

# **Алгоритми та складність**

I семестр

Лекція 5

# Пірамідальне сортування (heapsort)

- *Піраміда (binary heap, бінарна купа)* – частково упорядкована структура даних, яку можна розглядати як бінарне дерево з певними властивостями.
- Дерево заповнене на всіх рівнях крім, можливо, останнього.
- Останній рівень заповнюється зліва направо до вичерпання елементів.
- Ключі у вузлах певним чином упорядковані.
- Бінарна піраміда може бути ефективно реалізована у вигляді масиву шляхом запису її елементів згори донизу зліва направо.

# Пірамідальне сортування (heapsort)

- Зручно зберігати елементи в масиві на позиціях з 1 до  $n$  (нульовий індекс або не використовується, або заповнюється значенням-обмежувачем).
- Індекси батька й синів  $i$ -го вузла обчислюються наступними функціями:  
$$\text{PARENT}(i)$$
$$\text{return } \lfloor i/2 \rfloor$$
$$\text{LEFT}(i)$$
$$\text{return } 2i$$
$$\text{RIGHT}(i)$$
$$\text{return } 2i + 1$$
- Ці операції будуть швидко виконуватись при реалізації через бітові зміщення.

# Пірамідальне сортування (heapsort)

- *Незростаюча піраміда* (max-heap property): для кожного некореневого вузла виконується

$$A[\text{PARENT}(i)] \geq A[i].$$

Тобто в корені знаходиться *найбільший* елемент.

- *Неспадаюча піраміда* (min-heap property): для кожного некореневого вузла виконується

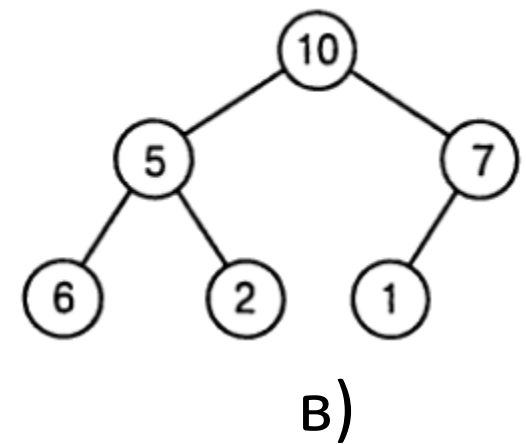
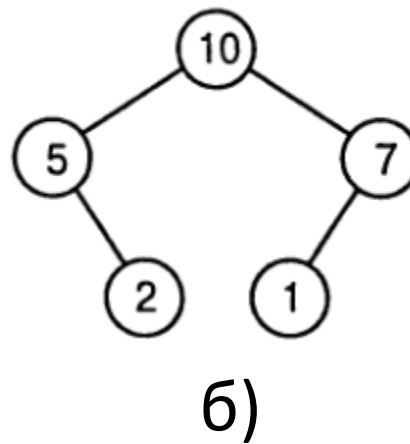
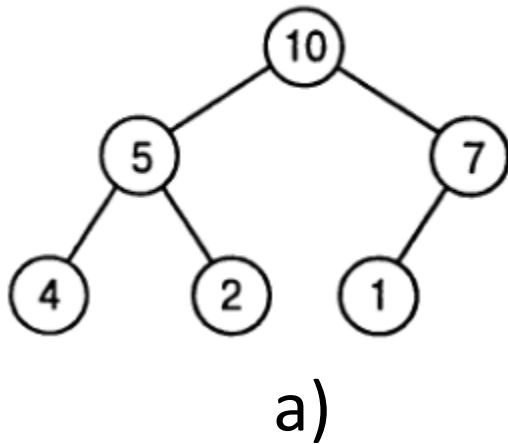
$$A[\text{PARENT}(i)] \leq A[i].$$

Тобто в корені знаходиться *найменший* елемент.

- Висота піраміди – висота її кореня.
- Для  $n$ -елементної піраміди це  $\Theta(\lg n)$ .
- Час виконання основних операцій в піраміді пропорційний висоті дерева.

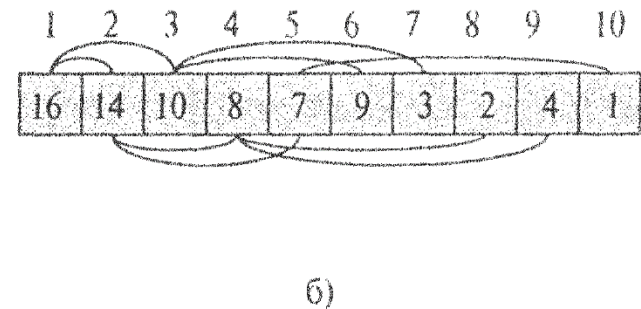
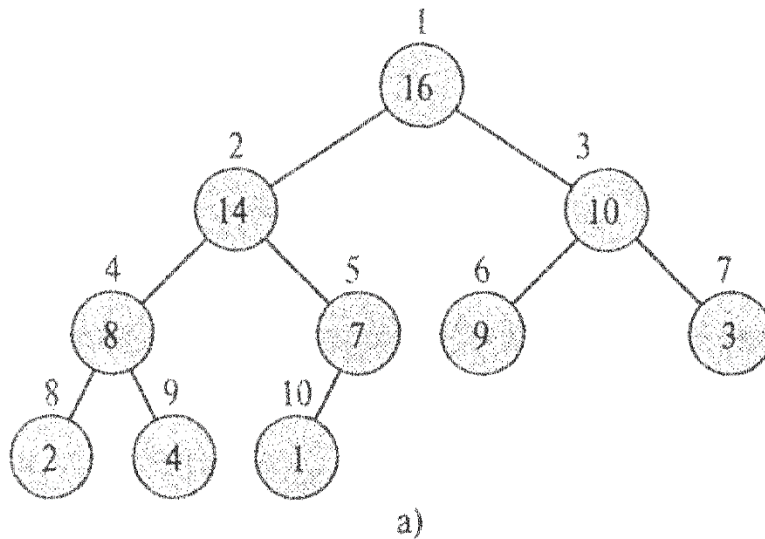
# Пірамідальне сортування (heapsort)

- Чи є вказані дерева пірамідами?



# Пірамідальне сортування (heapsort)

- Приклад представлення піраміди бінарним деревом (а) та масивом (б):



# Пірамідальне сортування (heapsort)

- Якою будуть найбільша та найменша кількості елементів в піраміді висотою  $h$ ?

# Пірамідальне сортування (heapsort)

- Якою будуть найбільша та найменша кількості елементів в піраміді висотою  $h$ ?
- Чи буде сам пірамідою довільний вузол піраміди разом з його потомками?



# Пірамідальне сортування (heapsort)

- Якою будуть найбільша та найменша кількості елементів в піраміді висотою  $h$ ?
- Чи буде сам пірамідою довільний вузол піраміди разом з його потомками?
- Чи є масив з відсортованими елементами неспадаючою пірамідою?

# Пірамідальне сортування (heapsort)

- Якою будуть найбільша та найменша кількості елементів в піраміді висотою  $h$ ?
- Чи буде сам пірамідою довільний вузол піраміди разом з його потомками?
- Чи є масив з відсортованими елементами неспадаючою пірамідою?
- Де в незростаючій піраміді, у якої всі елементи різні, може знаходитися найменший елемент?

# Пірамідальне сортування (heapsort)

- Якою будуть найбільша та найменша кількості елементів в піраміді висотою  $h$ ?
- Чи буде сам пірамідою довільний вузол піраміди разом з його потомками?
- Чи є масив з відсортованими елементами неспадаючою пірамідою?
- Де в незростаючій піраміді, у якої всі елементи різні, може знаходитися найменший елемент?
- Які індекси масиву будуть мати листки  $n$ -елементної піраміди?

