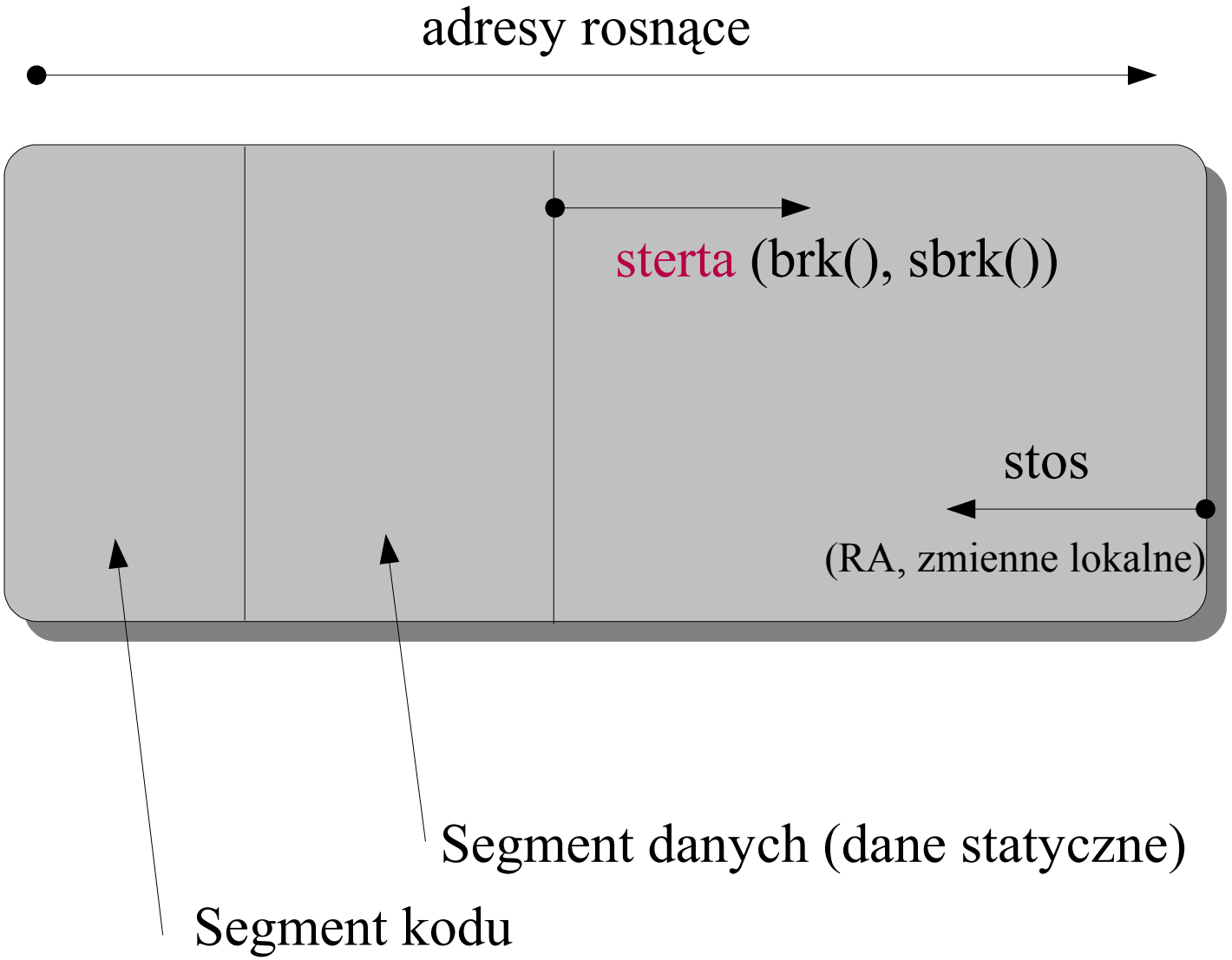
Mechanizmy odśmiecania w językach obiektowych

Artur Popławski

Referat na seminarium magisterskie
Zagadnienia Programowania Obiektowego

Wydział Matematyki, Informatyki i Mechaniki
Uniwersytet Warszawski

Przypomnienie: organizacja pamięci



Alokacja pamięci

```
static int i;

void foo(void)

{

    long j;

    char ptr = (char*) malloc(10);

    ...

}
```

Alokacja statyczna

```
static int i;
void foo(void)
{
    long j;
    char ptr = (char*) malloc(10);
    ...
}
```

- alokacja przez kompilator
- obiekt dostępny przez cały czas działania programu
- stały rozmiar

Alokacja automatyczna

```
static int i;
void foo(void)
{
    long j;
    char ptr = (char*) malloc(10);
    ...
}
```

- alokacja na stosie podczas wołania procedury
- dostępna przez czas działania procedury
- stały rozmiar

Alokacja dynamiczna

```
static int i;
void foo(void)
{
    long j;
    char ptr = (char*) malloc(10);
    ...
}
```

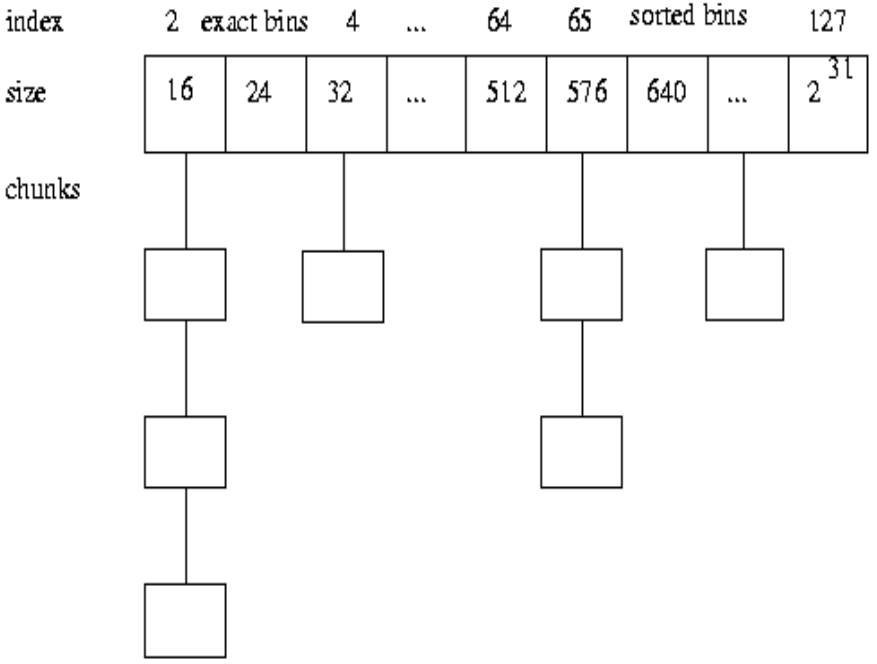
- alokacja na stercie podczas działania programu
- dostępna, aż do dealokacji
- możliwa modyfikacja rozmiaru

Manualne zarządzanie stertą

- Programista odpowiedzialny za alokowanie bloków pamięci
  - malloc( ) - funkcja biblioteki libc
  - new( )
- Programista odpowidzialny za zwolnienie zaalokowanych bloków pamięci
- najczęściej implementacja to odmiana algorytmów operujących na listach: first-fit, worst-fit, next-fit...

Doug Lea malloc

- na obu końcach informacja o rozmiarze i stanie
  - można łączyć, i przechodzić w obu kierunkach
- podobne do systemu bloków bliźniaczych
- szukanie bloku alg. best-first z łączeniem bloków sąsiednich podczas zwalniania



Problemy z manualnym zarządzaniem pamięcią

- wiszący wskaźnik:

```
char *ptr=(char*)malloc(3);
ptr1 = ptr;
...
free(ptr1);
...
ptr[5] = 'x';
```

- wyciek pamięci:

```
char *a=malloc(4);
...
a = malloc(4);
...
free(a);
```

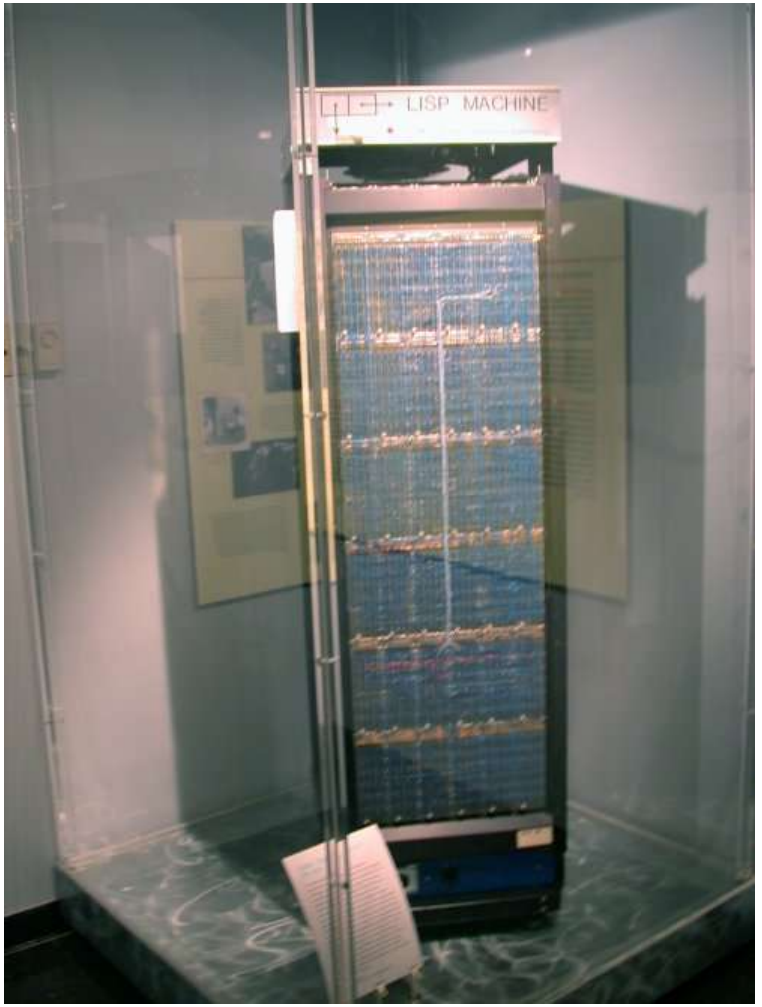
Problemy z manualnym zarządzaniem pamięcią (2)

- problemy z pamięcią są trudno debugowalne
- tworzenie zależności między modułami
- często trzeba ręcznie implementować niby-GC w dużym systemie
- uproszczenie API (kto zwalnia pamięć?)
- produktywność programisty

Trochę historii...

- początki GC wiążą się z Lispem i maszynami lispowymi

- Pierwszy GC powstał w 1958 roku, napisany przez Johna McCarthy'ego jako fragment implementacji Lispa



Podstawowe pojęcia

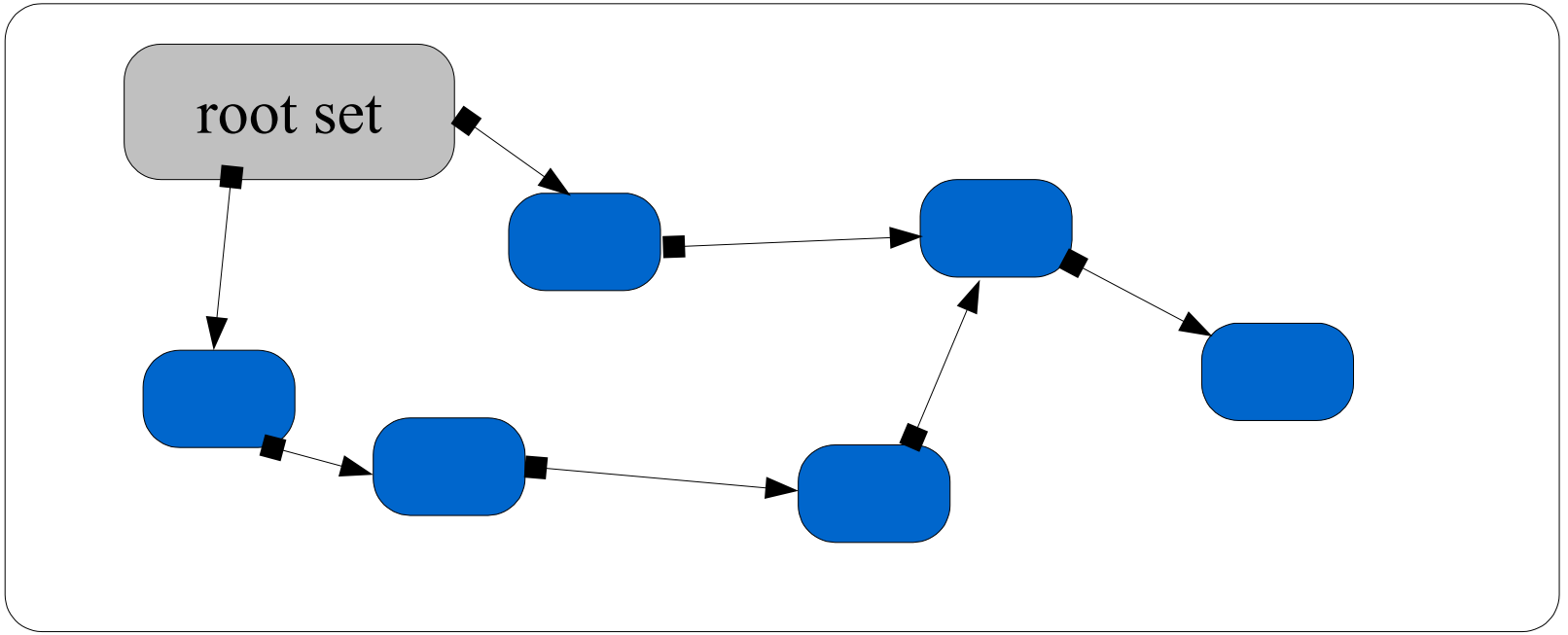
- **obiekt** – ciągły fragment pamięci, niekoniecznie obiekt w rozumieniu OOP!
- **referencja** – link do innego obiektu, wskaźnik
- **obiekt martwy** (śmieć) – obiekt, który już nie zostanie wykorzystany w uruchomionym programie
- **odśmiecanie** – automatyczne odzyskiwanie dynamicznie zaalokowanych zasobów
- **osiągalność** – obiekt jest osiągalny, jeśli istnieje łańcuch referencji prowadzący do niego

root set – zbiór domyślnie żywych obiektów w programie

W jego skład wchodzą:

1. rejestry procesora
2. zmienne statyczne
3. zmienne lokalne na stosie

Przykład osiągalności obiektów:



Dwie fazy odśmiecania

Niezależnie od implementacji możemy wyróżnić dwie fazy działania GC:

1. identyfikacja nieosiągalnych przez program obiektów w pamięci
2. odzyskanie pamięci

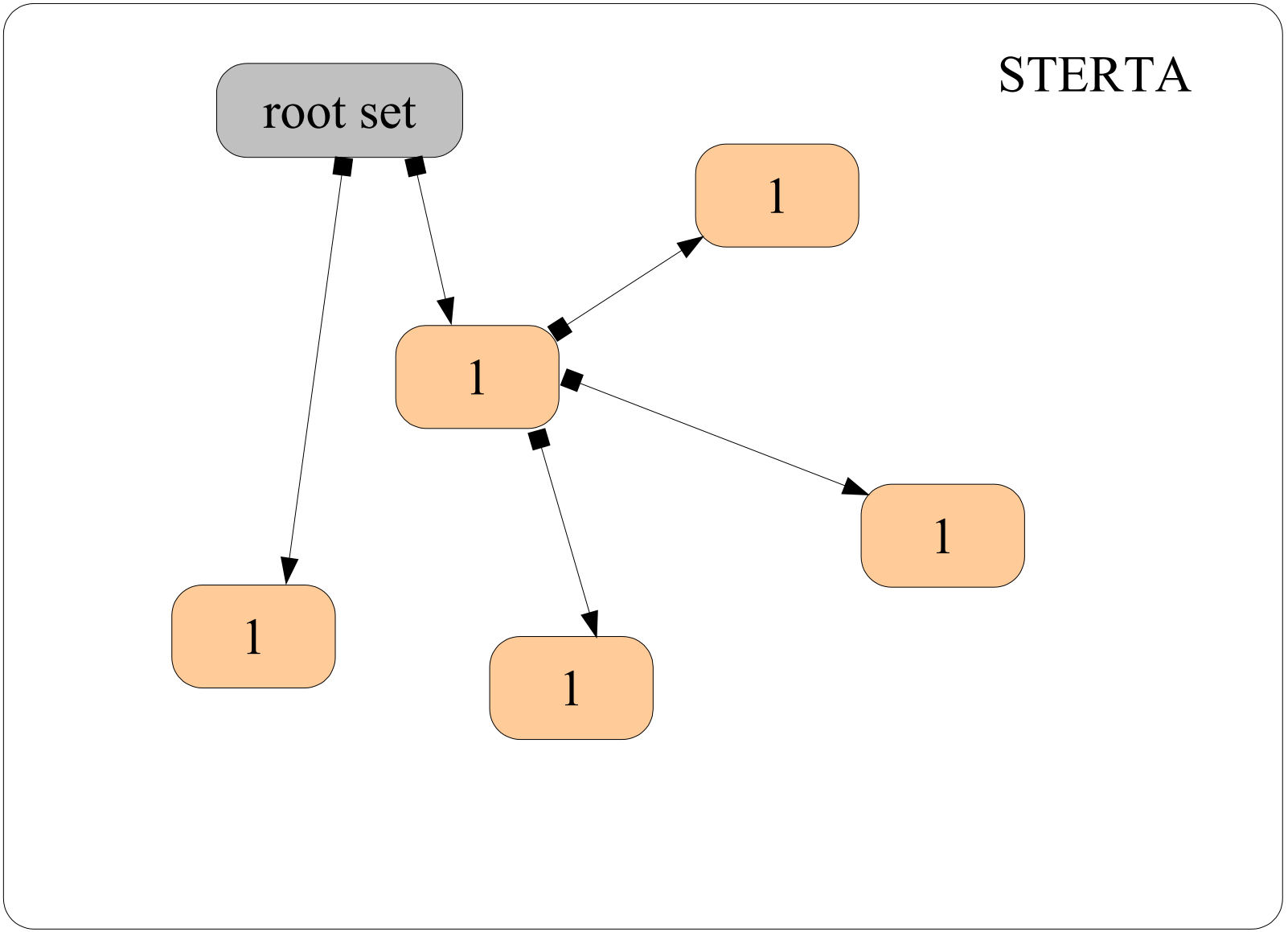
- w praktyce, te dwie fazy mogą się przenikać
- zakładamy, że obiekty są samoidentyfikujące się

