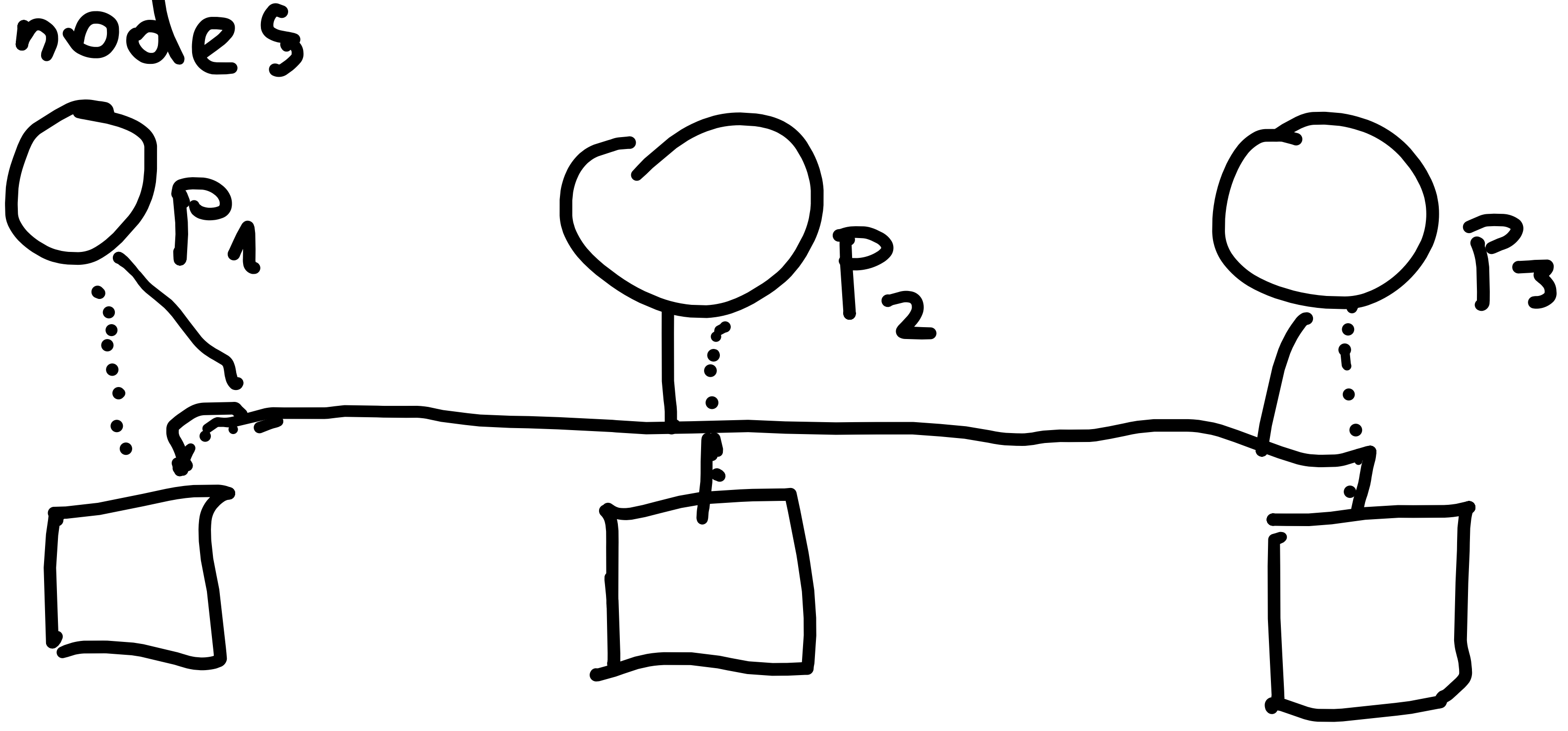
Shared memory

Algo lecture 21

Memory

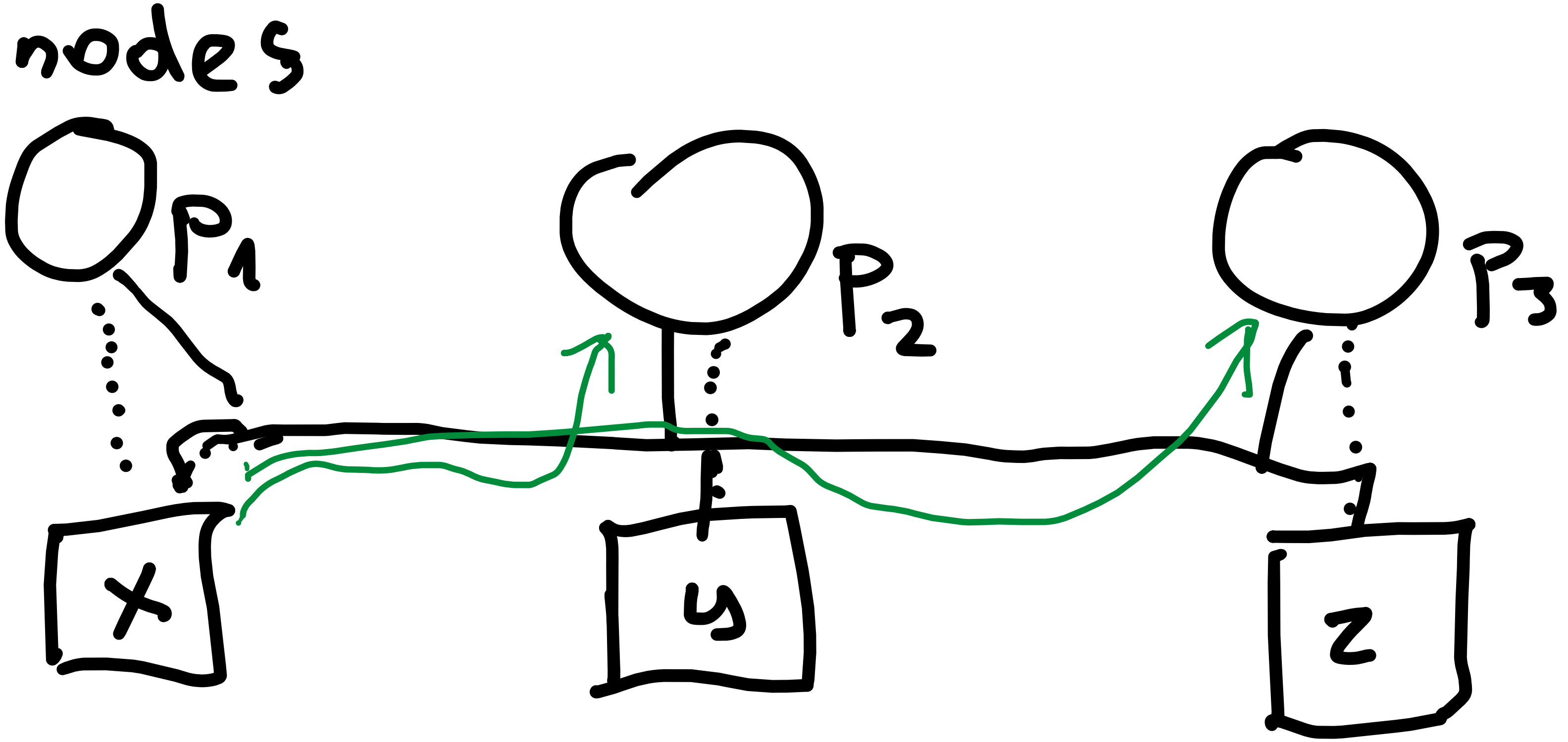


Operations

Read

Write

Memory



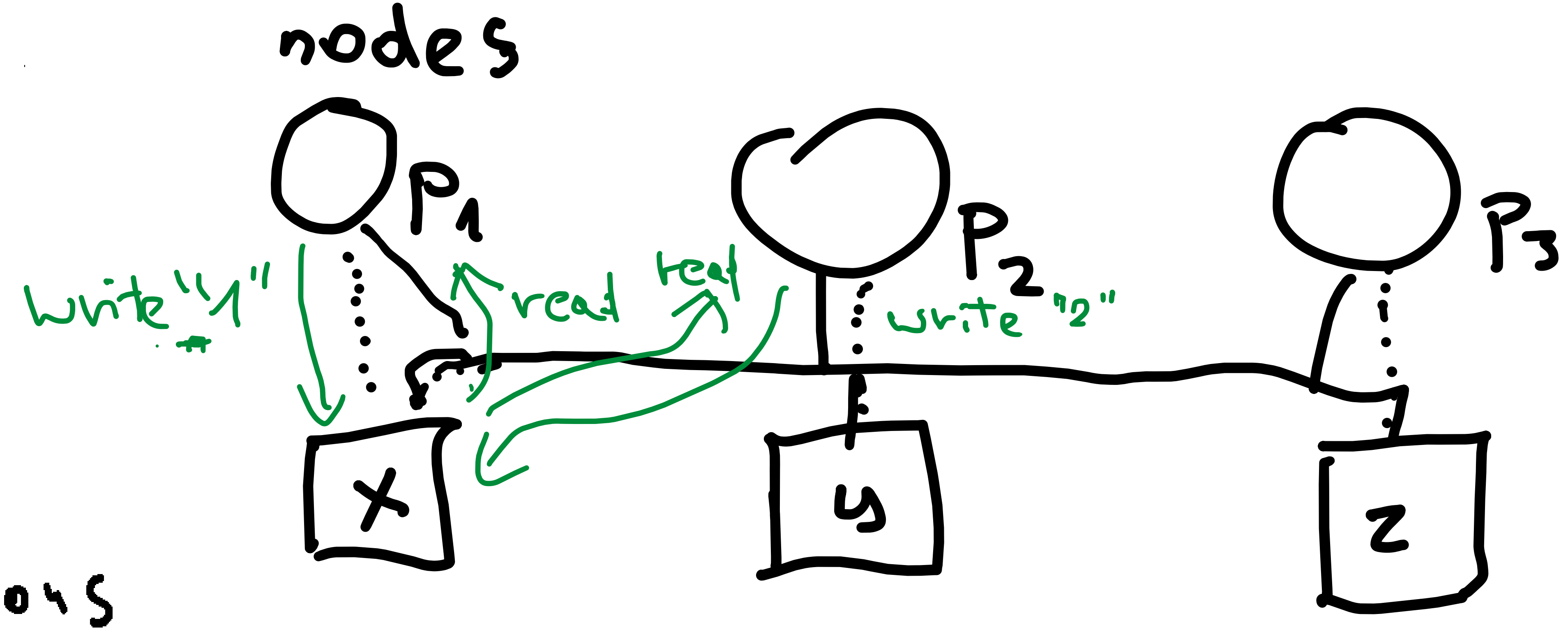
Operations

read

time / latency

no misunderstandings

Memory



Operations

write

time sequence

write 1	write „1"
read 1	write "2"
write 2	read "2"
read 2	read "2"

