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:

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

RIGHT(i)
    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

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

Тобто в корені знаходиться *найбільший* елемент.

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

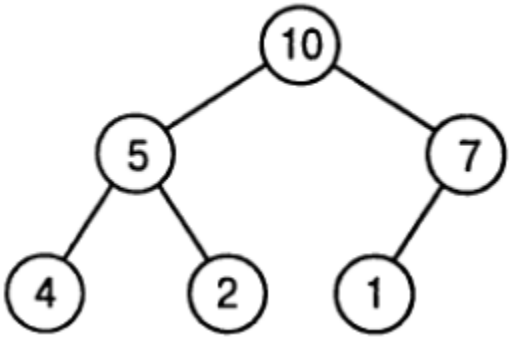
$$A[\text{PARENT}(i)] \leqslant A[i].$$

Тобто в корені знаходиться *найменший* елемент.

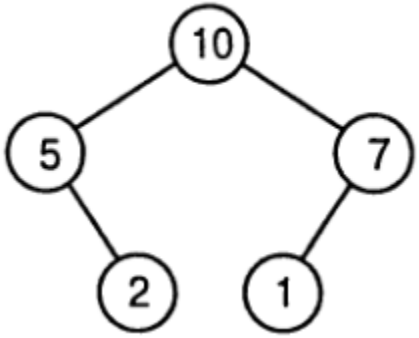
- Висота піраміди – висота її кореня.
- Для n-елементної піраміди це $\Theta(\lg n)$.
- Час виконання основних операцій в піраміді пропорційний висоті дерева.

Пірамідальне сортування (heapsort)

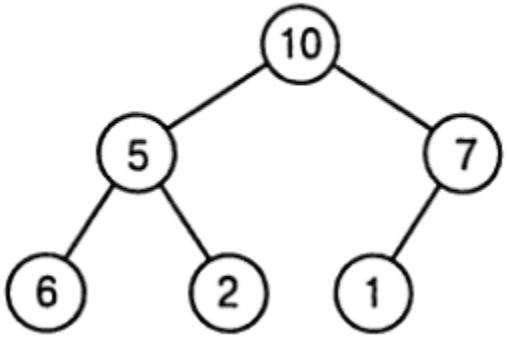
- Чи є вказані дерева пірамідами?



а)



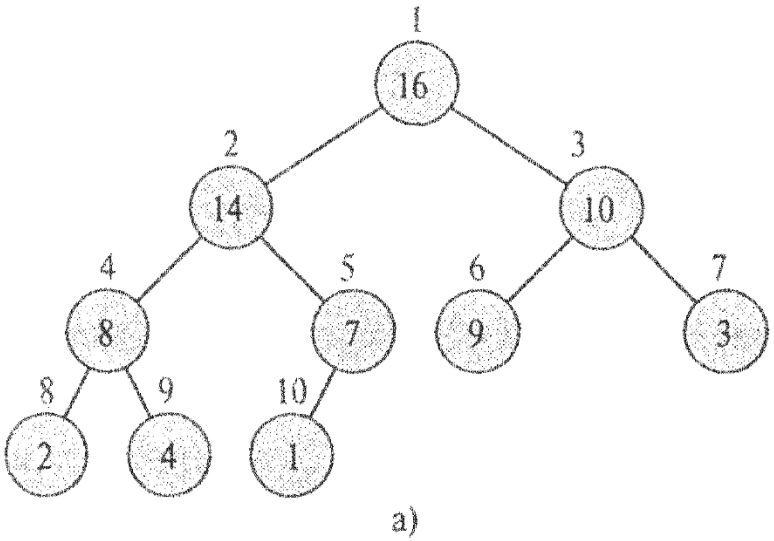
б)



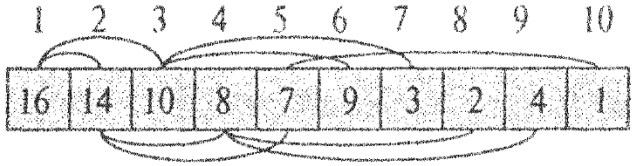
в)

Пірамідальне сортування (heapsort)

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



а)



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)