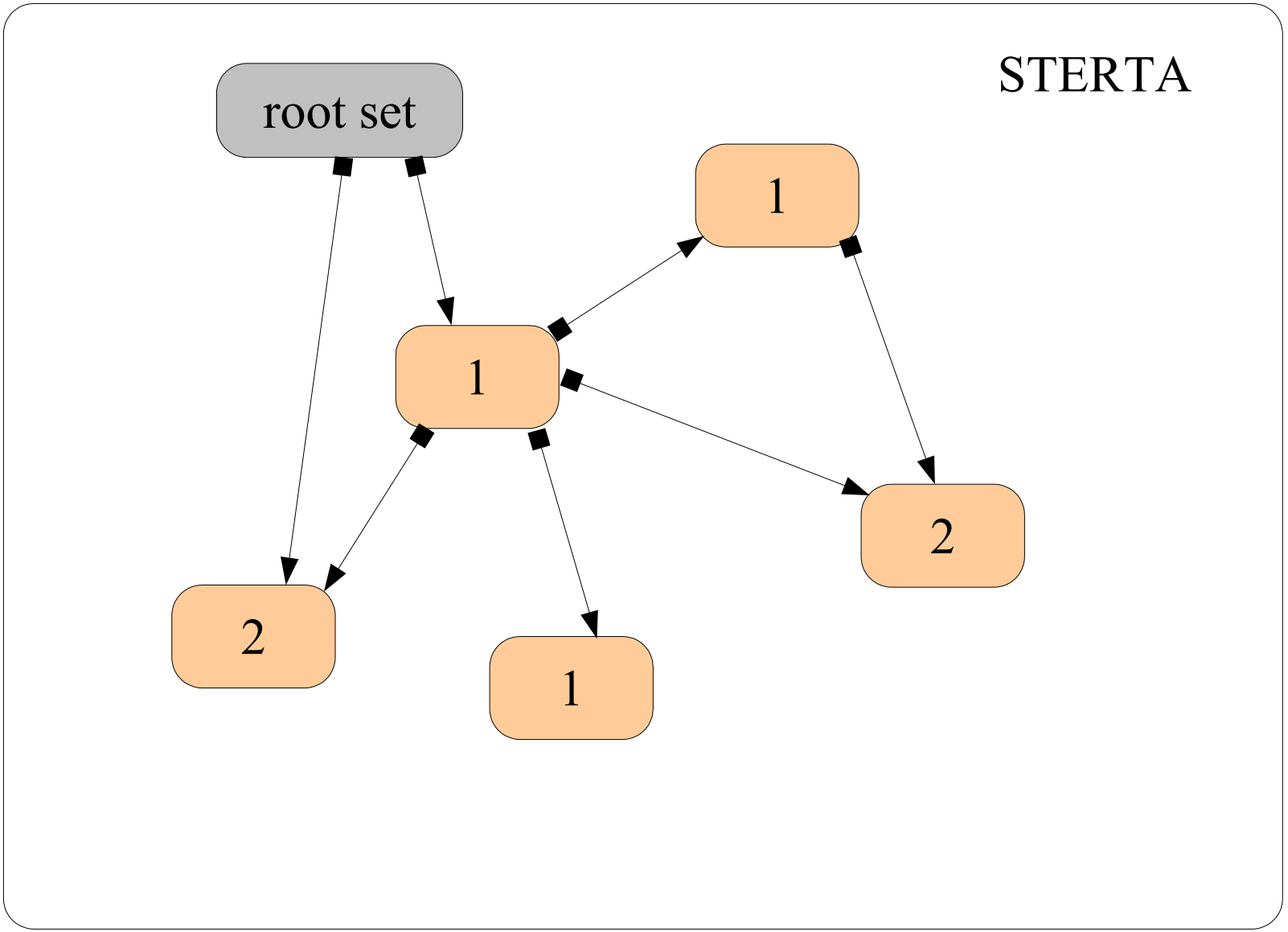
Podstawowe algorytmy

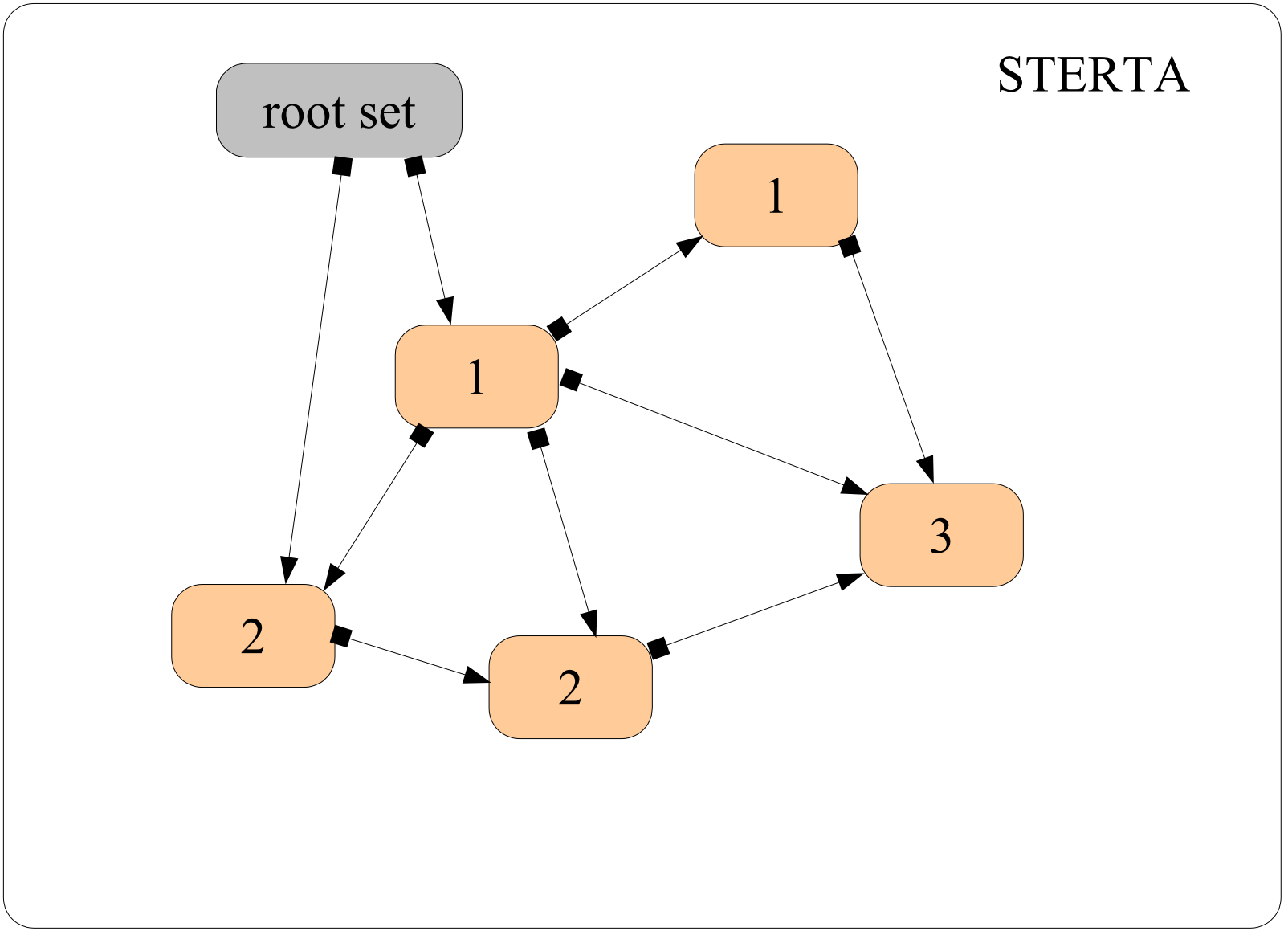
1. Zliczanie referencji

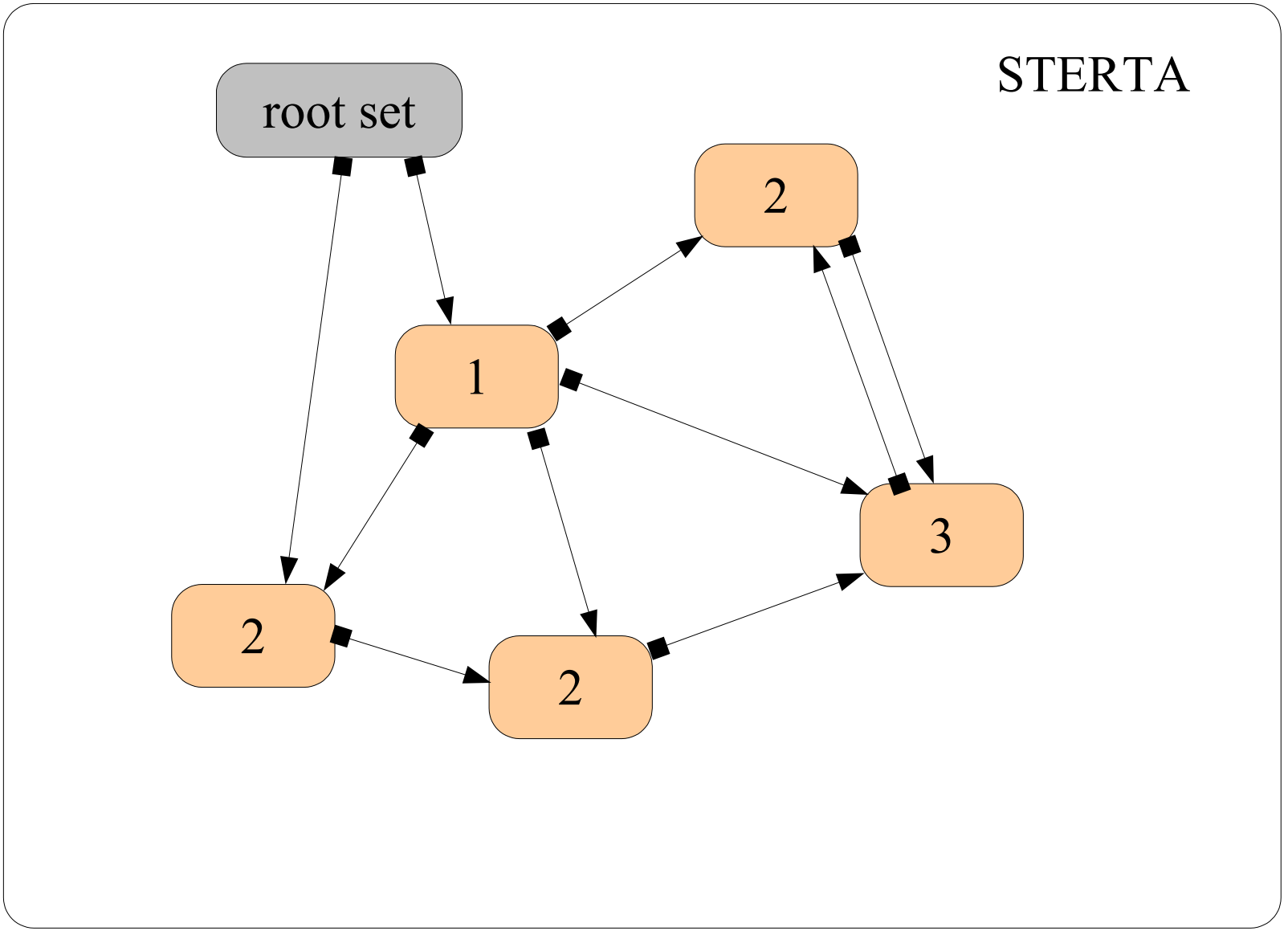
Najprostszy pomysł z możliwych:

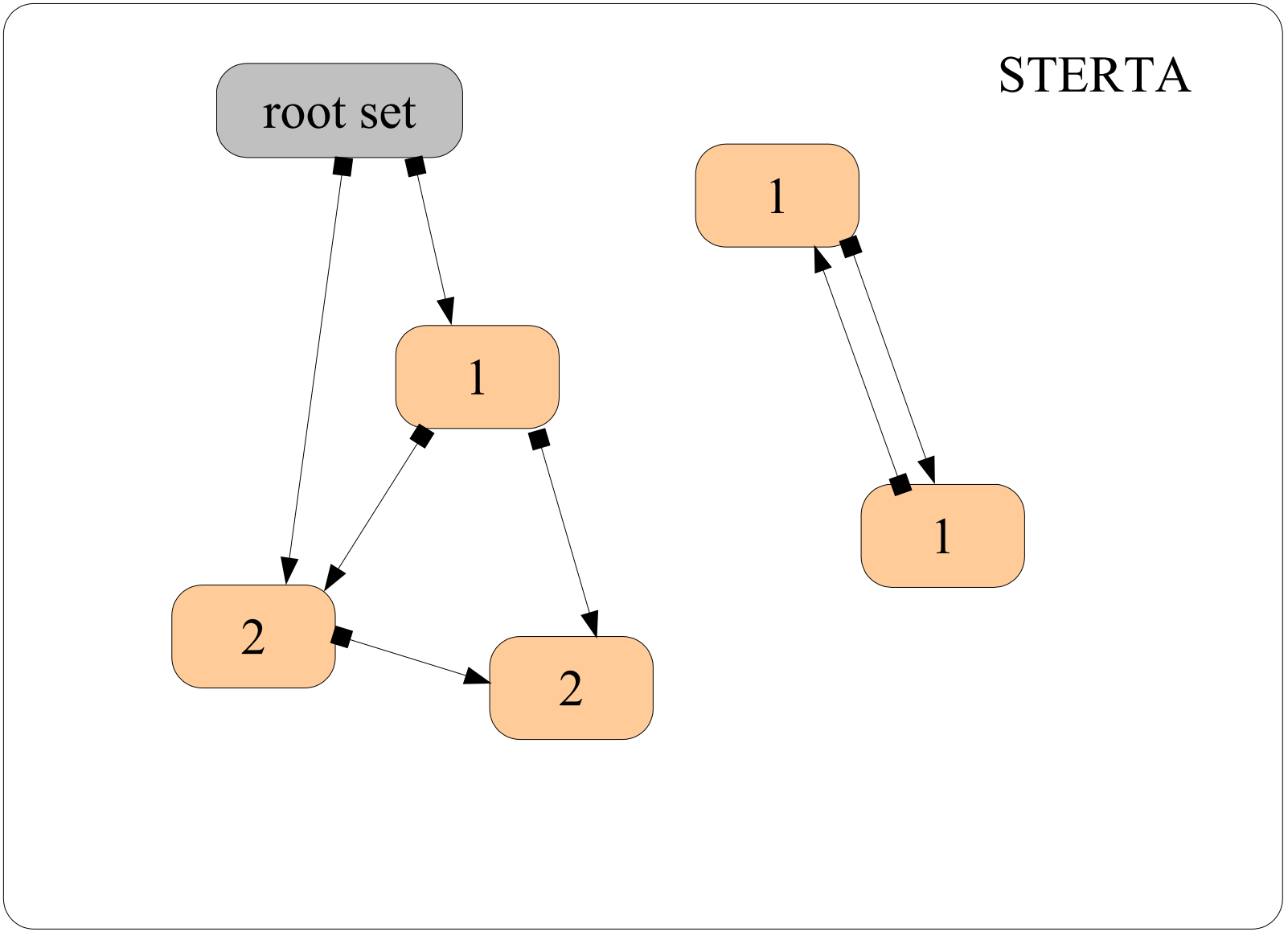
- każdy obiekt w pamięci ma przypisany licznik odniesień
- kiedy tworzymy nową referencję do obiektu, zwiększamy także licznik
- analogicznie, podczas eliminowania referencji zmniejszamy także licznik
- dealokacja możliwa, kiedy licznik odniesień wynosi zero
- podczas drugiej fazy uaktualniamy liczniki obiektów do których prowadzą referencje z odzyskiwanego obiektu

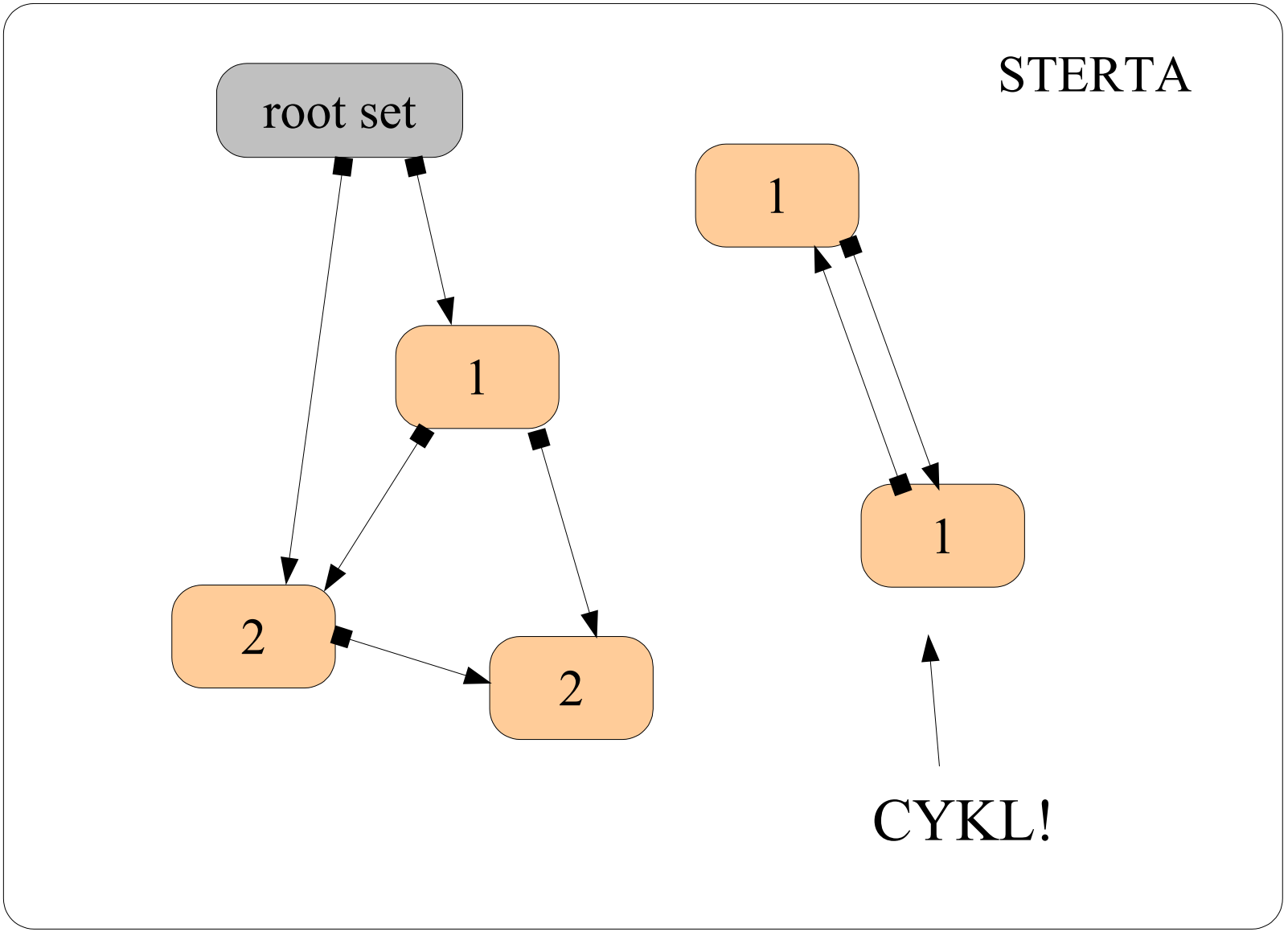

STERTA
root set
1
1
1
1
1


STERTA
root set
1
1
2
2
1


STERTA
root set
1
1
3
2
2


STERTA
root set
2
1
3
2
2


STERTA
root set
1
2
2
1
1


STERTA
root set
1
2
2
1
1
CYKL!

- powstawanie cykli wymusza stosowanie dodatkowej techniki odśmiecania
- problem z wydajnością
  - uaktualnianie liczników
  - wydajność proporcjonalna do ilości pracy programu
  - zmienne na stosie i uaktualnianie licznika
  - większy koszt pamięciowy
- możliwość przyrostowego odśmiecania
- prostota implementacji
- problemy z fragmentacją

Zliczanie referencji – wariacje

- deferred reference counting
  - nie liczymy referencji z obiektów lokalnych
  - Zero Count Table
- one-bit reference count
  - heurystyka
  - wymaga zastosowania innych technik GC

Popularność tej metody GC we współczesnych językach skryptowych

Algorytmy śledzące

Mark-Sweep Collection

- pierwszy algorytm GC!

