Для визначення ефективних структур даних розглядаються комбінаторно-імовірнісні методи. Методи визначення показників обчислювальної складності алгоритмів адаптовані та застосовані щодо структур даних. Визначаються показники для найкращого, середнього та найгіршого випадків. Вказані умови застосування показників обчислювальної складності. Наведено приклад застосування методу імовірнісного підрахунку операцій при вирішені задач розробки ефективних структур даних

Ключові слова: структура даних, обчислювальна складність, ефективність, операції з даними, імовірність, показники

Для определения эффективных структур данных рассматриваются комбинаторно-вероятностные методы. Методы определения показателей вычислительной сложности алгоритмов адаптированы применительно к структурам данных. Показатели определяются для лучшего, среднего и худшего случаев. Указаны условия, при которых применимы показатели вычислительной сложности. Приведен пример применения метода вероятностного подсчета операций при решении задач разработки эффективных структур данных

Ключевые слова: структура данных, вычислительная сложность, эффективность, операции над данными, вероятность, показатели

УДК 681.3.06

ОЦІНКА ЧАСОВИХ ХАРАКТЕРИСТИК СТРУКТУР ДАНИХ НА ПРОЕКТНОМУ РІВНІ

В. І. Шинкаренко
Доктор технічних наук, професор*
E-mail: shinkarenko_vi@ua.fm
Д. О. Пєтін
Аспірант*
E-mail: f1.minardi@mail.ru
Г. В. Забула
Аспірант*
E-mail: zabulus12@gmail.com
*Кафедра комп'ютерних інформаційних технологій
Дніпропетровський національний університет залізничного транспорту ім. Академіка В. А. Лазаряна
вул. Лазаряна, 2, м. Дніпропетровськ, Україна, 49010

1. Вступ

За визначенням Н. Вірта, програма – це сукупність алгоритму та структур даних. Визначення відображає важливість проектування структур [1].

Як відомо, можна виділити 4 рівня (етапу) проектування даних. Абстрактний рівень, на якому визначають фізичну сутність, зв'язки між елементами і базові операції над даними, абстрагуючись від структур даних та їх реалізації. Проектний (концептуальний) рівень, який передбачає створення розробленої та обґрунтованої логічного організації даних, з урахуванням вимог щодо ефективності, безпеки та іншим обмеженням. Рівень реалізації, який надає логічну організацію даних в термінах середовища розробки (наприклад, це середовище програмування). Фізичний рівень – розміщення даних на носіях інформації, включаючи допоміжні дані, такі як: адреси, посилання, покажчики, індекси і тощо. Вочевидь, що при вирішенні конкретних завдань проектування не всі етапи і не для всіх даних обов'язкові.

Кожен рівень проектування передбачає і відповідний рівень операцій обробки даних. На абстрактному рівні визначені всі операції над даними, які представлені у вигляді специфікацій відповідних процедур і функцій. На проектному рівні – це операції над структурованими даними. На рівні реалізації операцій визначається базовими операціями середовища розробки (мови програмування), такі як присвоювання, додавання і т. д. На фізичному рівні операції визначені машинними командами процесора, контролерів і тощо. [2].

Розрізняють прості та структуровані дані. Прості – це дані доступ до яких можливий виключно цілком. Структуровані дані складаються з простих даних (елементів), пов'язаних між собою зв'язками, що визначають порядок розташування та можливу послідовність обробки елементів структури.

Найбільш вживаними серед класичних структур даних є такі, як індексований масив, запис, список, дерево, стек, дек та черга. На даний час поширеними структурами є асоціативний масив, колекція та інші структури. В мовах програмування такі структури є наперед визначеними, примітивними.

Проектування та розробка структур даних зводиться або до вибору серед наперед визначених структур, або до конструювання більш складних структур. Конструювання полягає у вкладеності одних структур в інші, тобто елементами структур також виступають структури.

