Лекція 23

Бінарні дерева пошуку

Дерева пошуку являють собою структури даних, що підтримують операції пошуку елемента, мінімального і максимального значення, попередника й наступника, вставку і видалення.

Основні операції в бінарному дереві пошуку виконуються за час, пропорційний його висоті. Для повного бінарного дерева з n вузлами ці операції виконуються за час $\Theta(\lg n)$ у найгіршому випадку. Математичне сподівання висоти побудованого випадковим образом бінарного дерева равно $O(\lg n)$, так що всі основні операції над динамічною множиною в такім дереві виконуються в середньому за час $\Theta(\lg n)$.

На практиці не завжди можна гарантувати випадковість побудови бінарного дерева пошуку, однак існують версії дерев, у яких гарантується гарний час роботи в найгіршому випадку, а саме — червоно-чорні дерева, висота яких $O(\lg n)$.

Структура бінарного дерева

Бінарне дерево може бути представлене за допомогою зв'язаної структури даних, у якій кожен вузол є об'єктом. На додаток до полів ключа *key* і супутніх даних, кожен вузол містить поля *left*, *right* і *p*, що вказують на лівий і правий дочірні вузли і на батьківський вузол відповідно. Якщо дочірній чи батьківський вузол відсутні, відповідне поле містить значення NULL. Єдиний вузол, вказівник p якого дорівнює NULL, — це кореневий вузол дерева. Ключі в бінарному дереві пошуку зберігаються таким чином, щоб у будь-який момент задовольняти наступній ***властивості бінарного дерева пошуку***.

> Якщо x — вузол бінарного дерева пошуку, а вузол y знаходиться в лівому піддереві вузла x, то $key[y] \le key[x]$. Якщо вузол y знаходиться в правому піддереві вузла x, то $key[x] \le key[y]$.

