Лабораторна робота № 7

Тема: програмування та аналіз алгоритму сортування за допомогою купи (пірамідальне сортування). Обчислення часу виконання алгоритму.
Мета: проаналізувати та дослідити алгоритм сортування методом купи.

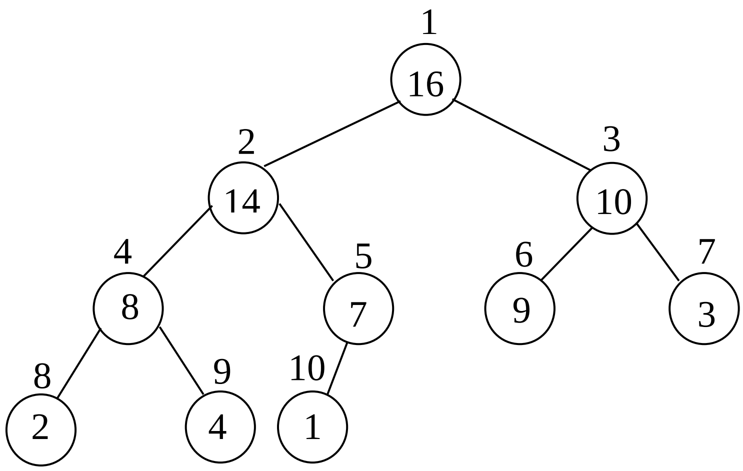
Основні теоретичні відомості

1 Сортування за допомогою купи

Як і алгоритм сортування злиттям (якій ми досліджували дещо раніше), алгоритм сортування за допомогою купи потребує часу $\theta(n \cdot log\, n)$ для сортування n об’єктів, але потребує додаткову пам’ять розміром $\theta(1)$ замість $\theta(n)$ для сортування злиттям. Таким чином, цей алгоритм містить переваги двох раніше розглянутих алгоритмів – сортування злиттям і сортування вставками [1].

Структура даних, яку використовує алгоритм (вона називається двійковою купою) буває корисною і в інших ситуаціях. Особливо ефективно на її основі можна організовувати чергу з пріоритетами тощо [1].

Двійковою купою називається масив з визначеними властивостями впорядкованості. Щоб сформулювати ці властивості, будемо розглядати масив як двійкове дерево (рис. 1).



а)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

б)

Рисунок 1

Кожна вершина дерева відповідає елементу масива. Якщо вершина має індекс i, то її батько має індекс $\lfloor i/2 \rfloor$, вершина з індексом *1* є коренем, а її дочірні вершини мають індекси $2i$ та $2i+1$. Будемо вважати, що купа може не займати всього масиву, і тому будемо зберігати не тільки масив A і його довжину *length[A]*, а й спеціальний параметр *heap-size[A]* (розмір купи), причому *heap-size[A]≤ length[A]*. Купа складається з елементів *A[1], ..., A[heap-size[A]]*.

Рух по дереву здійснюється за допомогою процедур:

PARENT(i)	LEFT(i)	RIGHT(i)
return $\lfloor i/2 \rfloor$;	return $2i$;	return $2i + 1$.

Купу можна розглядати як дерево (рис. 1 а) або масив (рис. 1 б). Усередині кожної вершини наведено її значення. Біля вершини наведено її індекс у масиві. Елемент *A[1]* є коренем дерева.

У більшості комп'ютерів для виконання процедур LEFT і PARENT можна використовувати команди лівого і правого зсуву, відповідно. Процедура RIGHT потребує лівого зсуву, після якого у молодший розряд записується одиниця.

Елементи, що зберігаються в купі повинні характеризуватися головною властивістю купи: для кожної вершини *i*, крім кореня (при *2≤i≤heap-size[A]}*,

$$A\left[PARENT(i)\right] \geq A\left[i\right] \tag{1}$$

Звідси випливає, що значення нащадка не перевищує значення батька. Таким чином, найбільший елемент дерева (або будь-якого піддерева) знаходиться у кореневій вершині цього дерева (або піддерева).

