношення $T(\nu/2) \le \chi\nu/2\,\lambda\gamma(\nu/2)$. Після підстановки даного виразу в рекурентне співвідношення отримуємо:

$T(n) \le 2c\,n/2$ $\lambda\gamma(\nu/2))+\nu\,\delta\,\chi\nu\,\lambda\gamma(\nu/2)+\nu = \chi\nu\,\lambda\gamma\,\nu - \chi\nu\,\lambda\gamma\,2 + \nu =$

$= \chi\nu\,\lambda\gamma\,\nu - \chi\nu + \nu \le \chi\nu\,\lambda\gamma\,\nu$, де остання нерівність виконується при $c \ge 1$.

Тепер, згідно методу математичної індукції, необхідно довести, що цей розв’ язок справедливий для граничних умов. Зазвичай для цього достатньо показати, що граничні умови є придатною базою для доведення за індукцією. В рекурентному співвідношенні (4.3) необхідно довести, що сталу c можна обрати достатньо великою для того, щоб співвідношення T(n) ≤ cn lgn виконувалось й для граничних умов. Така вимога інколи призводить до проблем. Припустимо, що T(1) = 1 – єдина гранична умова цього співвідношення. Для n = 1 співвідношення T(n) ≤ cn lgn дає T(1) ≤ c × 1 × lg1 = 0, що суперечить припущенню T(1) = 1. Відповідно, даний базис індукції нашого доведення не виконується.

Цю складність, яка виникає при доведенні припущення індукції для вказаної граничної умови, легко обійти. Наприклад, в рекурентному співвідношенні можна використовувати переваги асимптотичних позначень, які вимагають довести нерівність T(n) ≤ cn lgn тільки для великих n ($n^3 n_0$, де n_0 – константа). Це дозволяє в доведенні за методом математичної індукції не враховувати граничну умову T(1) = 1. Так, при n>3 наведене рекурентне співвідношення явним чином від T(1) не залежить. Таким чином, обравши $n_0 = 2$, в якості бази індукції можна розглядати не T(1), а T(2) та T(3). З рекурентного співвідношення слідує, що T(2) = 4, а T(3) = 5. Тепер доведення за математичною індукцією співвідношення T(n) ≤ cn lgn для деякої константи $c^3 1$ можна завершити, обравши її достатньо великою для того, щоб були справедливі нерівності T(2) ≤ 2c lg2 та T(3) ≤ 3c lg3. Для цього достатньо обрати c≥2. В більшості рекурентних співвідношень легко розширити граничні умови таким чином, щоб припущення ін дукції виявилось вірним для малих n.

На жаль, не існує єдиного рецепту як вгадати розв’ язок рекурентного співвідношення. Для цього потрібний досвід, вдача та творче мислення. Проте існують певні евристичні прийоми, які можуть допомогти зробити правильну здогадку.

Якщо рекурентне співвідношення подібне до того, що ми щойно розглянули, то можна припустити, що розв’ язки цих співвідношень будуть подібними.

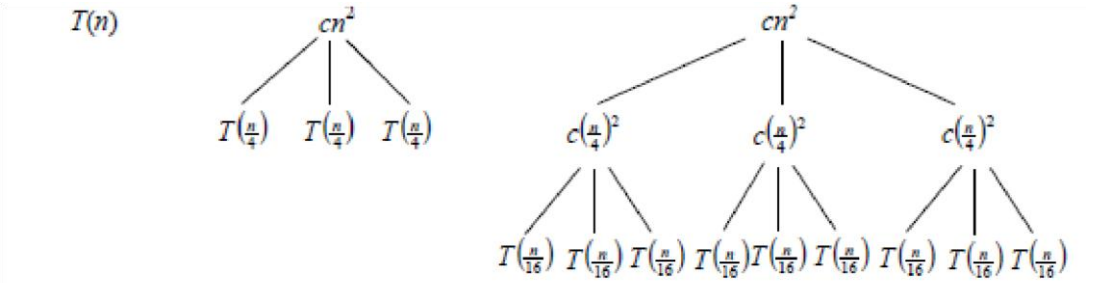
Метод дерев рекурсії.

