Лекція 10. Метод перетворення
План.

1. Визначення.
2. Попереднє сортування.
3. Збалансовані дерева пошуку.
4. Визначення піраміди.
5. Пірамідальне сортування.

1.

Деякі автори називають цю загальну технологію "перетвори й пануй", оскільки такі методи працюють у дві стадії. Спочатку, на стадії перетворення, екземпляр задачі перетвориться в інший, по тій або іншій причині легше піддається рішенню, після чого на стадії "володарювання" вирішується отриманий у результаті перетворення екземпляр задачі.
Є три основних варіанти цього методу, що відрізняються способом перетворення (мал. 6.1).

- Перетворення в більше простий або більше зручний для рішення екземпляр тої ж задачі - спрощення екземпляра.
- Зміна подання наявного екземпляра задачі.
- Приведення задачі, тобто перетворення до екземпляра іншої задачі, для якої є алгоритм рішення.

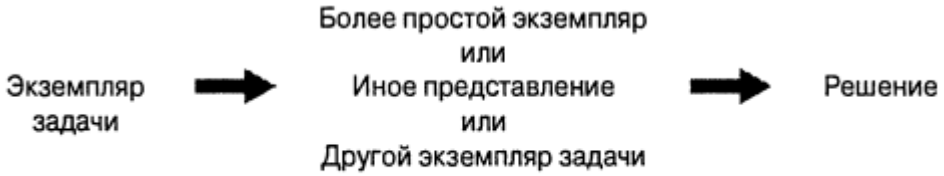


Рис. 6.1. Стратегия “преобразуй и властвуй”

2.

Попереднє сортування.
Попереднє сортування - стара ідея в кібернетиці. Насправді інтерес до алгоритмів сортування в значній мірі обумовлений саме тим, що ряд задач за участю списків вирішуються істотно простіше, якщо списки відсортовані. Зрозуміло, що часова ефективність алгоритму, що включає як етап сортування, може залежати від ефективності використаного алгоритму сортування. Для простоти в цьому розділі ми думаємо, що всі списки реалізовані у вигляді масивів, оскільки багато алгоритмів сортування реалізуються простіше при використанні цього подання.
Ми вже розглядали три елементарних алгоритми сортування — сортування вибором, бульбашкове сортування й сортування вставкою, — які квадратичні як у найгіршому, так і в середньому випадку, і два більше ефективних алгоритми — сортування злиттям, ефективність якого в будь-якому випадку дорівнює $\Theta(n\log n)$, і швидке сортування, ефективність якого в середньому

випадку також дорівнює $\Theta(n \log n)$, але в гіршому — квадратична. Чи є більше швидкі алгоритми сортування? У загальному випадку жоден алгоритм сортування, заснований на порівнянні, не може мати ефективність, що перевищує *nlogn* у найгіршому або в середньому випадку.
Нижче наведені три приклади використання попереднього сортування.

Приклад 1 (Перевірка одиничності елементів масиву).
Алгоритм на основі грубої сили для перевірки того, що всі елементи масиву різні, попарно порівнює всі елементи цього масиву, поки не будуть знайдені два однакових або поки не будуть переглянуті весь можливі пари. У найгіршому випадку ефективність такого алгоритму дорівнює $\Theta(n^2)$.
До рішення задачі можна підійти й по-іншому - спочатку відсортувати масив, а потім порівнювати тільки послідовні елементи: якщо в масиві є однакові елементи, то вони повинні випливати у відсортованому масиві один за іншим.

АЛГОРИТМ $PresortElementUniqueness\,(A\,[0..n-1])$

```
// Проверка единственности элементов массива
// Входные данные:   Массив A[0..n − 1] упорядочиваемых элементов
// Выходные данные: true, если в A нет одинаковых элементов,
//                  и false, если есть
Сортировка массива A
for i ← 0 to n − 2 do
   if A[i] = A[i + 1]
      return false
return true
```

Час роботи даного алгоритму являє собою суму часу, витраченого на сортування, і часу на перевірку сусідніх елементів. Оскільки для сортування потрібно як мінімум *nlog n* порівнянь, а для перевірки сусідніх елементів — не більше *n-1*, саме сортування й визначає загальну ефективність алгоритму. Так, якщо ми використаємо тут квадратичний алгоритм сортування, то алгоритм у цілому виявиться не ефективніше методу грубої сили. Але якщо скористатися гарним алгоритмом сортування, таким як сортування злиттям, ефективність якого в найгіршому разі становлять $\Theta(n \log n)$, то весь алгоритм перевірки одиничності елементів масиву також буде мати ефективність $\Theta(n \log n)$:

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

Приклад 2 (Обчислення моди). Модою (mode) називається значення, що зустрічається в даному списку частіше інших. Наприклад, у випадку значень 5, 1,5, 7, 6, 5, 7 модою є значення 5 (якщо однаково часто зустрічається кілька значень, модою може бути обране кожне з них). Алгоритм на основі грубої сили сканує весь список і обчислює кількість появ у списку кожного з різних значень, після чого шукається найбільше зі знайдених кількостей появ у списку. При реалізації такого підходу зустрінуті значення й кількість їхніх появ можна

зберігати в окремому списку. При кожній ітерації *i-ий* елемент вихідного списку рівняється зі значеннями вже, що зустрічалися елементів, шляхом сканування допоміжного списку. Якщо значення *i-го* елемента є в допоміжному списку, збільшується лічильник кількості елементів; якщо — ні, елемент додається в допоміжний список, а його лічильнику привласнюється значення 1.
Неважко побачити, що в найгіршому випадку вхідні дані являють собою список з неповторюваних елементів. У такому списку його *i-й* елемент рівняється з *i - 1* різними елементами допоміжного списку, перед тим як бути доданим до цього списку. У результаті кількість порівнянь, виконуваних алгоритмом у найгіршому випадку, при створенні допоміжного списку становить

$$C(n) = \sum_{i=1}^{n} (i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)\,n}{2} \in \Theta\left(n^2\right).$$

Крім того, для виявлення елемента з найбільшим значенням лічильника в допоміжному списку потрібно виконати *n - 1* порівнянь, але вони не впливають на квадратичність розглянутого алгоритму в найгіршому випадку.
Розглянемо альтернативний варіант, що починається із сортування списку. У такому випадку всі рівні значення будуть сусідити один з одним, і для обчислення моди треба тільки знайти найбільшу підпослідовність однакових сусідніх значень у відсортованому списку.

Алгоритм $PresortMode\,(A\,[0..n-1])$
// Вычисляет моду массива с использованием
// предварительной сортировки
// **Входные данные:** Массив $A[0..n-1]$ упорядочиваемых элементов
// **Выходные данные:** Мода массива
Сортировка массива A
$i \leftarrow 0$ // Текущее сканирование начинается с позиции i
$modefrequency \leftarrow 0$ // Максимальное количество одинаковых элементов
while $i \leqslant n-1$ **do**
  $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
  while $i + runlength \leqslant n-1$ **and** $A[i + runlength] = runvalue$ **do**
    $runlength = runlength + 1$
  if $runlength > modefrequency$
    $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
  $i \leftarrow i + runlength$
return $modevalue$

Аналіз цього алгоритму аналогічний аналізу алгоритму із приклада 1: час роботи алгоритму визначається часом сортування, оскільки час виконання іншої частини алгоритму — лінійне (чому?). Отже, при використанні сортування, що належить класу ефективності *n log n*, ефективність описаного алгоритму в найгіршому випадку буде вище ефективності в найгіршому випадку алгоритму з використанням грубої сили.
Приклад 3. Обчислення перетину двох множин. (Самостійно)

3.
Збалансовані дерева пошуку

Бінарне дерево пошуку — це бінарне дерево, вузли якого містять елементи множини елементів, що впорядковуються, по одному елементі у вузлі, причому всі елементи у лівому піддереві менше елемента в корені піддерева, а елементи в правому піддереві - більше його. Відзначимо, що таке перетворення множини в бінарне дерево пошуку являє собою приклад *методу зміни подання*. Чого ми досягаємо таким перетворенням у порівнянні із простою реалізацією словника, наприклад, за допомогою масиву? Ми одержуємо більше **високу ефективність пошуку, вставки й видалення** — час виконання всіх цих операцій дорівнює $\Theta(\log n)$, але тільки в середньому випадку. У найгіршому випадку ці операції виконуються за час $\Theta(n)$, оскільки дерево може виродитися в повністю незбалансоване, з висотою, рівної *n - 1*.

Необхідно було знайти структуру даних, що зберігає важливі властивості класичних бінарних дерев пошуку, - у першу чергу логарифмічну ефективність словникових операцій і відсортованість елементів, - але при цьому уникає вирожденності в найгіршому випадку. Для цього використаються два підходи.
Перший підхід являє собою варіант спрощення екземпляра задачі — незбалансоване бінарне дерево пошуку перетвориться в збалансоване. Конкретні реалізації цієї ідеї розрізняються по їхніх визначеннях того, що таке збалансованість. **AVL-дерево** (AVL tree) вимагає, щоб різниця висот лівого й правого поддеревьев кожного вузла не перевищувала 1. **Червоно-чорне дерево** (red-black tree) допускає, щоб висота одного піддерева була у два рази більше висоти іншого піддерева того ж самого вузла. Якщо вставка нового вузла або видалення наявного приводить до того, що порушується умова збалансованості, таке дерево перебудовується за допомогою одного із сімейства спеціальних перетворень, які називаються поворотами (rotation) і які відновлюють умови збалансованості. Ми розглянемо тільки AVL-дерева.
Другий підхід являє собою варіант зміни подання: допускається наявність більш ніж одного елемента у вузлі дерева пошуку. Окремими випадками таких дерев є **2-3-дерева**, **2-3-4- дерева** й більше загальний і важливий випадок - **B-дерева**. Вони розрізняються кількістю елементів, які припустимі в одному вузлі дерева пошуку, але всі вони є ідеально збалансованими.

AVL-дерева

AVL-дерева були відкриті в 1962 р. двома радянськими математиками - Г.М. Адельсон-Вельским і Е.М. Ландисом; винайдена структура одержала назву по перших буквах їхніх прізвищ.
Визначення 1. AVL-дерево являє собою бінарне дерево пошуку, у якому показник збалансованості (balance factor) кожного вузла, обумовлений як різниця висот лівого й правого піддерев вузла, дорівнює 0, +1 або -1 (висота порожнього дерева вважається рівною -1).

Наприклад, бінарне дерево пошуку на мал. 6.2а - AVL-дерево, у той час як бінарне дерево пошуку на мал. 6.2б таким не є.

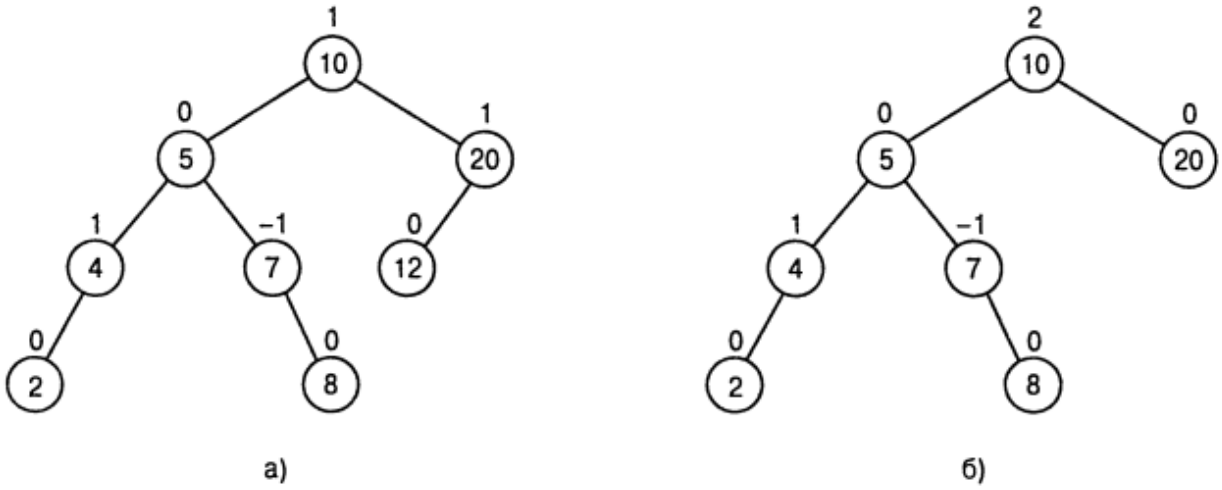
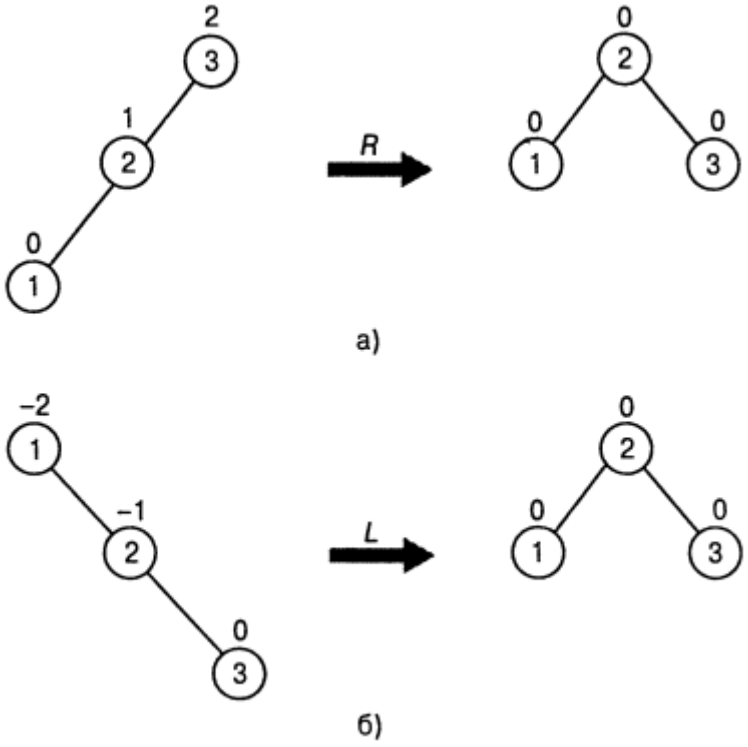
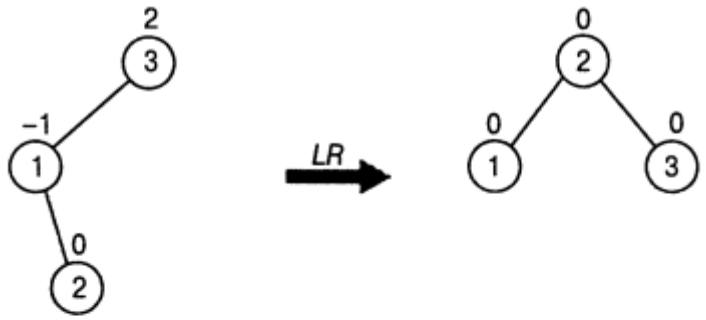


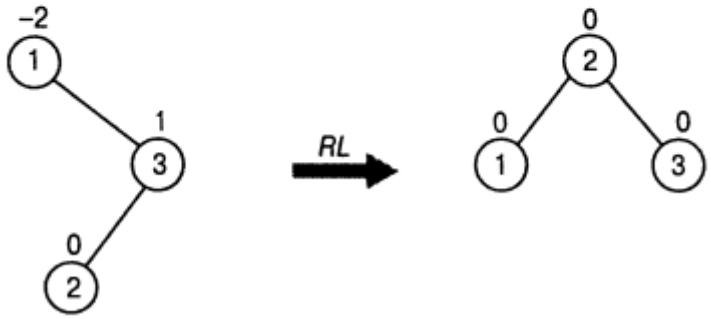
Рис. 6.2. а) AVL-дерево. б) Бинарное дерево поиска, не являющееся AVL-деревом. Показатели сбалансированности показаны над узлами деревьев

Якщо вставка нового вузла робить AVL-дерево незбалансованим, воно перетвориться за допомогою **повороту**. Поворот в AVL-дереві являє собою локальне перетворення піддерева, корінь якого має показник збалансованості, рівний +2 або -2; якщо таких вузлів трохи, ми повертаємо дерево з незбалансованим коренем, що найбільш близький до знову вставленого аркуша. Усього є тільки чотири типи поворотів, причому два з них являють собою дзеркальне відбиття двох інших. У найпростішій формі чотири можливих повороти показані на мал. 6.3.





в)



г)

Рис. 6.3. Четыре типа поворотов AVL-деревьев с тремя узлами. а) Одиночный R-поворот. б) Одиночный L-поворот. в) Двойной LR-поворот. г) Двойной RL-поворот

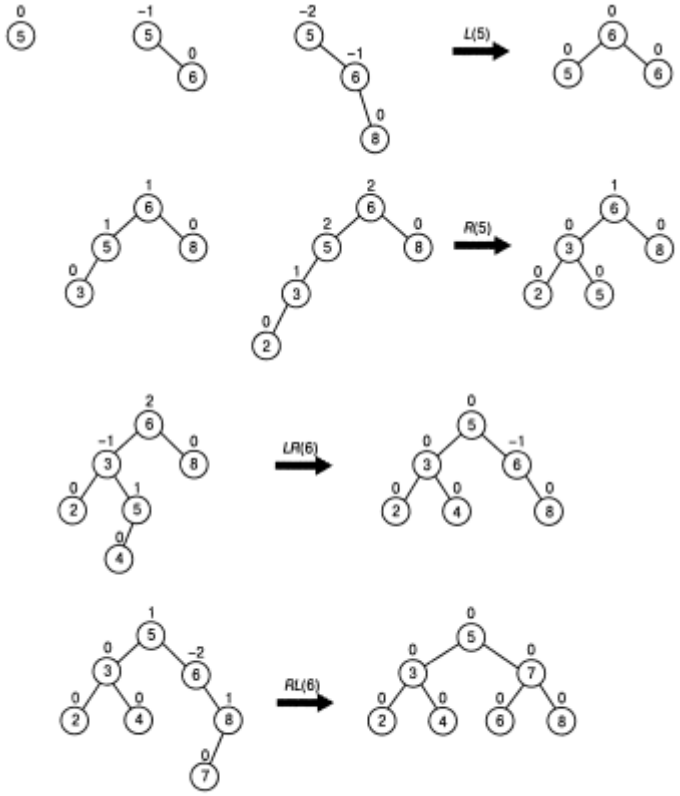


Рис. 6.6. Построение AVL-дерева путем последовательной вставки чисел из списка 5, 6, 8, 3, 2, 4, 7. Число в скобках после указания типа поворота — корень перестраиваемого дерева

2-3-дерева

Друга ідея балансування дерев пошуку полягає в тім, щоб дозволити вузлу одночасно містити кілька ключів. Найпростішою реалізацією цієї ідеї є 2-3-дерева, розроблені в 1970 р. американським кібернетиком Дж. Хопкрофтом (John Hopcroft). **2-3-дерево** являє собою дерево, що може мати вузли двох видів — 2-вузли й 3-вузли. **2-вузол** містить єдиний ключ *К* и має два нащадків: лівий дочірній вузол служить коренем поддерева, всі ключі в якому менше *К*, а правий — коренем піддерева, всі ключі в якому більше *К*. (Інакше кажучи, 2-вузол точно такиї ж, як і вузол у класичному бінарному дереві пошуку.) **3-вузол** містить два впорядкованих ключі $К_1$ і $К_2$ ($К_1 < К_2$) і має три дочірніх вузли. Лівий дочірній вузол служить коренем піддерева, ключі в якому менше $К_1$ середнього — коренем піддерева, ключі в якому більше $К_1$ і менше $К_2$, а правий — коренем піддерева, всі ключі в якому більше $К_2$
(мал. 6.7).

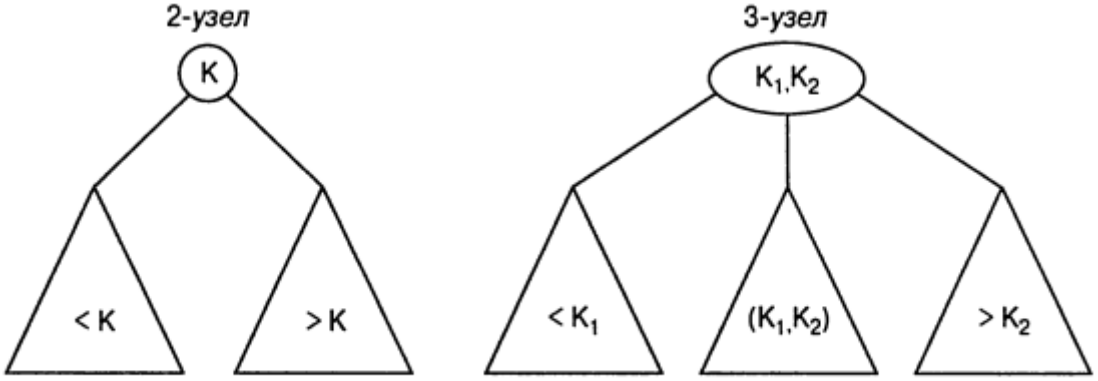


Рис. 6.7. Два вида узлов в 2-3-дереве

Остання вимога до 2-3-дерева полягає в тім, що всі його листи повинні перебувати на одному рівні, тобто 2-3-дерево **завжди збалансован по висоті** (height-balanced): довжина шляхи від кореня дерева до аркуша повинна бути однакова для всіх листів дерева. Ця властивість досягається ціною дозволу мати вузли із трьома дочірніми вузлами.

Пошук заданого ключа *К* у 2-3-дереві досить простий. Він починається з кореня. Якщо корінь являє собою 2-вузол, то ми діємо так само, як і у випадку бінарного дерева пошуку, — або припиняємо пошук, якщо значення *К* дорівнює значенню ключа кореня, або продовжуємо пошук у левом або правом піддереві, залежно від того, чи менше значення *К*, чим ключ кореня, або більше. Якщо ж корінь являє собою 3-вузол, то після не більш ніж двох порівнянь ми знаємо, чи варто припинити пошук (якщо *К* дорівнює одному із ключів 3-вузла) або в якому із трьох піддерев він повинен бути продовжений.

Вставка нового ключа в 2-3-дерево виконується в такий спосіб. Новий ключ *К* завжди вставляється в аркуш, за винятком случаючи порожнього дерева. Відповідний аркуш ми знаходимо, виконуючи пошук ключа *К*. Якщо шуканий аркуш — 2-вузол, ми вставляємо *К* або як першого, або як другий ключ — залежно від того, чи менше *К*, чим старий ключ, або більше. Якщо ж шуканий аркуш - 3-вузол, то ми розділяємо його на два: найменший із трьох ключів (двох старих і нового) міститься в перший аркуш, найбільший - у другий

аркуш, а середній ключ переноситься у вузол, батьківський стосовно старого аркуша (якщо аркуш - корінь дерева, то для вставки нового ключа створюється новий корінь). Помітимо, що переміщення середнього ключа в батьківський вузол може привести до переповнення батьківського вузла (якщо він був 3-вузлом) і, отже, викликати ряд поділів вузлів уздовж ланцюжка предків аркуша.

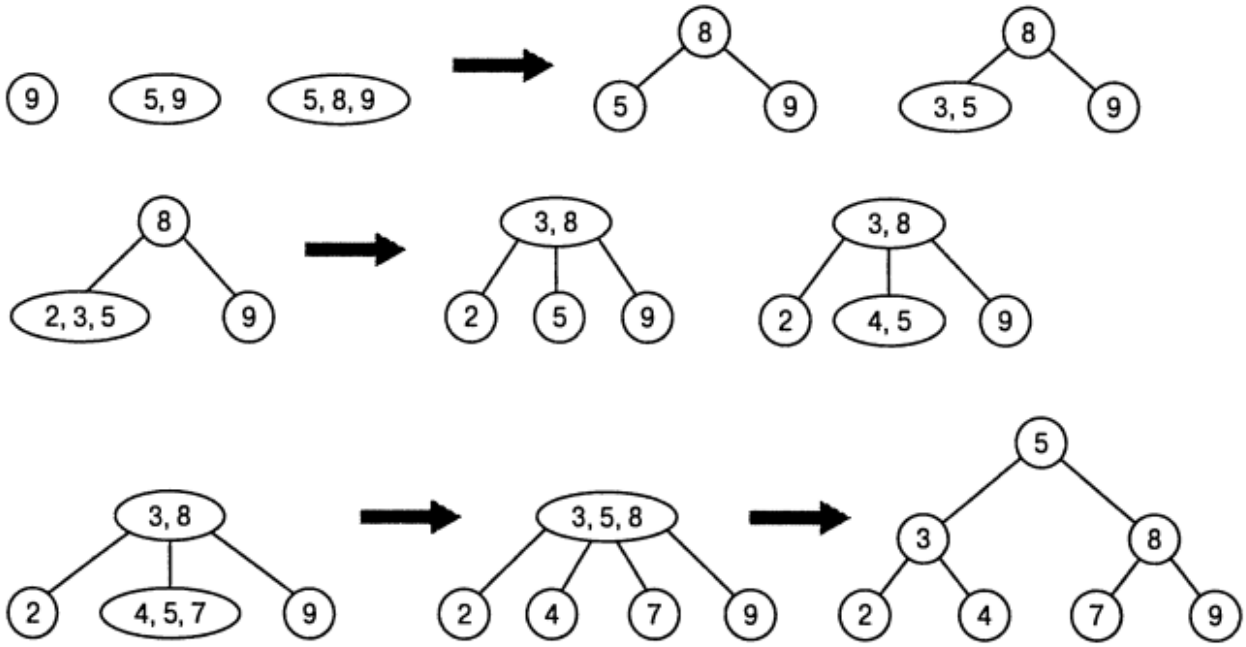


Рис. 6.8. Построение 2-3-дерева путем внесения ключей 9, 5, 8, 3, 2, 4, 7

Як і в будь-якому дереві пошуку ефективність словникових операцій залежить від його висоти. Знайдемо верхню границю ефективності. Для будь-якого 2-3-дерева висотою h з n вузлами ми одержуємо нерівність

$$h \leqslant \log_2 (n+1) - 1.$$

Для будь-якого 2-3-дерева з *n* вузлами

$$h \geqslant \log_3 (n+1) - 1.$$

З отриманих у такий спосіб верхньої й нижньої границь висоти *h*

$$\log_3 (n+1) - 1 \leqslant h \leqslant \log_2 (n+1) - 1$$

випливає, що тимчасова ефективність пошуку, вставки й видалення як у найгіршому, так і в середнє випадку - $\Theta(\log n)$

4

Піраміди й пірамідальне сортування

Структура даних за назвою піраміда (heap) являє собою частково впорядковану структуру даних, що особливо добре підходить для реалізації черг із пріоритетами. Згадаємо, що **черга із пріоритетами** (priority queue) являє собою множину елементів з упорядкованою характеристикою, що називається пріоритетом (priority) елемента, і виконання, що забезпечує, наступних операцій:

- друк елемента з найвищим (тобто з найбільшим) пріоритетом;

- видалення елемента з найбільшим пріоритетом;
- додавання нового елемента в множину.

Піраміди становлять особливий інтерес у першу чергу завдяки ефективній реалізації перерахованих операцій. Піраміда також є структурою даних, що служить наріжним каменем теоретично важливого алгоритму - пірамідального сортування (heapsort). Ми розглянемо цей алгоритм пізніше, після того як дамо визначення піраміди й вивчимо її основні властивості.

Поняття піраміди

Визначення 1. Піраміда (heap) може бути визначена як бінарне дерево с ключами, призначеними її вузлам (по одному ключі на вузол), для якого виконуються дві наступні умови.

1. Вимога до форми дерева. Бінарне дерево практично повне (essentially complete) або просто повне (complete), тобто всі його рівні заповнені, за винятком, можливо, останнього рівня, у якому можуть бути відсутніми деякі крайні праворуч листи.
2. Вимога домінування батьківських вузлів. Ключ у кожному вузлі не менше ключів у його дочірніх вузлах (умова вважається автоматично виконується для всіх листів). (Деякі автори вимагають, щоб ключ у кожному вузлі не перевищував ключі в дочірніх вузлах.)

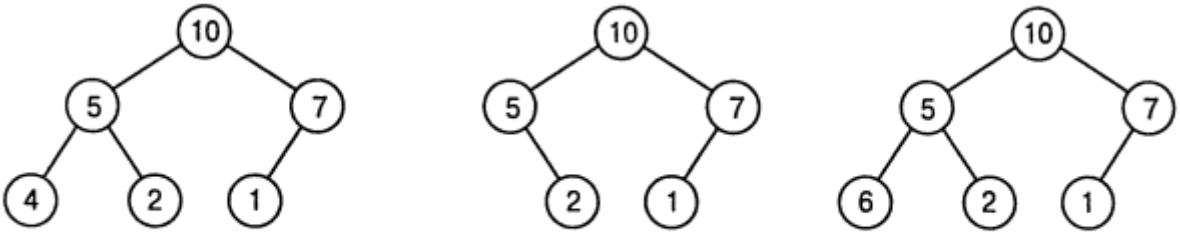


Рис. 6.9. Иллюстрация к определению пирамиды: таковой является только крайнее слева дерево

Розглянемо, наприклад, дерева на мал. 6.9. Перше дерево є пірамідою. Друге дерево - не піраміда, оскільки порушено вимогу до форми дерева. Третє дерево також не є пірамідою, оскільки в ньому порушена вимога домінування батьківських вузлів для вузла із ключем 5.

Зверніть увагу на впорядкованість значень у піраміді зверху вниз - тобто послідовність значень на будь-якому шляху від кореня до листа убутна (незростаюча, якщо допускається наявність однакових ключів). Однак упорядкованості ключів ліворуч праворуч ні, тобто немає ніяких співвідношень між значеннями ключів у вузлах на одному рівні дерева або, у загальному випадку, у левом і правом піддеревах одного вузла.

От список важливих властивостей пірамід, які нескладно довести (у якості приклада перевірте їхнє виконання для піраміди, показаної на мал. 6.10.

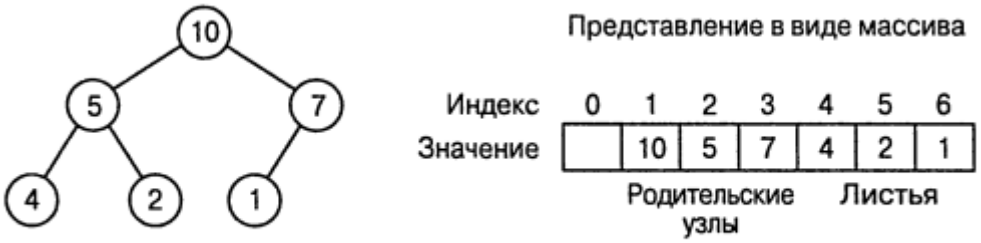


Рис. 6.10. Пирамида и ее представление в виде массива

1. Є рівно одне практично повне бінарне дерево з n вузлами. Його висота дорівнює $\lfloor \log_2 n \rfloor$.
2. Корінь піраміди завжди містить її найбільший елемент.
3. Будь-який вузол піраміди з усіма його нащадками також є пірамідою.
4. Піраміда може бути реалізована у вигляді масиву шляхом запису її елементів зверху вниз ліворуч праворуч. Зручно зберігати елементи піраміди в позиціях такого масиву з 1 по n, залишаючи $H[0]$ або невикористовуваним, або розміщаючи в ньому обмежник, значення якого перевищує значення будь-якого елемента піраміди. При використанні такого подання

а) ключі батьківських вузлів займають перші $\lfloor n/2 \rfloor$ позицій у масиві, а ключі листів — останні $\lceil n/2 \rceil$ позицій.

б) дочірні ключі стосовно батьківського в позиції i ($1 \le i \le \lfloor n/2 \rfloor$) перебувають у позиціях $2i$ і $2i + 1$, відповідно; батьківський ключ для ключа в позиції i ($2 \le i \le n$) перебуває в позиції $[i/2]$.

Таким чином, ми можемо визначити піраміду як масив H [1..n], у якому кожний елемент у позиції i у першій половині масиву більше або дорівнює елементам у позиціях $2i$ і $2i + 1$, тобто

$$H[i] \geqslant \max\{H[2i], H[2i+1]\} \quad \text{для } i = 1, 2, \ldots, \lfloor n/2 \rfloor .$$

(Звичайно, якщо $2i + 1 > n$, то виконуватися повинне тільки нерівність $H[i] \ge H[2i]$) У той час як ідеї, що лежать в основі алгоритмів з використанням пірамід, простіше для розуміння при поданні пірамід у вигляді бінарних дерев, реальні реалізації ці алгоритмів звичайно істотно простіше й ефективніше при використанні подання у вигляді масиву.

Як же побудувати піраміду для заданої множини ключів? Є два основних методи виконати цю роботу. Перший називається всхідною побудовою піраміди (bottom-up heap construction) і проілюстроване на мал. 6.11. При цьому практично повне бінарне дерево ініціалізується шляхом розміщення n ключів у заданому порядку, а потім дерево "пірамідизується" у такий спосіб. Починаючи з останнього батьківського вузла й закінчуючи коренем алгоритм перевіряє, чи виконується для розглянутого вузла вимога домінування батьківського вузла. Якщо ні, то алгоритм обмінює ключ вузла

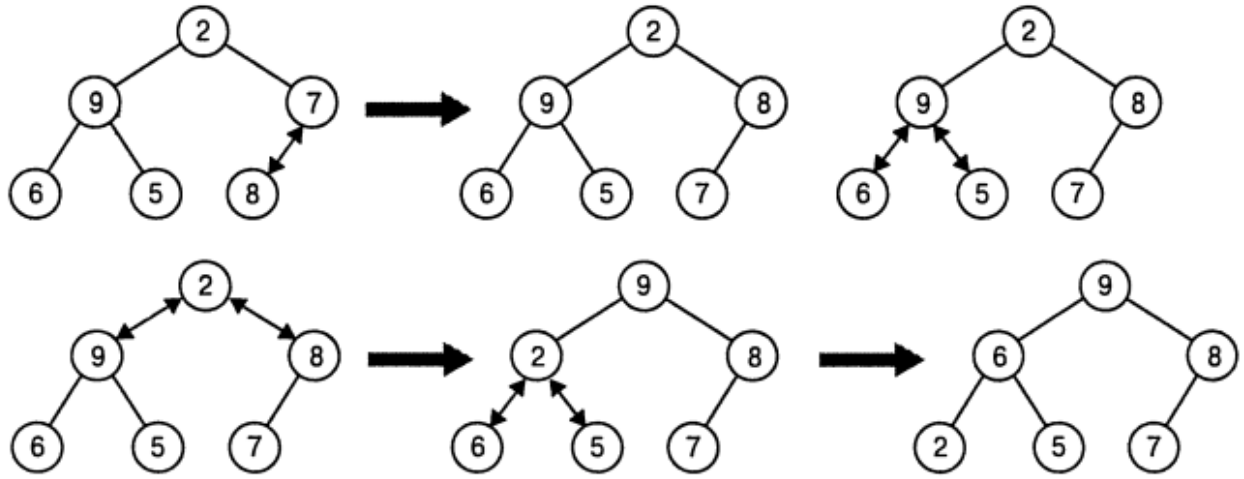


Рис. 6.11. Восходящее построение пирамиды для множества ключей 2, 9, 7, 6, 5, 8

K з найбільшим ключем серед його дочірніх вузлів і перевіряє виконання вимоги домінування батьківського вузла для ключа К у нової позиції. Цей процес триває доти, поки для ключа До не буде виконана вимога домінування батьківського вузла (в остаточному підсумку ця вимога буде виконано, тому що воно завжди виконується для ключів у листах). Після завершення "пірамідизації" піддерева, коренем якого є даний батьківський вузол, алгоритм виконує ті ж дії з безпосереднім предком цього вузла. Алгоритм завершує свою роботу після обробки кореня дерева.

Перед тим як буде наведений псевдокод висхідної побудови піраміду, варто зробити одне зауваження. Оскільки значення ключа не змінюється при його переміщенні вниз по дереву, немає необхідності виконувати проміжні обміни. Таке поліпшення алгоритму можна представити як обмін порожнього вузла з більшими ключами серед нащадків (або як переміщення порожнього вузла вниз по дереву) до досягнення кінцевої позиції, куди й вставляється раніше збережений ключ.

АЛГОРИТМ $HeapBottomUp\,(H\,[1..n])$

```
// Построение пирамиды из элементов заданного массива при
// помощи восходящего алгоритма
// Входные данные:  Массив H[1..n] упорядочиваемых элементов
// Выходные данные: Пирамида H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i; v ← H[k]
    heap = false
    while not heap and 2 * k ⩽ n do
        j ← 2 * k
        if j < n        // Имеется два дочерних узла
            if H[j] < H[j + 1]
```

```
        j ← j + 1
    if v ⩾ H[j]
        heap ← true
    else
        H[k] ← H[j]; k ← j
H[k] ← v
```

Наскільки ефективний даний алгоритм у найгіршому випадку? (зробіть аналіз)

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\substack{\text{Ключи на} \\ \text{уровне } i}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)\,2^i = 2\left(n - \log_2(n+1)\right),$$

Альтернативний (і менш ефективний) алгоритм будує піраміду шляхом послідовних вставок нового ключа в раніше побудовану; деякі автори називають цей алгоритм **спадною побудовою піраміди** (top-down heap construction). Яким же образом можна додати новий ключ у піраміду? Почнемо з додавання нового вузла із ключем *K* після останнього листа наявної піраміди, а потім перемістимо *K* у відповідному його значенню місце в новій піраміді в такий спосіб. Зрівняємо *K* с батьківським ключем: якщо він не менше *K*, алгоритм припиняє роботу (отримана структура є пірамідою). У противному випадку обміняємо ці два ключі й будемо порівнювати *K* с новим батьком. Цей процес триває доти, поки *K* не перестане перевищувати значення ключа в батьківському вузлі або не досягне кореня (цей процес проілюстрований на мал. 6.12). У цьому алгоритмі також можна переміщати порожній вузол до досягнення їм коректної позиції, а потім привласнити йому значення *K*.

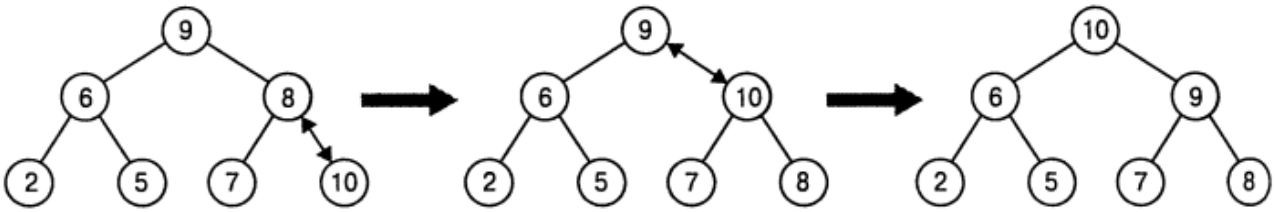


Рис. 6.12. Вставка ключа 10 в пирамиду, построенную на рис. 6.11

Очевидно, що така вставка не може вимагати більшої кількості порівнянь ключів, чим висота піраміди. Оскільки висота піраміди з *n* вузлами біля $\log_2 n$, тимчасова ефективність вставки становить *O (logn)*.

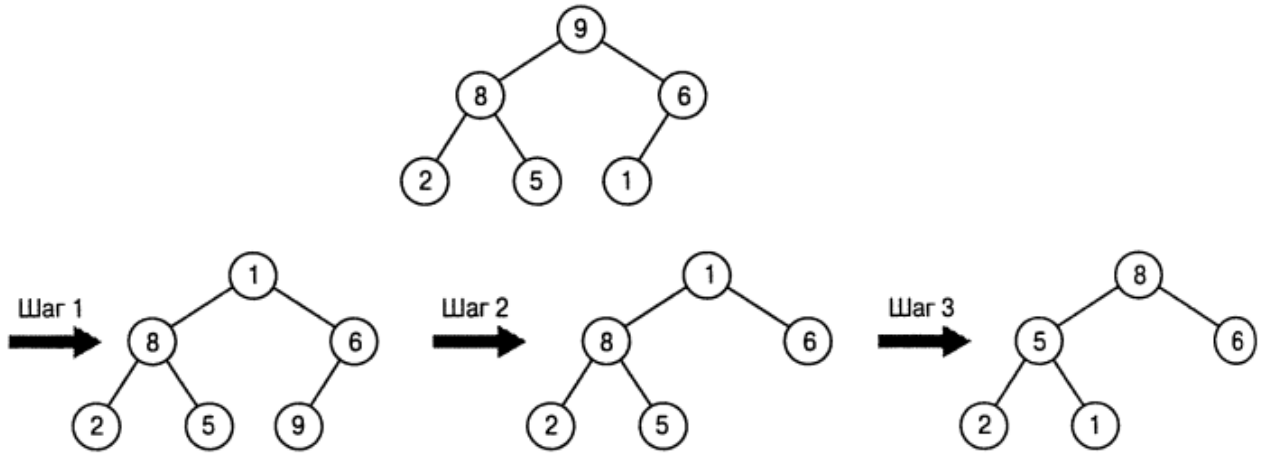


Рис. 6.13. Удаление корневого ключа из пирамиды

Як виконується видалення вузла з піраміди? Ми розглянемо тут тільки найбільш важливий випадок видалення ключа з кореня, залишаючи питання про видалення довільного ключа як самостійну вправу. Отже, видалення кореневого ключа з піраміди можна виконати за допомогою наступного алгоритму (проілюстрованого на мал. 6.13).

Крок 1. Обміняти ключ у корені з останнім ключем піраміди.

Крок 2. Зменшити розмір піраміди на 1.

Крок 3. "Пірамідизувати" зменшене дерево шляхом переміщення *K* униз по дереву так само, як ми робили це у висхідному алгоритмі побудови піраміди, — тобто перевіряючи виконання вимоги домінування батьківських вузлів: якщо воно виконується, алгоритм завершує роботу, якщо немає — обмінюємо *K* с найбільшим з дочірніх вузлів і повторюємо дану операцію доти, поки в черговій позиції *K* вимоги домінування батьківських вузлів не виявиться виконаним.

Ефективність операції видалення визначається кількістю виконуваних порівнянь ключів, необхідних для "пірамідизації" дерева після того, як був зроблений обмін і розмір піраміди був зменшений на 1. Оскільки не може знадобитися порівнянь більше, ніж подвоєна висота піраміди, тимчасова ефективність видалення з піраміди — *O (logn)*.

5

Пірамідальне сортування

Тепер ми можемо описати пірамідальне сортування (heapsort) - цікавий алгоритм сортування, відкритий Дж. Вільямсом (J. W. J. Williams). Цей двухетапний алгоритм працює в такий спосіб.

Етап 1 (побудова піраміди). Будуємо піраміду для заданого масиву.

Етап 2 (видалення найбільших елементів). Застосовуємо операцію видалення кореня *n - 1* раз.

У результаті елементи масиву видаляються в порядку зменшення. Але оскільки при реалізації піраміди з використанням масиву видаляється элемент, що,

розташовується останнім, масив, що виходить у результаті пірамідального сортування, виявляється відсортований у порядку зростання.

Приклад покрокового виконання пірамідального сортування наведений на мал. 6.14 (на мал. 6.11 навмисно використалися ті ж вхідні дані, для того щоб ви могли зрівняти висхідну побудову піраміди при її реалізації у вигляді дерева й з використанням масиву).

$$C(n) \leqslant 2\lfloor \log_2 (n-1) \rfloor + 2\lfloor \log_2 (n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \leqslant 2\sum_{i=1}^{n-1} \log_2 i \leqslant$$

$$\leqslant 2\sum_{i=1}^{n-1} \log_2 (n-1) = 2(n-1)\log_2 (n-1) \leqslant 2n \log_2 n.$$

Це означає, що для другого етапу пірамідального сортування $C(n) \in O(n \log n)$. Більше докладний аналіз показує, що в дійсності тимчасова ефективність пірамідального сортування дорівнює $\Theta(n \log n)$ як у середньому, так і в найгіршому випадках. Таким чином, тимчасова ефективність пірамідального сортування попадає в той же клас, що й сортування злиттям, але, на відміну від останньої, виконується "на місці", без залучення додаткової пам'яті. Експерименти, проведені над випадковими файлами, показують, що пірамідальне сортування працює повільніше швидкої, однак цілком може суперничати із сортуванням злиттям.