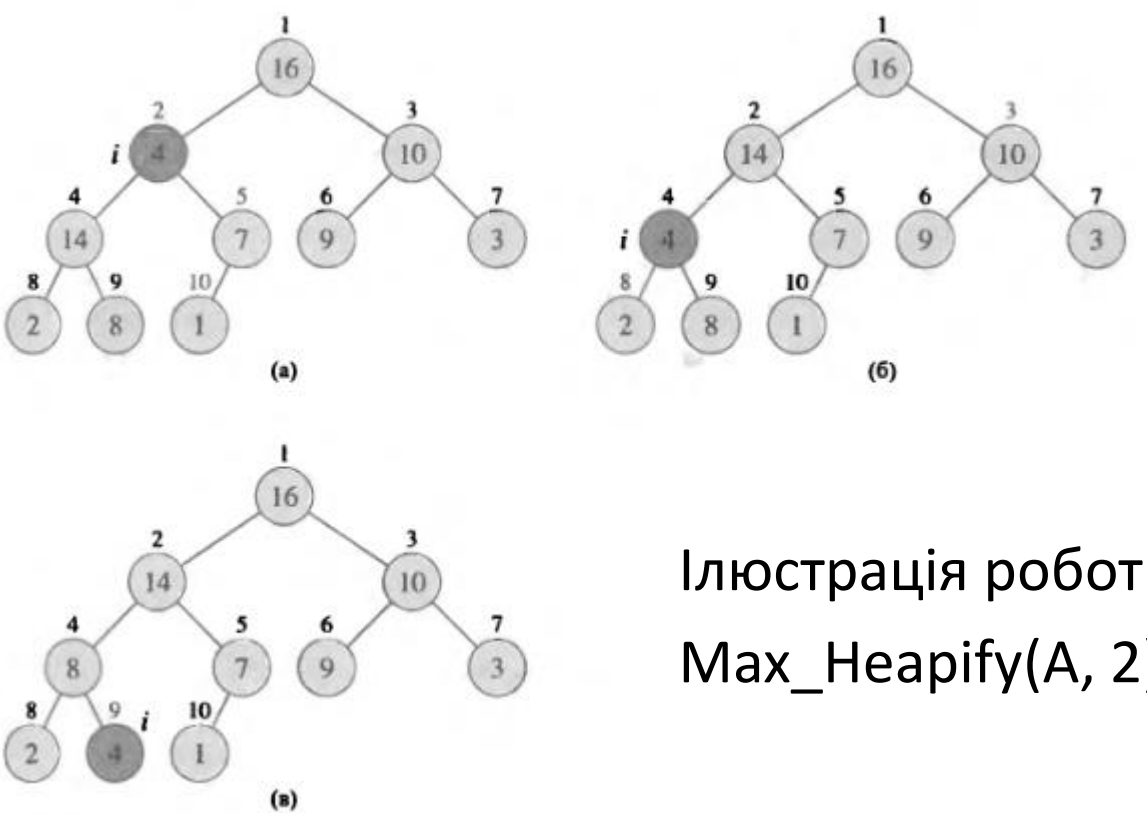
Процедура Max_Heapify

- Розглядається незростаюча піраміда
- Служить для підтримки властивості незростання піраміди

АЛГОРИТМ *MAX_HEAPIFY* (A, *i*)
1 *l* <= LEFT(*i*)
2 *r* <= RIGHT(*i*)
3 **if** $l \le heap\_size[A]$ та A[*l*] > A[*i*]
4 **then** *largest* <= *l*
5 **else** *largest* <= *i*
6 **if** $r \le heap\_size[A]$ та A[*r*] > A[*largest*]
7 **then** *largest* <= *r*
8 **if** $largest \ne i$
9 **then** Обміняти A[*i*] <=> A[*largest*]
10 MAX_HEAPIFY(A, *largest*)

Пірамідальне сортування (heapsort)

- Вважається, що дерева з коренями Left(i) та 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

Max_Heapify(A, 2)

Пірамідальне сортування (heapsort)

Час роботи Max_Heapify:

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

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

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 <= ⌊length[A]/2⌋ downto 1
3      do  MAX_HEAPIFY(A, i)
```

Покажемо коректність алгоритму через інваріант циклу:

який?

Пірамідальне сортування (heapsort)

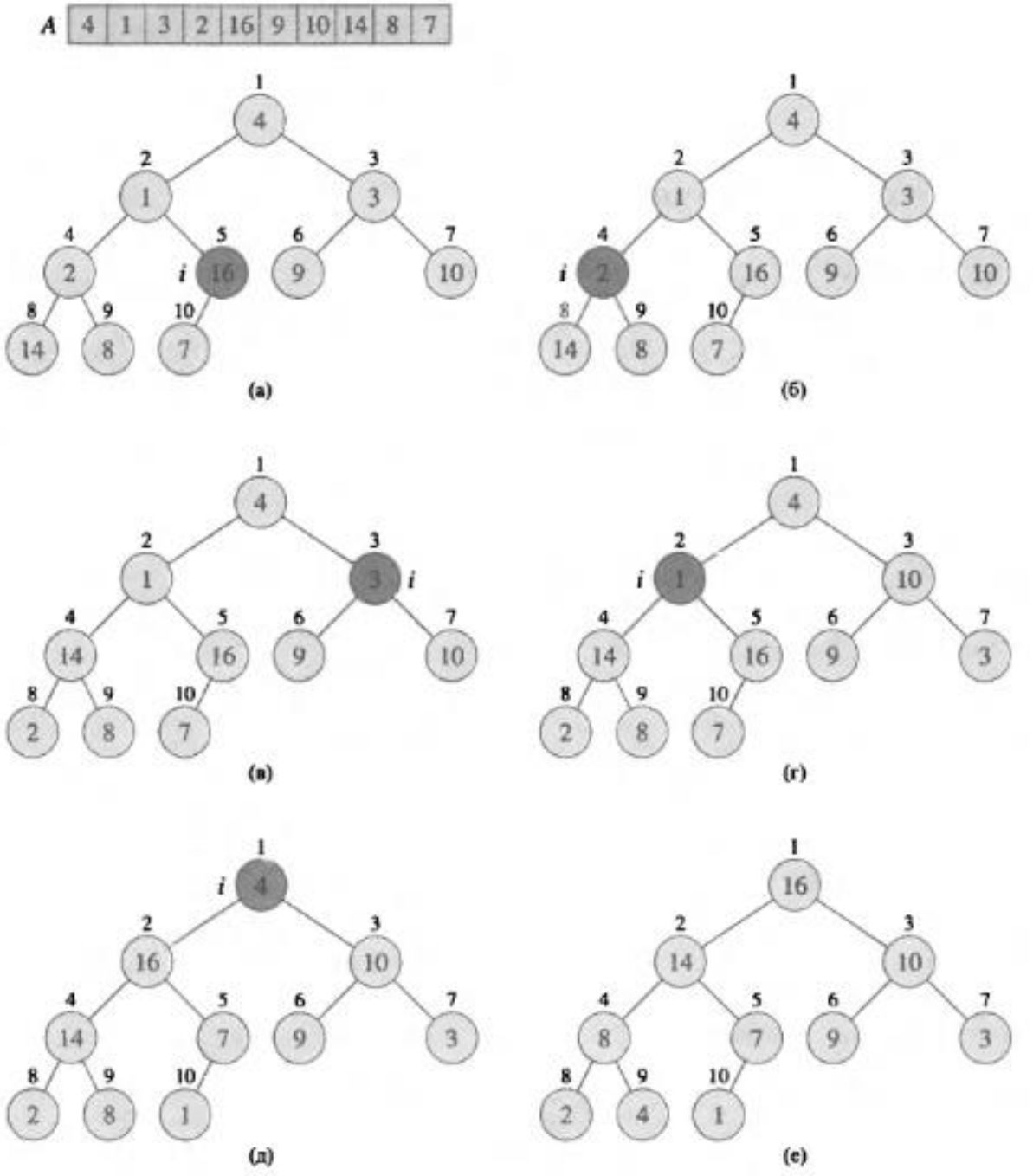
Процедура Build_Max_Heap:

- 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 <= ⌊length[A]/2⌋ 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
(а)
(б)
(в)
(г)
(д)
(е)

Пірамідальне сортування (heapsort)

Коректність Build_Max_Heap:

- *Ініціалізація*: перед першою ітерацією $i = \lfloor n/2 \rfloor$; усі вузли з індексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ – 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 $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ – 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 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 $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ – 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 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,…, *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 $\lfloor \lg n \rfloor$;
- 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 $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ 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:

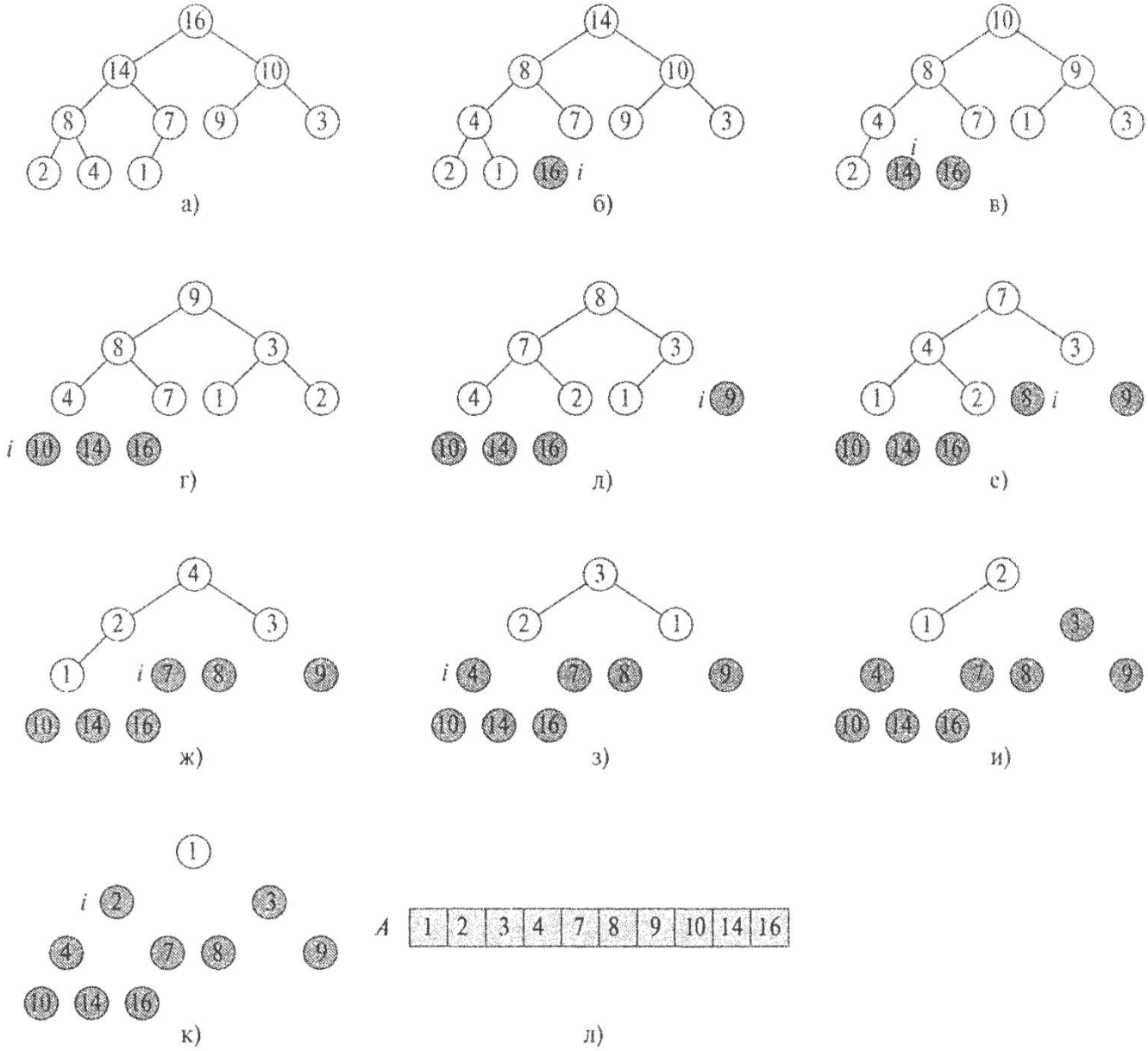
```
АЛГОРИТМ HEAPSORT (A)
1  BUILD_MAX_HEAP(A)
2  for  i <= length[A]  downto  2
3     do  Обміняти A[1] <=> A[i]
4         heap_size[A] <= heap_size[A] – 1
5         MAX_HEAPIFY(A,1)
```

- 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



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:

- Insert(S,x) – додає елемент *x* до множини S.
- Maximum(S) – повертає елемент S з найбільшим ключем.
- Extract_Max(S) – повертає елемент S з найбільшим ключем, видаливши його.
- 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 Θ(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] <= key
4  while  i > 1  та  A[PARENT(i)] < A[i]
5        do  Обміняти  A[i] <=> A[PARENT(i)]
6            i <= PARENT(i)
```

