

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

Циклова комісія програмної інженерії

**МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
до виконання лабораторних робіт з дисципліни**

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

спеціальність	121 Інженерія програмного забезпечення
освітня програма	Розробка програмного забезпечення (назва освітньої програми)
відділення	Комп'ютерної та програмної інженерії (назва відділення)

**Дніпро
2024 рік**

ЗМІСТ

1.	Лабораторна робота №№ 1-2. Складання та налагодження програм обробки однозв'язного списку.	2
2.	Лабораторна робота №№ 3-4. Складання та налагодження програм обробки черги та стеку.	15
3.	Лабораторна робота №№ 5-6. Складання та налагодження програми рішення задачі створення та обходу бінарних дерев.	21
4.	Лабораторна робота №№ 7-8. Складання та налагодження програми з використанням рекурсивного перебору з поверненням.	29
5.	Лабораторна робота № 9. Складання та налагодження програм реалізації алгоритмів сортування елементів масиву даних.	33
6.	Лабораторна робота № 10. Складання та налагодження програм пошуку в лінійних масивах даних.	45
7.	Лабораторна робота №№ 11-12. Складання програм пошуку по збалансованому бінарному дереву.	51
8.	Лабораторна робота №№ 13-14. Складання програми обходу неорієнтованого графу.	59
9.	Лабораторна робота № 15. Складання та налагодження програми реалізації алгоритму Дейкстри пошуку найкоротших шляхів.	65
10.	Лабораторна робота № 16. Складання програми рішення задачі комівояжера методом гілок та границь.	69

Методичні рекомендації до виконання лабораторної роботи №№ 1-2

Тема: Розробка основних функцій обробки зв'язаного списку. Складання та налагодження програм обробки однозв'язних списків.

Мета: Отримання практичних навиків в формуванні та обробці динамічної структури даних – однозв'язний список.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Лінійний список - це динамічна структура даних, кожен елемент якої за допомогою покажчика зв'язується з наступним елементом.

З визначення треба, що кожен елемент списку містить поле даних (Data) (воно може мати складну структуру) і поле посилання на наступний елемент (next). Поле посилання останнього елемента повинне містити порожній покажчик (NULL).

Тому що посилання всього одне (тільки на наступний елемент), те такий список є однозв'язним. Коли говорять про лінійний список, як правило, мають на увазі саме однозв'язний список.

Приклад. Сформувати список, що містить цілі числа 3, 5, 1, 9.

Рішення. При роботі з динамічними структурами даних **можна рекомендувати** наступний порядок дій.

а) Насамперед необхідно визначити дві структури:

1. структура, що містить характеристики даних, тобто всі ті поля з даними, які необхідні для рішення поставленого завдання (у нашому випадку є всього одне поле цілого типу). Назвемо цю структуру Data;
2. структура, що містить поле типу Data і поле - адреса наступного елемента next. Другу структуру назвемо List.

Тексти цих структур необхідно розташувати на початку програми (до main() і інших функцій). От можлива реалізація структур:

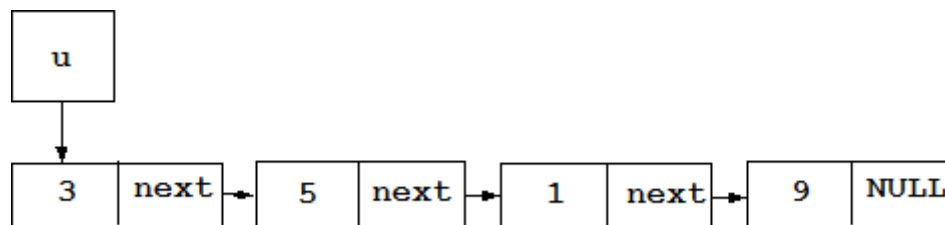
```

struct Data
{
    int a;
};
struct List
{
    Data d;
    List *next;
};

```

Такий підхід дозволить надалі змінювати в широких межах структуру із власно даними, ніяк не зачіпаючи при цьому основну структуру **List**.

Отже, ми описали структурний тип, за допомогою якого можна створити наш однозв'язний список. Графічно створюваний список можна зобразити так, як це показано на малюнку нижче:



б) У програмі (звичайно у функції `main()`) визначаємо покажчик на початок майбутнього списку:

```
List *u = NULL;
```

Поки список порожній, покажчик явно заданий рівним константі **NULL**.

в) Виконуємо первісне заповнення списку.

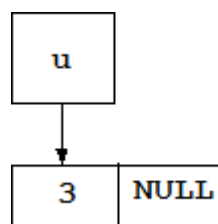
Створимо перший елемент:

```
u = new List; // Виділяємо пам'ять під елемент списку
```

```
u->d.a = 3; // Заповнюємо поле даних
```

```
u->next = NULL; // Покажчик на наступний елемент пустий
```

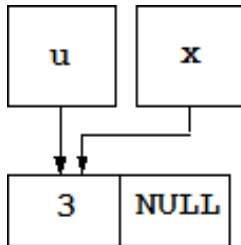
Після включення першого елемента список можна зобразити так:



Продовжимо формування списку, додаючи нові елементи в його кінець. Для зручності введемо допоміжний показчик, що буде зберігати адресу останнього елемента списку:

```
List *x;
```

```
x = u; // Зараз останній елемент списку співпадає з його початком
```



Таким чином, до області пам'яті можна звернутися через два показчики.

Виділяємо місце в пам'яті для наступного елемента списку й перенаправляємо показчик **x** на виділену область пам'яті:

```
x->next = new List;
```

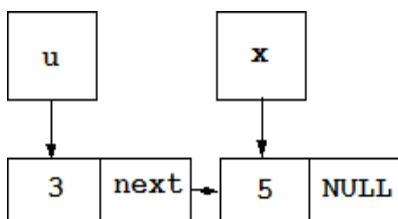
```
x = x->next;
```

Потім визначаємо значення цього елемента списку:

```
x->d.a = 5;
```

```
x->next = NULL;
```

Отримуємо таку схему:



Цей процес продовжуємо доти , поки не буде сформований весь список.

Дії зі сформованим списком

Сформувавши початковий список, можна виконувати з ним різні дії. Настійно рекомендується кожен операцію зі списком оформляти у вигляді окремої функції.

Такий підхід помітно спрощує розробку програми й можливо її подальшу модифікацію. Зрозуміло, що й тільки що розглянуте формування початкового списку також краще записати у вигляді функції.

1. *Перегляд списку* - здійснюється послідовно, починаючи з його початку. Показчик послідовно посилається на перший, другий, і т.д. елементи списку доти, поки весь список не буде пройдений. Приведемо приклад реалізації перегляду, наприклад, для виведення вмісту списку на екран монітора:

```
void Print(List *u)
{
    List *p = u;
    cout << "Spisok:" << endl;
    while(p)
    {
        cout << p->d.a << endl;
        p = p->next;
    }
}
```

Звернутися до функції можна так:

```
Print(u);
```

Тут і далі в прикладах *u* - це показчик на початок списку.

2. *Пошук першого входження* в список елемента, що відповідає заданим вимогам:

```
List * Find(List *u, Data &x)
{
    List *p = u;
    while(p)
    {
        if(p->d.a == x.a) // умова для пошуку
            return p;
    }
}
```

```

    p = p->next;
}
return 0;
}

```

Можливий виклик функції:

```
List * v = Find(u, x);
```

где **x** — об'єкт типа **Data**.

3. Додавання нового елементу в початок списку:

```
void Add_Beg(List **u, Data &x)
```

```

{
    List *t = new List;
    t->d.a = x.a;
    t->next = *u;
    *u = t;
}

```

Можливий виклик функції:

```
Add_Beg(&u, x);
```

Де **x** — об'єкт типу **Data**.

4. *Вставка нового елемента в довільне місце списку* по якому-небудь принципу, наприклад, вставка у відсортований по зростанню список:

```
void Insert(List **u, Data &x)
```

```

{
    /* вставка в список одного елемента перед элементом,
    меншим або рівним даному x */

    List *p = new List;
    p->d.a = x.a;
    if(*u == 0) // якщо список пустий – вставляємо в початок
    {
        p->next = 0;
    }
}

```

```

    *u = p;
    return;
}

```

```

List *t = *u;
if(t->d.a >= p->d.a) // список не пустий - вставка в початок
{
    p->next = t;
    *u = p;
    return;
}

```

```

List *t1 = t->next;
while(t1)
{
    if(t->d.a < p->d.a && p->d.a <= t1->d.a)
    { // вставка в середину
        t->next = p;
        p->next = t1;
        return;
    }
    t = t1;
    t1 = t1->next;
}

```

```

t->next = p; // додаємо в кінець списку
p->next = 0;
}

```

Можливий виклик функції:

Insert(&u, x)

де **x** — об'єкт типу **Data**.

Ця функція дозволяє відразу формувати впорядкований список.

5. Видалення елементу із лінійного списку:

void Delete(List **u, Data &x)

```
{
    if(*u == 0) // список пустий – видаляти нічого!
    {
        return;
    }
    List *t = *u;
    if(t->d.a == x.a) // список не пустий – видаляється початок
    {
        *u = t->next;
        delete t;
        return;
    }
    List *t1 = t->next;
    while(t1)
    {
        if(t1->d.a == x.a)
        // список не пустий - видаляється елемент із середини
        {
            t->next = t1->next;
            delete t1;
            return;
        }
        t = t1;
        t1 = t1->next;
    }
}
```

Можливий виклик функції:

Delete(&u, x);

де **x** — об'єкт типу **Data**.

6. *Видалення (очищення)* усього списку

Коли дані, що зберігаються в списку, стають непотрібними, можна очистити весь список, тобто звільнити пам'ять, що займали всі елементи списку. Виконувати цю операцію бажано відразу після того, як список став не потрібний. Реалізація цієї операції може бути такою:

```
void Clear(List **u)  
{  
    if(*u == 0) return;  
    List *p = *u;  
    List *t;  
    while(p)  
    {  
        t = p;  
        p = p->next;  
        delete t;  
    }  
    *u = 0;  
}
```

Можливий виклик функції:

Clear(&u);

Ми розглянули найбільш важливі дії зі списками. За аналогією з ними можна реалізувати й інші дії, які можуть знадобитися для рішення практичних завдань.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Скласти та налагодити програму обробки однозв'язного списку за алгоритмом згідно Вашого варіанту. Програма повинна задовольняти наступним вимогам:

- організувати користувацьке меню, яке повинно містити наступні пункти:
 1. Формування списку.
 2. Перегляд вмісту списку.
 3. Обробка списку.
 4. Видалення списку.
- забезпечити коректне введення користувачем вхідних даних;
- при обробці списку враховувати, що шукані елементи можуть бути відсутні. В цьому випадку вивести користувачеві відповідне повідомлення;
- введення та виведення вхідних та вихідних даних повинно містити необхідні для користувача повідомлення.

2 Розробити та налагодити програму рішення задачі.

3 Оформити звіт з лабораторної роботи.

Звіт повинен містити наступні розділи:

- 1 Постановка задачі: загальна постановка завдання + Ваш варіант.
- 2 Тексти реалізованих функцій обробку списку з відповідними коментарями (приклади лістингів функцій обробки списку дивись в лекції).
- 3 Текст програми з відповідними коментарями.
- 4 Копії вікон виконання програми.

Розробити тестові набори вхідних даних, що демонструють всі можливі варіанти роботи програми та помістити в звіт копії вікон виконання програми для кожного тестового набору.

5 Висновок.

