Dinamikus gráfok

Iván Szabolcs

2017 tavasz

Recap: dinamikus gráfok

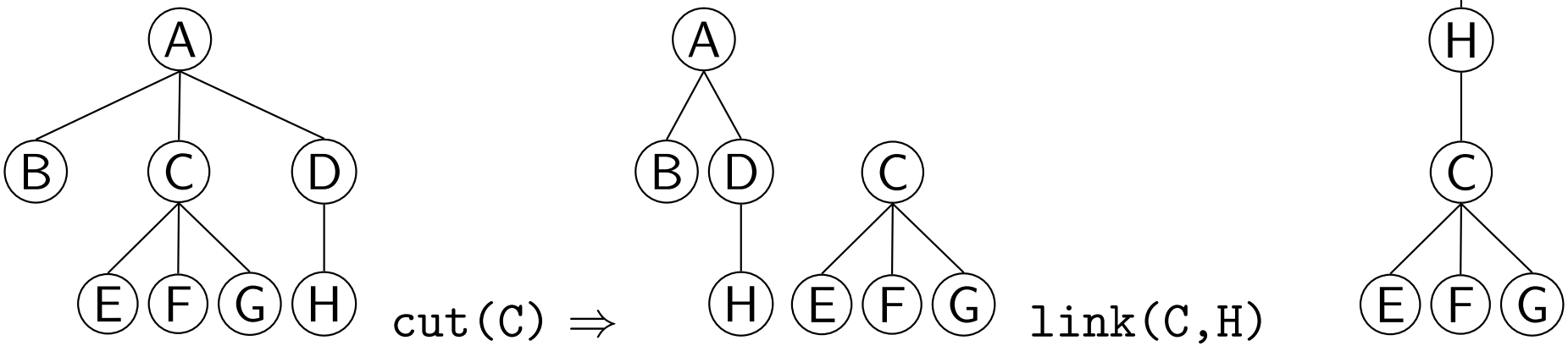
- Van valami gráf adatszerkezetünk (irányított vagy nem, fa vagy nem, etc)
- Jönnek be updateek: a gráf változik (kicsit)
- Jönnek be queryk: valami lekérdezés a gráfban (összefüggő-e, két csúcs egy komponensben van-e, etc.)

A mai task

- A gráf egy gyökér felé irányított erdő
- Updatek:
  - `maketree`: plusz egy új csúcs létrehozása
  - `link(v,w)`: egy aktuális gyökér bekötése egy másik fa egy csúcsa alá
  - `cut(v)`: a `v`-ből a szülőjébe menő él törlése
  - `find(v)`: a `v`-t tartalmazó fa gyökere
  - `sum(v)`: a `v`-ből a fájának gyökeréig menő úton (mondjuk) a csúcsok egy adattagjának az összege (vagy minimuma, maximuma etc is lehetne)

A naiv megoldás

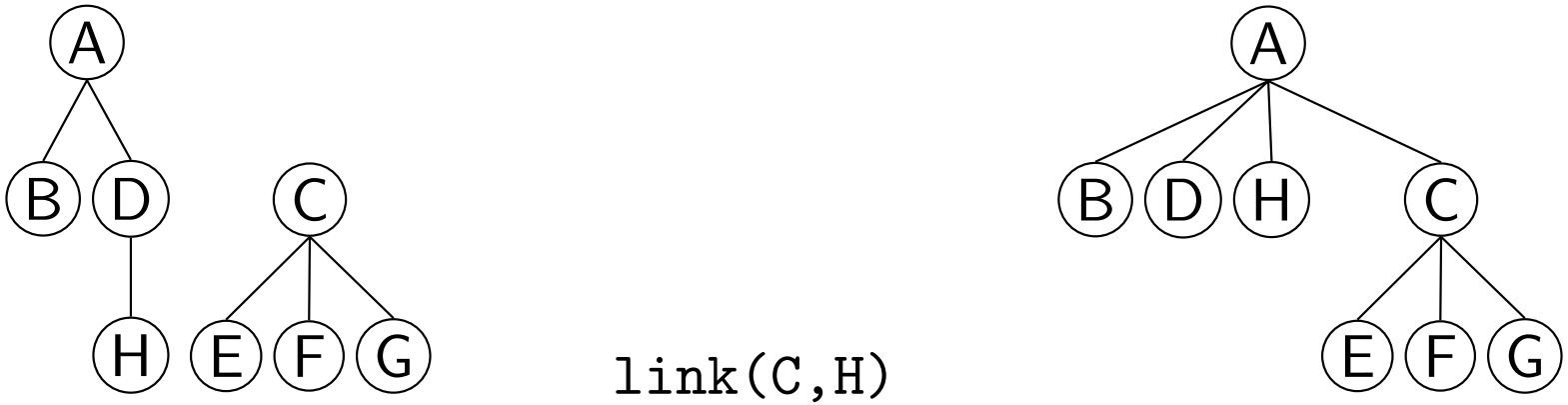
- Az erdőt ténylegesen csak mint pointereket tároljuk
- Update cost: $O(1)$
- Query cost: $O(n)$



- A find lehet így $O(n)$ drága.

Ha nem lenne cut...

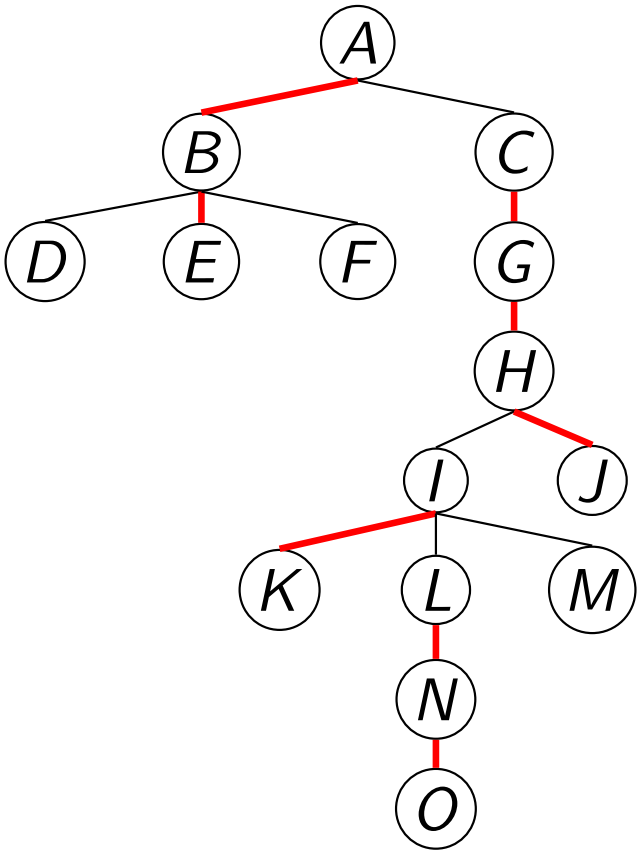
- A Union-Find forest (amortizált) majdnem konstans időben támogatná a `maketree`, `link(v,w)`, `find(v)` lekérdezéseket
- De elrontja a fát! `sum(v)` nem megy
- `cut(v)` sem



