Sakupljanje đubreta

60 godina mučenja

Andrej Ivašković

Matematička gimnazija

NEDELJA INFORMATIKE v5.0

17. decembar 2018.


"SINDE

Koji su ovi profesori?

Koji su ovi profesori?

Edsger W. Dijkstra

John McCarthy

LISP!

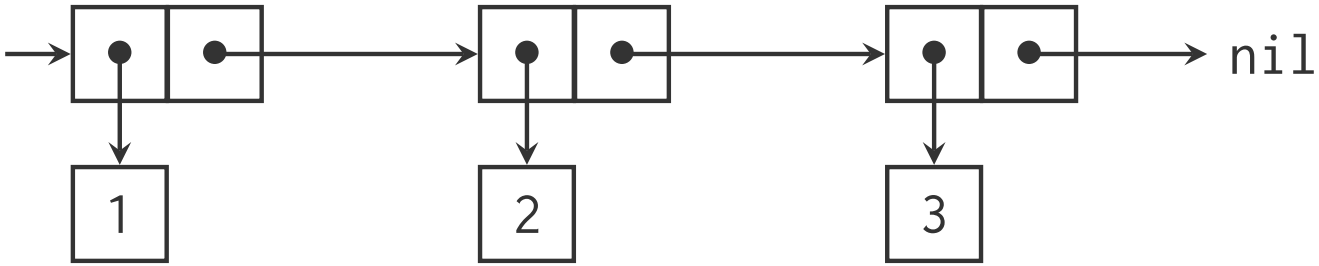
- Prva implementacija **LISP** 1958. godine.
- **LISt Processor**, zasnovan na λ računu i nekim rezultatima teorije izračunljivosti.
- Nekoliko dijalekata i implementacija (Common Lisp, Scheme), i dalje živi preko `emacs` tekstualnog editora.
- Inspiracija za funkcionalne jezike (Haskell, OCaml, F#). Značajan jer je uveo koncept ***heap* memorije**.

Primer LISP koda

- Sintaksa ne zahteva komplikovan parser (takozvani S-izrazi imaju jednostavnu gramatiku).
- Lots of Irritating Silly Parentheses...

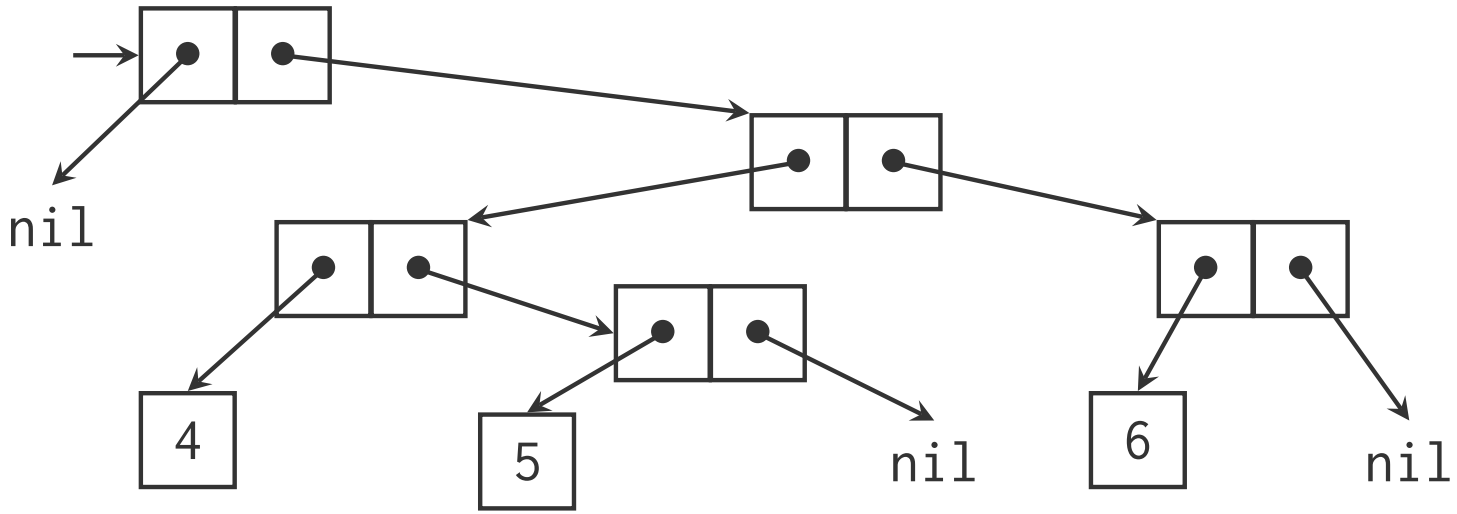
```
> (defun times_two (n) (* 2 n))
TIMES_TWO
> (times_two 7)
14
> (defun fact (x)
  (if (= x 0) 1
              (* x (fact (- x 1)))))
FACT
> (fact 5)
120
```

LISP liste




```
> (list 5 8 1)
(5 8 1)
> (car (list 1 2 3 4))
1
> (cdr (list 1 2 3 4))
(2 3 4)
> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
> (list (list 1 2 3) (list 4 5 6))
((1 2 3) (4 5 6))
```

Stablo?




```
(cons nil (cons (cons 4 (cons 5 nil)) (cons 6 nil)))
```

- Dinamički alocirana ***heap*** memorija – prvi **sakupljač đubreta**.
- Nije teško zamisliti da dobijemo i graf zavisnosti. Neophodna je strategija da se “očiste” podaci.

Pregled ostatka predavanja

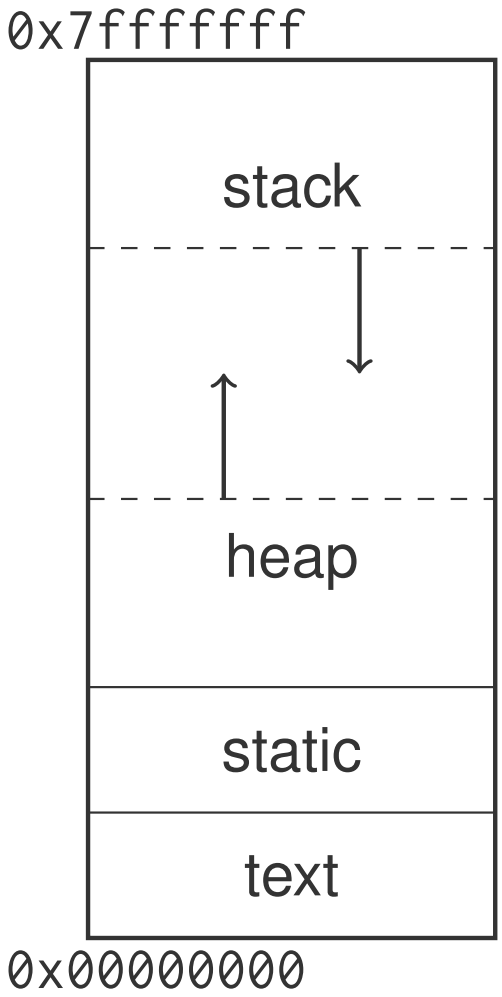
- Osnovne ideje sakupljanja đubreta:
  - Memorijski model programa
  - Vrste sakupljača đubreta
- Implementacija sakupljača đubreta koji rade “praćenje”
- Napredniji dizajn sakupljača đubreta
- Alternative sakupljanju đubreta

Manuelna alokacija memorije

- Ako imamo C strukturu `list_node` sa poljima `val` i `next` i deklarišemo promenljivu sa `list_node x;`, nećemo joj imati pristup nakon izlaza iz trenutnog opsega (*scope*, najčešće funkcija).
- Pri izlasku iz opsega se oslobodi prostor predviđen za ovu strukturu.
- Način da promenljiva preživi opseg je korišćenje pokazivača i dinamičke alokacije memorije, ali to onda zahteva poziv `free`. U suprotnom nastaje *memory leak*.

```
list_node* p = (list_node*) malloc(sizeof(list_node));
// ... nakon mnogo instrukcija
free(p);
```

Adresni prostor C programa




```
list_node p = {v, &q};
```

(pozivi funkcija, lokalne promenljive)

```
list_node* p = (list_node*)
  malloc(sizeof(list_node));
p->val = v; p->next = &q;
```

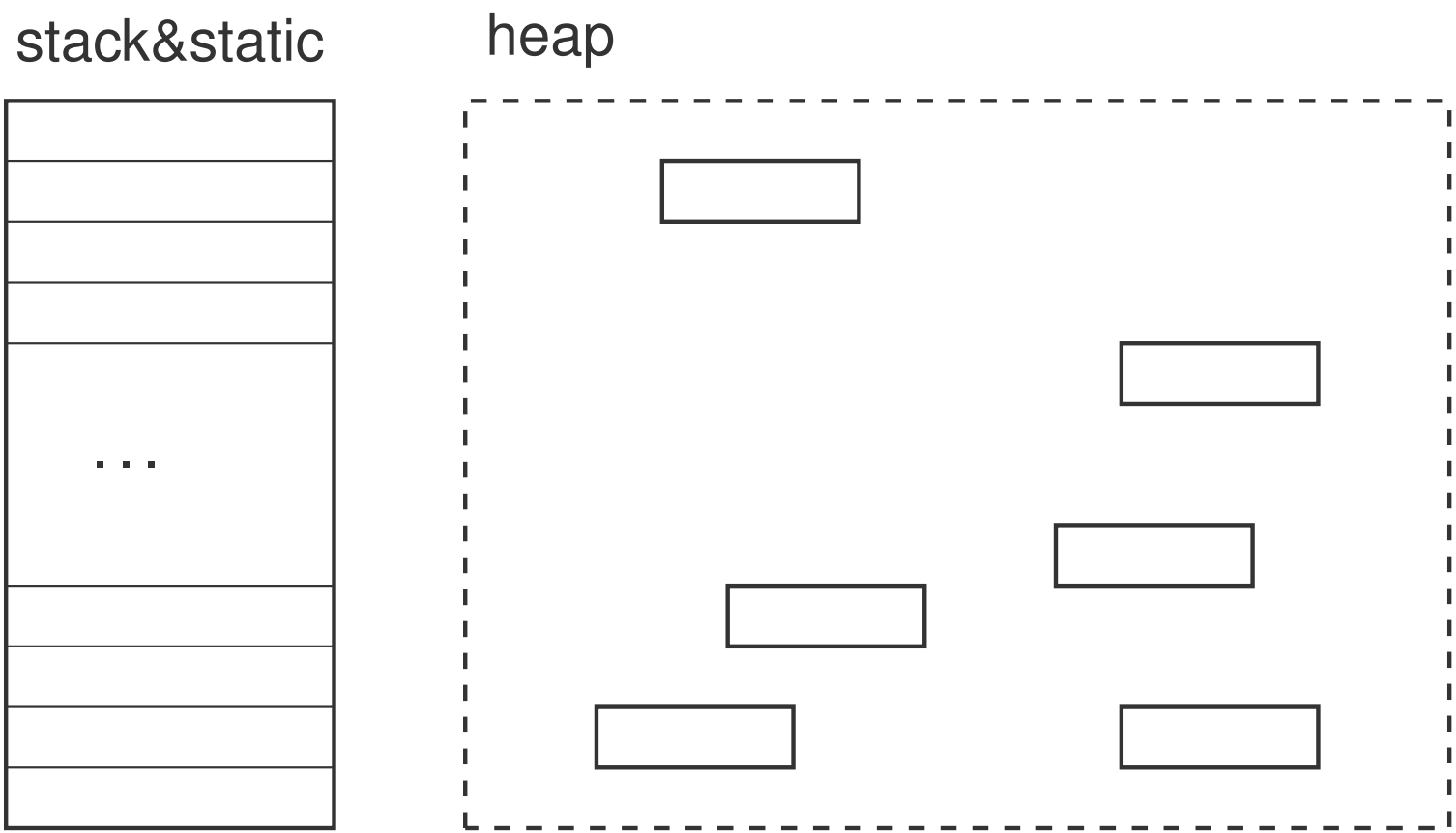
Veoma uprošćena slika!

Memorija je virtuelna.

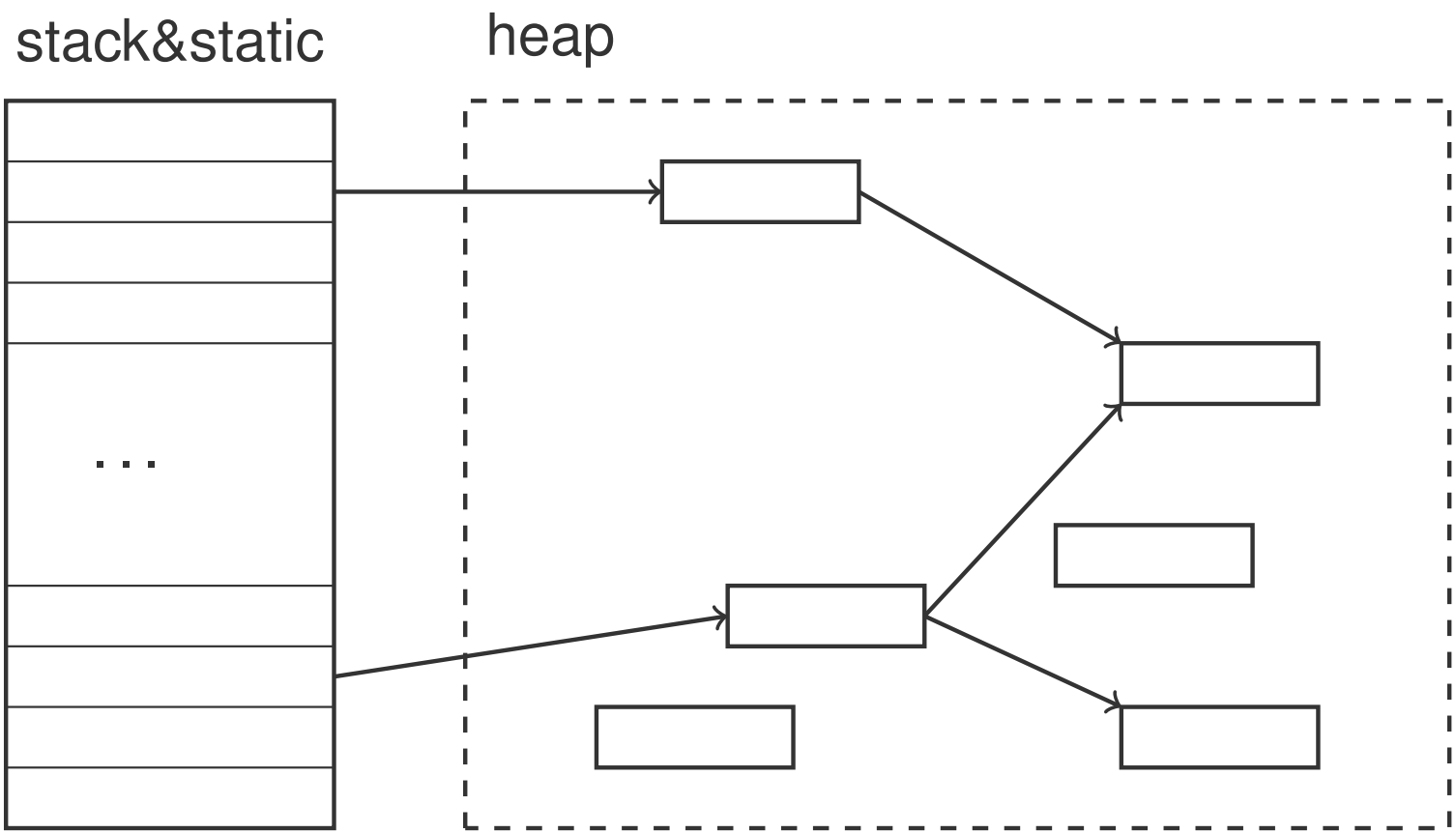
Zadatak sakupljača đubreta

- Želimo da prestanemo da zaboravimo o svim `free` pozivima i da to umesto nas uradi *sakupljač đubreta* (*garbage collector* – skraćeno GC).
- Deo *runtime* sistema jezika.
- **Bezbedna aproksimacija**: ne sme da se oslobodi nešto što ćemo možda koristiti u budućnosti.
- Alokacija na *heap*-u je relativno nepredvidiva, smatramo da nema nekih jasnih pravila gde se nužno šta nalazi.
- Želimo razumne performanse i cenu korišćenja GC (verovatno se poziva onda kada ponestane memorije).

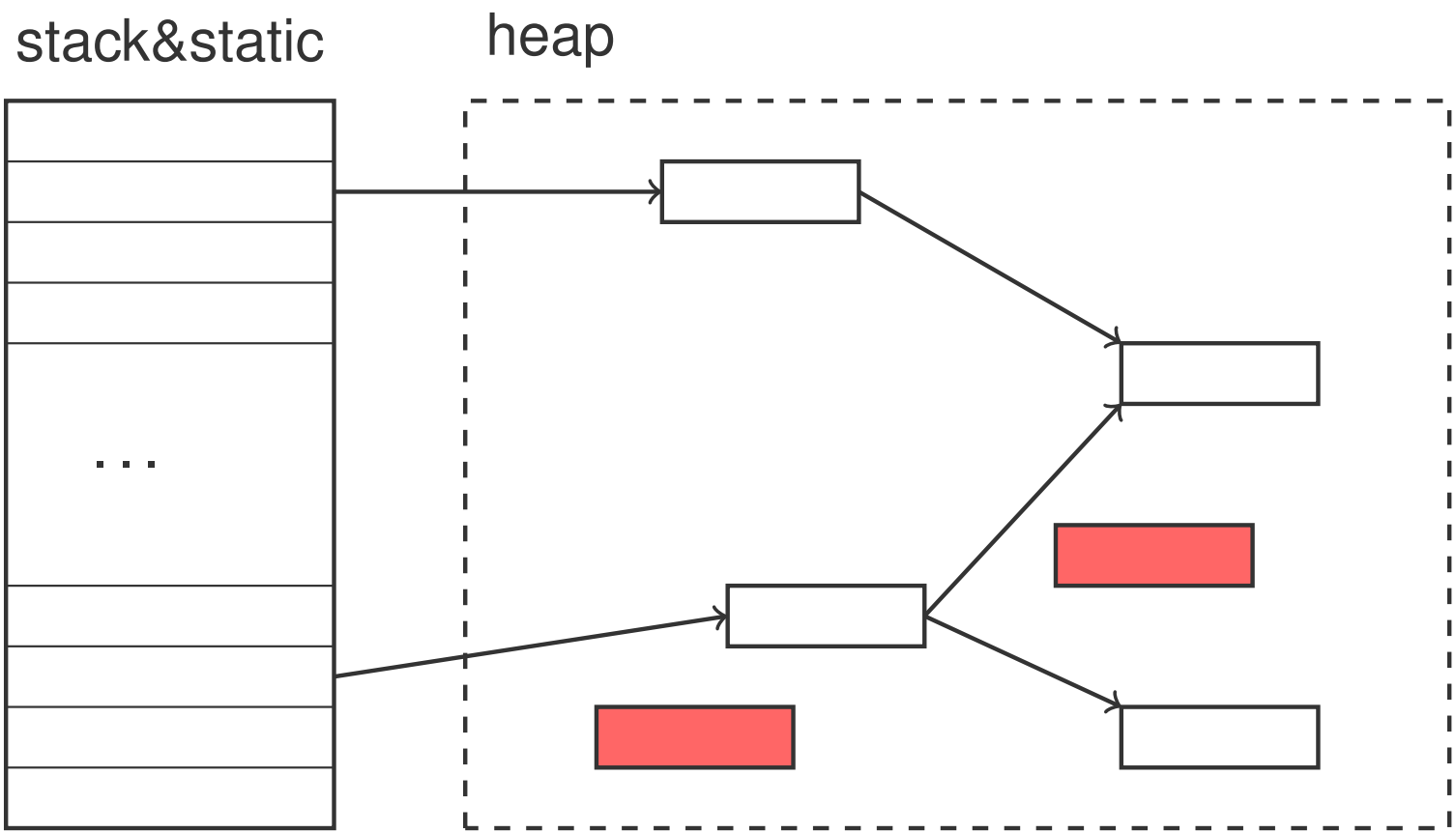
Još svedeniji model memorije



Još svedeniji model memorije



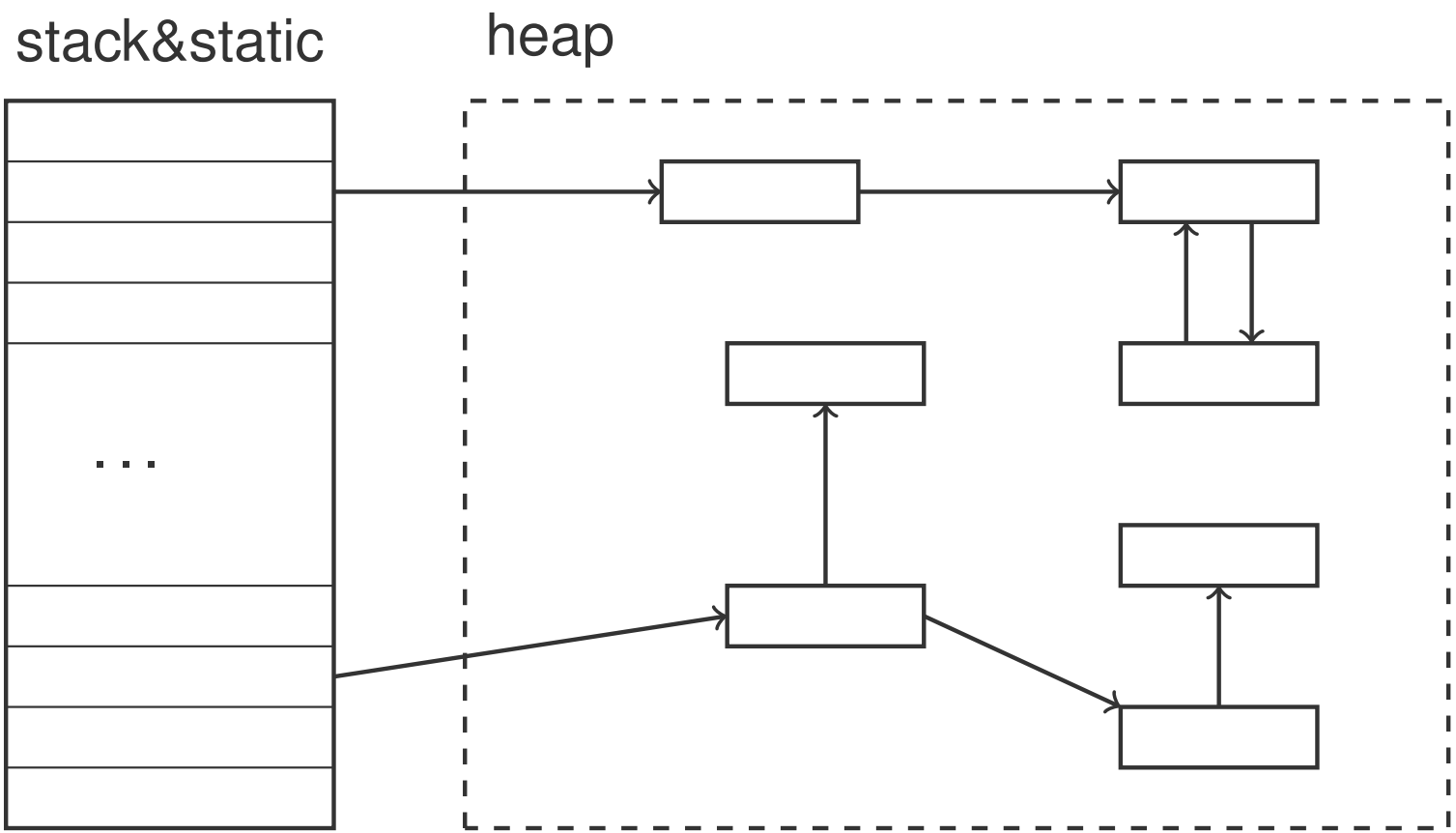
Još svedeniji model memorije



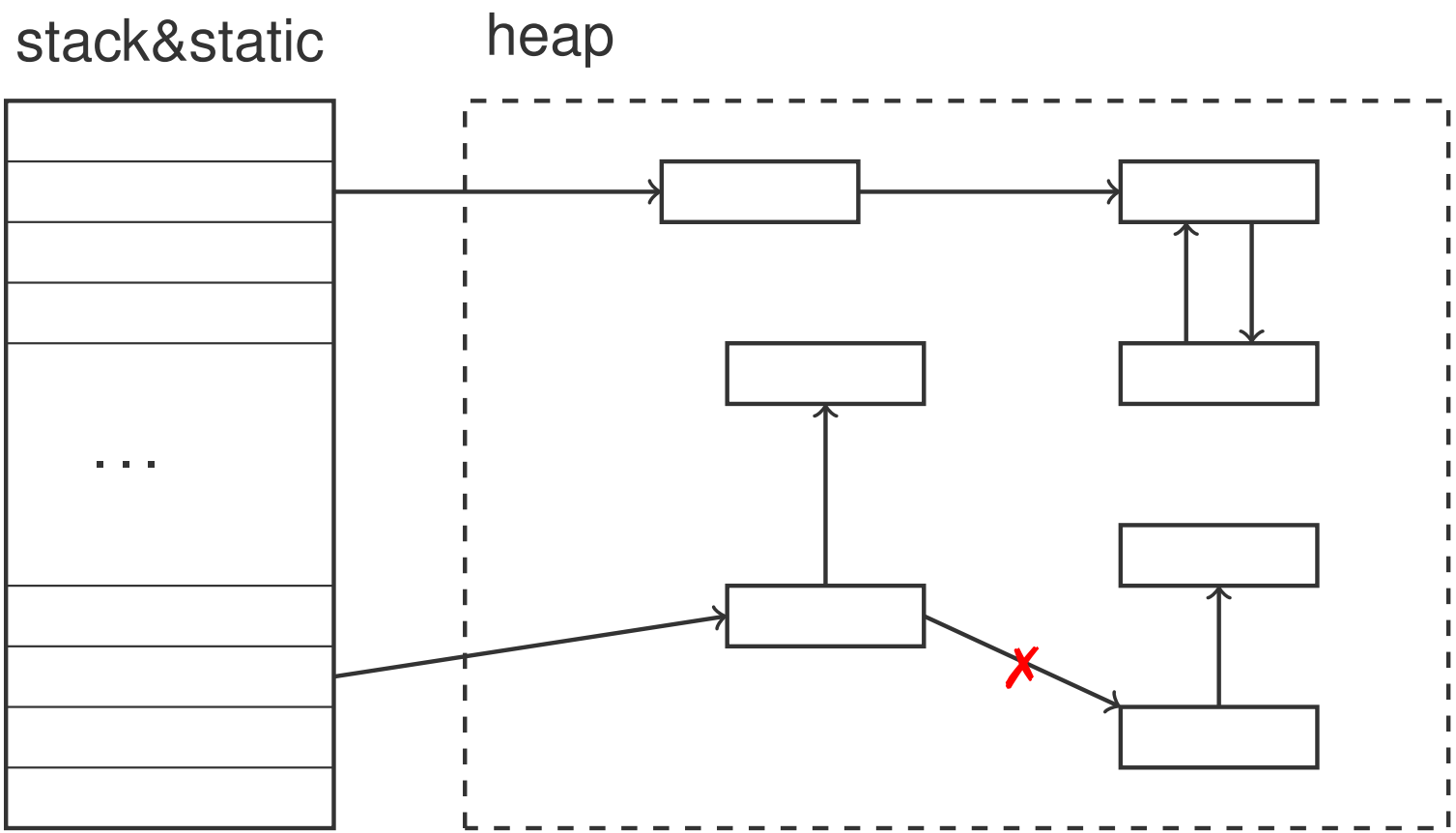
Brojanje referenci

- Šta ako svakom objektu alociranom u *heap* memoriji pridružimo i broj referenci na taj objekat?
- Bezbedno je sakupiti svaki objekat od koga niko ne zavisi (broj referenci je 0).
- Ovaj broj naš *runtime* sistem treba da ažurira i proveri svaki put kada se promeni ili podesi bilo koja referenca/pointer.

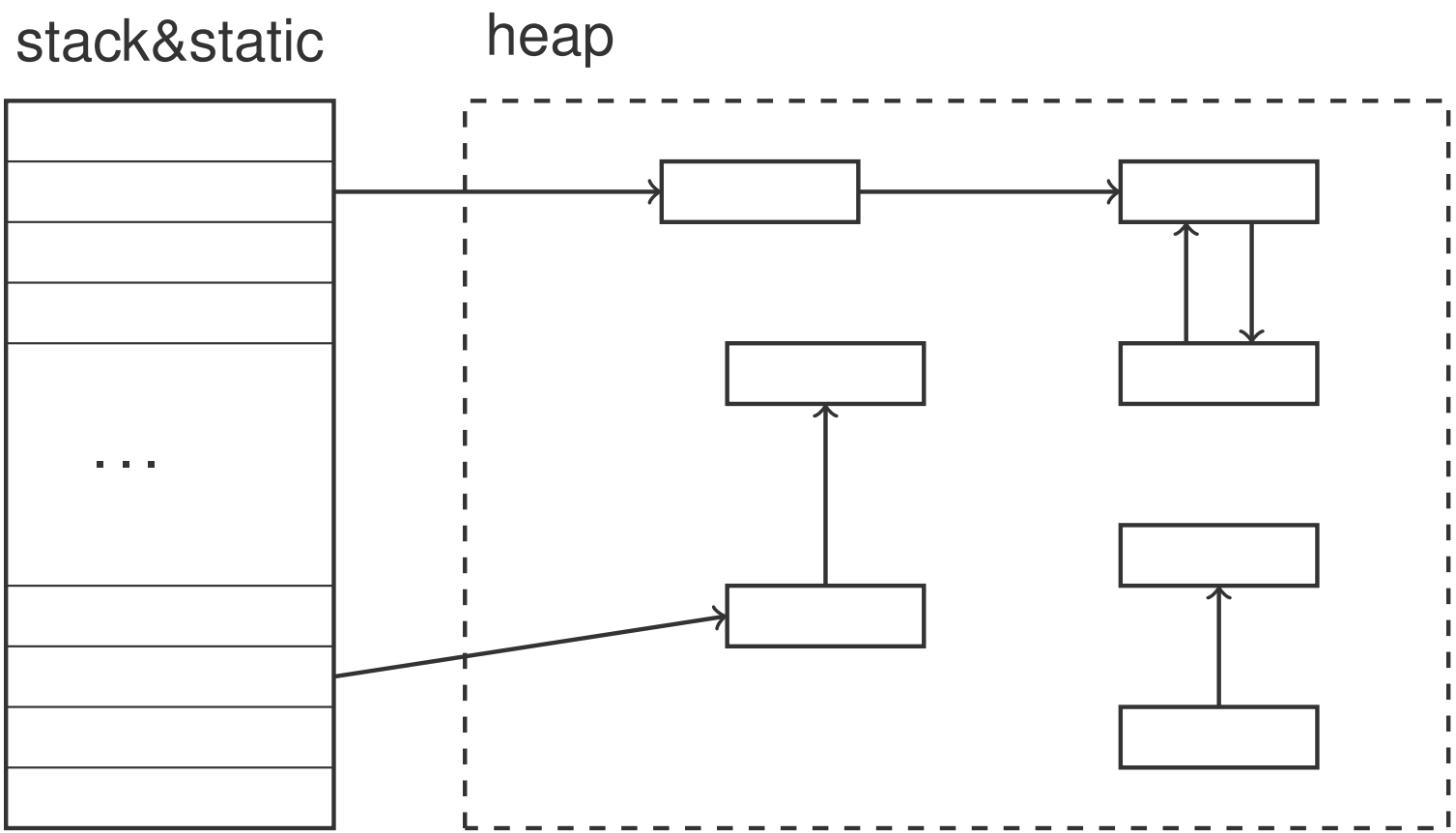
Brojanje referenci: primer



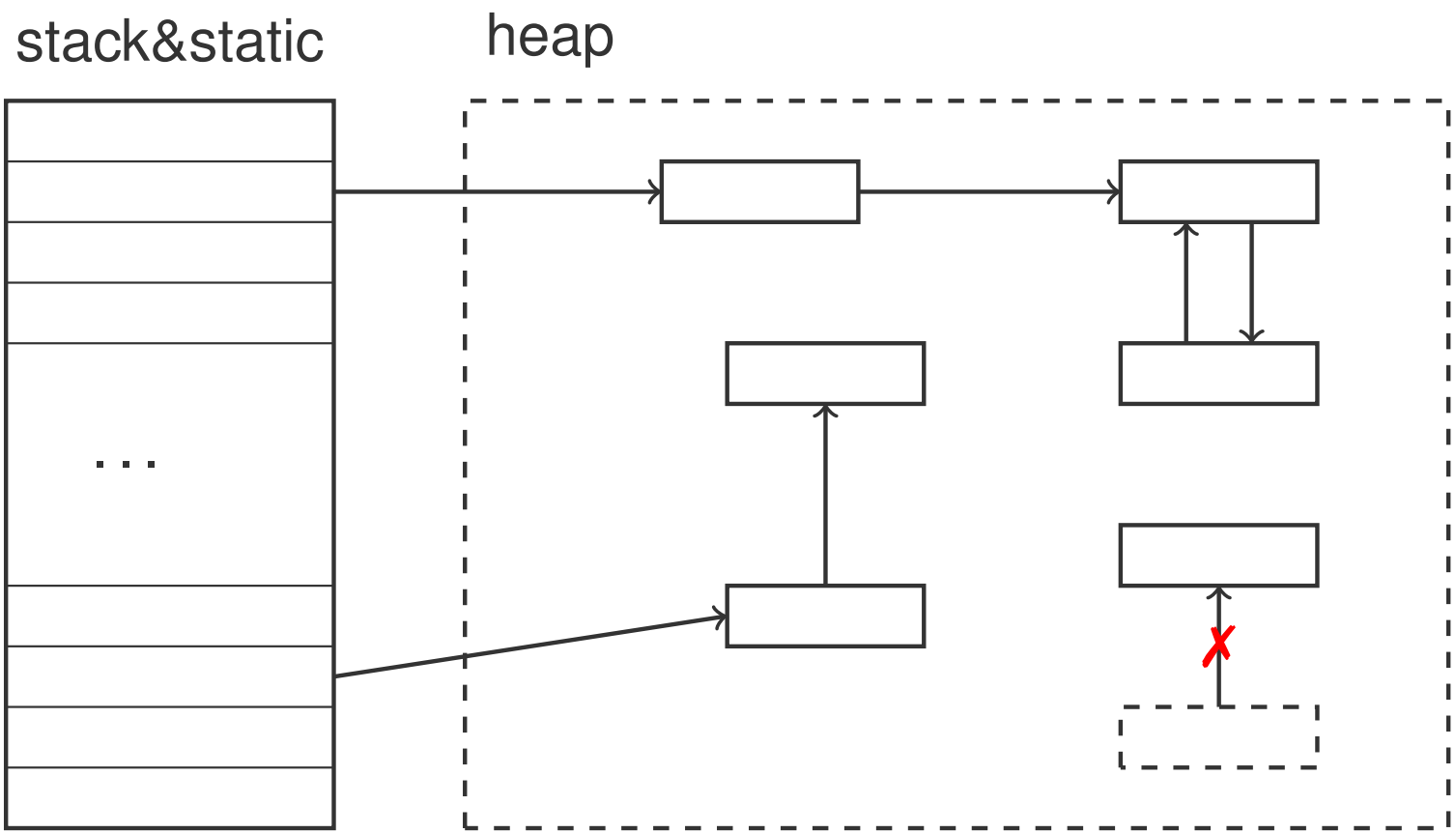
Brojanje referenci: primer



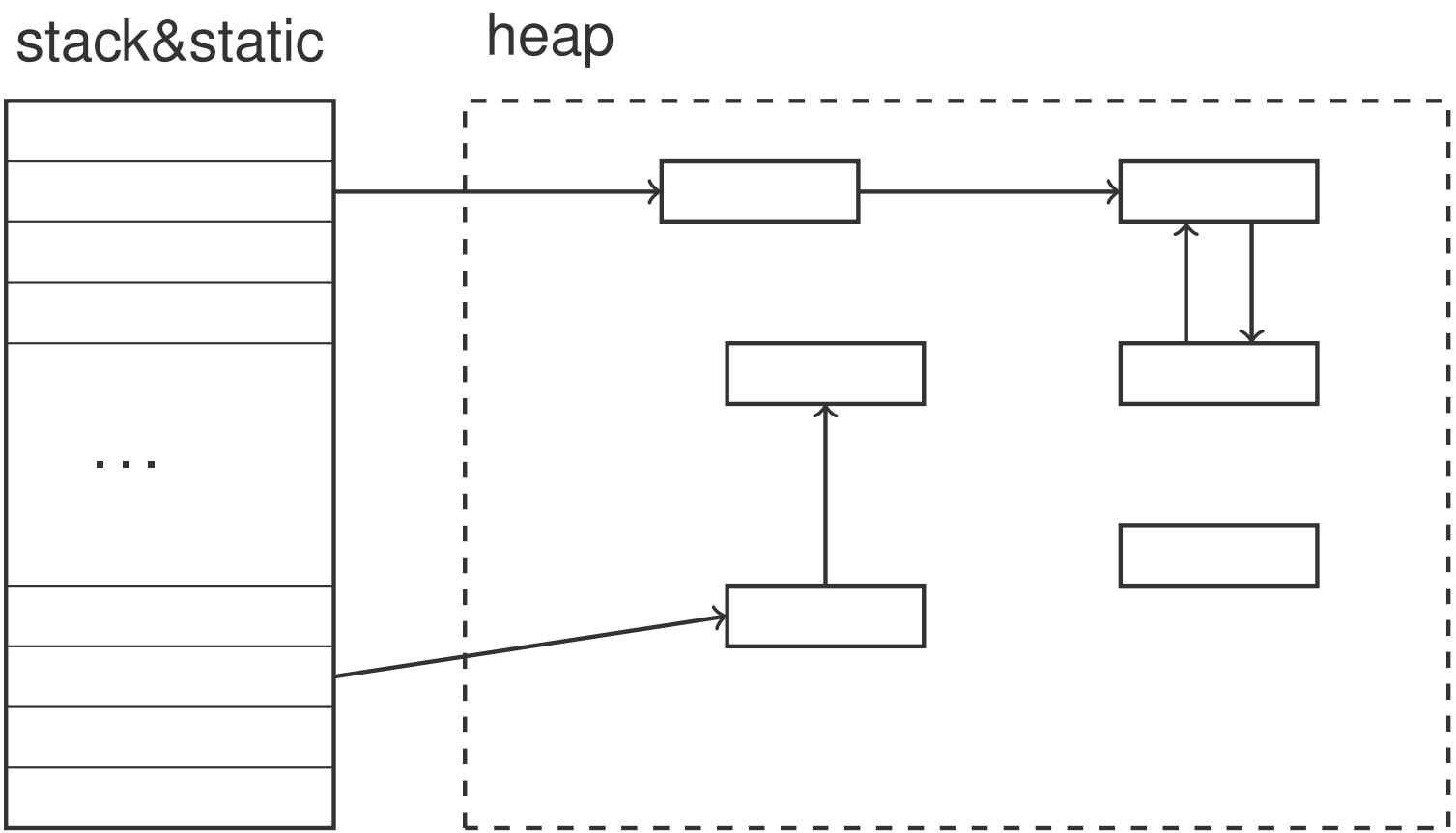
Brojanje referenci: primer



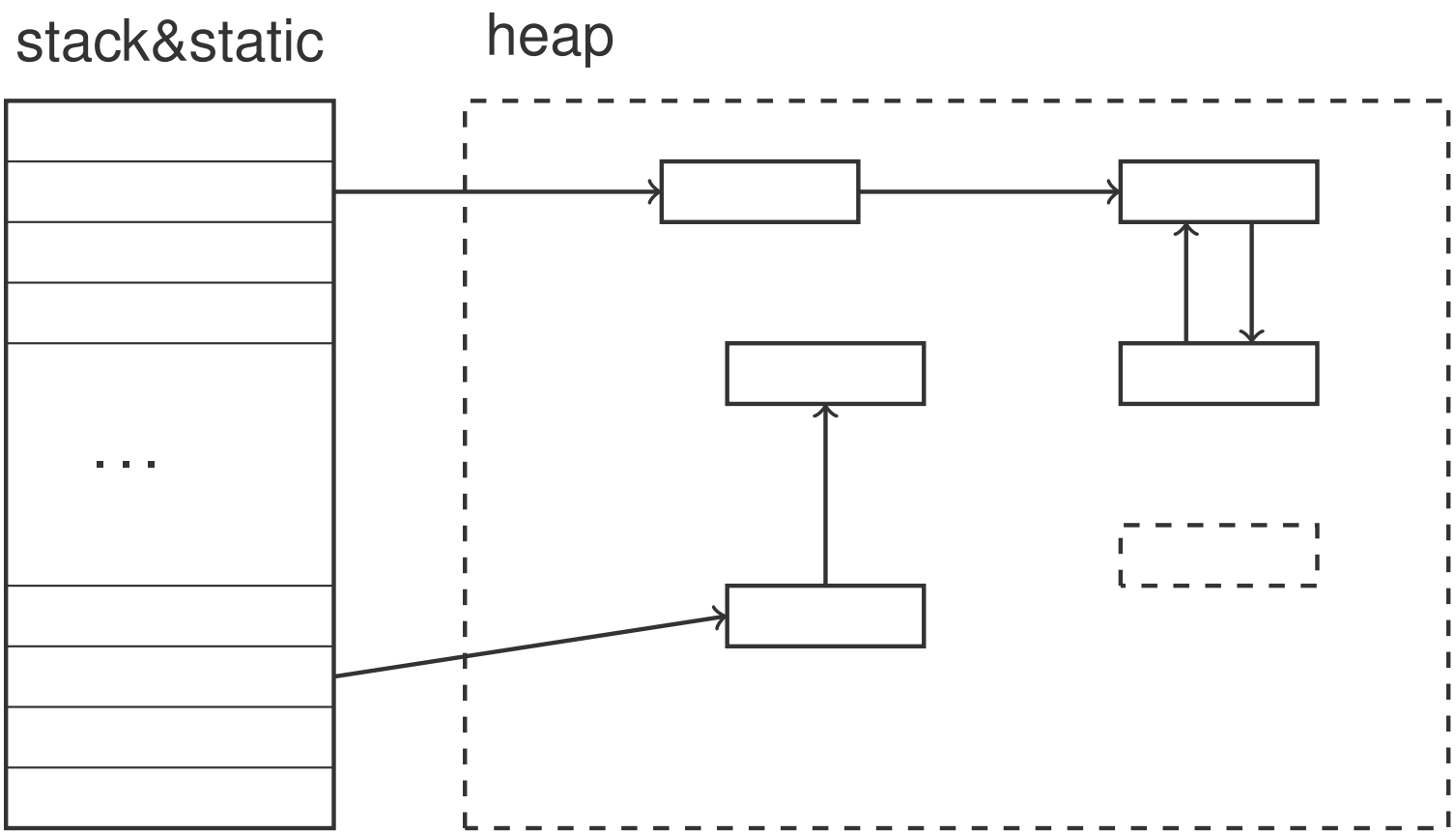
Brojanje referenci: primer



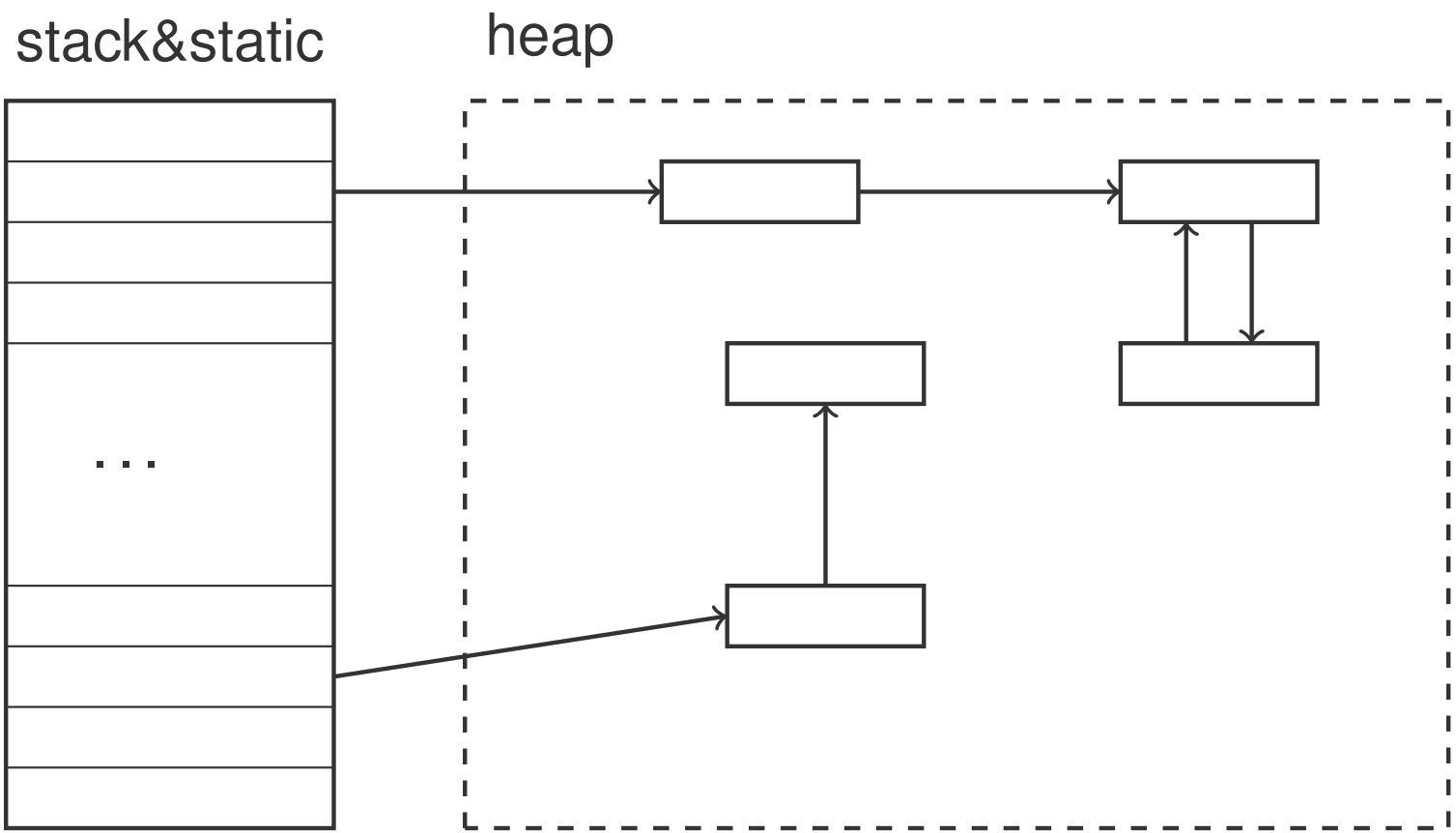
Brojanje referenci: primer



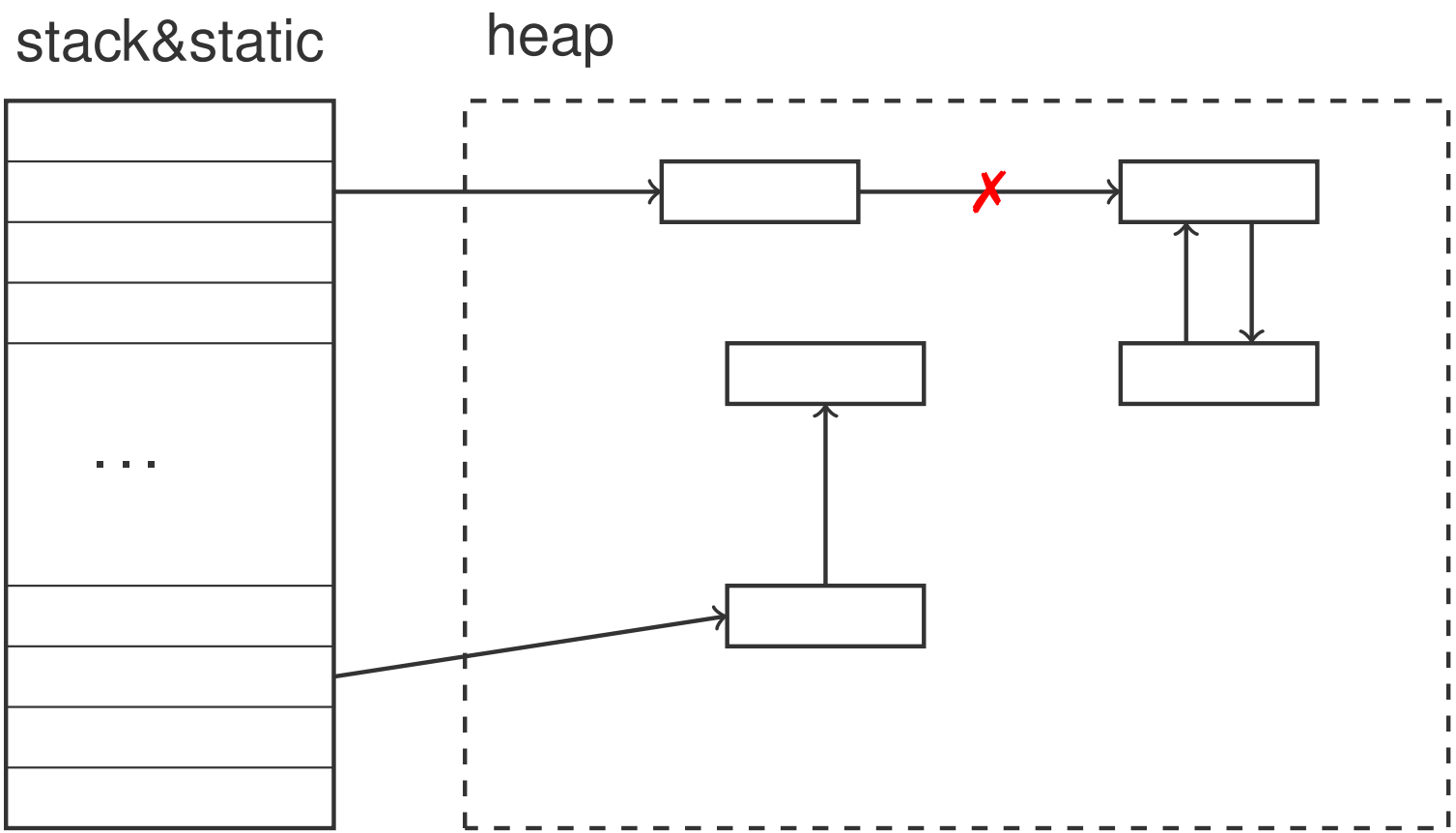
Brojanje referenci: primer



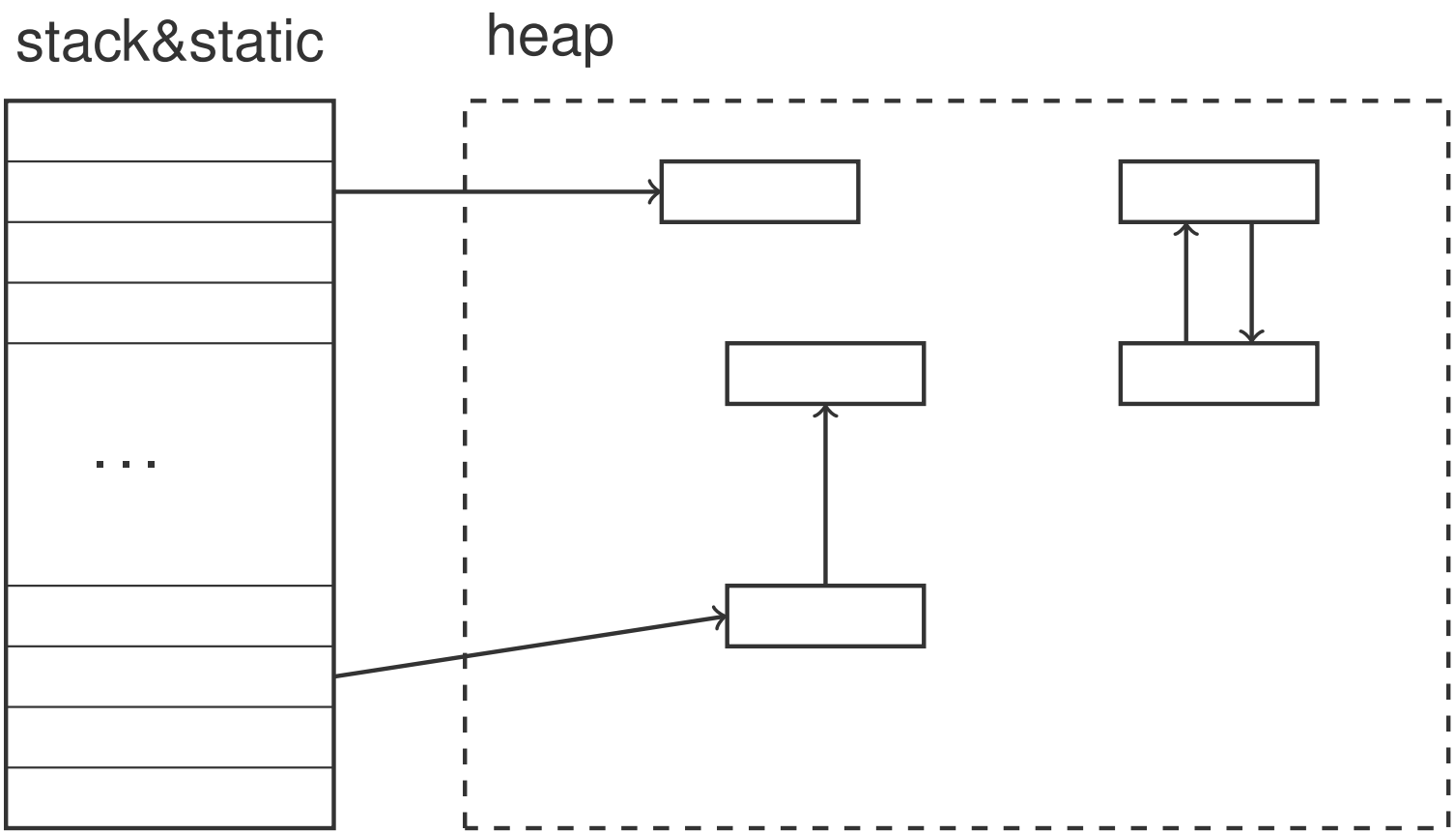
Brojanje referenci: primer



Brojanje referenci: primer



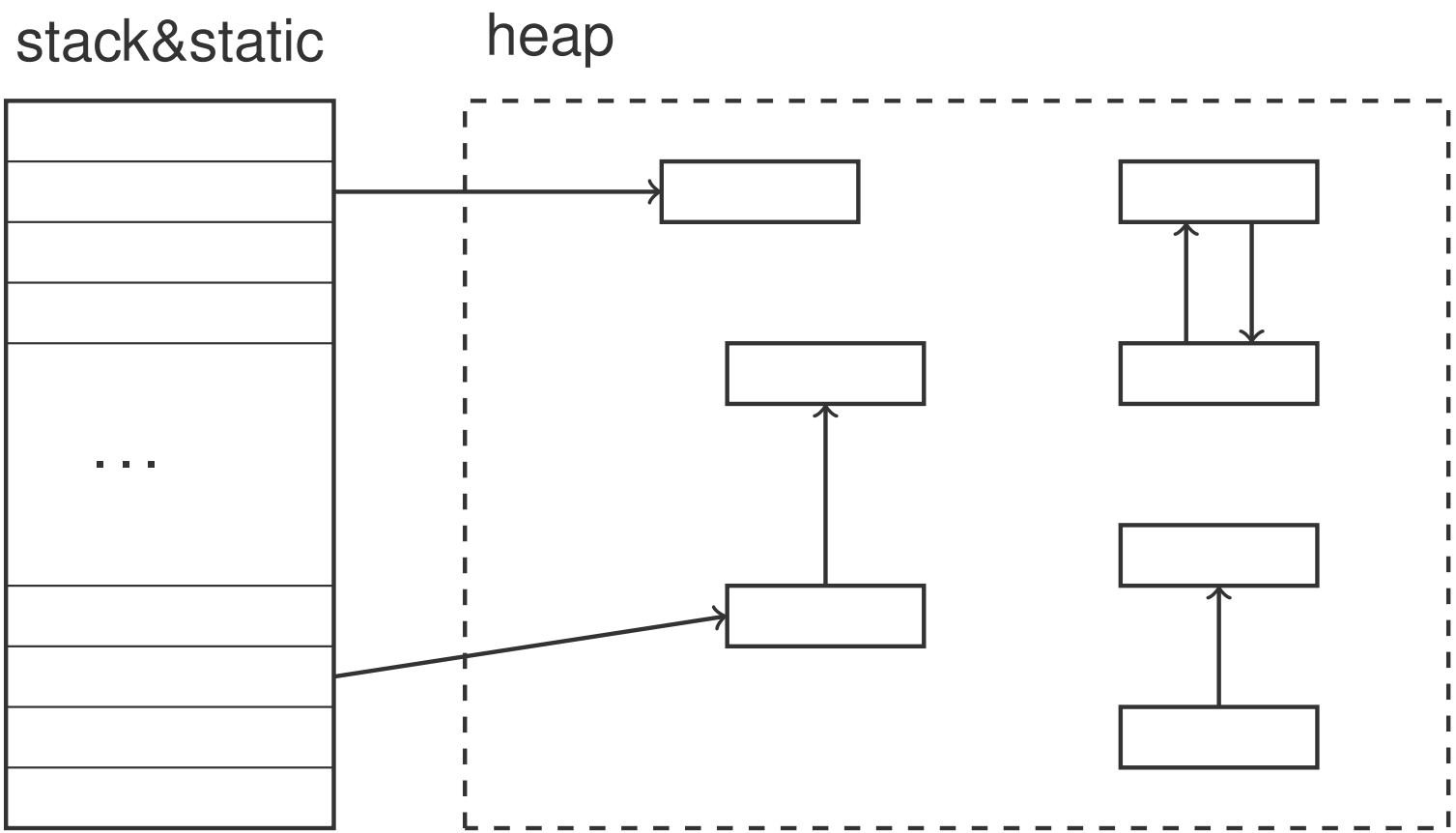
Brojanje referenci: primer



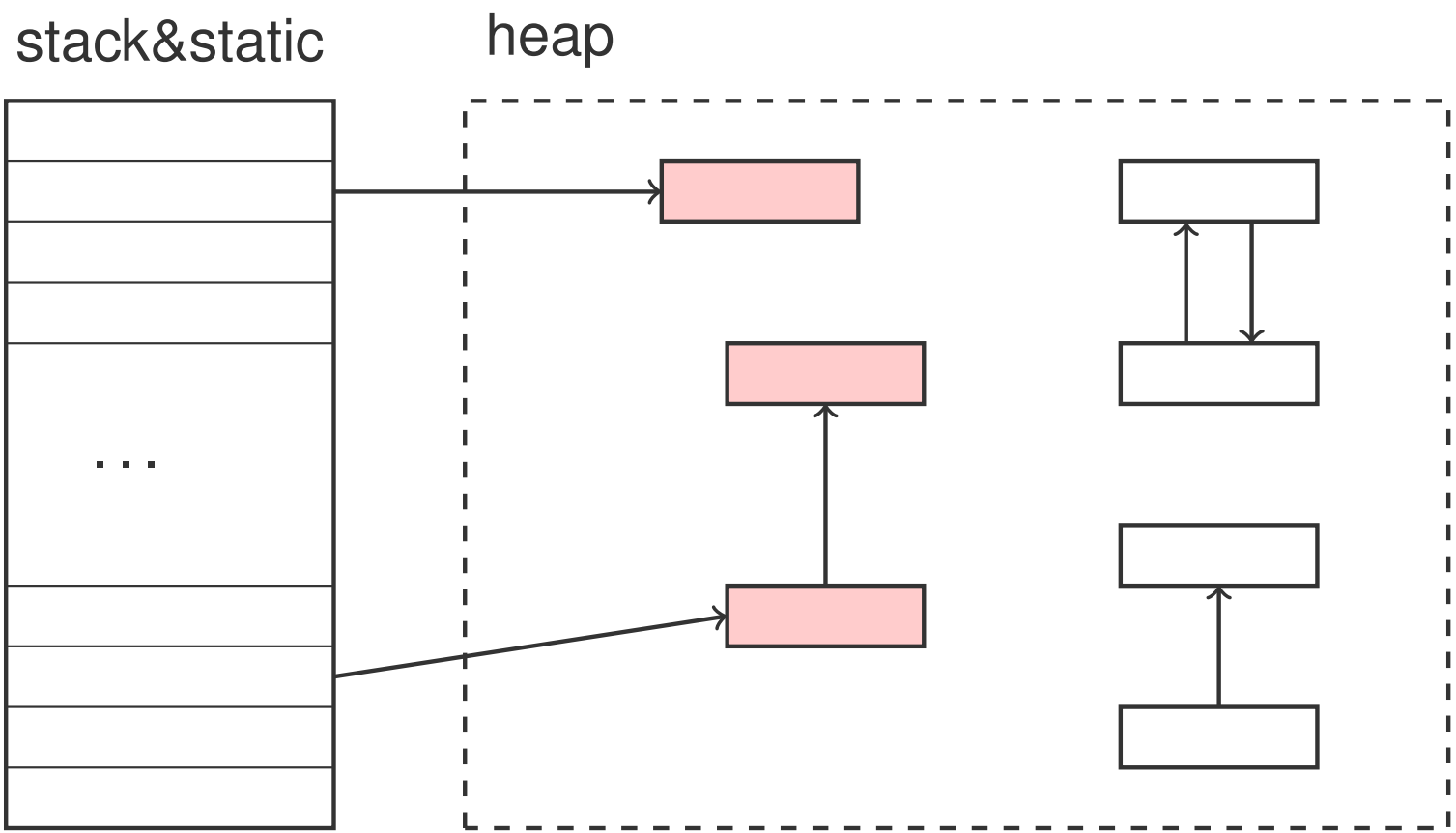
GC praćenjem

- Brojanje referenci neće sakupiti cikluse u grafu referenci (koji mogu da se pojave ako, na primer, imamo dvostruko povezanu listu).
- Zato je neophodno markirati sve objekte u *heap* segmentu do kojih je moguće doći polazeći od *root* skupa (stack&static). Kandidati za sakupljanje su svi objekti do kojih ne možemo da dođemo.
- Ovo može da se radi bilo kojom jednostavnom grafovskom pretragom (potencijalno optimizovanom) – DFS je jedno moguće rešenje.
- Ovo vodi do familije sakupljača đubreta koji vrše **praćenje**.

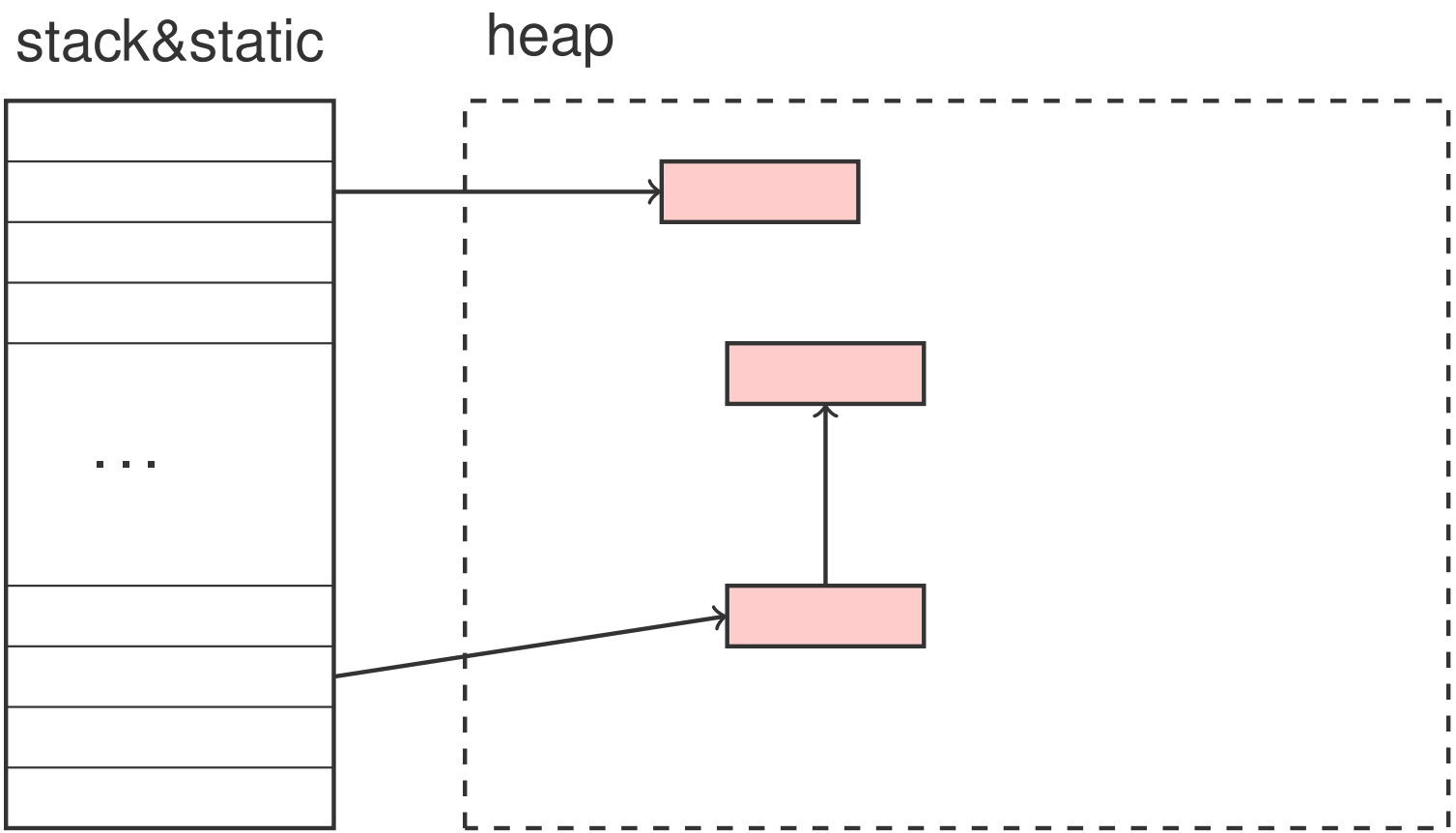
GC praćenjem: primer



GC praćenjem: primer



GC praćenjem: primer



Pomerajući i nepomerajući GC

- Šta uraditi sa preostalim *heap* objektima nakon što se oslobodi prostor?
- **Nepomerajući sakupljač đubreta**: ništa.
- **Pomerajući sakupljač đubreta**: promeniti lokacije nekih objekata u *heap* memoriji, i promeniti vrednosti svih referenci na njih.
- Pomerajući sakupljači đubreta zahtevaju više posla i komplikacija. Međutim, zbog *fragmentacije* pomerajući GC često imaju bolje performanse.

If Java had true garbage collection, most programs would delete themselves upon execution.

—Robert Sewell

Kopirajući GC

- **Kopirajući sakupljač đubreta** deli *heap* na dva dela: *from-space* i *to-space*. Ovo znači da nam je *heap* efektivno prepolovljen.
- Objekti se alociraju u *from-space* sve dok ne treba da se izvrši GC.
- GC vrši markiranje prisutnih objekata. Kada se markira dostupan objekat, on se prebace u *to-space*. Ovo implicitno vrši kompakciju (nema više šupljina). Pošto je reč o pomerajućem GC, treba da se ažuriraju i sve reference.
- Na kraju ciklusa, *from-space* i *to-space* zamene mesta, i nov *to-space* može da se isprazni.

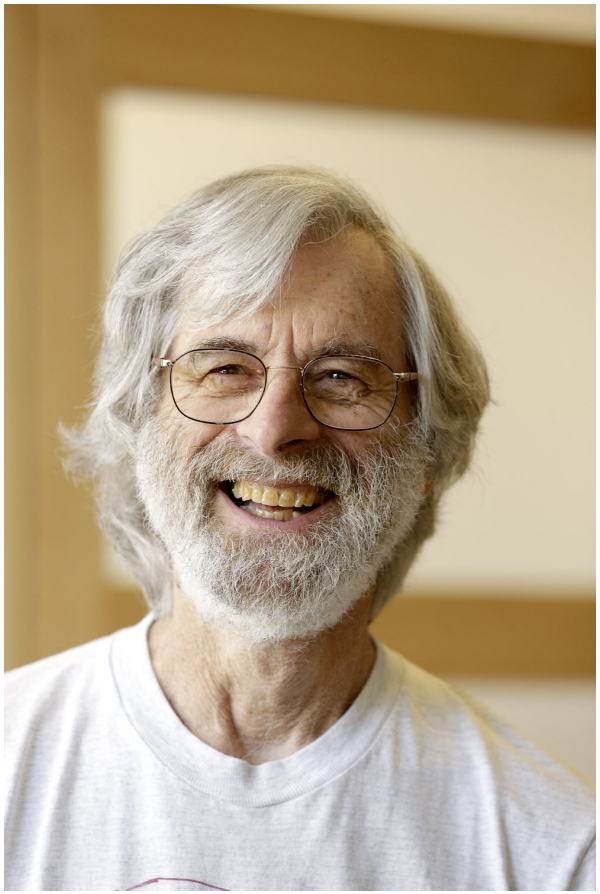
Mark-and-sweep GC

- ***Mark-and-sweep* sakupljač đubreta** se sastoji od faze markiranja i faze čišćenja.
- Faza markiranja proverava do kojih objekata možemo da stignemo.
- Faza čišćenja oslobađa prostor alociran za objekte do kojih ne može da se stigne polazeći iz *root* skupa.
- Ovaj sakupljač đubreta može da postaje pomerajući ako se na kraju izvrši kompakcija.

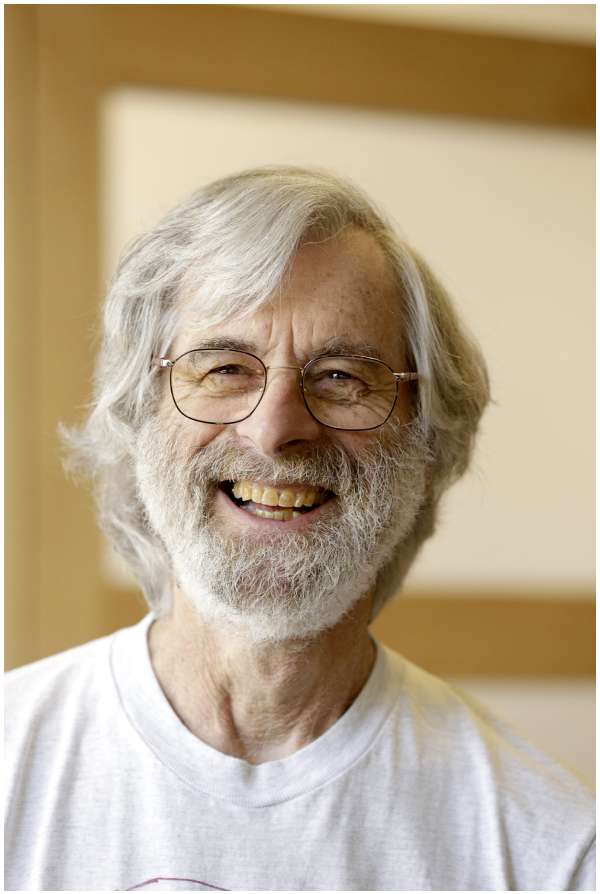
Pregled osnovnih strategija

- **Brojanje referenci**: jednostavno, nepotpuno, neki ne smatraju pravim sakupljanjem đubreta.
- **Praćenje**: pretraga grafa
  - **Kopirajući**: deli *heap* na dva dela, vrši kompakciju
  - ***Mark-and-sweep***: koristi ceo *heap*, mora da se doda kompakcija da bi bio pomerajući, zahteva iteraciju kroz sve objekte da bi u potpunosti izvršio sakupljanje
- *Računarske nauke se tiču ispravnog odabira i kompromisa!*

Koji su ovi informatičari?



Koji su ovi informatičari?



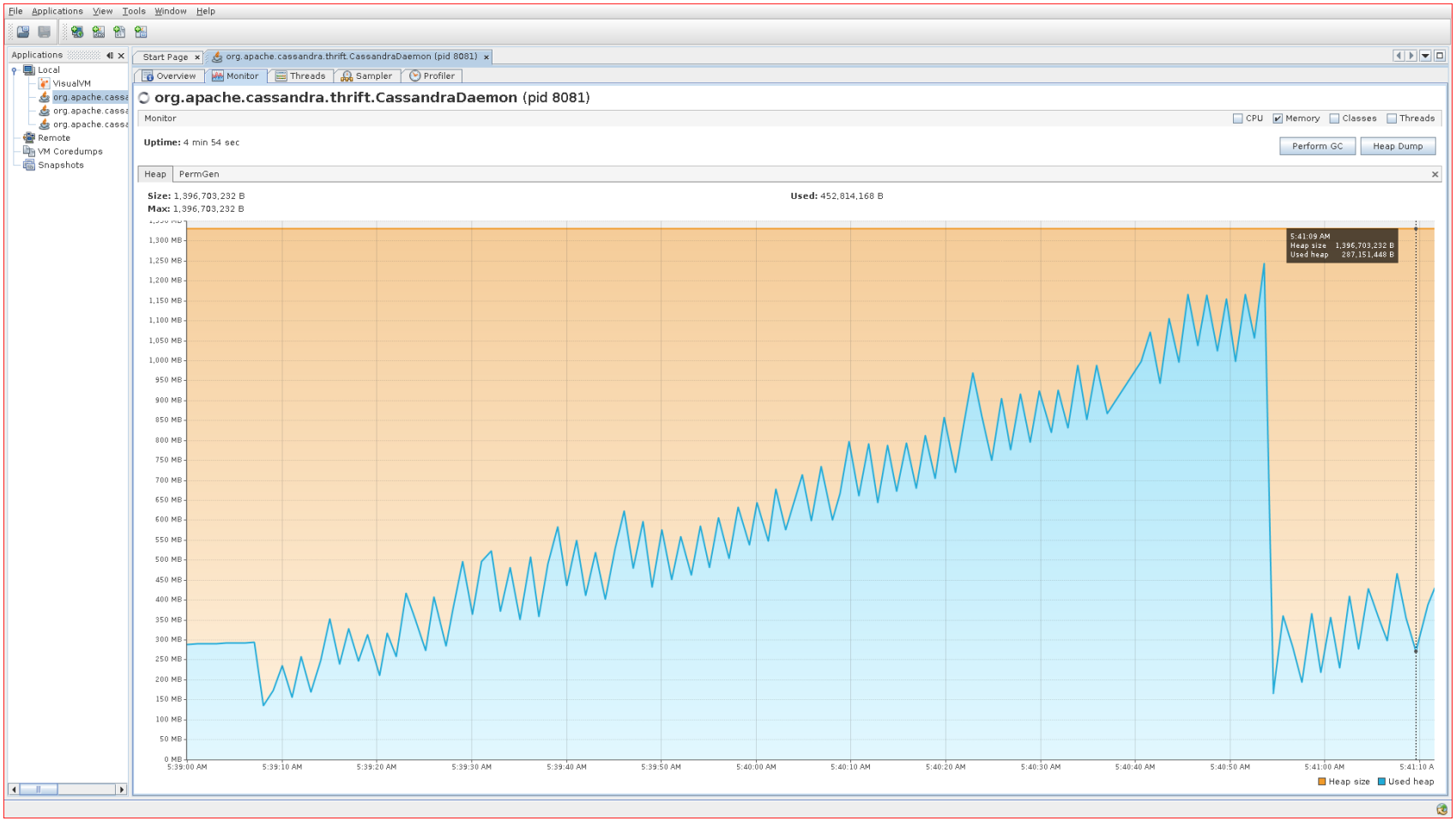
Leslie Lamport

Simon Peyton Jones

Generacijski sakupljač đubreta

- Većina objekata alociranih u heap segmentu *živi jako kratko*.
- Ostali objekti uglavnom žive dosta duže.
- To vodi do dva (ili više) GC nivoa: jedan koji treba da “pokupi” kratko živeće objekte, i jedan koji treba da “pokupi” druge (i ređe se poziva).
- Ovo dovodi do ideje **generacijskog GC**:
  - svi objekti su inicijalno u generaciji 1 i, ako “prežive” GC, postanu deo generacije 2;
  - za generaciju 1 se koristi jednostavna GC strategija, za generaciju 2 neka preciznija.

GC sawtooth



Slika preuzeta sa `javacodegeeks.com` (Peter Schuller, 2012.)

Konkurentni GC

- Najstariji sakupljači su *stop-the-world*.
- Analiza konkurentnih sakupljača đubreta je isto stara tema: 1975. rad Dijkstre, Lamporta, Martina, Šoltena, Stefens: *On-the-fly garbage collection: an exercise in cooperation*.
- Najjednostavniji algoritam je pažljiva, *thread-safe* implementacija *mark-and-sweep* (koristi tri boje za markiranje).
- Savremeni multiprocesorski sistemi su danas posebno relevantni.
- 2009. Zhao et al.: postoji granica nakon koje dodavanje novih niti i procesora ne pomaže.
- 2011. Marlow i Peyton Jones: svaki procesor ima svoj lokalni *heap*.

Distribuirani GC

- Šta raditi ako nam se objekti nalaze na više mašina? Šta ako postoje reference između više mašina?
- Ukratko: teško.
- Jedna mogućnost je da mašine obavljaju redovnu sinhronizaciju i da vrše lokalno sakupljanje đubreta u fiksnim intervalima. Ali i tu je neophodno dosta opreza.

Idle GC

- Haskell ima zanimljiv pristup u paralelnima aplikacijama (uglavnom interaktivnim, bitno u *real time*).
- Ako se u jednoj niti ne dešava nikakvo procesiranje, nakon nekog vremena će se aktivirati GC.
- Ovo bi trebalo da smanji količinu GC onda kada je neophodno brzo izvršiti neko izračunavanje i dobre *real time* performanse.
- Trenutno eksperimentalno, nije poznato da li će ući u širu upotrebu.

Problem fragmentacije

- Nepomerajući sakupljači ostavljaju veliki broj “rupa” u *heap* memoriji. Ovo je *fragmentacija*.
- Jedan mogući problem je da veliki broj malih rupa može da onemogući alociranje velikog objekta, ali do toga dolazi retko.
- Mnogo je važnije to što su onda objekti “razbacani” u različitim linijama u kešu i u različitim stranicama (*paging*).
- Ne koristimo moćne hardverske mogućnosti, a česti *page fault* problemi zahtevaju česte pristupe disku.

Interakcija GC i OS

- Fragmentacija ima lošu interakciju sa hardverom, kao i sa operativnim sistemom koji vodi računa o relevantnoj logici.
- Operativni sistem procesima i nitima daje određene prioritete, i dodeljuje im procesorsko vreme u zavisnosti od prioriteta.
- Ako je jedna nit zadužena za sakupljanje đubreta i ima neodgovarajući prioritet, dolazi do svakakvih problema.

Sakupljanje đubreta u C?

- Izvor svih problema sa memorijom u C su pozivi poput `malloc` i `calloc`.
- Šta ako bismo ih zamenili nekim svojim verzijama, `GC_MALLOC` i `GC_CALLOC`, koje bi pozivale sakupljač đubreta po potrebi?
- Postoje silne moguće užasne stvari koje možemo da radimo sa pokazivačkom aritmetikom. Zato ćemo svaku vrednost upisanu u memoriji programa potencijalno smatrati adresom.
- Ovo je ideja ***Boehm garbage collector***. Ispitajte `bdwgc`!

Sakupljanje đubreta u C++?

- Novi standardi C++ nam omogućavaju da radimo jako zanimljive, bezbednije stvari.
- `unique_ptr`: "samo ja imam pokazivač na ovu adresu, automatski uradi brisanje kada i ja budem obrisan"
- `shared_ptr`: "ja sam u grupi koja deli ovaj pokazivač, briši objekat kada ta grupa postane prazna"
- Ovi **pametni pokazivači** govore o tome šta kome *pripada*, te ciklusi u grafu referenci nemaju mnogo smisla.
- U C++ ne treba da koristimo `malloc` i `free`, već `new`, `delete`, `delete[]`.

Memorijski regioni

- Umesto da nam logičke celine (strukture podataka poput povezanih lista) budu "razbacane" po *heap* memoriji, što ne bi sve bile u jednom manjem **regionu**?
- Unapred rezervišemo `malloc` pozivom neki prostor u kom će biti implementirana cela struktura podataka, koju onda polako "popunjavamo".
- Sakupljanje svih objekata strukture podataka se radi samo jednim pozivom `free` rutine.

GC u čisto funkcionalnim jezicima

- Haskell je i ovde inovativan.
- Promenljive su “nepromenljive” (imutabilne) i vrednost im se dodeljuje tačno jednom.
- To znači da nijedan objekat `a` ne može da sadrži referencu na objekat `b` koji je kreiran nakon kreacije objekta `a`.
- Odlična interakcija sa generacijskim sakupljanjem đubreta.
- Kontraintuitivno ponašanje: što je više đubreta, to je sakupljač đubreta brži.

Šta uzeti kao koren praćenja?

- Možda je naš izbor *root* skupa prevelik i treba da ga svedemo.
- Postoje u *stack* segmentu i neki objekti za koje možemo da garantujemo da više neće biti korišćeni (takozvane mrtve promenljive).
- Detekcija mrtvih i živih promenljivih je standardna analiza koju vrše savremeni kompajleri.
- 2014. Asati, Sanyal, Karkare, Mycroft: *Liveness-Based Garbage Collection*. Aktuelno istraživanje, do sada samo u funkcionalnim jezicima.

Formalna verifikacija GC

- Teoretičari vole da dokazuju da su algoritmi i programi korektni.
- Dijkstrin i Lamportov sakupljač đubreta su odavno dokazano tačni.
- Savremene tehnike formalne verifikacije programa (*Concurrent Separation Logic* i slične metode) omogućavaju pisanje formalnih dokaze da su konkretne implementacije tačne.
- 2015. Gammie, Hosking, Engelhardt: *Relaxing safely: verified on-the-fly garbage collection for x86-TSO*.
- Formalna semantika – neophodni ljudi koji vole grčka slova u računarstvu.

Zaključci

- Sakupljanje đubreta je stara ideja i paradigma, koja je postala mnogo poznatija širenjem Jave, C# i Python-a. Da li to znači da će neke druge stare ideje isto tako postati popularne tek u budućnosti?
- Veliki broj različitih pristupa problemu GC, povezuje teoretičare, sistemske programere, dizajnere kompajlera.
- Za nekog je nešto đubre, a za nekog drugog neprocenjivo bogatstvo. £ $ RSD € ¥