Countermeasures Hardware

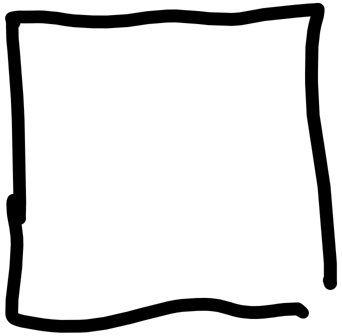
- test-and-set(R): $t := R$; $R := 1$; return t
- $x=1.$ fetch-and-add(R, x): $t := R$; $R := R + x$; return t
- compare-and-swap(R, x, y): if $R = x$ then $R := y$; return **true**; else return **false**; endif;

– load-link(R)/store-conditional(R, x): Load-link returns the current value of the specified register R. A subsequent store-conditional to the same register will store a new value x (and return **true**)

Mutual exclusion

<Entry> → *<Critical Section>* → *<Exit>* → *<Remaining Code>*

only 1 has control



101010....

Algorithm 5.3 Mutual Exclusion: Test-and-Set

Input: Shared register $R := 0$

<Entry>

R=1
r=0

repeat

$\quad r :=$ test-and-set(R)

until $r = 0$

r=1, R=1

<Critical Section>

...

<Exit>

$R := 0$

<Remainder Code>

...

green: P_1

red: P_2

1) unfair

2) test-and-set

Algorithm 5.3 Mutual Exclusion: Test-and-Set

Input: Shared register $R := 0$

<Entry>

repeat

  $r :=$ test-and-set(R)

until $r = 0$

<Critical Section>

…

<Exit>

$R := 0$

<Remainder Code>

…

$r=0$, $R=1$

green: P_1

red: P_2

Without hardware support

Algorithm 5.5 Mutual Exclusion: Peterson's Algorithm

Initialization: Shared registers W_0, W_1, Π, all initially 0.

Code for process p_i , $i = \{0, 1\}$ $i=1$

<Entry>

1: $W_i := 1$

2: $\Pi := 1 - i$ $1=0$

3: **repeat until** $\Pi = i$ or $W_{1-i} = 0$

<Critical Section>

4: ...

<Exit>

5: $W_i := 0$

<Remainder Code>

6: ...

$W_0 = 0$
$W_1 = 1$
$W_1 = 0$

$W_0 := 1$
$\Pi := 1$
$\Pi = 0 \vee W_1 = 0$

Π, W_0, W_1

Without hardware support

Algorithm 5.5 Mutual Exclusion: Peterson's Algorithm

Initialization: Shared registers W_0, W_1, Π, all initially 0.

Code for process p_i , $i = \{0, 1\}$

<Entry>

$W_i := 1$

$\Pi := 1 - i$

repeat until $\Pi = i$ or $W_{1-i} = 0$

<Critical Section>

. . .

<Exit>

$W_i := 0$

<Remainder Code>

. . .

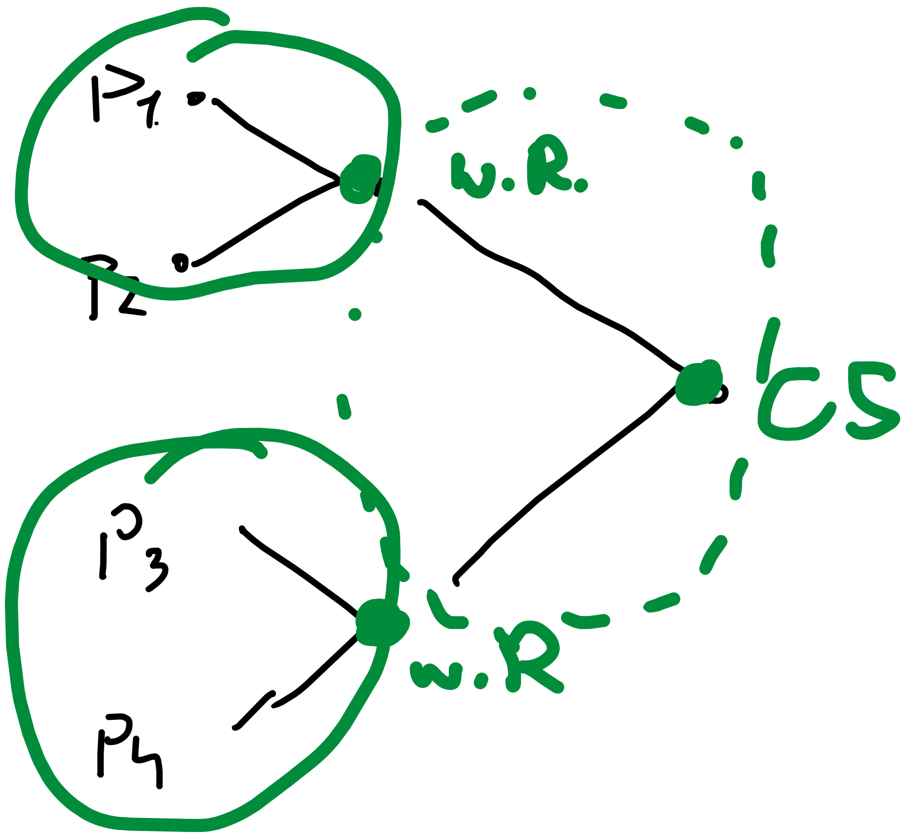
$W_1 = 1$
$\Pi = 0$
CS

$W_1 = 0$
$W_1 = 1$
$\Pi = 0$
Waiting

$W_0 = 1$
$\Pi = \cancel{1}\ 0$
waiting
CR

for 2 nodes!

