14
Garbage collection 1.

Motivation.
Begreber.
Klassiske teknikker.
- Reference counting
- Mark and sweep garbage collection.
- Kopierende garbage collection.

PS4 - Garbage collection

Noter

Der findes en meget stor mængde tilgængelig litteratur om garbage collection. Til dette kursus læser vi oversigtsartiklen

> *"Uniprocessor Garbage Collection Techniques"* fra **The 1992 International Workshop on Memory Managment,** udgivet på Springer's Lecture Notes in Computer Science, nr. 637.

Artiklen er skrevet af Paul R. Wilson fra University of Texas.

Artiklen er tilgængelig på postscript format fra

> cs.utexas.edu:/pub/garbage/gcsurvey.ps

Motivation.

- Deallokering af objekter kan realiseres på to forskellige måder:
  1. **Eksplicit**: En del af en programbeskrivelse, ligesom allokering.
  2. **Implicit**: Ved "automatisk" opsamling af objekter, hvorom det kan bevises at de ikke længere har nogen mulighed for at påvirke programmets videre udførsel.
- Automatisk opsamling af spild er en forudsætning for 'modulære programmer'.
  - Hvilket modul afgør, hvornår fælles data skal deallokeres?
- Eksplicit programmeret deallokering er vanskeligt at gennemføre konsekvent.
  - Stor fare for, man glemmer noget spild.
- Nogle store applikationer anvender en applikationsspecifik garbage collection.
  - Ad hoc, ofte fejlbehæftet og ineffektivt.
- Det er attraktivt én gang for alle, og applikationsuafhængigt, at løse deallokeringsproblemet.
- Garbage collection i forhold til 'Dynamiske Sprog og Omgivelser':
  - En selvfølge i dynamiske sprog så som Lisp og Smalltalk.
  - Er med til at højne abstraktionsniveauet, idet visse (vigtige) detaljer kan ignoreres.

Noter

Garbage collection er som regel bandlyst i sprog, hvori man programmerer systemer møntet mod stor effektivitet. Vi ved alle, at C og C++ ikke undersøtter garbage collection. Garbage collection er heller ikke et emne i sprog, hvor dynamisk allokeret data spiller en mindre rolle, såsom Pascal.

I sprog hvor de fleste data er dynamisk allokerede er garbage collection næsten "et must". Langt de fleste dynamiske sprog er omfattet af dette, men også mere traditionelle "statiske" omgivelser, såsom Simula og Eiffel, er baseret på garbage collection.

Garbage collection begreber.

- *Levende objekter* (live objects).
- *Døde objekter*: spild fordi det blot fylder i lageret (garbage).
- Forskellige, mulige kriterier til afgørelse af om et objekt er dødt eller levende:
  - **Reelt kriterium**: Et objekt er dødt hvis det aldrig kan bruges af (og dermed kan påvirke) det kørende program.
  - **Konservativt kriterium**: Et objekt er levende hvis det kan *nåes* fra en mængde af objekter, der betegnes som *rødder*.
    - Rødder: Globale variable, og lokale variable af aktiverede procedurer.
- Antagelse: Dynamisk allokerede objekter er selv-identificerende hvad angår udstrækning, og indhold af referencer til andre objekter.

Noter

Den naturlige danske betegnelse for 'garbage collection' er *spildopsamling*. Termen 'garbage collection' er dog også almindelig og brugbar på datalogisk dansk, og vi vil veksle mellem de to betegnelser.

De levende og døde objekter udgør en klassedelning af den samlede mængde af objekter.

Reference counting.