Конструювання використовують за потребою задачі, яка розв'язується алгоритмічно. Приклади сконструйованих структур: масив масивів, дерево списків, стек записів, колекція дерев масивів та інші.

2. Постановка проблеми

У практиці програмування задача розробки структур даних вирішується на інтуїтивному рівні, але коли

до часових характеристик програмного забезпечення висуваються підвищені вимоги слід спиратись на об'єктивні показники ефективності.

Задача наведеної роботи полягала у визначені статичних методів оцінки часових характеристик структур даних, тобто методів, що не потребують виконання програми та статистичних досліджень. При цьому відомі методи визначення показників обчислювальної складності алгоритму слід було адаптувати до вирішення задач оцінки часових характеристик структур даних.

Доцільність визначення показників обчислювальної складності

Ефективність структури даних визначається ефективністю операцій обробки, які реалізовані у вигляді певних алгоритмів та можуть складатись з операцій доступу, зміни порядку та розташування структурованих елементів. Розмежуємо операції обробки даних та переторення даних. Ефективність операцій обробки пов'язана з певними критеріями, такими як: час виконання, потреба пам'яті, надійність, складність, тощо. Для об'єктивного визначення критеріїв слід використовувати значення їх показників.

Розглянемо у якості прикладу структури даних довідник, інформація у якому відсортована за алфавітом.

Операція пошуку може мати декілька реалізацій: пошук перебором, бінарний пошук [3] та інші. Для пошуку перебором можлива кількість повторень може складати від 1 до n, де n – кількість сторінок у довіднику, у середньому $n/2$. Бінарний пошук полягає у розділенні області пошуку навпіл.

Та частина, що містить потрібні дані знову ділиться навпіл. У середньому кількість повторень буде складати $\log_2 n$.

Порівняємо операції пошуку: пошук перебором при $n = 10$ матиме у середньому 5 повторень, бінарний – 3...4, при $n = 128$, показники алгоритмів дорівнюватимуть 64 та 6, відповідно. При зростанні n, різниця стає більш вираженою, але реалізація другого варіанту складніша, тому для вибору потрібна коректна оцінка обох операцій.

Сенс застосування показників обчислювальної складності є якщо:

– передбачається обробка значного об'єму даних;

– необхідно задовольнити підвищені показники до часу обробки;

– відсутнє розпаралелювання обчислень програмними та апаратними засобами.

Методи та підходи до оцінки операцій обробки структур даних

Як було показано раніше кожною операцією обробки структур даних є процедура, функція або частка програми, розроблена за певним алгоритмом. Тому ключову роль у ефективності структур даних відіграють часові характеристики алгоритмів. Існує декілька підходів до визначення часових показників [4 – 11]. Їх основою є:

– безпосередній аналіз алгоритму;

– імовірнісний підрахунок операцій;

– асимптотична обчислювальна складність;

– оцінка амортизаційної вартості.

Кожен з наведених підходів має свої особливості та власну специфіку використання. Основна перевага зазначених підходів: можливість отримання оцінки ще до виконання програми.

3. Основна частина

Безпосередній аналіз алгоритму

Застосування підходу базується на досвіді оцінки типових алгоритмів. Наприклад, група операторів знаходиться у тілі циклу, виконання якого не переривається. На основі досвіду можна припустити, що складність такого циклу дорівнює n, де n – кількість виконань операторів тіла циклу. Якщо тіло циклу – цикл, кількість виконань якого m, складність становитиме – $n * m$ (або n^2 у випадку, коли $n = m$).

Розглянемо, як приклад, пошук мінімального елементу матриці:

```
for(i = 0; i < n; i++)
{
for(j = 0; j < n; j++)
{
if(matrix[i][j] < min)
{
min = matrix[i][j];
}
}
}
```

Згідно з умовою, оператори тіла зовнішнього циклу виконаються n разів. Внутрішній цикл також виконається n разів для кожного виконання зовнішнього циклу. Відповідно, перевірка умови, яка належить внутрішньому циклу, виконається n^2 разів. Недоліком підходу є низька придатність для великих алгоритмів, та алгоритмів, виконання яких може змінюватись в залежності від комбінації вхідних даних.