- `cut(H)` egész mást ad, mint az előbb adott volna

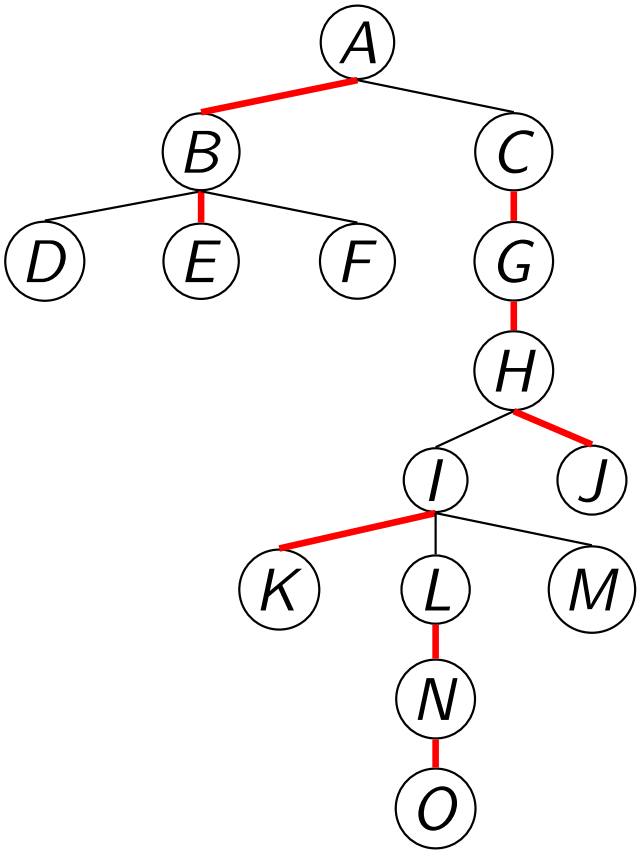
Preferált gyerekek

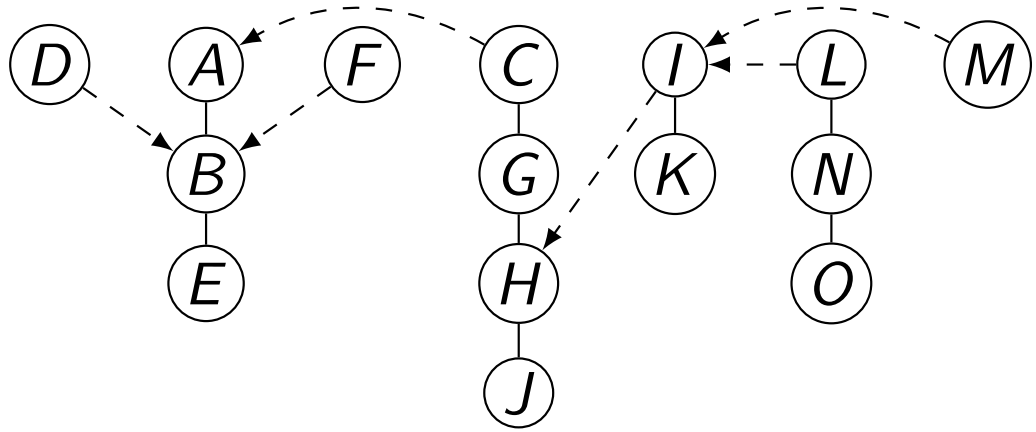
Ötlet: a fában minden csúcsnak nyilvántartunk (legfeljebb) egy preferált gyereket is.



Preferált utak

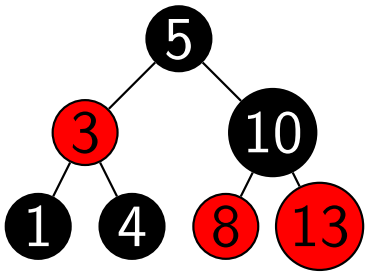
- Mindegyik fa a preferált élekből álló preferált utak diszjunkt uniója
- Terv: egy preferált útból gyorsan tudjunk fellépni az út fölé

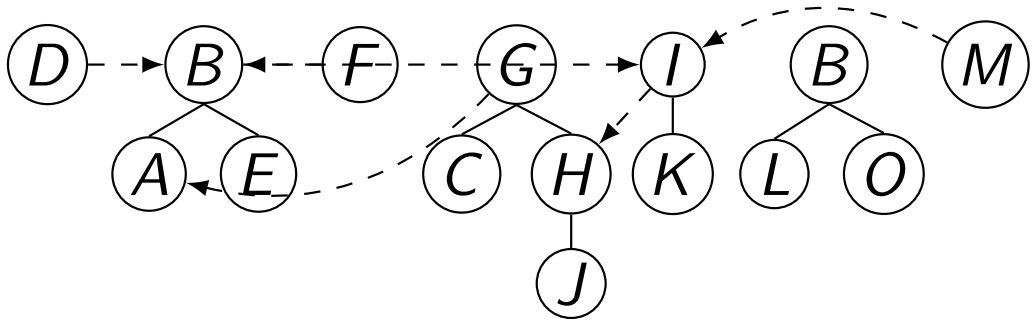




Preferált utak

- A preferált utakat valamilyen halmazként tároljuk
- Szóba jöhet pl: egy keresőfa

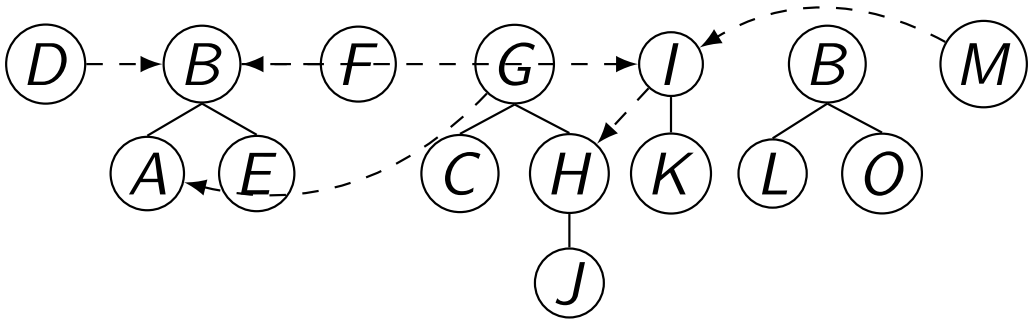




- De ne veszítsük el az út sorrendjéről az infót!

