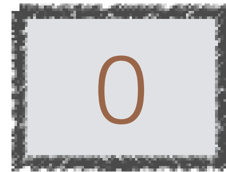
An Optimal Implementation of Fetch-and-Increment

Faith Ellen, University of Toronto
Philipp Woelfel, University of Calgary

Fetch&Inc Object

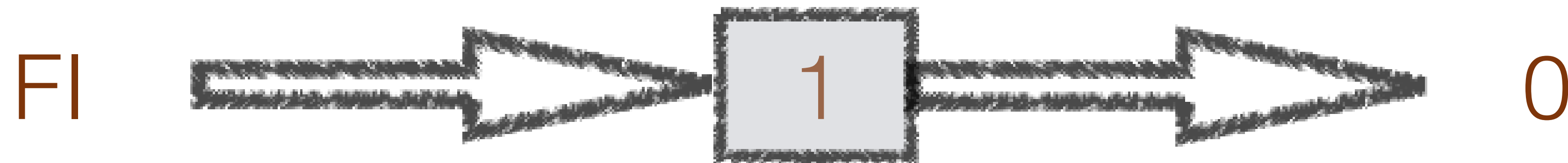


stores a non-negative integer

FI: returns value of object

increments value of object

Fetch&Inc Object



stores a non-negative integer

FI: returns value of object
increments value of object

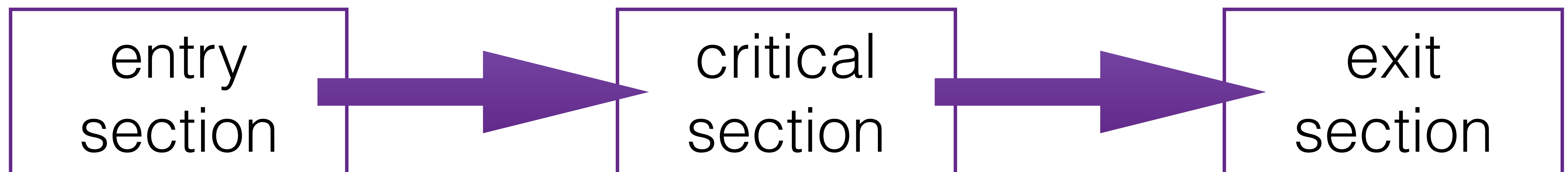
Applications

Renaming: each process wants to get a new name from a small range

GETNAME: return FI()

Applications

Mutual Exclusion: processes want exclusive use of a shared resource




```
i := FI()
if i > 0 then
   wait until A[i]
```

```
A[i+1] := true
```

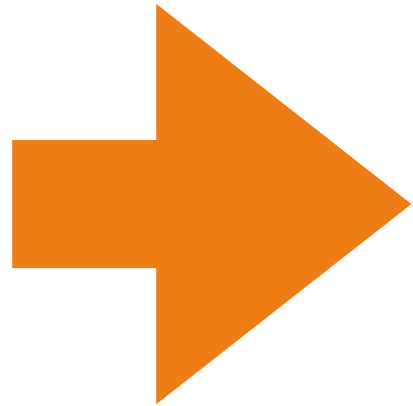
Model

asynchronous shared memory

n deterministic processes: 1,....,n

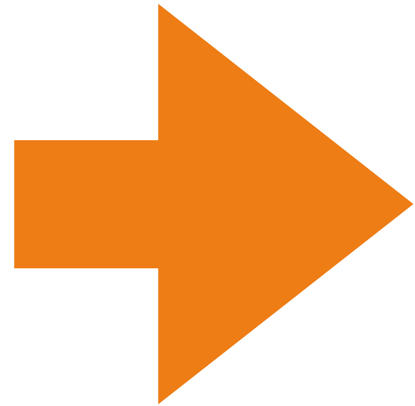
linearizable, wait-free implementations

Fetch&Inc Objects have consensus number 2



cannot be implemented from registers

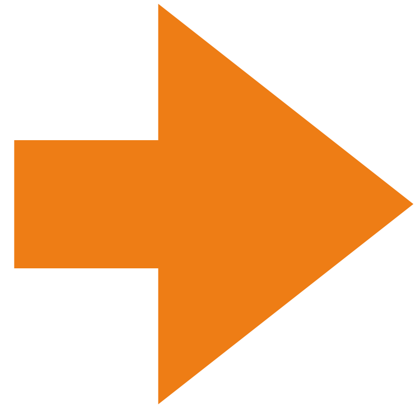
Fetch&Inc Objects have consensus number 2



cannot be implemented from registers

can be implemented from Test&Set objects or Swap objects and registers [AWW]