Extension for an arbitrary number of processors



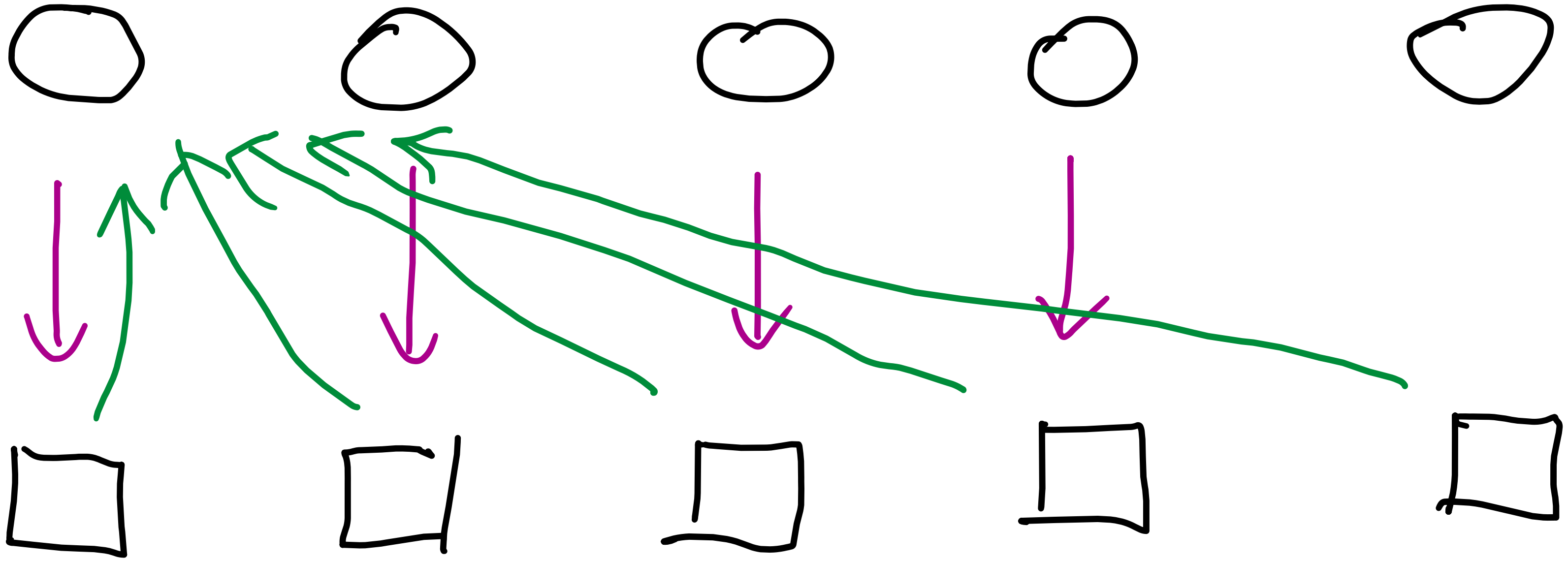
Save and Collect

processors

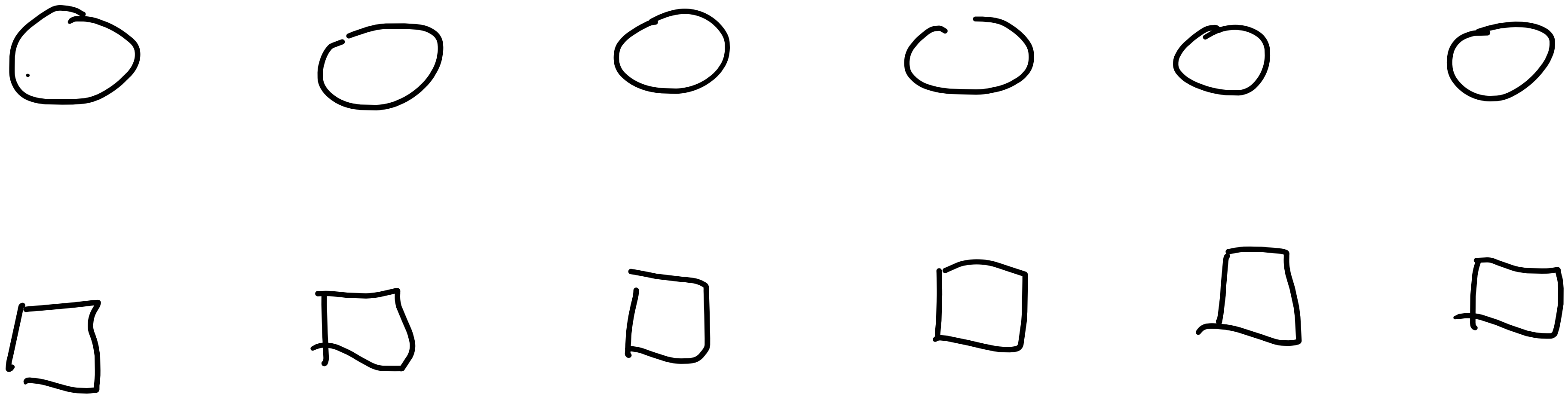
write

collect

registers:



Problem of the order of operations



Algorithm 5.8 Collect: Simple (Non-Adaptive) Solution

Operation STORE(val) (by process p_i) **:**

$R_i := val$

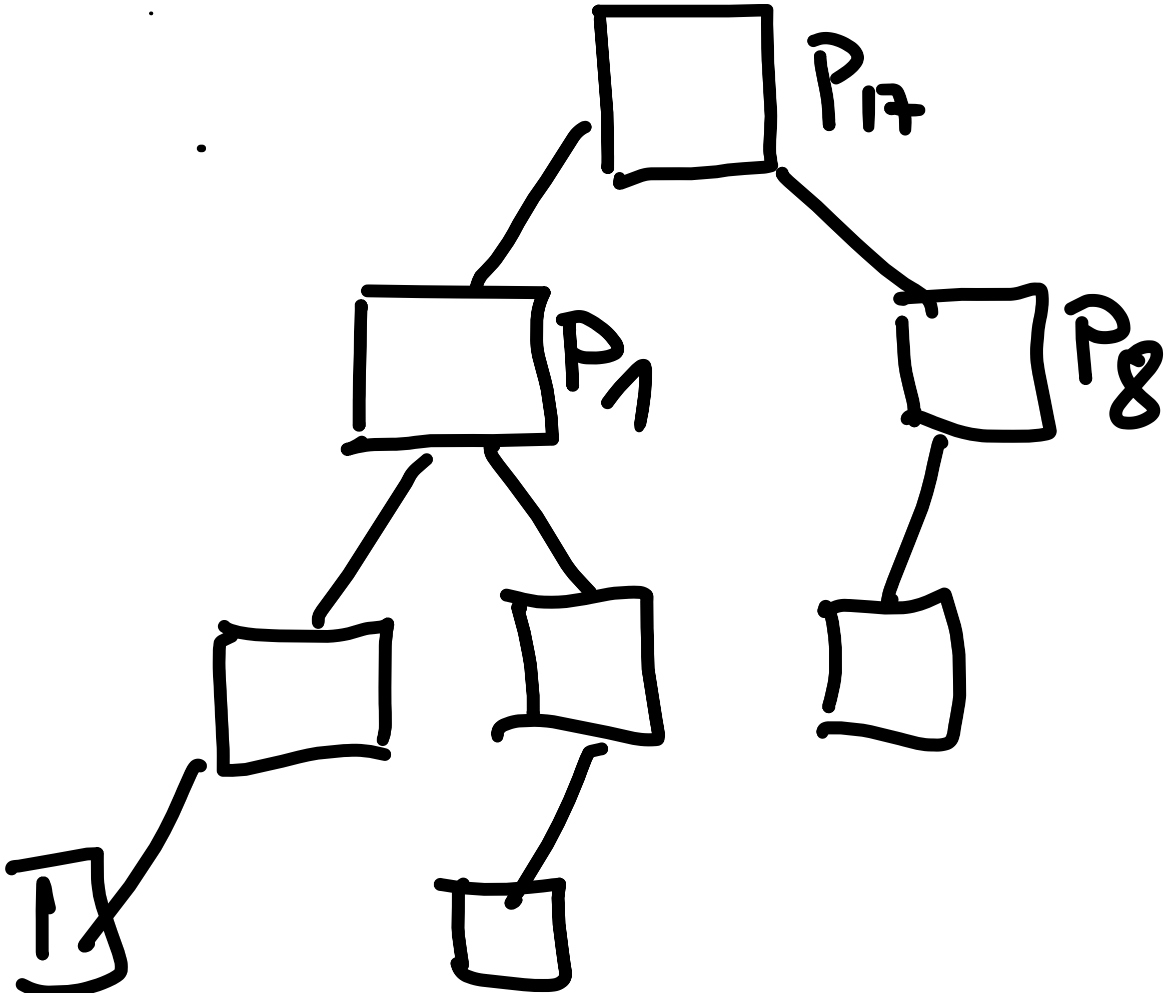
Operation COLLECT**:**

for $i := 1$ **to** n **do**

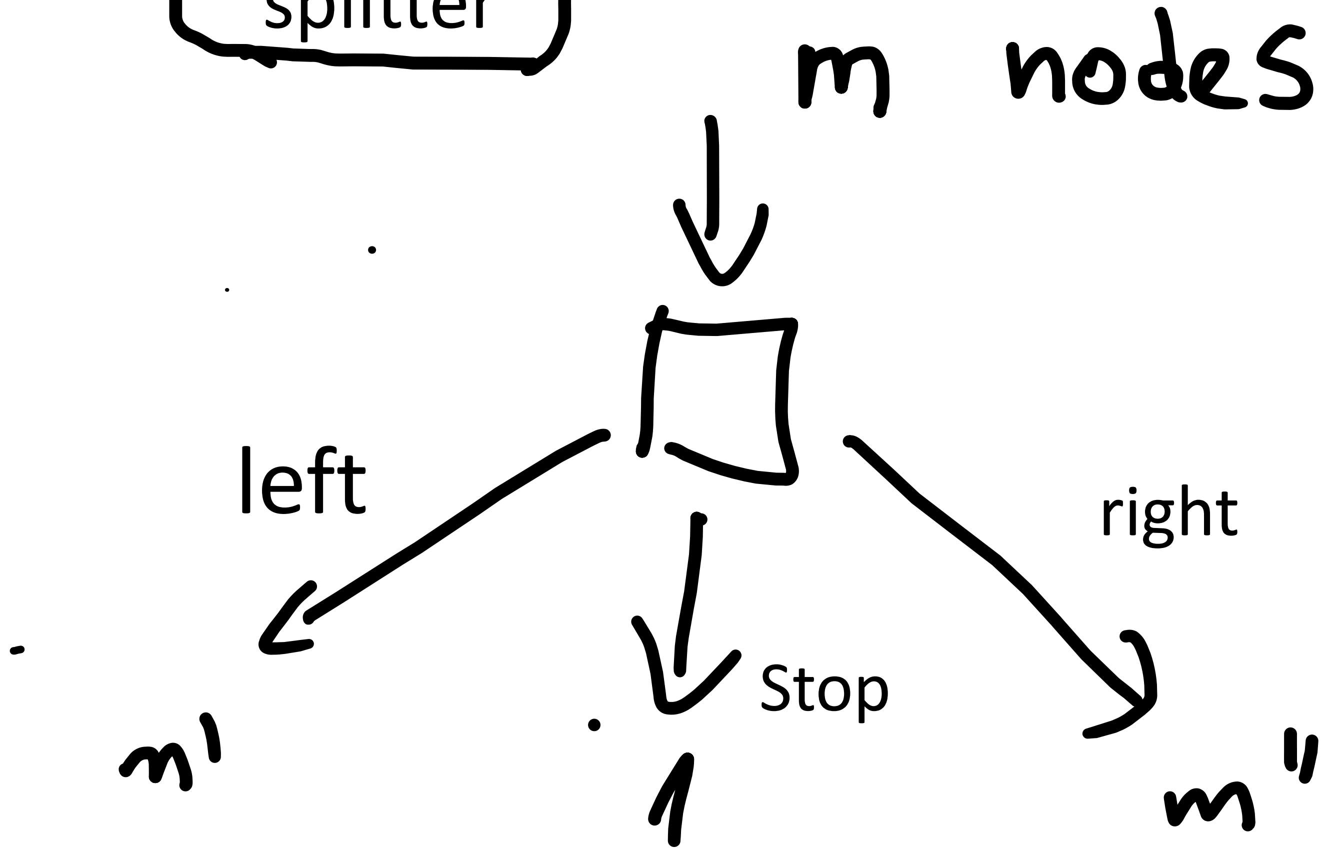
  $V(p_i) := R_i$

end for

time for collect if few „Store" operations splitting memory idea



splitter



$$m' + m'' + 1 = m$$

ideal: $m' \approx m''$

Algorithm 5.9 Splitter Code

Shared Registers: $X : \{\perp\} \cup \{1, \dots, n\}$; Y : boolean
Initialization: $X := \perp$; $Y :=$ false

Splitter access by process p_i:

```
X := i;
if Y then
   return right
else
   Y := true
   if X = i then
      return stop
   else
      return left
   end if
end if
```

last node that assigns $X := i$

Y=true ⇒ returns right

Y=false ⇒ returns stop

it does not return ‚left‘

Algorithm 5.9 Splitter Code

Shared Registers: $X : \{\perp\} \cup \{1, \ldots, n\}$; Y : **boolean**
Initialization: $X := \perp$; $Y :=$ false

Splitter access by process p_i:

```
if Y then
   return right
else
   Y := true
   if X = i then
      return stop
   else
      return left
   end if
end if
```

```
X := i;
```

p_0

$X := 0$, $Y = \cancel{\text{false}}$ true

return stop

P_1

$X := 1$

