ЛЕКЦІЯ 4. ПІРАМІДИ

4.1. Піраміди

Піраміда або *купа, стіс* (англ. *binary heap*) – це структура даних, яка представляє собою масив, який можна розглядати як майже повне бінарне дерево. Кожний вузол цього дерева відповідає певному елементу масиву. На всіх рівнях, окрім, можливо, останнього, дерево повністю заповнено (заповнений рівень – це такий, який містить максимальну можливу кількість вузлів). Останній рівень заповнюється зліва направо до тих пір, доки в масиві не закінчаться елементи. Масив A, який представляє піраміду, є об'єктом з двома атрибутами: $length[A]$ – кількість елементів масиву, та $heap\_size[A]$ – кількість елементів піраміди, які містяться в масиві A. Іншими словами, не дивлячись на те, що в масиві $A[1 \ldots length[A]]$ всі елементи можуть бути коректними числами, жоден з елементів, які слідують після елементу $A[heap\_size[A]]$, де $heap\_size[A] \le length[A]$, не є елементом піраміди.

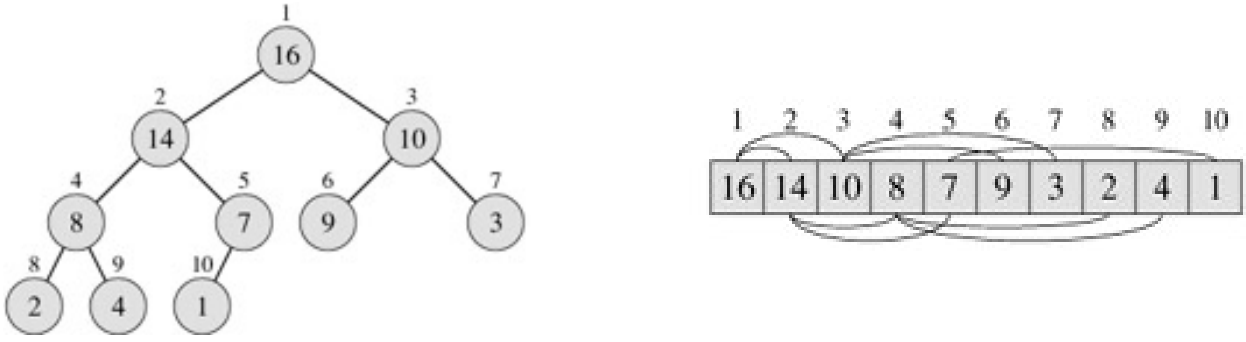
В корні дерева знаходиться елемент $A[1]$, а далі воно будується за наступним принципом: якщо якомусь вузлу відповідає індекс i, то індекс його батьківського вузла обраховується за допомогою наведеної нижче процедури `Parent(i)`, індекс лівого дочірнього вузла – за допомогою процедури `Left(i)`, а індекс правого дочірнього вузла – за процедурою `Right(i)`:

```
Parent(i)
1     return ⌊i / 2⌋

Left(i)
1     return 2i

Right(i)
1     return 2i + 1
```

Лістинг 4.1. Процедури `Parent, Left`та `Right`для піраміди.

В піраміді, яка представлена на рис. 4.1, число в колі є значенням елементу масиву, індекс якого відображається біля відповідного вузла дерева. Лінії, які попарно з'єднують елементи масиву, позначають зв'язок „батько-нащадок".

Батьківські елементи завжди розташовані зліва від дочірніх.



а) *б)*

Рис. 4.1. Піраміда у вигляді *а)* бінарного дерева та *б)* масиву.

На більшості комп'ютерах операція $2i$ в процедурі `Left` виконується за допомогою однієї команди процесору – побітового зсуву числа i на один біт вліво. Операція $2i + 1$ в процедурі `Right` також виконується дуже швидко – достатньо зсунути число i на один біт вліво та додати одиницю в молодший розряд. Процедура `Parent` виконується шляхом зсуву числа i на один біт вправо.

Розрізняють два види бінарних пірамід: *незростаючі* (англ. *max-heaps*) та *неспадні* (англ. *min-heaps*). В пірамідах обох видів значення, які розташовані у вузлах, задовольняють властивості піраміди, яке вирізняє один тип пірамід від інших. Властивість незростаючих пірамід полягає в тому, що для кожного відмінного від кореня вузла з індексом i виконується наступна нерівність:

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

Іншими словами, значення вузла не перевищує значення батьківського для нього вузла. Таким чином, у незростаючих пірамідах найбільший елемент знаходиться в корені дерева, а значення вузлів піддерева, яке бере початок в деякому елементі, не більше значення самого цього елементу.

Принцип організації неспадної піраміди прямо протилежний. Властивість таких пірамід полягає в тому, що для всіх відмінних від кореня вузлів з індексом i виконується нерівність:

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

Таким чином, найменший елемент такої піраміди знаходиться в її корені.

Для сортування масивів можуть використовуватись незростаючі піраміди (алгоритм пірамідального сортування). Неспадні піраміди часто застосовуються у пріоритетних чергах.