ВАРІАНТИ ЗАВДАНЬ ЛАБОРАТОРНОЇ РОБОТИ

1. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Список повинен вміщати як додатні, так і від'ємні числа. Знайти суму додатних елементів списку. Видалити із списку всі додатні елементи.
2. Створити однозв'язний список натуральних чисел введенням нового елементу в кінець списку. Знайти та вивести суму парних елементів списку. Видалити із списку всі парні елементи.
3. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Список повинен вміщати як додатні, так і від'ємні числа. Знайти та вивести добуток та підрахувати кількість від'ємних елементів списку. Видалити із списку всі від'ємні елементи.
4. Створити однозв'язний список натуральних чисел введенням нового елементу в кінець списку. Знайти та вивести суму квадратів парних елементів списку. Видалити із списку всі парні елементи та вивести оновлений список. Видалити список.
5. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Замінити всі від'ємні числа в списку на їх модулі. Видалити із списку всі додатні елементи.
6. Створити однозв'язний список дійсних чисел введенням нового елементу в кінець списку. Замінити всі додатні числа їх квадратами. Видалити всі від'ємні елементи списку.
7. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку.. Знайти та вивести максимальний та мінімальний елементи списку. Видалити із списку перший мінімальний та останній максимальний елементи.
8. Створити однозв'язний список дійсних чисел введенням нового елементу в кінець списку. Знайти суму квадратів від'ємних елементів списку. Видалити із списку перший від'ємний елемент.

9. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Знайти та вивести максимальний та мінімальний елементи списку та їх суму. Видалити із списку всі мінімальні елементи.
10. Створити однозв'язний список дійсних чисел введенням нового елементу в кінець списку. Замінити всі від'ємні числа списку максимальним елементом. Видалити із списку останній елемент.
11. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Створити, введенням нового елементу в кінець, новий список, який вміщає тільки додатні елементи першого.
12. Створити однозв'язний список дійсних чисел введенням нового елементу в кінець списку. Знайти та вивести мінімальний елемент списку. Видалити із списку всі мінімальні елементи.
13. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Створити новий список, який вміщає тільки квадрати від'ємних елементів першого списку. Видалити із нового списку два перші елементи.
14. Створити однозв'язний список натуральних чисел введенням нового елементу в кінець списку. Знайти та вивести елементи списку, кратні 3. Видалити із списку всі ці елементи.
15. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Знайти та вивести максимальний елемент списку. Видалити із списку всі максимальні елементи.
16. Створити однозв'язний список дійсних чисел введенням нового елементу в кінець списку. Знайти суму додатних елементів списку. Видалити із списку всі додатні елементи.
17. Створити однозв'язний список натуральних чисел введенням нового елементу в початок списку. Знайти та вивести добуток парних елементів списку. Видалити із списку всі парні елементи.
18. Створити однозв'язний список дійсних чисел введенням нового елементу в початок списку. Знайти добуток та підрахувати кількість від'ємних елементів списку. Видалити із списку всі від'ємні елементи.

19. Створити однозв'язний список натуральних чисел введенням нового елемента в кінець списку. Знайти та вивести суму квадратів парних елементів списку. Видалити із списку всі парні елементи.
20. Створити однозв'язний список дійсних чисел введенням нового елемента в початок списку. Замінити всі від'ємні числа в списку на їх модулі. Видалити із списку ті елементи, значення яких належать інтервалу $[a, b]$, заданому користувачем.
21. Створити однозв'язний список дійсних чисел введенням нового елемента в кінець списку. Замінити всі додатні числа їх квадратами. Видалити із списку перших два від'ємних числа.
22. Створити однозв'язний список дійсних чисел введенням нового елемента в початок списку. Знайти та вивести максимальний та мінімальний елементи списку. Видалити із списку всі мінімальні та максимальні елементи.
23. Створити однозв'язний список дійсних чисел введенням нового елемента в кінець списку. Знайти суму квадратів від'ємних елементів списку. Видалити із списку перший від'ємний елемент та останній елемент списку.
24. Створити однозв'язний список дійсних чисел введенням нового елемента в початок списку. Знайти та вивести максимальний та мінімальний елементи списку та їх суму. Видалити із списку перший мінімальний елемент та останній максимальний елемент.
25. Створити однозв'язний список дійсних чисел введенням нового елемента в початок списку. Замінити всі від'ємні числа їх квадратами. Знайти перший максимальний елемент списку та видалити його.

КОНТРОЛЬНІ ПИТАННЯ

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

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

- 1 Як визначити об'єкт для списку, елементами якого є числа.

- 2 Як визначити об'єкт для списку, елементами якого є слова.
- 3 Як визначити об'єкт для списку, елементами якого є покажчики.
- 4 Визначено змінні: `zvp *fst = NULL, *en, *r`. Покажчики `fst, en` - відповідно адреси першого й останнього елемента списку; `r` - для значень адрес змінних типу `zvp`. Сформовано список з елементами типу `zvp`. Напишіть коди перебору елементів списку.
- 5 Перелічите інформацію, що необхідна, для того щоб видалити зі списку зі значенням рівним деякому числу `k`.
- 6 Написати фрагмент програми, що видаляє необхідний елемент зі списку.
- 7 Перелічите інформацію, що необхідна, для того щоб у список додати елемент зі значенням рівним деякому числу `k`.

Методичні рекомендації до виконання лабораторної роботи №№ 3-4

Тема: Розробка основних функцій обробки черги та стеку. Складання та налагодження програм обробки черги та стеку.

Мета: Отримання практичних навиків в формуванні та обробці «черги» та «стеку» на основі однозв'язного списку.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Всі лінійні динамічні структури даних можуть бути віднесені до спискових структур. Саме такими є різні види черг і стеки. Динамічними ці структури вважаються з тієї причини, що їхній розмір у процесі роботи може змінюватися - список може подовжуватися або коротшати, - а елементи структур (включаючи найперші) створюються й руйнуються за допомогою операцій або процедур динамічного виділення й звільнення пам'яті.

Почнемо розгляд цих структур даних з найбільш простих, тобто із черг. У загальному випадку, черга - це лінійний список, доступ до елементів якого відбувається за принципом FIFO (First In and First Out - першим прийшов і першим пішов). Для черги характерні дві операції:

- занесення елемента в чергу;
- витяг (зчитування) елемента із черги.

У простій черги для роботи з даними доступні дві позиції - початок (із цієї позиції відбувається витяг) і кінець (у цю позицію заноситься вхідний елемент) або "голова" й "хвіст". Довільний доступ до елементів, у відмінності від масивів, формально не допускається.

Операція витягу (зчитування) формально є руйнуючою. Це означає, що оброблені дані стають недоступними. Можливо, явного руйнування (знищення) даних і не відбувається (у разі реалізації черги на лінійному масиві), але до них немає доступу.

Області застосування черг можуть бути розділені на дві групи - системне застосування й прикладне. До застосування черг у системних цілях ставляться:

- диспетчеризація завдань операційною системою;
- буферизація введення/виведення;

Прикладне застосування:

- моделювання процесів (наприклад, систем масового обслуговування);
- використання черг як допоміжних структур даних у яких-небудь алгоритмах (наприклад, при пошуку в графах).

Стек - різновид черги (а значить і різновид списку), але з іншим правилом доступу - LIFO (Last In and First Out - останнім прийшов і першим пішов). Ще один термін

(рідко використовуваний) для позначення цієї структури даних - магазин. У стеці доступна єдина позиція - та, у якій перебуває останній введений елемент - вершина. Іноді позицію, від якої стек почав заповнюватися даними, називають дном стека (хоча особливого застосування цей термін не має, оскільки дно стека, у якому втримується хоча б один елемент даних, формально недоступно. Одне з деяких раціональних застосувань терміна "дно" - ознака порожнього стека, коли дно й вершина збігаються).

Для стека, як і для черги, характерні дві операції - збереження й витяг, але застосовуються вони до однієї й тій же позиції. Операція витягу видаляє елемент зі списку й знищує його вміст. Довільний доступ до елементів стека не допускається. Області застосування стеків такі ж, як й у черг:

Системні:

- передача процедур і функціям аргументів за значенням;
- збереження й відновлення вмісту регістрів загального призначення процесора при виклику процедур (прологи й епілоги функцій).

Прикладні:

- стекова (магазинна) пам'ять, наприклад у мові програмування Forth;
- допоміжні структури даних (наприклад, при пошуку в графі).

Приклади стеків: Друга назва стека - магазин, - приводить до механічного прикладу стека - магазин стрілецької зброї. Ще один простий приклад - склянка. Стеки, також як і черги, можуть реалізовуватися за допомогою одномірних масивів або зв'язних списків.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Скласти та налагодити програму обробки «черги» та «стеку» на основі одностов'язного списку за алгоритмом згідно Вашого варіанту. Програма повинна задовольняти наступним вимогам:

- організувати користувацьке меню, яке повинно містити наступні пункти:
 1. Додавання елемента в чергу.
 2. Обробка всієї черги. (Результатом роботи цього пункту повинно бути виведення результату обробки черги згідно Вашого варіанту).
 3. Обробка всього стеку. (Результатом роботи цього пункту повинно бути виведення результату обробки стеку згідно Вашого варіанту).
- забезпечити коректне введення користувачем вхідних даних;
- при обробці списку враховувати, що шукані елементи можуть бути відсутні. В цьому випадку вивести користувачеві відповідне повідомлення;

— введення та виведення вхідних та вихідних даних повинно містити необхідні для користувача повідомлення.

2 Розробити та налагодити програму рішення задачі.

3 Оформити звіт з лабораторної роботи.

Звіт повинен містити наступні розділи:

1 Постановка задачі.

2 Текст програми з відповідними коментарями.

3 Копії вікон виконання програми.

Розробити тестові набори вхідних даних, що демонструють всі можливі варіанти роботи програми та помістити в звіт копії вікон виконання програми для кожного тестового набору.

4 Висновок.

ВАРІАНТИ ЗАВДАНЬ ЛАБОРАТОРНОЇ РОБОТИ

1 Знайти добуток тих елементів черги, значення яких належать діапазону $[x; y]$ (x та y вводяться користувачем). Ті елементи черги, які не належать діапазону $[x; y]$, помістити в стек. Підрахувати кількість елементів в стеку.

2 Знайти суму непарних елементів черги. Парні елементи черги помістити в стек. Підрахувати добуток елементів в стеку.

3 Знайти кількість від'ємних елементів черги. Додатні елементи черги помістити в стек. Підрахувати суму елементів в стеку.

4 Знайти середнє арифметичне парних додатних елементів черги. Непарні елементи черги помістити в стек. Підрахувати суму елементів в стеку.

5 Порахувати кількість елементів черги, значення яких належать діапазону $[x; y]$ (x та y вводяться користувачем). Ті елементи черги, які не належать діапазону $[x; y]$, помістити в стек. Підрахувати кількість елементів в стеку.

6 Знайти добуток від'ємних елементів черги. Додатні елементи черги помістити в стек. Підрахувати кількість елементів в стеку.

7 Порахувати кількість від'ємних елементів черги. Додатні елементи черги помістити в стек. Підрахувати добуток елементів в стеку.

8 Порахувати кількість парних додатних елементів черги. Від'ємні елементи черги помістити в стек. Підрахувати кількість елементів в стеку.

9 Знайти суму елементів черги з діапазону $[x; y]$ (x та y вводяться користувачем). Ті елементи черги, які не належать діапазону $[x; y]$, помістити в стек. Підрахувати кількість додатних елементів в стеку.

10 Знайти суму парних елементів черги. Непарні елементи черги помістити в стек. Підрахувати добуток від'ємних елементів в стеку.

11 Порахувати кількість парних елементів черги. Непарні додатні елементи черги помістити в стек. Підрахувати кількість елементів в стеку.

12 Знайти середнє арифметичне елементів черги з діапазону $[x; y]$ (x та y вводяться користувачем). Ті елементи черги, які не належать діапазону $[x; y]$, помістити в стек. Підрахувати суму елементів в стеку.

13 Знайти добуток парних елементів черги. Непарні елементи черги помістити в стек. Підрахувати суму додатних елементів в стеку.

14 Знайти середнє арифметичне непарних елементів черги. Парні додатні елементи черги помістити в стек. Підрахувати суму елементів в стеку.

15 Знайти суму від'ємних елементів черги. Додатні елементи черги помістити в стек. Підрахувати суму парних елементів в стеку.

16 Порахувати середнє арифметичне додатних елементів черги. Від'ємні елементи черги помістити в стек. Підрахувати суму непарних елементів в стеку.

17 Знайти суму ненульових елементів черги. Їх квадрати помістити в стек. Знайти добуток парних елементів стеку.

18 Знайти середнє арифметичне парних елементів черги. Додатні елементи черги помістити в стек. Знайти кількість елементів в стеку.