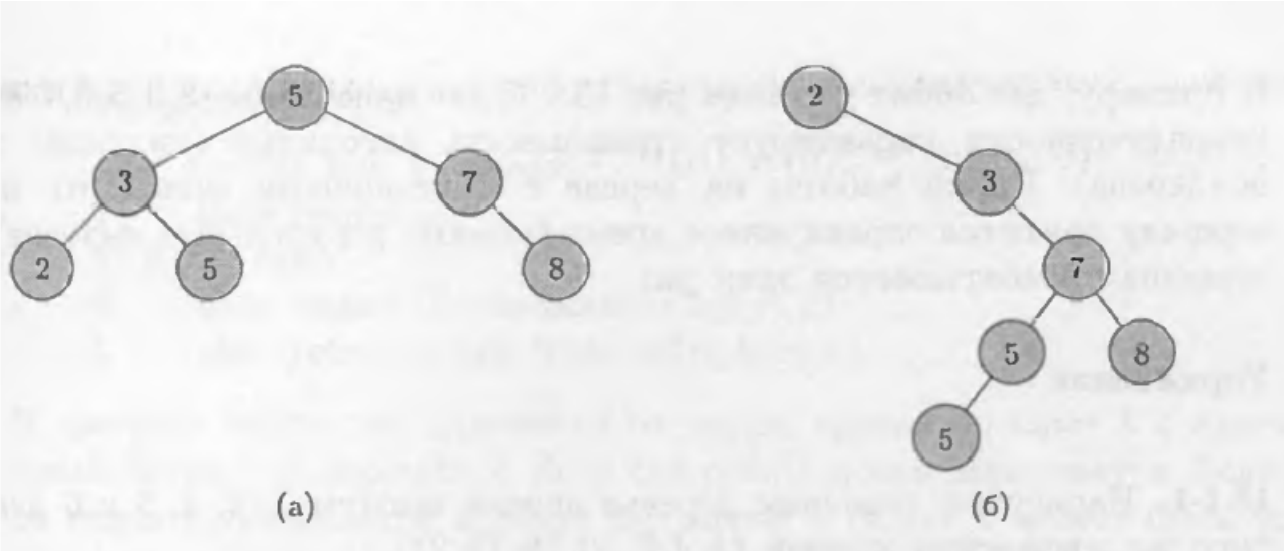

Рис. 23.1. Бінарні дерева пошуку

Так, на рис. 23.1*а* ключ кореня дорівнює 5, ключі 2, 3 і 5, що не перевищують значення ключа в корені, знаходяться в його левом піддереві, а ключі 7 і 23, що не менше, ніж ключ 5, — у його правом піддереві. Та ж властивість, як легко переконатися, виконується для кожного іншого вузла дерева. На рис. 23.1*б* показане дерево з тими ж вузлами, що має ту ж властивість, однак менш ефективне в роботі, оскільки його висота дорівнює 4, на відміну від дерева на рис. 23.1.*а*, висота якого дорівнює 2.

Властивість бінарного дерева пошуку дозволяє нам вивести всі ключі, що знаходяться в дереві, у відсортованому порядку за допомогою простого рекурсивного алгоритму, називаного ***симетричним обходом дерева*** (inorder tree walk). Цей алгоритм одержав дану назву в зв'язку з тим, що ключ у корені піддерева виводиться між значеннями ключів лівого піддерева і правого піддерева. Існують й інші способи обходу, а саме — ***обхід у прямому порядку*** (preorder tree walk), при якому спочатку виводиться корінь, а потом — значення лівого і правого піддерева, і ***обхід у зворотному порядку*** (postorder tree walk), коли першими виводяться значення лівого і правого піддерева, а уже потім — кореня. Симетричний обхід дерева T реалізується процедурою Inorder_Tree_Walk($root[T]$):

```
Inorder_Tree_Walk(x)
1  if x≠ NULL
2       then Inorder_Tree_Walk(left[x])
```

```
3           print key[x]
4           Inorder_Tree_Walk(right[x])
```

Симетричний обхід дерев, показаних на рис. 23.1, дає в обох випадках той самий порядок ключів, а саме 2, 3, 5, 5, 7, 23. Коректність описаного алгоритму випливає безпосередньо з властивості бінарного дерева пошуку.

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

Теорема 23.1. Якщо x *— корінь піддерева, у якому є* n *вузлів, то процедура Inorder_Tree_Walk(*x*) виконується за час* $\Theta(n)$.

Пошук

Для пошуку вузла з заданим ключем у бінарному дереві пошуку використовується наступна процедура Tree_Search, що одержує як параметри вказівник на корінь бінарного дерева і ключ *k*, а повертає вказівник на вузол з цим ключем (якщо такий існує; інакше повертається значення NULL):

```
Tree_Search(x,k)
 1  if x = NULL or k = key[x]
 2       then return x
 3  if k < key[x]
 4       then return Tree_Search(left[x],k)
 5       else return Tree_Search(right[x],k)
```

Процедура пошуку починається з кореня дерева і проходить униз по дереву. Для кожного вузла x на шляху вниз його ключ $key[x]$ порівнюється з переданим як параметр ключем k. Якщо два ключі однакові, пошук завершується. Якщо k менше $key[x]$, пошук продовжується в левом піддереві x; якщо більше — то пошук переходить у праве піддерево. Так, на рис. 23.2 для пошуку ключа 13 ми повинні пройти наступний шлях від кореня: $15 \to 6 \to 7 \to 13$. Вузли, що ми відвідуємо при рекурсивному пошуку, утворять спадний шлях від кореня дерева, так що час роботи процедури Tree_Search складає $O(h)$, де h — висота дерева.