# Пірамідальне сортування (heapsort)

## Процедура Max Heapify

- Розглядається незростаюча піраміда
- Служить для підтримки властивості незростання піраміди

АЛГОРИТМ *MAX\_HEAPIFY* (  $A, i$  )

1  $l \leq \text{LEFT}(i)$

2  $r \leq \text{RIGHT}(i)$

3 **if**  $l \leq \text{heap\_size}[A]$  та  $A[l] > A[i]$

4     **then**  $\text{largest} \leq l$

5     **else**  $\text{largest} \leq i$

6 **if**  $r \leq \text{heap\_size}[A]$  та  $A[r] > A[\text{largest}]$

7     **then**  $\text{largest} \leq r$

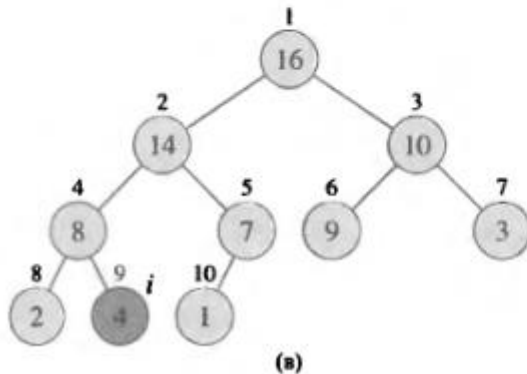
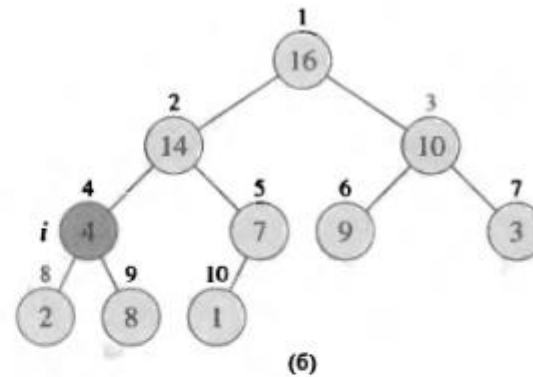
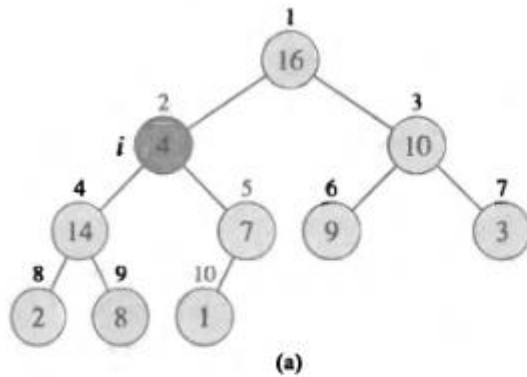
8 **if**  $\text{largest} \neq i$

9     **then** Обміняти  $A[i] \Leftrightarrow A[\text{largest}]$

10         *MAX\_HEAPIFY*( $A, \text{largest}$ )

# Пірамідальне сортування (heapsort)

- Вважається, що дерева з коренями  $\text{Left}(i)$  та  $\text{Right}(i)$  – незростаючі піраміди, але елемент  $A[i]$  може порушувати цю властивість.
- Значення  $A[i]$  просувається вниз, поки відповідне дерево з цим коренем не стане незростаючою пірамідою.



Ілюстрація роботи  
 $\text{Max\_Heapify}(A, 2)$

# Пірамідальне сортування (heapsort)

Час роботи Max\_Heapify:

- час виправлення відношень між елементами  $A[i]$ ,  $A[\text{Left}(i)]$  або  $A[\text{Right}(i)]$  складає  $\Theta(1)$ ;
- час роботи процедури з піддеревом з коренем в одному з дочірніх вузлів вузла  $i$ : розмір кожного з них не перевищить  $2n/3$  (в найгіршому випадку останній рівень буде заповнений наполовину):

$$T(n) \leq T(2n/3) + \Theta(1).$$

Розв'язок співвідношення (випадок 2 основної теореми):  $T(n) = O(\lg n)$ .

Час роботи процедури з вузлом на висоті  $h$ :  $O(h)$ .

# Пірамідальне сортування (heapsort)

## Процедура Build Max Heap:

- кожен лист дерева можна вважати одноелементною пірамідою;
- будує піраміду знизу вгору, викликаючи для кожного вузла-нелиста Max\_Heapify:

АЛГОРИТМ *BUILD\_MAX\_HEAP* (A)

```
1  heap_size[A] <= length[A]
2  for  $i \leq \lfloor \text{length}[A] / 2 \rfloor$  downto 1
3    do MAX_HEAPIFY(A,  $i$ )
```

Покажемо коректність алгоритму через інваріант циклу:

*який?*

# Пірамідальне сортування (heapsort)

## Процедура Build Max Heap:

- кожен лист дерева можна вважати одноелементною пірамідою;
- будує піраміду знизу вгору, викликаючи для кожного вузла-нелиста Max\_Heapify:

АЛГОРИТМ *BUILD\_MAX\_HEAP* (A)

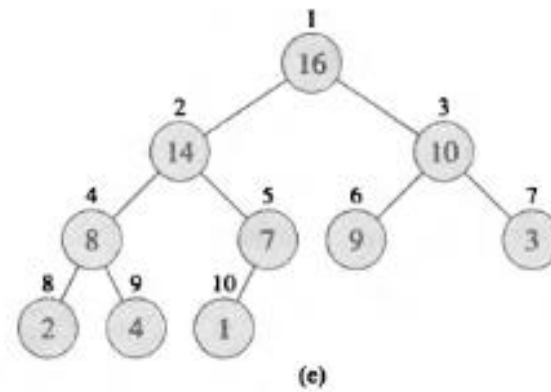
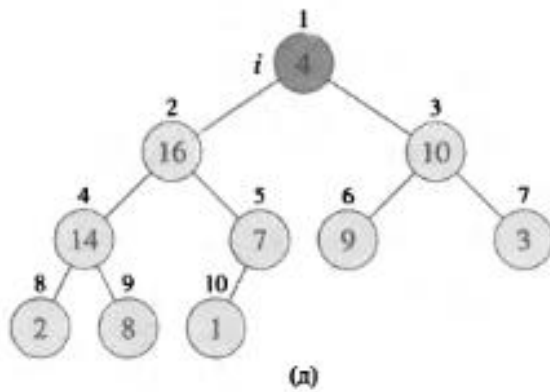
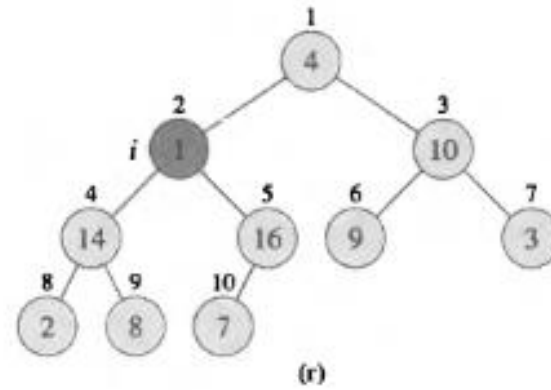
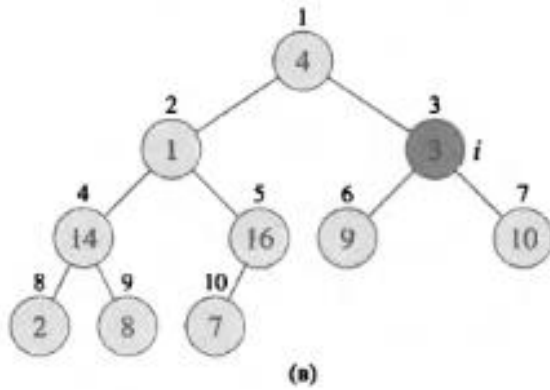
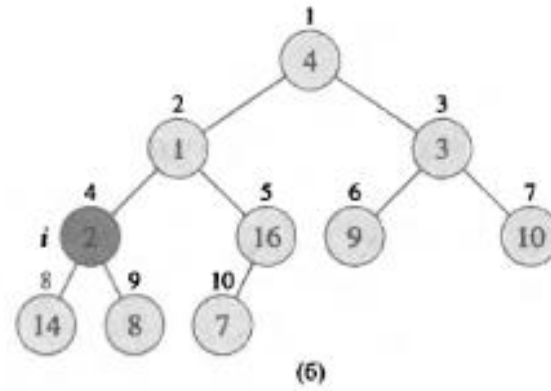
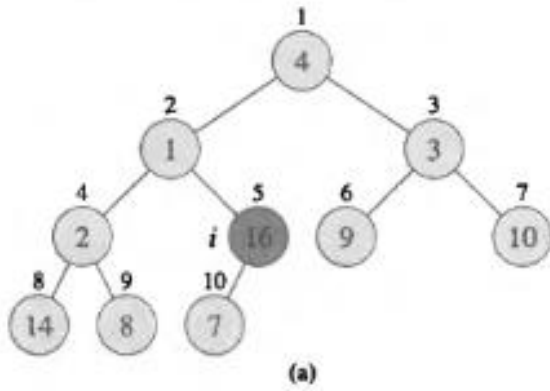
```
1  heap_size[A] <= length[A]
2  for  $i \leq \lfloor \text{length}[A] / 2 \rfloor$  downto 1
3    do MAX_HEAPIFY(A,  $i$ )
```