Імовірнісний підрахунок операцій

У поданому розділі розглянуто метод запропонований Д.Кнутом [4, 5]. Обчислювальна складність у методі підраховується за допомогою кількості операцій, вважаючи, що час їх виконання однаковий.

Припустимо, що для деякої структури даних реалізовано m операцій. Тоді ймовірність використання будь-якої з операцій дорівнює:

$$P_j = \frac{k_j}{\sum_{i=1}^{m} k_i}, \qquad (1)$$

де P_j – ймовірність виконання j -ої операції обробки даних, k_i – кількість виконань і -ої операції обробки даних. У (1) повинні бути враховані усі реалізовані операції обробки структури даних.

На наступному кроці, за допомогою отриманих значень P_j, визначається обчислювальна складність операцій за комбінаторно-імовірнісним методом:

$$S = \sum_{j=1}^{m} P_j \cdot k_j, \qquad (2)$$

де S – обчислювальна складність операції обробки структури даних, P_j – ймовірність виконання j -ої операції, k_j – розрахована кількість виконань операторів у j -ій операції обробки структури даних. Для кожної структури даних оцінка S розраховується тричі: для найкращого, середнього та найгіршого випадків виконання відповідно.

Правильність розрахунку перевіряється за умовою $\sum P_j = 1$.

Виконані дослідження дають змогу обрати кращу структуру за критерієм обчислювальної складності.

Якщо потрібно врахувати час виконання операцій з урахуванням часу виконання операцій (2) перетворюється на

$$S = \sum_{j=1}^{m} P_j \cdot k_j \cdot t_j , \qquad (3)$$

де t_j – час виконання j-ої операції.

Якщо не враховувати розпаралелювання на конвеєрі процесора та інші методи прискорення обчислень, можна встановити час виконання операцій за латентністю команд процесору.

Наприклад, у табл. 1 наведений асемблерний код, отриманий з вікна відлагодження у середовищі програмування.

Таблиця 1

Підрахунок кількості операцій з урахуванням латентності команд процесору

Оператори мови С++	Адреса у ОП	Команди процесору	Латентність
int* Search List (int data)	003914E0	push ebp	1.5
	003914E1	mov ebp,esp	0.5
	003914E3	sub esp,0D8h	1
	003914E9	push ebx	1.5
	003914EA	push esi	1.5
	003914EB	push edi	1.5
	003914EC	lea edi,[ebp-0D8h]	3
	003914F2	mov ecx,36h	0.5
	003914F7	mov eax,0CCCCCCCCh	0.5
	003914FC	rep stos dword ptr es:[edi]	15+36
	003914FE	mov eax,dword ptr [first	0.5
	00391503	(3980DCh)]	0.5
	00391506	mov dword ptr [tmp],eax	0.5
	0039150D	mov dword ptr [res],0	0.5
	00391511	cmp dword ptr [tmp],0	2
	00391513	je SearchList+58h (391538h)	0.5
	00391516	mov eax,dword ptr [tmp]	0.5
	00391518	mov ecx,dword ptr [eax]	0.5
	0039151B	cmp ecx,dword ptr [data]	2
	0039151D	jg SearchList+58h (391538h)	0.5
	00391520	mov eax,dword ptr [tmp]	0.5
	00391522	mov ecx,dword ptr [eax]	0.5
	00391525	cmp ecx,dword ptr [data]	2
	00391527	jne SearchList+4Dh	0.5
	0039152A	(39152Dh)	0.5
	0039152D	mov eax,dword ptr [tmp]	0.5
	00391530	mov dword ptr [res],eax	0.5
	00391533	mov eax,dword ptr [tmp]	0.5
	00391536	mov ecx,dword ptr [eax+4]	2
	00391538	mov dword ptr [tmp],ecx	0.5
	0039153B	jmp SearchList+2Dh	1.5
	0039153C	(39150Dh)	1.5
	0039153D	mov eax,dword ptr [res]	1.5
	0039153E	pop edi	0.5
	00391540	pop esi	1.5
	00391541	pop ebx	8
		mov esp,ebp	
		pop ebp	
		ret	
Разом			93

