МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЦЕНТРАЛЬНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Методичні вказівки до виконання лабораторних робіт з навчальної дисципліни “Технології розробки алгоритмів” для студентів денної та заочної форми навчання спеціальностей 6.050102 та 123 “Комп’ютерна інженерія”, 6.170103 та 125 “Кібербезпека”

ЗАТВЕРДЖЕНО
на засіданні кафедри кібербезпеки та програмного забезпечення,
протокол від 29 серпня 2018 року № 1

КРОПИВНИЦЬКИЙ
2018

Методичні вказівки до виконання лабораторних робіт з навчальної дисципліни “Технології розробки алгоритмів” для студентів денної та заочної форми навчання спеціальностей 6.050102 та 123 “Комп’ютерна інженерія”, 6.170103 та 125 “Кібербезпека” / уклад. Гермак В.С.; Кропивницький: ЦНТУ – 2018.– 81 с.

Укладач: Гермак В.С.

Рецензенти: Смірнов О.А., докт. техн. наук, професор;
Дрєєв О. М., канд. техн. наук.

Вступ

Методичні вказівки вміщують теоретичний матеріал, приклади та рекомендації по технологіям розробки алгоритмів. Приводяться контрольні питання та завдання для виконання лабораторних робіт з дисципліни “Технології розробки алгоритмів”.

Рекомендовано студентам ЦНТУ денної та заочної форми навчання спеціальностей 6.050102 та 123 “Комп’ютерна інженерія”, 6.170103 та 125 “Кібербезпека”.

Метою виконання лабораторних робіт є закріплення та поглиблення знань, отриманих в процесі вивчення дисципліни “Технології розробки алгоритмів”.

Студентам необхідно ознайомитися з теоретичним матеріалом, відповісти на питання та виконати всі завдання, що даються у лабораторних роботах.

5. Теми лабораторних занять

№ з/п	Назва теми
1	Аналіз часу виконання програми.
2	Рекурсія.Рекурсивні процедури і функції.
3.	Алгоритми сортування.
4	Алгоритми пошуку. Хешування.
5	Дерева. Побудова бінарного дерева та реалізація основних операцій над його елементами.
6	Дерева. Видалення елементів з бінарного дерева. Невузлове представлення бінарних дерев, збалансовані бінарні дерева, дерево з випадковим пошуком.
7	Тестування і відладка програмного засобу.

Рекомендована література

Базова

1. Кнут Д. Искусство программирования для ЭВМ, т. 1. Основные алгоритмы. – С.-П.: Вильямс, 2000.
2. Кнут Д. Искусство программирования для ЭВМ, т. 2. Получисленные алгоритмы. – С.-П.: Вильямс, 2000.
3. Кнут Д. Искусство программирования для ЭВМ, т. 3. Сортировка и поиск. – С.-П.: Вильямс, 2000.
4. Непейвода Н.Н. Стили и методы программирования Интернет-университет информационных технологий - ИНТУИТ.ру, 2005
5. Борисенко В.В. Основы программирования Интернет-университет информационных технологий - ИНТУИТ.ру, 2005
6. Терехов А.Н. Технология программирования БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2007

7. Симонович С.В., Евсеев Г.А. Занимательное программирование:Delphi. – М.: АСТ-Пресс Книга; Издательство «Развитие», 2003. – 368 с.
8. Закревский А.Д. Алгоритмы синтеза дискретных автоматов. М., Наука, 1971
9. Голицына О.Л., Попов И.И. Основы алгоритмизации и программирования: Учебное пособие. - М.: Форум: Инфра-М, 2004.
10. Семакин И.Г., Шестаков А.П. Основы программирования: Учебник. - М.: Мастерство, 2001.

Допоміжна

1. Алгоритмизация и программирование вычислительных задач: А. И. Заковряшин — Санкт-Петербург, 2002 г.- 80 с.
2. Программирование в алгоритмах: С. Окулов — Санкт-Петербург, Бином. Лаборатория знаний, 2007 г.- 384 с.
3. Программирование. Математические основы, средства, теория: С. Лавров — Санкт-Петербург, БХВ-Петербург, 2001 г.- 320 с.
4. Касьянов В.Н., Евстегнеев В.А. "Графы в программировании: обработка, визуализация и программирование" – Санкт-Петербург, БХВ-Петербург, 2003 г.- 1104 с.
5. Стивен С. Скиена "Алгоритмы. Руководство по разработке." – Санкт-Петербург, БХВ-Петербург, 2011 г.- 719 с.

Інформаційні ресурси

1. http://algolist.manual.ru/
2. http://moodle.kntu.kr.ua/course/view.php?id=469

Лабораторна робота № 1.

Тема: **“Аналіз часу виконання програми”.**

Мета: Опанувати навичками визначення часу та порядку степеня росту алгоритму, набути навичок в розробці блок-схем, псевдокодів алгоритмів та написанні програм з використанням розгалужень та циклів.

Теоретичні відомості:

Для більшості проблем існує багато різних алгоритмів. Який з них вибрати для вирішення конкретного завдання? Це питання дуже ретельно опрацьовується в програмуванні.

Ефективність програми (коду) є дуже важливою її характеристикою. Користувач завжди віддає перевагу ефективнішому рішенню навіть в тих випадках, коли ефективність не є вирішальним чинником.

Ефективність програми має дві складові: пам'ять (або простір) і час.

Просторова ефективність вимірюється кількістю пам'яті, потрібної для виконання програми.

Комп'ютери володіють обмеженим об'ємом пам'яті. Якщо дві програми реалізують ідентичні функції, то та, яка використовує менший об'єм пам'яті, характеризується більшою просторовою ефективністю. Іноді пам'ять стає домінуючим чинником в оцінці ефективності програм. Проте в останній роки у зв'язку з швидким її здешевленням ця складова ефективності поступово втрачає своє значення.

Часова ефективність програми визначається часом, необхідним для її виконання.

На час виконання програми мають вплив наступні фактори:

Введення вхідної інформації в програму.

Кількість скомпільованого коду програми, що виконується.

Машинні інструкції, які використовуються для виконання програми.

Часова складність алгоритму відповідної програми.

Час виконання програми можна визначити як функцію від вхідних даних.

Для опису швидкості росту функції використовується O – символіка. Тобто, коли ми говоримо, що час виконання $T(n)$ деякої програми має порядок $O(n^2)$, то розуміємо, що існують додатні c і n_0 такі, що для всіх n, більших або рівних n_0, виконується нерівність $T(n) \le cn^2$. Усі функції часу виконання визначені на множині невід’ємних цілих чисел і їх значення також невід’ємні, але необов’язково цілі.

Будемо казати, що $T(n)$ має порядок $O(f(n))$, якщо існують константи c і n_0 такі, що для всіх $n \ge n_0$ виконується нерівність $T(n) \le cf(n)$.

Для програм, у яких час виконання має порядок $O(f(n))$, говорять, що вони мають порядок росту $f(n)$. Коли використовують символіку $O()$, мають на увазі не точний час виконання, а лише верхню межу, або час виконання в найгіршому випадку, при чому з точністю до постійного множника.

O – функції виражають відносну швидкість алгоритму в залежності від

деякої змінної.

Кращий спосіб порівняння ефективності алгоритмів полягає в зіставленні їх порядків складності. Цей метод застосовний як до часової, так і просторової складності. Порядок складності алгоритму виражає його ефективність звичайно через кількість оброблюваних даних.

Наприклад, деякий алгоритм може істотно залежати від розміру оброблюваного масиву. Якщо, скажімо, час обробки подвоюється з подвоєнням розміру масиву, то порядок часової складності алгоритму визначається як розмір масиву.

Порядок алгоритму - це функція, домінуюча над точним виразом часової складності.

Функція f(n) має порядок O(g(n)), якщо є константа K і лічильник n_0, такі, що f(n)<=(K*g(n), для $n>n_0$.

O-функції виражають відносну швидкість алгоритму залежно від деякої змінної (або змінних).

Існують три правила для визначення складності:

1. $O(c\,f(n)) = O(f(n))$;

2. Правило добутку.

Якщо $T_1(n)$ і $T_2(n)$ мають степені росту $O(f(n))$ і $O(g(n))$ відповідно, то добуток $T_1(n)\cdot T_2(n)$ має степінь росту $O(f(n)\cdot g(n))$; або O(f/g)=O(f)/O(g);

3. Правило сум.

Якщо $T_1(n)$ і $T_2(n)$ мають степені росту $O(f(n))$ і $O(g(n))$ відповідно, тоді час послідовного виконання фрагментів P_1 і P_2, тобто $T_1(n)+T_2(n)$, має степінь росту $O(\max(f(n),g(n)))$.

Перше правило декларує, що постійні множники не мають значення для визначення порядку складності. O(1,5*N)=O(N)

З другого правила виходить, що порядок складності добутку двох функцій рівний добутку їх складностей.

O((17*N)*N)= O(17*N)*O(N) = O(N)*O(N)=O(N*N) = $O(N^2)$

З третього правила виходить, що порядок складності суми функцій визначається як порядок домінанти (максимуму) першого і другого доданків, тобто вибирається найбільший порядок.

$O(N^5+N^2)$=O(N5)

O - аналіз складності отримав широке застосування в багатьох практичних додатках. Але необхідно чітко розуміти його обмеженість.

До основних недоліків можна віднести наступні:

для складних алгоритмів отримання O - оцінок як правило, або дуже трудоємне або практично неможливе;

зазвичай складно визначити складність „в середньому”;

O - оцінка дуже не точна для відображення більш тонких відмінностей алгоритмів;

O - аналіз дає мало інформації для аналізу поведінки алгоритмів при обробці невеликих об'ємів даних.

Класифікація алгоритмів

Постійні – складність не залежить від n: $O(1)$;

Лінійні – складність $O(n)$;

Поліноміальні – складність: $O(n^m)$, де $m-const$;

Екпоненціальні – складність: $O(t^{t(n)})$, де $t-const$ більша за 1, а $f(n)$ - поліноміальна функція;

Суперполіноміальні – складність: $O(c^{f(n)})$, де $c-const$, а $f(n)$ - функція яка зростає швидше за постійну, але повільніше лінійної.

Приклад виконання лабораторної роботи

Написати алгоритм, псевдокод та програму на Delphi 7 для обчислення функції F(x,y): $f(x,y) = \begin{cases} x^2 + 7y, & x < 5 \\ 2x + y, & x \geq 5 \end{cases}$

Блок-схема

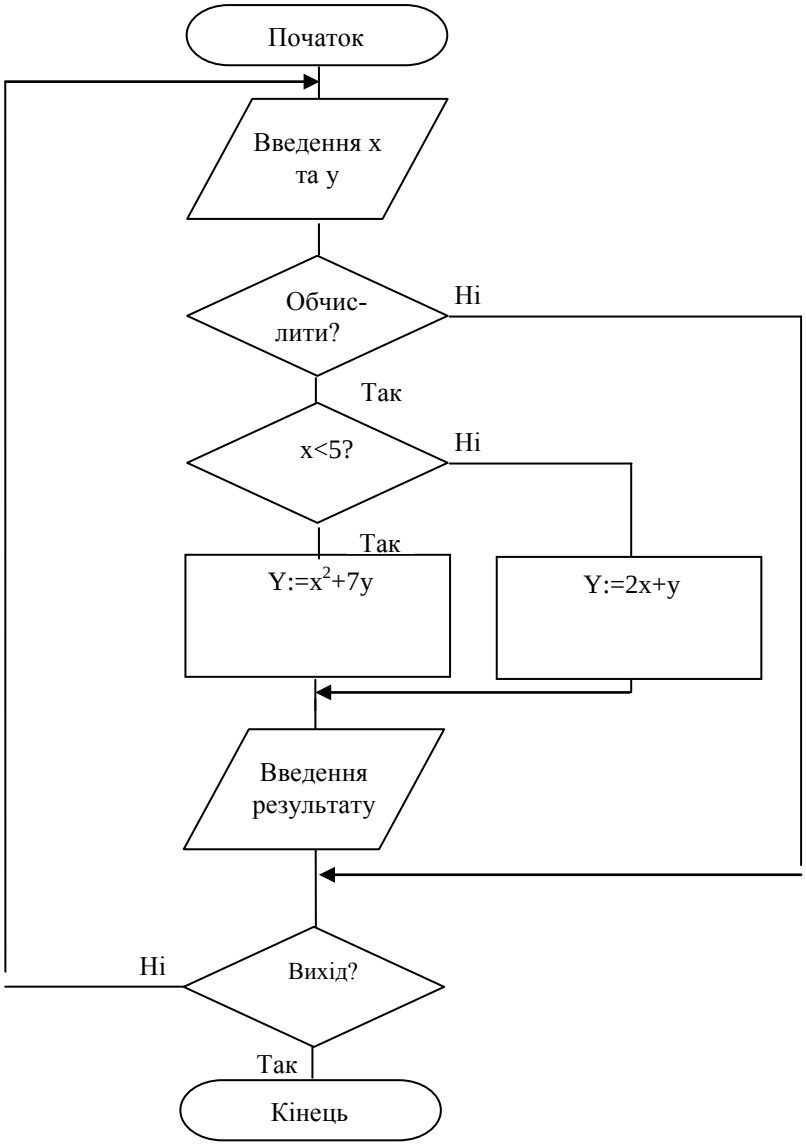


Рисунок 1 – Приклад блок-схеми алгоритму

Псевдокод

алгоритм Обчислення функції F (**арг ціле** x, **арг ціле** y, **рез ціле** f)
початок
  введення x, y
  якщо x<5 **тоді** F:= x*x+7*y
  інакше f:=2*x+y
 виведення f
кінець

Примітка. У псевдокоді базові керуючі структури представляються наступним чином:

- розгалуження if: **якщо** умова **тоді** дія1 **інакше** дія2
- цикл while: **поки** умова **тоді** дія
- цикл for: **для** і від n1 до n2

Лістинг програми

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes,
Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Image1: TImage;
    Label2: TLabel;
    Label3: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    Label4: TLabel;
    Edit3: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

//підпрограма натиснення кнопки «Обчислити»
procedure TForm1.Button1Click(Sender: TObject);
var x, y, f: integer; //ініціалізація змінних
begin