Як зазначалось, недоліком методу підстановки є необхідність робити припущення щодо розв’ язку рекурентного співвідношення. Метод дерев рекурсії – прямий шлях до вдалого припущення. В дереві рекурсії кожний вузол представляє час, необхідний для виконання окремо взятої підзадачі, яка розв’ язується при одному з багатьох рекурсивних викликах функції. Далі значення часу роботи окремих етапів підсумовується в межах кожного рівня, а потім – по всім рівням дерева, в результаті чого отримуємо повний час роботи алгоритму.

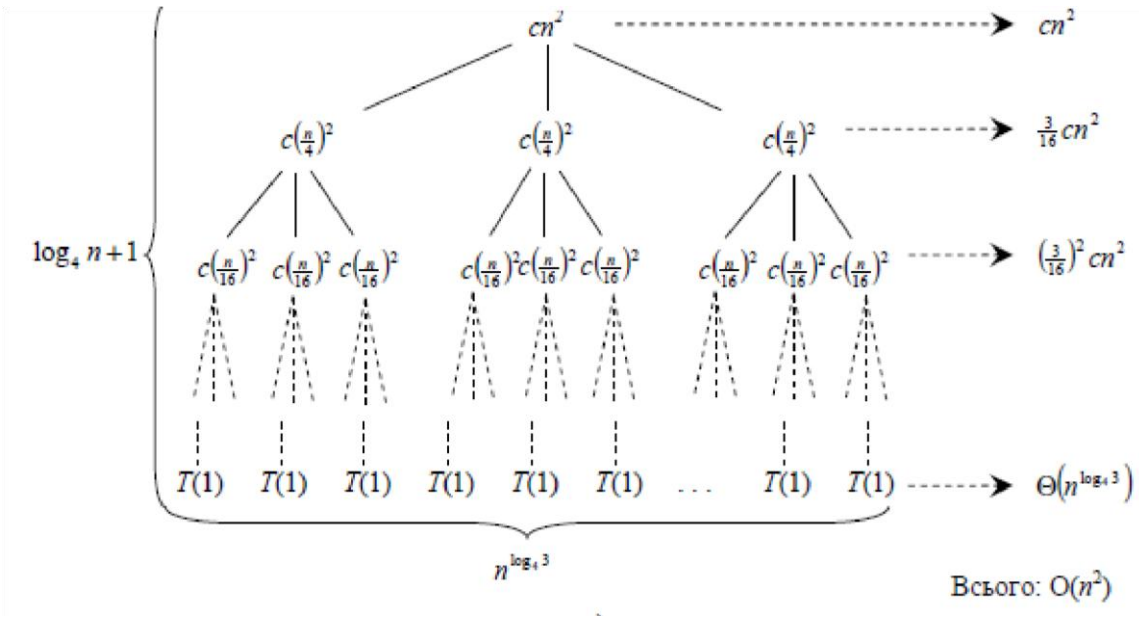
Дерева рекурсії краще за все підходять для того, щоб допомогти зробити припущення про вигляд розв’ язку, який потім перевіряється методом підстановок. При цьому у припущенні часто дозволяються наявність невеликих неточностей, оскільки воно потім все одно перевіряється. Якщо ж побудова дерева рекурсії та підсумування часу роботи по всім складовим відбувається достатньо ретельно, то саме дерево рекурсії може стати джерелом доведення коректності розв’ язку.

Наприклад, розглянемо, як за допомогою дерева рекурсії можна здогадатись про вигляд розв’ язку рекурентного співвідношення $T(n) = 3T(\nu/4) + \Theta(\nu^2)$. Почнемо з того, що оцінимо розв’язок згори. Враховуючи те, що округлення до цілої частини аргументів функцій є несуттєвим, то побудуємо дерево рекурсії для рекурентного співвідношення

$T(\nu) = 3T(\nu/4) + \chi\nu^2$, ™$\Sigma\ \chi > 0$.



a) b) c)



г)

Рис. Побудова дерева рекурсії для рівняння $T(n) = 3T(n/4) + cn^2$

На рис. продемонстрований процес побудови дерева рекурсії для рекурентного співвідношення $T(\nu) = 3T(\nu/4) + \chi\nu^2$. Для зручності вважатимемо, що n – ступінь четвірки. В частині а показана функція T(n), яка потім все більш детально розкривається в частинах б – г у вигляді еквівалентного дерева рекурсії. Член cn2 в корені дерева представляє час верхнього рівня рекурсії, а три піддерева, які починаються в корені, – час виконання підзадач розміром n/4.

