Binary Indexed Trees, Splay Trees and Scapegoat Trees

Iván Szabolcs

2017 tavasz

A Task

A Task

- Bejön sok int: $a_1, a_2, \ldots, a_n$ (sok: $O(n)$ memóriánk van, $\Omega(n^2)$ memóriánk nincs)

A Task

- Bejön sok int: $a_1, a_2, \ldots, a_n$ (sok: $O(n)$ memóriánk van, $\Omega(n^2)$ memóriánk nincs)
- Query: i, j, $1 \leq i \leq j \leq n$, adjuk vissza $a_i + a_{i+1} + \ldots + a_n$-t.

Triviális megoldás:

A Task

- Bejön sok int: $a_1, a_2, \ldots, a_n$ (sok: $O(n)$ memóriánk van, $\Omega(n^2)$ memóriánk nincs)
- Query: i, j, $1 \leq i \leq j \leq n$, adjuk vissza $a_i + a_{i+1} + \ldots + a_n$-t.

Triviális megoldás:

- letesszük tömbbe az a_i-ket: $O(n)$ idő a beolvasás (preprocess)

A Task

- Bejön sok int: $a_1, a_2, \ldots, a_n$ (sok: $O(n)$ memóriánk van, $\Omega(n^2)$ memóriánk nincs)
- Query: i, j, $1 \leq i \leq j \leq n$, adjuk vissza $a_i + a_{i+1} + \ldots + a_n$-t.

Triviális megoldás:

- letesszük tömbbe az a_i-ket: $O(n)$ idő a beolvasás (preprocess)
- intervallum querynél kiszámítjuk: $O(n)$ idő a query.

Okosabb megoldás

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$
- kitöltés $O(n)$ idő: $s[0] = 0$, $s[i+1] = s[i] + a_{i+1}$

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$
- kitöltés $O(n)$ idő: $s[0] = 0$, $s[i+1] = s[i] + a_{i+1}$
- ezek után query $O(1)$ idő: $a_i + \ldots + a_j = s[j] - s[i-1]$.

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$
- kitöltés $O(n)$ idő: $s[0] = 0$, $s[i+1] = s[i] + a_{i+1}$
- ezek után query $O(1)$ idő: $a_i + \ldots + a_j = s[j] - s[i-1]$.

Új feature request: támogatni kéne az updatet is, a_i változhat.

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$
- kitöltés $O(n)$ idő: $s[0] = 0$, $s[i+1] = s[i] + a_{i+1}$
- ezek után query $O(1)$ idő: $a_i + \ldots + a_j = s[j] - s[i-1]$.

Új feature request: támogatni kéne az updatet is, a_i változhat.

- a triviális megoldás időigénye $O(1)$ update, $O(n)$ query

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$
- kitöltés $O(n)$ idő: $s[0] = 0$, $s[i+1] = s[i] + a_{i+1}$
- ezek után query $O(1)$ idő: $a_i + \ldots + a_j = s[j] - s[i-1]$.

Új feature request: támogatni kéne az updatet is, a_i változhat.

- a triviális megoldás időigénye $O(1)$ update, $O(n)$ query
- a prefixösszeges megoldásé $O(n)$ update (!), $O(1)$ query

Okosabb megoldás

Prefixösszeg-tömbbel sokkal gyorsabb a lekérdezés:

- létrehozunk egy $s[0 \ldots n]$ tömböt
- $s[i] := a_1 + \ldots + a_i$
- kitöltés $O(n)$ idő: $s[0] = 0$, $s[i+1] = s[i] + a_{i+1}$
- ezek után query $O(1)$ idő: $a_i + \ldots + a_j = s[j] - s[i-1]$.

Új feature request: támogatni kéne az updatet is, a_i változhat.

- a triviális megoldás időigénye $O(1)$ update, $O(n)$ query
- a prefixösszeges megoldásé $O(n)$ update (!), $O(1)$ query

Próbáljunk meg egy kiegyensúlyozattabb megoldást adni.

Gyorsabb prefixösszeg

Gyorsabb prefixösszeg

- A prefixösszeges megoldás $O(n)$ update költsége onnan jön, hogy egy a_i szerepel az összes s_j, $j \geq i$-ben tagként és ha mindet letároljuk, mindet változtatni kell.

Gyorsabb prefixösszeg

- A prefixösszeges megoldás $O(n)$ update költsége onnan jön, hogy egy a_i szerepel az összes s_j, $j \geq i$-ben tagként és ha mindet letároljuk, mindet változtatni kell.
- Tehát a gond: a letárolt intervallum túl hosszúak lehetnek, túl előre néznek vissza.

Gyorsabb prefixösszeg

- A prefixösszeges megoldás $O(n)$ update költsége onnan jön, hogy egy a_i szerepel az összes s_j, $j \geq i$-ben tagként és ha mindet letároljuk, mindet változtatni kell.
- Tehát a gond: a letárolt intervallum túl hosszúak lehetnek, túl előre néznek vissza.
- Ötlet: tároljunk intervallum-összeget

Gyorsabb prefixösszeg

- A prefixösszeges megoldás $O(n)$ update költsége onnan jön, hogy egy a_i szerepel az összes s_j, $j \geq i$-ben tagként és ha mindet letároljuk, mindet változtatni kell.
- Tehát a gond: a letárolt intervallum túl hosszúak lehetnek, túl előre néznek vissza.
- Ötlet: tároljunk intervallum-összeget
  - ,,ügyesen" megválasztott intervallumokra

Gyorsabb prefixösszeg

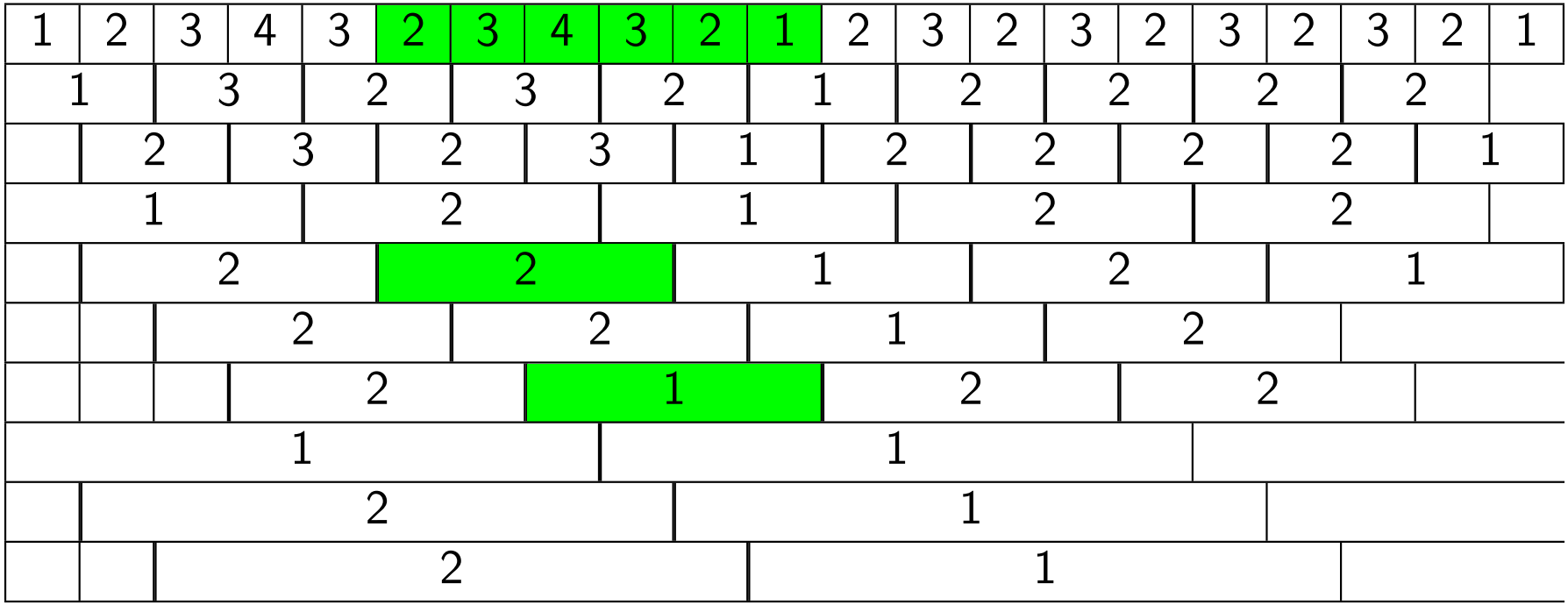
- A prefixösszeges megoldás $O(n)$ update költsége onnan jön, hogy egy a_i szerepel az összes s_j, $j \geq i$-ben tagként és ha mindet letároljuk, mindet változtatni kell.
- Tehát a gond: a letárolt intervallum túl hosszúak lehetnek, túl előre néznek vissza.
- Ötlet: tároljunk intervallum-összeget
  - ,,ügyesen" megválasztott intervallumokra
  - ,,ügyes": a prefixösszeg kevés ilyen intervallumból kirakható legyen

Gyorsabb prefixösszeg

- A prefixösszeges megoldás $O(n)$ update költsége onnan jön, hogy egy a_i szerepel az összes s_j, $j \geq i$-ben tagként és ha mindet letároljuk, mindet változtatni kell.
- Tehát a gond: a letárolt intervallum túl hosszúak lehetnek, túl előre néznek vissza.
- Ötlet: tároljunk intervallum-összeget
  - ,,ügyesen" megválasztott intervallumokra
  - ,,ügyes": a prefixösszeg kevés ilyen intervallumból kirakható legyen
  - ,,ügyes": egy elem megváltoztatása kevés ilyen intervallumra hasson

Recall RMQ

Volt valami Range Minimum Query...



