An introduction to the workspace

Richard Smith

Coming up ...

- A look at what goes inside a workspace
- A look at how the workspace is managed
- Why?
  - I've been asked for "how it works" presentations
  - It *really* affects performance
  - We've made it fast, but sometimes tuning can help further

What you are about to see is based on the way Dyalog APL actually works.

Some dramatic licence has been taken and sequences have been shortened for simplicity.

The workspace

A big contiguous block of memory which the interpreter asks the OS to allocate.

The workspace

The interpreter manages what is in it.

DYALOG

The workspace

The interpreter tries to keep the workspace small.

The workspace

The workspace shrinks and grows from time to time, but never gets bigger than MAXWS.

Workspace allocation

The interpreter *reserves* MAXWS bytes in the computer's address space to keep the range free.

Workspace allocation

The interpreter *reserves* MAXWS bytes in the computer's address space to keep the range free. But it initially only *allocates* a fraction of that.

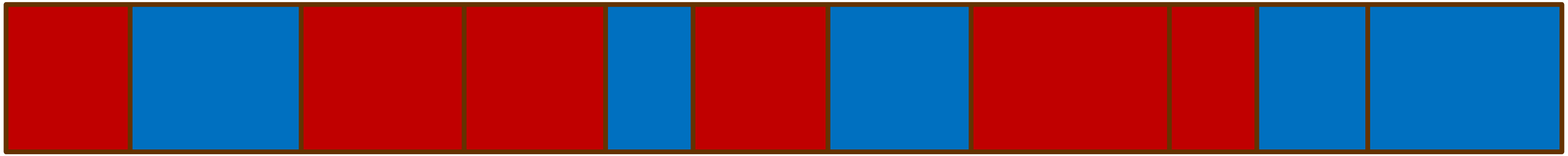
What goes into the workspace?

Pretty much everything:

- Arrays.
- Symbols (names).
- Functions.
- The APL stack.
- ... etc.

All of these things are made up of Pockets.

Pockets

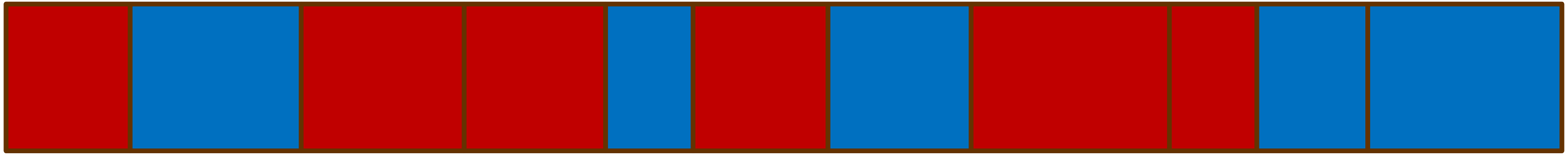


In the allocated part of the workspace there are:

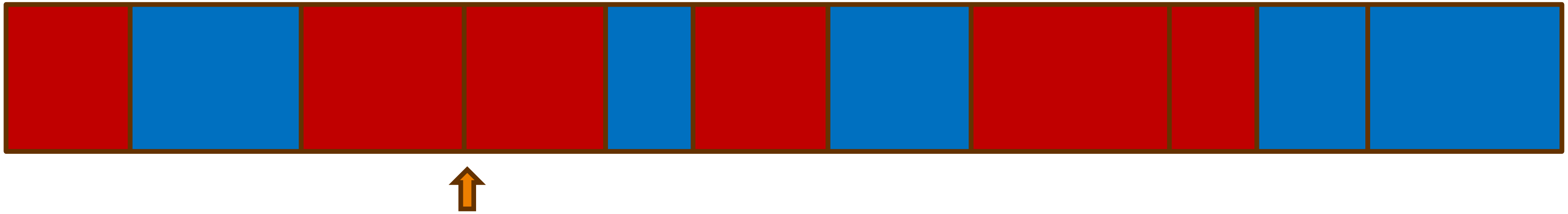
- FREE POCKETS.
- ALLOCATED POCKETS.

… and there lots of types of allocated pocket – but more on that later.

Pocket allocation algorithm

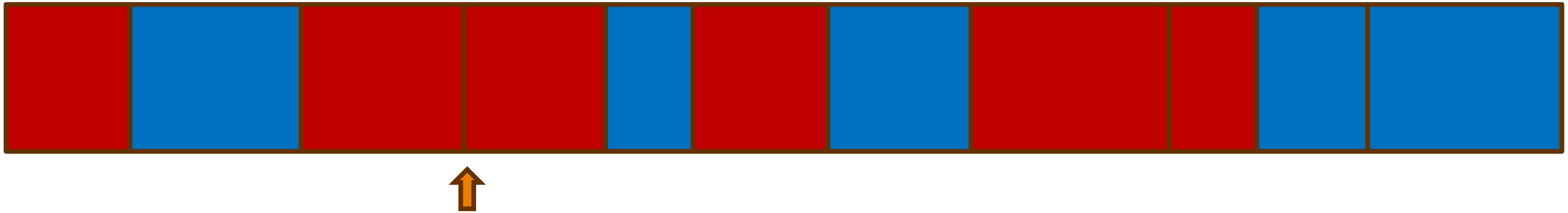


Pocket allocation algorithm



Starting at the pocket after the previous allocation:

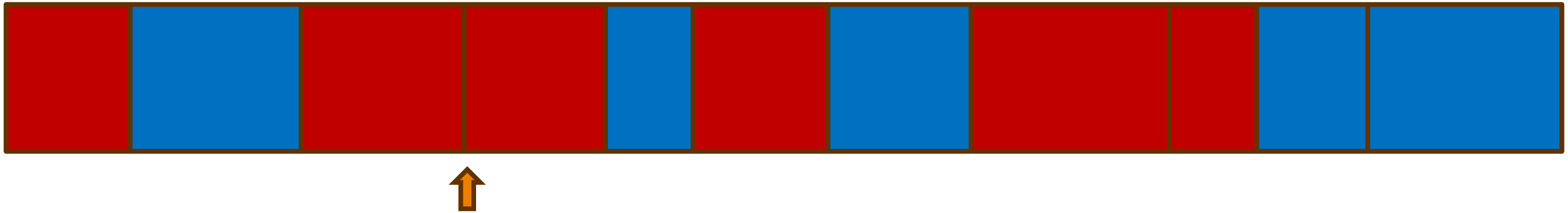
Pocket allocation algorithm



Starting at the pocket after the previous allocation:

- If it is free and big enough: allocate at that point, and anything left over becomes a new free pocket.

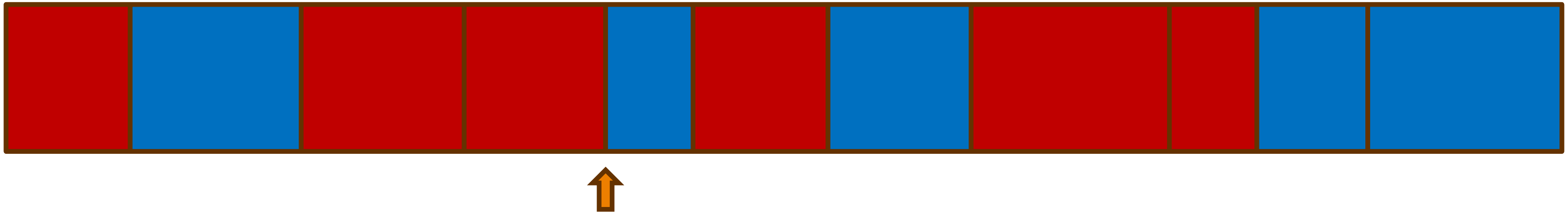
Pocket allocation algorithm



Starting at the pocket after the previous allocation:

- If it is free and big enough: allocate at that point, and anything left over becomes a new free pocket.
- Otherwise: skip to the next pocket and try again.

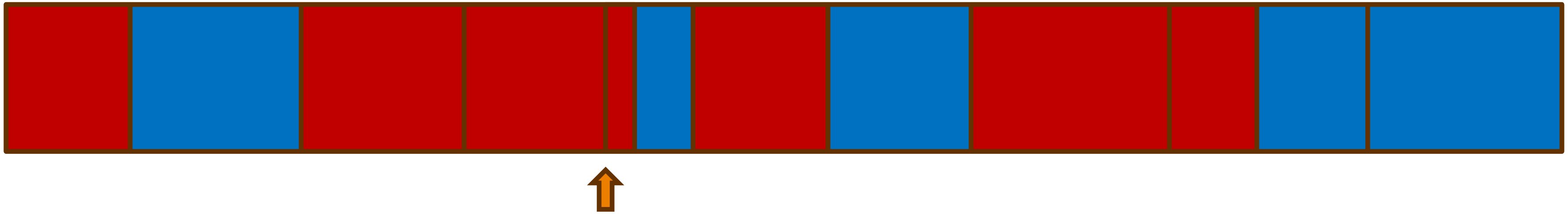
Pocket allocation algorithm



Starting at the pocket after the previous allocation:

- If it is free and big enough: allocate at that point, and anything left over becomes a new free pocket.
- Otherwise: skip to the next pocket and try again.

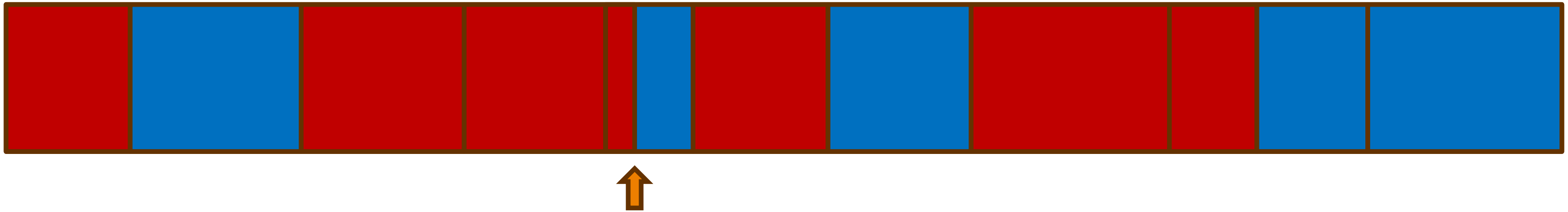
Pocket allocation algorithm



Starting at the pocket after the previous allocation:

- If it is free and big enough: allocate at that point, and anything left over becomes a new free pocket.
- Otherwise: skip to the next pocket and try again.

Pocket allocation algorithm



Starting at the pocket after the previous allocation:

- If it is free and big enough: allocate at that point, and anything left over becomes a new free pocket.
- Otherwise: skip to the next pocket and try again.

Next time, restart from the next pocket.

Pocket allocation (and deallocation)

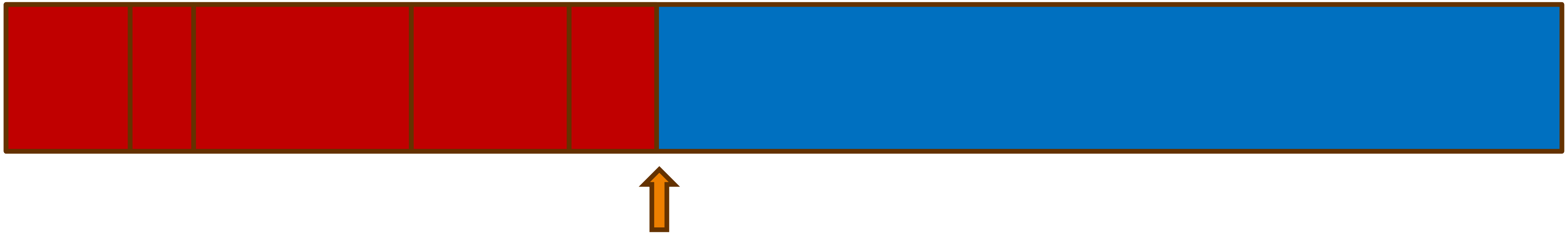
Pocket allocation (and deallocation)

Pocket allocation (and deallocation)

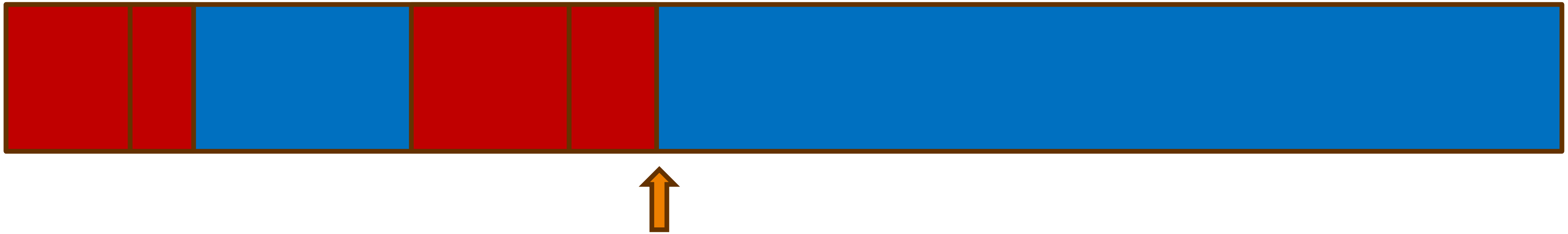
Pocket allocation (and deallocation)

Pocket allocation (and deallocation)

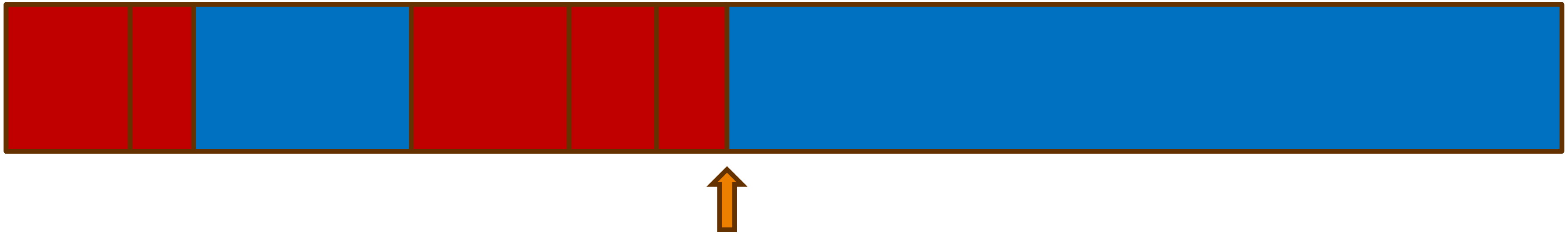
Pocket allocation (and deallocation)



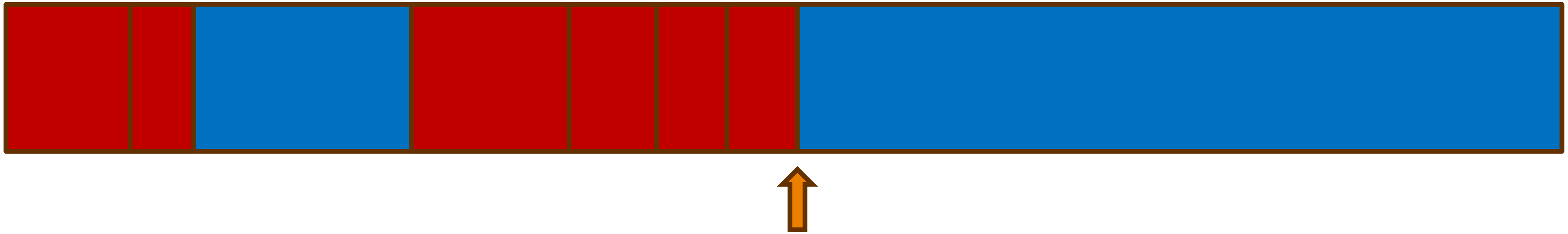
Pocket allocation (and deallocation)



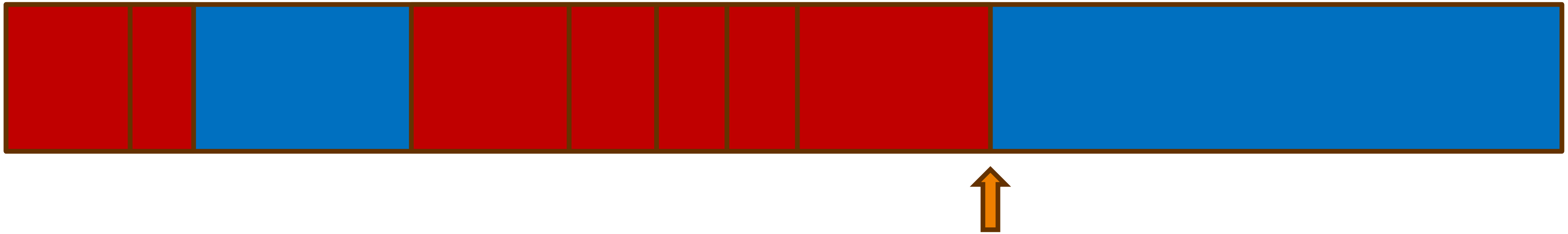
Pocket allocation (and deallocation)



Pocket allocation (and deallocation)

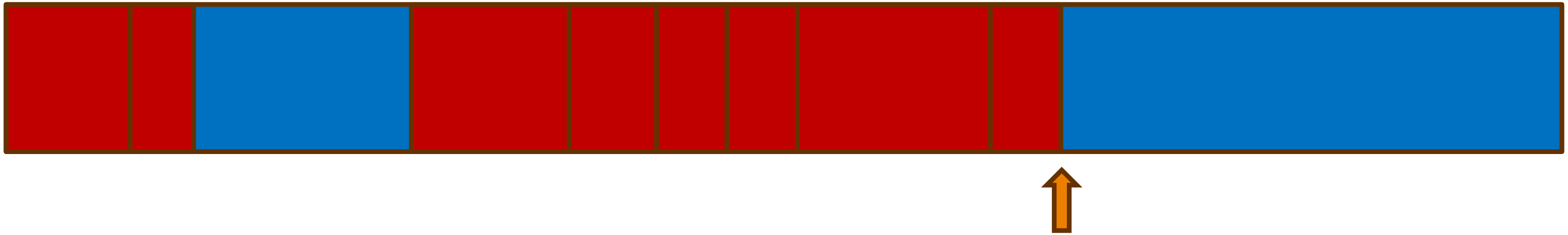


Pocket allocation (and deallocation)

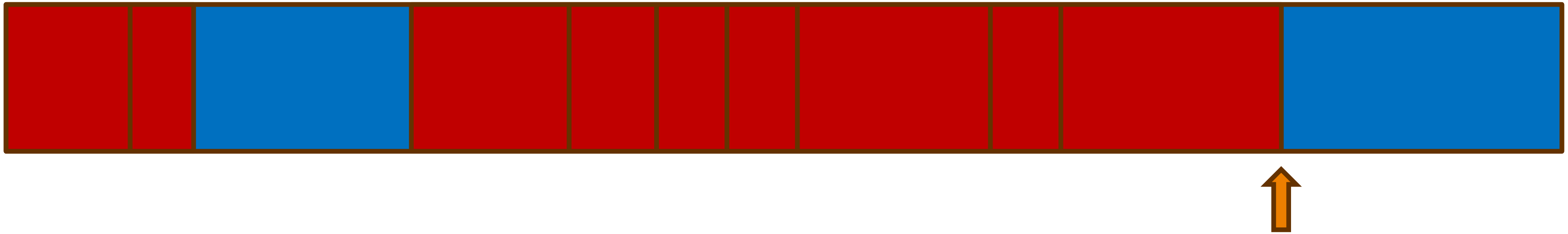


DYALOG

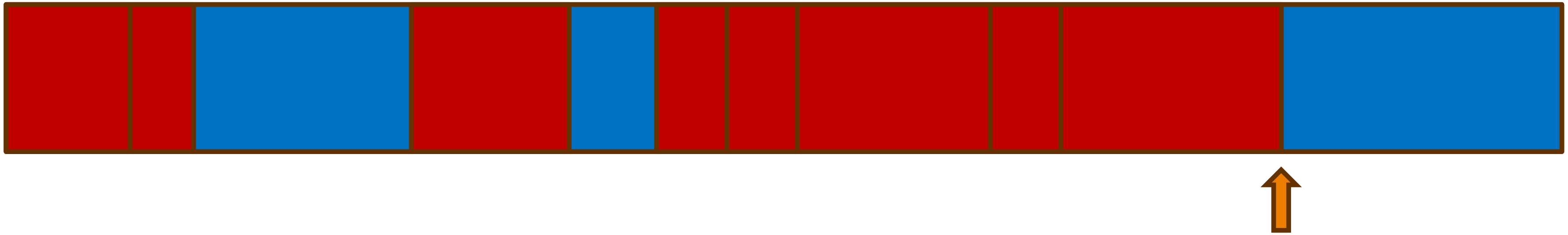
Pocket allocation (and deallocation)