Покажемо коректність алгоритму через інваріант циклу:

перед кожною ітерацією циклу всі вузли з індексами  $(i+1)$ ,  $(i+2)$ , ...,  $n$  є коренями незростаючих пірамід.



A [ 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 ]



# Пірамідальне сортування (heapsort)

Коректність Build\_Max\_Hear:

- *Ініціалізація*: перед першою ітерацією  $i = \lfloor n/2 \rfloor$ ; усі вузли з індексами  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  – листи, тому кожен з них є коренем тривіальної піраміди.

# Пірамідальне сортування (heapsort)

Коректність Build\_Max\_Heap:

- *Ініціалізація*: перед першою ітерацією  $i = \lfloor n/2 \rfloor$ ; усі вузли з індексами  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  – листи, тому кожен з них є коренем тривіальної піраміди.
- *Збереження*: відносно вузла  $i$  його дочірні вершини мають більший номер та за інваріантом циклу є коренями незростаючих пірамід – умова виклику  $\text{Max\_Heapify}(A, i)$ , щоб зробити вузол  $i$  коренем незростаючої піраміди; виклик зберігає факт, що вузли з  $(i+1)$ -го по  $n$ -й є коренями незростаючих пірамід, а зменшення індексу  $i$  в циклі забезпечує виконання інваріанту перед наступною ітерацією.

# Пірамідальне сортування (heapsort)

Коректність Build\_Max\_Heap:

- *Ініціалізація*: перед першою ітерацією  $i = \lfloor n/2 \rfloor$ ; усі вузли з індексами  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  – листи, тому кожен з них є коренем тривіальної піраміди.
- *Збереження*: відносно вузла  $i$  його дочірні вершини мають більший номер та за інваріантом циклу є коренями незростаючих пірамід – умова виклику  $\text{Max\_Heapify}(A, i)$ , щоб зробити вузол  $i$  коренем незростаючої піраміди; виклик зберігає факт, що вузли з  $(i+1)$ -го по  $n$ -й є коренями незростаючих пірамід, а зменшення індексу  $i$  в циклі забезпечує виконання інваріанту перед наступною ітерацією.
- *Завершення*: після виконання циклу  $i=0$ . За інваріантом всі вузли з індексами  $1, 2, \dots, n$  – корені незростаючих пірамід, зокрема і кореневий вузол 1.

# Пірамідальне сортування (heapsort)

Груба оцінка процедури Build\_Max\_Heap:

- кожен виклик Max\_Heapify займає час  $O(\lg n)$ ;
- усього таких викликів  $O(n)$ ;
- значить верхня оцінка  $O(n \lg n)$ .

# Пірамідальне сортування (heapsort)

Груба оцінка процедури Build\_Max\_Heap:

- кожен виклик Max\_Heapify займає час  $O(\lg n)$ ;
- усього таких викликів  $O(n)$ ;
- значить верхня оцінка  $O(n \lg n)$ .

Факти для точної оцінки:

- висота  $n$ -елементної піраміди  $\lceil \lg n \rceil$ ;
- рівень на висоті  $h$  містить не більше  $\lceil n/2^{h+1} \rceil$  вузлів;
- час виклику Max\_Heapify для вузла висоти  $h$ :  $O(h)$ .

# Пірамідальне сортування (heapsort)

Верхня оцінка процедури Build\_Max\_Heap:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

З співвідношення  $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$  при  $x = 1/2$  отримуємо

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

Звідки

$$O \left( n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

# Пірамідальне сортування (heapsort)

## Алгоритм Heapsort:

АЛГОРИТМ *HEAPSORT* (A)