return right

Algorithm 5.9 Splitter Code

Shared Registers: $X : \{\perp\} \cup \{1, \ldots, n\}$; Y : **boolean**

Initialization: $X := \perp$; $Y :=$ false

Splitter access by process p_i**:**

```
1:  X := i;
2:  if Y then
3:      return right
4:  else
5:      Y := true
6:      if X = i then
7:          return stop
8:      else
9:          return left
10:     end if
11: end if
```

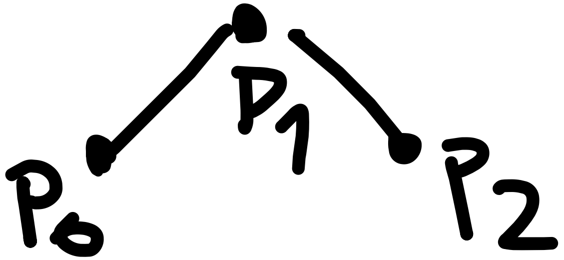
P_0

1. $X := 0$
2.
4.
5. $Y = true$
9. return left

P_1

$X := 1$

2.
4
5. $Y = true$
6.
7 return stop



1) no two nodes get "stop"

2) not all get „right"

3) not all get "left"

4) at a least 1 gets stop

not all get "right"

not all return "left"

¬ two nodes return "stop"

Algorithm 5.9 Splitter Code

Shared Registers: $X : \{\perp\} \cup \{1, \ldots, n\}$; Y : boolean
Initialization: $X := \perp$; $Y :=$ false

Splitter access by process p_i:

```
X := i;
if Y then
   return right
else
   Y := true
   if X = i then
      return stop
   else
      return left
   end if
end if
```

last node that assigns $X := i$

Y=true ⇒ returns right

Y=false ⇒ returns stop

it does not return ‚left'

Algorithm 5.9 Splitter Code

Shared Registers: $X : \{\perp\} \cup \{1, \ldots, n\}$; Y : boolean
Initialization: $X := \perp$; $Y :=$ false