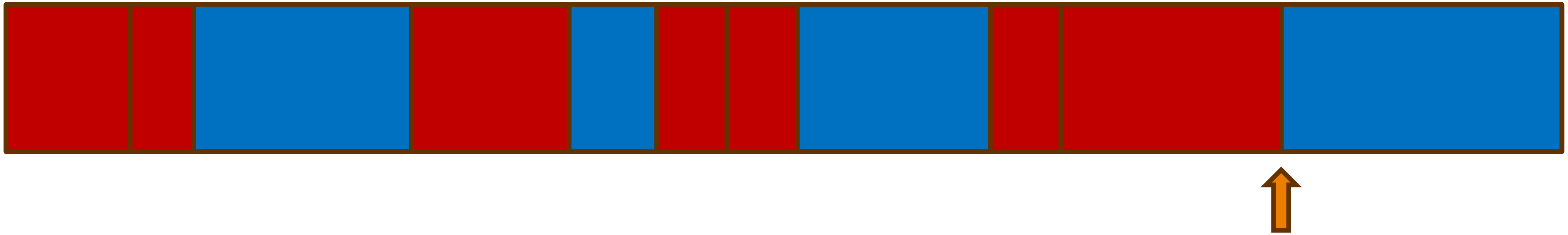
Pocket allocation (and deallocation)



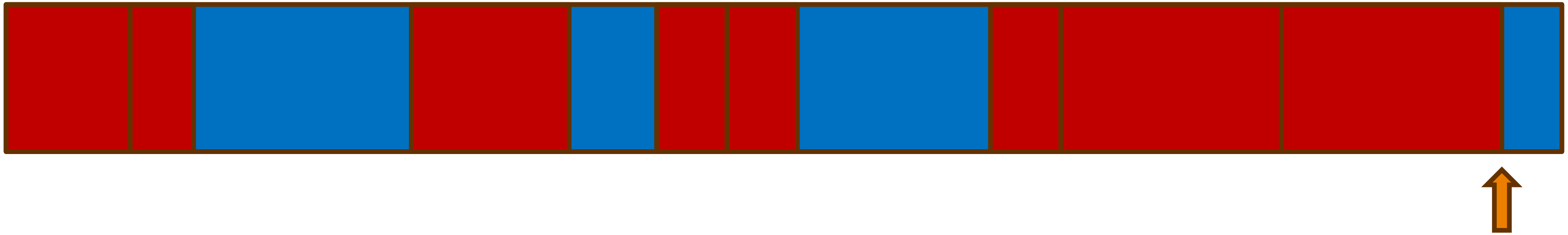
Pocket allocation (and deallocation)



Pocket allocation (and deallocation)



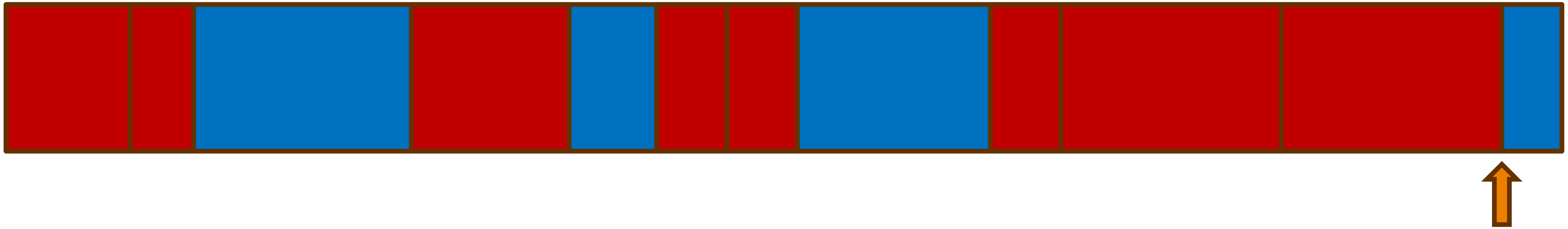
Pocket allocation (and deallocation)



Next allocation request

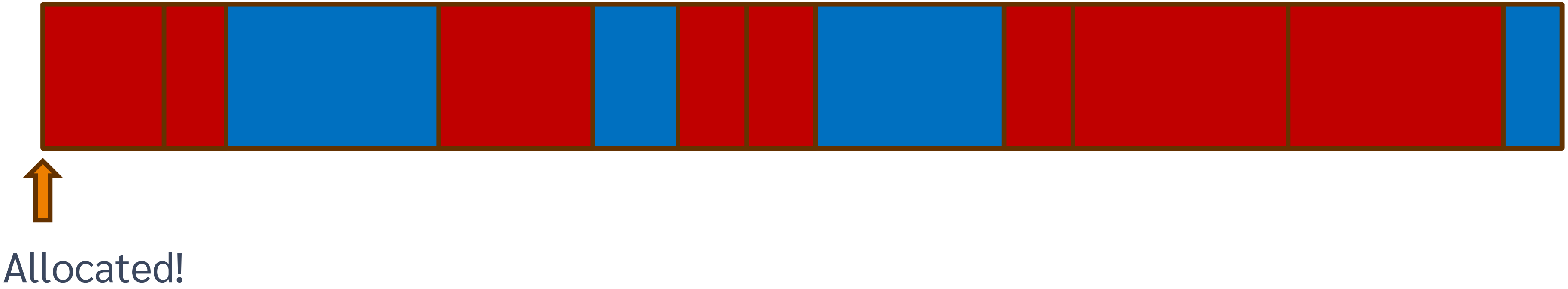
DYALOG

Pocket allocation (and deallocation)

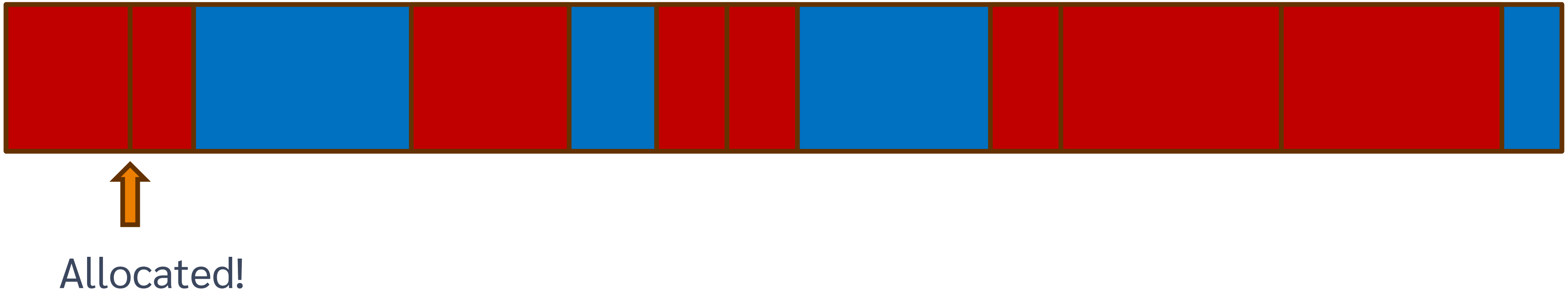


Too small!

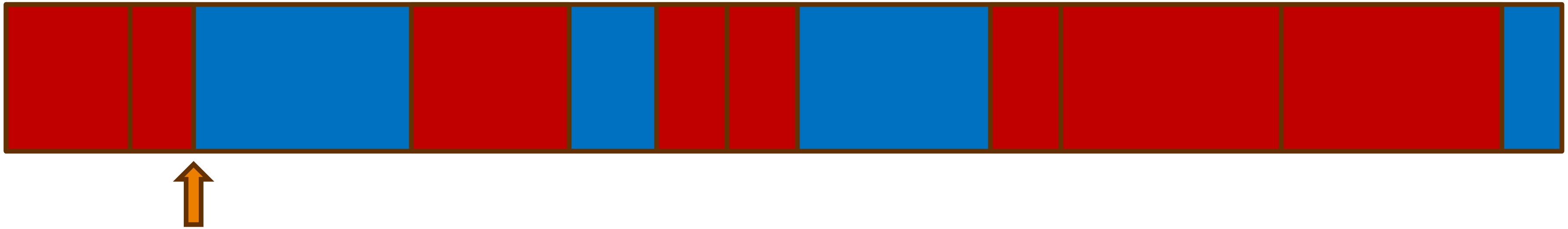
Pocket allocation (and deallocation)



Pocket allocation (and deallocation)

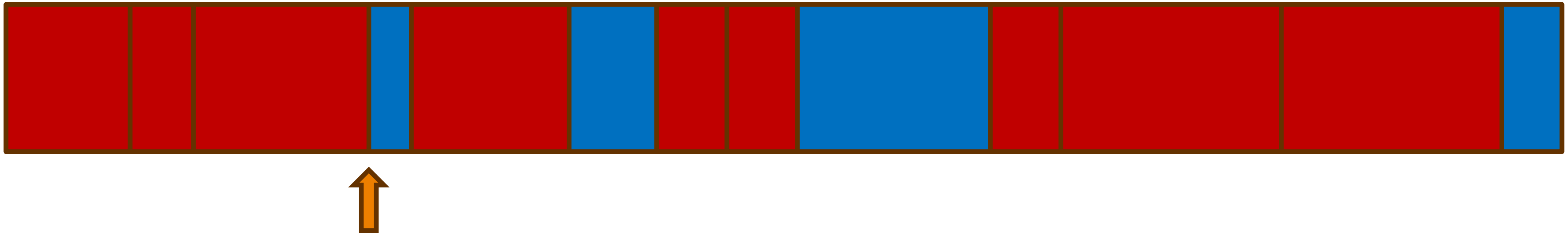


Pocket allocation (and deallocation)



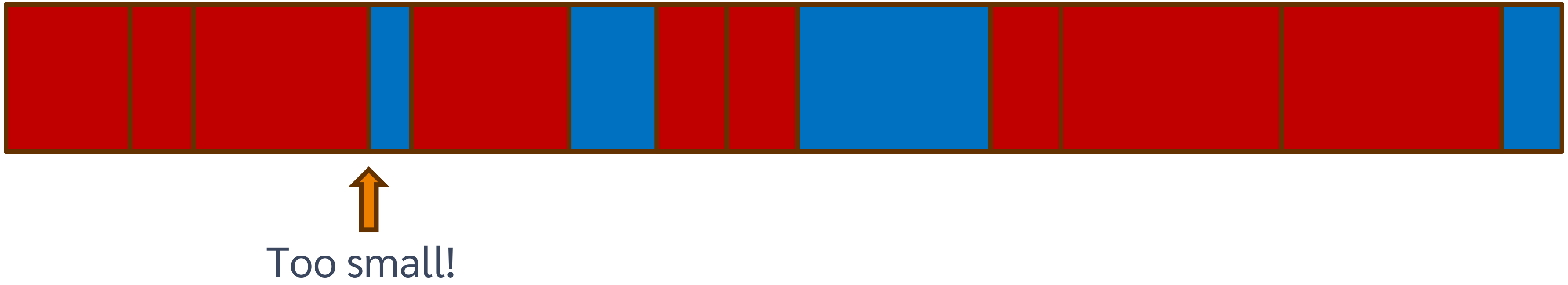
Will fit!!

Pocket allocation (and deallocation)

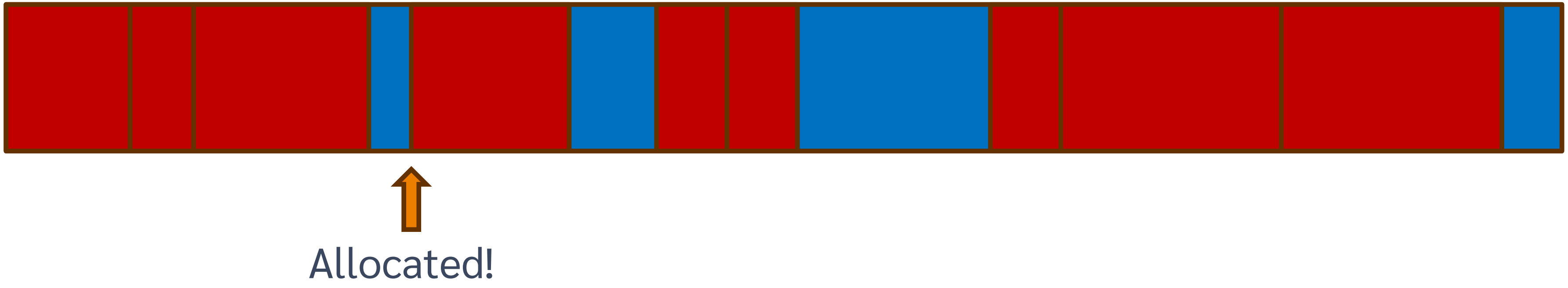


Next allocation request

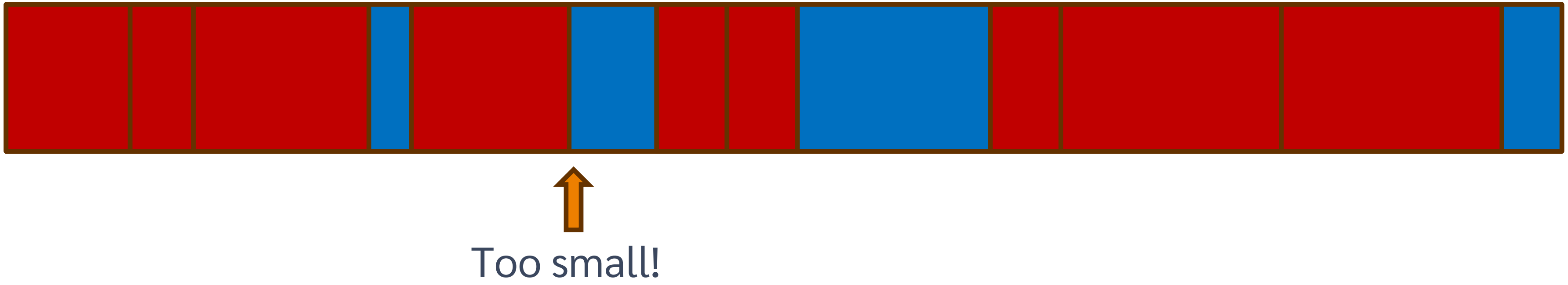
Pocket allocation (and deallocation)



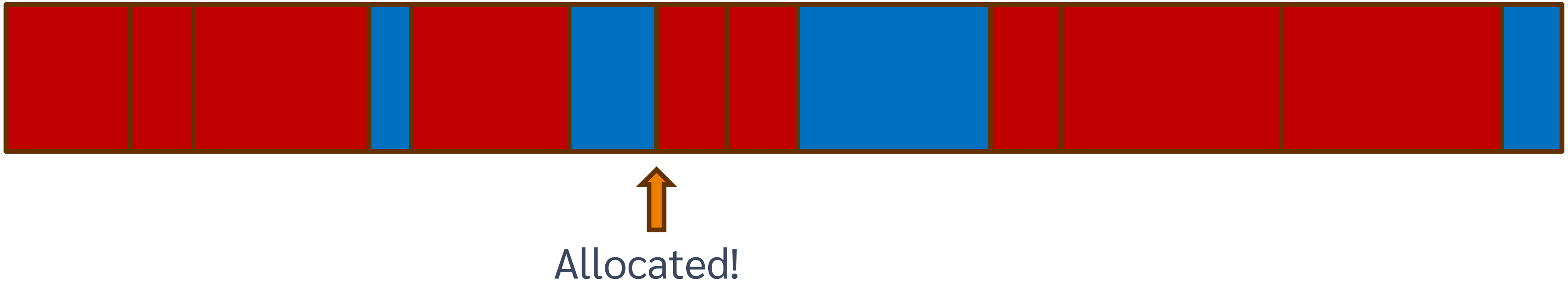
Pocket allocation (and deallocation)



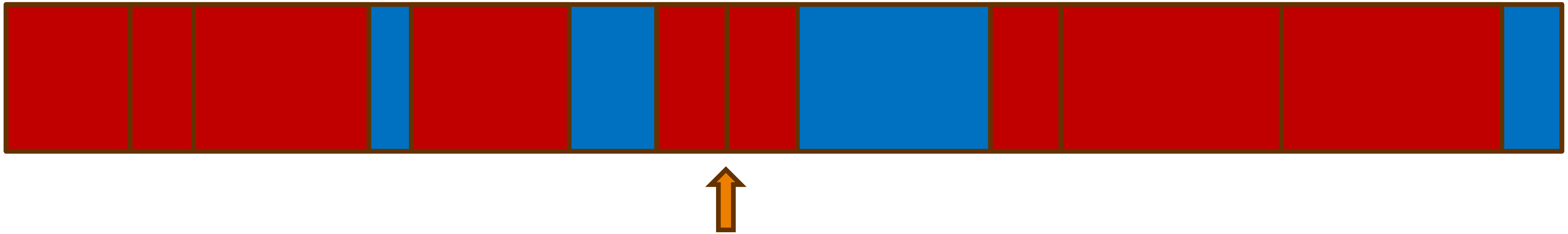
Pocket allocation (and deallocation)



Pocket allocation (and deallocation)

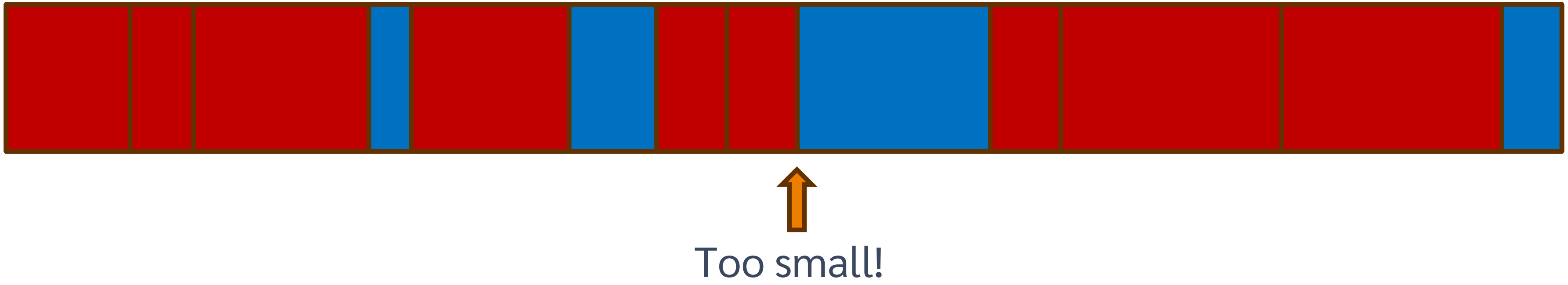


Pocket allocation (and deallocation)



Allocated!