Висотою вершини дерева є висота піддерева з коренем в цій вершині (число ребер в найдовшому шляху збігається з початком у цій вершині вниз по дереву до листка). Висота дерева, таким чином, збігається з висотою його кореня. У дереві, що утворює купу, всі рівні (крім останнього) заповнені повністю. Тому висота цього дерева дорівнює $\theta(\log n)$, де n – число елементів купи. Як побачимо нижче, час роботи основних операцій над купою пропорційний висоті дерева і, відповідно, складає $\theta(\log n)$.

Перерахуємо основні операції над купою [1]:

• Процедура HEAPIFY дозволяє підтримувати головні властивості купи. Час її роботи складає $\theta(\log n)$.

• Процедура BUILD-HEAP будує купу з вихідного (невідсортованого) масиву. Час її роботи $\theta(n)$.

• Процедура HEAPSORT сортує масив, не використовуючи допоміжної пам'яті. Час її роботи $\theta(n \cdot log\, n)$.

• Процедури EXTRACT-MAX (отримання найбільшого) і INSERT (доповнення елементу) використовується при моделюванні черги з пріоритетами на базі купи. Час роботи обох процедур складає $\theta(log\, n)$.

Збереження головної властивості купи. Процедура HEAPIFY – важливий метод роботи з купою [1]. Її параметрами є масив *A* та індекс *i*. Вважається, що піддерева з коренями LEFT(*i*) і RIGHT(*i*) вже мають головні властивості. Процедура переміщує елементи піддерева з вершиною *i*, після чого воно буде мати головну властивість. Ідея проста: якщо головна властивість не виконана для вершини *i*, то її слід поміняти із старшим з її дочірніх вершин і т. д., доти, поки елемент A[*i*] не „завантажиться" до потрібного місця.

Процедура HEAPIFY наведена на рисунку 2. У рядках 3 – 7 змінна *largest* набуває значення, що дорівнює індексу найбільшого з елементів *A*[*i*], *A*[LEFT(*i*)] та *A*[RIGHT(*i*)]. Якщо *largest=i*, то елемент *A*[*i*] вже „завантажився" до потрібного місця, і робота процедури закінчена. Інакше процедура міняє місцями *A*[*i*] та *A*[*largest*] (що забезпечує виконання властивості (1) у вершині *i*, але, можливо, порушує цю властивість у вершині *largest*) і рекурсивно викликає себе для вершини *largest* (рядок 10), щоб виправити можливі порушення.

HEAPIFY(*A, i*)
1. *l*←LEFT(*i*)
2. *r*←RIGHT(*i*)
3. **if** *l* ≤ (*heap-size*[*A*] *and* *A*[*l*] > *A*[*i*])
4. **then** *largest* ← *l*
5. **else** *largest* ← *i*
6. **if** *r* ≤ (*heap-size*[*A*] *and A*[*r*] > *A*[*largest*])
7. **then** *largest* ← *r*
8. **if** *largest* ≠ *i*
9. **then** виконати обмін *A*[*i*] ↔ *A*[*largest*]
10. HEAPIFY(*A*, *largest*)

Рисунок 2 – Процедура збереження головної властивості купи

Приклад виконання процедури HEAPIFY наведений на рис. 3. Робота процедури HEAPIFY(*A,2*) при *heap-size*[*A*]=10 (a). У вершині *i=2* головна властивість порушена. Щоб відновити її, необхідно поміняти місцями

$A[2]$ і $A[4]$. Після цього (б) головна властивість порушується у вершині з індексом 4. Рекурсивний виклик процедури HEAPIFY(A,4) відновлює головну властивість у вершині з індексом 4 шляхом перестановки $A[4] \leftrightarrow A[9]$ (в). Після цього головна властивість виконана для усіх вершин, тому процедура HEAPIFY(A,9) вже нічого не виконує.

