# FUNKCIJSKE PODATKOVNE STRUKTURE

Pozorni bomo na dva koncepta podatkovnih struktur:

- Nespremenljivost struktur
- Lenost izvajanja

# NESPREMENLJIVOST STRUKTUR

Do zdaj smo delali s spremenljivimi podatkovnimi strukturami (minljive / ephemeral / mutable).

V funkcijskem programiranju si zaradi referenčne transparentnosti (vsaka vrednost je enaka svoji definiciji) želimo nespremenljivih (trajnih / persistent / immutable) podatkovnih struktur.

Primer:
```
r = ref 0
f x = x + x
g x = incr r ; !r + x
```

Ali je $f(gx) = gx + gx$? Ne.

Zato v OCamlu nimamo referenčne transparentnosti, če uporabljamo reference, tabele, ...

Opomba: Referenčna transparentnost in reference niso zares v sorodu.

Nespremenljivost dosežemo s podvajanjem podatkovnih struktur, učinkovitost pa z deljenjem nespremenjenih podatkov.

Primer: $0 \to 1 \to 2 \to \bullet$ ++ $3 \to 4 \to 5 \to \bullet$

$= 0 \to 1 \to 2$ 

Preden nadaljujemo, ločimo med:

- Abstraktnimi podatkovnimi strukturami (specifikacija operacij in njihovih lastnosti):

  sklad, vrsta s prednostjo, slovar, ...

- Implementacijo teh abstraktnih struktur:

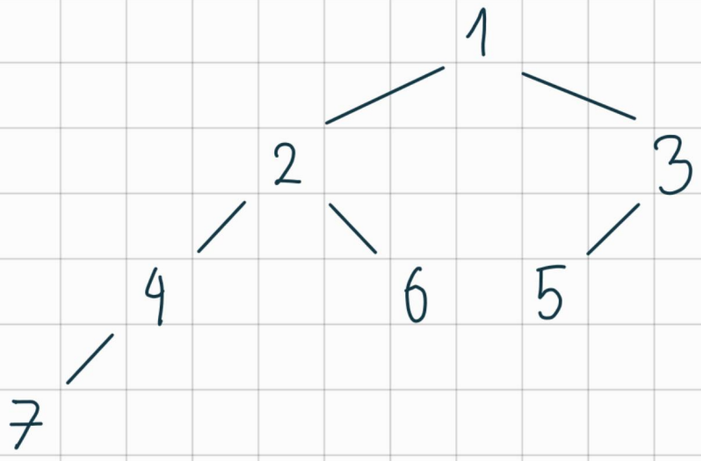
  sklad z verižnim seznamom, sklad s tabelo in kazalcem, vrsta s prednostjo z dvojiškimi kopicami, slovar z zgoščevalnimi tabelami, slovar z dvojiškimi iskalnimi drevesi, ...

# LEVIČARSKE KOPICE

Lastnosti:

- Dvojiška drevesa
- Lastnost min-kopice (koren manjši od vseh potomcev)
- Nagnjenost v levo (rang (dolžina desnega roba - pot od korena, ki gre samo desno) levega otroka je večji od ranga desnega)
- Rekurzivno za otroke

Primer:




Trditev: rang drevesa $\leq \lfloor \log_2(n+1) \rfloor$

Dokaz: DN

Ključna operacija na levičarskih kopicah je zlivanje, saj je:

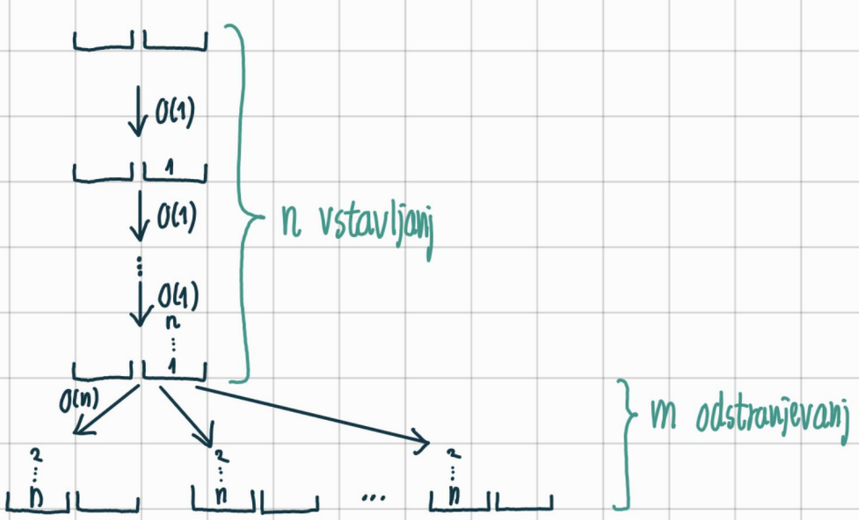
dodaj $(x, h)$ = zlij $(\overset{x}{\cdot}, h)$