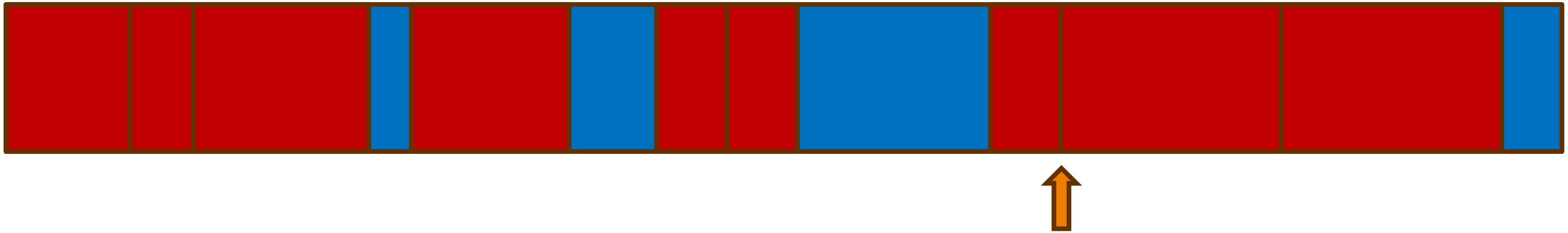
Pocket allocation (and deallocation)



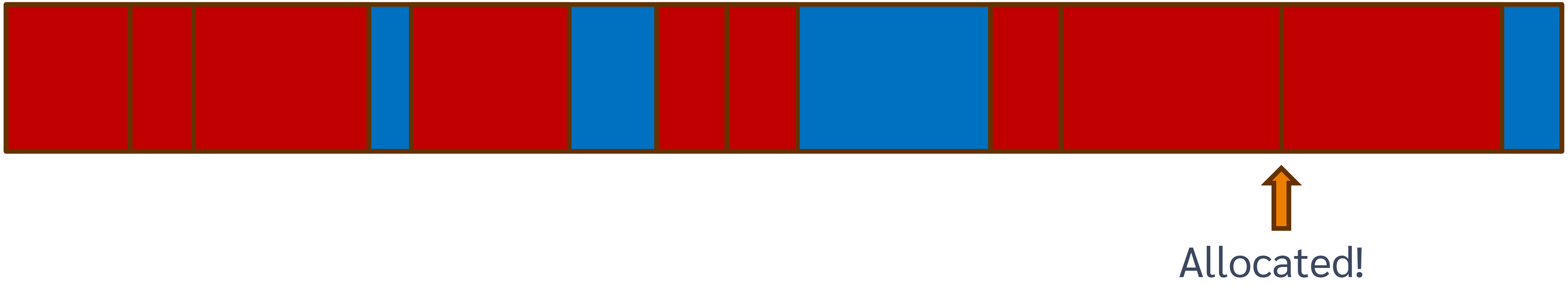
DYALOG

Pocket allocation (and deallocation)

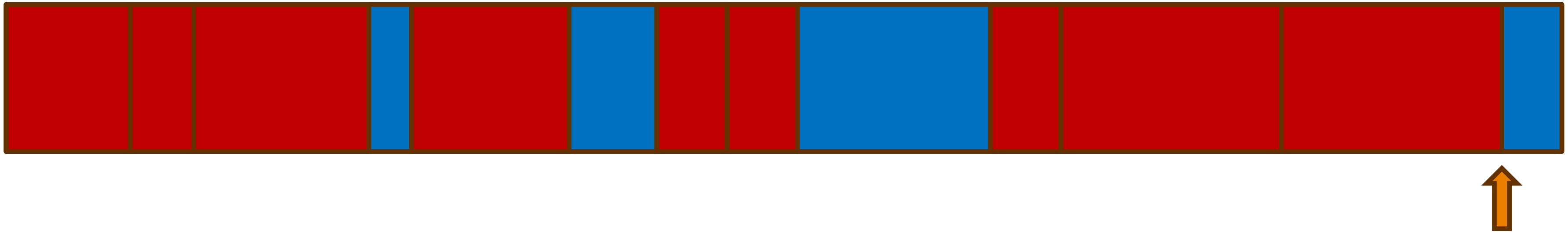
Pocket allocation (and deallocation)



Pocket allocation (and deallocation)

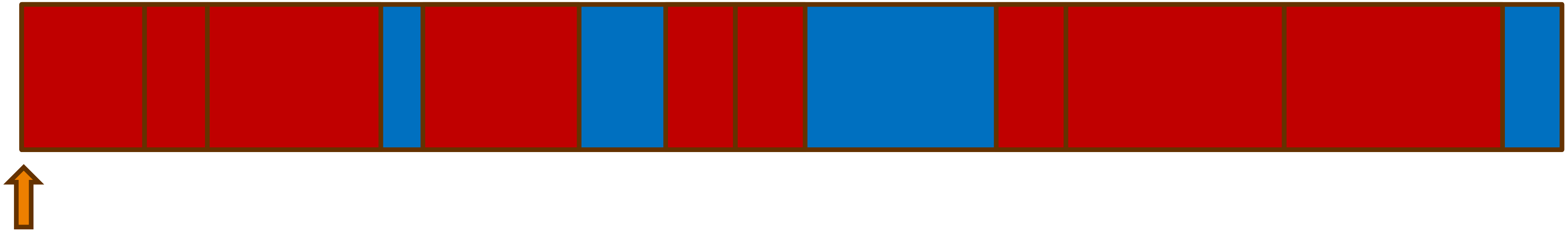


Pocket allocation (and deallocation)



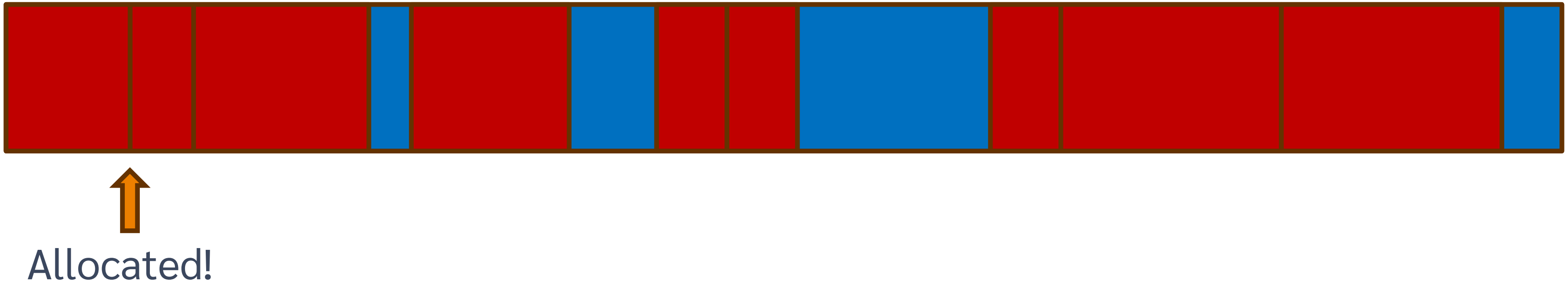
DYALOG

Pocket allocation (and deallocation)

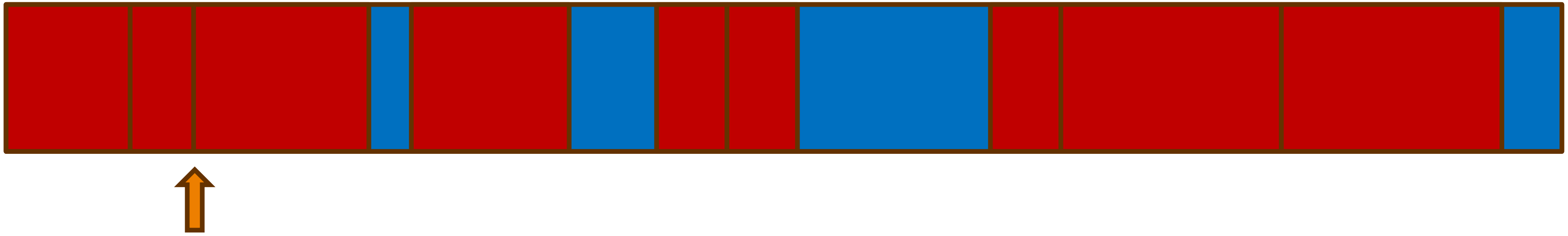


Allocated!

Pocket allocation (and deallocation)

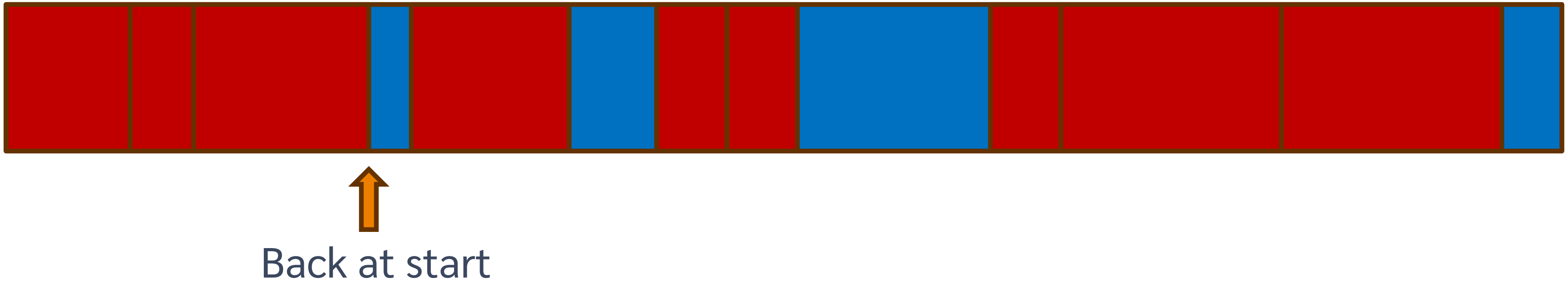


Pocket allocation (and deallocation)

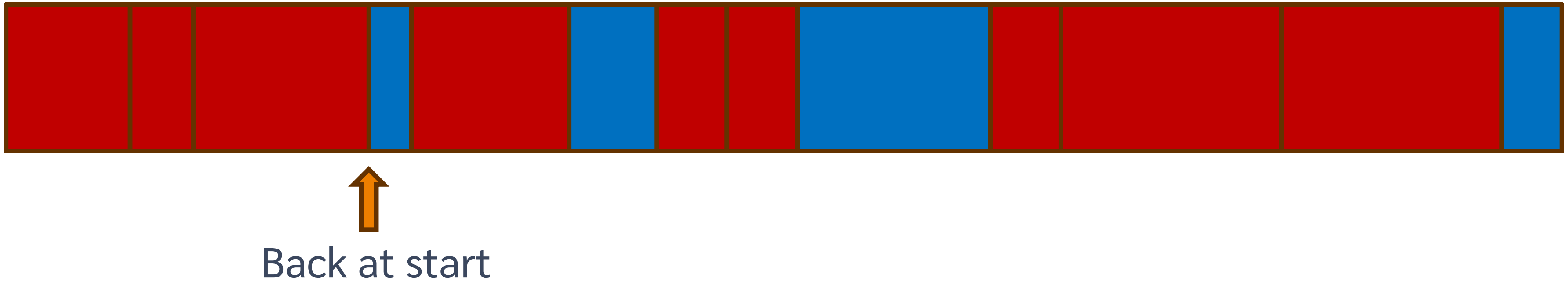


Allocated!

Pocket allocation (and deallocation)

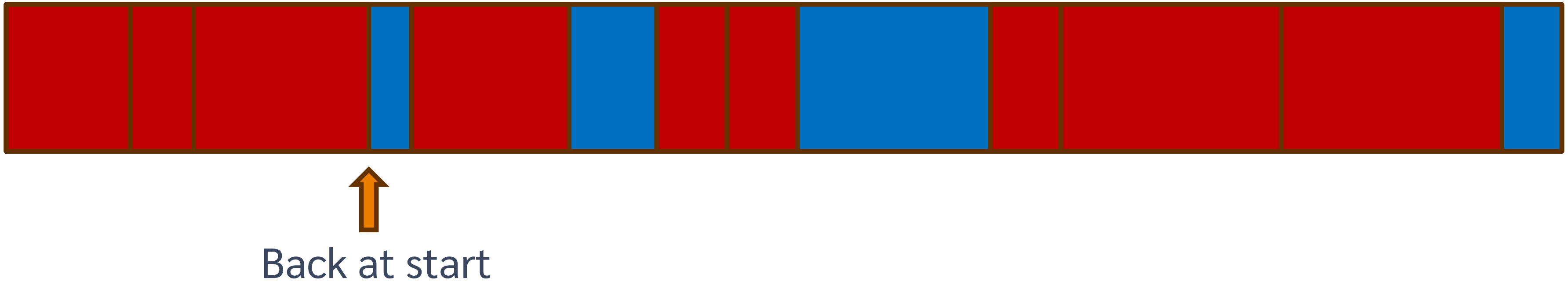


Pocket allocation (and deallocation)



Space could not be allocated.

Pocket allocation (and deallocation)



Space could not be allocated.
Not necessarily a WSFULL… we'll see what happens next later.

A look inside some pockets

Free pockets

Free pockets

Allocated pockets

Allocated pockets

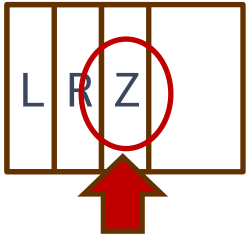
Allocated pockets

1 word long (64-bits).
Includes the main pocket type.
There are 15 major pocket types in all.

Arrays

A simple array

A simple array - ⍳8

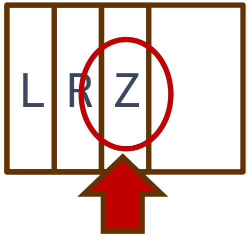


Simple array pocket type.

Rank 1.

A simple array - ⍳8

NB – array contains:
1 2 3 4 5 6 7 8

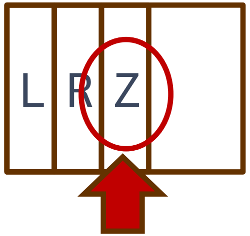


Simple array pocket type.

Rank 1.

A simple array - ⍳8

NB – array contains:
1 2 3 4 5 6 7 8

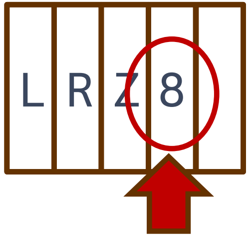


Simple array pocket type.

Rank 1.

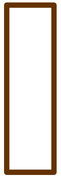
8-bit integers.

A simple array - ⍳8



Shape 8.

A simple array - ⍳8



A simple array - ⍳8

A simple array - ⍳ 8

1	2	3	4	5	6	7	8

8 × 8 bits = 1 word

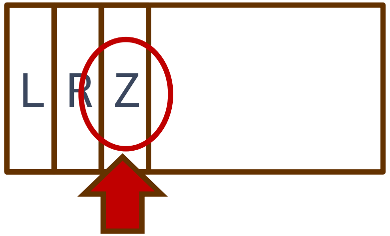
A simple array - ⍳ 8

1
2
3
4
5
6
7
8

A simple array - ⍳8

A simple array

A simple array –(⍳7),100000

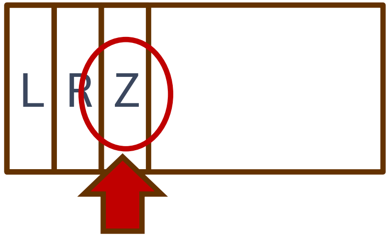


Simple array pocket type.

Rank 1.

32-bit integers.

A simple array -(⍳7),100000

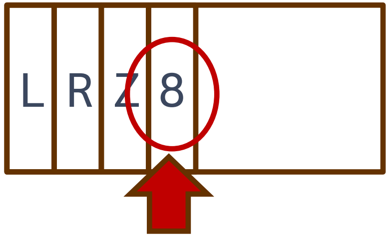


Simple array pocket type.

Rank 1.

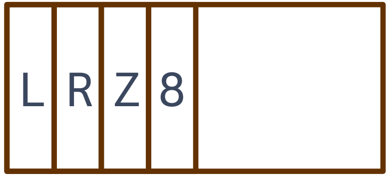
32-bit integers.

A simple array –(⍳7),100000



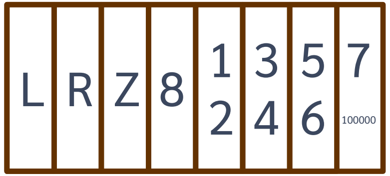
Shape 8.

A simple array `-(⍳7),100000`



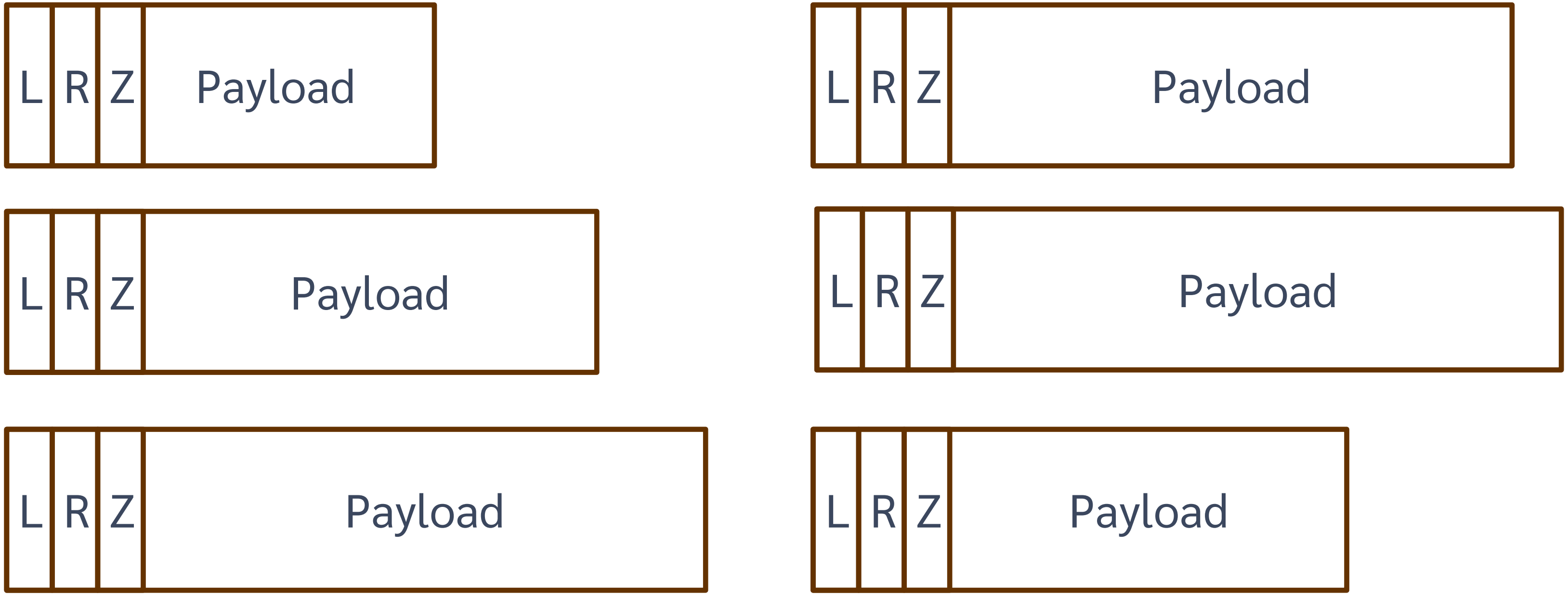
Each element is now 32-bit, rather than 8-bit before.

A simple array – (⍳7),100000

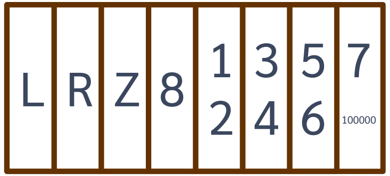


Each element is now 32-bit, rather than 8-bit before.
8 x 32 bits = 4 words.

