Wiktor Kuchta

6/6

Zauważmy, że każdy ciąg operacji rotacji jest odwracalny, zatem wystarczy pokazać, że możemy zmienić kształt każdego drzewa BST w ścieżkę (pochyloną w prawo).

Tezę pokażemy indukcją względem liczby wierzchołków. Drzewa o < 2 wierzchołkach już są takimi ścieżkami.

Weźmy drzewo o $\geqslant 2$ wierzchołkach. Z tezy indukcyjnej jego lewe poddrzewo L możemy zmienić w ścieżkę i korzystając z rotacji otrzymać drzewo o pustym lewym poddrzewie. Teraz korzystając z założenia indukcyjnego jego prawe poddrzewo zamieniamy w ścieżkę.

6/7

Nie jest oczywiste, jak interpretować *split* jako operacje modyfikującą drzewo w miejscu, zatem zaimplementujemy trwałe drzewo AVL z taką operacją.

```
(*
  node(l, v, r) - konstruktor, który obliczy wysokość itp.
  insert(t, v)  - drzewo t ze wstawionym v
*)

split(t, k) =
  case t of
  | Empty         => (Empty, Empty)
  | Node(l, v, r) =>
    case compare(k, v) of
    | LESS    => let (ll, lr) = split(l, k) in (ll, join(lr, v, r))
    | EQUAL   => (l, r)
    | GREATER => let (rl, rr) = split(r, k) in (join(l, v, rl), rr)

join(l, v, r) =
  if height(l) > height(r) + 1 then join_right(l, v, r) else
  if height(r) > height(l) + 1 then join_left(l, v, r) else
  node(l, v, r)

(* height(node(ll, lv, lr)) > height(r) + 1 *)
join_right((ll, lv, lr), v, r) =
  if height(lr) <= height(r) + 1 then
    let t' = node(lr, v, r) in
    if height(t') <= height(ll) + 1
      then node(ll, lv, t')
      else rotate_left(node(ll, lv, rotate_right(t')))
  else
    let t' = join_right(lr, v, r) in
    let t = node(ll, lv, t') in
    if height(t') <= height(l) + 1 then t
    else rotate_left(t)
```

Koszt operacji *join* to różnica między wysokościami drzew.

6/8

Do implementacji struktury możemy użyć BST (np. AVL), w którym węzły dodatkowo spamiętują minimum, maksimum i mindiff w poddrzewie.

W implementacji drzew AVL, kiedy konstruujemy nowy węzeł (nową wersję węzła), to potrzebujemy mieć tylko lewe poddrzewo, prawe poddrzewo i element, który ma być trzymany w tym węźle. Z tych informacji łatwo wyliczyć żądane wartości:

$$
\begin{aligned}
&\mathrm{minval}(\mathit{Empty}) &&= \infty\\
&\mathrm{minval}(\mathit{Node}(l,v,r)) &&= \min\{\mathrm{minval}(l), v\}\\
&\mathrm{maxval}(\mathit{Empty}) &&= -\infty\\
&\mathrm{maxval}(\mathit{Node}(l,v,r)) &&= \max\{v, \mathrm{maxval}(r)\}\\
&\mathrm{mindiff}(\mathit{Empty}) &&= \infty\\
&\mathrm{mindiff}(\mathit{Node}(l,v,r)) &&= \min\{\mathrm{mindiff}(l), v-\mathrm{maxval}(l), \mathrm{minval}(r)-v, \mathrm{mindiff}(r)\}
\end{aligned}
$$

Operacje *insert*, *delete* pozostają bez zmian. Operacja *mindiff* to po prostu odczytanie tej wartości w korzeniu.

6/9

Nie potrzebujemy dwóch bitów informacji w *każdym* węźle: Jeśli któryś z synów jest pusty, to współczynnik zrównoważenia łatwo wyliczyć. Tylko wezły z dwoma niepustymi synami potrzebują dwóch bitów informacji.

Jeśli potrzebujemy mieć jednorodnej reprezentacji dla każdego węzła, to zauważmy, że te dwa bity dla węzła z dwoma niepustymi synami możemy „zepchnąć” do synów – jeden bit do lewego, jeden bit do prawego. Wtedy każdy węzeł potrzebuje spamiętywać jeden dodatkowy bit.

7/1

Chcemy zminimalizować oczekiwany koszt wykonywania operacji, a więc $\sum_{i=1}^{n} p_i d_i$, gdzie d_i to liczba wierzchołków odwiedzonych na ścieżce z korzenia BST do wierzchołka i. Inaczej mówiąc, minimalizujemy ważoną sumę długości ścieżek do liści drzewa.

Niech $E_{i,j}$ oznacza optymalną sumę ważoną dla podproblemu $a_i, a_{i+1}, \ldots, a_j$ z wagami $p_i, p_{i+1}, \ldots, p_j$. Mamy

$$
\begin{aligned}
E_{i,i} &= p_i, && (1 \leqslant i \leqslant n)\\
E_{i,j} &= \sum_{k=i}^{j} p_k + \min_{i<r<j}(E_{i,r-1} + E_{r+1,j}), && (1 \leqslant i < j \leqslant n)
\end{aligned}
$$

co nam daje algorytm sześcienny. Aby odtworzyć drzewo, musimy dodatkowo spamiętywać, jaki r został wybrany w każdym $E_{i,j}$. To nam daje węzeł drzewa i rekurencyjnie możemy odtworzyć jego poddrzewa.

7/3

Rozważmy wstawienie n-elementów do n-elementowej tablicy haszującej. Niech

$$X_j = \begin{cases} 1 & \text{jeśli } j\text{-ty indeks w tablicy pozostał pusty,} \\ 0 & \text{w p. w.} \end{cases}$$

Mamy $E(X_j) = (1-\frac{1}{n})^n$, bo każdy klucz musi trafić do indeksu różnego od j. Liczba oczekiwanych pustych kubełków to $X_1 + \ldots + X_n$, a więc

$$E(X_1 + \ldots + X_n) = E(X_1) + \ldots + E(X_n) = n\left(1 - \frac{1}{n}\right)^n \to \frac{n}{e}.$$