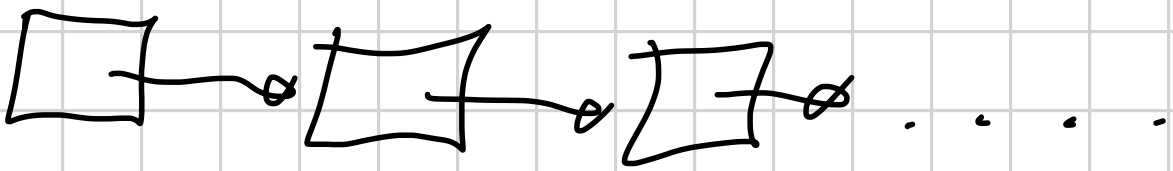
1) GESTIONE ESPLICITA (C)

- uso dopo dealloc. } CRASH
- doppia dealloc. } CRASH
- mancata dealloc. } MEMORY LEAK
- CONCORRENZA DISCIPLINA

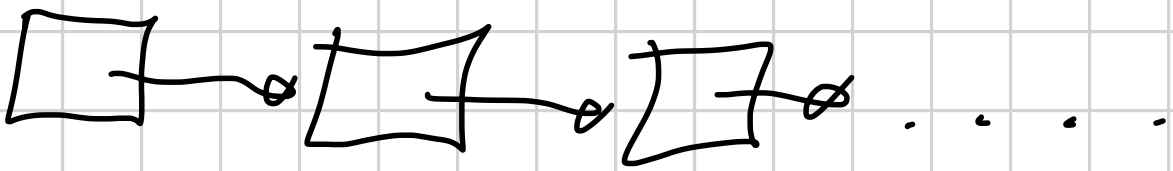
+ MASSIMO CONTROLLO QUANDO



1) GESTIONE ESPLICITA (C)

- uso dopo dealloc. } CRASH
- doppia dealloc. } CRASH
- mancata dealloc. } MEMORY LEAK
- CONCORRENZA DISCIPLINA

+ MASSIMO CONTROLLO QUANDO



2) GESTIONE VIA SMART POINTERS (C++, RUST)

- il programmatore alloca la memoria dicendo che uso ne farà
  - ACCESSO SINGOLO / CONCORRENTE
  - ACCESSO READ / WRITE / READ WRITE } ALLOC.
  - " TEMPORANEO / PERMANENTE } USO
  - SHARING SÌ / NO
- il compilatore
  1) dealloca autom. i dati non più in uso
  2) garantisce correttezza