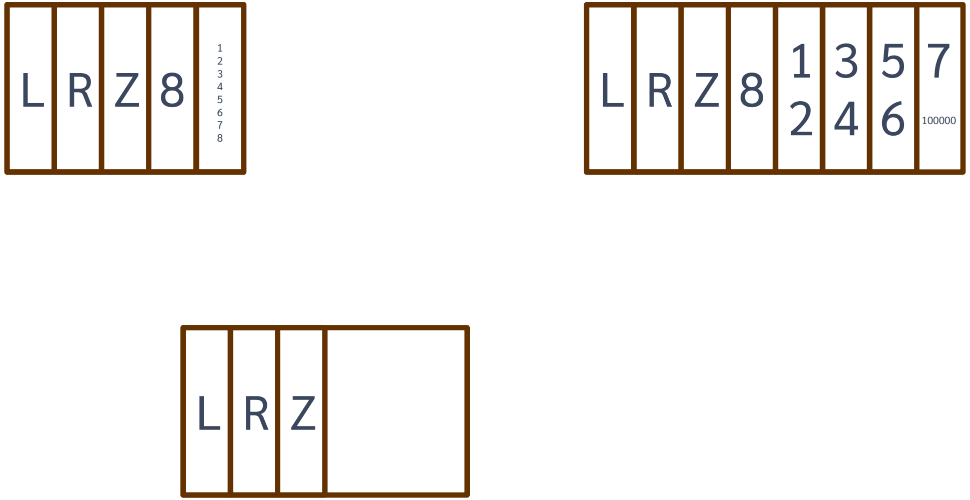
A non-simple array: multiple pockets



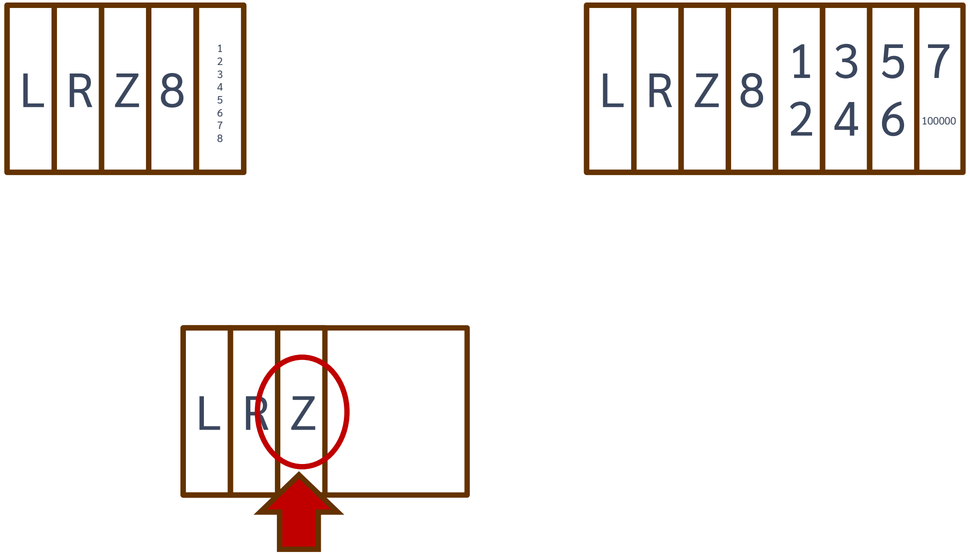
```
(⍳8)((⍳7),100000)
```



```
(⍳8)((⍳7),100000)
```

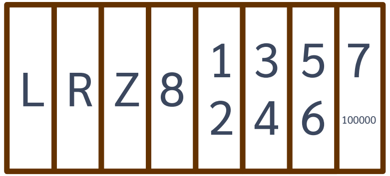


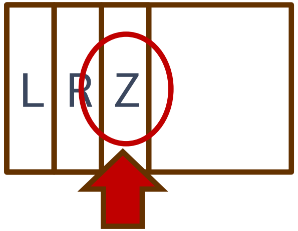
(⍳8)((⍳7),100000)



“Non-simple” array pocket type.

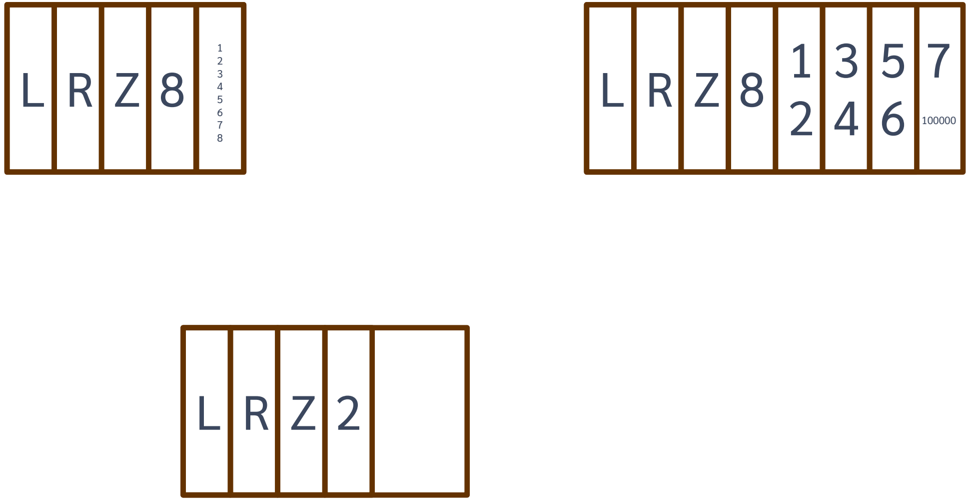
```
(⍳8)((⍳7),100000)
```



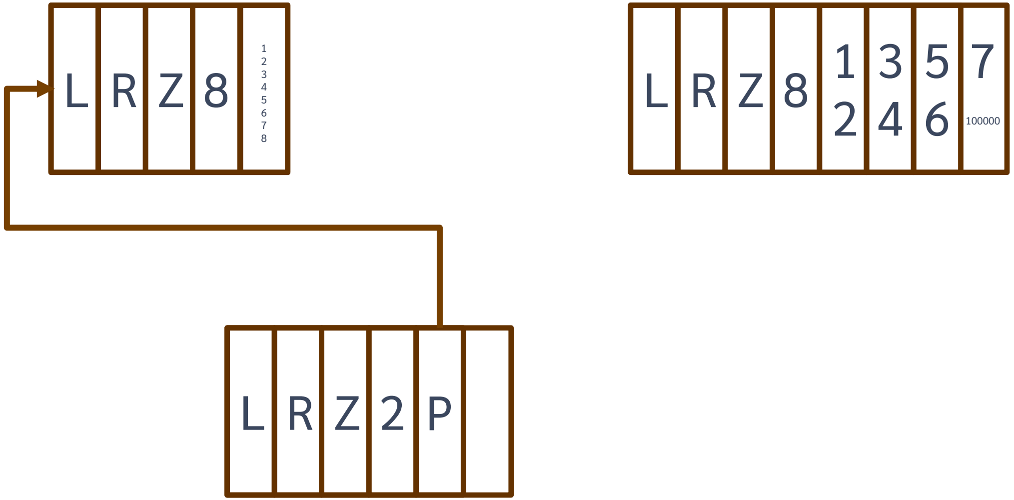


Rank 1.

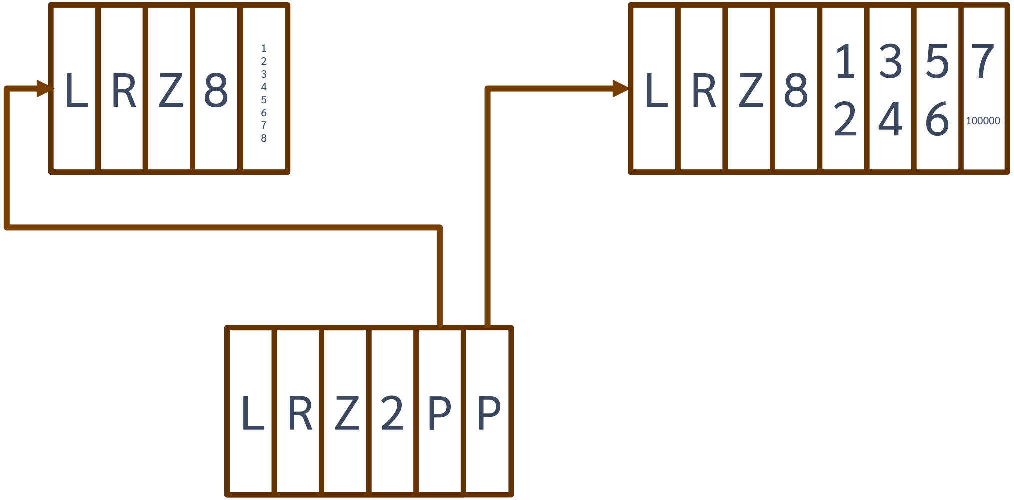
```
(⍳8)((⍳7),100000)
```



(⍳8)((⍳7),100000)

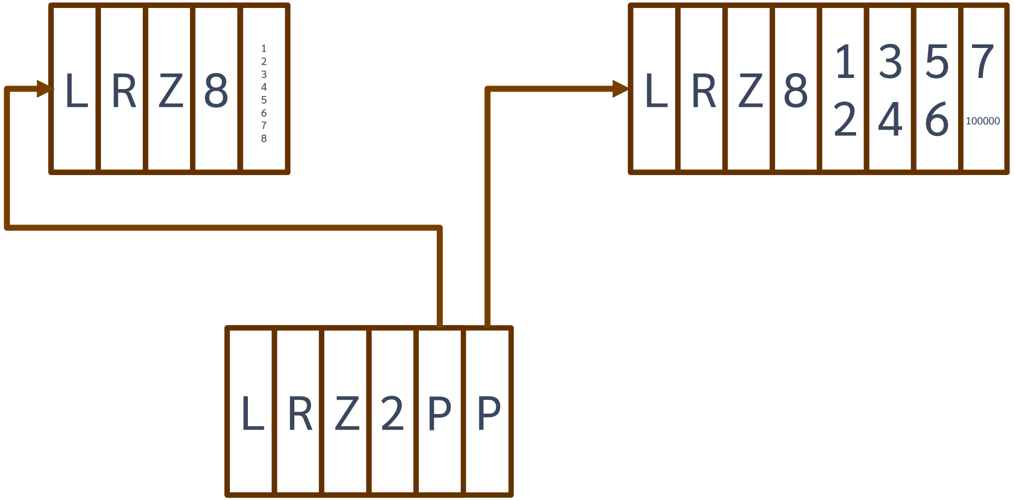


(⍳8)((⍳7),100000)

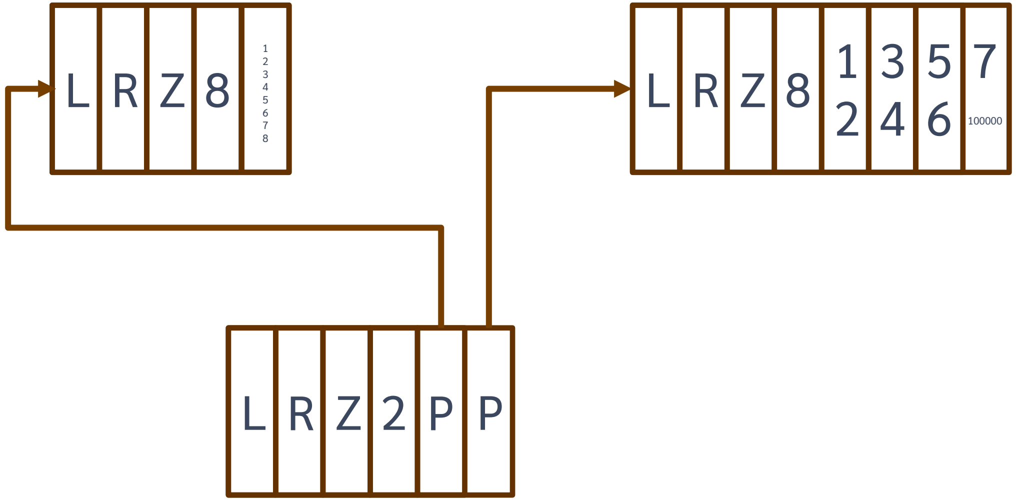


Other pocket types

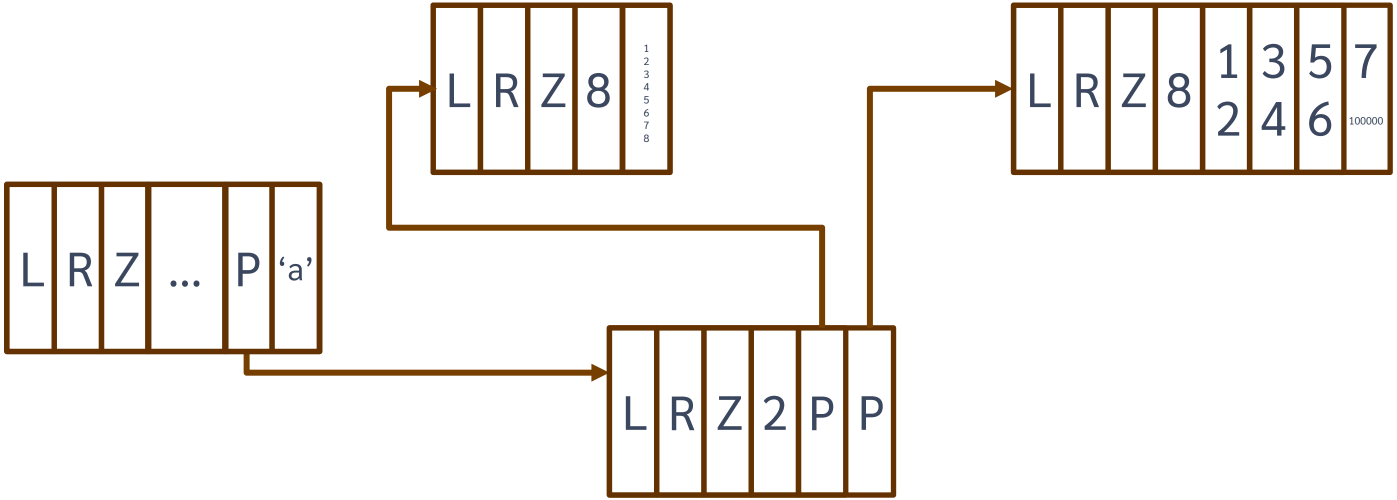
(⍳8)((⍳7),100000)