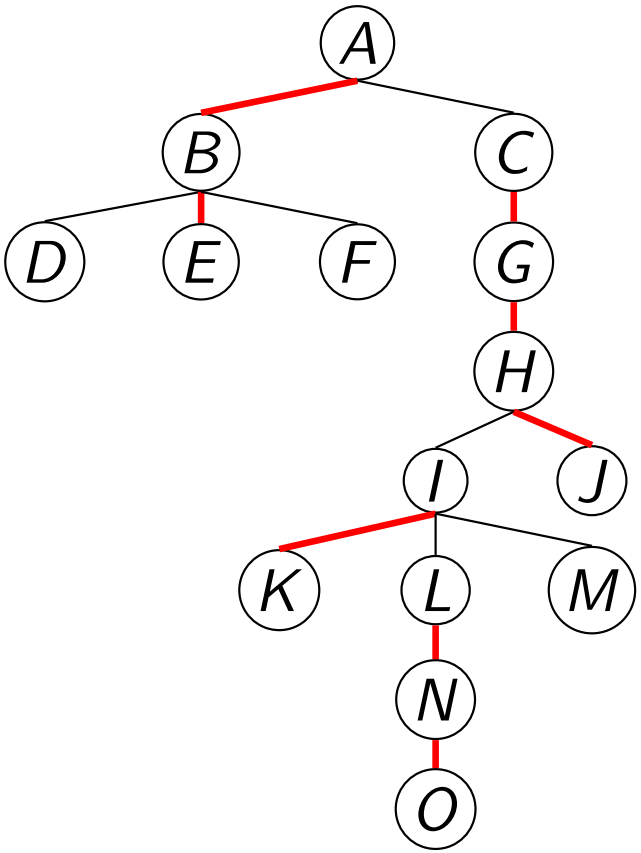
Rendezés?

- Bármilyen rendezés megfelelne a preferált utak keresőfáiban
- Legyen $u < v$, ha u van feljebb az úton
- Ezt nyilvántartani persze nehezebb lesz, de a `cut`nál majd megéri

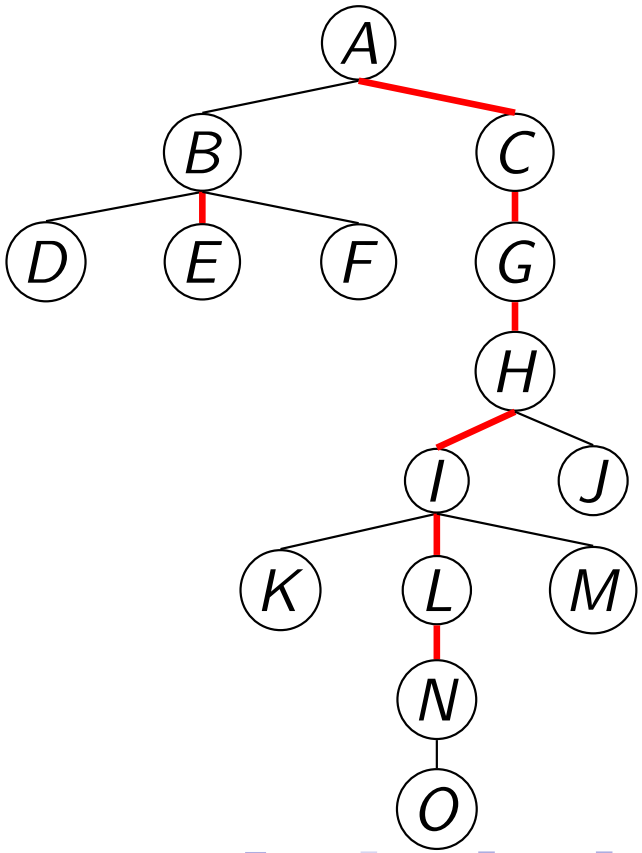


Preferred?

- `access(v)`: ,,érintsük meg" `v`-t
- hatása: `v`-ből a gyökérig terjedő úton legyenek az új preferred gyerekek
- és: `v`-nek ne legyen preferred gyereke (why? see later)

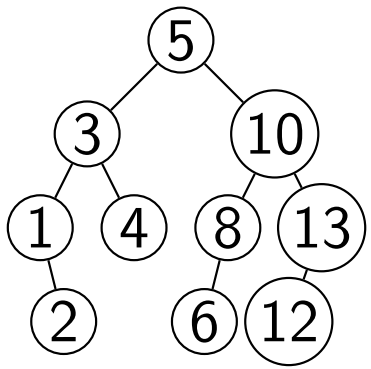


`access(N)`



find(v)

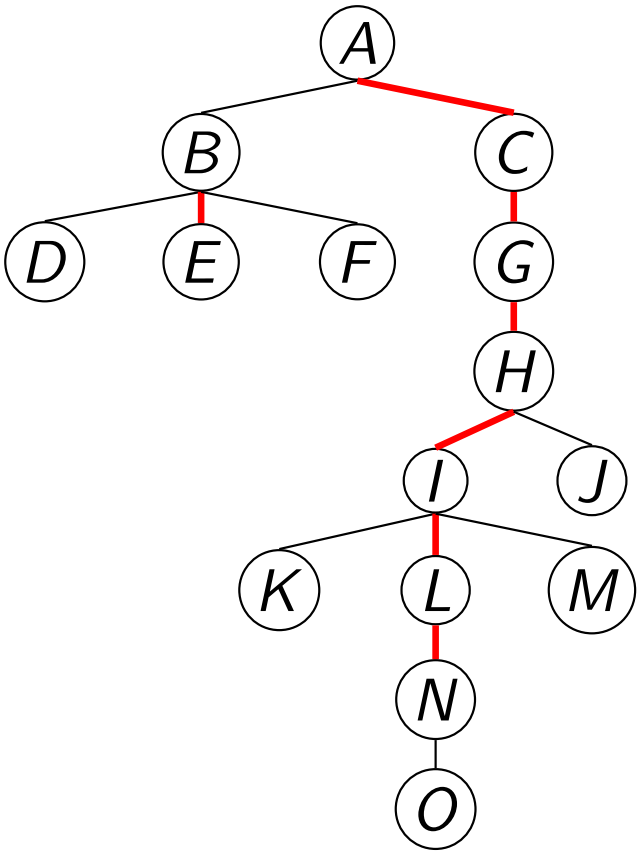
- Érintsük meg v-t
- Most v rajta van a gyökérig menő preferred pathon
- Ennek a legkisebb eleme (recall: rendezés mélység szerint van) lesz a gyökér

