АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник

ЗМІСТ

ВКАЗІВНИК ТЕРМІНІВ

Наведемо деякі поняття термінів, які будуть використовуватися при викладенні матеріалів посібника, при необхідності більш поширені поняття цих термінів і їх приклади використання будуть запропоновані у подальшому викладенні.

Автомат – формальна конструктивна структура, яка визначається входом, виходом, множиною станів і функціями переходів та виходів.

Алгоритм – на інтуїтивному рівні, послідовність дій, за якою досягається певна мета і який має дві сторони: представлення і реалізацію. Загального точного визначення алгоритму не існує. Часткове коректне визначення алгоритму можна дати для певного предмету і механізму, який реалізує цей алгоритм. Наприклад, алгоритм Т'юрінга, визначений на натуральній множині обчислених функцій і, який реалізується на механізмі Т'юрінга.

Атрибут – в перекладі з латинської /присукупленний/, істотний признак, властивість будь чого, невід'ємна приналежність предмету тощо. Атрибут завжди знаходиться в деякому відношенні до імені елемента x предметної області або до множини імен X, тобто елемент з іменем $x \in X$ та атрибутом $a \in A$, можна записати як відношення $\varphi(x, a)$, $\varphi_1(X, a)$, або такax, aX.

Атрибутика – сукупність атрибутів.

Властивість (у програмуванні) – засіб доступу до внутрішнього стану об'єкта, імітуючого змінну деякого типу. Властивість (в побуті, математиці і логіці) — атрибут предмету (об'єкта). Властивість можна розглядати як форму предмета саму по собі, притому, що він може мати і інші властивості.

Граф – сукупність елементів і пар елементів, в якій елементи звуть вершинами і пари вершин – ребрами. Якщо пара вершин упорядкована, тоді ребро зветься дугою.

Дані – інформація спеціальним чином представлена технічними пристроями, наприклад, ЕОМ або людиною.

Дерево – граф з однією спільною вершиною (корінь дерева), який немає контурів.

Елемент – неподільний об'єкт – число, буква тощо, однорідні елементи мають однакову природу (тип), так, сукупність букв і цифр – не однорідна.

Інформація – відображення реального світу, яке має властивості: здобування, передавання, збереження (інформації). Носієм інформації є повідомлення, яке формально складається з символів, позначок та іншого.

Категорія [грец.] – найбільш суттєве поняття предмету, явища та іншого.

Кількість інформації – вимір інформації, який здійснюється в бітових одиницях або ін.

Клас – як математична категорія – сукупність предметів (категорій) виділених за деяким признаком за звичай перелічених. Частковий випадок – множина, сімейство.

Ключ – деяка сутність за якою ведеться пошук у структурі даних

Конструктивна структура – формальна породжуюча структура призначена для конструювання об’єктів предметних областей.

Контур графу – послідовність зв’язаних ребрами вершин таких, що початкова вершина і заключна співпадають.

Критерій [грец.] – признак, на основі якого проводиться оцінка, визначення або класифікація чого-небудь.

Масив – індексно упорядкована структура даних.

Метод – множина алгоритмів призначених для вирішення певної задачі.

Методи проходження дерев – сукупність алгоритмів пошуку даних на деревах, які передбачають правила проходження дерев і відвідування їх вершин.

Методи сортування – алгоритми призначені для упорядкування за відповідним показником даних. Методи сортування використовують різноманітні алгоритми пошуку, за якими їх класифікують і алгоритми обміну даних.

Модель – система, яка адекватно відтворює призначення об’єктів предметної області.

Множини – сукупність різних однорідних і вільних (як завгодно розташованих у сукупності) елементів.

Мультомножина – множина, в якій окремі елементи можуть повторюватися.

Нечітка величина – величина, значення якої визначаються множиною даних.

Об’єкт – предмет пізнання і розвитку, предметами пізнання є сутності їх властивості, признаки тощо. У програмуванні — деяка сутність у віртуальному просторі, котра має певні стани і поведінку, які задаються значеннями властивостей і операцій над ними. Як правило, при розгляді об’єктів вважається, що об’єкти належать одному або декільком класам, котрі визначають поведінку об’єкту.

Оцінка алгоритму – здійснюється за різними показниками: його часу реалізації, об’ємом представлення, логічною складністю представлення, його кількістю дій та інше.

Піраміда даних – структура даних, у якій числові дані розташовані у вершинах дерева так, що значення даних попереднього рівня більші (менші) за значення наступного рівня. При цьому, якщо виконується умова «більше», тоді піраміда зветься максимальною і відповідно мінімальною у протилежному випадку.

Послідовність – множина з упорядкованими за яким-то признаком елементами.

Предметна область – область визначення предмета (область визначення алгоритму, функції тощо).

Признак – категорія, призначена для виділення чого-небудь із набору сутностей. Признак – одна із сторін об’єкту (якість, властивість) або багато

його сторін, сам об’єкт, його категорія та інше. Конструктивно признак можна задавати відношеннями на сукупностях властивостей об’єкту або об’єктів.

Рівень дерева – сукупність його вершин, які знаходяться на однаковій відстані від коріння. Номер рівня дерева визначається відстанню від коріння до вершин рівня. Так перший рівень дерева містить вершини розташовані на відстані один від його корню.

Система – множина зв’язаних яким-то чином елементів.

Системний підхід до предметної області – погляд на предметну область як на систему.

Сімейство – систематизована категорія будь-чого. Систематизація може досягатися завдяки критеріям, властивостям, параметрам тощо.

Список – структура зв’язаних полем показника даних, існують одно зв’язані і багато зв’язані списки.

Стек – автомат пам’яті, у якому вхід і вихід один і той же.

Структура – будова дечого.

Структури даних – будова, яка визначається за представленням даних, представленням у пам’яті ЕОМ, типом даних тощо.

Структуризація предметної області – визначення складових предметної області, зв’язків між ними, формалізація і інше необхідне для побудови моделі предметної області.

Тип [грец.] – зразок як модель дечого, форма як вид дечого.

Формула – точне визначення деякого правила (операції, відношення, відображення та інш.), коротко представлене.

Черга – автомат пам’яті, у якому вхід і вихід окремо різні.

Штучна граматика – формальна система визначена послідовністю: термінального і не термінального алфавітів, початкового не термінального символа та множиною правил виводу. Наприклад, множина правил $P = \{p_1 : \sigma \to a\sigma b, p_2 : \sigma \to ab\}$

граматики виводе ланцюжок a^3b^3

$$C(a^3b^3) = \sigma \xrightarrow{p_1} a\sigma b \xrightarrow{p_1} a^2\sigma b^2 \xrightarrow{p_2} a^3b^3.$$

Штучна мова – утворена множиною виведених ланцюжків у деякій формальній граматиці.

ВСТУП

Сучасний рівень програмування передбачає застосування структур даних (СД), як необхідних атрибутів програмних конструкцій. Отже програмування передбачає формулу представлення: *структуризація предмета програмування + структури даних + алгоритми + програма + реалізація алгоритму.*

Так як дані можуть бути різними (числа, символи, строки тощо), тому вони мають різну структуру представлення, що необхідно враховувати в програмуванні. Історично можна виділити три етапи розвитку структуризації (організації структур даних) в мовах програмування.

- *Перша спроба організації структур* – через адресність (пам'яті) машин.
- *Друга спроба* – через типи даних.
- *Третя спроба* – повна типізація на всіх рівнях обробки програм і конструювання типовості структур даних.

Існує понад 5 тис. середовищ програмування (інженерні, економічні, інтелектуальні, тощо) зі своїми технологіями організації структур даних.

Сучасне системне програмування застосовує технологію третього етапу організації структур даних.

Виходячи з цього, знання теоретичних питань СД, практичних навичок їх представлення та обробки є важливим елементом професійних знань для фахівців з програмної інженерії.

Вирішення задач предметних областей потребують її структуризації, розробки алгоритмів, представленням яких є алгоритмічна програма (програма) написана на тій чи іншій мові. Реалізація алгоритмів за їх представленням проводиться людиною, штучним виконавцем тощо. Ефективність представлення і виконання алгоритму рішення задачі залежить від вибраних структур даних.

Алгоритми структур даних – прикладний розділ науки «Теорія програмування», в якому вивчаються основи структур даних та алгоритми їх обробки. З даної дисципліни існує досить поширена література, частина з якої [1 – 4] застосовується в матеріалах розділів посібника. Фундаментальним джерелом з предмету викладання навчального посібника є робота [1], котрій надана перевага у виборі програм та ілюстраційних матеріалів.

В матеріалах посібника висвітлюються як базові питання з понять інформації та даних, структур та даних, алгоритмів та їх ефективності, так і спеціальні питання структур даних масивів, списків, дерев, графів та алгоритмів і їх обробки. Матеріали представлені у шести розділах та трьох додаткових розділах, у першому з яких наводяться питання з закріплення і контролю набутих знань, і у другому – пропонується практичні завдання для набуття навичок застосування отриманих читачем знань у розробці елементів програмного забезпечення. У останньому дев'ятому розділі розглянуто приклади створення програм для складних гібридних структур даних.

РОЗЛІЛ 1. БАЗОВІ ПОНЯТТЯ ПРЕДМЕТУ

У розділі наведені базові відомості необхідні здобувачеві знань для засвоєння предмету «Алгоритми, дані і структури». Дано попереднє знайомство з інформацією та даними, надаються відомості про структури і дані та визначається поняття алгоритму, його роль у обробці даних тощо. Матеріали розділу наведені у трьох параграфах: *інформація і данні, структури і данні* та *алгоритми їх представлення, реалізація і аналіз*

1.1. Інформація і данні

Будь який предмет природи або штучний об'єкт «породжують» інформацію. Людина сприймає інформацію з інформаційного простору через свої сенсори зір, слух та інше. Останні наукові досягнення свідчать про те, що людина має 11 сенсорів для сприйняття інформації і людина може існувати тільки в інформаційному просторі. Тому інформація може бути визначена як відображення реального світу, яке має властивості: збереження, обробки, передачі. Інформація має носій, на який вона накладається. Носієм інформації є текстове, звукове, відео або інше повідомлення.

Програма, як об'єкт ЕОМ є не тільки послідовність строк, операторів деякої штучної мови програмування, але і набір інформаційних об'єктів імен-даних над якими виконуються ті чи інші дії операторів програми. Отже програма є носієм інформації. Програмний носій як правило представляється у вигляді текстів повідомлення, утворених на алфавітах природних або штучних мов, або конструктивних об'єктів, за допомогою яких будуються фрагменти мов. Прикладом конструктивних об'єктів є конструкції на вертикальних рисках (|). За допомогою риски можливо побудувати множину натуральних чисел (числовий ряд): {1 – |, 2 – ||, 3 – |||, ...} або за набором з вертикальної (|), горизонтальної (−) та нахиленої (/) рисок є можливість побудувати цілі та раціональні числа [5], шифри поштових індексів також утворюються за набором {|, −,/}.

Перейдемо до детального розгляду базових понять: *алфавіт*, *інформація*, *данні*.

Так як природа об'єкту пізнання різна (буква, знак, цифра, риска і інше), тому будемо використовувати уніфіковану назву імені об'єкту *символ.* Таким чином, символ – *базовий об'єкт* для побудови конструктивних об'єктів текстів повідомлення [5].

Визначення 1.1. Множину символів (елементів) будемо називати *алфавітом.*

Наприклад, $A = \{а, б, в, \dots, я\}$- алфавіт української мови, $B = \{|\}$- алфавіт натурального ряду. У мовах програмування алфавіт утворюється символами:

$a, b, c, \ldots >, <, \ldots 1,2, \ldots$. В подальшому будуть розглядатися алфавіти, які задовольняють наступним умовам:

- алфавіт складається з скінченої кількості символів, тобто він скінчений;
- порядок символів в множині не суттєвий;
- порожній символ ε завжди належить алфавітові, хоча не завжди виділяється в переліку його символів;
- символи алфавіту різні, тобто однакові символи в алфавіті не допускаються;
- елементами множини можуть бути різні комбінації символів цього ж алфавіту;
- алфавіт, котрий складається тільки з символу ε – порожній.

Згідно з умовами елементи алфавітів можуть бути *простими* і *сполученими*. Прості елементи неподільні, тобто не можуть бути сполучені з символів даного алфавіту. Сполучені ж елементи складаються з більше як одного символів алфавіту. Сполучення $\varepsilon a = a$ будемо відносити до простих елементів. Наприклад, алфавіт $A = \{\varepsilon, a, b, ab, aab\}$містить в собі три простих елементи (символи) ε, a, b і два сполучених елементи ab, aab. У програмуванні сполученні елементи алфавіту звуть *ключовими словами,* наприклад, if, for, і ін. Наведемо декілька важливих визначень.

Визначення 1.2. Розміром алфавіту A зветься кількість його простих елементів (символів) і позначається це так $dim\, A$.

Так для алфавіту $A = \{\varepsilon, a, b, ab, aab\}$ його $dim\, A = 2$, так як приймемо, що $dim\{\varepsilon\} = dim\, \emptyset = 0$.

Визначення 1.3. Алфавіт зветься *базисним або стандартним*, якщо він складається тільки з символів, тобто усі його елементи прості.

Наприклад, двійковий алфавіт $B = \{a, b\}$- стандартний.

Якщо $A \subseteq B$, тоді кажуть, що алфавіт A є під алфавітом алфавіту B і в разі $A \subset B$ кажуть, що A є власним *підалфавітом* алфавіту B.

Визначення 1.4. Алфавіти A і B звуть *рівними* $(A = B)$, якщо виконуються включення $A \subseteq B$ і $B \subseteq A$.

На заданому алфавіті можливо побудувати нові конструктивні об’єкти. Алфавітні конструктивні об’єкти є ідентифікаторами і елементами структур в програмуванні і тому приділимо їм відповідну увагу.

Нехай задано деякий алфавіт A.

Визначення 1.5. Словами (ланцюжками) над алфавітом A будемо називати конструктивні об’єкти побудовані за допомогою *індуктивного* процесу:

- слово $l = \varepsilon$ є порожнім словом;

- якщо конструктивний об’єкт l – слово над деяким алфавітом A, тоді конструктивний об’єкт $l\alpha,\ \alpha \in A$ є також словом побудованим над цим алфавітом A.

З визначення 1.5 маємо:

- що $l\varepsilon = \varepsilon l = l$, тобто порожнє слово слугує одиницею конструкції l;

- якщо l слово над алфавітом A, тоді за індуктивним процесом $lll \dots l = l^i$ також слово над алфавітом A;

- приймається, що $l^0 = \varepsilon$.

- над алфавітом $A = \{\varepsilon, a, b\}$ можливо отримати нескінченну сукупність слів $\{\varepsilon, a, b, aa, bb, ab, \dots\}$.

Множину слів побудованих над алфавітом A будемо позначати через $\mathbb{F}(A)$.

Нехай l слово над алфавітом A, тобто $l = a_1 a_2 \dots a_k \in \mathbb{F}(A)$ тоді

Визначення 1.6. Довжиною слова l над алфавітом A є кількість символів, з яких складається це слово і позначають її так $|l| = k$.

Приймемо довжину порожнього слова ε за $|\varepsilon| = 0$. Зрозуміло, що будь яке не порожнє слово має довжину $k \geq 1$. Якщо l_1 і l_2 слова множини $\mathbb{F}(A)$, тоді для довжини слів мають виконуватися властивості

$$|l_1 l_2| = |l_1| + |l_2| \text{ і } \left|l_1^i\right| = i|l_1|.$$

Визначення 1.7. Два слова $l = b_1 b_2 \dots b_m \in \mathbb{F}(B)$ і $q = a_1 a_2 \dots a_n \in \mathbb{F}(A)$ *однакові* $l = q$, якщо $|l| = |q|$ або $n = m$ і $a_i = b_i$ для всіх значень індексів $1 \leq i \leq n$.

Таким чином ми визначили формальну систему над алфавітом A, побудовану за допомогою індуктивного (*рекурсивного*) правила $l = l\alpha, \quad \alpha \in A$, на основі якого побудована вільна мова $A^* = \mathbb{F}(A)$.

Якщо слова $l, q \in \mathbb{F}(B)$, то може статися, що слово l є частиною слова q, тобто $q = xly$, де $x, y \in \mathbb{F}(B)$, у цьому випадку слово l звуть *підсловом* (*підланцюжком*) слова q.

Розглянемо деяку конструктивну множину W – це може бути алфавіт (конструктивний алфавіт) або будь яка множина слів.

Визначення 1.8. Упорядкована послідовність елементів (взагалі, різної природи) $w_i \in W, \quad i = 1,2, \dots, n$, тобто кожен елемент w_i займає i -те місце у цій послідовності, зветься *кортежем розміру* n і позначається як $(w_1, w_2, \dots, w_n)$.

Звертаємо увагу на те, що кортеж відрізняється від множини, в якій елементи не упорядковані. Якщо $n = 2$, тоді кортеж (w_1, w_2) звуть *упорядкованою парою.* Вважається, що два кортежі $(w_1, w_2, \dots, w_n)$, $(y_1, y_2, \dots, y_m)$ співпадають, якщо вони мають однаковий розмір $n = m$ і $w_i = y_i, \quad i = 1,2, \dots n$. Очевидно, що кортежі $(w_1, w_2, \dots, w_n)$ і $(w_2, w_1, \dots, w_n)$ різні, а пара $(w, \varepsilon) = (\varepsilon, w) = w \in$ *виродженою.* Кортеж порожній, якщо він утворений з порожніх елементів $(\varepsilon, \varepsilon, \dots, \varepsilon) = \varepsilon$. Таким чином кортежі як об'єкти можуть утворювати звичайні конструктивні послідовності. Частковим випадком кортежу є *масив, список*, у яких об'єкти мають однакову природу. Якщо об'єкти кортежу повторюються, тоді він зветься мультокортежом. Загалом, конструктивний об'єкт-програма представляється мультокортежом тому, що оператори в ній можуть декілька разів повторюватися.

У неоднорідному кортежі W неоднорідності його елементів w_i визначаються *атрибутом* α_i, як його невід'ємною частиною. Елемент w_i з атрибутом α_i позначимо так $\alpha_i w_i$ [5]. Наприклад, якщо елементи кортежу W слова, тоді довжина слова w_i є його атрибутом, тобто $|w_i|w_i$. Запис $|w|W$ означає, що кортеж W однорідний по довжинам його елементів.

Для зручності обробки об'єктів у програмуванні їх кодують. Кодування об'єктів l_i скінченної множини $\bar{A} \subset A^*$ завдається відображенням φ, за яким об'єкту l_i ставиться у відповідність об'єкт q_i сконструйований на тому ж чи іншому алфавітові, тобто $\varphi: l_i(A) \to q_i(B)$.

Розглянемо приклад кодування об'єктів восьмирічної системи числення *к*онструкціями створеними на двійковому алфавітові $\{0,1\}$.

Так як розглядається восьмирічна система числення, то кількість різних конструктивних об'єктів a_i таких, що $|a_i| = 1$, дорівнює восьми, тобто $(a_1, a_2, \dots, a_8)$. Виходячи з того, що $b_i = \varphi(a_i)$ повинні будуватися над двійковим алфавітом, тому довжина коду $|b_i| = 3$ $(2^3 = 8)$ для $i = \overline{1,8}$. Процес побудови кодових слів b_i відбувається за схемою:

- спочатку набір $(a_1, a_2, \dots, a_8)$ поділяється навпіл,
- першій групі набору приписується символ 0, а другій – 1, як зображено на рис. 1.1;
- потім знову кожна з груп поділяється навпіл і першій приписується – 0, другій – 1 і так далі поки в кожній групі не зостанеться по одному об'єкту;
- на при кінець необхідно записати слова читаючи символи 0 або 1 спочатку процесу ділення (див. рис. 1.1).

$$
\begin{array}{ll}
a_1 & 000 \\
a_2 & 001 \\
a_3 & 010 \\
a_4 & 011 \\
a_5 & 100 \\
a_6 & 101 \\
a_7 & 110 \\
a_8 & 111
\end{array}
$$

Рис. 1.1. Кодування восьмирічної системи числення

Отже восьмирічна система числення закодована рівномірними кодами довжини три. Аналогічно кодується рівномірними кодами на двійковому алфавіті шістнадцятирічна, тридцятидвохрічна та інші системи числення.

Після введення понять алфавіту, конструктивного об'єкту маємо можливість ввести поняття інформації.

Визначення 1.9. Під інформацією, яку несе на собі деяке повідомлення розуміється відображення, котре задовольняє умовам:

- воно не матеріальне;
- здобування, збереження, обробки, передачі.

Як було вказано, носієм інформації є текстове повідомлення. Щоб здобути інформацію з тексту повідомлення, написаного на відповідній мові, людині необхідно вміти читати, знати граматику цієї мови, мати досвід розуміння прочитаного тощо. Послідовність вмінь здобуття, збереження, обробки і передачі інформації утворює технологію опрацювання інформації. Одним із напрямків навчання людини є засвоєння технологічних навичок опрацювання інформації. В штучних системах у технології опрацювання інформації приймає участь транслятор або інший інтелектуальний пристрій.

Визначення 1.10. Інформаційне повідомлення спеціальним чином підготовлене для сприйняття технічними пристроями або людиною зветься *даними* (данні інформації).

Крім наведеного визначення поняття данні, існує ще декілька ідентичних визначень цього поняття. Наприклад, за міжнародним стандартом ISO *данні* – це представлення фактів, понять, інструкцій, ідей або будь якої інформації у формалізованому вигляді, сприятливому для обробки, інтерпретації, спілкування або передачі, як людиною, так і технічними засобами при здійснені відповідних послідовностей дій або процесів.

Здобуття інформації буває пов’язано з вимірювальними пристроями для визначення кількісних параметрів відповідних даних, кольору, звучання, геометричних розмірів тощо. Виходячи з того, що пристрої визначають виміри не точно і мислення людини також розмите, тому данні можуть представлятися нечітким чином. Нечіткі дані представляються, як правило, парою утвореною даним d і його характеристикою $\mu(x)$, значення якої належать проміжку $[0,1]$. При значені $\mu(x) = 1$ – дане d є чітким. Отже характеристика є атрибутом даного $\mu(x)d$. В свою чергу, данні є атрибутами будь якої частини програми або програми взагалі. Атрибутами об’єктів програми можуть бути окремі біти, бітові послідовності, числа різної форми представлення, байти і групи байтів визначаючих символи у різних формах представлення у пам’яті ЕОМ та зовнішній пам’яті у вигляді окремих файлів або у вигляді зв’язаних груп файлів тощо.

Різноманіття природи даних ускладнює їх обробку на обчислювальних машинах, виходом з цієї незручності є кодування інформації. Засоби кодування даних залежать від якості каналів зв’язку. Данні в обчислювальних машинах передаються по каналах зв’язку (шинах), які взагалі мають завади зовнішнього середовища (електор-магнітне випромінювання тощо). Так як довжина шин у комп’ютері невелика, тому його канали зв’язку вважаються без завад або з наявністю малих завад. У сучасних каналах зв’язку великої довжини передача даних відбувається за оптоволоконною технологією, котра не реагує на електромагнітне випромінювання. Засобів ефективного кодування інформації запропонований Шенноном [7] при передачі даних в каналах зв’язку без завад, наведено на рис. 1.1. При наявності завад у каналах зв’язку використовують інші моделі кодування даних.

Завади каналів зв'язку приймаються за випадкові події і данні моделюються ансамблем повідомлення B, котре утворено елементами повідомлення (алфавітом повідомлень) A і ймовірностями (алфавітом ймовірностей) настання подій елементів алфавіту повідомлення P. Ансамбль повідомлень представляється так:

$$B = \begin{cases} A: (a_1 & a_2 & \cdots & a_n) \\ P: (p_1 & p_2 & \cdots & p_n) \end{cases}, \sum_{i=1}^{n} p_i = 1.$$

За відомим ансамблем повідомлень можливо побудувати ефективні коли для передачі даних у каналах з завадами. Продемонструємо це на прикладі.

Нехай задано двійковий кодовий алфавіт $\{0,1\}$ і ансамбль повідомлень $B = \begin{cases} A: (a_1 & a_2 & a_3 & a_4) \\ P: (0.5 & 0.125 & 0.25 & 0.125) \end{cases}$, тоді побудова ефективних кодів утворюється за схемою

$$\begin{array}{l} a_1 0.51 \\ \frac{1}{0} \\ a_3 0.2501 \\ \frac{1}{0} \\ a_2 0.125001 \\ \frac{1}{0} \\ a_4 0.125000. \end{array}$$

Рис. 1.2. Кодування даних для передачі у каналах з завадами

Наведений приклад кодування задає різний рівень організації даних, чим більшу ймовірність має елемент повідомлення, тим довжина його коду буде меншою. Зрозуміло, що такий засіб кодування, запропонований Шенноном, має перевагу при передачі даних, але має недоліки при декодуванні інформації. Щоб обґрунтовано отримати результати кодування інформації, розглянемо деякі її показники.

Показник кількості (невизначеності) інформації I, діє за правилом: чим довше повідомлення Пв тим більше інформації воно несе і при рівномірному повідомлені виміряється як $I = log_c n$. Де $n = |\text{Пв}|$, c – одиниця виміру кількості інформації, якщо $c = 2$, тоді маємо *біт,* якщо $c = 10$ тоді – *дит* і інші виміри. При нерівномірному повідомлені його кількість визначається – як $I = -\sum_{i=1}^{n} p_i \, log_c \, p_i \geq 0$.

Можливо виміряти час T проходження сигналу по каналах зв'язку. Так $T = \sum_{i=1}^{n} |\bar{a}_i| p_i \tau_i$ – *часовий показник* проходження повідомлення Пв по каналу зв'язку, де $\bar{a}_i$ – код елементу a_i, τ_i – час проходження елементу коду по каналу.

Показник швидкості проходження повідомлення по каналу зв'язку підраховують за формулою $\bar{I} = \frac{I}{\tau_{cp}}$.

Пропускна спроможність каналу зв'язку η визначається за правилом $\eta = max\{\bar{I}\}$

За цими показниками для розглянутого прикладу Рис.1.2. для часу проходження одного елементу двійкового коду $\tau = \tau_i = 1c.$, маємо $I = 1.75$ біт, $T = 1.75c.$, $\bar{I} = 1.75$ біт/с., I_{max}, $\eta = 1.75$ біт/с. Отже отримані за схемою Шеннона коди є ефективними (оптимальними), тому що швидкість передачі даних співпадає з пропускною спроможністю каналу зв'язку.

Матеріали цього параграфу познайомили читача з первинними базовими поняттями алфавіту і можливостями конструювання слів та деяких їх характеристик. Розглянуті питання рівномірного і ефективного не рівномірного кодування повідомлень. Введено поняття інформації та її спеціального виду даних. Виконано знайомство читачів з основними вимірами і деякими одиницями вимірювання даних.

1.2. Структури і данні

В попередньому параграфі розглянуто поняття даних як синонім поняття інформація і пов’язане з цим питання обробки даних, зокрема кодування, тощо. У цьому параграфі розглядаються поняття даних з структурної позиції Виходячи з того, що данні в загалі, конструктивні об’єкти котрі утворені змістом і задають будову конструкції, тому в подальшому їх будемо розглядати під цими двома кутами зору.. Зміст даних пов'язаний з пам’яттю, в якій розташовуються дані у ЕОМ, їх структура ж визначається програмістом. Хоча і представлення змісту даних у пам’яті також відбувається не без його участі. Поняття структури і змісту даних відносяться до базових понять структур даних, яким присвячено цей пункт.

Під даними можна розуміти носії інформації, наприклад, змінні, константи тощо. Терміни змінна і константа у математиці розуміються, як «ім’я + значення» і відрізняються тим, що значення змінної змінюється, константи – ні. З інформаційної точки зору ці терміни визначають ім’я, значення та змінне чи постійне значення імені. У програмуванні ці дані представляються ім’ям та типом, за якими у пам’яті виділяється місце під дані, об’єм пам’яті тощо і при потребі імені даних присвоюється відповідне числове, символьне, логічне та інше значення. Отже у програмуванні данні мають структуру представлення у програмі і структуру організації пам’яті під них у ЕОМ. При відсутності даних не можуть бути реалізовані обчислювальні процеси у ЕОМ.

Структура грецький термін, що означає «будова», «порядок». У програмуванні надають перевагу структурі, як будові. Представлення структури даних (скорочено СД) може бути різним формульним, графічним та іншим. Наприклад, формульне представлення загального даного може мати вигляд «ім’я + значення + пам’ять + місце у пам’яті».

Як вказано у вступі до навчального посібника, історично розвиток структур даних проходив через етапи адресності обчислювальної машини, типізацію даних і повну типізацію на всіх рівнях розроблення програм і конструювання типовості СД. На сучасному рівні програмування перевага надається технології повної типізації обробки програм.

У сучасному програмуванні використовується також структуризація різноманітних значень даних. Розглянемо деякі прийоми структуризації значень числового числення.

Натуральні числа множини $\mathbb{N}$: 1, 2, 3, …, можна задати на алфавіті $A = \{1, +\}$ за допомогою математичної операції підстановки $(\rightarrow)$ або операції присвоєння у програмуванні $(=)$ і структурного правила.

1) $x = 1, x \in \mathbb{N}$,

2) $x \rightarrow x + 1$.

Числа: $0, \pm1, \pm2, \pm3, \ldots$ цілої множини $\mathbb{Z}$ також можна задати структурою обчислень, але на алфавіті $B = \{0,1, +, -\}$,

1) $x = 0, x \in \mathbb{Z}$,

2) $x \to x \mp 1$.

Числа $r = a/b; a, b \in \mathbb{Z}$ і $b \neq 0$, утворюють множину раціональних чисел $\mathbb{Q}$. Має місце послідовних включень $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$ і будь-яке $r \in \mathbb{Q}$ може бути представлено скінченним або нескінченним десятковим дробом.

Сучасна уніфікація представлення дійсних чисел множини $\mathbb{R}$ досягається завдяки введенню множини гіпердійсних чисел $\mathbb{R}^*$. $x^* \in \mathbb{R}^*$, таких що $x^* = x + o(\varepsilon)$, $x \in \mathbb{R}$,

де $o(\varepsilon) < \varepsilon$ – нескінченно мале число, порядок малості якого менше за $\varepsilon \in \mathbb{R}$ і яке читається як «о мале порівняно зі значенням епсілон» (*не плутати з порожнім символом раніше введеним у алфавітах*).

У машинній пам'яті застосовують число x, як число обмеженої довжини ($o(\varepsilon)$ нехтується). Нехтування величиною $o(\varepsilon)$ може призводити до машинних помилок при обчисленнях. Щоб віддалити машинну похибку від потрібного результуючого числа, застосовують типи даних *float, double, long double* з розмірами пам'яті під число – 4 байту, 8 байт і 16 байт відповідно.

Перейдемо до розгляду множино-подібних структур даних *множин*, *масивів*, *списків*.

Відомо, що під множиною розуміється абстрактна сукупність однорідних вільно розташованих у цій сукупності елементів. Елементи множини мають різні імена і різні або деякі однакові значення імен, причому одного і того ж типу (числового, символьного, строкового і ін.) і ніяк не зв'язані з місцем їх розташування у пам'яті ЕОМ. Остання вимога ускладнює функцію задання множини set у програмуванні. Похідними структурами множини є масиви та списки і інші структури даних.

Масив це множина, елементи якої закріплюються «місцем» у переліку елементів, тобто масив упорядковує множину по послідовності індексів. Під масив транслятор виділяє місце у пам'яті одним «неперервним» куском. Наприклад, одномірний набір даних $M = \{m1, m2, \dots mn\}$ розміру $dim\, M = n$, може бути заданий наступним представленням:

```
td codeMass [] = {…};  // td – визначений програмістом тип даних, {…}
– перелік елементів mk;
void doCode (td * mfist, td * mend)
// * mfist, * mend вказівники початку і кінця масиву даних;
{
   for (int i = 0; mfist [i]; i++)  {
       mend [i] = codeMass [mfist [i]];   // індексація даних;
   }
}
```

Наведена функція в залежності від вибраного типу даних індексно кодує символи, слова, строки та ін. Функція doCode реалізує тільки одну операцію «присвоєння індексу». При необхідності застосування декількох операцій до СД зручно застосовувати абстрактні моделі конструктивних породжувальних структур [5] і абстрактні типи даних ADT [1 – 3]. Наприклад, множину можна

представити абстрактним типом масив з набором множинних операцій, що, як правило, і робиться при задані множини.

Конструювання ADT даних відбувається за схемою-інтерфейсом, який містить набори операторів і методів реалізації операторів. Формат схеми містить заголовок з іменем ADT, опис типу даних та список операцій:.

ADT *ADT_name*
Дані
Опис структури даних
Операції
Конструктор
Початкові значення; // дані необхідні для ініціалізації об'єкта,
Процес; // ініціалізації об'єкта,
Операція 1
Вхід; // дані клієнта-об'єкта,
Передумова; // стан системи перед для застосування операції,
Процес; // правила виконання дій з даними,
Вихід; // дані, які повертаються форматом,
Після умова; // стан системи після завершення операції.
Операція 2
…
Операція 3
…
Кінець ADT.

Організація даних ADT у середовищі C++ використовується як структура при створені класів.

Так клас множини Set реалізується з використанням масиву і множинних операцій за об'явою:

```
const int SetSise = m;  // розмір множини m задається користувачем
const int True = 1, False =0;
class Set
{
    private:
    // дані – члени класу
          Int member [SetSise];
    Public:
    // перший конструктор створює порожню множину
             Set (void);
    // другий конструктор формує множину із n елементів, як масив
    // b[0],  b[n-1] об'єктів типу td, котрий визначається
    // програмістом. Кожен елемент множини перевіряється на
    // входження в діапазон [0, m] за значенням true індексів масиву
    Set (td b[], int n);
```

```
    Set operator+ (Set x) const; // + – символ операції об’єднання
множин
    Set operator* (Set x) const; // * – символ операції перетину
множин
    friend int operator^ (int elt, int Set x); //
}
```

Упорядкування елементів множини можливо виконати вказавши який елемент за яким слідує. Тобто елемент списку утворюється деяким ім’ям і вказівником на наступне ім’я елементу за формулою: «ім’я + (вказівник на наступне ім’я)». Наведена формула визначає конструктивний вузол даних.

Так організована послідовна структура даних зветься *лінійним списком*. Елементи лінійного списку даних можуть розташовуватися у різних частинах пам’яті. Щоб обмежити кількість елементів у лінійному списку вказівнику останнього елементу, присвоюється значення 0. Виходячи з того, що реалізація списку у машинах відтворюється над адресами пам’яті тому посилання в останньому вузлу здійснюється на адресу NULL. Посилання на початковий вузол (голову) списку відбувається за *вказівником голови* (head). Список без вузлів представляється формулою «head → NULL» або більш математизованою формулою $\pi(head, NULL)$, в яких символи → і π – задають відношення посилання. Структура лінійного списку з відношенням → наведена на рис. 2.1.

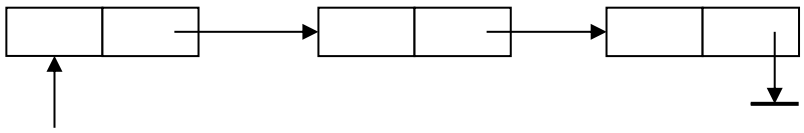


Рис. 2.1 Структура лінійного списку

Організація вузлів лінійного списку і деяких операцій задається наступною ADT

ADT *ADT Node*

Дані

Вузол Node утворюється полем даних Date і полем вказівника Next. Значення Node не використовується ніде крім ініціалізації. Конструктор класу Node ініціалізує поле відкритих даних Date і поле закритих вказівників Next. По замовчуванню значення Next встановлюється на адресу NULL. Доступ до поля Next і змінити його значення дозволяється функціям членам InsertAfter і DeleteAfter.

Операції

Конструктор

Початкові значення: Значення даних і вказівника на наступний вузол.

Процес: Ініціалізація полів вузла.

NextNode

Вхід:	Немає.
Передумова:	Немає.
Процес:	Вибірка значення поля Next.
Вихід:	Повертає значення поля Next.
Після умова:	Немає.

InsertAfter

Вхід:	Вказівник на новий вузол.
Передумова:	Немає.
Процес:	Налаштування значення Next для встановлення адреси вказівника на новий наступний вузол.
Вихід:	Немає.
Після умова:	Вузол вказує на новий вузол.

DeleteAfter

Вхід:	Немає.
Передумова:	Немає.
Процес:	Від'єднання наступного вузла і присвоєння значення Next для посилання на вузол, котрий є наступним за видаленим вузлом.
Вихід:	Вказівник вказує на видалений вузол.
Після умова:	Вузол має нове значення Next.

Кінець *ADT Node.*

Конкретне представлення *ADT Node* виконується об'явою класу Node.

Об'ява

```
template <class T>
class Node
{
     private :
            // next вказує на адресу
            // наступного вузла
               Node<T> *next;
     public :
            // відкриті дані до доступу
            T data;
            // конструктор
            Node (const  T&  item,  Node <T>*  ptrnext  =  NULL) ;
            // методи модифікації списку :
            void  InsertAfter (Node<T> *p) ;
            Node<T>  *DeleteAfter (void) ;
            // отримання адреси наступного вузла
            Node<T> *NextNode (void) const ;
```

}

Розглянуті лінійні списки не вичерпують усіх різновидів можливостей представлення списків. На рис. 2.2. наведена структура простого кільцевого списку, який організовано за допомогою орієнтації його голови head на останній вузол і присвоєння значенню заключного вказівника NULL адреси першого вузла.

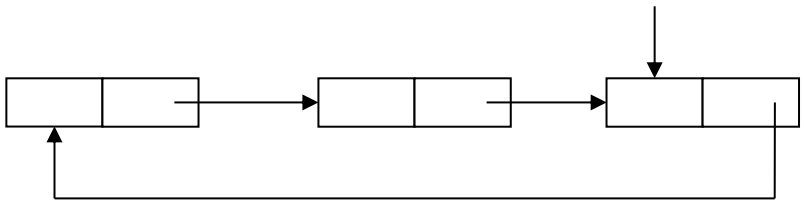


Рис.2.2. Структура простого кільцевого спису

Зрозуміло, що процес проходження кільцевого списку нескінчений, обмежити кількість проходів можна за допомогою введення у процес лічильника або певної умови виходу з процесу.

Існує ціла низка базових структур даних зі своїми методами обробки, котрі визначаються відповідним ADT. Застосування інструментарію проектування структур даних з використанням ADT для складних конструктивних об'єктів не завжди зручно. Для цього краще підходить методика конструктивних структур КС [5].

Наведемо декілька конструктивних формул, за якими можна побудувати складні об'єкти даних (ОД).

ОД – масив масивів.

$\varphi(tda, int\ i) = a[i]$ // функція індексації даного типу td

$\{a[i]\}$ // формування одно мірного масиву

$\varphi(\{a[i]\}, int\ j) = \{a[i]\}[j]$ // індексація масиву

$\{\{a[i]\}[j]\}$ // формування двомірного масиву масивів

$\varphi(\{\{a[i]\}[j]\}, int\ k) = \{\{a[i]\}[j]\}[k]$ // індексація масиву масивів

$\{\{\{a[i]\}[j]\}[k]\}$ // формування тримірного масиву масивів

Кінець ОД

Аналогічно можуть бути побудовані, важливі для застосування в предметних галузях формули конструювання багатомірних таблиць даних з елементами $a[i][j][k]\cdots[n]$.

Будуються також гібридні ОД такі як списки масивів, масиви списків, кільцеві багаторівневі списки множин тощо. До гібридних ОД можливо застосовувати комбінований підхід проектування на основі ОД і ADT. Так структура конструювання масиву лінійних списків буде наступною.

ОД – таблиця списків

ADT Node→ s // конструювання вузла

$\psi(s, *next) = \{s\}$ //формування списку за вказівником $next$

$\varphi(\{s\}, int\ i) = \{s\}[i]$ // функція індексації списку
$\{\{s\}[i]\}$ // формування одно мірного масиву списків
Кінець ОД.

Параграф знайомить читача з поняттям структури і її застосуваннях при відтворені даних. Розглянуто формування деяких базових структур даних і конструктивних об’єктів СД. Наведені деякі технології проектування структур даних і розглянуто декілька прикладів конструювання складних структур даних.

1.3. Алгоритми: представлення, реалізація, аналіз

Метою даного параграфу є надання загальних відомостей з базового поняття алгоритм, його властивостей, конструктивних можливостей, представленням та реалізації, знайомство з методологію аналізу для встановлення алгоритмічних характеристик. Надання деяких відомостей з прийомів організації рекурсивних обчислень для прискорення обробки даних.

Поняття алгоритму досить важливе в діяльності і здобутті знань людиною. Це поняття виникає природним чином у людини в результаті її діяльності і проявляється різним чином в залежності від обставин, в яких вона знаходиться, від досвіду людини та іншого.

На інтуїтивному рівні під алгоритмом розуміють деяку послідовність дій для досягнення поставленої мети. Але розмитість такого визначення не дає можливості застосовувати його у програмуванні, хоча б з того, що невідомо про які дії іде мова та хто є виконавцем дій?

На сучасному рівні інтелекту людини не вдається загально визначити алгоритм, але його можна коректно визначити в часткових випадках. Наприклад, якщо задати об'єкт, для якого формулюють алгоритм, виконавчий механізм (виконавець), оператори-дії, котрі може виконувати цей механізм, тоді визначити алгоритм можна чітко. Існує велика кількість механізмів для задання алгоритмів, як правило, в теоретичних дослідженнях використовують прості механізми з невеликою кількістю операторів, у практичній інтелектуальній діяльності людина використовує ЕОМ з великою кількістю операторів (декілька сотень операторів).

Перейдемо до коректнішого визначення алгоритму обчислення функції.

Нехай задана *числова функція* $f: x \to y$, $x, y \in \mathbb{N}$ і деякий механізм M, який може виконувати дії за операторами скінченної множини $\{O_i\}$. Тепер можна визначити алгоритм обчислення функції f. При цьому необхідно врахувати основну властивість алгоритму – масовість, тобто його виконання на деякій множині вхідних даних.

Визначення 3.1. Функція f обчислена у точці $\boldsymbol{x}$, якщо її можна представити скінченною послідовністю операторів $\langle O_j O_k \cdots O_m \rangle$ такою, що за кінцеву кількість кроків дій механізм M зупиняється і видає результат b, який приймається за значення функції $f(x) = b \in Y$.

Визначення 3.2. Алгоритмом A_X^Y обчислення функції f називають її обчисленність у будь-якій точці $x \in X$.

Останнє визначення враховує властивість масовості алгоритму, за яким множина $X = DomA_X^Y$ визначає область його визначеності $DomA_X^Y$ і область значень $RunA_X^Y$ таку, що $Y = RunA_X^Y$. Між областями визначення та значень алгоритму і функції f існують включення $X \subseteq Domf(x)$, $Y \subseteq Runf(x)$.

Характерним для алгоритму є його конструктивність, тобто він сприймається як конструктивний об'єкт, який має вхід і вихід, має представлення і реалізацію, конструктивно формує предметні об'єкти (рішення, функції, процеси тощо), має директивний напрямок виконання і ін. Алгоритм здобуває, переробляє, зберігає та сприяє передачі інформації і даних. На алгоритм можна дивитися, як на процес за яким відбувається представлення і його реалізація. Представлення алгоритму може бути задане на будь якому носію інформації. Зокрема, у програмуванні це програма написана на відповідній алгоритмічній мові, структурні схема програми різного рівня деталізації, файли та інше.

Застосування алгоритмів необхідно для здобуття знань (інформації) при чому відповідних знань можна досягти різним чином за допомогою різних алгоритмів. Звідси виникає необхідність оцінки алгоритмів за часом виконання (показник швидкості), за об'ємом використаної пам'яті та інше.

Дамо декілька визначень оцінки алгоритму за його представленням і реалізацією.

Визначення 3.3. Оцінка алгоритму за представленням або *об'єм програми* Pr – $V(Pr)$ = (кількість строк програми).

Будь який алгоритм за своїм представленням задає структуру, яка утворюється послідовностями лінійних операторами, операторів керування (розгалуження і циклів) і ін. В загалі, структура програми може бути лінійною або нелінійною при наявності в ній керуючих операторів. Отже структура програми утворюється операторами і зв'язками між ними.

Визначення 3.4. *Структурна складність* алгоритму і програми Pr або *цикломатичне число* за Мак-Кейбом $\nu(Pr) = n - m + 2$, де n – кількість зв'язків між операторами, m – кількість операторів програми.

За показником ν оцінюється не тільки алгоритм але й професійна якість колективів програмістів тощо.

Розроблений алгоритм вирішення задачі може видавати наближений результат, який оцінюється за допомогою показника «о мале по відношенню до малого значення ε» $o(\varepsilon)$. Прикладам таких алгоритмів є наближені алгоритми вирішення рівнянь тощо. Як правило, така оцінка робиться математичними прийомами на основі порівнянь між уявленим точним і наближеним рішеннями.

Оцінити складність алгоритму за його реалізацією можна за допомого показника (порядок) *складність реалізації* – $O(n)$, котрий читається як «О велике від n або порівняно з n». Показник $O(n)$ вказує на узагальнену кількість реалізацій алгоритму, наприклад, якщо розрахунок кількості дій дорівнює $(3n + 2)n + 2n + 4 = 3n^2 + 4n + 4$ тоді, нехтуючи константами і членами цього виразу у степені меншій за 2, отримаємо $O(n^2)$. В залежності від особливостей задачі, для якої розробляється алгоритм показник його складності реалізації може розраховуватися по найгіршій, середній або

найкращій реалізаціях. Як правило, для характеристики алгоритму вибирається показник складності за найгіршим варіантом реалізації.

Розглянемо для прикладу простий механізм – машину з необмеженими регістрами (МНР) [5], котра утворена пів-обмеженою стрічкою, поділену на регістри $R_1, R_2, R_3, \dots,$ $R_1 R_2 R_3$ r_1 r_2 r_3, в яких містяться значення даних r_i і набір операторів реалізації $\langle S, Z, T, J, E \rangle$ на ній.

Реалізація алгоритмів на МНР машині виконується за численням, оператори в якому виконуються за правилами:

$S(k): r_k = r_k + 1$, одномісний оператор слідування, за яким зміст k -го регістру збільшується на одиницю,

$Z(k): r_k = 0$, оператор обнуління змісту k -го регістру,

$T(k, m): r_k = r_m$, двомісний оператор пересилки даних з m –го регістру у k -й регістр,

$J(k, m, q)$ – тримісний умовний оператор, за яким при виконанні умови $r_k = r_m$, відбувається перехід до оператора з позначкою q, інакше виконується наступний оператор після умовного оператора,

E – тотожній оператор, який вказує на зупинку «роботи» МНР;

Оператори S, Z, T змінюють стани регістрів, сукупність станів регістрів стрічки утворює конфігурацію стрічки МНР машини,

- вхід МНР алгоритму визначається початковою конфігурацією стрічки машини,
- послідовність конфігурацій стрічки визначає протокол реалізації алгоритму,
- конфігурація стрічки заключна, якщо результат реалізації алгоритму міститься у першому регістрі і машина зупинена, при цьому, стан першого регістру приймається за вихід алгоритму,
- послідовність операторів з позначками $q_i, 1 \le i \le n$ утворює програму Pr представлення алгоритму.

Задача 1. Побудувати МНР алгоритм A_1 обчислення функції. $f(x, y) = x + y, x, y \in \mathbb{N}$.

Нехай вхід алгоритму задано початковою конфігурацією $R_1 R_2 R_3$ x y 0. Тоді програма Pr1 представлення і протокол реалізації алгоритму при $x = 2$ і $y = 3$ наступні:

$$q_1: J(2,3,q_5),$$
$$q_2: S(1),$$
$$q_3: S(3),$$
$$q_4: J(1,1,q_1),$$

$$q_5: E; \quad \begin{matrix} R_1 & R_2 & R_3 \\ 3 & 2 & 0 \\ 4 & 2 & 0 \\ 4 & 2 & 1 \\ 5 & 2 & 1 \\ 5 & 2 & 2 \end{matrix}.$$

Для алгоритму A_1 об'єм програми $V = 5$ і кількість дій $2y = 2n$ реалізації, тому складність алгоритму $O(n)$. Цикломатичне число за Мак-Кейбом $\nu = 5 - 5 + 2 = 2$.

Задача 2. Побудувати МНР алгоритм A_2 обчислення функції $f(x, y) = xy$, $x, y \in \mathbb{N}$.

Початкова конфігурація і програма алгоритму A_2 для неї будуть такими:

$$0 \quad y \quad x \quad 0 \quad 0 \quad , \qquad \begin{matrix} R_1 R_2 R_3 R_4 R_5 \\ q_1: J(2,5, q_9), q_6: J(1,1, q_3), \\ q_2: S(5), q_7: Z(4), \\ q_3: J(3,4, q_7), q_8: J(1,1, q_1), \\ q_4: S(1), q_9: E. \\ q_5: S(4), \end{matrix}$$

Для алгоритму A_2 кількість операцій $2xy = 2n^2$, тому $O(n^2)$ і цикломатичне число за програмою Pr2 є $\nu = 3$. Алгоритм A_2 складніший ніж – A_1 за показниками V, O і ν.

В деяких випадках вдається зменшити алгоритмічну складність обчислення функцій, завдяки попередньому спрощенню вигляду функції або застосуванню штучних прийомів обчислень. Один з таких прийомів пов'язаний з рекурсивним представленням функції і подальшим обчисленням. Розглянемо прийоми рекурсивного обчислення функцій

Функція f рекурсивно обчислена, якщо її вдається представити виразами $f(x_k) = f(x_{k-1})$, за яким значення функції у наступній точці x_k розраховують через значення функції у попередній точці x_{k-1} або $f(x_k) = f(g(x_{k-1}))$, може статися, що $f = g$. Рекурсивний процес обчислення алгоритмічно можна задати явно і неявно циклічно. У програмуванні ці процеси реалізується за допомогою операторів циклів, або рекурсивною процедурою.

Розглянемо приклад обчислення числової функції $f(n) = n!$, $n \in \mathbb{N}$. При умові, що $0! = 1$.

Схематичний фрагмент рекурсивного обчислення факторіала можна записати через повторення операції множення $k \cdot f(k-1)$

```
long f(int n)
{
     if (n < 0) return 0;
     if (n == 0) return 1;
     return n * f(n-1);
}
```

У математичному забезпечені середовищ програмування обробка масивів, списків, дерев, трансцендентних функцій і ін., виконується за допомогою рекурсивного процесу.

Розглянемо ряд $\sum_{k=1}^{\infty}\frac{x^k}{k}$ при умові $|x|<1$. Сума цього ряду існує і дорівнює S, але це значення вдається обчислити приблизно. Безпосереднє знаходження суми за рядом вимагає багато часу. Скоротити час обчислення для приблизного знаходження S можна за допомогою рекурентного правила:$S_{n+1}=S_n+\frac{y_{n+1}}{n+1}, y_{n+1}=y_n x,$ в якому $n=0,1,2,\ldots$ і $S_0=0,\ y_0=1.$ Рекурентне правило дозволяє організувати рекурсивний процес і знайти наближену оцінку $o(\varepsilon)$ по заданому значенню ε, за результатом порівняння значень сум S_n і S_{n+1}.

Складніший випадок обчислень маємо для тригонометричних функцій cos і sin, які обчислюються через ряди $cos\,x=\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n}}{(2n)!}$, $sin\,x=\sum_{n=1}^{\infty}(-1)^{n-1}\frac{x^{2n-1}}{(2n-1)!}$. Радіус збігання рядів нескінчений, отже за цими формулами знаходяться їх значення при довільних значеннях змінної x.

Можливо обчислити функції $cos\,x$ і $sin\,x$ безпосередньо за наведеними рядами, але для цього необхідно організувати дві рекурсії обчислень виразів x^n та $n!$ і організувати прямий цикл для знаходження суми, тому такий процес обчислень нераціональний. Краще виконати обчислення за однією рекурсією, перетворивши ряд за схемою Горнера.

Для прикладу, скористуємося схемою Горнера: при перетворенні ряду тригонометричної функції cos

$$1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^2}{6!}+\cdots=1+\frac{x^2}{1\cdot2}\left(-1+\frac{x^2}{3\cdot4}\left(1+\frac{x^2}{5\cdot6}(-1+\cdots)\right)\right). \quad (3.1)$$

З цього представлення видно, що вирази перед дужками мають загальний вигляд $(-1)^k+\frac{t}{(2k+1)(2k+2)}$, $k=0,1,2,\ldots$, де $t=x^2$. Звернемо також увагу на те, що добутки у формулі (3.1) зростають, тому рекурсивна формула для розрахунку функції $cos\,x$повинна мати вигляд

$$f(n)=1+\frac{(-1)t}{k\cdot2k}f(n-1,k+2), \quad (3.2)$$

де n – кількість членів ряду, k поточне значення таке, що $k\le n$, і $f(0,k)=0$.

Розглянемо процес формування ряду за формулою (3.2):

$n=1, k=1, f(1)=1;$

$n=2, k=1, f(2)=1+\frac{(-1)t}{1\cdot2}, f(2,k)=1+\frac{-t}{k\cdot(k+1)},$

$n=3, k=1, f(3)=1+\frac{-t}{1\cdot2}\left(1+\frac{-t}{3\cdot4}\right)=1+\frac{-t}{1\cdot2}f(2,k+2)=f(3,k),$

$$n = 4, \qquad k = 1, \qquad f(4) = 1 + \frac{-t}{1\cdot 2}\left(1 + \frac{-t}{3\cdot 4}\left(1 + \frac{-t}{5\cdot 6}\right)\right) =$$

$$= 1 + \frac{-t}{1 \cdot 2} f(3, k+2) = f(4, k), \ldots$$

Отже за формулою (3.2) можна побудувати рекурсивну програму для обчислення функції $cos\, x$. Процес обчислення повинен завершуватися при умові, що різниця $f(n-1) - f(n)$ є малою величиною $o(\varepsilon)$ відносно заданої помилки ε.

Однак, процедура обчислення трансцендентних функцій за допомогою рядів, також буває не доречною. Пояснимо це на прикладі функції tgx, яка представляється рядом $tgx = \sum_{n=1}^{\infty} \frac{2^{2n}(2^{2n}-1)|B_{2n}|}{(2n)!} x^{2n-1}$, $x^2 < \frac{\pi^2}{4}$, у якому $|B_{2n}|$ абсолютна величина рекурентно визначених чисел Бернуллі

$B_{2n} = -\frac{1}{2n+1} + \frac{1}{2} - \sum_{k=2}^{2n-2} \frac{2n(2n-1)\cdots(2n-2k+2)}{k!} B_k$.

Зрозуміло, що обчислення функції tgx у такий спосіб складно. Тому обчислення цієї функції краще виконувати за формулою $tgx = \frac{sin\, x}{cos\, x}$, у якій ряди для функцій $sin\, x$ і $cos\, x$ одно типові, і можна організувати один рекурсивний процес, з якого вибирати необхідні значення по черзі для функцій sin і cos .

Однак ситуація покращується, якщо скористатися представленням функції tg ланцюговим дробом:

$$tgx = \cfrac{1}{\cfrac{1}{x} - \cfrac{1}{\cfrac{3}{x} - \cfrac{1}{\cfrac{5}{x} - \cfrac{1}{\cfrac{7}{x} - \ddots}}}}$$

Зауважимо, що від ланцюгового дробу легко перейти до ряду, але перехід від ряду до дробу це мистецтво.

Особливості ланцюгових дробів їх простота, рекурсивне представлення і обчислення, оптимальна кількість операцій і їх типізація. Так у наведеному дробу явно використовується дві операції: віднімання і ділення та одна неявна операція складання $k_{i-1} + 2 \to k_i$ для отримання чисел 3, 5, 7, ….

Для алгоритмів існує проблема алгоритмічної вирішеності, тобто не для всякої задачі існує алгоритм її вирішення. Наприклад, для будь якого рівняння $f(x) = 0$ не існує загального алгоритму знаходження його рішення,
проблема зупинки реалізації програми алгоритмічно не вирішується, проблема захисту даних взагалі алгоритмічно не вирішується і ін. Однак, у багатьох випадках проблема алгоритмічного вирішення розв'язується частково, наприклад розглянуті вище проблеми.

Введене поняття алгоритму знайомить здобувача знань з основами його визначення, сторонами формування та реалізації, розглянуто питання

знаходження характеристичних показників та проблем вирішення алгоритмів. Наведені деякі прийоми рекурсивного обчислення функцій.

РОЗДІЛ 2. ПРОСТІ СТРУКТУРИ ДАНИХ І ЇХ ПРЕДСТАВЛЕННЯ

Організація пам’яті машини під певні дані необхідна як для збереження даних, так і для їх обробки. Таку організацію надають автоматні структури, прості спискові структури і складні нелінійні структури даних Організація та операції обробки таких даних розглянуто у наступних трьох параграфах цього розділу: *автоматні структури даних, обробка множинних структур даних, особливі списки*

2.1. Автоматні структури даних

Структури даних можуть бути пов’язані з певною структурою організації пам’яті і доступу до неї. У випадку організації даних подібно захищеному списку з відповідним доступом до них і обробкою, утворюють стеки, черги і ін. Такі структури даних моделюються автоматами. Нижче дано знайомство з автоматами над пам’яттю, методами доступу, алгоритмами обробки і їх програмною реалізацією.

Поняття автомату займає важливе місце в теорії моделювання поведінки реальних систем. Абстрактний автомат представляється упорядкованою п’ятіркою $A = \langle X, Y, Q, \varphi, \psi \rangle$, в якій X і Y вхідний та вихідний алфавіти, Q множина станів автомату, $\varphi: X \times Q \to Q$ функція зміни станів і $\psi: X \times Q \to Y$ функція виходу.

Типи автоматів можуть бути різними, автомати без входів, автомати без виходів і ін. Представлення автоматів визначається способом задання функцій φ та ψ і може бути табличним, графічним або іншим. Так як нас цікавить структурна частина даних тому будемо розглядати графічне представлення автомату у вигляді нескінченної стрічки або декількох стрічок. З моделлю автомату ми вже зустрічалися при задані МНР механізму у третьому параграфі попереднього розділу.

Як вказувалося у матеріалах параграфу 1.3, конструктивний об’єкт визначається змістом і його структурою, тому зміст об’єкту будемо ототожнювати із станом автомату та структуру – з графічним представленням автомату. На рис. 1.1. наведено зміст (стан) даного і структура (граф) однострічкового автомату [5].

$a_1 a_2 a_3 q_i a_4 \cdots a_n$, ∇_q

$a_1 \quad a_2 \quad a_3 \quad a_4$

Рис. 2.1. Зміст даного, стан стрічки і структурний граф однострічкового автомату

На рис 2.1. символом q_i позначено i -й стан вказівника, за яким проглядається елемент змісту об'єкту з іменем a_4 та виконуються відповідні операції над цим екземпляром СД.

Однострічковими автомати моделюються множини, мультомножини, масиви, списки, кортежі та інші структури даних. Тому, що їх представлення просте та наглядне і досить зручно демонструє процеси реалізації операцій над даними у цих структурах.

Багатострічкові автомати також є жаданими при розгляді складних СД таких, як множини множин, масиви масивів, таблиць, масиви списків тощо.

Кожна стрічка багатострічкового автомату є звичайною, як у одно стрічковому автоматові, але стрічки можуть оброблятися синхронно або як-то інакше. При синхронній обробці стрічок вказівники зв'язані (переміщення одночасне в одному напрямку) і зміна стану вказівників відбувається синхронно, хоча операції оброби даних на кожній стрічці свої.

На рис. 2.2. наведено графічне зображення багатострічкового автомату.

$$\begin{array}{l} a_1 a_2 a_3 q_i a_4 \cdots a_n \\ \vdots \\ b_1 b_2 b_3 q_j b_4 \cdots b_n \quad \nabla_{q1} \\ \\ \nabla_{q2} \\ \\ \vdots \\ \nabla_{qk} \end{array}$$

Рис.2.2. Значення, стани і граф багатострічкового автомату.

Розглянемо деякі типи одно стрічкових автоматів пам'яті і структури даних, які моделюються цими автоматами.

Конструкція пам'яті типу магазин має графічне представлення зображене на рис. 2.3.

$$X, Y \rightleftarrows$$

Рис. 2.3. Автомат типу магазин

Характерним для магазинного автомату є те, що вхід і вихід з автомату здійснюється через один кінець стрічки (голова стрічки) і автомат немає вказівника, тобто стан магазину змінюється тільки через його голову. Отже над магазином виконуються дві операції (в) і (із) магазину за принципом LIFO (last-in/first-out). Магазин може бути організовано як для статичної, так і динамічної СД. Магазин у якому дозволяється доступ до даних за вказівником називається стеком. Стек має представлення (див. Рис.2.4.)

∇_q

$X, Y \rightleftarrows$ ∇_q

$X, Y \rightleftarrows$ $\rightleftarrows X, Y$

Рис. 2.4. Графічне представлення структури стеку і деку

На рис.4.4. також представлено СД дек, як стек з двома головами.

Якщо у магазині роз'єднати голову на дві частини вхідну і вихідну, або в декові дозволити прохід тільки в одному напрямі, тоді будемо мати тип структури даних черга, яка зображена на рис. 2.5.

$a_{k-1} \leftarrow$ a_k $a_{k+1} \leftarrow a_i$

Рис. 2.5. Структура автомату черга

Організація СД стек (stack) і деяких колекцій на ньому наступна. Стек задається певним розміром (StackSize) і над ним виконуються дві операції-функції: «із стеку» (push) і «в стек» (pop). Вказівник стеку (top) вказує на порядкове місце елементу в стеку при його видалені або вставці. Якщо , top$=-1$, стек порожній.

Розглянемо приклад організації [1] та реалізації даних і операції над стекам

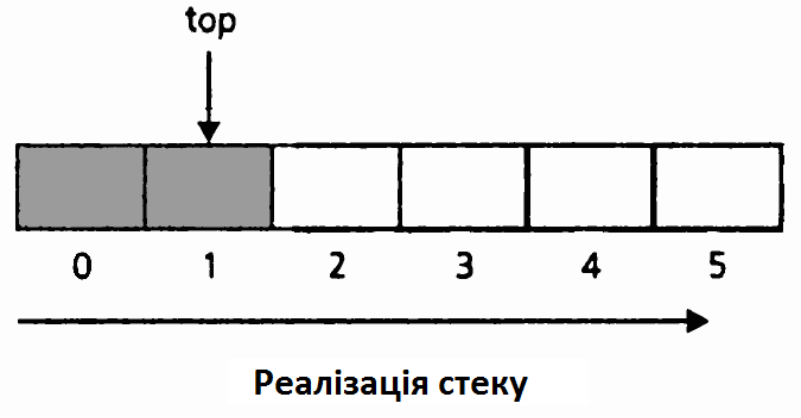


Рис. 2.6. Структура стеку

В ній top – вказівник стеку (індекс стеку), який може приймати значення $top = -1|0|1|2|\cdots|n$; при $top = -1$, стек порожній. По замовченню розмір стеку $n = MaxSize = 50$.

Об'явлення об'єкту типу Stack включає розмір стеку. При цьому список (stacklist), максимальна кількість елементів. У стеку (size) та індекс (top) є закритими членами, а операції – відкритими.

Операції (функції): $Push$ – в стек, $top = 0,1,2,\cdots$;

Pop – із стеку, $top = \cdots,2,1,0$ є вказівником на комірку стеку.

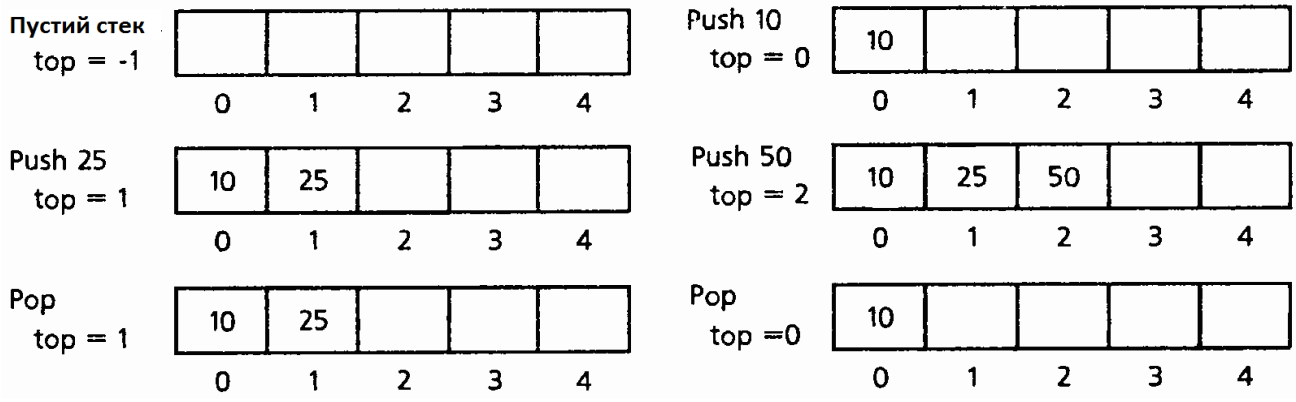


Рис. 2.7. Робота стеку

Специфікація класу Stack

Об’явлення

```
#include <iostream.h>
#include <stdlib.h>
const int MaxStackSize = 50;
class Stack
{
private:
// закриті данні-члени, масив стека та вершина (індекс)
DataType stacklist[MaxStackSize];
int top;
public:
// конструктор; ініціалізує вершину
Stack (void);
// операції модифікації стека
void Push (const DataType& item);
DataType Pop (void);
void ClearStack(void);
// доступ до стека
DataType Peek (void) const;
// методи перевірки стеку
int StackEmpty(void) const;
int StackFull(void) const; // реалізація масиву;
};
```

Стек – зберігає проміжні дані різних типів, дані результатів операцій і ін. Розглянемо обчислення арифметичних виразів за допомогою стеку:

Вирази можуть представлятися різним чином. *Інфіксною* формою представлення $y = a * b + c$, *постфіксною* формою представлення $y = ab * c +$.

Сканується вираз $\overrightarrow{ab*}c+$ у постфікс ній формі і заноситься у стек $\overset{*}{\underset{a}{b}}$, обробляється алгоритмом стеку – d. Отже маємо вираз $dc +$. Продовжується

сканування $dc\overrightarrow{+}$ і занесення у стек $\begin{matrix}+\\c\\d\end{matrix}$. Після обробки маємо результат обчислення у стеку – y.

Над деком, аналогічно стеку, виконуються операції «в дек» PuchFront PuchBack (через голову і хвіст), «з деку» PopFront PopBack і Empty – «перевірка змісту деку на порожність».

Перейдемо до питання обробки черг (queue), як – спеціально представленого списку.

Розглянемо демонстраційний приклад побудови списку черги.

Вказівниками черги є: front і rear – початок і кінець черги.

Операції: QInsert і QDelete вставки і видалення даних черги.

Рис. 2.8. Зміни у черзі при виконанні операцій [1]

Крім наведених операцій можуть використовуватися і інші операції над чергами. Наведемо специфікацію класу для організації черги.

Специфікація класу Queue

Об’явлення

```
# include <iostream.h>
# include <stdlib.h>
// Максимальний розмір списку черги
const int MaxQSize = 50;
class Queue
{
private:
// Масив і параметри черги
    int front, rear, count; // count містить число елементів в черзі
    DataType qlist [MaxQSize];
public:
// Конструктор
```

```
     Queue (void);     // initialize integer data members
// Операції модифікації черги
     void QInsert (const DataTypes item);
     DataType QDelete (void);
     void ClearQueue (void);
// Операція доступу, повертає значення на початку черги.
     DataType QFront (void) const;
// Методи тестування черги
     int QLength (void) const; // перевіряє довжину черги
     int QEmpty (void) const; // перед видаленням
     int QFull (void) const; // перед вставкою
};
```

Крім наведених лінійних черг існують і інші, зокрема кругові черги.

Розглянемо приклад кругової черги довжиною 4, в яку необхідно вставити послідовність елементів (3,5,6,2,8)

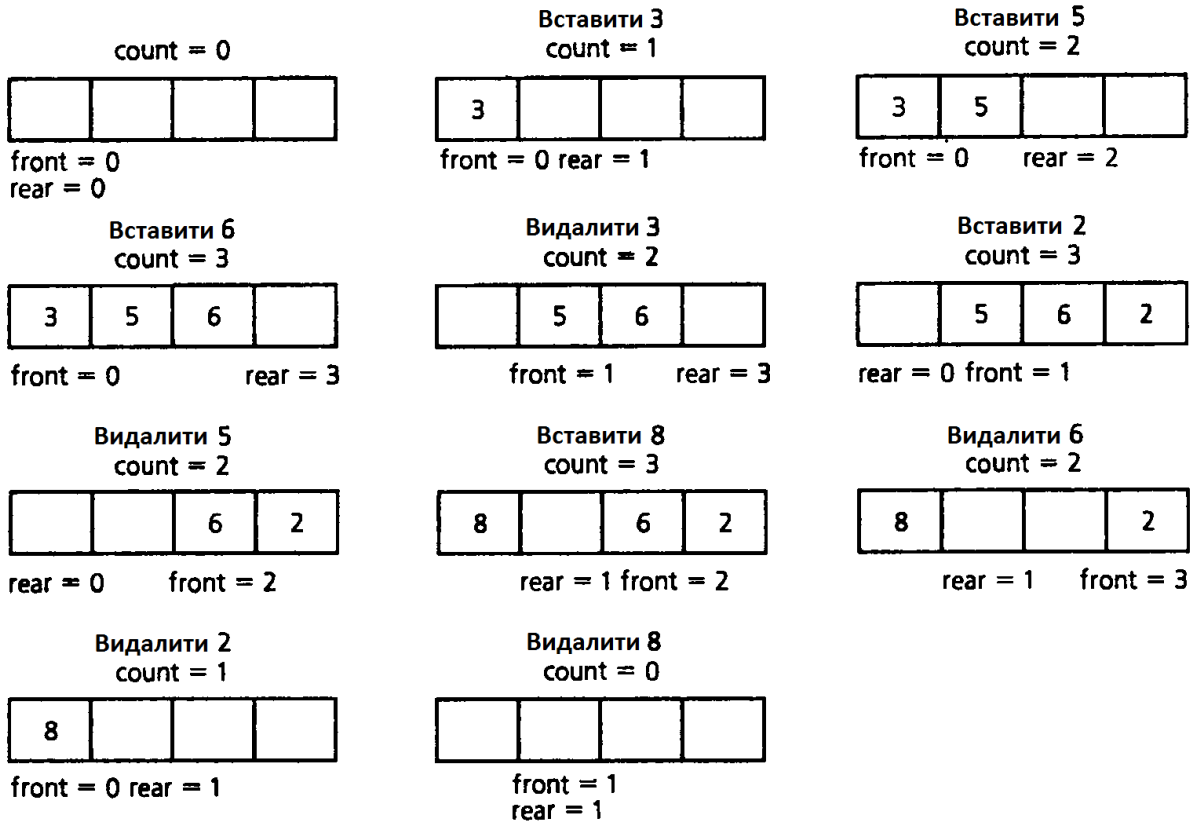


Рис. 2.9. Операції вставки та видалення у круговій черзі [1]

Наведемо фрагмент програми застосування операцій (функцій) при обробці черги.

```
typedef int DataType;
# include aqueue.h
Queue Q;  // оголошуємо чергу
Q.QInsert(30);  // вставляємо 30 в чергу
Q.QInsert(70);  // вставляємо 70 в чергу
cout <<Q.QLength()<< endl;  // друкує 2
```

```
cout <<Q.QFront()<< endl;  // друкує 30
// QEmpty – функція тестування черги перед операцією видалення
if (! Q.QEmpty())
cout <<Q.QDelete();  // друкує значення 30
cout <<Q.QFront()<< endl;  // друкує 70
Q.ClearQueue();  // очищення черги
```

На основі автоматного підходу, дано ознайомчі матеріали з основними СД пам’яті. Показано як можна застосувати структуру стеку для виконання обчислень арифметичних виразів. Розглянуто структуру представлення стеків, лінійних та нелінійних черг і операцій над ними.

2.2. Обробка множинних структур даних

У попередньому параграфі введено поняття структур списків як даних утворених зв’язаними вузлами Node, котрі складаються з полів значень Date і Next. Вузли з їх полями даних і покажчиків є будівельними блоками пов'язаного списку. Структура вузла дає можливість застосування операцій, які ініціалізують дані-члени, і методи управління вказівниками для доступу до наступного вузла. У цих матеріалах розглянуті структури операцій над списками і зокрема над їх вузлами, дається знайомство з основними алгоритмами обробки пов'язаних списків, які застосовуються у більшості алгоритмічних програм. Метою параграфу є знайомство з структурою списків, їх конструюванням, організацією основних операцій і алгоритмів та програмної реалізації цих операцій.

Нагадаємо, що вузол (node) складається з поля даних і вказівника на наступний об’єкт у списку. Зв’язаний список складається з вузлів, перший елемент якого (front), - це вузол, на який вказує голова (head). Список пов'язує вузли разом від першого до кінця або хвоста (rear) списку.