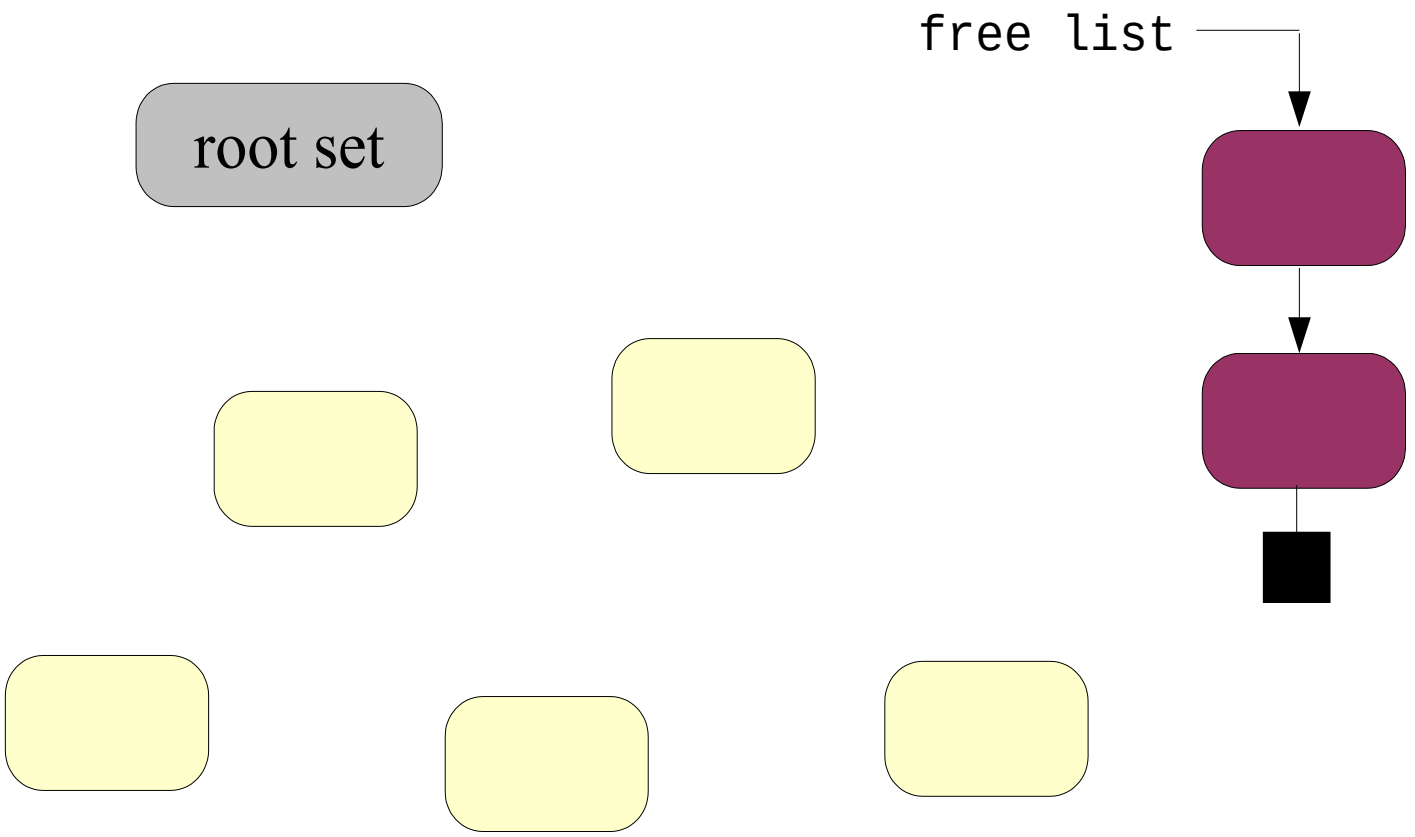
Pierwsza faza – markowanie obiektów

- rekursywnie przechodzimy graf referencji obiektów, zaczynając od root set
- obiekty osiągalne w jakiś sposób markujemy

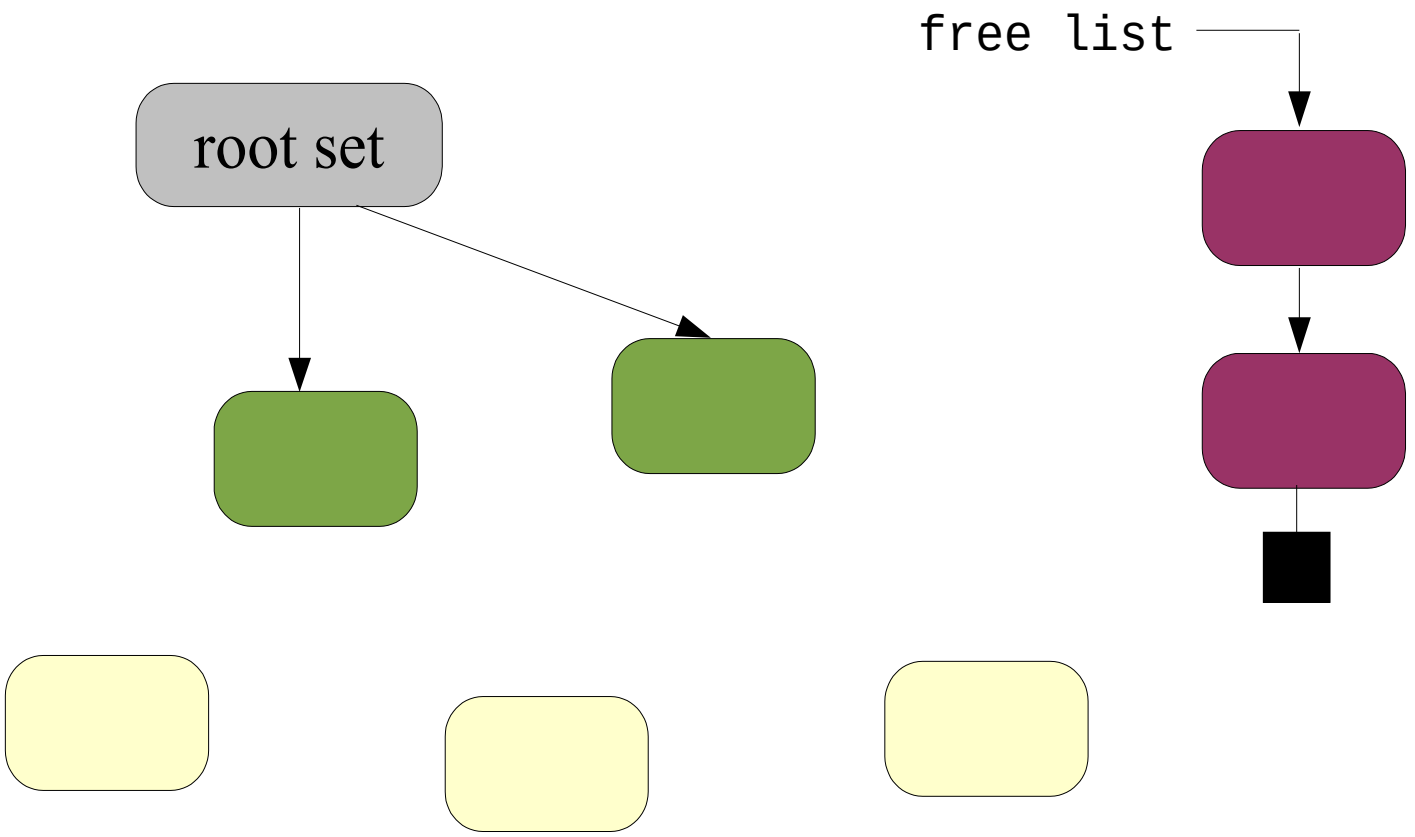
Druga faza – zamiatanie obiektów

- szukamy wszystkich niezaznaczonych obiektów w pamięci
- odzyskujemy pamięć dołączając obiekty do odpowiedniej listy

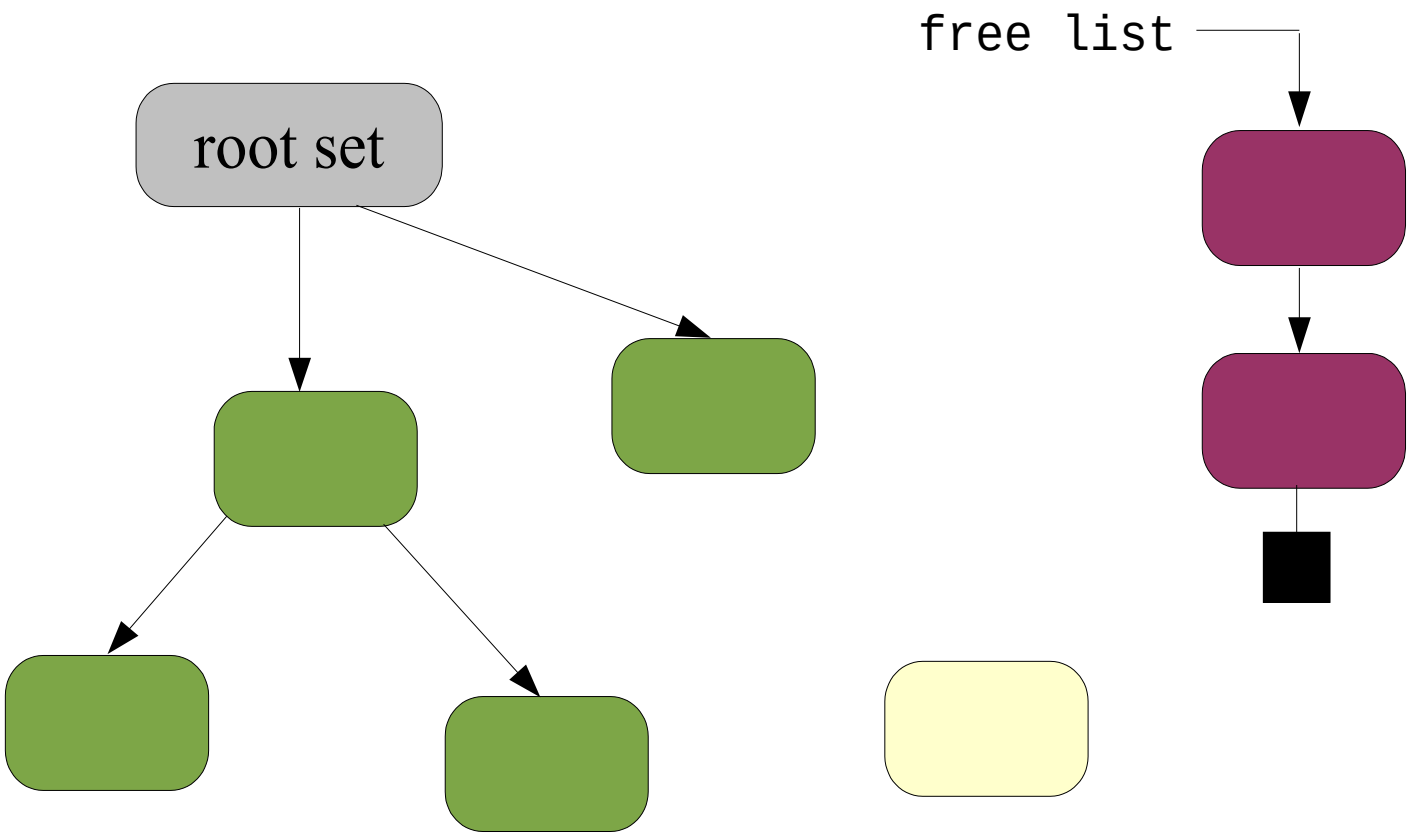
Faza pierwsza: markowanie



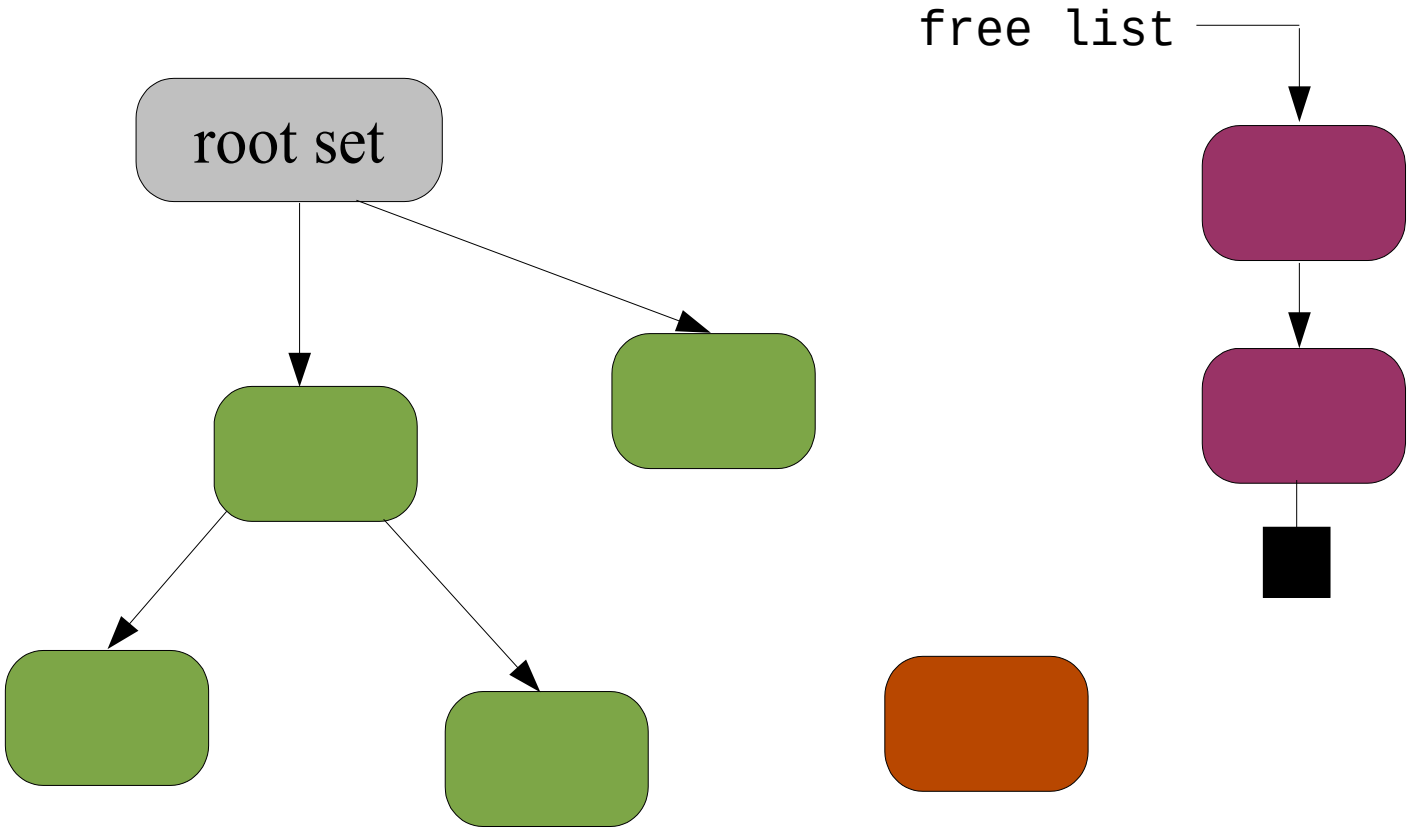
Faza pierwsza: markowanie



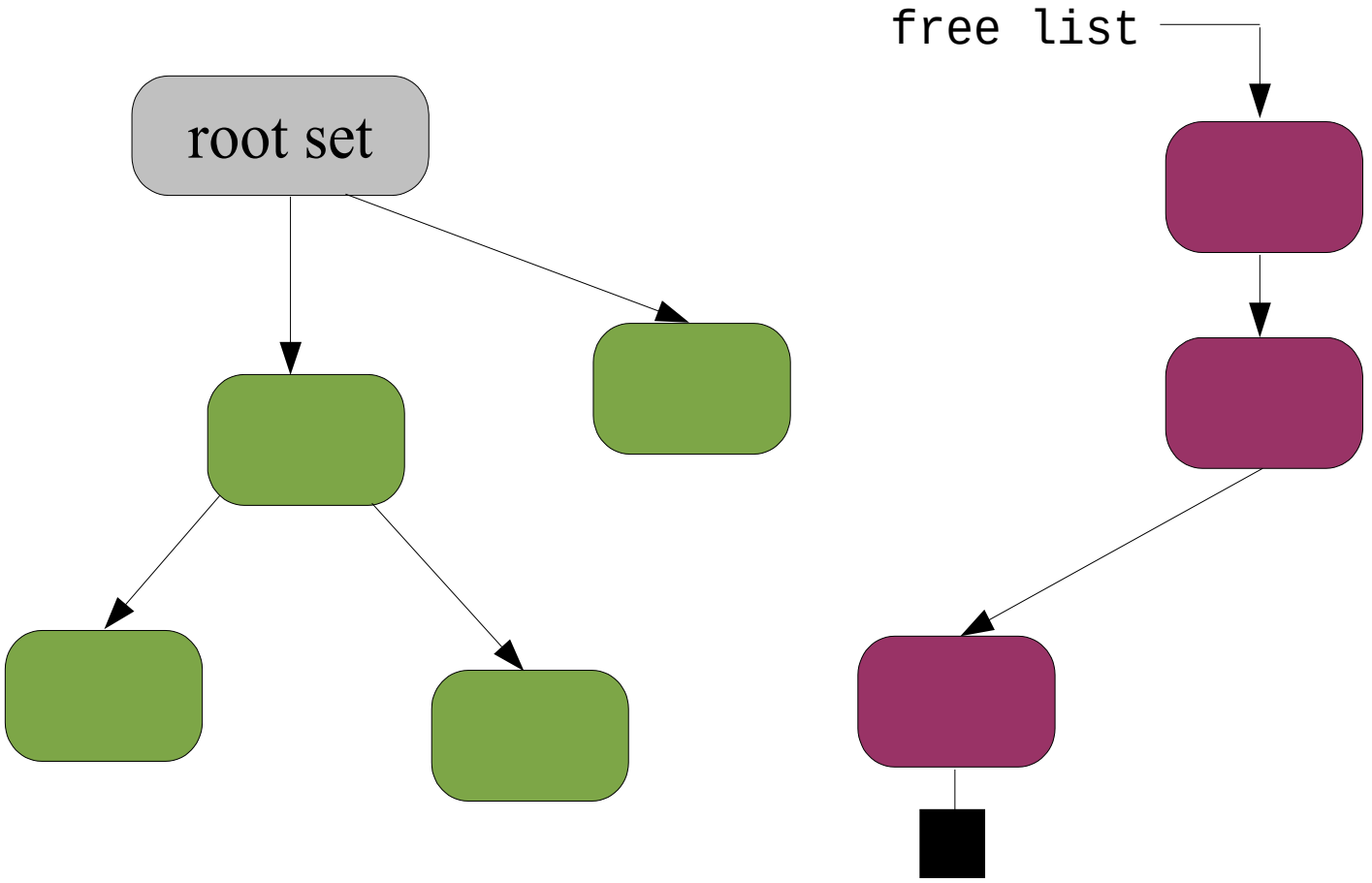
Faza pierwsza: markowanie



Faza druga: zamiatanie



Faza druga: zamiatanie



Złożoność

- koszt markowania: O(n), n – liczba osiągalnych obiektów
  - stała równa liczbie instrukcji przy każdym obrocie pętli podczas przechodzenia grafu
- koszt zamiatania: O(m), m – liczba wszystkich obiektów na stercie
  - stała j.w.
- uwaga na złożoność pamięciową rekursywnego algorytmu przechodzenia grafu!
  - algorytm Deutsch-Schorr-Waite

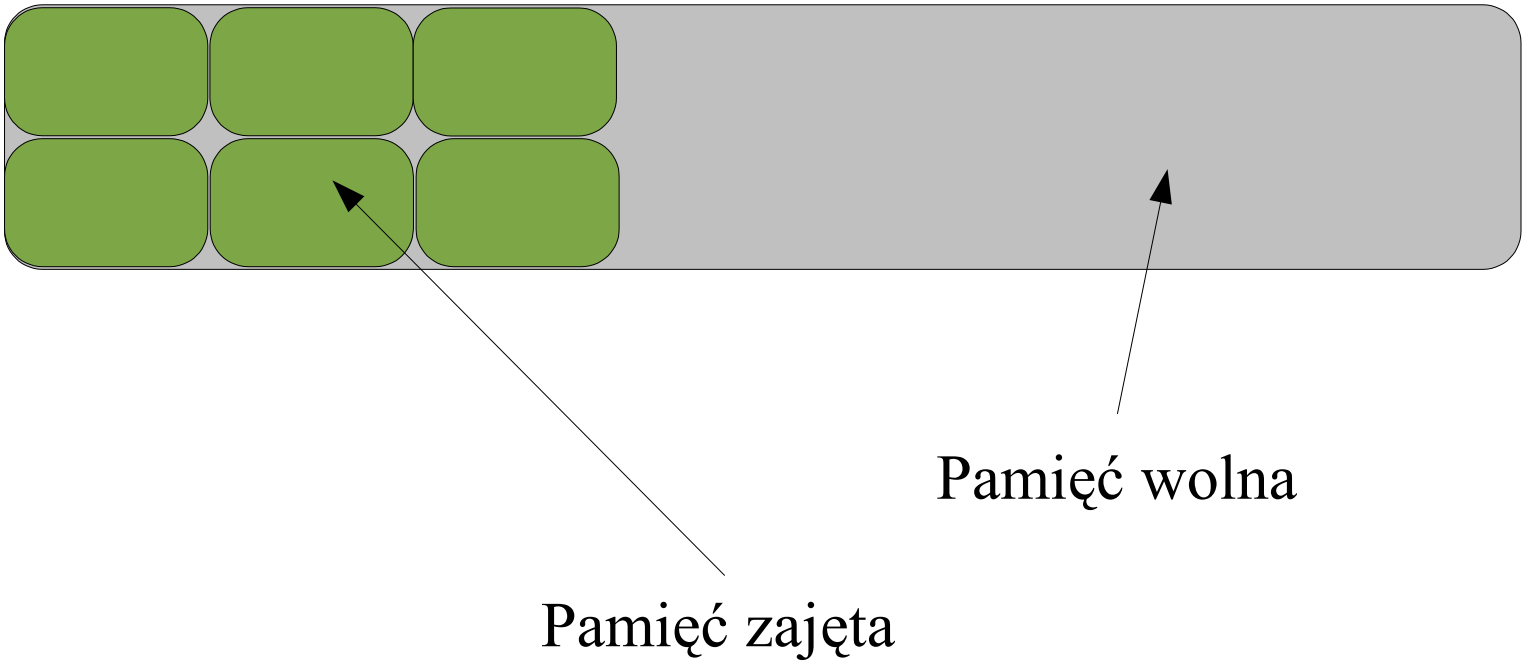
Plusy i minusy algorytmu mark-sweep

- Minusy
  - obiekty nie zmieniają adresów
  - problemy z fragmentacją
  - problemy z lokalnością odniesień
  - koszt proporcjonalny do rozmiaru sterty
- Plusy
  - można korzystać z trików algorytmów malloc/free
  - możliwa praca w środowisku współbieżnym
  - radzi sobie ze strukturami cyklicznymi

Algorytm mark-compact

- modyfikacja mark-sweep, mająca na celu poprawić problemy z fragmentacją i lokalnością odwołań
- pierwsza faza, markowanie, bez zmian
- druga faza – wszystkie żywe obiekty są upakowywane do ciągłej przestrzeni adresowej
- po drugiej fazie reszta pamięci jest wolna
- wskaźniki do skopiowanych obiektów muszą zostać uaktualnione

Po zadziałaniu algorytmu pamięć wygląda tak:



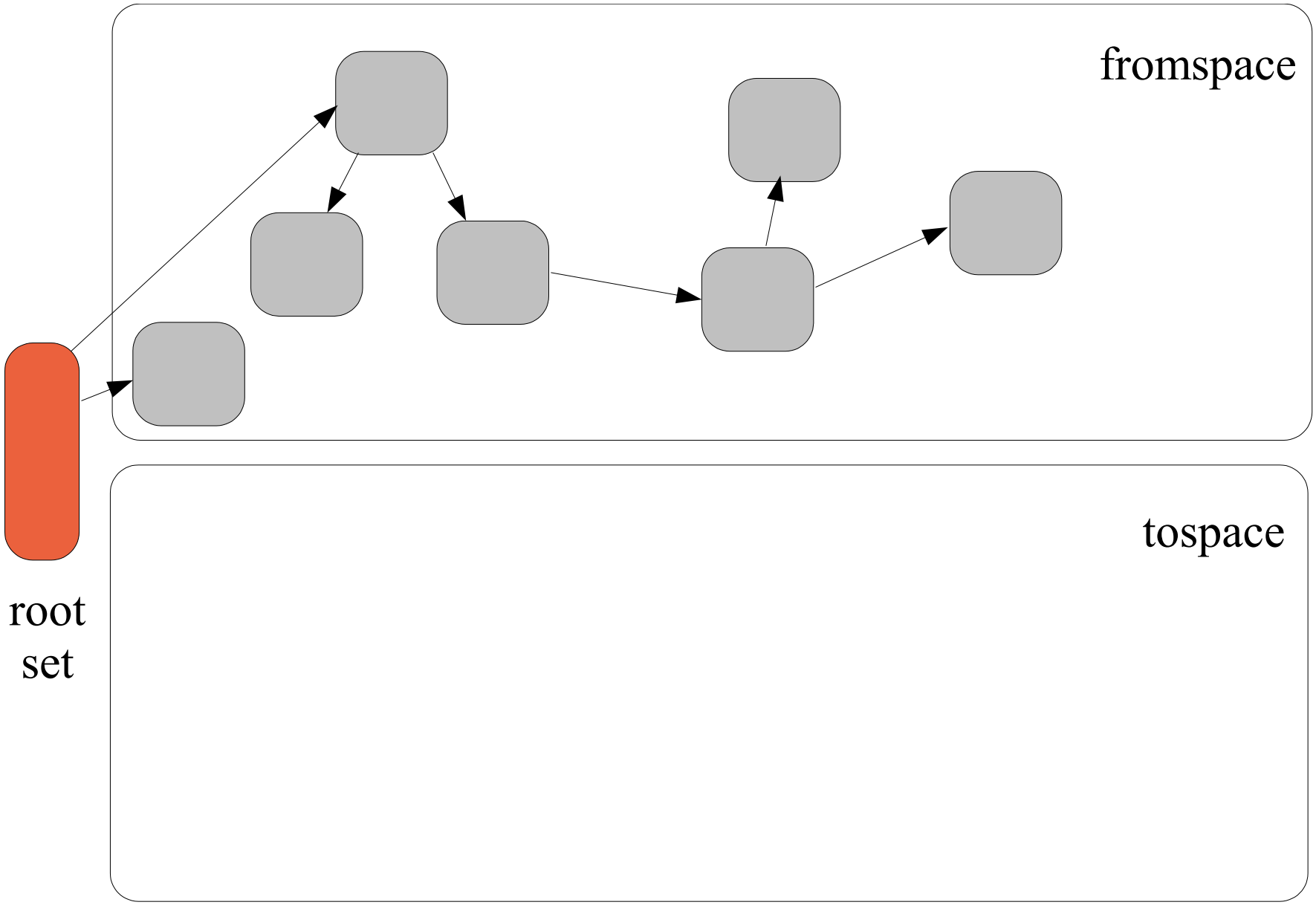
Uwagi do algorytmu mark-compact

- wyeliminowane problemy z fragmentacją
- prosta alokacja – wolna pamięć jest ciągła, więc alokacja podobna do alokacji na stosie
- efektem alokacji jest dobra lokalność obiektów
- niestety lokalność odwołań ciągle zła – faza upakowywania i uaktualniania referencji wymusza odwołania do rozrzuconych fragmentów pamięci
- faza kopiowania obiektów bywa bardzo kosztowna, wymaga trzech przejść po żywych obiektach
  - obliczenie nowych adresów
  - uaktualnienie referencji
  - przeniesienie obiektów

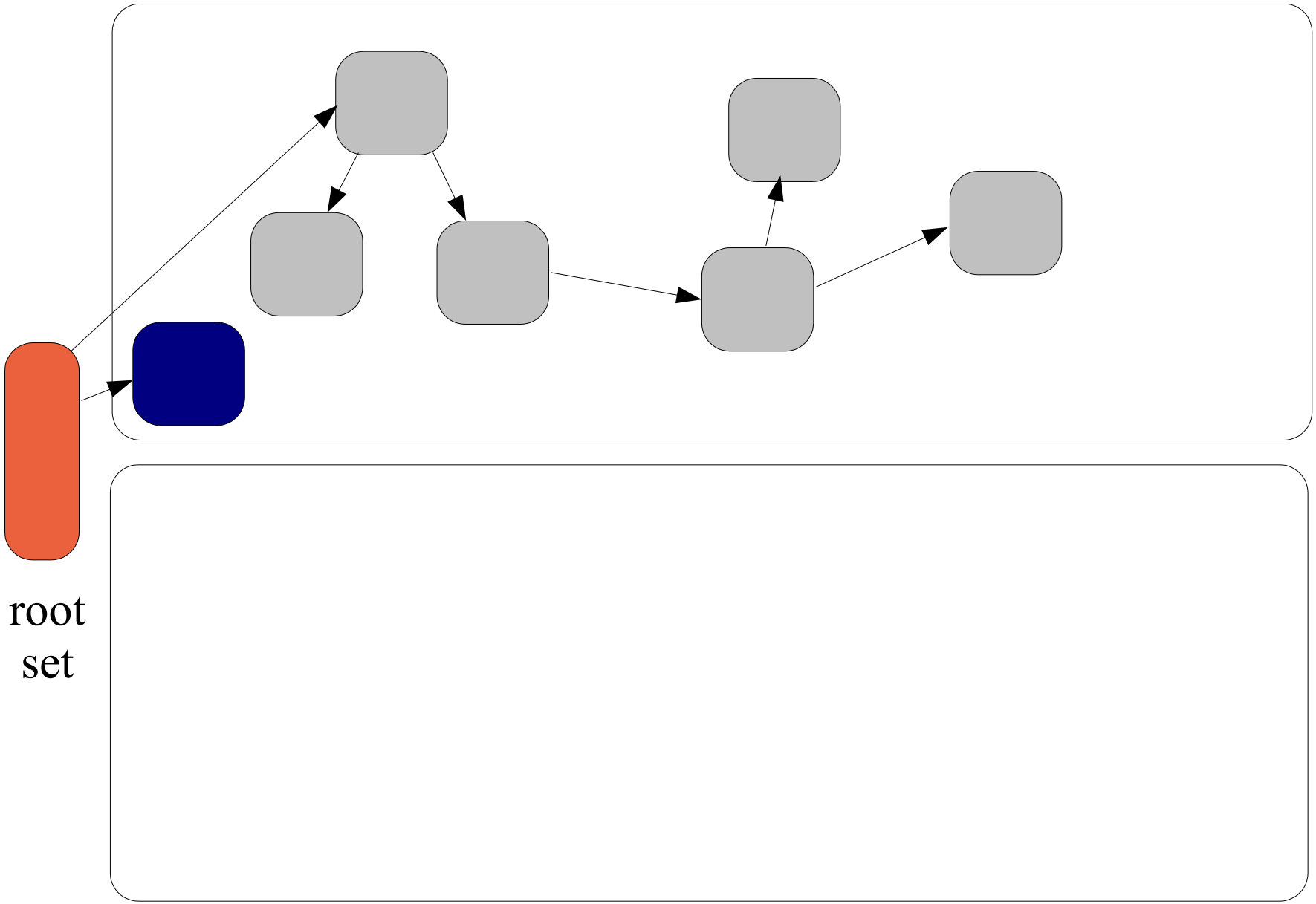
Algorytm mark-copy

- podobnie do mark-compact, kopiuje zajęte obiekty do ciągłego obszaru pamięci
- zajmiemy się odmianą algorytmu mark-copy, algorytmem Cheney'a, tzw. “semispace collector”
- stertę dzielimy na dwie przestrzenie, z których wykorzystujemy tylko jedną (na zmianę)
- pamięć alokujemy liniowo (jak w mark-compact)
- podczas działania GC markujemy obiekty żywe, po czym kopiujemy je z jednej przestrzeni do drugiej i uaktualniamy referencje
- ... i zamieniamy przestrzenie

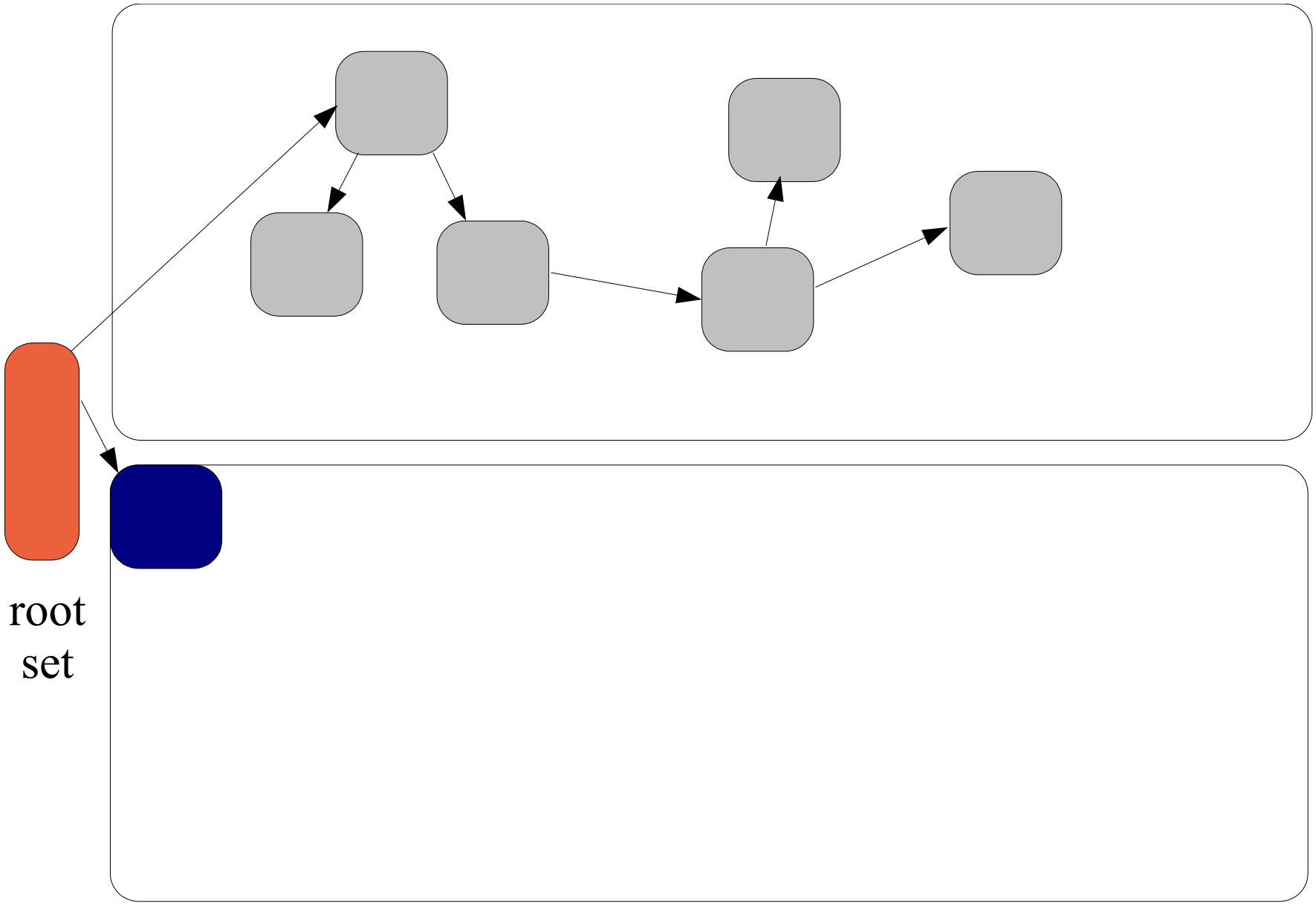
Przed odśmiecaniem...



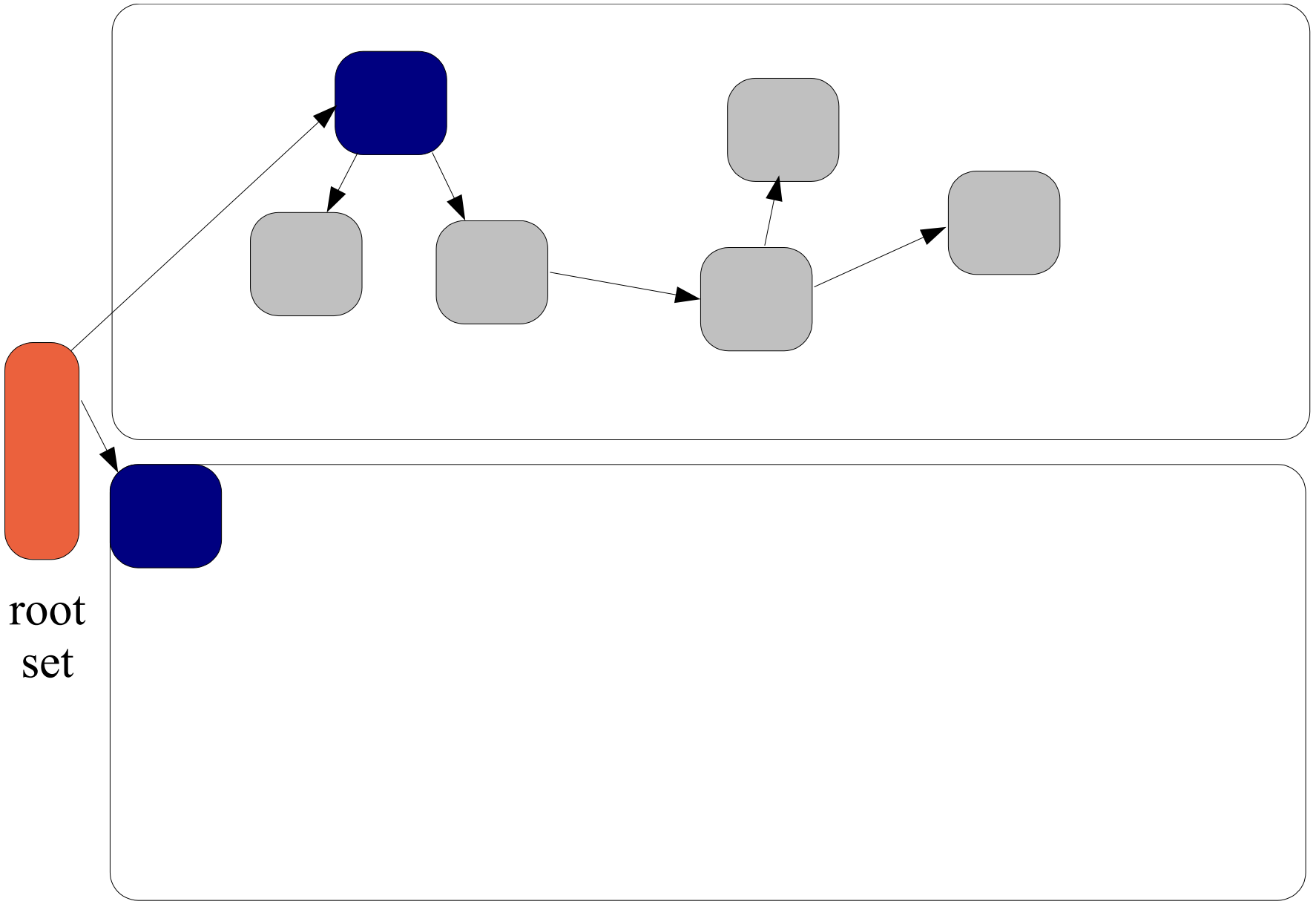
W trakcie odśmiecania...



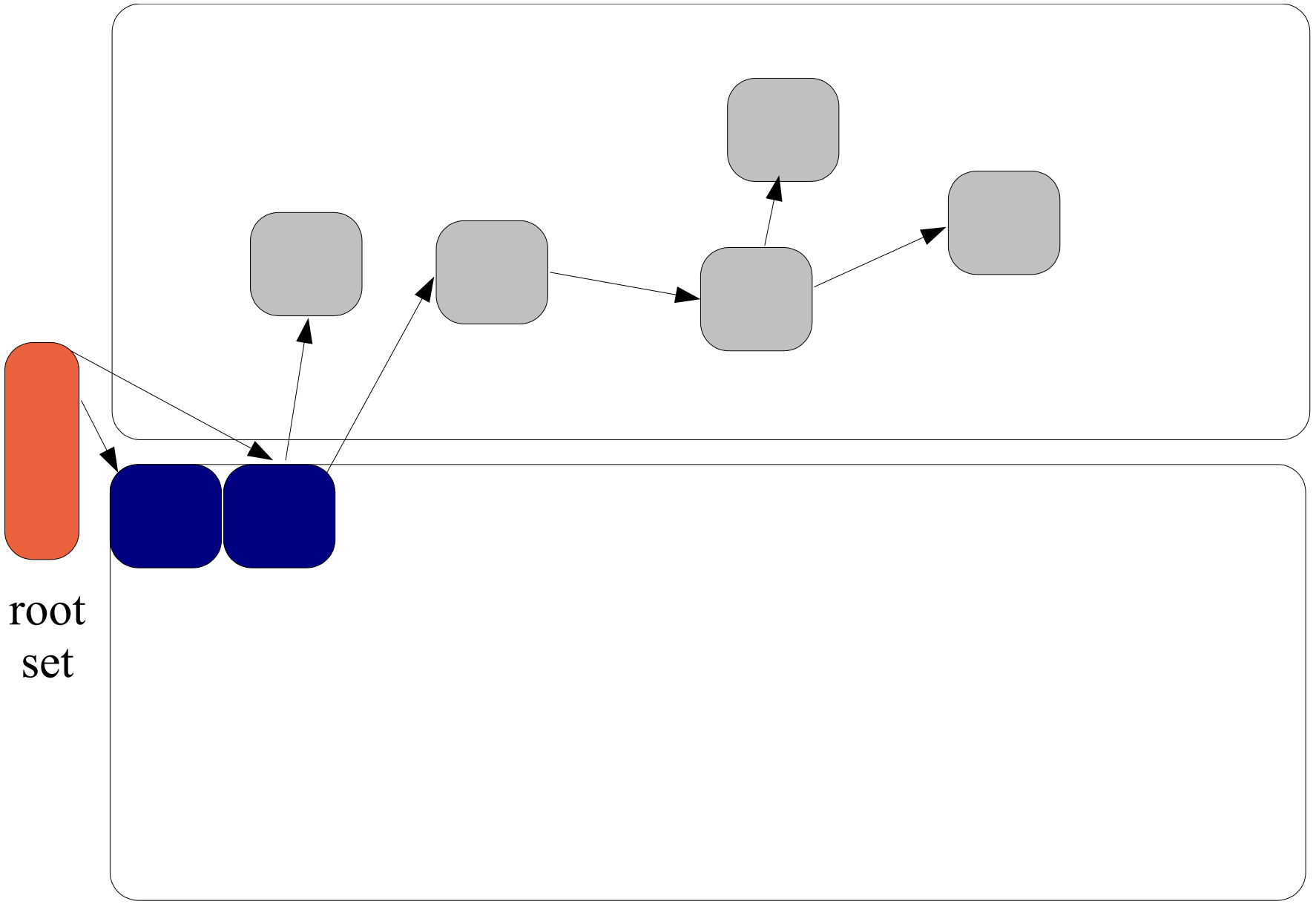
W trakcie odśmiecania...