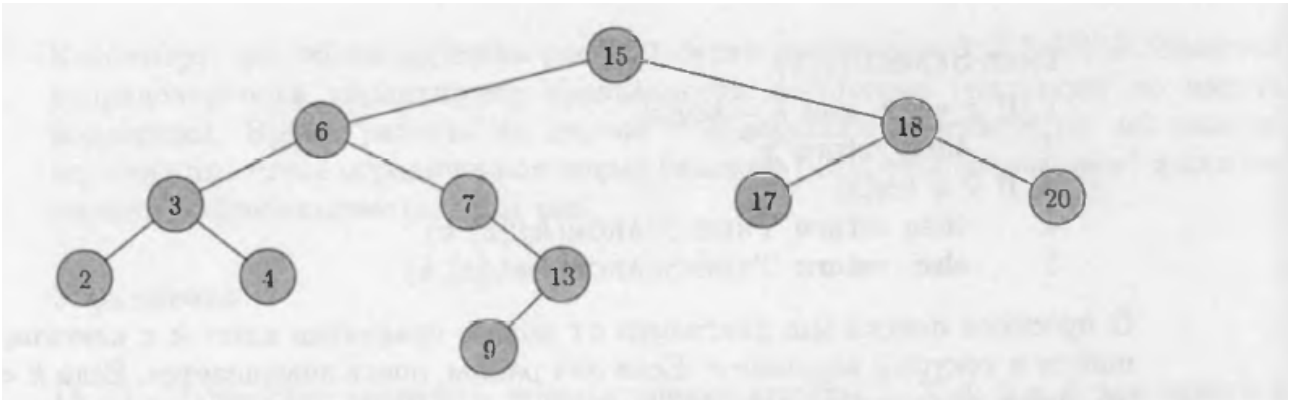

Рис. 23.2. Запити в бінарному дереві пошуку

Ту ж процедуру можна записати ітеративно, замінивши рекурсію циклом **while**.

Iterative_Tree_Search (*x*, *k*)

```
1  while x≠NULL and k≠key[x]
2      do if k < key[x]
3           then x ← left[x]
4           else x ← right[x]
5  return x
```

Пошук мінімуму і максимуму

Елемент із мінімальним значенням ключа легко знайти, переходячи по вказівниках *left* від кореневого вузла доти, поки не зустрінеться значення NULL. Так, на рис. 23.2, проходячи по вказівниках *left*, ми пройдемо шлях 15→6→3→2 до мінімального ключа в дереві, що дорівнює 2. Ось как виглядає реалізація описаного алгоритму:

Tree_Minimum (*x*)

```
1  while left[x] ≠ NULL
2      do x ← left[x]
3  return x
```

Властивість бінарного дерева пошуку гарантує коректність процедури Tree_Minimum. Якщо у вузла *x* немає лівого піддерева, то оскільки всі ключі в правом піддереві *x* не менше ключа $key[x]$, мінімальний ключ піддерева з коренем у вузлі *x* знаходиться в цьому вузлі. Якщо ж у вузла є ліве поддерево, то, оскільки в правом піддереві не може бути вузла з ключем, меншим $key[x]$, а всі ключі у вузлах лівого піддерева не перевищують $key[x]$, вузол з мінімальним значенням ключа знаходиться в піддереві, коренем якого є вузол $left[x]$.

Алгоритм пошуку максимального елемента дерева симетричний алгоритму пошуку мінімального елемента:

```
Tree_Maximum(x)
1  while right[x] ≠ NULL
2      do x ← right[x]
3  return x
```

Обидві представлені процедури знаходять мінімальний (максимальний) елемент дерева за час $O(h)$, де h — висота дерева, оскільки, як і в процедурі Tree_Search, послідовність вузлів, що перевіряються, утворить спадний шлях від кореня дерева.

Попередній і наступний елементи

Іноді, маючи вузол у бінарному дереві пошуку, потрібно визначити, який вузол слідує за ним у відсортованій послідовності, обумовленої порядком симетричного обходу бінарного дерева, і який вузол передує даному. Якщо всі ключі різні, наступним стосовно вузла x є вузол з найменшим ключем, що є більшим ніж $key[x]$. Структура бінарного дерева пошуку дозволяє нам знайти цей вузол навіть не виконуючи порівняння ключів. Приведена далі процедура повертає вузол, що слідує за вузлом x у бінарному дереві пошуку (якщо такий існує) і NULL, якщо x має найбільший ключ у бінарному дереві:

```
Tree_Successor(x)
1  if right[x] ≠ NULL
2      then return Tree_Minimum(right[x])
3  y ← p[x]
4  while y ≠ NULL and x = right[y]
5      do x ← y
6         y ← p[y]
7  return y
```

Код процедури Tree_Successor розбивається на дві частини. Якщо праве піддерево вузла x непорожнє, то наступний за x елемент є крайнім лівим вузлом у правому піддереві, що виявляється в рядку 2 викликом процедури Tree_Minimum($right[x]$). Наприклад, на рис. 23.2 наступним за вузлом із ключем 15 є вузол із ключем 23.