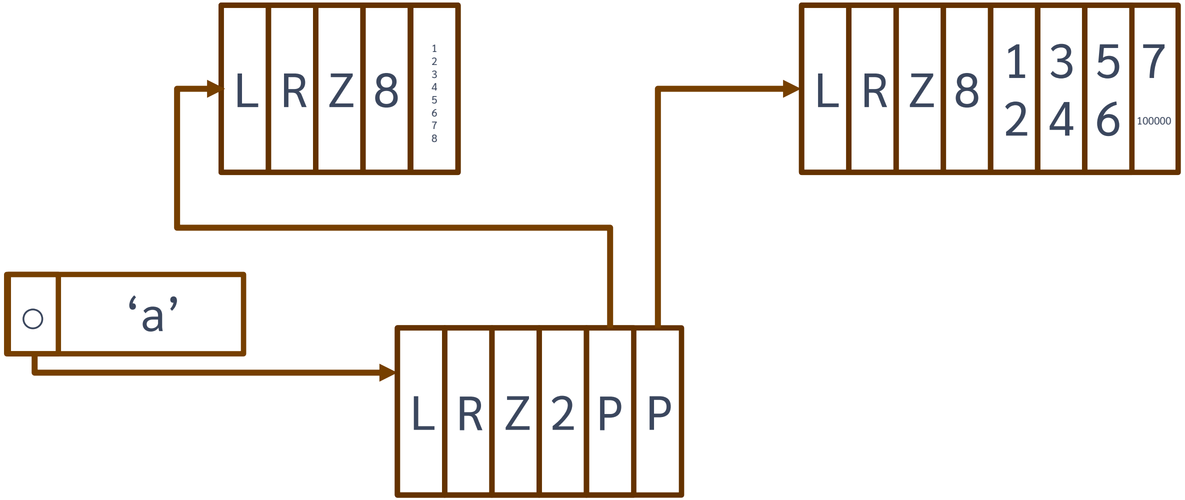
DYALOG


L
R
Z
8
1
2
3
4
5
6
7
8
L
R
Z
8
1
2
3
4
5
6
7
100000
L
R
Z
2
P
P

Symbols



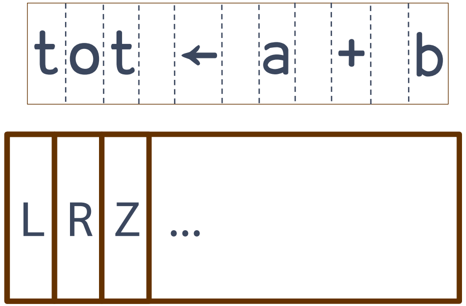
Symbols



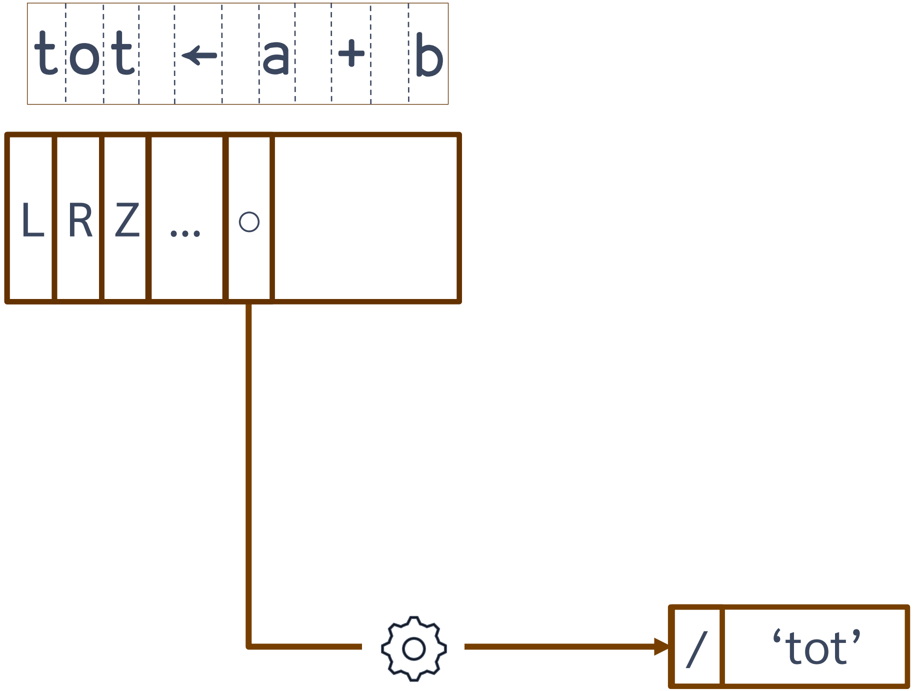
Code

```
tot ← a + b
```

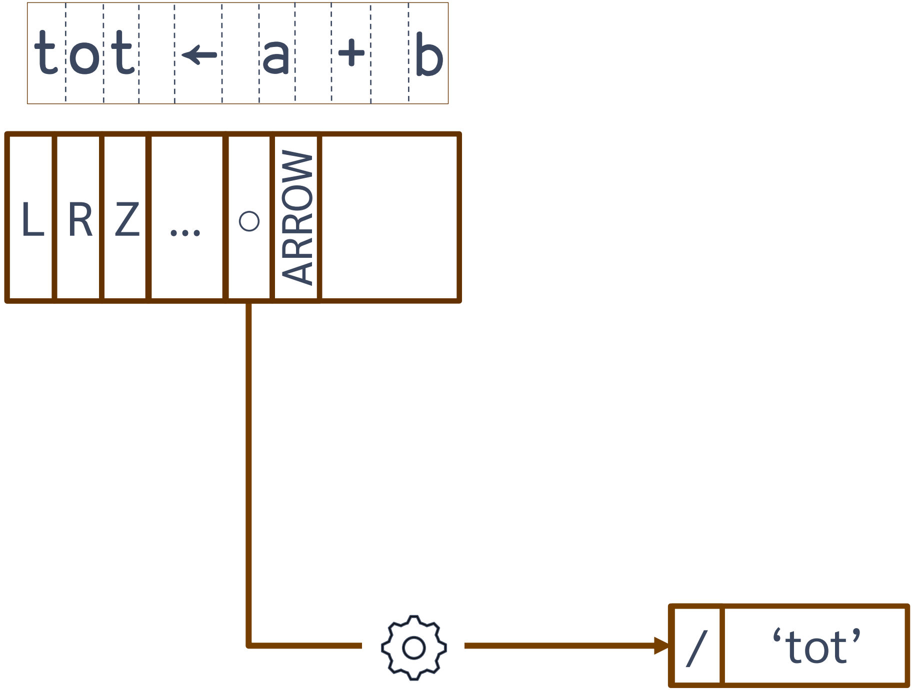
Code



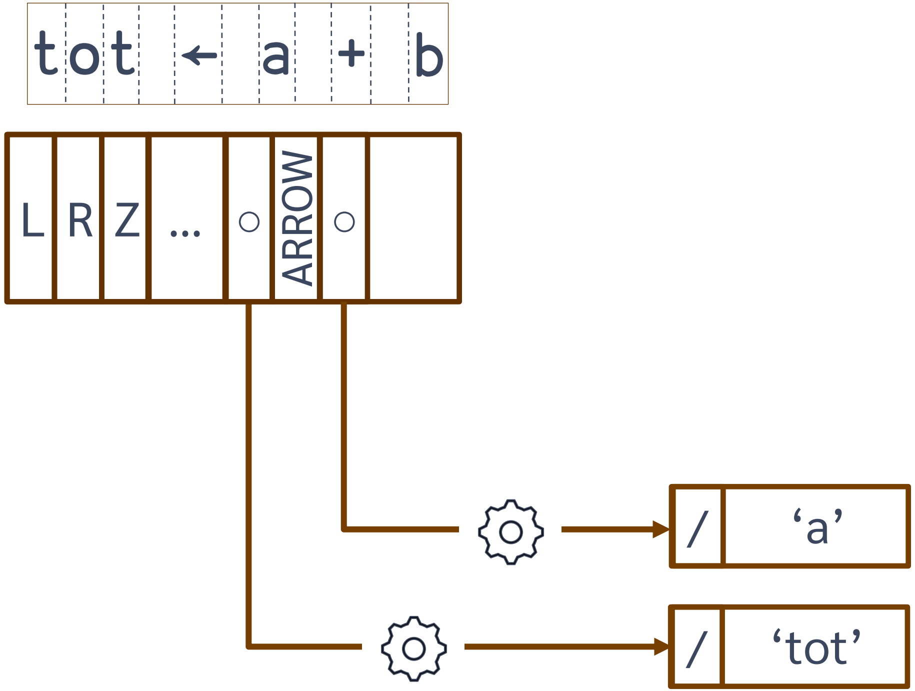
Code



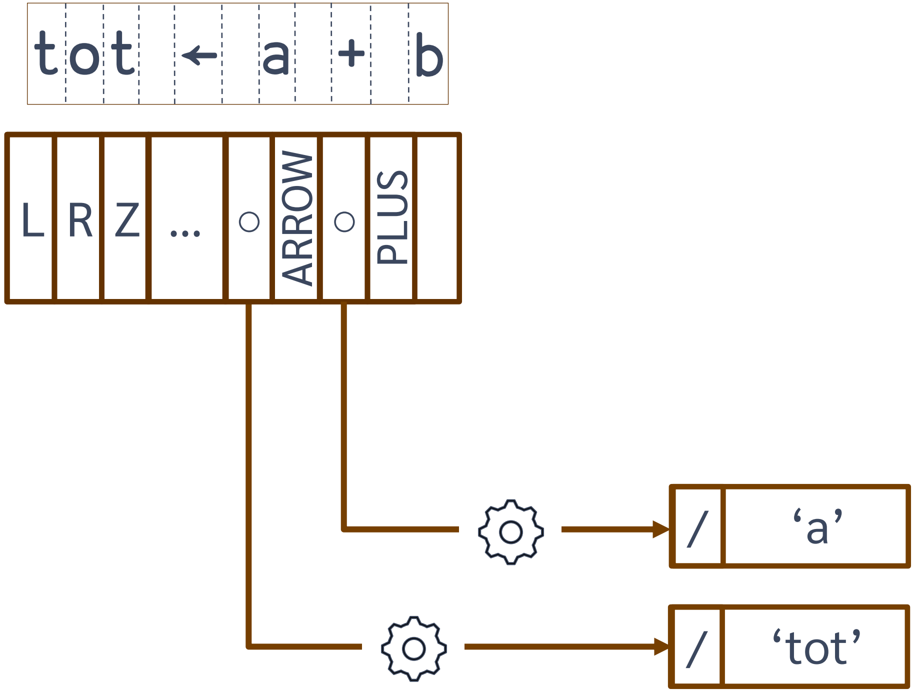
Code



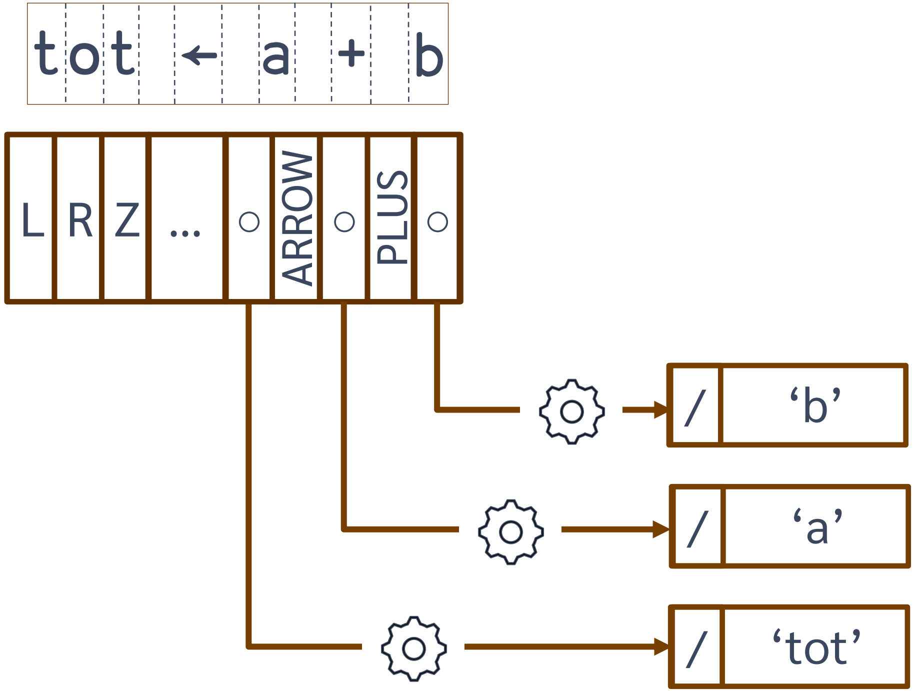
Code



Code



Code



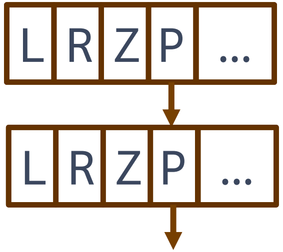
DYALOG

Stack

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

Stack

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

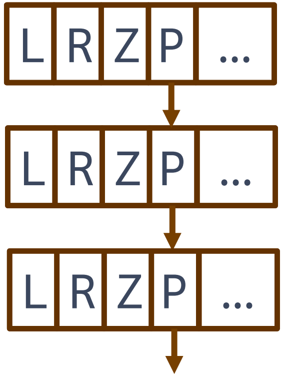


Function “Mode” frame

“Shadow” block

Stack

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

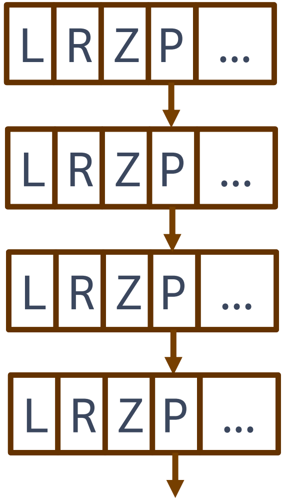


Each

Function “Mode” frame

“Shadow” block

Stack



+/

Each

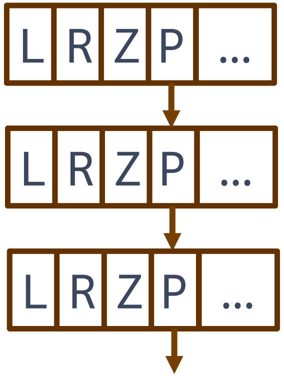
Function “Mode” frame

“Shadow” block

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

Stack

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

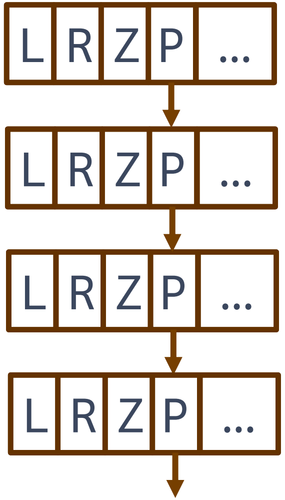


Each

Function “Mode” frame

“Shadow” block

Stack



+/

Each

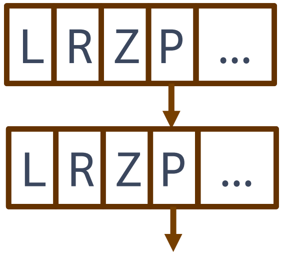
Function “Mode” frame

“Shadow” block

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

Stack

```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```



Function “Mode” frame

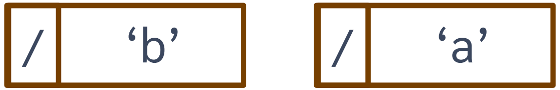
“Shadow” block

Stack

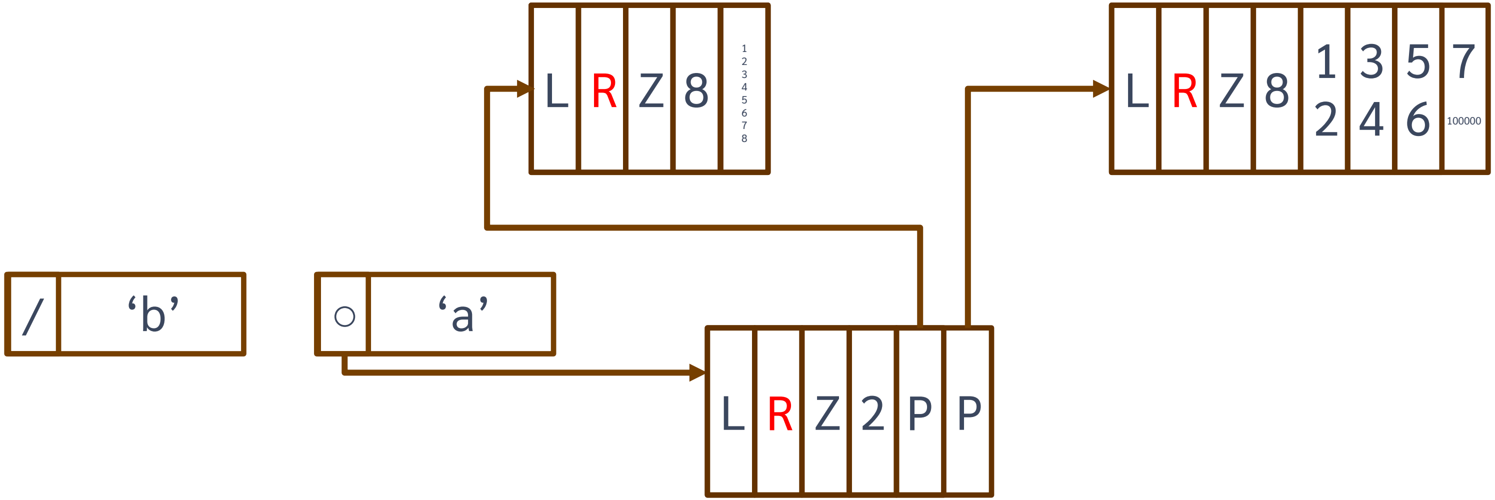
```
r←f;a
a←(1 2)(3 4)
r←+/¨a
```

Reference counts

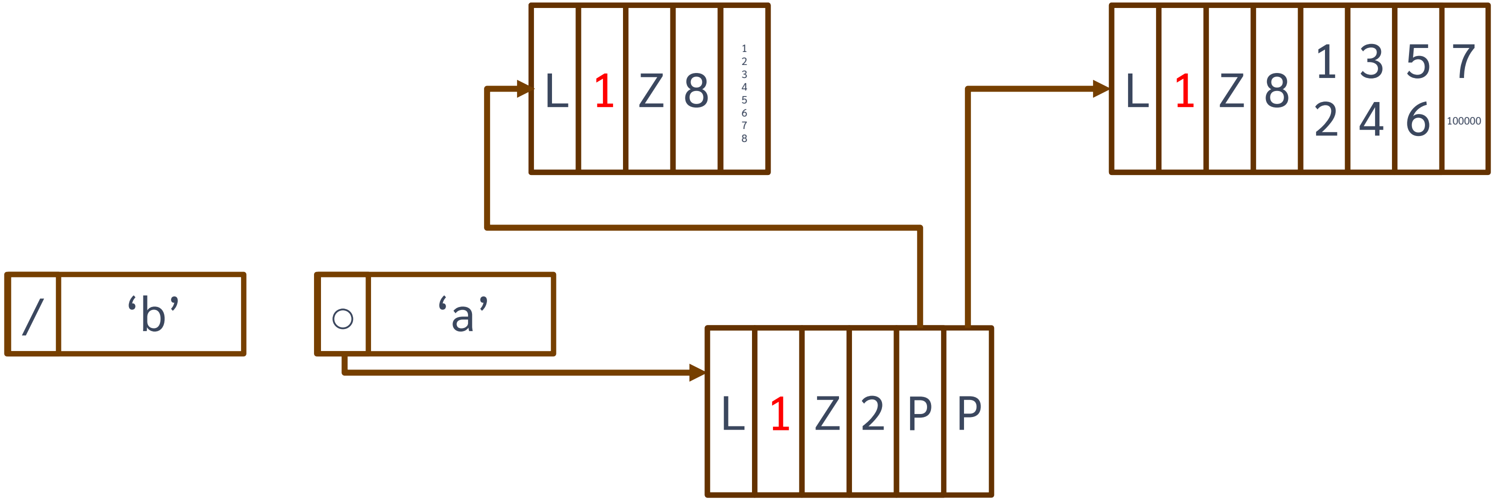
```
a←(⍳8)((⍳7),100000)
```



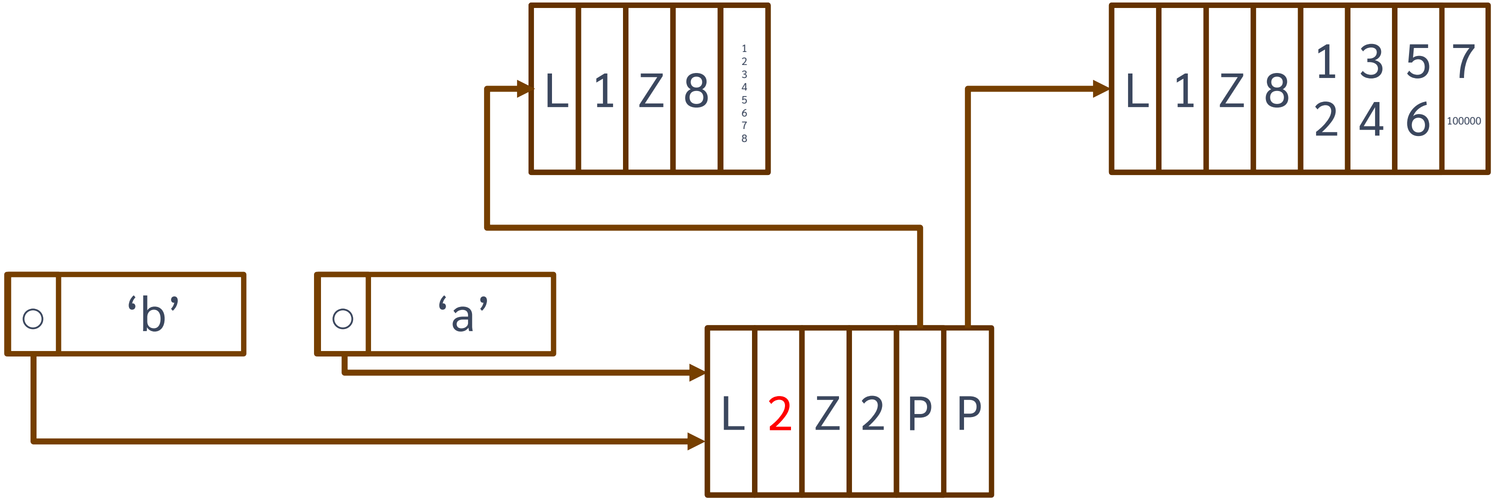
```
a←(⍳8)((⍳7),100000)
```



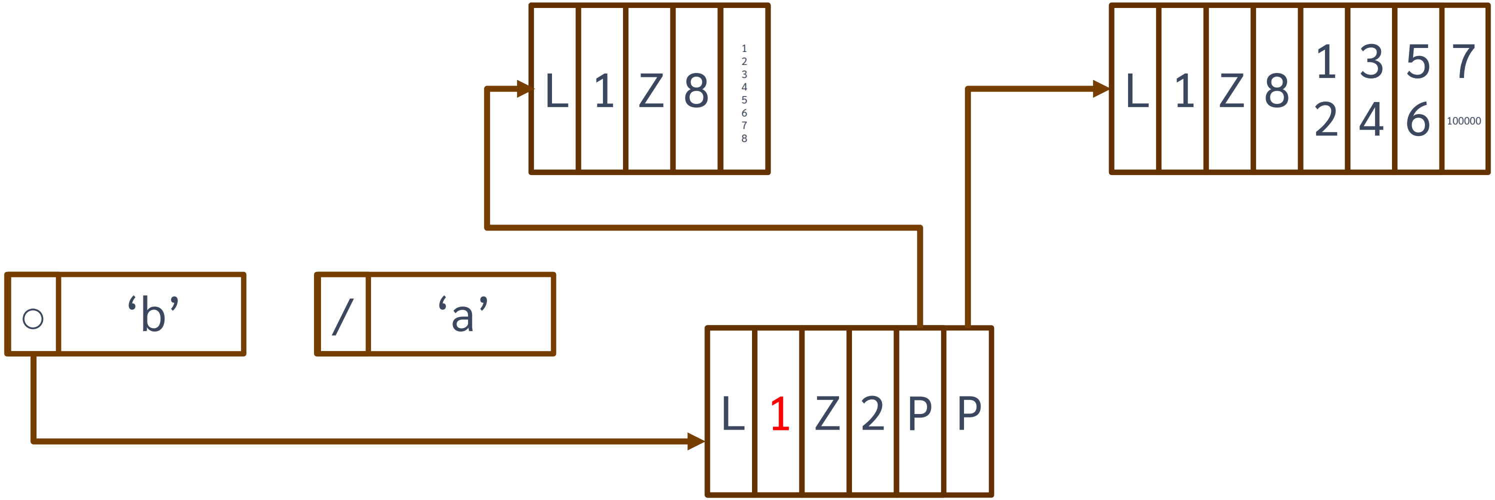
a←(⍳8)((⍳7),100000)



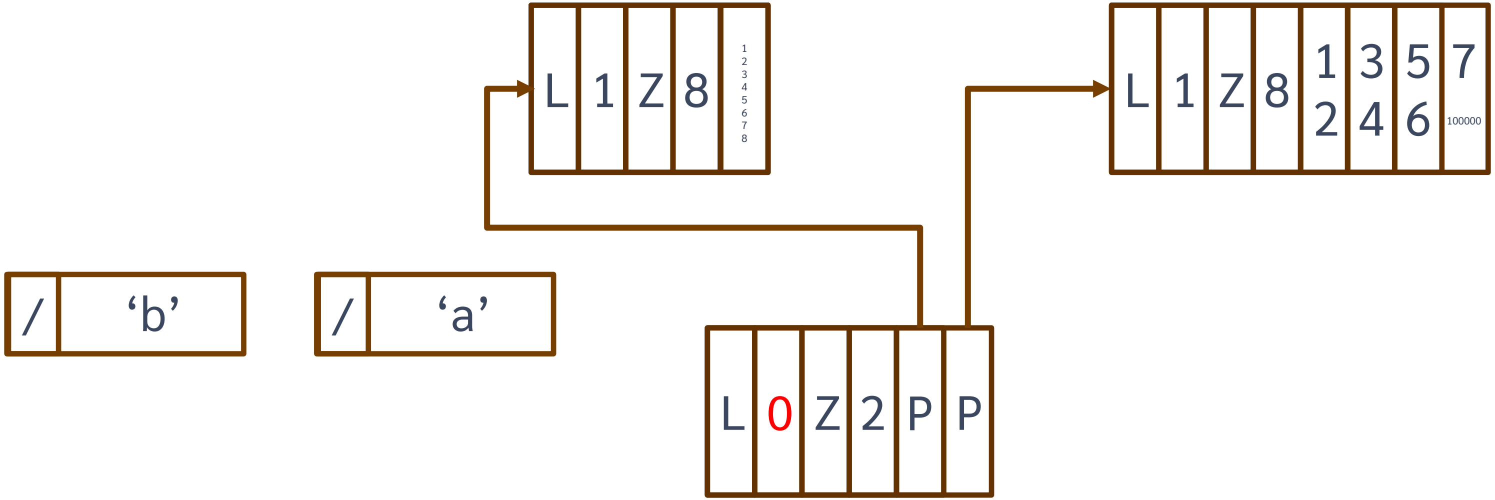
b←a



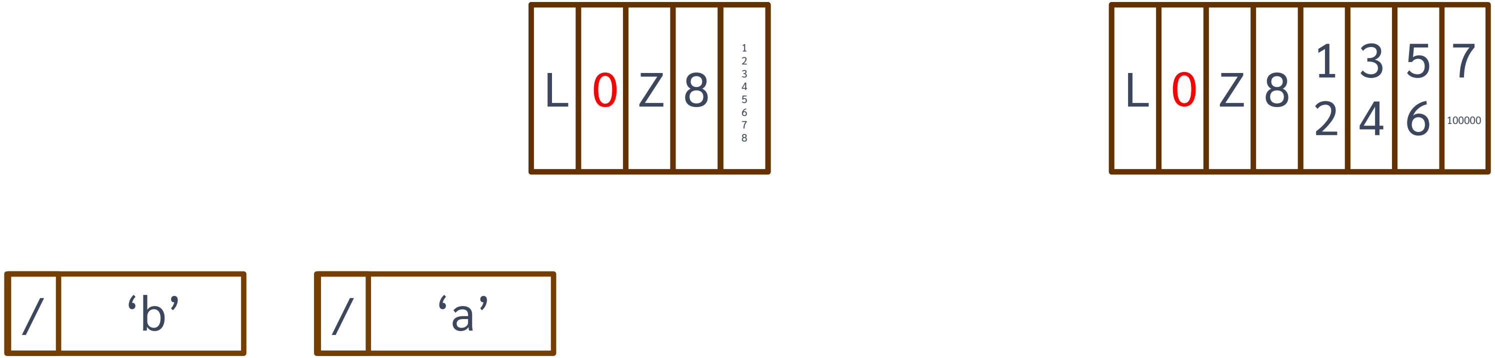