Fetch&Inc Objects have consensus number 2



cannot be implemented from registers

can be implemented from Test&Set objects or Swap objects and registers [AWW]

but not efficiently

Theorem [JTT]
Any implementation of a Fetch&Inc object from historyless objects and resettable consensus has worst case step complexity $\Omega(n)$

Model

asynchronous shared memory

n deterministic processes: 1,....,n

linearizable, wait-free implementations

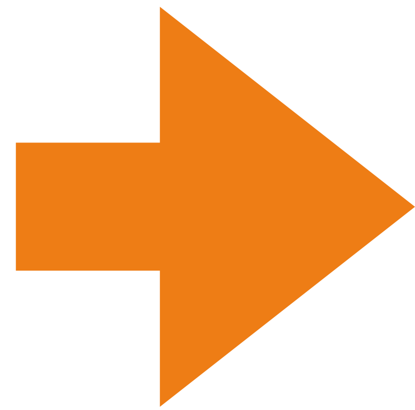
LL/SC objects and registers

LL/SC OBJECT

LL(v): returns the value of object v

SC(v,x): if, since the process last performed LL(v), no process has performed a successful SC on v, it sets the value of v to x and returns T; otherwise it just returns F

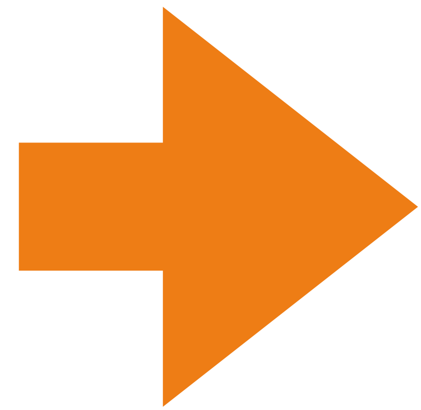
LL/SC Objects have infinite consensus number



implementations of Fetch&Inc objects from LL/SC objects and registers exist

Theorem [Jayanti]
Any implementation of a Fetch&Inc object from LL/SC objects and registers has worst case step complexity $\Omega(\log n)$

LL/SC Objects have infinite consensus number

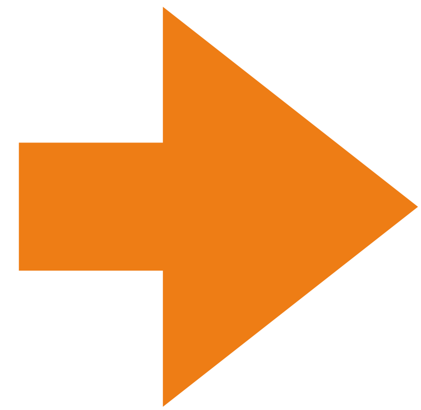


implementations of Fetch&Inc objects from LL/SC objects and registers exist

universal construction [ADT,J]

O(log n) step complexity

LL/SC Objects have infinite consensus number



implementations of Fetch&Inc objects from LL/SC objects and registers exist

universal construction [ADT,J]

$O(\log n)$ step complexity

uses $\Omega(n \log n)$-bit objects

DISC 2012

Ellen, Ramachandran, Woelfel

implementation of a Fetch&Inc object with $O(\log^2 n)$ step complexity using $O(\log m)$-bit LL/SC objects and registers,
where m = number of FIs performed.