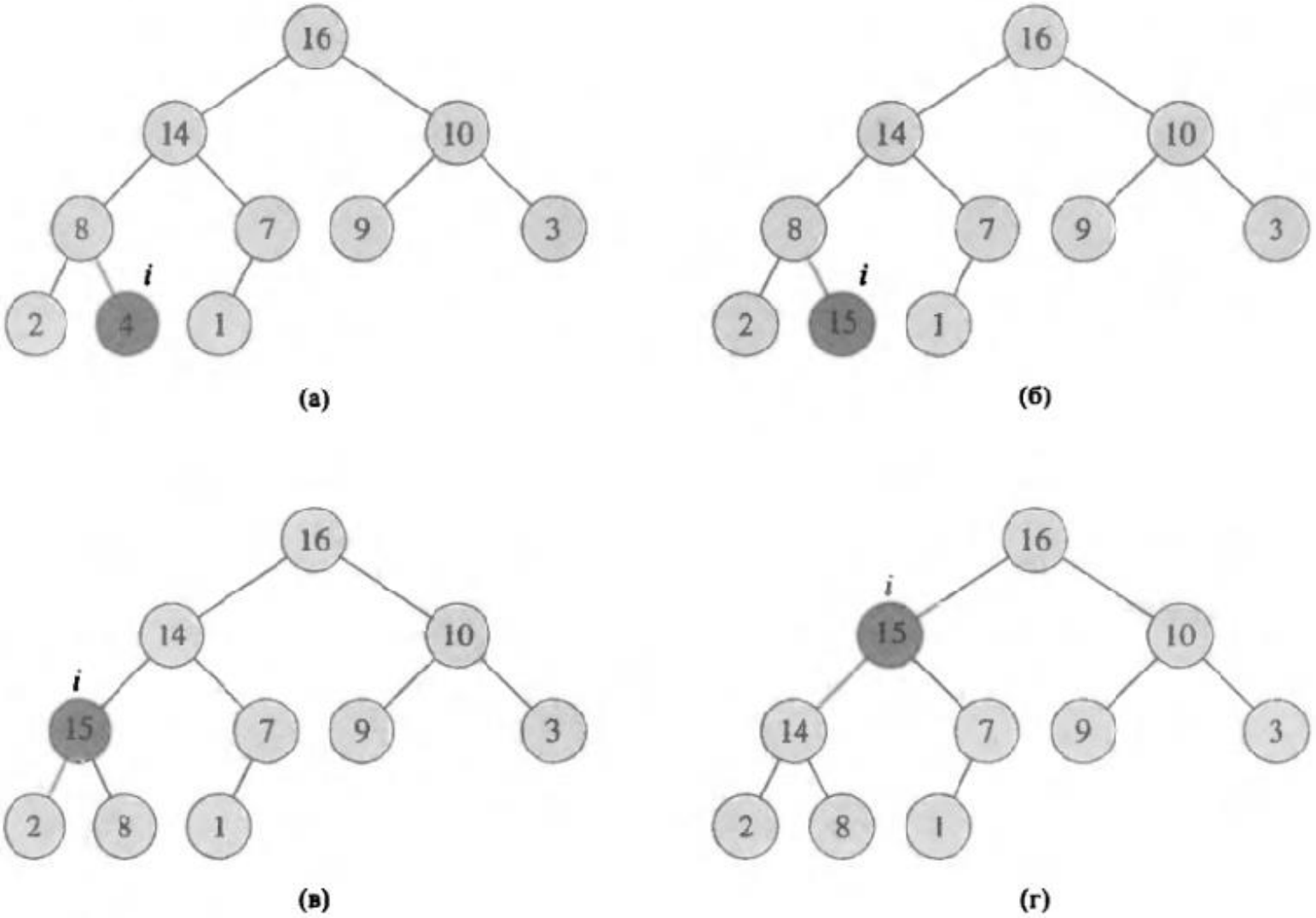
i – позиція в піраміді, *key* – новий ключ.