1 BUILD\_MAX\_HEAP(A)

2 **for**  $i \leq \text{length}[A]$  **downto** 2

3   **do** Обміняти  $A[1] \rightleftharpoons A[i]$

4        $\text{heap\_size}[A] \leftarrow \text{heap\_size}[A] - 1$

5       MAX\_HEAPIFY(A,1)

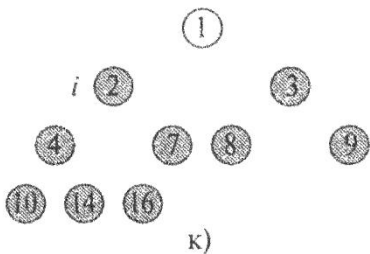
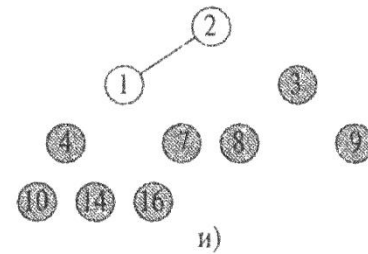
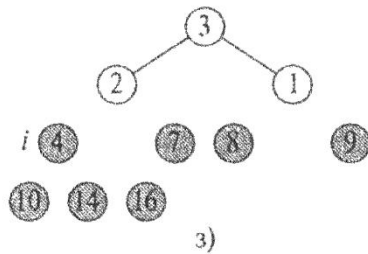
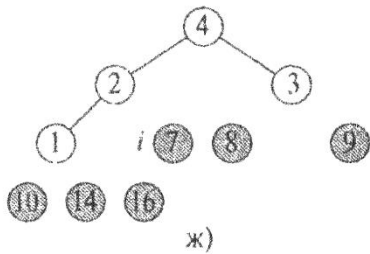
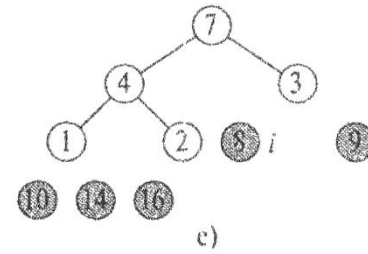
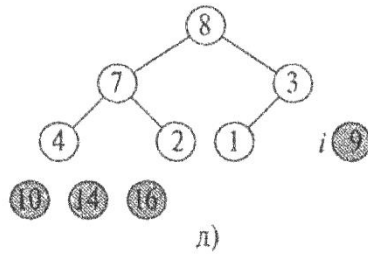
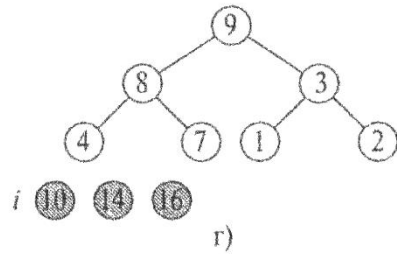
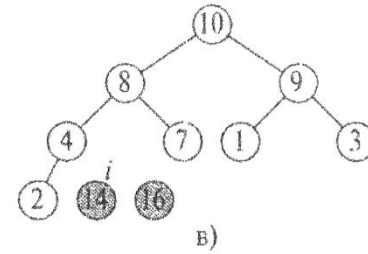
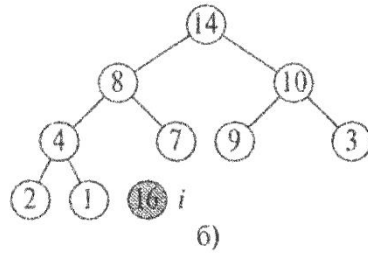
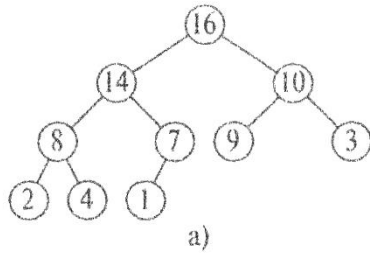
- Build\_Max\_Heap має час  $O(n)$ ;
- $(n-1)$  виклик Max\_Heapify по  $O(\lg n)$ .

Отже, загальний час  $O(n \lg n)$ .

Сортування пірамідою не використовує додаткову пам'ять.



# Ілюстрація роботи алгоритму Heapsort



А

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

л)

# Черги з пріоритетами

На основі пірамід можна ефективно реалізувати чергу з пріоритетами.

Черга з пріоритетами – структура даних, призначена для обслуговування множини  $S$ , з кожним елементом якої пов'язане значення-ключ.

Операції незростаючої черги з пріоритетами:

- $\text{Insert}(S, x)$  – додає елемент  $x$  до множини  $S$ .
- $\text{Maximum}(S)$  – повертає елемент  $S$  з найбільшим ключем.
- $\text{Extract\_Max}(S)$  – повертає елемент  $S$  з найбільшим ключем, видаливши його.
- $\text{Increase\_Key}(S, x, k)$  – замінює значення ключа, що відповідає елементу  $x$ , на ключ з більшим значенням  $k$ .

# Черги з пріоритетами

- Незростаюча черга з пріоритетами: планування завдань на комп'ютері, що спільно використовується різними користувачами (наступним вибирається завдання з найбільшим пріоритетом).
- Неспадаюча черга з пріоритетами: моделювання систем, що керують подіями (ключем є час настання події; вибирається подія, що відбулася раніше за інші).

На практиці часто може виникати необхідність визначати за об'єктом відповідний елемент черги чи навпаки. Тому має бути передбачена належна організація і обробка взаємних ідентифікаторів.

# Черги з пріоритетами

АЛГОРИТМ *HEAP\_MAXIMUM* (A)

1 **return** A[1]

Константний час  $\Theta(1)$ .

АЛГОРИТМ *HEAP\_EXTRACT\_MAX* (A)

1 **if** *heap\_size*[A] < 1

2     **then error** “Порожня черга”

3 *max* <= A[1]

4 A[1] <= A[*heap\_size*[A]]

5 *heap\_size*[A] <= *heap\_size*[A] – 1

6 MAX\_HEAPIFY(A,1)

7 **return** *max*

Час  $O(\lg n)$ , бо такий час виклику Max\_Heapify, а перед ним константні кроки.

# Черги з пріоритетами

АЛГОРИТМ *HEAP\_INCREASE\_KEY* ( $A, i, key$ )

```
1  if  $key < A[i]$   
2    then error “Новий ключ менший за поточний”  
3   $A[i] \leq key$   
4  while  $i > 1$  та  $A[PARENT(i)] < A[i]$   
5    do Обміняти  $A[i] \Leftrightarrow A[PARENT(i)]$   
6     $i \leftarrow PARENT(i)$ 
```

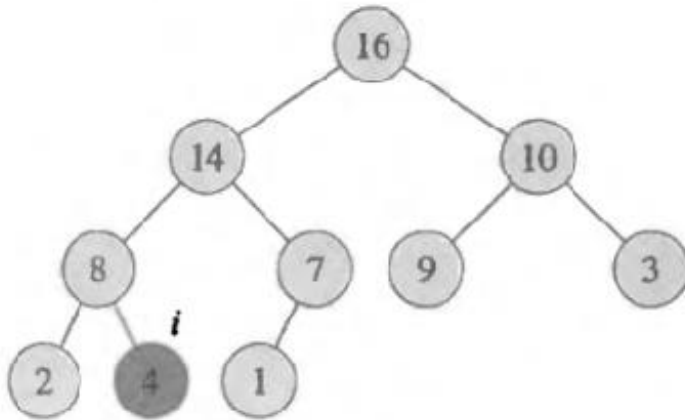
$i$  – позиція в піраміді,  $key$  – новий ключ.

Після заміни ключа йде перевірка, чи не порушилась властивість незростання. В такому випадку відбувається обмін з батьківським елементом, поки властивість не відновиться.

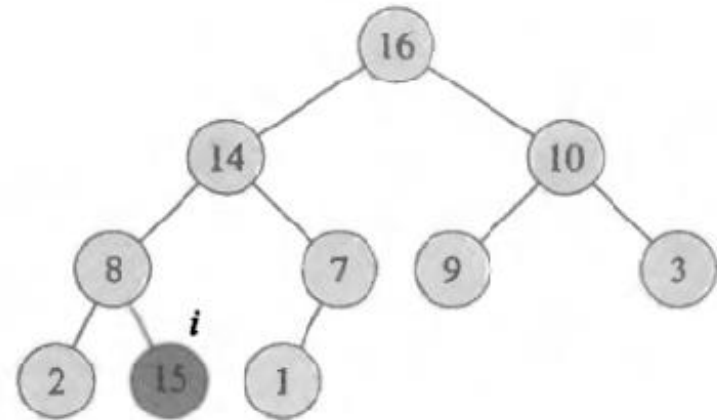
Час роботи  $O(\lg n)$  – пропорційно висоті пірамід.

# Черги з пріоритетами

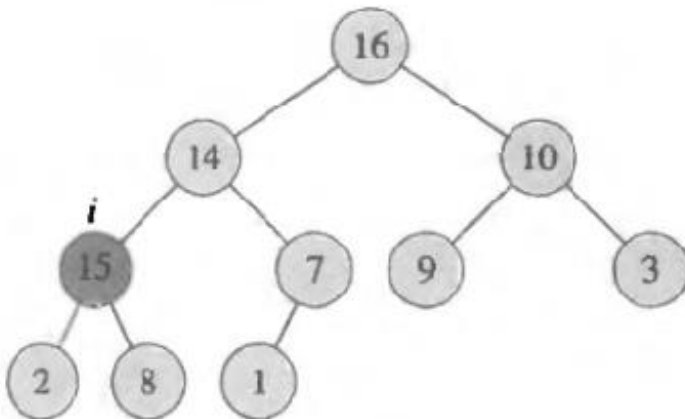
Приклад роботи для  $\text{Heap\_Increase\_Key}(A, i, 15)$ ,  $A[i] = 4$ .