⎕EX'a'



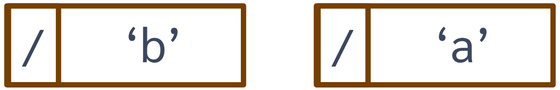
⎕EX'b'



⎕EX'b'



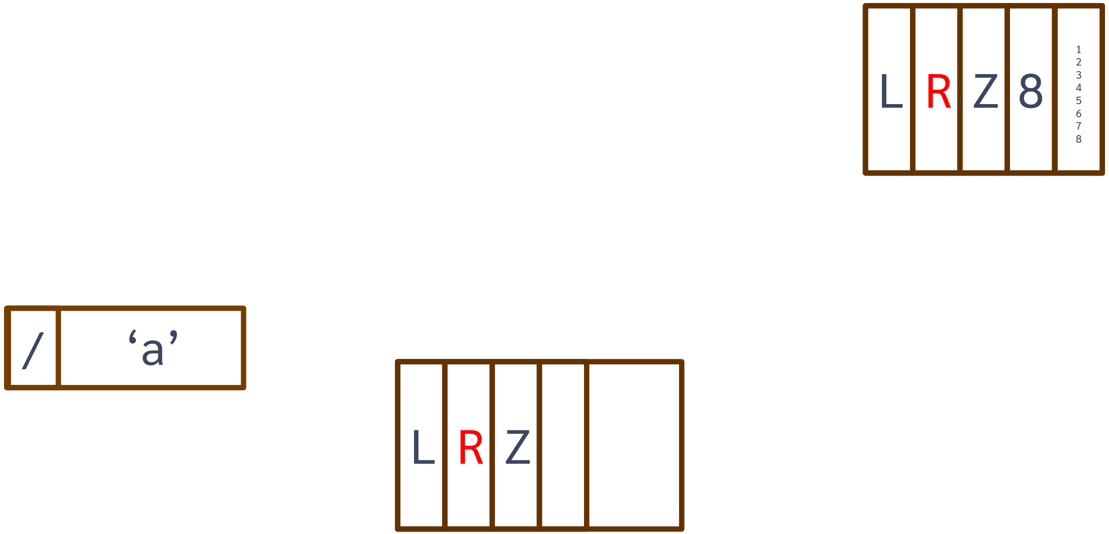
⎕EX'b'



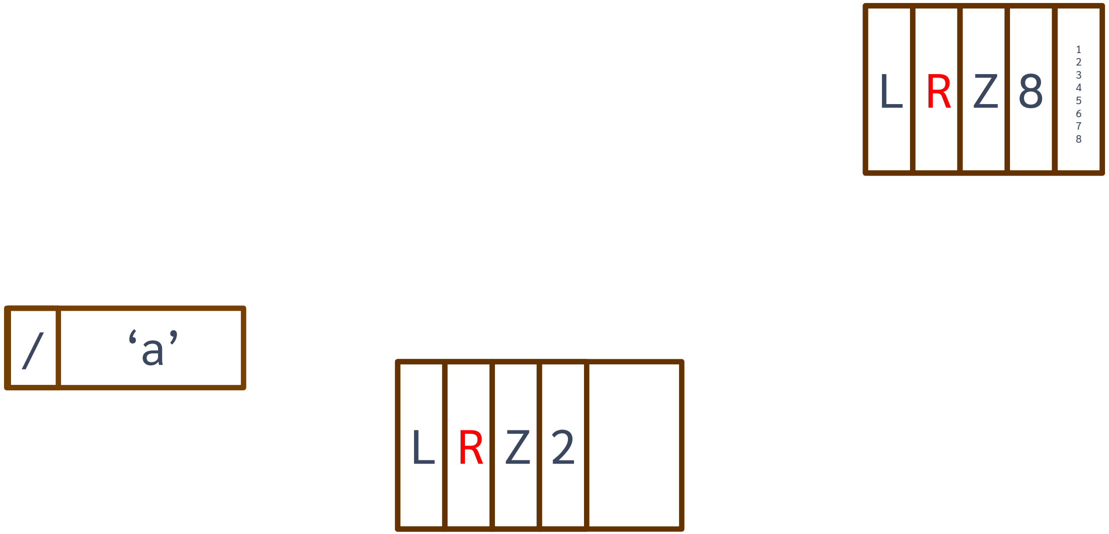
```
a←2/⊂⍳8
```

a←2/⊂⍳8

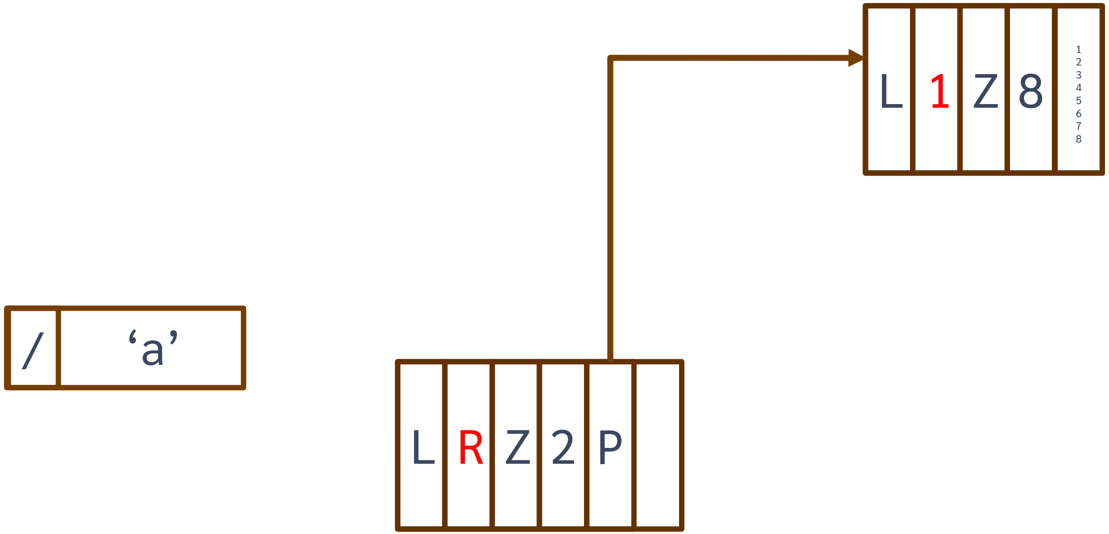
a←2/⊂⍳8



a←2/⊂⍳8

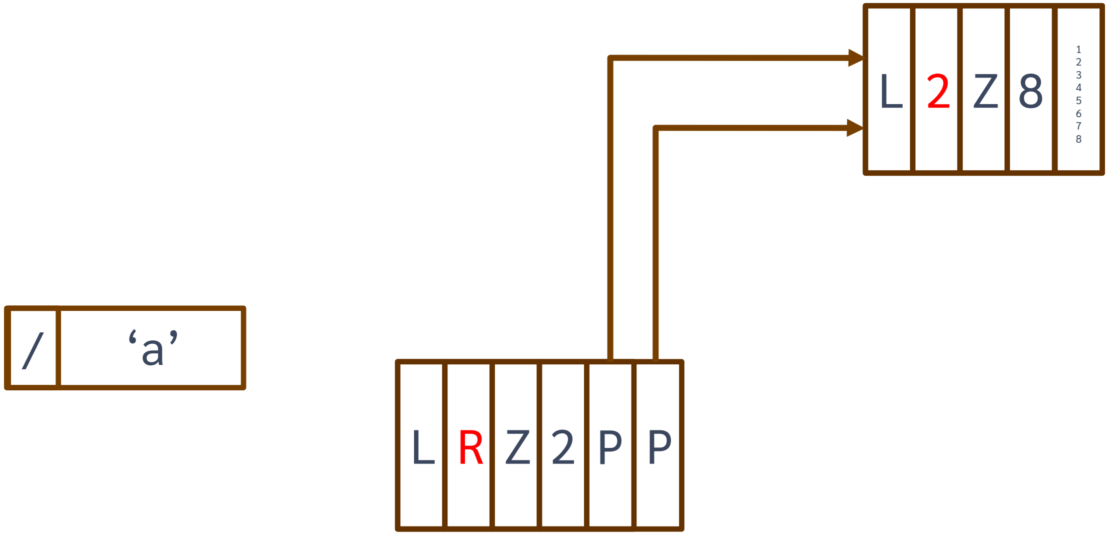


a←2/⊂⍳8

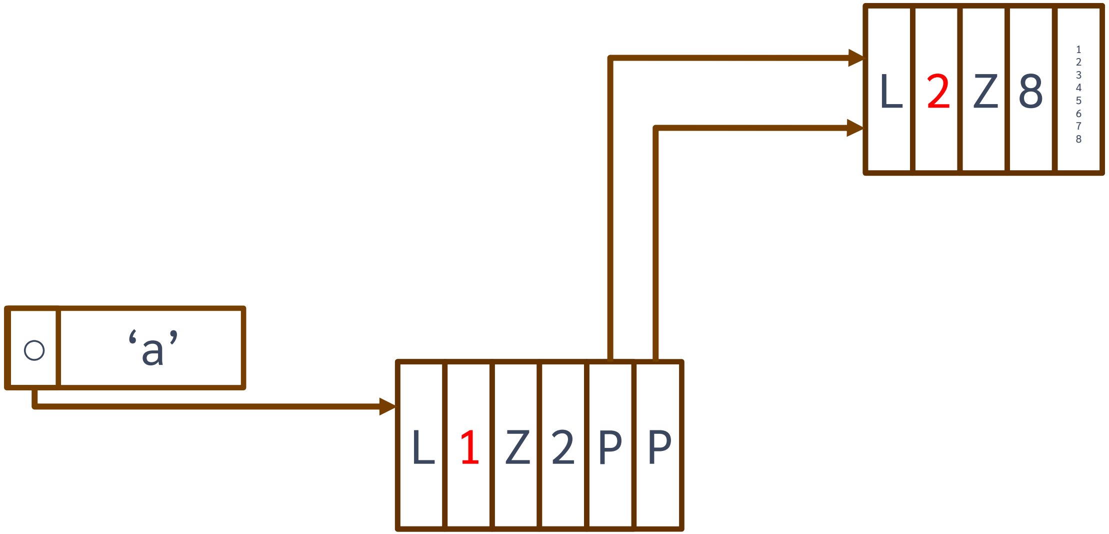


DYALOG

```
a←2/⊂⍳8
```



a←2/⊂⍳8



Refcounts

- Save space.
- Make assignment fast.

APL without them would be impractical.

Refcounts vs optimisations

- Pockets with high refcounts cannot be modified.

```
r←f;a
a←⍳100
r←1+a
```

Refcounts vs optimisations

- Pockets with high refcounts cannot be modified.

```
r←f;a
a←⍳100
r←1+a
```


DYALOG

Refcounts vs optimisations

- Pockets with high refcounts cannot be modified.


```
r←f;a
a←⍳100
r←1+a
```

Refcounts vs optimisations

- Pockets with high refcounts cannot be modified.

```
r←f ;a
a←⍳100
r←1 + a
```

Refcounts vs optimisations

- Pockets with high refcounts cannot be modified.


```
r←f;a
a←⍳100
r←1+a
```

Refcounts vs optimisations

- Pockets with low refcounts can be modified.

```
r←f;a
a←⍳100
r←1+a
```

Refcounts vs optimisations

- Pockets with low refcounts can be modified.

```
r←f

r←1+⍳100
```

Refcounts vs optimisations

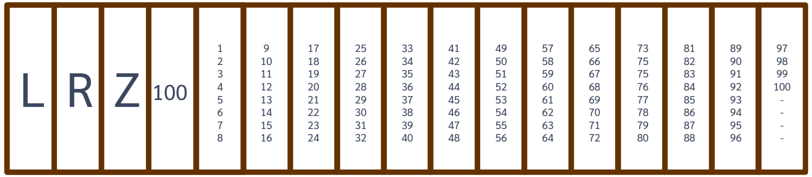
- Pockets with low refcounts <u>can</u> be modified.

```
r←f

r←1+⍳100
```

Refcounts vs optimisations

- Pockets with low refcounts can be modified.

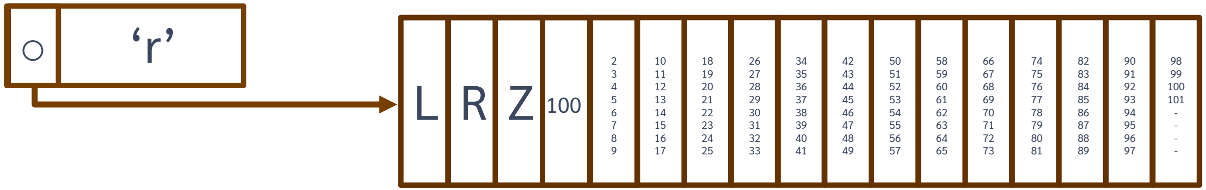
```
r←f

r←1+⍳100
```



Refcounts vs optimisations

- Pockets with low refcounts can be modified.

```
r←f

r←1+⍳100
```



Refcounts vs optimisations

- Pockets with low refcounts can be modified.

```
r←f

r←1+⍳100
```

- 20% faster!

Refcounts vs optimisations

- Pockets with low refcounts <u>can</u> be modified.

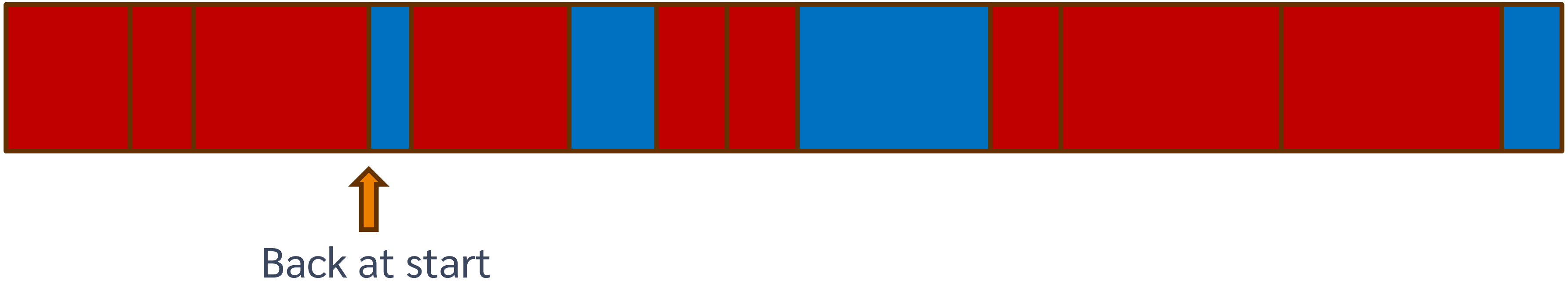
```
r←f

r←1+⍳100
```

- 20% faster!
- Only possible when refcount is low!

Garbage

- Garbage occurs when there are “reference loops”
  - The only thing that references the pockets in the loop is the pockets in the loop
- Traditional APL does not create garbage but OO features can.
- Why, and how it is removed, is a whole other presentation!

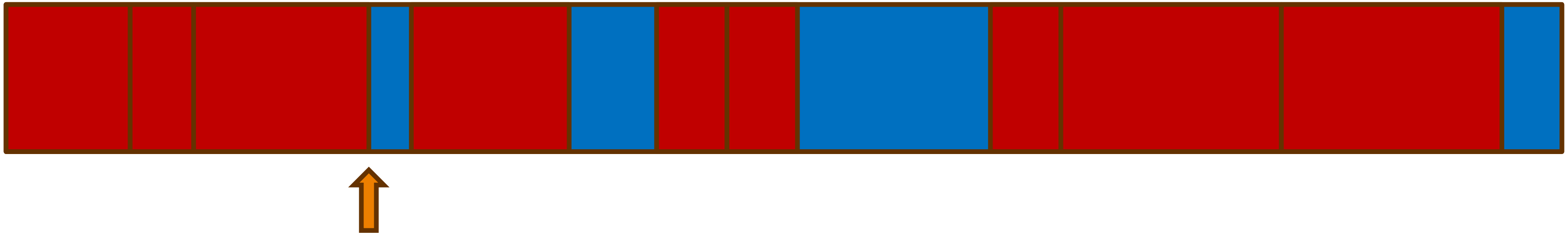
Pocket allocation (and deallocation)



Space could not be allocated.

Not necessarily a WSFULL… we'll see what happens next later.

Pocket compression (“squeeze”)



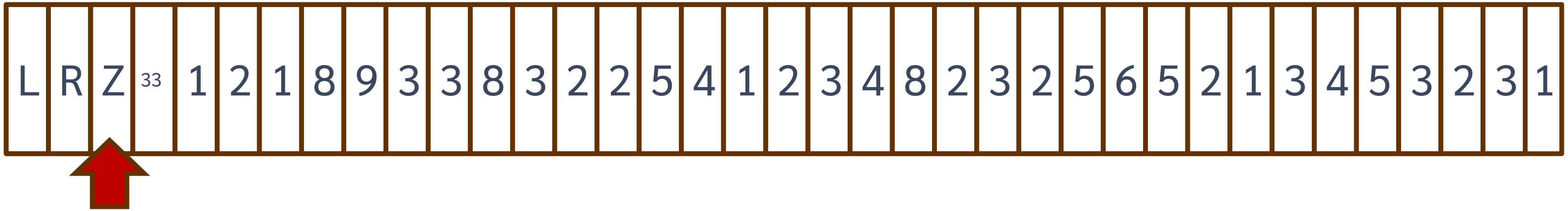
Pocket compression

Pocket compression

Pocket compression

Simple array.
Rank 1.
64-bit doubles.

Pocket compression

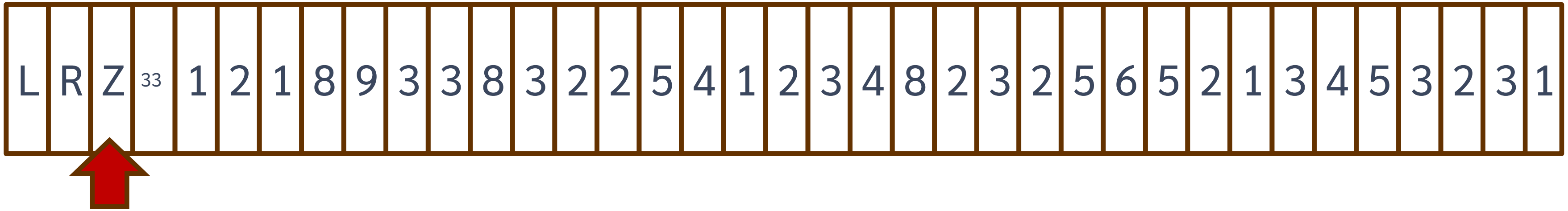


Simple array.

Rank 1.

64-bit doubles.

Pocket compression



Simple array.

Rank 1.

64-bit doubles.

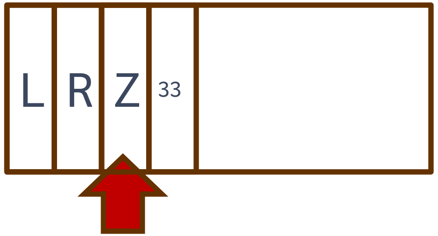
Pocket compression

Simple array.

Rank 1.

64-bit doubles.

Pocket compression

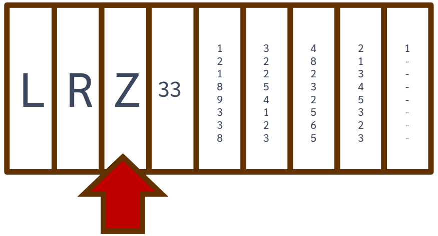


Simple array.

Rank 1.

64-bit doubles.

Pocket compression



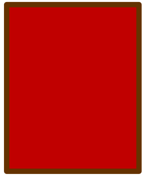
Simple array.

Rank 1.

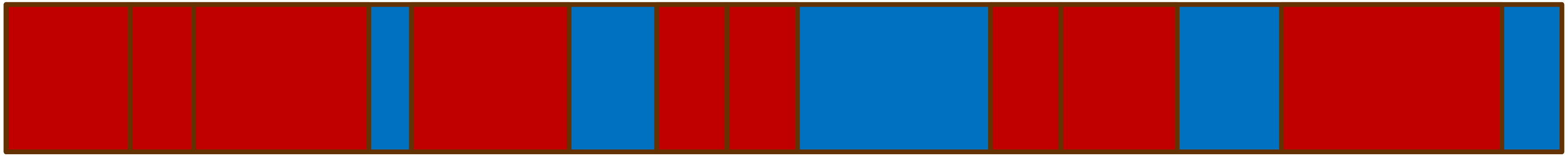
8-bit ints.

Pocket compression

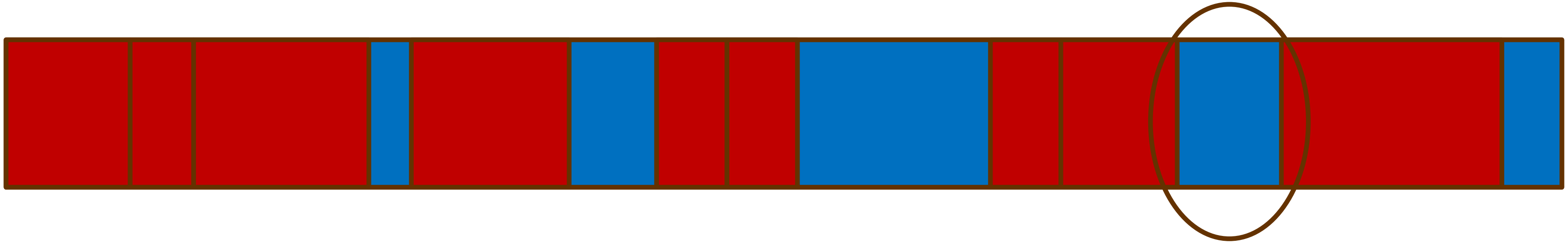
Pocket compression



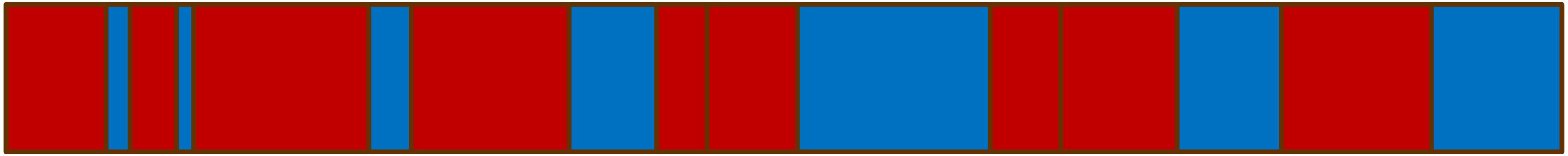
Pocket compression



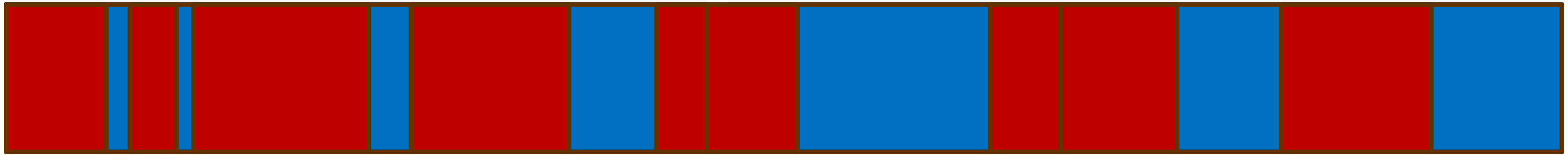
Pocket compression



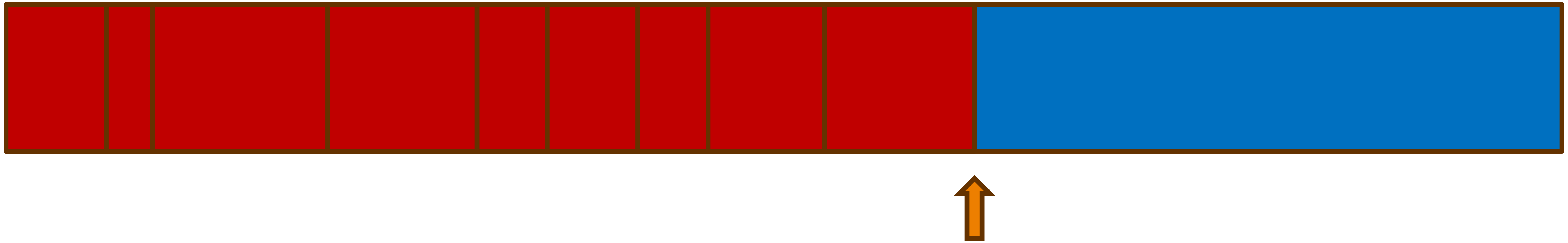
Pocket compression



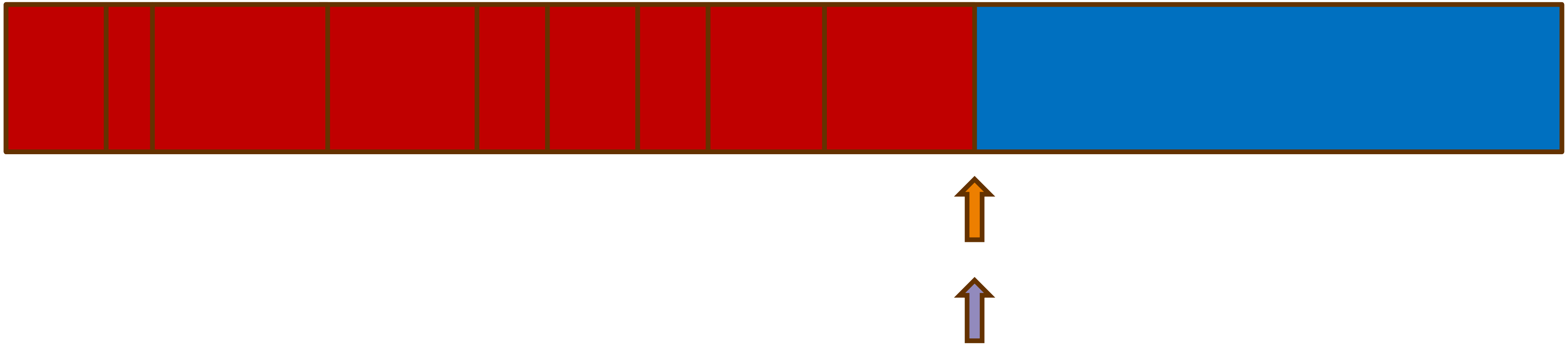
Workspace compaction



Workspace compaction

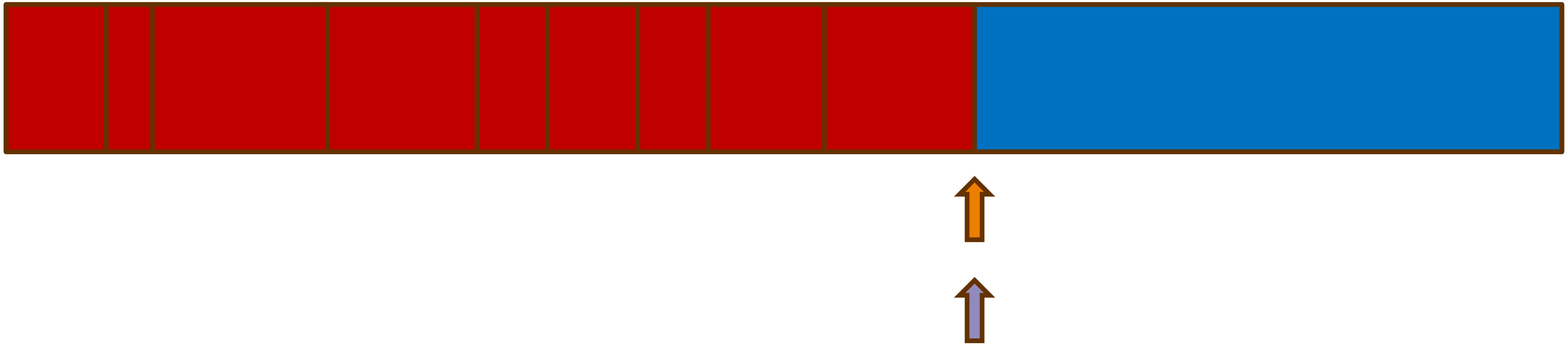


Workspace compaction

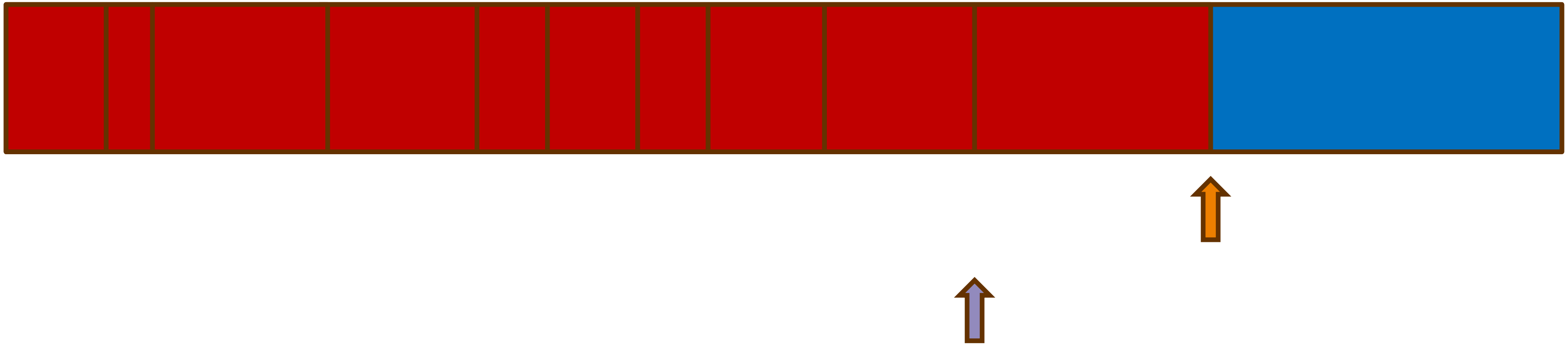


The allocation request

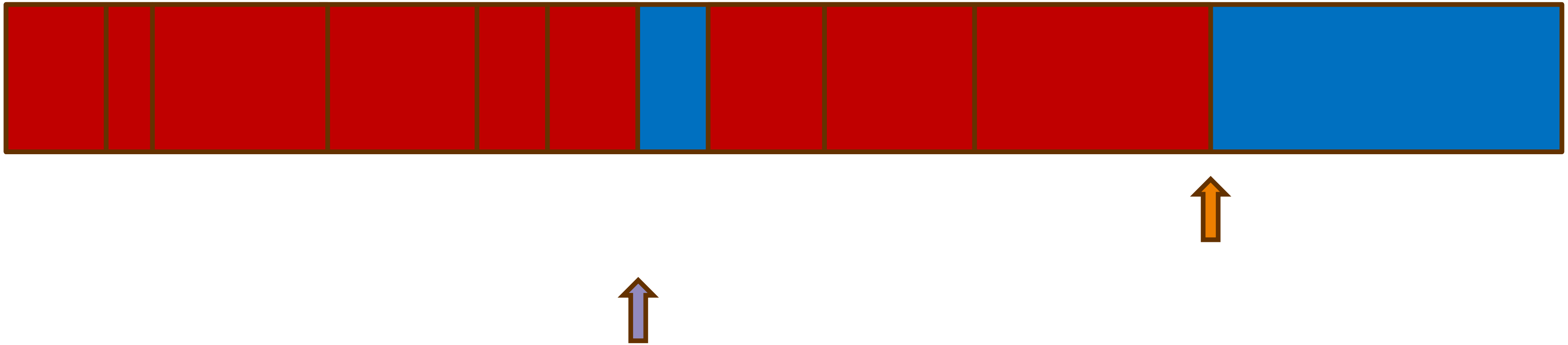
Pocket allocation (and deallocation)



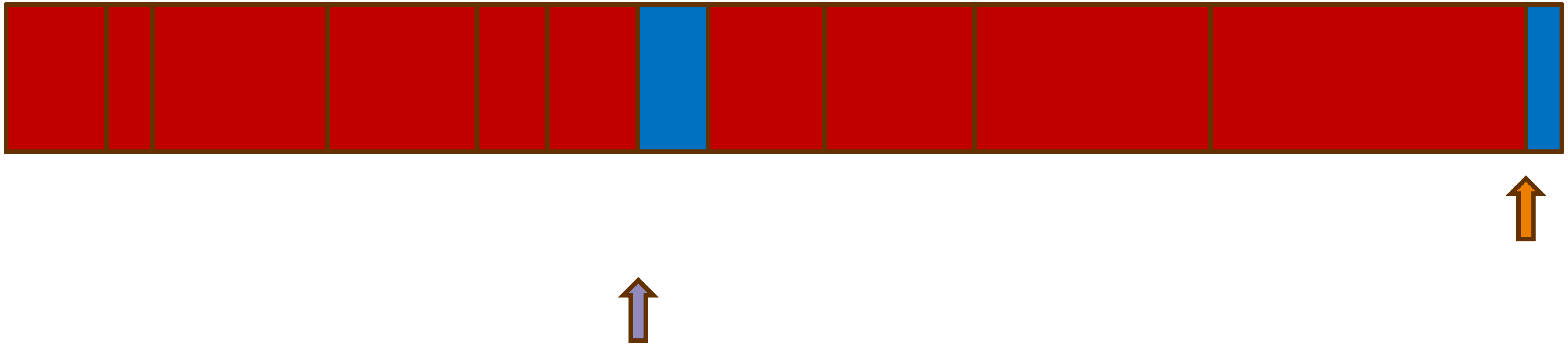
Pocket allocation (and deallocation)



Pocket allocation (and deallocation)