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 піраміди.

Черги з пріоритетами

Приклад роботи для Heap_Increase_Key(A, *i*, 15), A[*i*]= 4.



Черги з пріоритетами

АЛГОРИТМ *MAX_HEAP_INSERT* (A, *key*)
1 *heap_size*[A] <= *heap_size*[A] + 1
2 A[*heap_size*[A]] <= – ∞
3 HEAP_INCREASE_KEY(A, *heap_size*[A], *key*)

Спочатку створюється новий лист зі значенням −∞, 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 <= PARTITION(A, p, r)
3            QUICKSORT (A, p, q–1)
4            QUICKSORT (A, q+1, r)
```

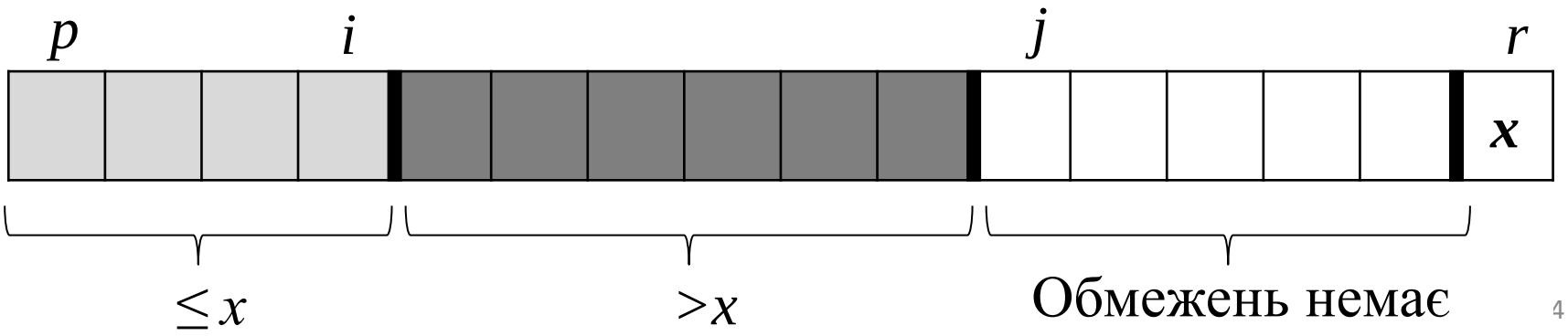
Початковий виклик QUICKSORT(A, 1, *length*[*A*])

Швидке сортування

Розбиття масиву (спосіб Ломуто):

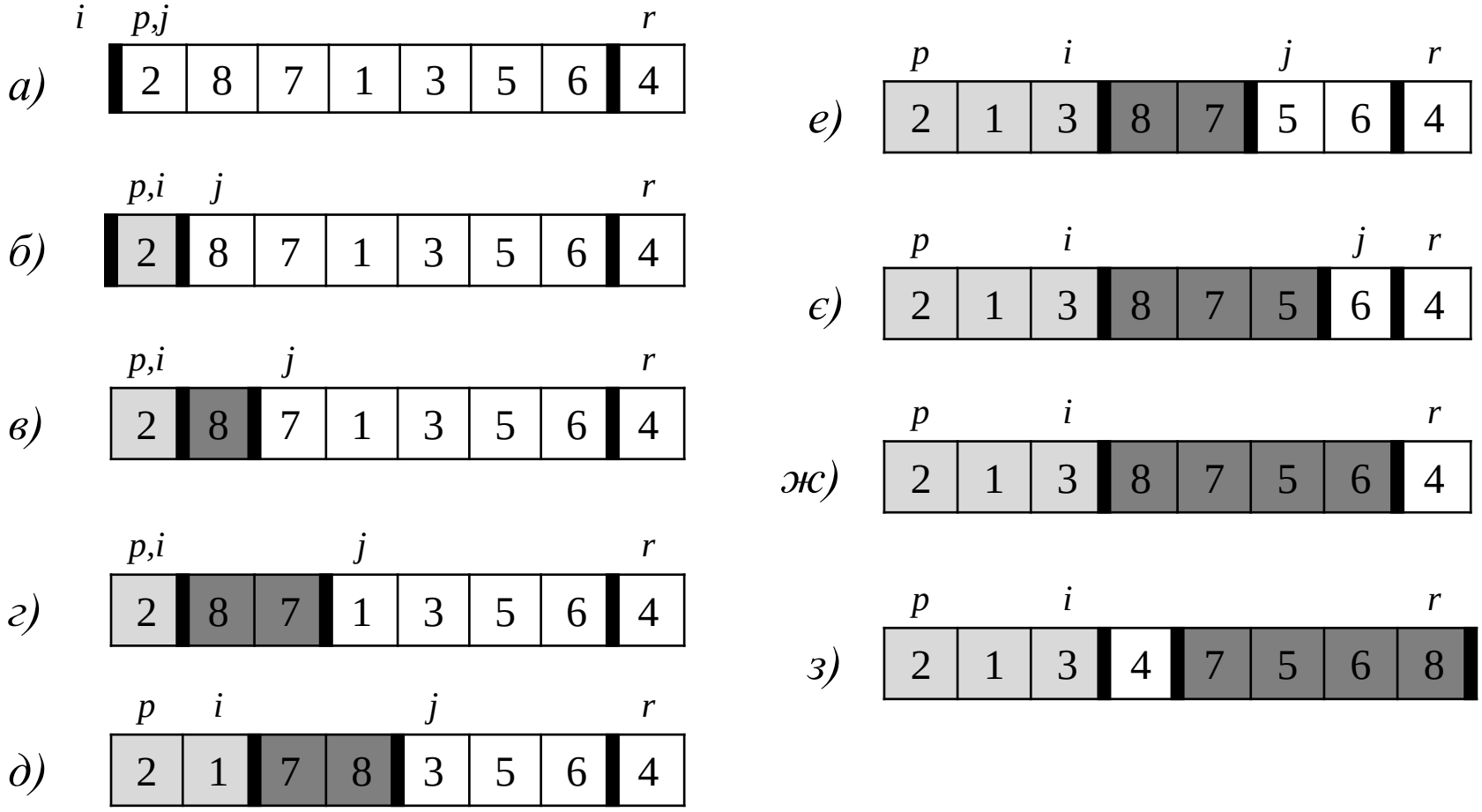
```
АЛГОРИТМ PARTITION (A, p, r)
1  x <= A[r]
2  i <= p – 1
3  for  j <= p  to  r – 1
4       do  if  A[j] ≤ x
5                then  i <= i + 1
6                      Обміняти  A[i] <=> A[j]
7  Обміняти  A[i + 1] <=> A[r]
8  return i + 1
```