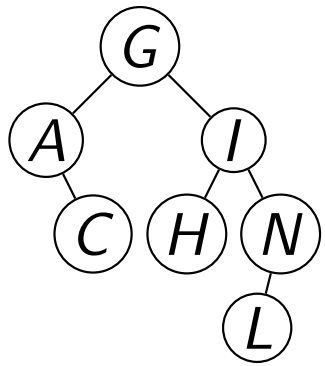


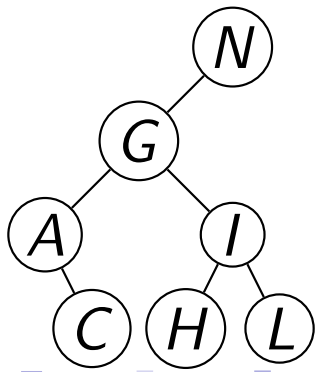
- Az időigényen még lesz mit javítani

cut(v)

- Érintsük meg v-t
- Most v a legnagyobb eleme a preferred fájának
- Ha v lenne a gyökér, könnyű lenne: csak ki kéne nullázni a bal fiát a preferred fában

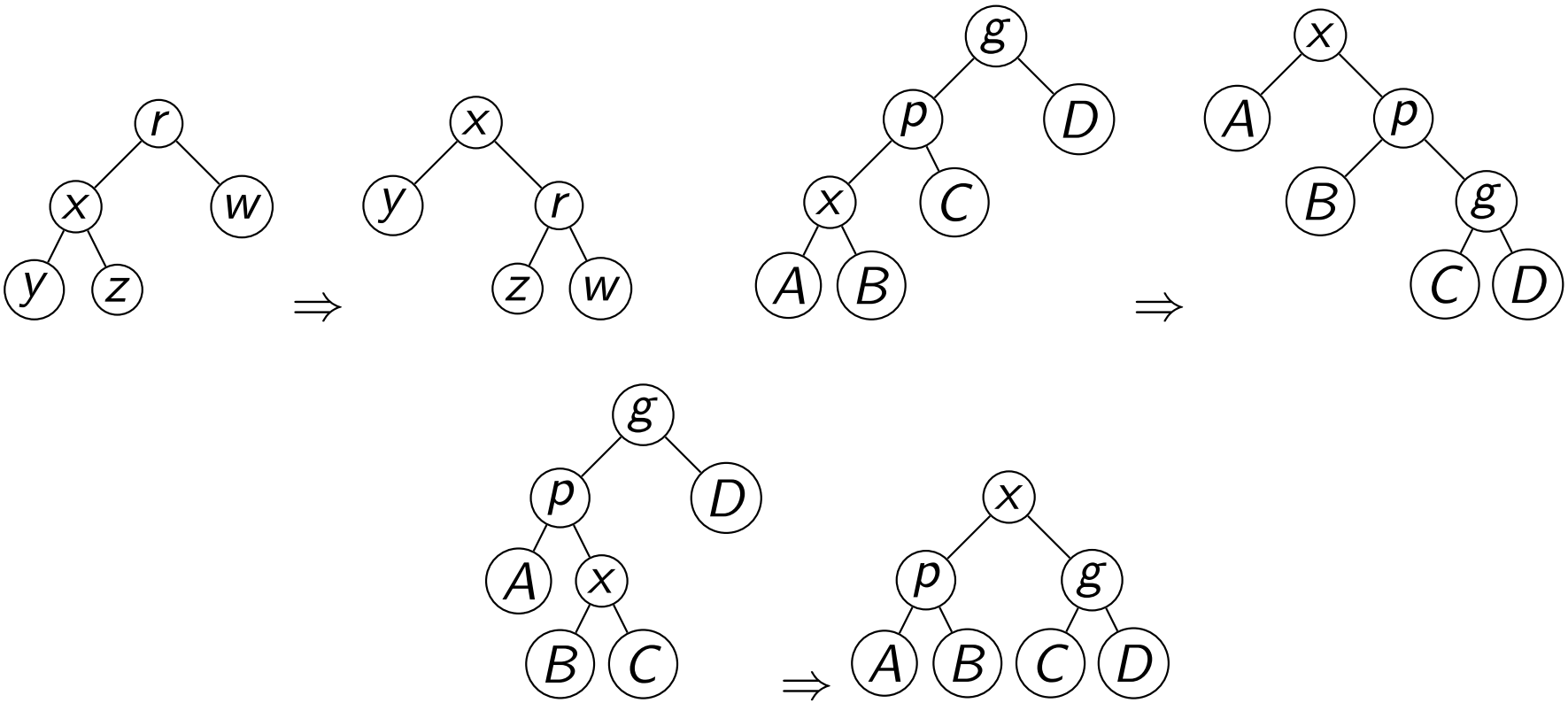






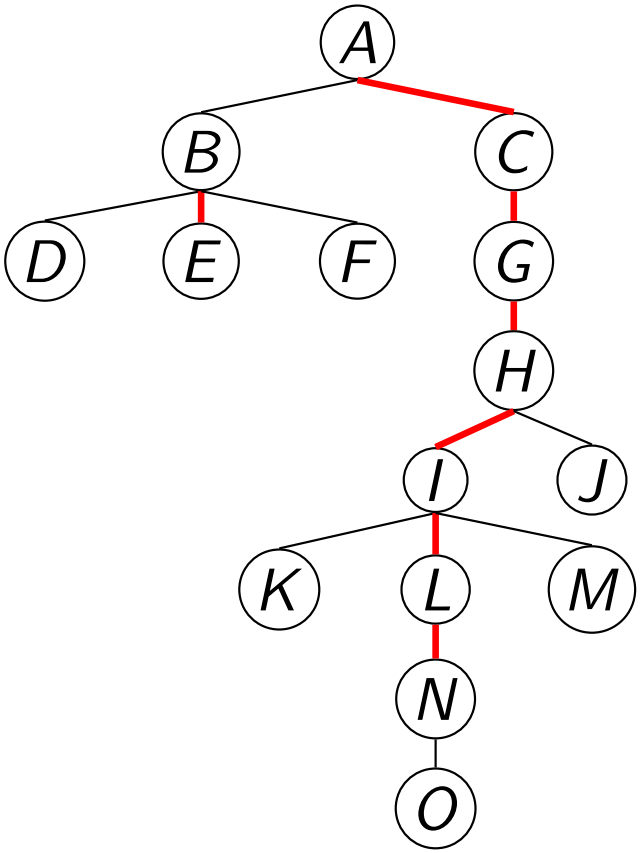
Splay tree

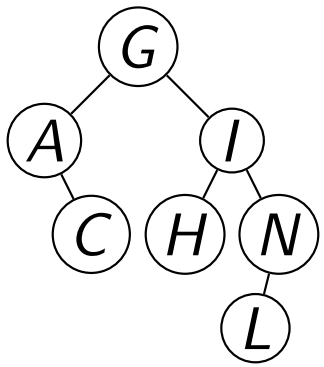
- Bináris keresőfa
- `find(x)`nál `x`-et felforgatja gyökérbe
- Egy kis módosítással ,,zig-zig step”