Розглядаючи піраміду як дерево, визначимо висоту її вузла як кількість ребер в найдовшому спадаючому простому шляху від цього вузла до одного з листків дерева. Висота піраміди визначається як висота її кореня. Оскільки n-елементна піраміда будується за принципом повного бінарного дерева, то її висота дорівнює $\Theta(\lg n)$. Час виконання основних операцій в піраміді приблизно пропорційний висоті її дерева, і, таким чином, ці операції потребують для роботи часу $O(\lg n)$:

- процедура `MaxHeapify()` виконується за час $O(\lg n)$ і слугує для підтримки властивості незростаючої піраміди;
- час роботи процедуру `BuildMaxHeap()` збільшується зі зростанням кількості елементів лінійно. Ця процедура призначена для створення незростаючої піраміди із невпорядкованого вхідного масиву;
- процедура `Heapsort()` виконується за час $O(n \lg n)$ і сортує масив без використання додаткової пам'яті;
- процедури `MaxHeapInsert()`, `HeapExtractMax()`, `HeapInsertKey()` та `HeapMaximum()` виконуються за час $O(\lg n)$ і дозволяють використовувати піраміду в якості черги за пріоритетами.

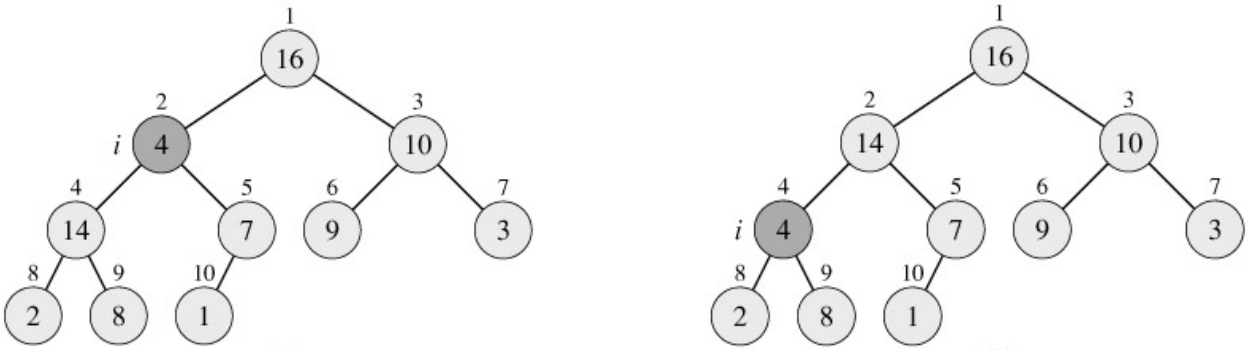
4.2. Підтримка властивості піраміди

Процедура `MaxHeapify()` є важливою підпрограмою, яка використовується для роботи з елементами незростаючих пірамід. На її вхід подається масив A та індекс i цього масиву. При виклику процедури передбачається, що бінарні дерева, коренями яких є елементи `Left(i)` та `Right(i)`, є незростаючими пірамідами, але сам елемент $A[i]$ може бути меншим своїх дочірніх елементів, порушуючи цим властивість незростаючої піраміди. Функція `MaxHeapify()` опускає значення елементу $A[i]$ вниз по піраміді до тих пір, доки піддерево з коренем, що відповідає індексу i, не стане незростаючою пірамідою.

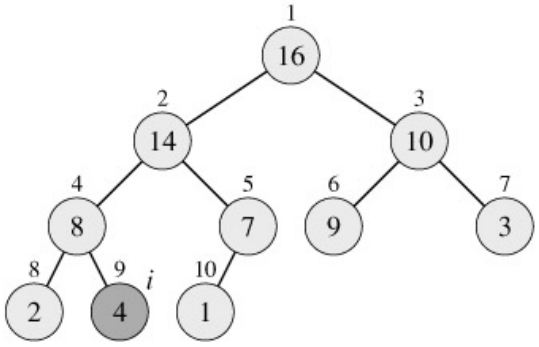
```
MaxHeapify(A, i)
1     l = Left(i)
2     r = Right(i)
3     if l ≤ heap_size[A] та A[l] > A[i]
4           then largest = l
5           else largest = i
6     if r ≤ heap_size[A] та A[r] > A[largest]
7           then largest = r
8     if largest ≠ i
9           then Обміняти A[i] ↔ A[largest]
10          MaxHeapify(A, largest)
```

Лістинг 4.2. Процедура підтримки властивості піраміди `MaxHeapify()`.

Робота процедури MaxHeapify проілюстрована на рис. 4.2. На кожному етапі її роботи визначається, який з елементів – $A[i]$, A[`Left(i)`] чи A[`Right(i)`] – є максимальним, і його індекс присвоюється змінній `largest`. Якщо найбільший елемент – $A[i]$, то піддерево, корінь якого знаходиться у вузлі з індексом i, - незростаюча піраміда, і процедура завершує свою роботу. У протилежному випадку максимальне значення має один з дочірніх вузлів, і елемент $A[i]$ міняється місцями з елементом $A[largest]$. Після цього вузол з індексом i та його дочірні вузли будуть задовольняти властивості незростаючої піраміди. Проте тепер початкове значення $A[i]$ приписане вузлу з індексом `largest`, і властивість незростаючої піраміди може порушитись в піддереві з цим коренем. Для виправлення порушення для цього дерева необхідно рекурсивно викликати процедуру `MaxHeapify()`.