3) GARBAGE COLLECTION

\+ dealloc. autom. e corretta

– (?) mancato controllo sui tempi della GC

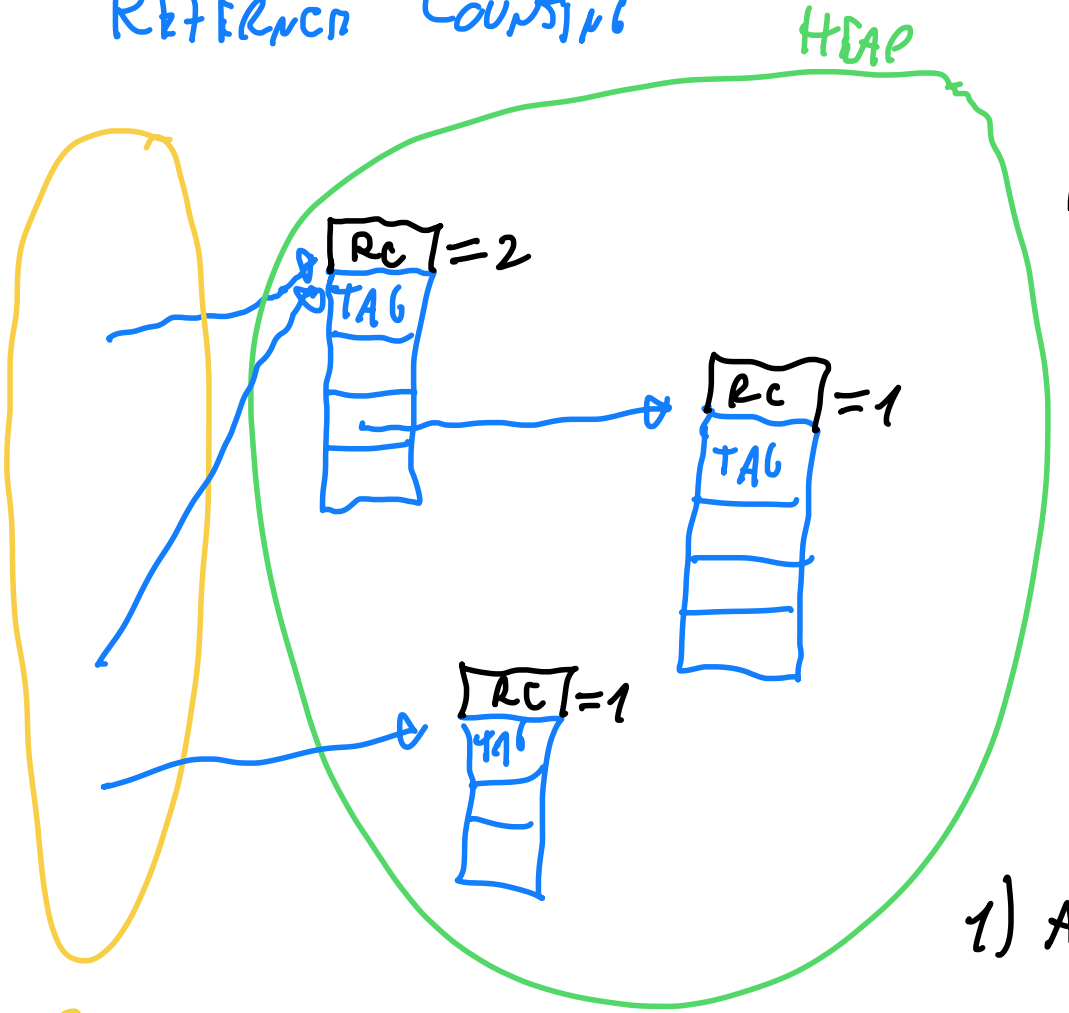
REFERENCE COUNTING

MARK & SWEEP

GENERAZIONALE

(ERLANG, OCAML, HASKELL)

REFERENCE COUNTING



RC = # DI PUNTATORI ENTRANTI

RC = 0 ⇒ DEALLOC.

1) ALLOC SETTA RC A 1

2) ASSEGNAMENTI

```
↑ = q;
IF(q != null) {
  q → RC++;}
IF (↑ != null) {
  ↑ → RC--;
  IF (↑ → RC == 0){
}} DEALLOC()
```

3) USCITA DI SCOPE VARIABILI LOCALI "}"

$\forall v.\ v = NULL;$ ⟹

REFERENCE COUNTING

+ IMPLEM. SEMPLICE

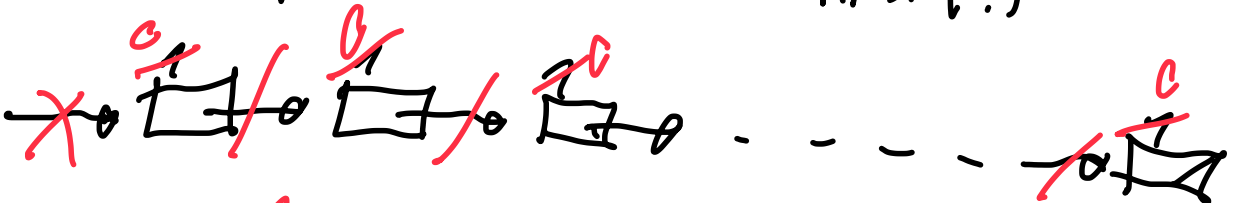
− IN SPAZIO: COSTA 1 WORD PER RECORD

− IN TEMPO: DIVERSE OPERAZIONE MACCHINA

E 2 ACCESSI ALLA HEAP

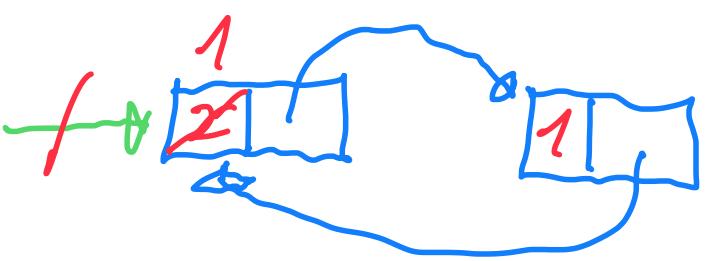
MOLTO COSTOSI

+ LENTO, MA NON CI SONO PAUSE (?)