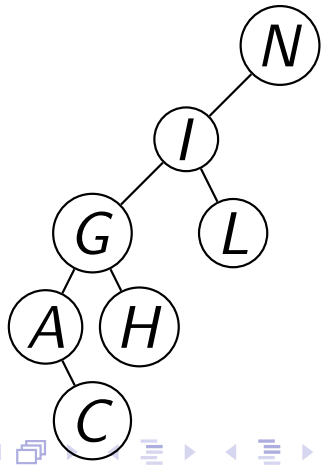


cut(v)

- Érintsük meg v-t
- A preferred pathon v-nél lejjebb levő csúcsok a v-nél nagyobb csúcsok a rendezésben
- Ha v lenne a gyökér, könnyű lenne
- Splay treet használunk, könnyű lesz

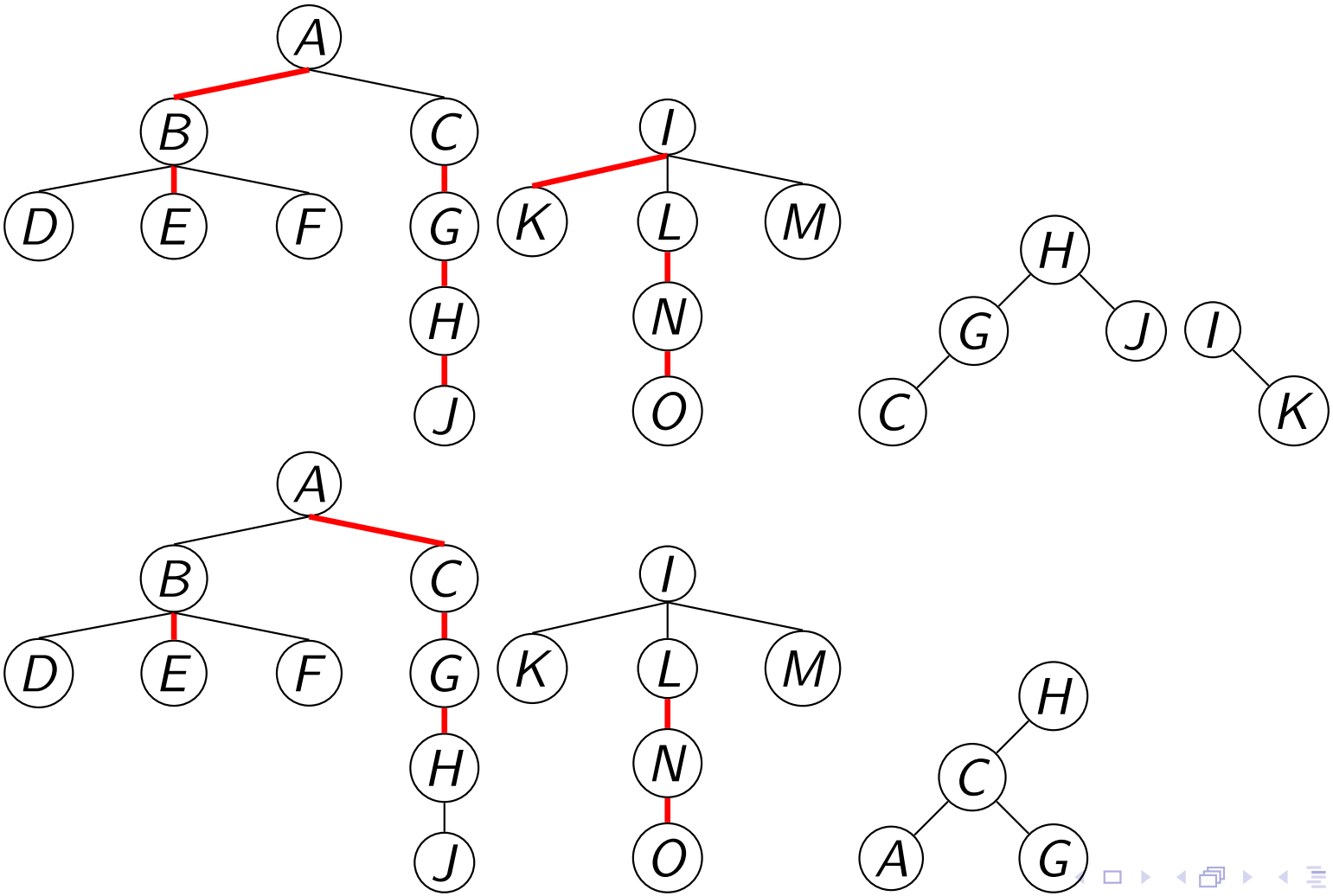




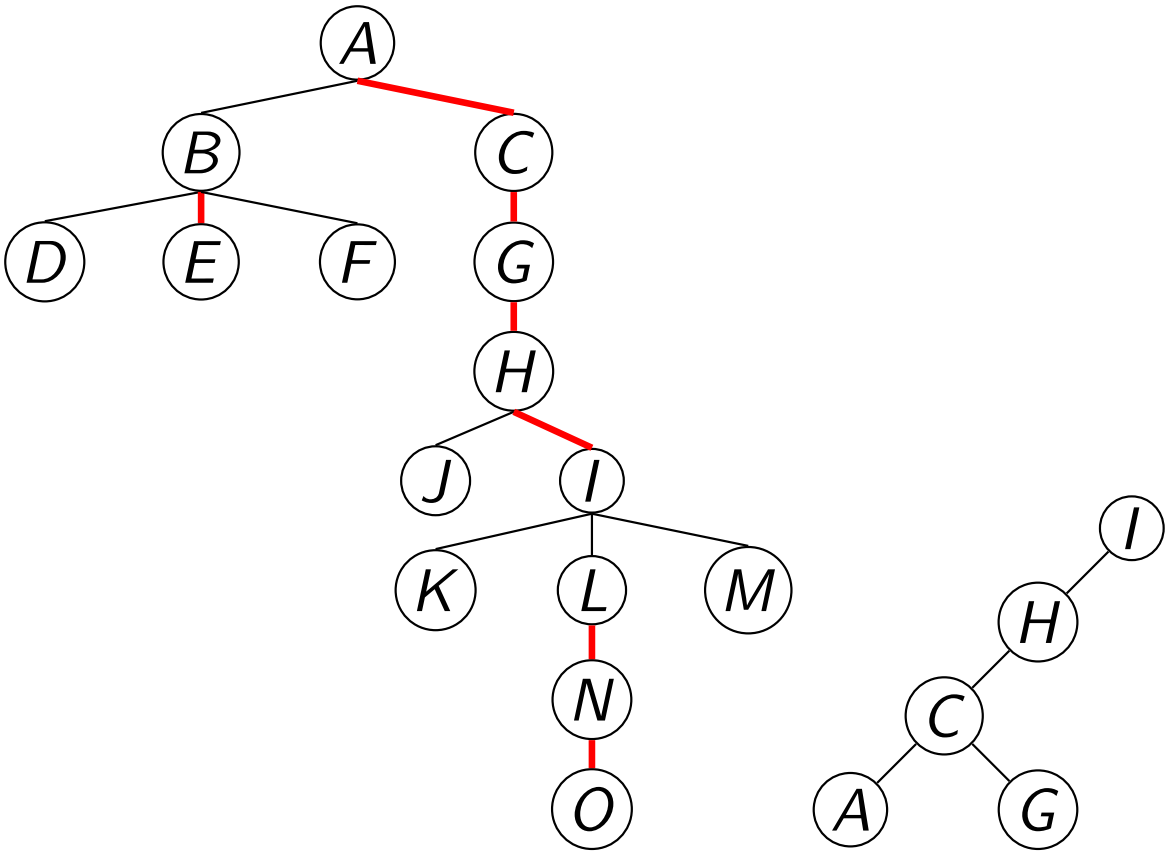


link(v,w)

- Érintsük meg v-t és w-t
- Mivel v gyökér, ő most egyedül lesz a splay fájában
- v-nek legyen w a bal fia, done. link(I,H):



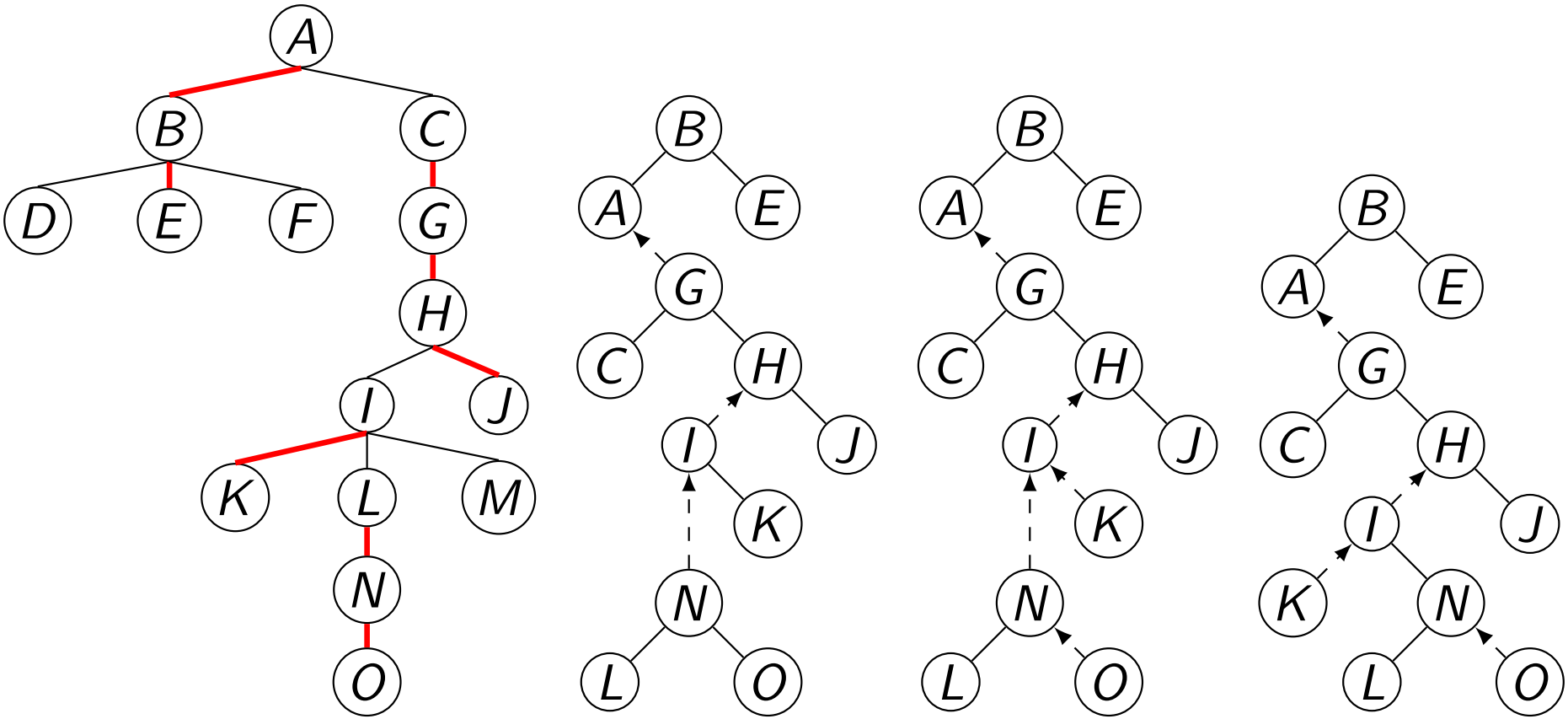
link(v,w)



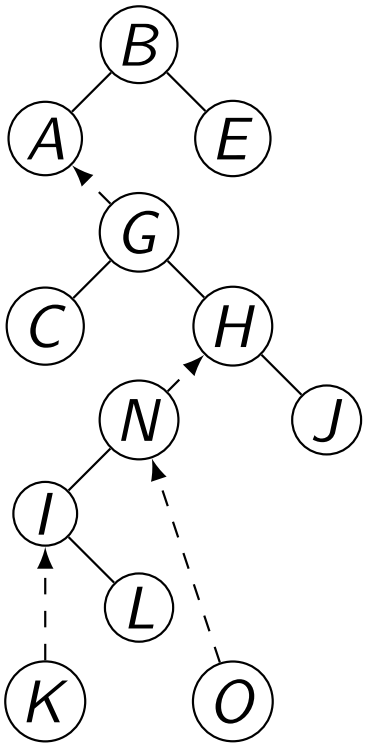
access(v)

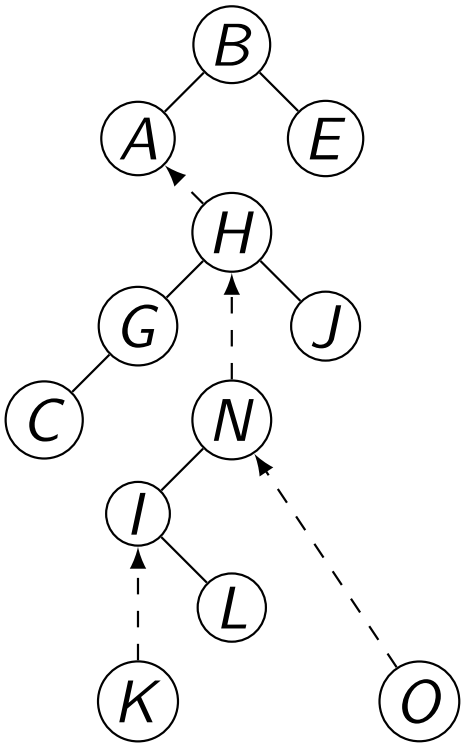
- Splay findoljuk v-t $\Rightarrow$ ő lesz a preferred path gyökere
- A jobb fiát elvágjuk, updateljük a gyerek `path parent` mezejét v-re
- (ezzel elvágtuk v alatt a preferred pathot)
- megyünk fel a fában: `w:=v.pathparent`
- Splay findoljuk w-t
- w jobb gyerekét levágjuk, mint az előbb
- w új jobb gyereke v legyen
- forgassuk fel v-t

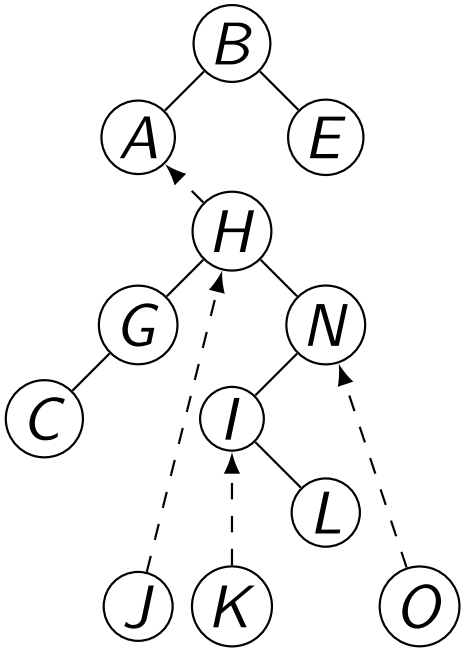
access(N)


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
B
A
E
G
C
H
I
J
K
N
L
O
B
A
E
G
C
H
I
J
K
N
L
O
B
A
E
G
C
H
I
J
K
N
L
O

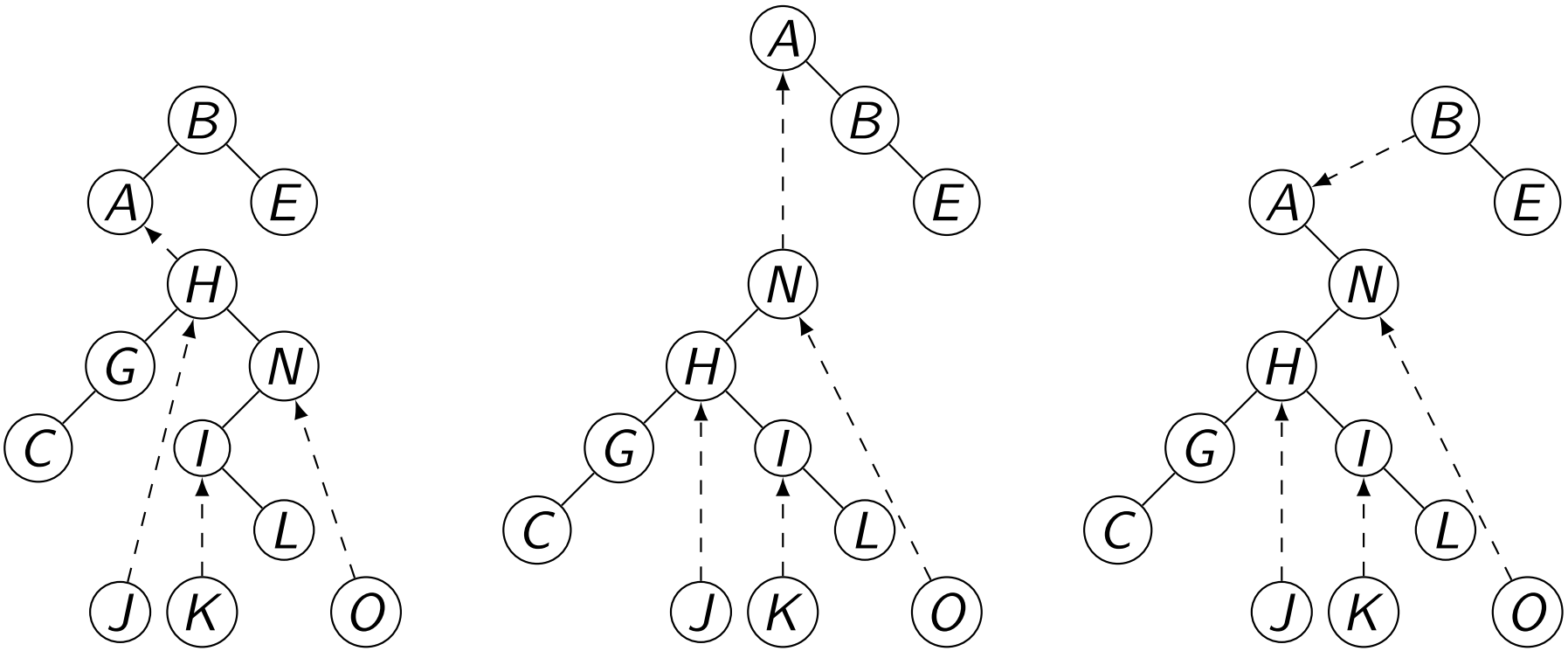
access(N)