4. Апробація результатів дослідження

Порядок проектування структур даних

Розглянемо використання *імовірнісного підрахунку операцій*, для проектування структур даних ефективних за часовими характеристиками. За цим підходом слід виконати наступну послідовність дій:

– проектування альтернативних структур даних;

– проектування структур даних на рівні представлення (реалізація операцій – над обраними структурами);

– аналіз операцій над елементами структур (визначення найкращого, середнього та найгіршого випадків обробки даних);

– розрахунок обчислювальної складності операцій обробки структур даних.

Далі більш детально розглянемо кожен з кроків.

Проектування альтернативних структур даних

Проектування даних виконується у наступній послідовності: на абстрактному, логічному, представлення та фізичному рівнях.

На абстрактному рівні визначаються призначення даних та операції над ними. Окрім цього можуть бути введені певні обмеження, наприклад, максимальна кількість, діапазон значень даних, тощо. Зв'язки між даними на абстрактному рівні не визначаються.

Приклад. Структура даних – таблиці ідентифікаторів у лексичному аналізаторі трансляторів. Призначення – зберігання інформації про ідентифікатори, які використовуються у програмі, що транслюється. Операції – додавання ідентифікатору, його адреси, типу; пошук ідентифікатору; визначення адреси, типу.

На основі абстрактного виконується логічне проектування. На цьому рівні здійснюється вибір серед примітивних або конструювання на їх основі більш складних структур даних. Для подальшої демонстрації оберемо у якості прикладу дві структури: упорядкований однозв'язний динамічний список та упорядкований статичний масив, а також операції над ними – додавання та пошук елементу.

Проектування структур даних на рівні представлення

Проектування структур даних на рівні представлення полягає у визначенні самих структур даних та їх операцій засобами мови програмування.

Приклад реалізації операцій над елементами упорядкованого однозв'язного динамічного списку на мові С++:

```
int *first = NULL; //вказівник на перший елемент списку
//реалізація операції додавання нового елементу
void InsertList(int n)
{
int *node; //вказівник для нового елементу
node = (int*)malloc(sizeof(int*)); //виділяється пам'ять під елемент:
//половина – під безпосереднє зберігання даних
//половина – під зберігання адреси наступного елементу
//заноситься значення до нового елементу
node[0] = n; //у перші 4 байта – нові дані
node[1] = NULL; //у інші 4 – NULL-адресу
//список порожній або дані нового елементу менші ніж дані першого? if(!first || (first[0] > node[0]))
{
```

```
node[1] = (int)first; //вставка нового елементу у початок
first = node;
}
Else //список не порожній
{
int *tmp = first; //тимчасовий вказівник

//пошук місця для вставки нового елементу
while(tmp[1] && ((int*)tmp[1])[0] <= node[0])
{
tmp = (int*)tmp[1];
}

node[1] = tmp[1]; //вставка нового елементу перед знайденим
tmp[1] = (int)node;

}
}

//реалізація операції пошуку по даним
int* SearchList(int data)
{
int *tmp = first; //тимчасовий вказівник для обходу
int *res = NULL; //вказівник на результат пошуку
//пошук серед елементів списку
while(tmp[1] && tmp[0] <= data)
{
if(tmp[0] == data) //дані співпали?
res = tmp; //запам'ятовується результат
tmp = (int*)tmp[1]; //наступний елемент
}
return res; //повернення результату пошуку
}
```