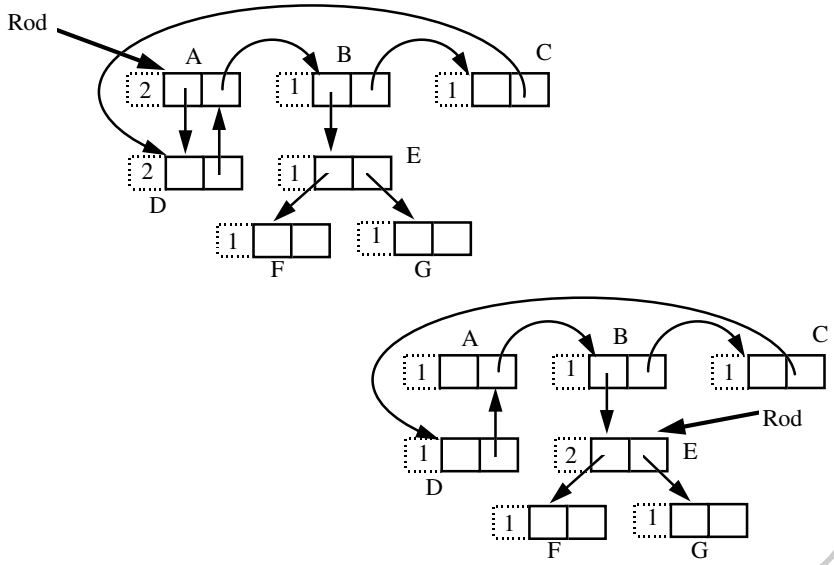
- Ethvert objekt indeholder et (skjult) *reference tæller felt*, der angiver hvormange referencer der peger på objektet.
- Hvis reference tæller feltet bliver 0 i et objekt, kan objektet deallokeres.
- Når et objekt P deallokeres, nedtælles reference tæller feltet i alle objekter, der direkte refereres fra P.
- Vurdering af reference counting:
  - ***Positive sider***:
    - Simpel at forstå og simpel at implementere.
    - Inkrementel af natur: kan realiseres side om side med normal programudførelse.
  - ***Negative sider***:
    - Cykliske datastrukturer bliver ikke deallokeret.
    - For kostbar i tid:
      - Tidsforbruget af 'reference counting' er proportional med antallet af primitive manipulationer af referencer.
  - Reference counting regnes ikke for en attraktiv teknik til opsamling af spild.

Noter

I nogle fremstillinger regnes reference counting ikke for garbage collection, idet gen-inddrivelsen af spild er så tæt integreret i det normale programforløb, at man ikke kan tale om en særlig spildopsamlingsproces eller -procedure. Vi vil dog, lige som i oversigtsartiklen om garbage collection, betragte reference counting som en spildopsamlingsteknik.

Tidsforbrug: Det observeres i artiklen at tidsforbruget ved reference counting er proportional med programmets køretid. Det bunder i iagttagelsen om, at et program typisk det meste af tiden ændrer på referencerne mellem objekter. Enhver sådan ændring afstedkommer en ændring på de involverede objekters referencetæller. Det ville være mere attraktivt, hvis tidsforbruget af 'garbage collection processen' kunne gøres proportional med mængden af spild, eller måske med mængden af det totale antal objekter (døde såvel som levende).

Scenarium af reference counting.



PS4 - Garbage collection

Noter

I dette eksempel, og fremover, vil vore dataobjekter være (isomorfe med) Lisp CONS celler. Referencetælleren er vist i den stiplede del af en CONS celle.

Den nederste del af tegningen er identisk med den øverste, blot har vi flyttet roden, og vi har fjernet referencen fra A til D. Bemærk cyklen A B C D A. Men den givne rod forneden til højre burde A, B, C og D deallokeres, men læg mærke til, at deres reference tæller ikke er nul.

Mark and sweep garbage collection.

- **Basale ide**:
  1. Mærk alle levende objekter ved et systematisk lagergennemløb.
  2. Fjern alle objekter, som ikke er mærkede: de døde objekter.
- ***Positive sider***:
  – Kan også fjerne cykliske datastrukturer.
- ***Negative sider***:
  – Ikke-inkrementel.
  – Kan føre til *fragmentering* af lageret.
  – Relativ kostbar: Proportional med det samlede antal objekter.
  – Kan føre til manglende *lokalitet* af lageret:
    – Opsamlingsteknikken indeholder ingen form for flytning eller samling af de levende objekter.
    – Logisk sammenhørende objekter bliver derfor let spredt i lageret.
    – Gamle og nye objekter bliver allokeret side om side.
- **Variation**: *Mark and compact* garbage collection:
  – I stedet for fase 2 flyttes levende objekter, således at fragmentering undgåes.
  – Dyrt, idet (endnu) flere gennemløb af objekterne er nødvendig.