Оцінимо час роботи процедури HEAPIFY. На кожному кроці потрібно провести $\theta(1)$ дій, не враховуючи рекурсивного виклику. Нехай $T(n)$ – час роботи для піддерева, що містить n елементів. Якщо піддерево з коренем i складається з n елементів, то піддерева з коренями LEFT(i) та RIGHT(i) містять не більше ніж по $2n/3$ елементів кожен (найгірший випадок – коли останній рівень в піддереві заповнений наполовину). Таким чином,

$$\mathrm{T}(n) \le \mathrm{T}(2\, n\, /3) + \theta(1).$$

З основної теореми (див наступну лабораторну роботу) про рекурентні оцінки (випадок 2) отримуємо, що $\mathrm{T}(n) = \theta(\log n)$. Цю ж оцінку можна отримати так: на кожному кроці ми опускаємось по дереву на один рівень, враховуючи, що висота дерева це $\theta(\log n)$.

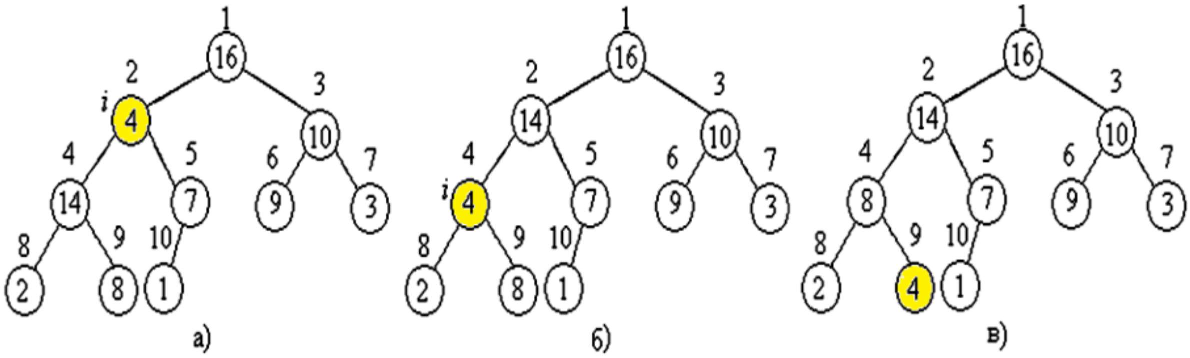


Рисунок 3 – Приклад виконання процедури HEAPIFY

Побудова купи. Нехай дано масив $A[1\,..\,n]$, який ми хочемо перетворити на купу, переставивши його елементи. Для цього можна використати процедуру HEAPIFY, застосовуючи її по черзі до усіх вершин, починаючи з нижніх. Оскільки вершини з номерами $\lfloor n/2 \rfloor + 1,\ ...,\ n$ є листям, піддерева з цими вершинами задовольняють головну властивість. Для кожної з вершин, що залишились, у порядку зменшення індексів, використовуємо процедуру HEAPIFY. Порядок обробки вершин гарантує, що кожен раз умови виклику процедури (виконання головної властивості для піддерев) будуть виконані.

Процедура BUILD-HEAP(A) наведена на рисунку 4

BUILD-HEAP (A)
1. *heap-size[A]* $\leftarrow$ *length[A]*
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. **do** HEAPIFY (A, i)

Рисунок 4 – Процедура побудови купи

Приклад роботи даної процедури наведено на рис. 5, де показано стан даних перед кожним викликом процедури HEAPIFY у рядку 3.