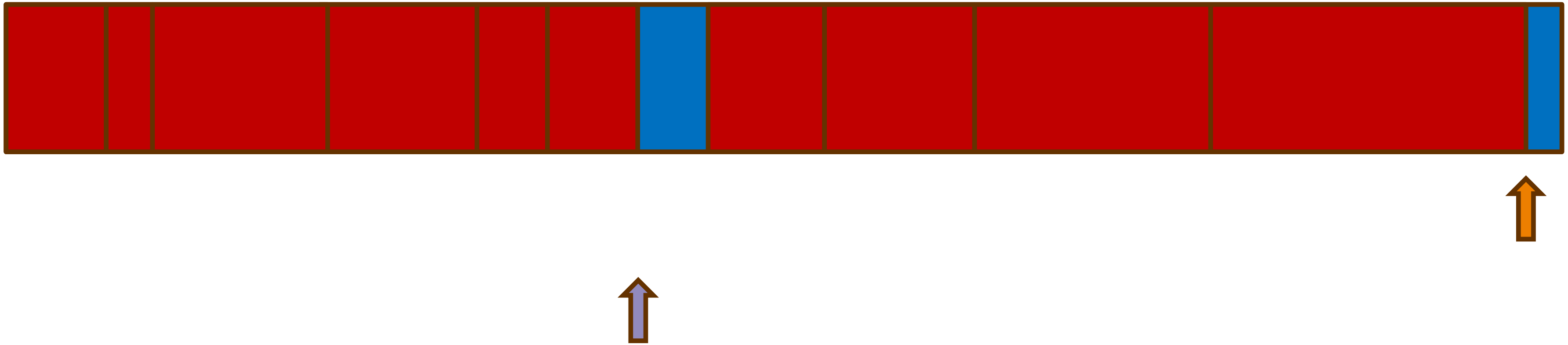


Pocket allocation (and deallocation)

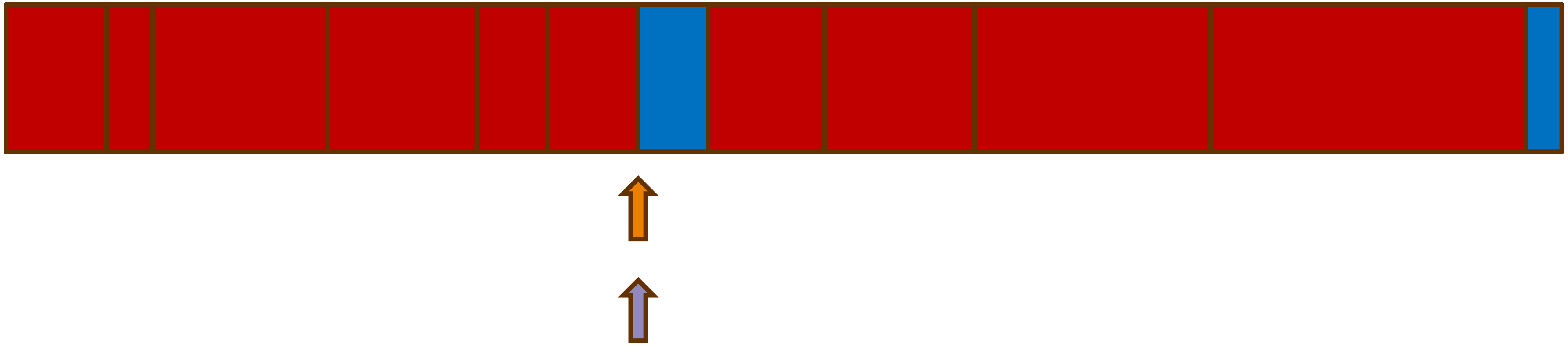


Next allocation request

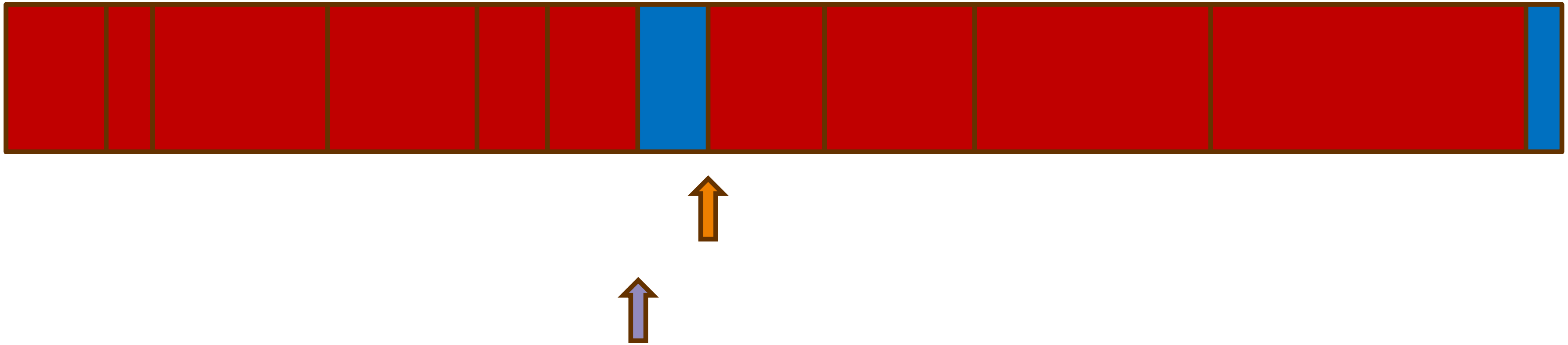
Walk workspace



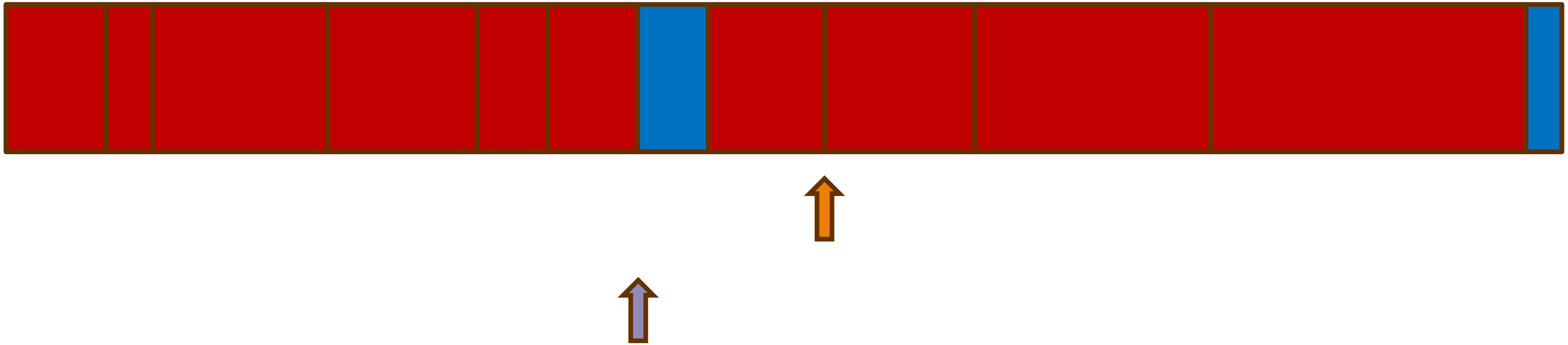
Walk workspace



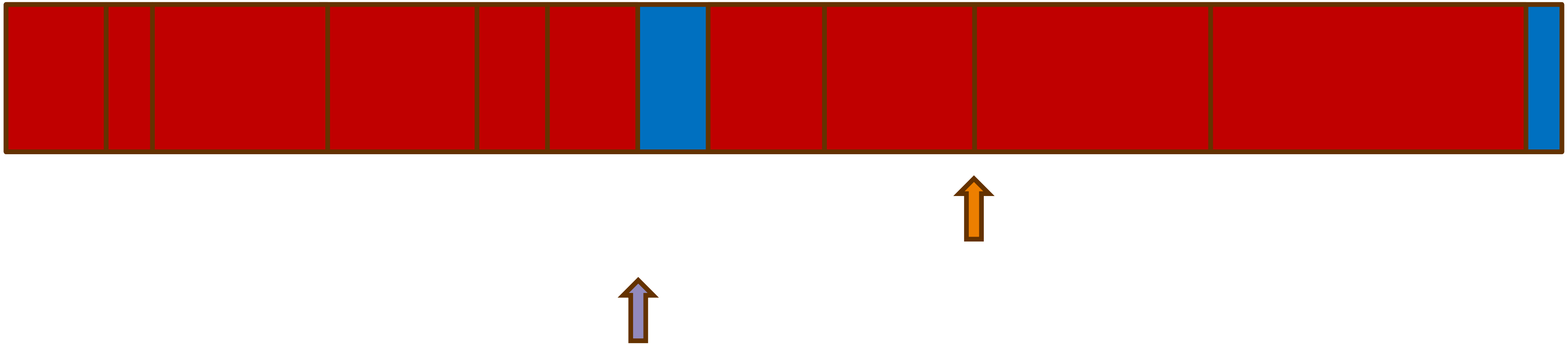
Walk workspace



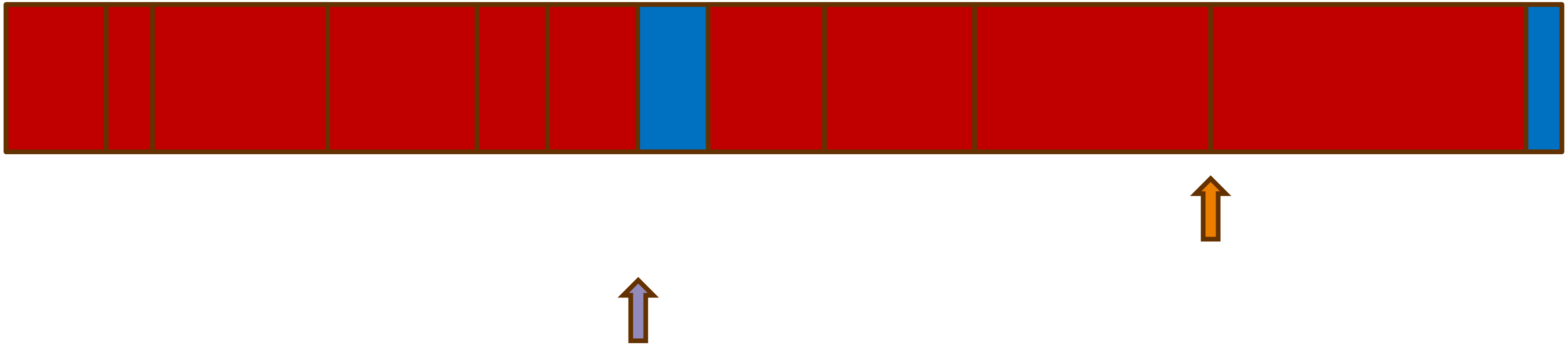
Walk workspace



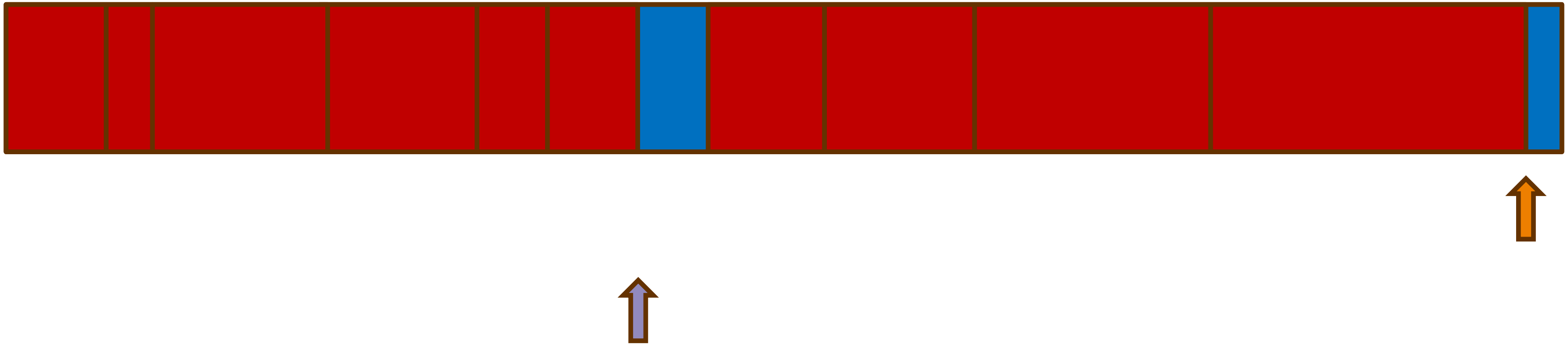
Walk workspace



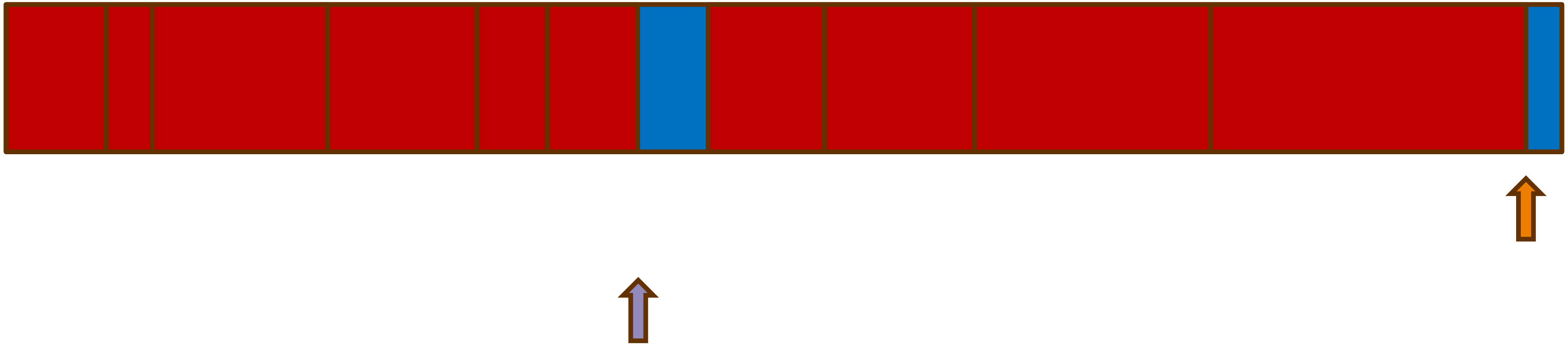
Walk workspace



Walk workspace

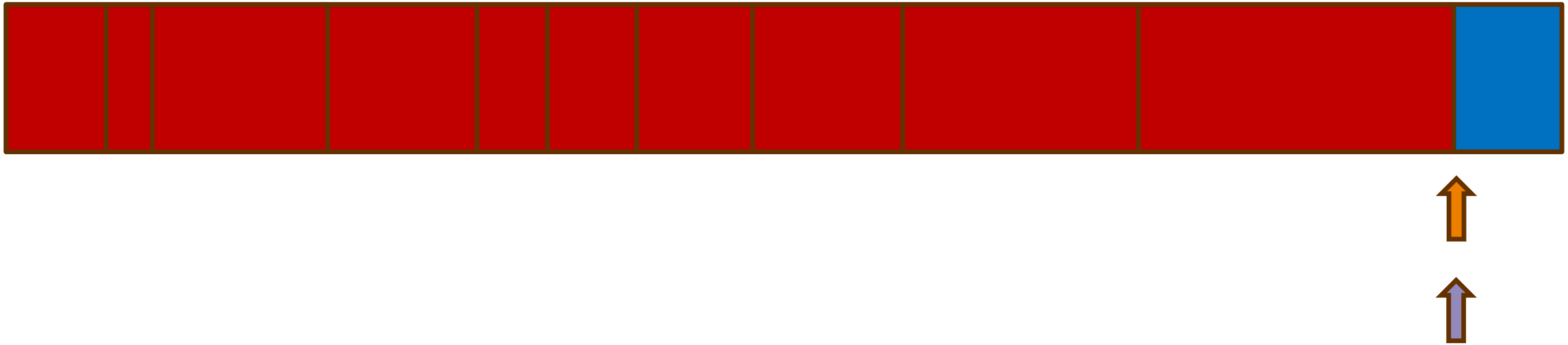


Walk workspace

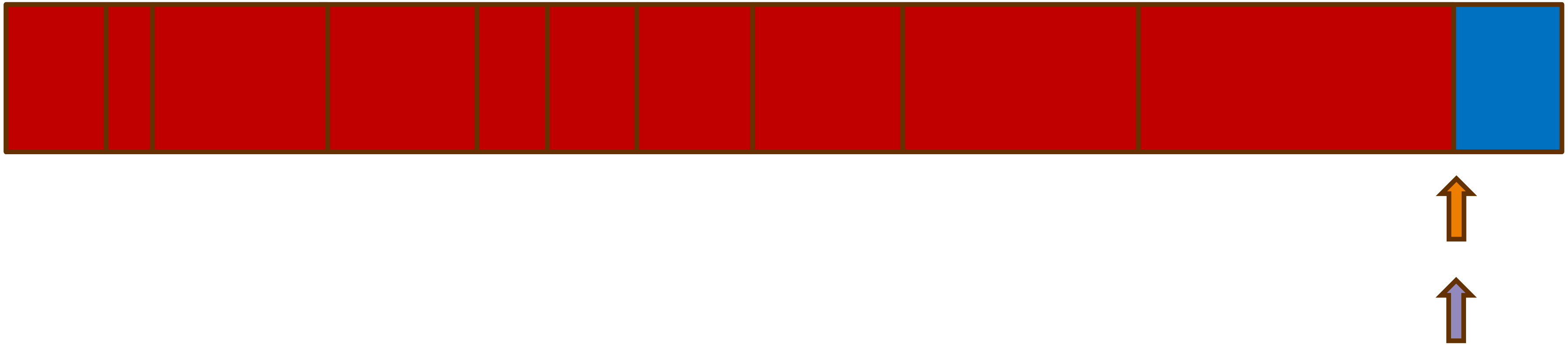


No room.

Compress and compact

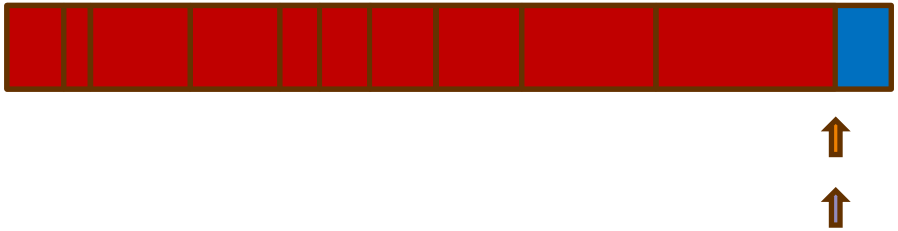


Compress and compact

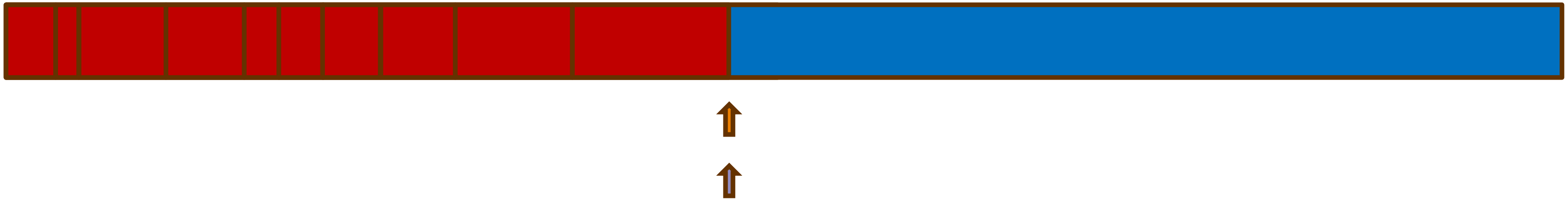


Still no room.

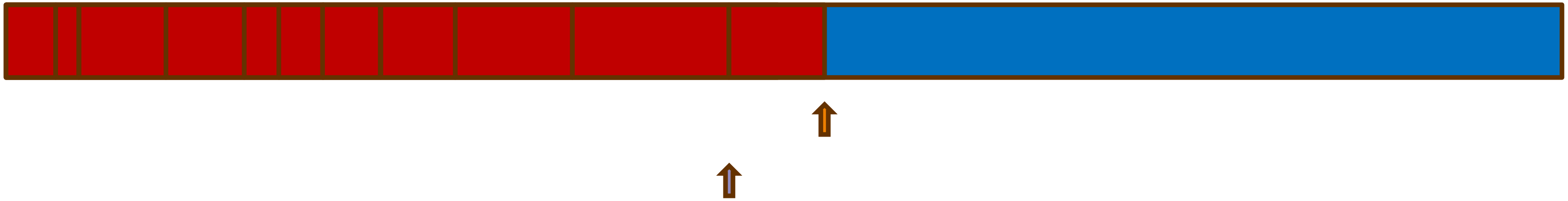
Worspace expansion



Worspace expansion



Worspace expansion



Pocket allocation algorithm

Incredibly simple.

Very fast.

Every new interpreter developer thinks they can improve it.

No-one has so far.

In 18.0 we almost did…

Reducing workspace allocation

- ⎕WA
  - Performs compression and compaction.
  - Resets to an “ideal” memory allocation.

Useful tools

```
2000⌶
```

- Number of free and allocated pockets.
- Number of compactions.
- Sediment size.
- Current allocation and allocation HWM.
- Set min/max allocation sizes.
- ⎕WA without compaction etc.

Useful tools

`2002⌶`

- `⎕WA` which allows the WS allocation to be specified.

Why 2000⌶ ?

QUID MM⌶ ?

QUID MM⌶ ?

MM⌶

QUID MM⌶ ?

M M

QUID MM⌶ ?

MEMORY MANAGER

The workspace

- Everything in it is a pocket.
- Pockets are refcounted.
- Pockets are allocated using a “rotating first fit” algorithm.
- Workspace is compressed and compacted only when space cannot be allocated.
- The workspace allocation increases only when compression and compaction don’t help.
- You can monitor when this happens and have some control over it.

Questions?