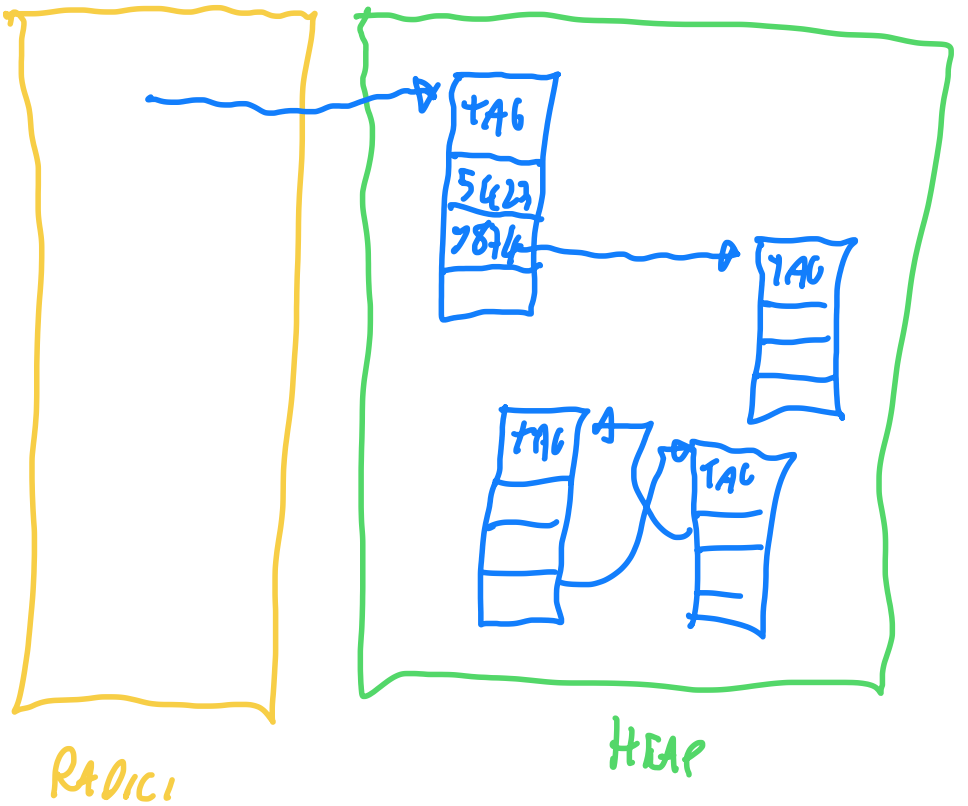


O(n)

− MEMORY LEAK



MARK & SWEEP



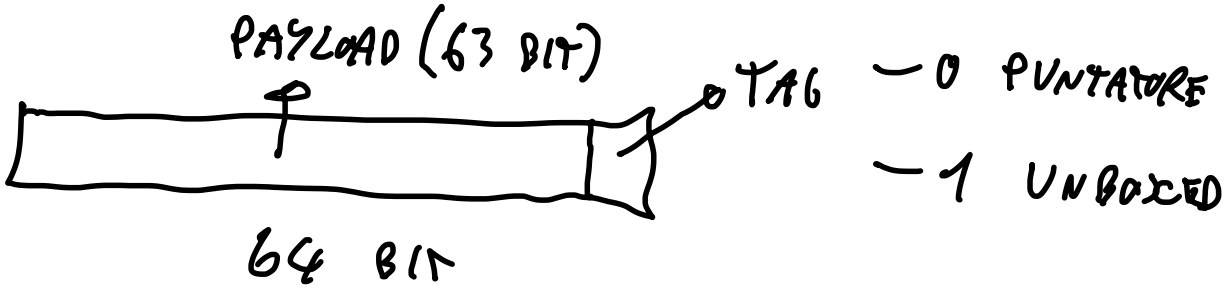
TIPO DI DATO:

- UNBOXED
- BOXED ⇒ ALLOCATO SULLO HEAP

WORD

DATO

PUNTATORE



- 0 PUNTATORE
- 1 UNBOXED

TIPI BOXED (PUNTATORI)

- USARE BIT MENO SIGNIF. per non perdere 50% RAM, ma solo indirizzi 1 ogni 2
- USARE 0 PER BOXED per indirizzi solo pari (ALLINEATI) raccomandato dal processore

TIPI UNBOXED

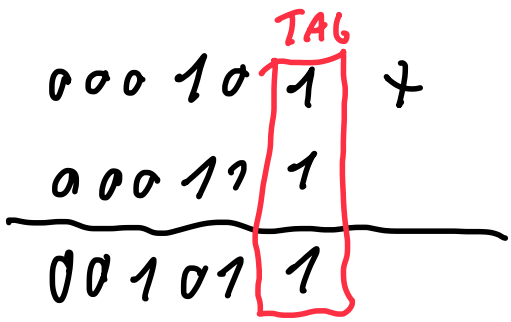
- SOLO NUMERI 63 BIT

PER APPLICAZIONI NUMERICHE

NON USARE QUESTA TECNICA

1) INTEROPER.
2) INEFFICIENZA ARITMETICA

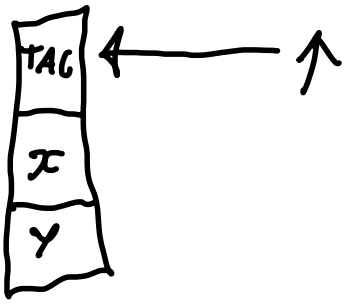
$2 + 3 = 5$



$x + y \quad \mathbb{N}$

$x + y - 1$

STRUCT { INT x; INT y; }

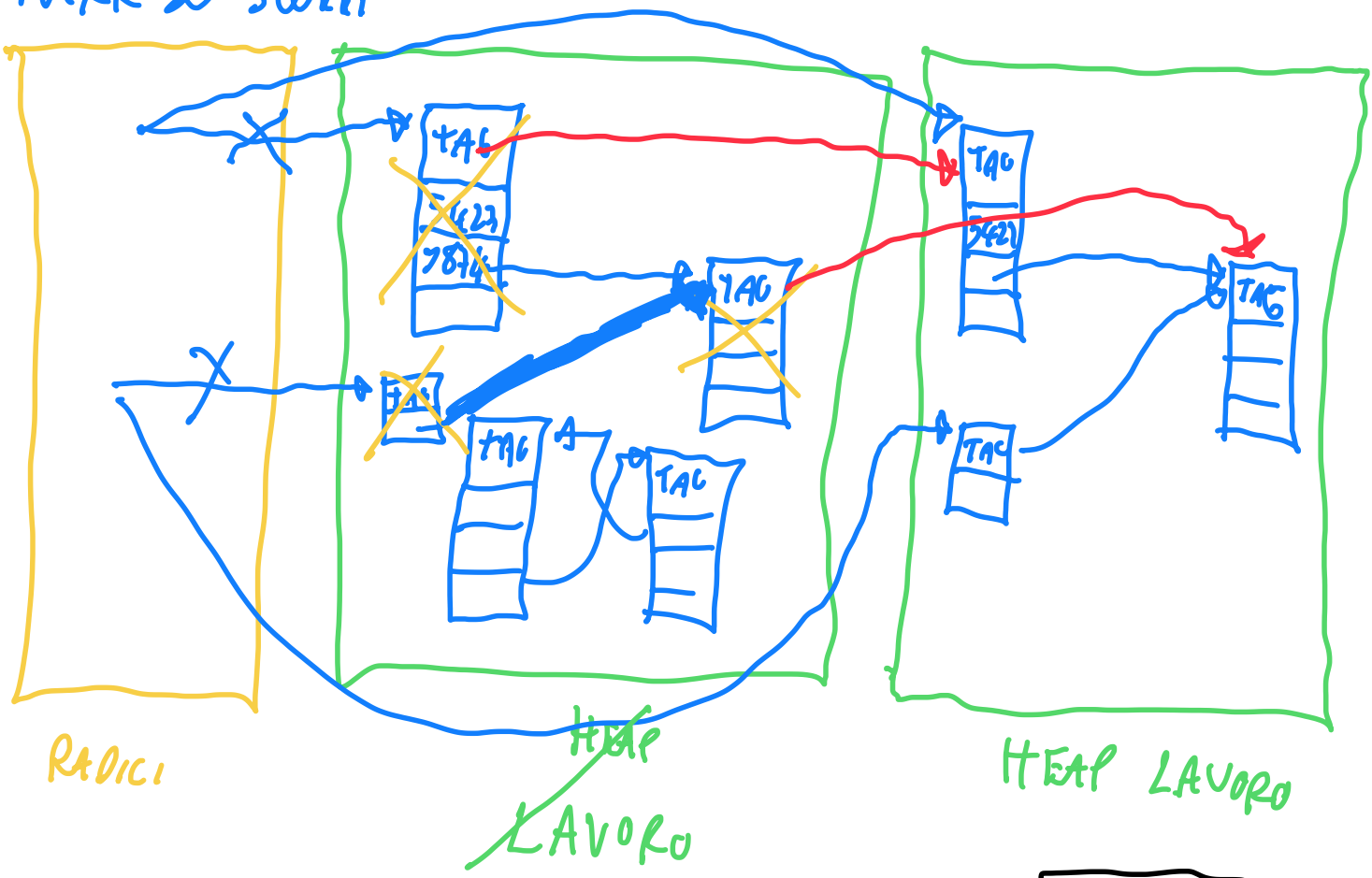


↑ = 

↑.x *(↑+2)

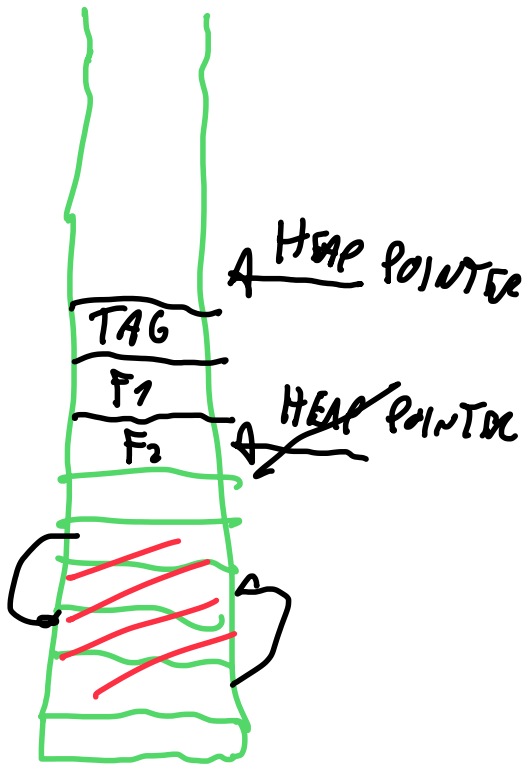
↑.y *(↑+3)

MARK & SWEEP

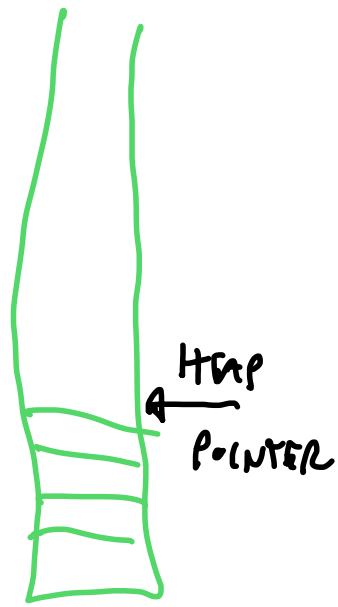


ALLOC:

O(1)



DEFRAG



MARK & SWEEP

1) trade off spazio - tempo

50% HEAP allocazione O(1)

2) trade off tecnica - costrutti linguaggio

- no puntatori ai campi dei record
- no aritmetica dei puntatori

+SÌ: CONFRONTARE CON == PUNTATORI

NO HASHTABLE E ALBERI DI RICERCA E.

- NO: CONFRONTARE CON <
- NO: SOTTRAZIONE ↑ - 9
- NO: INDIRIZZI STATICI
- NO: HASH DI UN PUNTATORE

CON CHIAVI PUNTATORI

3) Debug fa bene alla cache

(MA LA COLLECTION È MOLTO IMPATTANTE)

COMPLESSITÀ COMPUTAZIONALE:

$$O(\#\text{RADICI} + \#\text{TUPLE VIVE})$$

- ottima per dati che vivono pochi (i morti non costano)
- pessima per dati che vivono a lungo (pagati a ogni collection)

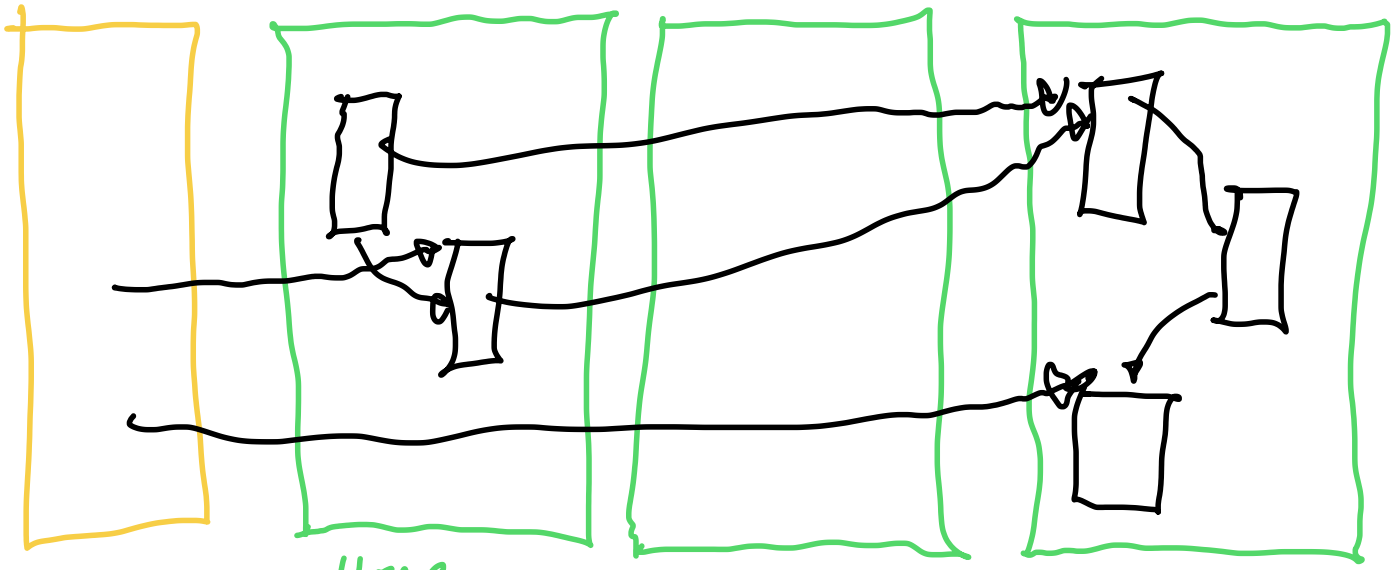
MARK & SWEEP GENERAZIONALE

- ipotesi generazionale (STATICA)

  i dati appartengono tipicamente a 2 categorie

  1) dati che muoiono velocemente

  2) dati che persistono molto a lungo



MARK & SWEEP

MINOR COLLECTION

FREQUENTEMENTE

(HEAP LAVORO PIENO)

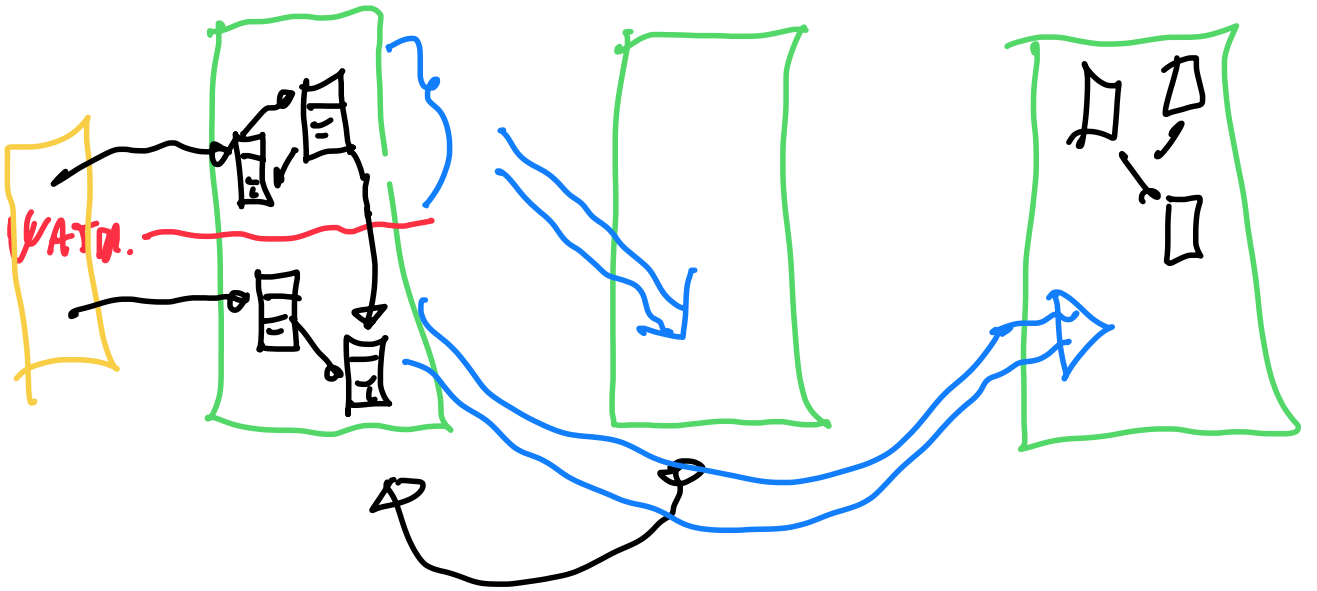
MAJOR COLLECTION

SUPER-RARAMENTE

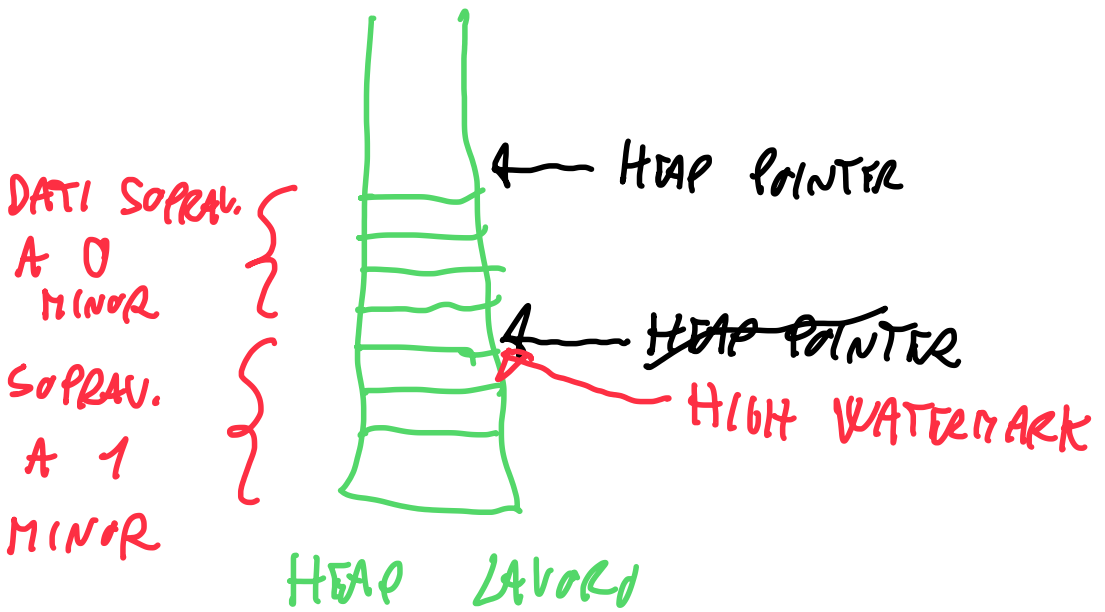
(HEAP PERSIST. PIENO SOLO RICHIESTA PIÙ HEAP AL SO)

- ECONOMICA
- MOLTO UTILE

- COSTOSA
- POCO UTILE



MINOR - COLLECTION



GENERATIONAL GC:

SUPPONENDO CHE VALGA L'IPOTESI GENER.

- MINOR: $O(\#\text{RADICI} + \#\text{VIVI NELL'HEAP DI LAVORO})$

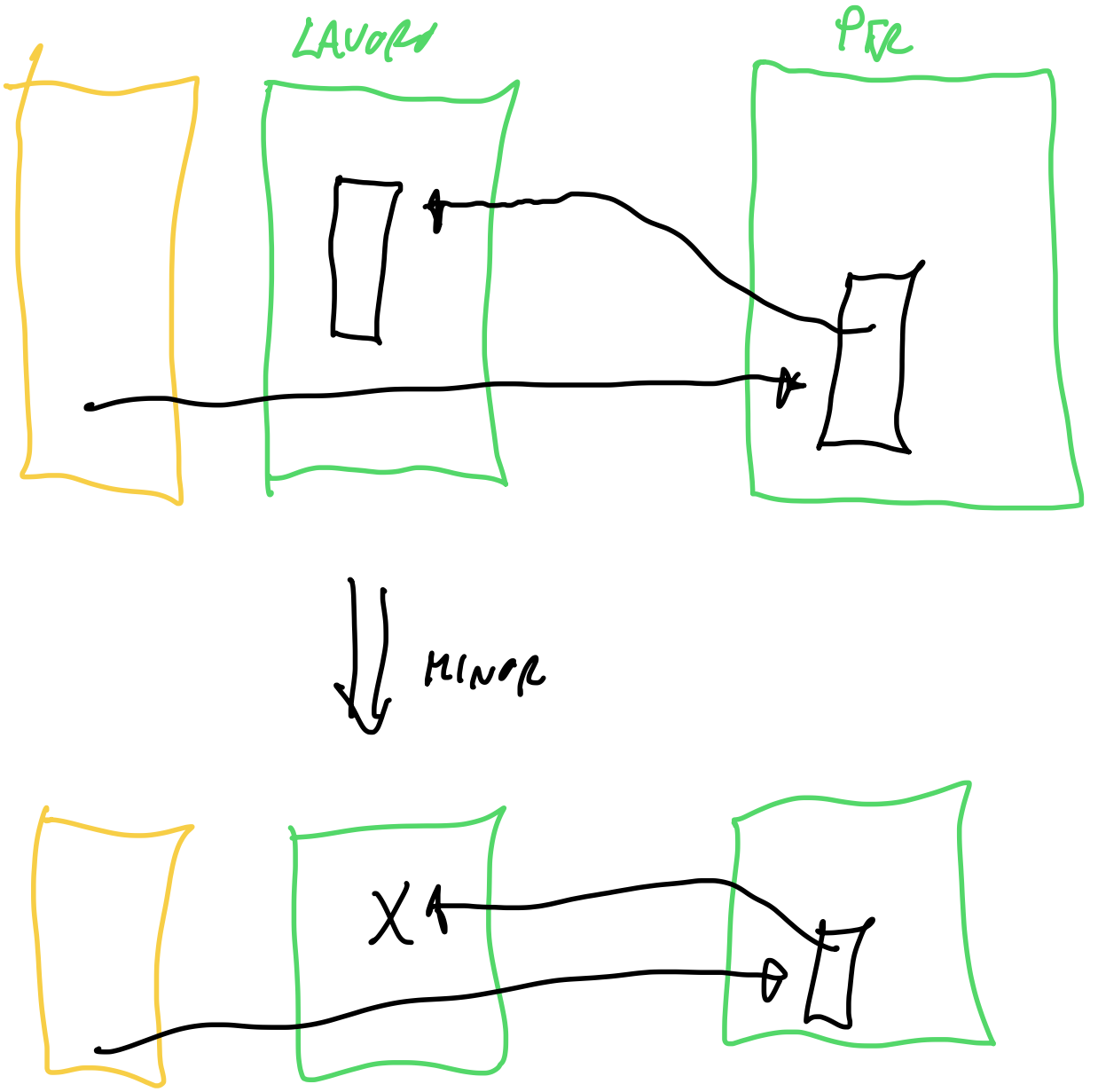
  ECONOMICA

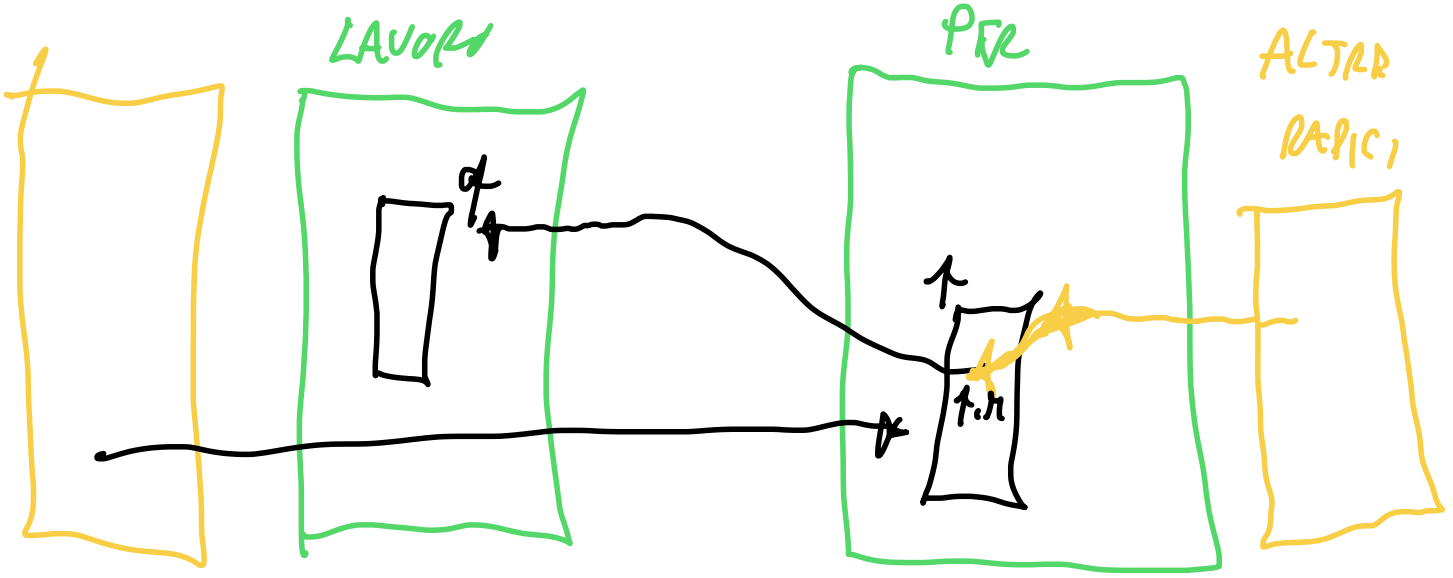
- MAJOR: $O(\#\text{RADICI} + \#\text{VIVI})$

  COSTOSA

TRADE OFF COMPLESSITÀ - COSTRUTTI:

- LINGUAGGIO PURO, I.E. NO ASSEGNAMENTI


LAVORI
PER
MINOR



⇓ MINOR

se il linguaggio fa mutazione:

p → n = q; // ASSEGNO UN PUNT.

⇓

IF (p ∈ HEAP PERS. && q ∈ HEAP LAVORO) {
    ADD (LISTA_RADICI, p);
}
p → n = q;

WRITE BARRIER

— trade off aggiuntivi

SÌ MUTAZIONI, MA PIÙ COSTOSE LETTURE

OCAML