Житомирський державний університет імені Івана Франка

О.М. Кривонос, Жуковський С.С.

Алгоритми та структури даних

Навчально-методичний посібник

Житомир
Вид-во ЖДУ ім. І. Франка
2020

Рекомендовано вченою радою Житомирського державного університету імені Івана Франка, протокол №10 від 29.11.19 р.

Рецензенти:

Струтинська М.В. – кандидат педагогічних наук, доцент кафедри теоретичних основ інформатики Інституту Інформатики Національного педагогічного університету імені М. П. Драгоманова

Коротун О.В. – кандидат педагогічних наук, доцент доцент кафедри комп'ютерних наук Держаного університету «Житомирська політехніка»

К82

Кривонос О.М. Жуковський С.С.

Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей. – Житомир : Вид-во ЖДУ ім. І. Франка, 2020. – 66 с.

У даному навчальному посібнику розглядаються основні алгоритми сортування, пошуку даних, роботи з графами, деревами та хеш-таблицями, а також алгоритми роботи клітинних автоматів. Для кожного алгоритму наведено опис, псевдокод та графічні пояснення. Посібник розрахований на студентів, що навчаються на технічних спеціальностях, пов'язаних зі створенням програмного забезпечення

ББК 74.26
УДК378

© Кривонос О.М., 2020
© Жуковський С.С., 2020

1. ПРОСТІ СТРУКТУРИ ДАНИХ

1.1. Арифметичні типи

Стандартні типи даних часто називають арифметичними, оскільки їх можна використовувати в арифметичних операціях. Для опису основних типів визначено ключові слова: цілий, символьний, логічний, дійсний.

За допомогою цілих чисел можна представити кількість об'єктів, яка є дискретною за своєю природою (тобто кількість об'єктів можна перерахувати). Внутрішнє представлення величин цілого типу – ціле число в бінарному коді. Внутрішнє представлення дійсного типу складається з двох частин – мантиси і порядку.

Результати логічного типу отримуються при порівнянні даних будь-яких типів. Величини логічного типу можуть приймати тільки значення **true** і **false**. Внутрішня форма представлення значення **false** – 0 (нуль). Будь-яке інше значення інтерпретується як **true**.

Значенням символьного типу є символи з деякої наперед визначеної множини. В більшості сучасних персональних комп'ютерів цією множиною є ASCII. Ця множина складається з 256 різних символів, які впорядковані певним чином і містить символи великих і малих букв, цифр і інших символів, включаючи спеціальні керуючі символи. Значення символьного типу займає в пам'яті 1 байт. Іншою широко використовуваною множиною для представлення символьних даних є код Unicode. В Unicode кожний символ кодується двома байтами.

Над арифметичними типами, як і над всіма іншими, можливі перш за все чотири основних операції: створення, знищення, вибір, поновлення. Специфічні операції над числовими типами – додавання, віднімання, множення і ділення.

Ще одна група операцій над арифметичними типами – операції порівняння: „рівно", „не рівно", „більше", „менше" і т.п. Говорячи про операції порівняння, потрібно звернути увагу на особливість виконання порівнянь на рівність/нерівність дійсних чисел. Оскільки ці числа представляються в пам'яті з деякою точністю, порівняння їх не завжди може бути абсолютно достовірним.

!

1.1.1. Фундаментальні типи в C++

Основні типи в C++ діляться на три категорії: цілочисельні, з плаваючою комою і *void*. Цілочисельні типи дозволяють обробляти цілі числа. Типи з плаваючою комою дозволяють задавати значення, які можуть мати дробові частини.

Типом *void* описується порожній набір значень. Задання змінних типу *void* неможливо. Цей тип служить в основному для оголошення функцій, які не повертають значення, або універсальних покажчиків на нетипізовані або довільно типізовані дані. Будь-який вираз можна явно перетворити або привести до типу *void*. Однак такі вирази можна використовувати тільки в наступних операторах і операндах:

- в операторі виразу;
- в лівому операнді оператора коми;
- в другому і третьому операндах умовного оператора;

У наступній таблиці пояснюються обмеження на розміри типів. Ці обмеження не залежать від реалізації.

Основні типи мови С++:

Категорія	**Тип**	**Опис**
Цілі числа	*char*	це цілочисельний тип, зазвичай містить члени кодування виконання (в Microsoft C++ це кодування ASCII).
		Компілятор С ++ обробляє змінні типу *char*, *signed char* і *unsigned char* як змінні різних типів. Змінні типу *char* підвищуються до *int*, якщо вони мають тип *signed char* за замовчуванням, і не використовується параметр компіляції */J*. У цьому випадку вони розглядаються як тип *unsigned char* і підвищуються до типу *int* без розширення знака.
	bool	це цілочисельний тип, який може мати одне з двох значень: true або false. Його розмір не визначений.
	short	*short int* (або просто *short*) - це цілочисельний тип, розмір якого більше або дорівнює розміру типу *char* і менше або дорівнює розміру типу *int*.
		Об'єкти типу *short* можуть оголошуватися як об'єкти типу *signed short* і *unsigned short*. *Signed short* - синонім *short*.
	int	це цілочисельний тип, розмір якого більше або дорівнює розміру типу *short int* і менше або

		дорівнює розміру типу *long*.
		Об'єкти типу *int* можуть бути оголошені в *unsigned int* або *signed int*. *Signed int* - синонім *int*.
	*__int***n**	Ціле значення заданої розмірності, де **n** - розмір цілого числа в бітах. Значення **n** може бути 8, 16, 32 або 64.
	long	Тип *long* (або *long int*) - це цілочисельний тип, який більше або дорівнює розміру типу *int*.
		Об'єкти типу *long* можуть оголошуватися як об'єкти типу *signed long* і *unsigned long*. *Signed long* - синонім *long*.
	long long	Більше, ніж беззнаковий *long*.
		Об'єкти типу *long long* можуть оголошуватися як об'єкти типу *signed long long* і *unsigned long long*. *Signed long long* - синонім *long long*.
Числа з плаваючою комою	*float*	це тип з плаваючою комою найменшого розміру.
	double	це тип з плаваючою комою, розмір якого більше або дорівнює розміру типу *float*, але менше або дорівнює розміру типу *long double*.
	long double*1*	це тип з плаваючою комою, розмір якого дорівнює розміру типу double.
Розширені символи	*__wchar_t*	Змінна *__wchar_t* позначає тип розширених символів або багатобайтних символів. За замовчуванням тип wchar_t є власним типом, але можна використовувати */Zc: wchar_t-*, щоб зробити *wchar_t* визначенням типу для *unsigned short*. Щоб вказати константу розширеного символьного типу, перед символьною або рядковою константою слід використовувати префікс L.

Представлення *long double* і *double* ідентичні. Проте *long double* і *double* є різними типами.

У наступній таблиці вказані обсяги пам'яті, необхідні для основних типів в Microsoft C ++.

Розміри основних типів:

Тип	**Розмір**
bool	1 байт
char, unsigned char, signed char	1 байт
short, unsigned short	2 байта
int, unsigned int	4 байта
__*int* **n**	8, 16, 32, 64 або128 біт в залежності від значення **n**. __*int***n** відноситься тільки до систем Microsoft.
long, unsigned long	4 байта
float	4 байта
double	8 байт
long double1	8 байт
long long	Аналогічно параметру __int64.

Варіанти завдань:

1. Оголосити змінні за допомогою яких можна буде порахувати загальну суму покупки декількох товарів. Наприклад плитки шоколаду, кави і пакету молока.

2. Оголосити дві змінні типу *int* і присвоїти першій числове значення, друга змінна дорівнює першій змінній збільшеної на 3, а третя змінна дорівнює сумі перших двох.

3. Оголосити змінні, для підрахунку загальної кількості предметів для сервірування столу. Наприклад чашки, такої ж кількості блюдець і ложок.

4. Дано два дійсних числа *a* і *b*. Вирахувати їх суму, різницю, добуток і частку.

5. Обчислити площу трапеції по заданій формулі $S = \frac{1}{2}(a+b)H$, якщо *a, b, H* - відомі.

6. З початку доби пройшло *N* секунд (*N* - ціле). Знайти кількість повних хвилин, що пройшли з початку доби.

7. Дано сторони прямокутника *a* і *b*. Знайти його площу *S* і периметр *P*.

8. Дано два невід'ємних числа *a* і *b*. Знайти їх середнє геометричне.

9. Дано три числа *a*, *b*, *c*. Знайти середнє арифметичне квадратів цих чисел.

10. Обчислити площу трикутника за трьома сторонами - *a*, *b*, *c*. Довжини сторін ввести з клавіатури.

1.2. Перерахований тип

При написанні програм часто виникає потреба визначити декілька іменованих констант, для яких потрібно, щоб усі вони мали різні значення. Для цього зручно використовувати перерахований тип.

Перерахований тип представляє собою впорядкований тип даних, який визначається програмістом, тобто програміст перераховує всі значення, які може приймати змінна цього типу.

Діапазон значень перечислень визначається кількістю біт, які необхідні для представлення всіх його значень. Будь-яке значення цілого типу можна явно привести до перерахованого типу, але при виході за межі його діапазону результат буде невизначеним. При відсутності ініціалізації перша константа приймає нульове значення, а кожній наступній присвоюється на одиницю більше значення від попереднього.

На фізичному рівні над змінними перерахованого типу визначені операції створення, знищення, вибору, поновлення. При цьому виконується визначення порядкового номера ідентифікатора за його значенням і, навпаки, за номером ідентифікатора його значення.

При виконанні арифметичних операцій перечислення перетворюються у ціле. Оскільки перечислення є типом, який визначається користувачем, для нього можна вводити власні операції.

!

В мові C++ для створення перерахування використовується ключове слово *enum*. Загальна форма перерахування має наступний вигляд:

enum ім'я_перерахування {список_імен} список_змінних

Тут ім’я_перерахування - ім'я типу даного перерахування. У списку імен, як і в списку змінних, елементи списку відокремлюються один від одного комами.

Наприклад, в наступному фрагменті програми спочатку визначаються перерахування міст, іменоване *cities*, і змінна *c* типу *cities*, а потім змінній *c* присвоюється значення Houston:

enum cities {Houston, Austin, Amarillo} c;

c = Houston;

У будь-якому перерахуванні значення першого (крайнього зліва) по імені за замовчуванням дорівнює 0, значення другого імені дорівнює 1 і т.д. Взагалі кожному імені присвоюється значення, на одиницю більше від значення попереднього імені. Додавши ініціалізатор, можна надати імені деяке конкретне значення. Наприклад, в наступному перерахуванні ім'я Austin матиме значення 10:

enum cities {Houston, Austin = 10, Amarillo}

У цьому прикладі ім'я Amarillo матиме значення 11.

Приклад програми:

Оголосимо перерахований тип річниць весілля.

```
#include <iostream>

using namespace std;

enum  weddingAnn {chintz = 1, paper, leather, linen, wooden} year; //визначаємо
перерахування і оголошуємо змінну

int main()

{

   setlocale(LC_ALL, "rus");

   cout << "Олег с Ольгой отмечают\t" << chintz << "-ю годовщину со дня
свадьбы!!!";

   cout << "\n";

   cout << "Андрей с Анной отмечают\t" << paper << "-ю годовщину со дня
свадьбы!!!";

   cout << "\n";
```

```
    cout << "Марк с Ириной отмечают\t" << leather << "-ю годовщину со дня свадьбы!!!";
    cout << "\n";
    cout << "Игорь с Юлией отмечают\t" << linen << "-ю годовщину со дня свадьбы!!!";
    cout << "\n";
    cout << "Олег с Аллой отмечают\t" << wooden << "-ю годовщину со дня свадьбы!!!";
    cout << "\n\n";
return 0;
}
```

Варіанти завдань:

1. Дана п'ятиповерхова будівля, на кожному поверсі якої знаходяться різні магазини. Написати програму симулятор ліфта з вибором поверха і отримання інформації про магазин який на ньому знаходиться.

2. Написати програму, яка при введені коду професії виводить опис цієї професії. Взяти 6 професій.

3. Написати програму, яка визначає рівень введеної заробітної плати: «дуже низька», «низька», «середня», «висока», «дуже висока»,.

4. Є автомат який продає каву різних ємностей. Залежно від того куди на яку кнопку натисне покупець, такий результат і отримає (текст сповіщення) – 1 (маленька), 2 (середня), 3 (велика), 4 (дуже велика).

5. Написати програму, яка при введені порядкового номера дня тижня (1, 2, 3 і тд) виводить назву цього дня тижня.

6. Написати програму, яка при введені року виводить інформацію про те, який це був рік по китайському гороскопу. Взяти останніх 5 років.

7. Написати програму, яка при введені порядкового списку по журналу виводить інформацію про студента (прізвище та ім'я). Взяти 7 порядкових номерів.

8. Написати програму, яка при введені номеру маршруту руху автобуса виводить інформацію про кінцеві зупинки. Взяти 5 маршрутів.

9. Написати програму, яка при введені порядкового номеру місяця (1, 2, 3 і тд) виводить назву цього місяця. Взяти 6 місяців.

10. Написати програму, яка при введені набраної кількості балів визначає який рівень його здобутків: «початківець», «любитель», «середняк», «професіонал».

1.3. Покажчики

Тип „покажчик" представляє собою адресу комірки пам'яті (в більшості сучасних обчислювальних систем розмір комірки – мінімальної адресованої одиниці пам'яті – складає один байт). При програмуванні на низькому рівні робота з адресами складає значну частину програм. При вирішенні прикладних задач з використанням мов високого рівня найбільш часті випадки, коли програміст може використовувати покажчики, наступні:

1. При необхідності представити одну й ту ж ділянку пам'яті, а значить, одні й ті ж фізичні дані, як дані різної логічної структури.
2. При роботі з динамічними структурами даних.

В С++ розрізняють три види покажчиків – покажчик на об'єкт, на функцію і на пустий тип, які відрізняються властивостями і набором допустимих операцій. Покажчик не є самостійним типом, він завжди зв'язаний з якимось іншим типом.

У програмах на мовах високого рівня покажчики можуть бути типізованими і нетипізованими. При оголошені типізованого покажчика визначається й тип об'єкту в пам'яті, який адресується цим покажчиком.

Хоча фізична структура адреси не залежить від типу й значення даних, які зберігаються за цією адресою, компілятор вважає покажчики на різні типи такими, що мають різний тип. Таким чином, коли йде мова про типізовані покажчики, правильніше говорити не про єдиний тип даних „покажчик", а про цілу сім'ю типів: „покажчик на ціле", „покажчик на символ".

Нетипізований покажчик використовується для представлення адреси, за якою містяться дані невідомого типу. Робота з нетипізованими покажчиками суттєво обмежена, вони можуть використовуватися тільки для збереження адреси, звертатися за такою адресою не можна.

Основними операціями, в яких беруть участь покажчики є присвоєння, отримання адреси, вибір.

Перерахованих операцій достатньо для вирішення задач прикладного програмування тому набір операцій над покажчиками в більшості мов програмування цим й обмежується. Системне програмування потребує більш гнучкої роботи з адресами, тому в C/C++ доступні також операції адресної арифметики.

!