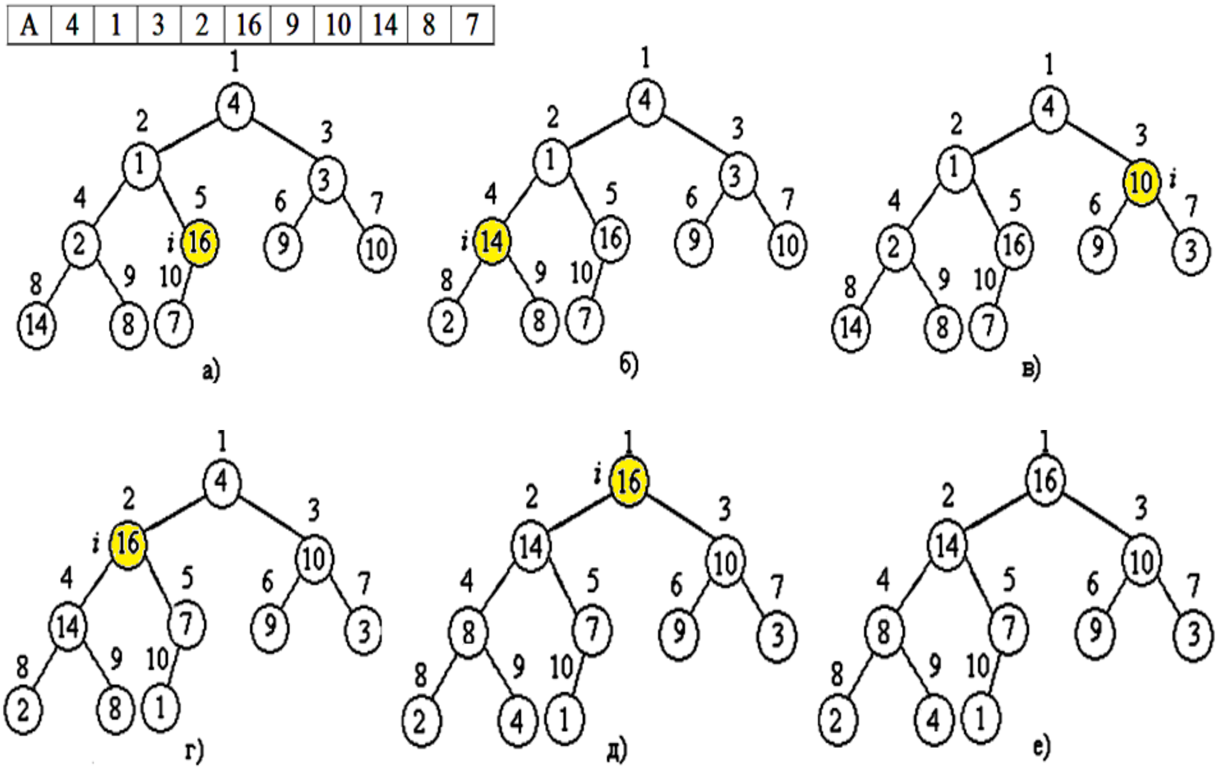


Рисунок 5 – Робота процедури BUILD-HEAP

Зрозуміло, що час роботи процедури BUILD-HEAP не перевищує $\theta(n \cdot \log n)$. Дійсно, процедура HEAPIFY викликається $\theta(n)$ раз, а кожне її виконання потребує часу $\theta(\log n)$. Однак цю оцінку можна покращити.

Справа в тому, що час роботи процедури HEAPIFY залежить від висоти вершини, для якої вона викликається (i пропорційна цій висоті). Оскільки число вершин висотою h в купі з n елементів не перевищує $\lceil n/2^{h+1} \rceil$, а висота всієї купи не перевищує $\lfloor \log n \rfloor$, час роботи процедури BUILD-HEAP не перевищує

$$\sum_{h=0}^{[\log n]} \left\lceil \frac{n}{2^{n+1}} \right\rceil \theta(h) = \theta\left(n \sum_{h=0}^{[\log n]} \frac{h}{2^h} \right). \qquad (2)$$