Спочатку розглянемо базові операції над вузлами списку InsertAfter і DeleteAfte. У будь-якому вузлу p ми можемо реалізувати операцію InsertAfter (див. рис.2.10.), яка приєднує новий вузол після поточного.

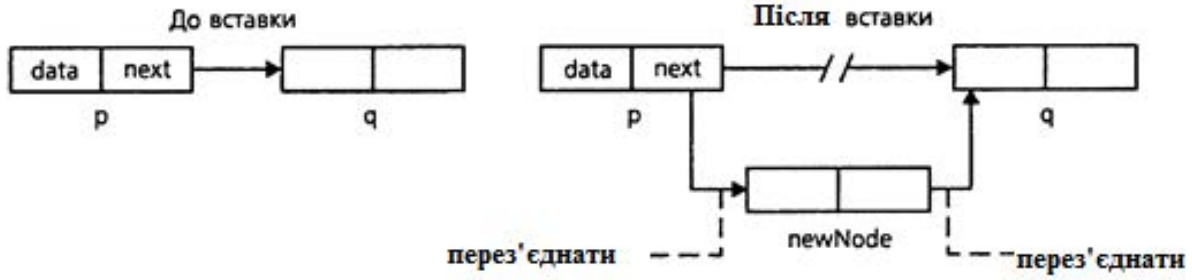


Рис. 2.10. Схема вставки вузла у список

Аналогічний процес описує операцію DeleteAfter, котра видаляє вузол, наступний за поточним, як це демонструє рис. 2.11. Спочатку роз'єднуються сусідні вузли p та q і потім з'єднується вузол p з вузлом, який слідує за – q.

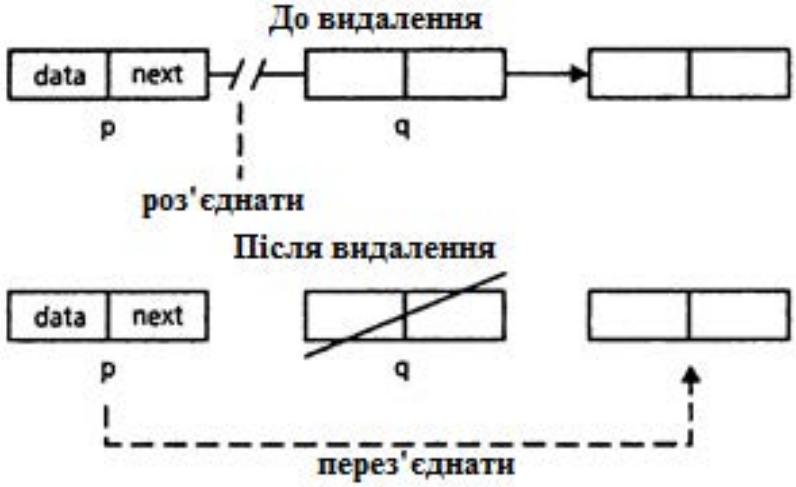


Рис. 2.11. Схема видалення вузла спису

Клас Node містить як відкриті, так і закриті дані-члени. Згадаємо, що відкриті дані-члени призначаються для того, щоб користувач і класи колекцій могли мати прямий доступ до їх значень. Поле next є закритим, і доступом до цього поля мають керуючі функції-члени. Якщо зробити це поле відкритим, тоді користувач отримає можливість порушувати зв'язок і знищувати зв'язаний список. Клас Node містить функцію-член NextNode, яка дозволяє клієнту проходити по зв'язаному списку.

```
// Конструктор, ініціалізація даних і вказівника
template <class T>
Node<T>::Node(const T& item, Node<T>* ptrnext) :
data(item), next(ptrnext)
{}
// Метод NextNode надає клієнтові доступ до поля next. Цей метод
повертає значення next і використовується для проходження по списку.
// Повернути закритий член next
template <class T>
Node<T> *Node<T>::NextNode(void) const
{
return next;
}
```

Представлення алгоритму вставки вузла. АлгInsertAfter приймає вузол p в якості параметра і додає його до списку в якості наступного вузла. Спочатку поточний об'єкт вказує на вузол, адресою якого є q (значення в полі next). Алгоритм змінює два вказівники. Поле вказівника p встановлюється на q, а полю вказівника в поточному об'єкті присвоюється значення p.

```
// вставити вузол p після поточного вузла
template <class T>
void Node<T>::InsertAfter(Node<T> *p)
{
     // p вказує на наступний за поточним вузол,
```

```
      // а поточний вузол вказує на p..
      p->next = next;
      next = p;
}
```

Порядок присвоєння вказівників, важливий. Наприклад, присвоювання має зворотний порядок.

```
      next = p; // Вузол, наступний за поточним об'єктом, втрачено!
      p->next = next
```

Алгоритм DeleteAfter видаляє вузол за поточним об'єктом і пов'язує його поле вказівника з наступним вузлом. Якщо після поточного об'єкту немає вузла (next == NULL), функція повертає NULL. Інакше, функція повертає адресу віддаленого вузла для випадку, якщо програмісту необхідно звільнити пам'ять цього вузла. Алгоритм DeleteAfter зберігає адресу наступного вузла в tempPtr. Поле next вузла tempPtr визначає вузол у списку, на який повинен тепер вказувати поточний об’єкт. Повертається вказівник на вузол tempPtr. Процес вимагає присвоєння тільки одного вказівника.

```
// видалити вузол, наступний за поточним, і повернути його адресу
template <class T>
Node<T> *Node<T>::DeleteAfter(void)
{
// зберегти адресу вузла, що видаляється
Node <T> * tempPtr = next;
// якщо немає наступного вузла, повернути NULL
if (next == NULL)
      return NULL;
// поточний вузол вказує на вузол, наступний за tempPtr.
next = tempPtr-> next;
// повернути вказівник на непов'язаний вузол
return tempPtr;
}
```

Після розгляду методів виконання стандартних функцій перейдемо до прийомів формування зв’язаних списків. Конструювання списку можна провести з його голови або з хвоста, або інакше. Спочатку розглянемо метод створення вузла з застосуванням шаблону функції GetNode, яка приймає початкові дані-значення і вказівник динамічно створює новий вузол. Якщо виділення пам'яті відбувається невдало, програма завершується, інакше, функція формує показник на новий вузол.

```
// Виділення вузла з даними-членом item і вказівником nextPtr
template <class T>
Node<T> *GetNode(const T& item, Node<T> *nextPtr - NULL)
{
Node<T> *newNode;
// Виділення пам'яті при передачі item і NextPtr конструктору.
// Завершення програми, якщо виділення пам'яті виконалося невдало
newNode * new Node<T>(item, nextPtr);
if (newNode == NULL)
{
```

```
      cerr << "Помилка виділення пам'яті!" << endl;
      exit(1);
}
return newNode;
}
```

Операція вставки вузла в початок списку (InsertFront) вимагає оновлення значення вказівника голови, так як список повинен тепер мати новий початок. Проблема збереження голови списку є основною для керування списками. Якщо голову втрачено, то втрачено і список!

Перед початком вставки голова визначає початок списку і після вставки новий вузол займе положення на початку списку, а попередній початок списку займе другу позицію. Отже, полю покажчиків нового вузла присвоюється поточне значення голови, а голові присвоюється адресу нового вузла. Нижче ця переорієнтація виконується з застосуванням шаблону GetNode для створення нового вузла,

```
head = GetNode(item,head);
```

і продемонстрована на рис. 2.3.

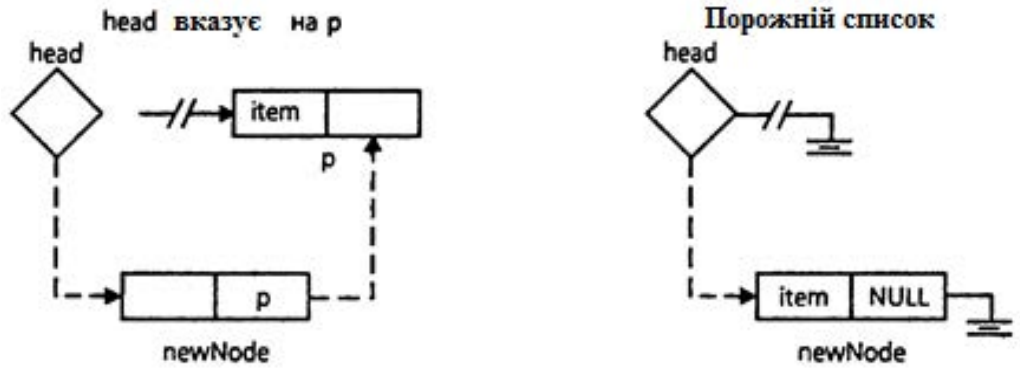


Рис. 2.12. Формування голови списку за шаблоном GetNode

Функція InsertFront приймає поточну голову списку, яка є вказівником, визначаючим список, а також приймає нове значення даних. Вона вставляє значення даних у вузол на початку списку. Так як голова буде змінюватися цією операцією, вона передається як контрольний параметр.

```
// Вставка елемента в початок списку
template <class T>
void InsertFront(Node<T>* & head, T item)
{
// створення нового вузла, щоб він вказував на первину голову списку
// зміна голови списку
head = GetNode (item, head);
} // Ця функція і GetNode знаходяться у файлі nodelib.h.
```

Початковою точкою будь-якого алгоритму проходження є вказівник голови, так як він визначає початок списку. При проходженні за списком, використовується вказівник currPtr для посилання на поточне положення в списку. Спочатку currPtr встановлюється на початок списку – currPtr = head;

При скануванні необхідно мати доступ до поля даних у поточному становищі. Так як data - це відкрите поле, є можливість виконати вибірку значення data або присвоїти цьому данному нове значення.

```
currentDataValue = currPtr->data;
currPtr->data= newdata;
```

Наприклад, простий оператор cout може бути включений в алгоритм проходження списку для друку значення кожного вузла:

```
cout << currPtr->data;
```

У процесі сканування безперервно переміщується вказівник currPtr до наступного вузла до тих пір, поки не досягнуто кінця списку.

```
currPtr = currPtr->nextNode();
```

Проходження за списком завершується, коли currPtr стає рівним NULL. Наприклад, функція PrintList (файл nodelib.h) друкує значення даних кожного вузла. Голова передається як параметр для визначення списку. Другий параметр користувацького типу AppendNewline вказує на те, чи повинні слідувати за висновком два пробіли або символ newline.

```
enum AppendNewline {noNewline,addNewline};
// Друкування зв'язаного списку
template <class T>
void PrintList(Node<T> *head, AppendNewline addnl = noNewline)
{
// CurrPtr пробігає за списком, починаючи з голови
Node <T> * currPtr = head;
// поки не кінець списку, друкувати значення даних поточного вузла
while(currPtr != NULL)
{
     if(addnl == addNewline)
          cout << currPtr->data << endl;
     else
          cout << currPtr->data << " ";
     currPtr = currPtr->NextNode(); // Перейти до наступного вузла
}
}
```

Приклад. Побудувати програму, яка генерує 10 випадкових чисел в діапазоні від 1 до 10 і вставляє ці значення як вузли в голову зв'язаного списку, використовуючи InsertFront. Для відображення списку використати функцію Printlist.

Програма містить код, який підраховує кількість входжень ключа до списку. У користувача спочатку запитується ключ, який при проходженні по списку порівнюється з полем даних у кожному вузлу списку та вводиться загальна кількість входжень цього ключа.

```
#include <iostream.h>
#pragma hdrstop
#include "node.h"
#include "nodelib.h"
#include "random.h"
```

```
void main(void)
{
// Встановити голову списку в NULL
Node <int> * head = NULL, * currPtr;
int i, key, count = 0;
RandomNumber rnd;
// Ввести 10-ть випадкових чисел в початок списку
for (i=0;i < 10;i++)
      InsertFront(head, int(1+rnd.Random(10)));
// Друкування вихідного списку
cout <<" Список: ";
PrintList (head, noNewline);
cout <<endl;
// Запросити введення ключа
cout << "Введіть ключ: ";
cin >> key;
// Цикл за списком
currPtr = head;
while (currPtr != NULL)
{
      // Якщо дані збігаються з ключем, збільшити count
      if (currPtr-> data - key)
      count + +;
      // Перейти до наступного вузла
      currPtr = currPtr->NextNode();
}
cout <<" Значення "<< key <<" з'являється "<< count
           << Раз (а) в цьому списку "<< endl;
}

<Виконання програми >
Список: 3 6 5 7 5 2 4 5 9 10
Введіть ключ: 5
Значення 5 з'являється 3 рази в цьому списку
```

На завершення лекції розглянемо додавання вузла в хвіст списку (InsertRear). Ця операція вимагає початкового тестування для визначення, чи порожній список. Якщо так, тоді створюється новий вузол з нульовим вказівником поля і присвоюється його адреса голові. Операція реалізується функцією InsertFront. При не порожньому списку необхідно сканувати вузли для виявлення хвостового вузла, у якому поле next містить значення NULL. currPtr->NextNode() == NULL;

Вставка виконується таким чином: спочатку створюється новий вузол (GetNode), потім він вставляється після поточного об'єкта Node (InsertAfter).

Так як вставка може змінювати значення вказівника голови, голова передається як контрольний параметр:

```
// Знайти хвіст списку і додати item
template <class T>
void InsertRear(Node<T>* & head, const T& item)
```

```
{     // InsertRear міститься у файлі nodelib.h.
      Node<T> *newNode, *currPtr = head;
      if (currPtr == NULL) // Якщо контактів немає, вставити item в
      початок
            InsertFront(head,item) ;
      else
      {
            while (currPtr-> NextNode ()! = NULL)
      // Знайти вузол з нульовим вказівником
                  currPtr =currPtr-> NextNode ();
            newNode = GetNode(item); // Створити вузол і вставити в
            кінець списку
            currPtr->InsertAfter(newNode); // InsertRear міститься у
            файлі nodelib.h.
      }
}
```

З огляду наведених матеріалів параграфу з'ясовується, що над зв'язаними списками і їх складовими можна виконувати операції вставки, видалення вузлів, конструювання нових списків або пере форматування старих тощо. Наведені програмні представлення алгоритмів операцій дозволяють здобувачеві навичок самостійно конструювати необхідні списки.

2.3. Особливі списки

Сучасні середовища програмування дозволяють конструювати досить різноманітні структури даних, зокрема одно-направлені та багато-направлені, одно та багато циклічні і інші спискові структури. Параграф знайомить здобувача знань з деякими структурами особливих списків, з їх засобами формування, обробки і технологічними операціями над такими списками.

Структура формування даних предметної області може бути лінійною, зірковою, мережевою та іншою. Характерним для таких структур є наявність двох елементних конструкцій (x, y) у послідовності формування. Перший елемент послідовності x є відповідне дане, а другий – y відтворює зв'язок з попереднім сформованим елементом, отже пара (x, y) є фрагментом списку. При безумовних парах (x, y) послідовності, формується одно-направлений список і при декількох умовних парах формується мережа зв'язаних даних. Так збудована структура даних може задавати автомобільні мережі з відстанями між населеними пунктами тощо.

Наведемо приклад класу формування одно-направленого списку.

```
class Element {
        int  data;
        Element  prev; // попередній елемент списку
```

```
        Element(int x, Element s) { data=x;  prev = s;} //
встановлюємо значення для елементів
}
class Stack {
     Element last;
    public void prin() {
     Element  current = last; // останній елемент робимо поточним
          if (last == null)
                System.out.println("Stack empty!"); // системне
повідомлення «Стек порожній»
          else
              while (current != null) {
             System.out.print(current.data +"\t"); // вивід на екран
поточних даних
             current = current.prev; // змінюємо елементи місцями
     }
   }
public boolean add(int data)  {
     last = new Element(data, last); // заносимо нове значення в last
        return true;
   }
public Integer delete() {
       int del;
       if (last != null) {
           del = last.data;
           last = last.prev;
     return del;
       }
       return null;
   }
}
```

Зв'язаний список, що закінчується NULL-символом – це послідовність вузлів, яка починається з головного вузла і закінчується вузлом, поле покажчика next яке має значення NULL. У лекції 5 наведена бібліотека функцій [1] для сканування такого списку і для вставки та видалення вузлів. У подальшому розробляється альтернативна модель, що зветься циклічним пов'язаним списком (circular linked list), яка спрощує розробку та кодування алгоритмів послідовних списків.

Порожній циклічний список містить вузол, який має неініціалізоване поле даних. Цей вузол називається заголовком (header) і спочатку вказує на самого себе. Роль заголовка - вказувати на перший реальний вузол у списку і, отже, на заголовок часто посилаються як на вузол (sentinel). У циклічній моделі зв'язаного списку порожній список фактично містить один вузол, і покажчик NULL ніколи не використовується. Ми наводимо схему (див. рис. 2.13.) заголовка, використовуючи кутові лінії в якості сторін вузла.

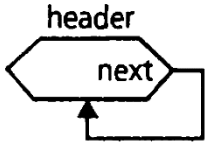


Рис 2.13. Голова порожнього списку

Зауважте, що для стандартного пов'язаного списку і циклічного зв'язаного списку тести, що визначають, чи є список порожнім, різні.

Так для стандартного зв'язаний списку: маємо (head == NULL) і для циклічного зв'язаного списку – (header-> next == header).

При додаванні вузлів до списку останній вузол вказує на заголовний вузол, як наведено на рис. 2.14.

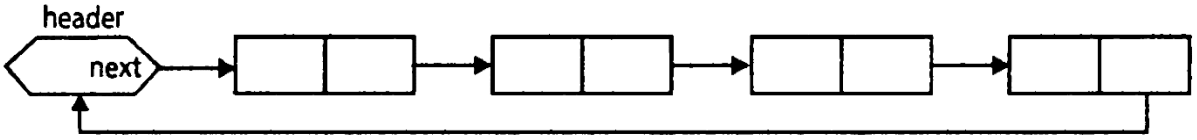


Рис. 2.14. Схема циклічного списку

У параграфі 2.1 був описаний клас Node і у параграфі 2.2 використовувалися його методи для створення зв'язаних списків. У цьомув параграфі оголошується клас Cnode, що створює вузли для циклічного списку. Цей клас створює конструктор за замовчуванням, що допускає неініціалізовані поля даних. Конструктор використовується для створення заголовка.

Специфікація класу CNode

Оголошення

```
template <class T>
class CNode
{
private:
// Циклічний зв'язок для наступного вузла
CNode <T> * next;
public:
T data; // Дані - відкриті
CNode (void); // Конструктор
CNode (const T & item);
// Методи модифікації списку
void InsertAfter (CNode <T> * p);
CNode<T> * DeleteAfter (void);
CNode<T> *NextNode(void) const; // Отримати адресу наступного вузла
};
```

Цей клас подібний до класу Node. У дійсності всі члени цього класу мають те ж ім'я і ті ж функції.

Конструктори ініціалізують вузол його вказівкою на самого себе, тому кожен вузол може служити в якості заголовка для порожнього списку. Покажчик на самого себе - це покажчик this, і, отже, присвоювання стає наступним:

next = this;

Для конструктора за замовчуванням поле data не ініціалізується. Інший конструктор приймає параметр і використовує його для ініціалізації поля data.

Ніякий конструктор не вимагає параметра, який би визначав початкове значення для поля next. Всі необхідні зміни поля next виконуються з використанням методів Insert After і Delete After.

```
// Конструктор, який створює порожній список та ініціалізує дані
template<class T>
CNode<T>::CNode(const T& item)
{
// Встановлює вузол для покажчика на самого себе і іціціалізує дані
next = this;
data = item;
}
```

Клас CNode надає метод NextNode, який використовується для проходження по списку. Подібно до методу класу Node функція NextNode повертає значення покажчика next.

Функція Insert After додає вузол p безпосередньо після поточного об'єкта. Для завантаження вузла в голову списку не потрібно ніякого спеціального алгоритму, так як просто виконується функція Insert After (у header). Демонстрація вставки наведена на рис. 2.15.

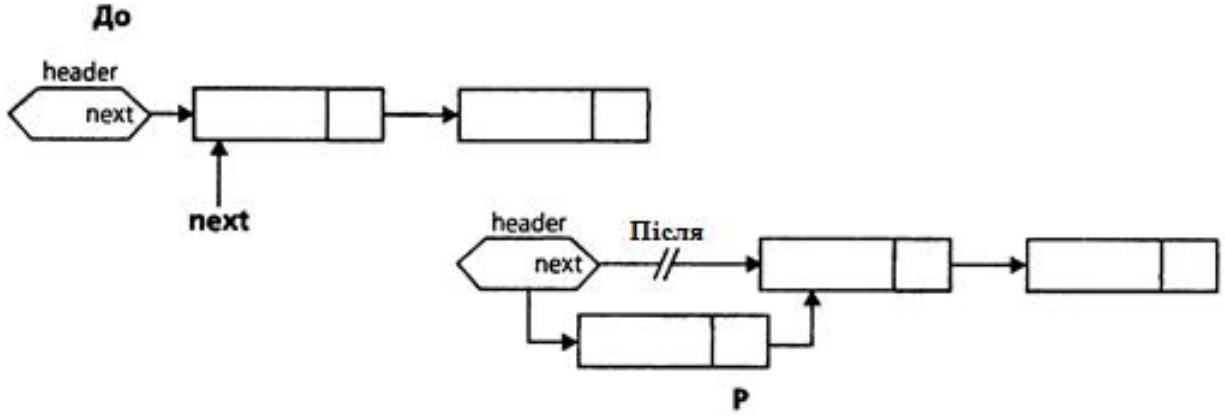


Рис. 2.15. Демонстрація вставки вузла через голову списку

```
// Вставка вузла p після поточного вузла
template <class T>
void CNode <T> :: InsertAfter (CNode <T> * p)
{
//P вказує на наступний вузол за поточним. Поточний вузол вказує на p.
     p->next = next;
     next = p;
}
```

Видалення вузла зі списку виконується методом DeleteAfter. DeleteAfter видаляє вузол, який слідує безпосередньо за поточним вузлом, і потім повертає покажчик на віддалений вузол. Якщо next має єдине this, тоді у списку немає ніяких інших вузлів, і вузол не повинен видаляти самого себе. У цьому випадку операція повертає значення NULL.

```
// Видалити вузол, наступний за поточним і повернути його адресу
template <class T>
CNode<T> *CNode<T>::DeleteAfter(void)
{
CNode <T> * tempPtr = next; // Зберегти адресу видаляється вузла
// Якщо в next адреса поточного об'єкта (this) / він
// вказує сам на себе, повернути NULL
if (next ==this)
     return NULL;
// Поточний вузол вказує на наступний за tempPtr.
next = tempPtr-> next;
return tempPtr; // Повернути вказівник на непов'язаний вузол
}
```

Сканування списку, що закінчується NULL-символом відбувається зліва направо. Циклічний список є більш гнучким і дозволяє починати сканування з будь-якого положенні в списку і продовжувати його до початкової позиції. Ці списки мають обмеження, так як вони не дозволяють користувачеві повертати пройдені кроки і виконувати сканування у зворотному напрямку (див. рис. 2.16.). Вони неефективно виконують просту задачу видалення вузла p, так як необхідно проходити за списком і знаходити покажчик на вузол, що передує p.

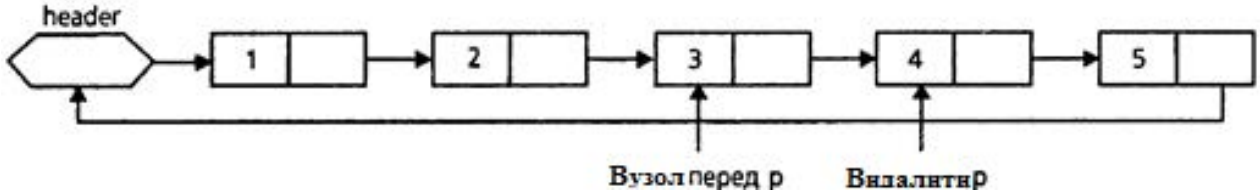


Рис. 2.16. Схема проходження списку для видалення вузла p

У випадках, коли виникає необхідно звертатися до вузлів в будь-якому напрямку, корисним є *двох-зв'язаний* список (doubly linked list),котрий зображено на рис. 2.17. Для створення потужної і гнучкої структури обробки двох-зв'язаного списку, вузли списку містить по два покажчики.

Рис. 2.17. Вузол двозв'язаного списку

Двох-зв'язані списки відносяться до нелінійних списків. Для двох-зв'язаного списку, операції вставки і видалення є в кожному напрямку. Рис.2.18 ілюструє проблему вставки вузла праворуч від поточного. При цьому необхідно встановити чотири нових зв'язки.

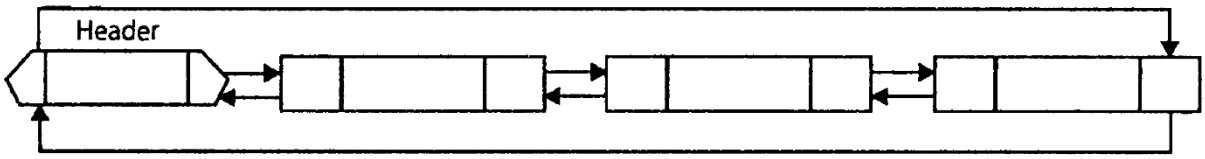


Рис. 2.18. Схема двозв’заного списку

У двох зв'язному списку вузол може видалити сам себе зі списку, змінюючи два покажчика. На наступному рис. 2.19 показані відповідні процедурні зміни при видалені і вставці вузла.:

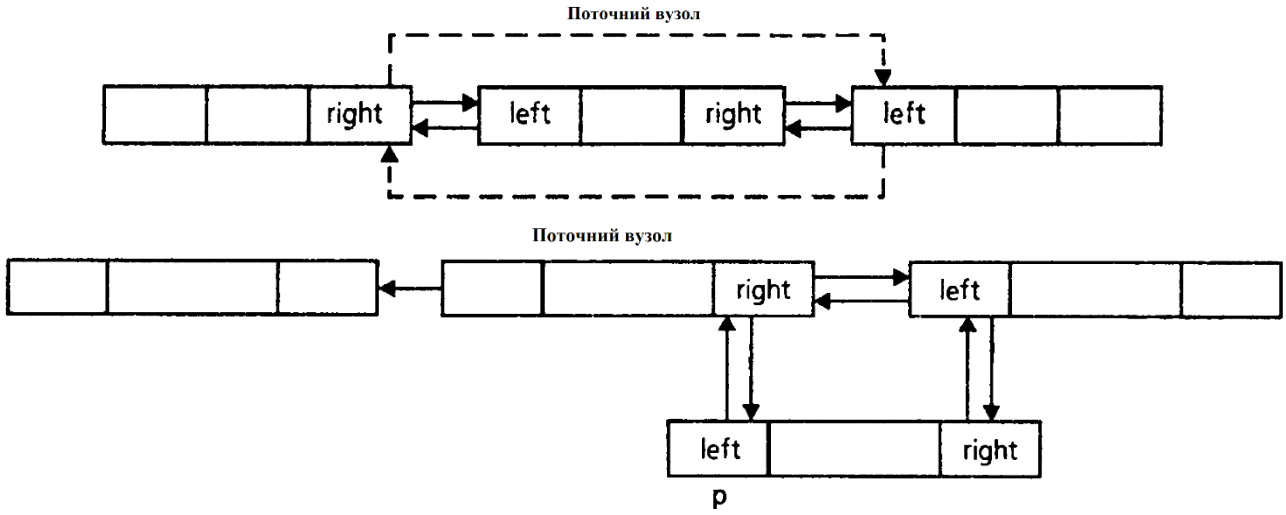


Рис. 2.19. Схеми видалення і вставки вузла

Специфікація класу DNode

Оголошення

```
template <class T>
class DNode
{
private:
// Циклічні зв'язки вліво і вправо
DNode<T> *left;
DNode<T> *right;
public:
// Дані - відкриті
T data;
// Конструктори
DNode (void);
DNode (const T & item);
// Методи модифікації списку
void InsertRight(DNode <T> * p);
void InsertLeft(DNode <T> * p);
DNode <T> * DeleteNode (void);
// Отримання адреси наступного (вправо і вліво) вузла
DNode<T> *NextNodeRight(void) const;
```

```
DNode<T> *NextNodeLeft(void) const;
};
```

У представленому класі дані-члени подібні членам одно-зв’язаного класу CNode, за винятком того, що тут використовуються два покажчика next. І тому застосовуються дві операції вставки (по одній для кожного напрямку) і операція видалення, яка видаляє поточний вузол зі списку. Значення закритого покажчика повертається за допомогою функцій NextNodeRight і NextNodeLeft.

На завершення розглянемо приклад формування двох-зв’язного списку.

```
DNode <int> dlist;              // двох-зв’язний список
// Сканувати, значення вузла, поки не повернемося до голови списку
DNode <int> * p = &dlist;       // ініціалізація покажчика
p = p-> NextNodeRight();        // установка p на перший вузол у
списку
while (p! = Slist)
{
     cout <<p-> data;           // вивід значення даних
     p=p-> NextNodeRight (); // установка p на наступний вузол у
списку
}
DNode <int> * newNodel (10);    // створення вузлів зі значеннями
DNode <int> * newNode2 (20);    // 10 і 20
DNode <int> * p = sdlist;       // p вказує на заголовний вузол
p-> InsertRight (newNodel);          // вставка в початок списку
p-> InsertLeft (newNode2);           // вставка у хвіст списку.
```

У запропонованому параграфі наведені матеріали, які дозволяють ознайомитися з технологічними можливостями формування одно-направлених, циклічних і двох-зв’язаних списків.

Розглянуто операції, які можна виконувати над такими списками. Для детального оволодіння прийомами створення спискових структур наводяться фрагменти програм.

РОЗДІЛ 3. МЕТОДИ СОРТУВАННЯ ДАНИХ

Як правило, для прискорення обробки множених структур даних їх спочатку сортують. Основні методи сортування масивів, списків і оцінка складності алгоритмів сортування, наведені у матеріалах двох параграфів розділу: *методи сортування даних у масивах, методи вставки, швидкого та порозрядного сортування,*

3.1. Методи сортування даних у масивах

У попередніх матеріалах розглядалися множинні структури даних, які можуть бути великими за розмірами і тому пошук в них відповідних компонент може бути тривалим. Для прискорення пошуку у множинних об'єктах, як правило, застосовують спочатку *сортування* (упорядкування) елементів. Існує велика кількість алгоритмів сортування, також як і алгоритмів пошуку, котрі описані у літературних джерелах [1 – 5, 8]. Нижче надається знайомство з поширеними алгоритмами вибору і бульбашки.

Почнемо знайомство з класичним алгоритмом *вибірки* на масивах даних. Хоча цей алгоритм на практиці не надто ефективний для великої кількості елементів, але він ілюструє основні підходи до сортування даних у масивах.

Нехай задано набір числових елементів $A = (a_1, a_2, a_3, \dots, a_n)$, необхідно впорядкувати за відношенням ρ =(>, <) ці дані. Сортування виконується за декількома проходами масиву даних зліва на право, з права на ліво або комбінованим методом.

Схема алгоритму прохід зліва на право наступна, спочатку присвоюється $\boldsymbol{i = 0}$:

1) розглядається відношення $a_i \rho a_j$, $i < j$;

2) виконується порівняння

$$a_i \rho a_j \rightarrow \begin{cases} true, j = j + 1; \\ false, a = a_i, a_i = a_j, a_j = a, j = j + 1. \end{cases}$$

3) змінюється значення індексу$i = i + 1$.

У другому пункті схеми при значенні false відбувається обмін даними.

Прохід з права на ліво починається з кінця масиву, $\boldsymbol{i = n}$ при умові $i > j$, і $i = i - 1$.

Для алгоритму вибірки передбачається, що n елементів даних зберігаються в масиві A і на ньому виконується $n - 1$ проходів. У нульовому проході зліва на право вибирається найменші елементи з двох можливих (сортування за зростанням значень), які потім міняються місцями з $A[0]$. Після цього неупорядкованими залишаються елементи $A[1]$ ... $A[n-1]$. У наступному проході проглядається неупорядкована хвостова частина масиву, звідки вибирається найменший елемент і запам'ятовується в $A[1]$. У наступному

проході проводиться пошук найменшого елемента у під масивові $(A[2], A[3], \dots, A[n-1])$. Знайдене мінімальне значення міняється місцями з $A[2]$. Таким чином виконується $n-1$ прохід, після чого невпорядкований хвіст масиву скорочується до одного елементу, який і є найбільшим.

Для прикладу розглянемо детально схематичний процес сортування масиву за зростанням методом вибірки з ліва на право, якщо масив містить п'ять чисел: 50, 20, 40, 75 і 35.

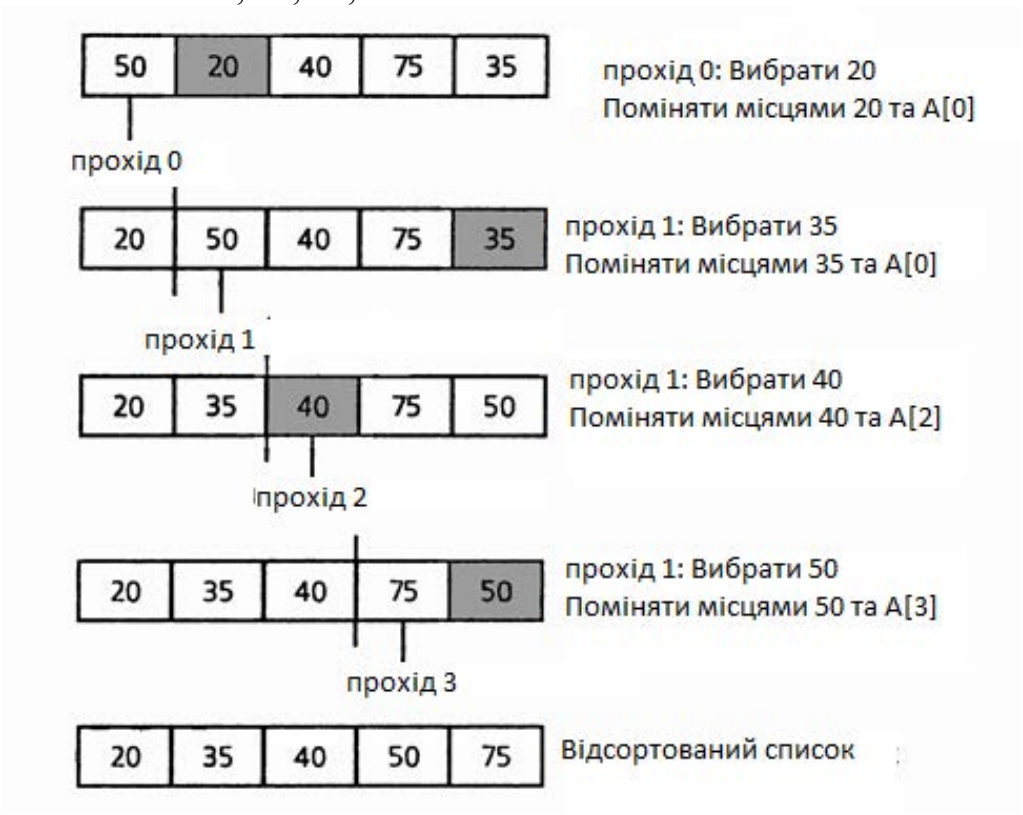


Рис. 3.1 Сортування за зростанням

Нижче наводиться представлення алгоритму сортування даних масиву A, котрий після сортування зостається на «місці». У i -му проході сканується частина масиву $A[i], \dots, A[n-1]$ і змінній smalllndex присвоюється індекс найменшого елементу в цьому масивові. Потім елементи $A[i]$ і A [smalllndex] міняються місцями. Функція SelectionSort і утиліта обміну даними Swap знаходяться у файлі arrsort.h.

```
// Впорядкування n-елементного масиву типу T,
// за допомогою алгоритму вибірки
template<class T>
void SelectionSort(T A[], int n)
{
// індекс найменшого елемента на кожному проході
int smallIndex;
int i, j;
for (i = 0; i<n-l; i++)
```

```
{
     // почати прохід з індексу i; встановити smallIndex в i
     smallIndex = i;
     for (j=i+l; j<n; j++)
     // оновити smalllndex, якщо знайдений менший елемент
          if (A[j] < A[smallIndex])
               smallIndex = j;
     //по закінченню помістити найменший елемент в A [i]
     Swap(A[i], A[smallIndex]);
     }
}
```

Порахуємо тепер складність алгоритму сортування за представленим методом. Сортування за допомогою методу вибірки вимагає фіксованого числа порівнянь і залежить тільки від розміру масиву, а не від початкового розподілу в ньому даних.

У і-му проході число порівнянь з елементами A [i +1] ... A [n-1] дорівнює

$$(\mathrm{n}-1)-(\mathrm{i}+1)+1=\mathrm{n}-\mathrm{i}-1$$

Загальна кількість порівнянь становить

$$\sum_{i=0}^{n-2}\big((n-1)-i\big)=(n-1)^2-\sum_{i=0}^{n-2}i=(n-1)^2-\frac{(n-1)(n-2)}{2}=\frac{n(n-3)}{2}+1$$

.

Отже складність алгоритму, що вимірюється числом порівнянь, дорівнює порядку $O(n^2)$, а число обмінів дорівнює $O(n)$. Найкращого і найгіршого випадків не існує, так як алгоритм робить фіксоване число проходів, в кожному з яких сканується певна кількість елементів.

Обговоримо тепер бульбашкове сортування за зростанням елементів, при якому також у кожному проході виконується ряд обмінів. Для сортування n елементного масиву A методом бульбашки потрібно до n-1 проходів. У кожному проході порівнюються сусідні елементи, і якщо перший з них більше другого, ці елементи міняються місцями. До моменту закінчення кожного проходу найбільший елемент піднімається до вершини поточного підмасиву, подібно пухирцю повітря в киплячій воді. Наприклад, по закінченні нульового проходу, хвіст масиву (A[n-1]) відсортований, а його головна частина залишається невпорядкованою.

Розглянемо ці проходи докладніше. Для збереження останнього індексу, який приймає участь у обміні даних введемо змінну lastExchangelndex. На початку кожного проходу елементів масиву, змінній lastExchangelndex присвоюється значення нуль. У нульовому проході порівнюються сусідні елементи (A[0], A[1]), (A[1], A[2]), ..., (A[n-2], A[n-1]). В кожній парі (A[i], A[i +1]) елементи міняються місцями за умови, що A[i +1] <A[i], а значення lastExchangeIndex стає рівним i. В кінці цього проходу найбільше значення виявляється в елементі A[n-1], а значення lastExchangelndex показує, що всі елементи в хвостовій частині списку від A[lastExchangeIndex + l] до A[n-1] відсортовані. У черговому проході порівнюються сусідні елементи в підмасиві

A[0] – A[lastExchangeIndex]. Процес сортування за методом бульбашки припиняється при lastExchangeIndex = 0. Отже алгоритм здійснює максимум n-1 прохід.

Проілюструємо за У. Топом і У. Фордом [1] процес бульбашкового сортування по зростанню на масивові, що містить п'ять попередніх неупорядкованих цілих чисел: 50, 20, 40, 75 і 35.

Рис. 3.2. Сортування бульбашками, прохід 0

Оскільки lastExchangeIndex не дорівнює нулю, процес сортування триває. У проході 1 сканується масив даних від A[0] до A[3] = A[lastExchangeIndex]. Новим значенням lastExchangeIndex стає 2.

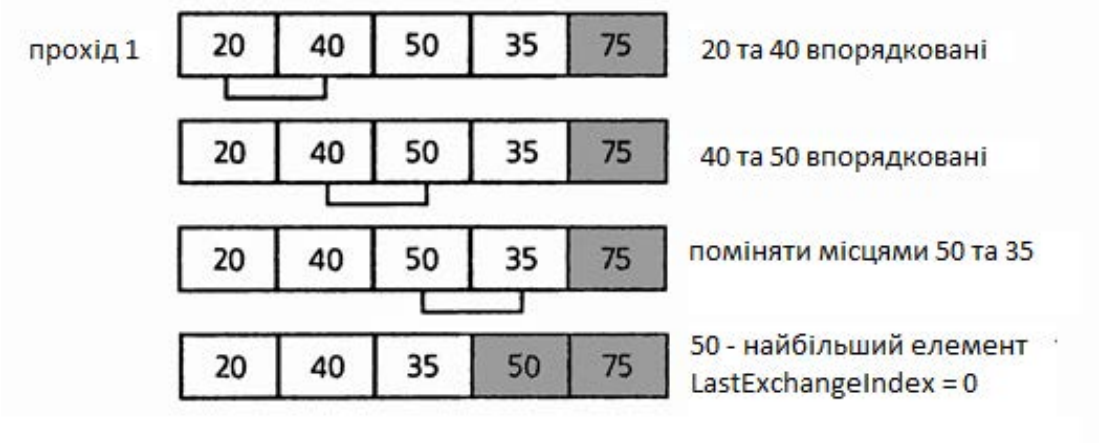


Рис. 3.3. Сортування бульбашками, прохід 1

При проході 2 сканується масив даних від A [0] до A [2] і елементи 40 і 35 міняються місцями. lastExchangeIndex стає рівним 1.

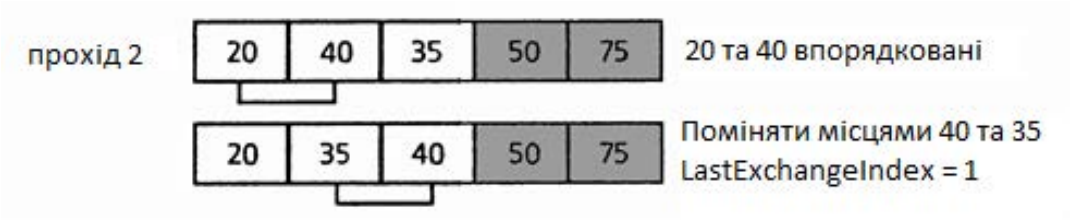


Рис. 3.4. Сортування бульбашками, прохід 2

У проході 3 виконується єдине порівняння (20 і 35). Так як при цьому обмінів не здійснюється, тому змінна lastExchangelndex = 0, і процес сортування припиняється.

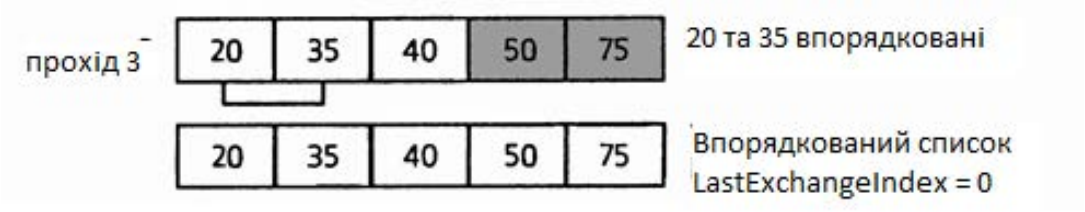


Рис. 3.5. Сортування бульбашками, прохід 3

Наведемо текст представлення бульбашкового алгоритму на мові С++.

```
// BubbleSort приймає масив A і його розмір n.
// Сортування виконується за допомогою ланцюга проходів, поки
// змінна lastExchangelndex = 0.
template<class T>
void BubbleSort (T A[], int n)
{
int i=n-1, j;
// Індекс останнього елемента, що брав участь в обміні
int lastExchangelndex;
// i – індекс останнього елемента в підсписку масиву є i = n-1;
// продовжувати процес, поки не буде зроблено жодного обміну
while (i > 0)
{
      // Початкове значення змінної
      lastExchangelndex = 0;
      // Сканувати підсписок A [0] - A [i]
      for (j=0; j < i; j++)
      // міняти місцями сусідні елементи і оновити lastExchangelndex
            If(A[j+l] < A[j])
            {
                  Swap(A[j], A[j+1]);
                  lastExchangelndex = j;
            }
      // присвоїти і значення індексу останнього обміну
      i = lastExchangelndex;
}
}
```

Оцінка ефективності впорядкування даних за методом бульбашки може бути виконана для гіршого і кращого початкових випадків даних. Якщо масив даних з n елементів на початку сортування упорядкований за зростанням, тоді достатньо одного проходу з n-1 порівняннями і складність алгоритму буде мати порядок O(n). Найгірший випадок для бульбашкового сортування виникає коли початковий масив даних впорядкований за збуванням. Тоді необхідно виконувати всі n-1 проходів. І на і-му проході виконується (n - i - 1) порівнянь та (n - i - 1) обмінів. Складність у найгіршому випадку становить $O(n^2)$ порівнянь і $O(n^2)$ обмінів. У загальному випадку сортування за допомогою методу вибірки перевершує бульбашкове впорядкування за рахунок меншої кількості обмінів.

Існує декілька модифікацій сортування методом бульбашки. Суть однієї з них полягає в тому, що напрями проходів масиву чергуються. При проходженні масиву у «прямому» напрямку його найбільший елемент заноситься у кінець масиву, а при проходженні у «зворотному» напрямку – найменший елемент заноситься у початок масиву. Цей алгоритм носить назву *шейкер-сортування* і є досить ефективним для задач відновлення впорядкованості, коли початкова послідовність вже була впорядкована, але піддалася не дуже значним змінам. Наприклад, впорядкованість масиву з одним початковим порушенням порядку відновлюється за два проходи.

Розглянемо ще одна модифікація методу сортування бульбашкою. Базовий метод бульбашки виконує порівняння елементів, відстань між якими дорівнює одиниці. Але порівняння можна виконувати між елементами, які знаходяться один від одного на деякій відстань *h*. Початковий розмір h звичайно вибирається рівним половині загального розміру масиву даних сортування. Далі виконується сортування бульбашкою з інтервалом порівняння h. Потім значення *h* зменшується удвічі і знов виконується сортування бульбашкою, після чого *h* зменшується ще удвічі і т.д. Останнє сортування бульбашкою виконується при *h=1*. Розглянутий прийом сортування зветься *методом Шелла*.

Порядок складності алгоритму сортування за методом Шелла залишається незмінним $O(n^2)$, середнє ж кількість порівнянь, визначається емпірично шляхом, – $n \log n^2$. Прискорення досягається за рахунок того, що виявленні «не на місцях» елементи при *h>1*, швидше «спливають на свої місця». Наступний приклад програми ілюструє сортування за методом Шелла.

```
template<class T>
void ShellSort(T a[], int n)
{
  int h, i; // n – розмір масиву даних
  bool k;               // ознака перестановки
  h = n / 2;      // початкове значення інтервалу
       // цикл із зменшенням інтервалу до 1
  while (h>0)
  {
```

```
        // сортування бульбашкою з інтервалом d
        k = true;
        // цикл, поки є перестановки
        while (k)
        {
              k = false;
              // порівняння елементів на інтервалі d
              for ( i=0; i<n-h; i++ )
                    if ( a[i] > a[i+h] )
                    {
                          Swap(a[i], a[i+h]);        // обмін даними
                          // оновлення ознаки
                          k = true;
                    };
        };
        // зменшення інтервалу у двічі
        h = h / 2;
  };
}
```

Розглянуто ознайомчі питання з простих методів сортування даних на масивах, таких як алгоритми вибірки у модифікаціях та метод бульбашки і його модифікації метод Шелла і шейкер-сортування. Наведені оцінки ефективності цих алгоритмів за показниками дій. Показники складності представлених алгоритмів загалом мають однакову складність, але модифіковані алгоритми мають деякі переваги по кількості обмінів (метод Шелла), або при частково упорядкованих списках даних (шейкер-сортування).

3.2. Методи вставки, швидкого та порозрядного сортування

В матеріалах цього параграфу продовжується знайомство з методами сортування даних, зокрема розглядається методи сортування вставками, порозрядне та швидке сортування. Метою матеріалів є ознайомлення читача з методикою впорядкування даних за допомогою алгоритмів вставки і швидкого сортування з використанням рекурсивної процедури та порозрядного числового або лінгвістичного сортування.

Алгоритм сортування вставками нагадує процес тасування карток з іменами. Реєстратор заносить кожне ім'я на картку, а потім впорядковує картки за алфавітом, вставляючи картку у підходяще місце карткової стопи.

Продемонструємо алгоритм вставки на комірках даних однострічкового автомату. Нехай комірки стрічка містять масив чисел A = (50, 20, 40, 75, 35).

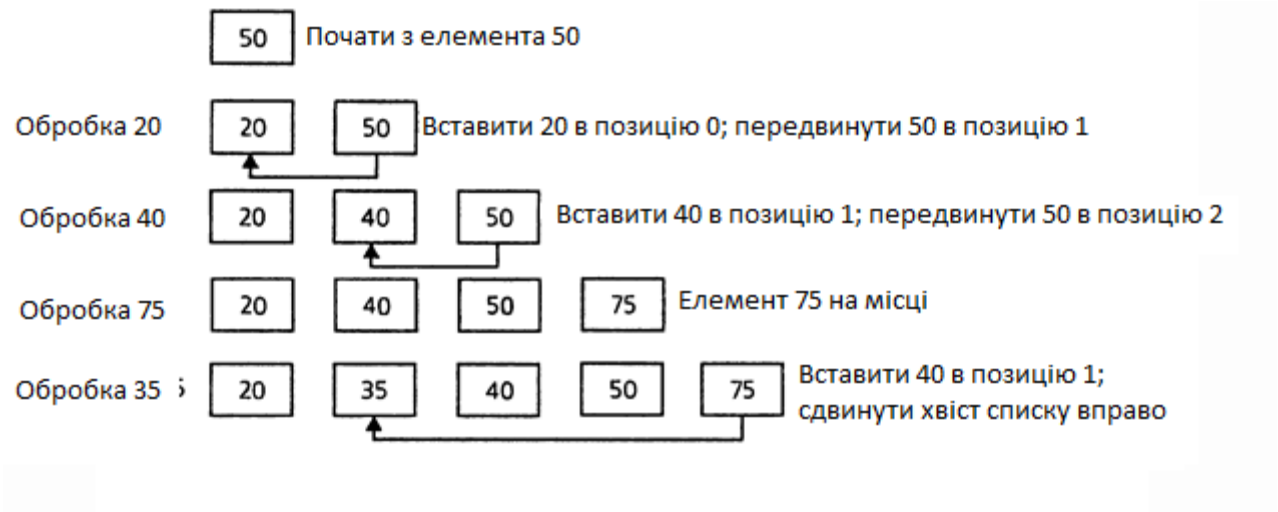


Рис. 3.6. Сортування вставками

Програма, яка представляє алгоритм вставки передбачає, що у функцію InsertionSort передається масив A і його довжина n.

За алгоритмом, припускається, що масив A[0], … A[i-1] вже відсортований по зростанню. Тоді на i-му проході (l <i: <n-l), змінний елемент temp, котрий вставляється у відповідне місце масиву, вибирається дане A[i] і далі змінна просувається до початку масиву, порівнюючись з елементами A[i-1], A[i-2] і т.д. Огляд елементів масиву завершується на елементі A[j], який менше або дорівнює temp, або знаходиться на початку масиву (j = 0). При просуванні змінної до початку списку кожен елемент зсувається вправо (A[j] = A[j-1]). Коли ж відповідне місце для A[i] буде знайдено, цей елемент вставляється на j місце масиву A.

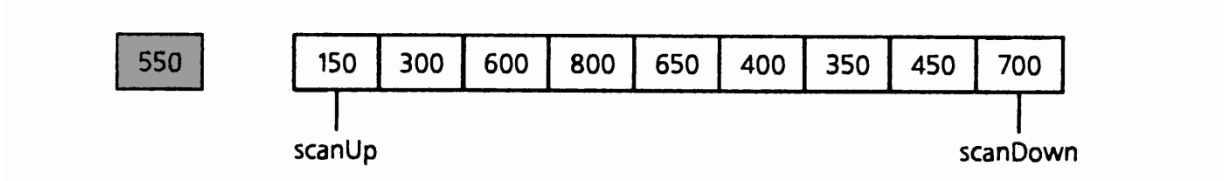
```
// Сортування вставками впорядковує масив даних A[0] ... A[i],
// 1 < i <= n-1. Для кожного i A[i] вставляється в підходящу позицію
A[j]
   template<class T>
   void InsertionSort(T A[], int n)
   {
      int i, j;
      T temp;
      // «i» визначає масив A[0] ... A[i]
      for (i = 1; i < n; i++) {
            // Індекс j пробігає масив вліво від A [i], розшукуючи
            // правильну позицію для значення,
            j = i;
            temp = A[i];
            // виявити відповідну позицію для вставки, скануючи частину
масиву,
            // поки temp <A [j-1] або поки не досягнуто початку списку
            while (j > 0 && temp < A[j - 1]) {
                  // зсунути елементи вправо, щоб звільнити місце для
вставки
                  A[j] = A[j - 1];
```

```
            j--;
        }
        // точка вставки знайдена; вставити temp
        A[j] = temp;
    }
}
```

Складність алгоритму сортування за методом вставки визначається тільки вставками на n-1 проходах (обміни не використовуються). Так як у і-му проході вставки формується послідовність A[0] … A[i] і застосовується в середньому i/2 порівнянь, тому загальна кількість порівнянь становить 1/2 + 2/2 + 3/2 + ... + (n-l)/2 = n (n-l)/4. Отже складність алгоритму вставки дорівнює $O(n^2)$. Найменшу складність маємо у випадку відсортованого за зростанням масиву. Тоді в і-му проході вставка відбувається в точці A [i], а загальне число порівнянь становить n-1, тобто складність становить O(n).

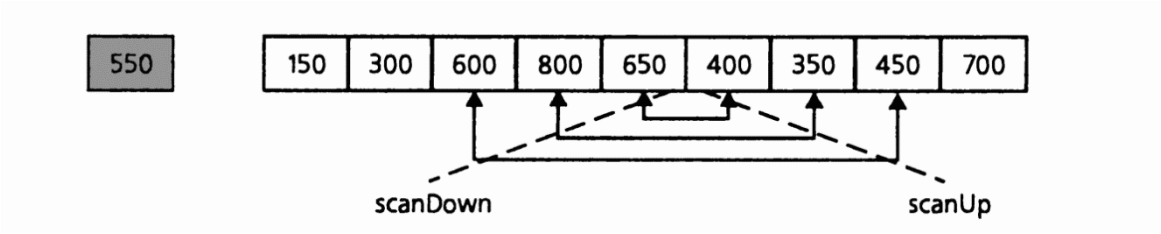
Розглянуті у цій і попередай лекціях методи сортування мають складність $O(n^2)$, але існує метод *швидкого сортування*, який є найшвидшим серед усіх існуючих методів. Ідея методу полягає в розбитті групи даних по відношенню до деякого її середнього значення mid на підгрупи, за правилом: менші елементи групи порівняно з mid утворюють одну підгрупу, а більші – другу.

Оригінальність методу полягає у взаємодії індексів масиву даних Індекс scanUp зростаючий і індекс scanDown збуваючий при проходженні масиву. Нехай дано масив з 10 чисел A = (800, 150, 300, 600, 550, 650, 400, 350, 450, 700). Його середина припадає на індекс mid = 4 з центральним елементом A[mid] = 550.



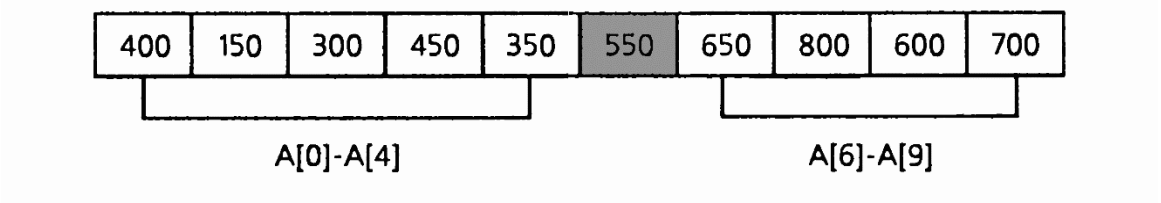
Таким чином, є можливість всі елементи масиву A розбити на дві групи. Просуваючи індекс scanUp вперед і шукаючи елемент A[scanUp] більший, ніж A[mid]. У цьому місці сканування зупиняється, і алгоритм готовий до переміщення знайденого елементу у верхню групу. Але перед тим як це переміщення відбудеться, робиться просування індексу scanDown вниз масиву і знаходиться елемент, менший або рівний центральному. Таким чином, є два елементи, які знаходяться не в тих групах. Обміняти їх місцями можна утилітою Swap (AfscanUp), A[scanDown]).

Цей процес продовжується до тих пір, поки scanUp і scanDown не зайдуть один за одного (scanUp = 6, scanDown = 5). У цей момент scanDown виявляється в підсписки, елементи якого менше або дорівнюють центральному. Ми потрапили в точку розбиття двох списків і вказали остаточну позицію для центрального елемента. У нашому прикладі поміняліся місцями числа 600 і 450, 800 і 350, 650 і 400.



Потім відбувається обмін значеннями елемента A [0] з A [scanDown], Swap (A[0], A[scanDown]). Отже в результаті вийшла група даних A[0] … A[4], елементи якої менші елементів групи A[6] ... A[9]. Центральний елемент (550) розділяє ці групи.

Обидві групи обробляються по попередньому алгоритму рекурсивно.



Отже в результаті рекурсивного впорядкування отримає масив даних A = (50, 300, 350, 400, 450, 550, 600, 650, 700, 800).

Представимо рекурсивний алгоритм швидкого сортування Quicksort у наступному вигляді.

```
// Quicksort приймає в якості параметрів масив та межі його індексів
template<class T>
void Quicksort(T A[], int low, int high) {
     // локальні змінні, що містять індекс середини mid,
     // центральний елемент та індекси сканування
     T pivot;
     int scanUp, scanDown;
     int mid;
     // якщо діапазон індексів не включає в себе як мінімум два елементи,
     // повернутися
     if (high - low <= 0)
          return;
     else
          // Якщо в частці масиву два елементи, порівняти між собою
          // і поміняти місцями при необхідності
          if (high - low == 1) {
               if (A[high] < A[low])
                    Swap(A[low], A[high]);
               return;
          }
     // Отримати індекс середини і присвоїти визначене їм значення
     // центральному значенню
     mid = (low + high) / 2; pivot = A[mid];
     // Поміняти місцями центральний і початковий елементи масиву
     // і ініціалізувати індекси scanUp та scanDown
     Swap(A[mid], A[low]); scanUp = low + 1; scanDown = high;
     // Шукати елементи, розташовані не в тих підсписках.
```

```
      // Зупинитися при scanDown <scanUp
      do {
            // Просуватися вгору по нижній частині масиву; зупинитися,
            // коли scanUp вкаже на верхній частині, або якщо
            // вказаний цим індексом елемент більше центрального
            while (scanUp <= scanDown && A[scanUp] <= pivot)
                  scanUp++;
            // Просуватися вниз по верхньому частині масиву; зупинитися,
            // якщо scanDown вкаже на елемент <= центрального.
            // зупинка на елементі A[low] гарантується
            while (pivot < A[scanDown])
                  scanDown--;
            // Якщо індекси все ще у своїх підсписках, то вони вказують
на
            // два елементи, що знаходяться не в тих підсписках. поміняти
їх місцями
            if (scanUp < scanDown) {Swap(A[scanUp], A[scanDown]);}}
      while (scanUp < scanDown);
      // Копіювати центральний елемент в точку розбиття
      A[low] = A[scanDown]; A[scanDown] = pivot;
      // Якщо нижня частина(low ... scanDown - 1) має два або більше
      // елементів, виконати рекурсивний виклик
      if (low < scanDown - 1)
            Quicksort(A, low, scanDown - 1);
      // якщо верхня частина (scanDown+1 ... high) має два або більше
елементів
      // виконати рекурсивний виклик
      if (scanDown + 1 < high)
            Quicksort(A, scanDown + 1, high);
}
```

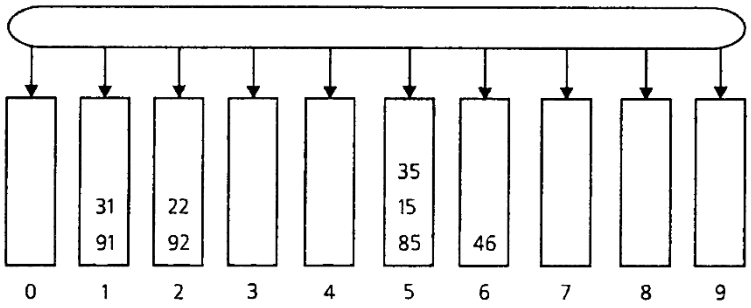
Аналіз ефективності "швидкого сортування" досить важкий, тому наведемо його результат $O(n\,log_2\,n)$. Цей показник складності менший ніж – $O(n^2)$ для методів вибірки, бульбашки, вставки і їх модифікацій.

На завершення розглянемо ще один досить простий метод порозрядного сортування. Суть, якого полягає у розбитті однорідних даних на сортувальні групи по кількості елементів алфавіту, на якому утворенні ці дані. Метод пебредбачає у якості груп застосовувати черги і масив для даних.

Наведемо приклад схемі алгоритму порозрядного сортування двох-розрядних чисел [1].

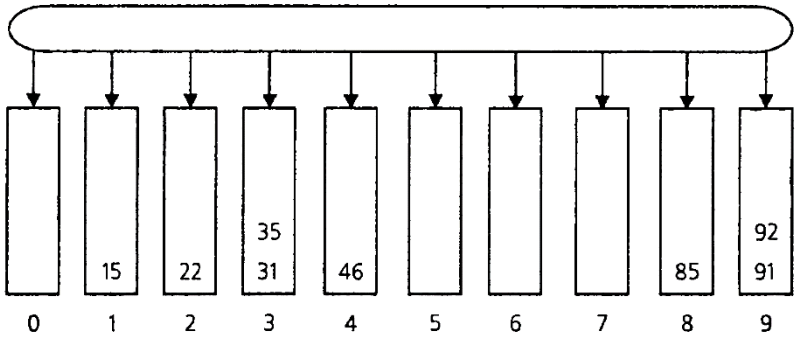
Нехай задана неупорядкована множина чисел M = {91, 46, 85, 15, 92, 35, 31, 22}

Кількість груп для організації сортування дорівнює десяти за алфавітом {0, 1, …, 9} і групи впорядковані. Виходячи з того, що множина утворена двох-розрядними числами, тому для її сортування достатньо два проходи множини даних У першому проході групи розподіляються по позиціям числових одиниць.



Дані з груп вибираються у порядку від 0 до 9. Отже після першого проходу відбулося сортування по молодих розрядах 91 31 92 22 85 15 35 46.

У другому проході дані першого проходу розподіляються у десяти групах по позиціям десятків.



Як і при першому проході вибираються дані із груп в порядку від 0 до 9, так, що множина M стає упорядкованою {15, 22, 31, 35, 46, 85, 91, 92}

Слід зауважити, що в якості груп при такому сортуванні зручно застосовувати черги, перший елемент якої є виходом і останній – входом, тобто елементи черг розташовуються в порядку FIFO.

Зрозуміло, що складність алгоритму для цього прикладу, становить $O(2n)$. У загальному випадку складність порозрядного сортування є $O(kn)$, $n = dim\, M$, k максимальна розрядність чисел множини M.

Наведений метод сортування є простим і може бути застосований для впорядкування лінгвістичних даних словарів, ключів та іншого.

Досі були розглянуті метоли сортування, які реалізовувалися на масивах, але ці прийоми впорядкування даних можна перенести і на списки або інші структури. Нижче розглянута програма сортування на лінійних списках.

Програма приймає як параметри масив елементів та його розмір і сортує список, використовуючи шаблонну функцію LinkSort. Отриманий упорядкований масив, виводиться за допомогою функції PrintArray.

```
#include <iostream.h>
#pragma hdrstop
#include "node.h"
#include "nodelib.h"
template <class T>
void LinkSort(T a[], int n)
{
Node<T> *ordlist = NULL, *currPtr;
```

```
int i;
// Вставляти елементи з масиву в список з упорядкуванням
for (i=0; i< n; i++)
      InsertOrder(ordlist, a[i]);
currPtr = ordlist; // Сканувати список і копіювати дані в масив
i = 0;
while(currPtr != NULL)
{
      a[i++] = currPtr->data;
      currPtr = currPtr->NextNode();
}
ClearList(ordlist); // Видалити всі вузли, створені в списку
}
void PrintArray(int a[], int n) // Сканувати масив і друкувати його
елементи
{
for(int i=0; i < n; i++)
      cout << a[i] << " ";
}
void main(void)
{
      int A [10] = {82,65,74,95,60,28,5,3,33,55}; // ініціалізація
      масиву даних
      LinkSort (A, 10);     // сортувати масив
      cout <<" Відсортований масив: ";
      PrintArray (A, 10);   // друкування масиву
      cout <<endl;
}
/*
<Виконання програми>
Відсортований масив: 3 5 28 33 55 60 65 74 82 95
*/
```

Наведені матеріали знайомлять читача з трьома методами сортування даних, які є різними за складністю реалізації. Найпростішим з них є метод порозрядного сортування, в якому використовуються дії з чергами та найшвидшим є метод швидкого сортування. У цих методах не застосовується функція обміну даних, що значно прискорює обробку даних. Але метод порозрядного сортування при великих розмірах алфавітів вимагає багато пам’яті.

РОЗДІЛ 4. АЛГОРИТМИ ПОШУКУ ДАНИХ У ВНУТРІШНІХ І ЗОВНІШНІХ СТРУКТУРАХ

Розділ присвячений питанням пошуку даних у множинних структурах і хеш-таблицях, вирішенню виникаючих колізій, а також обробки даних на зовнішніх носіях. Методика пошуку даних викладена у трьох параграфах розділу: *методи пошуку даних і хешування, вирішення колізій та клас хеш-таблиць, зовнішні носії і файлові операції*

4.1. Методи пошуку даних і хешування

Після розгляду алгоритмів упорядкування даних настала черга познайомитися з алгоритмами пошуку інформації. Метою послідуючого викладення матеріалу є знайомство з методами пошуку даних як на простих, так і складних відповідних структурах, навчити читачів застосовувати прийоми оцінки складності алгоритмів пошуку та ознайомити їх з деякими прикладами організації пошукових програм і формування хеш-функцій.

Базовою позицією обробки структур даних є пошук. Методика пошуку може бути різноманітною простою і складною в залежності від складності структур даних. Розглянемо спочатку два прості універсальні методи пошуку (Find) даних лінійний та бінарний. Ці пошуки прив'язуються до деякого ключа. Ключ (key) – ім'я даних або його частина. Наприклад $key \mapsto 7|a|abc$.

Метод лінійного пошуку полягає в послідовному огляді комірок структури даних і перевірки на співпадіння даних із значенням ключа:

$$\begin{matrix} & & & & & & & key\downarrow & \\ a & b & 6 & e & df & 10 & & & \end{matrix}, return\begin{cases} key|i \\ -1 \end{cases}.$$

Результатом пошуку є значення ключа або індекс елементу масиву, в якому міститься дане. У випадку відсутності даного ключа у структурі масиву, результат пошуку повертає –1.

Очевидна складність цього алгоритму – $O(n)$.

При бінарному пошуку береться деякий ключ і проглядається упорядкований масив з n елементів починаючи з середини (mid) послідовності даних або списку

$$\begin{matrix} & & & & key\downarrow & & & \\ 3 & 5 & 6 & 8 & 8.7 & 10 & 12 & 15 \\ \uparrow mid & & & & & & & \end{matrix}$$

на предмет співпадіння значення з цим ключем. Якщо значення ключа менше за середнє, тоді обробляється ліва частина списку, у якій mid приймає нове середнє значення.

$key < mid$ $key \downarrow$

3 5 6 8 8.7 10 12 15

$\uparrow mid$

Так як процес обробки становить $n, \frac{n}{2}\frac{n}{4}\frac{n}{8}, \ldots, 1 \sim log_2 n$, тому складність бінарного пошуку $O(log_2 n)$. Кращий варіант $O(1)$ і найгірший варіант $O(n)$, тому що порівняння складності алгоритмів пошуку наступне $log_2 n < n$.

Алгоритм бінарного пошуку може бути описаний рекурсивно.

Припустимо, що відсортований список A характеризується нижнім граничним індексом low і верхнім – high. Маючи ключ, починається пошук на збігання значень в середині списку (індекс mid).

mid = (low + high) / 2. Порівняти A [mid] з ключем.

Якщо збіг стався, тоді виконується умова зупинки, що дозволяє припинити пошук і повернути індекс mid. Якщо ж збіг не відбувся, можна скористатися тим фактом, що список впорядкований, і обмежити діапазон пошуку «нижнім підсписком» (зліва від mid) або «верхнім підсписком» (праворуч від mid).

У тому випадку коли ключ key < A[mid], тоді збігання значень може відбутися тільки в лівій половині списку в діапазоні зачень індексів від low до mid –1. Якщо ключ key > A[mid], збігання може відбутися тільки в правій половині списку в діапазоні індексів від mid +1 до high.

Крок рекурсії направляє бінарний пошук для продовження в один з підсписків. Рекурсивний процес переглядає все менші й менші списки. Зрештою пошук закінчується невдачею, якщо підсписки зникли. Це відбувається тоді, коли верхня межа списку стає менше ніж нижня межа. Умова low > high є другою умовою зупинки. У цьому випадку алгоритм повертає значення-1.

Розглянемо шаблонову версії бінарного пошуку, в якості параметрів якої використовується масив елементів типу T, значення ключа, а також верхній і нижній граничні індекси. Оператор If обробляє дві умови зупинки: 1) збіг стався, 2) ключового значення немає в списку. У блоці $Else$ оператора If виконується крок рекурсії, який направляє подальший пошук в лівий (ключ < A[mid]) або в правий підсписок (ключ > A[mid]). Той же алгоритм застосовується за принципом «розділяй і володій» до послідовності все менших інтервалів, поки не відбудеться успіх (збіг) або невдача.

```
template<class T>
int BinSearch(T A[], int low, int high, T key) {
  int mid;
  T midvalue;
  // умова зупинки: ключ не знайдено
  if (low > high)
```

```
        return -1;
    // Порівняти ключ з елементом у середині списку.
    // Якщо збігу немає, розділити на підсписки.
    // Застосувати процедуру бінарного пошуку до відповідного підсписку
    else {
        mid = (low + high) / 2;
        midvalue = A[mid];
        // умова зупинки: ключ не знайдено
        if (key == midvalue)
            return mid; // ключ знайдено по індексу mid
        // переглядати лівий підсписок, якщо key < midvalue;
        // в іншому випадку – правий підсписок
        else if (key < midvalue)
            // крок рекурсії
                return BinSearch(A, low, mid - 1, key);
        else
            // крок рекурсії
                return BinSearch(A, mid + 1, high, key);
    }
}
```

У попередніх параграфах було розглянуто ряд множинних структур, котрі дозволяють програмі-клієнту здійснювати пошук і вибірку даних. У кожній такій структурі метод Find виконує обхід списку і шукає елемент даних, що збігається з ключем. При цьому ефективність пошуку залежить від структури списку. У разі послідовного списку метод Find гарантовано переглядає елементи з порядком $O(n)$, у той час як у випадку бінарного пошуку забезпечується більш висока ефективність пошуку $O(log_2 n)$. В ідеалі хотілося б вибирати дані за показником $O(1)$. Але у цьому випадку кількість необхідних порівнянь не залежить від кількості елементів даних. Вибірка елемента здійснюється за показником $O(1)$ при використанні в якості індексу масиву деякого ключа.

Крім наведених базових методів пошуку даних існують комбіновані методи. Наприклад, пошук слова за ключем k_1 у словникові S_1, $dim\, S_1 = n$ і пошук слова за ключем k_2 фразеологічного речення у фразеологічному словникові S_2, $dim\, S_2 = m$. У цьому випадку, доцільно виконувати пошук за моделлю двох-стрічкового двох-головкового автомату. При лінійній обробці пошуку складність кожного алгоритму стрічки становить $O(n)$ і $O(m)$ і для загального двох-стрічкового автомату – $O(n + m)$.

Існує велика кількість прикладів зв’язку ключів з даними великого розміру, це телефонні номери власників стаціонарних телефонів міста або власників мобільного зв’язку відповідного оператора, власників банківських платіжних карток тощо. Пошук у спеціальних базах даних також відбувається за відповідними ключами.

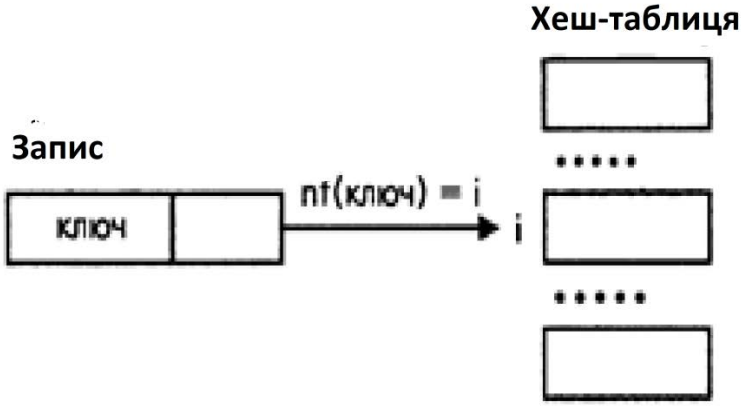
Ключі пошуку не обов'язково повинні бути числовими, символьними або словами. Наприклад, сформована компілятором таблиця символів (symbol

table) містить всі використовувані в програмі ідентифікатори разом з супутньою кожному з них інформацією. Ключем для доступу до конкретного запису є сам ідентифікатор.

У загальному випадку ключі не настільки прості, як у розглянутих прикладах. Незважаючи на те, що ключі забезпечують доступ до даних, вони, як правило, не є безпосередніми індексами в масиві записів. наприклад, телефонний номер може ідентифікувати клієнта, але навряд чи окрема АТС зберігає багатомільйонний масив телефонів громадян держави.

У більшості додатків ключ забезпечує непряме посилання на дані. Ключ k відображається в множині цілих чисел за допомогою *хеш-функції* (hash function) χ, тобто $\chi(k) = m \in \mathbb{N}_0$, де $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Отримане в результаті значення потім використовується для доступу до даних.

Припустимо, є множина записів з ціло-чисельними ключами. Хеш-функція χ відображає ключ k у ціло-чисельний індекс i з діапазону $[\overline{0, n-1}]$. З хеш-функцією пов'язана так звана *хеш-таблиця* (hash table), елементи якої пронумеровані від 0 до $n-1$ і зберігають самі дані або посилання на них.



Припустимо, що ключ k – позитивне ціле, а $\chi(k)$ – значення молодшого розряду числа k, тоді діапазон індексів дорівнює $[\overline{0,9}]$.

Приклад 4.1. Нехай $k = 49$, то значення функції $\chi(k) = 9$. Таким чином хеш-функція в якості свого значення провертає залишок від ділення значення ключа 49 на число 10.

Наведемо просте програмне представлення алгоритму Хеш-функції, котра повертає значення молодшого розряду ключа

```
// Key – значення ключа, яке обробляє хеш-функция HF
```

```
int HF(int Key)
{
    return Key % 10;
}
```

Функція χ є частковим випадком відображення «багато до одного» або задається суперпозицією відображень ключа за правилом $\chi_i(\chi_j(k))$, або ще більш складно за допомогою топологічних відображень і їх математичних фрагментів вузлів, кос, струн та іншого.

Так як відображення, здійснюване хеш-функцією, може бути не взаємно однозначним, то це призводить до *колізій* (collisions).

Визначення 4.1. Колізією за ключами k_i і k_j називається ситуація, при якій для $k_i \neq k_j$ має місце $\chi(k_i) = \chi(k_j)$.

Для прикладу 4.1 і значенні ключа Key = 39 програмна функція HF повертає однакові значення HF10 (49) = HF10 (39) = 9. Отже при виникненні колізії два або більше ключа асоціюються з одним і тим самим осередком хеш-таблиці. Оскільки декілька ключів не можуть займати одну і ту ж комірку в таблиці, повинна бути розроблена стратегія для вирішення колізій. Схеми вирішення колізій обговорюються після знайомства з деякими типами умов, яким повинні задовольняти хеш-функції.

Хеш-функція повинна відображати ключ в ціле число з діапазону $[\overline{0, n-1}]$. При цьому кількість колізій повинна бути обмеженою, а обчислення самої хеш-функції – дуже швидким. Розглянемо деякі методи побудови хеш-функцій, які задовольняють цим вимогам.

Найбільш часто у програмуванні використовується *метод поділу* (division method), що вимагає двох кроків. Спершу ключ з довільними даними повинен бути перетворений в ціле число, а потім отримане значення вписується в діапазон $[\overline{0, n-1}]$ за допомогою оператора отримання залишку. На практиці метод поділу використовується в більшості додатків з застосуванням хешування.

Наприклад, нехай ключ є п'ятизначним числом. Хеш-функція будується наступним чином. Спочатку вибирається молодше двох-розрядне число. Так, якщо це число дорівнює 56389, тоді HF100 (56389) = 89.

```
int HF (int Key)
{
 return Key %  100                  //метод ділення на 100
}
```

Ефективність хеш-функції залежить від того, чи забезпечує вона рівномірне розсіювання ключів в діапазоні $[\overline{0, n-1}]$. Якщо дві останні цифри відповідають року народження, то буде занадто багато колізій при ідентифікації платників відповідних податків.

Ключ – символьний рядок мови С++. Хеш-функція відображає цей рядок в ціле число за допомогою підсумовування першого і останнього символів і подальшого поділу на 101 (розмір таблиці).

```
// хеш-функція для символьного рядка.
// повертає значення в діапазоні від 0 до 100
int HF(char *key)
{
int len = strlen(key), hashf = 0;
//якщо довжина ключа дорівнює 0 або 1, повернути key(0).
//інакше додати перший та останній символ
  if (len <= 1)
     hashf = key[0];
  else
     hashf = key[0] + key[len-l];
  return hashf % 101;
}
```

У загальному випадку при великих значеннях n індекси мають більший розкид. Крім того, математична теорія стверджує, що розподіл буде більш рівномірним, якщо n – просте число.

Розглянемо інший поширений метод *середини квадрата* (midsquare technique), котрий передбачає перетворення ключа в ціле число, зведення його в квадрат і повернення в якості значення функції послідовності бітів, взятих з середини цього квадрата. Припустимо, що ключ є ціле 32-бітове число. Тоді наступна хеш-функція вибирає середні 10 біт зведеного в квадрат перетвореного ключа.

```
//повернути середні 10 біт добутку key*key
int HF (int key)
{
key* = key;    // довести ключ в квадрат
key>>=11;    //віднести 11 молодших біт
return key % 1024;      // повернути 10 молодших біт
}
```

Наведемо ще один *мультиплікативний метод* (multiplicative method) побудови функції хешування. У цьому методі використовується випадкове дійсне число f в діапазоні 0 <f <l. Дробова частина добутку f * key лежить в діапазоні від 0 до 1. Якщо цей добуток помножити на n (розмір хеш-таблиці), то ціла частина отриманого результату дасть значення хеш-функції в діапазоні від 0 до n-1.

```
// хеш-функція, що використовує мультиплікативний метод;
// повертає значення в діапазоні 0…700
int HF (int key)
{
static RandomNumber rnd;
float f;
// помножити ключ на випадкове число з діапазону 0…1
```

```
f = key * rnd. F.Random ( );
// взяти дробову частину числа
f = f – int (f);
// повернути число в діапазоні 0…n-1
return 701*f;
}
```

Наведені у параграфі ознайомчі матеріали дозволяють розширити знання слухачеві з методології пошуку даних, організації пошукових ключів та з деякими навичками формування хеш-функцій.

4.2. Вирішення колізій та клас хеш-таблиць

В попередньому параграфі було введено поняття колізії між ключами, за якими відбувається пошук даних у хеш-таблицях. Метою цього параграфу є ознайомлення читача з деякими прийомами вирішення колізій та оволодіння алгоритмічними можливостями розв’язання колізій і їх програмної реалізації.

Незважаючи на те, що два або більше ключів можуть хешувати однаково, вони не можуть займати в хеш-таблиці одну і ту ж комірку. Тому залишається два шляхи вирішення цієї колізії: або знайти для нового ключа іншу позицію в хеш-таблиці, або створити для кожного значення хеш-функції окремий список всіх ключів, які відображаються при хешуванні в це значення. Обидва варіанти являють собою дві класичні стратегії вирішення колізій – *відкриту адресацію з лінійним опробування* (linear probe open addressing) та *метод ланцюжків* (chaining with separate lists).

Розглянемо спочатку метод відкритої адресації для вирішення питання ключової колізії при пошуку даних у табличних структурах.

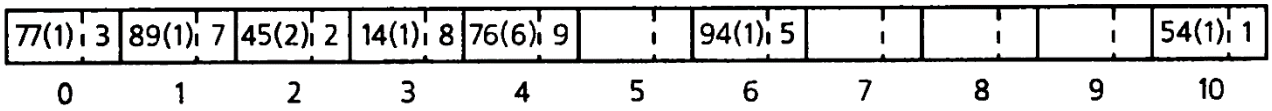
Суть методу відкритої адресації з лінійним опробування полягає в застосуванні методики вставки ключа у вільне місце хеш-таблиці. Зокрема, методика передбачає, що кожна клітина таблиці віртуально позначена як незайнята. Тому при додаванні нового ключа завжди можна визначити, чи дійсно зайнята дана осередок таблиці чи ні. Якщо так (клітина зайнята), алгоритм здійснює «опробування» по колу, поки не зустрінеться «відкрита адреса» (вільне місце). Звідси і назва методу. Якщо розмір таблиці великий щодо числа збережених там ключів, метод працює добре, оскільки хеш-функція буде рівномірно розсіювати ключі по всьому діапазону даних і число колізій буде мінімальним. По мірі того як статистичний коефіцієнт заповнення таблиці наближається до одиниці, ефективність процесу помітно знижується. Продемонструємо методику лінійного випробування на прикладі семи записів таблиці.

Приклад 4.2. Припустимо, що дані записів мають тип DataRecord і зберігаються в одинадцяти-елементній таблиці із структурою:

```
struct DataRecord
{
int key;
int data;
};
// В якості хеш-функції HF використовуються залишок від ділення на 11,
що приймає значення в діапазоні [0,10].
HF(item) = item.key % 11
     // У таблиці зберігаються такі дані. Кожен елемент позначений
парою
      // (дане, число спроб), де число спроб, вказує на кількість
необхідних
     // спроб для його розміщення в таблиці.
```

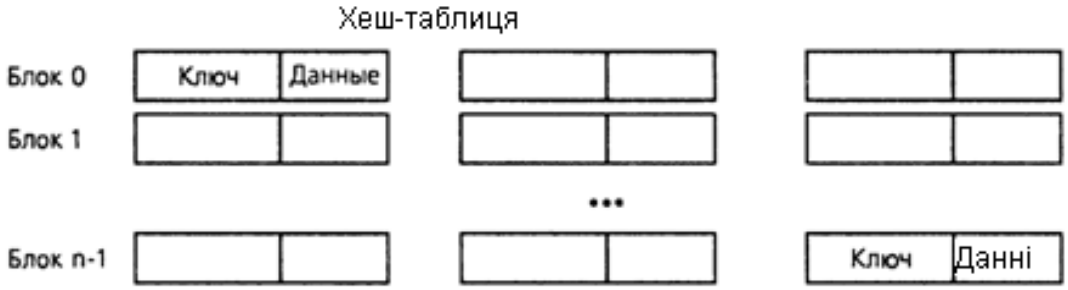
Нехай за списком пар: (54,1), (77,3), (94,5), (89,7), (14,8), (45,2), (76,9) сформована таблиця

Хеш-таблиця



Хешування перших п'яти ключів дає п'ять різних індексів, по яких ці ключі запам'ятовуються в таблиці. Наприклад, HF11 (54,1)= 10, і цей елемент потрапляє в Table[10]. Перша колізія виникає між ключами 89 і 45, так як обидва вони відображаються в номер комірки (індекс) 1. Тому, що елемент даних (89,7) йде першим у списку, то він займає позицію Table[1]. При спробі записати (45,2) виявляється, що місце Table[1] вже зайнято. Тоді починається послідовне опробування осередків таблиці з метою знаходження вільного місця. У даному випадку це Table[2]. На ключі 76 ефективність алгоритму сильно падає. Цей ключ хешується в індекс 10, для якого місце, вже зайняте. Тому у процесі опробування здійснюється оглядання ще п'яти комірок, перш ніж буде знайдено вільне місце в клітині Table[4]. Загальна кількість спроб для розміщення в таблиці всіх елементів списку дорівнює 13, тобто в середньому 1,5 спроб на один елемент хеш-таблиці.

Зосередимо тепер увагу на методі ланцюжків, як на домінуючій стратегії вирішення колізій. При цьому підході до хешування, таблиця розглядається як масив пов'язаних списків. Кожен такий об’єкт (ланцюжок) називається блоком (bucket) і містить записи, які відображаються хеш-функцією в один і той же табличний адрес. Тому ця стратегія вирішення колізій і називається методом ланцюжків.



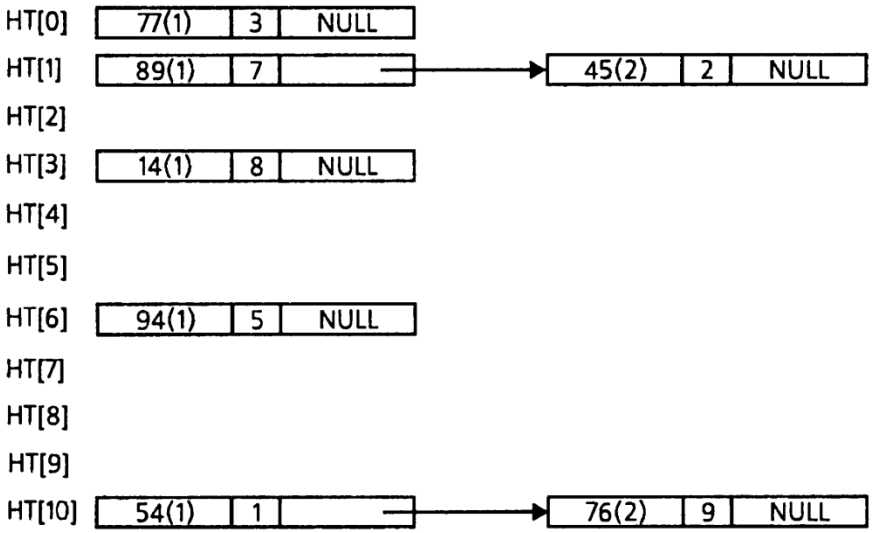
Якщо таблиця є масивом пов'язаних списків, то елемент даних просто вставляється у відповідний список в якості нового вузла. Щоб виявити елемент даних, потрібно застосувати хеш-функцію для визначення потрібного пов'язаного списку і виконати там послідовний пошук.

Розглянемо стратегію методу ланцюжків на записах типу DataRecord і хеш-функції HF із прикладу 4.2. Нагадаємо, що список утворено послідовністю пар (54,1), (77,3), (94,5), (89,7), (14,8), (45,2), (76,9), а функція HF визначена програмним фрагментом

HF (item) = item.key % 11

Кожен новий елемент даних (пара даних) вставляється в хвіст відповідного пов'язаного списку.

У наступній таблиці кожне значення даних супроводжується числом спроб, необхідних для запам'ятовування цього значення.



Якщо вважати спробу як вставку нового вузла, тоді їх загальна кількість при включенні семи елементів дорівнює 9, тобто в середньому 1,3 спроби на елемент даних (порівняйте з попереднім методом).

У загальному випадку метод ланцюжків швидше формує таблицю ніж за методом відкритої адресації, оскільки переглядає тільки ті ключі, які потрапляють в одну й ту саму табличну адресу. Крім того, відкрита адресація передбачає наявність таблиці фіксованого розміру, в той час як у методі ланцюжків елементи таблиці створюються динамічно, а довжина списку обмежена лише кількістю пам'яті. Основним недоліком методу ланцюжків є

додаткові витрати пам'яті на поля покажчиків. У загальному випадку динамічна структура методу ланцюжків більш краща для хешування.

Перейдемо тепер до створення загального класу хеш-таблиць (HashTable), який здійснює хешування методом ланцюжків. Цей клас утворюється на базовому абстрактному класі List і забезпечує механізм зберігання з дуже ефективними методами доступу. Клас допускає дані будь-якого типу з тим лише обмеженням, що для цих типів даних повинен бути визначений оператор порівняння (==), щоб порівняти ключові поля двох елементів даних. Причому прикладна програма повинна вміти перевантажувати оператор (==).

Крім класу хеш-таблиць вводиться також клас HashTablelterator, який полегшує обробку даних хеш-таблиць. Об'єкт типу HashTablelterator знаходить важливе застосування і при сортуванні та друкуванні даних.

Оголошення та реалізації цих класів знаходяться у файлі hash.h.

Спочатку запровадимо об’яву класів для створення таблиці

```
#include “array.h”
#include “list.h”
#include “link.h”
#include “iterator.h”
template <class T>
class HashTableIterator;
temperate<class T>
class HashTable: public List <T>
{
  protected:
    // кількість блоків; визначає розмір таблиці
     int numBuckets ;
    // хеш-таблиця є масив зв'язаних списків
    Array< LinkedList<T> buckets;
//хеш-функція й адреса елемента даних,
//до якого зверталися останній раз
Unsigned long (*hf) (T key)
T  *current;
public:
      // конструктор з параметрами, що включають
      // розмір таблиці і хеш-функцію
    HashTable (int nbuckets, unsigned lond hashf (T key) ) ;
      // методи обробки списків
   virtual void Insert (const T& key) ;
   virtual int Find (T& key) ;
   virtual void Delete (const T& key) ;
   virtual void ClearList (void) ;
   void Update (const T& key) ;
      // дружній ітератор, що має доступ до
      // данних-членів
  Friend class HashTableIterator<T>
}
```

Описання наведеної об’яви.

Об'єкт типу HashTable є список елементів типу T. В ньому реалізовані всі методи, які вимагає абстрактний базовий клас List. Прикладна програма повинна задати розмір таблиці і хеш-функцію, перетворюючу елемент типу T в довге ціле без знака. Такий тип значень, які повертає хеш-функція допускається для широкого діапазону даних.

Методи Insert, Find, Delete і ClearList є базовими методами обробки списків. Окремий метод Update слугує для оновлення елемента, який вже є наявним в хеш-таблиці.

Методи ListSize і ListEmpty реалізовані в базовому класі. Елемент даних current завжди вказує на останнє доступне значення даних. Він використовується методом Update і похідними класами, які повинні повертати посилання.

Розглянемо приклад програми, яка формує хещ-таблицю.

Припустимо, що NameRecord є запис, котрий містить поле найменування і поле лічильника.

```
struct NameRecord
{
String name;
int count;
};
//  101-елементна таблиця, що містить дані типу NameRecord
//  і має хеш-функцію hash
HashTable < NameRecord>  HF(101, hash);
// вставити запис (“Betsy”, 1) в таблицю
NameRecord rec ;                        //  змінна типу NameRecord
rec.name = “Betsy” ;                    //  присвоєння name = ”Betsy
rec.cunt = 1 ;                          // i count = 1
HF. Insert (rec) ;                      // Вставити запис
count << HF.ListSize ( ) ;              // роздрукувати розмір таблиці
// вибрати значення даних, що відповідають ключу “Betsy”,
// збільшити поле лічильника на 1 і поновити запис
rec.name = “Betsy”;
if (HF.Find (rec))                              //  знайти “Betsy”
{
  rec.count  += 1;                              // поновити поле данних
  HF.Update (rec);                              //  поновити запис в
таблиці
}
else
   cerr  << “Помилка:  \ “Ключ  Betsy повинен бути в таблиці. \ “\ n;
```

Клас HashTablIteration утворений із абстрактного класу Iterator і містить методи для проглядання даних у таблиці.

Розглянемо клас Iterator.

Специфікація класу HashTableIterator

Об’ява класу

```
template <class T>
class HashTableIterator: public Iterator<T>
{
  private:
   // покажчик таблиці, що підлягає обходу
   HashTable<T>  *HashTable;
   // індекс поточного блоку, що проглядається
   // і покажчик, що вказує на зв'язаний список
   int currentBucket;
   LinkeList<T>  *currBucketPtr;
   //утиліта для реалізації методу Next
   void SearchNextNode (int cb) ;
 public:
   // конструктор
   HashTableIterator  (HashTable<T>& ht) ;
  // базові методи інтератора
  virtual void Next (void) ;
   virtual void Reset (void) ;
   virtual T& Data (void) ;
   //  підготувати ітератор для сканування другої таблиці
   void SetList (HashTable< T>& Ist) ;
} ;
```

Наведемо описання методу Next у об’яві класу Iterator.

Метод Next виконує проходження списку (блоку) за списком хеш-таблиці, по вузлах кожного списку, причому значення даних, що видаються ітератором ніяк не впорядковані. Для виявлення чергового списку, що підлягає проходженню, метод Next використовує функцію SearchNextNode.

Нижче подано приклад фрагменту програми.

```
// оголосити ітератор об'єкта HF типу HashTable
 HashTableIterator<NameRecord> hiter (HF)  ;
// сканувати всі елементи даних
for (hiter.Reset ( ) ; !hiter/EndOfList ; hiter.Next ( ) )
{
   rec = hiter.Data( ) ;
   count << rec.name <<  “: “  << rec.count <<  endl;
}
```

У параграфі розглянуті методи вирішення колізій, наведено порівняння їх ефективності і вказано на деякі недоліки. Запропоновані прийоми організації програм з формування хеш-таблиць. Зокрема класу створення таблиць з застосуванням бібліотеки класів середовища C++.

4.3. Зовнішні носії і файлові операції

До сих пір розглядалися множини даних, які можуть одночасно розташовуватися у пам’яті машини. Але великі набори даних, з якими приходиться стикатися програмістам, можуть містити мільйони записів, котрі

неможливо розмістити в пам'яті одночасно і тому такі дані розташовують в файлах на диску або на інших носіях. Мета цього параграфу полягає в наданні знайомства з організацією операцій вводу та виводу даних бінарних файлів на дисках, отримання загальних відомостей з конструювання алгоритмів зовнішнього сортування та пошуку даних.

Розглянемо спочатку загальні відомості відносно бінарних файлів. Текстовий файл містить рядки ASCII-символів, розділені символами кінця рядка. Бінарний файл складається із записів, які варюються від одного байту до складних структур, які включають цілі числа, числа з плаваючою точкою і масиви. З апаратної точки зору записи файлу являють собою блоки даних фіксованої довжини, що зберігаються на диску. Блоки, як правило, несуміжні. Однак з логічної точки зору записи розташовуються у файлі послідовно. Файлова система дозволяє здійснювати доступ як до окремих записів, так і до всього файлу цілком, розглядаючи останній як масив записів. Під час вводу або виводу даних система підтримує файловий вказівник (file pointer) – поточну позицію у файлі.

Файл як структура прямого доступу з компонентами даних $R_0, R_1, \dots, R_{n-1}$ має стрічкову структуру

Файл є також послідовною структурою, яка зберігає файловий покажчик в поточній позиції всередині даних стрічки. Операції вводу або виводу звертаються до даних у поточній позиції, а потім застосовуються до наступного файлового запису.

Файл даних визначається як структура послідовного доступу до його набору даних:

В середовищі програмування C++ мається клас fstream, який описує файлові об'єкти, котрі можуть використовуватися як для вводу, так і для виводу даних. При створені якого то об'єкту, можна використовувати метод open для призначення файлу фізичного імені та режиму доступу. Можливі режими визначені в базовому класі ios.

Режим	Дія
in	відкрити файл для читання

out	відкрити файл для запису
trunc	видалити запис до читання або запису
nocreate	якщо файл не існує, не створювати порожній файл; повернути помилковий стан потоку
binary	відкрити файл, вважаючи його бінарним (не текстовим)

Розглянемо приклад використання класу fstream.

```
#include < fstream.h>
     // Об’ява файлу
fstream f;
// Відкрити текстовий файл Phone для вводу.
// Якщо такого файлу немає, повідомити про помилку
f.open("Phone", ios::in | ios: mocreate) ;
// Оголошення файлу
fstream f;
// Відкрити бінарний файл для вводу
f.open("DataBase", ios::in | ios::out | ios:ibinary);
```

Кожен файловий об'єкт має асоційований з ним файловий покажчик, який вказує на поточну позицію для введення або виведення. Функція tellg() повертає зсув в байтах від початку файлу до поточної позиції у вхідному файлі. Функція tellp() повертає зсув в байтах від початку файлу до поточної позиції у вихідному файлі. Функції seekg() і seekp() дозволяють пересувати поточний файловий вказівник. Всі ці функції приймають в якості параметру кількість позицій зсуву покажчика, який вимірюється числом байтів відносно початку файлу (beg), кінця файлу (end) або поточної позиції у файлі (cur). Якщо файл використовується як для введення, так і для виводу даних рекомендується застосовувати функції tellg і seekg.

Наступний код демонструє дії функцій seekg і tellg:

```
// Бінарний файл цілих чисел
fstream f;
f.open("datafile", ios::in | ios::nocreate | ios::binary);
// скинути поточну позицію на початок файлу
f.seekg(0, ios::beg);
// встановити поточну позицію на останній елемент даних
f.seekg(-sizeof(int), ios:rend);
// просунути поточну позицію до наступного запису
f.seekg(-sizeof(int), ios::cur);
          . . .
```

```
// переміститися до кінця файлу
f.seekg(0, ios::end);
// роздрукувати число байтів у файлі
cout « f.tellg() << endl;
// роздрукувати число елементів-даних у файлі
cout << f.tellg()/sizeof(int);
```

Клас fstream має базові методи read і write, що виконують введення та виведення потоку байтів. Кожному методу передається адреса буфера і лічильник байтів, що пересилаються. Буфер є масивом символів, в якому дані запам'ятовуються в тому ж вигляді, в якому вони прийняті або надіслані на диск. Операції вводу та виводу даних не символьного типу потребують приведення до типу char.

Наприклад, наступні операції передають блок цілих чисел:

```
// Оголошення файлу
fstream f;
int data = 30, A[20];
// Записати число 30 як блок символів довжиною sizeof (int)
f.write((char*) &data, sizeof(int));
// прочитати 20 чисел із файла f в масив A
f.read((char*)A, 20*sizeof(int));
```

Файловий ввід та вивід використовується в багатьох додатках і тому необхідно визначити деякий клас, який би забезпечив загальні операції обробки бінарних файлів. Таким є клас BinFile, який повністю приховує від користувача системні деталі нижнього рівня. Оскільки цей клас визначений як шаблон, породжувані ним файли можуть містити різні типи даних.

Розглянемо специфікацію цього класу.

Оголошення.

```
// Системні файли, що містять методи для обробки файлів
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "strclass.h"
// Типи доступу до файлу
enuiri Access {IN, OUT, INOUT};
// Типи зсуву в операціях пошуку
enum SeekType {BEG, CUR, END};
template <class T>
class BinFile
{
private:
// файловий потік зі своїм ім'ям і типом доступу
fstream f;
Access accessType; // тип доступу
String fname; // фізичне ім'я файлу
int fileOpen; // файл відкритий?
// параметри, що характеризують файл як структуру прямого доступу
```

```
int Tsize; // розмір запису
int filesize; // число записів
// видає повідомлення про помилку і завершує програму
void Error{char *msg);
public:
// конструктори і деструктор
BinFile(const Strings fileName, Access atype = OUT);
~BinFile(void);
// конструктор копіювання.
// об'єкт повинен передаватися за посиланням.
// завершується програмно
BinFile(BinFile<T>& bf);
// утиліти обслуговування файлу
void Clear(void); // очистити файл від записів
void Delete(void); // закрити файл і видалити його
void Close(void); // закрити файл
int EndFile(); // перевірити умову кінця файлу
long Size(); // повернути кількість записів  файлу
void Reset(void); // встановити файл на перший запис
// перемістити файловий вказівник на pos записів щодо
// початку файлу, поточної позиції або кінця файлу
void Seek(long pos, SeekType mode);
 // блочне читання n елементів даних в буфер з адресою A
int Read( *A, int n);
// блоковий запис n елементів даних з буфера з адресою A
void Write(T *A, int n);
// Вибрати запис, розташований в поточній позиції
T Peek(void);
// копіювати data в запис, розташовану в позиції pos
void Write (const T& data, long pos);
// читати запис за індексом pos
T Read (long pos);
// занеести запис в кінець файлу
void Append(T item);
}
```

Деякі пояснення до цієї програми, приклади реалізації і інше наведено у роботі [1].

Раніше розглянуті прийоми збереження даних у внутрішніх спискових структурах подібним чином можна виконати і для даних, що зберігаються у файлі. Ефективність методів зовнішнього сортування і пошуку залежить від організації записів файлу. Поширимо концепцію хешування на файлові структури і застосуємо методи класу BinFile для доступу до даних.

Методики хешування відкритої адресації та методу ланцюжків забезпечують високу ефективність пошуку і можуть бути застосована до зовнішніх структур. В подальшому буде розглядатися метод ланцюжків, при якому файл містить зв'язані списки записів.

Нехай у пам’яті створена хеш-таблиця, за допомогою якої відбувається звертання до дискового пристрою і кожен запис на диску представляється парою полів (data, nextindex) даного і файлового індексу. Ці записи зберігаються на диску у вигляді зв'язаного списку. Поле nextlndex вказує позицію наступного запису файлу.

Хеш-функція відображає кожен запис в табличний індекс так, що:

```
int hashtable[n];                    // масив файлових індексів
```

Хеш-таблиця представляється в пам'яті у вигляді n-елементного масиву (див. рис.4.1). Спочатку таблиця порожня (кожна клітинка містить -1), показуючи тим самим, що записів у файлі немає. Як тільки вводиться запис з бази даних, хеш-функція визначає індекс в таблиці. Якщо відповідна комірка таблиці порожня, запам'ятовується сам запис на диску, а його позицію зберігається у файлі – в таблиці.

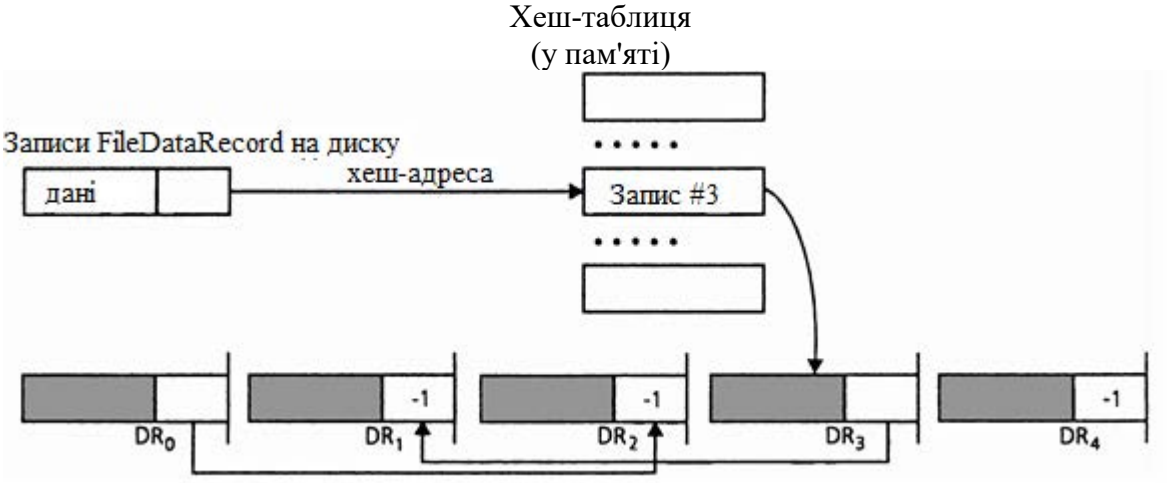


Рис. 4.1. Записи FileDataRecord на диску

Тепер комірка таблиці містить дискову адресу першого відображеного у цю комірку запису. Ця адреса може використана для доступу до відповідного зв'язаного списку і вставки туди нового запису. Процес вставки полягає у виведенні запису на диск і оновленні вказівника в полі nextlndex. Проілюструємо цей процес на простому прикладі, який, тим не менше, виражає головні його особливості. Нехай дані мають цілий тип і запам'ятовуються у файлі у вигляді списку записів FileDataRecord.

Наведемо схему алгоритму зовнішнього хешування.

```
// Вузол списку, в якому зберігається запис
struct FileDataRecord
{
// Поле data є ціле число, на практиці
// найчастіше поле data є складним записом
int data;
// посилання на наступний запис у файлі
int nextlndex;
};
```

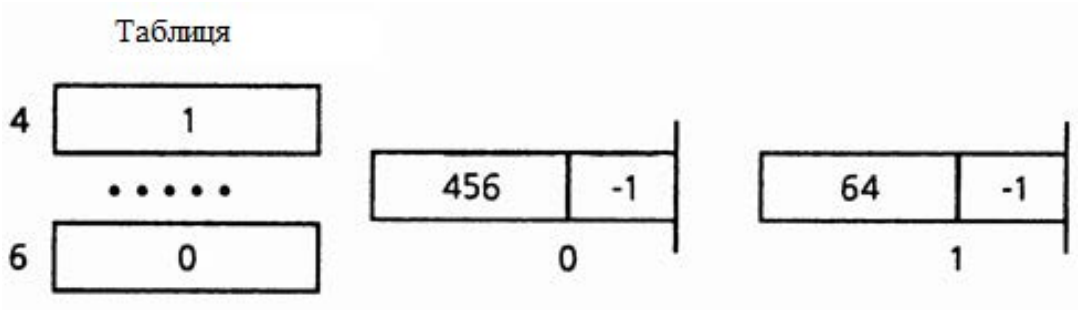
Нехай у файлі запам’ятовуються числові дані: 456, 64, 84, 101, 144.

Хеш-функція відображає кожне значення даних в інше ціле число, що виражається молодшою цифрою вихідного числа.

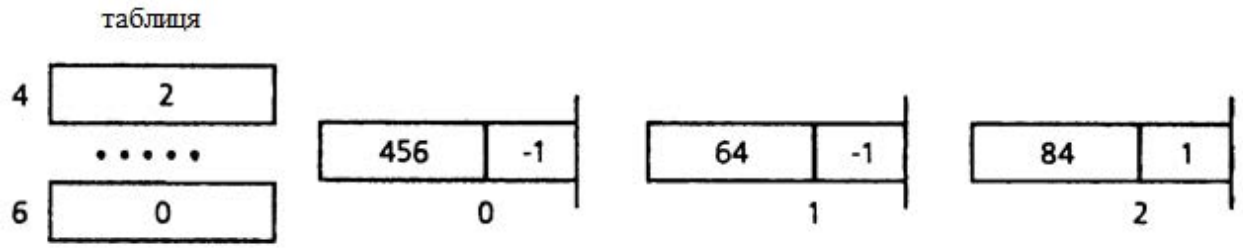
```
h(data) = data % 10;   // h(456) = 6; . . . h(101) = 1; h(144) = 4
```

Перші два числа відображаються у порожні клітинки таблиці, отже, можуть бути відразу вставлені у файл в якості вузлів.

Перший вузол запам'ятовується в позиції 0, а другий - у позиції 1. номери позицій заносяться у відповідні комірки таблиці.



Комірка таблиці, відповідна числу 84, містить 1 – номер запису файлу, що є першим в деякому пов'язаному списку записів. Новий запис (84) вставляється в початок цього списку.



Після завантаження чисел 101 і 144 файл містить п'ять записів FileDataRecord, які логічно представляють собою три пов'язаних списки. Голови цих списків містяться у відомих осередках таблиці. У цьому методі зберігання ефективно використовується прямий доступ до файлу. Файл формується шляхом послідовного додавання записів і поновлення відповідних елементів таблиці. Часто сама таблиця створюється у вигляді окремого файлу і завантажується звідти в пам'ять по потребі. З програмою зовнішнього хешування за наведеним алгоритмом можна ознайомитися у [1].

Наведені матеріали ознайомили здобувачів знань з методикою організації даних у файлових системах, з операціями доступу та пошуку на основі методології хешування. Розглянуто приклади розробки необхідних для цього алгоритмів і окремі програмні засоби обробки даних.

РОЗДІЛ 5. ДЕРЕВА, ПРЕДСТАВЛЕННЯ ТА ОБРОБКА

Компактне представлення даних у вигляді дерев і порівняно швидкий пошук даних основна перевага дерев порівняно з множинними структурами масивами, списками тощо. У матеріалах наступних трьох параграфах розглядаються питання представлення дерев, їх проходження, методи класичної та спеціальної обробки деревовидних структур даних.

5.1. Дерева та засоби їх проходження

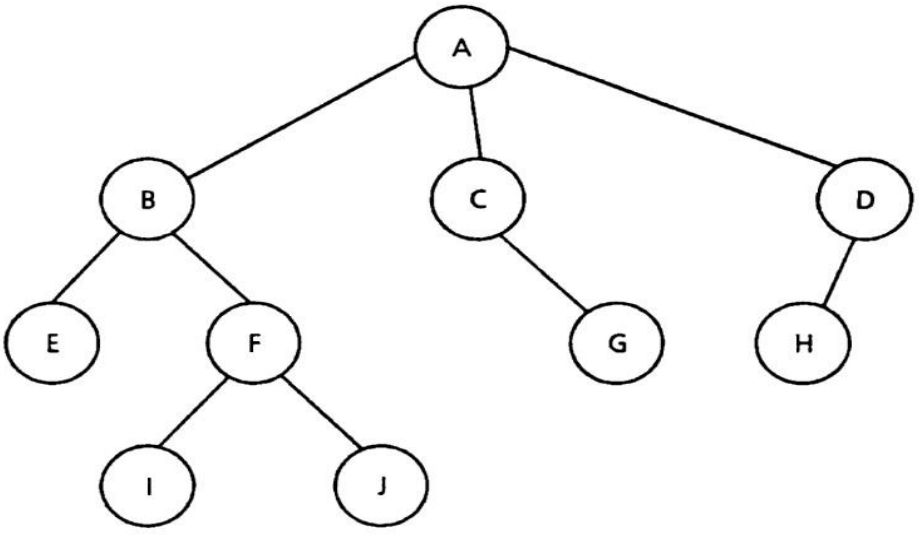
Деревовидна структура займає центральне місце в організації і обробці даних, з тієї причини, що задає можливість представляти нелінійні зв'язані дані і зручно та прискорено їх обробляти. Тому у параграфі ставиться задача засвоєння навичок програмного конструювання дерев і роботи з ними. Метою є знайомство з деякими прийомами формування і представлення та проходженням деревовидних структур.

Структура дерева задається сукупністю зв'язаних *вершин* (vertices) або *вузлів* (nodes), які походять від єдиної початкової або декількох початкових вершин, котрі називаються *коренями* (roots) таких структур.

Визначення. 5.1. Деревовидна структура з одним коренем називається *деревом*, а з декількома коренями – *лісом*.

В подальшому зосередимо увагу на деревах. Дерево можна задати декількома засобами, наприклад, формульно, графічно, табличко або інакше.

Нехай задана сукупність вершиш {A,D,C,D,E,F,G,H,I,J}. Якщо ввести відношення (|) – «або», (&) – «і», тоді формульне дерево представиться так: (A&(B&(E|(F&(I|J)))|(C&G)|(D&H))), це ж дерево має і графічний вигляд.



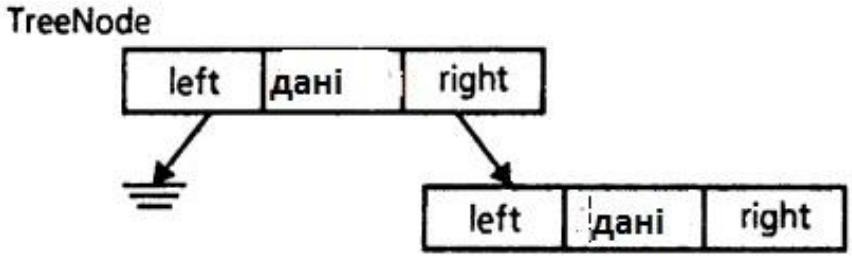
Графічні представлення дерев є наглядним і використовується на практиці, зокрема для виявлення їх характеристик:

- дві вершини дерева є *суміжними*, якщо між ними є прямий зв'язок, так вершини (B,F), (I,F), (F,J) є суміжними;
- *степінь вершини* дерева (St) визначається кількістю суміжних з нею вершин, отже St(F)=3;
- вершини дерева, степінь яких дорівнює одиниці утворюють його *листя*, так вершини {E,I,J,G,H} є листям наведеного дерева;
- *шлях дерева* між двома вершинами утворюють його прямо зв'язані вершини, отже шлях h(A,J)=(A,B,F,J);
- *довжина шляху* дерева між двома вершинами |h()| визначається кількістю вершин шляху без однієї, тобто |h(A,B,F,J)|=3;
- вершини дерева, які мають однакову довжину шляху, відносно його кореня, утворюють *рівень дерева*, корінь дерева знаходиться на рівні 0, а його вершини B,C,D на першому рівні, наведене у прикладі дерево має чотири рівні (0, 1, 2, 3);
- вершини, які знаходяться на рівні k по відношенню до вершин рівня $k-1$ прийнято називати *батьківськими,* а вершини рівня $k-1$ по відношенню до вершин рівня k синами, отож корінь A у розглянутому дереві є батьківською вершиною для вершин рівня один.

Визначення 5.2. Дерево степінь вершин якого не перевищує трьох звуть *бінарним* (binary tree) або скорочено (BT) деревом.

Вивчення бінарних дерев дає можливість вирішувати найбільш загальні завдання, пов'язані з деревами. Кожна вершина BT, степінь якої три, має *лівого* і *правого* синів (left child і right child). Так як кожне піддерево BT дерева є BT, то побудова бінарного дерева може бути виконана рекурсивно.

BT дерево, степінь кореня якого та його листа є одиниця, а всі інші вершини мають степінь 2, утворює звичайний зв'язаний список. Тому вузлові зв'язки такого дерева мають вигляд,



а листовий вузол містить NULL в полі правого і лівого покажчиків.

Отже вузол BT складається з поля даних, яке є відкритим (public) елементом і двох полів з вказівниками, котрі є закритими (private) елементами, доступ до яких здійснюється за допомогою функцій Left() і Right(). Оголошення і визначення класу TreeNode містяться у файлі treenode.h середовища C++.

Специфікація класу TreeNode.

Об'явлення

```
// BinSTree залежить від TreeNode
template <class T>
class BinSTree;
// Оголошення об'єкта для вузла бінарного дерева
template <class T>
class TreeNode
{
private: // вказівники лівого і правого синів
TreeNode <T> * left;
TreeNode <T> * right;
public: // Відкритий елемент, що допускає оновлення
T data; // Конструктор
TreeNode (const T & item, TreeNode <T> * lptr = NULL,
TreeNode <T> * rptr = NULL);
TreeNode <T> * Left (void) const; // Методи доступу до полів
вказівників
TreeNode <T> * Right (void) const;
// Зробити BinSTree дружнім, оскільки необхідний доступ до left і
right
friend class BinSTree <T>;
};
```

Реалізація класу TreeNode супроводжується ініціалізацією поля об'єкта. Для цього конструктор має параметр item, за яким вказівники призначають вузлу лівого і правого сина відповідні значення.

```
// Конструктор активізує поля даних і вказівників
// Значення NULL відповідає порожньому піддереву
template <class T>
TreeNode <T> :: TreeNode (const T & item, TreeNode <T> * lptr,
TreeNode <T> * rptr): data (item), left (lptr), right (rptr)
{}
```

Бінарне дерево складається з колекції об'єктів TreeNode, пов'язаних за допомогою своїх полів з вказівниками.

```
TreeNode <int> * p; // оголошення вказівника
p - new TreeNode (item); // лівий і правий вказівники дорівнюють NULL
```

Виклик функції new обов'язково повинен включати значення даних. Якщо як параметр передається також вказівник об'єкта TreeNode, то він використовується новоствореним вузлом для приєднання синовнього вузла. Функція GetTreeNode приймає дані і нуль або більше вказівників об'єкта TreeNode, для створення та ініціалізації вузла ВТ. При недостатній кількості доступної пам'яті виконання програми припиняється.

```
// Створити об'єкт TreeNode з вказівними полями lptr і rptr.
// За замовчуванням вказівники містять NULL.
template <class T>
TreeNode <T> * GetTreeNode (T item, TreeNode <T> * lptr <<NULL,
TreeNode <T> * rptr - NULL)
{
```

```
TreeNode <T> * p;
// Викликати new для створення нового вузла
// Передати туди параметри lptr і rptr
p = new TreeNode <T> (item, lptr, rptr);
// Якщо пам'яті обмаль, повідомити про помилку і завершити програму
if (p == NULL)
{
cerr <<" Помилка при виділенні пам'яті! \ п ";
exit (l);
}
return p; // Повернути вказівник на виділену системою пам'ять
}
```

FreeTreeNode приймає вказівник на TreeNode і звільняє займану вузлом пам'ять, викликаючи функцію delete. Функції містяться у файлі treelib.h

```
// Звільнити динамічну пам'ять, займану даним вузлом
template <class t>
void FreeTreeNode (TreeNode <T> * p)
{
     delete p;
}
```

Алгоритми проходження ВТ дерев (прямий, симетричний і зворотний) суттєво впливають на ефективність використання дерева. Реалізація методів проходження ВТ передбачає параметр-функцію visit, яка здійснює доступ до даних вузла. Передаючи в якості параметра функцію, можна задати дію, яка повинна виконуватися в кожному вузлі в процесі проходження дерева.

```
template <class T>
void <Метод_прохода> (TreeNode <T> * t, void visit (T & item));
Функція може бути аргументом, якщо вказати її ім'я, список параметрів
і її значення.
// Обчислити f(t) за допомогою функції f і параметру t.
int G (int t, int f (int x)) // параметр-функція f
{
// Повернути значення функції f(t) і значення її аргументу t
     return t * f (t) /
}
```

Викликаючи функцію G, клієнт повинен передати функцію f з тією ж структурою. Нехай у нашому прикладі клієнт визначив функцію XSquared, яка обчислює x2.

```
// XSquared – цілочислова функція з цілочисловим параметром x
int XSquared (int x)
{
     return x * x;
}
```

Як вказувалося раніше ВТ дерева можна обробляти рекурсивно. Сила рекурсії проявляється разом з методами проходження ВТ. Кожен алгоритм

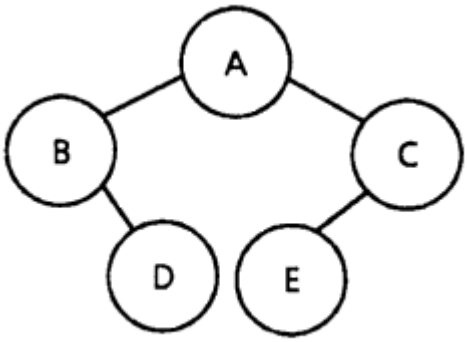
проходження дерева виконує в вузлі три дії: заходить вузол, рекурсивно спускається по лівому піддереву і по правому піддереву. Спустившись до піддерева, алгоритм визначає, що він знаходиться у вузлі, і може виконати ті ж три дії. Спуск припиняється після досягнення порожнього дерева (покажчик == NULL). Різні алгоритми рекурсивного проходження відрізняються порядком, в якому вони виконують свої дії у вузлі. Наведемо симетричний і зворотний методи, в яких спочатку здійснюється спуск по лівому піддереву, а потім по правому.

Симетричний метод проходження починає свої дії у вузлі спуском по його лівому піддереву. Потім виконується друга дія – обробка даних у вузлі. Третя дія – рекурсивне проходження правого піддерева. У процесі рекурсивного спуску дії алгоритму повторюються в кожному новому вузлі.

Отже, порядок операцій при симетричному методі наступний:

1. Проходження лівого піддерева.
2. Відвідування вузла.
3. Проходження правого піддерева.

Таке проходження має назву LNR (left, node, right) [1]. Для дерева Tree_0 у функції MakeCharTree "відвідування" означає друкування значення поля даних вузла.



Tree_0

При симетричному методі проходження дерева Tree_0 виконуються наступні операції.

Дія	Друк	Зауваження
Спуститися від A до B: Відвідати B;	B	Лівий син вузла B дорівнює NULL
Спуститися від B до D: Відвідати D;	 D	D листовий вузол Кінець лівого піддерева вузла A
Відвідати корінь A:	A	
Спуститися від A до C:		

Спуститися від C до E:	E	E листовий вузол
Відвідати E;	E	
Відвідати C;	C	Готово!

Вузли відвідуються в порядку B D A E C, Рекурсивна функція спочатку спускається по лівому дереву [t-> Left ()], а потім відвідує вузол. Другий крок рекурсії спускається по правому дереву [t-> Right ()].

```
// Симетричне рекурсивне проходження вузлів дерева
template <class T>
void Inorder (TreeNode <T> * t, void visit (T & item))
{
// Рекурсивне проходження завершується на порожньому піддереві
if (t! - NULL)
{
     Inorder (t-> Left (), visit); // спуститися по лівому піддереві
     visit (t-> data); // відвідати вузол
     Inorder (t-> Right (), visit); // спуститися по правому піддереві
}
}
```

Функція проходження міститься у файлі treescan.h.

При зворотному проходженні ВТ відвідування вузла відкладається до тих пір, поки не будуть рекурсивно пройдені обидва його піддерева. Порядок операцій дає так зване LRN (left, right, node) сканування.

1. Проходження лівого піддерева.
2. Проходження правого піддерева.
3. Відвідування вузла.

При зворотному проходженні дерева Tree_0 вузли відвідуються в порядку D B E C A.

Дія	Друк	Зауваження
Спуститися від A до B:		Лівий син вузла B дорівнює NULL
Спуститися від B до D:		D - листовий вузол
Відвідати D;	D	Усі сини вузла B пройдені
Відвідати B;	B	Ліве піддерево вузла A пройдено
Спуститися від A до C:		
Спуститися від C до E:		E - листовий вузол
Відвідати E;	E	Лівий син вузла C
Відвідати C;	C	Правий син вузла A
Відвідати корінь A:	A	Готово!

Функція сканує дерево знизу вгору. Ми спускаємося вниз по лівому дереву [t-> Left ()], а потім вниз по правому [t-> Right ()]. Останньою операцією є відвідування вузла.

```
// Зворотне рекурсивне проходження вузлів дерева
template <class T>
void Postorder (TreeNode <T> * t, void visit (T & item))
{
// Рекурсивне проходження завершується на порожньому поддереві
if (t!= NULL)
{
     Postorder (t-> Left (), visit); // спуститися по лівому піддереві
     Postorder (t-> Right (), visit); // спуститися по правому
     піддереві
     visit (t-> data); // відвідати вузол
}
} // Функція проходження містяться у файлі treescan.h.
```

Прямий метод проходження визначається відвідуванням вузла і подальшим проходженням спочатку лівого, а потім правого піддерев (NLR).

Питання загального характеру будови дерев, висвітлені у наведених матеріалах. Зокрема розглянуто питання програмного формування бінарних дерев та деякі рекурсивні методи їх обробки. Представлені, важливі для подальшого виконання операцій на деревах та алгоритми проходження бінарних дерев.

5.2. Обробка дерев

На рекурсивних алгоритмах проходження дерев засновано багато алгоритмів їх обробки. Ці алгоритми забезпечують упорядкований доступ до вузлів, копіювання, вставки або пошук вузлів дерева. При викладені лекції використано матеріали розділів 11.3 – 11.5 роботи [1]. Мета лекції полягає в ознайомленні з прийомами і організацією програм для виконання алгоритмів обробки дерев.

Обробка дерев може бути пов’язана з різними задачами: пошуку певного вузла або вузлів, наприклад, листових, знаходження відповідних вузлових шляхів тощо. Розглянемо спочатку важливу задачу копіювання дерев. Ця задача є підготовчою для програміста перед тим, як почати проектування класу дерев. Тут використовується функція CopyTree, яка міститься у файлі treelib.h середовища програмування C++.

Функція CopyTree використовує для відвідування вузлів зворотній метод проходження дерева, розглянутий у попередньому пароаграфі. Цей метод гарантує спуск по дереву на максимальну глибину та подальше відвідування для створення вузла нового дерева. Функція CopyTree будує нове дерево знизу вгору. Спочатку створюються сини, а потім вони приєднуються до своїх батьків, після того як останні будуть створені. Розглянемо на прикладі BT

дерева Tree_0 попереднього параграфу, порядок операцій, який при цьому необхідно виконати.

1. d = GetTreeNode(‘D’);
2. e = GetTreeNode(‘E’);
3. b = GetTreeNode(‘B’, NULL, d);
4. c = GetTreeNode(‘C’, e, NULL);
5. a = GetTreeNode(‘A’, b, c);
6. root = a;

За порядком операцій, спочатку створюється син ‘D’ ВТ дерева, який потім приєднується до свого батька ‘B’. На наступному кроці створюється вузол ‘E’ і приєднується до свого батька ‘C’ під час створення останнього. Нарешті, створюється корінь і приєднується до своїх синів ‘B’ і ‘C’.

Алгоритм копіювання дерева починає діяти з кореня і будує ліве піддерево вузла, а потім - праве. Після цього створюється новий вузол. Той же рекурсивний процес повторюється для кожного вузла. Відповідно вузлу t дерева-орігінала створюється новий вузол з вказівниками newlptr і newrptr.

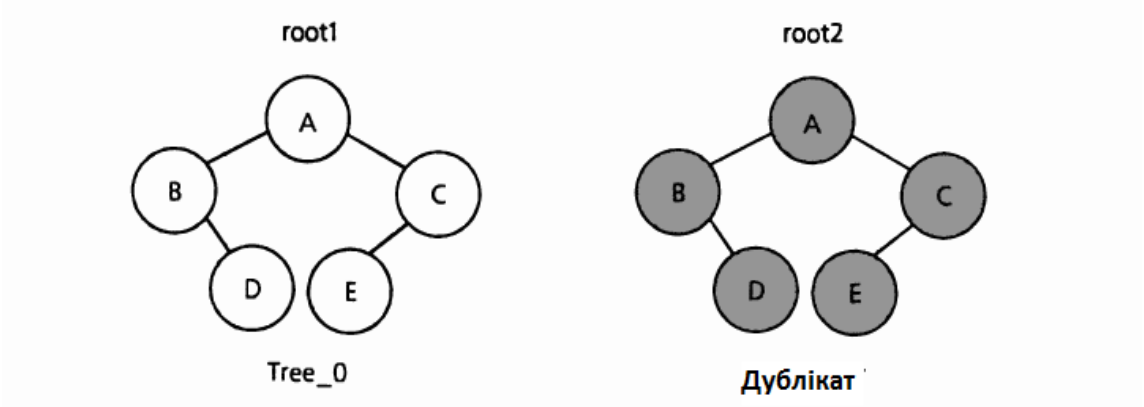
При зворотному методі проходження сини відвідуються раніше їх батьків. В результаті у новому дереві створюються піддерева, відповідні t-> Left () і t-> Right (). Сини приєднуються до своїх батьків під час їх створення.

```
newlptr = CopyTree (t-> Left ());
newrptr = CopyTree (t-> Right ());
// Створити батька та приєднати до нього його синів
newnode = GetTreeNode (t-> data, newlptr, newrptr);
```

Суть відвідування вузла t у дереві-оригіналі полягає у створенні нового вузла на дереві-дублікаті.

Символьне дерево Tree_0 є прикладом, котрий ілюструє рекурсивну функцію CopyTree. Припустимо, що головна процедура визначає корені rootl і root2 і створює дерево Tree_0. Функція CopyTree створює нове дерево з коренем root2. Простежимо алгоритм і проілюструємо процес створення п'яти вузлів на дереві-дублікаті.

```
TreeNode <char> * rootl, * root2; // оголосити два дерева
MakeCharTree (rootl, 0); // rootl вказує на Tree_0
root2 = CopyTree (rootl); // створити копію дерева Tree_0
```



Наведемо тепер схему алгоритму, за якою створюються вузли дерева-дубліката.

1. Пройти нащадків вузла A, починаючи з лівого піддерева з вузлом B і далі до вузла D, який є правим піддерево вузла B. Створити новий вузол D з полем даних (оригінала вузла D), і лівим і правим покажчиками, зі значеннями NULL.

2. Сини вузла B пройдені. Створити новий вузол B з даними і лівим вказівником, рівним NULL та правим вказівником, що вказує на вузол D.

3. Ліве піддерево вузла A пройдено, почати проходження його правого піддерева і дійти до вузла E. Створити новий вузол з даними з вузла E і вказівними полями, рівними NULL.

4. Після обробки вузла E перейти до його батька. Створити новий вузол з даними з оригінала C. У полі правого вказівника помістити NULL, а лівому вказівникові привласнити посилання на вузол сина E.

5. Останній крок виконується у вузлі A. Створити новий вузол з даними з оригінала вузла A і приєднати до нього сина B зліва та сина C зправа.

Копіювання дерева завершено.

Функція CopyTree повертає вказівник на новостворений вузол. Що дозволяє при потребі продовжити формування вузлів дерева. Крім того функція повертає корінь дерева програмі яка його викликає.

Розглянемо тепер програму, яка реалізує метод CopyTree.

```
// Створити дублікат дерева t і повернути корінь нового дерева
template <class T>
TreeNode <T> * CopyTree (TreeNode <T> * t)
{
// Змінна newnode вказує на новий вузол, створюваний
// за допомогою виклику GetTreeNode і приєднується в подальшому
// до нового дерева, вказівники newlptr і newrptr адресують синів
// нового вузла і передаються в якості параметрів в GetTreeNode
TreeNode <T> * newlptr, * newrptr, * newnode;
// Зупинити рекурсивне проходження при досягненні порожнього дерева
if (t ==NULL)
return NULL;
// CopyTree будує нове дерево в процесі проходження вузлів t.
// Функція CopyTree перевіряє наявність лівого сина, якщо він є,
// створюється його копія, в іншому випадку повертається NULL.
// CopyTree створює копію вузла за допомогою GetTreeNode і
// приєднує до нього копії синів.
if (t-> Left ()! = NULL)
      newrptr - CopyTree(t-> Left ());
else
      newlptr - NULL;
if (t-> Right ()! = NULL)
      newrptr - CopyTree(t-> Right ());
else
      newrptr = NULL;
```

```
// Побудувати нове дерево знизу вгору, спочатку створюючи
// двох синів, а потім їх батька
newnode=GetTreeNode(t->data, newlptr, newrptr);
// Повернути вказівник на новостворене дерево
return newnode;
}
```

Після створення копії дерева-оригінала виникає необхідність звільнення пам'яті, що можна зробити за допомогою функції DeleteTree, в якій застосовується зворотний метод проходження і використовується операція відвідування, котра викликає функцію FreeTreeNode для видалення вузла.

```
// Використати зворотний алгоритм і видалити вузол при його відвіданні
template <class T>
void DeleteTree (TreeNode <T> * t)
{
if (t! <<NULL)
{
     DeleteTree(t-> Left ());
     DeleteTree(t-> Right ());
     FreeTreeNode(t);
}
}
```

ВТ дерево може містити велику кількість даних і разом з тим забезпечувати швидкий їх пошук. Як відомо з попередніх матеріалів посібника, складність алгоритму пошуку у лінійних структурах становить $O(n)$, що неефективно для великих колекцій. У загальному випадку деревовидні структури, за бінарним деревом пошуку (binary search tree) BST, забезпечують більшу ефективність пошуку порядку $O(log_2 n)$.

BST будується за правилом так що, для кожного вузла значення даних в лівому піддереві менше, ніж у батьківському вузлі, а в правому піддереві – більше або є рівним значенню предка. Наприклад, пошук числа 37 (ключ) у наведеному BST вимагає чотирьох порівнянь, починаючи з кореня.

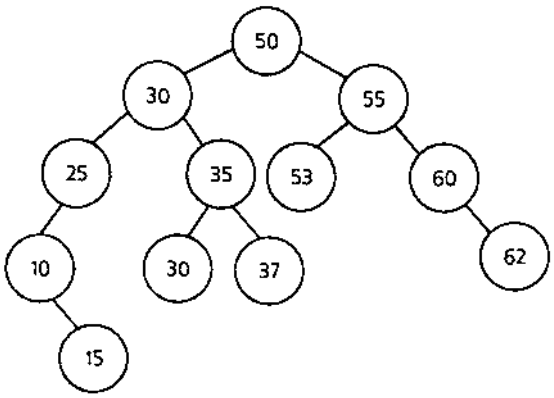


Рис. 5.1. Дерево бінарного пошуку

Поточний вузол	Дія
Корінь = 50	Порівняти ключ 37 і 50
	оскільки 37 <50, перейти в ліве піддерево

Вершина = 30	Порівняти ключ 37 і 30
	оскільки 37> 30, перейти в праве піддерево
Вершина = 35	Порівняти ключ 37 і 35
	оскільки 37> 35, перейти в праве піддерево
Вершина = 37	Порівняти ключ 37 і 37. Вершина знайдена.

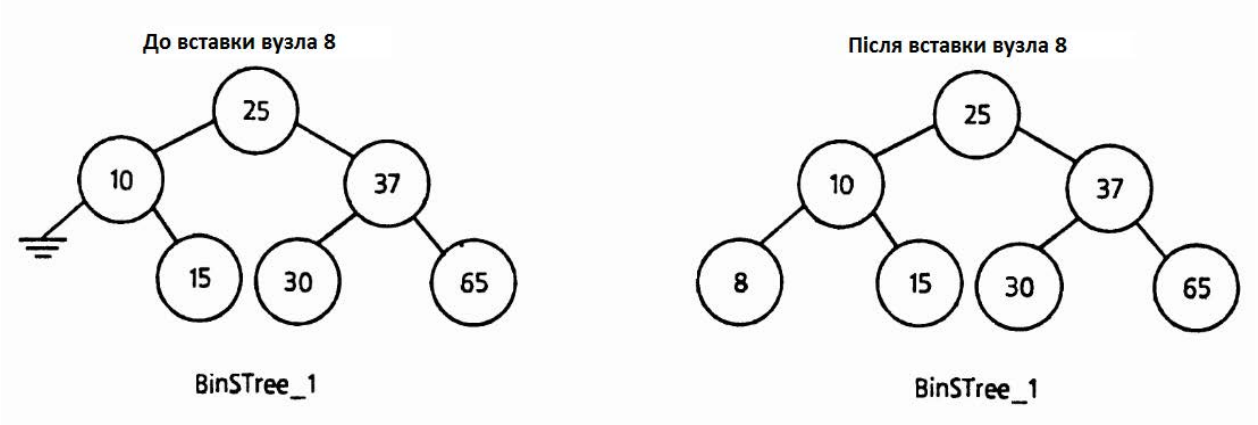


Рис. 5.2. Вставка до бінарного дерева

Ключем може бути все поле даних або його частина.

BST дерево, крім операції пошуку, повинно допускати операції включення (вставки) і видалення вершин. Причому операція включення повинна правильно вставляти новий елемент у дерево. Наприклад, включення вузла 8 в дерево BinSTree_1 починається з кореневого вузла 25, від якого за ланцюжком порівнянь (8 <25, 8 <10) отримаємо, що число 8 повинно бути лівим піддеревом вузла 10.

У зв'язаному списку операція видалення від'єднує вузол і з'єднує його попередника з наступним вузлом. На BST дереві подібна операція набагато складніше, тому що вузол може порушити впорядкування елементів дерева. Розглянемо операцію видалення кореня 25 з BinSTree_l. У результаті з'являються два роз'єднаних піддерева, яким потрібен новий корінь.

На перший погляд напрошується рішення вибрати сина вузла 25 -скажімо, 37 - і замінити його батька. Однак це просте рішення невдале, так як деякі вузли виявляються не з того боку кореня.

На перший погляд напрошується рішення вибрати сина вузла 25 -скажімо, 37 - і замінити його батька. Однак це просте рішення невдале, так як деякі вузли виявляються не з того боку кореня.

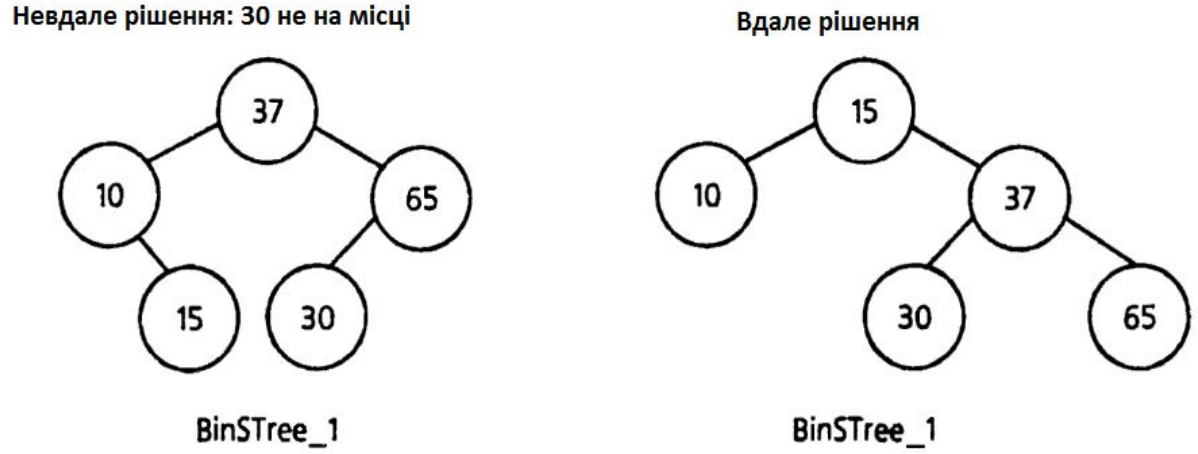


Рис. 5.3. Видалення з бінарного дерева

Оскільки дане дерево відносно невелике, можливо встановити, що числа 15 і 30 є допустимими замінами кореневого вузла дерева BST_1.

Розглянемо специфікацію класу BinSTree

Об’явлення. Наводиться реалізація BST у вигляді класу з динамічними списковими структурами. Цей клас містить стандартний деструктор, конструктор копіювання і перевантажені оператори присвоєння, котрі дозволяють ініціювати об’єкти, що відіграють роль операторів присвоєння. Деструктор відповідає за очищення списку, після закриття області дії об’єкту. Деструктор і оператори присвоєння разом з методом ClearList викликають закритий метод DeleteTree. Клас також містить закритий метод CopyTree, який використовується конструктором копіювання та операторі перевантаження.

```
#include <iostream.h>
#include <stdlib.h>
#include "treenode.h"
template <class T>
class BinSTree
{
protected: // потрібна для наслідування
// вказівники встановлюються на корінь і на поточний вузол
TreeNode <T> * root;
TreeNode <T> * current;
// Число елементів дерева
int size;
// Розподіл або звільнення пам'яті
TreeNode <T> * GetTreeNode (const T & item,
     TreeNode <T> * lptr, TreeNode <T> * rptr);
void FreeTreeNode (TreeNode <T> * p);
// Використовується конструктором копіювання і оператором присвоєння
```

```
void DeleteTree (TreeNode <T> * t);
// Використовується деструктором, оператором присвоювання
// і функцією ClearList
TreeNode <T> * FindNode (const T & item,
     TreeNode <T> * & parent) const;
public:
// Конструктор і деструктор
BinSTree (void);
BinSTree (const BinSTree <T> & tree);
~BinSTree (void);
// Оператор присвоєння
BinSTree <T> & operator = (const BinSTree <T> & rhs);
int Find (T & item); // Стандартні методи обробки списків
void Insert (const T & item);
void Delete (const T & item);
void ClearList (void);
int ListEmpty (void) const;
int ListSize (void) const;
// Методи, специфічні для дерев
void Update (const T & item);
TreeNode <T> * GetRoot (void) const;
}
```

Цей клас має захищені дані. Захищений доступ функціонально еквівалентний закритому доступу для даного класу. Змінна root вказує на кореневий вузол дерева. Вказівник current посилається на точку останньої зміни в списку. Метод Find заносить в current посилання на вузол, значення якого співпав з елементом даних.

Стандартні операції обробки списків використовують ті ж імена і параметри, що і в класі SeqList.

Клас BinSTree містить дві операції, специфічні для дерев. Метод Update присвоює новий елемент даних поточного вузла або включає в дерево новий елемент, якщо той не співпадає з даними в поточній позиції. Метод GetRoot надає доступ до кореня дерева. Маючи корінь дерева, користувач отримує доступ до бібліотечних функцій з treelib.h, treescan.h і treeprint.h.

Розглянуто алгоритми обробки дерев з застосуванням прийомів проходження, зокрема копіювання, вставки, пошук вузлів дерева тощо. Наведена ефективна реалізація цих алгоритмів з застосуванням бібліотеки середовища C++.

5.3. Пірамідальні дерева і їх застосування

У третьому розділі обговорювалися питання сортування даних на лінійних структурах, зокрема ефективний метод швидкого сортування. Наступний параграф матеріалів знайомить нас ще з одним ефективним методом

сортування даних на пірамідальних структурах закінчених дерев, які моделюють масиви даних. Метою параграфу є знайомство з структурами пірамідальних дерев, їх формуванням на масивах даних, операцій на пірамідах, застосуванням пірамідальних дерев до сортування списків і особливостями розробки відповідних алгоритмів.

Раніше було розглянуто класичне представлення дерев через вузлові конструкції з трьома полями, одним полем даних і двома полями покажчиків необхідних для зв'язку з іншими вузлами дерева. Але для ВТ дерева є можливість організувати вузли за допомогою масиву, зберігаючи дані відповідного поля вузла у масиві, а поля покажчиків зв'язати індексами масиву. При цьому утворюється закінчене бінарне дерево

На рис. 5.4 показано закінчене бінарне дерево для масиву A

int A[10] = {5, 1, 3, 9, 6, 2, 4, 7, 0, 8}

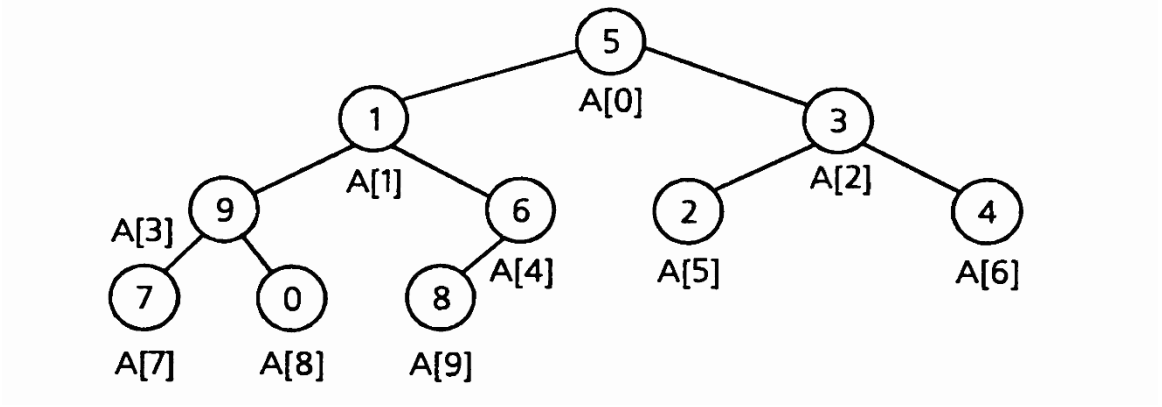


Рис. 5.4. Закінчене бінарне дерево масиву A

Отже під таким кутом розглянутий масив A забезпечує природне уявлення дерева але виникає проблема, пов'язана з відсутніми вузлами, яким повинні відповідати невикористані елементи масиву.

Переваги вказаних масивами дерев виявляються тоді, коли потрібен прямий доступ до вузлів. Індекси, що ідентифікують синів і батька даного вузла, обчислюються просто. Для кожного вузла A[i] в масиві розміру n, індекси його синів обчислюється за правилами: 1) індекс лівого сина $l = 2i + 1$, невизначений при $2i \geq n - 1$; 2) індекс правого сина $p = 2i + 2$, невизначений при $2i \geq n - 2$.

У таблиці 5.1 представлено дерево, зображене на рис. 3.1 для кожного рівня якого вказані батьківські і синовні вузли.

Піднімаючись від синів до батьків, помічаємо, що батьком вузлів A[3] і A[4] є вузол A[1] та вузлам A[5] і A[6] відповідає вузол батька A[2] і т.д. Отож загальна формула для обчислення індексу батьківського вузла A[i] наступна: індекс батька дорівнює $\frac{(i-1)}{2}$ і невизначений при $i = 0$.

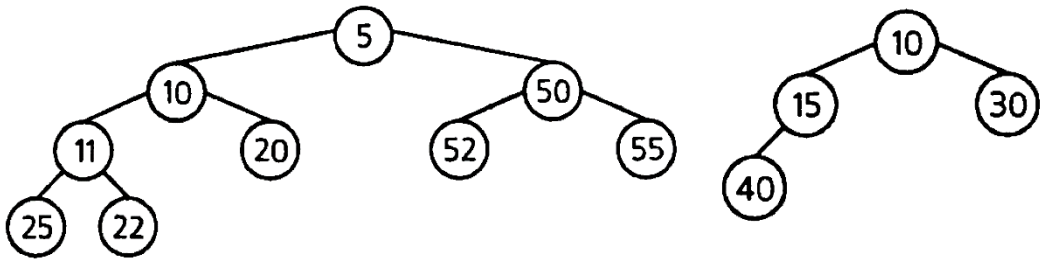
Таблиця 5.1.

Рівень	Батько	Значення	Лівий син	Правий син
0	0	A[0] = 5	1	2
1	1	A[1] = 1	3	4
	2	A[2] = 3	5	6
2	3	A[3] = 9	7	8
	4	A[4] = 6	9	10=NULL
	5	A[5] = 2	11=NULL	12=NULL
3	6	A[6] = 4	13=NULL	14=NULL
	7	A[7] = 7	-	-
	8	A[8] = 0	-	-
	9	A[9] = 8	-	-

Послідовне проходження дерева можна виконувати вниз від «батьків до синів» або вгору « від синів до батьків».

Представленні масивами дерева знаходять застосування в додатках з пірамідами (heaps), що мають велике значення, та які є закінченими ВТН деревами, що мають впорядкування вузлів за рівнями. В максимальній піраміді (maximum heap) батьківський вузол більше або дорівнює кожному з своїх синів. У мінімальній піраміді (minimum heap) батьківський вузол менше або дорівнює кожному зі своїх синів, такого типу піраміди зображені на рис. 5.5. У подальшому розглядаються тільки мінімальні піраміди.

Піраміда є списком, який зберігає деякий набір даних у вигляді ВТН дерева. Як абстрактна списова структура піраміда допускає додавання і видалення вузлів. Процес включення не має на увазі, що новий вузол займає конкретне місце, а лише вимагає, щоб підтримувалося пірамідальне впорядкування. Однак при видаленні зі списку викидається найменший елемент (корінь). Піраміда використовується в тих додатках, де клієнту потрібно прямий доступ до мінімального значення даних. Як список, піраміда не має операції пошуку і здійснює прямий доступ до мінімального вузла в режимі "тільки читання". Всі алгоритми обробки пірамід самі повинні оновлювати дерево і підтримувати пірамідальне впорядкування.



(A) (B)

Рис. 5.5. Мінімальні піраміди

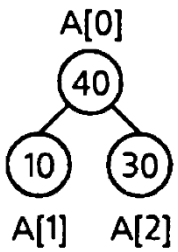
Піраміда є дуже ефективною структурою управління списками, яка користується перевагами закінченого бінарного дерева. При кожній операції

вставки або видаленні вузла піраміда поновлює своє впорядкування за допомогою сканування тільки коротких шляхів від кореня вниз до листя дерева. Важливими додатками ВТН пірамід є черги пріоритетів та сортування даних списку.

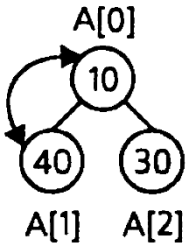
Замість того щоб використовувати довільні алгоритми сортування, можна включити елементи списку в ВТН піраміду і відсортувати їх, постійно видаляючи кореневий вузол. Це дає надзвичайно швидкий алгоритм сортування.

Обговоримо внутрішню організацію ВТН піраміди в класі Heap. Алгоритми включення і виключення вузлів представляються реалізаціями методів Insert і Delete. Розглянемо спочатку схематично операції з деревом (В) рис. 5.6.

Операція *створення піраміди* (мінімальної).на вхідному спискові (40, 10, 30). Спочатку на даних масиву створюється ВТ дерево, яке в загалі не утворює піраміду. Піраміда створюється після упорядкування даних 10 і 40.



Дерево за вхідним списком



Піраміда

Операція *вставки у піраміду* передбачає додавання нового вузла в кінець списку, а потім ВТ дерево реорганізується з метою відновлення пірамідальної ВТН структури. Наприклад, для додавання в список числа 15 проводяться наступні дії: 1) записати число 15 у елемент А[3]; 2) упорядкувати створене дерево.

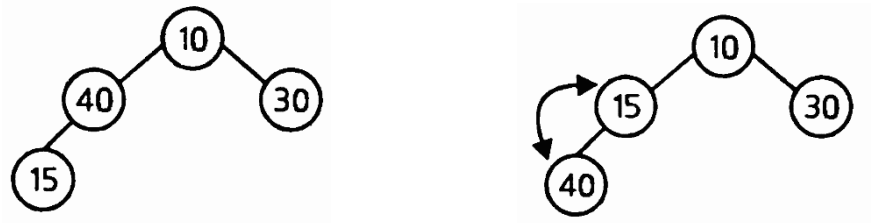


Рис. 5.6. Операція вставки у піраміду

Під операцією *видалення вузла з ВТН* розуміється видалєння коріня дерева А[0]. Потім виконуються дії: 1) на звільнене місце вставляється останній елемент списку; 2) дерево реорганізується в ВТН структуру. Так для останньої ВТН структури маємо послідовність дій:

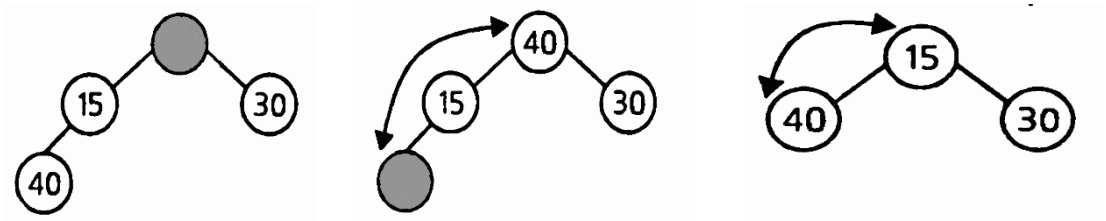


Рис. 5.7. Операція видалення з піраміди

Як список, клас Heap повинен мати операції вставки, видалення і операції, які повертають інформацію про стан об'єкта, наприклад, розмір списку.

Розглянемо специфікацію класу Heap.

Об'ява

```
#include <iostream.h>
#include <stdlib.h>

template <class T>
class Heap
{
private:
// hlist вказує на масив, котрий може бути створений динамічно
// конструктором (inArray == 0) або переданий як параметр (inArray ==
1)
     T *hlist;
     int inArray;
     // максимальний та поточний розміри піраміди
     int maxheapsize;
     int heapsize; // визначає кінець списку
     void error(char errmsg[]); // функція повідомляє про помилку
     // утиліти відновлення пірамідальної структури
     void FilterDown(int i);
     void FilterUp(int i);
public: // конструктори та деструктор
     Heap (int maxsize); // створити порожню піраміду
     Heap (T arr[], int n); // перетворити arr у піраміду
     Heap (const Heap<T>& H); // конструктор копії
     ~Heap(void); // деструктор
     // перевантажені оператори: "=", "[]", "т*"
     Heap<T> operator= (const Heap<T>& rhs);
     const T& operator[] (int i);
     // методи обробки списків
     int ListSize(void) const;
     int ListEmpty(void) const;
     int ListFull(void) const;
     void Insert(const T& item);
     T Delete(void);
     void ClearList(void);
};
```

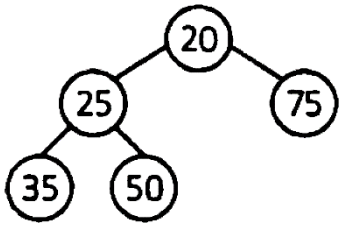
Скорочено розглянемо описання цієї програми.

Перший конструктор приймає параметр size і використовує його для динамічного виділення пам'яті під масив. Другий конструктор приймає в якості параметра масив і перетворює його в піраміду. Перевантажений оператор індексу "[]" дозволяє клієнту звертатися до об'єкту типу піраміди як до масиву. Оскільки цей оператор повертає посилання на константу, доступ здійснюється лише в режимі "тільки читання". Методи ListEmpty, ListSize і ListFull повертають інформацію про поточний стан піраміди.

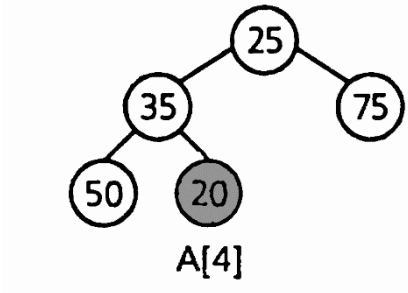
Перейдемо до розгляду застосування ВТН структури в задачі сортування.

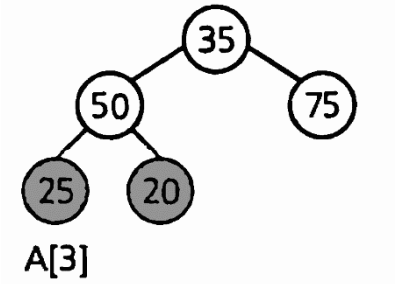
Для здійснення пірамідального сортування масиву А необхідно, спочатку, оголосити об'єкт типу Heap з масивом А і його dimA = n як параметрами. Конструктор перетворить масив А в ВТН піраміду. Сортування здійснюється методом послідовного виключення екземпляра А[0] і включенням його в послідовність (А[n-1], А[n-2], ..., А[1]). Нагадаємо, що після виключення екземпляра з піраміди елемент, який був до цього хвостовим, заміщується кореневим і з цього моменту більше не є частиною піраміди. Тепер мається можливість скопіювати видалений елемент в цю позицію. Отже у процесі пірамідального сортування чергові найменші елементи видаляються і послідовно запам'ятовуються в хвостовій частині масиву. Таким чином, масив А сортується за збуванням його екземплярів.

Наведемо приклад пірамідального сортування масиву А: Нехай масив А є int A[] = {50, 20, 75, 35, 25}, тоді ВТН структура буде мати вигляд:

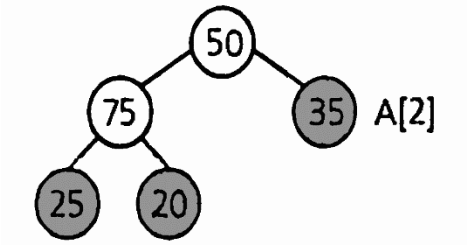


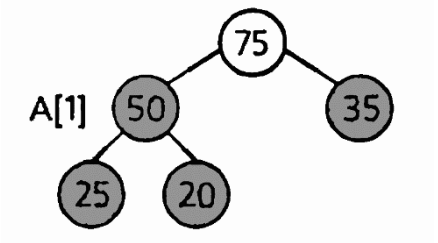
Видалити 20 та запам'ятати в А [4], видалити 25 і запам'ятати в А [3]





Знову видалити 35 і запам'ятати в А [2], видалити 50 та запам'ятати в А [1]





Оскільки зостався єдиний елемент 75 і він є коренем, тому масив відсортований: A = {75, 50, 35, 25, 20}.

Алгоритм сортування використовує той факт, що найменший елемент піраміди знаходиться в корені (індекс 0) і що метод Delete повертає це значення. Реалізація алгоритму пірамідального сортування виконується функцією HeapSort, знаходиться у файлі heapsort.h.

```
// Функція HeapSort
#include  "heap.h"  // клас Heap
// сортування масиву A за збуванням
template <class T>
void HeapSort (T A[], int n)
{
     Heap<T> H(A, n);  // конструктор, котрий перетворює A в піраміду
     T elt;
     for (int i=n-l; i>=l; i–)//цикл заповнення елементів A[n-1] A[1]
          {
      // виключити найменший елемент з піраміди та запам’ятати його в
A[i]
              elt = H.DeleteO ;  A[i] = elt;
           }
}
```

Пірамідальне сортування має ефективність $O(n\,log_2\,n)$. Причому сортування не потребує додатної пам’яті, тому що знаходиться «на місті».

Щоб при сортуванні помістити вузол в правильну позицію, операція видалення Delete використовує метод FilterDown. Ця функція отримує в якості параметра індекс i, з якого починається сканування. При видаленні метод FilterDown викликається з параметром 0, так як значення, шо заміщається копіюється з останнього елемента піраміди в її корінь. Метод FilterDown використовується також конструктором для побудови піраміди.

```
// утиліта для відновлення піраміди, починаючи з індексу i
template <class T>
void Heap<T>::FilterDown (int i)
{
int currentpos, childpos;
T target; // почати з вузла i та присвоїти його значення змінній
target
currentpos = i;
target = hlist[i];
childpos = 2 * i + 1; // індекс лівого сина і рух вниз
while (childpos < heapsize) // поки не кінець списку
{
// індекс правого сина =  childpos+1. присвоїти змінній childpos < з
синів
     if ((childpos+1 < heapsize) &&
         (hlist[childpos+1] <= hlist[childpos]))
     childpos = childpos + 1; // якщо батько є менший сина, вихід
     if (target <= hlist[childpos])  break;
```

```
      else
      { // перемістити значення меншого сина у батьківський вузол.
            hlist[currentpos] = hlist[childpos];
            currentpos = childpos; // відновити індекси та продовжити
            childpos = 2 * currentpos + 1;
      }
} // помістити target у позицію що тільки що стала незайнятою
hlist{currentpos] = target;
}
```

Метод Delete копіює значення з кореневого вузла в тимчасову змінну, та заміщує корінь останнім елементом піраміди і heapsize зменшується на 1.

Розглянуті ознайомчі питання відносно особливих конструкцій бінарних дерев, свідчать про те, що дерева можуть мати більш складну структуру, завдяки чому прискорюється їх обробка. Так пірамідальні дерева дозволяють обробляти дані зі швидкостями близькими до методу швидкого сортування. Матеріали параграфу знайомлять здобувача з алгоритмами формування пірамід, їх операційною обробкою та методологією пошуку і сортування даних за допомогою ВТН структур.

РОЗДІЛ 6. AVL-ДЕРЕВА І ГРАФИ

Спіціальним чином сформовані дерева обробляються швидше, одне з таких представлень AVL-дерево. Інша особливість представлення даних у вигляді графів потребує розгляду специфічних прийомів обходу дерев та пошуку даних. Ці питання обговорюються у двох останніх параграфах наведених матеріалах посібника, при цьому нагадуємо, що матеріали побудовані в основному на даних роботи [1].

6.1. Збалансовані дерева. Графи і їх представлення

ВТ дерева пошуку призначені для швидкого доступу до даних. В ідеалі дерево є розумно збалансованим і має висоту (максимальна довжина шляху дерева) близько $O(\log_2 n)$. Однак при деяких даних дерево може виявитися виродженим. Тоді висота його буде порядку O(n), і доступ до даних істотно сповільниться. У цьому параграфі розглядається модифікований клас AVL-дерев і її метою є знайомство з методикою побудови збалансованих дерев, методами пошуку, організацію програмних засобів. Також тут відбувається знайомство з узагальненням дерев графами і засобами їх представлення.

Під збалансованим (AVL) деревом розуміється таке дерево, для кожного вузла якого висоти його піддерев розрізняються не більше ніж на одиницю.

Методи вставки та видалення в класі AVL-дерев гарантують, що всі вузли залишаться збалансованими по висоті. На рис. 1.1 показані еквівалентні подання масиву даних бінарним деревом пошуку та AVL-деревом. Пара дерев представляє простий п’ятиелементний масив A = {1, 2, 3, 4, 5}, відсортований за зростанням. Бінарне дерево пошуку має висоту 5, в той час як висота AVL-дерева дорівнює 2. У загальному випадку висота збалансованого дерева не перевищує $O(\log_2 n)$. Таким чином, AVL-дерево є потужною структурою зберігання, що забезпечує швидкий доступ до даних.

Перейдемо до питання проектування класу AVL-дерева. Спочатку розробимо клас AVLTreeNode, а потім використовуємо об'єкти цього типу для конструювання класу AVLTree. При цьому особлива увага приділяється методам Insert і Delete тому, що вони вимагають ретельного проектування. Оскільки ці методи повинні гарантувати, що всі вузли нового дерева залишаються збалансованими по висоті.

Вузли AVL-дерева мають уявлення, схоже на бінарні дерева пошуку. Всі операції ідентичні, за винятком методів Insert і Delete, котрі повинні постійно відслідковувати співвідношення висот лівого і правого піддерев вузла. Для збереження цієї інформації проводиться розширення визначення об'єкта TreeNode, до якого включено поле balanceFactor (показник збалансованості), яке містить різницю висот правого і лівого піддерев.

balanceFactor = height (right subtree) - height (left subtree)

Якщо balanceFactor від'ємний, то вузол "переважує вліво", так як висота лівого піддерева більше, ніж висота правого піддерева. При позитивному balanceFactor вузол "переважує вправо". Збалансований за висоті вузол має balanceFactor = 0. Отже у AVL-дереві показник збалансованості повинен належати діапазону [-1, 1].

На рис. 6.1. зображені AVL-дерева з позначками -1, 0 та +1 на кожному вузлі, що показують відносний розмір лівого і правого піддерев.

-1: Висота лівого піддерева на 1 більше висоти правого піддерева.

0: Висоти обох піддерев однакові.

+1: Висота правого піддерева на 1 більше висоти лівого піддерева.

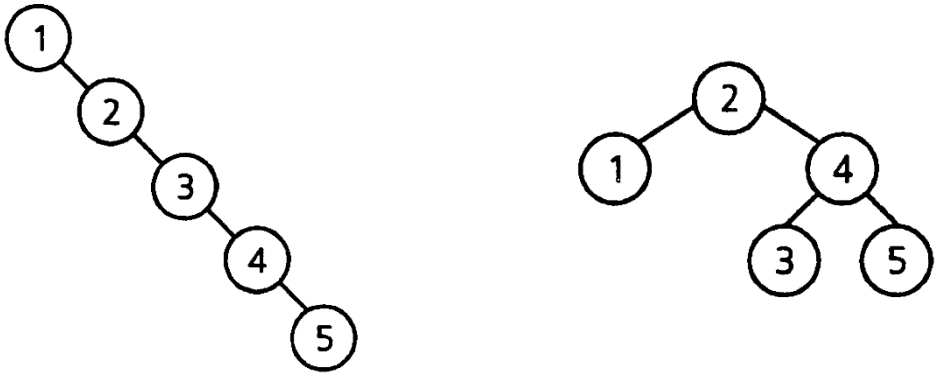


Рис. 6.1. Представлення масиву деревом пошуку та AVL-деревом.

Використовуючи властивості спадкування, можна утворити клас AVLTreeNode на базі класу TreeNode. Об’єкт типу AVLTreeNode успадковує поля з класу TreeNode і додає до них поле balanceFactor.

Дані-члени left і right класу TreeNode є захищеними, тому AVLTreeNode або інші похідні класи мають до них доступ. Клас AVLTreeNode і всі супроводжуючі його програми знаходяться у файлі avltree.h.

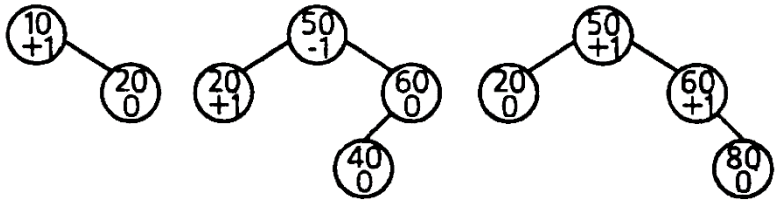


Рис. 6.2. AVL-дерева з показниками збалансованості

Наведемо специфікацію класу AVLTreeNode

Оголошення

```
// наследник класса TreeNode
template <class T>
class AVLTreeNode: public TreeNode<T>
{
```

```
          private:
               // допоміжний член класу
               int balanceFactor/
               // застосовуються методами класу VLTree і сприяють
               // запобіганню «переваження» вузлів
               AVLTreeNode<T>* & Left(void);
               AVLTreeNode<T>* & Right(void);
          public:
               // конструктор
               AVLTreeNode(const T& item, AVLTreeNode<T> *lptr =
NULL,
                                          AVLTreeNode<T> *rptr = NULL,
int balfac = 0);
               // повернути лівий/правий вказівник вузла типу
TreeNode,
// для вкзівника вузла типу AVLTreeNode; застосува приведення типів
             AVLTreeNode<T> *Left(void) const;
             AVLTreeNode<T> *Right(void) const;
        int GetBalanceFactor(void); // метод доступу до нового поля
даних
             // методи класу AVLTree повинні мати доступ до Left та
Right
             friend class AVLTree<T>;
};
```

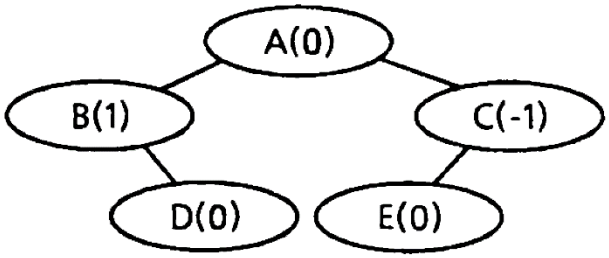
Наведемо деякі пояснення до програми

Покажчик даних balanceFactor є закритим, тому оновлювати його повинні тільки збалансовані операції включення і виключення. Параметри, передані конструктору, містять дані для базової структури типу TreeNode. По замовченню параметр balfac дорівнює 0. Доступ до полів вказівників здійснюється за допомогою методів Left і Right. Нові визначення цих методів обов'язкові, оскільки вони повертають вказівник на структуру AVLTreeNode.

Деструктор оголошується віртуальним. Оскільки клас AVLTree утворений на базі класу BinSTree, використовується деструктор базового класу і ClearList. Ці методи видаляють вузли за допомогою оператора delete. У кожному випадку вказівник посилається на об'єкт типу AVLTreeNode, а не TreeNode. Тому що деструктор базового класу TreeNode віртуальний, то при виклику delete використовується динамічне зв'язування і віддаляється об'єкт типу AVLTreeNode.

Розглянемо декілька прикладів використання AVL-дерев і їх методів.

```
  AVLTreeNode<char> *root; // корінь AVL-дерева, дерево, зображене
нижче.
// кожному вузлу дерева присвоюється вказівник збалансованості
```




```
void MakeAVLCharTree(AVLTreeNode<char>* &root)
{
          AVLTreeNode<char> *a, *b, *c, *d, *e;
          e = new AVLTreeNode<char>(/E', NULL, NULL, 0);
          d = new AVLTreeNode<char>('D', NULL, NULL, 0);
          c = new AVLTreeNode<char>('C, e, NULL, -1);
          b = new AVLTreeNode<char>('B', NULL, d, 1) ;
          a * new AVLTreeNode<char> (' A' , b, c, 0);
         root = a;
}
```

Реалізація класу AVLTreeNode. Конструктор класу AVLTreeNode викликає конструктор базового класу і ініціалізує balanceFactor.

```
// конструктор, ініціаліє balanceFactor та базовий клас.
  // початкові значення вказівників – нульові, вузол є лист, template
<class T>
AVLTreeNode<T>::AVLTreeNode (const T& item,
           AVLTreeNode<T> *lptr, AVLTreeNode<T> *rptr, int balfac):
        TreeNode<T>(item, lptr, rptr), balanceFactor(balfac)
{}
```

Методи Left і Right в класі AVLTreeNode спрощують доступ до полів даних. При спробі звернутися до лівого сина за допомогою базового методу Left повертається вказівник на об'єкт типу TreeNode. Щоб покажчик вказував на вузол збалансованого дерева, потрібно виконати перетворення типів.

Наприклад,

```
AVLTreeNode<T> *p, *q;
q = p->Left();                                                    //
недопустима операція
q = (AVLTreeNode<T> *)p->Left();        // необхідне приведення типу
```

Щоб уникнути постійного перетворення типом вказівників визначаються методи Left и Right для класу AVLTreeNode, які повертають вказівники на об'єкти типу AVLTreeNode.

```
template <class T>
AVLTreeNode<T>* AVLTreeNode::Left(void)
{
          return ((AVLTreeNode<T> *) left;
}
```

Оскільки AVL-дерево є розширеним бінарним деревом пошуку, клас AVLTree будується на базі класу BinSTree і є його спадкоємцем.

Методи Insert і Delete повинні підмінятися для виконання AVL-умови. Крім того, в похідному класі визначаються конструктор копіювання і перевантажений оператор привласнення.

Наведемо специфікацію класу AVLTree

Об’явлення

```
// Значення показників збалансованості вузла
const int leftheavy = -1;
const int balanced =1;
const int rightheavy =1;
// похідний клас пошукових дерев
template <class T>
class AVLTree: public BinSTree<T>
{
  private:
      // виділення пам'яті
     AVLTreeNode<T> *GetAVLTreeNode(const T& item,
                 AVLTreeNode<T> *lptr, AVLTreeNode<T> *rptr);
// використовується конструктором копіювання та оператором присвоєння
     AVLTreeNode<T> *CopyTree(AVLTreeNode<T> *t) ;
    // використовується методами Insert та Delete для відновлення
    // AVL-умов після операцій включення або виключення
    void SingleRotateLeft (AVLTreeNode<T>* &p);
    void SingleRotateRight (AVLTreeNode<T>* &p);
    void DoubleRotateLeft (AVLTreeNode<T>* &p);
    void DoubleRotateRight (AVLTreeNode<T>* &p);
    void UpdateLeftTree (AVLTreeNode<T>* &tree,
                                          int &reviseBalanceFactor);
    void UpdateRightTree (AVLTreeNode<T>* &tree,
                                             int
SreviseBalanceFactor);
     // спеціальні версії методів Insert и Delete
     void AVLInsert(AVLTreeNode<T>* &tree,
                       AVLTreeNode<T>* newNode, int
SreviseBalanceFactor);
     void AVLDelete(AVLTreeNode<T>* &tree,
                       AVLTreeNode<T>* newNode, int
sreviseBalanceFactor);
  public:  // конструктори
      AVLTree(void);
      AVLTree(const AVLTree<T>& tree);
      // оператор присвоєння
      AVLTree<T>& operator^ (const AVLTree<T>i tree);
      // стандартні методи обробки списків
      virtual void Insert(const T& item);
      virtual void Delete(const T& item);
};
```

Прокоментуємо текст програми. Константи leftheavy, balanced і rightheavy використовуються в операціях вставки і видалення для опису показника

збалансованості вузла. Метод GetAVLTreeNode керує виділенням пам'яті для класу. По замовченню показник balanceFactor нового вузла дорівнює нулю.

У цьому класі оновлюється визначення функції CopyTree для використання з конструктором копіювання і перевантаженим оператором присвоєння. Незважаючи на те, що алгоритм ідентичний алгоритму для функції CopyTree класу BinSTree, нова версія коректно створює розширені об'єкти типу AVLTreeNode при побудові нового дерева.

Функції AVLInsert і AVLDelete реалізують методи Insert і Delete, відповідно. Вони використовують закриті методи на зразок SingleRotateLeft. Відкриті методи Insert і Delete оголошені як віртуальні та підміняють відповідні функції базового класу. Інші операції успадковуються від класу BinSTree.

В прикладних питаннях програмної інженерії розглядаються більш складні дерева, наприклад *B – дерева*, вузли яких містять комбінації даних d_i та ключів $k_j, j > i$ і зв'язок між вузлами здійснюється через ключі на дані суміжних вузлів. Прикладом вузла B – дерева є:

$$k_1 \quad d_1 \quad k_2 \quad d_2 \quad k_3 \quad d_3 \quad k_4.$$

Узагальнюючою структурою дерева є граф (graph). Граф G визначається, як сукупність вершин $V = \{v_i\}$ і пар вершин $\mathrm{C} = \{(v_i, v_j)\}$: $G = \langle V, D \rangle$

Упорядкована (орієнгтована) пара вершин (v_i, v_j) графу зветься *дугою*, а неупорядкована – *ребром* (edges) і граф з відповідними парами звуться орієнтованими або орграфами та реберними.

Існує декілька форм структурного представлення графів. Найбільше поширена графічна форма (звідси походить і назва) графу (див рис.6.3):

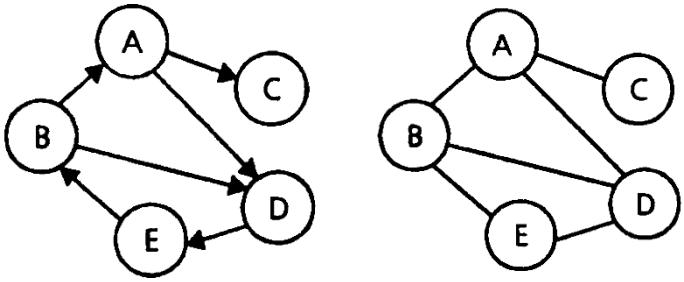


Рис. 6.3. Орграф і реберний графи

Графи зображені на рис. 6.3 можна задати табличними або матричними формами:

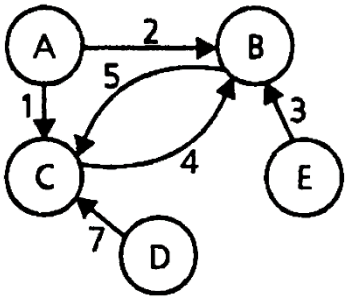
↵	A	B	C	D	E
A		1			
B					1
C	1				
D	1	1			
E				1	

,

	A	B	C	D	E
A		1	1	1	
B	1			1	1
C	1				
D	1	1			1
E		1		1	

Для орієнтованого графу (перша таблиця) читається у напрямку стрілки, а для реберного графу читається довільно.

Графи можуть бути навантаженими відповідною атрибутикою (вагою)



0	2	1	0	0
0	0	5	0	0
0	4	0	0	0
0	0	7	0	0
0	3	0	0	0

(A)

Рис. 6.4. Навантажений граф і його матрична форма

При обробці графів корисним є їх представлення у формі L -графів, у яких кожна вершина пов’язана із списком дуг, котрі виходять з вершини. Наприклад, для графа на рис.6.4. L -граф має вигляд:

$$\{A: (A \xrightarrow{2} B, A \xrightarrow{1} C), B: (B \xrightarrow{5} C), C: (C \xrightarrow{4} B), D: (D \xrightarrow{7} C), E: (E \xrightarrow{3} B)\}$$

.

Також, як і дерево, граф характеризується шляхами, довжинами шляхів, степенями вершин, та особистими характеристиками: *петлею графу* – (v, v), *контуром* – шлях, у якому початкова і кінцева вершини однакові та ін.

У параграфі наведено знайомство з пірамідальними структурами, котрі дають можливість здійснювати швидкий пошук даних і зберігати пам'ять оброблюваних даних. Представлено алгоритми формування пірамід на масивах і обробки пірамідальних дерев. Крім того розглянуто графове узагальнення дерев та засоби представлення графів.

6.2. Обробка графів

В матеріалах попереднього параграфу наведено визначення графу, його типи, способи задання та деякі характеристики графів. Метою наступного матеріалу є знайомство з алгоритмами представлення графів, методами їх обробки та пошуку найменших за вагою шляхів на графових мережах.

Наявність у графах петель, контурів, незв’язаності та іншого суттєво відрізняє, в деяких випадках, методику обробка графів від методики обробки дерев. Далі розглядається клас Graph, який використовує матричне представлення графу. Тут використовується статична модель графу, яка передбачає скінчену кількість вершин. Матричне подання спрощує реалізацію

класу і дозволяє зосередитися на цілому ряді алгоритмів обробки графів. Основними особливостями класу Graph є подання у класі Graph, методу ReadGraph і ряду пошукових алгоритмів, котрі здійснюють проходження вершин способами «в глибину» та «в ширину». Даний клас включає також ітератор списку вершин для використання клієнтами наведоною програмою.

Розглянемо специфікацію цього класу.

Оголошення

```
const int MaxGraphSize = n; // задати кількість вершин графу
template <class T>
class Graph
{
  private:
    // основні дані включають список вершин, матрицю суміжності
    // і поточний розмір (число вершин) графа
     SeqList<T> vertexList;
     int edge [MaxGraphSize];
     int graphsize;
    // методи для пошуку вершини і вказівки їх позиції в списку
     int FindVertex(SeqList<T> &L, const T& vertex);
     int GetVertexPos(const T& vertex);
  public: // конструктор
     Graph(void);
       // методи тестування графа
       int GraphEmpty(void) const;
       int GraphFull(void) const;
       // методи обробки даних
       int NumberOfVertices(void) const;
       int NumberOfEdges(void) const;
       int GetWeight(const T& vertexl, const T& vertex2);
       SeqList<T>& GetNeighbors(const T& vertex);
       // методи модифікації графа
       void InsertVertex(const T& vertex);
       void InsertEdge(const T& vertexl, const T& vertex2, int
weight);
       void DeleteVertex(const T& vertex);
       void DeleteEdge(const T& vertexl, const T& vertex2);
       // утиліти
       void ReadGraph(char *filename);
       int MinimumPath(const T& sVertex, const T& sVertex);
       SeqList<T>& DepthFirstSearch(const T& beginVertex);
        SeqList<T>& BreadthFirstSearch(const T& beginVertex);
        // ітератор для обходу вершин
        friend class VertexIterator<T>;
};
```

Тепер надамо деякий коментар до програми

Дані-члени класу містять вершини у вигляді послідовного списку, дуги або ребра, представлені матрицею суміжності, і змінну graphsize, яка є

лічильником вершин. Значення graphsize повертається функцією NumberOfVertices.

Утиліта FindVertex перевіряє наявність вершини у списку L і використовується в пошукових методах. Метод GetVertexPos обчислює позицію вершини vertex в vertexList. Ця позиція відповідає індексу рядка або стовпця в матриці суміжності.

Методу ReadGraph передається як параметр ім'я файлу з вхідним описом вершин і ребер графа. Клас Vertexlterator є похідним від класу SeqListlterator і дозволяє здійснювати проходження вершин. Ітератор спрощує додатки.

Перейдемо до методики проходження графів. Узагальненням прямого методу проходження для графів є пошук *спочатку в глибину* (depth-first), у якому початкова вершина передається як параметр і стає першою оброблюваної вершиною. По мірі просування вниз, суміжні вершини запам'ятовуються в стеку, з тим щоб можна було до них повернутися і продовжити пошук по іншому шляху у випадку, якщо ще залишилися необроблені вершини. Оброблені вершини утворюють список всіх вершин, досяжних з початкової вершини.

Характерне для дерев поперечне сканування починається з кореня, а обхід вузлів здійснюється рівень за рівнем зверху вниз. Аналогічна стратегія застосовується при пошуку *спочатку в ширину* (breadth-first**)** на графах.

Проходження починаються з деякої початкової вершини рівня і обробляються суміжні з нею вершини на цьому рівні. Потім сканування триває на наступному рівні суміжних вершин і т.д. При цьому оброблені суміжні вершини зберігаються у черзі.

Дамр представлення алгоритму пошуку методом у глибину на графові. Програма формує список оброблених вершин L та використовує для збереження суміжних вершин – стек S.

```
   // починаючи з початкової вершини, сформувати список вершин,
   // оброблєних в порядку обходу за методом в глибину
template <class T>
SeqList <T> & Graph<T>::DepthFirstSearch{const T& beginVertex)
{      // стек для тимчасового зберігання вершин, які очікують обробки
        Stack<T> S;
   // L – список пройдених вершин. adjL містить вершини, суміжні з
поточною.
   // L створюється динамічно, тому можна повернути його адресу
       SeqList<T> *L, adjL;
   // iteradjL – ітератор списку суміжних вершин
       SeqListIterator<T> iteradjL(adjL);
       T vertex;
   // ініціалізувати вихідний список і помістити початкову вершину в
стек
   L = new SeqList<T>;
   S.Push(beginVertex);
```

```
   // продовжувати сканування, поки не спорожніє стек
      while (IS.StackEmpty())
         {
   // видалити чергову вершину із стеку
            vertex = S.Pop() ;
   // перевірити її на наявність у списку L
            if (!FindVertex(*L, vertex))
              {
   // якщо vertex відсутня у списку, включити її в L і отримати всі
суміжні
                  (*L).Insert(vertex);
                  adjL = GetNeighbors(vertex);
   // встановити ітератор на поточний adjL
                  iteradjL.SetList(adjL);
   // сканувати список суміжних вершин і помістити в стек ті, які
відсутні у L
                  for (iteradjL.Reset(); !iteradjL.EndOfList()/
iteradjL.Next())
                    if (!FindVertex(*L, iteradjL.Data()))
                        S.Push(iteradjL.Data());
                }
         }
   // повернути вихідний список
            return *L;
}
```

Нагадаємо, що при пошуку в ширину для зберігання вершин графу використовується черга і ітераційний процес пошуку продовжується до тих пір, поки черга не спорожніє.

За схемою алгоритму пошуку у графі видалена вершина з черги перевіряється на її наявність у списку оброблених вершин. Якщо такої вершини немає в списку L, вона включається до списку. Одночасно отримуються всі суміжні з нею вершини і включаються в чергу ті з них, які відсутні у списку оброблених вершин L.

В розглянутих алгоритмах пошуку відвідування кожної вершини вимагає дій порядку O(n). При додаванні вершини в список оброблених вершин для виявлення суміжних з нею вершин перевіряється рядок матриці суміжності, отже тут складність також дорівнює O(n). Тому загальна складність є $O(n^2)$.

Кількість порівнянь, у разі матричного подання графу, не залежить від кількості його ребер і для кожного пошуку буде O(n + n) = O(n). Таким чином алгоритм пошуку має порядок $O(n^2)$.

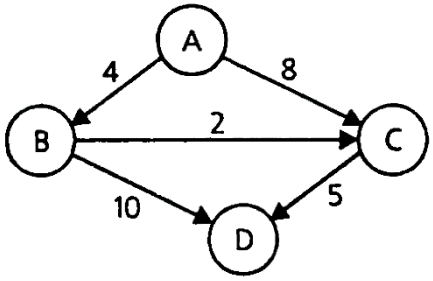
На кінець параграфу розглянемо задачу пошук мінімального шляху у графові.

Методи проходження графів в глибину і в ширину знаходять вершини, досяжні з початкової вершини, при цьому шлях від вершини до вершини не оптимізується. Але у багатьох додатках потрібно вибрати шлях з мінімальної

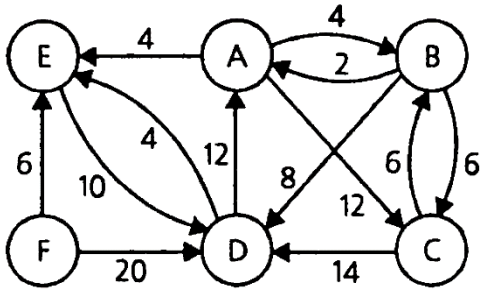
вагою, що складається з ваг ребер цього шляху. Для вирішення цієї задачі пропонується клас Pathlnfo. Об’єкт, породжений цим класом, описує навантажений шлях, між двома вершинами. Об'єкти типу Pathlnfo зберігаються в черзі пріоритетів, яка забезпечує прямий доступ до об'єкта з мінімальною вагою.

```
template <class T>
struct Pathlnfo
 {
        T startV, endV;
 };
template <class T>
int operator <= (const PathInfo<T>& a, const PathInfo<T>& b)
{
      return a.cost <= b.cost;
}
```

Так як між вершинами графа може існувати кілька різних шляхів, об'єкти типу Pathlnfo можуть відповідати одним і тим же вершинам, але мати шляхи з різні вагами. Наприклад, у наведеному графі між вершинами A і D існують три різно-вагові шляхи: (A,B,D)↦ 14, (A,C,D)↦ 13, і (A,B,C,D)↦ 11.



Для порівняння ваги у класі Pathlnfo визначений оператор "<=". Алгоритм перевіряє об'єкти типу Pathlnfo, в черзі пріоритетів, і вибирає об'єкт з мінімальною вагою. Визначення мінімального шляху між початковою (sVertex) і заключною (eVertex) вершинами ілюструється на графі.



Нехай на графові вершина A буде початковою, а D – заключною.

Процес пошуку починається з створення першого об'єкту типу Pathlnfo, що з'єднує початкову вершину саму з собою при нульовій початковій вазі і включенням цього об'єкту до черги пріоритетів: (A, A)↦ 0.

Для знаходження мінімального шляху використовується ітераційний процес, за яким видаляються об'єкти з черги пріоритетів. Якщо кінцева вершина в об'єкті є eVertex, то мінімальний шлях знайдено, вага якого знаходиться в полі cost. В іншому випадку проглядаються всі вершини, суміжні з поточної кінцевою вершиною endV, і до шляху, який починається з sVertex, додається ще одна дуга.

У нашій задачі розшукується мінімальний шлях від вершини A до D. Отже спочатку видаляється єдиний об'єкт Pathlnfo, в якому endV = A. Якби вершина A була заданою кінцевою вершиною eVertex, процес завершився б з нульової мінімальною вартістю. Оскільки вершина A не є eVertex, то вона заноситься до списку L, який містить всі вершини мінімального шляху. Суміжними (sectendV) з вершиною A є вершини B, C і E графу. Для кожної з них створюється об'єкт типу Pathlnfo, і всі ці об'єкти заносяться в чергу. Тому маємо об’єкти (A, B)↦ 4, (A, E)↦ 4, (A, C)↦ 12, які утворюють чергу.

На наступному кроці об’єкт (A, B)↦ 4 видаляється з черги і так як B є endV з вагою 4 та вона відсутня у списку L, тому B заноситься до списку L..

sectendV з вершиною B є вершини A, C і D. Вершина A вже в L, тому утворюються об’єкти типу Pathlnfo для вершин C і D. Потому результуюча черга містить чотири створені об’єкти, причому два з них закінчуються вершиною C. Шлях (A, C), визначений на першому кроці, має вагу 12. Шлях (A, B, C) має вагу 10 і об’єкт додано в чергу наступним. Отже маємо чергу

{(A, E)↦ 4, (A, C)↦ 12, (B, C)↦ 10, (B, D)↦ 12}.

Після видалення об'єкта (A, E)↦ 4 з черги створюється об’єкт (E, D)↦ 14, що дає наступну чергу {(B, C)↦ 10, (A, C)↦ 12, (B, D)↦ 12, (E, D)↦ 14}. При наступному видаленні об’єкту (B, C)↦ 10 , вершина C додається до списку L, так як 10 є мінімальною вагою на шляху від A до C.

Оскільки кінцевою вершиною мінімального шляху є D, очікується видалення об'єкта з endV = D. У вершини C є суміжні вершини B і D. Так як B вже оброблена, в чергу включається тільки об'єкт (C, D)↦ 24. І після видалення об'єкта (A, C)↦ 12 з черги він відкидається, так як C вже є в списку. Тепер черга має вигляд {(B, D)↦ 12, (E, D)↦ 14, (C, D)↦ 24}.

Видалення (B, D)↦ 12 з черги, дає мінімальну вагу 12 шляху від A до D.

Реалізація алгоритму пошуку шляху мінімальної ваги наведена нижче.

```
template <class T>
   int Graph<T>::MinimumPath(const T& sVertex, const T& eVertex)
{
    // черга пріоритетів об'єктами, які містять вагу шляхів з sVertex
   PQueue< PathInfo<T> > PQ(MaxGraphSize);
    // використовується при вставці або видаленні об'єктів Pathlnfo в
черзі
    PathInfo<T> pathData/
    // L – список вершин, досяжних з sVertex і вага, яких врахована
    // adjL – список вершин, суміжних з відвідуваною в даний момент
    // Для сканування adjL використовується ітератор adjLiter
```

```
    SeqList<T> L, adjL;    SeqListIterator<T> adjLiter(adjL);
    T sv, ev;    int mincost;
    // сформувати початковий елемент черги пріоритетів
    pathData.startV = sVertex;    pathData.endV = sVertex;
    // вага шляху з sVertex в sVertex дорівнює 0
    pathData.cost = 0;
    PQ.PQInsert(pathData);
    // обробляти вершини, поки не буде знайдений мінімальний шлях
    // до eVertex або поки не спорожніє черга
    while ( IPQ.PQEmptyO )
    {
   // видалити елемент черги і зберегти кінцеву вершину і вагу від
sVertex
        pathData = PQ.PQDelete();
        ev = pathData.endV;  mincost = pathData.cost;
    // якщо це eVertex, то мінімальний шлях від sVertex до eVertex
знайдений
        if (ev == eVertex)  break;
        // якщо кінцева вершина вже є в L, не розглядати її знову
        if (!FindVertex(L, ev))
           {
             // Включити ev до списку L
             L.Insert(ev);
  // знайти всі суміжні з ev вершини. для тих з них, яких немає в L і
створити
  // Pathlnfo з початковими вершинами, рівними ev, і включити їх у
чергу
           sv = ev;     adjL = GetNeighbors(sv);
   // новий список adjL сканується ітератором adjLiter
           adjLiter.SetList(adjL);
           for (adjLiter.Reset(); ladjLiter.EndOfList();
adjLiter.Next())
                {
                   ev = adjLiter.Data();
                   if (!FindVertex(L,ev))
                      {
   // створити новий елемент пріоритетною черги
    pathData.startV = sv;   pathData.endV = ev;
   // вага дорівнює поточній мінімальній вазі плюс вага переходу від
sv до ev
   pathData.cost = mincost + GetWeight (sv, ev);
PQ.PQInsert(pathData);
                      }
               }
         }
  }
if (ev « eVertex)
    return mincost;
    else
```

```
return -1; }
```

Наведені матеріали знайомлять читача з можливостями представлення алгоритмів обробки площинних графів, за якими проводиться проходження графів і пошуки необхідних вершин. Розглянуто технологію пошуку оптимальних шляхів на графовій мережі та наведені програмні алгоритми, які реалізують ці технології.

РОЗДІЛ 7. КОНТРОЛЬ ЗНАННЬ.

У цьому розділі пропонуються питання з контролю знань, які повинен набути читач під час знайомства з матеріалами посібника. Загалом ці питання стосуються загальних понять викладеного предмету по всім розділам посібника і частково окремим питанням, необхідним для глибшого розуміння і набуття навичок володіння структурами даних у їх використанні при проектуванні алгоритмів та програмуванні.

Матеріали розділу можуть бути використані викладачами для формування контрольних запитань з перевірки знань здобувачами під час поточного контролю і залишкових знань.

7.1. До розділу 1.

7.1.1. Контрольні запитання.

Надати відповіді на наступні питання:

1) алфавіти і їх призначення,
2) основні характеристики алфавітів,
3) поняття інформації,
4) інформаційні характеристики,
5) дані, види, представлення у ЕОМ,
6) поняття структури,
7) дані і їх структури,
8) поняття алгоритму та його властивості,
9) структури даних масиви і їх організація у програмуванні,
10) організація порядку даних у масивах,
11) формальне визначення алгоритму,
12) форми алгоритмів,
13) характеристики алгоритмів,
14) МНР алгоритми,
15) рекурсивні алгоритми, переваги і недоліки,
16) організація рекурсивних процесів і їх алгоритмічна реалізація,
17) поняття атрибуту та їх типів, атрибутивні алфавіти, формування атрибутивних алфавітів;
18) операції над атрибутивними алфавітами;
19) основні виміри інформації;
20) зміст і структура конструкції схеми програми;
21) виміри алгоритмів; одиниці вимірів;
22) застосування схеми Горнера для організації рекурсивного обчислювального процесу;
23) різновиди представлення алгоритмів;
24) статичне та динамічне представлення даних

25) похибки алгоритмів обчислення, причини та методології їх зменшення.

7.1.2. Тестові запитання

1. Які вирази а) аб+ба; б) ½+1,23; в) абвс-1; г) (а(б(с*д)+а)+б); д) 1*2*3; утворюють слова?
2. Яка довжина слова (а(б(с*д)+а)+б)?
3. Яку кількість біт інформації передає повідомлення «я студент», якщо символи повідомлення мають однакову ймовірність?
4. Яку структуру даних утворює послідовність $\{A[1], A[2], \ldots, A[10]\}$? Яку структуру даних утворює послідовність $\{A[1], B[2], \ldots, A[10]\}$? Що формує функція $\psi(s, *next) = \{s\}$?
5. Що визначає цикломатичне число алгоритмічної програми?
6. Що визначає показник $O(.)$алгоритму?
7. Чи можливо рекурсивно обчислити вираз $\sum_{k=0}^{n} \frac{x^k}{k!}$?
8. Чи утворюють вирази а) аб+ба; б) ½+1,23; в) абвс-1; г) (а(б(с*д)+а)+б); д) 1*2*3; алфавіт?
9. Що означає показник $O(n)$ алгоритму А?
10. Що означає показник $O(n^2)$ алгоритму А?
11. На якому алфавіті утворено вираз ½+1,23?
12. На якому алфавіті утворено слово (а(б(сд)+а)+б)?
13. Яка довжина слова (а(б(сд)+а)+б)?
14. Чи можливо рекурсивно обчислити функцію x^n? Де n – натуральне число.
15. Чи можливо рекурсивно обчислити вираз $n!$?
16. Яку кількість біт інформації передає повідомлення «я студент», якщо символи повідомлення мають однакову ймовірність?
17. Яку кількість біт інформації несе повідомлення «тут гарно навчають», якщо символи повідомлення мають однакову ймовірність?
18. Яку структуру даних формує функція $\psi(n, *next)$, якщо $n = 1..10$?
19. Яку структуру даних формує функція $\psi(n, *next) + \varphi(m, *next)$, якщо $n, m = 1..10$?
20. Яку структуру даних утворює вираз $\{A[1], \ldots, A[10]\} + \{B[1], \ldots, B[7]\}$?
21. Які структури даних утворює наступний вираз $(n, *next; k, *Null), (m, *next; j, *Null)$?
22. Яку кількість інформації передає код 10011101?

23. Яку кількість інформації передає трійковий код 102011101?

7.2. До розділу 2.

7.2.1. Контрольні запитання.

Визначити поняття:

1) автоматів їх будови і способи задання;
2) типи автоматів над пам'яттю;
3) стек, його структура і операції на ньому;
4) черга представлення, операції;
5) кругові (циклічні) черги, представлення і обробка;
6) списки, способи задання списків;
7) структура однозв'язаного списку;
8) двохзв'язані списки і структура задання;
9) способи формування списків
10) обробка списків;
11) операції на двозв'язаних списках;____
12) формування черг;
13) створення лінійних списків;
14) створення не лінійних списків;
15) операції на лінійних і не лінійних списках, їх особливості.

7.2.2. Тестові запитання

1. Яке значення має покажчик top, якщо стек порожній?
2. Яке значення має покажчик top після виконання операції push, якщо до цього стек був порожній?
3. Яка операція застосовується при внесені даних у чергу?
4. Які значення приймають вказівники черги, якщо черга порожня?
5. Який покажчик вказує на кількість елементів черги?
6. Яка типова позначка прийнята для означення вузла списку?
7. За якою базовою функцією організується вставка вузла у список?
8. Що означає запис «Node <T> *ptnext=NULL»?
9. .Яке ім'я має голова циклічного списку?
10. Що задає показник count у черзі?
11. Яка операція виконується у списку за функцією insertafter?
12. Що позначається ключовим словом Node у списку?
13. Що означає значення NULL у списку?
14. На що вказує ім'я header у циклічному списку?
15. Яка операція застосовується при вилученні даних з черги?

7.3. До розділу 3.

7.3.1. Контрольні запитання.

Надати відповіді на наступні питання:

1) сортування даних на масивах методом вибірки;
2) модифіковані методи вибірки;
3) оцінка складності алгоритмів вибірки;
4) бульбашковий метод сортування;
5) метод Шелла і переваги по відношенню до базового алгоритму;
6) метод шейкер-сортування і його переваги;
7) складність методів бульбашки;
8) метод швидкого сортування;
9) визначення складності алгоритму швидкого сортування;
10) метод порозрядного сортування;
11) оцінка складності порозрядного сортування, переваги і недоліки методу;
12) складність алгоритму сортування Шелла.
13) складність методу шейкер-сортування.
14) особливості сортування даних за методом вставки.
15) оцінка складності алгоритму сортування методом вставкм.
16) особливості сортування даних у масивах.
17) особливості сортування даних у списках.

7.3.2. Тестові завдання

1. Яку найменшу кількість обмінів потрібно виконати при сортуванні за збуванням даних $\{1,2,8,7,5\}$ методом вибірки?
2. Яку найменшу кількість обмінів потрібно виконати при сортуванні за збуванням даних $\{1,2,8,7,5\}$ методом швидкого сортування?
3. Яку найменшу кількість обмінів потрібно виконати при сортуванні за збуванням даних $\{1,2,8,7,5\}$ методом порозрядного сортування?
4. Чому дорівнює складність алгоритму за методом порозрядного сортування для відсортування даних $\{1,2,8,7,5\}$?
5. Яка кількість обмінів даних відбудеться при сортуванні по зростанню послідовності $\{1,3,9,7,6\}$ за шейкер-сортуванням?
6. Яка кількість обмінів даних відбудеться при сортуванні по зростанню послідовності $\{1,3,9,7,6,4\}$ за методом Шелла?
7. Яку максимальну складність має алгоритм сортування даних $\{a_1, a_2, a_3, a_4, a_5\}, a_i \in N$ методом бульбашки.

8. Яку максимальну складність має алгоритм сортування даних $\{a_1, a_2, a_3, a_4, a_5\}, a_i \in N$ методом вставки.
9. Яку максимальну складність має алгоритм сортування даних $\{a_1, a_2, a_3, a_4, a_5\}, a_i \in N$ методом Шелла.
10. Яку максимальну складність має алгоритм сортування даних $\{a_1, a_2, a_3, a_4\}, a_i \in N$ гібридним методом вибірки.
11. Чому дорівнює максимальна складність алгоритму за методом порозрядного сортування для відсортування даних п’яти цілих чисел?
12. Чому дорівнює максимальна складність алгоритму за методом шейкер-сортуванням для відсортування даних п’яти цілих чисел?
13. Яку найменшу кількість обмінів потрібно виконати при сортуванні за зростанням даних $\{1,2,8,7,5\}$ методом швидкого сортування?
14. Яку найменшу кількість обмінів потрібно виконати при сортуванні за зростанням даних $\{9,2,8,7,1\}$ методом порозрядного сортування?
15. Яку найменшу кількість обмінів потрібно виконати при сортуванні за зростанням даних $\{1,3,5,2,8,7,5\}$ методом швидкого

7.4. До розділу 4.

7.4.1. Контрольні запитання.

Надати відповіді на наступні питання:

1) лінійний спосіб пошуку даних у масиві, оцінка складності методу пошуку;
2) бінарний метод поуку і його алгоритм та ефективність;
3) алгоритм рекурсивного пошуку за методом бінарного пошуку;
4) особливості пошуку даних за ключем;
5) призначення хеш-функцій при пошуку даних;
6) колізії при хешуванні даних;
7) метод поділу для вирішення ключової колізії;
8) вирішення колізії за метод середини квадрата;
9) мультіплікативний метод вирішення колізій;
10) прийом відкритої адресації при вирішенні колізії;
11) метод ланцюжків у вирішенні колізії хешування;
12) файлова структура даних, організація та доступ до даних;
13) базові методи обробки файлів;
14) методи зовнішньої обробки (сортування і пошук) даних;___-
15) оцінка складності алгоритмів пошуку.

7.4.2. Тестові завдання

1. Яке значення повертає хеш-функція return Key % 10, якщо Key=31?

2. Яке значення повертає хеш-функція return hashf % 10, якщо hashf = key[0] + key[n-l] і Key=1331?
3. Яке значення повертає хеш-функція return (Key % 10+ Key % 11), якщо Key=31?
4. У якій послідовності в хеш-таблиці будуть розташовані ключі 45, 66 і 56, якщо хеш-функція return key % 11?
5. У якій послідовності в хеш-таблиці будуть розташовані ключі 10, 45, 63, якщо хеш-функція return key % 9?
6. Яке значення повертає хеш-функція return Key % 9 – Key % 10, якщо Key=52?
7. Яке значення повертає хеш-функція return Key % 10 + Key % 7, якщо Key=52?
8. Яке значення повертає хеш-функція return hashf % 10, якщо hashf = key[1] + key[n-l] і Key=13?
9. Якщо Хеш-функція f(data) = data % 7 + 1, тоді чому дорівнює значення функції f(79)?
10. Файл тримає цілі числові дані в діапазоні 7 .. 14. Яке найбільше значення має Хеш-функція f(data) = data % 7 на цьому діапазоні?

7.5. До розділу 5.

7.5.1. Контрольні запитання.

Надати відповіді на наступні питання:

1) структури даних дерева, визначення, типи;
2) характеристики дерев;
3) бінарні дерева їх структура і представлення;
4) прямий метод проходження бінарного дерева;
5) симетричний метод проходження дерева;
6) технологія проходження бінарного дерева зворотним методом;
7) застосування рекурсії у методах проходження дерев;
8) обробка дерев, алгоритм копіювання;
9) бінарні дерева пошуку;
10) обробка дерев , видалення і вставка вузлів;
11) пірамідальні дерева і їх будова.
12) створення пірамідального дерева;
13) операції обробки вузлів піраміди;
14) алгоритми пошуку і сортування даних пірамід.
15) бінарні дерева і нелінійні списки.

7.5.2. Тестові завдання

1. Чому дорівнює степінь листа бінарного дерева?
2. Степінь кореня бінарного дерева не перевищує.
3. Степінь будь-якої вершини бінарного дерева не перевищує.
4. Вершинами бінарного дерева є вузли зі скількома полями?
5. Скільки рівнів має мінімальне пірамідальне бінарне дерево утворене даними {10,12,3,7,2}?
6. Скільки рівнів має максимальне пірамідальне бінарне дерево утворене даними {20,10,12,3,7,2}?
7. За скільки кроків буде знайдено число 7 у максимальному пірамідальному бінарному дереві, утвореному на даних. {20,10,12,3,7,2}?
8. За скільки кроків буде знайдено число 7 у мінімальному пірамідальному бінарному дереві, утвореному на даних. {20,10,12,3,7,2}.
9. Пірамідальне бінарне дерево, утворене на даних. {20,10,12,3,7,2}, є збалансованим?.
10. Чому дорівнює максимальна довжина шляху пірамідального бінарного дерева, утвореного на даних. {20,10,12,3,7,2}?

7.6. До розділу 6.

7.6.1. Контрольні запитання.

Надати відповіді на наступні питання:

1) ВТ і AVL дерева;
2) проектування і створення збалансованих дерев;
3) показники збалансованості і їх використання при створені AVL дерева;
4) В-дерева, особливості будови вузлів і між вузлові зв’язки;
5) графи і їх характеристики;
6) способи задання графів, графічний спосіб;
7) табличні способи задання графів;
8) навантажені графи;
9) представлення графів L-графами;
10) алгоритми представлення графів і їх особливості;
11) метод проходження графу у глибину;
12) алгоритм проходження графу у ширину;
13) складність алгоритму пошуку на графах;
14) графові мережі їх будова;
15) алгоритм пошуку оптимального за вагою шляху мережі.

7.6.2. Тестові завдання

1. Яка мінімальна глибина орієнтованого графу заданого таблицею,

↵	a	b	c	d	e
a		1			
b	1		1		
c		1		1	
d			1		1
e				1	

якщо a початкова вершина графу та e – заключна.?

2. Яка мінімальна глибина орієнтованого графу заданого таблицею

↵	a	b	c	d	e
a		1			
b	1		1		
c				1	
d		1			1
e				1	

? Якщо a початкова вершина графу та e – заключна.

3. У якому порядку проходяться вершини графу, заданого таблицею

↵	0	1	2	3	4	5
0			1			
1	1					
2					1	
3	1	1				
4	1			1		
5		1		1		

методом у глибину, якщо завжди відбувається прохід до вершини з найменшим номером?

4. У якому порядку проходяться вершини графу, заданого таблицею

↵	0	1	2	3	4	5
0			1			
1	1					
2					1	
3	1	1				
4	1			1		
5		1		1		

методом у ширину, якщо завжди відбувається прохід до вершини з найменшим номером?

5. Які вершини графу, заданого таблицею

↵	0	1	2	3	4	5
0			1			
1	1					
2					1	
3	1	1				
4	1			1		
5		1		1		

мають найбільшу степінь?

6. Які вершини графу, заданого таблицею

↵	0	1	2	3	4	5
0			1			
1	1					
2					1	
3	1	1				
4	1			1		
5		1		1		

мають найменшу степінь?

7. Які слова найбільшої довжини утворює граф, заданий таблицею

↵	0	1	2	3	4	5
0						
1	a					
2				a	?	
3	b	a				
4	c			c		
5		b	d	d	b	

8. Які слова найменшої довжини утворює граф, заданий таблицею

↵	0	1	2	3	4	5
0						
1	a					
2				a	?	
3	b	a				
4	c			c		
5		b	d	d	b	

9. Знайти найкоротший шлях (за відстанню) між вершинами, c і d навантаженого відстанями графу, заданого таблицею

↵	a	b	c	d	e	f
a			1			
b	2					
c					3	.
d	7	3				
e	2			4		
f		5		4		

10. Знайти найкоротший шлях (за відстанню) між вершинами, c і f навантаженого відстанями графу, заданого таблицею

↵	a	b	c	d	e	f
a			1			
b	2					
c					3	.
d	7	3				
e	2			4		
f		5		4		

РОЗДІЛ 8. САМОСТІЙНА РОБОТА

У розділі запропоновані теми і завдання для самостійної роботи читачів з набуття практичних навичок вирішення задач і засвоєних ним матеріалів з формування та обробки різноманітних структур даних у програмуванні та реалізації розроблених програм на ЕОМ для подальшого їх тестування і аналізу за визначеними характеристиками складностями алгоритмів, виконання часу, тощо.

8.1. Обробка даних у масивах

8.1.1 Мета роботи:

1) засвоєння методології обробки і пошуку даних у масивах,
2) набуття навичок розробки алгоритмів сортування,
3) конструювання програм алгоритмів сортування,
4) реалізація алгоритмів на ЕОМ,
5) організація і проведення чисельних експериментів по визначенню часової ефективності методів сортування.

8.1.2. Порядок виконання роботи:

1) згенерувати випадкову послідовність і заповнити нею масив,
2) розробити схеми Насі-Шнайдермана алгоритмів сортування даних за зростанням або за збуванням для непарних значень,
3) написати тексти програм сортування методами вибору і бульбашки та відлагодити їх на ЕОМ,
4) розробити інтерфейс до чисельної реалізації сортування,
5) порівняти за часом методи сортування, зробити висновки щодо отриманих результатів,
6) виконати розрахунки складності розроблених алгоритмів,

8.2. Обробка даних з застосуванням списків, стеків і черг.

8.2.1 Мета роботи:

1) засвоєння методології створення списків та обробки і пошуку даних у спискових структурах даних,
2) набуття навичок розробки алгоритмів для роботи із списками, стеками і чергами,
3) конструювання програм алгоритмів операцій та сортування даних у списках і чергах,
4) реалізація алгоритмів на ЕОМ,
5) організація і проведення чисельних експериментів по визначенню часової ефективності методів обробки даних у списках і чергах.

8.2.2. Порядок виконання роботи:

1) згенерувати випадкову послідовність,
2) сформувати порожній список відповідного розміру,
3) заповнити список випадковими даними, які було створено;
4) виконати сортування даних списку методом вставки з використанням стеку для обміну даними;
5) розробити програму, яка реалізує цей метод;
6) виконати аналіз алгоритмів сортування за швидкістю;
7) створити чергу відповідно розміру,
8) наповнити чергу даними з масиву або послідовності,
9) виконати сортування даних за методом порозрядного сортування,
10) розробити програму, яка реалізує цей метод,
11) результати роботи програми представити у вигляді відсортованих даних черг и і часу реалізації сортування,
12) результати звести у таблицю і виконати аналіз отриманих даних,

8.3. Формування дерев і їх обробка

8.3.1 Мета роботи:

1) засвоєння методології з розробки і створення дерев,
2) набуття навичок розробки алгоритмів операцій над деревами,
3) засвоєння методології з організації пірамідального сортування даних,
4) конструювання програм алгоритмів обробки дерев,
5) реалізація алгоритмів на ЕОМ,
6) організація і проведення чисельних експериментів.

8.3.2. Порядок виконання роботи:

1) згенерувати випадкову послідовність

2) розробити структурну схему будови бінарного дерева за сформованою послідовністю випадкових даних,

3) створити програмну функцію формування порожнього вузлового бінарного дерева, на основі якого за схемою пункту 2) побудувати дерево випадкових даних,

4) розробити алгоритм формування пірамідального дерева,

5) виконати пошук даних на пірамідальному дереві,

6) виконати тестування програми;

7) реалізувати пункти 4) – 6) на ЕОМ;

8) виконати аналіз алгоритмів за їх складністю.

РОЗДІЛ 9. ДОДАТКИ

Раніше було приділено основну увагу класичним простим структурам даних але на практиці можуть виникати потреби у створенні більш складних СД. Тому у цьому розділі розглянуто деякі ускладнення структур даних, які можуть виникати при розробці алгоритмів у прикладних задачах. Це різноманітні структури на основі базових даних такі, як масиви масивів, масиви черг, масиви списків або списки масивів, списки списків, списки черг тощо, або інші складні комбінації класичних структур даних. Далі запропоновані прості програми формування деяких структур даних і наводяться приклади створення на їх основі представлення алгоритмів складних структур даних. Представлення алгоритмів наведене для мови С++, наявність коментарів до них дозволяє зрозуміти їх змістовну частину.

9.1. Формування класів складних даних

Наведемо практичне представлення гібридних класів структур даних масивів, списків і черг та їх комбінацій (У створені програм приймав участь студент кафедри «Комп'ютерні інформацій технології» Самарцов Д.М.).

9.1.1 Створення класу черг

```
class my_quere {
      struct node
      {
            int info;       // інформаційна частина вузла
            node * next;    // вказівник на наступний вузол
      };
      node *root;
      int count;
public:
      //конструктор без параметрів
      my_quere(){ count = 0; }
      //додавання у чергу числа
      void add(int i)
      {
            node *work, *last;
            last = root;
            if (last == NULL) // черга порожня
            {// додавання першого елементу у чергу
                  work = new node;
                  work->next = NULL;
                  root = work; last = work;
                  root->info=i;
```

```
          }while (last->next != NULL){ last = last->next; }//перехід
до останнього елементу черги
          //додаванняд останнього елементу черги
               work = new node;
               work->next = NULL;
               last->next = work;
               work->info=i;
               last = work;
               count++;}
     //отримання елементу з черги
     int getElem(){
          if (count != 0){
               int num;
               node *work;
               work = root;
               num = work->info;
               root = work->next;
               delete work;
               count--;
               return num; }
          cout << endl << "";
          return INT_MIN; }
     // отримання довжини черги
     int getCount()
{
return count;
}
};
```

9.1.2. Створення класу списків

```
// Клас списків цілих чисел
class node_int{
     int info;// інформація про значення даних
     node_int *next; // вказівник на наступний вузол
public:
     //конструктор без параметрів
     node_int(){}
     //int i // значення поля даних вузла списку
     node_int(int i)
{
info = i;
}
     int getInfo()
{
return info;
}
     void setInfo(int inf)
```

```
{
info = inf;
}
      node_int* getNext(){ return next; }
      void setNext(node_int *n)
{
next = n;
}
};
```

9.1.3. Створення класу вузла списку з даними масиву.

```
// Клас списку масиву цілих чисел
class node_intmas{
      int *info;// вказівник на перший елемент масиву
      int count;// довжина масиву
      node_intmas *next;// вказівник на наступний вузол
public:
      node_intmas();
      //int *mas – вказівник на перший елемент масиву
      //int c – довжина масиву
      node_intmas(int *mas, int c){
            for (int i = i; i < c; i++)
            { info[i] = mas[i]; }
            count = c; }
      //int *c - довжина масиву
      int* getInfo(int *c){
            c=&count;
            return info; }
      //int *inf – вказівник на перший елемент масиву
      //int c - довжина масиву
      void setInfo(int* inf, int count){
            for (int i = i; i < count; i++)
            { info[i] = inf[i]; } }
      node_intmas* getNext(){ return next; }
      void setNext(node_intmas *n){ next = n; }
};
```

9.1.4. Створення класу вузла списку черг

```
// Клас списку черг
class node_quere{
      my_quere *info;// утворення черги
      node_quere *next;// вказівник на наступний елемент
public:
      node_quere();
```

```
      node_quere(my_quere q){
            info = q;
      }
      my_quere* getInfo(){return info;}
      void setInfo(my_quere *q){info = q;}
      node_quere* getNext(){ return next; }
      void setNext(node_quere *n){ next = n; }
};
```

9.2. Дії над складними структурами даних

Дії, які виконуються над простими структурами даних (структурами нульового рівня) є традиційними і вважаються стандартними. Дії ж над сконструйованими структурами виконуються за спеціально створеними діями. Наведемо деяку демонстрацію деяких дій.

9.2.1. Дії з масивами списків та черг

```
node_int add(node_int *root,int i)
{
      node_int *work, *last;
      last = root;
      if (last == NULL) // список порожній
      {// додавання першого елементу у список
            work = new node_int();
            work->setNext(NULL);
            root = work; last = work;
            root->setInfo(i);
      }
      else{
            while (last->getNext() != NULL){ last = last->getNext(); }//
перехід до останнього о елементу списку
            // додавання останнього елементу у список
            work = new node_int();
            work->setNext(NULL);
            last->setNext(work);
            work->setInfo(i);
            last = work;
      }
}
void review(node_int * head)
{
      cout << endl << "Список:" << endl;
      node_int *work;
      // перевірка на те чи є порожнім список
```

```
      if (head == NULL) cout << " Список порожній!" << endl;
      else
      {
            work = head;
            while (work != NULL)
            {
                  cout << work->getInfo() << endl;
                  work = work->getNext(); // просування по списку
            }
      }
      _getch();
}
void review(my_quere* head)
{
      cout << endl << " Дія з чергою" << endl;
      if (head->getCount()==0) cout << "Черга порожня!" << endl;
      else
      {
            while (head->getCount()!=0)
            {
                  cout << head->getElem();
            }
      }
      _getch();
}
```

9.2.2. Дії з масивом списків

```
node_int *mas = new node_int[10];
for (int j = 0; j < 5; j++)
            for (int i; i < 10; i++){
                  mas[i] = add(&mas[i], 12 + j);
            }// заповнення даними масиву списків
      for (int i; i < 10; i++){
            review(&mas[i]);
      }// виведення даних з масиву списків.
```

9.2.3. Дії з масивом черг

```
      my_quere *mas1 = new my_quere[10];
      for (int j = 0; j < 5; j++)
            for (int i; i < 10; i++){
                  mas1[i].add(12 + j);
            }// заповнення масиву черг
      for (int i; i < 10; i++){
            review(&mas1[i]);
      }// виведення масиву черг
```

```
}
```

9.2.4. Дії зі списками масивів та черг

```
node_intmas* add(node_intmas *root, int *mas,int len)
{
      node_intmas *work, *last;
      last = root;
      if (last == NULL) // список порожній
      {// додавання першого елементу у список
            work = new node_intmas();
            work->setNext(NULL);
            root = work; last = work;
            root->setInfo(mas,len);
      }
      else{
            while (last->getNext() != NULL){ last = last->getNext(); }//
перехід до останнього елементу списку
            // додавання останнього елементу у список
            work = new node_intmas();
            work->setNext(NULL);
            last->setNext(work);
            work->setInfo(mas, len);
            last = work;
      }
}
node_quere* add(node_quere *root, my_quere *i)
{
      node_quere *work, *last;
      last = root;
      if (last == NULL) // список порожній
      {// додавання першого елементу у список
            work = new node_quere();
            work->setNext(NULL);
            root = work; last = work;
            root->setInfo(i);
      }
      else{
            while (last->getNext() != NULL){ last = last->getNext(); }//
перехід жо останнього елементу списку
            // додавання останнього елементу у список
            work = new node_quere();
            work->setNext(NULL);
            last->setNext(work);
            work->setInfo(i);
            last = work;
      }
}
```

```
void review(node_intmas * head)
{
	cout << endl << "Список:" << endl;
	node_intmas *work;
	// перевіряється на наявність пустого списку
	if (head == NULL) cout << "Список порожній!" << endl;
	else
	{
		work = head;
		while (work != NULL)
		{
			int count = 0, *mas = work->getInfo(&count);// int
count – довжина масиву
			for (int i = 0; i < count; i++){ cout << mas[i] << "
"; }
			work = work->getNext(); // просування по спписку
		}
	}
	_getch();
}
void review(node_quere * head)
{
	cout << endl << "Список:" << endl;
	node_quere *work;
	// перевірка на наявність порожнього списку
	if (head == NULL) cout << "Список порожній!" << endl;
	else
	{
		work = head;
		while (work != NULL)
		{
			review(work->getInfo());
			work = work->getNext();  // просування списком
		}
	}
	_getch();
}
```

9.2.5. Дії зі списком черг

```
	node_quere *root1;
	my_quere *q = new my_quere();
	for (int i = 0; i < 5;i++){ q->add(i); }//заповнення черги
root1 = add(root1, q);//добавленя вузла списку черг
review(root1);// виведення списку черг.
```

9.2.6. Дії зі списком масивів

```
node_intmas *root;
int count = 5;
      int *mas = new int[count];
      root = add(root, mas, count);// додавання вузла списку масивів
      review(root);// виведення списку масивів цілих чисел
```

Розглянуті приклади конструювання структур даних утворюють структури першого рівня. Структури даних вищих рівнів таких, як вектори списків, матриці масивів тощо можна утворити за методикою розглянутих методик на наведених прикладах.

ЛІТЕРАТУРА

1. Топп У., Форд У. Структуры данных в C++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2000. – 816 с.
2. Кубенский А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++.– СПб.: БХВ-Петербург, 2004. – 464 с.
3. Подбельский В.В. Язык C++: Учеб. пособие.– М.: Финансы и статистика. 2000. – 560 с.
4. Шаховська Н.Б. Алгоритми та структури даних. Пос. «Видавництво Ліра – К.», 2016 – 216 с.
5. Скалозуб В.В., Ільман В.М., Івченко Ю.М., Андрющенко В.О. Дискретні та алгоритмічні структури в інструментарії програмної інженерії: навч. посіб. – Д., 2016. – 254 с.
6. Алгоритми та структури даних: методичні вказівки до виконання лабораторних робіт. Уклад. Ільман В.М., Панік Л.О., Звоненко Н.В. – Д. 2016. – 68 с.
7. Дмитриев В.И. Прикладная теория информации .– М.: Высш. шк., 1989. – 320 с.
8. Кнут Д. Искусство программирования, том 3. Сортировка и поиск. – М.: Издательский дом «Вильямс», 2000. – 832 с.