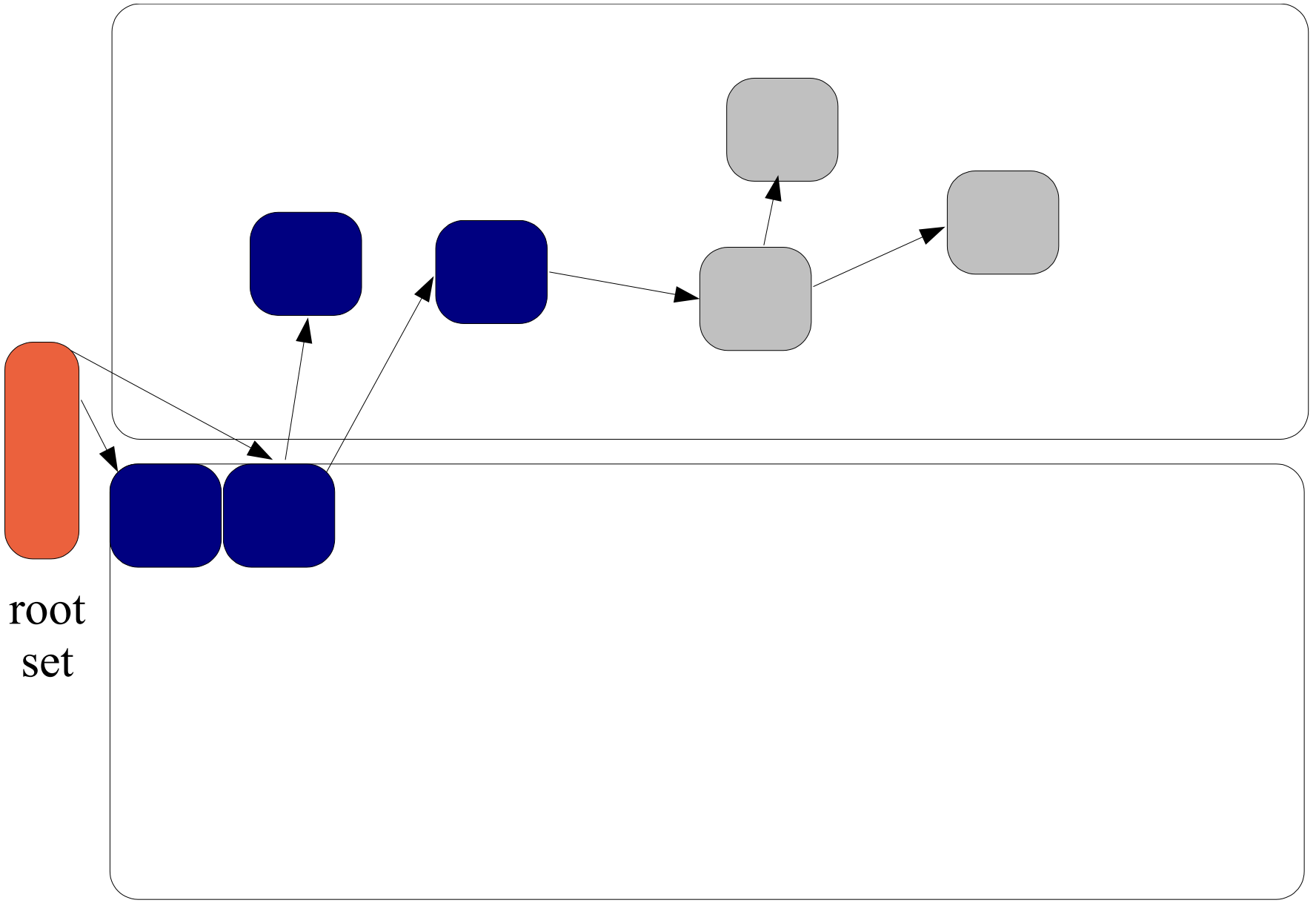
W trakcie odśmiecania...



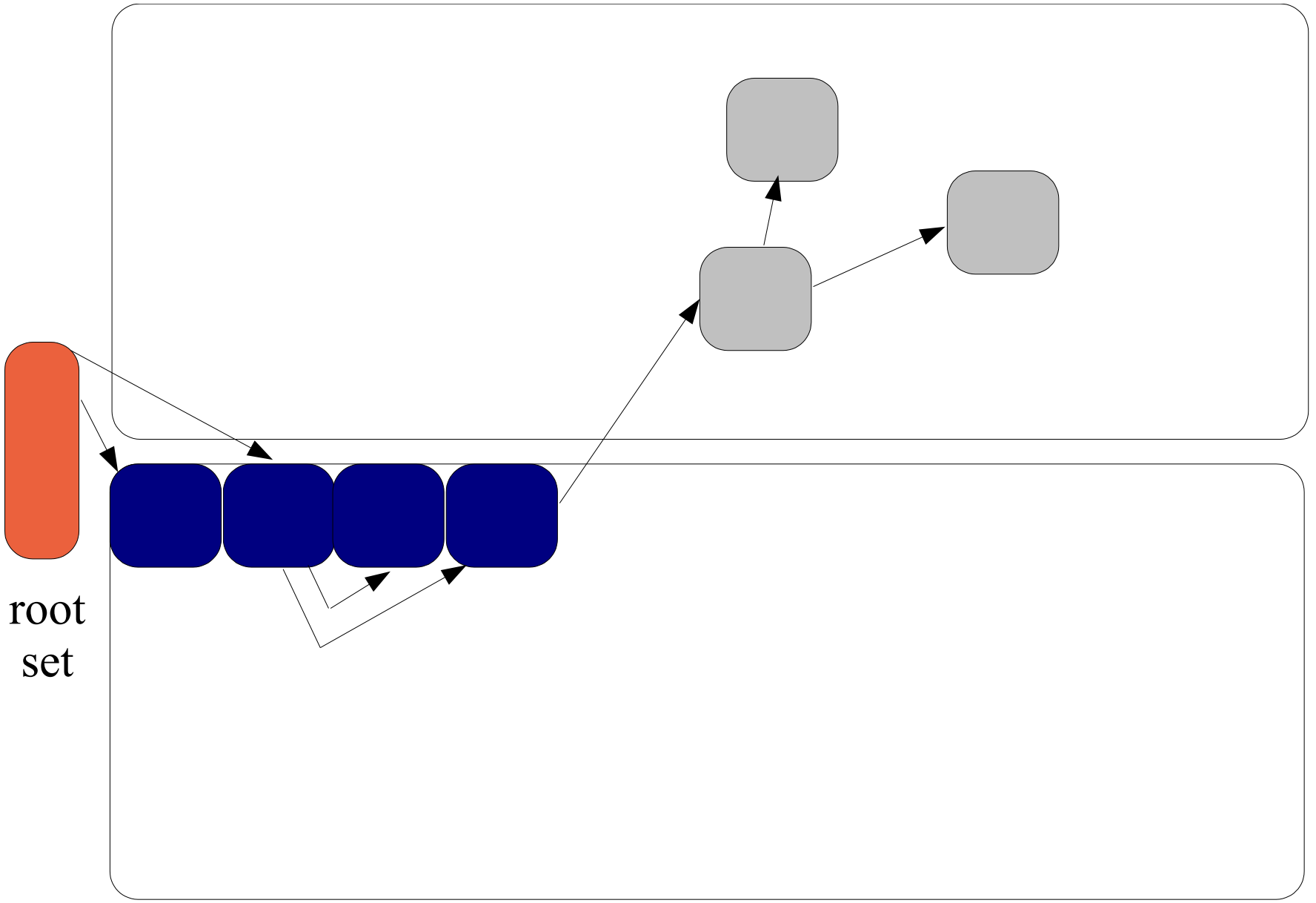
W trakcie odśmiecania...



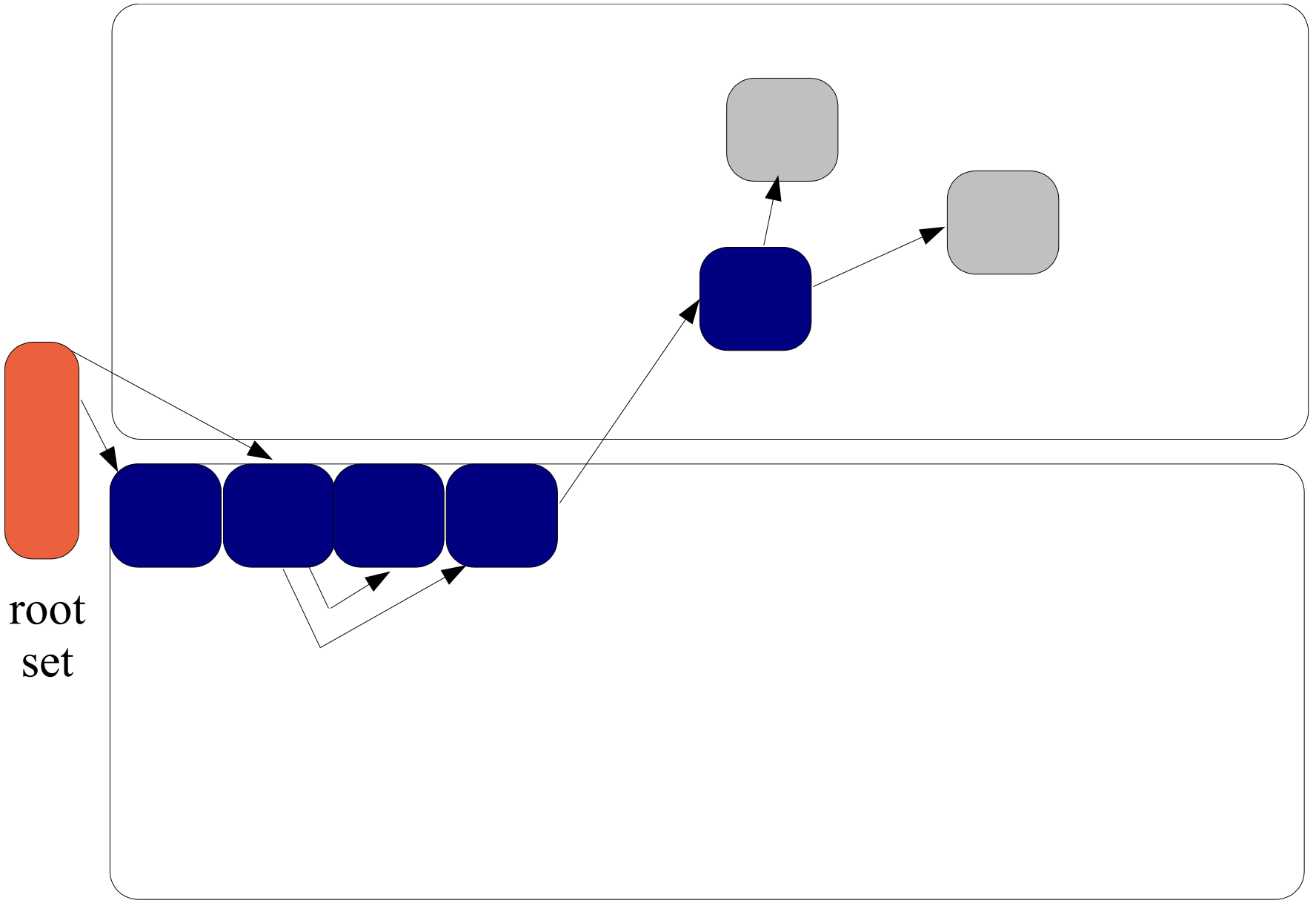
W trakcie odśmiecania...



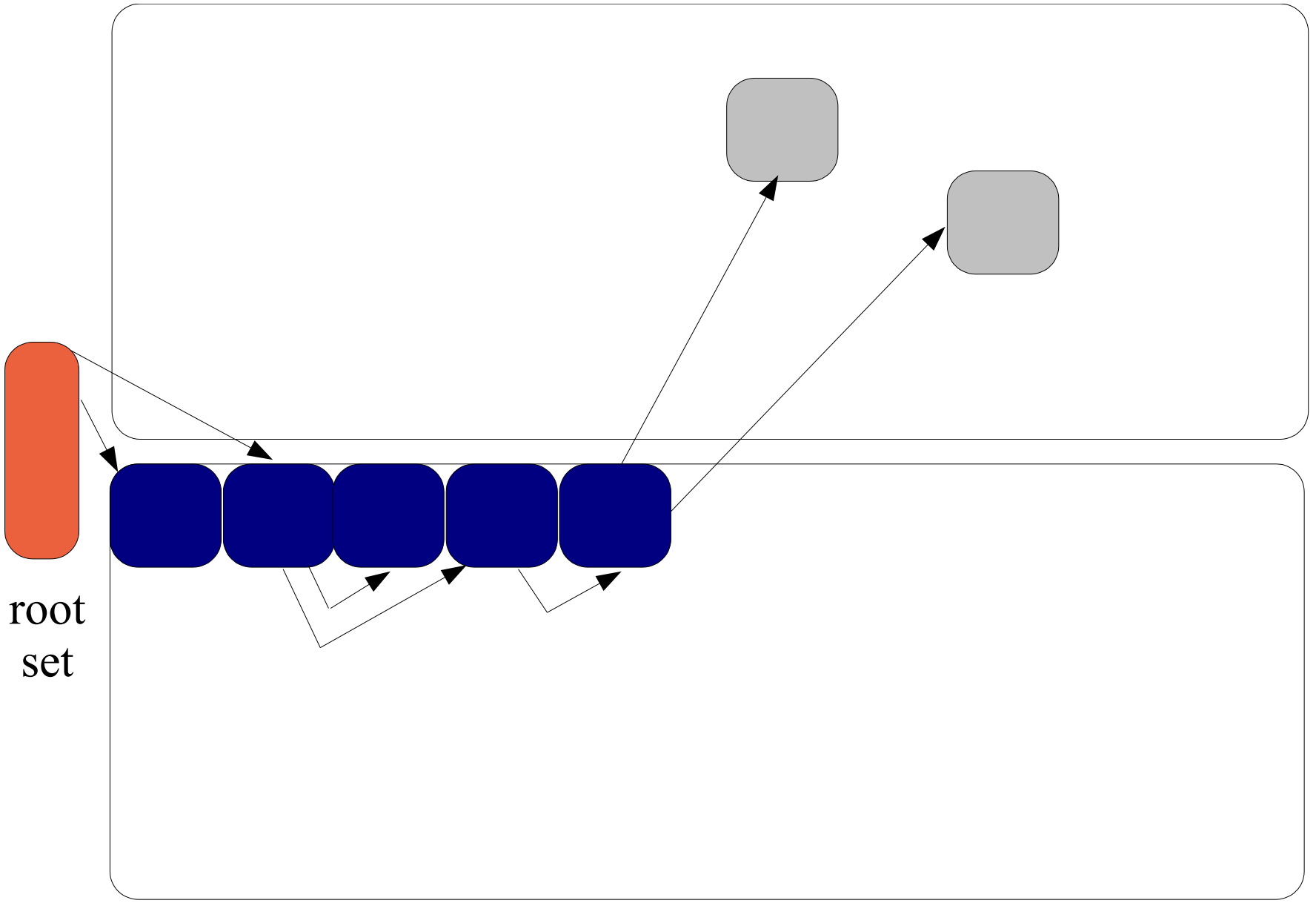
W trakcie odśmiecania...



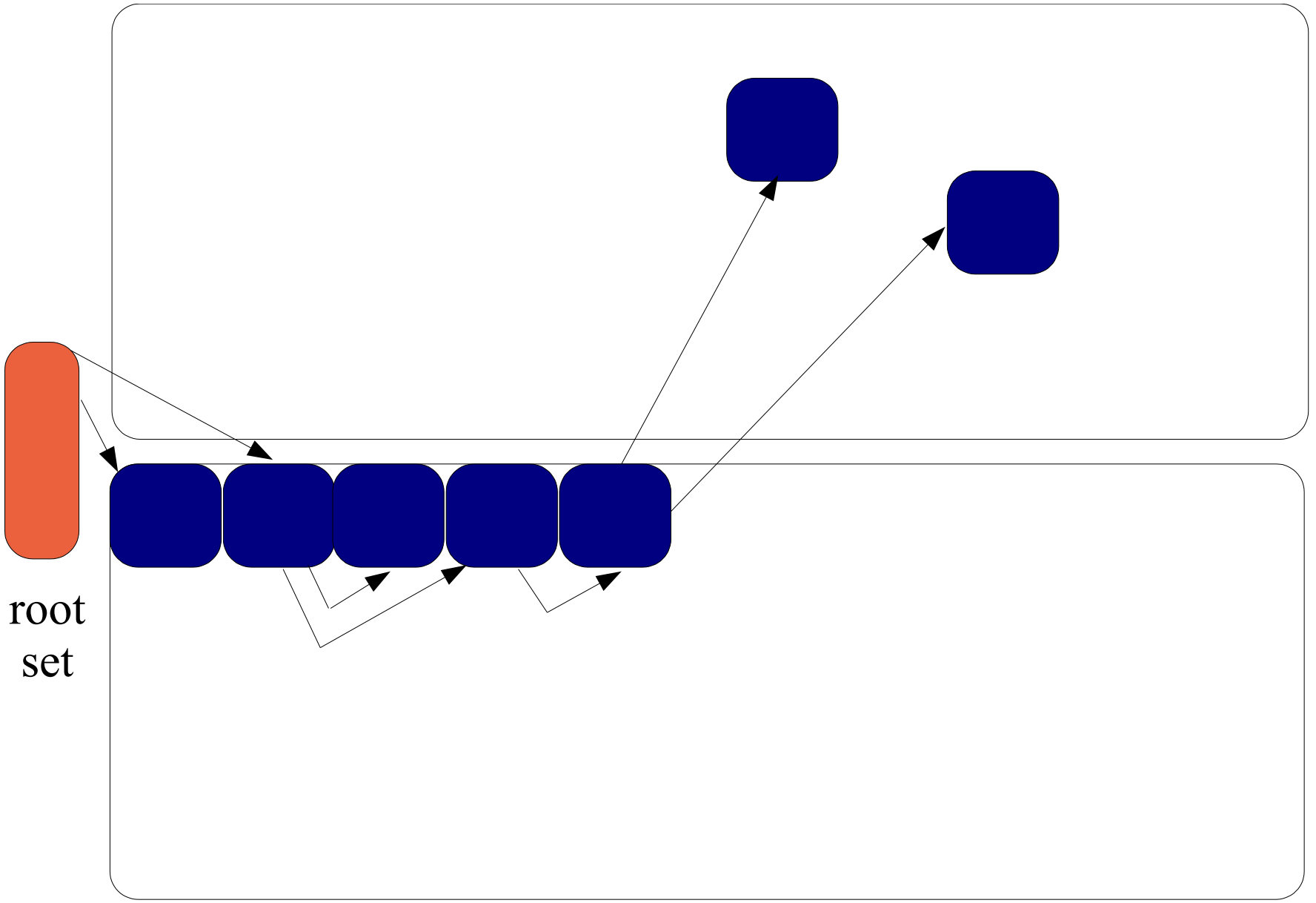
W trakcie odśmiecania...