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 $p \leqslant k \leqslant i$, то $A[k] \leqslant x$;

2. Якщо $i+1 \leqslant k \leqslant j-1$, 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

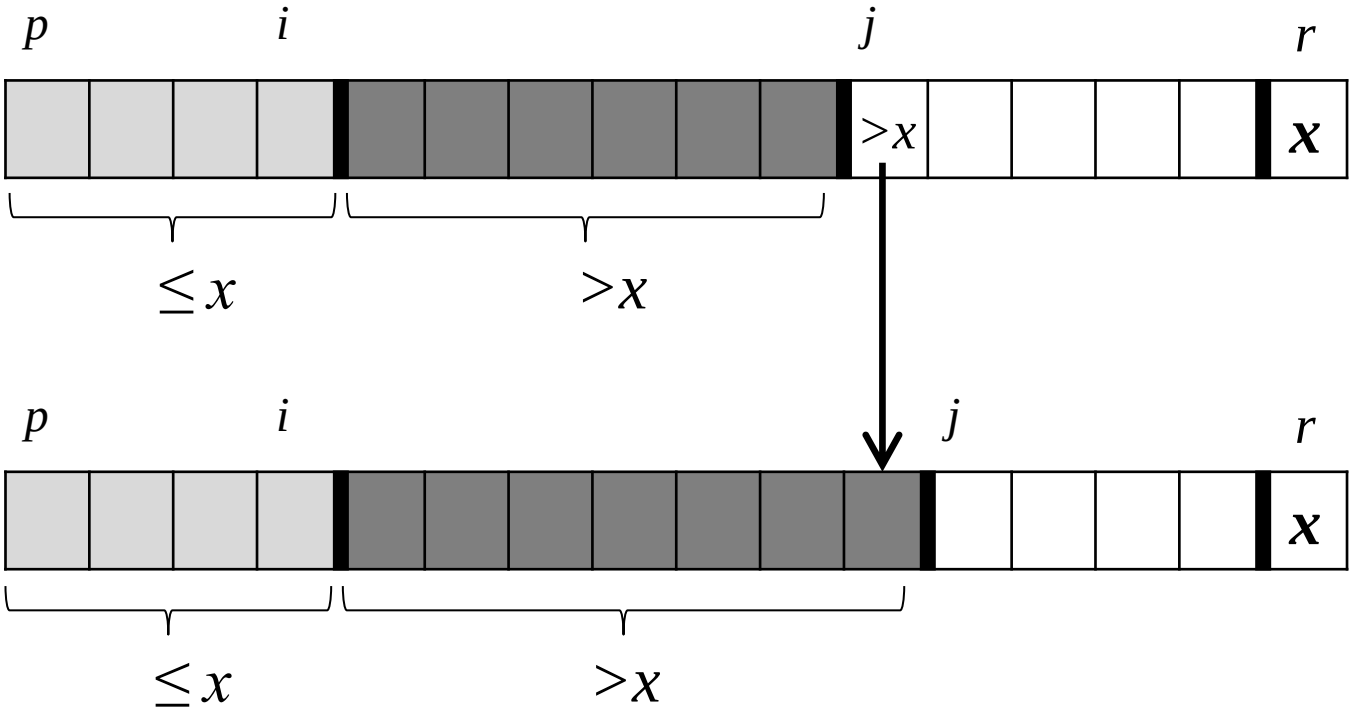
Коректність роботи процедури Partition.

- *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.

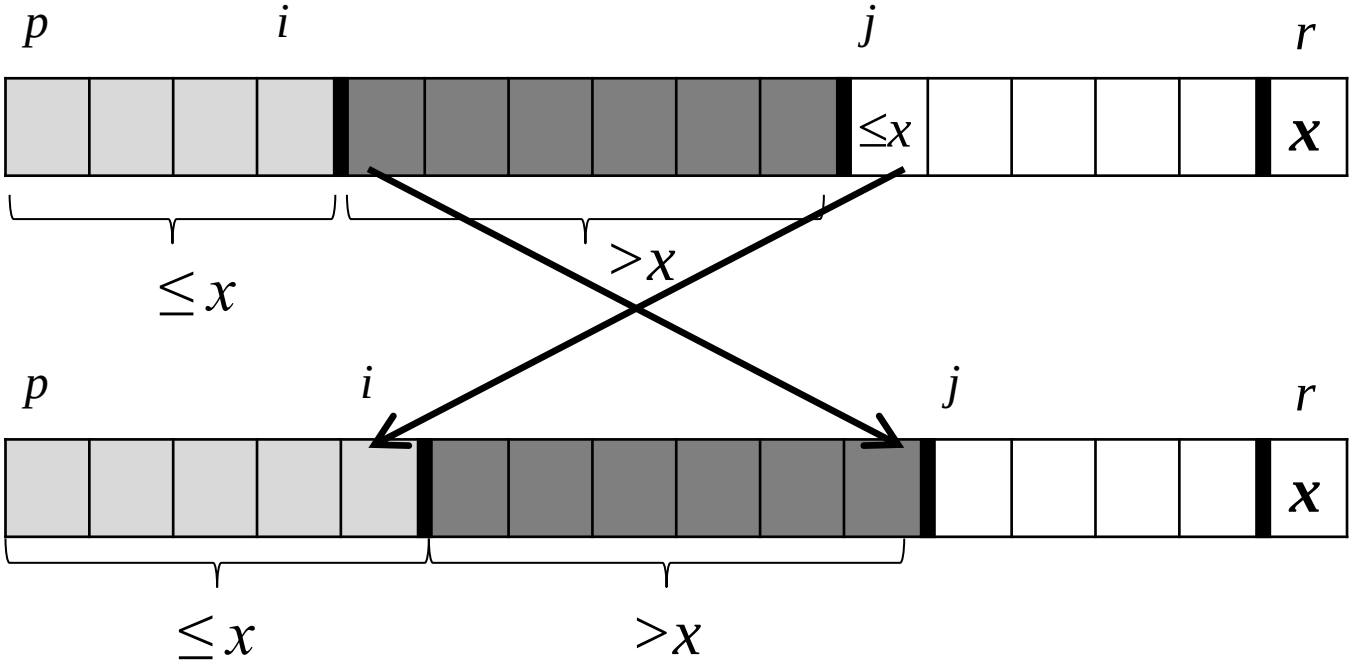
- *Збереження*. Нехай A[*j*]>*x*. Тоді значення *j* збільшується на одиницю. При цьому для елемента A[*j*–1] виконується умова 2, а інші елементи залишаються незмінними.



Швидке сортування

Коректність роботи процедури Partition.

- *Збереження (далі)*. Нехай A[*j*]≤*x*. Тоді збільшується значення *i*, елементи A[*i*] та A[*j*] міняються місцями і на одиницю збільшується *j*. В результаті перестановки A[*i*]≤*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 $O(n^2)$.

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

Нехай $T(n) \leqslant cn^2$, тоді