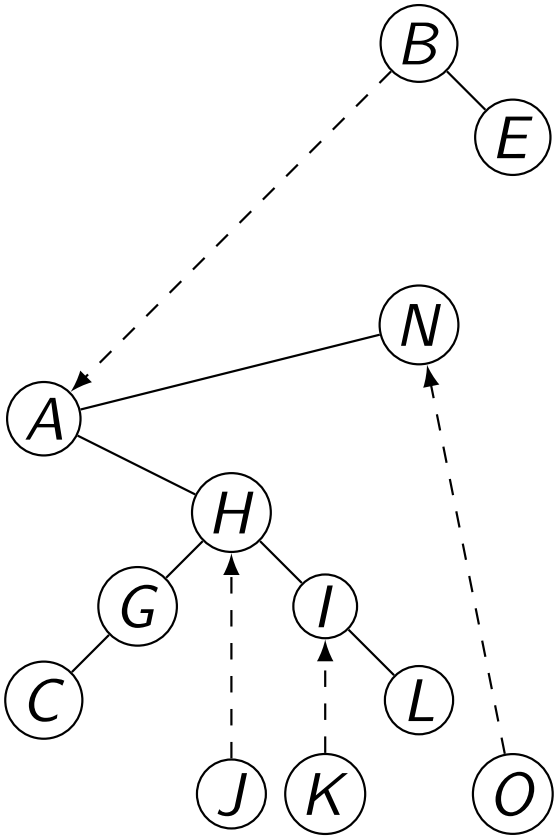


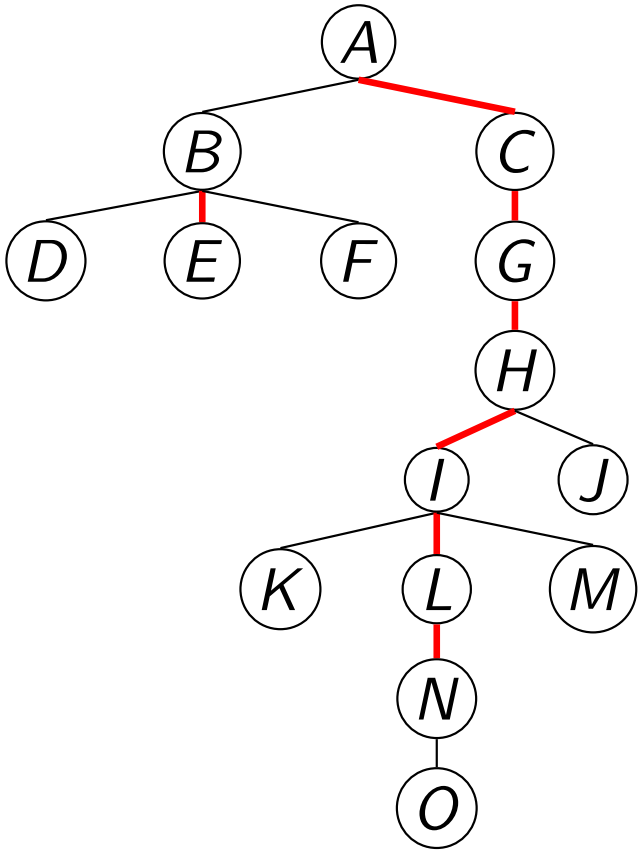


access(N)



access(N)



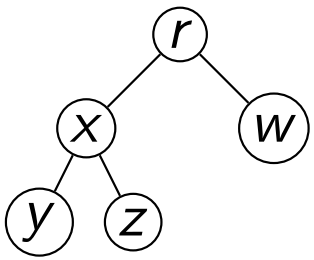


Amortizált időigény

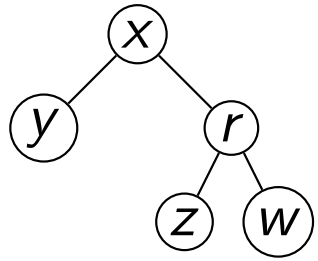
Most kiszámoljuk egy `splay` művelet amortizált költségét egy n-csúcsú splay fában.

- Vezessük be a következőket:
- `size`(x): az x gyökerű részfában a csúcsok száma
- `rank`(x): ennek a kettes alapú logaritmusa
- Φ: az összes csúcs rangjának összege

Hogy változik a potenciál egy-egy cikk, cikk-cikk és cikk-cakk forgatásnál?



$\Rightarrow$

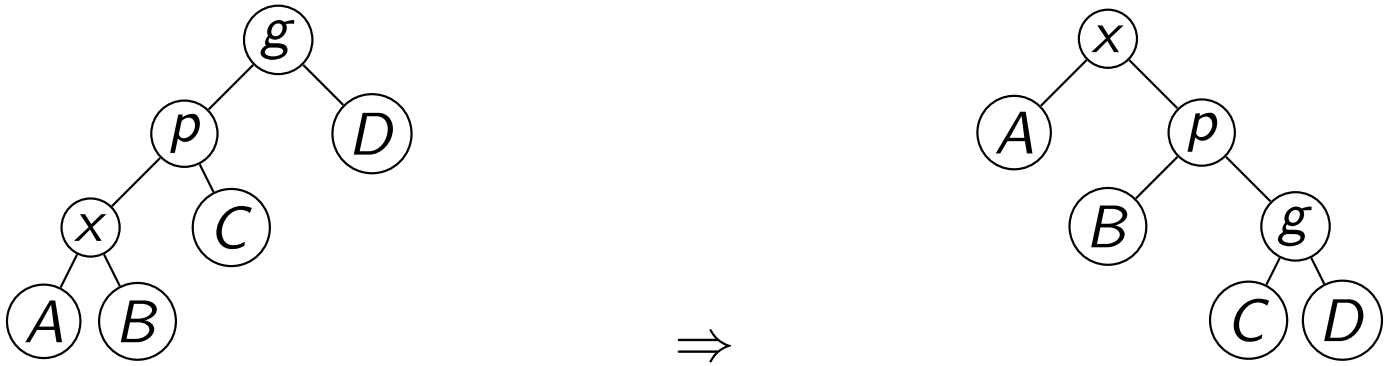


`rank'(r)-rank(r)+rank'(x)-rank(x)`

$=$`rank'(r)-rank(x)`

$\leq$`rank'(x)-rank(x)`

Amortizált időigény



$$
\begin{aligned}
&\mathrm{rank}'(g) - \mathrm{rank}(g) + \mathrm{rank}'(p) - \mathrm{rank}(p) + \mathrm{rank}'(x) - \mathrm{rank}(x) \\
&= \mathrm{rank}'(g) + \mathrm{rank}'(p) - \mathrm{rank}(p) - \mathrm{rank}(x) \\
&\leq \mathrm{rank}'(g) + \mathrm{rank}'(x) - \mathrm{rank}(x) - \mathrm{rank}(x) \\
&= \mathrm{rank}'(g) + \mathrm{rank}'(x) - 2\mathrm{rank}(x) \\
&\leq 3(\mathrm{rank}'(x) - \mathrm{rank}(x))
\end{aligned}
$$

a másik eset is hasonló

Ebből: egy n-csúcsú splay fán m művelet költsége $O(m \log n + n \log n)$. (amortizált $\log n$)

Teljes időigény

- Az `access` műveletek során amortizált $\log n$-szer haladunk szaggatott nyilakon.
- A teljes költség így műveletenként amortizált $\log n$.
- A splay fákat ki lehet bővíteni intervallum-összeg (-minimum, maximum stb) műveletekkel, amiket forgatásnál updatelünk – az aggregált csúcs–gyökér lekérdezéseket is tudjuk amortizált $O(\log n)$-ben támogatni.

`%EOF`