Припускаючи, що x=1/2, отримуємо верхню оцінку для суми у правій частині

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

Таким чином, час роботи процедури BUILD-HEAP складає:

$$\theta\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = \theta(n).$$

Алгоритм сортування за допомогою купи. Алгоритм сортування за допомогою купи складається з двох частин. Спочатку викликається процедура BUILD-HEAP, після виконання якої масив стає купою. Ідея другої частини така: максимальний елемент масиву тепер знаходиться у корені дерева $A[1]$. Його слід поміняти з елементом $A[n]$, зменшити розмір купи на 1 і відновити головну властивість у кореневій вершині (оскільки піддерева з коренями LEFT(1) і RIGHT(1) не втратили головної властивості купи, це можна зробити з допомогою процедури HEAPIFY. Після цього в корені буде знаходитися максимальний з елементів, що залишився. Так робиться доти, поки в купі не залишиться лише один елемент [1].

HEAPSORT(*A*)
1. BUILD-HEAP (*A*)
2. **for** *i* ← *length*[*A*] downto 2
3. **do** замінити *A*[*1*]↔ *A*[*i*]
4. *heap-size*[*A*] ←*heap-size*[*A*] - 1
5. HEAPIFY(*A*, 1)

Робота другої частини алгоритму наведена на рис. 6. Тут зображені стани купи перед кожним виконанням циклу for (рядок 2), а заштриховані елементи вже не входять до купи. Час роботи процедури HEAPSORT складає $\theta(n \cdot \log n)$. Дійсно, перша частина (побудова купи) потребує часу $\theta(n)$, а кожне з n-1 виконань циклу for потребує часу $\theta(\log n)$.