Noter

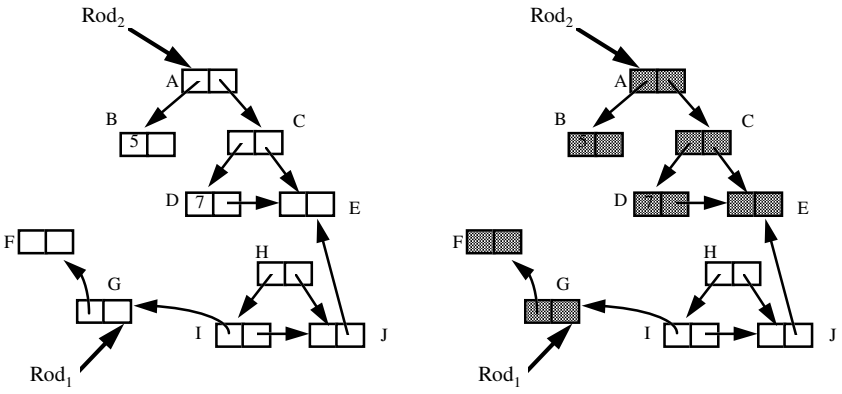
Bemærk at fase to af teknikken (sweep fasen) skal realiseres ved et nyt gennemløb af alle objekter, hvor man objekt for objekt undersøger om objektet er levende eller dødt. Et linært gennemløb er godt nok, hvis et sådant kan realiseres.

Fragmenteringsproblemet: Lageret fyldes op med objekter af forskellig størrelse, og med indbyrdes mellemrum (frit lager) af forskellig størrelse.

Lokalitet: Tæt forbundne objekter i nettet kan blive allokeret i vidt forskellige dele af lageret. Der er ikke indbygget nogen form for compaction i 'mark & sweep'. Dette er specielt uheldigt, når vi har *virtuelle lagre* (med paging).

Sweeping betyder at feje (gøre rent).

Scenarium af ‘mark and sweep’ garbage collection (1).



Oprindelig situation.

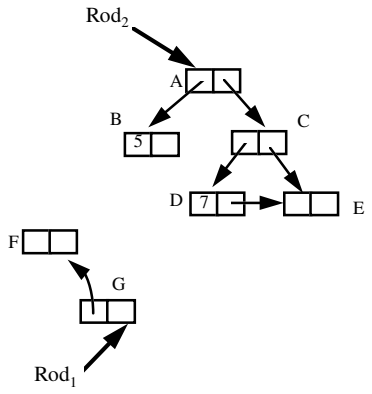
Efter ‘mark’.

Noter

På næste side fortsættes eksemplet, idet vi der viser situationen efter sweep fasen.

Man må observere, at garbage collection indtræder, når vi er løbet ud for lager. I de fleste realiseringer af garbage collection gælder der imidlertid, at der skal bruges noget lager til administration af et gennemløb, såsom ‘marking’ i ‘mark & sweep’. Vi skal altså iværksætte en garbage collection på det tidspunkt, hvor vi har så lidt frit lager tilbage, at vi netop vil være istand til at administrere et gennemløb. Der findes snedige udformninger af garbage collection teknikker, som er optimeret til at kræve meget lidt lager til administration af gennemløb.

Scenarium af ‘mark and sweep’ garbage collection (2).



Efter ‘sweep’.

Noter

Kopierende garbage collection.