а) *б)*



в)

Рис. 4.2. Робота процедури `MaxHeapify(A, 2)` при *heap_size*[*A*] = 10.

На рис. 4.2 показана робота процедури `MaxHeapify(A, 2)`. В частині *а)* цього рисунку показана початкова конфігурація, в якій елемент *A*[2] порушує властивість незростаючої піраміди, оскільки він менше, ніж обидва дочірніх вузла. Помінявши місцями елементи *A*[2] та *A*[4], ми відновлюємо цю властивість у вузлі 2, проте порушуємо її у вузлі 4 (частина *б)* рисунку). Тепер у рекурсивному виклику процедури `MaxHeapify(A, 4)` в якості параметру виступає значення $i = 4$. Після перестановки елементів *A*[4] та *A*[9] (частина *в)* рисунку) ситуація у вузлі 4 виправляється, а рекурсивний виклик процедури `MaxHeapify(A, 9)` не вносить жодних змін у розглянуту структуру даних.

Час роботи процедури `MaxHeapify()` на піддереві розміру n з коренем в заданому вузлі i обраховується як час $\Theta(1)$, необхідний для виправлення відношень між елементами *A*[*i*], *A*[`Left(i)`] або *A*[`Right(i)`], плюс час роботи цієї процедури з піддеревом, корінь якого знаходиться в одному з дочірніх вузлів вузла i. Розмір кожного з таких дочірніх піддерев не перевищує величину $2n/3$, причому найгірший випадок – це коли останній рівень дерева заповнений наполовину. Таким чином, час роботи процедури `MaxHeapify()` описується наступним рекурентним співвідношенням:

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

Розв'язком цього рекурентного співвідношення згідно з другим випадком основної теореми (тема 4) є $T(n) = O(\lg n)$. По-іншому час роботи процедури `MaxHeapify()` з вузлом, який знаходиться на висоті h, можна виразити як $O(h)$.

4.3. Створення піраміди

За допомогою процедури `MaxHeapify()` можна перетворити масив $A[1\ldots n]$, де $n = length[A]$, у незростаючу піраміду в напрямку знизу вгору. Можна показати, що всі елементи підмасиву $A\left[\left(\lfloor n/2 \rfloor + 1\right)\ldots n\right]$ є листками дерева, тому кожний з них можна вважати одноелементною пірамідою, з якої можна почати процес побудови. Процедура `BuildMaxHeap()` проходить по іншим вузлам та для кожного з них виконує процедуру `MaxHeapify()`:

```
BuildMaxHeap(A)
1    heap_size[A] = length[A]
2    for i = ⌊length[A]/2⌋ downto 1
3         do MaxHeapify(A, i)
```

Лістинг 4.3. Процедура побудови піраміди `BuildMaxHeap`.

Приклад роботи процедури `BuildMaxHeap()` показаний на рис. 4.3. В частині *а)* цього рисунку показаний 10-елементний вхідний масив *A* та його бінарне дерево. З рисунку видно, що перед викликом процедури `MaxHeapify(A, i)` індекс циклу *i* вказує на 5-й вузол. Отримана структура даних показана в частині *б)*. В наступній ітерації індекс циклу *i* вказує на вузол 4. Наступні ітерації циклу **for** в процедурі `BuildMaxHeap()` показані в частинах *в)–д)* рисунку. При виклику процедури `MaxHeapify()` для довільного вузла піддерева з коренями в його дочірніх вузлах є незростаючими пірамідами. В частині *е)* показана незростаюча піраміда, отримана в результаті роботи процедури `BuildMaxHeap()`.

Щоб показати, що процедура `BuildMaxHeap()` працює коректно, скористаємось наведеним нижче інваріантом циклу.