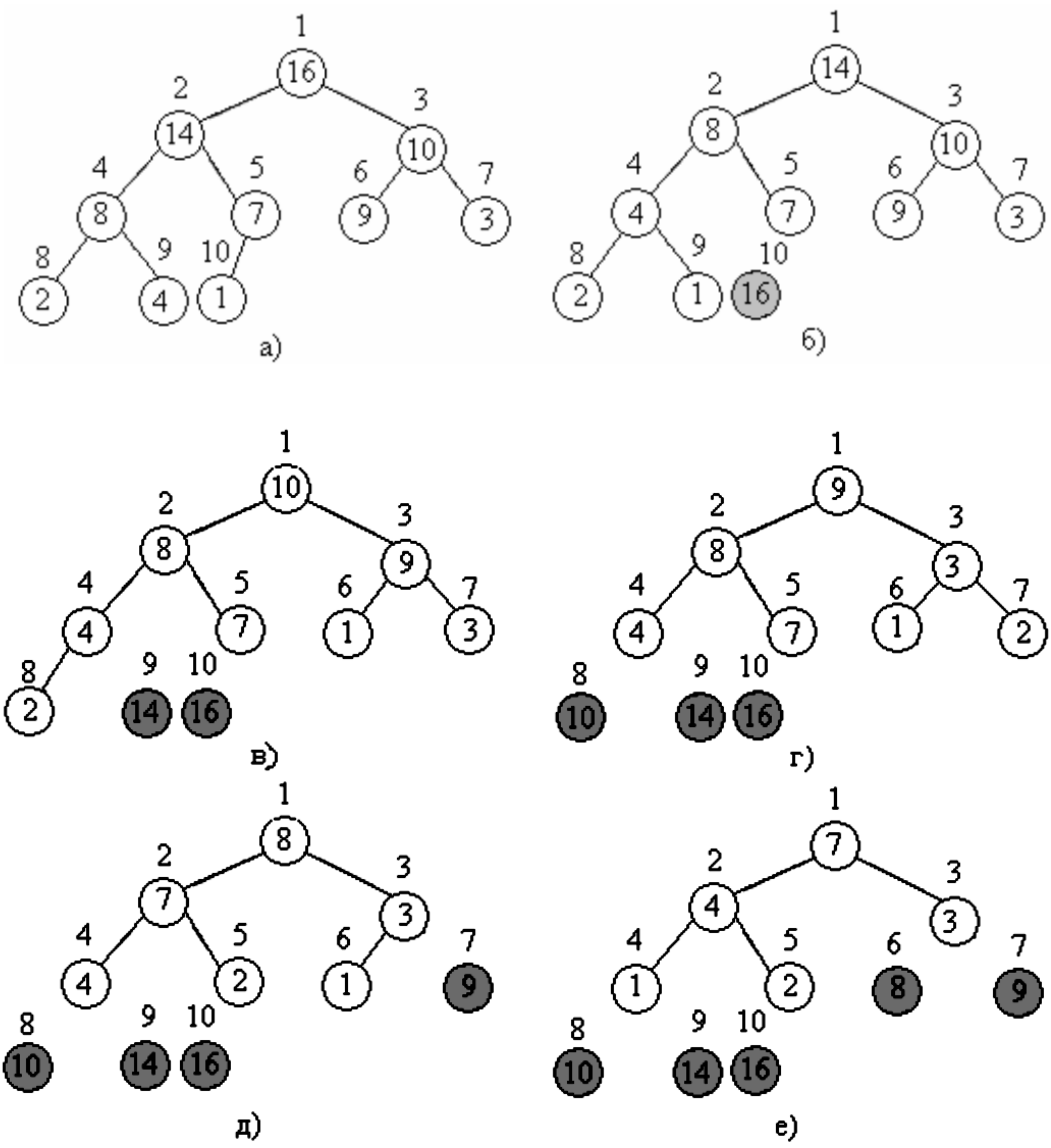
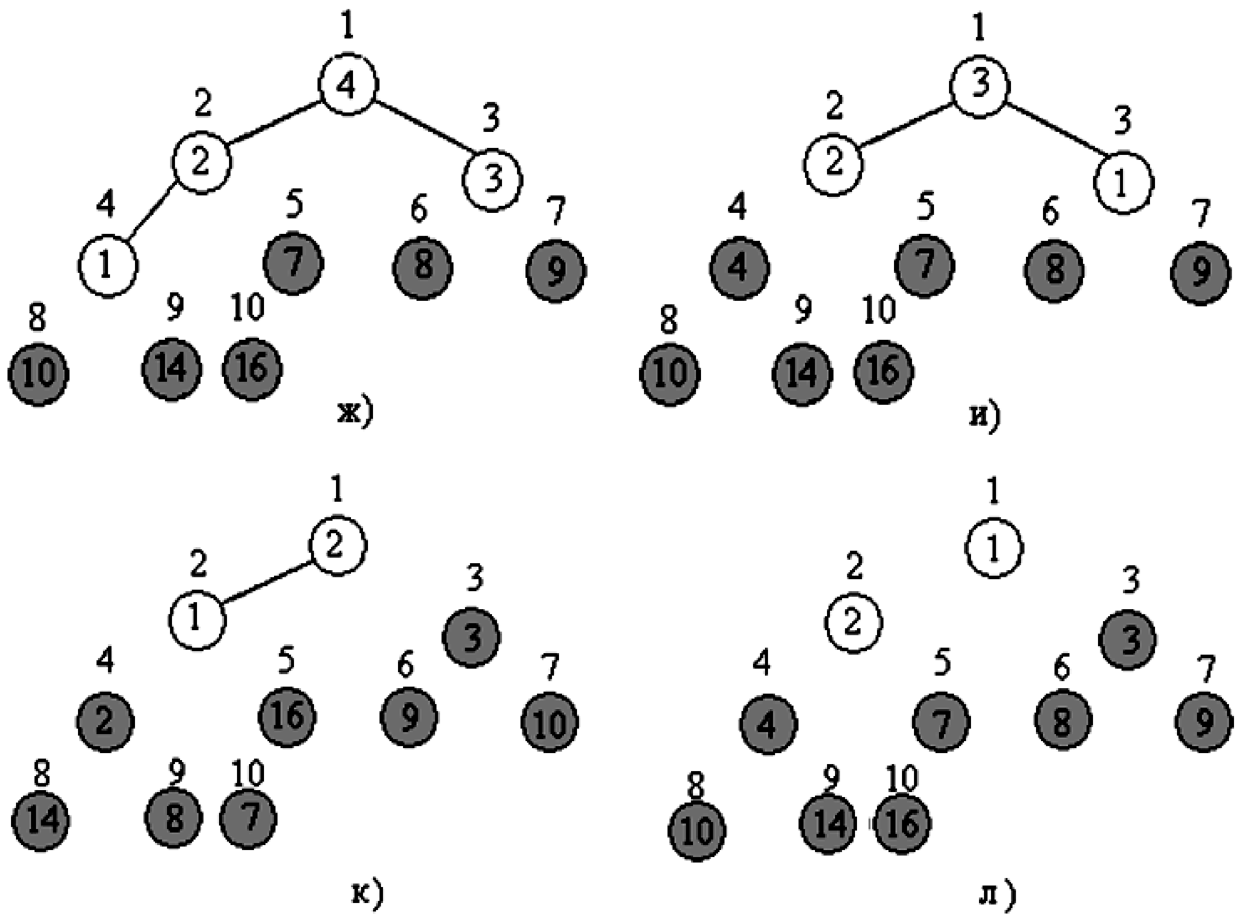


Рисунок 6 – Приклад сортування за допомогою купи



A	1	2	3	4	7	8	9	10	14	16

Рисунок 6 – (продовження)

Насамкінець доцільно зазначити, що на практиці алгоритм сортування за допомогою купи не є найшвидшим – як правило, швидке сортування, яке ми розглянемо нижче, працює швидше. Однак сама купа, як структура даних, часто виявляється досить корисною [1].

Контрольні питання

1. Поясніть чому задача сортування елементів є однією з найцікавіших та показових задач для курсу теорії алгоритмів.
2. Поясніть, що таке стійкість алгоритму сортування.
3. За якими критеріями на Ваш погляд можна класифікувати алгоритми сортування?
4. Наведіть класифікацію алгоритмів сортування.
5. Перерахуйте та порівняйте відомі Вам алгоритми сортування за псевдолінійний час.

6. Поясніть чому при оцінці складності алгоритму нас частіше за все цікавить робота у найгіршому випадку.
7. Наведіть основні операції над купою.
8. Наведіть та поясніть як працює процедура збереження головної властивості купи.
9. Поясніть яким чином з довільного масиву побудувати купу. Скільки часу потребує така процедура?
10. Наведіть власний приклад сортування за допомогою купи.
11. Наведіть переваги та недоліки алгоритму сортування методом купи.

Звіт повинен містити

1. Тему, мету та порядок виконання роботи.
2. Ідея відповідного алгоритму сортування.
3. Власний приклад роботи відповідного алгоритму сортування на масиві з 10 чисел.
4. Алгоритм сортування у графічному вигляді.
5. Теоретична оцінка складності алгоритму.
6. Сирець алгоритму відповідного сортування.
7. Практична оцінка складності відповідного алгоритму в трьох випадках для масивів різного розміру. Навести таблиці та відповідні графіки часу виконання програми для трьох випадків. Кількість значень часу має бути не менше семи.
   Увага! Мінімальне та максимальне значення часу ***кожним студентом підбирається індивідуально****, залежно від потужностей Вашого ПК. При цьому врахуйте, що мінімальне значення часу повинно бути порядка кількох секунд (до 10), а максимальне – не більше 5 – 10 хвилин (за Вашим бажанням останній час може бути збільшено).*
8. Порівняльний аналіз теоретично та практично отриманих графіків залежностей часів виконання програми від розмірів входу.
9. Переваги та недоліки дослідженого Вами алгоритму сортування та Ваші міркування.
10. Розширені висновки з роботи.