По мірі віддалення від кореня розмір підзадач зменшується, зрештою ми дійдемо до граничних умов. Скільки рівнів дерева потрібно побудувати, щоб їх досягнути? Розмір допоміжної задачі, як відповідає рівню i, відповідає n/4i. Таким чином, розмір підзадачі стає рівним 1, коли n/4i = 1 або, що теж саме, коли $i = \log_4 n$. Отже, в дереві всього log 4 n +1 рівнів.

Далі необхідно визначити час виконання кожного рівня дерева. На кожному рівні в три рази більше вузлів, ніж на попередньому, тому кількість вузлів на рівні i дорівнює 3i. Оскільки розміри допоміжних підзадач при спуску на один рівень зменшуються в чотири рази, час виконання кожного вузла на рівні i (i = 0, log 4 n −1) дорівнює c(n / 4i)2 . Помноживши кількість вузлів на рівні на час виконання одного вузла, отримуємо, що сумарний час виконання

допоміжних підзадач, які відповідають вузлам рівня i, дорівнює

3 ι χ(ν / 4ι)2= (3 /16)ι χν 2 . Останній рівень, який знаходиться на глибині log 4 n має $3^{\log_4 n} = n^{\log_4 3}$ вузлів, кожний з яких дає у загальний час роботи внесок T (1).

Тепер необхідно підсумувати час роботи всіх рівнів дерева та визначити загальний час роботи дерева:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta\left(n^{\log_4 3}\right) =$$
$$= \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1}cn^2 + \Theta\left(n^{\log_4 3}\right).$$

Враховуючи, що ми маємо справу в цій формулі із геометричною прогресією, знаменник якої менше одиниці, а отже прогресія спадає. Таким чином, ми можемо обмежити згори суму у формулі сумою нескінченної спадаючої геометричної прогресії:

$$T(n) = \sum_{i=0}^{\log_4 n-1}\left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) < \sum_{i=0}^{\infty}\left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) =$$
$$= \frac{1}{1-(3/16)}cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{16}{13}cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2).$$

Отже, для початкового рекурентного співвідношення T (ν) = 3T (ν / 4)+ χν2 отримане припущення щодо розв’язку у вигляді T(n) = O(n2). Коефіцієнти при cn2 утворюють геометричну прогресію, а їх сума, обмежена згори константою 16/13. Оскільки вклад кореня в повний час роботи

дерева становить cn2, то час роботи кореня є деякою сталою частиною від загального часу роботи дерева в цілому. Іншими словами, повний час роботи дерева здебільшого визначається часом роботи його кореня. 2

Насправді, оцінка O(n) повинна бути також асимптотично точною. Адже перший рекурсивний виклик дає внесок в загальний час роботи алгоритму, який виражається як Θ(n2), тому нижня границя часу роботи дерева також буде

 (n2).

Тепер за допомогою методу підстановок перевіримо коректність нашого припущення. Необхідно показати, що для деякої константи d > 0 виконується нерівність T (n) dn 2 . Використовуючи сталу c > 0, отримуємо:

$$T(n) \le 3T(\lfloor n/4 \rfloor) + cn^2 \le 3d \lfloor n/4 \rfloor^2 + cn^2 \le 3d\,(n/4)^2 + cn^2 = \frac{3}{16} dn^2 + cn^2 \le dn^2,$$

де

остання нерівність виконується при d ≥ (16 /13)c .

Основний метод.

Основний метод є свого роду «збіркою рецептів», за якими будується розв' язок рекурентних співвідношень вигляду

T (n) = aT (n / b) + f (n) ,

де a³1 та b>1 – константи, а f(n) – асимптотично додатна функція. Метод складається лише з трьох випадків, які легко запам' ятати та застосовувати на практиці для аналізу рекурентних співвідношень.