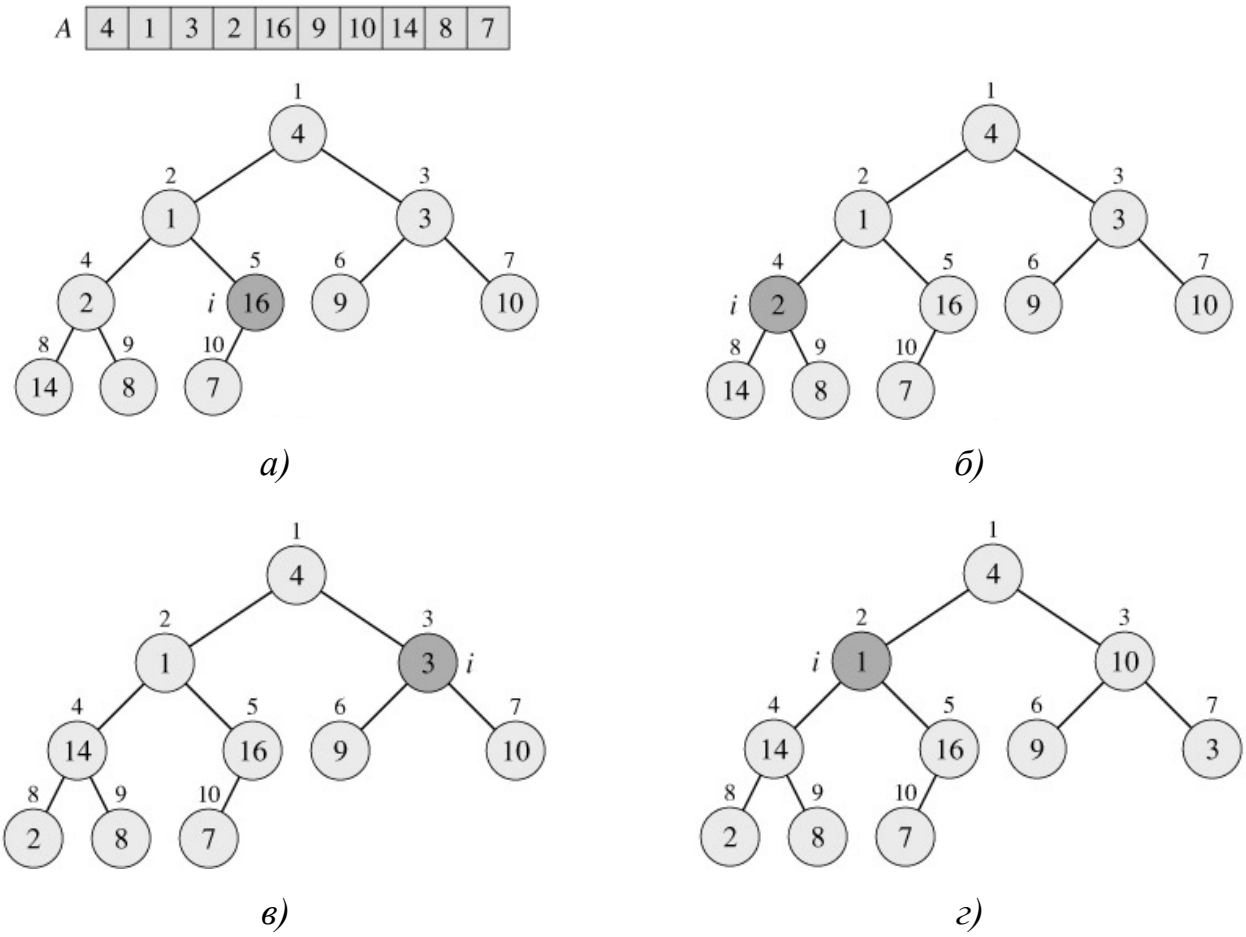
Перед кожною ітерацією циклу **for** в рядках 2-3 процедури `BuildMaxHeap()` всі вузли з індексами $i + 1, i + 2, \ldots, n$ є коренями незростаючих пірамід.

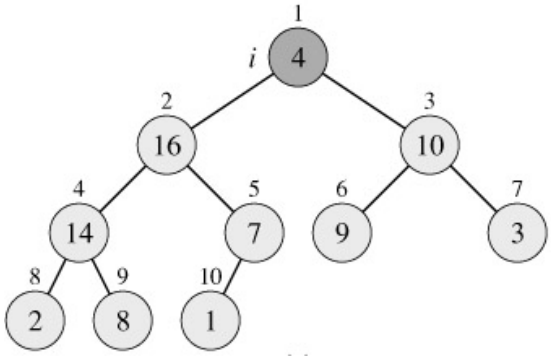
Необхідно показати, що цей інваріант виконується перед першою ітерацією циклу, що він зберігається при кожній ітерації, і що він дозволяє продемонструвати коректність алгоритму після його завершення.

1. *Ініціалізація*. Перед першою ітерацією циклу $i = \lfloor n/2 \rfloor$. Всі вузли з

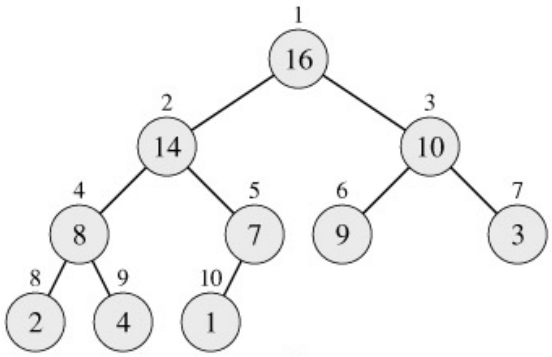
індексами $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ – листки, тому кожний з них є коренем тривіальної незростаючої піраміди.

2. *Збереження*. Щоб пересвідчитись, що кожна ітерація зберігає інваріант циклу, відмітимо, що дочірні по відношенню до вузла i вузли мають номери, які більші i. Згідно інваріанту циклу, обидва ці вузли є коренями незростаючих пірамід. Це саме та умова, яка потрібна для виклику процедури `MaxHeapify(A, i)`, щоб перетворити вузол з індексом i в корінь незростаючої піраміди. Окрім того, при виклику процедури `MaxHeapify()` зберігається властивість піраміди, яка полягає в тому, що всі вузли з індексами $i + 1$, $i + 2, \ldots, n$ є коренями незростаючих пірамід. Зменшення циклу i в циклі **for** забезпечує виконання інваріанту для наступної ітерації.
3. *Завершення*. Після завершення циклу $i = 0$. У відповідності з інваріантом циклу, всі вузли з індексами $1, 2, \ldots, n$ є коренями незростаючих пірамід.