19 Порахувати суму додатних елементів черги. Непарні елементи черги помістити в стек. Знайти добуток елементів в стеку.

20 Порахувати добуток додатних елементів черги. Непарні елементи черги помістити в стек. Знайти кількість елементів в стеку.

21 Знайти добуток непарних елементів черги. Від'ємні елементи черги помістити в стек. Знайти кількість парних елементів в стеку.

22 Порахувати кількість додатних елементів черги. Від'ємні елементи черги помістити в стек. Знайти добуток парних елементів в стеку.

- 23 Знайти добуток непарних елементів черги. Парні елементи черги помістити в стек. Знайти суму елементів в стеку.
- 24 Знайти суму парних додатних елементів черги. Від'ємні елементи черги помістити в стек. Знайти добуток непарних елементів в стеку.
- 25 Порахувати кількість непарних елементів черги. Від'ємні парні елементи черги помістити в стек. Знайти суму елементів в стеку.
- 26 Знайти добуток ненульових елементів черги. Ці елементи, зменшені на 1, помістити в стек. Знайти суму непарних елементів в стеку.
- 27 Порахувати добуток додатних елементів черги. Від'ємні парні елементи черги помістити в стек. Знайти суму елементів в стеку.
- 28 Знайти середнє арифметичне ненульових елементів черги. Ці елементи, збільшені на 3, помістити в стек. Знайти суму непарних елементів в стеку.
- 29 Порахувати середнє арифметичне від'ємних елементів черги.

КОНТРОЛЬНІ ПИТАННЯ

- 1 Яка структура даних називається чергою?
- 2 Як структура даних «черга» відображається на пам'ять комп'ютера?
- 3 Який елемент називають «головою» черги?
- 4 Який елемент називають «хвостом» черги?
- 5 Як відбувається читання елементів черги? Запишіть алгоритм читання елемента з черги.
- 6 Як відбувається запис елементів у чергу? Запишіть алгоритм запису елемента в чергу.
- 7 Яким чином організовано ефективну обробку елементів черги?
- 8 Зобразіть схематично роботу з елементами черги.
- 9 Запишіть процедуру запису елемента в чергу.
- 10 Запишіть процедуру читання елемента з черги.
- 11 Запишіть процедуру перегляду елементів черги.

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

Методичні рекомендації до виконання лабораторної роботи №№ 5-6

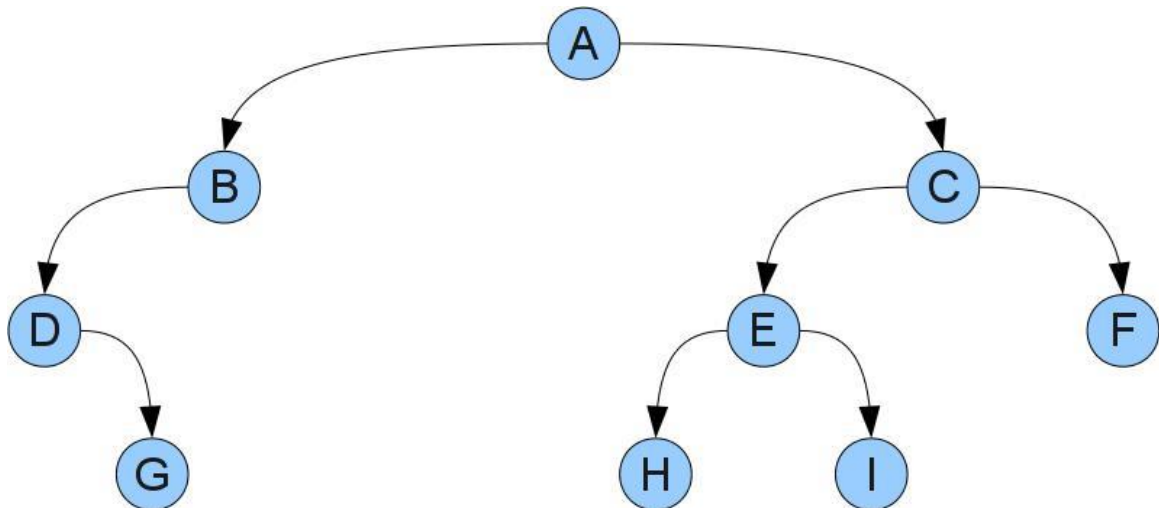
Тема: Складання та налагодження програми рішення задачі створення та обходу бінарних дерев.

Мета: Отримання навиків в організації динамічної структури даних – бінарного дерева.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Бінарне дерево – це структура даних, кожен елемент якої окрім самих даних містить покажчики на два наступних елементи структури. Один з цих наступних елементів умовно називається лівим, а інший - правим.

Конкретніше, у дерева є виділена вершину-корінь й у кожної вершини може бути лівий і правого синів. На словах звучить трохи складно, але якщо глянути на картинку все стає зрозумілим:



У цього дерева коренем буде вершина A. Видно, що у вершини D відсутній лівий син, у вершини B - правий, а у вершин G, H, F й I - обоє. Вершини без синів прийнято називати листами.

Кожній вершині X можна зіставити своє дерево, що складається з вершини, її синів, синів її синів, і т.д. Таке дерево називають піддеревом з коренем X. Лівим і правим піддеревими X називають піддерева з коріннями відповідно в левом і правом

синах X. Помітимо, що такі піддерева можуть виявитися порожніми, якщо в X немає відповідного сина.

Дані в дереві зберігаються в його вершинах. У програмах вершини дерева звичайно представляють структурою, що зберігає дані й два посилання на лівого й правого сина. Відсутні вершини позначають null.

Множина всіх вузлів, рівновіддалених від кореня, називається рівнем. Вузол, з якого не починається жодна вітка, називається кінцевим або листовим вузлом.

Оскільки дерево бінарне, кожен вузол може породжувати два вузли наступного рівня. Породжені вузли є дочірніми по відношенню до вузла, що їх породив. Породжуючий вузол є батьківським по відношенню до своїх дочірніх вузлів. Батьківський вузол разом із своїми дочірніми складає ланку.

Сумарна кількість рівнів дерева називається висотою дерева.

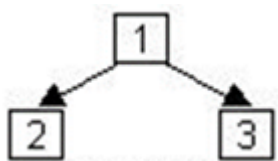
Основними операціями при роботі з деревами є:

- додавання елемента до дерева;
- пошук елемента дерева, що відповідає заданому критерію пошуку;
- сортування елементів дерева;
- вилучення елемента дерева.

Процес доступу до елементів дерева називається проходженням дерева. Існує три способи проходження дерев:

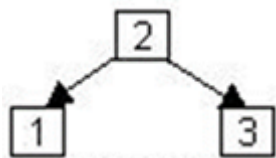
- обхід у прямому порядку,
- обхід у зворотному порядку,
- кінцевий (симетричний) обхід.

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



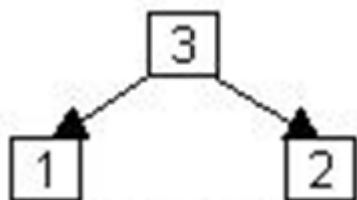
1. потрапити в корінь,
2. пройти в прямому порядку ліве піддерево,
3. пройти в прямому порядку праве піддерево.

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



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

Кінцевий (симетричний) обхід дерева



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

Видалення бінарного дерева.

Слід враховувати, що при видаленні вузла дерева спочатку повинні бути видалені його лівий та правий нащадок, а потім видаляється сам вузол. Отже для реалізації цієї операції слід використовувати кінцевий (симетричний) обхід дерева.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

- 1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Скласти та налагодити програму створення та обробки бінарного дерева на основі нелінійного списку за алгоритмом згідно Вашого варіанту. Програма повинна задовольняти наступним вимогам:

- організувати користувацьке меню, яке повинно містити наступні пункти:
 1. Створення бінарного дерева.
 2. Перегляд вмісту бінарного дерева.
 3. Обробка дерева згідно Вашого варіанту.

- забезпечити коректне введення користувачем вхідних даних;
- при обробці дерева враховувати, що шукані елементи можуть бути відсутні. В цьому випадку вивести користувачеві відповідне повідомлення;
- введення та виведення вхідних та вихідних даних повинно містити необхідні для користувача повідомлення.

2 Розробити та налагодити програму рішення задачі.

3 Оформити звіт з лабораторної роботи.

Звіт повинен містити наступні розділи:

1 Постановка задачі.

2 Текст програми з відповідними коментарями.

3 Копії вікон виконання програми.

Розробити тестові набори вхідних даних, що демонструють всі можливі варіанти роботи програми та помістити в звіт копії вікон виконання програми для кожного тестового набору.

4 Висновок.

Варіанти завдань:

1. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, підрахувати кількість парних чисел в дереві та вивести ці числа на екран.

2. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, від'ємні числа дерева заміни їх квадратами. Вивести оновлене дерево на екран.

3. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, знайти максимальний елемент дерева та вивести його на екран.

4. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, підрахувати кількість непарних чисел в дереві та вивести ці числа на екран.

5. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, знайти максимальний та мінімальний елементи дерева та вивести ці значення на екран.

6. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, знайти найбільше від'ємне число дерева та вивести його на екран.

7. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, додатні числа дерева піднести до третього ступеня. Вивести оновлене дерево на екран.

8. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, підрахувати кількість листових вузлів дерева.

9. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Вивести створене дерево на екран. Використовуючи симетричний обхід дерева, кожному листовому вузлу дерева додати лівого прямого нащадка зі значенням, вказаним користувачем. Вивести оновлене дерево на екран.

10. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, підрахувати кількість вузлів дерева, які мають тільки правого прямого нащадка.

11. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, підрахувати суму парних чисел в дереві.

12. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, підрахувати суму парних чисел в дереві. Значення кореневого вузла дерева замінити на значення отриманої суми.

13. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, знайти максимальний елемент дерева та вивести його на екран. Всі додатні числа дерева заміни максимальним елементом. Вивести оновлене дерево на екран.

14. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, підрахувати добуток непарних чисел

в дереві. Замінити значення кореневого вузла цим добутком. Вивести оновлене дерево на екран.

15. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, знайти суму максимального та мінімального елементів дерева та вивести це значення на екран.

16. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, підрахувати кількість вузлів дерева, в яких правий або лівий прямий потомок є листовим вузлом.

17. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Вивести створене дерево на екран. Кожному листовому вузлу дерева додати правого прямого нащадка зі значенням, вказаним користувачем. Вивести оновлене дерево на екран.

18. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, підрахувати кількість вузлів дерева, які мають тільки лівого прямого нащадка.

19. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, знайти максимальний та мінімальний елементи дерева та вивести ці значення на екран.

20. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, додатні числа дерева помножити на -1 . Вивести оновлене дерево на екран.

21. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, підрахувати кількість непарних чисел в дереві та вивести ці числа на екран.

22. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, знайти мінімальний елемент дерева та вивести його на екран. Всі від'ємні числа дерева заміни мінімальним елементом. Вивести оновлене дерево на екран.

23. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, підрахувати кількість вузлів дерева, які мають як лівого так і правого прямих потомків.

24. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, знайти мінімальний елемент дерева та вивести його на екран. Значення всіх листових вузлів дерева замінити мінімальним елементом. Вивести оновлене дерево на екран.

25. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Вивести на екран значення лівого та правого прямих потомків кореневого вузла дерева.

26. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, значення листових вузлів дерева заміни їх квадратами. Вивести оновлене дерево на екран.

27. Створити бінарне дерево натуральних чисел введенням даних з клавіатури. Використовуючи симетричний обхід дерева, підрахувати суму парних чисел в дереві. Значення кореневого вузла дерева замінити на значення отриманої суми.

28. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи прямий обхід дерева, знайти максимальний елемент дерева. Підрахувати кількість вузлів дерева, значення яких дорівнює максимальному.

29. Створити бінарне дерево дійсних чисел введенням даних з клавіатури. Використовуючи зворотній обхід дерева, значення листових вузлів дерева помножити на -1. Вивести оновлене дерево на екран.