odstrani min $(\overset{x}{n_1}\ n_2)$ = zlij $(h_1, h_2)$

# AMORTIZACIJA NESPREMENLJIVIH STRUKTUR

Amortizacija v primeru nespremenljivih podatkovnih struktur vodi v težave, saj lahko drage operacije izvedemo večkrat na istem stanju.

Primer:



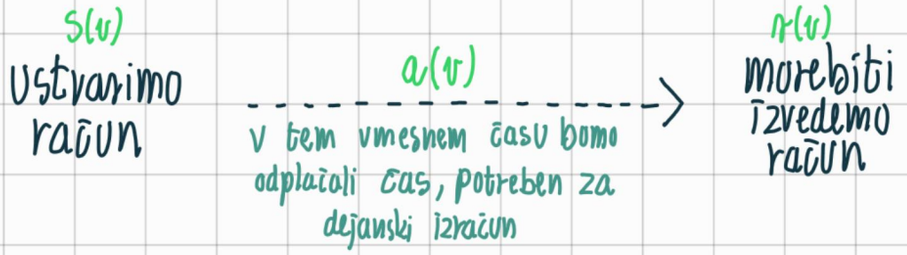

Za $n$ dodajanj in $m$ odstranjevanj bi pričakovali čas $O(n+m)$, dejanski čas pa je $O(n + m \cdot n)$.

Rešitev ponuja leno izvajanje v kombinaciji z ustrezno premišljeno podatkovno strukturo.

Primer: Stvari obrnemo in premaknemo na prvi seznam, ko je drugi daljši.


Kako utemeljimo amortizirano časovno zahtevnost?

Ideja: Pri nevčakanem izvajanju se račun izvede takoj, ko ga poženemo. Pri lenem izvajanju pa sta ti dve točki ločeni v času:




Ko ustvarimo račun, si zabeležimo dolg, ki ga moramo z amortiziranimi operacijami odplačati, preden se izvede.

Formalno definiramo sled izvajanja kot graf, v katerem so vozlišča izvedene operacije, povezave pa odvisnosti med njimi.

$\hat{v}$ je preteklost operacije $v$, torej vsi predhodniki v grafu.

Vsakemu vozlišču $v$ moramo dodeliti:

- Multimnožico potencialnih stroškov $S(v)$ (različni kovanci)

- Multimnožico odplačil $a(v)$
- Multimnožico izvedenih stroškov $a(v)$

Primer: $check(f, r) =$ if $|r| > |f|$
then $(f +\!\!+ rev\, r, [])$
else $(f, r)$

$$x : xs +\!\!+ ys := x : (xs +\!\!+ ys)$$

Če velja:

1) $v \neq v' \Rightarrow s(v) \cap s(v') = \emptyset$

(različne operacije ustvarjajo ločene dolgove)

2) $a(v) \subseteq \bigcup_{w \in \hat{v}} s(w)$

$\hat{v} := \{w ;\ w \text{ prednik } v\}$

(operacija lahko odplačuje le dolgove predhodnikov)

3) $r(v) \subseteq \bigcup_{w \in \hat{v}} a(w)$

(poženemo lahko le izračune brez neodplačanih dolgov)

Tedaj velja:

$$\underbrace{\left|\bigcup_{v \in V} r(v)\right|}_{\text{dejanski strošek skupnih operacij}} \overset{(3)}{\leq} \left|\bigcup_{v \in V} a(v)\right| \leq \sum_{v \in V} |a(v)|$$

Vsaka operacija porabi še $\ell(v) \in \mathbb{N}$ časa za lastne nedeljene izračune.

Dejanski strošek je torej:

$$\sum_{v \in V} \ell(v) + \left| \bigcup_{v \in V} r(v) \right| \leq \sum_{v \in V} (\ell(v) + |a(v)|)$$