Кожна змінна, яку ви оголошуєте в програмі, має адресу - номер комірки пам'яті, в якій вона розташована. Адреса є невід'ємною характеристикою змінної. Можна оголосити іншу змінну, яка буде зберігати цю адресу і яка називається покажчиком. Покажчики застосовуються при передачі у функцію параметрів, які ми хочемо змінити, при роботі з масивами, при роботі з динамічною пам'яттю і в ряді інших випадків.

В мові C++ оголошення покажчика має наступний синтаксис:

<тип> *<ідентифікатор> [= <ініціалізатор>];

Покажчик може вказувати на значення базового, перерахованого типу, структури, об'єднання, функції, покажчика.

```
int *pi;                              //Покажчик на int
char *ppc;                            //Покажчик на покажчик на char
int* p, s;                            //Поганий стиль оголошення, s - НЕ покажчик!
int *p, s;                            //Видно, що s - НЕ покажчик
int *p, *s;                           //Два покажчика
char *names[] = {"John", "Anna"};     //Масив покажчиків
```

В останньому оголошенні для формування типу використовуються два оператори: * і [], один з яких стоїть перед ім'ям, а інший - після. Використання операторів оголошення значно спростилося б, якби вони були всі або префіксами, або суфіксами. Однак, *, [] і () розроблялися так, щоб відображати їх сенс у виразах. Таким чином, * є префіксом, а [] і () - суфіксами. Суфіксні оператори «міцніше пов'язані» з ім'ям, ніж префіксні. Отже, *names[] означає масив покажчиків на які-небудь об'єкти, а для визначення типів на кшталт «покажчик на функцію», необхідно використовувати дужки.

Існують дві операції, які мають відношення до роботи з покажчиками. Цими операціями є:

- операція взяття адреси (адресація) &;
- операція взяття значення за адресою (непряма адресація або розіменування) *.

```
int a, *p;
p = &a;                    //Змінній p привласнюється адреса змінної a
*p = 0;                    //Значення за адресою, що знаходиться у змінній p
                           (тобто значення змінної a), стає рівним 0
```

Покажчики можна використовувати для обробки масивів:

У наступному прикладі всі елементи масиву будуть виведені на екран без використання індексів (зверніть увагу, що при цьому параметри циклу можуть бути будь-якими, аби цикл виконався потрібну кількість разів):

```
int ar[] = {-72, 3, 402, -1, 55, 132};
int* p = ar;
for (int i=101; i<=106; i++) {
    cout << *p << ' ';
    p++;
}
```

Покажчики можуть посилатися на інші покажчики. При цьому в комірках пам'яті, на які будуть посилатися перші покажчики, будуть міститися не значення, а адреси других покажчиків. Число символів * при оголошенні покажчика показує порядок покажчика. Щоб отримати доступ до значення, на яке посилається покажчик його необхідно разименовать відповідну кількість разів. Розробимо програму, яка буде виконувати деякі операції з покажчиками порядку вище першого:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
```

```
int var = 123; //ініціалізація змінної var числом 123

int *ptrvar = &var; //покажчик на змінну var

int **ptr_ptrvar = &ptrvar; //покажчик на покажчик на змінну var

int ***ptr_ptr_ptrvar = &ptr_ptrvar;

cout << " var\t\t= " << var << endl;

cout << " *ptrvar\t= " << *ptrvar << endl;

cout << " **ptr_ptrvar   = " << **ptr_ptrvar << endl; //два раза разименовуєм
покажчик, так як він другого порядку

cout << " ***ptr_ptrvar  = " << ***ptr_ptr_ptrvar << endl; //покажчик третього
порядку

cout << "\n ***ptr_ptr_ptrvar -> **ptr_ptrvar -> *ptrvar ->      var -> "<< var <<
endl;

cout << "\t  " << &ptr_ptr_ptrvar<< " -> " << "    " << &ptr_ptrvar << " ->" <<
&ptrvar << " -> " << &var << " -> " << var << endl;

return 0;

}
```

При цьому на екрані ми побачимо:

```
var           = 123
 *ptrvar       = 123
 **ptr_ptrvar   = 123
 ***ptr_ptrvar  = 123


 ***ptr_ptr_ptrvar -> **ptr_ptrvar -> *ptrvar ->      var -> 123
         0x22ff00 ->     0x22ff04 ->0x22ff08 -> 0x22ff0c -> 123
```

Варіанти завдань:

Для обходу масиву використовувати покажчики (заборонено звертатися до елементів масиву за індексами).

1. Заданий масив, що складається з 15 елементів дійсного типу. Визначити кількість елементів, значення яких більше першого елемента.

2. Заданий масив, що складається з 16 елементів дійсного типу. Визначити індекси (місцерозташування) максимального і мінімального елементів

3. Ввести цілочисельний масив, що складається з 14 елементів. Обчислити кількість і суму парних за значенням додатніх елементів.

4. Ввести масив, що складається з 14 елементів цілого типу. Знайти кількість елементів парних за значенням.

5. Ввести масив, що складається з 12 елементів цілого типу. Отримати новий масив, замінивши значення п'ятого елементу середньоарифметичним вихідного масиву.

6. Ввести цілочисельний масив, що складається з 10 елементів. Поміняти місцями максимальний і перший елементи.

7. Ввести цілочисельний масив, що складається з 9 елементів. Поміняти місцями максимальний і мінімальний елементи масиву.

8. Ввести масив, що складається з 20 елементів цілого типу. Визначити яких елементів більше парних або непарних за значенням.

9. Ввести масив, що складається з 12 елементів дійсного типу. Визначити кількість чисел, що стоять між максимальним і мінімальним елементами.

10. Ввести цілочисельний масив, що складається з 14 елементів. Обчислити кількість і суму непарних за значенням від'ємних елементів.

2. СТАТИЧНІ СТРУКТУРИ ДАНИХ

Статичні структури відносяться до класу структур, які представляють собою структуровану множину примітивних, базових, структур. Оскільки статичні структури відрізняються відсутністю змінності, пам'ять для них виділяється автоматично – як правило, на етапі компіляції, або при виконанні – в момент активізації того програмного блоку, в якому вони описані. Ряд мов програмування допускають розміщення статичних структур в пам'яті на етапі виконання за явною вимогою програміста, але й у цьому випадку обсяг виділеної пам'яті залишається незмінним до знищення структури. Виділення пам'яті на етапі компіляції є такою зручною властивістю статичних структур, що у ряді задач програмісти використовують їх навіть для представлення

об'єктів, які мають властивість змінності. Наприклад, коли розмір масиву невідомий наперед, для нього резервується максимально можливий розмір.

Статичні структури в мовах програмування зв'язані із структурованими типами. Структуровані типи в мовах програмування є тими засобами інтеграції, які дозволяють будувати структури даних будь-якої складності. До таких типів відносяться масиви, структури та їхні похідні типи.

2.1. Масиви

Логічно масив об'єднує елементи одного типу даних, тобто належить до однорідного типу даних. Більше формально його можна визначити як впорядковану сукупність елементів деякого типу, які адресуються за допомогою одного або декількох індексів.

Масиви можна класифікувати за кількістю розмірностей масиву масиви поділяються на одновимірні масиви (вектори), двохвимірні (матриці) і багатовимірні (трьох, чотирьох і більше).

Логічно масив – це така структура даних, яка характеризується:

- фіксованим набором елементів одного і того ж типу;
- кожний елемент має унікальний набір значень індексів;
- кількість індексів визначають мірність масиву;
- звернення до елемента масиву виконується за ім'ям масиву і значенням індексів для даного елемента.

Фізична структура масиву – це спосіб розміщення елементів масиву в пам'яті комп'ютера. Під елемент масиву виділяється кількість байт пам'яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам'яті для зберігання масиву.

Сама найважливіша операція фізичного рівня над масивом – доступ до заданого елемента. Як тільки реалізовано доступ до елемента, над ним може бути виконана будь-яка операція, що має сенс для того типу даних, якому відповідає елемент. Перетворення логічної структури масиву у фізичну називається процесом лінеаризації, в ході якого багатовимірна логічна структура масиву перетвориться в одновимірну фізичну структуру.

Адресою масиву є адреса першого байту початкового компоненту масиву. Індексація масивів в C/C++ обов'язково починається з нуля.

До операцій логічного рівня над масивами необхідно віднести такі як сортування масиву, пошук елемента за ключем.

!

В мові С++ оголошення масиву має наступний синтаксис:

<специфікація типу> <ім'я> [<константний вираз>];

<специфікація типу> <ім'я> [];

Тут квадратні дужки є елементом синтаксису, а не ознакою необов'язковості конструкції.

Оголошення масиву можуть мати одну з двох синтаксичних форм, зазначених вище. Квадратні дужки, які слідують за *ім'ям*, - ознака того, що змінна є масивом. Константний вираз, укладений в квадратні дужки визначає число елементів у масиві. Індексація елементів масиву в мові С ++ починається з нуля. Таким чином, останній елемент масиву має індекс на одиницю менше, ніж число елементів масиву.

У другій синтаксичній формі константний вираз у квадратних дужках опущено. Ця форма може бути використана, якщо в оголошенні масиву присутній ініціалізатор, або масив оголошується як формальний параметр функції, або дане оголошення є посиланням на оголошення масиву десь в іншому місці програми. Однак для багатовимірного масиву може бути опущена тільки перша розмірність.

Багатомірний масив, або масив масивів, оголошується шляхом задання послідовності константних виразів у квадратних дужках, яка слідує за ім'ям:

<специфікація типу> <ім'я> [<констант. вираз>] [<констант. вираз>] ...;

Кожний константний вираз визначає кількість елементів в даному вимірі масиву, тому оголошення двовимірного масиву містить два константних вирази, тривимірного - три тощо.

Масив може складатися з елементів будь-якого типу, крім типу *void* і функцій, тобто елементи масиву можуть мати базовий, перерахований, структурний тип, бути об'єднанням, покажчиком або масивом.

Приклади оголошень масивів:

int x [10];	// Одновимірний масив з 10 цілих чисел. Індекси змінюються від 0 до 9.
double y [2] [10];	// Двовимірний масив дійсних чисел з 2 рядків і 10 стовпців.

Як і прості змінні, масиви можуть бути ініціалізовані при оголошенні. Ініціалізатор для об'єктів складових типів (яким є масив) складається зі списку

ініціалізаторов, розділених комами і укладених у фігурні дужки. Кожен ініціалізатор в списку являє собою або константу відповідного типу, або, у свою чергу, список ініціалізаторов. Ця конструкція використовується для ініціалізації багатовимірних масивів.

Наявність списку ініціалізаторов в оголошенні масиву дозволяє не вказувати число елементів по його першій розмірності. У цьому випадку кількість елементів у списку ініціалізаторов і визначає число елементів по першій розмірності масиву. Тим самим визначається розмір пам'яті, яка необхідна для зберігання масиву. Число елементів по решті розмірностям масиву, окрім першої, вказувати обов'язково.

Якщо в списку ініціалізаторов менше елементів, ніж у масиві, то залишилися елементи неявно ініціалізуються нульовими значеннями. Якщо ж число ініціалізаторов більше, ніж потрібно, то видається повідомлення про помилку.

Приклади ініціалізації масивів

int a[3] = {0, 1, 2}; // Число ініціалізаторов дорівнює числу елементів

double b[5] = {0.1, 0.2, 0.3}; // Число ініціалізаторов менше числа елементів

int c [] = {1, 2, 4, 8, 16}; // Число елементів масиву визначається за кількістю ініціалізаторов

int d [2][3] = {{0, 1, 2}, {3, 4, 5}}; // Ініціалізація двовимірного масиву. Масив складається з двох рядків, в кожному з яких по 3 елементи. Елементи першого рядка отримують значення 0, 1 і 2, а другий - значення 3, 4 і 5.

int e[3] = {0, 1, 2, 3}; // Помилка - число ініціалізаторов більше числа елементів

Зверніть увагу, що не існує присвоювання масиву, відповідного описаному вище способу ініціалізації.

int a[3] = {0, 1, 2}; // Оголошення і ініціалізація

a = {0, 1, 2}; // Помилка

Для доступу до конкретного елементу масиву використовуються так звані індексні вирази:

<ім'я масиву> [<цілочисельний вираз>]

Тут квадратні дужки є вимогою синтаксису мови, а не ознакою необов'язковості конструкції.

Індекс масиву може бути не тільки константою, а й виразом, який має цілочисельний тип, наприклад, *a[i + 1]* (тут *a* повинно бути ім'ям раніше оголошеного масиву, а *i* - змінної цілого типу).

Для обробки елементів масиву зазвичай використовується оператор покрокового циклу for:

```
for (i = 0;        // Привласнюємо лічильнику циклу значення індексу
                   першого елемента
    i<n;           // Умова продовження циклу - поки значення лічильника
                   менше кількості елементів масиву
    i++)           // Збільшуємо лічильник циклу на 1 для переходу до
                   наступного елементу масиву
  <тіло циклу>     // У тілі циклу відбувається обробка одного елемента
                   масиву
```

Для обробки багатовимірного масиву використовується відповідна кількість циклів.

У мові С++ немає можливості вводити і виводити весь масив одним оператором вводу/виводу. Можна вводити і виводити тільки один елемент масиву. Отже, для того щоб ввести весь масив, треба використовувати цикл.

```
int a [10], n;
cout<<("Введіть кількість елементів масиву (від 0 до 9):");
                   // Оголошуємо масив і змінну для кількості елементів
                   масиву
cin>> (n);         // Введення кількості елементів масиву
if (n <0 || n> 9)
 {cout<< ("Кількість елементів масиву повинно бути від 0 до 9! \ ");
  return;
 }                 // Якщо вхідні дані невірні, то друкуємо відповідне
                   повідомлення і виходимо з програми
```

```
for (i = 0; i<n; i++)        // Введення масиву по одному елементу
 cin>> (a[i]);
```

Вивід також здійснюється в циклі.