д)



е)

Рис. 4.3 Приклад робота процедури `BuildMaxHeap()`.

Просту верхню оцінку час роботи процедури `BuildMaxHeap()` можна отримати наступним простим способом. Кожний виклик процедури `MaxHeapify()` займає час O(lg*n*) і всього є O(*n*) таких викликів. Таким чином, час роботи алгоритму становить O(*n*lg*n*). Ця верхня границя достатньо коректна, проте не є асимптотично точною.

Щоб отримати більш точну оцінку, відмітимо, що час роботи `MaxHeapify()` залежить від висоти вузла, і при цьому більшість вузлів розташовані на малій висоті. При більш детальному аналізі приймається до уваги той факт, що висота *n*-елементної піраміди становить $\lfloor \lg n \rfloor$ і що на будь-якому рівні, який знаходиться на висоті *h*, міститься не більше $\lceil n / 2^{h+1} \rceil$ вузлів.

Час роботи `MaxHeapify()` при її виклику для вузла, який знаходиться на висоті *h*, дорівнює O(*h*), тому верхню оцінку повного часу роботи процедури `BuildMaxHeap()` можна записати наступним чином

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

Суму в останньому виразі можна оцінити, скориставшись диференціюванням суми спадної геометричної прогресії

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2},$$

і підставивши замість $x = 1/2$. В результаті отримуємо:

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

Таким чином, час роботи процедури `BuildMaxHeap()` можна обмежити наступним чином:

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

Отже, час потрібний для перетворення невпорядкованого масиву у незростаючу піраміду, лінійно залежить від розміру вхідних даних.

Неспадну піраміду можна створити за допомогою процедури `BuildMinHeap()`, яка отримується перетворенням процедури `BuildMaxHeap()` шляхом заміни в рядку 3 виклику функції `MaxHeapify()` відповідної функції `MinHeapify()`. Процедура `BuildMinHeap()` створює неспадну піраміду з невпорядкованого лінійного масиву за час, який лінійно залежить від розмірності вхідних даних.

4.4. Алгоритм пірамідального сортування

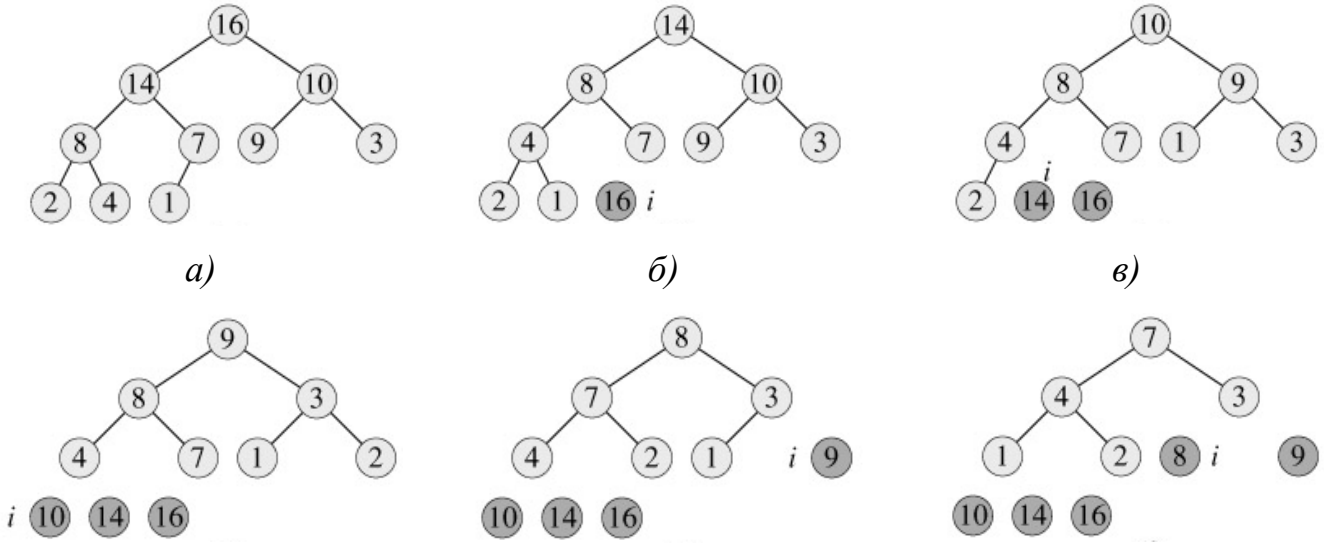
За допомогою піраміди можна виконувати сортування вхідного масиву *A*. Робота пірамідального сортування починається з виклику процедури `BuildMaxHeap()`, за допомогою якої з вхідного масиву $A[1\ldots n]$ створюється незростаюча піраміда. Оскільки найбільший елемент масиву знаходиться в корені, тобто в елементі $A[1]$, його можна помістити у кінцеву позицію у відсортованому масиві, помінявши місцями з елементом $A[n]$. Видаливши з піраміди вузол *n* (шляхом зменшення величини *heap_size*[*A*] на одиницю, підмасив $A[1\ldots(n-1)]$ легко перетворюється на незростаючу піраміду. Піраміди, дочірні по відношенню до кореневого вузла, після обміну елементів $A[1]$ та $A[n]$ і зменшення розміру масиву, лишаються незростаючими, проте новий кореневий елемент може порушувати властивість незростаючих пірамід. Для відновлення цієї властивості достатньо викликати процедуру `MaxHeapify(A, 1)`, після чого підмасив $A[1\ldots(n-1)]$ перетвориться на незростаючу піраміду. Потім алгоритм пірамідального сортування повторює описаний процес для незростаючих пірамід

розміру $n-2, n-3, \ldots, 2$. Нижче наведений псевдокод процедури `HeapSort()` пірамідального сортування.

```
HeapSort(A)
1     BuildMaxHeap(A)
2     for i = length[A] downto 2
3           do Обміняти A[1] ↔ A[i]
4                heap_size[A] = heap_size[A] - 1
5                MaxHeapify(A, 1)
```

Лістинг 4.4. Процедура пірамідального сортування `HeapSort()`.

На рис. 4.4 показаний приклад пірамідального сортування після попередньої побудови незростаючої піраміди. В кожній частині цього рисунку зображена незростаюча піраміда перед виконанням чергової ітерації циклу **for** в рядках 2 – 5. В частині *а)* цього рисунка показана початкова незростаюча піраміда, отримана за допомогою процедури `BuildMaxHeap()`. В частинах *б) – і)* наведені піраміди, які отримані в результаті виклику процедури `MaxHeapify()` в рядку 5. В кожній з цих частин вказані значення індексу *i*. В піраміді містяться тільки ті вузли, які зафарбовані світло-сірим кольором. В частині *и)* показаний відсортований масив *A*.



а) *б)* *в)*

г) *д)* *е)*

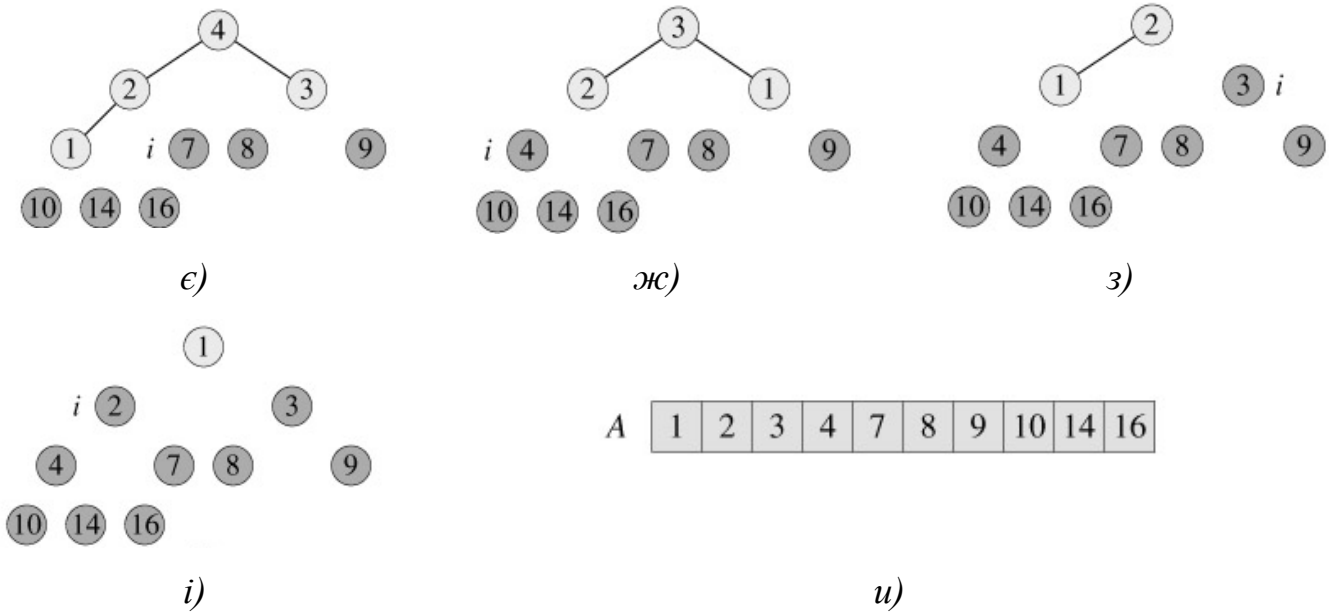


Рис. 4.4. Робота процедури `HeapSort`().

Час роботи процедури `HeapSort()` становить O($n$lg$n$), оскільки виклик процедури `BuildMaxHeap()` потребує часу O(n), а кожний з n – 1 викликів процедури `MaxHeapify()` – часу O(lg$n$).

4.5. Черги з пріоритетами

Як і піраміди, черги з пріоритетами бувають двох видів: незростаючі та неспадні. *Черга з пріоритетами* (англ. *priority queue*) – це структура даних, призначена для обслуговування множини S, з кожним елементом якого пов'язане певне значення, яке називається ключем. У незростаючій черзі з пріоритетами підтримуються наступні операції:

- Операція `Insert(S, x)` вставляє елемент x в множину S.
- Операція `Maximum(S)` повертає елемент множини S з найбільшим ключем.
- Операція `ExtractMax(S)` повертає елемент з найбільшим ключем, при цьому видаляючи його з множини S.
- Операція `IncreaseKey(S, x, k)` збільшує значення ключа, відповідного елементу x, шляхом заміни його значенням k. Передбачається, що величина k не менше поточного ключа елементу x.

Одна з галузей застосування незростаючих черг – планування завдань на комп'ютері, який разом використовується декількома користувачами. Черга

дозволяє слідкувати за завданнями, які потребують виконання, та за їх пріоритетами. Якщо завдання перервано або завершило свою роботу, з черги за допомогою операції `ExtractMax()` обирається наступне завдання з найбільшим пріоритетом. В чергу в будь-який час можна додати нове завдання, використовуючи операцію `Insert()`.

У неспадній черзі з пріоритетами підтримуються операції `Insert()`, `Minimum()`, `ExtractMin()` та `DecreaseKey()`. Черги такого типу можуть використовуватись в моделюванні систем, які керують подіями. В ролі елементів черги в такому випадку виступають змодельовані події, для кожної з яких зіставляється час початку, що відповідає ключу елемента. Елементи повинні моделюватись послідовно згідно часу подій, оскільки процес моделювання може виявити генерацію інших подій. Програма з моделювання обирає наступну подію за допомогою операції `ExtractMin()`. Коли ініціюються нові події, вони розміщуються в чергу за допомогою процедури `Insert()`.

Пріоритетна черга можна реалізувати за допомогою піраміди. В кожному окремому випадку елементи черги з пріоритетами відповідають об'єктам, з якими працює відповідне програмний застосунок. Часто виникає необхідність виявити, який з об'єктів застосунку відповідає тому або іншому елементу черги, чи навпаки. Якщо черга з пріоритетами реалізується за допомогою піраміди, то в кожному елементі піраміди доводиться зберігати ідентифікатор відповідного об'єкту застосунку. Те, яким буде конкретний вигляд цього ідентифікатору, – залежить від застосунку. В кожному об'єкті застосунку так само необхідно зберігати ідентифікатор відповідного елементу піраміди (наприклад, індекс масиву). Оскільки під час операцій над пірамідою її елементи змінюють своє розташування в масиві, при переміщенні елементу піраміди необхідно також оновлювати значення індексу у відповідному об'єкті застосунку.

Розглянемо реалізацію операцій у незростаючій черзі з пріоритетами. Процедура `HeapMaximum()` реалізує виконання операції `Maximum()` за час $\Theta(1)$.

```
HeapMaximum(A)
1    return A[1]
```

Лістинг 4.5. Процедура `HeapMaximum()`.

Процедура `HeapExtractMax()` реалізує операцію `ExtractMax()`. Вона нагадує тіло циклу **for** в рядках 3-5 процедури `HeapSort()`.

```
HeapExtractMax(A)
1    if heap_size[A] < 1
2         then error "Черга порожня"
3    max = A[1]
4    A[1] = A[heap_size[A]]
5    heap_size[A] = heap_size[A] - 1
6    MaxHeapify(A, 1)
7    return max
```

Лістинг 4.5. Процедура виймання максимального елементу з черги `HeapExtractMax()`.

Час роботи процедури `HeapExtractMax()` дорівнює O(lg*n*), оскільки перед запуском процедури `MaxHeapify()`, яка виконується за час O(lg*n*), в ній виконується тільки стала підготовча робота.

Процедура `HeapIncreaseKey()` реалізує операцію `IncreaseKey()`. Елемент черги з пріоритетами, ключ якого буде збільшуватись, ідентифікується у масиві за допомогою індексу *i*. Спочатку процедура оновлює ключ елементу *A*[*i*]. Оскільки це може порушити властивість незростаючих пірамід, після цього процедура проходить шлях від зміненого вузла до кореня в пошуках відповідного місця для нового ключа. Ця операція нагадує реалізовану в циклі процедури `InsertionSort()` (рядки 5 - 7) з теми 2. В процесі обходу виконується порівняння поточного елементу з батьківським. Якщо виявляється, що ключ поточного елементу більше значення ключа батьківського елементу, то відбувається обмін ключами елементів та процедура продовжує свою роботу на більш високому рівні. В протилежному випадку процедура закінчує свою роботу, оскільки їй вдалось відновити властивість незростаючих пірамід.

```
HeapIncreaseKey(A, i, key)
1    if key < A[i]
2         then error "Новий ключ менше поточного"
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)
```

Лістинг 4.6. Процедура збільшення ключа в черзі `HeapIncreaseKey()`.