// обчислення функкії f(x,y)
x:=StrToInt(Edit1.Text); //присвоення змінній x
введеного значення
y:=StrToInt(Edit2.Text); //присвоення змінній y
введеного значення
if(x<5) then f:=x*x+7*y // обчислення f(x,y), якщо x<5
else f:=2*x+y;          // обчислення f(x,y), якщо x>=5
Edit3.Text:=IntToStr(f); // виведення результатів
end;
end.
```

Скриншот програми

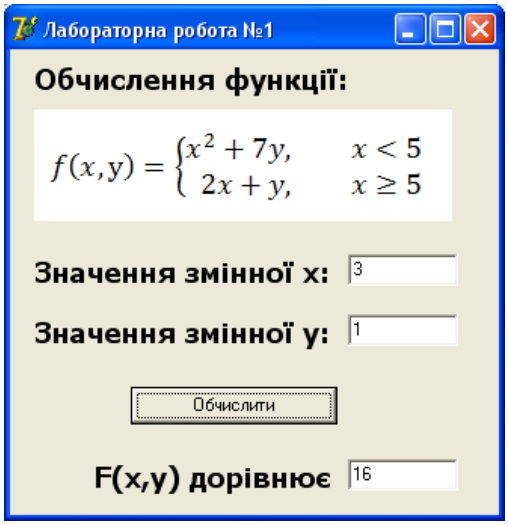


Рисунок 2 – Приклад інтерфейсу програми

Завдання:

Постановка задачі

у відповідності зі своїм варіантом написати блок-схеми, псевдокод алгоритмів та програми (на довільній мові програмування, наприклад, Pascal, C++, Delphi, або будь-якій іншій *структурній* мові програмування) для обчислення вказаних функцій та рішення заданих задач, оцінити їх

складність, використовуючи О-символіку.

Варіант 1

1. Вводячи в циклі по 4 оцінки, які студенти отримали в сесію, визначити число невстигаючих студентів і середній бал групи по всім екзаменам.

2. Скласти алгоритм визначення найменшого елемента матриці A(m,n), вказавши номер його строки і стовпця.

Варіант 2

1. В області 10 районів. Задані площі, які в кожному році засівають пшеницею, і врожай, який зібраний в кожному районі. Визначити середню врожайність пшениці по кожному району і по області в цілому (за три роки).

2. Зріст учнів в класі представлений у вигляді масиву. Зріст дівчат кодується знаком „+", зріст хлопців знаком „-". Визначити середній зріст хлопців.

Варіант 3

1. Визначити, скільки різних сигналів може бути подано m прапорцями різних кольорів. Відмінність сигналів заключається в порядку розташування різнокольорових прапорців на щоглі. Розв'язати при m = 6.

2. Нехай маємо ряд спостережень $x_1, x_2, ..., x_n$. Скласти алгоритм визначення середнього значення x_{cp} і середнього квадратичного відхилення S_x по формулам

$$x_{cp} = \frac{\sum_{i=1}^{n} x_i}{n}; \qquad S_x = \sqrt{\frac{\sum (x_i - x_{cp})^2}{n}}$$

Варіант 4

1. Біля стінки стоїть нахилена палиця довжиною x. Один її кінець знаходиться на відстані y від стінки. Визначити значення кута α між палицею і підлогою для значень 4,5 м. і y, який змінюється від 2 до 3 м. з кроком 0,2 м.

2. Скласти програму для обробки результатів кросу на 500 м. для жінок. В кросі брали участь не більше за 100 учасниць. Для кожної учасниці внесіть її прізвище, шифр групи, прізвище викладача, результат. Потрібно отримати результуючу таблицю, яка впорядкована за результатом, і в якій є інформація про виконані норми 1 розряду. Обчислити сумарну кількість студенток, які виконали цю норму.

Варіант 5

1. Скласти таблицю вартості порцій сиру вагою 50, 100, 150, ...,1000 грамів

(ціна 1 кг. – 15 грн.)

2. Прізвища учасників змагань по фігурному ковзанню після короткої програми в порядку, який відповідає зайнятому місцю. Скласти список учасників в порядку їх стартових номерів для вільної програми (учасники виступають в порядку зворотному зайнятим місцям).

Варіант 6

1. Почавши тренування, спортсмен в перший день пробіг 10 км. Кожний наступний день він збільшував денну норму на 10 % від норми попереднього дня. Через скільки днів спортсмен буде пробігати в день більше 20 км.

2. Зміни основних фондів галузі описується рівнянням:

$k_{i+1} = \alpha k_1 + \beta I_i + \gamma I_{i-1}$.

Заданий план інвестицій $I_0, I_1, ..., I_n$ і початкове значення основних фондів k_1. Скласти алгоритм визначення значень $k_2, ..., k_n$.

Варіант 7

1. Визначити середній зріст дівчаток і хлопчиків одного класу. В класі навчається n учнів.

2. В пам'яті ПЕОМ зберігаються списки номерів телефонів і прізвища абонентів, впорядковані за номерами телефонів, для кожного з п'яти телефонних вузлів міста. Один телефонний вузол включає декілька АТС (не більше 10). Номери АТС (перші дві цифри номери телефону), які відносяться до кожного телефонного вузла, також зберігаються в пам'яті ПЕОМ. Скласти програму, яка забезпечує швидкий пошук прізвища абонента по заданому номеру телефону.

Варіант 8

1. Вводячи в циклі по 5 оцінок кожного студента, отримати число студентів, які не мають оцінок 2 і 3. В групі навчається n студентів.

2. Результати перепису населення зберігаються в пам'яті ПЕОМ. Використовуючи масиви прізвищ і роки народження, надрукувати прізвища і підрахувати загальне число мешканців, які народилися раніше 1930 року.

Варіант 9

1. Скласти програму для визначення віку, який підходить в кандидати для вступу в шлюб, використовуючи наступну умову: вік дівчини дорівнює половині віку чоловіка плюс 7, вік чоловіка визначається відповідно як подвоєний вік дівчини мінус 14. Дані для перевірки роботи задачі задати самостійно.

2. Японська радіокомпанія провела опитування 250 радіослухачів по питанню: "Яку тварину Ви пов'язуєте з Японією і японцями?". Скласти програму отримання п'яти відповідей, які найбільше зустрічаються і їх часток в (%).

Варіант 10

1. Скласти програму, яка реалізує уривок казки: питає: куди бажає піти герой (направо, наліво і прямо), і друкує, що його чекає в кожному випадку. Відповідь ПЕОМ присвоїти символьній змінній і надрукувати. Текст питань і відповідей ПЕОМ задати самостійно.

2. Задані дві вибірки $x_1, x_2, ..., x_n$; $y_1, y_2, ..., y_n$. Скласти ефективний алгоритм визначення вибіркового коефіцієнту кореляції:

$$r_{xy} = \frac{\sum x_i y_i / n - \overline{xy}}{S_x S_y}$$

де $\bar{x} = \sum \frac{x_i}{n}$, $\bar{y} = \sum \frac{y_i}{n}$ - середнє значення вибірок.

$S_x = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$ - незміщене середнє квадратичне відхилення спостережень першої вибірки (аналогічно для другої вибірки).

Варіант 11

1. Задано n трійок чисел a, b, c. Вводячи їх по черзі і інтерпретуючи як довжини сторін трикутника визначити скільки трійок може бути використано для побудови трикутника (числа a, b, c при введені розташувати в порядку зростання.)

2. Відомо, що в Києві самими теплими є дні з 15 липня по 15 серпня. Для проведення фестивалю були обрані 7 днів підряд найбільш теплих по даним за останні 10 років. Складіть програму для виконання цієї роботи на ПЕОМ.

Варіант 12

1. Почавши тренування спортсмен в перший день пробіг 10 км. Кожний наступний день він збільшував денну норму на 10 % від норми попереднього дня. Через скільки днів спортсмен пробіжить сумарний шлях 150 км.

2. При виборі місця будівництва житлового комплексу при металургійному заводі необхідно враховувати „розу вітрів” в даній місцевості. На основі даних щоденного визначення направлення вітру, яке проводилося протягом року, визначити доцільне взаємне розташування промислової і житлової зони.

Вказівка: Напрямок вітру координується наступним чином: 1 – північний, 2 – південний, 3 – східний, 4 – західний, 5 – північно – західний, 6 – північно – східний, 7 – південно – західний, 8 – південно – східний.

Варіант 13

1. Визначити, скільки різних сигналів може бути подано m прапорцями різних кольорів. Відмінність сигналів заключається в порядку розташування різнокольорових прапорців на щоглі. Розв’язати при m = 6.

2. Скласти алгоритм визначення найменшого елемента матриці A(m,n), вказавши номер його строки і стовпця.

Варіант 14

1. Почавши тренування, спортсмен в перший день пробіг 10 км. Кожний наступний день він збільшував денну норму на 10 % від норми попереднього дня. Через скільки днів спортсмен буде пробігати в день більше 20 км.

2. Визначити середній зріст дівчаток і хлопчиків одного класу. В класі навчається n учнів.

Варіант 15

1. Біля стінки стоїть нахилена палиця довжиною x. Один її кінець знаходиться на відстані y від стінки. Визначити значення кута α між палицею і підлогою для значень 4,5 м. і y, який змінюється від 2 до 3 м. з кроком 0,2 м.

2. Результати перепису населення зберігаються в пам’яті ПЕОМ. Використовуючи масиви прізвищ і роки народження, надрукувати прізвища і

підрахувати загальне число мешканців, які народилися раніше 1930 року.

Контрольні питання:

1. Поняття алгоритму, його визначення.
2. Назвіть властивості алгоритму.
3. Яких умов необхідно дотримуватись щоб побудувати алгоритм?
4. Які алгоритмічно розв'язні чи нерозв'язні проблеми відомі на даний час?
5. Як розв'язуються алгоритмічно нерозв'язні проблеми?
6. Які форми представлення алгоритмів Ви знаєте?
7. Які фактори впливають на час виконання програм?
8. Коли і для чого використовують О-символіку?
9. При якому значенні n алгоритм, який потребує $100n^2$ операцій, ефективніший за алгоритм, який потребує 2^n операцій.
10. Знайдіть використовуючи О-символіку, час виконання в найгіршому випадку наступних процедур як функцій від n:

а)
```
procegure matmpy  (n: integer);
var
i, j, к: integer; begin
for i: = 1 to n do
for j:=1 to n do begin
C[i,j]:= 0;
for k:= 1 to n do
C[i,j]:= C[i,j] + A[i,k] * B[k,j]
end
end
```

б)
```
procegure mystery (n: integer);
var
i, j, к: integer;
begin
for i:= 1 to n - 1 do
for j:= i + 1 to n do
for k:= 1 to j do
{группа операторів з часом виконання 0(1)}
end
```

в)
```
procegure veryodd (n - integer);
var
i, j, x, y: integer;
begin
for i:= 1 to n do
if нечетное(i) then begin
for j:=1 to n do
x: = x + 1 ;
for j:= 1 to i do
```

```
y:= y + 1
end
end
```

*г) **procegure** *recursive* (n: integer): integer;

```
begin
if n <= 1 then
return(1)
else
return (recursive(n - 1) + recursive{n - 1))
end
```

11. Розташуйте наступні функції в порядку зростання а) n, б) $\sqrt{n}$, в) $\log n$, г) $\log\log n$, д) $\log^2 n$, е) $n/\log n$, ж) $\sqrt{n}\log^2 n$, з) $(1/3)^n$, и) $(3/2)^n$, к) 17.

12. Дана послідовність чисел $x_1, x_2, ..., x_n$. Покажіть, що за час $O(n\log n)$ можна визначити, чи є в цій послідовності два однакові числа.

13. Дано масив S який складається з n дійсних чисел, а також число x. Як за час $O(n\log n)$ визначити, чи можна представити x у вигляді суми двох елементів масиву S?

Лабораторна робота № 2.

Тема: **“Рекурсія.Рекурсивні процедури і функції”.**

Мета: Розлянути поняття рекурсії, дослідити доцільність застосування рекурсивних алгоритмів та заміну рекурсивних алгоритмів ітеративними.

Теоретичні відомості:

Визначення називається *рекурсивним*, якщо воно задає елементи множини за допомогою інших елементів цієї ж множини. Об'єкти, задані рекурсивним визначенням, також називаються рекурсивними. І нарешті, рекурсія – це використання рекурсивних визначень.

Приклад 1

Значення функції факторіал задаються виразом: 0!=1, n=n*(n-1)!. Вони утворюють множину {1,2,6.}: 0!=1, 1=1*0!, 2=2*1!, 6=3*2! і т.д. Всі його елементи, окрім першого, визначаються рекурсивно.

Приклад 2

Арифметичні вирази з константами і знаком операції “+” в інфіксній дужковій формі задаються таким визначенням:константа є виразом;якщо E і F є виразами , то (E)+(F) також є виразом.Такими виразами є, наприклад, 1, 2 (1)+(2), ((1)+(2))+(1). Всі вони, окрім констант 1 і 2, визначаються рекурсивно.

Об'єкти, визначені в прикладах є рекурсивними.

У рекурсивному визначенні не повинно бути “зачарованого кола”, коли об'єкт визначається за допомогою себе самого або за допомогою інших, але заданих через нього ж.

Приклад 3

Змінимо визначення функції факторіал на наступне: n!=n*(n-1)! При n>0, 0!=1!. Спочатку значення функції від 1 виражається через її ж значення від 0, яке, у свою чергу, – через значення від 1. За таким визначенням так і не дізнатися, чому ж рівне 1!.

Щоб подібна нескінченність не виникала в рекурсивному визначенні, повинні виконуватися наступні умови:

-безліч визначуваних об'єктів є частково впорядкованою;

-кожна убуваюча по цьому впорядкуванню послідовність елементів закінчується деяким мінімальним елементом;

-мінімальні елементи визначаються нерекурсивно;

-немінімальні елементи визначаються за допомогою елементів, які менше за них по цьому впорядкуванню.

Неважко дійти переконання, що визначення з прикладів 1 і 2 задовольняють цим умовам, а з прикладу 3 – ні.

Відношення називається відношенням часткового порядку, якщо воно має такі властивості:

1) Для кожного елементу a множини у відношенні є пара (a, a);

2) Якщо у відношенні є пара (a, b) з різними елементами a і b, то пари (b, a) там немає. При цьому ми говоримо, що a менше за b.

3) Якщо a менше за b, a b менше з, то a менше c.

Втім, елементів a, b, c таких, що a менше за b, a b менше c, в множині

може і не бути – при виконанні властивостей (1) і (2) відношення буде відношенням часткового порядку.

Множина із заданим на ньому відношенням часткового порядку називається частково впорядкованою. Елемент частково впорядкованої множини називається мінімальним, якщо в множині немає елементів, менших його.

Рекурсивні алгоритми в програмуванні реалізовані в механізмі так званих рекурсивних підпрограм. **Рекурсивною** називається підпрограма, виконання якої приводить до її ж **повторного** виклику з зміненими параметрами.

Якщо підпрограма просто викликає сама себе, то така рекурсія називається прямою. Якщо ж декілька підпрограм викликають одна одну, але ці виклики "замкнуті в кільце", то така рекурсія називається непрямою.

З рекурсивними підпрограмами пов'язані два важливі поняття – *глибина рекурсії* і *загальна кількість викликів, породжених викликом рекурсивної підпрограми*.

Розглянемо перше з них. При обчисленні факторіалу виклик функції з аргументом, наприклад 4, закінчується лише після закінчення виклику з аргументом 3, а той, у свою чергу, після виклику з аргументом 2 і т.д. Такі виклики називаються *вкладеними*. Таким чином, виклик з аргументом 4 породжує ще три вкладені виклики. Взагалі, при виклику цієї функції з аргументом n породжується ще n-1 виклик, і загальна кількість незавершених викликів досягає n. Таким чином, *глибиною рекурсії виклику підпрограми* називається максимальна кількість незавершених рекурсивних викликів при виконанні її виклику.

При виконанні виклику з глибиною рекурсії m одночасно існує m екземплярів локальної пам'яті. Кожен екземпляр має певний розмір, і якщо глибина буде занадто великою, то пам'яті, наданої процесу виконання програми, може не вистачити.

Друге поняття можна назвати *загальною кількістю вкладених викликів*, породжених викликом рекурсивної підпрограми. Ця кількість впливає на час виконання виклику.

Таким чином, вживання рекурсивних підпрограм вимагає обережності і уміння оцінити можливу глибину рекурсії і загальну кількість викликів.

Рекурсивний виклик може бути непрямим. В цьому випадку підпрограма звертається сама до себе опосередковано, шляхом виклику іншої підпрограми, в якій звернення до першої.

Обмеження глибини рекурсії

Теоретично, рекурсія може бути нескінченною. Проте такий варіант навряд чи кого-небудь влаштує: рекурсивний алгоритм, як і будь-який нерекурсивний його побратим, зобов'язаний видавати результат своєї роботи за якийсь осяжний час. Крім того, пам'ять у комп'ютера не гумова, в ній може поміститися лише кінцеве число контекстів одночасно відкритих екземплярів рекурсивної підпрограми.

Отже, кожна рекурсивна підпрограма повинна містити в собі **ознаку**

закінчення - своєрідну "огорожу", що визначає максимальну глибину вкладеності для цієї рекурсії. Ознака кінця рекурсії може бути як явною, так і неявною.

Рекурсія і ітерація в деякому розумінні протилежні. Рекурсія вирішує задачу від складного до простого, ітерація - від простого до складного. Текст рекурсивного алгоритму виражає складний об'єкт через більш простий (простіші) такого ж типа. Текст ітеративного алгоритму описує процес будівництва, починаючи з дрібних деталей. Рекурсивний алгоритм виражає невідоме через невідоме, але зв'язок між цими двома невідомими досить простий, математично прозорий, тому довести правильність алгоритму легко. Стани об'єкту на двох різних кроках ітеративного алгоритму - як два стани недобудованого будинку: поки не покладена остання цеглина, не очевидно, що вийде в кінці. Тому довести правильність такого алгоритму звичайно досить складно.

Якщо обчислення починається з відшукання значення функції при мінімальному значенні аргументу, використовує його для визначення наступного і так далі, таке обчислення, записане у вигляді:

```
  y := a;
for i:= 1 to n do
  y := f(y, i);
```

називається ітерацією.

Важлива відмінність між рекурсією і ітерацією полягає в механізмі реалізації.

Виконавець рекурсивного алгоритму зводить невідоме до іншого невідомого, накопичуючи інформацію і відкладаючи реальні обчислення до моменту зведення аргументу функції до такого значення, при якому значення функції відоме.

Виконавець ітеративного алгоритму починає з відомого значення і крок за кроком перетворить відомі вже значення функції в значення для великих аргументів. В цьому відношенні ітерація подібна зворотному ходу в реалізації рекурсії. З цих слів (ітераційний процес є частина рекурсивного) можна зробити висновок про те, що ітерація простіше (як частина цілого) і необхідно вирішувати задачі ітераційними методами.

Насправді ситуація складніша. У декількох словах її можна охарактеризувати так: рекурсивне рішення задачі складається з двох частин - прямого ходу (зведення завдання до ітераційної форми) і зворотного (виконання ітерацій). Звільнивши машину від виконання прямого ходу, полегшуємо її роботу, але перекладаємо цю частину роботи на програміста, якому ставиться в обов'язок перетворити програму. Іноді це не складає труднощів для програміста, і він відразу може написати ітераційний варіант, а іноді проблема буде важко вирішуваною.

Розглянемо докладніше зв'язок між рекурсією і ітерацією на прикладі обчислення чисел Фібоначчі.

Рекурсивний варіант.

```
function Fr (n: integer): integer;
```

```
 begin
 if n = 1 or n = 2 then
Fr:= 1
  else Fr:=Fr(n-l) + Fr(n-2);
 end;
```

Ітеративний варіант

```
   function Fi (n: integer): integer;
     var i N1, N2, Result: integer ;
   begin
     N1:= 1; N2:= 1; { Установка початкових значень}
     Result:= 1; { на той випадок, якщо n = 1 або 2.}
     for i:= 3 to n do { Не виконується, якщо n < 3.}
     begin
       Result:= N1 + N2; { Відповідає основній формулі чисел Фібоначчі.}
       N2:= N1; { Зрушення значень на один індекс.}
       N1:= Result { Зрушення значень на один індекс.}
     end;
     Fi:= Result { Установка значення функції.}
   end;
```

До порівняльної оцінки цих двох алгоритмів можна підходити з двох позицій: використання ресурсів ЕОМ і складності написання і відладки програми.

А. Ресурси. В даному випадку під ресурсами розуміється час процесора і об'єм зайнятої програмою (і даними) пам'яті.

Рекурсивна програма включає наступні операції, що займають час процесора: виклик функції, порівняння n з одиницею і двійкою, складання; при кожному виклику відбуваються порівняння і в частині викликів (якщо n>2) відбуваються складання. Тому вирішальним параметром в оцінці часу є кількість викликів функції. Всі виклики можна розбити на рівні таким чином. На першому рівні знаходиться єдиний перший виклик з параметром n. На другому рівні знаходяться два виклики (з параметрами n-1 і n-2), проведені при першому виклику. На третьому рівні знаходяться чотири виклики з параметрами n-2, n-3, n-3, n-4 і т.д. із збільшенням кількості на кожному рівні в два рази по відношенню до попереднього рівня до тих пір, поки величини n-і не почнуть досягати значення 2. Ці виклики можна зобразити у вигляді дерева (малюнок 1).

Тут для визначеності узято n=7. Непросто оцінити, скільки вершин (викликів) в цьому дереві, але можна швидко одержати нерівності, що обмежують цю кількість зверху і знизу. Ліва гілка помічена числами n n-1, n-2, n-3, ..., 2. Отже, кількість рівнів не перевершує величини n-1. Права гілка помічена числами n n-2, n-4, n-6 = 1. Можна зробити висновок, що кількість рівнів не менша n/2. Якщо дерево має к рівнів, то в ньому 1+2+4+8+...+2k-1=2k вершин або, враховуючи нерівність n-1 > до > n/2, одержуємо 2n-1-1 > число викликів > 2n/2.

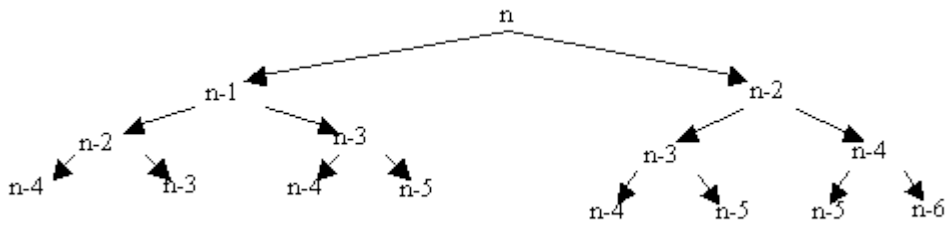


Рис.1. Дерево викликів рекурсивної функції обчислення чисел Фібоначчі

Як верхня, так і нижня межі швидко ростуть із збільшенням n. Вже при n=20 потрібно не менше 1000 викликів, хоча, як це ясно видно з дерева, багато викликів дублюють один одного - текст функції не передбачає контролю дублювання. З кожним незавершеним викликом функції пов'язано зберігання в стеку певних даних (це обговорювалося, коли розглядався механізм реалізації рекурсії). З огляду на те, що прохід по дереву проводиться в глибину, у будь-який момент часу потрібно зберігати від n-1 до n/2 груп даних.

Ітеративна програма включає складання, пересилки (привласнення) і перевірки (неявні) циклу на закінчення. Зайнятість процесора в цьому випадку зручно оцінити кількістю виконань тіла циклу. Заголовок циклу ясно указує на те, що тіло циклу (одне складання, три пересилки і команди управління циклом) виконується n - 2 рази; окрім цього три пересилки (привласнення) на початку і одне в кінці. Таким чином, кількість операцій може бути оцінена формулою an+b. Кількість використаної пам'яті - пам'ять для чотирьох змінних.

Ці розрахунки показують явну перевагу ітеративного методу порівняно з рекурсивним по використанню ресурсів ЕОМ для вирішення даного завдання.

В. Складність написання і відладки програми. По суті це питання про витрати праці програміста. Формальні числові оцінки тут важкі. Але порівняння двох текстів показує, що рекурсивна програма пишеться автоматично на основі постановки завдання, тоді як ітеративна вимагає перетворення завдання і "винаходження" алгоритму, введення додаткових змінних, відсутніх в постановці завдання, і неочевидних операцій. Все це приводить до необхідності доказу того, що одержаний алгоритм дійсно вирішує поставлену задачу, а не якусь іншу, що він правильно працює для всіх допустимих початкових даних. Положення ускладнюється тим, що замість статичного аналізу тексту (у разі рекурсії) необхідно представляти, як алгоритм розгортається в часі за допомогою якогось виконавця, тобто застосовувати модель комп'ютера.

З цієї точки зору рекурсивний алгоритм кращий ітеративного.

Аналіз пунктів (А) і (Б) дає нам протилежні висновки. У разі таких простих завдань як обчислення чисел Фібоначчі питання звичайно розв'язується на користь ресурсів комп'ютера.

Заміна рекурсивних алгоритмів ітеративними

Оскільки новий контекст створюється кожного разу, коли черговий

екземпляр рекурсивної підпрограми (сам ще залишаючись незавершеним) наново викликає себе ж, то пам'ять комп'ютера витрачається дуже швидко. Тому рекурсію при всієї її наочності не можна віднести до економічних способів програмування. Існує навіть спеціальна наука - **динамічне програмування**, що вивчає способи заміни рекурсивних алгоритмів адекватними ітеративними алгоритмами.

Якщо виконання підпрограми приводить тільки до одного виклику цієї ж самої підпрограми, то така рекурсія називається лінійною. Лінійну рекурсію досить легко замінити ітеративним алгоритмом. Наприклад, можна реалізувати функцію факторіалу двояко.

Рекурсивна реалізація	**Ітеративна реалізація**

function fact(k:byte):longint;
var x: longint;
begin
  if к = 0
  then fact:= 1
  else begin x:= fact(k-1)*k;
  fact:=x;
  end;
end;
	fact:= 1
	for i:= 2 to до do
	fact:= fact * i;

Якщо ж кожен екземпляр підпрограми може викликати себе кілька разів, то рекурсія називається нелінійною або такою, що розгалуджується. Для того, щоб перетворити таку рекурсію на ітеративний алгоритм, доведеться докласти багато зусиль і, можливо, навіть внести деякі зміни в оброблювану структуру даних.

Завдання:

Варіант 1

1) Напишіть рекурсивну підпрограму піднесення числа до степеня.

2) Напишіть рекурсивну підпрограму переведення натурального числа з десяткової системи в двійкову.

Варіант 2

1) Напишіть рекурсивну підпрограму визначення, чи є дане натуральне число простим.

2) Написати рекурсивну підпрограму друку десяткового запису цілого додатнього числа n.

Варіант 3

1) Напишіть рекурсивну підпрограму визначення кількості цифр натурального числа.

2) Запрограмуйте рекурсивний алгоритм розв'язання задачі про Ханойські вежі. Ця задача полягає в наступному:

Є три стержня і *n* дисків різного розміру. Диски можна надівати на стержні, утворюючи з них вежі. Спочатку всі диски в спадаючому порядку розташовані на стержні А. Задача полягає в тому, щоб перенести *n* дисків зі

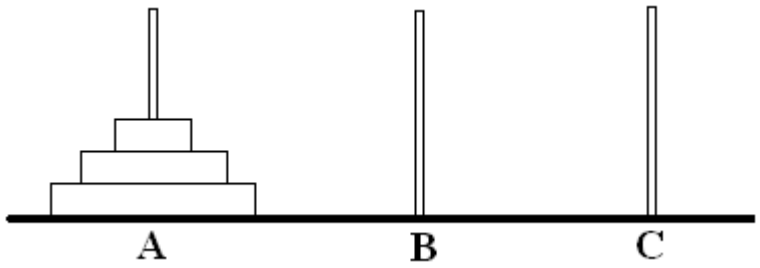
стержня A на стержень C, зберігши їх початковий порядок. При перенесенні потрібно керуватись наступними правилами:

а) На кожному кроці зі стержня на стержень переноситься рівно один диск;

б) Диск не можна класти на диск меншого розміру;

в) Для тимчасового зберігання можна використовувати стержень B.

Вежу зручно розглядати як таку, що складається з одного верхнього диску і вежі, утвореної іншими дисками.



Варіант 4

1) Напишіть рекурсивну підпрограму знаходження цифрового кореня натурального числа. (Додати всі цифри числа, потім всі цифри знайденої суми і т.д. доки не отримається одна цифра).

2) Написати рекурсивну підпрограму підрахунку числа вершин в дереві.

Варіант 5

1) Напишіть рекурсивну підпрограму знаходження суми перших n членів арифметичної прогресії.

2) Написати рекурсивну програму підрахунку числа листків в дереві.

Варіант 6

1) Напишіть рекурсивну підпрограму знаходження індекса мінімального елемента в масиві з n елементів.

2) Написати рекурсивну програму підрахунку висоти дерева (корінь має висоту 0, його сини - висоту 1 і т.п.; висота дерева - це максимум висот його вершин).

Варіант 7

1) Написати рекурсивну підпрограму для обчислення функції Аккермана для невід'ємних чисел n m, яка визначається таким чином:

A(n,m)=m+1 якщо n=0;

A(n,m)=A(n-1,1) якщо n<>0, m=0;

A(n,m)=A(n-1,A(n,m-1)) якщо n>0, m>0.

2) Напишіть рекурсивну підпрограму знаходження n-го члена геометричної прогресії.

Варіант 8

1) Написати рекурсивну підпрограму яка зчитує з клавіатури послідовність чисел і виводить її на екран в зворотньому порядку (закінчення послідовності при введенні 0).

2) Написати рекурсивну програму, яка знаходить всі перестановки чисел 1..n по одному разу.

Варіант 9

1) Написати рекурсивну підпрограму визначення найбільшого спільного дільника двох чисел.

2) Написати рекурсивну програму, яка дозволяє перерахувати всі представлення позитивного цілого числа n у вигляді суми послідовності незростаючих цілих позитивних доданків.

Варіант 10

1) Написати рекурсивну підпрограму що виводить цифри натурального числа в зворотньому порядку.

2) Написати рекурсивну програму, яка дозволяє розкласти натуральне число на співмножники всіма можливими способами (без повторень)

Варіант 11

1) Написати рекурсивну програму, яка по заданому n рахує число всіх вершин висоти n у заданому дереві.

2) Написати рекурсивну програму, яка дозволяє реалізувати індійський алгоритм піднесення до степеня. Цей алгоритм обчислення натурального *n-го (n*>1) степеня цілого числа *a* виглядає таким чином:

$a^n=a$ при n=1,

$a^n=a^{n \bmod 2} * (a^{n \operatorname{div} 2})^2$ при n>1.

Основна мета цього алгоритму – скоротити кількість множень при піднесенні до степеня.

Обчислення a^n зводиться до обчислення $a^{n \operatorname{div} 2}$, запам'ятовуванню результату, зведенню в квадрат і множенню його на *x* при непарному *n*.

Варіант 12

1) Напишіть рекурсивну підпрограму знаходження суми перших n членів геометричної прогресії.

2) Запрограмувати рекурсивний алгоритм сортування злиттям. Цей алгоритм полягає в наступному:

а) нехай К - індекс середнього елемента масиву А;

б) відсортуйте елементи до А(к) включно;

в) відсортуйте елементи після А(к);

г) з'єднайте ці два підмасиви в один відсортований масив

Варіант 13

1) Використовуючи лише команди write(x) при x=0..9, написати рекурсивну підпрограму друку десяткового запису цілого додатнього числа n.

2) Є деяка сума грошей S і набір монет з номіналами a_1, ., a_n. Монета кожного номінала є в єдиному екземплярі. Написати рекурсивну програму, яка дозволяє знайти всі можливі способи розміняти суму S за допомогою цих монет.

Варіант 14

1) Написати рекурсивну підпрограму для визначення, чи є симетричною частина рядка s починаючи з i-го елемента і закінчуючи j-м.

2) Написати рекурсивну програму відшукання максимального з N елементів, збережених в масиві a[1],. . ., a[N].

Варіант 15

1) Напишіть рекурсивну підпрограму обчислення суми цифр натурального числа.

2) Написати рекурсивну програму що здійснює пошук елементу у впорядкованому масиві (двійковий пошук). Є впорядкований масив і еталонний елемент. Потрібно визначити, чи міститься еталон в масиві. Якщо так, то повернути відповідний номер позиції. Якщо ні - вивести повідомлення. Для вирішення завдання використовується метод ділення початкового масиву навпіл. З еталоном порівнюється «середній» (розташований посередині) елемент масиву. Якщо він менше еталону – пошук продовжується в правій половині масиву. Якщо більше – в лівій. Пошук ведеться до тих пір, поки не буде виявлена відповідність, або поки довжина ділянок масиву, в яких ведеться пошук, не стане менше 1.

Контрольні питання:

1) Що таке рекурсія;
2) Що таке глибина рекурсії;
3) Які види рекурсії ви знаєте.

Лабораторна робота № 3

Тема: **“Алгоритми сортування”.**

Мета: Розлянути алгоритми сортування, дослідити доцільність застосування різних алгоритмів в конкретних випадках виходячи з їх ефективності.

Сортування бульбашкою

Розташуємо масив зверху вниз, від нульового елементу - до останнього.

Ідея методу: крок сортування полягає в проході від низу до верху по масиву. По дорозі переглядаються пари сусідніх елементів. Якщо елементи деякої пари знаходяться в неправильному порядку, то міняємо їх місцями.

Після нульового проходу по масиву "вгорі" виявляється "найлегший" елемент - звідси аналогія з бульбашкою. Наступний прохід робиться до другого зверху елементу, таким чином другий за величиною елемент піднімається на правильну позицію...

Робимо проходи по нижній частини масиву, що зменшується, до тих пір, поки в ній не залишиться тільки один елемент. На цьому сортування закінчується, оскільки послідовність впорядкована за зростанням.

```
{ Сортуються записи типу item по ключу item.key }
{ для допоміжних змінних використовується тип index }
procedure BubbleSort;
   var i,j: index; x:item;
begin for i:=2 to n do
   begin for j:=n downto i do
       if a[j-1].key > a[j].key then
    begin x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x;
    end;
    end;
end;
```

Середнє число порівнянь і обмінів має квадратичний порядок зростання: $O(n^2)$, звідси можна зробити висновок, що алгоритм бульбашки дуже повільний і малоефективний. Проте, у нього є величезний плюс: він простий і його можна по-всякому покращувати.

Якісно інше поліпшення алгоритму можна одержати з наступного спостереження. Хоча легка бульбашка знизу підніметься вгору за один прохід, важкі бульбашки опускаються з мінімальною швидкістю: один крок за ітерацію. Отже масив 2 3 4 5 6 1 буде відсортований за 1 прохід, а сортування послідовності 6 1 2 3 4 5 вимагає 5 проходів.

Щоб уникнути подібного ефекту, можна міняти напрям слідуючих один за іншим проходів. Отриманий алгоритм іноді називають *шейкерним* сортуванням.

```
procedure ShakerSort;
```

```
   var j,k,l,r: index; x: item;
begin l:=2; r:=n; k:=n;
   repeat
      for j:=r downto l do
      if a[j-1].key > a[j].key then
    begin x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x;
          k:=j;
    end;
    l:=k+1;
    for j:=l to r do
      if a[j-1].key > a[j].key then
    begin x:=a[j-1]; a[j-1]:=a[j]; a[j]:=x;
      k:=j;
    end;
    r:=k-1;
    until l>r;
end;
```

Аналіз алгоритму простої бульбашки:

Число порівнянь C = 1/2 (n^2 - n)

Числа пересилок: Mmin = 0, Mсереднє = 3/4 (n^2 - n), Mmax = 3/2 (n^2 - n).

Аналіз сортування шейкера:

Мінімальне число порівнянь Cmin = n-1

Середнє число проходів пропорційно n - k1* $\sqrt{n}$,

Среднее число порівнянь пропорційно 1/2 (n^2 - n(k2+ln(n))).

Наскільки описані зміни вплинули на ефективність методу? Середня кількість порівнянь, хоч і зменшилася, але залишається $O(n^2)$, тоді як число обмінів не помінялося взагалі ніяк. Середня (вона ж гірша) кількість операцій залишається квадратичною.

Додаткова пам'ять, очевидно, не потрібна. Поведінка вдосконаленого (але не початкового) методу досить природна, майже відсортований масив буде відсортований набагато швидше випадкового. Сортування бульбашкою стійке, проте шейкер-сортування втрачає цю якість.

Сортування вибором

Ідея методу полягає в тому, щоб створювати відсортовану послідовність шляхом приєднання до неї одного елементу за іншим в правильному порядку.

Будуватимемо готову послідовність, починаючи з лівого кінця масиву. Алгоритм складається з n послідовних кроків, починаючи від нульового і закінчуючи (n-1) -м.

На i-му кроці вибираємо найменший з елементів a[i]... a[n] і міняємо його місцями з a[i]. Послідовність кроків при n=5 зображена на малюнку нижче.

Незалежно від номера поточного кроку i, послідовність a[0]...a[i] (виділена курсивом) є впорядкованою. Таким чином, на (n-1) -му кроці вся послідовність, окрім a[n] виявляється відсортованою, а a[n] стоїть на останньому місці по праву: всі менші елементи вже пішли вліво.

Сортування вибором - простий алгоритм, на практиці не використовуваний зважаючи на низьку швидкодію. Натомість застосовують її поліпшення - пірамідальне сортування (Heapsort), а також (іноді) 'деревне сортування' (TreeSort).

```
{ Сортуються записи типу item по ключу item.key }
{ для допоміжних змінних використовується тип index }

procedure SelectSort;
   var i, j, k: index; x:item;
begin for i:=1 to n-1 do
   begin k:=i; x:=a[i];
     for j:=i+1 to n do
     if a[j].key < x.key then
  begin k:=j; x:=a[j];
  end;
  a[k]:=a[i]; a[i]:=x;
  end;
end;
```

Для знаходження найменшого елементу з n+1 елементу алгоритм здійснює n порівнянь. З урахуванням того, що кількість елементів, що розглядаються на черговому кроці, зменшується на одиницю, загальна кількість операцій:

$n + (n-1) + (n-2) + (n-3) + ... + 1 = 1/2 * (n^2+n) = O(n^2)$.

Таким чином, оскільки число обмінів завжди буде менше числа порівнянь, час сортування росте квадратично щодо кількості елементів.

Алгоритм не використовує додаткової пам'яті: всі операції відбуваються "на місці".

Сортування вставками

Сортування простими вставками в чомусь схоже на вищевикладені методи.

Аналогічним чином робляться проходи по частині масиву, і аналогічним же чином на його початку "зростає" відсортована послідовність...

Проте в сортуванні бульбашкою або вибором можна було чітко заявити, що на i-му кроці елементи a[0]...a[i] стоять на правильних місцях і нікуди більш не перемістяться. Тут же подібне твердження буде слабкішим: послідовність a[0]...a[i] впорядкована. При цьому по ходу алгоритму в неї вставлятимуться все нові елементи.

Розбиратимемо алгоритм, розглядаючи його дії на i-му кроці. Як мовилося вище, послідовність до цього моменту розділена на дві частини: готову a[0]...a[i] і неврегульовану a[i+1]...a[n].

На наступному, (i+1) -му кроці алгоритму беремо a[i+1] і вставляємо на потрібне місце в готову частину масиву. Пошук відповідного місця для чергового елементу вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що стоїть перед ним. Залежно від результату порівняння елемент або залишається на поточному місці (вставка завершена), або вони міняються місцями і процес повторюється.

Таким чином, в процесі вставки ми "просіваємо" елемент x до початку масиву, зупиняючись у разі, коли

1. Найдено елемент, менший x або
2. Досягнуто початку послідовності.

Аналогічно сортуванню вибором, середнє, а також гірше число порівнянь і пересилок оцінюються як $O(n^2)$, додаткова пам'ять при цьому не використовується.

Хорошим показником сортування є вельми природна поведінка: майже відсортований масив буде досортовано дуже швидко. Це, укупі із стійкістю алгоритму, робить метод хорошим вибором у відповідних ситуаціях.

Порозрядне сортування

Алгоритм, що розглядається нижче, істотно відрізняється від описаних раніше.

По-перше, він зовсім не використовує порівнянь сортованих елементів.

По-друге, ключ, по якому відбувається сортування, необхідно розділити на частини, *розряди* ключа. Наприклад, слово можна розділити по буквах, число - по цифрах...

До сортування необхідно знати два параметри: k і m, де

- k - кількість розрядів в найдовшому ключі
- m - розрядність даних: кількість можливих значень розряду ключа

При сортуванні російських слів m = 33, оскільки буква може приймати не більше 33 значень. Якщо в найдовшому слові 10 букв, k = 10.

Аналогічно, для для шістнадцяткових чисел m=16, якщо як розряд брати цифру, і m=256, якщо використовувати побайтове ділення.

Ці параметри не можна змінювати в процесі роботи алгоритму. У цьому - ще одна відмінність методу від вищеописаних.

Припустимо, що елементи лінійного списку L є к-розрядні десяткові числа, розрядність максимального числа відома наперед. Позначимо d(j,n) - j-у справа цифру числа n

Хай L0, L1,..., L9 - допоміжні списки (кишені), спочатку порожні. Порозрядне сортування складається з двох процесів, званих розподіл і збірка і виконуваних для j=1,2,...,k.

Фаза розподілу розносить елементи L по кишенях: елементи li списку L послідовно додаються в списки Lm, де m = d(j, li). Таким чином одержуємо десять списків, в кожному з яких j-ті розряди чисел однакові і рівні m.

Фаза збірки полягає в об'єднанні списків L0, L1,..., L9 у загальний список

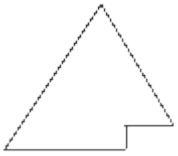
L = L0 => L1 => L2 => ... => L9

Пірамідальне сортування

Поступово переходимо від більш-менш простих до складних, але ефективних методів. Пірамідальне сортування одним з таких методів, швидкодія яких оцінюється як O(n log n).

Отже, назвемо пірамідою(Heap) бінарне дерево висоти k, в якому

- всі вузли мають глибину k або k-1 - дерево збалансоване.
- при цьому рівень k-1 повністю заповнений, а рівень k заповнений зліва направо, т.б форма піраміди має приблизно такий вигляд:



- виконується "властивість піраміди": кожен елемент менше, або рівний батькові.

Як зберігати піраміду? Найпростіше - помістити її в масив.

Відповідність між геометричною структурою піраміди як дерева і масивом встановлюється по наступній схемі:

- у a[0] зберігається корінь дерева
- лівий і правий сини елементу a[i] зберігаються, відповідно, в a[2i+1] і a[2i+2]

Таким чином, для масиву, що зберігає в собі піраміду, виконується наступна характеристична властивість: a[i]>= a[2i+1] і a[i]>= a[2i+2].

Плюси такого зберігання піраміди очевидні:

- ніяких додаткових змінних, потрібно лише розуміти схему.
- вузли зберігаються від вершини і далі вниз, рівень за рівнем.
- вузли одного рівня зберігаються в масиві зліва направо.

Відновити піраміду з масиву як геометричний об'єкт легко - досить пригадати схему зберігання і намалювати, починаючи від кореня.

Фаза 1 сортування: побудова піраміди

Почати побудову піраміди можна з a[k]...a[n], до = [size/2]. Ця частина масиву задовольняє властивості піраміди, оскільки не існує індексів i,j: i = 2i+1 (або j = 2i+2)... Просто тому, що такі i,j знаходяться за межею масиву.

Слід відмітити, що неправильно говорити про те, що a[k]..a[n] є пірамідою як самостійний масив. Це, взагалі кажучи, не вірно: його елементи можуть бути будь-якими. Властивість піраміди зберігається лише в рамках початкового, основного масиву a[0]...a[n].

Далі розширюватимемо частину масиву, що володіє такою корисною властивістю, додаючи по одному елементу за крок. Наступний елемент на кожному кроці додавання - той, який стоїть перед вже готовою частиною.

Щоб при додаванні елементу зберігалася пірамідальність, використовуватимемо наступну процедуру розширення піраміди a[i+1]..a[n] на елемент a[i] вліво:

1. Дивимося на синів зліва і справа - в масиві це a[2i+1] і a[2i+2] і вибираємо найбільшого з них.

2. Якщо цей елемент більше a[i] - міняємо його з a[i] місцями і йдемо до кроку 2, маючи на увазі нове положення a[i] в масиві. Інакше кінець процедури.

Новий елемент "просівається" крізь піраміду.

Фаза 2: власне сортування

Отже, завдання побудови піраміди з масиву успішно вирішене. Як видно з властивостей піраміди, в корені завжди знаходиться максимальний елемент. Звідси витікає алгоритм фази 2:

1. Беремо верхній елемент піраміди a[0]...a[n] (перший в масиві) і міняємо з останнім місцями. Тепер "забуваємо" про цей елемент і далі розглядаємо масив a[0]...a[n-1]. Для перетворення його в піраміду досить просіяти лише новий перший елемент.

2. Повторюємо крок 1, поки оброблювана частина масиву не зменшиться до одного елементу.

Очевидно, в кінець масиву кожного разу потрапляє максимальний елемент з поточної піраміди, тому в правій частині поступово виникає впорядкована послідовність.

```
procedure Heapsort;
 var i,l: index; x: item;

procedure sift;
 label 13;
 var i, j: index;
begin i := l; j := 2*i; x := a[i];
 while j <= r do
 begin if j < r then
  if a[j].key < a[j+1].key then j := j+1;
  if x.key >= a[j].key then goto 13;
  a[i]:= a[j]; i := j; j := 2*i
 end;
 13: a[i]:= x
 end;

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

  while r > 1 do
   begin x := a[l]; a[l]:= a[r]; a[r]:= x;
     r := r - 1; sift
 end
end { heapsort }
```

Як швидкодія алгоритму, що вийшов ?

Побудова піраміди займає O(n log n) операцій, причому точніша оцінка дає навіть O(n) за рахунок того, що реальний час виконання downheap залежить від висоти вже створеної частини піраміди.

Друга фаза займає O(n log n) часу: O(n) раз береться максимум і відбувається просіювання елементу що був останнім. Плюсом є стабільність методу: середнє число пересилок (n log n) /2, і відхилення від цього значення порівняно малі.

Пірамідальне сортування не використовує додаткової пам'яті.

Метод не є стійким: по ходу роботи масив так "перетрушується", що початковий порядок елементів може змінитися випадковим чином.

Поведінка неприродна: часткова впорядкованість масиву ніяк не враховується.

Швидке сортування

Удосконаливши метод сортування, заснований на обмінах, К.Хоар запропонував алгоритм QuickSort сортування масивів, що дає на практиці відмінні результати і дуже просто програмований. Автор назвав свій алгоритм швидким сортуванням.

Ідея К. Хоара полягає в наступному:

1 Виберемо деякий елемент x масиву A випадковим чином;

2.Проглядаємо масив в прямому напрямі (i = 1, 2...), шукаючи в ньому елемент A[i] не менший, ніж x;

3.Проглядаємо масив у зворотному напрямі (j = n n-1,..), шукаючи в ньому елемент A[j] не більший, ніж x;

4.Міняємо місцями A[i] і A[j];

Пункти 2-4 повторюємо до тих пір, поки i < j;

В результаті такого стрічного проходу початок масиву A[1..i] і кінець масиву A[j..n] виявляються розділеними “бар'єром” x: A[k] < x при k < i, A[k] > x при k > j, причому на розділення ми витратимо не більше n/2 перестановок. Тепер залишилося виконати ті ж дії з початком і кінцем масиву, тобто застосувати їх рекурсивно.

Таким чином, описана нами процедура Hoare залежить від параметрів k і m - початкового і кінцевого індексів оброблюваного відрізка масиву.

Методи поліпшення швидкого сортування.

1. Невеликі підфайли.

Перше поліпшення в алгоритмі швидкого сортування виникає із спостереження, що програма гарантовано викликає себе для величезної кількості невеликих підфайлів, тому слід використовувати найкращий метод сортування коли ми зустрічаємо невеликий підфайл. Очевидний спосіб добитися цього, це змінити перевірку на початку рекурсивної функції з "if r>l then" на виклик сортування вставкою (відповідно зміненої для сприйняття меж сортованого підфайлу): "if r-l<=M then insert(l, r)." Значення для M не зобов'язане бути "самим-самим" кращим: алгоритм працює приблизно однаково для M від 5 до 25. Час роботи програми при цьому знижується приблизно на 20% для більшості програм.

2. Ділення по медіані з трьох

Друге поліпшення в алгоритмі швидкого сортування полягає в спробі використання кращого ділячого елементу. У нас є декілька можливостей. Найбільш безпечною з них буде спроба уникнути гіршого випадку за допомогою вибору довільного елементу масиву як ділячого елементу. Тоді вірогідність гіршого випадку стає нехтівно малою. Це простий приклад "імовірнісного" алгоритму, який майже завжди працює незалежно від вхідних даних. Довільність може бути хорошим інструментом при розробці алгоритмів, особливо якщо можливі підозрілі вхідні дані.

Корисніше поліпшення полягає в тому, щоб узяти з файлу три елементи, і потім використовувати середнє з них як ділячий елемент. Якщо елементи узяті з початку, середини, і кінця файлу, то можна уникнути використання сторожових елементів: сортуємо узяті три елементи, потім обмінюємо центральний елемент з a[r-1], і потім використовуємо алгоритм ділення на масиві a[l+1..r-2]. Це поліпшення називається діленням по медіані з трьох.

3. Нерекурсивна реалізація.

Можна переписати даний алгоритм без використання рекурсії використовуючи стек.

Комбінація нерекурсивної реалізації ділення по медіані з трьох з відсіканням на невеликі файли може поліпшити час роботи алгоритму від 25% до 30%.

Злиття

Зовнішнє сортування сортує файли, які не поміщаються цілком в оперативну пам'ять.

Зовнішнє сортування сильно відрізняється від внутрішнього. Річ у тому, що доступ до файлу є послідовним, а не паралельним як це було в масиві. І тому прочитувати файл можна тільки блоками і цей блок відсортувати в пам'яті і знову записати у файл.

Принципову можливість ефективно відсортувати файл, працюючи з його частинами і не виходячи за межі частини забезпечує алгоритм злиття.

Під злиттям розуміється об'єднання двох (або більш) впорядкованих послідовностей в одну впорядковану послідовність за допомогою циклічного вибору елементів, доступних в даний момент.

Злиття - набагато простіша операція, ніж сортування.

Ми розглянемо 2 алгоритми злиття:

- Пряме злиття. Алгоритм Боуза - Нельсона
- Природне (Нейманівське) злиття.

Пряме злиття. Алгоритм Боуза - Нельсона

Послідовність a розбивається на дві половини b і c.

Послідовності b і c зливаються за допомогою об'єднання окремих елементів у впорядковані пари.

Одержаній послідовності привласнюється ім'я a, після чого повторюються кроки 1 і 2; при цьому впорядковані пари зливаються у впорядковані четвірки.

Попередні кроки повторюються, при цьому четвірки зливаються у вісімки і т.д., поки не буде впорядкована вся послідовність, оскільки довжини послідовностей кожного разу подвоюються.

Природне (Нейманівське) злиття.

Воно об'єднує впорядковані частини, що спонтанно виникли в початковому масиві; вони можуть бути також наслідком попередньої обробки даних. Розраховувати на однаковий розмір зливаних частин не доводиться.

Записи, що йдуть у порядку неубування ключів, зчіплюються, утворюючи підсписок. Мінімальний підсписок один запис.

Сортування двійковим деревом, 'деревне сортування'.

Двійковим (бінарним) деревом назвемо впорядковану структуру даних у якій кожному елементу - попереднику або кореню (під) дерева - поставлені у відповідність принаймні два інші елементи (наступника).

Причому для кожного попередника виконано наступне правило: лівий наступник завжди менше, а правий наступник завжди більше або рівний попереднику.

При додаванні в дерево нового елементу його послідовно порівнюють з нижчестоячими вузлами, таким чином вставляючи на місце.

Якщо елемент >= кореня - він йде в праве піддерево, порівнюємо його вже з правим сином, інакше - він йде в ліве піддерево, порівнюємо з лівим і так далі, поки є сини, з якими можна порівняти.

Якщо ми рекурсивно обходитимемо дерево за правилом "лівий син - батько - правий син", то, записуючи всі елементи, що зустрічаються, в масив, ми одержимо впорядковану у порядку зростання множину. На цьому і заснована ідея сортування деревом.

Завдання:

При виконанні завдання потрібно:

- сортувати множину, що складається не менш ніж з 100 елементів;
- оцінити складність виконаного алгоритму;
- скласти блок-схему результуючої програми.

Варіант 1

Реалізувати алгоритм TreeSort, застосувавши метод вкладення дерева в масив.

Варіант 2

Реалізувати ітеративну версію алгоритму пірамідального сортування.

Варіант 3

Реалізувати ітеративну версію алгоритму Хоара.

Варіант 4

Реалізувати алгоритм шейкерного сортування.

Варіант 5

Реалізувати алгоритм сортування вибором.

Варіант 6

Реалізувати алгоритм сортування вставками.

Варіант 7

Реалізувати рекурсивну версію алгоритму пірамідального сортування.

Варіант 8

Реалізувати рекурсивну версію алгоритму Хоара.

Варіант 9

Реалізувати модифікований алгоритм швидкого сортування з врахуванням сортування невеликих підмножин.

Варіант 10

Реалізувати модифікований алгоритм швидкого сортування з використанням ділення по медіані з трьох.

Варіант 11

Реалізувати алгоритм сортування прямим злиттям.

Варіант 12

Реалізувати алгоритм сортування природнім злиттям.

Варіант 13

Реалізувати ітеративний алгоритм швидкого сортування.

Варіант 14

Реалізувати алгоритм бульбашкового сортування.

Варіант 15

Реалізувати алгоритм порозрядного сортування

Контрольні питання:

1. Що таке зовнішнє сортування?
2. Що таке внутрішнє сорування?
3. Суть методу бульбашкового сортування?
4. В чому полягає особливість алгоритму шейкерного сортування?
5. Алгоритм швидкого сортування Хоара?
6. Суть сортування злиттям?
7. Властивості алгоритмів сортування?

Лабораторна робота № 4.

Тема: **“Алгоритми пошуку. Хешування”.**

Мета: Розлянути алгоритми пошуку, дослідити доцільність застосування різних алгоритмів в конкретних випадках виходячи з їх ефективності. Розглянути поняття хеш-функції, суть та цілі процесу хешування.

Теоретичні відомості:

Кажучи про пошук, ми матимемо на увазі якийсь масив даних, а шукати будемо певний елемент в цьому масиві. Оптимальність пошуку для простоти визначимо дуже конкретно - це швидкість роботи алгоритму.

Загальна класифікація алгоритмів пошуку

- Всі методи можна розділити на **статичні і динамічні**. При статичному пошуку масив значень не міняється під час роботи алгоритму. Під час динамічного пошуку масив може перебудовуватися або змінювати розмірність.

Спробуємо вирішити просте завдання - знайти потрібне значення в масиві.

Якщо немає ніякої додаткової інформації про розшукувані дані, то очевидний підхід простий послідовний перегляд масиву із збільшенням крок за кроком тієї його частини, де бажаного елементу не виявлено. Такий метод називається лінійним пошуком. Умови закінчення пошуку такі:

Елемент знайдений, тобто ai = x.

Весь масив проглянутий і збігу не виявлено.

Це дає нам лінійний алгоритм:

Алгоритм 1.