Рекурентне співвідношення (4.4) описує час роботи алгоритму, в якому задача розміру n розбивається на а допоміжних підзадач, розміром n/b кожна. Отримані в результаті розбиття а підзадач розв' язуються рекурсивним методом, причому час їх розв' язку становить T (n / b) . Час, необхідний на розбиття задач та об' єднання результатів цих підзадач, описується функцією f(n). Наприклад, для рекурентного рівняння, яке виникає при аналізі процедури MergeSort, a = 2, b = 2, а f(n) = Θ(n).

Строго кажучи, при визначення наведеного вище рекурентного співвідношення допущена неточність, оскільки число n/b може не бути цілим. Однак заміна кожного з а доданків T (n / b) виразом T (n / b) або T (n / b) не впливає на асимптотичну поведінку рішення.

Основний метод ґрунтується на наступній теоремі.

Основна теорема. Нехай a³1 та b>1 – константи, f(n) – довільна функція, а T(n) – функція, визначена на множні невід' ємних цілих чисел за допомогою рекурентного співвідношення T (n) = aT (n / b) + f (n) , де вираз n/b інтерпретується або як n / b , або як n / b . Тоді асимптотичну поведінку функції T(n) можна виразити наступним чином.

Якщо ϕ (ν) = O(ν λοgβ α –ε) для деякої сталої ε>0, то T (n) = Θ(ν λοgβ α).

Якщо ϕ (ν) = Θ(νλοgβ α), ∫⌊ T (ν) = Θ(ν λοgβ α λγ ν).

Якщо f (n) = W(νλογ β α +ε) для деякої сталої ε>0, і якщо af (n / b) ≤ cf (n) для деякої сталої c<1 і всіх достатньо великих n, то T (ν) = Θ(ϕ (ν)).

Ідея основної теореми полягає в наступному. В кожному з трьох означених випадків функція f(n) порівнюється з функцією n logb a . Інтуїтивно зрозуміло, що

асимптотична поведінка розв’ язку рекурентного співвідношення визначається більшою з двох функцій. Якщо більшою є функція n b (випадок 1), то розв’ язок – T (n) nb .

Якщо швидше зростає функція f(n) (випадок 3), то розв’ язок – T (n) = Θ(ϕ(ν)). Якщо ж обидві функції зрівнянні (випадок 2), то відбувається множення на логарифмічний додаток.

В першому випадку функція f(n) повинна бути не просто менше функції n logb a , а поліноміально менше. Це означає, що функція f(n) повинна бути асимптотично менше функції n logb a в nε разів. Аналогічно, в третьому випадку функція f(n) повинна бути поліноміально більшою, а окрім того задовольняти умові регулярності: af (n / b) ≤ cf (n) . Ця умова виконується

для більшості поліноміально обмежених функцій.

Важливо розуміти, що цими трьома випадками поведінка функції f(n) не обмежується. Між випадками 1 та 2 є проміжок, в якому функція f(n) менше за n logb a , але не поліноміально менше. Аналогічний проміжок є між випадками 2 та 3, коли функція f(n) більше n logb a , але не поліноміально більше. Якщо функція f(n) потрапляє в один з цих проміжків, або якщо для неї не виконуються умови регулярності випадку 3, основний метод не можна застосовувати для рекурентних співвідношень.

Питання для підсумкового контролю

1. Асимптотична оцінка алгоритму
2. Методика оцінки не рекурсивних алгоритмів. Приклад
3. Рекурсивні алгоритми, побудова асимптотичної оцінки. Приклад
4. Методика оцінки рекурсивних алгоритмів
5. Алгоритми сортування масивів за поліноміальний час
6. Швидкі алгоритми сортування масивів, засновані на порівнянні елементів
7. Алгоритми сортування масивів за лінійний час
8. Лінійні структури даних, основні операції та характеристики
9. Дерева, види, способи представлення, структури даних, обходи дерева
10. Двійкові дерева пошуку, операції додавання елементів
11. Двійкові дерева пошуку, операції видалення елементів і пошуку наступних
12. АВЛ-дерева, основні операції
13. Двійкові купи, основні операції та характеристики
14. Хеш-таблиці, метод ланцюжків
15. Хеш-таблиці, відкрита адресація
16. Динамічне програмування, основні особливості
17. Жадібні алгоритми, основні особливості
18. Побудова коду Хаффмена
19. Алгоритм обходу графів в ширину
20. Алгоритм обходу графів у глибину
21. Остове дерево, класифікація ребер, топологічне сортування
22. Сильно зв'язкові компоненти, алгоритм пошуку
23. Побудова мінімального покриваючого дерева, алгоритм Крускала
24. Побудова мінімального покриваючого дерева, алгоритм Прима
25. Пошук найкоротшого шляху. Алгоритм Дейкстри
26. Пошук найкоротшого шляху. Алгоритм Беллмана-Форда
27. Проблематика, поняття, призначення та засоби АСД.
28. Списки. Формування та оброблення.
29. Організація роботи зі структурами стек, черга й дек.