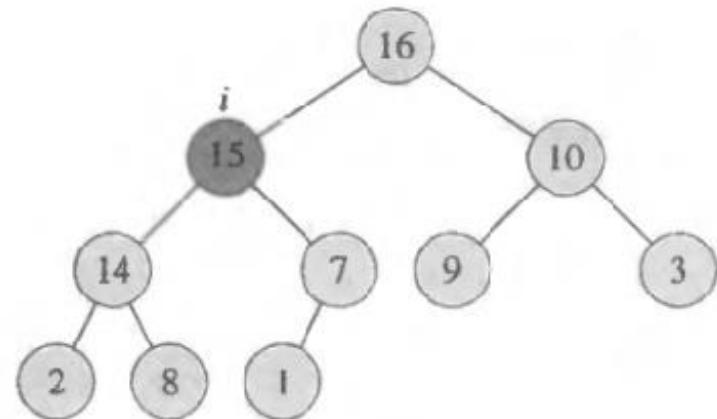
(a)



(б)



(в)



(г)

# Черги з пріоритетами

АЛГОРИТМ *MAX\_HEAP\_INSERT* ( $A, key$ )

- 1  $heap\_size[A] \leq heap\_size[A] + 1$
- 2  $A[heap\_size[A]] \leq -\infty$
- 3 *HEAP\_INCREASE\_KEY*( $A, heap\_size[A], key$ )

Спочатку створюється новий лист зі значенням  $-\infty$ , а потім через процедуру *Heap\_Increase\_Key* йому надається потрібне значення.

Тому час роботи алгоритму  $O(\lg n)$ .

- Всі операції над чергою з пріоритетами виконуються за час  $O(\lg n)$ .

# Швидке сортування

Належить до алгоритмів типу «розділяй та владарюй»

- *Поділ.* Масив  $A[p..r]$  розбивається шляхом перерозподілу елементів на (можливо порожні) підмасиви  $A[p..(q-1)]$  та  $A[(q+1)..r]$ . При цьому кожен елемент  $A[p..(q-1)]$  не перевищує  $A[q]$ , а кожен елемент  $A[(q+1)..r]$  не менше  $A[q]$ . Індекс  $q$  обчислюється в ході розбиття.
- *Підкорення.* Рекурсивно сортуються масиви  $A[p..(q-1)]$  та  $A[(q+1)..r]$ .
- *Комбінування.* Підмасиви сортуються на місці, об'єднання не потрібне: весь масив  $A[p..r]$  буде відсортований.



# Швидке сортування

Схема алгоритму:

АЛГОРИТМ *QUICKSORT* ( $A, p, r$ )

```
1  if  $p < r$   
2    then  $q \leq \text{PARTITION}(A, p, r)$   
3        QUICKSORT ( $A, p, q-1$ )  
4        QUICKSORT ( $A, q+1, r$ )
```

Початковий виклик *QUICKSORT*( $A, 1, \text{length}[A]$ )

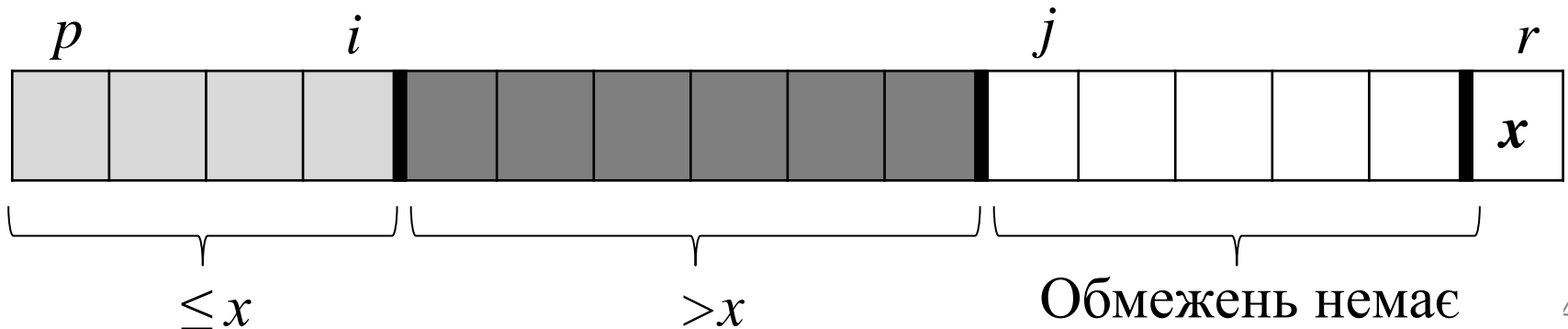
# Швидке сортування

## Розбиття масиву (спосіб Ломуто):

АЛГОРИТМ *PARTITION* ( $A, p, r$ )

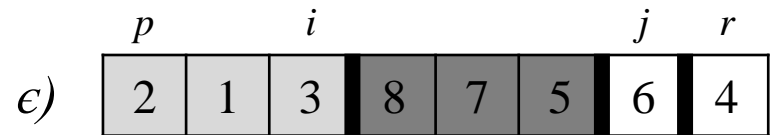
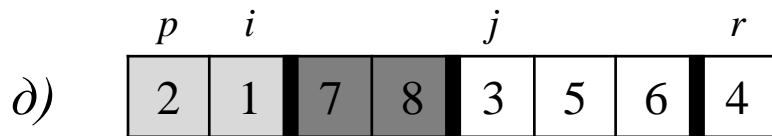
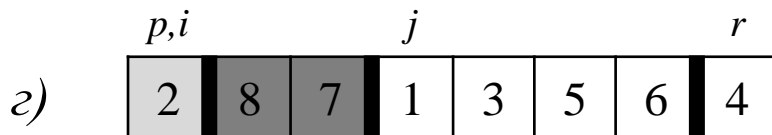
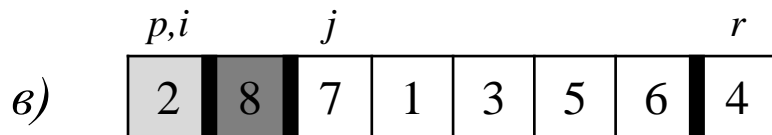
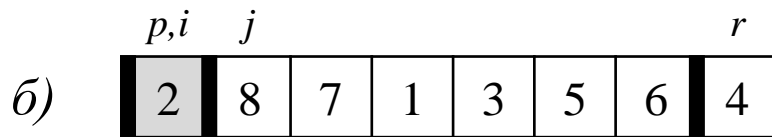
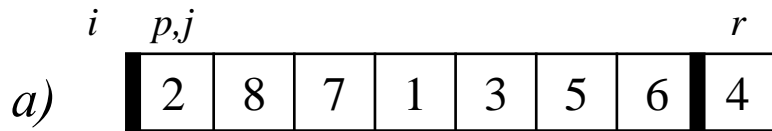
```
1   $x \leq A[r]$ 
2   $i \leq p - 1$ 
3  for  $j \leq p$  to  $r - 1$ 
4      do if  $A[j] \leq x$ 
5          then  $i \leq i + 1$ 
6              Обміняти  $A[i] \Leftrightarrow A[j]$ 
7  Обміняти  $A[i + 1] \Leftrightarrow A[r]$ 
8  return  $i + 1$ 
```

За опорний елемент завжди береться останній:  $x = A[r]$ .