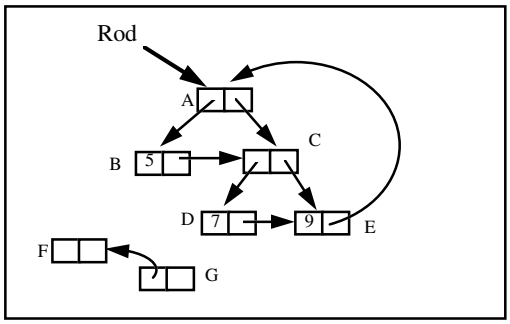
- **Basal ide**:
  - Lageret opdeles i to halvdele (from space, to space), hvoraf kun den ene kan anvendes effektivt.
  - Når én lagerdel er fyldt op, kopieres levende objekter over i den anden del, og rollerne mellem de to lagerhalvdele ombyttes.
- Mulig konkret realisering (stop-and-copy, Cheneys algoritme):
  - Bredde først gennemløb af objekter.
  - Særligt bogholderi skal sikre, at "delte objekter" ikke kopieres mere end én gang.
- ***Positive sider***:
  - God tidskompleksitet: proportional med mængden af levende dataobjekter.
  - Naturlig integration med "compaction": fører ikke til fragmentering.
- ***Negative sider***:
  - Kun halvdelen af lageret kan udnyttes effektivt.
  - Flytning af data kan være farlig, hvis pointere ikke kan identificeres 100% sikkert.

PS4 - Garbage collection

Noter

Teknikken, som beskrives på denne side, er alment kendt som "stop & copy" garbage collection.

Scenarium af kopierende garbage collection (1).

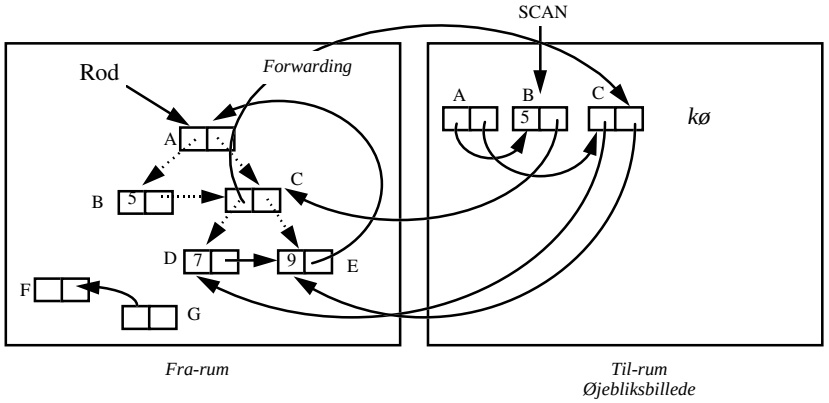


Udgangspunktet.

Noter

Scenariet, som starter her og går over de næste 2 sider, findes i en noget lignende form også i artiklen. Det er figur 5, som dog ikke viser referencerne fra to-space til from-space, og der vises heller ikke forward pointere.

Scenarium af kopierende garbage collection (2).



Referencer, hvis rolle er udspillet.

Forwarding pointer

Ideen i Cheneys algoritme:

Rødderne udgør initielt en kø i til-rummet.
Iterativt: Fronten af køen scannes:
Objekter der refereres kopieres til til-rum.
Slut når SCAN når enden af køen.

Der er fare for, at C kopieres for 2. gang!