```
for (i = 0; i <n; i ++)
 cout<<("a[", i+1, "]=", a [i]);
```

В результаті на екрані ми побачимо приблизно наступний текст:

```
a [1] = 4
a [2] = 15
a [3] = -2
...
```

Приклад програми пошуку максимального елемента масиву:

```
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;
int main() {
	int X[20];
	int max=0;
	for(int i=0; i<20; i++){
		cout<<"\nВведіть елемент масиву X[",i+1,"]=";
		cin>>X[i];
	}
	max=x[0];
	for(int i=1; i<20; i++){
```

```
            if(max<x[i]) max=x[i];
        }
        cout<<"Максимальний елемент масиву - "<<max;
        return 0;
}
```

Варіанти завдань:

1. Заданий масив, що складається з 15 елементів дійсного типу. Визначити кількість елементів, значення яких більше першого елемента.

2. Заданий масив, що складається з 16 елементів дійсного типу. Визначити індекси (місцерозташування) максимального і мінімального елементів

3. Ввести цілочисельний масив, що складається з 14 елементів. Обчислити кількість і суму парних за значенням додатніх елементів.

4. Ввести масив, що складається з 14 елементів цілого типу. Знайти кількість елементів парних за значенням.

5. Ввести масив, що складається з 12 елементів цілого типу. Отримати новий масив, замінивши значення п'ятого елементу середньоарифметичним вихідного масиву.

6. Ввести цілочисельний масив, що складається з 10 елементів. Поміняти місцями максимальний і перший елементи.

7. Ввести цілочисельний масив, що складається з 9 елементів. Поміняти місцями максимальний і мінімальний елементи масиву.

8. Ввести масив, що складається з 20 елементів цілого типу. Визначити яких елементів більше парних або непарних за значенням.

9. Ввести масив, що складається з 12 елементів дійсного типу. Визначити кількість чисел, що стоять між максимальним і мінімальним елементами.

10. Ввести цілочисельний масив, що складається з 14 елементів. Обчислити кількість і суму непарних за значенням від'ємних елементів.

2.2. Розріджені масиви

На практиці зустрічаються масиви, які через певні причини можуть займати пам'ять не повністю, а частково. Це особливо актуально для масивів великих

розмірів, таких що для їхнього зберігання в повному об'ємі пам'яті може бути недостатньо. Розріджений масив – це масив, більшість елементів якого рівні між собою, так що зберігати в пам'яті достатньо лише невелику кількість значень відмінних від основного (фонового) значення інших елементів. При роботи з розрідженими масивами питання розташування їх в пам'яті реалізуються на логічному рівні з врахуванням їхнього типу.

Розрізняють два типи розріджених масивів:

- масиви, в яких розташування елементів із значеннями відмінними від фонового, можуть бути описані математично;
- масиви з випадковим розташуванням елементів.

До **масивів з математичним описом розташування елементів** відносяться масиви, в яких існує закономірності в розташуванні елементів із значеннями відмінними від фонового.

Елементи, значення яких є фоновими, називають нульовими; елементи, значення яких відмінні від фонового, – ненульовими. Фонове значення не завжди рівне нулю. Ненульові значення зберігаються, як правило, в одновимірному масиві, а зв'язок між розташуванням у розрідженому масиві і в новому, одновимірному, описується математично за допомогою формули, що перетворює індекси масиву в індекси вектора.

На практиці для роботи з розрідженим масивом розробляються функції:

- для перетворення індексів масиву в індекс вектора;
- для отримання значення елемента масиву з його упакованого представлення за індексами;
- для запису значення елемента масиву в її упаковане представлення за індексами.

До **масивів з випадковим розташуванням елементів** відносяться масиви, в яких не існує закономірностей у розташуванні елементів із значеннями відмінними від фонового.

Один з основних способів зберігання подібних розріджених матриць полягає в запам'ятовуванні ненульових елементів в одновимірному масиві і ідентифікації кожного елемента масиву індексами.

Дане представлення масиву скорочує вимоги до об'єму пам'яті більш ніж в 2 рази. Спосіб послідовного розподілу має також ту перевагу, що операції над матрицями можуть бути виконані швидше, ніж це можливо при представленні у вигляді послідовного масиву, особливо якщо розмір матриці великий.

Методи послідовного розміщення для представлення розріджених матриць звичайно дозволяють швидше виконувати операції над матрицями і більш

ефективно використати пам’ять, ніж методи із зв’язаними структурами. Проте послідовне представлення матриць має певні недоліки. Так включення і виключення нових елементів матриці викликає необхідність переміщення великої кількості інших елементів. Якщо включення нових елементів і їхнє виключення здійснюється часто, то повинен бути вибраний метод зв’язаних структур.

Метод зв’язаних структур, проте, переводить структуру даних, що представляється, в інший розділ класифікації. При тому, що логічна структура даних залишається статичною, фізична структура стає динамічною.

!

Представлення розрідженого масиву у вигляді зв'язного списку.

При реалізації розрідженого масиву за допомогою зв'язаного списку насамперед необхідно створити структуру, яка містить такі елементи:

- Дані, які зберігаються в комірці
- Логічна позиція комірки в масиві
- Посилання на попередній і наступний елементи

Кожна нова структура розміщається у список так, що елементи залишаються впорядкованими по індексу в масиві. Доступ до масиву проводиться шляхом переходу по посиланнях.

Наприклад, в якості носія елемента розрідженого масиву в електронній таблиці можна використовувати наступну структуру:

```
struct cell{
  char cell_name[9];       //Ім'я комірки, напр., A1, B34
  char formula[128];       //Інформація, напр., 10/B2
  struct cell *next;       //Покажчик на наступний запис
  struct cell *prior;      //Покажчик на попередній запис
};
```

Поле *cell_name* містить рядок, який відповідає імені комірки, наприклад, A1, B34 або Z19. Рядкове поле *formula* зберігає формулу (дані) з відповідної комірки таблиці.

Представлення розрідженого масиву у вигляді двійкового дерева

По суті, двійкове дерево - це просто видозмінений двохзв'язний список. Його основна перевага полягає в можливості швидкого пошуку. Саме завдяки цьому вдається дуже швидко виконувати вставки і затрачати зовсім небагато часу на доступ до елементів.

Щоб використовувати двійкове дерево для реалізації електронної таблиці, необхідно змінити структуру *cell* наступним чином:

```
struct cell{
  char cell_name[9];          //Ім'я комірки, напр., A1, B34
  char formula[128];          // Дані, напр., 10 / B2
  struct cell *left;          // Покажчик на ліве піддерево
  struct cell *right;         // Покажчик на праве піддерево
} List_entry;
```

Представлення розрідженого масиву у вигляді масиву покажчиків

Припустимо, що електронна таблиця має розмір 26x100 (від A1 до Z100), тобто складається з 2600 елементів. Теоретично можна зберігати елементи таблиці в наступному масиві структур:

```
struct cell{
  char cell_name[9];
  char formula[128];
}List_entry[2600];                //2600 комірок
```

Але 2600, помножене на 137 байтів (розмір цієї структури в байтах), дорівнює 356200 байтів пам'яті. Це занадто великий обсяг пам'яті, щоб витрачати його на неповністю використовуваний масив. Тим не менш, можна створити масив покажчиків (*pointer array*) на структури типу cell. Для зберігання масиву покажчиків потрібно набагато менше пам'яті, ніж для масиву структур. При кожному присвоєнню комірці логічного масиву даних під ці дані виділяється пам'ять, а відповідному покажчику в масиві покажчиків привласнюється адреса виділеного фрагмента. Така схема дозволяє домогтися більш високої продуктивності, ніж при зв'язаному списку або двійковому дереві. Опис масиву покажчиків виглядає наступним чином:

```
struct cell {
  char cell_name[9];
  char formula[128];
}List_entry;
struct cell *sheet[2600];               //Масив з 2600 покажчиків
```

Цей менший за обсягом займаної пам'яті масив використовується для зберігання покажчиків на дані, які вводяться в електронну таблицю. При введенні чергового запису в відповідну клітинку масиву заноситься покажчик на введені дані.

Варіанти завдань:

1. Реалізувати розріджений масив у вигляді зв'язного списку, що складається з 1000 елементів дійсного типу. Визначити кількість елементів, значення яких більше першого елемента.

2. Реалізувати розріджений масив у вигляді двійкового дерева, що складається з 1000 елементів дійсного типу. Визначити кількість елементів, значення яких більше першого елемента.

3. Реалізувати розріджений масив у вигляді масиву покажчиків, що складається з 1000 елементів дійсного типу. Визначити кількість елементів, значення яких більше першого елемента.

4. Реалізувати розріджений масив у вигляді зв'язного списку, що складається з 1500 елементів. Обчислити кількість і суму парних за значенням додатніх елементів.

5. Реалізувати розріджений масив у вигляді двійкового дерева, що складається з 1500 елементів. Обчислити кількість і суму парних за значенням додатніх елементів.

6. Реалізувати розріджений масив у вигляді масиву покажчиків, що складається з 1500 елементів. Обчислити кількість і суму парних за значенням додатніх елементів.

7. Реалізувати розріджений масив у вигляді зв'язного списку, що складається з 1000 елементів цілого типу. Знайти кількість елементів парних за значенням.

8. Реалізувати розріджений масив у вигляді двійкового дерева, що складається з 1400 елементів. Обчислити кількість і суму непарних за значенням від'ємних елементів.

9. Реалізувати розріджений масив у вигляді масиву покажчиків, що складається з 2000 елементів цілого типу. Визначити яких елементів більше парних або непарних за значенням.

10. Реалізувати розріджений масив у вигляді зв'язного списку, що складається з 1400 елементів. Обчислити кількість і суму непарних за значенням від'ємних елементів.

2.3. Множини

Тип даних „множина" не реалізований як стандартний тип мови програмування C/C++, але дуже часто використовується в програмуванні і реалізовується засобами визначення типів користувача, тому дамо йому короткий опис.

Множина – така структура, яка є набором даних одного і того ж типу, що не повторюються (кожен елемент множини є унікальним). Порядок слідування елементів множини на має принципового значення.

До множин застосовується стандартний принцип виключення. Це означає, що конкретний елемент або є членом множини, або ні. Множина може бути пустою, таку множину називають нульовою.

Множина є підмножиною іншої множини, якщо в цій другій множині можна знайти усі елементи, які є в першій множині. Відповідно, множина вважається надмножиною іншої множини, якщо вона містить усі елементи цієї другої множини.

Кожен окремий елемент є членом множини, якщо він входить до складу елементів множини.

Над множинами визначені наступні специфічні операції:

1. Об'єднання множин. Результатом є множина, що містить елементи початкових множин.
2. Перетин множин. Результатом є множина, що містить спільні елементи початкових множин.
3. Різниця множин. Результатом є множина, яка містить елементи першої множини, які не входять в другу множину.
4. Симетрична різниця. Результатом є множина, яка містить елементи, які входять до складу однієї або другої множини (але не обох).
5. Перевірка на входження елемента в множину. Результатом цієї операції є значення логічного типу, що вказує чи входить елемент в множину.

2.4. Структури

На відміну від масивів чи множин, усі елементи яких однотипні, структура може містити елементи різних типів.

Елементи структури називаються полями структури і можуть мати довільний тип, крім типу цієї ж структури, але можуть бути покажчиками на неї. Якщо при описі структури відсутній тип структури, обов'язково повинен бути вказаний список змінних, покажчиків або масивів визначеної структури.

Звернення до окремих полів структури замінюються на їхні адреси ще на етапі компіляції.

Самою найважливішою операцією для структури є операція доступу до вибраного поля структури – операція кваліфікації.

Над вибраним полем структури можливі будь-які операції, які допустимі для типів цього поля.

Більшість мов програмування підтримує деякі операції, які працюють із структурою, як з єдиним цілим, а не з окремими її полями. Це операція присвоєння значення одного запису іншому однотипному запису, при цьому відбувається по елементне копіювання.

!

В мові C++ визначення структури складається з двох кроків:

- оголошення структури (завдання нового типу даних певного користувачем), структура складається з полів;

struct Point{

double x; double y; //визначені поля структури

}

struct RGBColor{

char red; char green; char blue; //визначені поля структури

}

- визначення змінних типу структура;

struct Circle{

Point center; //визначено поле структури як структуру

double radius; //визначено звичайне поле структури

int thickness; //визначено звичайне поле структури

RGBColor color; //визначено поле структури як структуру

}

Як ми бачимо структури можна вкладати в інші структури.

Для звернення до полів структури треба вказати ім'я змінної і через крапку ім'я поля:

Circle crcl;	//створенно змінну *crcl* з типом даних *Circle*
crcl.radius = 10.0;	//полю *radius*, структури *Circle*, присвоєно значення
crcl.center.x = -1.5;	
crcl.center.y = 0.7;	//так, як поле *center* структури *Circle* оголошене структурою *Point*, то воно має два відповідних поля, таким чином ми звертаємось через структуру *Circle* до структури *Point*

Приклад задачі та програми:

Задано n комплексних чисел, знайти число найбільшого модуля.

```
#include <iostream>

#include <math.h>

using namespace std;

int main ()

{

        struct complex

        {

                float x;                    // дійсна частина

                float y;                    // уявна частина

        };

        // Оголошення масиву комплексних чисел

        complex p[100];

        int i, n, nmax;
```

```
	float max;
	cout<<"n =";
	cin>>n;
	for (i = 0; i<n; i ++)
	{
		cout<< "Vvedit complex chislo \ n";
		//введення дійсної частини і-го комплексного числа
		cin>> p[i].x;
		//введення уявної частини і-го комплексного числа
		cin>>p[i].y;
		cout<<p[i].x<<"+"<< p[i].y<<"i"<<endl;
	}
	max = pow(p[0].x*p[0].x + p[0].y*p[0].y, 0.5); nmax = 0;
	for (i = 1; i<n; i ++)
		if (pow(p[i].x*p[i].x + p[i].y*p[i].y, 0.5)>max)
			{
			max = pow(p[i].x*p[i].x + p[i].y*p[i].y, 0.5); nmax = i;
			}
	cout<<"Nomer maxsimalnogo modulya"<<nmax<<"\nYogo velichina"
<<max<< endl;
	return 0;
}
```

Варіанти завдань:

Написати програму згідно із завданням

№ вар	Поля структури	Задача

1	Прізвище Амплуа Вік Кількість ігор Кількість голів	Визначити кращого форварда, і вивести відомості про футболістів, які зіграли менше 5-ти ігор.
2	Прізвище Група Фізика Інформатика Історія	Визначити середній бал оцінок по усіх предметах, і вивести відомості про студентів, середній бал яких більше 4.
3	Продавець Найменування Кількість Ціна Дата_продажу	Визначити кількість товарів, які продані менше року тому і вивести відомості про них.
4	Найменування Кількість Ціна Виробник Дата_надходження_на_склад	Визначити кількість всіх товарів, кількість яких більше 5 і вивести відомості про ці товари.
5	Найменування Виробник Рік_випуску Кількість Ціна	Визначити загальну вартість усіх товарів, випущених в поточному році і вивести відомості про ці товари.
6	Найменування Кількість Ціна Виробник	Вивести на екран найменування товару з максимальною загальною вартістю.

	Дата_випуску	
7	Прізвище Група Фізика Інформатика Історія	Визначити середній бал оцінок з фізики, кількість студентів з оцінкою 5 з інформатики та вивести відомості про них.
8	Продавець Найменування Кількість Ціна Дата_продажу	Визначити кількість товарів, проданих продавцем «Іванов», вивести відомості про них і визначити товар з максимальною вартістю.
9	Найменування Кількість Ціна Виробник Дата_надходження_на_склад	Вивести відомості про товари з ціною вище середньої.
10	Автор Кількість_сторінок Тираж Рік_видання	Вивести дані про книги, в яких кількість сторінок більше 150.

2.5. Об'єднання

Об'єднання представляють собою частковий випадок структури, усі поля якої розміщуються за однією ж і тою ж адресою. Формат опису такий же, як і в структури. Довжина об'єднання рівна найбільшій із довжин його полів. У кожен момент часу в змінній типу об'єднання зберігається тільки одне значення, і відповідальність за його правильне використання лягає на програміста.

Об'єднання застосовуються для економії пам'яті в тих випадках, коли відомо, що більше одного поля одночасно не потрібно, а також для різної інтерпретації одного і того ж бітового представлення.

Дуже часто деякі об'єкти програми відносяться до одного й того ж класу, відрізняючись лише деякими деталями. У цьому випадку застосовують комбінацію структурного типу і об'єднання. Об'єднання використовують як поля структури, при цьому в структурі включають поле, яке визначає, який саме елемент об'єднання використовується в кожний момент.

У загальному випадку змінна структура буде складатися з трьох частин: набір спільних компонентів, мітки активного компоненту і частини зі змінними компонентами.

!

В мові C++ загальна форма оголошення об'єднання виглядає наступним чином.

union [<тег>] {<список оголошень елементів>} <описувач> [, <описувач> ...];

union <тег> <описувач> [, <описувач> ...];

Тег призначений для розрізнення декількох об'єднань, оголошених в одній програмі.

Пам'ять, яка виділяється змінній типу об'єднання, визначається розміром найдовшого елемента об'єднання. Всі елементи об'єднання розміщуються в одній і тій же області пам'яті з одної і тої ж адреси. Значення поточного елемента об'єднання втрачається, коли іншому елементу об'єднання присвоюється значення.

У мові C ++ існує спеціальний тип об'єднання, який називаэться *анонімним об'єднанням*. В анонімному об'єднанні не міститься імені класу і не оголошуються ніякі об'єкти. Анонімне об'єднання просто повідомляє компілятор про те, що його змінні-члени повинні мати одну і ту ж саму область пам'яті. Однак до самих змінним можна звертатися безпосередньо, не вдаючись до звичайного синтаксису операторів "точка" і "стрілка".

```
union{
int a;
float f;
};
// ...
a = 10;                                  // доступ до змінної a
```

```
cout<<f;                    // доступ до змінної f
```

Приклад програми:

```
#include <stdio.h>
#include <stdio.h>
#include <iostream>
using namespace std;
void main() {
        union {
                float f;
                long int i;
        } u;
        cout<<"Input float number: ";
        cin>>&u.f;
        cout<<"Internal:\n\n"<<u.i;
 }
```

<u>Варіанти завдань:</u>

1. Написати програму яка запрошує ввести з клавіатури числа різного типу: ціле та дійсне. Додайте свій номер по списку та виведіть по черзі їх на екран використовуючи об'єднання.

2. Написати програму яка запрошує ввести з клавіатури відстань в милях та кілометрах. Додайте до них будь-яке значення та виведіть по черзі їх на екран використовуючи об'єднання.

3. Написати програму яка запрошує ввести з клавіатури суми грошей в гривнях та доларах. Додайте до них будь-яке значення та виведіть по черзі їх на екран використовуючи об'єднання.

4. Написати програму яка запрошує ввести з клавіатури сантиметри та дюйми. Додайте до них будь-яке значення та виведіть по черзі їх на екран використовуючи об'єднання.

5. Написати програму яка запрошує ввести з клавіатури об'єм рідини в літрах та пінтах. Додайте до них будь-яке значення та виведіть по черзі їх на екран використовуючи об'єднання.

6. Написати програму яка виводить на екран відстань від Києва до Харкова в милях та кілометрах. Виведіть по черзі їх на екран використовуючи об'єднання.

7. Написати програму яка виводить на екран швидкість автомобіля в км/год та милях/год. Виведіть по черзі їх на екран використовуючи об'єднання.

8. Написати програму яка виводить на екран цілочисельну та дробову частину числа 327679.623647. Виведіть по черзі їх на екран використовуючи об'єднання.

9. Написати програму яка виводить на екран назви років (теперішнього та наступного) по китайському гороскопу. Виведіть по черзі їх на екран використовуючи об'єднання.

10. Написати програму яка виводить на екран число та місяць вашого народження. Виведіть по черзі їх на екран використовуючи об'єднання.

2.6. Бітові типи

В ряді задач може стати в нагоді робота з окремими бінарними розрядами даних. Частіше всього такі задачі виникають в системному програмуванні, коли, наприклад, окремий розряд зв'язаний з станом окремого апаратного перемикача або окремої шини передачі даних. Дані такого типу представляються у вигляді набору бітів, які упаковані в байти або слова, і логічно не зв'язаних один з одним. Операції над такими даними забезпечують доступ до вибраного біта даного.

Бітові поля – це особливий вид полів структури. Вони використовуються для компактного розміщення даних, наприклад, прапорців типу „так/ні". Мінімально адресована комірка пам'яті – 1 байт, а для зберігання прапорця достатньо одного біта. Бітові поля описуються за допомогою структурного типу.

Бітові поля можуть бути довільного цілого типу. Ім'я поля може бути відсутнім, такі поля використовуються для вирівнювання на апаратну межу. Доступ до поля здійснюється звичайним способом – за іменем.

Над бітовими типами можливі три групи специфічних операцій: операції алгебри логіки, операції зсуву, операції порівняння.

Операції бульової алгебри – *НІ* („~"), *АБО* („|"), *І* („&"), *виключне АБО* („^"). Ці операції і за назвою, і за змістом подібні на операції над логічними

аргументами, але відмінність у їх застосуванні до бітових аргументів полягає в тому, що операції виконуються над окремими розрядами. В мові C/C++ для побітових і загальних логічних операцій використовуються різні позначення.

Операції зсуву виконують зміщення бінарного коду на задану кількість розрядів ліворуч або праворуч. Із трьох можливих типів зсуву (арифметичний, логічний, циклічний) в мовах програмування частіше реалізується лише логічний.

2.7. Таблиці

Елементами векторів і масивів можуть бути інтегровані структури. Одна з таких складних структур – таблиця. З фізичної точки зору таблиця є вектором, елементами якого є структури. Характерною логічною особливістю таблиць є те, що доступ до елементів таблиці проводиться не за номером (індексом), а за ключем – значення однієї з властивостей об'єкту, який описується структурою-елементом таблиці. Ключ – це властивість, що ідентифікує дану структуру в множині однотипних структур і є, як правило, унікальним в даній таблиці. Ключ може включатися до складу структури і бути одним з його полів, але може і не включатися в структуру, а обчислюватися за деякими її властивостями. Таблиця може мати один або декілька ключів.

Основною операцією при роботі з таблицями є операція доступу до структури за ключем. Вона реалізовується процедурою пошуку. Оскільки пошук може бути значне більш ефективним в таблицях, впорядкованих за значеннями ключів, досить часто над таблицями необхідно виконувати операції сортування.

Іноді розрізняють таблиці з фіксованою і із змінною довжиною структури. Очевидно, що таблиці, які об'єднують структури ідентичних типів, будуть мати фіксовані довжини структур. Необхідність в змінній довжині може виникнути в задачах, подібних до тих, які розглядалися для об'єднань. Як правило таблиці для таких задач і складаються із структур до складу яких входять об'єднання, тобто зводяться до фіксованої (максимальній) довжини структури. Значно рідше зустрічаються таблиці з дійсно змінною довжиною структури. Хоча в таких таблицях і економиться пам'ять, але можливості роботи з такими таблицями обмежені, оскільки за номером структури неможливо визначити її адресу. Таблиці із структурами змінної довжини обробляються тільки послідовно – в порядку зростання номерів структур. Доступ до елемента такої таблиці звичайно здійснюється в два кроки. На першому кроці вибирається постійна частина структури, в якій міститься, – в явному чи неявному вигляді – довжина структури. На другому кроці вибирається змінна частина структури у відповідності з її довжиною. Додавши до адреси поточної структури її довжину, одержують адресу наступної структури.

3. НАПІВСТАТИЧНІ СТРУКТУРИ ДАНИХ

3.1. Характерні особливості напівстатичних структур

Напівстатичні структури даних характеризуються наступними ознаками:

- вони мають змінну довжину і прості процедури її зміни;
- зміна довжини структури відбувається в певних межах, не перевищуючи якогось максимального (граничного) значення.

Якщо напівстатичну структуру розглядати на логічному рівні, то це послідовність даних, зв'язана відносинами лінійного списку. Доступ до елемента може здійснюватися за його порядковим номером.

Фізичне представлення напівстатичних структур даних в пам'яті – це звичайно послідовність комірок в пам'яті, де кожний наступний елемент розташований в пам'яті в наступній комірці. Фізичне представлення може мати також вид одно-направленого зв'язного списку (ланцюжки), де кожний наступний елемент адресується покажчиком, який знаходиться в поточному елементі. У цьому випадку обмеження на довжину структури менш строгі.

3.2. Стеки

Стеком називається множина деякої змінної кількості даних, над якою виконуються наступні операції:

- Поповнення стеку новими даними;
- Перевірка, яка визначає чи стек пустий;
- Перегляд останніх добавлених даних;
- Знищення останніх добавлених даних.

На основі такого функціонального опису, можна сформувати логічний опис. Стек – це такий послідовний список із змінній довжиною, включення і виключення елементів з якого виконуються тільки з одного боку списку. Застосовуються і інші назви стеку – магазин, пам'ять що функціонує за принципом LIFO (Last – In – First – Out – „останнім прийшов – першим вийшов”).

Самий „верхній” елемент стеку, тобто останній добавлений і ще не знищений, відіграє особливу роль: саме його можна модифікувати й знищувати. Цей елемент називають вершиною стеку. Іншу частину стеку називають тілом стеку. Тіло стеку, само собою, є стеком: якщо виключити зі стеку його вершину, то тіло перетворюється в стек.

Основні операції над стеком – включення нового елемента (**push** – заштовхувати) і виключення елемента зі стеку (**pop** – вискакувати).

Корисними можуть бути також допоміжні операції:

- визначення поточної кількості елементів в стеку;
- очищення стеку;
- „неруйнуюче" читання елемента з вершини стека, яке може бути реалізоване, як комбінація основних операцій – виключити елемент зі стеку та включити його знову в стек.

При представленні стеку в статичній пам'яті для стеку виділяється пам'ять, як для вектора. В описі цього вектора окрім звичайних для вектора параметрів повинен знаходитися також покажчик стеку – адреса вершини стека. Обмеження даного представлення полягає в тому, що розмір стеку обмежений розмірами вектора.

Покажчик стеку може вказувати або на перший вільний елемент стеку, або на останній записаний в стек елемент. Однаково, який з цих двох варіантів вибрати, важливо надалі строго дотримуватися його при обробці стеку.

При занесенні елементу в стек елемент записується на місце, яке визначається покажчиком стеку, потім покажчик модифікується так, щоб він вказував на наступний вільний елемент (якщо покажчик вказує на останній записаний елемент, то спочатку модифікується покажчик, а потім проводиться запис елемента). Модифікація покажчика полягає в надбавці до нього або у відніманні від нього одиниці (стек росте у бік збільшення адреси).

Операція виключення елемента полягає в модифікації покажчика стеку (в напрямку, зворотному модифікації при включенні) і вибірці значення, на яке вказує покажчик стеку. Після вибірки комірка, в якій розміщувався вибраний елемент, вважається вільною.

Операція очищення стеку зводиться до запису в покажчик стеку початкового значення – адреси початку виділеної ділянки пам'яті.

Визначення розміру стека зводиться до обчислення різниці покажчиків: покажчика стеку й адреси початку ділянки.

При зв'язному представленні стеку кожен елемент стеку складається із значення і покажчика, який вказує на попередньо занесений у стек елемент. Зв'язне представлення викликає втрату пам'яті, що викликане наявністю покажчика в кожному елементі стеку, і представляє інтерес тільки у випадку, коли важко визначити максимальний розмір стеку.

Отже, для зв'язного представлення стеку потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Для виконання операцій над стеком потрібен один покажчик на вершину стеку. Створення пустого стеку полягатиме у присвоєнні покажчику на вершину нульового значення, що означатиме, що стек пустий.

Послідовність кроків для добавлення елементу в стек складається з декількох кроків:

1. Виділити пам'ять під новий елемент стеку;
2. Занесення значення в інформаційне поле;
3. Встановлення зв'язку між ним і „старою" вершиною стеку;
4. Переміщення вершини стеку на новий елемент:.

Вилучення елементу зі стеку також проводять за кілька кроків:

1. Зчитування інформації з інформаційного поля вершини стеку;
2. Встановлення на вершину стеку допоміжного покажчика;
3. Переміщення покажчика вершини стеку на наступний елемент;
4. Звільнення пам'яті, яку займає „стара" вершина стеку.

3.3. Черга

Чергою називається множина змінної кількості даних, над якою можна виконувати наступні операції:

- Поповнення черги новими даними;
- Перевірка, яка визначає чи пуста черга;
- Перегляд перших добавлених даних;
- Знищення самих перших добавлених даних.

На основі такого функціонального опису, можна сформувати логічний опис. Чергою FIFO (First – In – First – Out – „першим прийшов – першим вийшов"). називається такий послідовний список із змінній довжиною, в якому включення елементів виконується тільки з одного боку списку (хвіст черги), а виключення – з другого боку (голова черги).

Основні операції над чергою – ті ж, що і над стеком – включення, виключення, визначення розміру, очищення, „неруйнуюче" читання.

При представленні черги вектором в статичній пам'яті на додаток до звичайних для опису вектора параметрів в ньому повинні знаходитися два покажчики: на голову і на хвіст черги. При включенні елемента в чергу елемент записується за адресою, яка визначається покажчиком на хвіст, після чого цей покажчик збільшується на одиницю. При виключенні елемента з черги вибирається елемент, що адресується покажчиком на голову, після чого цей покажчик зменшується на одиницю.

Очевидно, що з часом покажчик на хвіст при черговому включенні елемента досягне верхньої межі тієї ділянки пам'яті, яка виділена для черги. Проте, якщо операції включення чергувати з операціями виключення елементів, то в початковій частині відведеної під чергу пам'яті є вільне місце. Для того, щоб місця, займані виключеними елементами, могли бути повторно використані, черга замикається в кільце: покажчики (на початок і на кінець), досягнувши

кінця виділеної області пам'яті, перемикаються на її початок. Така організація черги в пам'яті називається кільцевою чергою. Можливі, звичайно, і інші варіанти організації: наприклад, всякий раз, коли покажчик кінця досягне верхньої межі пам'яті, зсовувати всі не порожні елементи черги до початку ділянки пам'яті, але як цей, так і інші варіанти вимагають переміщення в пам'яті елементів черги і менш ефективні, ніж кільцева черга.

У початковому стані покажчики на голову і хвіст вказують на початок ділянки пам'яті. Рівність цих двох покажчиків є ознакою порожньої черги. Якщо в процесі роботи з кільцевою чергою кількість операцій включення перевищує кількість операцій виключення, то може виникнути ситуація, в якій покажчик кінця „наздожене" покажчик початку. Це ситуація заповненої черги, але якщо в цій ситуації покажчики порівняються, ця ситуація буде така ж як при порожній черзі. Для розрізнення цих двох ситуацій до кільцевої черги пред'являється вимога, щоб між покажчиком кінця і покажчиком початку залишався „проміжок" з вільних елементів. Коли цей „проміжок" скорочується до одного елемента, черга вважається заповненою і подальші спроби запису в неї блокуються. Очищення черги зводиться до запису одного і того ж (не обов'язково початкового) значення в обидва покажчики. Визначення розміру полягає в обчисленні різниці покажчиків з урахуванням кільцевої природи черги.

При зв'язному представленні черги кожен елемент черги складається із значення і покажчика, який вказує на попередньо занесений у чергу елемент.

Зв'язне представлення викликає втрату пам'яті, що викликане наявністю покажчика в кожному елементі черги, і представляє інтерес тільки у випадку, коли важко визначити максимальний розмір черги. Для зв'язного представлення черги потрібно, щоб кожен його елемент описувався структурою, яка поєднує дані і покажчик на наступний елемент.

Для виконання операцій над чергою потрібно два покажчики: на голову і хвіст черги. Створення пустої черги полягатиме у присвоєнні покажчикам на голову і хвіст черги нульових значень, що означатиме, що черга пуста.

Послідовність кроків для добавлення елемента в кінець черги складається з декількох кроків:

1. Виділити пам'ять під новий елемент черги;
2. Занесення значення в інформаційне поле;
3. Занесення нульового значення в покажчик;
4. Встановлення зв'язку між ним і останнім елементом черги і новим, враховуючи випадок пустої черги;
5. Переміщення покажчика кінця черги на новий елемент.

Вилучення елементу з черги також проводять за кілька кроків:

1. Зчитування інформації з інформаційного поля голови черги;
2. Встановлення на голову черги допоміжного покажчика;
3. Переміщення покажчика початку черги на наступний елемент;
4. Звільнення пам'яті, яку займав перший елемент черги.

В реальних задачах іноді виникає необхідність у формуванні черг, відмінних від приведених структур. Порядок вибірки елементів з таких черг визначається пріоритетами елементів. Пріоритет в загальному випадку може бути представлений числовим значенням, яке обчислюється або на підставі значень яких-небудь полів елемента, або на підставі зовнішніх чинників. Так попередньо наведені структури стек і черги можна трактувати як пріоритетні черги, в яких пріоритет елемента залежить від часу його включення в структуру. При вибірці елемента всякий раз вибирається елемент з щонайбільшим пріоритетом.

Черги з пріоритетами можуть бути реалізовані на лінійних структурах – в суміжному або зв'язному представленні. Можливі черги з пріоритетним включенням – в яких послідовність елементів черги весь час підтримується впорядкованою, тобто кожний новий елемент включається на те місце в послідовності, яке визначається його пріоритетом, а при виключенні завжди вибирається елемент з голови. Можливі і черги з пріоритетним виключенням – новий елемент включається завжди в кінець черги, а при виключенні в черзі шукається (цей пошук може бути тільки лінійним) елемент з максимальним пріоритетом і після вибірки вилучається з послідовності. І в тому, і в іншому варіанті потрібний пошук, а якщо черга розміщується в статичній пам'яті – ще і переміщення елементів.

3.4. Деки

Дек – особливий вид черги. Дек (deq – double ended queue, тобто черга з двома кінцями) – це такий послідовний список, в якому як включення, так і виключення елементів, може здійснюватися з будь-якого з двох кінців списку. Так само можна сформулювати поняття деку, як стек, в якому включення і виключення елементів може здійснюватися з обох кінців.

Деки рідко зустрічаються у своєму первісному визначенні. Окремий випадок деку – дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури деку аналогічні логічній і фізичній структурі кільцевої черги. Проте, стосовно деку доцільно говорити не про голову і хвіст, а про лівий і правий кінець.

Над деком доступні наступні операції:

- включення елемента праворуч;

- включення елемента ліворуч;
- виключення елемента з права;
- виключення елемента з ліва;
- визначення розміру;
- очищення.

Фізична структура деку в статичній пам'яті ідентична структурі кільцевої черги.

3.5. Лінійні списки

Лінійні списки є узагальненням попередніх структур; вони дозволяють представити множину так, щоб кожний елемент був доступний і при цьому не потрібно було б зачіпати деякі інші.

Списки є досить гнучкою структурою даних, так як їх легко зробити більшими або меншими, і їх елементи доступні для вставки або вилучення в будь-якій позиції списку. Списки також можна об'єднувати або розділяти на менші списки.

Лінійний список – це скінчена послідовність однотипних елементів (вузлів), можливо, з повторенням. Кількість елементів у послідовності називається довжиною списку. Вона в процесі роботи програми може змінюватися. Лінійний список L, що складається з елементів $d_1, d_2, \ldots, d_n$, які мають однаковий тип, записують у вигляді $L = \langle d_1, d_2, \ldots, d_n \rangle$, або зображають графічно.

Важливою властивістю лінійного списку є те, що його елементи можна лінійно впорядкувати у відповідності з їх позицією в списку.

Для формування абстрактного типу даних на основі математичного визначення списку потрібно задати множину операторів, які виконуються над об'єктами типу список. Проте не існує однієї множини операторів, які виконуються над списками, які задовольняють відразу всі можливі застосування.

Найчастіше зі списками доводиться виконувати такі операції:

- Знайти елемент із заданою властивістю;
- Визначити i-й елемент у лінійному списку;
- Внести додатковий елемент до або після вказаного вузла;
- Вилучити певний елемент списку;
- Впорядкувати вузли лінійного списку в певному порядку.

У реальних мовах програмування не існує якої-небудь структури даних для зображення лінійного списку так, щоб усі операції над ним виконувалися в однаковій мірі ефективно. Тому при роботі з лінійними списками важливе значення має подання лінійних списків, які використовуються в програмі,

таким чином, щоб була забезпечена максимальна ефективність і за часом виконання програми, і за обсягом потрібної їй пам'яті.

Лінійний список є послідовність об'єктів. Позиція елемента в списку має інший тип даних, відмінний від типу даних елемента списку, і цей тип залежить від конкретної фізичної реалізації.

Над лінійним списком допустимі наступні операції.

Операція вставки – вставляє елемент в конкретну позицію в списку, переміщуючи елементи від цієї позиції і далі в наступну, більш вищу позицію.

Операція локалізації – повертає позицію об'єкта в списку. Якщо в списку об'єкт зустрічається декілька разів, то повертається позиція першого від початку списку об'єкта. Якщо об'єкта немає в списку, то повертається значення, яке рівне довжині списку, збільшене на одиницю.

Операція вибірки елемента з списку – повертає елемент, який знаходиться в конкретній позиції списку. Результат не визначений, якщо в списку немає такої позиції.

Операція вилучення – вилучає елемент в конкретній позиції зі списку. Результат невизначений, якщо в списку немає вказаної позиції.

Операції вибірки попереднього і наступного елемента – повертають відповідно наступній і попередній елемент списку відносно конкретної позиці в списку.

Функція очистки списку робить список пустим.

Основні методи зберігання лінійних списків поділяються на методи послідовного і зв'язного зберігання. При виборі способу зберігання в конкретній програмі слід враховувати, які операції і з якою частотою будуть виконуватися над лінійними списками, вартість їх виконання та обсяг потрібної пам'яті для зберігання списку.

Найпростіша форма представлення лінійного списку — це вектор. Визначивши таким чином список можна по чергово звертатися до них в циклі і виконувати необхідні дії. Однак при такому представленні лінійного списку не вдасться уникнути фізичного переміщення елементів, якщо потрібно добавляти нові елементи, або вилучати існуючі. Набагато швидше вилучати елементи можна за допомогою простої схеми чистки пам'яті. Замість вилучення елементів із списку, їх помічають як невикористані.

Більш складною організацією при роботі зі списками є розміщення в масиві декількох списків або розміщення списку без прив'язки його початку до першого елемента масиву.

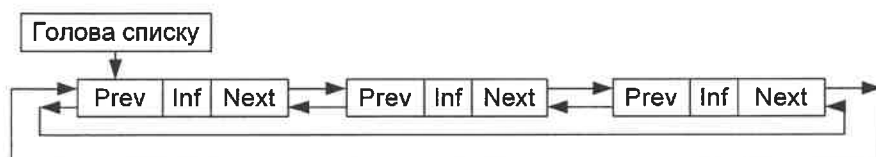
При зв'язному представленні лінійного списку кожен його елемент складається із значення і покажчика, який вказує на наступний елемент у списку.

На наступному рисунку приведена структура однозв'язного списку. Кожний список повинен мати особливий елемент, який називається покажчиком на початок списку, або головою списку.

Проте, обробка однозв'язного списку не завжди зручна, оскільки відсутня можливість просування в протилежну сторону. Таку можливість забезпечує двох-зв'язний список, кожний елемент якого містить два покажчики: на наступний і попередній елементи списку.

Для зручності обробки списку додають ще один особливий елемент – покажчик кінця списку. Наявність двох покажчиків в кожному елементі ускладнює список і приводить до додаткових витрат пам'яті, але в той же час забезпечує більш ефективне виконання деяких операцій над списком.

Різновидом розглянутих видів лінійних списків є кільцевий список, який може бути організований на основі як однозв'язного, так і двох-зв'язного списків. При цьому в однозв'язному списку покажчик останнього елемента повинен вказувати на перший елемент; в двох-зв'язному списку в першому і останньому елементах відповідні покажчики змінюються.

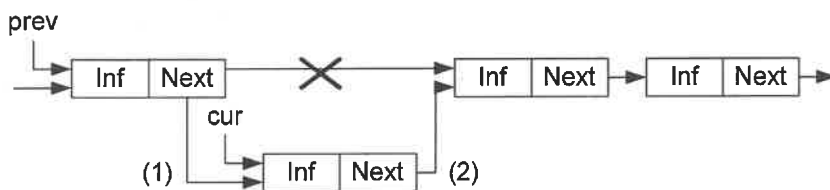


При роботі з такими списками дещо спрощуються деякі процедури, проте, при перегляді такого списку слід приймати деякі запобіжні засоби, щоб не потрапити в нескінченний цикл.

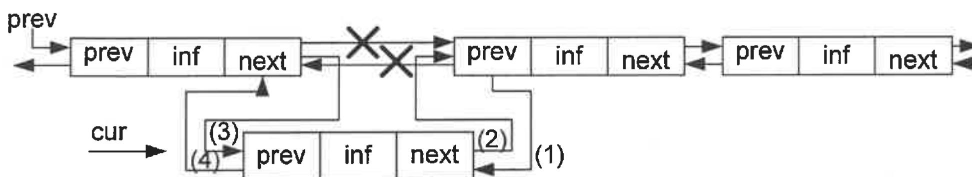
В пам'яті список є сукупністю опису однакових за розміром і форматом структур, які розміщені довільно в деякій ділянці пам'яті і пов'язані одна з одною в лінійно впорядкований ланцюжок за допомогою покажчиків. Структура містить інформаційні поля і поля покажчиків на сусідні елементи списку, причому деякими полями інформаційної частини можуть бути покажчики на блоки пам'яті з додатковою інформацією, що відноситься до елемента списку.

Розглянемо деякі прості операції над лінійними списками.

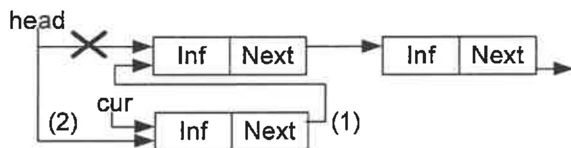
Вставка елемента в середину однозв'язного списку:



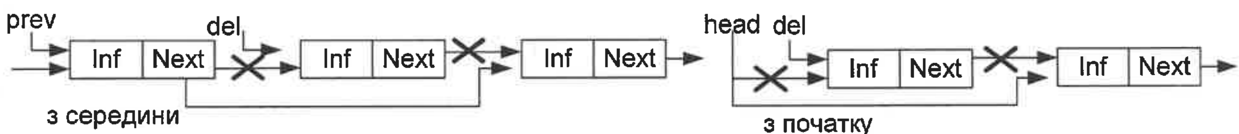
Вставка елемента в двох-зв'язний список:



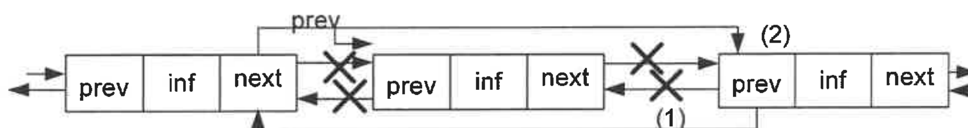
Наведені приклади забезпечують вставку в середину списку, але не можуть бути застосовані для вставки на початок списку. При такій операції повинен модифікуватися покажчик на початок списку:



Видалення елемента з однозв'язного списку для двох варіантів – з середини і з голови:



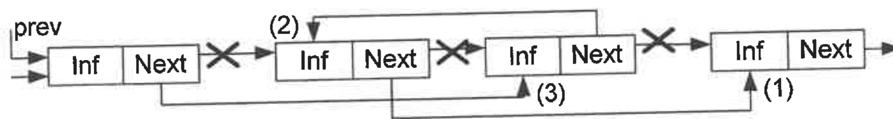
Видалення елемента з двох-зв'язного списку вимагає корекції більшої кількості покажчиків:



Процедура видалення елемента з двох-зв'язного списку виявиться навіть простішою, ніж для однозв'язного, оскільки в ній не потрібно шукати попередній елемент, він вибирається за покажчиком назад.

Змінність динамічних структур даних допускає не тільки зміни розміру структури, але і зміни зв'язків між елементами. Для зв'язних структур зміна зв'язків не вимагає пересилки даних в пам'яті, а тільки зміни покажчиків в елементах зв'язної структури. В якості прикладу приведена перестановка двох сусідніх елементів списку. В алгоритмі перестановки в однозв'язному списку

виходили з того, що відома адреса елемента, який передує парі, в якій проводиться перестановка. В приведеному алгоритмі також не враховується випадок перестановки початкових елементів списку.



У процедурі перестановки для двох-зв'язного списку неважко врахувати і перестановку на початку списку.

3.6. Мультисписки

В програмних системах, які обробляють об'єкти складної структури, можуть вирішуватися різні підзадачі, кожна з яких вимагає обробки можливо не всієї множини об'єктів, а лише якоїсь його підмножини.

Для того, щоб при вибірці кожної підмножини не виконувати повний перегляд з відсіванням записів, які до необхідної підмножини не відносяться, в кожний запис включаються додаткові поля посилань, кожне з яких зв'язує в лінійний список елементи відповідної підмножини. В результаті виходить багато-зв'язковий список або мультисписок, кожний елемент якого може входити одночасно в декілька однозв'язних списків.

До переваг мультисписків крім економії пам'яті (при множині списків інформаційна частина існує в єдиному екземплярі) слід віднести також цілісність даних – в тому сенсі, що всі підзадачі працюють з однією і тією ж версією інформаційної частини і зміни в даних, зроблені одній підзадачей негайно стають доступними для іншої підзадачі.

Кожна підзадача працює з своєю підмножиною як з лінійним списком, використовуючи для цього певне поле зв'язку. Специфіка мультисписку виявляється тільки в операції виключення елемента із списку. Виключення елемента з якого-небудь одного списку ще не означає необхідності видалення елемента з пам'яті, оскільки елемент може залишатися у складі інших списків. Пам'ять повинна звільнятися тільки у тому випадку, коли елемент вже не входить ні в один з приватних списків мультисписку. Звичайно задача видалення спрощується тим, що один з приватних списків є головним – в нього обов'язково входять всі наявні елементи. Тоді виключення елемента з будь-якого неголовного списку полягає тільки в зміні покажчиків, але не в звільненні пам'яті. Виключення ж з головного списку вимагає не тільки звільнення пам'яті, але і зміні покажчиків як в головному списку, так і у всіх неголовних списках, в які елемент, що видаляється, входив.

3.7. Стрічки

Стрічка – це лінійно впорядкована послідовність символів, які належать до скінченої множини символів, яка називається алфавітом.

Стрічки мають наступні важливі властивості:

- їхня довжина, як правило, змінна, хоч алфавіт фіксований;
- звичайне звернення до символів стрічки йде з будь-якого одного боку послідовності (важлива впорядкованість послідовності, а не її індексація);
- метою доступу до стрічки є на окремий її елемент, а ланцюжок символів.

Кажучи про стрічки, звичайно мають на увазі текстові стрічки – стрічки, що складаються з символів, які входять в алфавіт якої-небудь вибраної мови, цифр, розділових знаків і інших службових символів. Текстова стрічка є найбільш універсальною формою представлення будь-якої інформації.

Хоча стрічки й розглядаються в частині, яка присвячена напівстатичним структурам даних, в тих або інших конкретних задачах змінність стрічок може варіюватися від повної її відсутності до практично необмежених можливостей зміни. Орієнтація на ту чи іншу міру мінливості стрічок визначає і фізичне представлення їх в пам'яті і особливості виконання операцій над ними. В більшості мов програмування стрічки представляються саме як напівстатичні структури.

Базовими операціями над стрічками є:

- визначення довжини стрічки;
- присвоєння стрічки;
- конкатенація (зчеплення) стрічок;
- виділення підстрічки;
- пошук входження.

Операція визначення довжини стрічки має вид функції, яка повертає значення – ціле число – поточна кількість символів в стрічці. Операція присвоєння має такий же сенс, що і для інших типів даних.

Операція порівняння стрічок має такий же сенс, що і для інших типів даних. Порівняння стрічок проводиться за наступними правилами. Порівнюються перші символи двох стрічок. Якщо символи не рівні, то стрічка, що містить символ, місце якого в алфавіті ближче до початку, вважається меншою. Якщо символи рівні, порівнюються другі, треті і т.д. символи. При досягненні кінця в одній з стрічок стрічка меншої довжини вважається меншою. При рівності довжин стрічок і попарній рівності всіх символів в них стрічки вважаються рівними.

Результатом операції зчеплення двох стрічок є стрічка, довжина якої рівна сумарній довжині стрічок-операндів, а значення відповідає значенню першого операнда, за яким безпосередньо слідує значення другого операнда.

Операція виділення підстрічки виділяє з початкової стрічки послідовність символів, починаючи із заданої позиції, із заданою довжиною.

Операція пошуку входження знаходить місце першого входження підстрічки еталону в початкову стрічку. Результатом операції може бути номер позиції в початковій стрічці, з яким починається входження еталону або покажчик на початок входження. У разі відсутності входження результатом операції повинне бути деяке спеціальне значення, наприклад, від'ємний номер позиції або порожній покажчик.

Найпростішим способом є представлення стрічки у вигляді вектора постійної довжини. При цьому в пам'яті відводиться фіксована кількість байт, в які записуються символи стрічки. Якщо стрічка менша відведеного під неї вектора, то зайві місця заповнюються пропусками, а якщо стрічка виходить за межі вектора, то зайві (праві) символи повинні бути відкинуті.

Можливе представлення стрічки вектором змінної довжини з ознакою завершення. Цей і всі подальші за ним методи враховують змінну довжину стрічок. Ознака завершення – це особливий символ, який належить до алфавіту (таким чином, корисний алфавіт виявляється меншим на один символ), і займає ту ж кількість розрядів, що і всі інші символи. Витрати пам'яті при цьому способі складають 1 символ на рядок.

Окрім ознаки завершення можна використати лічильник символів – це ціле число, і для нього відводиться достатня кількість бітів, щоб їх з надлишком вистачало для представлення довжини найдовшої стрічки, яку можна представити. При використовуванні лічильника символів можливий довільний доступ до символів в межах стрічки.

Представлення стрічок списком у пам'яті забезпечує гнучкість у виконанні різноманітних операцій над ними (зокрема, операцій включення і виключення окремих символів і цілих ланцюжків) і використовування системних засобів управління пам'яттю при виділенні необхідного об'єму пам'яті для стрічки. Проте, при цьому виникають додаткові затрати пам'яті. Іншим недоліком такого представлення стрічок є те, що логічно сусідні елементи стрічки не є фізично сусідніми в пам'яті. Це ускладнює доступ до груп елементів стрічки в порівнянні з доступом у векторному представленні.

При представленні стрічки однозв'язним лінійним списком кожний символ стрічки представляється у вигляді елемента зв'язного списку; елемент містить

код символу і покажчик на наступний елемент. Одностороннє зчеплення представляє доступ тільки в одному напрямі уздовж стрічки.

При використанні двох-зв'язних лінійних списків у кожний елемент списку додається також покажчик на попередній елемент. Двостороннє зчеплення допускає двосторонній рух уздовж списку, що може значно підвищити ефективність виконання деяких стрічкових операцій.

Блочно-зв'язне представлення стрічок дозволяє в більшості операцій уникнути витрат, які пов'язані з управлінням динамічною пам'яттю, але в той же час забезпечує достатньо ефективне використовування пам'яті при роботі з стрічками змінної довжини.

4. ДИНАМІЧНІ СТРУКТУРИ ДАНИХ

4.1. Зв'язне представлення даних в пам'яті

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

Оскільки елементи динамічної структури розташовуються за не передбачуваними адресами пам'яті, адресу елемента такої структури не можна обчислити за адресою початкового або попереднього елемента. Для встановлення зв'язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв'язки між елементами. Таке представлення даних в пам'яті називається зв'язним. Елемент динамічної структури складається з двох полів:

- інформаційного поля або поля даних, в якому містяться ті дані, заради яких і створюється структура;
- поле зв'язку, в якому міститься один або декілька покажчиків, які зв'язують даний елемент з іншими елементами структури.

Коли зв'язне представлення даних використовується для вирішення прикладної задачі, для кінцевого користувача „видимим" робиться тільки вміст інформаційного поля, а поле зв'язку використовується тільки програмістом-розробником.

Переваги зв'язного представлення даних:

- можливість забезпечення значної змінності структур;
- розмір структури обмежується тільки доступним об'ємом машинної пам'яті;
- при зміні логічної послідовності елементів структури потрібно виконати не переміщення даних в пам'яті, а тільки корекцію покажчиків.

Разом з тим зв'язне представлення не позбавлене й недоліків, основні з яких:

- робота з покажчиками вимагає більш високої кваліфікації від програміста;

- на поля зв'язку витрачається додаткова пам'ять;
- доступ до елементів зв'язної структури може бути менш ефективним за часом.

Останній недолік є найбільш серйозним і саме ним обмежується застосування зв'язного представлення даних. Якщо в суміжному представленні даних для обчислення адреси будь-якого елемента у всіх випадках достатньо номера елемента і інформації, яка міститься в описі структури, то для зв'язного представлення адреса елемента не може бути обчислена з початкових даних. Опис зв'язної структури містить один або декілька покажчиків, які дозволяють увійти до структури, далі пошук необхідного елемента виконується проходженням ланцюжком покажчиків від елемента до елемента. Тому зв'язне представлення практично ніколи не застосовується в задачах, де логічна структура даних має вигляд вектора або масиву – з доступом за номером елемента, але часто застосовується в задачах, де логічна структура вимагає іншої початкової інформації доступу (таблиці, списки, дерева і т.д.).

5. НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ

5.1. Графи

Граф – це складна нелінійна багато-зв'язна динамічна структура, що відображає властивості і зв'язки складного об'єкту.

Ця багато-зв'язна структура має наступні властивості:

- на кожний елемент (вузол, вершину) може бути довільна кількість посилань;
- кожний елемент може мати зв'язок з будь-якою кількістю інших елементів;
- кожний зв'язок (ребро, дуга) може мати напрям і вагу.

У вузлах графа міститься інформація про елементи об'єкту. Зв'язки між вузлами задаються ребрами графа. Ребра графа можуть мати спрямованість, тоді вони називаються орієнтованими, в іншому випадку – неорієнтовані. Граф, усі зв'язки якого орієнтовані, називається орієнтованим графом; граф зі всіма неорієнтованими зв'язками – неорієнтованим графом; граф із зв'язками обох типів – змішаним графом.

Існує два основні методи представлення графів в пам'яті комп'ютера: матричний і зв'язними нелінійними списками. Вибір методу представлення залежить від природи даних і операцій, що виконуються над ними. Якщо задача вимагає великої кількості включень і виключень вузлів, то доцільно представляти граф зв'язними списками; інакше можна застосувати і матричне представлення.

При використанні матриць суміжності їхні елементи представляються в пам'яті комп'ютера елементами масиву. При цьому, для простого графа матриця складається з нулів і одиниць, для мультиграфа – з нулів і цілих чисел, які

вказують кратність відповідних ребер, для зваженого графа – з нулів і дійсних чисел, які задають вагу кожного ребра.

Орієнтований граф представляється зв'язним нелінійним списком, якщо він часто змінюється або якщо півміри входу і виходу його вузлів великі.

Багато-зв'язна структура – граф – знаходить широке застосування при організації банків даних, управлінні базами даних, в системах програмного імітаційного моделювання складних комплексів, в системах штучного інтелекту, в задачах планування і в інших сферах.

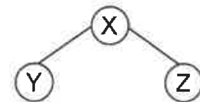
5.2. Дерева

Дерево – це граф, який характеризується наступними властивостями:

- Існує єдиний елемент (вузол або вершина), на який не посилається ніякий інший елемент, – він називається коренем.
- Починаючи з кореня і слідуючи по певному ланцюжку покажчиків, що містяться в елементах, можна здійснити доступ до будь-якого елемента структури.
- На кожний елемент, крім кореня, є єдине посилання, тобто кожний елемент адресується єдиним покажчиком.

Назва „дерево" виникла з логічної еквівалентності дерево-видної структури абстрактному дереву з теорії графів. Лінія зв'язку між парою вузлів дерева називається гілкою. Ті вузли, які не посилаються ні на які інші вузли дерева, називаються листям. Вузол, що не є листком або коренем, вважається проміжним або вузлом галуження.

В багатьох застосування відносний порядок проходження вершин на кожному окремому ярусі має певне значення. При представленні дерева в пам'яті комп'ютера такий порядок вводиться автоматично, навіть якщо він сам по собі довільний. Порядок проходження вершин на деякому ярусі можна легко ввести, позначаючи одну вершину як першу, іншу – як другу і т.д. Замість впорядковування вершин можна задавати порядок на ребрах. Якщо в орієнтованому дереві на кожному ярусі заданий порядок проходження вершин, то таке дерево називається впорядкованим деревом.



Введемо ще деякі поняття, пов'язані з деревами. Вузол X називається предком, або батьком, а вузли Y і Z називаються нащадками, або синами, їх, відповідно, між собою називають братами. Причому, лівий син є старшим сином, а правий – молодшим. Кількість піддерев даної вершини називається мірою цієї вершини.

Якщо з дерева прибрати коріння і ребра, що сполучають коріння з вершинами першого ярусу, то вийде деяка множина незв'язаних дерев. Множина незв'язаних дерев називається лісом.

Є ряд способів графічного зображення дерев. Перший спосіб полягає у використовуванні для зображення піддерев відомого методу діаграм Венна, другий – методу дужок, що вкладаються одна в одну, третій спосіб – це спосіб, який використовується при складанні змісту книг. Останній спосіб, що базується на форматі з нумерацією рівнів, схожий з методами, які використовуються в мовах програмування. При застосуванні цього формату кожній вершині приписується числовий номер, який повинен бути менший номерів, приписаних кореневим вершинам приєднаних до неї піддерев.

Повне дерево містить максимально можливу кількість вузлів на кожному рівні, крім нижнього. Повні дерева мають ряд важливих властивостей.

По-перше, це найкоротші дерева, які можуть містити задану кількість вузлів. Досить корисна властивість повних дерев полягає в тому, що вони можуть бути дуже компактно записані в масивах. Якщо пронумерувати вузли в „природному" порядку, зверху вниз і зліва направо, то можна помістити елементи дерева в масив у цьому ж порядку.

Існують m-арні дерева, тобто такі дерева, в яких півміра виходу кожної вершини менша або рівна m. Якщо півміра виходу кожної вершини в точності рівна або m, або нулю, то таке дерево називається повним m-арним деревом. При $m=2$ такі дерева називаються відповідно бінарними, або повними бінарними.

Представити m-арне дерево в пам'яті комп'ютера складно, так як кожен елемент дерева повинен містити стільки покажчиків, скільки ребер, виходить з вузла. Це приведе до підвищеної витрати пам'яті, різноманітності початкових елементів і ускладнить алгоритми обробки дерева. Тому m-арні дерева, ліс необхідно привести до бінарних для економії пам'яті і спрощенню алгоритмів. Усі вузли бінарного дерева представляються в пам'яті однотипними елементами з двома покажчиками, крім того, операції над бінарними деревами виконуються просто і ефективно.

Правило побудови бінарного дерева з будь-якого дерева:

1. В кожному вузлі залишити тільки гілку до старшого сина;

2. З'єднати горизонтальними ребрами всіх братів одного батька;

3. Таким чином перебудувати дерево за правилом:

лівий син – вершина, розташована під даною;

правий син – вершина, розташована праворуч від даної (тобто на одному ярусі з нею).

4. Розвернути дерево так, щоб усі вертикальні гілки відображали лівих синів, а горизонтальні – правих.

У результаті перетворення будь-якого дерева, в бінарне, виходить дерево у вигляді лівого піддерева, підвішеного до кореня.

У процесі перетворення правий покажчик кожного вузла бінарного дерева буде вказувати на сусіда по рівню. Якщо такого немає, то правий покажчик – **NULL**. Лівий покажчик буде вказувати на вершину наступного рівня. Якщо такої немає, то покажчик встановлюється на **NULL**.

Описаний вище метод представлення довільних впорядкованих дерев за допомогою бінарних дерев можна узагальнити на представлення довільного впорядкованого лісу.

Правило побудови бінарного дерева з лісу: корені всіх піддерев лісу з'єднати горизонтальними зв'язками. В отриманому дереві вузли в даному прикладі будуть розташовуватися на трьох рівнях. Далі перебудовувати по раніше розглянутому плану. В результаті перетворення впорядкованого лісу в бінарне дерево виходить повне бінарне дерево з лівим і правим піддеревом.

Дерева можна представляти за допомогою зв'язних списків і масивів (або послідовних списків).

Частіше всього використовується зв'язне представлення дерев, так як воно дуже сильно нагадує логічне. Зв'язне зберігання полягає в тому, що задається зв'язок від батька до синів. В бінарному дереві є два покажчики, тому зручно вузол представити у вигляді структури в якій *left* – покажчик на ліве піддерево, *right* – покажчик на праве піддерево, *inf* – містить інформацію, яка зв'язана з вершиною і має наперед визначений тип – *data*.

Над деревами визначені наступні основні операції:

1) Пошук вузла із заданим ключем.

2) Додавання нового вузла.

3) Видалення вузла (піддерева).

4) Обхід дерева в певному порядку:

Низхідний обхід;

Змішаний обхід;

Висхідний обхід.

Потрібна вершина в дереві шукається за ключем. Пошук в бінарному дереві здійснюється таким чином.

Нехай побудовано деяке дерево і вимагається знайти вузол з ключем X. Спочатку порівнюємо з X ключ, що знаходиться в корені дерева. У разі рівності пошук закінчений і потрібно повернути покажчик на корінь в якості результату пошуку. Інакше переходимо до розгляду вершини, яка знаходиться зліва внизу, якщо ключ X менший тільки що розглянутого, або справа внизу, якщо ключ X більший тільки що розглянутого. Порівнюємо ключ X з ключем, що міститься в цій вершині, і т.д. Процес завершується в одному з двох випадків:

1) знайдена вершина, що містить ключ, рівний ключу X;

2) в дереві відсутня вершина, до якої потрібно перейти для виконання чергового кроку пошуку.

В першому випадку повертається покажчик на знайдену вершину. В другому – покажчик на вузол, де зупинився пошук (що зручне для побудови дерева).

Для включення запису в дерево перш за все потрібно знайти в дереві ту вершину, до якої можна приєднати нову вершину, відповідну запису, що включається. При цьому впорядкованість ключів повинна зберігатися.

Алгоритм пошуку потрібної вершини, взагалі кажучи, той же самий, що і при пошуку вершини із заданим ключем. Ця вершина буде знайдена в той момент, коли в якості чергового покажчика, який визначає гілку дерева, в якій треба продовжити пошук, виявиться покажчик **NULL**.

В багатьох задачах, пов'язаних з деревами, вимагається здійснити систематичний перегляд всіх його вузлів в певному порядку. Такий перегляд називається проходженням або обходом дерева.

Бінарне дерево можна обходити трьома основними способами: низхідним, змішаним і висхідним (можливі також зворотний низхідний, зворотний змішаний і зворотний висхідний обходи). Прийняті назви методів обходу зв'язані з часом обробки кореневої вершини: До того як оброблено обидва його піддерева, після того, як оброблено ліве піддерево, але до того як оброблено праве, після того, як оброблено обидва піддерева. Використовувані назви методів відображають напрям обходу в дереві: від кореневої вершини вниз до листя – низхідний обхід; від листя вгору до кореня – висхідний обхід, і

змішаний обхід – від найлівішого листка дерева через корінь до найправішого листка.

Схемно алгоритм обходу бінарного дерева відповідно до низхідного способу може виглядати таким чином:

1. В якості чергової вершини взяти корінь дерева. Перейти до пункту 2.

2. Провести обробку чергової вершини відповідно до вимог задачі. Перейти до пункту 3.

3.а) Якщо чергова вершина має обидві гілки, то в якості нової вершини вибрати ту вершину, на яку посилається ліва гілка, а вершину, на яку посилається права гілка, занести в стек; перейти до пункту 2;

3.б) якщо чергова вершина є кінцевою, то вибрати в якості нової чергової вершини вершину із стека, якщо він не порожній, і перейти до пункту 2; якщо ж стек порожній, то це означає, що обхід всього дерева закінчений, перейти до пункту 4;

3.в) якщо чергова вершина має тільки одну гілку, то в якості чергової вершини вибрати ту вершину, на яку ця гілка вказує, перейти до пункту 2.

4. Кінець алгоритму.

Алгоритм істотно спрощується при використовуванні рекурсії. Так, низхідний обхід можна описати таким чином:

1). Обробка кореневої вершини;

2). Низхідний обхід лівого піддерева;

3). Низхідний обхід правого піддерева.

Змішаний обхід можна описати таким чином:

1) Спуститися по лівій гілці із запам'ятовуванням вершин в стеку;

2) Якщо стек порожній те перейти до п.5;

3) Вибрати вершину із стеку і обробити дані вершини;

4) Якщо вершина має правого сина, то перейти до нього; перейти до п.1.

5) Кінець алгоритму.

Рекурсивний змішаний обхід описується таким чином:

1) Змішаний обхід лівого піддерева;

2) Обробка кореневої вершини;

3) Змішаний обхід правого піддерева.

Трудність реалізації висхідного обходу полягає в тому, що на відміну від попереднього методу в цьому алгоритмі кожна вершина запам'ятовується в стеку двічі: вперше – коли обходиться ліве піддерево, і другий раз – коли обходиться праве піддерево. Таким чином, в алгоритмі необхідно розрізняти два види стекових записів: 1-й означає, що в даний момент обходиться ліве піддерево; 2-й – що обходиться праве, тому в стеку запам'ятовується покажчик на вузол і ознаку (код-1 і код-2 відповідно).

Алгоритм висхідного обходу можна представити таким чином:

1) Спуститися по лівій гілці із запам'ятовуванням вершини в стеку як 1-й вид стекових записів;

2) Якщо стек порожній, то перейти до п.5;

3) Вибрати вершину із стека, якщо це перший вид стекових записів, то повернути його в стек як 2-й вид стекових записів; перейти до правого сина; перейти до п.1, інакше перейти до п.4;

4) Обробити дані вершини і перейти до п.2;

5) Кінець алгоритму.

Рекурсивний змішаний обхід описується таким чином:

1). Висхідний обхід лівого піддерева;

2). Висхідний обхід правого піддерева;

3). Обробка кореневої вершини.

Якщо в розглянутих вище алгоритмах поміняти місцями поля покажчики на лівого і правого сина, то отримують процедури зворотного низхідного, зворотного змішаного і зворотного висхідного обходів.

6. МЕТОДИ ШВИДКОГО ДОСТУПУ ДО ДАНИХ

6.1. Хешування даних

Для прискорення доступу до даних можна використовувати попереднє їх впорядкування у відповідності зі значеннями ключів. При цьому можуть використовуватися методи пошуку в упорядкованих структурах даних, наприклад, метод дихотомічного пошуку, що суттєво скорочує час пошуку

даних за значенням ключа. Проте при добавленні нового запису потрібно дані знову впорядкувати. Втрати часу на повторне впорядкування можуть значно перевищувати виграш від скорочення часу пошуку. Тому для скорочення часу доступу до даних використовується так зване випадкове впорядкування або хешування. При цьому дані організуються у вигляді таблиці за допомогою хеш-функції *h*, яка використовується для „обчислення” адреси за значенням ключа.

адрес=h(ключ)   Таблиця

Ключ	поле даних	

поля (мають фіксований тип)

У попередній главі описувався алгоритм інтерполяційного пошуку, який використовує інтерполяцію для пришвидшення пошуку. Порівнюючи шукане значення зі значеннями елементів у відомих точках, цей алгоритм може визначити імовірне розміщення шуканого елемента. По суті, він створює функцію, яка встановлює відповідність між шуканим значенням і індексом позиції, в якій він повинен знаходитися. Якщо перше передбачення помилкове, то алгоритм знову використовує цю функцію, передбачаючи нове розміщення, і так далі, до тих пір, поки шуканий елемент не буде знайдено.

Хешування використовує аналогічний підхід, відображаючи елементи в хеш-таблиці. Алгоритм хешування використовує деяку функцію, яка визначає імовірне розміщення елемента в таблиці на основі значення шуканого елементу.

Ідеальною хеш-функцією є така хеш-функція, яка для будь-яких двох неоднакових ключів дає неоднакові адреси. Підібрати таку функцію можна у випадку, якщо всі можливі значення ключів відомі наперед. Така організація даних носить назву „досконале хешування”.

Наприклад, потрібно запам’ятати декілька записів, кожен з яких має унікальний ключ зі значенням від 1 до 100. Для цього можна створити масив з 100 комірками і присвоїти кожній комірці нульовий ключ. Щоб добавити в масив новий запис, дані з нього просто копіюються у відповідну комірку масиву. Щоб добавити запис з ключем 37, дані з нього копіюються в 37 позицію в масиві. Щоб знайти запис з певним ключем – вибирається відповідна комірка масиву. Для вилучення запису ключу відповідної комірки масиву просто присвоюється нульове значення. Використовуючи цю схему, можна добавити, знайти і вилучити елемент із масиву за один крок.

У випадку наперед невизначеної множини значень ключів і обмеженні розміру таблиці підбір досконалої функції складний. Тому часто використовують хеш-функції, які не гарантують виконанні умови.

Наприклад, база даних співробітників може використовувати в якості ключа ідентифікаційний номер. Теоретично можна було б створити масив, кожна комірка якого відповідала б одному з можливих чисел; але на практиці для цього не вистарчить пам'яті або дискового простору. Якщо для зберігання одного запису потрібно 1 КБ пам'яті, то такий масив зайняв би 1 ТБ (мільйон МБ) пам'яті. Навіть якщо можна було б виділити такий об'єм пам'яті, така схема була б дуже неекономною. Якщо штат фірми менше 10 мільйонів співробітників, то більше 99 процентів масиву буде пустою.

Щоб вирішити цю проблему, схеми хешування відображають потенційно велику кількість можливих ключів на достатньо компактну хеш-таблицю. Якщо на фірмі працює 700 співробітників, можна створити хеш-таблицю з 1000 комірок. Схема хешування встановлює відповідність між 700 записами про співробітників і 1000 позиціями в таблиці. Наприклад, можна розміщати записи в таблиці у відповідності з трьома першими цифрами ідентифікаційного номеру. При цьому запис про співробітника з номером 123456789 буде знаходитися в 123 комірці таблиці.

Очевидно, що оскільки існує більше можливих значень ключа, ніж комірок в таблиці, то деякі значення ключів можуть відповідати одним і тим коміркам таблиці. Даний випадок носить назву „колізія", а такі ключі називаються „ключі-синоніми".

Щоб уникнути цієї потенційної проблеми, схема хешування повинна включати в себе алгоритм вирішення конфліктів, який визначає послідовність дій у випадку, якщо ключ відповідає позиції в таблиці, яка вже зайнята іншим записом.

Усі методи використовують для вирішення конфліктів приблизно однаковий підхід. Вони спочатку встановлюють відповідність між ключем запису і розміщенням в хеш-таблиці. Якщо ця комірка вже зайнята, вони відображають ключ на іншу комірку таблиці. Якщо вона також вже зайнята, то процес повторюється знову до тих пір, поки нарешті алгоритм не знайде пусту комірку в таблиці. Послідовність позицій, які перевіряються при пошуку або вставці елемента в хеш-таблицю, називається тестовою послідовністю.

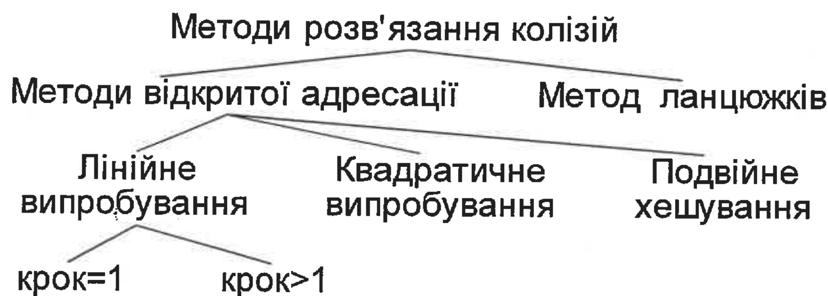
В результаті, для реалізації хешування необхідні три речі:

- Структура даних (хеш-таблиця) для зберігання даних;

- Функція хешування, яка встановлює відповідність між значеннями ключа і розміщенням в таблиці;
- Алгоритм вирішення конфліктів, який визначає послідовність дій, якщо декілька ключів відповідають одній комірці таблиці.

6.1.1. Методи розв'язання колізій

Для розв'язання колізій використовуються різноманітні методи, які в основному зводяться до методів „ланцюжків" і „відкритої адресації".

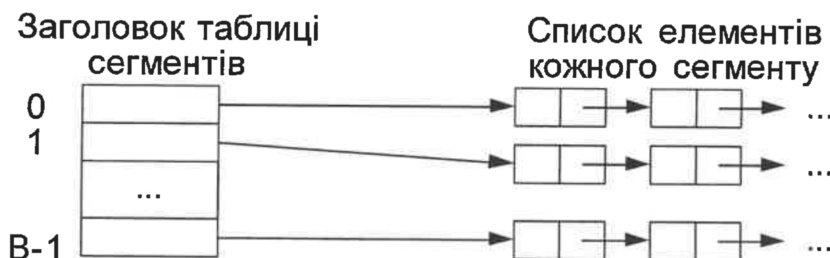


Методом ланцюжків називається метод, в якому для розв'язання колізій у всі записи вводяться покажчики, які використовуються для організації списків – „ланцюжків переповнення". У випадку виникнення колізій при заповненні таблиці в список для потрібної адреси хеш-таблиці добавляється ще один елемент.

Пошук в хеш-таблиці з ланцюжками переповнення здійснюється наступним чином. Спочатку обчислюється адреса за значенням ключа. Потім здійснюється послідовний пошук в списку, який зв'язаний з обчисленим адресом.

Процедура вилучення з таблиці зводиться до пошуку елемента в його вилучення з ланцюжка переповнення.

Схематичне зображення хеш-таблиці при такому методі розв'язання колізій приведене на наступному рисунку.



Якщо сегменти приблизно однакові за розміром, то у цьому випадку списки усіх сегментів повинні бути найбільш короткими при даній кількості сегментів. Якщо вихідна множина складається з N елементів, тоді середня довжина

списків буде рівна *N/B* елементів. Якщо можна оцінити *N* і вибрати *B* якомога ближчим до цієї величини, то в списку буде один-два елементи. Тоді час доступу до елемента множини буде малою постійною величиною, яка залежить від *N*.

Одна з переваг цього методу хешування полягає в тому, що при його використанні хеш-таблиці ніколи не переповнюються. При цьому вставка і пошук елементів завжди виконується дуже просто, навіть якщо елементів в таблиці дуже багато. Із хеш-таблиці, яка використовує зв'язування, також просто вилучати елементи, при цьому елемент просто вилучається з відповідного зв'язного списку.

Один із недоліків зв'язування полягає в тому, що якщо кількість зв'язних списків недостатньо велика, то розмір списків може стати великим, при цьому для вставки чи пошуку елемента необхідно буде перевірити велику кількість елементів списку.

Метод відкритої адресації полягає в тому, щоб, користуючись якимось алгоритмом, який забезпечує перебір елементів таблиці, переглядати їх в пошуках вільного місця для нового запису.

Лінійне випробування зводиться до послідовного перебору елементів таблиці з деяким фіксованим кроком

$$a = h(key) + c*i ,$$

де i – номер спроби розв'язати колізію. При кроці рівному одиниці відбувається послідовний перебір усіх елементів після поточного.

Квадратичне випробування відрізняється від лінійного тим, що крок перебору елементів нелінійно залежить від номеру спроби знайти вільний елемент

$$a = h(key2) + c*i + d*i^2$$

Завдяки нелінійності такої адресації зменшується кількість спроб при великій кількості ключів-синонімів. Проте навіть відносно невелика кількість спроб може швидко привести до виходу за адресний простір невеликої таблиці внаслідок квадратичної залежності адреси від номеру спроби.

Ще один різновид методу відкритої адресації, яка називається подвійним хешуванням, базується на нелінійній адресації, яка досягається за рахунок сумування значень основної і додаткової хеш-функцій.

$$a = h1(key) + i*h2(key).$$

Розглянемо алгоритми вставки і пошуку для методу лінійного випробування.

Вставка:

1. i = 0
2. a = h(key) + i*c
3. Якщо t(a) = вільно, то t(a) = key, записати елемент і зупинитися
4. i = i + 1, перейти до кроку 2

Пошук:

1. i = 0
2. a = h(key) + i*c
3. Якщо t(a) = key, то зупинитися – елемент знайдено
4. Якщо t(a) = вільно, то зупинитися – елемент не знайдено
5. i = i + 1, перейти до кроку 2

Аналогічно можна було б сформулювати алгоритми добавлення і пошуку елементів для будь-якої схеми відкритої адресації. Відмінності будуть лише у виразі, який використовується для обчислення адреси (крок 2).

З процедурою вилучення справа складається не так просто, так як вона в даному випадку не буде оберненою до процедури вставки. Справа в тому, що елементи таблиці знаходяться в двох станах: вільно і зайнято. Якщо вилучити елемент, перевівши його в стан вільно, то після такого вилучення алгоритм пошуку буде працювати некоректно. Нехай ключ елемента, який вилучається, має в таблиці ключі синоніми. У цьому випадку, якщо за ним в результаті розв'язання колізій були розміщені елементи з іншими ключами, то пошук цих елементів після вилучення завжди буде давати негативний результат, так як алгоритм пошуку зупиняється на першому елементі, який знаходиться в стані вільно.

Скоректувати цю ситуацію можна різними способами. Самий простий із них полягає в тому, щоб проводити пошук елемента не до першого вільного місця, а до кінця таблиці. Проте така модифікація алгоритму зведе нанівець весь виграш в прискоренні доступу до даних, який досягається в результаті хешування.

Інший спосіб зводиться до того, щоб відслідкувати адреси всіх ключів-синонімів для ключа елемента, що вилучається, і при необхідності розмістити відповідні записи в таблиці. Швидкість пошуку після такої операції не зменшиться, але затрати часу на саме розміщення елементів можуть виявитися значними.

Існує підхід, який не має перерахованих недоліків. Його суть полягає в тому, що для елементів хеш-таблиці добавляється стан „вилучено". Даний стан в процесі пошуку інтерпретується, як зайнято, а в процесі запису як вільно.

Тепер можна сформулювати алгоритми вставки, пошуку і вилучення для хеш-таблиці, яка має три стани елементів.

Вставка:

1. i = 0
2. a = h(key) + i*c
3. Якщо t(a) = вільно або t(a) = вилучено, то t(a) = key, записати елемент і стоп
4. i = i + 1, перейти до кроку 2

Вилучення:

1. i = 0
2. a = h(key) + i*c
3. Якщо t(a) = key, то t(a) = вилучено, стоп елемент вилучений
4. Якщо t(a) = вільно, то стоп елемент не знайдено
5. i = i + 1, перейти до кроку 2

Пошук:

1. i = 0
2. a = h(key) + i*c
3. Якщо t(a) = key, то стоп – елемент знайдено
4. Якщо t(a) = свободно, то стоп – елемент не знайдено
5. i = i + 1, перейти до кроку 2

Алгоритм пошуку для хеш-таблиці, яка має три стани, практично не відрізняється від алгоритму пошуку без врахування вилучення. Різниця полягає в тому, що при організації самої таблиці необхідно відмічати вільні і вилучені елементи. Це можна зробити, зарезервувавши два значення ключового поля. Інший варіант реалізації може передбачати введення додаткового поля, в якому фіксується стан елемента. Довжина такого поля може складати всього два біти, що достатньо для фіксації одного з трьох станів.

6.1.2. Переповнення таблиці і повторне хешування

Очевидно, що в міру заповнення хеш-таблиці будуть відбуватися колізії і в результаті їх розв'язання методами відкритої адресації чергова адреса може вийти за межі адресного простору таблиці. Щоб це явище відбувалося рідше, можна піти на збільшення розмірів таблиці у порівнянні з діапазоном адрес, які обчислюються хеш-функцією.

З однієї сторони це приведе до скорочення кількості колізій і прискоренню роботи з хеш-таблицею, а з іншої – до нераціональних витрат адресного простору. Навіть при збільшенні таблиці в два рази у порівнянні з областю значень хеш-функції нема гарантій того, що в результаті колізій адреса не перевищить розмір таблиці. При цьому в початковій частині таблиця може залишатися достатньо вільних елементів. Тому на практиці використовують циклічний перехід на початок таблиці.

Розглянемо даний спосіб на прикладі методу лінійного випробування. При обчисленні адреси чергового елементу можна обмежити адресу, взявши в якості такої остачу від цілочисельного ділення адреси на довжину таблиці N.

Вставка:

1. i = 0
2. a = (h(key) + c*i) % N
3. Якщо t(a) = вільно або t(a) = вилучено то t(a) = key, записати елемент і стоп
4. i = i + 1, перейти до кроку 2

В даному алгоритмі не враховується можливість багатократного перевищення адресного простору. Більш коректним буде алгоритм, який використовує зсув адреси на 1 елемент у випадку кожного повторного перевищення адресного простору. Це підвищує імовірність знаходження вільного елемента у випадку повторних циклічних переходів до початку таблиці.

Вставка:

1. i = 0
2. a = ((h(key) + c*i) / n + (h(key) + c*i) % n) % n
3. Якщо t(a) = вільно або t(a) = вилучено, то t(a) = key, записати елемент і стоп
4. i = i + 1, перейти до кроку 2

Розглядаючи можливість виходу за межі адресного простору таблиці, ми не враховували фактори наповненості таблиці й вдалого вибору хеш-функції. При великій наповненості таблиці виникають часті колізії і циклічні переходи на початок таблиці. При невдалому виборі хеш-функції відбуваються аналогічні явища. В найгіршому випадку при повному заповнені таблиці алгоритми циклічного пошуку вільного місця приведуть до зациклювання. Тому при використанні хеш-таблиць необхідно старатися уникати дуже густого заповнення таблиць. Звичайно довжину таблиці вибирають із розрахунку дворазового перевищення передбачуваної максимальної кількості записів. Не завжди при організації хешування можна правильно оцінити потрібну довжину таблиці, тому у випадку великої наповненості таблиці може знадобитися рехешування. У цьому випадку збільшують довжину таблиці, змінюють хеш-функцію і впорядковують дані.

Проводити окрему оцінку густини заповнення таблиці після кожної операції вставки недоцільно, тому можна проводити таку оцінку непрямим способом – за кількістю колізій під час однієї вставки. Достатньо визначити деякий поріг кількості колізій, при перевищенні якого потрібно провести рехешування. Крім того, така перевірка гарантує неможливість зациклювання алгоритму у випадку повторного перегляду елементів таблиці. Розглянемо алгоритм вставки, який реалізує описаний підхід.

Вставка:

1. i = 0
2. a = ((h(key) + c*i) / n + (h(key) + c*i) % n) % n
3. Якщо t(a) = вільно або t(a) = вилучено, то t(a) = key, записати елемент і стоп

4. Якщо i > m , то стоп – потрібно рехешування
5. i = i + 1, перейти до кроку 2
В даному алгоритмі номер ітерації порівнюється з пороговим числом m. Варто зауважити, що алгоритми вставки, пошуку і вилучення повинні використовувати ідентичне утворення адреси чергового запису.

Вилучення:

1. i = 0
2. a = ((h(key) + c*i) / n + (h(key) + c*i) % n) % n
3. Якщо t(a) = key, то t(a) = вилучено і стоп елемент вилучено
4. Якщо t(a) = вільно або i>m, то стоп – елемент не знайдено
5. i = i + 1, перейти до кроку 2

Пошук:

1. i = 0
2. a = ((h(key) + c*i) / n + (h(key) + c*i) % n) % n
3. Якщо t(a) = key, то стоп – елемент знайдено
4. Якщо t(a) = вільно або i>m, то стоп – елемент не знайдено
5. i = i + 1, перейти до кроку 2

6.1.3. Оцінка якості хеш-функції

Як вже було відмічено, дуже важливий правильний вибір хеш-функції. При вдалій побудові хеш-функції таблиця заповнюється більш рівномірно, зменшується кількість колізій і зменшується час виконання операцій пошуку, вставки і вилучення. Для того щоб попередньо оцінити якість хеш-функції можна провести імітаційне моделювання.

Моделювання проводиться наступним чином. Формується вектор цілих чисел, довжина якого співпадає з довжиною хеш-таблиці. Випадково генерується достатньо велика кількість ключів, для кожного ключа обчислюється хеш-функція. В елементах вектора підраховується кількість генерацій даної адреси. За результатами такого моделювання можна побудувати графік розподілу значень хеш-функції. Для отримання коректних оцінок кількість генерованих ключів повинна в декілька разів перевищувати довжину таблиці.

Якщо кількість елементів таблиці достатньо велика, то графік будується не для окремих адрес, а для груп адрес. Великі нерівномірності засвідчують високу імовірність колізій в окремих місцях таблиці. Зрозуміло, така оцінка є наближеною, але вона дозволяє попередньо оцінити якість хеш-функції і уникнути грубих помилок при її побудові.

Оцінка буде більше точною, якщо генеровані ключі будуть більш близькими до реальних ключів, які використовуються при заповненні хеш-таблиці. Для символьних ключів дуже важливо добитися відповідності генерованих кодів

символів тим кодам символів, які є в реальному ключі. Для цього потрібно проаналізувати, які символи можуть бути використані в ключі.

Наприклад, якщо ключ представляє собою прізвище українською мовою, то будуть використані українські букви. Причому перший символ може бути великою буквою, а інші – малими. Якщо ключ представляє собою номерний знак автомобіля, то також нескладно визначити допустимі коди символів в певних позиціях ключа.

Розглянемо більш загальний випадок. Нехай необхідно генерувати ключ із *m* символів з кодами в неперервному діапазоні від *n1* до *n2*.

```
for (i=0; i<m; i++)
    str[i]:=(char)(rand()%(n2-n1)+n1);
```

На практиці можливі варіанти, коли символи в одних позиціях ключа можуть належати до різних діапазонів кодів, причому між цими діапазонами може існувати розрив. Наприклад генерація ключа з *m* символів з кодами в діапазоні від *n1* до *n4* (діапазон має розрив від *n2* до *n3*).

```
for (int i=0; i<m; i++)
{
    x = rand() % ((n4-n3)+(n2-n1));
    if ( x<=(n2-n1) )
        str[i] = (char)(x+n1);
    else
        str[i] = (char)(x+n1+n3-n2);
}
```

Розглянемо ще один конкретний приклад. Нехай відомо, що ключ складається з 7 символів. Із них три перші символи – великі латинські букви, далі йдуть дві цифри, інші – малі латинські.

Приклад: довжина ключа 7 символів;

3 великі латинські (коди 65-90);

2 цифри (коди 48-57);

2 малі латинські (коди 97-122).

```
char key[7];
for (i=0; i<3; i++) key[i] = (char)(rand()%(90-65)+65);
for (i=3; i<5; i++) key[i] = (char)(rand()%(57-48)+57);
for (i=5; i<7; i++) key[i] = (char)(rand{}%(122-97)+97);
```

6.2. Організація даних для прискорення пошуку за вторинними ключами

До тепер розглядалися способи пошуку в таблиці за ключами, які дозволяють однозначно ідентифікувати запис. Такі ключі називають первинними ключами. Можливий варіант організації таблиці, при якому окремий ключ не дозволяє однозначно ідентифікувати запис. Така ситуація часто зустрічається в базах даних. Ідентифікація запису здійснюється за деякою сукупністю ключів. Ключі, які не дозволяють однозначно ідентифікувати запис в таблиці, називаються вторинними ключами.

Навіть при наявності первинного ключа, для пошуку запису можуть використовуватися вторинні. Наприклад, пошукові системи Internet часто організовані як набори записів, які відповідають Web-сторінкам. В якості вторинних ключів для пошуку виступають ключові слова, а сама задача пошуку зводиться до вибірки з таблиці деякої множини записів, які містять потрібні вторинні ключі.

6.2.1. Інвертовані індекси

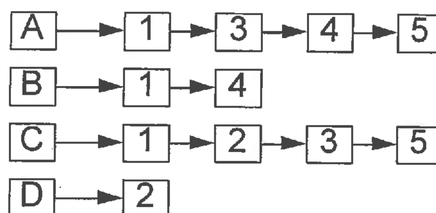
Розглянемо метод організації таблиці з інвертованими індексами. Для таблиці будується окремий набір даних, який містить так звані інвертовані індекси. Допоміжний набір містить для кожного значення вторинного ключа відсортований список адрес записів таблиці, які містять даний ключ.

Пошук здійснюється у допоміжній структурі достатньо швидко, так як фактично відсутня необхідність звернення до основної структури даних. Ділянка пам'яті, яка використовується для індексів, є відносно невеликою у порівнянні з іншими методами організації таблиць.

Таблиця (пряма адресація)

1	A	B	C	
2	C	D		
3	A	C		
4	A	B		
5	A	C		

Структура інфертованих індексів (спискова структура)



Недоліками даної системи є великі затрати часу на складання допоміжної структури даних і її поновлення. Причому ці затрати зростають зі збільшенням об'єму бази даних.

Система інвертованих індексів є досить зручною й ефективною при організації пошуку в великих таблицях.

6.2.2. Бітові карти

Для таблиць невеликого об'єму використовують організацію допоміжної структури даних у вигляді бітових карт. Для кожного значення вторинного ключа записів основного набору даних записується послідовність бітів. Довжина послідовності бітів рівна кількості записів. Кожен біт в бітовій карті відповідає одному значенню вторинного ключа і одному запису. Одиниця означає наявність ключа в запису, а нуль – відсутність.

Основною перевагою такої організації є дуже проста й ефективна організація обробки складних запитів, які можуть об'єднувати значення ключів різними логічними предикатами. У цьому випадку пошук зводиться до виконання логічних операцій запиту безпосередньо над бітовими стрічками й інтерпретації результуючої бітової стрічки. Іншою перевагою є простота поновлення карти при добавленні записів.

Таблиця (пряма адресація)

1	A	B	C	
2	C	D		
3	A	C		
4	A	B		
5	A	C		

Бітові карти

Бітові стрічки

A →	1	0	1	1	1
B →	1	0	0	1	0
C →	1	1	1	0	1
D →	0	1	0	0	0

До недоліків бітових карт варто віднести збільшення довжини стрічки пропорційно довжині файлу. При цьому наповненість карти одиницями зменшується зі збільшенням довжини файлу. Для великої довжини таблиці і ключів, які рідко зустрічаються, бітова карта перетворюється в велику розріджену матрицю, яка складається, в основному, з одних нулів.

1. ПРОСТІ СТРУКТУРИ ДАНИХ 2
1.1. Арифметичні типи 2
1.1.1. Фундаментальні типи в С++ 2
1.2. Перерахований тип 6
1.3. Покажчики 9
2. СТАТИЧНІ СТРУКТУРИ ДАНИХ 13
2.1. Масиви 14
2.2. Розріджені масиви 19
2.3. Множини 24
2.4. Структури 24
2.5. Об’єднання 29
2.6. Бітові типи 32
2.7. Таблиці 33
3. НАПІВСТАТИЧНІ СТРУКТУРИ ДАНИХ 34
3.1. Характерні особливості напівстатичних структур 34
3.2. Стеки 34
3.3. Черга 36
3.4. Деки 38
3.5. Лінійні списки 39
3.6. Мультисписки 43
3.7. Стрічки 44
4. ДИНАМІЧНІ СТРУКТУРИ ДАНИХ 46
4.1. Зв'язне представлення даних в пам'яті 46
5. НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ 47
5.1. Графи 47
5.2. Дерева 48

6. МЕТОДИ ШВИДКОГО ДОСТУПУ ДО ДАНИХ **53**

6.1. Хешування даних **53**
6.1.1. Методи розв'язання колізій 56
6.1.2. Переповнення таблиці і повторне хешування 59
6.1.3. Оцінка якості хеш-функції 61

6.2. Організація даних для прискорення пошуку за вторинними ключами **63**
6.2.1. Інвертовані індекси 63
6.2.2. Бітові карти 64