Приклад реалізації операцій над елементами упорядкованого статичного масиву на мові C++:

```
const int full_size = 1000; //максимальна кількість елементів масиву
int current_size; //поточна кількість елементів
int array[full_size]; //статичний масив
//реалізація операції додавання нового елементу
void InsertArray(int data)
{
if(current_size + 1 <= full_size) //масив повний?
//(поточна кількість дорівнює максимальній?)
{
int i, temp; //тимчасові змінні
i = current_size; //лічильник для обходу масиву
//пошук позиції та звільнення місця для вставки
while(i > 0 && array[i - 1] > data)
{
array[i] = array[i - 1]; //зсув елементів
i--;
}
array[i] = data; //вставка нових даних
current_size++; //збільшення поточної кількості елементів
}
}
//операція пошуку
int SearchArray(int data)
{
int res = -1; //результат пошуку
int i = 0; //лічильник
//пошук серед елементів масиву
while(i < current_size && array[i] <= data)
{
if(array[i] == data) //дані співпали?
res = i; //запам'ятовується позиція знайденого елементу
i++;
}
return res; //повернення результату пошуку
}
```

Аналіз операцій над елементами структур даних

Аналіз операцій полягає у визначенні найкращого, середнього та найгіршого випадків обробки даних у розроблених структурах.

Результат аналізу операції над структурами, що реалізовані у прикладі наведені нижче.

Для **операції додавання**:

– *найкращий випадок* – додавання вузла у початок списку, що передбачає виконання лише першого умовного оператору; для масиву – додавання елементу у кінець, для якого не потрібне виконання циклічного зсуву елементів;

– *середній випадок* – додавання вузла у середину списку, для якого потрібно виконати обхід половини вузлів та операцію вставки; для масиву – додавання елементу у середину, яке потребує виконання зсуву половини елементів;

– *найгірший випадок* – додавання у кінець списку, для якого необхідно виконати обхід усіх вузлів; для масиву – додавання елементу у початок, яке потребує виконання зсуву усіх елементів.

Для **операції пошуку** у списку та масиві:

– *найкращий випадок* – пошук неіснуючого елементу, меншого за початковий, при якому не потрібно виконувати цикл по перегляду вмісту структури даних;

– *середній випадок* – пошук існуючого елементу, що знаходиться у середині, при якому потрібно виконати перегляд половини вмісту структури даних;

– *найгірший випадок* – пошук неіснуючого елементу, більшого за кінцевий, при якому необхідно здійснити перегляд усього вмісту структури даних.

Підрахунок кількості виконаних операторів

Будемо вважати, що на поточний момент у структурах даних міститься n елементів.

Здійснимо підрахунок кількості виконаних операторів при обробці лінійного списку. Розглянемо приклад, використавши оператор циклу:

```
while(tmp[1] && ((int*)tmp[1])[0] <= node[0])
{
tmp = (int*)tmp[1];
}
```

У прикладі:

– у найкращому випадку цикл не виконується, тому як він знаходиться у альтернативній гілці оператору розгалуження, що має істинну умову;

– середній випадок має $n/2$ виконань операторів тіла циклу, тому як це відповідає середині списку та $n/2+1$ виконання перевірки умови, відповідно до правил роботи циклів з передумовою;

– найгірший випадок має n виконань перевірки умови, тому що на кожному кроці перевіряється наявність наступного вузла та $n-1$ виконань операторів

тіла циклу (на останньому кроці наступний вузол відсутній).

Для упорядкованого однозв'язного динамічного списку результат оцінювання операцій наведено у табл. 2.