КОНТРОЛЬНІ ПИТАННЯ

- 1 Яку структуру мають вершини двійкового дерева?
- 2 Чому для дерев існує кілька правил обходу вершин?
- 3 Які правила обходу вершин дерева є основними?
- 4 Як виконується обхід дерева в прямому напрямку?
- 5 Як виконується обхід дерева в симетричному напрямку?
- 6 Як виконується обхід дерева у зворотному напрямку?
- 7 Як виконується обхід дерева в назад-симетричному напрямку?
- 8 Чому рекурсивна реалізація правил обходу є найбільш зручної?
- 9 Що відбувається при рекурсивному виконанні обходу дерева?
- 10 Як програмно реалізується обхід дерева в прямому напрямку?
- 11 Як програмно реалізується обхід дерева в симетричному напрямку?

- 12 Як програмно реалізується обхід дерева у зворотному напрямку?
- 13 Який формальний параметр необхідний для рекурсивної реалізації правил обходу і як він використовується?
- 14 Як правильно виконати знищення всієї деревоподібної структури?

Методичні рекомендації до виконання лабораторної роботи №№ 7-8

Тема: Розробка та реалізація програм з використанням методу пошуку з поверненнями.

Мета: Отримання навичок в розробці та реалізації програм на основі алгоритму пошуку (перебору) з поверненнями.

Теоретичні відомості

Алгоритми з поверненням і їх властивості.

Пошук з поверненням (англ. Backtracking) - загальний метод знаходження рішень задачі, в якій потрібно повний перебір усіх можливих варіантів в деякій множині. Як правило дозволяє вирішувати завдання, в яких ставляться питання типу, : "Перерахуйте усі можливі варіанти ", "Скільки існує способів ", чи "Є спосіб ", чи "Існує об'єкт". і т. п.

Термін backtrack був введений в 1950 році американським математиком Дерриком Генрі Лемером. Незначні модифікації методу пошуку з поверненням, пов'язані з представленням даних або особливостями реалізації, мають і інші назви: метод гілок і меж, пошук в глибину, метод проб і помилок і т. д. Пошук з поверненням практично одночасно і незалежно був винайдений багатьма дослідниками ще до його формального опису.

Опис методу

Рішення задачі методом пошуку з поверненням зводиться до послідовного розширення часткового рішення. Якщо на черговому кроці таке розширення провести не вдається, то повертаються до попереднього часткового рішення і продовжують пошук далі. Цей алгоритм дозволяє знайти усі рішення поставленої задачі, якщо вони існують. Для прискорення методу стараються обчислення організувати так, щоб якомога раніше виявляти свідомо невідповідні варіанти.

Використання методу

Класичним прикладом використання алгоритму пошуку з поверненням є завдання про вісім ферзів. Її формулювання таке: "Розставити на стандартній 64-клітинній шахівниці 8 ферзів так, щоб жоден з них не знаходився під боєм іншого". Спершу на дошку ставлять одного ферзя, а потім намагаються поставити кожного наступного ферзя так, щоб його не били вже встановлені ферзі. Якщо на черговому кроці таку установку зробити не можна - повертаються на крок назад і намагаються поставити раніше встановленого ферзя на інше місце.

Окрім цього, метод пошуку з поверненням дозволяє вирішувати безліч інших переборних завдань. Наприклад, за допомогою його можна отримати усі підмножини, розміщення, перестановки, поєднання цієї множини M .

Недоліки

Метод пошуку з поверненням є універсальним. Досить легко проектувати і програмувати алгоритми рішення завдань з використанням цього методу. Проте час знаходження рішення може бути дуже великий навіть при невеликій розмірності завдання (кількості початкових даних), причому настільки велике (може складати роки або навіть віки), що про практичне застосування не може бути і мови. Тому при проектуванні таких алгоритмів, обов'язково треба теоретично оцінювати час їх роботи на конкретних даних. Існують також завдання вибору, для вирішення яких можна побудувати унікальні, "швидкі" алгоритми, що дозволяють швидко отримати рішення навіть при великій розмірності завдання. Метод пошуку з поверненням в таких завданнях застосовувати неефективно.

Властивість

Характерна властивість таких алгоритмів полягає в тому, що в них робляться кроки у напрямі загального рішення, але усі вони фіксуються (записуються) таким чином, що пізніше можна повернутися назад, відкидаючи тим самим кроки, які не ведуть не до загального рішення. Такий процес називається поверненням або відкатом (backtracking).

Варіанти завдань для перебору з поверненнями

Усі завдання, для вирішення яких можна застосувати перебір з поверненнями, можна умовно розділити на декілька великих груп:

- Пошук одного рішення
- Пошук усіх рішень
- Пошук оптимального рішення

Для вирішення деяких завдань вимагається знайти одне рішення, для цього можна просто перебрати усі варіанти і рішенням буде перший відповідний варіант. У завданнях пошуку усіх рішень справа йде трохи інакше: необхідно перебрати усі варіанти і вибрати усі відповідні. Третій клас завдань схожий напредыдущий: нам також потрібно буде знайти усі рішення, і потім вибрати з них оптимальне. Зазвичай вибір оптимального рішення відбувається в процесі самого перебору.

Завдання про лабіринт

Щоб не бути голослівним, розберемо конкретне завдання - завдання про лабіринт. Формулювання.

Є прямокутна матриця $n \times m$, яка задає лабіринт. Нулі в матриці означають прохід, мінус одиниці - стіни. Треба знайти шлях із заданої клітини (вхід) в іншу задану клітину (вихід). Ходи дозволяється робити тільки по горизонталі і вертикалі, і тільки по проходах. Гарантується, що шлях існує.

Реальні завдання, звичайно, зазвичай складніше, ніж просто ходіння по лабіринту. Проте в цілому пошук рішення практично такий же : ми робимо черговий крок в пошуку рішення, фіксуємо його і намагаємося на його основі зробити наступний. Не знайшовши рішення, ми повинні спробувати інший допустимий крок. Якщо усі можливі кроки вже перебрані, а рішення так і не знайдене, слід відступити на крок назад і повторити процедуру пошуку.

Вхідні дані.

Кількість стовпців і кількість рядків N і M ($1 \leq N, M \leq 6$).

Координати старту і координати фінішу.

Матриця лабіринту.

Вихідні дані.

Матриця лабіринту, в якій помічений шлях до виходу.

Порядок виконання роботи

1. Реалізувати програму пошуку шляху проходження по лабіринту.
2. Оформити та захистити звіт з лабораторної роботи.

Звіт з лабораторної роботи повинен вміщати наступні розділи:

1. Тема роботи.
2. Мета роботи.
3. Постановка задачі.
4. Текст програми.
5. Копія вікна виконання програми.
6. Висновки.

Контрольні запитання:

1. У чому проявляється рекурсивність методу перебору з поверненням?
2. Чому повний метод перебору з поверненням гарантує відшукування усіх рішень задачі?
3. Дати визначення методу решета.
4. Який алгоритм називається недетермінованим?

Методичні рекомендації до виконання лабораторної роботи № 9

Тема: Складання та налагодження програм реалізації алгоритмів сортування елементів масиву даних.

Мета: Здобуття навичок роботи зі структурованим типом даних масив та реалізації алгоритмів внутрішнього сортування.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Сортування обмінами

Стандартний обмін («бульбашковий» метод сортування)

Одномірний масив із n цілих чисел відсортуємо за зростанням значень елементів. Алгоритм полягає в тому, що при перегляді вхідного масиву попарно порівнюються сусідні елементи. Якщо порядок їхнього проходження не відповідає заданому критерію впорядкованості, то елементи міняються місцями. В результаті одного такого перегляду, при сортуванні за збільшенням елементів, елемент з найбільшим значенням переміститься на останнє місце в масиві. При наступному проході на своє місце переміститься другий за величиною елемент і т.д. Для постановки на свої місця n елементів слід зробити $n-1$ проходів. При кожному черговому проході кількість елементів, що порівнюються, треба зменшувати на одиницю.

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

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

Лістинг 1.

```
. . .  
//Цикл проходів по масиву до  $i$ -того елементу (до границі перегляду)  
for (i=n-1; i>1; i--)  
{ // цикл попарного порівняння сусідніх елементів  
  for (j =0; j<=i-1; j++)  
    if (b[j]>b[j+1])
```

```

        { // обмін елементів з номерами j та j+1
          int a = b[j];
          b[j] = b[j+1];
          b[j+1] = a;
        }
      }
    }
  }

```

Способи модифікації сортування стандартним обміном

1) *Дотримання умови Айверсона.*

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

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

Лістинг 2.

```

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

        for (j =0; j<=i-1; j++)
            if (b[j]>b[j+1])
            {
                int a = b[j];
                b[j] = b[j+1];
                b[j+1] = a;
                k=1; // обмін відбувся
            }
        i--; // зменшення нижньої границі перегляду
    }
    . . .

```

2) *Шейкерне сортування*

Очевидний прийом поліпшення алгоритму стандартного обміну - запам'ятовувати, були або не були перестановки в процесі чергового проходу. Якщо в останньому

проході перестановок не було, то сортування можна закінчувати. Це поліпшення можна знову ж поліпшити, якщо запам'ятовувати не тільки сам факт, що обмін мав місце, але й індекс останнього обміну. Ясно, що всі пари сусідніх елементів нижче цього індексу вже впорядковані. Тому перегляди можна закінчувати на цьому індексі, а не йти до заздалегідь певної нижньої границі.

Фрагмент програми шейкерного сортування за зростанням представлено в лістингу 3

Лістинг 3.

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

. . .

Сортування простими вставками

Одномірний масив $b[]$ із n цілих чисел відсортуємо за зростанням значень елементів методом вибору. Ідея алгоритму: для кожного елемента масиву $b[i]$ вважаємо, що попередні елементи вже відсортовані. Елемент $b[i]$ потрібно вставити на відповідне місце в вже відсортовану послідовність $b[0] \dots b[i-1]$. Це місце визначається послідовним порівнянням $b[i]$ з вже впорядкованими елементами і, якщо необхідно - елементи міняються місцями.

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

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

Лістинг 4.

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

Сортування методом вибору.

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

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

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

Лістинг 1.6

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

```

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

```

Сортування методом Шелла

Сортування Шелла була названа на честь її винахідника - Дональда Шелла, що опублікував цей алгоритм в 1959 році. Загальна ідея сортування Шелла складається в порівнянні на початкових стадіях сортування пари значень, що розташовані досить далеко друг від друга в наборі даних, який необхідно впорядкувати. Така модифікація методу сортування дозволяє швидко переставляти далекі неупорядковані пари значень (сортування таких пар звичайно вимагає великої кількості перестановок, якщо використасться порівняння тільки сусідніх елементів).

Загальна схема методу полягає в наступному.

Крок 1. Відбувається впорядкування елементів $n/2$ пар $(x_i, x_{n/2+i})$ для $1 < i < n/2$.

Крок 2. Упорядковуються елементи в $n/4$ групах із чотирьох елементів $(x_i, x_{n/4+i}, x_{n/2+i}, x_{3n/4+i})$ для $1 < i < n/4$.