$$T(n) \leqslant \max_{0 \leqslant q \leqslant n-1} \left(cq^2 + c(n-q-1)^2 \right) + \Theta(n) =$$

$$= c \cdot \max_{0 \leqslant q \leqslant n-1} \left(q^2 + (n-1-q)^2 \right) + \Theta(n).$$

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

Тоді $\max_{0 \leqslant q \leqslant n-1} \left(q^2 + (n-q-1)^2 \right) \leqslant (n-1)^2 = n^2 - 2n + 1$

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

Швидке сортування

Найкраще розбиття:

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

Час роботи:

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

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

Швидке сортування

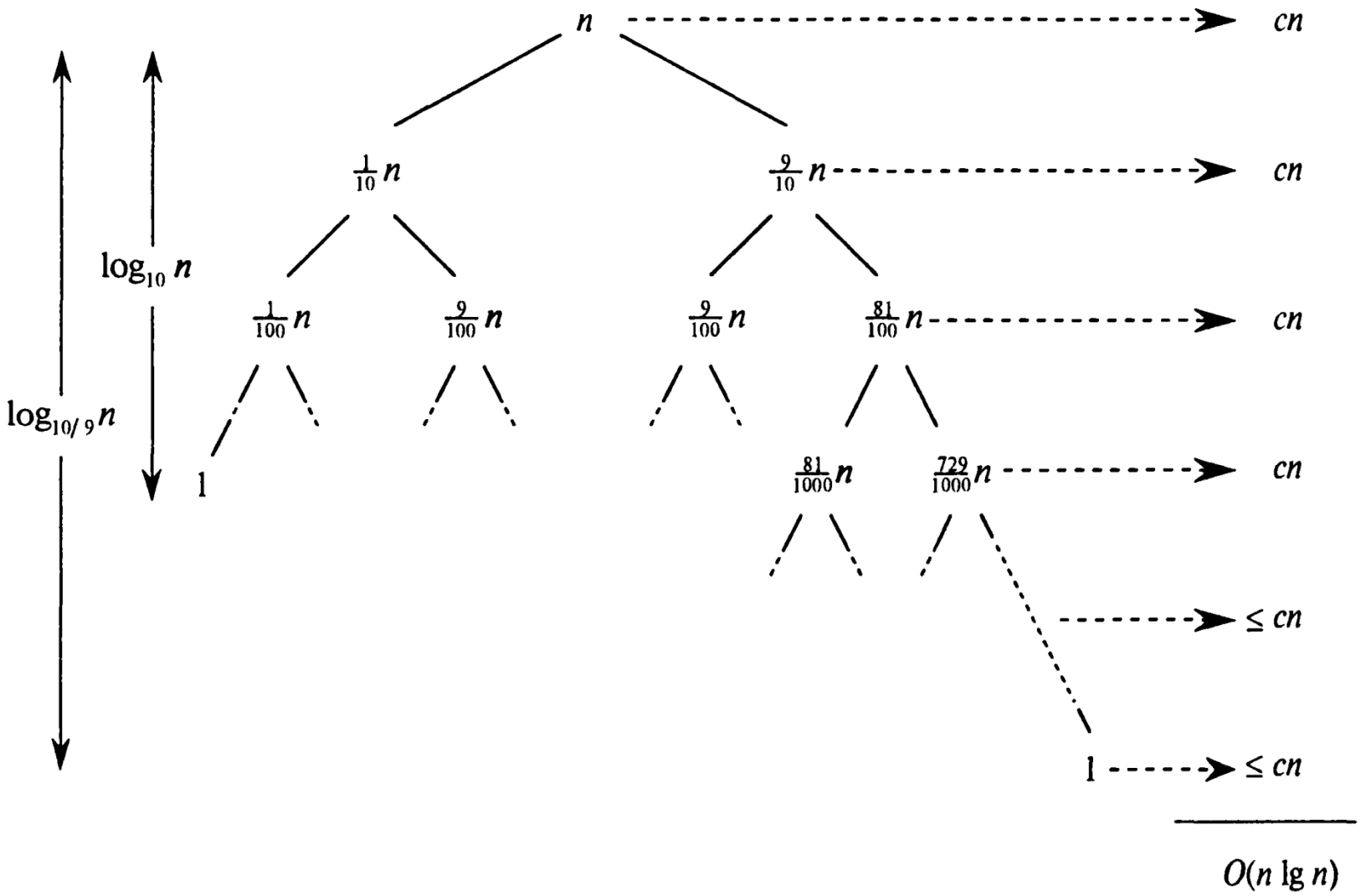
Збалансоване розбиття:

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

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

Розглянемо дерево рекурсії, що відповідає цьому рекурентному співвідношенню.

Швидке сортування



Швидке сортування

Збалансоване розбиття:

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