Таблиця 2

Підрахунок кількості виконаних операторів при обробці лінійного списку

Операції та оператори	Кількість виконаних операторів		
	Найкращий випадок	Середній випадок	Найгірший випадок
Додавання нового елементу			
int *node;	1	1	1
node = (int*)malloc(sizeof(int*));	1	1	1
node[0] = n;	1	1	1
node[1] = NULL;	1	1	1
if(!first ‖ (first[0] > node[0]))	1	1	1
{			
node[1] = (int)first;	1	0	0
first = node;	1	0	0
}			
else	0	1	1
{			
int *tmp = first;	0	1	1
while(tmp[1] && ((int*)tmp[1])[0] <= node[0])	0	n/2 + 1	n
{	0	n/2	n - 1
tmp = (int*)tmp[1];			
}	0	1	1
node[1] = tmp[1];	0	1	1
tmp[1] = (int)node;			
}			
Разом	7	10 + n	8 + 2n
Пошук елементу			
int *tmp = first;	1	1	1
int *res = NULL;	1	1	1
while(tmp[1] && tmp[0] <= data)	1	n/2 + 1	n
{			
if(tmp[0] == data)	0	n/2	n - 1
res = tmp;	0	1	0
tmp = (int*)tmp[1];	0	n/2	n
}			
return res;	1	1	1
Разом	4	5 + 3n/2	2 + 3n

Виконаємо підрахунок кількості операторів при обробці статичного масиву. Розглянемо підрахунок на прикладі оператору циклу:

```
while(i > 0 && array[i - 1] > data)
{
array[i] = array[i - 1];
i--;
}
```

Для наведеного оператору:

– у найкращому випадку перевірка умови виконується один раз, для виявлення того, що зсув елементів не потрібен; оператори тіла циклу не виконуються;

– у середньому випадку оператори тіла циклу мають $n/2$ виконань, тому як це відповідає середині масиву; умова перевіряється $n/2+1$ разів, відповідно до правил роботи циклів з передумовою;

– у найгіршому випадку оператори тіла циклу мають n виконань, тому як кожен елемент має бути зсунутий: умова перевіряється $n+1$ разів, відповідно до правил роботи циклів з передумовою.

Для статичного масиву результат оцінювання операцій наведено у табл. 3.

Таблиця 3

Підрахунок кількості виконаних операторів при обробці статичного масиву

Операції та оператори	Кількість виконаних операторів		
	Найкращий випадок	Середній випадок	Найгірший випадок
Додавання нового елементу			
if(current_size + 1 <= full_size)	1	1	1
{			
int i, temp;	1	1	1
i = current_size;	1	1	1
while(i > 0 && array[i - 1] > data)	1	n/2 + 1	n + 1
{			
array[i] = array[i - 1];	0	n/2	n
i--;	0	n/2	n
}			
array[i] = data;	1	1	1
current_size++;	1	1	1
}			
Разом	6	6 + 3n/2	6 + 3n
Пошук елементу			
int res = -1;	1	1	1
int i = 0;	1	1	1
while(i < current_size && array[i] <= data)	1	n/2 + 1	n + 1
{	0	n/2	n
if(array[i] == data)	0	1	0
res = i;	0	n/2	n
i++;			
}	1	1	1
return res;			
Разом	4	5 + 3n/2	4 + 3n

Кращий випадок для обох операцій – це константне значення, яке не залежить від кількості елементів у структурах. При середньому та гіршому випадку отримані показники є функціями від n .

Розрахунок обчислювальної складності операцій обробки даних

Визначимо обчислювальну складність операцій для розроблених структур за комбінаторно-імовірнісним методом.

Для прикладу розглянемо задачу формування та використання таблиці ідентифікаторів при розробці деякого транслятору. Під час обробки таких таблиць використовують лише операції додавання та пошуку (в інших структурах даних можуть бути задіяні операції видалення, сортування та інші). Кількість операторів k_j з (1) може бути визначена статистичним методом, за допомогою підрахунку ідентифікаторів та відповідних операцій у зразках текстів програм.

Використаємо зразок програми на мові Pascal, наведений у табл. 4.

Під час обробки наведеного тексту лексичним аналізатором операція додавання у таблицю ідентифікаторі буде виконана 17 разів, а операція пошуку – 42.

Тоді розрахуємо ймовірність використання операцій згідно з (1):

$$P_+ = \frac{17}{17+42} = 0.3 \text{ та } P_? = \frac{42}{17+42} = 0.7,$$

де P_+ – ймовірність виконання операції додавання,
$P_?$ – ймовірність виконання операції пошуку.