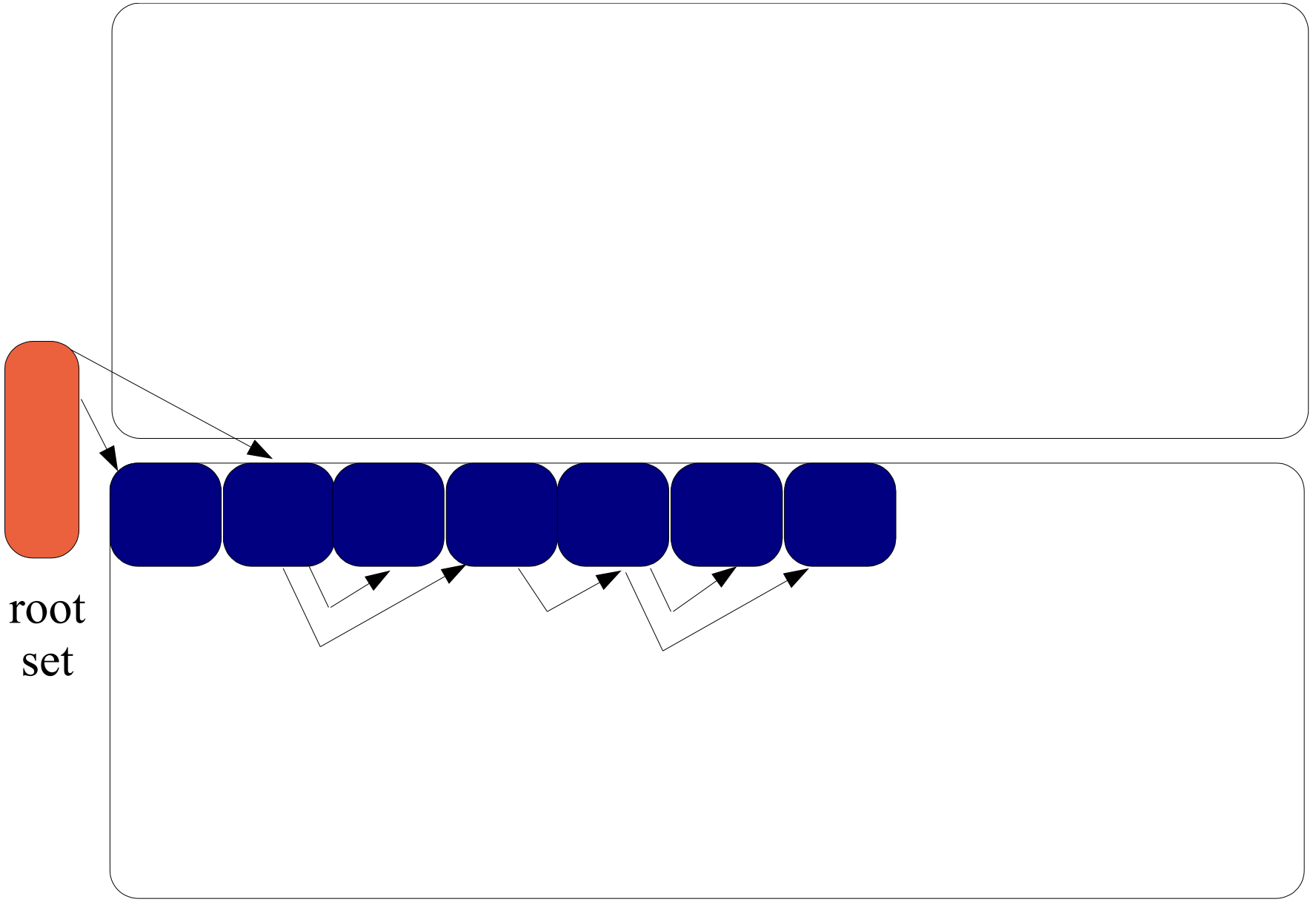
W trakcie odśmiecania...



W trakcie odśmiecania...



Odśmiecanie zakończone



Uwagi do algorytmu mark-copy

- nie ma problemu fragmentacji
- żywe obiekty odwiedzane zaledwie jednokrotnie
- szybka alokacja
- algorytm jest dość skomplikowany
- jeśli większość obiektów jest żywych, wtedy algorytm jest mało wydajny (kopiowanie dużych obiektów)
- stare obiekty także są ciągle kopiowane
- pamięciożerność
- niepraktyczny dla dużych stert
- koszt jest proporcjonalny do ilości osiągalnych obiektów w FromSpace

Podstawowe algorytmy – podsumowanie

- złożoność obliczeniowa algorytmów – podobna
- w zasadzie wszystkie algorytmy mają problemy z lokalnością odwołań
- fragmentacja pamięci
  - w przypadku algorytmu liczników i mark-sweep fragmentacja jest problemem
  - pozostałe algorytmy stosują pakowanie obiektów, co minimalizuje fragmentacje
- praca w trybie przyrostowym

Techniki złożone

Odśmiecanie pokoleniowe

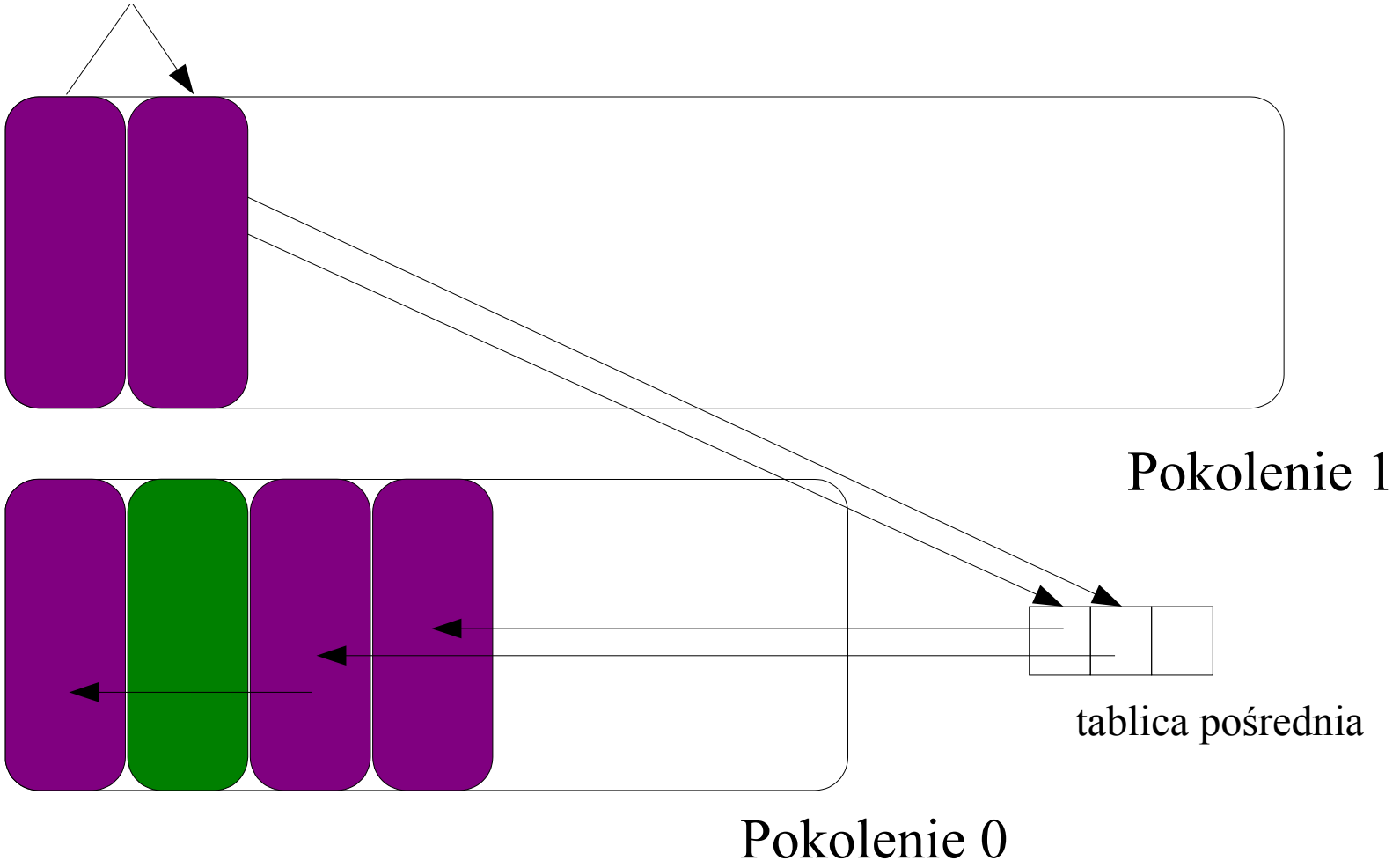
- Trochę uwag statystycznych:
  - większość obiektów żyje bardzo krótko i nie przetrwa nawet jednego odśmiecania (80%-98%)
  - te które przetrwają, bardzo często przetrwają także kilka kolejnych odśmieceń
  - mała liczba referencji prowadzi z obiektów starych do młodych (dużo częściej w drugą stronę)
- Pomysł – podzielmy stertę na “pokolenia” i odpowiednio rozmieśćmy obiekty
- Zysk:
  - unikamy niepotrzebnego kopiowania i skanowania obiektów
  - czas odśmiecania!

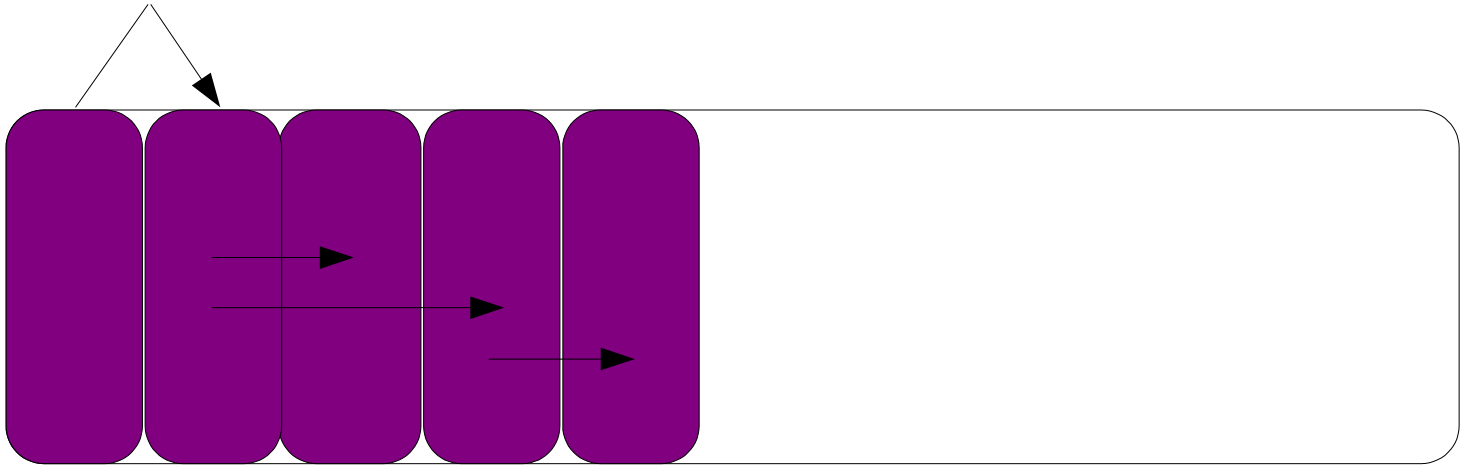
Odśmiecanie pokoleniowe (2)

- jednostką długości życia obiektu jest ilość kolekcji, które przetrwał obiekt
- fragmenty sterty zawierające młode obiekty są odśmiecane częściej, natomiast fragmenty sterty zawierające starsze obiekty rzadziej
- obiekty, które przetrwały odśmiecanie zostają awansowane do starszego pokolenia
- dla poszczególnych generacji możemy stosować różne techniki odśmiecania
- odśmiecanie najstarszych pokoleń możemy odwlekać w czasie

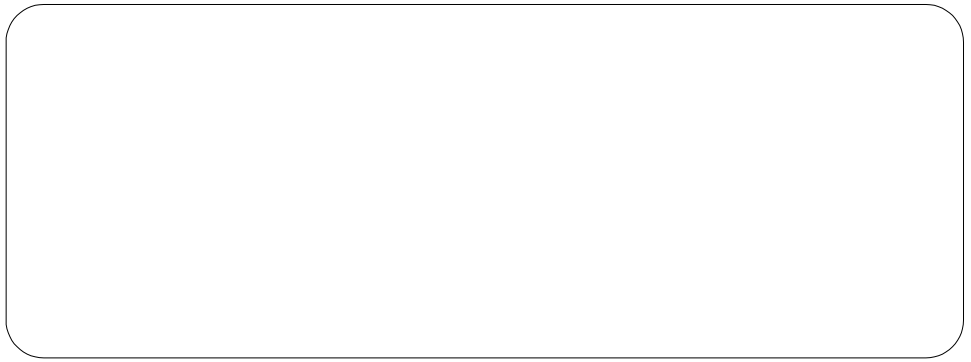
Referencje międzypokoleniowe

- referencje międzypokoleniowe to referencje z obiektu w starszym pokoleniu do obiektu w młodszym pokoleniu (< 2%)
- takie referencje muszą być śledzone, żeby zapobiec sytuacji “zniknięcia” obiektu wskazywanego
- przykładowo może to być zrealizowane przy użyciu tablicy pośredniej, do której prowadzą referencje ze starszych obiektów i w której są adresy do młodszych obiektów


Pokolenie 1
tablica pośrednia
Pokolenie 0
obiekt zajęty
śmieć

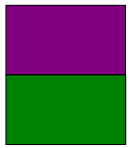


Pokolenie 1



tablica pośrednia

Pokolenie 0



obiekt zajęty

śmieć

Odśmiecanie pokoleniowe – podsumowanie

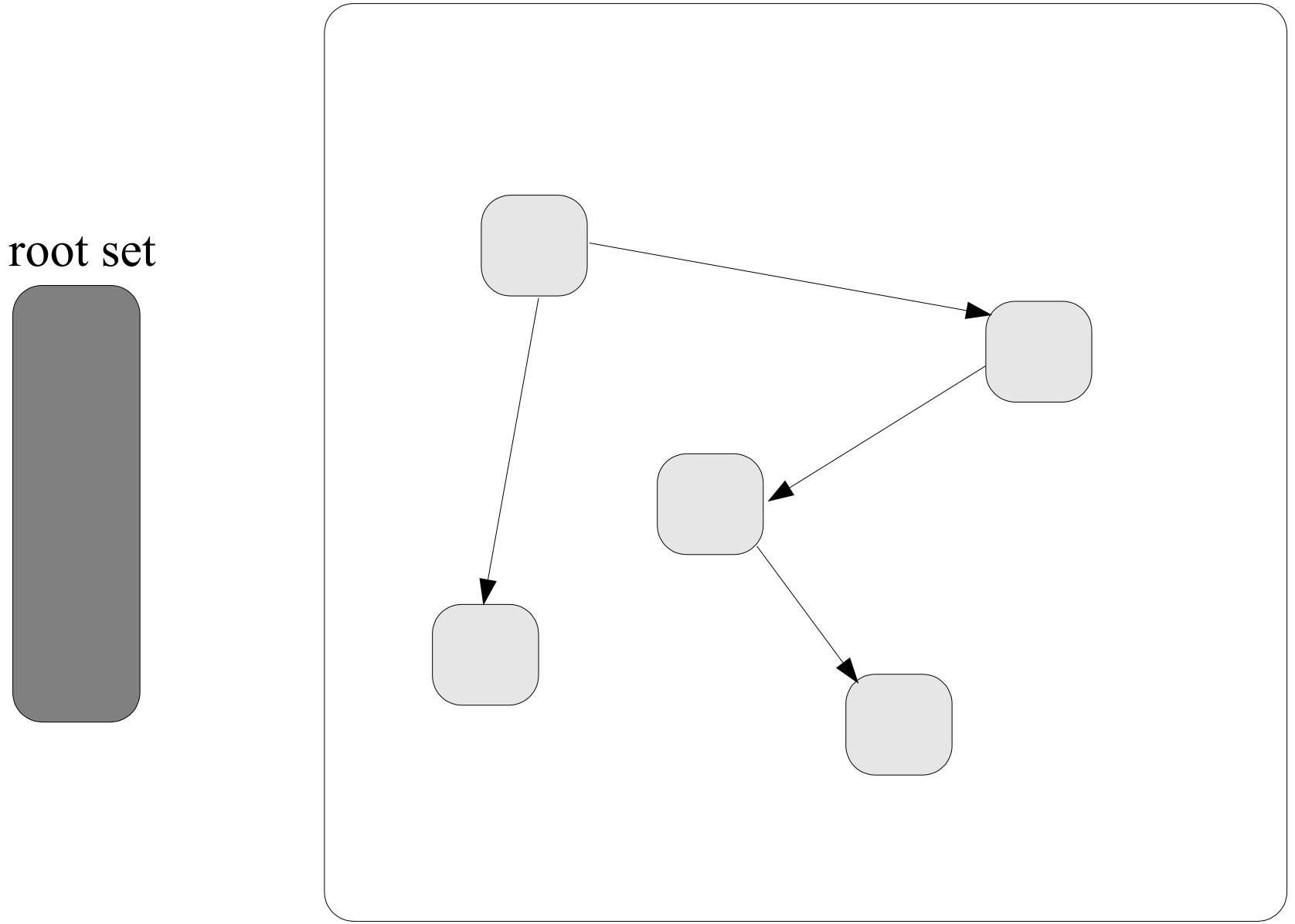
- typowa konfiguracja:
  - dwie lub trzy generacje
  - mark-copy w młodszej generacji
  - mark-compact w starszej generacji
- odśmiecacz bardziej skomplikowany i wymaga solidnego wsparcia kompilatora
- strojenie takiego GC pracochłonne
- jednak wydajność odśmiecania znacznie lepsza

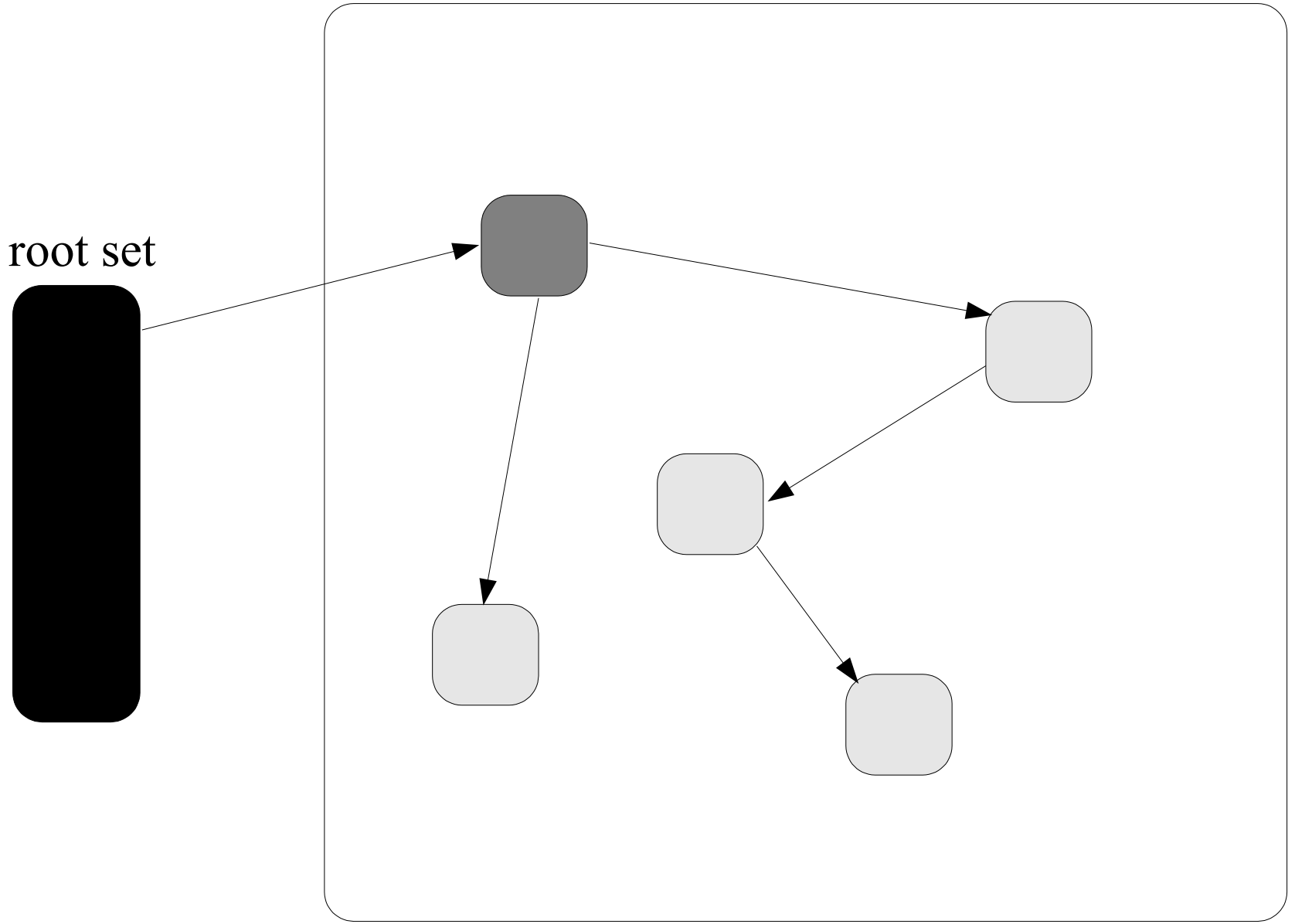
Odśmiecanie przyrostowe

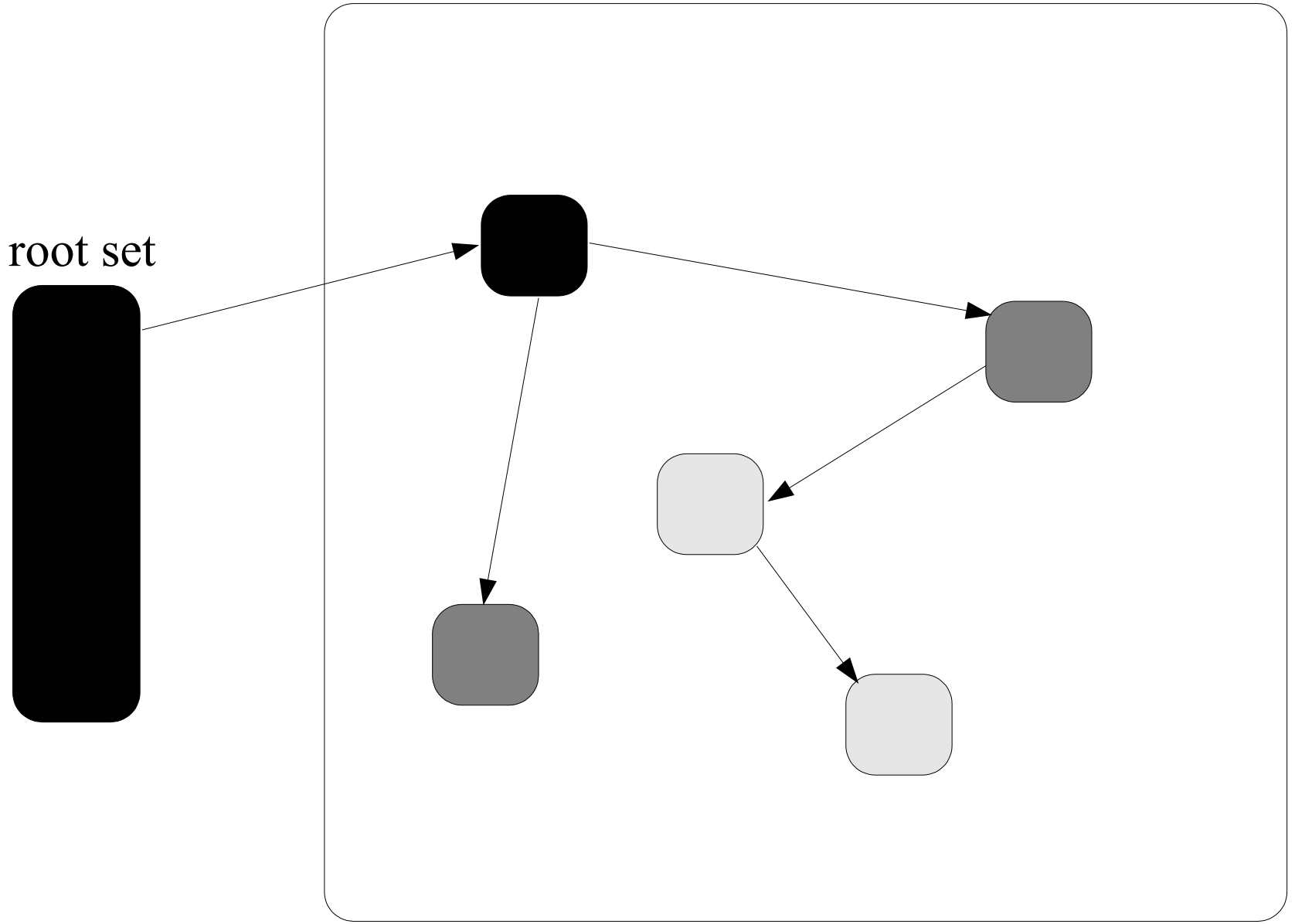
- klasyczne odśmiecanie wymusza zatrzymanie działania programu, co w pewnych przypadkach jest niedopuszczalne
- pomysł – przeplatanie wykonania programu z wątkiem GC
- problem – działający program może zmodyfikować graf osiągalności
- rozwiązanie jest śledzenie zmian w grafie osiągalności

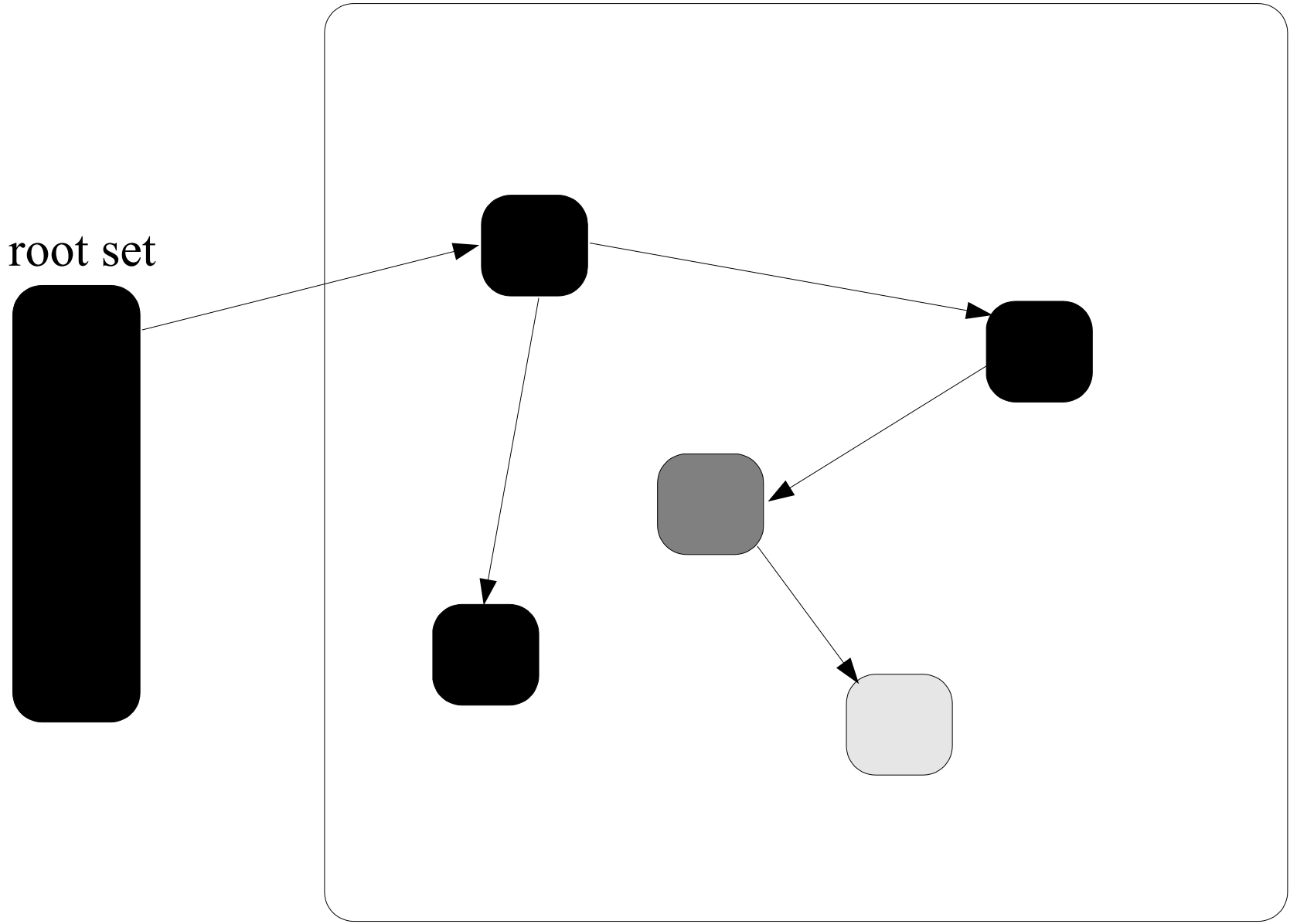
Śledzenie zmian – algorytm “tricolor marking”

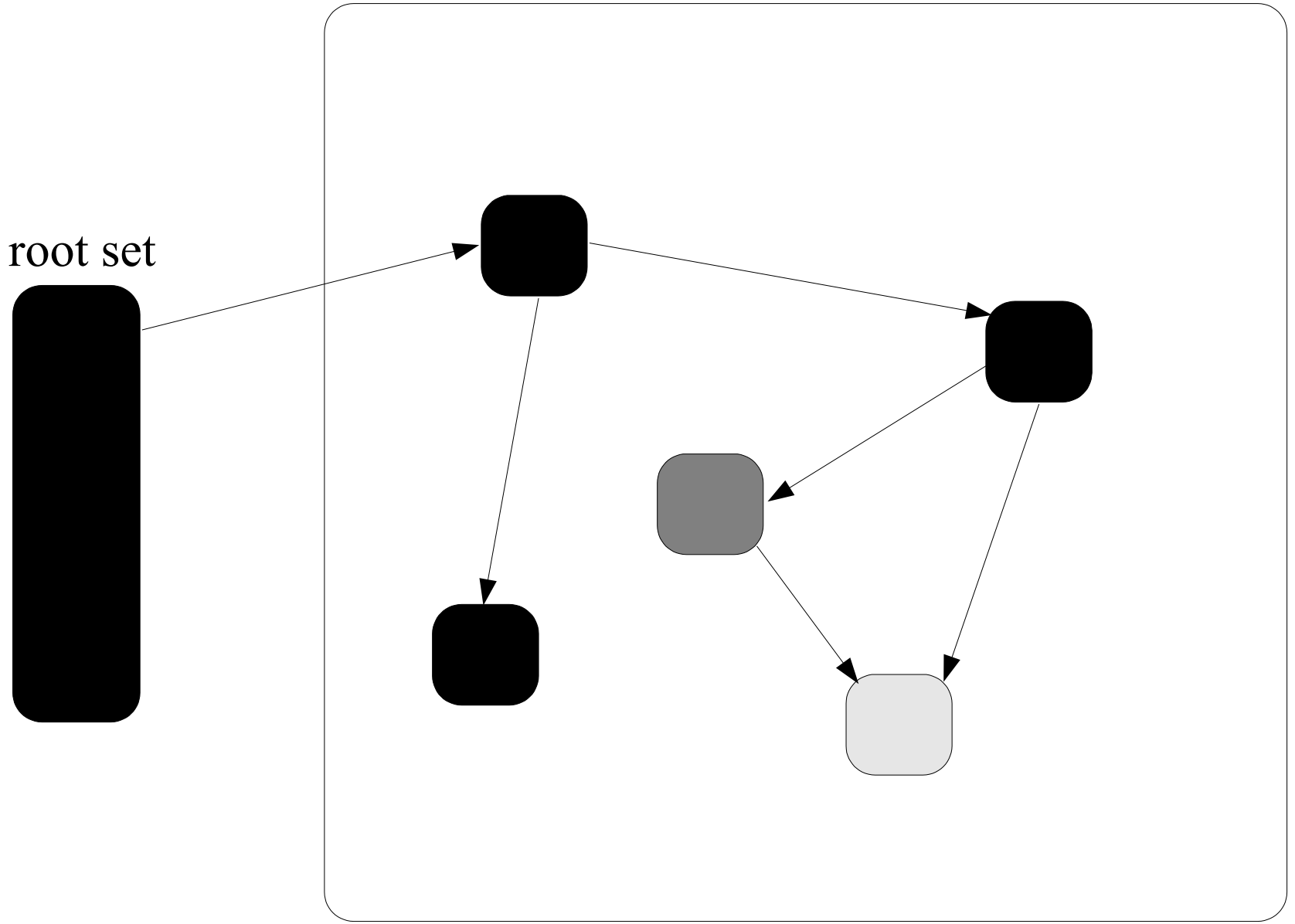
- przechodząc przez graf osiągalności zaznaczamy każdy węzeł jednym z trzech kolorów:
  - biały: węzeł nie zamarkowany
  - szary: zamarkowany, ale potomkowie węzła niezamarkowani
  - czarny: zamarkowany węzeł i jego potomkowie
- zaczynamy z wszystkimi węzłami białymi

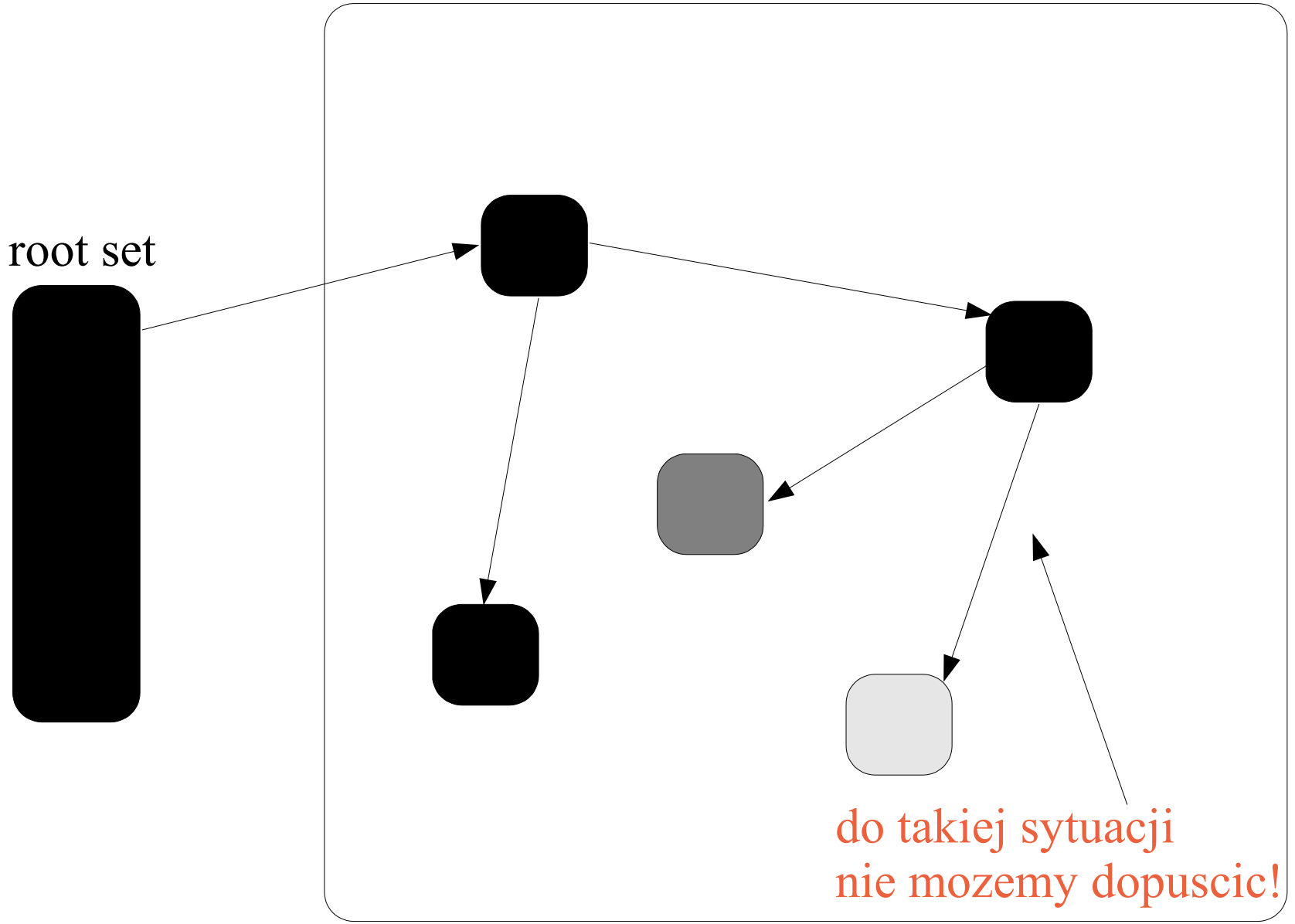

root set


root set


root set

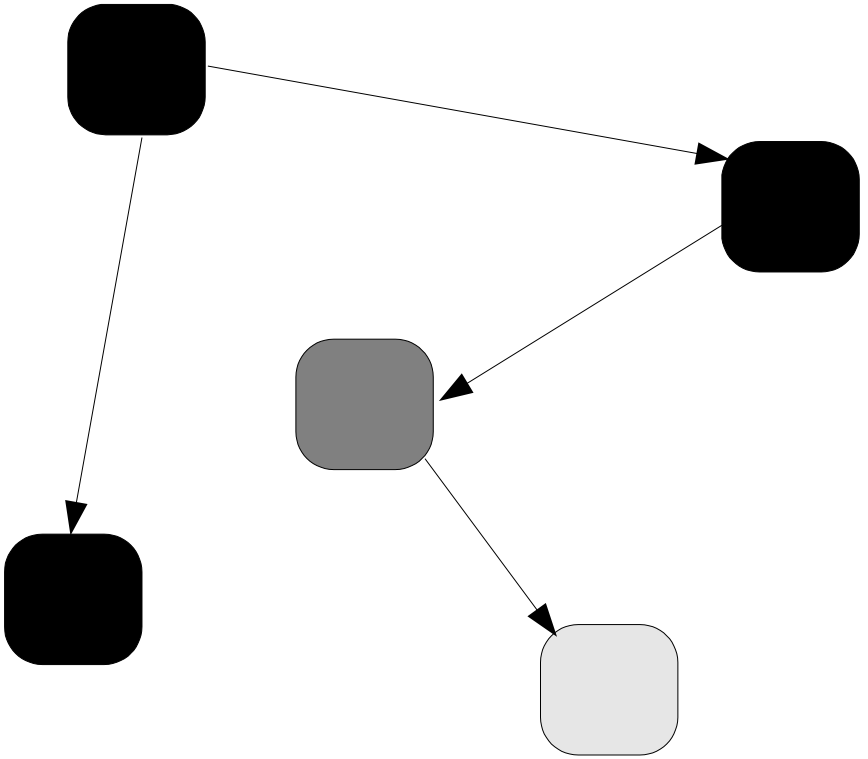

root set


root set

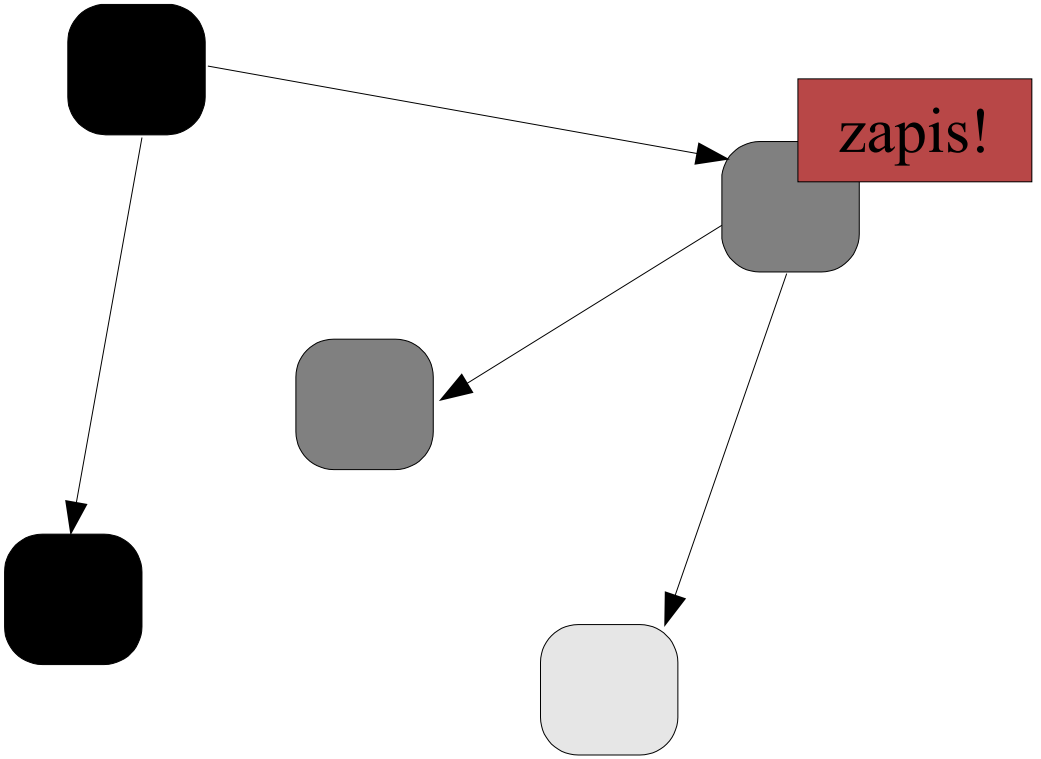

root set
do takiej sytuacji
nie mozemy dopuscic!