Query

Az $[i \ldots j]$ intervallum minumum eleme:

- vegyük a legnagyobb 2^k kettő-hatványt, ami belefér az intervallumba;
- a válasz $A[i, i + 2^k - 1]$ és $A[j + 1 - 2^k, j]$ minimuma lesz – konstans idő.

RMQ gondok

RMQ gondok

- Az összegre nem kéne átlapoló intervallumokkal számolnunk

RMQ gondok

- Az összegre nem kéne átlapoló intervallumokkal számolnunk
- Egy-egy index sok intervallumban is benne lehet (akár lineáris sokban is)

RMQ gondok

- Az összegre nem kéne átlapoló intervallumokkal számolnunk
- Egy-egy index sok intervallumban is benne lehet (akár lineáris sokban is)
- Másképp érdemes kiválasztanunk az intervallumokat

A legkisebb beállított BIT

Kis kitérő: írjunk függvényt, ami kinullázza az input $m > 0$ egész szám legkisebb bitjét.

A legkisebb beállított BIT

Kis kitérő: írjunk függvényt, ami kinullázza az input $m > 0$ egész szám legkisebb bitjét. Pl. $5 \mapsto 4$, $7 \mapsto 6$, $10 \mapsto 8$, $100 \mapsto 96$.
$101 \mapsto 100$, $111 \mapsto 110$, $1010 \mapsto 1000$, $1100100 \mapsto 1100000$.

A legkisebb beállított BIT

Kis kitérő: írjunk függvényt, ami kinullázza az input $m > 0$ egész szám legkisebb bitjét. Pl. $5 \mapsto 4$, $7 \mapsto 6$, $10 \mapsto 8$, $100 \mapsto 96$.
$101 \mapsto 100$, $111 \mapsto 110$, $1010 \mapsto 1000$, $1100100 \mapsto 1100000$.

- Ha kivonunk m-ből 1-et, az ezt a bitet kinullázza, a mögötte levő nullákból 1-es lesz, a többi nem változik

A legkisebb beállított BIT

Kis kitérő: írjunk függvényt, ami kinullázza az input $m > 0$ egész szám legkisebb bitjét. Pl. $5 \mapsto 4$, $7 \mapsto 6$, $10 \mapsto 8$, $100 \mapsto 96$.
$101 \mapsto 100$, $111 \mapsto 110$, $1010 \mapsto 1000$, $1100100 \mapsto 1100000$.

- Ha kivonunk m-ből 1-et, az ezt a bitet kinullázza, a mögötte levő nullákból 1-es lesz, a többi nem változik
- Tehát : `m & (m-1)`

A legkisebb beállított BIT

Kis kitérő: írjunk függvényt, ami kinullázza az input $m > 0$ egész szám legkisebb bitjét. Pl. $5 \mapsto 4$, $7 \mapsto 6$, $10 \mapsto 8$, $100 \mapsto 96$.
$101 \mapsto 100$, $111 \mapsto 110$, $1010 \mapsto 1000$, $1100100 \mapsto 1100000$.

- Ha kivonunk m-ből 1-et, az ezt a bitet kinullázza, a mögötte levő nullákból 1-es lesz, a többi nem változik
- Tehát : `m & (m-1)`
- Ez elég gyors

A legkisebb beállított BIT

Kis kitérő: írjunk függvényt, ami kinullázza az input $m > 0$ egész szám legkisebb bitjét. Pl. $5 \mapsto 4$, $7 \mapsto 6$, $10 \mapsto 8$, $100 \mapsto 96$.
$101 \mapsto 100$, $111 \mapsto 110$, $1010 \mapsto 1000$, $1100100 \mapsto 1100000$.

- Ha kivonunk m-ből 1-et, az ezt a bitet kinullázza, a mögötte levő nullákból 1-es lesz, a többi nem változik
- Tehát : `m & (m-1)`
- Ez elég gyors

Nevezzük ezt a függvényt $f(m)$-nek. Nyilván $f(m) < m$.

Bit kinullázásának iterálása

A terv: $s[m]$ tárolja az $a[f(m)+1]+\ldots+a[m]$ intervallum-összeget.

Bit kinullázásának iterálása

A terv: $s[m]$ tárolja az $a[f(m)+1]+\ldots+a[m]$ intervallum-összeget.

- PI. $s[5]=a[5]$, $s[7]=a[7]$, $s[10]=a[9]+a[10]$, $s[100]=a[97]+\ldots+a[100]$.

Bit kinullázásának iterálása

A terv: $s[m]$ tárolja az $a[f(m)+1]+\ldots+a[m]$ intervallum-összeget.

- PI. $s[5]=a[5]$, $s[7]=a[7]$, $s[10]=a[9]+a[10]$, $s[100]=a[97]+\ldots+a[100]$.
- Akkor $a[1]+\ldots+a[m]$-et hogy kapjuk?

Bit kinullázásának iterálása

A terv: $s[m]$ tárolja az $a[f(m)+1]+\ldots+a[m]$ intervallum-összeget.

- PI. $s[5]=a[5]$, $s[7]=a[7]$, $s[10]=a[9]+a[10]$, $s[100]=a[97]+\ldots+a[100]$.
- Akkor $a[1]+\ldots+a[m]$-et hogy kapjuk?
- $s[m]+s[f(m)]+s[f(f(m))]+\ldots+s[0]$. (legyen $s[0]=0$.)

Bit kinullázásának iterálása

A terv: $s[m]$ tárolja az $a[f(m)+1]+\ldots+a[m]$ intervallum-összeget.

- PI. $s[5]=a[5]$, $s[7]=a[7]$, $s[10]=a[9]+a[10]$, $s[100]=a[97]+\ldots+a[100]$.
- Akkor $a[1]+\ldots+a[m]$-et hogy kapjuk?
- $s[m]+s[f(m)]+s[f(f(m))]+\ldots+s[0]$. (legyen $s[0]=0$.)
- Ezt mennyi idő kiszámolni input $0 \leq m \leq n$-re? $\log n$.

Bit kinullázásának iterálása

A terv: $s[m]$ tárolja az $a[f(m)+1]+\ldots+a[m]$ intervallum-összeget.

- PI. $s[5]=a[5]$, $s[7]=a[7]$, $s[10]=a[9]+a[10]$, $s[100]=a[97]+\ldots+a[100]$.
- Akkor $a[1]+\ldots+a[m]$-et hogy kapjuk?
- $s[m]+s[f(m)]+s[f(f(m))]+\ldots+s[0]$. (legyen $s[0]=0$.)
- Ezt mennyi idő kiszámolni input $0 \leq m \leq n$-re? $\log n$.
- Tehát ilyen reprezentációval a query cost $O(\log n)$-re áll be.

Intervallum-összeg kicsit okosabban

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 1100100110010$ és $j = 1100101000101$:

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010$0110010 és $j = 110010$1000101:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010$0110010 és $j = 110010$1000101:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010$0110010 és $j = 110010$1000101:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`!

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig?

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig? Ha i-ből elindulunk, akkor is ide kell érjünk erre a számra

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig? Ha i-ből elindulunk, akkor is ide kell érjünk erre a számra
  - `sum -= s[110010 0110010]`, nullázzuk `i := f(i)`

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig? Ha i-ből elindulunk, akkor is ide kell érjünk erre a számra
  - `sum -= s[110010 0110010]`, nullázzuk `i := f(i)`
  - `sum -= s[110010 0110000]`, nullázzuk

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 1100100110010$ és $j = 1100101000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig? Ha i-ből elindulunk, akkor is ide kell érjünk erre a számra
  - `sum -= s[110010 0110010]`, nullázzuk `i := f(i)`
  - `sum -= s[110010 0110000]`, nullázzuk
  - `sum -= s[110010 0100000]`, nullázzuk,

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 1100100110010$ és $j = 1100101000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig? Ha i-ből elindulunk, akkor is ide kell érjünk erre a számra
  - `sum -= s[110010 0110010]`, nullázzuk `i := f(i)`
  - `sum -= s[110010 0110000]`, nullázzuk
  - `sum -= s[110010 0100000]`, nullázzuk,
  - `sum -= s[110010 0000000]`, és most `i=j`, vége, az összeg helyes

Intervallum-összeg kicsit okosabban

- Nem feltétlen kell kiszámolnunk a teljes prefixösszeget, ha a célunk az intervallum-összeg!
- Például, ha $i - 1 = 110010\,0110010$ és $j = 110010\,1000101$:
- Elindulunk j-ből, nullázzuk az eddigi összeget `sum := 0`
  - `sum += s[110010 1000101]`, nullázzuk az utolsó bitet (`j := f(j)`)
  - `sum += s[110010 1000100]`, nullázzuk az utolsó bitet
  - `sum += s[110010 1000000]`, nullázzuk az utolsó bitet
  - `sum += s[110010 0000000]`, és most már `j<i`! Minek mennénk le 0-ig? Ha i-ből elindulunk, akkor is ide kell érjünk erre a számra
  - `sum -= s[110010 0110010]`, nullázzuk `i := f(i)`
  - `sum -= s[110010 0110000]`, nullázzuk
  - `sum -= s[110010 0100000]`, nullázzuk,
  - `sum -= s[110010 0000000]`, és most `i=j`, vége, az összeg helyes
- Továbbra is $O(\log n)$ persze a költség, de gyorsabb: a közös bináris prefix 1-eseivel nem számolunk.

Az update cost

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik a `[110011]`.

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték
- Hogy generáljuk ezeket ki gyorsan?

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték
- Hogy generáljuk ezeket ki gyorsan?
- Vegyük a legutolsó 1-es helyiértéket és... adjuk hozzá az eredeti indexhez

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték
- Hogy generáljuk ezeket ki gyorsan?
- Vegyük a legutolsó 1-es helyiértéket és... adjuk hozzá az eredeti indexhez
- pl. 110011+000001=110100, 110100+000100=111000, 111000+001000=1000000 stb.

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték
- Hogy generáljuk ezeket ki gyorsan?
- Vegyük a legutolsó 1-es helyiértéket és... adjuk hozzá az eredeti indexhez
- pl. 110011+000001=110100, 110100+000100=111000, 111000+001000=1000000 stb.
- A legutolsó 1-es helyiértéket (amit kinullázni ki tudtunk) hogy kapjuk?

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték
- Hogy generáljuk ezeket ki gyorsan?
- Vegyük a legutolsó 1-es helyiértéket és... adjuk hozzá az eredeti indexhez
- pl. 110011+000001=110100, 110100+000100=111000, 111000+001000=1000000 stb.
- A legutolsó 1-es helyiértéket (amit kinullázni ki tudtunk) hogy kapjuk?
- `n & (-n)`

Az update cost

- Megváltoztatjuk $a[i]$-t. Melyik $s[i]$-k változnak?
- Pl. megváltozik `a[110011]`.
- Akkor változik: `s[110011]`, `s[110100]`, `s[111000]`, `s[1000000]`, `s[10000000]`,... amíg van helyiérték
- Hogy generáljuk ezeket ki gyorsan?
- Vegyük a legutolsó 1-es helyiértéket és... adjuk hozzá az eredeti indexhez
- pl. 110011+000001=110100, 110100+000100=111000, 111000+001000=1000000 stb.
- A legutolsó 1-es helyiértéket (amit kinullázni ki tudtunk) hogy kapjuk?
- `n & (-n)`
- Ez így $O(\log n)$ update costot (és $O(\log n)$ query costot) ad.

Inicializálás

Inicializálás

- Ha az `a[1],...,a[n]` tömbbel inicializálunk, persze ez ad egy $O(n \log n)$-es inicializálási costot

Inicializálás

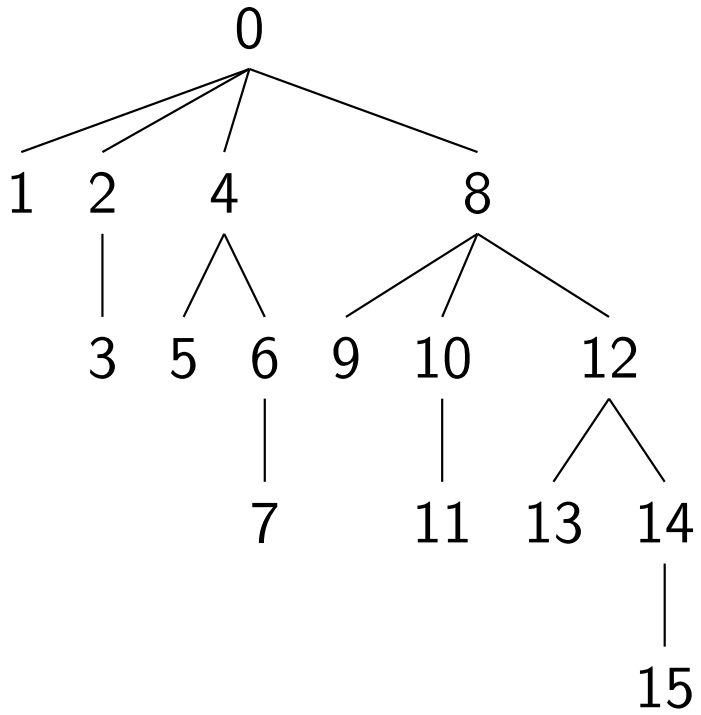
- Ha az `a[1],...,a[n]` tömbbel inicializálunk, persze ez ad egy $O(n \log n)$-es inicializálási costot
- Ennél gyorsabban is lehet meghatározni `s[m]`-et:

```
for i = 1 .. n s[i] := a[i]
for i = 1 .. n
  j := i + (i & (-i)) //a leghatso 0-t 1-re allitjuk
  if( j <= n ) s[j] += s[i]
```

Binary Indexed Tree, Fenwick Tree

Binary Indexed Tree, Fenwick Tree

Ha fába rendezzük a tömbindexeket és az m elem apja az $f(m)$ elem, akkor így hívjuk.



(a csúcsokon a számok a tömbindexek, az adatot nem jelenítettem meg)

A Splay Treeről még egy kicsit

Emlékeztető: a Splay Tree

A Splay Treeről még egy kicsit

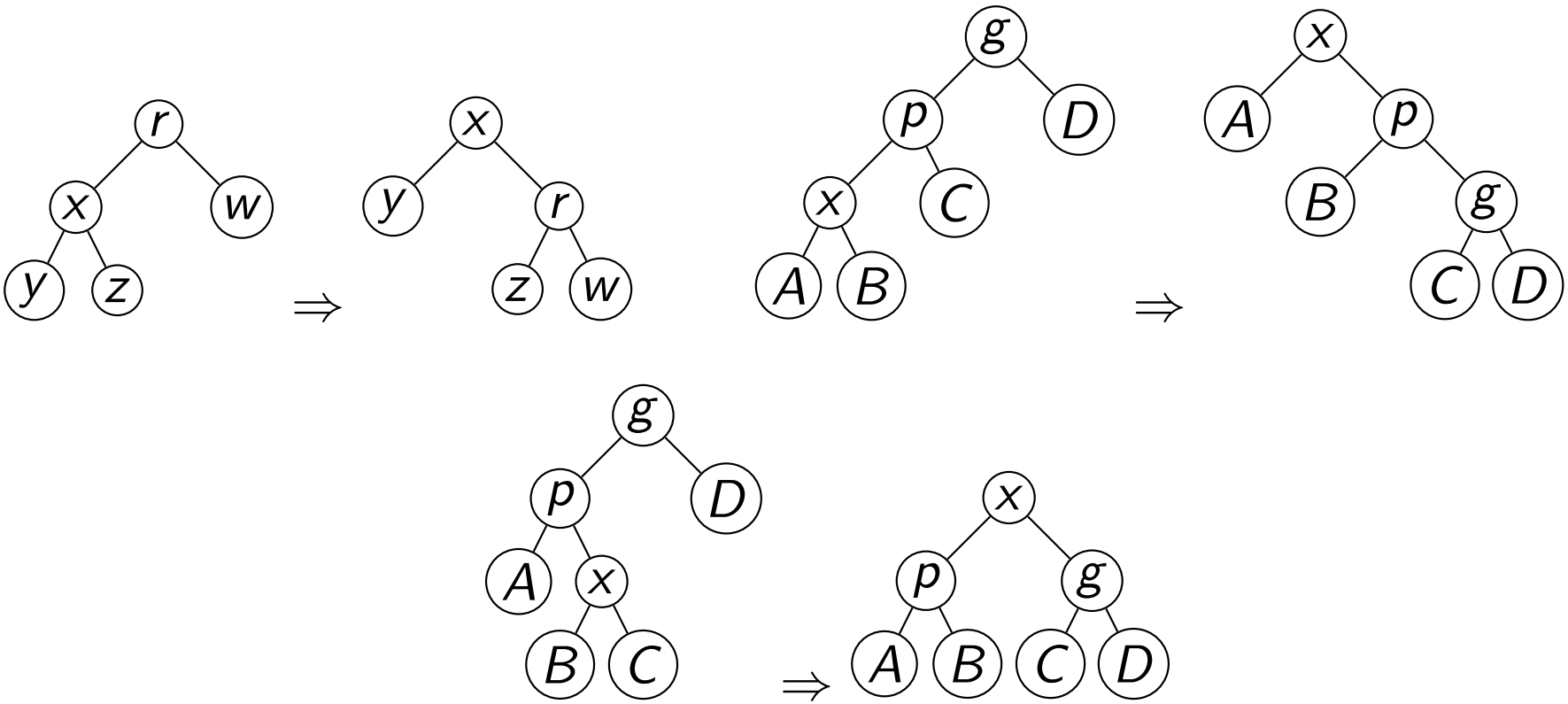
Emlékeztető: a Splay Tree

- bináris keresőfa

A Splay Treeről még egy kicsit

Emlékeztető: a Splay Tree

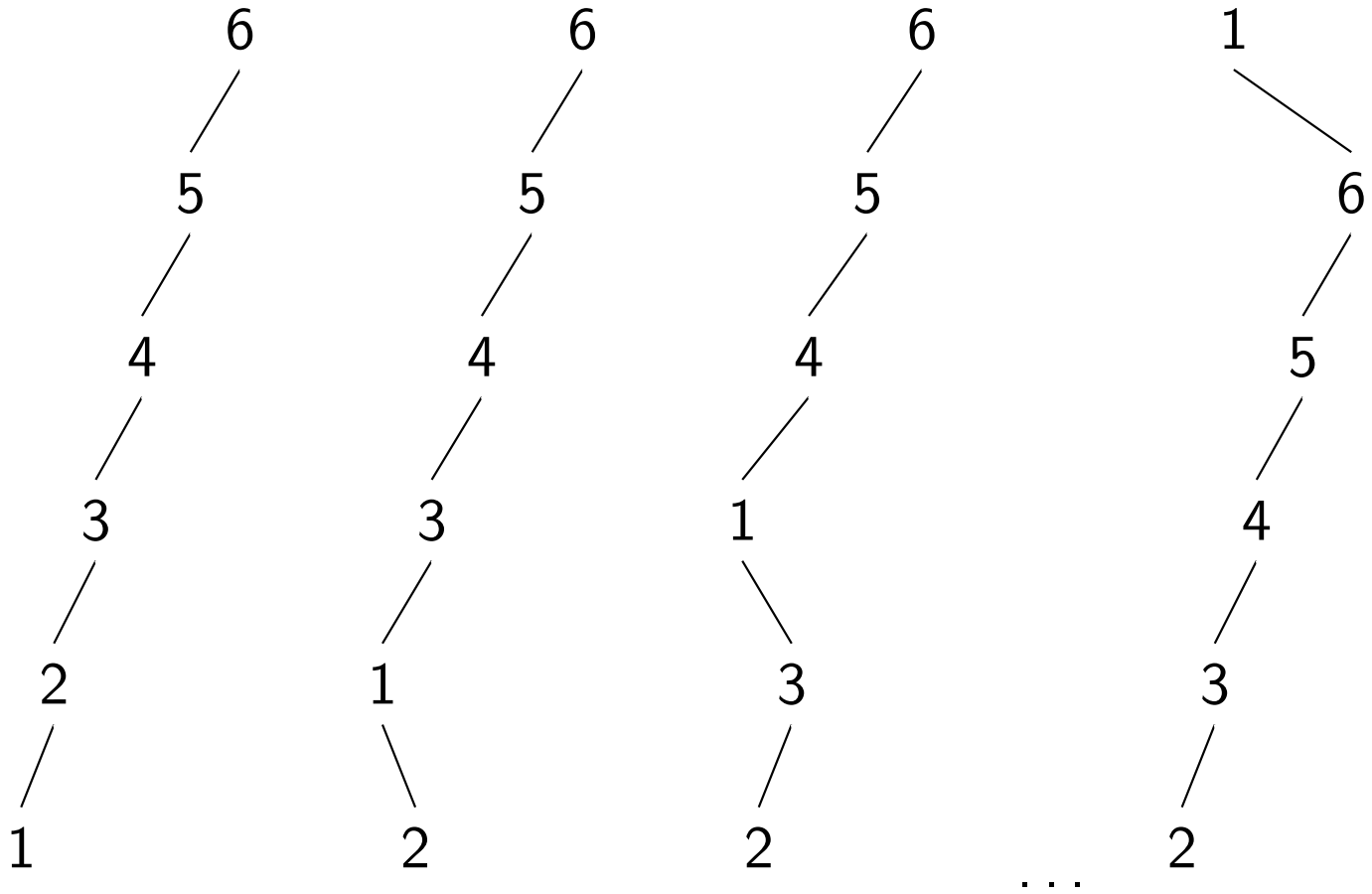
- bináris keresőfa
- az utoljára keresett csúcsot felforgatja gyökérnek



Módosított forgatás ám ez!

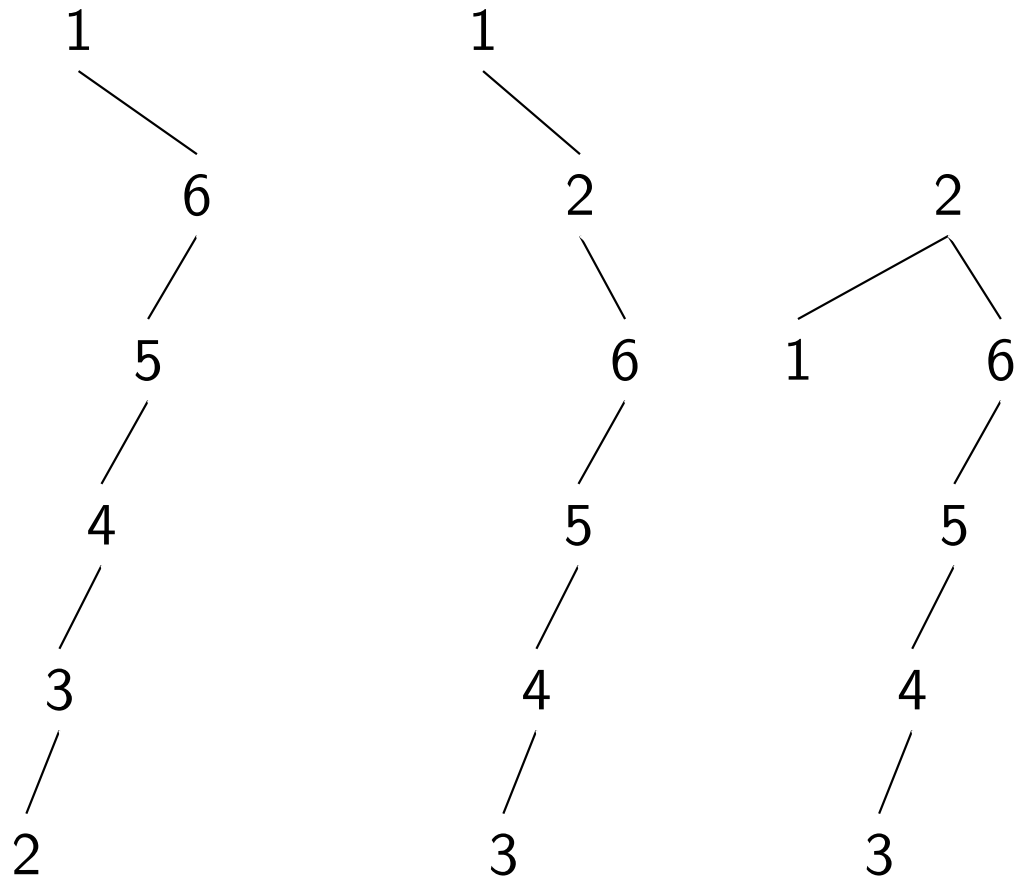
Default forgatással

Touch 1:



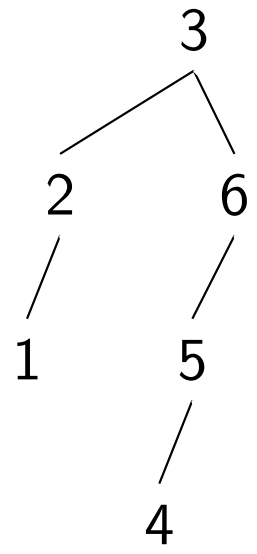
Default forgatással

Touch 2:

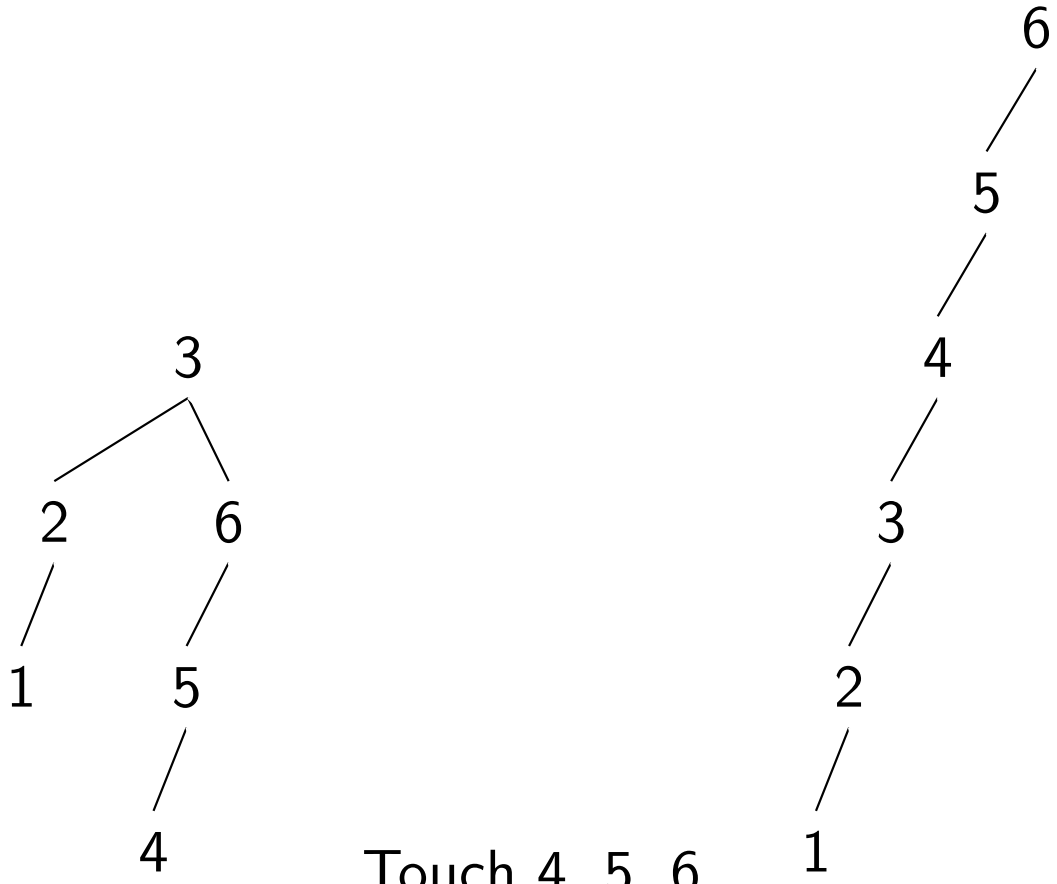


...

Touch 3...



Default forgatással

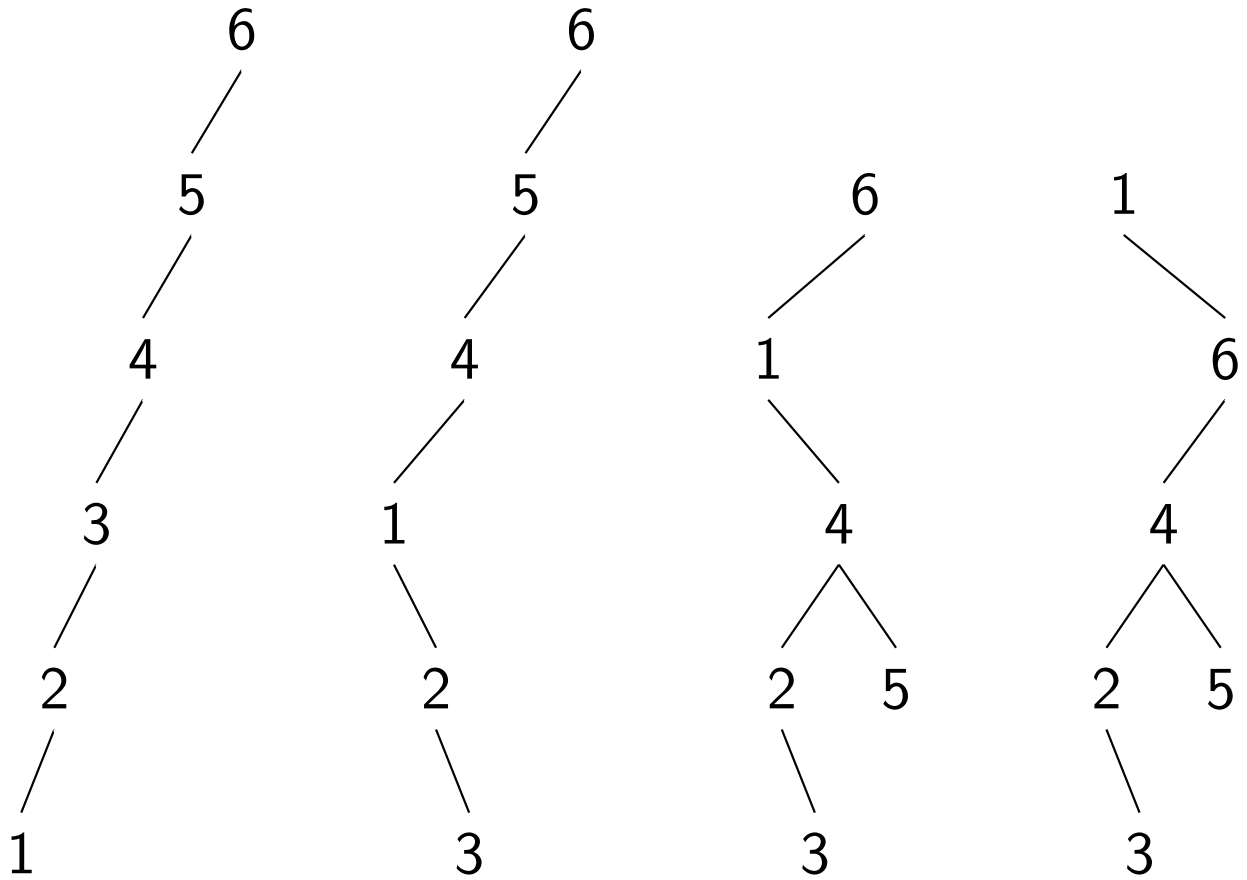


Touch 4, 5, 6...

Ez így n lekérdezés összesen $\Theta(n^2)$ költséggel! És ismételhető!

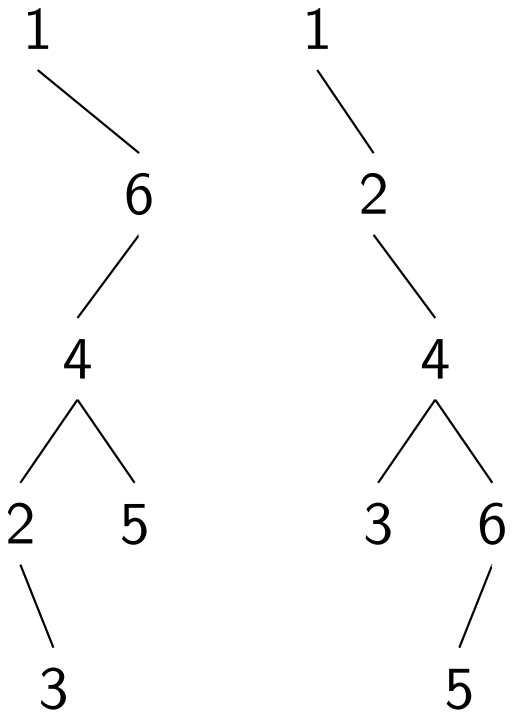
Splay fával

Touch 1:

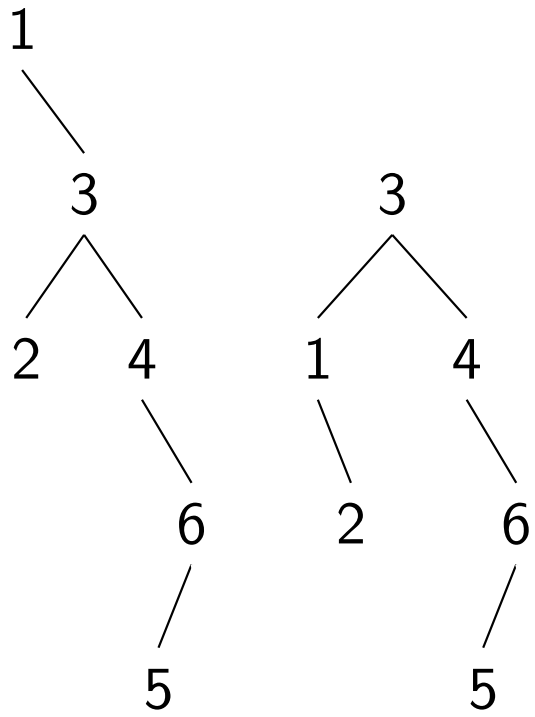


Splay fával

Touch 2:

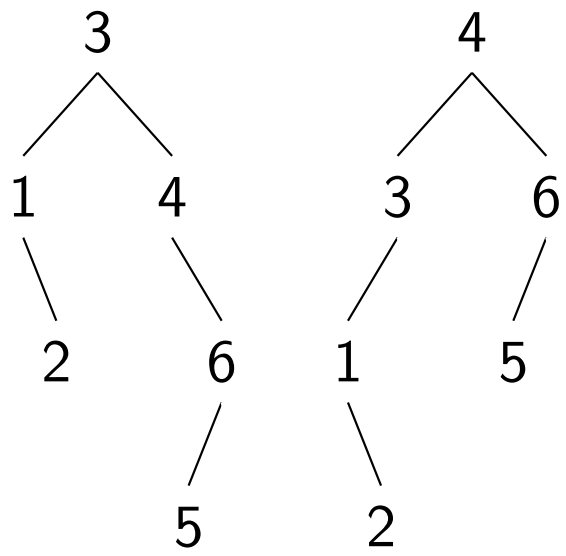


Touch 3:

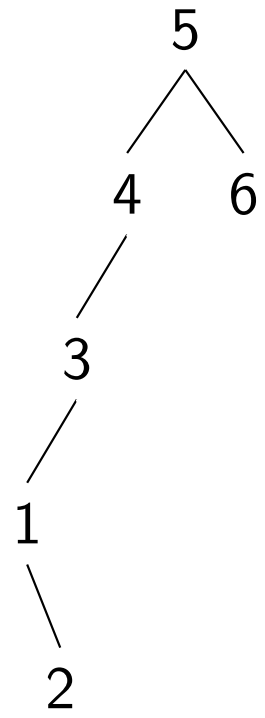


Splay fával

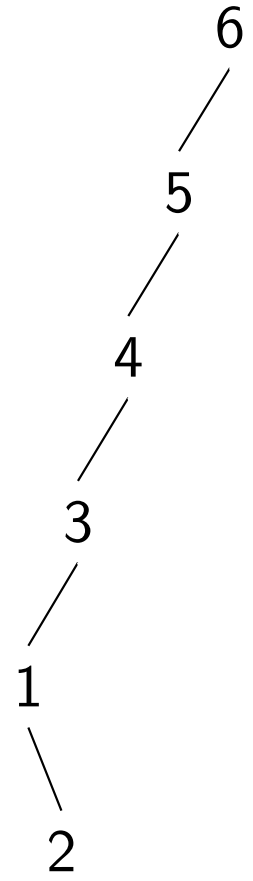
Touch 4:



Touch 5:

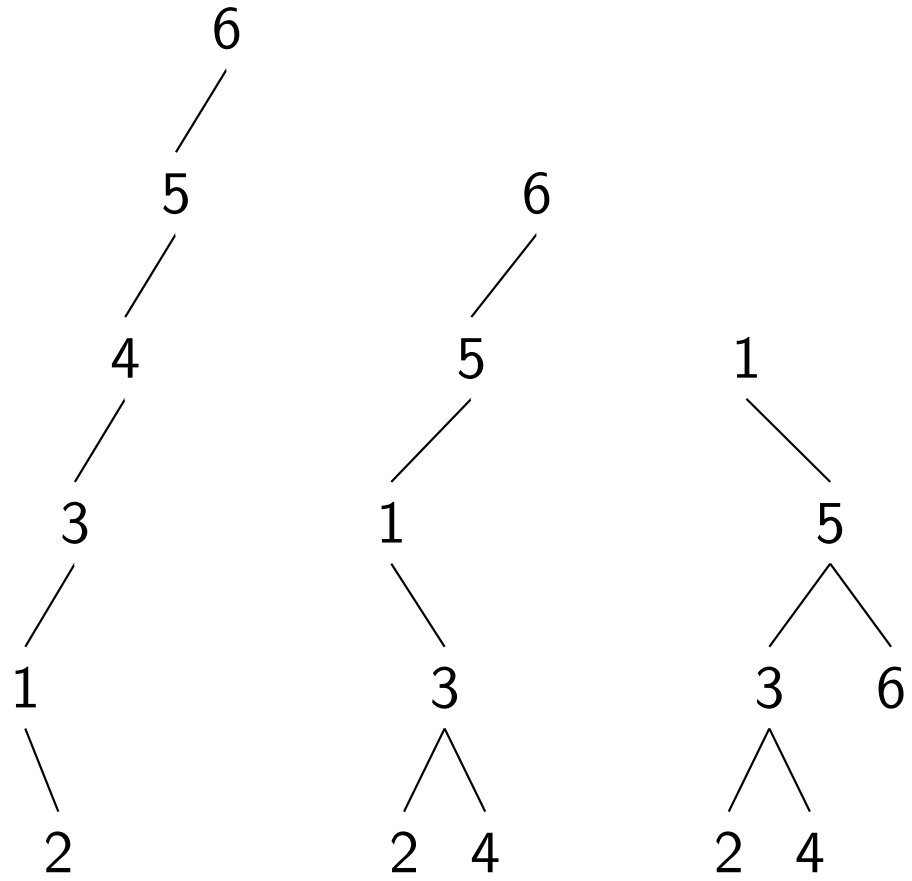


Touch 6:

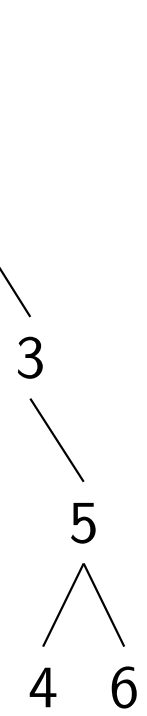


Splay fával

Touch 1, megint:

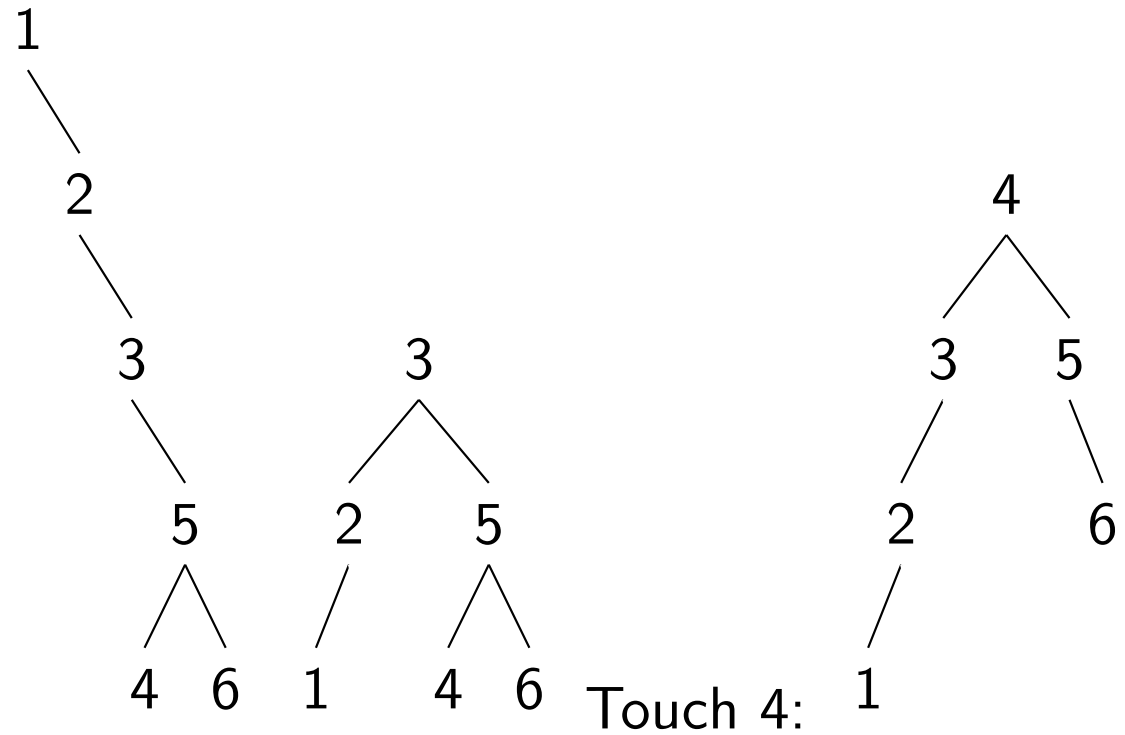


Touch 2:



Splay fával

Touch 3:

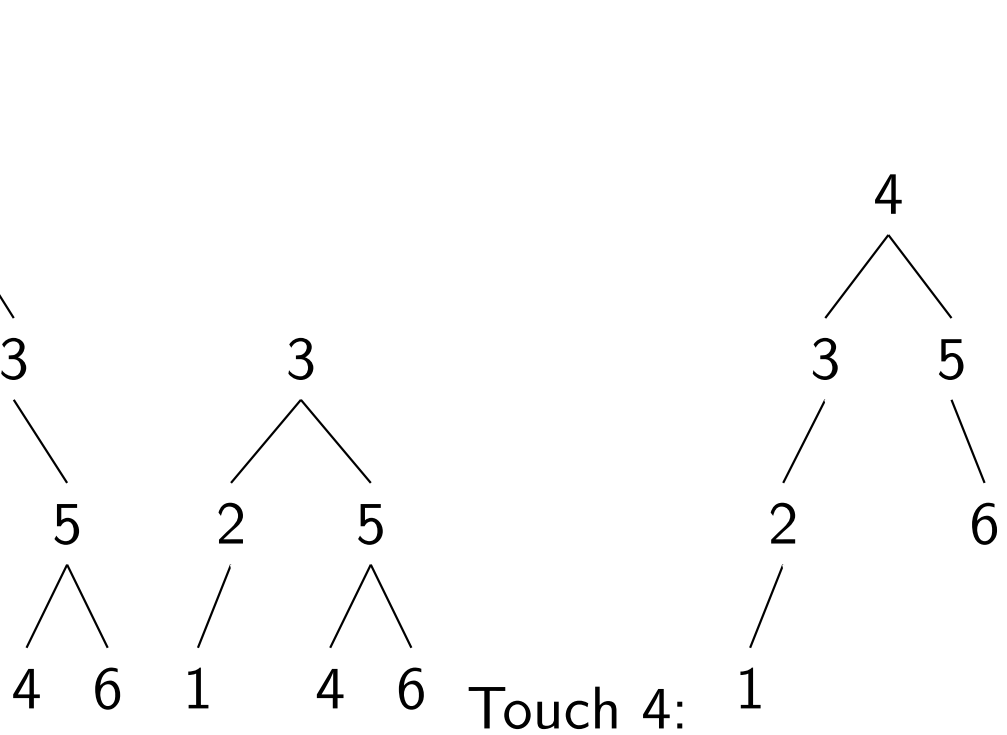


Touch 4:

Touch 5, 6: vissza az eredetihez.

Splay fával

Touch 3:



Touch 4:

Touch 5, 6: vissza az eredetihez. De az összköltség csak $O(n \log n)$ lesz.

A Splay Tree műveletei

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t
- split: Ha S Splay fa és x egy érték, akkor kettévághatjuk a $\leq x$ és a $> x$ értékeket tartalmazó fákra:

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t
- split: Ha S Splay fa és x egy érték, akkor kettévághatjuk a $\leq x$ és a $> x$ értékeket tartalmazó fákra:
  - Rákeresünk x-re (ha nincs, akkor az x-nél első kisebb elemet vesszük, ha az sincs, akkor a $\leq x$ fa üres, a $> x$ fa S)

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t
- split: Ha S Splay fa és x egy érték, akkor kettévághatjuk a $\leq x$ és a $> x$ értékeket tartalmazó fákra:
  - Rákeresünk x-re (ha nincs, akkor az x-nél első kisebb elemet vesszük, ha az sincs, akkor a $\leq x$ fa üres, a $> x$ fa S)
  - Levágjuk a gyökér jobb fiát, az lesz a $> x$ fa, ami marad, a $\leq x$ fa

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t
- split: Ha S Splay fa és x egy érték, akkor kettévághatjuk a $\leq x$ és a $> x$ értékeket tartalmazó fákra:
  - Rákeresünk x-re (ha nincs, akkor az x-nél első kisebb elemet vesszük, ha az sincs, akkor a $\leq x$ fa üres, a $> x$ fa S)
  - Levágjuk a gyökér jobb fiát, az lesz a $> x$ fa, ami marad, a $\leq x$ fa
- insert: S-be x beszúrásakor mint rendes keresőfába megérintjük x-et, így ő lesz az új gyökércsúcs

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t
- split: Ha S Splay fa és x egy érték, akkor kettévághatjuk a $\leq x$ és a $> x$ értékeket tartalmazó fákra:
  - Rákeresünk x-re (ha nincs, akkor az x-nél első kisebb elemet vesszük, ha az sincs, akkor a $\leq x$ fa üres, a $> x$ fa S)
  - Levágjuk a gyökér jobb fiát, az lesz a $> x$ fa, ami marad, a $\leq x$ fa
- insert: S-be x beszúrásakor mint rendes keresőfába megérintjük x-et, így ő lesz az új gyökércsúcs
- delete: törlünk mint bináris keresőfából, majd megérintjük a törölt node apját.

A Splay Tree műveletei

- join: Ha S és T Splay fák, S minden eleme kisebb T minden eleménél, akkor uniójuk:
  - megérintjük S legnagyobb elemét (jobb lent)
  - ezután ő lesz S gyökere, nincs jobb fia
  - idekötjük T-t
- split: Ha S Splay fa és x egy érték, akkor kettévághatjuk a $\leq x$ és a $> x$ értékeket tartalmazó fákra:
  - Rákeresünk x-re (ha nincs, akkor az x-nél első kisebb elemet vesszük, ha az sincs, akkor a $\leq x$ fa üres, a $> x$ fa S)
  - Levágjuk a gyökér jobb fiát, az lesz a $> x$ fa, ami marad, a $\leq x$ fa
- insert: S-be x beszúrásakor mint rendes keresőfába megérintjük x-et, így ő lesz az új gyökércsúcs
- delete: törlünk mint bináris keresőfából, majd megérintjük a törölt node apját. Alternatíva: megérintjük x-et, és joinoljuk a kapott két részfát

Miért pont a Splay Tree?

Miért pont a Splay Tree?

A legutóbbi Splay Tree-s talkon láttunk egy mondást:

Balance Theorem

Egy n-csúcsú Splay Tree-n m művelet összköltsége $O(m \log n + n \log n)$.

Miért pont a Splay Tree?

A legutóbbi Splay Tree-s talkon láttunk egy mondást:

Balance Theorem

Egy n-csúcsú Splay Tree-n m művelet összköltsége $O(m \log n + n \log n)$.

Ennél több minden is ismert a Splay Tree-kre:

Miért pont a Splay Tree?

A legutóbbi Splay Tree-s talkon láttunk egy mondást:

Balance Theorem

Egy n-csúcsú Splay Tree-n m művelet összköltsége $O(m \log n + n \log n)$.

Ennél több minden is ismert a Splay Tree-kre:

Static Optimality Theorem

Jelölje q_x azt a számot, ahányszor az x elemet lekérdezzük. Ha minden elemet legalább egyszer lekérünk, akkor a teljes műveletsor összköltsége

$$O\left(m + \sum_x q_x \log \frac{m}{q_x}\right).$$

Miért pont a Splay Tree?

A legutóbbi Splay Tree-s talkon láttunk egy mondást:

Balance Theorem

Egy n-csúcsú Splay Tree-n m művelet összköltsége $O(m \log n + n \log n)$.

Ennél több minden is ismert a Splay Tree-kre:

Static Optimality Theorem

Jelölje q_x azt a számot, ahányszor az x elemet lekérdezzük. Ha minden elemet legalább egyszer lekérünk, akkor a teljes műveletsor összköltsége

$$O\left(m + \sum_x q_x \log \frac{m}{q_x}\right).$$

(gyakrabban lekérdezett elemek átlag elérési ideje kisebb)

Miért pont a Splay Tree?

A legutóbbi Splay Tree-s talkon láttunk egy mondást:

Balance Theorem

Egy n-csúcsú Splay Tree-n m művelet összköltsége $O(m \log n + n \log n)$.

Ennél több minden is ismert a Splay Tree-kre:

Static Optimality Theorem

Jelölje q_x azt a számot, ahányszor az x elemet lekérdezzük. Ha minden elemet legalább egyszer lekérünk, akkor a teljes műveletsor összköltsége

$$O\left(m + \sum_x q_x \log \frac{m}{q_x}\right).$$

(gyakrabban lekérdezett elemek átlag elérési ideje kisebb)
Az pl. ismert, hogy ennél jobbat statikus keresőfával nem lehet elérni.

Miért pont a Splay Tree?

Miért pont a Splay Tree?

Static Finger Theorem

Legyenek az elemek a fában az $1, 2, \ldots, n$ és legyen f az egyik rögzített közülük. Akkor a sorozat összköltsége

$$O\left(m + n \log n + \sum_x \log(|x - f| + 1).\right)$$

Miért pont a Splay Tree?

Static Finger Theorem

Legyenek az elemek a fában az $1, 2, \ldots, n$ és legyen f az egyik rögzített közülük. Akkor a sorozat összköltsége

$$O\left(m + n \log n + \sum_x \log(|x - f| + 1).\right)$$

Finger Search Tree: egy elemre nyilvántartunk egy pointert (a gyökéren kívül) és onnan indítjuk a keresést

Miért pont a Splay Tree?

Static Finger Theorem

Legyenek az elemek a fában az $1, 2, \ldots, n$ és legyen f az egyik rögzített közülük. Akkor a sorozat összköltsége

$$O\left(m + n \log n + \sum_x \log(|x - f| + 1).\right)$$

Finger Search Tree: egy elemre nyilvántartunk egy pointert (a gyökéren kívül) és onnan indítjuk a keresést

Az ismert, hogy ennél jobbat statikus Finger Search Tree nem tud.

Miért pont a Splay Tree?

Static Finger Theorem

Legyenek az elemek a fában az $1, 2, \ldots, n$ és legyen f az egyik rögzített közülük. Akkor a sorozat összköltsége

$$O\left(m + n \log n + \sum_x \log(|x - f| + 1).\right)$$

Finger Search Tree: egy elemre nyilvántartunk egy pointert (a gyökéren kívül) és onnan indítjuk a keresést

Az ismert, hogy ennél jobbat statikus Finger Search Tree nem tud.

Dynamic Finger Theorem: ha a fingert mozgatja a Finger Search Tree mindig az utoljára lekérdezett node-ra, akkor se lesz jobb, mint a Splay Tree.

Miért pont a Splay Tree?

Miért pont a Splay Tree?

Dynamic Optimality Conjecture

Legyen egy m lekérdezésből álló sorozatunk egy n-csúcsú keresőfán és

Miért pont a Splay Tree?

Dynamic Optimality Conjecture

Legyen egy m lekérdezésből álló sorozatunk egy n-csúcsú keresőfán és legyen A egy tetszőleges algoritmus, amiben

Miért pont a Splay Tree?

Dynamic Optimality Conjecture

Legyen egy m lekérdezésből álló sorozatunk egy n-csúcsú keresőfán és legyen A egy tetszőleges algoritmus, amiben

- minden lekérdezés költsége a lekérdezett csúcs aktuális mélysége plusz egy,

Miért pont a Splay Tree?

Dynamic Optimality Conjecture

Legyen egy m lekérdezésből álló sorozatunk egy n-csúcsú keresőfán és legyen A egy tetszőleges algoritmus, amiben

- minden lekérdezés költsége a lekérdezett csúcs aktuális mélysége plusz egy,
- és a lekérdezések közt A tetszőleges sok forgatást végezhet, forgatásonként 1 költséggel.

Miért pont a Splay Tree?

Dynamic Optimality Conjecture

Legyen egy m lekérdezésből álló sorozatunk egy n-csúcsú keresőfán és legyen A egy tetszőleges algoritmus, amiben

- minden lekérdezés költsége a lekérdezett csúcs aktuális mélysége plusz egy,
- és a lekérdezések közt A tetszőleges sok forgatást végezhet, forgatásonként 1 költséggel.

Akkor ha ugyanezt Splay fával végezzük el, az legfeljebb konstansszor lehet rosszabb plusz $O(n)$, mint A költsége.

Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

- berakjuk az ábécénk betűit egy (tetszőleges, de előre rögzített) keresőfa leveleibe

Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

- berakjuk az ábécénk betűit egy (tetszőleges, de előre rögzített) keresőfa leveleibe
- ha egy *a* betűt elkódolunk:

Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

- berakjuk az ábécénk betűit egy (tetszőleges, de előre rögzített) keresőfa leveleibe
- ha egy *a* betűt elkódolunk:
  - lemegyünk az *a* betűt tároló csúcshoz, közben kiírjuk az útvonalat odafele (0: balra lépek, 1: jobbra lépek)

Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

- berakjuk az ábécénk betűit egy (tetszőleges, de előre rögzített) keresőfa leveleibe
- ha egy *a* betűt elkódolunk:
  - lemegyünk az *a* betűt tároló csúcshoz, közben kiírjuk az útvonalat odafele (0: balra lépek, 1: jobbra lépek)
  - splay *a* parentjét

Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

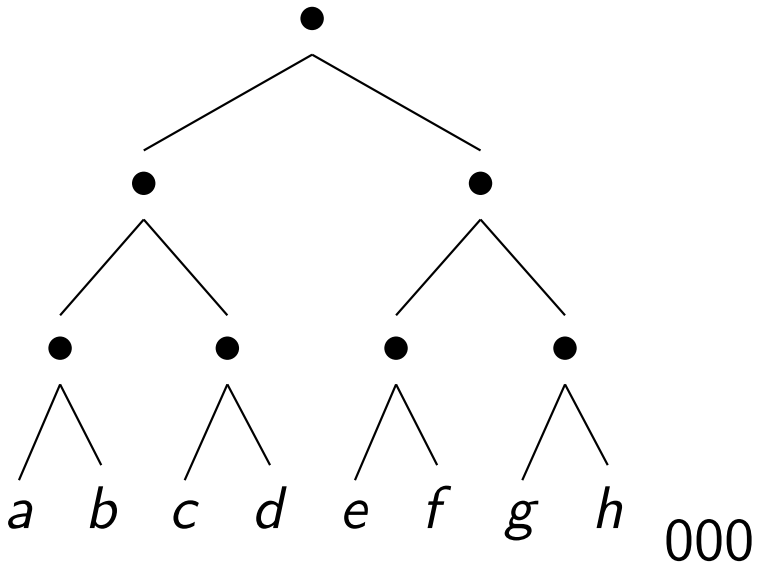
- berakjuk az ábécénk betűit egy (tetszőleges, de előre rögzített) keresőfa leveleibe
- ha egy *a* betűt elkódolunk:
  - lemegyünk az *a* betűt tároló csúcshoz, közben kiírjuk az útvonalat odafele (0: balra lépek, 1: jobbra lépek)
  - splay *a* parentjét
- (azért kell, hogy csak levelekben legyenek a betűk végig, hogy prefixmentes kódolásunk legyen)

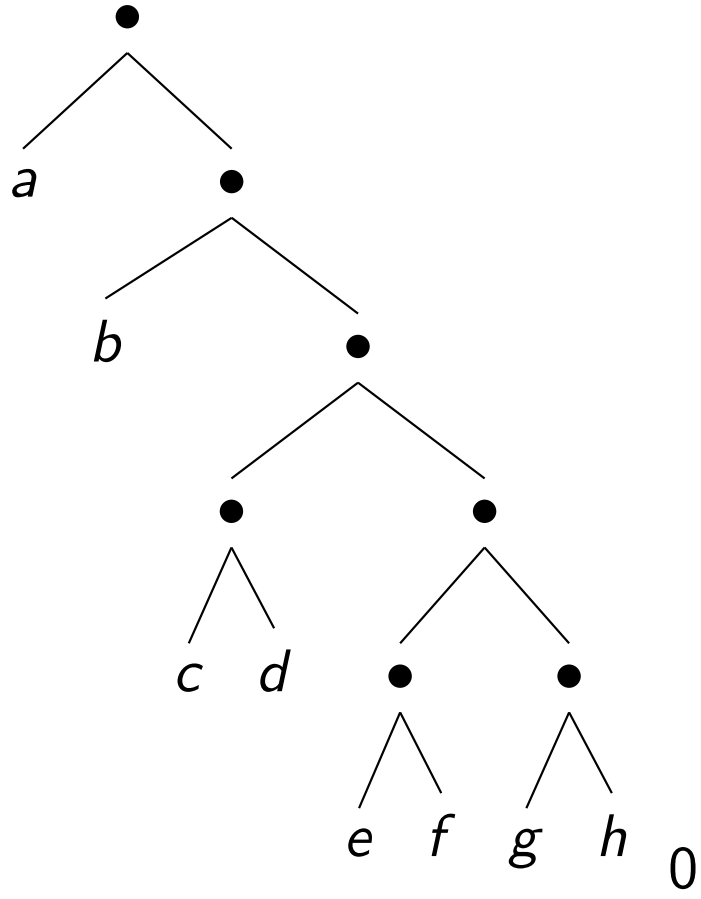
Mesélj még a Splay Treeről

A Splay fákat akár egy Huffman-like kódolásra is lehet használni:

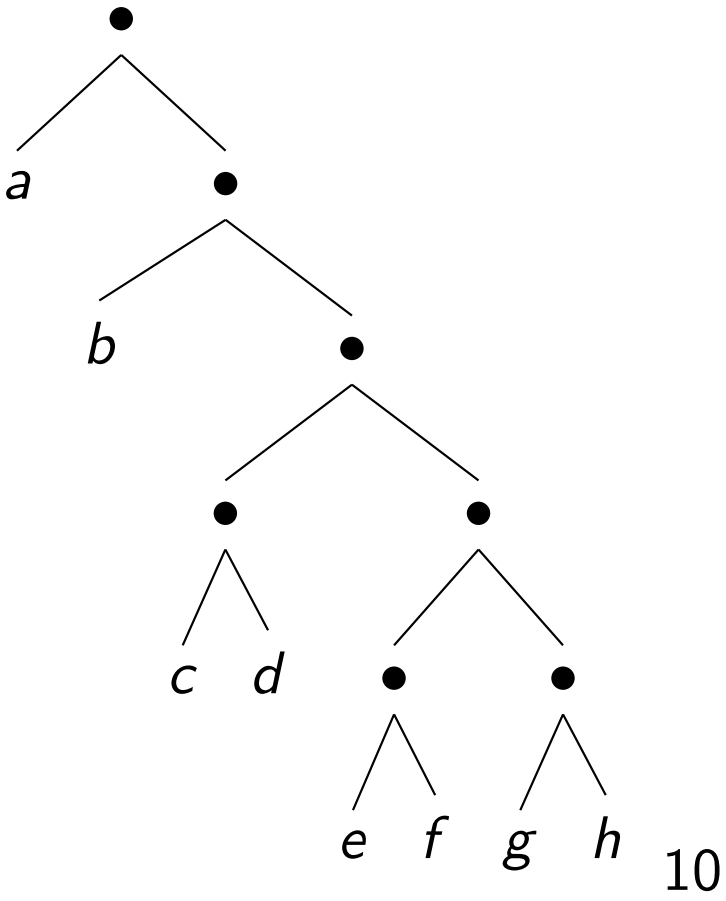
- berakjuk az ábécénk betűit egy (tetszőleges, de előre rögzített) keresőfa leveleibe
- ha egy *a* betűt elkódolunk:
  - lemegyünk az *a* betűt tároló csúcshoz, közben kiírjuk az útvonalat odafele (0: balra lépek, 1: jobbra lépek)
  - splay *a* parentjét
- (azért kell, hogy csak levelekben legyenek a betűk végig, hogy prefixmentes kódolásunk legyen)
- Dekódolás: ugyanígy

Példa: *aabg* . . .

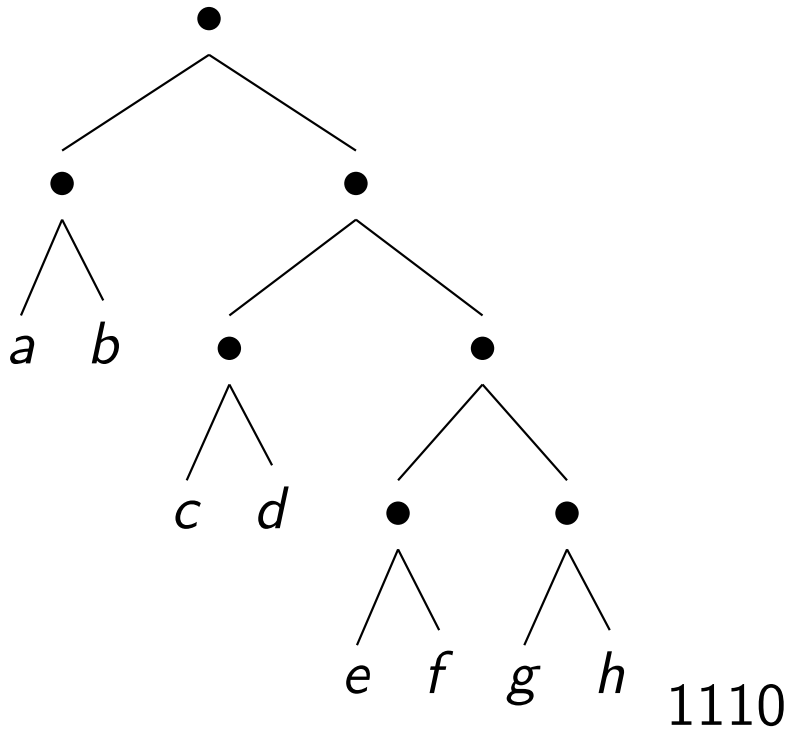




Példa: *aabg* . . .



10



1110

Mennyire jó ez?

A Static Optimality Theorem szerint kódolás hossza, ha m karakter van és az a karakter q_a-szor fordul elő:

$$O\left(m + \sum_a q_a \frac{m}{q_a}\right)$$

és van olyan ismert tétel, miszerint a sorozat entrópiája,

$$\sum_a q_a \frac{m}{q_a}$$

egy alsó korlát. (A Huffman kódolásé is legalább ennyi. De ahhoz tudnunk kell előre a frekvenciákat!)
Használják még: cache (gyakran elért elemek felül lesznek, locality of reference), routerek packet classification algoritmusaiban stbn.