З іншого боку, якщо праве піддерево вузла x порожнє, і в x є наступний за ним елемент y, то y є найменшим предком x, чий лівий спадкоємець також є предком x. На рис. 23.2 наступним за вузлом із ключем 13 є вузол із ключем 15. Для того щоб знайти y, ми просто піднімаємося нагору по дереву доти, поки не зустрінемо вузол, що є лівим дочірнім вузлом свого батька. Ця дія виконується в рядках 3–7 алгоритми.

Час роботи алгоритму Tree_Successor у дереві висотою h складає $O(h)$, оскільки ми або рухаємося по шляху униз від вихідного вузла, або по шляху нагору. Процедура пошуку наступного вузла в дереві Tree_Predecessor симетрична процедурі Tree_Successor і також має час роботи $O(h)$.

Якщо в дереві існують вузли з однаковими ключами, ми можемо просто визначити наступний і попередній вузли як такі, що повертаються процедурами Tree_Successor і Tree_Predecessor відповідно.

Теорема 23.2. Операції пошуку, визначення мінімального і максимального елемента, а також попереднього і наступного, у бінарному дереві пошуку висоти h можуть бути виконані за час $O(h)$.

Вставка і видалення

Операції вставки і видалення приводять до внесення змін у динамічну множину, що представлена бінарним деревом пошуку. Структура даних повинна бути змінена таким чином, щоб відбивати ці зміни, але при цьому зберегти властивості бінарних дерев пошуку.

Вставка

Для вставки нового значення v у бінарне дерево пошуку T ми скористаємося процедурою Tree_Insert. Процедура одержує як параметр вузол z, у якого $key[z] = v$, $left[z] = \text{NIL}$ і $right[z] = \text{NIL}$, після чого вона в такий спосіб змінює T і деякі поля z, щоz виявляється вставленим у відповідну позицію в дереві:

```
Tree_Insert(T, z)
 1  y ← NULL
 2  x ← root[T]
```

```
3  while x ≠ NULL
4      do y ← x
5          if key[z] < key[x]
6              then x ← left[x]
7              else x ← right[x]
23  p[z] ← y
9  if y = NULL
10      then root[T] ← z       // Дерево T — порожнє
11      else if key[z] < key[y]
12          then left[y] ← z
13          else right[y] ← z
```

На рис. 23.3 показана робота процедури Tree_Insert. Подібно процедурам Tree_Search і Iterative_Tree_Search, процедура Tree_Insert починає роботу з кореневого вузла дерева і проходить по спадному шляху. Вказівник x відзначає прохідний шлях, а вказівник y указує на батьківський стосовно x вузол. Після ініціалізації цикл **while** у рядках 3–7 переміщає ці вказівники вниз по дереву ліворуч чи праворуч в залежності від результату порівняння ключів $key[x]$ і $key[z]$, доти пока x не стане рівним NULL. Це значення знаходиться саме в тій позиції, куди варто помістити елемент z. У рядках 23–13 виконується установка значень вказівників для вставки z.

Рис. 23.3. Вставка елемента з ключем 13 у бінарне дерево пошуку. Світлі вузли вказують шлях від кореня до позиції вставки; пунктиром зазначена зв'язок, що додається при вставці нового елемента

Так само, як і інші елементарні операції над бінарним деревом пошуку, процедура Tree_Insert виконується за час $O(h)$ у дереві висотою h.

Видалення

Процедура видалення даного вузла z з бінарного дерева пошуку одержує як аргумент вказівник на z. Процедура розглядає три можливі ситуації, показані на

рис. 23.4. Якщо у вузла *z* немає дочірніх вузлів (рис. 23.4*а*), то ми просто змінюємо його батьківський вузол $p[z]$, заміняючи в ньому вказівник на *z* значенням NULL. Якщо у вузла *z* только один дочірній вузол (рис. 23.4*б)*, то ми видаляємо вузол *z*, створюючи новий зв'язок між батьківським і дочірнім вузлом вузла *z*. І нарешті, якщо у вузла *z* два дочірніх вузли (рис. 23.4*в)*, то ми знаходимо наступний за ним вузол *y*, у якого немає лівого дочірнього вузла, прибираємо його з позиції, де він знаходився раніше, шляхом створення нового зв'язку між його батьком і нащадком, і заміняємо їм вузол *z*.