Synchronizacja programu z GC

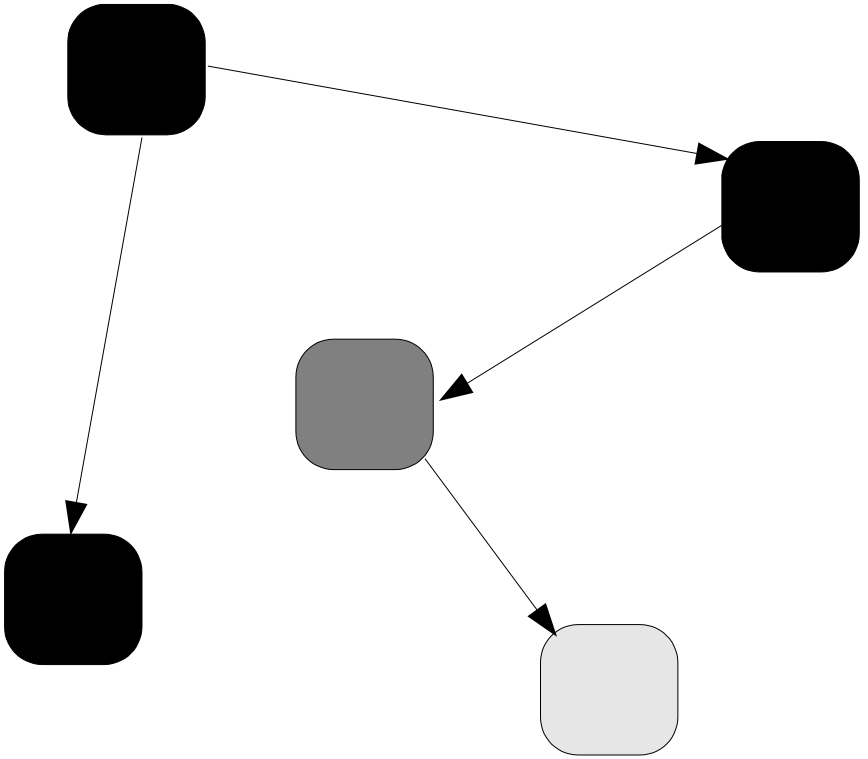
- żeby zapobiec takim sytuacjom możemy:
- zastosować barierę odczytu
  - kontroluje sytuację, kiedy dokonywana jest próba odczytu z węzła koloru białego
  - zmieniamy kolor na szary (z białego)
- zastosować barierę zapisu
  - kontroluje sytuację, kiedy dokonywana jest próba zapisu do węzła koloru czarnego
  - zmianiamy kolor na szary (z czarnego)
- konieczna jest współpraca ze strony kompilatora



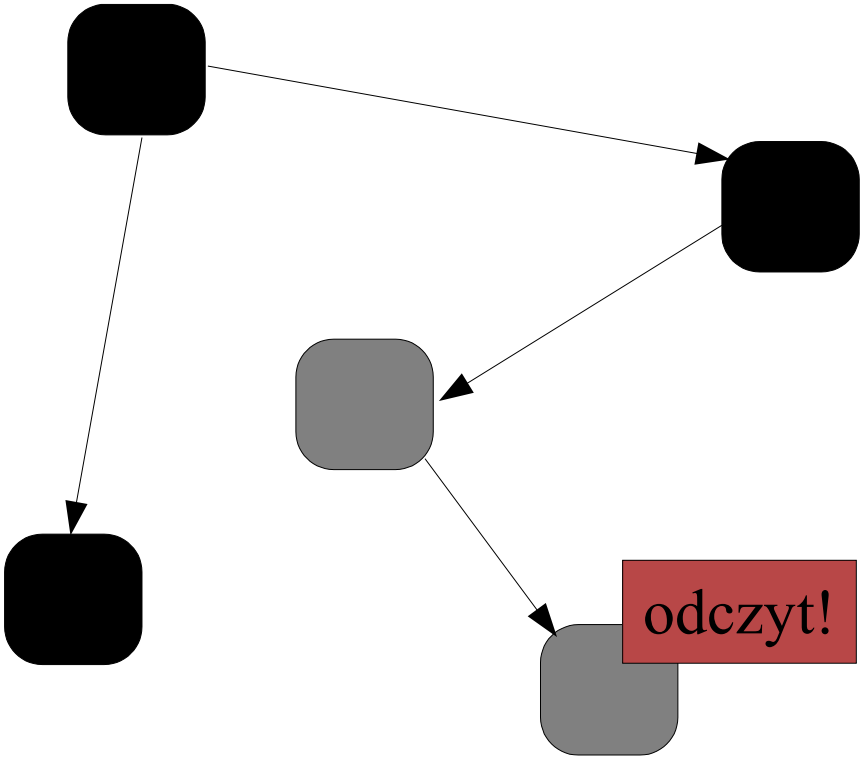
przykład zastosowania bariery zapisu



przykład zastosowania bariery zapisu



przykład zastosowania bariery odczytu



przykład zastosowania bariery odczytu

Podsumowanie przyrostowego GC

- zyskujemy bardzo krótkie okresy pauz
- kosztem
  - zwiekszonych wymagań pamięci
  - wydłużenia czasu odśmiecania
  - skomplikowania algorytmu odśmiecania
- istnieją przyrostowe warianty algorytmów mark-sweep, mark-compact, mark-copy
- zliczanie liczników przyrostowe z definicji
- mimo wszystko do RTS odśmiecanie może się nie nadawać

Odśmiecanie w praktyce: .NET

Odśmiecanie w .NET

- żywe obiekty znajdowane standardowo – śledzenie korzeni
- algorytm GC:
  - obiekty podzielone na pokolenia, plus tzw. Large Object Heap
  - w najmłodszym pokoleniu algorytm mark-copy (bardzo prosta alokacją)
  - w pokoleniu starszym mark-sweep
  - GC jest uruchamiany w przypadku braku zasobów
- sterta jest podzielona na segmenty, w razie potrzeby można zwiększyć liczbę segmentów

- obiekt po przetrwaniu cyklu odśmiecania jest awansowany do starszego pokolenia
- odśmiecanie starszych generacji odbywa sie rzadziej niż tych młodszych
- o tym w jakim pokoleniu znajduje się dany obiekt decyduje jego adres
- najmłodsze pokolenie składa sie tylko z jednego segmentu
- starsze pokolenia mogą tworzyć łańcuch segmentów
- po każdym cyklu Pok-0 jest tworzona na nowo

Large Object Heap

- Large object heap jest wewnętrznie podzielona na
  - obiekty nie zawierające wskaźników
  - obiekty zawierające wskaźniki
- w ten sposób GC nie skanuje niepotrzebnie obiektów bez referencji do innych obiektów
- LOH do zarządzania stosuje malloc-like implementację zarządzania pamięcią
  - konsekwencją jest możliwość fragmentacji

Bariera zapisu

- Obiekty zawierające ref. międzypokoleniowe są kontrolowane przy pomocy bariery zapisu
- pamiętając które to obiekty, wiadomo także gdzie poprawić referencje podczas przenoszenia obiektów
- kompilator JIT emituje stosowny kod podczas napotkania instrukcji modyfikującej referencje

Odzyskiwanie pamięci

- zawieszenie wszystkich wątków programu
- awansowanie żywych obiektów
- uaktualnienie wskaźników
- zmiecenie obiektów martwych i uaktualnienie struktur (list wolnych pamięci)

Finalizacja

- GC w .NET umożliwia finalizowanie obiektów
- Minusy stosowania Finalize()
  - brak kontroli nad czasem i kolejnością wykonania metody
  - dłuższa alokacja
  - narzut na wykonanie metody
  - jeśli Finalize() odnosi się do innych obiektów, to autmatycznie uniemożliwia ich odśmiecenie
  - odkładanie odśmiecania
  - odśmiecenie obektów zajmuje dwa cykle GC

Boehm-Demers-Weiser conservative GC

Algorytmy zachowawcze

- pracują w niesprzyjających warunkach
  - konieczna niezależność od kompilatora
  - bez 100% pewności czy dany obiekt na stercie jest referencją
  - bez informacji o zawartości rejestrów i stosu
  - często bez możliwości przeniesienia obiektu
  - często musi współpracować z malloc()/free()
- przykłady: GC dla C/C++

Algorytmy zachowawcze (2)

- rozwiązanie – musimy założyć, że każde słowo w pamięci potencjalnie może być referencją
- jeśli analiza zawartości danego słowa w pamięci sugeruje że to może być referencja, to traktujemy je jakby nią było
- Problemy:
  - kompresowanie utrudnione, ponieważ musimy modyfikować wskaźniki, które nie muszą nimi być
  - wycieki pamięci – w przypadku, kiedy interpretujemy słowo jako referencje, choć nią nie jest

Boehm-Demers-Weiser GC

- niekopiujący, stosujący algortym mark-sweep GC
- w pełni zachowawczy i niezależny od kompilatora
  - w szczególności, alokowane obiekty nie zawierają żadnych dodatkowych informacji o zawartości
- może pracować w trybie detektora wycieków pamięci
- modyfikację kodu to najczęściej:
  - zamieniamy malloc() na GC_malloc()
  - zamieniamy realloc() na GC_realloc()
  - usunięcia free()
  - plus dodanie pliku nagłówkowego i GC_init()

Sterta

- sterta standardowa
  - kompatybilna z malloc
  - bez referencji do sterty GC
- sterta GC
  - dla GC_malloc()
  - ignorowane wskaźniki do sterty malloc
  - podzielona na bloki, np. wielkości 4K
  - każdy blok zawiera obiekty określonego rodzaju

Fazy GC

1. Przygotowanie – wyzerowanie flagi informującej o tym, czy obiekt jest zamarkowany
2. Faza markowania
3. Faza zamiatania
4. Faza finalizowania – obiekty zarejestrowane do sfinalizowania są kolejkowane do finalizacji

Alokowanie pamięci

- alokator pobiera z systemu pamięć przy pomocy standardowego malloca
- każdy blok ma określony rodzaj, np. blok którego nie trzeba skanować
- do alokowania dużych przestrzeni pamięci używana jest strategia best-fit
- bloki zawierające mniejsze obiekty zawierają obiekty tego samego rodzaju
- w przypadku braku zasobów najpierw następuje próba powiększenia sterty, a w przypadku niepowodzenia proces odśmiecania

Faza markowania

- W poszukiwaniu referencji skanowane są:
  - rejestry procesora, stos, segment danych statycznych
- możliwe jest markowanie przyrostowe
- procedura decydowania czy dane słowo jest referencją:
  - czy wskazywany adres mieści się w granicach sterty?
  - adres jest dekodowany i sprawdzany w tablicy zawierającej informację o zaalokowanych obiektach

Faza zamiatania

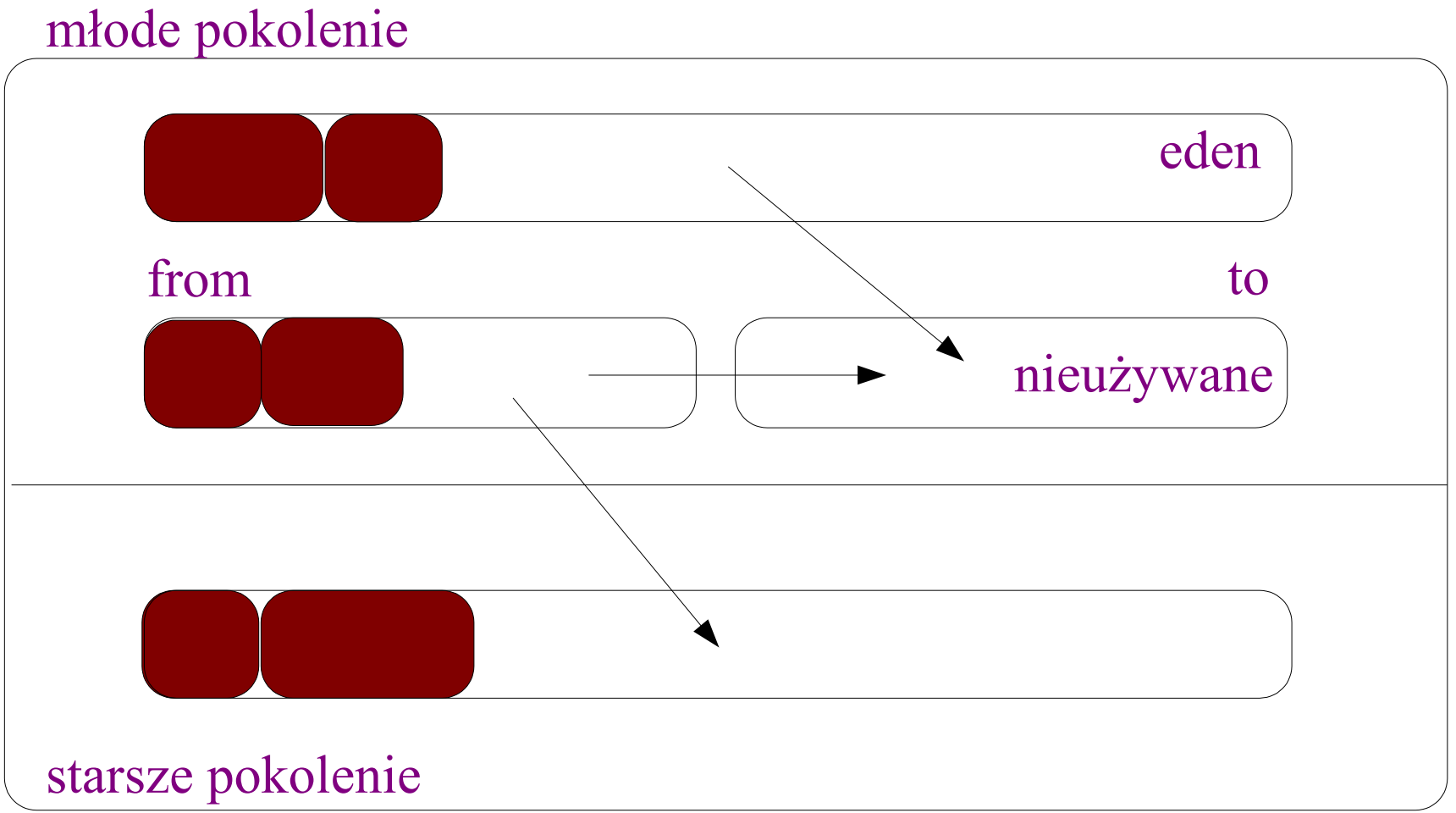
- oglądamy wszystkie obiekty
  - obiekty wielkie natychmiast dodajemy do listy wolnych obiektów
  - obiekty małe, jeśli znajdują sie w segmencie z żywymi obiektami muszą czekać

Java HotSpot GC

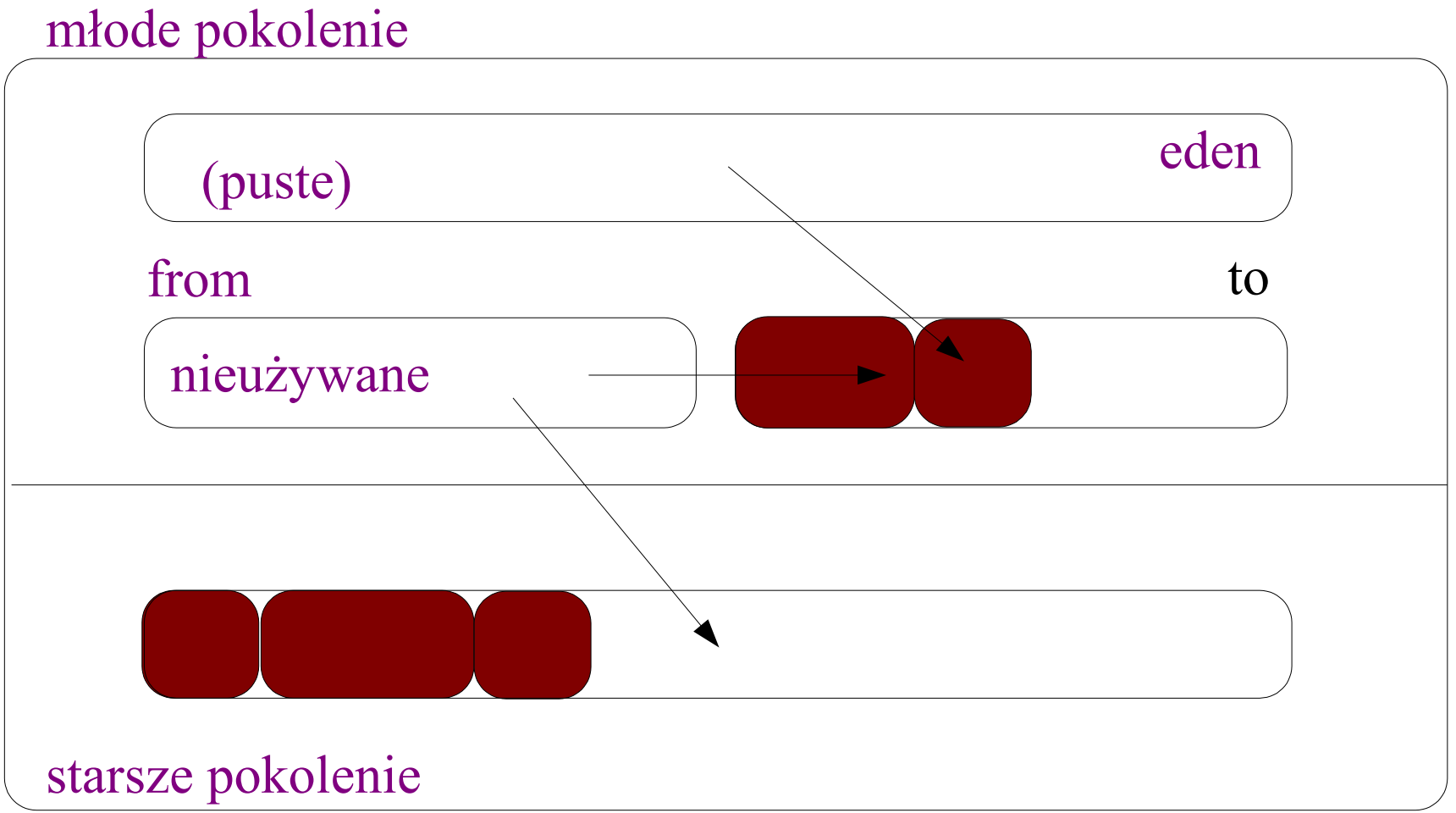
Java HotSpot GC

- zaimplementowano wiele algorytmów odśmiecania
  - stop-the-world
  - algorytm równoległy
  - algorytm przyrostowy
- schemat pokoleniowy
  - młodsze pokolenie – mark-copy
  - starsze pokolenie – mark-compact
- szybka alokacja (przesunięcie wskaźnika)

Pokolenia


młode pokolenie
eden
from
to
nieużywane
starsze pokolenie

Pokolenia


młode pokolenie
(puste)
eden
from
to
nieużywane
starsze pokolenie

- Algorytmy:
  - młodsze pokolenie – mark-copy
  - starsze pokolenie – mark-compact
- Obiekty są przenoszone z młodszego pokolenia do starszego, po przetrwaniu kilku cykli kopiowania w młodszym pokoleniu
- do śledzenie referencji międzypokoleniowych stosowana jest odmiana bariery zapisu
  - sterta podzielona jest na “karty” (cards)
  - w tablicy bitowej zaznaczany jest fakt modyfikacji referencji w obiekcie znajdującym się w karcie
  - podczas odśmiecania skanowane na okoliczność istnienia referencji do młodzego pokolenia są tylko obiekty w których dokonano zapisu

Wydajność odśmiecania

- wczesne implementacje odśmiecania były bardzo kosztowne
  - zarządzanie pamięcią kosztowało nawet 40% więcej niż przy manualnym zarządzaniu pamięcią
- aktualnie odśmiecanie jest bardzo wydajne i w dobrych implementacjach koszt nie przekracza 10% (a często dużo mniej)
- największy koszt w przypadku odśmiecania zachowawczego
  - alg. BHW średnio 20%, w pesymistycznym przypadku 57%
- oczywiście narzut jest, jednak sztuczka polega na
  - skróceniu GC (pokolenia, odśmiecanie przyrostowe)
  - wykonywaniu odśmiecania w mało uciążliwym momencie

Odnośniki

- wstęp do algorytmów odśmiecania:
  http://www.cs.utexas.edu/ftp/pub/garbage/gcsurvey.ps

  http://www.memorymanagement.org/
- odśmiecanie w .NET

http://msdn.microsoft.com/msdnmag/issues/1100/GCI/default.aspx

http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/default.aspx

- odśmiecanie w Javie HotSpot:

http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs884/GCinJava.html

http://www-106.ibm.com/developerworks/java/library/j-jtp11253/

- Boehm-Demers-Weiser GC:

http://www.hpl.hp.com/personal/Hans_Boehm/gc/