Таблиця 4

Зразок програми для аналізу та підрахунку кількості використань операторів обробки структур даних

Текст програми	Кількість використань	
	Оператор додавання	Оператор пошуку
procedure initRound(round:integer);		
var		
i,j: byte;	2	
isChosen: boolean;	1	
a,b,c,d:word;	4	
selectedPin: integer;	1	
attempt, round, wins, field: integer;	4	
winner: boolean;	1	
pins:array[0..maxAttempts-1,0..3] of integer;	1	
answer: array[0..maxAttempts-1,0..3] of integer;	1	
passwordPin: array[0..3] of integer;	1	
passwordHelp: array[0..1,0..3] of boolean;	1	
begin		
randomize;		
for i := 0 to maxAttempts-1 do begin		2
for j:=0 to 3 do begin		1
pins[i][j] := -1;		3
answer[i][j] := -1;		3
end;		
end;		
for i := 0 to 3 do begin		1
isChosen := true;		1
while (isChosen) do begin		1
isChosen := false;		1
gettime(a,b,c,d);		4
passwordPin[i] := (random(a*60+d)) mod 6;		4
for j:=0 to i do		2
if (i<>j) and (passwordPin[j] =		
=passwordPin[i])		6
then		
isChosen := true;		1
end;		
end;		
for i := 0 to 3 do begin		1
pins[attempt][i] := -1;		3
end;		
nrButtonActive := false;		1
acceptButtonActive := false;		1
roundEnd := false;		1
round := round;		2
attempt := 0;		1
field := -1;		1
selectedPin := -1;		1
end;		
Разом	17	42

Правильність розрахунку перевіримо за умовою $P_+ + P_? = 1$.

Після операцій обробки кількість елементів у структурах (n) буде дорівнювати дев'яти. Використаємо отримані значення згідно з (2) та визначимо обчислювальну складність операцій зі списком:

$$\hat{S} = 0,3 \cdot 7 + 0,7 \cdot 4 = 2,1 + 2,8 = 4,9;$$

$$\bar{S} = 0,3 \cdot (10 + n) + 0,7 \cdot (5 + 3n/2) = 0,3 \cdot 19 + 0,7 \cdot 13,5 = 15,15;$$

$$\check{S} = 0,3 \cdot (8 + 2n) + 0,7 \cdot (2 + 3n) = 0,3 \cdot 26 + 0,7 \cdot 29 = 28,1.$$

Знову використаємо (2) та визначимо обчислювальну складність операцій з масивом:

$$\hat{S} = 0,3 \cdot 6 + 0,7 \cdot 4 = 1,8 + 2,8 = 4,6;$$

$$\bar{S} = 0,3 \cdot (6 + 3n / 2) + 0,7 \cdot (5 + 3n/2) = 0,3 \cdot 19,5 + 0,7 \cdot 18,5 = 18,8;$$

$$\check{S} = 0,3 \cdot (6 + 3n) + 0,7 \cdot (4 + 3n) = 0,3 \cdot 33 + 0,7 \cdot 31 = 31,6.$$

Підсумкові результати розрахунку представимо у вигляді табл. 5.

Таблиця 5

Результати розрахунку обчислювальної складності

Структура даних	Обчислювальна складність		
	Найкращий випадок	Середній випадок	Найгірший випадок
Упорядкований однозв'язний список	4,9	15,15	28,1
Упорядкований статичний масив	4,6	18,8	31,6

Виконані дослідження дають змогу обрати кращу структуру за критерієм обчислювальної складності. У даному випадку кращою структурою для представлення таблиці ідентифікаторів є упорядкований однозв'язний список.

5. Висновки

Добре відомий метод визначення обчислювальної складності алгоритмів Д. Кнута [4, 5] дозволяє опосередковано, за кількістю обчислень оцінювати часові характеристики алгоритмів. Але часові характеристики програм у значній мірі залежать і від того, наскільки вдало обрані або сконструйовані структури даних, які застосовані у програмі.

Запропонований у даній роботі метод визначення обчислювальної складності структур даних спирається на метод Д. Кнута.

Обчислювальна складність структур даних визначається як обчислювальна складність сукупності алгоритмів (операцій) обробки даних. Операціями обробки даних вважаються операції доступу до позиції та до даних, та виключаються операції перетворення даних.

За розробленим методом обґрунтування вибору структур даних спирається на проведені дослідження, аналіз результатів базується на об'єктивній оцінці виконаної роботи для виявлення переваг та недоліків кожної зі структур даних. Оскільки недоліки присутні у будь-якій обробці структур даних, здатність їх виявляти та передбачати є шляхом усунення проблем з ефективністю у тому числі й по показникам обчислювальної складності.

Однак слід пам'ятати, що показники мають лише відносну оцінку, яку слід інтерпретувати в залежності від умов, у яких буде використовуватись структура даних. До таких умов відносяться необхідність вибору або конструювання ефективних структур даних, що може бути пов'язаним з обробкою великих обсягів даних, багатократним їх використанням. Важливо враховувати, що імовірнісні методи оцінки як алгоритмів, так і структур даних достовірні лише в умовах відсутності розпаралелювання обчислень програмними та апаратними засобами.

У наведеному прикладі показано, що інтуїтивне уявлення, що динамічні структури даних «швидші» за статичні частково підтверджується тим, що для обраної мови програмування, транслятору та порядку застосування структур даних (за кількістю різних операцій обробки) статична структура за показником обчислювальної складності виявилась гіршою у середньому випадку на 17,5 %.

Але в кращому випадку ефективніше на 7 % саме статичний масив.

Слід зауважити, що інші умови застосування структур даних вимагають окремого дослідження, і результати цих досліджень заздалегідь непередбачувальні. Це ще раз підтверджує практичну значимість запропонованого методу для визначення об'єктивних оцінок.

Література

1. Вирт, Н. Алгоритмы и структуры данных [Текст] / Н. Вирт – М.: ДМК, 2010. – 274 с.
2. Шинкаренко, В. И. Экспериментальные исследования алгоритмов в программно-аппаратных средах [Текст] / В. И. Шинкаренко – Д.: Изд-во Днепропетр. нац. ун-та железнодор. трансп. им. акад. В. Лазаряна, 2009. – 279 с.
3. Макконнелл, Дж. Анализ алгоритмов. Вводный курс [Текст] / Дж. Макконнелл. – М.: Техносфера, 2002. – 304 с.
4. Кнут, Д. Искусство программирования, том 1. Основные алгоритмы [Текст] / Д. Кнут. – [3-е изд.]. – М.: Издательский дом "Вильямс", 2000. – 720 с.
5. Кнут, Д. Искусство программирования, том 3. Сортировка и поиск [Текст] / Д. Кнут. – [3-е изд.]. – М.: Издательский дом "Вильямс", 2000. – 832 с.
6. Кормен, Т. Алгоритмы: построение и анализ [Текст] / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: МЦНМО, 2001. – 960 с.
7. Ахо, А. Построение и анализ вычислительных алгоритмов [Текст] / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Мир, 1979. – 536 с.
8. Ахо, А. В. Структуры данных и алгоритмы [Текст] / А. В. Ахо, Дж. Хопкрофт, Дж. Д. Ульман. – М.: Изд. дом «Вильямс», 2001. – 384 с.
9. Грин, Д. Математические методы анализа алгоритмов [Текст] / Д. Грин Д. Кнут. – М.: Мир, 1987. – 120 с.
10. Гудман, С. Введение в разработку и анализ алгоритмов [Текст] / С. Гудман, С. Хидетниеми. – М.: Мир, 1981. – 366 с.
11. Гудрич, М. Т. Структуры данных и алгоритмы в Java [Текст] / М. Т. Гудрич, Р. Тамассия. – Мн.: Новое знание, 2003. – 671 с.