# Швидке сортування

## Приклад роботи алгоритму



# Швидке сортування

Інваріант циклу:

Перед кожною ітерацією для довільного індексу  $k$  виконується:

1. Якщо  $p \leq k \leq i$ , то  $A[k] \leq x$ ;
2. Якщо  $i + 1 \leq k \leq j - 1$ , то  $A[k] > x$ ;
3. Якщо  $k = r$ , то  $A[k] = x$ .

Час роботи процедури Partition для підмасиву  $A[p..r]$  складає  $\Theta(n)$ , де  $n = r - p + 1$

# Швидке сортування

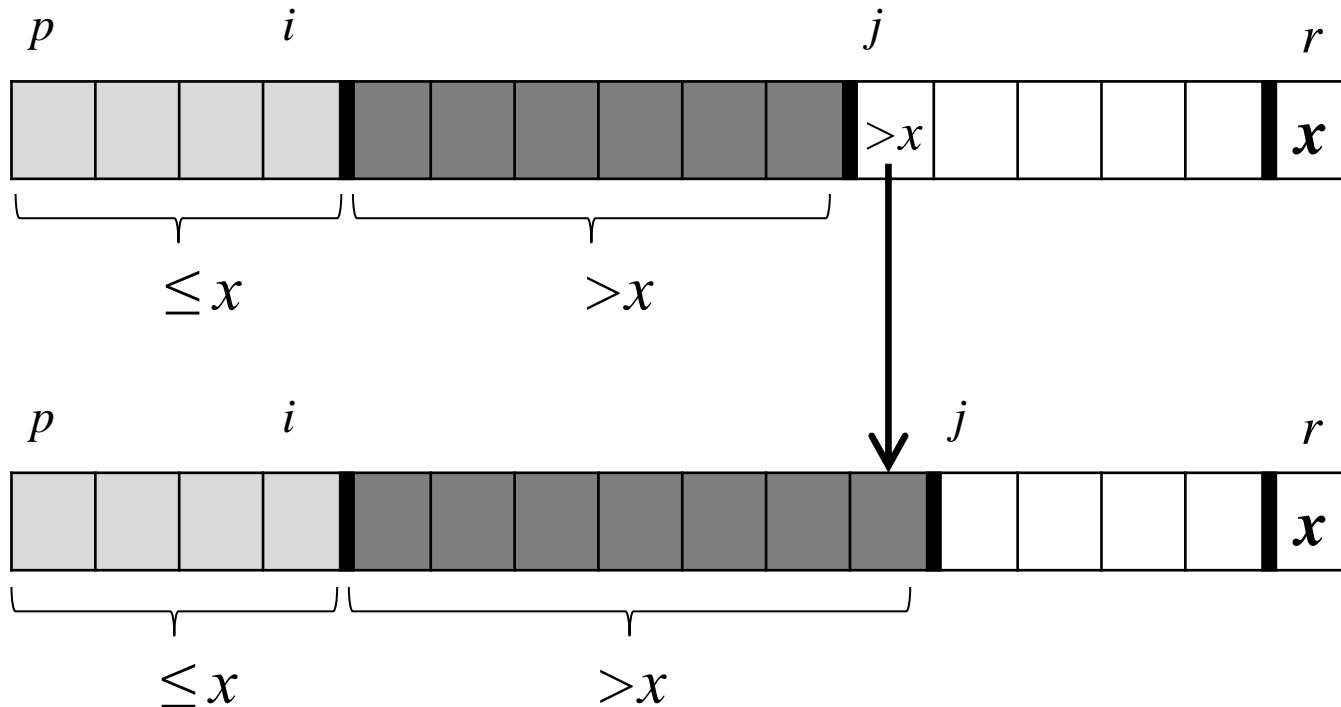
## Коректність роботи процедури Partition.

- *Ініціалізація.* Перед першою ітерацією циклу  $i=p-1$  та  $j=p$ . Між елементами з індексами  $p$  та  $i$ , а також  $i+1$  та  $j-1$  немає елементів, отже перші дві умови інваріанту виконуються. Присвоювання в рядку 1 робить справедливою третю умову (рівність з опорним елементом).

# Швидке сортування

## Коректність роботи процедури Partition.

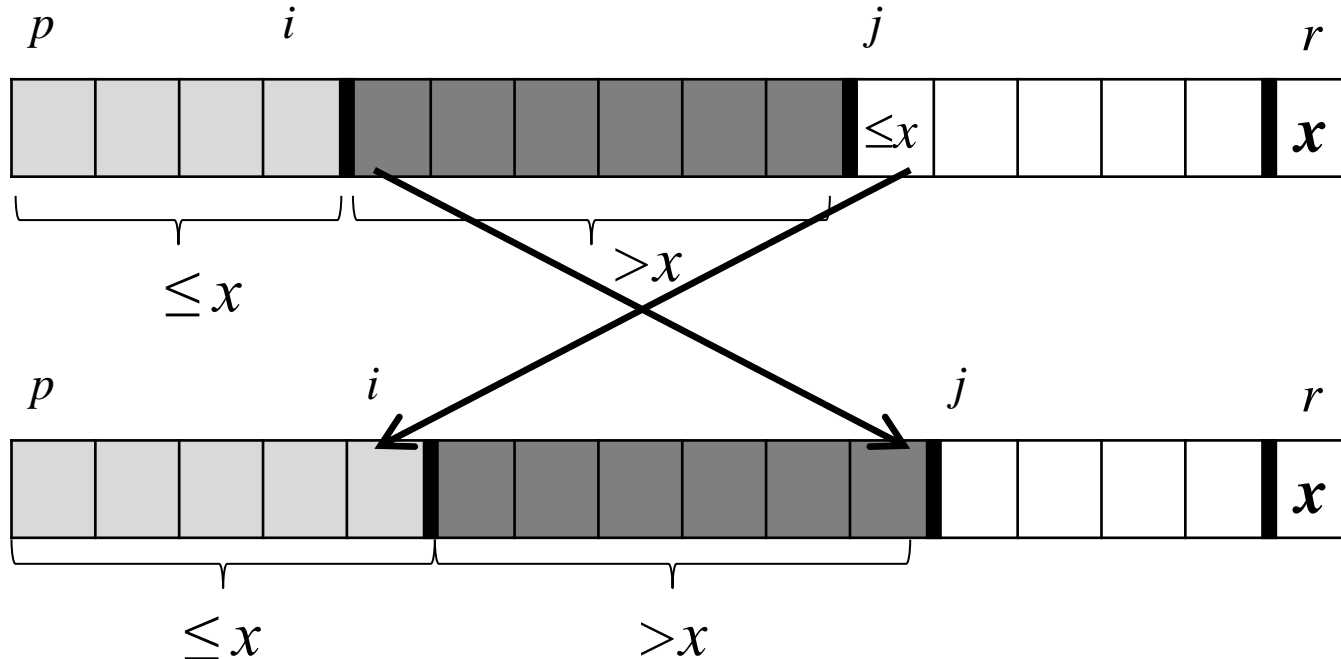
- *Збереження.* Нехай  $A[j] > x$ . Тоді значення  $j$  збільшується на одиницю. При цьому для елемента  $A[j-1]$  виконується умова 2, а інші елементи залишаються незмінними.