Splitter access by process p_i:

1: $X := i$;
2: if Y then
3:     return right
4: else
5:     $Y :=$ true
6:     if $X = i$ then
7:         return stop
8:     else
9:         return left
10:     end if
11: end if

P0

X=0

2: Y=false

5: Y:=true

6: ~~i~~ if X=i

P1

X=1

2: Y=false

Y:=true

if y=i

either

X=1

or

X=0

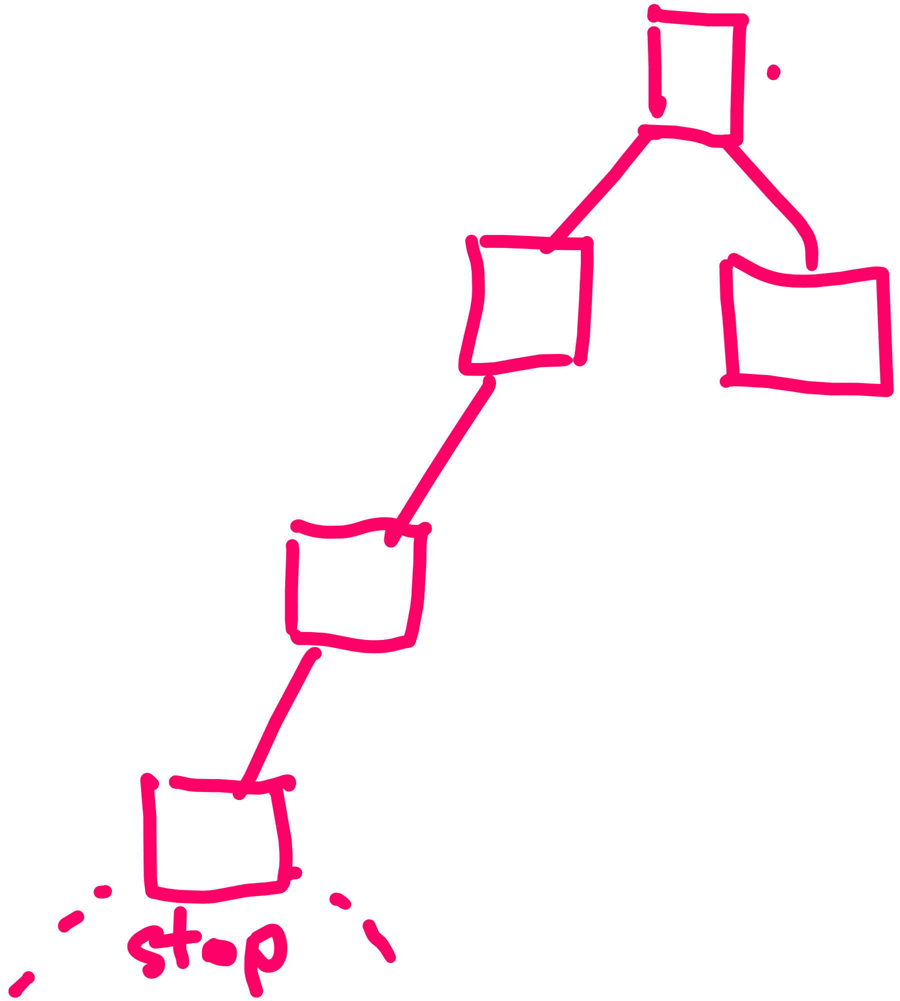
one true

one false

Algorithm 5.13 Adaptive Collect: Binary Tree Algorithm

Operation STORE(val) (by process p_i) **:**

$R_i := val$
if first STORE operation by p_i **then**
    $v :=$ root node of binary tree
    $\alpha :=$ result of entering splitter $S(v)$;
    $M_{S(v)} :=$ **true**
    while $\alpha \neq$ **stop do**
        if $\alpha =$ **left then**
            $v :=$ left child of v
        else
            $v :=$ right child of v
        end if
        $\alpha :=$ result of entering splitter $S(v)$;
        $M_{S(v)} :=$ **true**
    end while
    $Z_{S(v)} := i$
end if



Operation COLLECT**:**

Traverse marked part of binary tree:

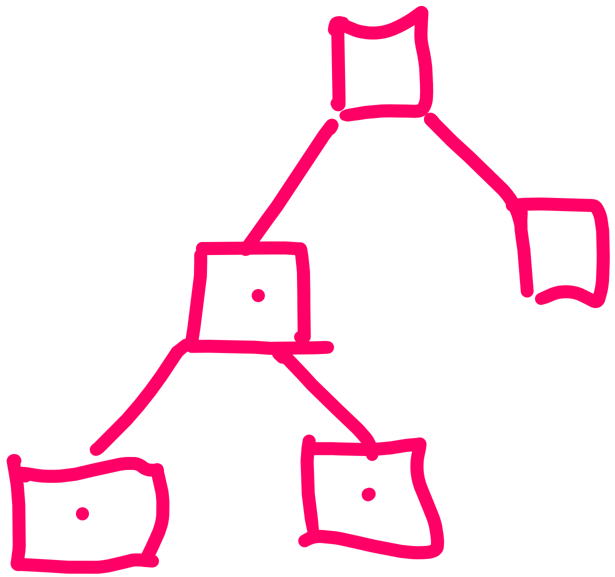
for all marked splitters S **do**

  if $Z_S \neq \perp$ **then**

    $i := Z_S$; $V(p_i) := R_i$

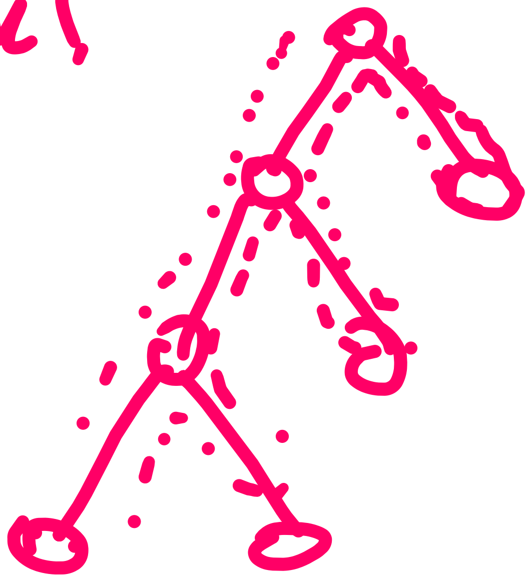
  end if

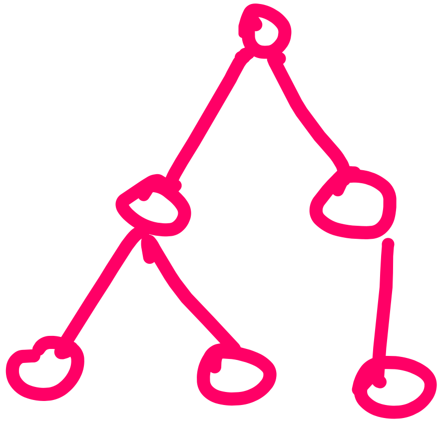
end for



space ok. tree with $\leq 2k$ node
for k active node

time
ok!





Splitter matrix:

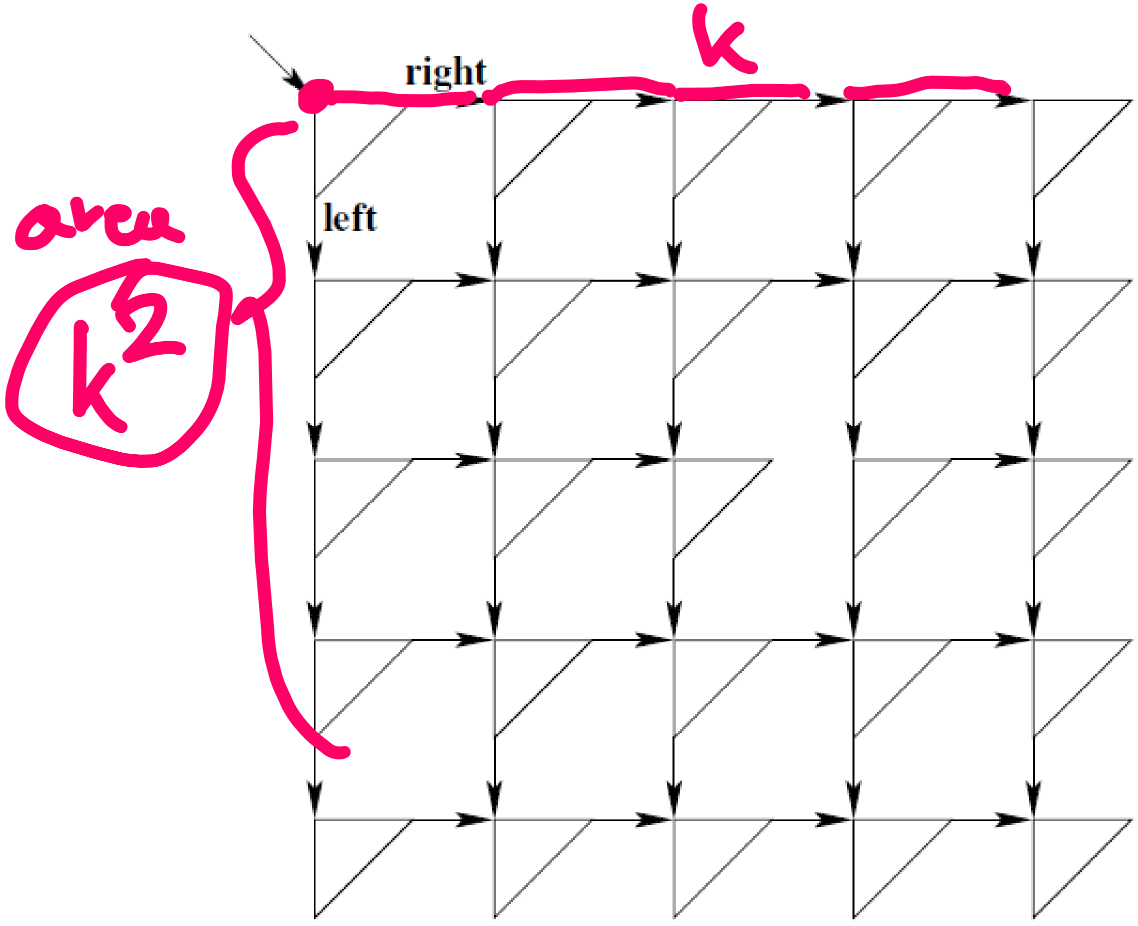


Figure 5.15: 5×5 Splitter Matrix

problem!
unbalanced trees

The

End

next episode:

shared objects