PS4 - Garbage collection

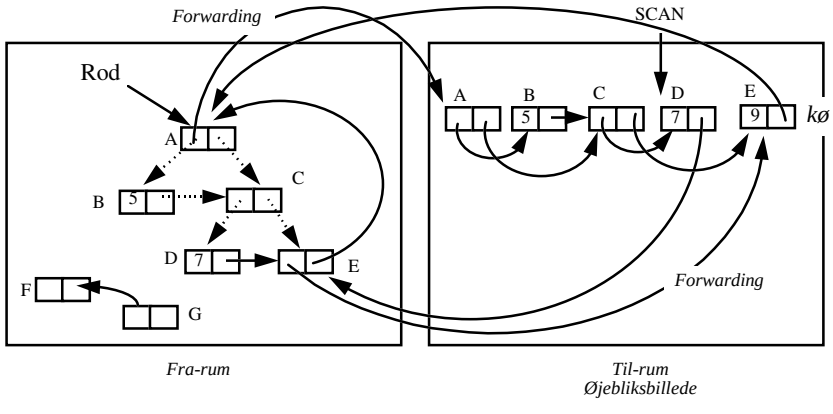
Noter

Hver gang vi kopierer et objekt fra fra-rummet til til-rummet etablerer vi en pointer fra det gamle objekt til det nye objekt. Lige inden vi går igang med at kopiere et objekt fra fra-rummet til til-rummet checker vi, om der eksisterer en forward pointer (som kan kendes på, at den går ind i til-rummet). Hvis en sådan findes, er det et tegn på, at objektet i fra-rummet ikke skal kopierers, idet det allerede er blevet kopieret én gang. Vi nøjes så blot med at justere en pointer i til-rummet.

Forward pointere kan frit etableres i de objekter, der er kopieret (f.eks. i CAR positionen). Årsagen er, at den oprindelige værdi af CAR positionen befinder sig i det kopierede objekt i til-rummet.

Bemærk, at vi ikke viser alle forward pointere på figurerne.

Scenarium af kopierende garbage collection (3).



Situation:
- Scanning tæt på at være færdig.
- Der bliver ikke kopieret flere objekter over i til-rummet.
- F og G er altså spild.

PS4 - Garbage collection

Noter

Der vises kun forwarding pointere for A og E.

Kopierende garbage collection: Algoritme skitse (1).

```
Copy(To-space-root: Cell);
var Q: Queue[Cell];
    N, M: Cell;
begin
 M := copy-cell(To-space-root);
 enter(Q,M)
 while not empty(Q)
 do
    let N be leave(Q)
    in begin
         if Ref?(car(N))
         then let M be copy-cell(car(N))
                 in set-car(N,M);
                    enter(Q,M)
         end;
         -- ditto for cdr(N):
          if Ref?(cdr(N))
          then let M be copy-cell(cdr(N))
                  in set-cdr(N,M);
                     enter(Q,M)
          end
       end
  end
end
```

```
copy-cell(from-node: cell);
var c: cell
begin
 c := alloker celle i to-space;
 set-car(c, car(from-node));
 set-cdr(c, cdr(from-node));
 set-car(from-node, nil);
 set-cdr(from-node, nil)
 returner c
end
```

PS4 - Garbage collection

Noter

Ovenfor viser vi en simpel skitse af algoritmen, der kopierer from-space til to-space. Q er en FIFO kø med enter og leave operationer. Cell typen skal opfattes som binære celler, ala Lisp cons celler. Routinen copy-cell foretager kopieringen af én celle (incl. etablering af referencer i den kopierede celle). Ref?(X) returnerer hvorvidt X er en pointer. Set-car og Set-cdr muterer hhv. det første og det andet felt i en celle.

Algoritmen er simpel og utilstrækkelig derved, at den ikke tager hensyn til faren for gentagen kopiering af den samme celle fra from-space til to-space. Ovenstående skitse er altså ikke bekymret om forward references, som omtalt på de forrige sider.

På næste side viser vi en skitse af en algoritme, som også tager hensyn til forward referencer.

Kopierende garbage collection: Algoritme skitse (2).

```
Copy(To-space-root: Cell);
var Q: Queue[Cell];
    N, M: Cell;
begin
 M := copy-cell(To-space-root);
 enter(Q,M)
 while not empty(Q)
 do
    let N be leave(Q)
    in begin
          if Ref?(car(N))
          then if forwarding-in(car(N))
                then set-car(N, forwarding-pointer(car(N)))
                else let M be copy-cell(car(N))
                     in set-car(N,M);
                        enter(Q,M)
                end
          end;

          -- ditto for cdr(N). Udeladt her.

       end
  end
end
```

```
copy_cell(from-node);
begin
 c := alloker celle i to-space;
 set-car(c, car(from-node));
 set-cdr(c, cdr(from-node));
 set-car(from-node, c);
                    -- forward pointer
                    -- etableres i from-node
 set-cdr(from-node, nil)
end
```