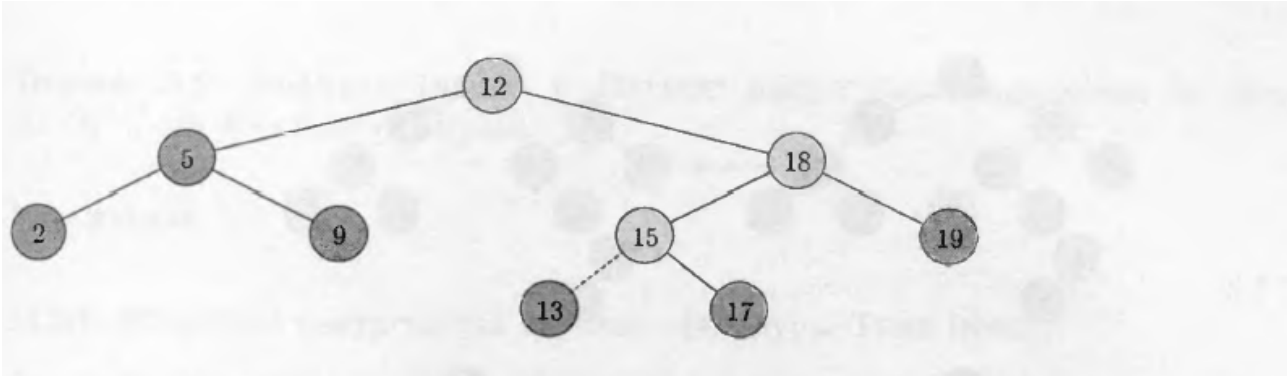

Рис. 23.3. Видалення вузла z *з бінарного дерева пошуку*

Код процедури Tree_Delete реалізує ці дії дещо не так, як вони описані:

```
Tree_Delete(T, z)
 1  if left[z] = NULL or right[z] = NULL
 2       then y ← z
 3       else y ← Tree_Sussessor(z)
 4  if left[y] ≠ NULL
 5       then x ← left[y]
 6       else x ← right[y]
 7  if x ≠ NULL
23        then p[x] ← p[y]
 9  if p[y] = NULL
10       then root[T] ← x
11       else if y = left[p[y]]
12            then left[p[y]] ← x
13            else right[p[y]] ← x
14  if y ≠ z
15       then key[z] ← key[y]
16            Копіювання супутніх даних y у z
23  return y
```

У рядках 1–3 алгоритм визначає вузол *y*, що вилучається, за допомогою “склейки” батька і нащадка. Цей вузол являє собою або вузол *z* (якщо у вузла *z* не більш одного дочірнього вузла), або вузол, що слідує за вузлом *z* (якщо в *z* два дочірніх вузли). Потім у рядках 4–6 вузлу *x* привласнюється вказівник на дочірній вузол вузла *y* або значення NULL, якщо в *y* немає дочірніх вузлів. Потім вузол *y* прибирається з дерева в рядках 7–13 шляхом зміни вказівників в $p[y]$ й *x*. Це видалення ускладнюється необхідністю коректного відпрацьовування граничних умов (коли *x* равно NULL чи коли *y* — кореневий вузол). І нарешті, у рядках 14–16, якщо вилучений вузол *y* був наступним за *z*, ми заміняємо ключ *z* і його значення ключем і значенням вузла *y*. Вилучений вузол *y* повертається в рядку 23, для того щоб процедура, що викликається могла при необхідності звільнити чи використати займану їм пам'ять. Час роботи описаної процедури з деревом висотою *h* складає $O(h)$.

Теорема 23.3. Операції вставки і видалення в бінарному дереві пошуку висоти h *можуть бути виконані за час* $O(h)$.

Література

Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ, 3-е издание. — М.: «Вильямс», 2013. — глава 12.