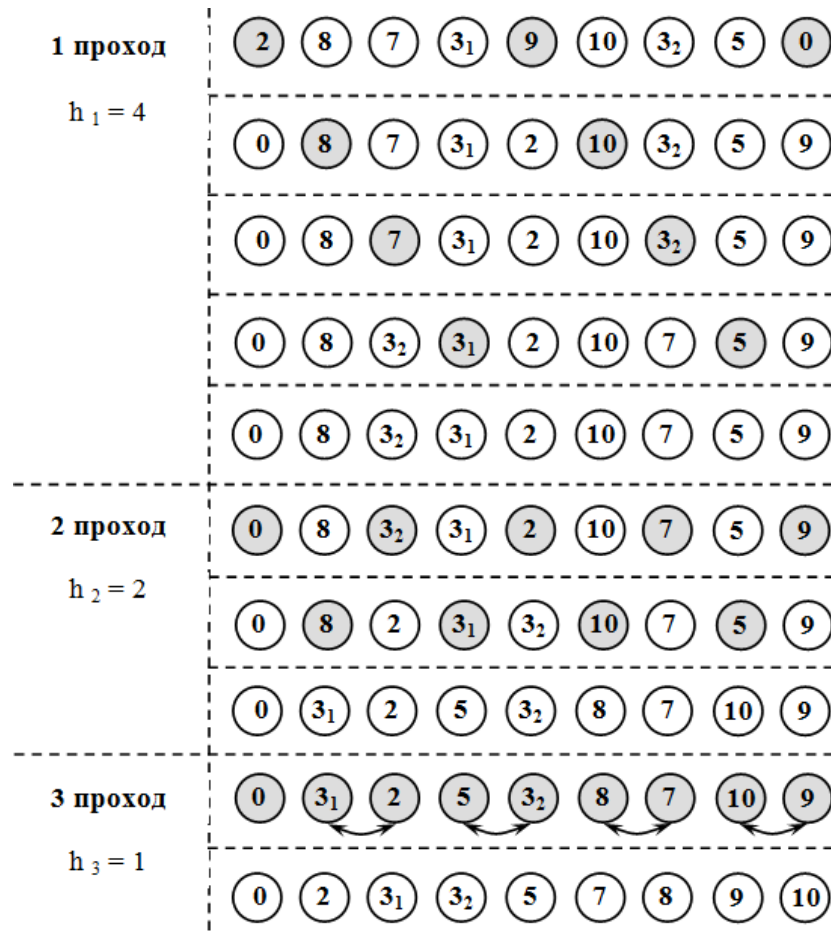
Крок 3. Упорядковуються елементи вже в $n/4$ групах з восьми елементів і т.д.

На останньому кроці впорядковуються елементи відразу у всьому масиві

$x_1, x_2, \dots, x_n$.

На кожному кроці для впорядкування елементів у групах використається метод сортування вставками.

Демонстрація сортування по неспаданню методом Шелла:



Опис функції сортування методом Шелла

```
void Shell_Sort (int n, int *x){
    int h, i, j;
    for (h = n/2 ; h > 0 ; h = h/2)
        for (i = 0 ; i < n-h ; i++)
            for (j = i ; j >= 0 ; j = j - h)
                if (x[j] > x[j+h])
                    Exchange (j, j+h, x);
                else j = 0;
    }
    //процедура обміну двох елементів
    void Exchange (int i, int j, int *x){
        int tmp;
        tmp = x[i];
        x[i] = x[j];
        x[j] = tmp;
    }
}
```

Швидке сортування Хоара

Метод швидкого сортування був уперше описаний Ч.А.Р. Хоаром в 1962 році. Швидке сортування - це загальна назва ряду алгоритмів, які відбивають різні підходи до одержання критичного параметра, що впливає на продуктивність методу.

При загальному розгляді алгоритму швидкого сортування, відзначимо, що цей метод ґрунтується на послідовному поділі набору даних на блоки меншого розміру таким чином, що між значеннями різних блоків забезпечується відношення впорядкованості (для будь-якої пари блоків всі значення одного із цих блоків не перевищують значень іншого блоку).

Опорним (ведучим) елементом називається деякий елемент масиву, що вибирається певний образом. З погляду коректності алгоритму вибір опорного елемента байдужний. З погляду підвищення ефективності алгоритму вибиратися повинна медіана, але без додаткових відомостей про дані, що сортуються, її звичайно неможливо одержати. Необхідно вибирати постійно той самий елемент (наприклад, середній або останній по положенню) або вибирати елемент із випадково обраним індексом.

Алгоритм швидкого сортування Хоара

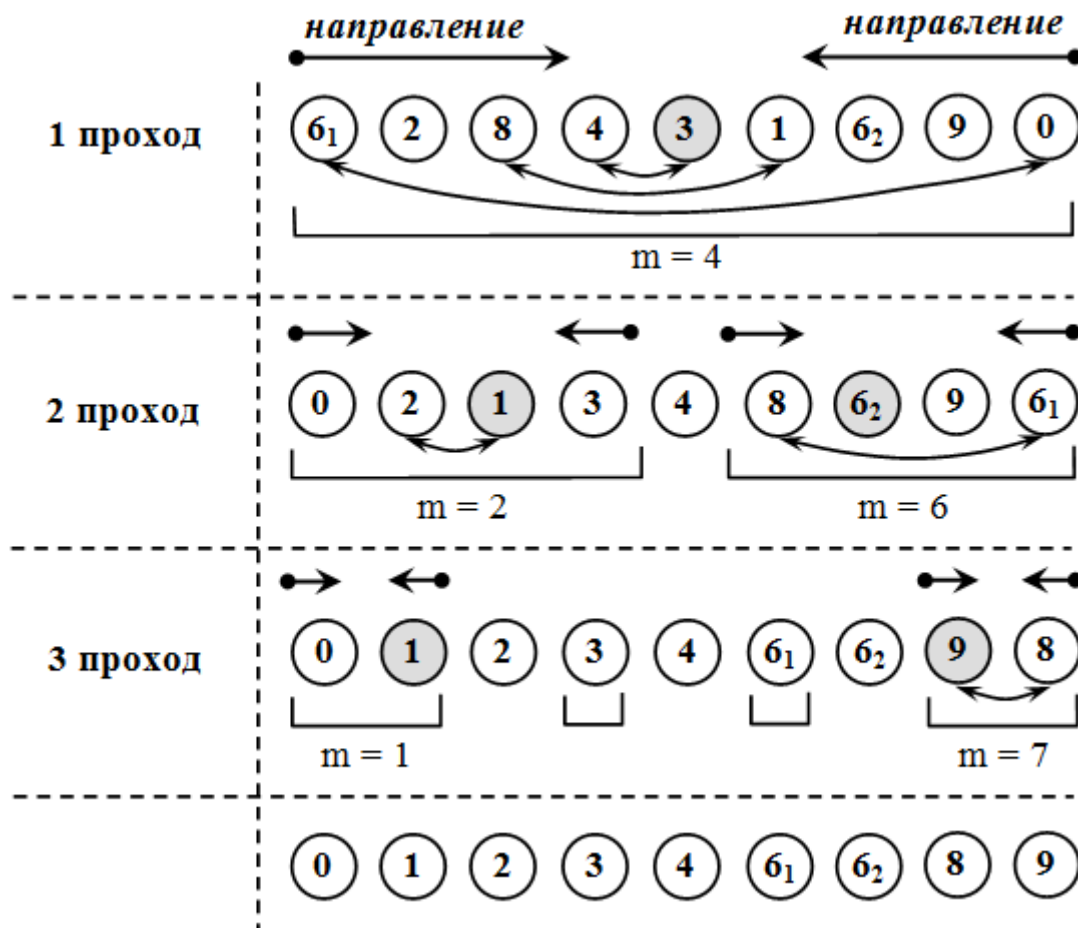
Нехай даний масив $x[n]$ розмірності n .

Крок 1. Вибирається опорний елемент масиву.

Крок 2. Масив розбивається на два - лівий і правий - щодо опорного елемента. Реорганізуємо масив таким чином, щоб всі елементи, менші опорного елемента, виявилися ліворуч від нього, а всі елементи, більші опорного - праворуч від нього.

Крок 3. Далі повторюється крок 2 для кожного із двох знову утворених масивів. Щораз при повторенні перетворення чергова частина масиву розбивається на два менших і т.д. , поки не вийде масив із двох елементів.

Демонстрація швидкого сортування Хоара по неспаданню. Виберемо в якості опорний елемент, розташована на середній позиції.



Швидке сортування стало популярної насамперед тому, що її неважко реалізувати, вона добре працює на різних видах вхідних даних й у багатьох випадках вимагають менше витрат ресурсів у порівнянні з іншими методами сортування.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Скласти та налагодити програму створення та сортування матриці за алгоритмом згідно Вашого варіанту. Програма повинна задовольняти наступним вимогам:

— організувати користувацьке меню, яке повинно містити наступні пункти:

1. Створення матриці.
2. Виведення матриці.
3. Обробка матриці згідно Вашого варіанту.

- забезпечити коректне введення користувачем вхідних даних. Тип матриці – динамічна або псевдодинамічна – обрати самостійно;
 - введення та виведення вхідних та вихідних даних повинно містити необхідні для користувача повідомлення.
- 2 Розробити та налагодити програму рішення задачі.
 - 3 Оформити звіт з лабораторної роботи.

Звіт повинен вміщати наступні розділи:

1. Постановка задачі.
2. Текст функції реалізації алгоритму сортування згідно Вашого варіанту.
3. Текст програми мовою реалізації алгоритму.
4. Копію вікна виконання програми на тестових вхідних даних.
5. Висновки.

Варіанти задач

1. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й та r -й рядки масиву використовуючи алгоритм сортування обмінами з дотриманням умови Айверсона (k та r вводяться користувачем).
2. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й та r -й рядки масиву використовуючи алгоритм сортування вибором (k та r вводяться користувачем).
3. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й та r -й рядки масиву використовуючи алгоритм сортування вставками (k та r вводяться користувачем).
4. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й та r -й стовпці масиву використовуючи алгоритм шейкерного сортування (k та r вводяться користувачем).

5. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й та r -й стовпці масиву використовуючи алгоритм сортування Хоара (k та l вводяться користувачем).

6. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням k -й та r -й рядки масиву використовуючи алгоритм сортування Шелла (k та l вводяться користувачем).

7. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем) Впорядкувати за зростанням k -й та r -й стовпці масиву використовуючи алгоритм сортування вставками (k та r вводяться користувачем).

8. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням k -й та r -й рядки масиву використовуючи алгоритм сортування обмінами з дотриманням умови Айверсона (k та r вводяться користувачем).

9. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням k -й та r -й рядки масиву використовуючи алгоритм шейкерного сортування (k та r вводяться користувачем).

10. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням k -й та r -й рядки масиву використовуючи алгоритм сортування вибором (k та r вводяться користувачем).

11. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням k -й та r -й стовпці масиву використовуючи алгоритм сортування Шелла (k та r вводяться користувачем).

12. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням k -й та r -

й стовпці масиву використовуючи алгоритм сортування Хоара (k та p вводяться користувачем).

13. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням всі стовпці масиву використовуючи алгоритм сортування обмінами з дотриманням умови Айверсона.

14. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й рядок масиву та за спаданням p -й рядок масиву використовуючи алгоритм сортування вибором (k та p вводяться користувачем).

15. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й рядок масиву та за спаданням p -й рядок масиву використовуючи алгоритм сортування вставками (k та p вводяться користувачем).

16. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й стовбець масиву та за спаданням p -й стовбець масиву використовуючи алгоритм сортування обмінами (k та p вводяться користувачем).

17. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й стовбець масиву та за спаданням p -й стовбець масиву використовуючи алгоритм сортування Хоара (k та p вводяться користувачем).

18. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням k -й стовбець масиву та за спаданням p -й стовбець масиву використовуючи алгоритм сортування вставками (k та p вводяться користувачем).

19. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням всі рядки масиву використовуючи алгоритм сортування обмінами.

20. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням всі рядки масиву використовуючи алгоритм сортування вибором.

21. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням всі рядки масиву використовуючи алгоритм шейкерного сортування.

22. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням всі стовпці масиву використовуючи алгоритм сортування Хоара.

23. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням всі стовпці масиву використовуючи алгоритм сортування Шелла.

24. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за зростанням всі стовпці масиву використовуючи алгоритм сортування вставками.

25. Дано двовимірний масив дійсних чисел $a[1..n, 1..m]$ (кількість рядків n та кількість стовпців m вводиться користувачем). Впорядкувати за спаданням всі рядки масиву використовуючи алгоритм сортування обмінами з дотриманням умови Айверсона.

КОНТРОЛЬНІ ПИТАННЯ

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

Методичні рекомендації до виконання лабораторної роботи № 10

Тема: Складання та налагодження програм пошуку в лінійних масивах даних.

Мета: Здобуття навичок реалізації алгоритмів пошуку.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Алгоритми пошуку є основою для ефективного і швидкого знаходження інформації в Інтернеті. Вони використовуються пошуковими системами, такими як Google, для індексації мільярдів веб-сторінок та надання користувачам найбільш релевантних результатів пошуку. У цій статті ми розглянемо основні види алгоритмів пошуку, їх роль у веб-пошуку та вплив на SEO.