PS4 - Garbage collection

Noter

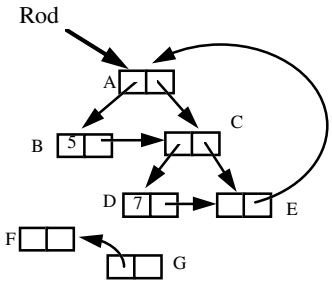
Her vises en udgave af algoritmen for kopierende garbage collection, som tager højde for den situation, hvor en celle refereres fra mere end ét sted. I denne situation ville algoritmen på forrige side kopiere vedkommende celle to (eller flere) gange til to-space. Forward referencer fra from-space til to-space lader os detektere situationen. Bemærk hvordan forward pointere etableres i copy-cell.

Forwarding-in(X) returnerer hvorvidt X refererer til en celle, som har en forward pointer. I fald dette er tilfældet vil Forwarding-pointer(X) returnere denne forward pointer (som jo går tilbage til to-space).

Det er ikke svært at se på en pointer, om det er en forward reference. En sådan krydser nemlig grænsen mellem de to rum (udgår fra, f.eks. det “lave rum” og går til det “høje rum”).

Baker's teknik: 'Non-copying implict collection'.

- **Et alternativ til kopierende garbage collection:**
  - Tilføj to nye, skjulte felter til hvert objekt.
  - Organiser objekterne i dobbeltkædede lister.
  - Tilføj et farve-felt til hvert objekt.
- Vurdering:
  - Fra- og til-rum ikke nødvendige.
  - Kræver overhead til dobbeltkædede lister.
  - Objekter flyttes aldrig:
    - Vil forårsage fragmentering.
    - Sprog-niveau pointere ændres aldrig.



- ***Under normal kørsel*:**
  - Fri-liste af celler
  - En liste af allokerede celler.
- ***Under garbage collection*** (fri-listen er tom):
  - Den hvide liste: allokerede, ikke sete objekter.
  - Den grå liste: Objekter som er under behandling.
  - Den sorte liste: Færdig behandlede objekter.

Noter

Bekrivelsen på denne slide svarer til afsnit 2.5 i artiklen: Baker's non-copying implicit collection. Den er isomorf til kopierende garbage collection, men de to "spaces" håndteres anderledes: ved dobbeltkædede lister. Endvidere introduceres farver (sort, grå og hvid) af objekter.

De to spaces i kopierende indsamling halverer det effektive lagerrum. I teknikken, som vi her beskriver, er det ekstra lagerforbrug udgjort af cellerne, som bruges til den dobbelte sammenkædning. Hvis alle objekter er binære, ala CONS celler i Lisp, er de to teknikker pladsmæssigt lige gode. Hvis objekterne er større end CONS celler, er der plads at spare ved at bruge dobbeltkædning.

Garbage collection forløber som følger. Rodsættet er givet, og placeres til en start i den grå liste. Den sorte liste er tom. Den hvide liste indeholder alle allokerede objekter.

Vi tager et objekt ud af den grå liste, og hægter den ind i den sorte liste. Alle hvide efterkommere af dette objekt sættes ind i den grå liste. Vi kan let opretholde en kø disciplin på den grå liste, hvilket befordrer et bredde-først gennemløb. En stak disciplin befordrer et dybde-først gennemløb

Når den grå liste bliver tom, er garbage collection slut.

Alle objekter i den hvide liste er nu spild, og disse deallokeres. Når normal udførsel fortsættes, bliver denne liste fri-listen.

Alle objekter i den sorte liste er data, som skal beholdes. Når normal udførelse fortsættes, bliver denne liste hvid (listen af allokerede celler).

Under forelæsningen vil vi illustrere farveændringen i ovenstående objektnetværk.

Initielt:

Hvid liste: B C D E F G

Grå liste: A

Sort liste: