

Тема 9. Бінарні дерева пошуку

9.1. Бінарні дерева пошуку

Дерева пошуку представляють собою структури даних, які підтримують більшість операцій з динамічними множинами: пошук елементів, мінімального та максимального значення, попереднього та наступного елементу, вставку та видалення. Таким чином, дерево пошуку може використовуватись як словник, так і як черга з пріоритетами.

Як слідує з назви, бінарне дерево пошуку в першу чергу є бінарним деревом (рис. 9.1) Таке дерево може бути представлене за допомогою зв'язаної структури даних, в якій кожний вузол є об'єктом. Ключі у бінарному дереві пошуку зберігаються таким чином, щоб в будь-який момент задовольняти наступній умові бінарного дерева пошуку: якщо x – вузол бінарного дерева пошуку, а вузол y знаходиться у лівому піддереві x , то $key[y] \leq key[x]$; якщо y знаходиться у правому піддереві x , то $key[y] \geq key[x]$.

На рис. 9.1, а ключ кореня дорівнює 5, ключі 2, 3 та 5, які не більші значення ключа в корені, знаходяться в його лівому піддереві, а ключі 7 та 8, які не менші кореня, – в правому піддереві. Та сама властивість виконується й для кожного внутрішнього вузла дерева. На рис. 9.1, б показане дерево з тими самими вузлами та яке має ту саму властивість, проте менш ефективне в роботі, адже його висота дорівнює 4, на відміну від дерева на рис. 9.1, а, висота якого дорівнює 2.

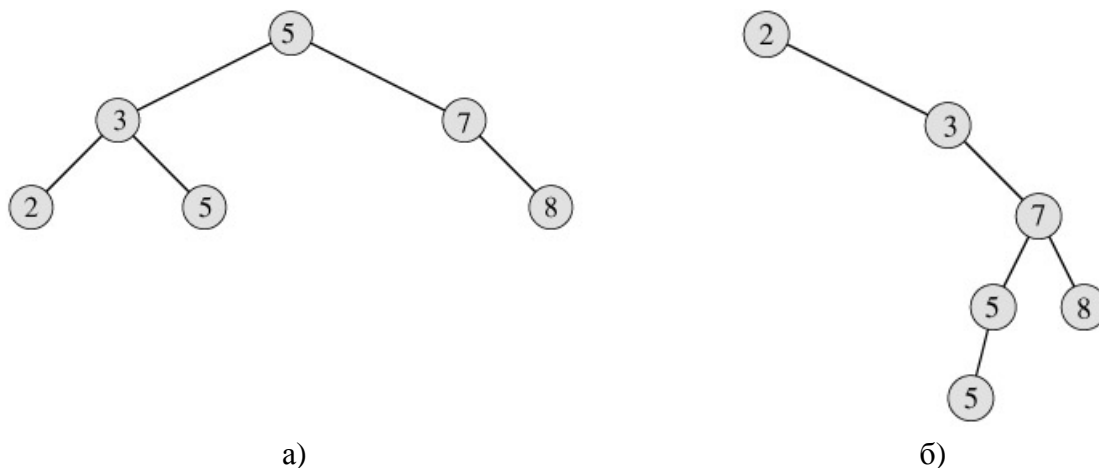


Рис. 9.1. Приклад бінарного дерева пошуку.

Властивість бінарного дерева пошуку дозволяє вивести всі ключі, які знаходяться у дереві, у відсортованому порядку за допомогою простого рекурсивного алгоритму, який називається **обходом дерева у внутрішньому порядку** (*inorder*). Цей алгоритм обходить вершини дерева таким чином, що корінь піддерева обходиться після свого лівого піддерева та перед правим піддеревом. Є й інші способи обходу дерева: обхід у прямому порядку (*preorder*), коли спочатку обходиться корінь, а потім його нащадки зліва направо, та у зворотному порядку (*postorder*), коли спочатку обходяться нащадки зліва направо, а потім корінь піддерева. Обхід дерева T у внутрішньому порядку реалізується процедурою `InorderTreeWalk`.

```

InorderTreeWalk(x)
1  if x ≠ NIL
2      then InorderTreeWalk(left[x])
3           print key[x]
4           InorderTreeWalk(right[x])
  
```

Лістинг 9.1. Процедура `InorderTreeWalk` обходу дерева у внутрішньому порядку.

Головний виклик даної процедури для дерева T має вигляд: `InorderTreeWalk(root[T])`. Для дерев з рис. 9.1 ми отримуємо в обох випадках один й той самий вивід: 2, 3, 5, 5, 7, 8. Коректність описаного алгоритму слідує безпосередньо з властивостей бінарного дерева пошуку.

Для обходу дерева необхідний час $\Theta(n)$, оскільки після початкового виклику процедура викликається рівно два рази для кожного вузла дерева: один раз для його лівого дочірнього вузла, та інший раз – для правого.

9.2. Робота з бінарними деревами пошуку

Найбільш розповсюдженою операцією, яка виконується з бінарним деревом пошуку, є пошук в ньому певного ключа. Окрім того, бінарні дерева пошуку підтримують такі запити, як пошук мінімального та максимального елементів, а також наступного та попереднього. Всі ці операції можуть бути виконані за час $O(h)$, де h – висота дерева.

Для пошуку вузла із заданим ключем у бінарному дереві пошуку використовується наступна процедура `TreeSearch`, яка отримує в якості параметрів покажчик на корінь бінарного дерева та ключ k , а повертає покажчик на вузол з цим ключем (якщо такий існує, або значення `NIL` у протилежному випадку).

```
TreeSearch(x, k)
1  if x = NIL або k = key[x]
2      then return x
3  if k < key[x]
4      then return TreeSearch(left[x], k)
5  else return TreeSearch(right[x], k)
```

Лістинг 9.2. Процедура пошуку ключа `TreeSearch`.

Процедура пошуку починається з кореня дерева та переходить униз по дереву. Для кожного вузла x на шляху вниз його ключ $key[x]$ порівнюється з ключем k , який був переданий у якості параметру. Якщо ключі однакові, то пошук завершений – ми знайшли елемент. Якщо k менше $key[x]$, то пошук продовжується у лівому піддереві x ; якщо більше – то у правому піддереві. На рис. 9.2 для пошуку ключа 13 ми повинні пройти наступний шлях від кореня: $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$. Вузли, які відвідуються при рекурсивному виклику, утворюють низхідний шлях від кореня дерева, так що час роботи процедури `TreeSearch` дорівнює $O(h)$, де h – висота дерева.

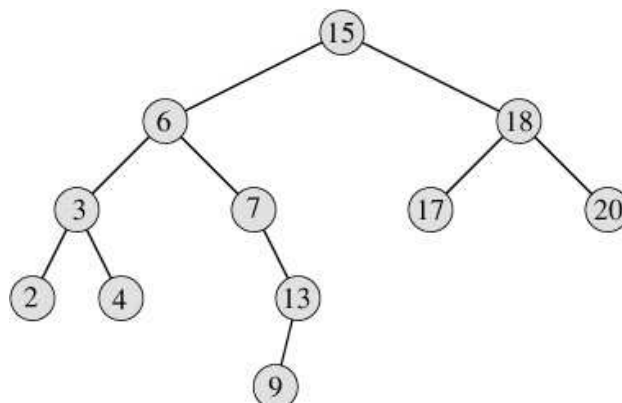


Рис. 9.2. Приклад бінарного дерева пошуку.

Ту саму процедуру можна записати ітеративно, якщо розгорнути рекурсію у цикл **while**. На більшості комп'ютерів така версія виявляється більш ефективною за рахунок того, що не потрібно витратити процесорний час на дочірні виклики функції.

```

IterativeTreeSearch(x, k)
1  while x ≠ NIL та k ≠ key[x]
2      do if k < key[x]
3          then x ← left[x]
4          else x ← right[x]
5  return x

```

Лістинг 9.3. Ітеративна реалізація процедури пошуку `IterativeTreeSearch`.

Елемент з мінімальним значенням ключа легко знайти, якщо рухатись за покажчиками *left* від кореневого вузла допоки не зустрінеться значення `NIL`. Так, на рис. 9.2, слідуючи таким чином, буде пройдений шлях $15 \rightarrow 6 \rightarrow 3 \rightarrow 2$ до мінімального ключа в дереві, який дорівнює 2. Нижче наведений псевдокод цього алгоритму.

```

TreeMinimum(x)
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x

```

Лістинг 9.4. Процедура `TreeMinimum` пошуку мінімального елементу в дереві.

Властивість бінарного дерева пошуку гарантує коректність процедури `TreeMinimum`. Якщо у вузла x немає лівого піддерева, то оскільки всі ключі в правому піддереві x не менші за $key[x]$, мінімальний ключ піддерева з коренем у вузлі x знаходиться в цьому вузлі. Якщо ж у вузла є ліве піддерево, то, оскільки, в правому піддереві не може бути вузла з ключем, який менший за $key[x]$, а всі ключі у вузлах лівого піддерева не більші за $key[x]$, то вузол з мінімальним значенням ключа знаходиться у піддереві, коренем якого є вузол $left[x]$.

Алгоритм пошуку максимального елементу дерева симетричний алгоритму пошуку мінімального елементу:

```

TreeMaximum(x)
1  while left[x] ≠ NIL
2      do x ← right[x]
3  return x

```

Лістинг 9.5. Процедура `TreeMaximum` пошуку максимального елементу в дереві.

Обидві представлені процедури знаходять мінімальний (максимальний) елементи дерева за час $O(h)$, де h – висота дерева, оскільки, як і в процедурі `TreeSearch`, послідовність досліджуваних вузлів утворює низхідний шлях від кореня дерева.

Іноді, маючи вузол у бінарному дереві пошуку, необхідно визначити вузол, який йде наступним або попереднім у відсортованій послідовності. Якщо всі ключі різні, то наступним по відношенню до вузла x є вузол із найменшим ключем, який більший за $key[x]$. Структура бінарного дерева пошуку дозволяє знайти такий вузол навіть не виконуючи порівняння ключів. Наведена нижче процедура повертає вузол, який є наступним за x у бінарному дереві пошуку (якщо такий існує) та `NIL`, якщо x має найменше значення ключа у дереві.

```

TreeSuccessor(x)
1  if right[x] ≠ NIL
2      then return TreeMinimum(right[x])
3  y ← parent[x]
4  while y ≠ NIL та x = right[y]
5      do x ← y
6      y ← parent[y]

```

7 **return** y

Лістинг 9.6. Процедура `TreeSuccessor` визначення вузла наступника.

Код процедури `TreeSuccessor` розбитий на дві частини. Якщо праве піддерево вузла x не порожнє, то наступний за x елемент є крайнім лівим вузлом у правому піддереві. Цей елемент визначається викликом процедури `TreeMinimum` у рядку 2. Наприклад, на рис. 9.2 наступний за вузлом з ключем 15 є вузол з ключем 7.

З іншого боку, якщо праве піддерево вузла x порожнє та в x є наступний за ним елемент y , то y є найменшим предком x , чий лівий нащадок також є предком x . На рис. 9.2 наступний за вузлом з ключем 13 є вузол з ключем 15. Для того щоб знайти y , необхідно просто підніматись вгору деревом допоки не зустрінемо вузол, який є лівим дочірнім вузлом свого батька. Це виконується у рядках 3-7 алгоритму.

Час роботи процедури `TreeSuccessor` у дереві висотою h складає $O(h)$, оскільки ми або рухаємось униз за деревом від початкового вузла, або вгору. Процедура пошуку наступного вузла в дереві `TreePredecessor` симетрична процедурі `TreeSuccessor` і так само працює за час $O(h)$.

Якщо в дереві є вузли з однаковими ключами, то можна просто визначити наступний та попередній вузли як такі, що повертаються процедурами `TreeSuccessor` та `TreePredecessor` відповідно.

9.3. Вставка та видалення

Операції вставки та видалення призводять до внесення змін у динамічну множину, яка представлена бінарним деревом. Структура даних повинна бути змінена таким чином, щоб відображати ці зміни, але при цьому зберігати властивість бінарних дерев пошуку.

Для вставки нового значення v у бінарне дерево пошуку T скористаємось процедурою `TreeInsert`. Процедура отримує в якості параметру вузол z , у якого $key[z] = v$, $left[z] = NIL$ та $right[z] = NIL$, після чого вона таким чином змінює T та деякі поля z , що z виявляється вставленим у відповідну позицію з дереві.

```
TreeInsert( $T$ ,  $z$ )
1   $y \leftarrow NIL$ 
2   $x \leftarrow root[T]$ 
3  while  $x \neq NIL$ 
4      do  $y \leftarrow x$ 
5          if  $key[z] < key[x]$ 
6              then  $x \leftarrow left[x]$ 
7              else  $x \leftarrow right[x]$ 
8   $parent[z] \leftarrow y$ 
9  if  $y = NIL$ 
10     then  $root[T] \leftarrow z$ 
11     else if  $key[z] < key[y]$ 
12         then  $left[y] \leftarrow z$ 
13         else  $right[y] \leftarrow z$ 
```

Лістинг 9.7. Процедура `TreeInsert` вставки нового елементу до дерева.

На рис. 9.3 наведена робота процедури `TreeInsert`. Подібно до процедур `TreeSearch` та `IterativeTreeSearch`, процедура `TreeInsert` починає роботу з кореневого вузла дерева та проходить низхідним шляхом. Показчик x відмічає пройдений шлях, а показчик y вказує на батьківський по відношенню до x вузол. Після ініціалізації цикл **while** у рядках 3-7 переміщує показники вниз деревом, пересуваючись ліворуч чи праворуч залежно від результату порівняння ключів $key[x]$ та $key[z]$ допоки x не стане

рівним NIL. Це значення знаходиться саме в тій позиції, куди слід помістити елемент z . У рядках 8-13 виконується приписування значень покажчикам для вставки z .

Як й інші базові операції над бінарними деревами пошуку, процедура `TreeInsert` виконується за час $O(h)$, де h – висота дерева.

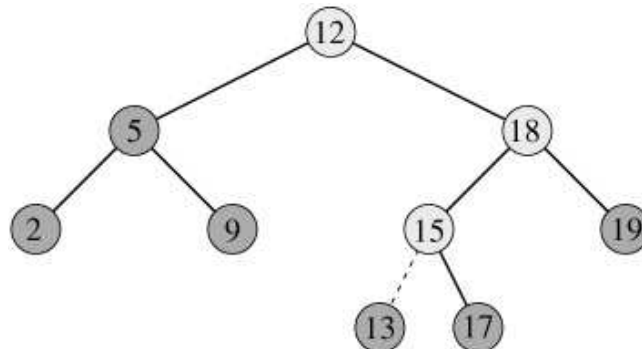


Рис. 9.3. Вставка елементу 13 у бінарне дерево пошуку.

Процедура видалення даного вузла z з бінарного дерева пошуку отримує в якості аргументу покажчик на z . Процедура переглядає три можливі ситуації, які наведені на рис. 9.4. Якщо у вузла z немає дочірніх вузлів (рис. 9.4, а), то ми просто змінюємо батьківський вузол $parent[z]$, замінюючи в ньому покажчик на z значенням NIL. Якщо у вузла z тільки один дочірній вузол (рис. 9.4, б), то ми видаляємо z , створюючи новий зв'язок між батьківським та дочірнім вузлом вузла z . Нарешті, якщо у вузла z два дочірніх вузла (рис. 9.4, в), то ми шукаємо наступний за ним вузол y , в якого немає лівого дочірнього вузла, видаляємо його з позиції, де він знаходився раніше, шляхом створення нового зв'язку між його батьком та нащадком, і замінюємо ним вузол z .

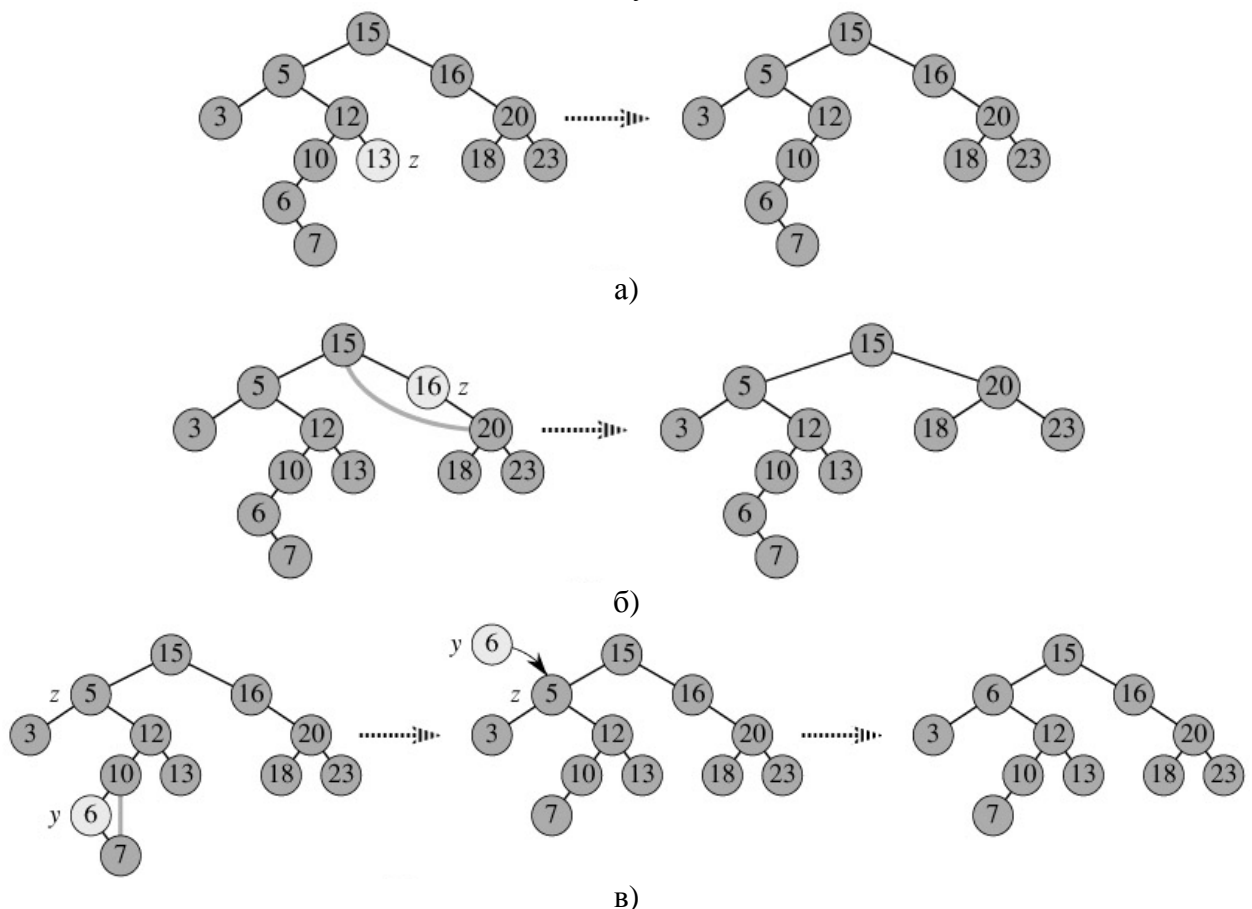


Рис. 9.4. Різні варіанти видалення елементу з бінарного дерева пошуку.

Код процедури `TreeDelete` реалізує ці дії трохи по-інакшому, ніж вони описані.

```

TreeDelete(T, z)
1  if left[z] = NIL або right[z] = NIL
2      then y ← z
3      else y ← TreeSuccessor(z)
4  if left[y] ≠ NIL
5      then x ← left[y]
6      else x ← right[y]
7  if x ≠ NIL
8      then parent[x] ← parent[y]
9  if parent[y] = NIL
10     then root[T] ← x
11     else if y = left[parent[y]]
12         then left[parent[y]] ← x
13         else right[parent[y]] ← x
14 if y ≠ z
15     then key[z] ← key[y]
16         Копіювання допоміжних даних з у в z
17 return y

```

Лістинг 9.8. Процедура `TreeDelete` видалення елементу з дерева.

У рядках 1-3 алгоритм визначає вузол y , який видаляється шляхом «склеювання» батька та нащадка. Цей вузол є або вузлом z (якщо у вузла z не більше одного дочірнього вузла), або вузлом, який є наступним за z (якщо у вузла z два дочірніх вузла). Потім у рядках 4-6 x приписується покажчик на дочірній вузол вузла y або значення `NIL`, якщо в y немає дочірніх вузлів. Далі вузол y видаляється у рядках 7-13 шляхом зміни покажчиків у `parent[y]` та x . Це видалення ускладнюється необхідністю коректної обробки граничних умов (коли x дорівнює `NIL` або коли y – кореневий вузол). Нарешті, у рядках 14-16, якщо видалений вузол y був наступним за z , ми переписуємо ключ z та допоміжні дані ключем та допоміжними даними y . Видалений вузол y повертається у рядку 17, з тим щоб процедура, яка викликала видалення, могла за необхідністю звільнити використану ним пам'ять. Час роботи описаної процедури становить $O(h)$, де h – висота дерева.

Отже, ми отримали, що всі базові операції у бінарному дереві пошуку виконуються за один й той самий час – $O(h)$, де h – висота дерева. Як було наведено вище, недоліком бінарних дерев пошуку є те, що вони для однакових значень ключів можуть мати різну висоту h . Зрозуміло, що оптимальне значення висоти буде дорівнювати $\log_2 n$, де n – кількість елементів у дереві. Древа, які мають таку висоту називаються збалансованими. У загальному випадку бінарні дерева пошуку не є збалансованими, адже додавання або вилучення елементів з дерева призводить до його розбалансування. Підтримка збалансованості дерева досить трудомістка операція. Тож замість постійного відновлення збалансованості, рекомендується використання інших деревоподібних структур (червоно-чорні дерева, AVL-дерева), які гарантують збереження властивості збалансованості під час роботи з ними.

Література

Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001) [1990]. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7. Глава 12.