# Швидке сортування

## Коректність роботи процедури Partition.

- Збереження (далі). Нехай  $A[j] \leq x$ . Тоді збільшується значення  $i$ , елементи  $A[i]$  та  $A[j]$  міняються місцями і на одиницю збільшується  $j$ . В результаті перестановки  $A[i] \leq x$ , і умова 1 виконується. Аналогічно  $A[j-1] > x$ , бо елемент, що був переставлений з елементом  $A[j-1]$  за інваріантом циклу більший за  $x$ .



# Швидке сортування

## Коректність роботи процедури Partition.

- *Завершення.* По завершенні роботи алгоритму  $j=r$ . Кожен елемент масиву належить до однієї з трьох множин, описаних в інваріанті: ті, що не перевищують  $x$ ; ті, що перевищують  $x$ , і одноелементна множина, що містить  $x$ .



# Швидке сортування

Найгірше розбиття:

Задача постійно розбивається на дві підзадачі величиною  $(n-1)$  та 0 елементів.

Для виконання розбиття потрібен час  $\Theta(n)$ . Робота над порожньою підзадачею виконається за час  $T(0) = \Theta(1)$ .

В описаній ситуації рекурентне співвідношення матиме вигляд

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$

Розв'язком його буде  $T(n) = \Theta(n^2)$ .

Найгірший час роботи буде, зокрема, тоді, коли сортується вже відсортований масив.

# Швидке сортування

Найгірше розбиття:

За допомогою метода підстановок покажемо, що час роботи алгоритму буде  $O(n^2)$ .

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

Нехай  $T(n) \leq cn^2$ , тоді

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) = \\ &= c \cdot \max_{0 \leq q \leq n-1} (q^2 + (n-1-q)^2) + \Theta(n). \end{aligned}$$

Вираз  $q^2 + (n-q-1)^2$  досягає максимуму на кінцях інтервалу  $0 \leq q \leq n-1$ ,

$$\text{Тоді } \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$$

та  $T(n) \leq cn^2 - c(2n-1) + \Theta(n) \leq cn^2$ , тобто  $T(n) = O(n^2)$ .

# Швидке сортування

Найкраще розбиття:

Кожного разу розбиття на підзадачі розміром  $\lfloor n/2 \rfloor$  та  $\lfloor n/2 \rfloor - 1$ .

Час роботи:

$$T(n) \leq 2T(n/2) + \Theta(n).$$

За випадком 2 основної теореми,  $T(n) = O(n \lg n)$ .

# Швидке сортування

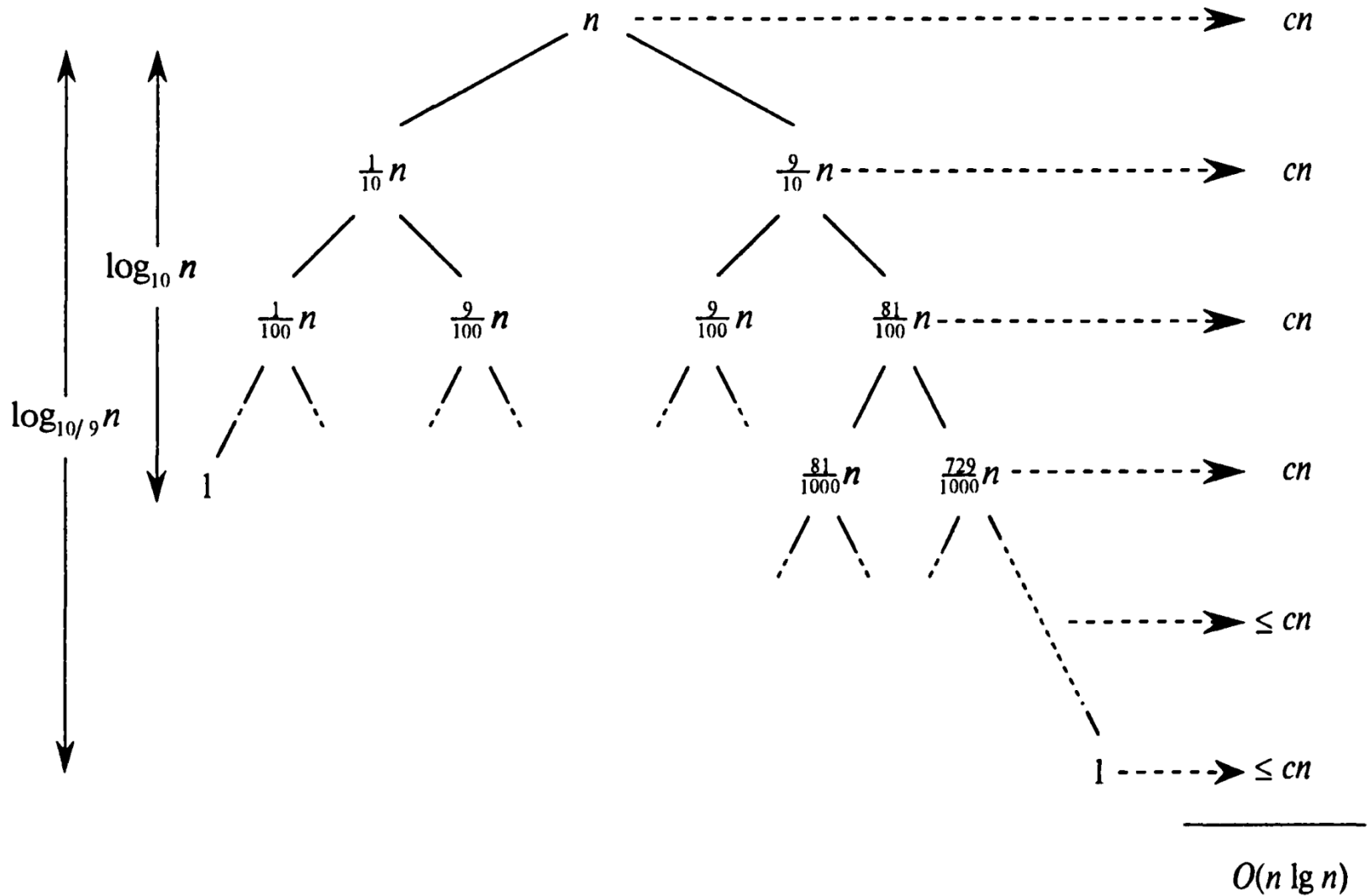
Збалансоване розбиття:

Нехай кожного разу розбиття відбувається у співвідношенні 1 до 9. Тоді

$$T(n) \leq T(9n/10) + T(n/10) + cn,$$

Розглянемо дерево рекурсії, що відповідає цьому рекурентному співвідношенню.

# Швидке сортування



# Швидке сортування

Збалансоване розбиття:

Час роботи все одно буде  $T(n) = O(n \lg n)$

Будь-яке розбиття зі скінченною константою пропорційності призводить до утворення рекурсивного дерева висоти  $\Theta(\lg n)$  й часом роботи кожного рівня  $O(n)$ .

Тобто в будь-якому випадку повний час виконання становитиме  $O(n \lg n)$ .

# Швидке сортування

Середній випадок:

Рекурентне співвідношення виглядатиме

$$T(n) = (n - 1) + \frac{1}{n} \left( \sum_{i=0}^{n-1} (T(i) + T(n - i - 1)) \right), n \geq 2 ,$$
$$T(1) = T(0) = 0$$

Звідси можна вивести співвідношення

$$T(n) = \frac{(n+1)T(n-1) + 2n - 2}{n}, n \geq 2 ,$$
$$T(1) = T(0) = 0$$

Його розв'язок  $T(n) \approx 1.4(n + 1) \log_2 n$ .

Отже, середня складність буде  $O(n \lg n)$ .