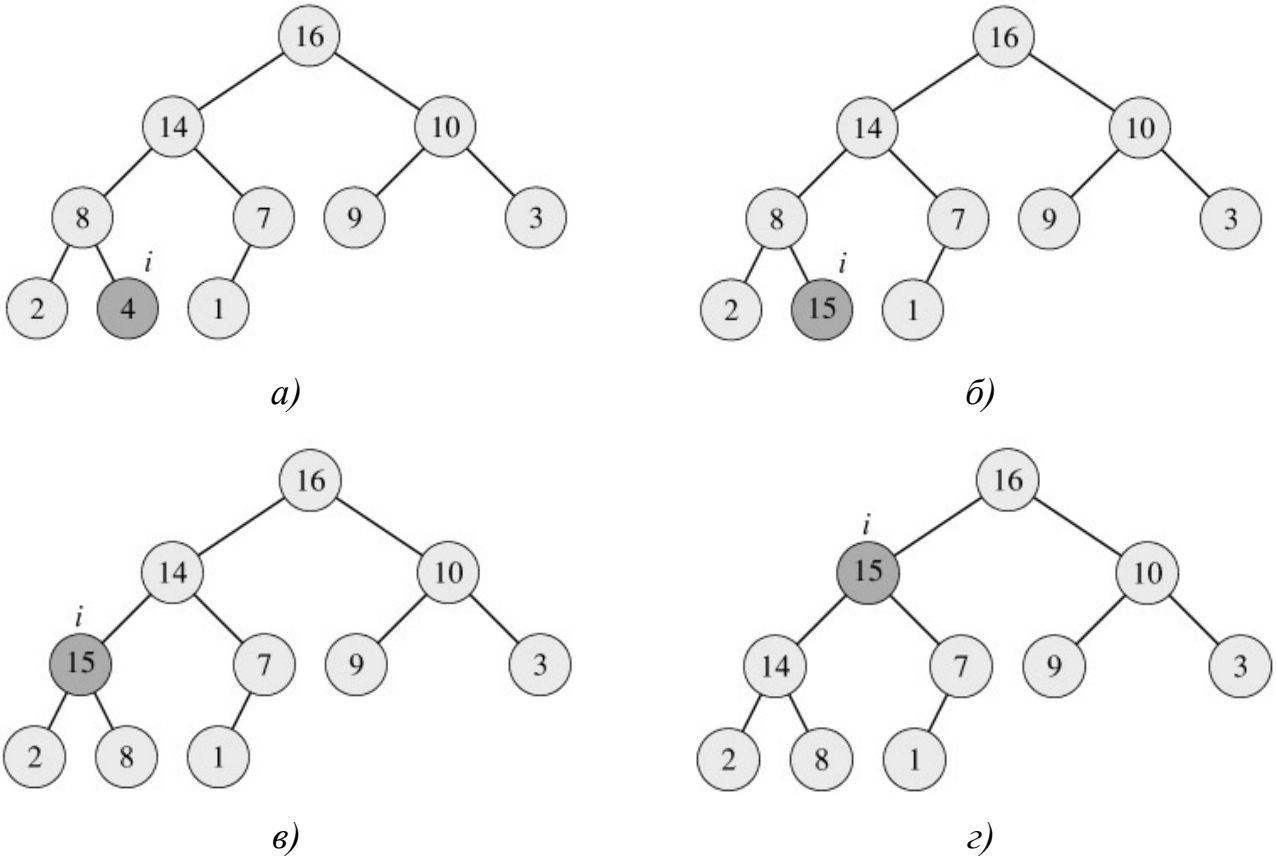


Рис. 4.5. Робота процедури `HeapIncreaseKey()`.

На рис. 4.5 наведений приклад роботи процедури `HeapIncreaseKey()`. В частині *а)* цього рисунку зображена незростаюча піраміда, в якій буде збільшений ключ вузла, виділеного темно-сірим кольором. В частині *б)* показана та сама піраміда після того, як ключ виділеного вузла був збільшений до 15. В частині *в)* піраміда зображена після першої ітерації циклу **while** (рядки 4 - 6). Тут видно, що поточний та батьківський по відношенню до нього вузли обмінялись ключами, і індекс *i* перейшов до батьківського вузла. В частині *г)* показана ця піраміда після ще однієї ітерації циклу. Тепер умова незростаючих пірамід виконується і процедура завершує свою роботу. Час обробки *n*-елементної піраміди за допомогою цієї процедури дорівнює $O(\lg n)$. Це пояснюється тим, що довжина шляху від елементу, який оновлюється в рядку 3 процедури, до кореня дорівнює $O(\lg n)$.

Процедура `MaxHeapInsert()` реалізує операцію `Insert()`. В якості

параметру цій процедурі передається ключ нового елементу. Спочатку процедура додає до піраміди новий листок та присвоює йому ключ із значенням $-\infty$. Потім виконується процедура `HeapIncreaseKey()`, яка присвоює коректне значення ключу та розміщує його у потрібне місце, щоб не порушувалась властивість незростаючих пірамід.

```
MaxHeapInsert(A, key)
1    heap_size[A] = heap_size[A] + 1
2    A[heap_size[A]] = −∞
3    HeapIncreaseKey(A, heap_size[A], key)
```

Лістинг 4.7. Процедура додавання елементу в чергу `MaxHeapInsert()`.

Час вставки в n-елементу піраміду за допомогою процедури `MaxHeapInsert()` становить $O(\lg n)$.

Отже, ми отримали, що в піраміді час виконання всіх операцій з обслуговування черги з пріоритетами дорівнює $O(\lg n)$.

Література

1. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Изд. дом "Вильямс", 2011. – 1296 с. (Глава 6)