Що таке алгоритми пошуку?

Алгоритми пошуку – це комплексні математичні моделі та логічні інструкції, які використовуються для пошуку та ранжування веб-сторінок. Вони базуються на різних факторах, таких як ключові слова, релевантність, авторитет веб-сторінки та інші параметри. Алгоритми пошуку постійно оновлюються та удосконалюються для забезпечення кращих результатів пошуку для користувачів.

Історія розвитку алгоритмів пошуку

Історія алгоритмів пошуку сягає своїми коріннями до початку розвитку Інтернету. У 1990-х роках було створено перші пошукові системи, такі як Yahoo! та AltaVista, які використовували прості алгоритми для індексації та пошуку веб-сторінок. Однак, з появою Google у 1998 році, пошукова індустрія зазнала значних змін.

Основні види алгоритмів пошуку

Лінійний пошук

Лінійний пошук – це найпростіший вид алгоритму пошуку, в якому кожен елемент послідовно перевіряється на наявність шуканого значення. Цей метод не є ефективним для великих обсягів даних, оскільки час пошуку залежить від кількості елементів.

Бінарний пошук

Бінарний пошук – це швидкий алгоритм пошуку, який працює з відсортованим списком елементів. Він використовує стратегію “половинного поділу” для знаходження шуканого значення. Бінарний пошук ефективний для великих обсягів даних, оскільки час пошуку зменшується логарифмічно залежно від кількості елементів.

Пошук за зразком

Пошук за зразком – це алгоритм, який шукає входження певного зразка в текстовий рядок. Він може бути використаний для пошуку певних слів або фраз у веб-сторінках.

Пошук на графах

Пошук на графах – це алгоритм, який використовується для пошуку шляху між двома вузлами у графі. Цей вид алгоритму широко використовується у сферах, таких як навігація та маршрутизація.

Індексація та пошук у текстах

Алгоритми індексації та пошуку у текстах використовуються для аналізу та індексації великих обсягів текстової інформації. Вони використовуються в пошукових системах для забезпечення швидкого та точного пошуку текстових документів.

Ключові фактори для ефективного пошуку

Релевантність результатів

Одним з ключових факторів ефективного пошуку є релевантність результатів. Алгоритми пошуку намагаються знайти найбільш відповідні веб-сторінки до запиту користувача, враховуючи різні фактори, такі як наявність ключових слів та контекст запиту.

Швидкість пошуку

Швидкість пошуку є важливим аспектом для задоволення потреб користувачів. Алгоритми пошуку оптимізуються для швидкого пошуку та відображення результатів у максимально короткий час.

Масштабованість

Алгоритми пошуку повинні бути масштабовані, щоб обробляти великі обсяги даних. З постійним зростанням кількості веб-сторінок та запитів користувачів, алгоритми повинні бути готові до обробки великих навантажень.

Адаптація до змінних умов

Алгоритми пошуку повинні бути гнучкими та адаптивними до змін умов. З постійним розвитком Інтернету та зміною популярних технологій, алгоритми повинні оновлюватись для забезпечення актуальності та ефективності результатів пошуку.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

На основі виконання Лабораторної роботи №№ 9-10 скласти та налагодити функцію реалізації алгоритму пошуку в відсортованих рядках або стовпцях матриці згідно Вашого варіанту. Додати відповідний пункт меню в програму попередньої лабораторної роботи та «підключити» до нього реалізовану функцію пошуку.

- 2 Розробити та налагодити програму рішення задачі.
- 3 Оформити звіт з лабораторної роботи.

Звіт повинен вміщати наступні розділи:

1. Постановка задачі.
2. Текст функції (з відповідними коментарями) реалізації алгоритму пошуку згідно Вашого варіанту.
3. Копію вікон виконання програми на тестових вхідних даних.
4. Висновки.

ВАРІАНТИ ЗАВДАНЬ

1. Для відсортованих рядків матриці реалізувати алгоритм бінарного пошуку введеного користувачем числа.
2. Для відсортованих рядків матриці реалізувати Фібоначчієв пошуку введеного користувачем числа..
3. Для відсортованих рядків матриці реалізувати алгоритм інтерполяційного пошуку введеного користувачем числа.
4. Для відсортованих стовпців матриці реалізувати алгоритм бінарного пошуку введеного користувачем числа.
5. Для відсортованих стовпців матриці реалізувати Фібоначчієв пошуку введеного користувачем числа.
6. Для відсортованих стовпців матриці реалізувати алгоритм інтерполяційного пошуку введеного користувачем числа.
7. Для відсортованих рядків матриці реалізувати алгоритм бінарного пошуку введеного користувачем числа.
8. Для відсортованих рядків матриці реалізувати Фібоначчієв пошуку введеного користувачем числа.
9. Для відсортованих рядків матриці реалізувати алгоритм інтерполяційного пошуку введеного користувачем числа.
10. Для відсортованих стовпців матриці реалізувати алгоритм бінарного пошуку введеного користувачем числа.
11. Для відсортованих стовпців матриці реалізувати Фібоначчієв пошуку введеного користувачем числа..
12. Для відсортованих стовпців матриці реалізувати алгоритм інтерполяційного пошуку введеного користувачем числа.

- [illegible]

КОНТРОЛЬНІ ПИТАННЯ

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

Методичні рекомендації до виконання лабораторної роботи №№ 11-12

Тема: Складання та налагодження програм побудови збалансованого бінарного дерева пошуку та реалізація пошуку ключового елементу.

Мета: Отримання навиків в організації динамічної структури даних – збалансованого бінарного дерева пошуку реалізації пошуку ключового елементу.

ТЕОРЕТИЧНІ ВІДОМОСТІ

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

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

Будемо представляти вузли АВЛ-дерева наступною структурою:

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

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

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

Визначимо три допоміжні функції, пов'язані з висотою.

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

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

Друга обчислює $balancefactor$ заданого вузла (і працює тільки з ненульовими покажчиками):

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

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

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

2 Балансування вузлів

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

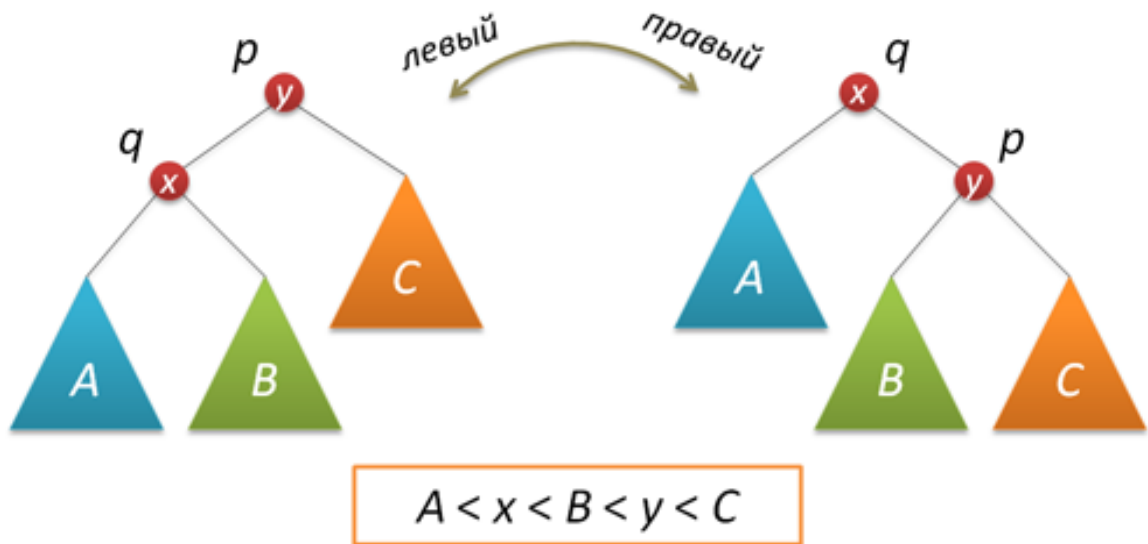


Рис. 2. Правий та лівий прості повороти.

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

Лістинг 1. Реалізація правого простого повороту.

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

Лівий поворот є симетричною копією правого (лістинг 2):

Лістинг 2. Реалізація лівого простого повороту.

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

Розглянемо тепер ситуацію дисбалансу, коли висота правого піддерева вузла p на 2 більше висоти лівого піддерева (зворотний випадок є симетричним і реалізується аналогічно). Нехай q - правий дочірній вузол вузла p , а s - лівий дочірній вузол вузла q (рисунок 3).

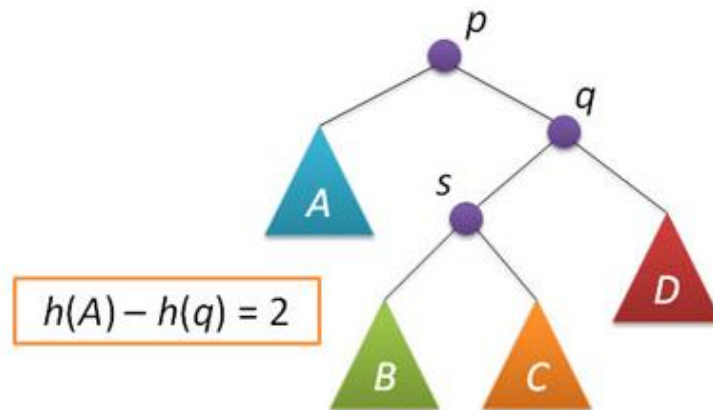


Рис. 3. Розбалансоване бінарне дерево пошуку.

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

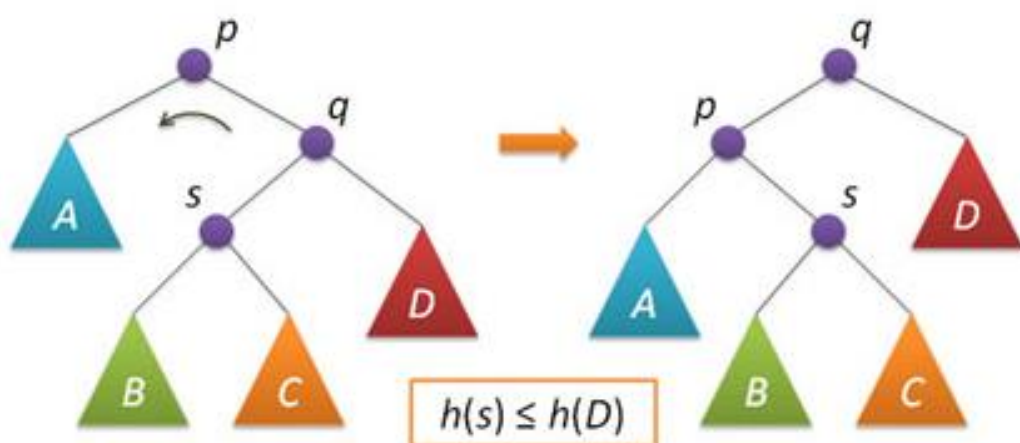


Рис. 4. Простий лівий поворот навколо вузла P .

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

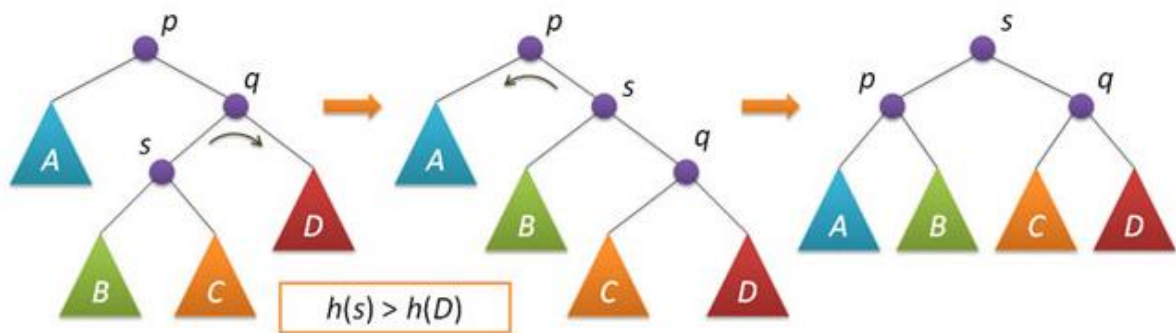


Рис. 5. Великий поворот навколо Р.

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

Лістинг 3. Функція балансування вузлу бінарного дерева пошуку.

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

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

3 Вставка ключів

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

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

Лістинг 4. Функція вставки нового вузла.

```
node* insert(node* p, int k) // вставка ключа k в дерево с корнем p
{
    if( p==NULL )
    {
        p=new node;
        p->info=k;
        p->h=0;
        p->left=NULL;
        p->right=NULL;
    }
    else
    {
        if( k<p->info )
            p->left = insert(p->left,k);
        else
            p->right = insert(p->right,k);
        return balance(p);
    }
}
```

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Скласти та налагодити програму створення збалансованого бінарного дерева на основі нелінійного списку та пошуку ключового елементу. Програма повинна задовольняти наступним вимогам:

- організувати користувацьке меню, яке повинно містити наступні пункти:
 1. Створення збалансованого бінарного дерева.
 2. Перегляд вмісту бінарного дерева.
 3. Пошук ключового елементу.
 4. Видалення дерева.
- забезпечити коректне введення користувачем вхідних даних;
- при обробці дерева враховувати, що шукані елементи можуть бути відсутні. В цьому випадку вивести користувачеві відповідне повідомлення;
- введення та виведення вхідних та вихідних даних повинно містити необхідні для користувача повідомлення.

2 Розробити та налагодити програму рішення задачі.

3 Оформити звіт з лабораторної роботи.

Звіт повинен містити наступні розділи:

- 1 Постановка задачі.
- 2 Текст програми з відповідними коментарями.
- 3 Копії вікон виконання програми.

Цей розділ повинен вміщати наступні скріншоти:

- введення невпорядкованої вхідної послідовності даних;
 - побудоване збалансоване бінарне дерево пошуку;
 - результати пошуку ключового (заданого користувачем) елементу.
- 4 Висновок.

КОНТРОЛЬНІ ПИТАННЯ

- 1 Дайте визначення АВЛ-дерева.
- 2 Чи є АВЛ-дерево деревом пошуку?
- 3 Яка висота АВЛ-дерева?
- 4 Для чого потрібні повороти АВЛ-дерева?
- 5 Яка трудомісткість включення й виключення вершини АВЛ-дерева?
- 6 Яке дерево називається деревом пошуку?
- 7 У чому складається практична важливість використання дерев пошуку?
- 8 Які переваги має використання дерев пошуку для зберігання впорядкованих даних у порівнянні з масивами й списками?
- 9 Приведіть алгоритм пошуку в дереві пошуку.
- 10 Як програмно реалізується пошук у дереві пошуку?

Методичні рекомендації до виконання лабораторної роботи № № 13-14

Тема: Складання та налагодження програми обходу неорієнтованого графу.

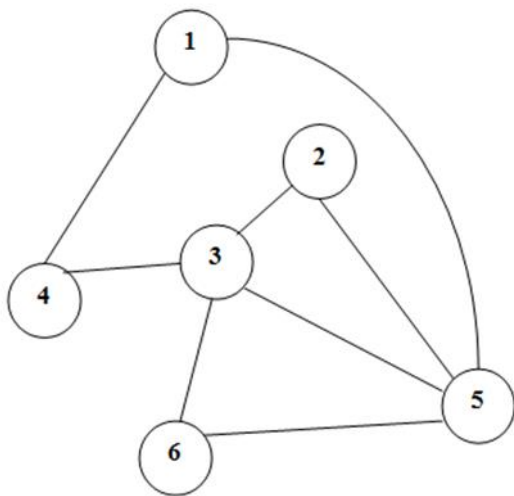
Мета: Отримання навичок в представленні в пам'яті комп'ютера та в реалізації основних процедур обробки незваженого неорієнтованого графа. Реалізація алгоритму обходу (пошуку) графа в ширину та в глибину.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Алгоритм обходу графу в ширину

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

Розглянемо дію алгоритму на прикладі графа G:



1. Вибираємо стартову вершину обходу графа (нехай це буде вершина 1) і привласнюємо їй мітку 0. Всі інші вершини вважаються непоміченими.

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

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

Таким чином, вершини 4 й 5 одержали мітку 1.

3. Розглянемо по черзі оточення вершин з міткою 1 і кожній непоміченій вершині із цього оточення привласнимо мітку 2. У такий спосіб вершини 3, 2 й 6 одержать мітки 2.

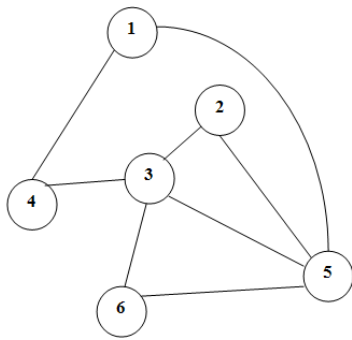
4. Далі, розглядаючи оточення вершин 3, 2, і 6, алгоритм не знайде ні однієї непоміченої вершини. Це означає, що пошук завширшки закінчений.

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

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

Реалізація алгоритму обходу (пошуку) графа завширишки

Для зображеного на малюнку неорієнтованого графа:



Будуємо матрицю сумісності::

MG:

	1	2	3	4	5	6
1	0	0	0	1	1	0
2	0	0	1	0	1	0
3	0	1	0	1	1	1
4	1	0	1	0	0	0
5	1	1	1	0	0	1
6	0	0	1	0	1	0

Будемо використовувати одномірний масив міток **MET**, довжина якого дорівнює кількості вершин у графі:

	1	2	3	4	5	6
MET:						

I-й елемент масиву буде дорівнює величині мітки, якою у процесі роботи програми буде позначатися I-а вершина графа.

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

	1	2	3	4	5	6
OCH:						

Для фіксації "голови" черги будемо використати змінну **first**, а для фіксації "хвоста" черги будемо використати змінну **last**.

Будемо використати змінну **M**, значення якої буде дорівнює величині мітки.

Опис алгоритму:

1. Уведенням із клавіатури заповнюємо матрицю суміжності **MG**.
2. Масив міток **MET** заповнюємо -1 (ознака того, що вершини непомічені).
3. Запитуємо користувача номер стартової вершини.
4. **M:=0**
5. У відповідний стартовій вершині елемент масиву **MET** заносимо **M**.
6. Номер стартової вершини заносимо в **OCH[1]** і встановлюємо **first** та **last** на **OCH[1]**

Нехай стартової буде вершина 2. Тоді:

	1	2	3	4	5	6
MET:	-1	0	-1	-1	-1	-1

M=0

	1	2	3	4	5	6
OCH:	2					

first **last**

1. Організуємо **цикл1** (поки черга не порожня, тобто поки **first** <= **last**)
 - 7.1. Збільшуємо **M** на 1 $M:=M+1$
 - 7.2. Організуємо **цикл2** руху по черзі (поки не переглянемо всі елементи, що містяться в черзі)
 - 7.2.1. Ідемо на рядок матриці суміжності, номер якої відповідає вершині, що перебуває в "голові" черги", тобто в **OCH[first]**. Якщо деякий елемент **MG[first][j]** дорівнює 1 а **MET[j]=-1**(вершина непомічена), заносимо **j** у чергу а елементу **MET[j]** привласнюємо значення **M**, тобто **MET[j]:=M**.
 - 7.2.2. Закінчивши перегляд рядка матриці суміжності, в масиві **OCH** пересуваємо **first** на наступний елемент масиву, тобто **first:=first+1**.
- Кінець цикл2** (коли **first** стане більше **last**)
- 7.3. Переміщаємо **last** на останній, занесений у чергу елемент (якщо такі були).

Кінець цикл1

В нашому прикладі після першого проходження **цикл1**

	1	2	3	4	5	6
MET:	-1	0	1	-1	1	-1

M=1

	1	2	3	4	5	6
OCH:	2	3	5			

first ↗
last ↖

Після другого проходження **цикл1**

	1	2	3	4	5	6
MET:	2	0	1	2	1	2

M=2

	1	2	3	4	5	6
OCH:	2	3	5	4	6	1

first ↗
last ↖

Після третього проходження **цикл1**

	1	2	3	4	5	6
MET:	2	0	1	2	1	2

M=3

	1	2	3	4	5	6
OCH:	2	3	5	4	6	1

first ↗
last ↖

Жодна вершина не була занесена в чергу, а значить всі вершини позначені - вихід із **цикл1**.

Ми обійшли всі вершини графа. При цьому масив **МЕТ** містить найкоротші відстані (кількість пройдених ребер) від стартової вершини до всіх вершин графа.

Обхід графа у глибину

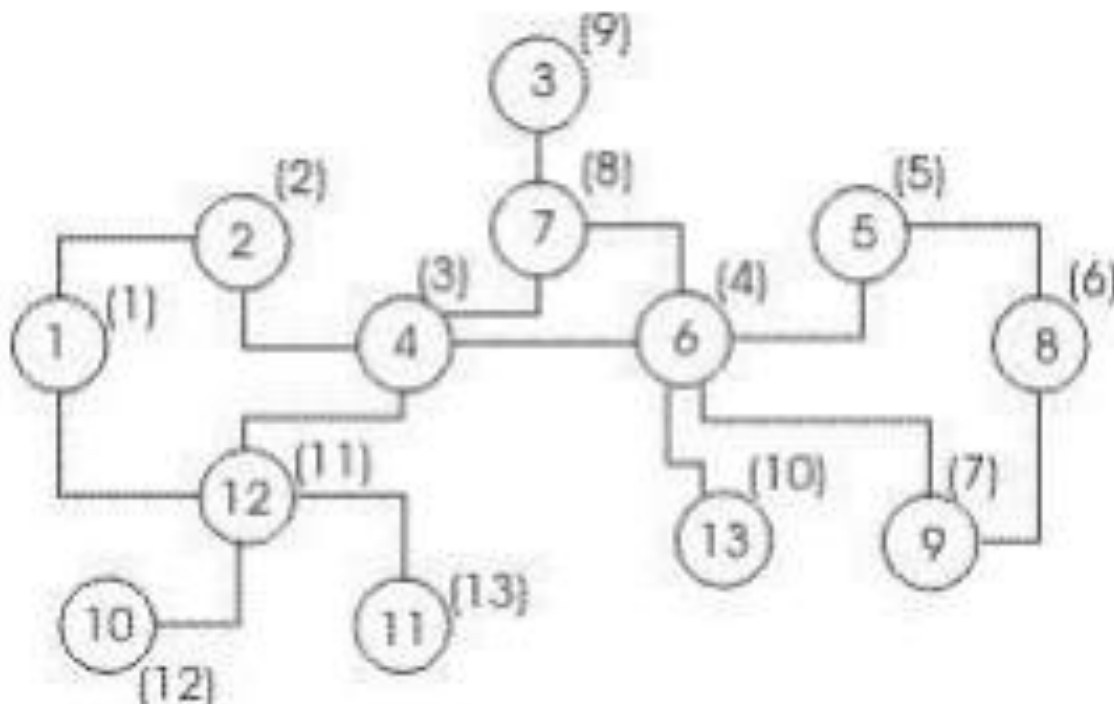
Обхід у глибину - це обхід графа за наступними правилами:

- 1) Перебуваючи у вершині x , потрібно рухатися в будь-яку іншу вершину, яка ще не відвідувалась (якщо така знайдеться), одночасно запам'ятовуючи ребро (дугу), по якій уперше потрапили в дану вершину;
- 2) Якщо з вершини x немає можливості потрапити в вершину, що не відвідувалась, або такої вершини немає, то повертаємося у вершину z , з якої вперше потрапили в x , і продовжуємо обхід у глибину з вершини z .

При виконанні обходу графа за цими правилами ми прагнемо проникнути "углиб" графа так далеко, як це можливо, потім відступаємо на крок назад і знову прагнемо пройти вперед і так далі.

Таким чином, допустимо, що ми перебуваємо у вершині графа v . Далі, вибираємо довільну, але ще не переглянуту вершину u , суміжну з вершиною v . Цю нову вершину розглядаємо тепер уже як v й, починаючи з неї, продовжуємо обхід. Якщо не існує ні однієї нової (ще не переглянутої) вершини, суміжної з v , то говорять, що вершина v переглянута й повертаються у вершину, з якої ми потрапили в v . Пошук триває доти, поки $v=v_0$.

Приклад. Нехай, наприклад, $v_0=1$ для графа, зображеного на малюнку.



Як вершини v послідовно виступають вершини **2,4,6,5,8,9** (у дужках зазначений також порядок відвідування вершин при пошуку в глибину).

Вершина 9 не має суміжних з нею не переглянутих вершин. Тому з вершини 9 повертаємося у вершину 6, минаючи вже використані вершини 8,5. Черговий етап починаємо з вершини 6, переходячи послідовно в вершини 7,3,13. На наступному етапі в якості v виступає вершина 4 і з неї здійснюється послідовний перехід у вершини 10,11 графу.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

- 1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

3. Розробити структуру (малюнок) неорієнтованого незваженого графу як тестовий приклад для програми.
4. Розробити та налагодити програму обходу неорієнтованого незваженого графу згідно вашого варіанту.
5. Оформити та захистити звіт з лабораторної роботи.

Звіт з лабораторної роботи повинен вміщати наступні розділи:

1. Постановка задачі.
2. Опис структури (малюнок) неорієнтованого незваженого графу – тестовий приклад.
3. Текст програми з коментарями.
4. Копія вікна виконання програми на тестових вхідних даних.
5. Висновки.

Варіанти завдань

1. Студенти з непарним номером варіанту реалізують алгоритм обходу графу в ширину.
1. Студенти з парним номером варіанту реалізують алгоритм обходу графу в глибину.

КОНТРОЛЬНІ ПИТАННЯ

- 1 Як зв'язані між собою різні способи подання графів?
- 2 Як від виду або подання графа залежить часова складність алгоритмів пошуку в глибину й завширшки ?

- 3 Як при реалізації в коді виконується повернення з тупикових вершин при обході графа?
- 4 Як виконується обхід у незв'язному графі?
- 5 Чи поширюються поняття "пошук завширшки" на незв'язний граф? Відповідь обґрунтуйте.
- 6 Як в алгоритмах обходу графа завширшки фіксується, що переглянута вершина не нова?
- 7 Як називається алгоритм обходу графа, що використовує для своєї роботи черга?
- 8 Що зберігається в черзі алгоритму обходу графа, що використовує для своєї роботи черга?
- 9 Поняття обходу графу.
- 10 Суть алгоритму обходу в глибину.
- 11 Рекурсивний алгоритм обходу в глибину.
- 12 Ітеративний алгоритм обходу в глибину (з використанням стеку).

Методичні рекомендації до виконання лабораторної роботи № 15

Тема: Складання та налагодження програми реалізації алгоритму Дейкстри пошуку найкоротших шляхів..

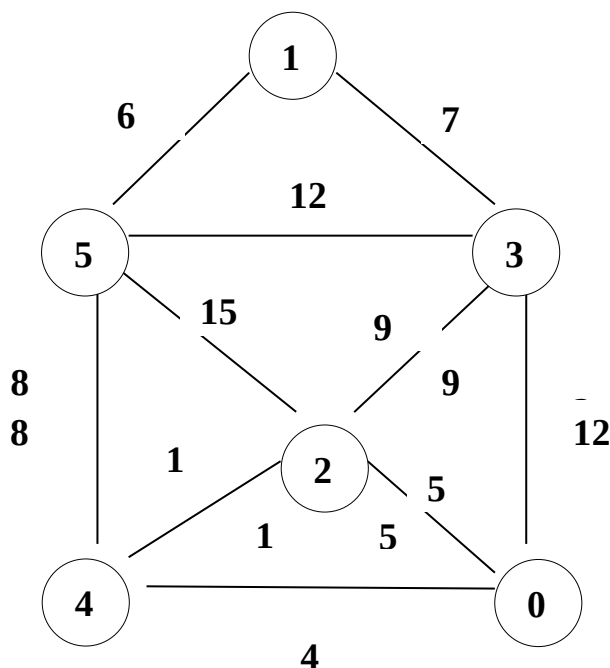
Мета: Отримання навичок в представленні в пам'яті комп'ютера та в реалізації основних процедур обробки зваженого неорієнтованого графа. Реалізація алгоритму Дейкстри пошуку найкоротших шляхів.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Завдання на побудову найкоротших шляхів у графі часто використовуються в побутових ситуаціях. Відомо, що на карті розташовані міста, причому деякі з них з'єднані дорогами. Водій їде з міста А до міста В. Потрібно визначити найкоротший маршрут поїздки.

Для вирішення цього завдання необхідно запровадити поняття зваженого графа. Граф називається зваженим, якщо кожному його ребру надано деяке число, зване довжиною ребра або вагою ребра.

Розглянемо граф G:



Найкоротшим шляхом між вершинами назвемо шлях мінімальної ваги.

Подання зваженого графа в пам'яті комп'ютера таке ж, як і незваженого - за допомогою матриці суміжності, в якій на перетині i -того рядка і j -го стовпця міститься вага ребра (i,j) . Матриця суміжності MG для нашого графа матиме вигляд:

	0	1	2	3	4	5
0	0	0	5	2	4	0
1	0	0	0	7	0	6
2	5	0	0	9	1	15
3	2	7	9	0	0	12
4	4	0	1	0	0	8
5	0	6	12	12	8	0

Алгоритм Дейкстри пошуку оптимального шляху у зваженому графі заснований на алгоритмі обходу графа завширшки. Розглянемо дію цього алгоритму на нашому графі G .

Для даного алгоритму необхідне використання двох одновимірних масивів – $Label$ та $Active$, розмір яких дорівнює кількості вершин у графі. У кожному осередку $Label[i]$ будемо зберігати поточну відстань від стартової вершини до i -тої вершини графу, а в масиві $Active$ будемо позначати 0 не пройдені вершини і 1 - пройдені.

Опис алгоритму:

1. У масиві $Active$ відзначимо всі вершини як пройдені, тобто. заповнимо масив $Active$ нулями. Масив $Label$ заповнимо неймовірно великими значеннями.

2. Запросимо у користувача номер стартової вершини (нехай у нашому прикладі стартової буде вершина 0). Зазначимо її в масиві $Active$ як пройдену (тобто в нашому прикладі $Active[0] = 1$) і в масиві $Label$ надамо їй значення 0 (відстань від вершини до себе самої). Таким чином стартовий стан масивів $Active$ і $Label$ такий:

	0	1	2	3	4	5
Active:	1	0	0	0	0	0

	0	1	2	3	4	5
Label:	0	55550	55550	55550	55550	55550

1. **Цикл 1** (поки у масиві Active є ще пройдені (тобто рівні 1) вершини.

1.1. Шукаємо вершину, яка має мінімальне значення в масиві Label і є пройденою. Нехай це буде перша вершина.

1.2. **Цикл 2** (рух рядком матриці суміжності)

Переглядаємо i-й рядок матриці MG і шукаємо суміжні з i-ї вершини. Для кожної суміжної вершини(j) перевіряємо: якщо справедлива нерівність $Label[j] > Label[i] + MG[i][j]$, то позначаємо її як пройдену, тобто. $Active[j]=1$ і заносимо в $Label[j] = Label[i] + MG[i][j]$.

Кінець Цикл 2

1.3 I-ю вершину отмечаем как не пройденную (т.е. $Active[i]=0$)

Кінець Цикл 1

Кінець алгоритму.

Описаний алгоритм обчислює довжину найкоротшого шляху, але зберігає сам шлях. Щоб отримати цей шлях, необхідно кожній вершини j за-поминати номер тієї вершини, з якої ми потрапили в j, тобто. для кожної вершини створити масив чи список «предків». Тепер, як тільки ми змінюємо значення Label для вершини j, в яку ми «прийшли» по ребру (i, j), необхідно записати i в масив або список «предків». Щоб отримати шлях у прямому, а не у зворотному порядку, необхідно розпочати виведення масиву або списку з кінця.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Розробити структуру (малюнок) неорієнтованого зваженого графу як тестовий приклад для програми.

Розробити та налагодити програму пошуку найкоротшого шляху в неорієнтованому зваженому графі. Номери стартової та кінцевої вершин графу задає користувач. Га виході програми необхідно отримати не тільки довжину найкоротшого шляху між вказаними вершинами, а й маршрут проходження вершин графу.

2. Оформити та захистити звіт з лабораторної роботи.

Звіт з лабораторної роботи повинен вміщати наступні розділи:

1. Постановка задачі.
2. Опис структури (малюнок) неорієнтованого незваженого графу – тестовий приклад.
3. Текст програми з коментарями.
4. Копія вікна виконання програми на тестових вхідних даних.
5. Висновки.

КОНТРОЛЬНІ ПИТАННЯ

- 1 Як зв'язані між собою різні способи подання графів?
- 2 Як від виду або подання графа залежить часова складність алгоритмів пошуку в глибину й завширшки ?
- 3 Дати поняття обходу графу.
- 4 Суть алгоритмів обходу в глибину та завширшки.
- 5 Який з методів обходу графу лежить в основі алгоритму Дейкстри?
- 6 Чи можна знайти найкоротший шлях в графі на основі обходу в глибину?

Методичні рекомендації до виконання лабораторної роботи № 16

Тема: Складання та налагодження програми рішення задачі комівояжера методом гілок та границь.

Мета: Отримання навичок в використанні методу гілок та границь. Реалізація програми рішення задачі комівояжера.

ТЕОРЕТИЧНІТВІДОМОСТІ

1 Загальний план рішення задачі комівояжера

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

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

Більш докладно ці етапи рішення завдання розкриті нижче.

2 Докладна методика рішення задачі комівояжера

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

Отже, методика рішення завдання комівояжера:

1. Побудова матриці з вихідними даними

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

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

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

2. Знаходження мінімуму по рядках

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

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

3. Редукція рядків

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

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

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

4. Знаходження мінімуму по стовпцях

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

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

5. Редукція стовпців

Віднімаємо з кожного елемента матриці відповідне йому d_j

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

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

6. Обчислення оцінок нульових кліток

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

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

7. Редуція матриці

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

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

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

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

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

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

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

9. Обчислення підсумкової довжини шляхи й побудова маршруту

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

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

Загальна довжина шляху: $L = 30$.

ЗМІСТ І ПОСЛІДОВНІСТЬ ВИКОНАННЯ ЗАВДАНЬ

1 Провести аналіз поставленої задачі.

Загальна постановка завдання:

Розробити структуру (малюнок) неорієнтованого зваженого графу як тестовий приклад для програми.

Розробити та налагодити програму пошуку найкоротшого шляху в неорієнтованому зваженому графі. Номери стартової та кінцевої вершин графу задає користувач. На виході програми необхідно отримати не тільки довжину найкоротшого шляху між вказаними вершинами, а й маршрут проходження вершин графу.

2. Оформити та захистити звіт з лабораторної роботи.

Звіт з лабораторної роботи повинен вміщати наступні розділи:

1. Постановка задачі.
2. Опис структури (малюнок) неорієнтованого незваженого графу – тестовий приклад.
3. Текст програми з коментарями.
4. Копія вікна виконання програми на тестових вхідних даних.
5. Висновки.

КОНТРОЛЬНІ ПИТАННЯ

1. Сформулюйте визначення завдання дискретного програмування.
2. Яка основна ідея методу гілок і границь?
3. У чому складаються особливості реалізації методу гілок і границь при рішенні конкретних класів завдань?
4. Що таке нижня оцінка множини?
5. У чому полягає процес розгалуження множини?
6. Які множини називаються кінцевими?
7. Сформулюйте ознаку оптимальності в завданні цілочисельного лінійного програмування.
8. Сформулюйте постановку завдання про комівояжера.
9. У чому полягає процедура приведення матриці в завданні про комівояжера?
10. Сформулюйте критерій оптимальності циклу для завдання про комівояжера.