```
i:=0;
while (i<N) and (a[i]<>x) do i:=i+1
```

1. Алгоритм пошуку по бінарному дереву.

Суть цього алгоритму достатньо проста. Уявимо собі, що у нас є набір карток з телефонними номерами і адресами людей. Картки відсортовані у порядку зростання телефонних номерів. Необхідно знайти адресу людини, якщо ми знаємо, що її номер телефону 222-22-22. Наші дії? Беремо картку з середини пачки, номер картки 444-44-44. Порівнюючи його з шуканим номером, ми бачимо, що наш менше і, отже, знаходиться точно в першій половині пачки. Сміливо відкладаємо другу частину пачки убік, вона нам не потрібна, масив пошуку ми звузили рівно в два рази. Тепер беремо картку з середини пачки, що залишилася, на ній номер 123-45-67. З чого виходить, що потрібна нам картка лежить в другій половині пачки, що залишилася... Таким чином, при кожному порівнянні, ми зменшуємо зону пошуку в два рази. Звідси і назва методу - половинного ділення або дихотомії.

Швидкість збіжності цього алгоритму пропорційна Log(2)N. Це означає буквально те, що не більше, ніж через Log(2) N порівнянь, ми або знайдемо

потрібне значення, або переконаємося в його відсутності.

Ті, хто має нещастя часто працювати з текстовими редакторами, знають ціну функції знаходження потрібних слів в тексті, що істотно полегшує редагування документів і пошук потрібної інформації. Пани програмісти теж це знають, благо аж до самої компіляції програми їм доводиться працювати в текстовому редакторі.

Отже, хай у нас є текст, що складається з n символів, який надалі домовимося називати T, а T[i] його i-й символ. Рядок або просто слово, що складається з m символів, назвемо S, де S[i] -i-й символ рядка. Нам потрібно перевірити, чи входить даний рядок в даний текст, і якщо входить, то починаючи з якого символу тексту. розглянемо декілька відомих алгоритмів, вирішуючих поставлене завдання.

Простий алгоритм

Суть методу, про який піде мова нижче, полягає в наступному: ми перевіряємо, чи співпадають m символів тексту (починаючи з вибраного) з символами нашого рядка, намагаючись приміряти шаблон куди тільки можливо.

Наступний метод працює набагато швидше простого, але, на жаль, накладає деякі обмеження на текст і шуканий рядок.

Алгоритм Рабіна-Карпа.

Ідея, запропонована Рабіном і Карпом, має на увазі поставити у відповідність кожному рядку деяке унікальне число, і замість того щоб порівнювати самі рядки, порівнювати числа, що набагато швидше. Проблема в тому, що шуканий рядок може бути довгим, рядків в тексті теж вистачає. А оскільки кожному рядку потрібно зіставити унікальне число, то і чисел повинно бути багато, а отже, числа будуть великими (порядку Dm, де D - кількість різних символів), і працювати з ними буде так само незручно.

Ще один важливий метод - алгоритм Кнута-Морріса-Пратта, один з кращих на нинішній момент, працює за лінійний час для будь-якого тексту і будь-якого рядка.

Алгоритм Кнута-Морріса-Пратта

Метод використовує передобробку шуканого рядка, а саме: на її основі створюється так звана *префікс-функція*. Суть цієї функції в знаходженні для кожного підрядка S[1..i] рядка S найбільшого підрядка S[1..j] (j<i), присутнього одночасно і на початку, і в кінці підрядка (як префікс і як суфікс). Наприклад, для підрядка abcHelloabc таким підрядком є abc (одночасно і префіксом, і суфіксом). Сенс функції префікса в тому, що ми можемо відкинути свідомо невірні варіанти, тобто якщо при пошуку співпав підрядок abcHelloabc (наступний символ не співпав), то нам має сенс продовжувати перевірку продовжити пошук вже з четвертого символу (перші три і так співпадуть).

Хешування

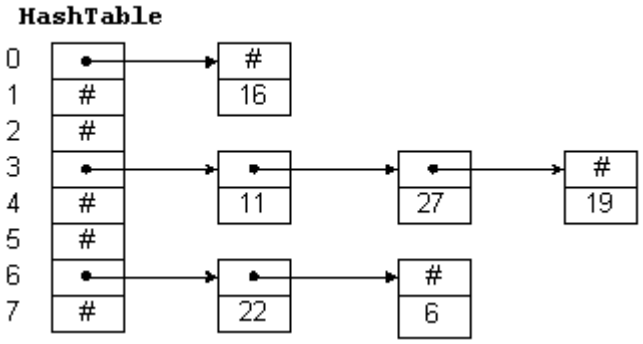
З хешуванням ми стикаємося чи не на кожному кроці: при роботі з браузером (список Web-посилань), текстовим редактором і перекладачем (словник), мовами скриптів (Perl, Python, PHP і ін.), компілятором (таблиця

символів). За словами Брайана Кернігана, це «один з найбільших винаходів інформатики». Заглядаючи в адресну книгу, енциклопедію, алфавітний покажчик, ми навіть не замислюємося, що впорядкування за абеткою є не чим іншим, як хешуванням. Хешування є розбиття безлічі ключів (що однозначно характеризують елементи зберігання і представлених, як правило, у вигляді текстових рядків або чисел) на непересічні підмножини (набори елементів), що володіють певною властивістю. Ця властивість описується функцією хешування, або хеш-функцією, і називається хеш-адресою. Рішення зворотної задачі покладене на хеш-структури (хеш-таблиці): по хеш-адресі вони забезпечують швидкий доступ до потрібного елементу. У ідеалі для завдань пошуку хеш-адреса повинен бути унікальним, щоб за одне звернення дістати доступ до елементу, що характеризується заданим ключем (ідеальна хеш-функція). Проте, на практиці ідеал доводиться замінювати компромісом і виходити з того, що набори, що виходять, з однаковою хеш-адресою містять більше одного елементу. Термін «хешування» (hashing) в роботах по програмуванню з'явився порівняно недавно (1967 р.), хоча сам механізм був відомий і раніше. Дієслово «hash» в англійській мові означає «рубати, кришити».

Хеш-функції

Хеш-функція – це деяка функція h(K), яка бере якийсь ключ K і повертає адресу, по якій проводиться пошук в хеш-таблиці, щоб одержати інформацію, пов'язану з K. Наприклад, K – це номер телефону абонента, а шукана інформація – його ім'я. Функція в даному випадку нам точно скаже, за якою адресою знайти шукане.

Наприклад, на **hashTable** мал. 1 - це масив з 8 елементів. Кожен елемент є покажчиком на лінійний список, що зберігає числа. Хеш-функція в даному прикладі просто ділить ключ на 8 і використовує залишок як індекс в таблиці. Це дає нам числа від 0 до 7. Оскільки для адресації в hashTable нам і потрібні числа від 0 до 7, алгоритм гарантує допустимі значення індексів.



Мал. 1: Хеш-таблиця

Щоб вставити в таблицю новий елемент, ми хешируем ключ, щоб визначити список, в який його потрібно додати, потім вставляємо елемент в початок цього списку. Наприклад, щоб додати 11, ми ділимо 11 на 8 і одержуємо залишок 3. Таким чином, 11 слід розмістити в списку, на початок якого указує **hashTable[3]**. Щоб знайти число, ми його хешируем і проходимо за відповідним списком. Щоб видалити число, ми знаходимо його

і видаляємо елемент списку, його що містить.

Колізія – це ситуація, коли h(K1)= h(K2), тоді як K1 <> K2. В цьому випадку, очевидно, необхідно знайти нове місце для зберігання даних. Очевидно, що кількість колізій необхідно мінімізувати. Хороша хеш-функція повинна задовольняти двом вимогам:

- її обчислення повинне виконуватися дуже швидко;
- вона повинна мінімізувати число колізій.

Завдання.

Варіант 1.

Реалізувати лінійний пошук елемента в масиві.

Варіант 2.

Реалізувати двійковий пошук елемента в масиві.

Варіант 3.

Реалізувати пошук елемента в бінарному дереві пошуку.

Варіант 4.

Реалізувати простий алгоритм пошуку слова в тексті.

Варіант 5.

Реалізувати алгоритм Рабіна-Карпа пошуку слова в тексті для коротких рядків.

Варіант 6.

Реалізувати поліпшений алгоритм Рабіна-Карпа пошуку слова в тексті.

Варіант 7.

Реалізувати алгоритм Кнута-Моріса-Пратта пошуку слова в тексті.

Варіант 8.

Реалізувати алгоритм Боуєра-Мура пошуку слова в тексті.

Варіант 9.

Написати програму, що організовує дані у вигляді хеш-таблиці, здійснює додавання нового елемента в таблицю, вилучення елемента з таблиці та пошук заданого елемента.

Варіант 10.

Реалізувати лінійний пошук елемента в масиві.

Варіант 11.

Реалізувати двійковий пошук елемента в масиві.

Варіант 12.

Реалізувати пошук елемента в бінарному дереві пошуку.

Варіант 13.

Реалізувати простий алгоритм пошуку слова в тексті.

Варіант 14.

Реалізувати алгоритм Рабіна-Карпа пошуку слова в тексті для коротких рядків.

Варіант 15.

Реалізувати поліпшений алгоритм Рабіна-Карпа пошуку слова в тексті.

Контрольні питання:

1. Які алорими пошуку ви знаєте?

2. Що таке хеш-функція?
3. В чому полягає алгоритм хешування?
4. Що таке колізія?
5. Які існують методи розв'язання колізій?

Лабораторна робота 5. **Дерева. Побудова бінарного дерева та реалізація основних операцій над його елементами.**

1. Дерева: основні поняття

Визначення 1. Дерево - це зв'язний ациклічний граф. Граф, що не містить циклів, називається лісом (ясно, що компонентами лісу є дерева). Якщо зафіксовано деяку вершину ao, то вона називається коренем дерева, а саме дерево називається деревом з коренем. Відомі й інші визначення поняття "дерево", які неявно містяться в наведеній нижче теоремі. Для графа G, що має p вершин й q ребер, наступні твердження еквівалентні: (1) граф G - дерево; (2) будь-які дві вершини в графі G з'єднані єдиним ланцюгом; (3) граф G - зв'язний граф й p=q+1; (4) граф G - ацикллічний граф й p=q+1; (5) граф G - ацикллічний граф, і якщо будь-яку пару несуміжних вершин з'єднати ребром x, то в графі G+x буде точно один цикл; (6) граф G - зв'язний граф, відмінний від K_p для p□3, і якщо будь-яку пару несуміжних вершин з'єднати ребром x, то в графі G+x буде точно один цикл. (7) граф G - граф, відмінний від K_3□K_1 й K_3□K_2, p=q+1, і якщо будь-яку пару несуміжних вершин з'єднати ребром x, то в графі G+x буде точно один цикл.Приведемо два очевидних наслідки даної теореми: (1) у будь-якому нетривіальному дереві є, принаймні, дві висячі вершини; (2) кожне дерево з n вершинами має рівно n-1 ребро.

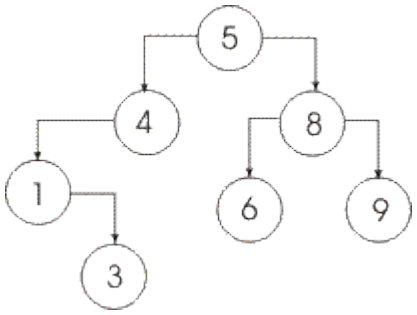
Визначення 2. Неорієнтований граф, що виходить у результаті "зняття орієнтації з дуг" орієнтованого графа G, називається лежачим в основі неорієнтованого графа G. Орієнтований граф G називається деревом, якщо лежачий у його основі неорієнтований граф також є деревом. Орієнтований граф G називається орієнтованим (кореневим) деревом, якщо він є деревом і має корінь. Піддеревом кореневого дерева T називається його частина T', що є кореневим деревом, що задовольняє наступній умові: разом з коренем v' дерево T' містить всі вершини й дуги, досяжні в T з v'. Вершини графа G з нульовим напівступенем виходу (іншими словами, що не мають наступників), називаються листами.

Кореневе дерево, як випливає з визначення, задовольняє наступним умовам: (а) є рівно одна вершина, називана коренем, у яку не входить жодна дуга; (б) у кожну вершину, відмінну від кореня, входить рівно одна дуга; (в) наступники всякої вершини впорядковані. У зв'язку із цим, має сенс наступне визначення.

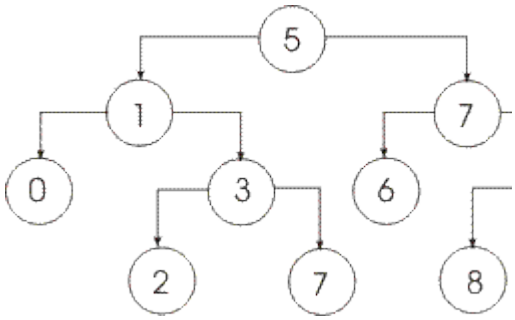
Визначення 3. Орієнтований скінчений граф (V,E) є прадерево з коренем x_1□V (кореневе дерево з коренем x_1□V), якщо: (1) у кожну вершину, відмінну від вершини x_1, заходить єдина дуга; (2) у вершину x_1 не заходить жодна дуга; (3) граф (V,E) не містить контурів. (Легко уявити собі зовнішній вигляд прадерева: це дерево, кожне ребро якого однозначно орієнтовано, і для будь-якої вершини x якого існує шлях від x_1 до x).

Визначення 4. Двійкове дерево - це кореневе дерево, у якому кожна вершина має не більше двох спадкоємців, називаних лівими й правими наступниками (синами).

2-3-дерево - це кореневе дерево, у якому відстані від кореня до всіх листів збігаються, а кожна вершина, відмінна від листа, має двох або трьох спадкоємців. Деревом двійкового пошуку для множини чисел S називається розмічене двійкове дерево, у якому кожна вершина v позначена числом l(v)OS, і яке задовольняє наступним умовам: (а) l(u)<l(v) для всіх вершин u, v, якщо вершина u перебуває в піддереві, корінь якого - лівий спадкоємець v; (б) l(u)>l(v) для всіх вершин u, v, якщо вершина u перебуває в піддереві, корінь якого правий спадкоємець v; (в) для всякого A, що належить S, існує єдина вершина v, для якої l(v)=A. Позначимо через r(T) максимум серед відстаней від кореня дерева T до його листів. АВЛ-деревом називається дерево двійкового пошуку, у якому для кожної вершини v виконана наступна умова: $|r(T_1)-r(T_2)|\leq 1$, де v_1,v_2 - наступники v, а T_1,T_2 - піддерева з коренями v_1,v_2. Приклади див.на малюнку: (а) дерево двійкового пошуку; (б) Авл-дерево.



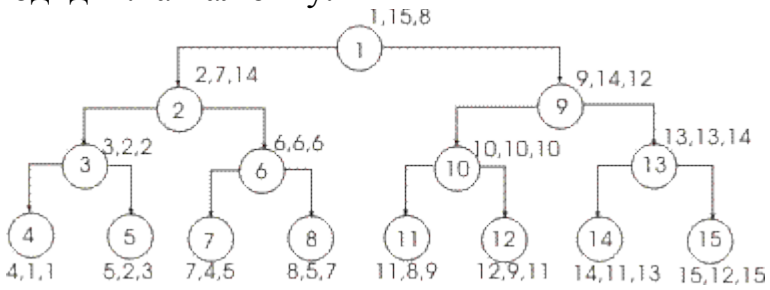
(а)



(б)

Визначення 5. Обхід кореневого дерева T складається у відвідуванні вершин цього дерева в деякому порядку. Прямий обхід (П-обход) кореневого дерева T описується так: (1) відвідати корінь v; (2) виконати П-обхід піддерев з коренями $v_1,...,v_n$ у зазначеній послідовності. Зворотний обхід (З-обхід) кореневого дерева T описується так: (1) виконати З-обхід піддерев з коріннями $v_1,...,v_n$ у зазначеній послідовності; (2) відвідати корінь v. Обхід у внутрішньому порядку (В-обхід або симетричний) кореневого дерева T визначається так: (1) виконати В-обхід лівого піддерева кореня дерева T; (2) відвідати корінь v дерева T; (3) виконати В-обхід правого піддерева кореня дерева T.

Приклад вказання порядку обходу вершин двійкового дерева при П-, З и В-обході див.на малюнку:



Процедура обходу бінарного дерева в прямому порядку

```
procedure preorder(var p:ukaz);
begin
write(p^.info);
if p^.left<>nil then preorder(p^.left);
if p^.right<>nil then preorder(p^.right);
end;
```

Нерекурсивний алгоритм обходу бінарного дерева в прямому порядку

t – вказівник на вершину бінарного дерева, A – стек, top – вершина стека, p – робоча змінна.

1. top:=0; p:=t;
2. Якщо p=nil then goto 4;
3. writeln(p^.info); top:=top+1; A[top]:=p; p:=p^.left; goto 2;
4. Якщо top=0 then кінець;
5. Дістати вершину з стека: p:=A[top]; top:=top-1; p:=p^.right; goto 2.

Процедура обходу бінарного дерева в симетричному порядку

```
procedure sumorder(var p:ukaz);
begin
if p^.left<>nil then sumorder(p^.left);
write(p^.info);
if p^.right<>nil then sumorder(p^.right);
end;
```

Процедура обходу бінарного дерева в оберненому порядку

```
procedure postorder(var p:ukaz);
begin
if p^.left<>nil then postorder(p^.left);
if p^.right<>nil then postorder(p^.right);
write(p^.info);
end;
```

Генерація дерева синтаксичного аналізу

Один і той же арифметичний вираз може бути записаний трьома способами:

1. Інфіксний спосіб запису (знак операції знаходиться між операндами):

((a / (b + c)) + (x * (y - z)))

Всі арифметичні операції, звичні нам з шкільних років, записуються саме таким чином.

2. Префіксний спосіб запису (знак операції знаходиться перед операндами):

+(/(a +(b,c)), *(x, -(y,z)))

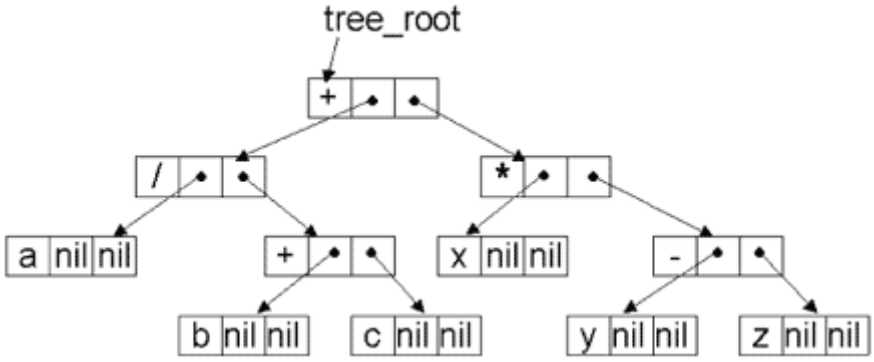
Із знайомих всім нам функцій префіксний спосіб запису використовується, наприклад, для sin(x), tg(x), f(x,y,z) і т.п.

3. Постфіксний спосіб запису (знак операції знаходиться після операндів):

((a(b,c)+)/,(x,(y,z) -)*)+

Цей спосіб запису менш поширений, проте і з ним багатьом з нас доводилося стикатися вже в школі: прикладом буде n! (факторіал).

Зрозуміло, вид дерева синтаксичного аналізу (ДСА) арифметичного виразу не залежить від способу запису цього виразу, оскільки його визначає не форма запису, а порядок виконання операцій. Але процес побудови дерева, звичайно ж, залежить від способу запису виразу. Далі ми розберемо всі три варіанти алгоритму побудови ДСА.



Мал. 1. Дерево синтаксичного аналізу і спосіб опису його елементів

```
type ukaz = ^tree;
    tree = record
            symbol: char;
            left: ukaz;
            right: ukaz;
     end;
```

Побудова з інфіксного запису

Для простоти ми вважатимемо, що правильний арифметичний вираз подається в одному рядку, без пропусків, а кожен операнд записаний однією буквою. Пріоритет операцій визначається розставленими дужками: ними повинна бути забезпечена кожна операція.

Алгоритм Infix

Якщо не досягнутий кінець рядка введення, прочитати черговий символ.

Якщо цей символ - відкриваюча дужка, то:

1. створити нову вершину дерева;
2. викликати **алгоритм Infix** для її лівого поддерева;
3. прочитати символ операції і занести його в поточну вершину;
4. викликати **алгоритм Infix** для правого поддерева;
5. прочитати символ закриваючої дужки (і нічого з ним не робити).

Інакше:

1. створити нову вершину дерева;
2. занести в неї цей символ.

Реалізація

```
procedure infix(var p: ukaz);
var c: char;
begin
  read(c);
  if c = '('
       then begin
         new(p);
         infix(p^.left);
         read(p^.symbol);          {'+', '-', '*', '/'}
         infix(p^.right);
         read(c);           {')'}
          end
     else begin   {'a'..'z','A'..'Z'}
       new(p);
       p^.symbol:= c;
       p^.right:= nil;
       p^.left:= nil
     end;
end;

begin
  ...
  infix(root);
  ...
end.
```

Побудова з префіксного запису

Для простоти припустимо, що правильний арифметичний вираз подається в одному рядку, без пропусків, а кожен операнд записаний однією буквою. Крім того, вважатимемо, що із запису видалені всі дужки: це цілком допустимо, оскільки операція завжди передує своїм операндам, отже, плутанина у порядку виконання неможлива.

Алгоритм Prefix

1. Якщо не досягнутий кінець рядка введення, прочитати черговий символ.
2. Створити нову вершину дерева, записати в неї цей символ.
3. Якщо символ - операція, то:

1. викликати **алгоритм Prefix** для лівого поддерева;
2. викликати **алгоритм Prefix** для правого поддерева.

Реалізація

```
procedure prefix(var p: ukaz);
begin
  new(p);
  read(p^.symbol);
  if p^.symbol in ['+','-','*','/']
```

```
      then begin prefix(p^.left);
                 prefix(p^.right);
           end
      else begin p^.left:= nil;
                 p^.right:= nil;
           end
end;

begin
  ...
  prefix(root);
  ...
end.
```

Побудова з постфіксного запису

Для простоти припустимо, що правильний арифметичний вираз подається в одному рядку, без пропусків, а кожен операнд записаний однією буквою. Крім того, знову вважатимемо, що із запису видалені всі дужки.

Алгоритм Postfix

1. Якщо не досягнутий кінець рядка введення, прочитати черговий символ, якщо цей символ - операнд, то занести його в стек, інакше (символ - операція):
1. створити новий елемент, записати в нього цю операцію;
2. дістати із стека два верхні (останніх) елементи, приєднати їх як лівий і правий операнди в новий елемент;
3. занести одержаний "трикутник" в стек.

Після закінчення роботи цього алгоритму в стеку міститиметься рівно один елемент - покажчик на корінь побудованого дерева.

Реалізація

Для того, щоб спростити роботу, додамо в структуру елементу дерева додаткове поле next:ukaz, яке служитиме для зв'язки стека:

```
stek:= nil;
while not eof(f) do
  begin
   new(p);
   read(f,p^.symbol);
   if p^.symbol in ['+','-','*','/']
     then begin
       p^.right:= stek;
       p^.left:= stek^.next;
       p^.next:= stek^.next^.next;
       stek:= p
     end
   else begin
     p^.left:= nil;
     p^.right:= nil;
```

```
      p^.next:= stek;
      stek:= p
    end;
  end;
```

Деревне сортування

Завдання. Упорядкувати заданий набір (можливо, з повтореннями) деяких елементів (чисел, слів, и т.п.).

Алгоритм TreeSort

1. Для сортованої безлічі елементів побудувати дерево двійкового пошуку:
   - перший елемент занести в корінь дерева;
   - для всієї решти елементів: почати перевірку з кореня; рухатися вліво (якщо даний елемент менший ніж поточна вершина дерева) або управо (якщо даний елемент більший ніж поточна вершина дерева) до тих пір, поки не зустрінеться такий же елемент, або поки не зустрінеться nil. У другому випадку потрібно створити новий лист в дереві, куди і буде записане значення нового елементу.
2. Зробити синтаксичний (симетричний) обхід побудованого дерева, друкуючи кожну зустрінуту вершину стільки разів, скільки було її входжень в сортований набір.

Завдання

У всіх варіантах 1 завдання стосується дерев, обходів дерев, 2 - остовів.

Варіант 1.

1. Напишіть процедуру обходу у внутрішньому (симетричному) порядку кореневого дерева.
2. Напишіть програму, яка будує впорядковане бінарне дерево та виконує в ньому операцію пошуку даного елемента.

Варіант2.

1. Перевірте на декількох деревах, що при додаванні до дерева нового ребра ми одержуємо рівно один цикл.
2. Напишіть програму, що дозволяє переводити арифметичний вираз з інфіксної форми запису в префіксну.

Варіант 3.

1. Визначте, чи є заданий граф бінарним деревом пошуку.
2. Напишіть нерекурсивну процедуру обходу бінарного дерева в прямому порядку.

Варіант 4.

1. Визначте, чи є заданий граф Авл-деревом.
2. Напишіть програму, яка будує впорядковане бінарне дерево та виконує в ньому операцію пошуку дублікатів (елементів, які зустрічались декілька разів в початковому масиві).

Варіант 5.

1. Визначте, чи є заданий граф 2-3-деревом.

2. Напишіть програму, що дозволяє переводити арифметичний вираз з постфіксної форми запису в інфіксну.

Варіант 6.

1. Спосіб обходу дерева завширшки, називаний іноді обходом у горизонтальному порядку, заснований на відвідуванні вершин дерева ліворуч праворуч, рівень за рівнем униз від кореня. Наведений нижче нерекурсивный алгоритм дозволяє зробити обхід дерева завширшки, використовуючи дві черги O1 й O2:

```
Взяти порожні черги O1 й O2.
Помістити корінь у чергу O1.
While Одна із черг O1 й O2 не порожня do
If O1 не є порожній
then begin
  Нехай p - вузол, що перебуває в голові черги O1.
  Відвідати вершину p і видалити її з O1.
  Помістити всіх синів вершини p у чергу O2,
  починаючи зі старшого сина
end else У якості O1 взяти непусту чергу O2,
а в якості O2 взяти порожню чергу O1.
```

Реалізуйте описаний алгоритм мовою Pascal

2. Напишіть програму, яка будує впорядковане бінарне дерево та реалізує операцію виведення всіх елементів даного дерева, менших ніж заданий (без перебору всіх без винятку елементів дерева).

Варіант 7.

1. Напишіть процедуру прямого обходу кореневого дерева.
2. Напишіть програму, яка будує бінарне дерево та реалізовує для нього операцію, яка дозволяє визначити, являється він лівим чи правим сином.

Варіант 8.

1. Напишіть процедуру зворотного обходу кореневого дерева.
2. Напишіть програму, що дозволяє переводити арифметичний вираз з префіксної форми запису в постфіксну.

Варіант 9.

1. Відомо, що заданий граф - не дерево. Перевірте, чи можна видалити з нього одну вершину (разом з інцидентними їй ребрами), щоб у результаті вийшло дерево.
2. Напишіть програму сортування масиву чисел за допомогою бінарного дерева

Варіант 10.

1. Перевірте на декількох деревах, що дерево, що має n вершин, є зв'язним і має n-1 ребро.

2. Напишіть програму, яка будує впорядковане бінарне дерево та реалізує операцію виведення всіх елементів даного дерева, більших ніж заданий (без перебору всіх без винятку елементів дерева).

Варіант 11.

1. Визначте, чи є заданий граф бінарним деревом.
2. Напишіть програму, яка будує бінарне дерево та виводить на екран всі листки (тобто висячі вершини) даного дерева.

Варіант 12.

1. Визначте, чи є заданий граф деревом.
2. Напишіть програму, яка дозволяє підраховувати значення арифметичного виразу, записаного в постфіксній формі (використовуючи стек).

Варіант 13.

1. Напишіть процедуру обходу у внутрішньому порядку кореневого дерева.
2. Напишіть програму, яка дозволяє переводити арифметичний вираз з постфіксної форми в префіксну.

Варіант 14.

1. Напишіть процедуру зворотного обходу кореневого дерева.
2. Напишіть програму, яка будує бінарне дерево та реалізовує для нього операції, що дозволяють знаходити для заданого елемента його правого, лівого сина та попередника.

Варіант 15.

1. Напишіть процедуру прямого обходу кореневого дерева.
2. Напишіть програму, яка будує бінарне дерево та виводить на екран всі внутрішні вершини даного дерева.

Контрольні питання

1. Що таке дерево?
2. Яке дерево називається двійковим?
3. Які способи обходу кореневого дерева ви знаєте?
4. Що таке остов?
5. Як побудувати остов найменшої вартості?

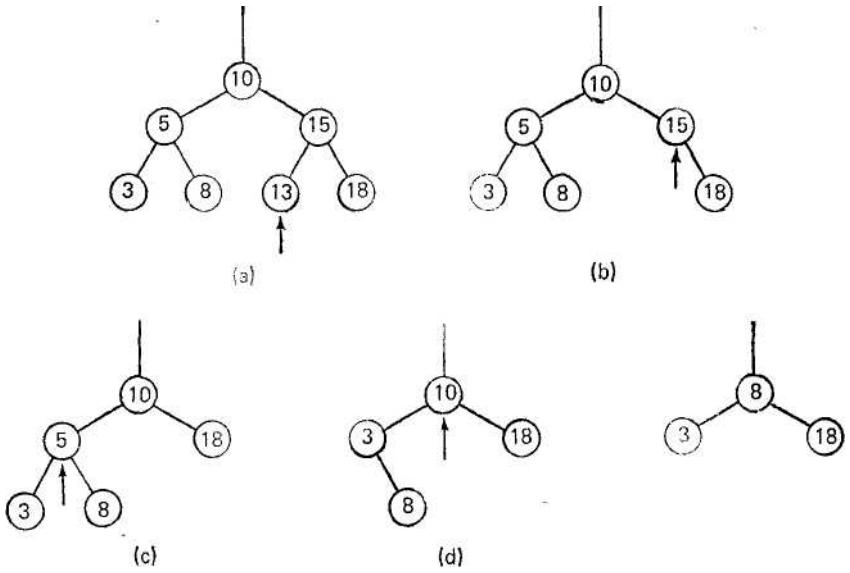
Лабораторна робота 6. **Дерева. Видалення елементів з бінарного дерева. Невузлове представлення бінарних дерев, збалансовані бінарні дерева, дерево з випадковим пошуком.**

Теоретичні відомості.

1. Видалення елементу з бінарного дерева.

Видалення елемента зазвичай виконується не так просто, як додавання нового елемента. Воно просто виконується у разі, коли елемент, що видаляється, є термінальним вузлом (тобто листом, не має нащадків) або має одного нащадка. Трудність полягає у видаленні елементів з двома нащадками, оскільки ми не можемо указувати одним посиланням на два напрями. Елемент, що в цьому випадку видаляється, потрібно замінити або на найбільший (найправіший) елемент його лівого піддерева, або на найменший (найлівіший) елемент його правого піддерева. Ясно, що такі елементи не можуть мати більше одного нащадка.

Задане початкове дерево (а), з якого послідовно віддаляються вузли з ключами 13, 15, 5, 10.



2. Невузлове представлення бінарного дерева.

Наряду з представленням бінарних дерев у вигляді динамічних структур даних використовується і представлення бінарних дерев у вигляді масивів. Таке невузлове представлення бінарного дерева можливо отримати шляхом нумерації вузлів таким чином, що номер назначений лівому сину дорівнює подвоєному номеру, присвоєному його батьку, а номер назначений правому синові на одиницю перевищує номер лівого сина. Тоді кожен вузол з номером p буде батьком вузлів 2p та 2p+1, а дерево з n-вершинами буде представлене масивом info розмірності 2n-1. Корінь дерева розташується в 1-му елементі масиву, довільний вузол в p-му елементі, а його сини, відповідно, в 2p-му та 2p+1-му елементах масиву.

3. Збалансовані бінарні дерева.

Реальний час виконання операцій з структурою, представленою у формі бінарного дерева залежить від форми дерева. Якщо воно схоже на список - маємо O(n), як в списку, якщо на дерево - то O(log n). Можна довести, що при послідовному включенні в дерево випадкових даних виходить саме середній час виконання операцій. Проте, якщо вхідні дані, наприклад, утворюють зростаючу послідовність, то виходить список і вся перевага дерева втрачена.

На практиці двійкові дерева пошуку дуже зручні в багатьох застосуваннях, де вхідні дані випадкові або є інші причини сподіватися на збереження деревом хорошої форми. Існують способи добитися часу O(log n) на будь-яких входах, вони корисні, але дають невеликі накладні витрати на зберігання спеціальної інформації.

Очевидно, пошук здійснюється тим швидше, чим ближче до кореня розташований шуканий елемент. У ідеалі двійкове дерево пошуку *збалансоване*, тобто його форма така, що кожен елемент розташовується порівняно близько до кореня. Збалансоване двійкове дерево пошуку розміру n має висоту O(log n), в припущенні, що шлях від кореня до кожного вузла має довжину більш, чим O(log n). Але зовсім незбалансоване дерево пошуку має висоту O(n); пошук в такому дереві так само неефективний, як в зв'язаному списку.

Висота дерева h(i) з кореневою вершиною **i** вимірюється $\max(l_{i,1}, l_{i,2},..., l_{i,n})$ – максимальною відстанню до інших вершин.

Аналогічно вимірюється висота лівого і правого піддерев $h(iD_L)$ і $h(iD_R)$.

Дисбалансом дерева δ_i з кореневою вершиною **i** називається різниця $|h(iD_L) - h(iD_R)|$

$$\delta_i = |h(iD_L) - h(iD_R)|$$

Якщо $h(iD_L) > h(iD_R) = \delta_L$ – дисбаланс вліво.

Якщо $h(iD_R) > h(iD_L) = \delta_R$ – дисбаланс управо.

Збалансовані дерева

БД називається **збалансованим**, якщо для кожного його вузла *x* значення дисбалансу менше або рівне певної константи.

Ідеально збалансовані БД

БД **ідеально збалансоване**, якщо для кожного його вузла *x* кількість вузлів в його правому і лівому піддеревах *xDL* і *xDR* розрізняються не більше, ніж на 1.

$\delta(x) = |\alpha(xD_L) - \beta(xD_R)| \le 1$ де α і β кількість вузлів.

Дерева з випадковим пошуком

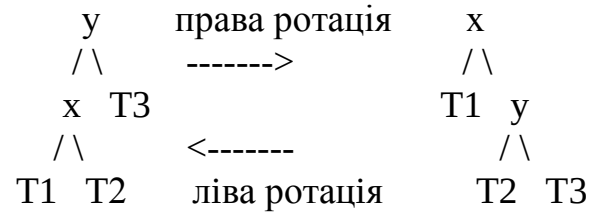
Кожному вузлу ставиться у відповідність пріоритет - випадкове число, що довизначає місце елементу додатково до значення самого елемента. Структура також забезпечує швидку операцію знаходження елементу.

Ідея розгляду дерев випадкового пошуку полягає в тому, щоб позбавитися залежності форми дерева від нових елементів, шляхом використання генератора випадкових чисел. При введенні нового елемента в дерево цьому елементу надається приорітет - певне випадкове дійсне чісло A з проміжку [0,1], 0<=A<=1.

Приорітети елементів в дереві випадкового пошуку визначають їх положення у відповідності з правилом:

для кожного елемента x (як кореня піддерева) всі вузли лівого піддерева будуть менші ніж x, а всі вузли правого піддеревабудуть більші ніж x.

Припустимо, що елемент x є лівим нащадком для y. Тоді виконаємо праву ротацію для елемента y, шляхом розміщення елементу x на один рівень вище. Наприклад:



В отриманому дереві можливе порушення правила приорітету для "нового" предка елемента x. Якщо це так, то застосуємо ротацію для цього предка, а саме:

якщо x є правим нащадком, то ліву ротацію предка;

якщо x є лівим нащадком, то праву ротацію.

Таким чином елемент x знову підніметься на один рівень вгору. Це триватиме доти, доки незнайдено вузол із меншим (ніж у x) пріоритетом.

Червоно-чорні дерева

Червоно-чорні дерева - один із способів балансування дерев. Назва походить від стандартного розфарбовування вузлів таких дерев в червоний і чорний кольори. Кольори вузлів використовуються при балансуванні дерева. Під час операцій вставки і видалення піддерева може знадобитися повернути, щоб досягти збалансованості дерева. Оцінкою як середнього часу, так і найгіршого є O(log n).

Червоно-чорне дерево - це бінарне дерево з наступними властивостями:

- Кожен вузол пофарбований або в чорний, або в червоний колір.
- Листям оголошуються **NIL-вузли (**тобто "віртуальні" вузли, спадкоємці вузлів, які звичайно називають листям; на них "указують" NULL покажчики). Листя пофарбоване в чорний колір.
- Якщо вузол червоний, то обидва його нащадка чорні.
- На всіх гілках дерева, що ведуть від його кореня до листя, число чорних вузлів однаково.

Кількість чорних вузлів на гілці від кореня до листа називається чорною висотою дерева. Перераховані властивості гарантують, що щонайдовша гілка від кореня до листа не більше ніж удвічі довше за будь-яку іншу гілку від кореня до листа. Щоб зрозуміти, чому це так, розглянемо дерево з чорною висотою 2. Найкоротша можлива відстань від кореня до листа рівна двом - коли обидва вузли чорні. Довга відстань від кореня до листа рівна чотирьом - вузли при цьому пофарбовані (від кореня до листа) так: червоний, чорний, червоний, чорний. Сюди не можна додати чорні вузли, оскільки при цьому порушиться властивість 4, з якої витікає коректність поняття чорної висоти. Оскільки згідно властивості 3 у червоних вузлів неодмінно чорні спадкоємці, в подібній послідовності недопустимі і два червоні вузли підряд. Таким чином, довгий шлях, який ми можемо сконструювати, складається з чергування червоних і чорних вузлів, що і приводить нас до подвоєної довжини шляху, що проходить тільки через чорні вузли. Всі операції над деревом повинні уміти працювати з перерахованими властивостями. Зокрема, при вставці і видаленні ці властивості повинні зберегтися.

Завдання

Варіант 1.

3. Написати програму, що видаляє заданий елемент з бінарного дерева.

Варіант2.

3. Написати програму, що моделює бінарне дерево за допомогою масиву і виконує операцію пошуку заданого елемента.

Варіант 3.

3. Написати програму, що моделює бінарне дерево за допомогою масиву і виконує операцію додавання заданого елемента.

Варіант 4.

3. Написати програму, що моделює бінарне дерево за допомогою масиву і виконує операцію видалення заданого елемента.

Варіант 5.

3. Написати програму побудови дерева з випадковим пошуком.

Варіант 6.

1. Написати програму побудови ідеально збалансованого невпорядкованого бінарного дерева.

Варіант 7.

3. Написати програму побудови впорядкованого збалансованого бінарного дерева.

Варіант 8.

3. Написати програму, що видаляє заданий елемент з бінарного дерева.

Варіант 9.

3. Написати програму, що моделює бінарне дерево за допомогою масиву і виконує операцію пошуку заданого елемента.

Варіант 10.

3. Написати програму, що моделює бінарне дерево за допомогою масиву і виконує операцію додавання заданого елемента.

Варіант 11.

3. Написати програму, що моделює бінарне дерево за допомогою масиву і виконує операцію видалення заданого елемента.

Варіант 12.

3. Написати програму побудови дерева з випадковим пошуком.

Варіант 13.

3. Написати програму побудови ідеально збалансованого невпорядкованого бінарного дерева.

Варіант 14.

3. Написати програму побудови впорядкованого збалансованого бінарного дерева.

Варіант 15.

3. Написати програму, що видаляє заданий елемент з бінарного дерева.

Контрольні питання

6. Які способи представлення бінарних дерев ви знаєте?
7. Яке дерево називається збалансованим?
8. Які способи збалансування бінарних дерев існують?
9. Як будується дерево з випадковим пошуком?
10. Як представити бінарне дерево у вигляді масиву?

Лабораторна робота № 7.

Тема: **“Тестування і відладка програмного засобу”.**

Мета: Розглянути суть та методи створення тестів для перевірки працездатності програми.

Теоретичні відомості:

Основні поняття.

Відладка ПС – це діяльність, направлена на виявлення і виправлення помилок в ПС з використанням процесів виконання його програм. *Тестування* ПС – це процес виконання його програм на деякому наборі даних, для якого наперед відомий результат застосування або відомі правила поведінки цих програм. Вказаний набір даних називається *тестовим або просто тестом*. Таким чином, відладку можна представити у вигляді багатократного повторення трьох процесів: тестування, в результаті якого може бути констатовано наявність в ПС помилки, пошуку місця помилки в програмах і документації ПС і редагування програм і документації з метою усунення виявленої помилки. Іншими словами:

Відладка = Тестування + Пошук помилок + Редагування.

Принципи і види відладки програмного засобу.

Успіх відладки ПС в значній мірі зумовлює раціональна організація тестування. При відладці ПС відшукуються і усуваються, в основному, ті помилки, наявність яких в ПС встановлюється при тестуванні. Як було вже відмічено, тестування не може довести правильність ПС, в кращому разі воно може продемонструвати наявність в ньому помилки. Іншими словами, не можна гарантувати, що тестуванням ПС практично здійснимим набором тестів можна встановити наявність тієї, що кожної є в ПС помилки. Тому виникає два завдання. Перше завдання: підготувати такий набір тестів і застосувати до них ПС, щоб виявити в ньому по можливості більше число помилок. Проте чим довше продовжується процес тестування (і відладки в цілому), тим більшою стає вартість ПС. Звідси друге завдання: визначити момент закінчення відладки ПС (або окремої його компоненти). Ознакою можливості закінчення відладки є повнота обхвату пропущеними через ПС тестами (тобто тестами, до яких застосовано ПС) безлічі різних ситуацій, що виникають при виконанні програм ПС, і відносно рідкісний прояв помилок в ПС на останньому відрізку процесу тестування. Останнє визначається відповідно до необхідного ступеня надійності ПС, вказаної в специфікації його якості.

Для оптимізації набору тестів, тобто для підготовки такого набору тестів, який дозволяв би при заданому їх числі (або при заданому інтервалі часу, відведеному на тестування) виявляти більше число помилок в ПС, необхідно, по-перше, наперед планувати цей набір і, по-друге, використовувати раціональну стратегію планування тестів. Проектування тестів можна починати відразу ж після завершення етапу зовнішнього опису ПС. Можливі різні підходи до вироблення стратегії проектування тестів, які можна умовно графічно розмістити (див. мал. 10.1) між наступними двома

крайніми підходами. Лівий крайній підхід полягає в тому, що тести проектуються тільки на підставі вивчення специфікацій ПС (зовнішнього опису, опису архітектури і специфікації модулів). Будова модулів при цьому ніяк не враховується, тобто вони розглядаються як чорні ящики. Фактично такий підхід вимагає повного перебору всіх наборів вхідних даних, оскільки інакше деякі ділянки програм ПС можуть не працювати при пропуску будь-якого тесту, а це означає, що помилки, що містяться в них, не виявлятимуться. Проте тестування ПС повною безліччю наборів вхідних даних практично нездійсненно. Правий крайній підхід полягає в тому, що тести проектуються на підставі вивчення текстів програм з метою протестувати всі шляхи виконання кожної програм ПС. Якщо взяти до уваги наявність в програмах циклів із змінним числом повторень, то різних шляхів виконання програм ПС може опинитися також надзвичайно багато, так що їх тестування також буде практично нездійсненно.

Оптимальна стратегія проектування тестів розташована усередині інтервалу між цими крайніми підходами, але ближче до лівого краю. Вона включає проектування значної частини тестів по специфікаціях, але вона вимагає також проектування деяких тестів і по текстах програм. При цьому в першому випадку ця стратегія базується на принципах:

- на кожну використовувану функцію або можливість – хоч би один тест
- на кожну область і на кожну межу зміни якої-небудь вхідної величини – хоч би один тест
- на кожну особливу (виняткову) ситуацію, вказану в специфікаціях, – хоч би один тест.

У другому випадку ця стратегія базується на принципі: кожна команда кожної програми ПС повинна пропрацювати хоч би на одному тесті.

Оптимальну стратегію проектування тестів можна конкретизувати на підставі наступного принципу: для кожного програмного документа (включаючи тексти програм), що входить до складу ПС, повинні проектуватися свої тести з метою виявлення в ньому помилок. В усякому разі, цей принцип необхідно дотримувати відповідно до визначення ПС і змістом поняття технології програмування як технології розробки надійних ПС (див. лекцію 1). У зв'язку з цим Майерс навіть визначає різні види тестування залежно від виду програмного документа, на підставі якого будуються тести. У нашій країні розрізняються два основні види відладки (включаючи тестування): автономну і комплексну відладку ПС. *Автономна* відладка ПС означає послідовне роздільне тестування різних частин програм, що входять в ПС, з пошуком і виправленням в них помилок, що фіксуються при тестуванні. Вона фактично включає відладку кожного програмного модуля і відладку сполучення модулів. Комплексна відладка означає тестування ПС в цілому з пошуком і виправленням помилок, що фіксуються при тестуванні, у всіх документах (включаючи тексти програм ПС), що відносяться до ПС в цілому. До таких документів відносяться визначення вимог до ПС, специфікації якості ПС, функціональна специфікація ПС, опис архітектури ПС і тексти програм ПС.

Заповіді відладки програмного засобу.

Але спочатку слід зазначити деякий феномен, який підтверджує важливість попередження помилок на попередніх етапах розробки: у міру зростання числа виявлених і виправлених помилок в ПС *росте* також відносна вірогідність існування в ньому невиявлених помилок. Це пояснюється тим, що при зростанні числа помилок, виявлених в ПС, уточнюється і наше уявлення про загальне число допущених в ньому помилок, а значить, в якійсь мірі, і про число невиявлених ще помилок.

Нижче приводяться рекомендації по організації відладки у формі заповідей.

Заповідь 1. Вважайте тестування ключовим завданням розробки ПС, доручайте його найкваліфікованішим і обдарованим програмістам; небажано тестувати свою власну програму.

Заповідь 2. Хороший той тест, для якого висока вірогідність виявити помилку, а не той, який демонструє правильну роботу програми.

Заповідь 3. Готуйте тести як для правильних, так і для неправильних даних.

Заповідь 4. Документуйте пропуск тестів через комп'ютер; детально вивчайте результати кожного тесту; уникайте тестів, пропуск яких не можна повторити.

Заповідь 5. Кожен модуль підключайте до програми тільки один раз; ніколи не змінюйте програму, щоб полегшити її тестування.

Заповідь 6. Пропускайте наново всі тести, пов'язані з перевіркою роботи якої-небудь програми ПС або її взаємодії з іншими програмами, якщо в неї були внесені зміни (наприклад, в результаті усунення помилки).

Автономна відладка програмного засобу.

При автономній відладці ПС кожен модуль насправді тестується в деякому програмному оточенні, крім випадку, коли відладжувана програма складається тільки з одного модуля. Це оточення складається з інших модулів, частина яких є модулями відладжуваної програми, які вже відладжені, а частина – модулями, що управляють відладкою (*налагоджувальними* модулями, див. нижче). Таким чином, при автономній відладці тестується завжди деяка програма (*тестована програма*), побудована спеціально для тестування відладжуваного модуля. Ця програма лише частково співпадає з відладжуваною програмою, крім випадку, коли відладжується останній модуль відладжуваної програми. В процесі автономної відладки ПС проводиться нарощування тестованої програми відладженими модулями: при переході до відладки наступного модуля в його програмне оточення додається останній відладжений модуль. Такий процес нарощування програмного оточення відладженими модулями називається *інтеграцією* програми. Налагоджувальні модулі, що входять в оточення відладжуваного модуля, залежать від порядку, в якому відладжуються модулі цієї програми, від того, який модуль відладжується і, можливо, від того, який тест пропускатиметься.

При висхідному тестуванні це оточення міститиме тільки один

налагоджувальний модуль (крім випадку, коли відладжується останній модуль відладжуваної програми), який буде головним в тестованій програмі. Такий налагоджувальний модуль називають *ведучим* (або драйвером). Провідний налагоджувальний модуль готує інформаційне середовище для тестування відладжуваного модуля (тобто формує її стан, потрібний для тестування цього модуля, зокрема, шляхом введення деяких тестових даних), здійснює звернення до відладжуваного модуля і після закінчення його роботи видає необхідні повідомлення. При відладці одного модуля для різних тестів можуть складатися різні ведучі налагоджувальні модулі.

При низхідному тестуванні оточення відладжуваного модуля як налагоджувальні модулі містить *налагоджувальні імітатори* (заглушки) деяких ще не відладжених модулів. До таких модулів відносяться, перш за все, всі модулі, до яких може звертатися відладжуваний модуль, а також ще не відладжені модулі, до яких можуть звертатися вже відладжені модулі (включені в це оточення). Деякі з цих імітаторів при відладці одного модуля можуть змінюватися для різних тестів.

На практиці в оточенні відладжуваного модуля можуть міститися налагоджувальні модулі обох типів, якщо використовується змішана стратегія тестування. Це пов'язано з тим, що як висхідне, так і низхідне тестування має свої достоїнства і свої недоліки.

До *достоїнств висхідного тестування відносяться:*

- простота підготовки тестів
- можливість повної реалізації плану тестування модуля.
- Це пов'язано з тим, що тестовий стан інформаційного середовища готується безпосередньо перед зверненням до відладжуваного модуля (провідним налагоджувальним модулем).
- *Недоліками висхідного тестування* є наступні його особливості:
- тестові дані готуються, як правило, не в тій формі, яка розрахована на користувача (крім випадку, коли відладжується останній, головний, модуль відладжуваної програм);
- великий об'єм налагоджувального програмування (при відладці одного модуля доводиться складати багато провідних налагоджувальних модулів, що формують відповідний стан інформаційного середовища для різних тестів);
- необхідність спеціального тестування сполучення модулів.
- До *достоїнств низхідного тестування* відносяться наступні його особливості:
- більшість тестів готуються у формі, розрахованій на користувача;
- у багатьох випадках відносно невеликий об'єм налагоджувального програмування (імітатори модулів, як правило, вельми прості і кожен придатний для великого числа, нерідко –для всіх, тестів);
- відпадає необхідність тестування сполучення модулів.

Недоліком низхідного тестування є те, що тестовий стан

інформаційного середовища перед зверненням до відладжуваного модуля готується побічно – воно є результатом застосування вже відладжених модулів до тестових даних або даних, що видаються імітаторами. Це, по-перше, утрудняє підготовку тестів і вимагає високої кваліфікації тестовика (розробника тестів), а по-друге, робить скрутним або навіть неможливим реалізацію повного плану тестування відладжуваного модуля. Вказаний недолік іноді вимушує розробників застосовувати висхідне тестування навіть у разі низхідної розробки. Проте частіше застосовують деякі модифікації низхідного тестування, або деяку комбінацію низхідного і висхідного тестування. Виходячи з того, що низхідне тестування, у принципі, є переважним, зупинимося на прийомах, що дозволяють в якійсь мірі подолати вказані труднощі.

Перш за все, необхідно організувати відладку програми так, щоб якомога раніше були відладжені модулі, що здійснюють введення даних, – тоді тестові дані можна готувати у формі, розрахованій на користувача, що істотно спростить підготовку подальших тестів. Далеко не завжди це введення здійснюється в головному модулі, тому доводиться в першу чергу відладжувати ланцюжки модулів, ведучі до модулів, що здійснюють вказане введення. Поки модулі, що здійснюють введення даних, не відладжені, тестові дані поставляються деякими імітаторами: вони або включаються в імітатор як його частина, або вводяться цим імітатором.

При низхідному тестуванні деякі стани інформаційного середовища, при яких потрібно тестувати відладжуваний модуль, можуть не виникати при виконанні відладжуваної програми ні при яких вхідних даних. У цих випадках можна було б взагалі не тестувати відладжуваний модуль, оскільки помилки, що виявляються при цьому, не виявлятимуться при виконанні відладжуваної програми ні при яких вхідних даних. Проте так поступати не рекомендується, оскільки при змінах відладжуваної програми (наприклад, при супроводі ПС) не використані для тестування відладжуваного модуля стани інформаційного середовища можуть вже виникати, що вимагає додаткового тестування цього модуля (а цього при раціональній організації відладки можна було б не робити, якщо сам даний модуль не змінювався). Для здійснення тестування відладжуваного модуля у вказаних ситуаціях іноді використовують відповідні імітатори, щоб створити необхідний стан інформаційного середовища. Частіше ж користуються модифікованим варіантом низхідного тестування, при якому відладжувані модулі перед їх інтеграцією заздалегідь тестуються окремо (в цьому випадку в оточенні відладжуваного модуля з'являється провідний налагоджувальний модуль, разом з імітаторами модулів, до яких може звертатися відладжуваний модуль). Проте, представляється доцільнішою інша модифікація низхідного тестування: після завершення низхідного тестування відладжуваного модуля для досяжних тестових станів інформаційного середовища слідує його окремо протестувати для решти необхідних станів інформаційного середовища.

Часто застосовують також комбінацію висхідного і низхідного тестування,

яку називають методом *сандвіча*. Суть цього методу полягає в одночасному здійсненні як висхідного, так і низхідного тестування, поки ці два процеси тестування не зустрінуться на якому-небудь модулі десь у середині структури відладжуваної програми. Цей метод при розумному порядку тестування дозволяє скористатися достоїнствами як висхідного, так і низхідного тестування, а також в значній мірі нейтралізувати їх недоліки.

Вельми важливим при автономній відладці є тестування сполучення модулів. Річ у тому, що специфікація кожного модуля програми, окрім головного, використовується в цієї програми в двох ситуаціях: по-перше, при розробці тексту (іноді говорять: тіла) цього модуля і, по-друге, при написанні звернення до цього модуля в інших модулях програми. І в тому, і в іншому випадку в результаті помилки може бути порушено необхідну відповідність заданої специфікації модуля. Такі помилки потрібно виявляти і усувати. Для цього і призначено тестування сполучення модулів. При низхідному тестуванні тестування сполучення здійснюється попутно кожним тестом, що пропускається, що вважають гідністю низхідного тестування. При висхідному тестуванні звернення до відладжуваного модуля проводиться не з модулів відладжуваної програми, а з провідного налагоджувального модуля. У зв'язку з цим існує небезпека, що останній модуль може пристосуватися до деяких "помилок" відладжуваного модуля. Тому, приступаючи (в процесі інтеграції програми) до відладки нового модуля, доводиться тестувати кожне звернення до раніше відладженого модуля з метою виявлення неузгодженості цього поводження з тілом відповідного модуля (і не виключено, що винен в цьому раніше відладжений модуль). Таким чином, доводиться частково повторювати в нових умовах тестування раніше відладженого модуля, при цьому виникають ті ж труднощі, що і при низхідному тестуванні.

Автономне тестування модуля доцільно здійснювати в чотири послідовно виконувані кроки.

Крок 1. На підставі специфікації відладжуваного модуля підготуйте тести для кожної можливості і кожної ситуації, для кожної межі областей допустимих значень всіх вхідних даних, для кожної області зміни даних, для кожної області неприпустимих значень всіх вхідних даних і кожної неприпустимої умови.

Крок 2. Перевірте текст модуля, щоб переконатися, що кожен напрям будь-якого розгалуження буде пройдено хоч би на одному тесті. Додайте бракуючі тести.

Крок 3. Перевірте текст модуля, щоб переконатися, що для кожного циклу існують тести, що забезпечують, принаймні, три наступні ситуації: тіло циклу не виконується жодного разу, тіло циклу виконується один раз і тіло циклу виконується максимальне число раз. Додайте бракуючі тести.

Крок 4. Перевірте текст модуля, щоб переконатися, що існують тести, перевіряючі чутливість до окремих особливих значень вхідних даних. Додайте бракуючі тести.

Комплексна відладка програмного засобу.

Як вже було сказано вище, при комплексній відладці тестується ПС в цілому, причому тести готуються по кожному з документів ПС. Тестування цих документів проводиться, як правило, в порядку, зворотному їх розробці. Виняток становить лише тестування документації по застосуванню, яка розробляється по зовнішньому опису паралельно з розробкою текстів програм – це тестування краще проводити після завершення тестування зовнішнього опису. Тестування при комплексній відладці є застосуванням ПС до конкретних даних, які у принципі можуть виникнути у користувача (зокрема, всі тести готуються у формі, розрахованій на користувача), але, можливо, в модельованому (а не в реальній) середовищі. Наприклад, деякі недоступні при комплексній відладці пристрої введення і висновку можуть бути замінені їх програмними імітаторами.

Тестування архітектури ПС. Метою тестування є пошук невідповідності між описом архітектури і сукупністю програм ПС. До моменту почала тестування архітектури ПС повинна бути вже закінчена автономна відладка кожної підсистеми. Помилки реалізації архітектури можуть бути зв'язані, перш за все, з взаємодією цих підсистем, зокрема, з реалізацією архітектурних функцій (якщо вони є). Тому хотілося б перевірити всі шляхи взаємодії між підсистемами ПС. При цьому бажано хоч би протестувати всі ланцюжки виконання підсистем без повторного входження останніх. Якщо задана архітектура представляє ПС як малої системи з виділених підсистем, то число таких ланцюжків буде цілком осяжно.

Тестування зовнішніх функцій. Метою тестування є пошук розбіжностей між функціональною специфікацією і сукупністю програм ПС. Не дивлячись на те, що всі ці програми автономно вже відладжені, вказані розбіжності можуть бути, наприклад, із-за невідповідності внутрішніх специфікацій програм і їх модулів (на підставі яких проводилося автономне тестування) функціональної специфікації ПС. Як правило, тестування зовнішніх функцій проводиться так само, як і тестування модулів на першому кроці, тобто як чорного ящика.

Тестування якості ПС. Метою тестування є пошук порушень вимог якості, сформульованих в специфікації якості ПС. Це найбільш важкий і найменше вивчений вид тестування. Ясно лише, що далеко не кожен примітив якості ПС може бути випробуваний тестуванням. Завершеність ПС перевіряється вже при тестуванні зовнішніх функцій. На даному етапі тестування цього примітиву якості може бути продовжено, якщо потрібно одержати яку-небудь імовірнісну оцінку ступеня надійності ПС. Проте, методика такого тестування ще вимагає своєї розробки. Можуть тестуватися такі примітиви якості, як точність, стійкість, захищеність, тимчасова ефективність, в якійсь мірі – ефективність по пам'яті, ефективність по пристроях, розширюваність і, частково, незалежність від пристроїв. Кожний з цих видів тестування має свою специфіку і заслуговує окремого розгляду. Ми тут обмежимося лише їх переліком. Легкість застосування ПС оцінюється при тестуванні документації по застосуванню ПС.

Тестування документації по застосуванню ПС. Метою тестування є пошук неузгодженості документації по застосуванню і сукупністю програм ПС, а також виявлення незручностей, що виникають при застосуванні ПС. Цей етап безпосередньо передує підключенню користувача до завершення розробки ПС (тестуванню визначення вимог до ПС і атестацій ПС), тому вельми важливо розробникам спочатку самим скористатися ПС так, як це робитиме користувач. Всі тести на цьому етапі готуються виключно на підставі тільки документація по застосуванню ПС. Перш за все, повинні тестуватися можливості ПС як це робилося при тестуванні зовнішніх функцій, але тільки на підставі документації по застосуванню. Повинні бути протестовані всі неясні місця в документацію, а також всі приклади, використані в документації. Далі тестуються найбільш важкі випадки застосування ПС з метою виявити порушення вимог відносності легкості застосування ПС.

Тестування визначення вимог до ПС. Метою тестування є з'ясування, якою мірою ПС не відповідає пред'явленому визначенню вимог до нього. Особливість цього виду тестування полягає в тому, що його здійснює організація-покупець або організація-користувач ПС як один з шляхів подолання бар'єру між розробником і користувачем. Звичайно це тестування проводиться за допомогою контрольних завдань – типових завдань, для яких відомий результат рішення. У тих випадках, коли що розробляється ПС повинне придти на зміну іншої версії ПС, яка вирішує хоч би частину завдань того, що розробляється ПС, тестування проводиться шляхом рішення загальних задач за допомогою як старого, так і нового ПС (з подальшим зіставленням отриманих результатів). Іноді як форму такого тестування використовують *дослідну* експлуатацію ПС – обмежене застосування нового ПС з аналізом використання результатів в практичній діяльності. По суті, цей вид тестування багато в чому перекликається з випробуванням ПС при його атестації, але виконується до атестації, а іноді і замість атестації.

Завдання:

Обрати будь-яку задачу зі свого варіанту з л.р. 1 -6 або власну оригінальну задачу. Скласти блок-схему програми, її граф-модель (управляючий граф). Визначити всі різні прості шляхи руху від входу до виходу граф-моделі програми. Для кожного шляху скласти свій набір вхідних даних та отриманий вручну набір результатів обробки цих вхідних даних. За допомогою одержаних тестових прикладів продемонструвати працездатність програми.

Контрольні питання:

1. Що таке тестування програми?
2. Які методи тестування існують?
3. В чому полягає процес відладки програми?