30. Реалізація списків на динамічній пам’яті.
31. Реалізація списків на базі масивів.
32. Ознозв’язні та двозв’язні списки.
33. Оцінювання складності та типи алгоритмів.
34. Дерева. Способи формування та оброблення. Бінарне дерево.
35. Алгоритми обходу дерев.
36. Графи. Формування та оброблення.
37. Хеш-таблиці. Побудова та застосування.
37. Квадратичні алгоритми сортування.
38. Логарифмічні алгоритми сортування.
39. Лінійні алгоритми сортування.
40. Спосіб кишенькового сортування.
41. Суть та види алгоритмів пошуку.
42. Пошук у хеш-таблиці.
43. Суть та типи алгоритмів пошуку послідовностей.
44. Алгоритм Карпа-Рабіна. Застосування.
45. Ефективність застосування знань з дисципліни АСД.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Шаховська, Н. Б. Алгоритми і структури даних – Львів : Магнолія, 2016 – 215 с.

2. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск / Д. Кнут. – [2е изд.]. – М. : Вильямс, 2014. – 824 с.

3. Алгоритмы: построение и анализ / [Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн]. – [3-е изд.]. – М. : Вильямс, 2013. – 1328 с.

4. Лафоре, Р. Структуры данных и алгоритмы в Java – СПб: Питер, 2013 – 702 с.

5. Томас Х. Кормен. Алгоритмы: вводный курс / Томас Х. Кормен. – М. : Вильямс, 2013. – 208 с.

6. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М. : Вильямс, 2009. – 400 с.

7. Т.Кормен, Ч.Лейзерсон, Р.Ривест. Алгоритмы: построение и анализ. – М.: Издат-во "Вильямс", 2005 – 1296 с.

8. Карпов Ю.Г. Теория и технология программирования. Основы построения трансляторов. – СПб.:БХВ-Петербург, 2005. – 272 с.

9. Стюарт Рассел, Питер Норвиг Искусственный интеллект: современный подход(AIMA) - 2-е узд.: Пер. с англ. - М.: Издательский дом «Вильямс», 2005. – 1424 с.

10. Кормен Т., Лейзерсон Ч. , Ривест Р. Алгоритмы: построение и анализ - 2-е изд. - М. : БИНОМ. Лаб. знаний : МЦНМО, 2004. - 955с.

Допоміжна

1 Потопахин В.В. Искусство алгоритмизации. – М.: ДМК Пресс, 2015. – 320с.

2.Генри С. Уоррен. Алгоритмические трюки для программистов / Генри С. Уоррен. – [2-е изд.]. – М. : Вильямс, 2013. – 512 с.

3. Измайлов А.Ф., Солодов М.В. Численные методы оптимизации. – М.: Физматлит, 2003.– 451с. 4. Самарский А.А. Введение в численные методы. – М.: Наука, 1982. – 269 с.

4. Кнут Д. Мистецтво програмування. Том 1,2,3. – М.: Видавничий будинок «Вільямс», 2003. – 720 с.

5. Р. Седжвик. Фундаментальные алгоритмы на С++ . – М.: 2001. – 687 с.

Інформаційні ресурси

1. Національна бібліотека України ім. В.І. Вернадського : веб-сайт. URL: http://www.nbuv.gov.ua.
2. Он-лайн бібліотека. URL: http://www.lib.com.ua
3. http://www.info-library.com.ua/books-book-149.html

Навчальне видання

Мірошник Марина Анатоліївни
Григор'єва Тетяна Ігорівна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Конспект лекцій