Час роботи в середньому лише на  $\approx 40\%$  гірший за ідеальний випадок!

# Швидке сортування

## Проблема вибору опорного елемента

Вибір крайнього елемента спричиняє найгіршу поведінку на вже відсортованому масиві.

- Обирається довільний елемент.
- Вибирається значення з середини розбиття.
- Правило «медіани-з-трьох»: середній між крайніми та центральним елементом.

Вибране значення переміщується на стандартну позицію для опорного елемента.

## Проблема повторів елементів

Масив розбивається на три ділянки: в середині додається область для елементів, рівних опорному.



# Рандомізований варіант Quicksort

Нехай опорний елемент буде вибиратися в масиві  $A[p..r]$  випадковим чином, тобто кожен з елементів може стати опорним з однаковою ймовірністю.

АЛГОРИТМ *RANDOMIZED\_PARTITION* ( $A, p, r$ )

- 1  $i \leftarrow \text{RANDOM}(p, r)$
- 2 Обміняти  $A[r] \Leftrightarrow A[i]$
- 3 **return** *PARTITION*( $A, p, r$ )

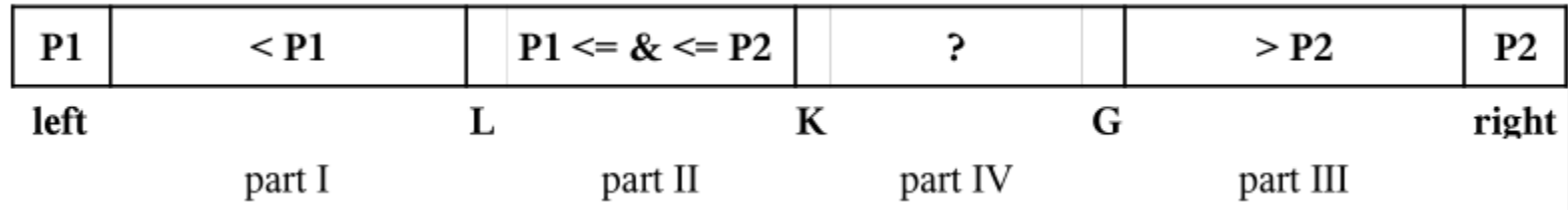
АЛГОРИТМ *RANDOMIZED\_QUICKSORT* ( $A, p, r$ )

- 1 **if**  $p < r$
- 2     **then**  $q \leftarrow \text{RANDOMIZED\_PARTITION}(A, p, r)$
- 3         *RANDOMIZED\_QUICKSORT* ( $A, p, q-1$ )
- 4         *RANDOMIZED\_QUICKSORT* ( $A, q+1, r$ )

Середня складність його буде  $O(n \lg n)$ .

# Quicksort з двома опорними елементами

- Dual-Pivot Quicksort (Yaroslavskiy, Bentley & Bloch).



- Обираються два опорних елементи  $P1 \leq P2$ .
- Масив розбивається на три ділянки:
  - елементи  $< P1$ ,
  - елементи між  $P1$  та  $P2$  (включно),
  - елементи  $> P2$ .
- Реалізація такого підходу буде ефективнішою за звичайне швидке сортування на великих масивах.

## Запитання і завдання

- Покажіть, що в будь-якій  $n$ -елементній піраміді на висоті  $h$  знаходиться не більше  $\lceil n/2^{h+1} \rceil$  вузлів.
- Доведіть коректність алгоритму Heapsort за допомогою наступного інваріанту цикла:  
на початку кожної ітерації в рядках 2-5 підмасив  $A[1..i]$  є незростаючою пірамідою, що містить  $i$  найменших елементів масиву  $A[1..n]$ , а в підмасиві  $A[(i+1)..n]$  містяться  $(n - i)$  відсортованих найбільших елементів масиву  $A[1..n]$ .
- Який час роботи алгоритму пірамідального сортування масиву  $A$  довжини  $n$ , у якого всі елементи відсортовані в порядку зростання? В порядку спадання?
- Покажіть, що час роботи алгоритму пірамідального сортування в найгіршому випадку дорівнює  $\Omega(n \lg n)$ .

## Запитання і завдання

- Доведіть коректність алгоритму `Heap_Increase_Key` за допомогою інваріанта циклу:

перед кожною ітерацією циклу масив  $A[1..heap\_size[A]]$  задовольняє властивості незростаючої піраміди, за винятком одного можливого порушення – елемент  $A[i]$  може бути більше елемента  $A[Parent(i)]$ .

- Створення піраміди через вставку

АЛГОРИТМ *BUILD\_MAX\_HEAP*\*(A)

1 *heap\_size*[A]  $\leftarrow$  1

2 **for**  $i \leq 2$  **to** *length*[A]

3     **do** *MAX\_HEAP\_INSERT*(A, A[i])

1) Чи завжди процедури *Build\_Max\_Heap* та *Build\_Max\_Heap*\* для однакових вхідних масивів створять однакові піраміди? Доведіть це або наведіть контрприклад. 2) Покажіть, що в найгіршому випадку для створення  $n$ -елементної піраміди процедурі *Build\_Max\_Heap*\* треба час  $\Theta(n \lg n)$ .

## Запитання і завдання

- Чому дорівнює час роботи процедури Quicksort у випадку, коли всі елементи масиву  $A$  однакові за величиною?
- Покажіть, що якщо всі елементи масиву  $A$  різняться за величиною і розташовані в порядку спадання, то час роботи процедури Quicksort дорівнює  $\Theta(n^2)$ .
- Доведіть методом підстановок, що розв'язок рекурентного співвідношення на слайді 41

$$T(n) = T(n-1) + \Theta(n)$$

має вигляд  $T(n) = \Theta(n^2)$ .

- Доведіть для рекурентного співвідношення на слайді 42, що  $T(n) = \Omega(n^2)$ .

## Запитання і завдання

- Розбиття масиву за Хоаром. Проаналізуйте первинний алгоритм для розбиття масиву при швидкому сортуванні:

АЛГОРИТМ *HOARE-PARTITION* ( $A, p, r$ )

```
1   $x \leftarrow A[p]$ 
2   $i \leftarrow p - 1$ 
3   $j \leftarrow r + 1$ 
4  while TRUE
5      repeat
6           $j \leftarrow j - 1$ 
7      until  $A[j] \leq x$ 
8      repeat
9           $i \leftarrow i + 1$ 
10     until  $A[i] \geq x$ 
11     if  $i < j$ 
12         Обміняти  $A[i] \Leftrightarrow A[j]$ 
13     else return  $j$ 
```

Такий підхід є ефективнішим за розглянуте розбиття за Ломуто, оскільки робить в середньому втричі менше обмінів, але при цьому є менш прозорим.

Порівняйте кількість здійснених обмінів в двох алгоритмах розбиття при сортуванні (1) відсортованого масиву; (2) масиву, що містить всі однакові елементи.

## Запитання і завдання

- *Аналіз пірамід, відмінних від бінарних.  $d$ -арні піраміди ( $d$ -ary heap) схожі на бінарні, лише їх вузли, відмінні від листя, мають не по 2, а по  $d$  дочірніх елементів. 1) Як би ви представили  $d$ -арну піраміду у вигляді масиву? 2) Як виражається висота  $d$ -арної  $n$ -елементної піраміди через  $n$  та  $d$ ? 3) Розробіть ефективні реалізації процедур Extract\_Max, Insert та Increase\_Key, призначених для роботи з  $d$ -арною незростаючою пірамідою. Проаналізуйте час роботи цих процедур і виразіть їх в термінах  $n$  та  $d$ .*