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

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

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 ≈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.

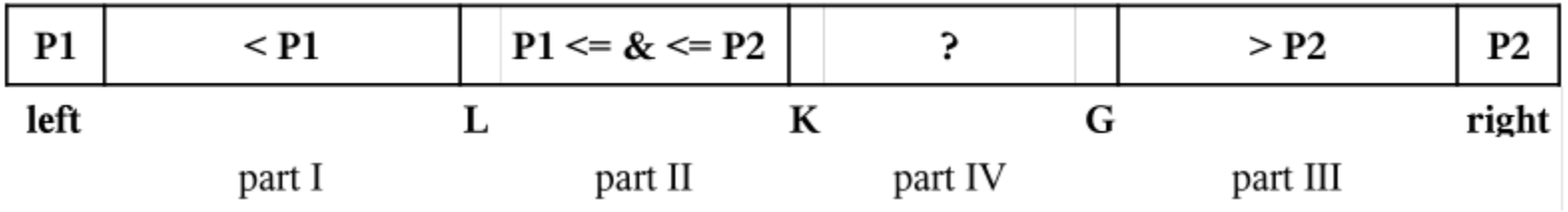
```
АЛГОРИТМ RANDOMIZED_PARTITION (A, p, r)
1   i <= RANDOM(p, r)
2   Обміняти  A[r] <=> A[i]
3   return  PARTITION(A, p, r)

АЛГОРИТМ RANDOMIZED_QUICKSORT (A, p, r)
1   if  p < r
2      then  q <= 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 ≤ 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 ітерацію 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] <= 1
2  for  i <= 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 <= A[p]
2   i <= p – 1
3   j <= r + 1
4   while TRUE
5       repeat
6           j <= j – 1
7       until A[j] ≤ x
8       repeat
9           i <= i + 1
10      until A[j] ≥ x
11      if i < j
12          Обміняти A[i] <=> 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*.