Our contribution

an implementation of a Fetch&Inc object
with $O(\log n)$ step complexity
using $O(\log m)$-bit LL/SC objects and
registers,
where m = number of FIs performed

A non-blocking implementation using an O(log m)-bit LL/SC object

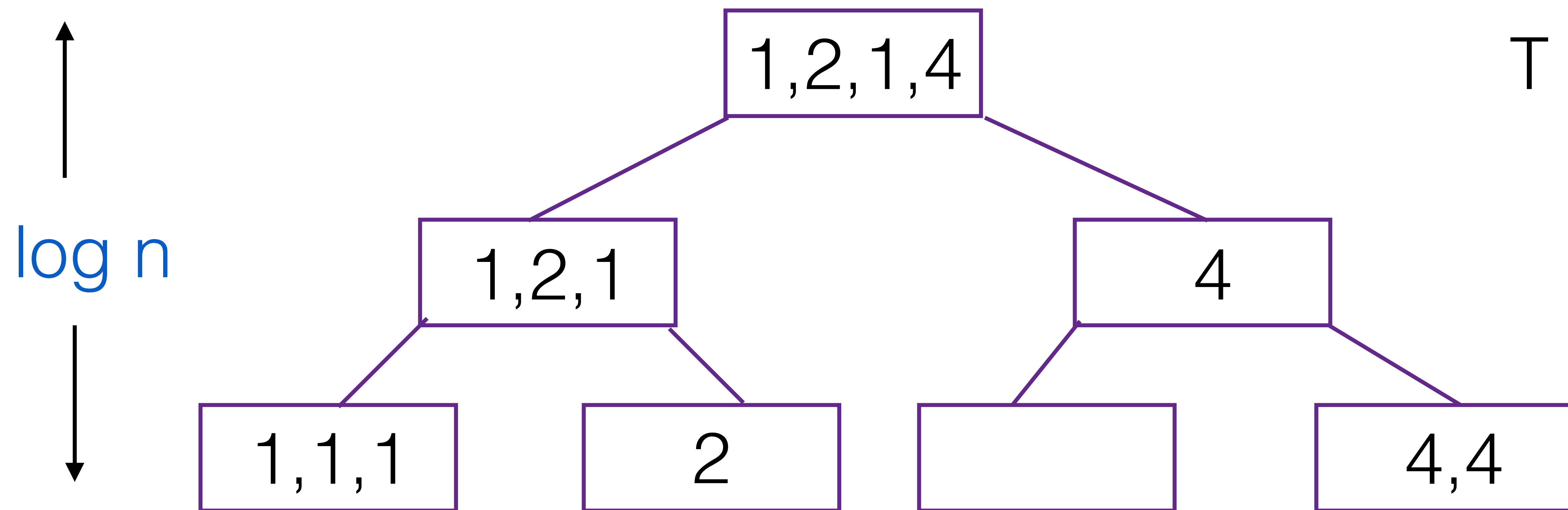
store the value of the Fetch&Inc in an LL/SC object v

```
repeat
    x ← LL(v)
until SC(v,x,x+1)
return x
```

A wait-free implementation with O(log n) step complexity using O(m log n)-bit LL/SC objects

represent the value of the Fetch&Inc object by its history: the sequence of ids of the processes that perform successful instances of FI

1,2,1,4,4,1



- the list at a vertex is a prefix of that vertex at all later times
- the list at an internal vertex is a prefix of the lists at its children

At each internal vertex v, a process does the following twice:

```
x := LL(v)
sl := READ(left(v))
sr := READ(right(v))
compute an interleaving x' of sl and sr with prefix x
SC(v,x')
```

At each internal node v, process i does the following twice:

```
x := LL(v)
sl := READ(left(v))
sr := READ(right(v))
compute an interleaving x' of sl and sr with prefix x
SC(v,x')
```

before: k copies of i in a child of v

after: k copies of i in v

At each internal node v, process i does the following twice:

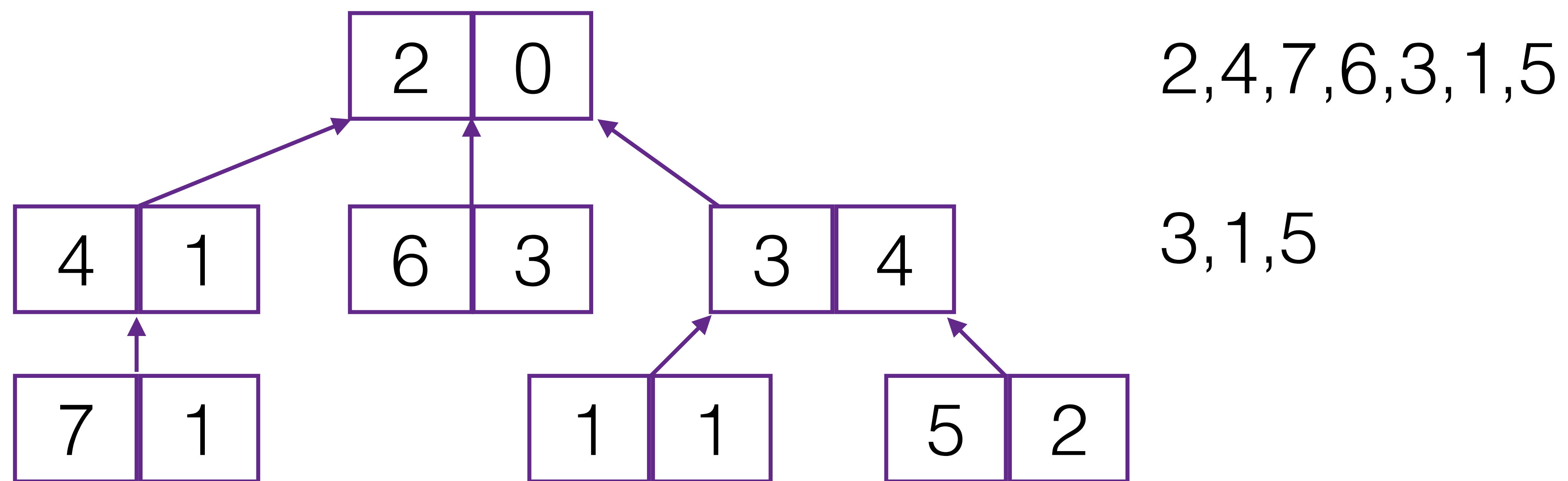
x := LL(v)
sl := READ(left(v))
sr := READ(right(v))
compute an interleaving x' of sl and sr with prefix x
SC(v,x')

before: k copies of i in a child of v
after: k copies of i in v

LL(v) LL(left(v)) LL(right(v)) SC(v) LL(v) LL(left(v)) LL(right(v)) SC(v)
[SC(v)] [SC(v)]

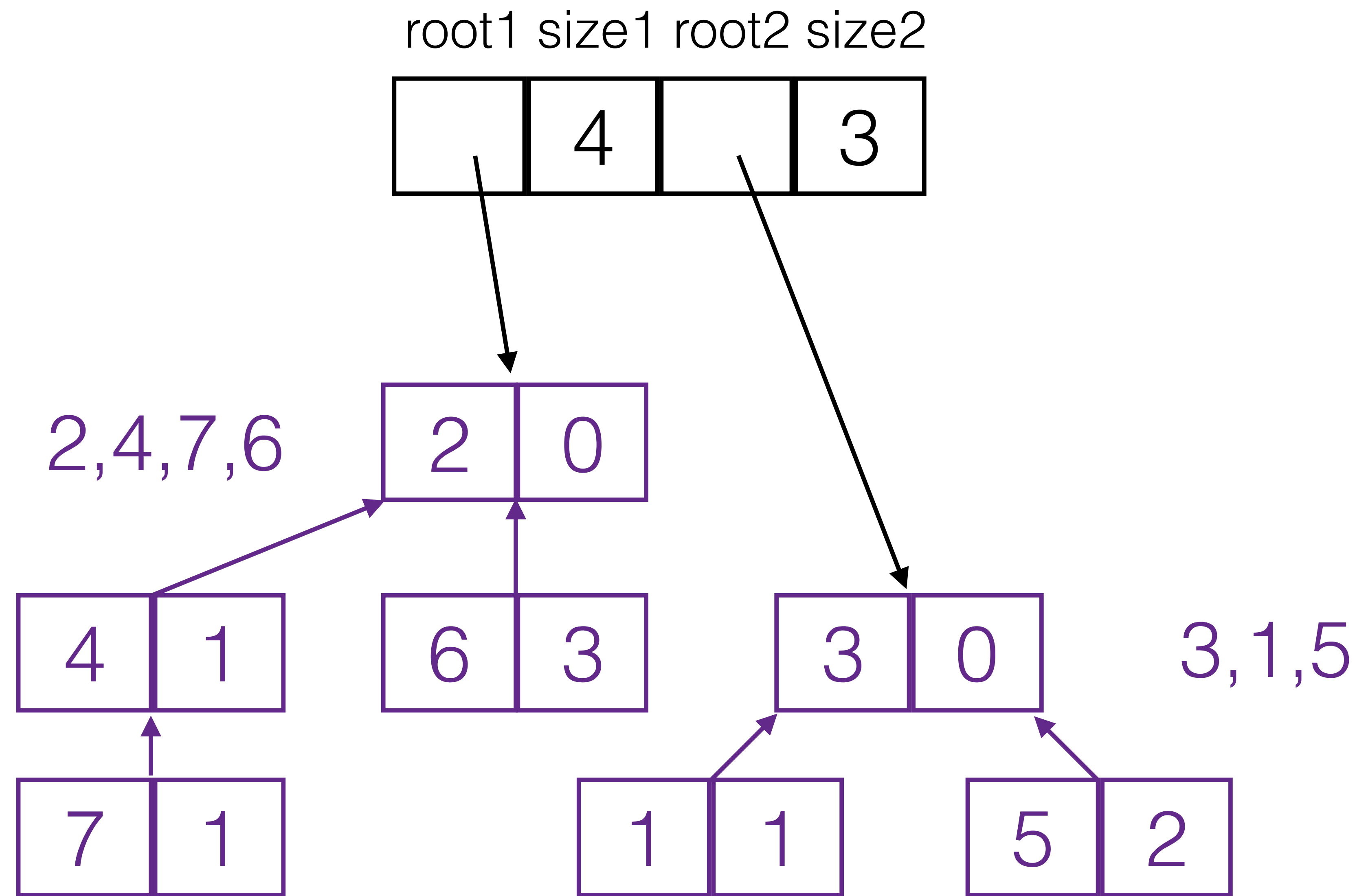
To reduce the size of the shared base objects:

Represent a list at a vertex using an ordered in-tree of nodes, each with a process ID, a parent pointer, and an offset



offset of a node is the number of elements that precede it in the list at its parent

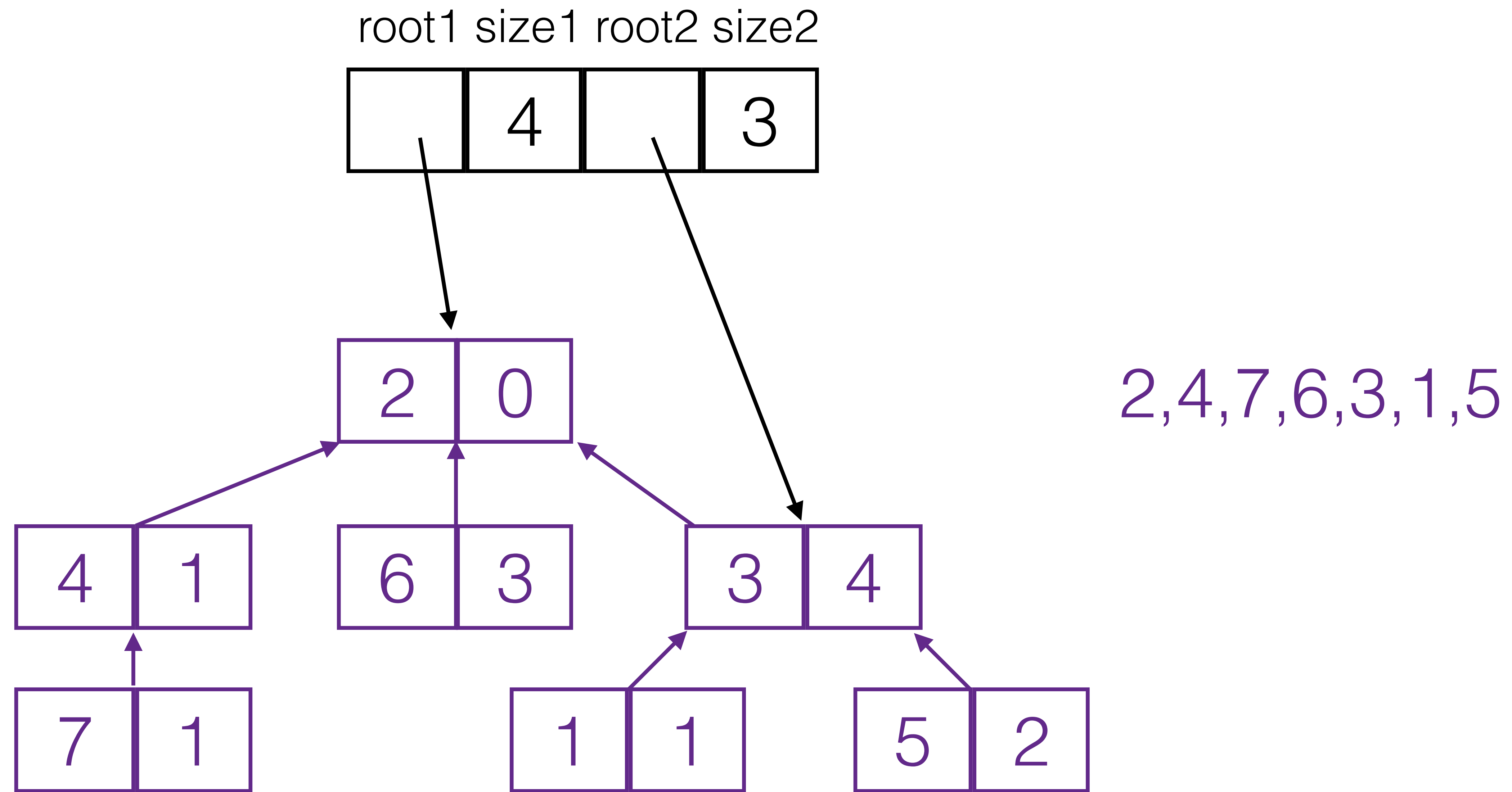
To concatenate 2 disjoint lists, given their sizes and pointers to their roots:



atomically, root2.ptr $\leftarrow$ root1

root2.offset $\leftarrow$ size1

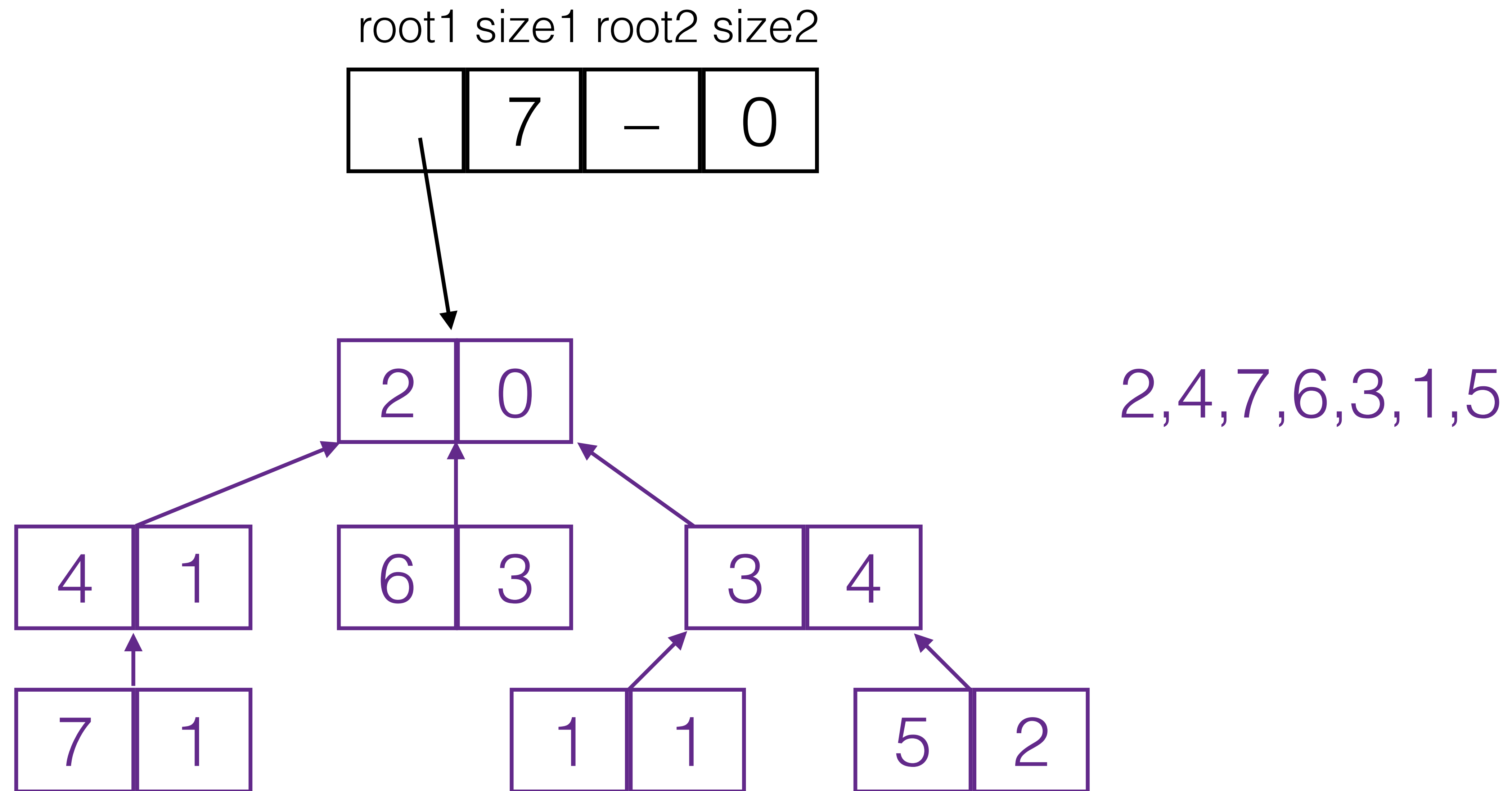
To concatenate 2 disjoint lists, given their sizes and pointers to their roots:



atomically, root2.ptr ← root1

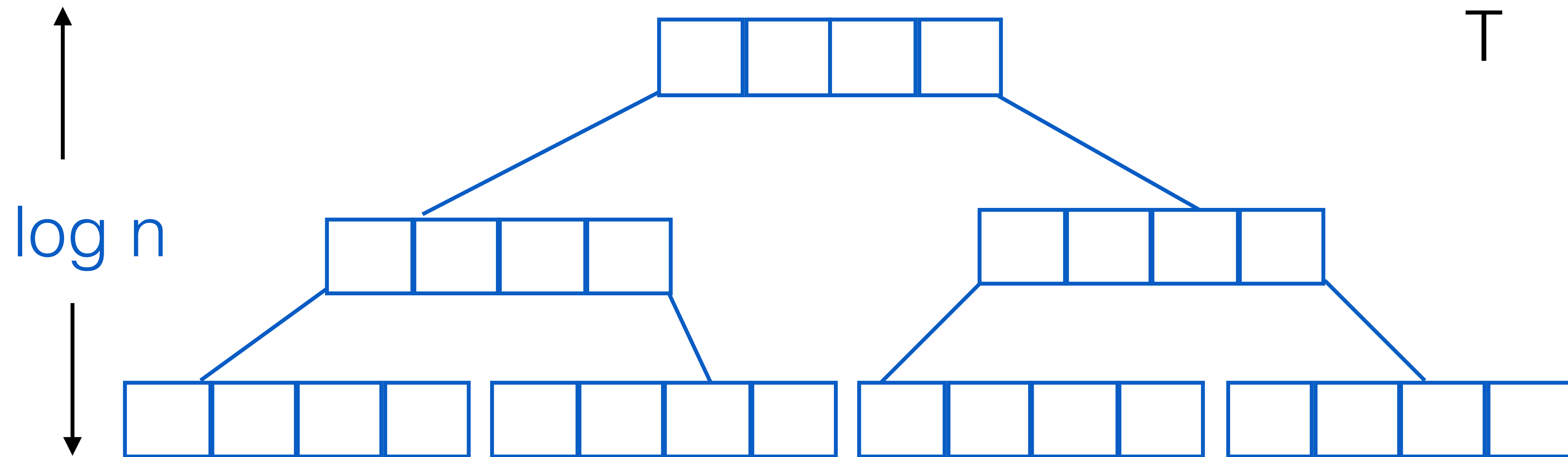
root2.offset ← size1

To concatenate 2 disjoint lists, given their sizes and pointers to their roots:



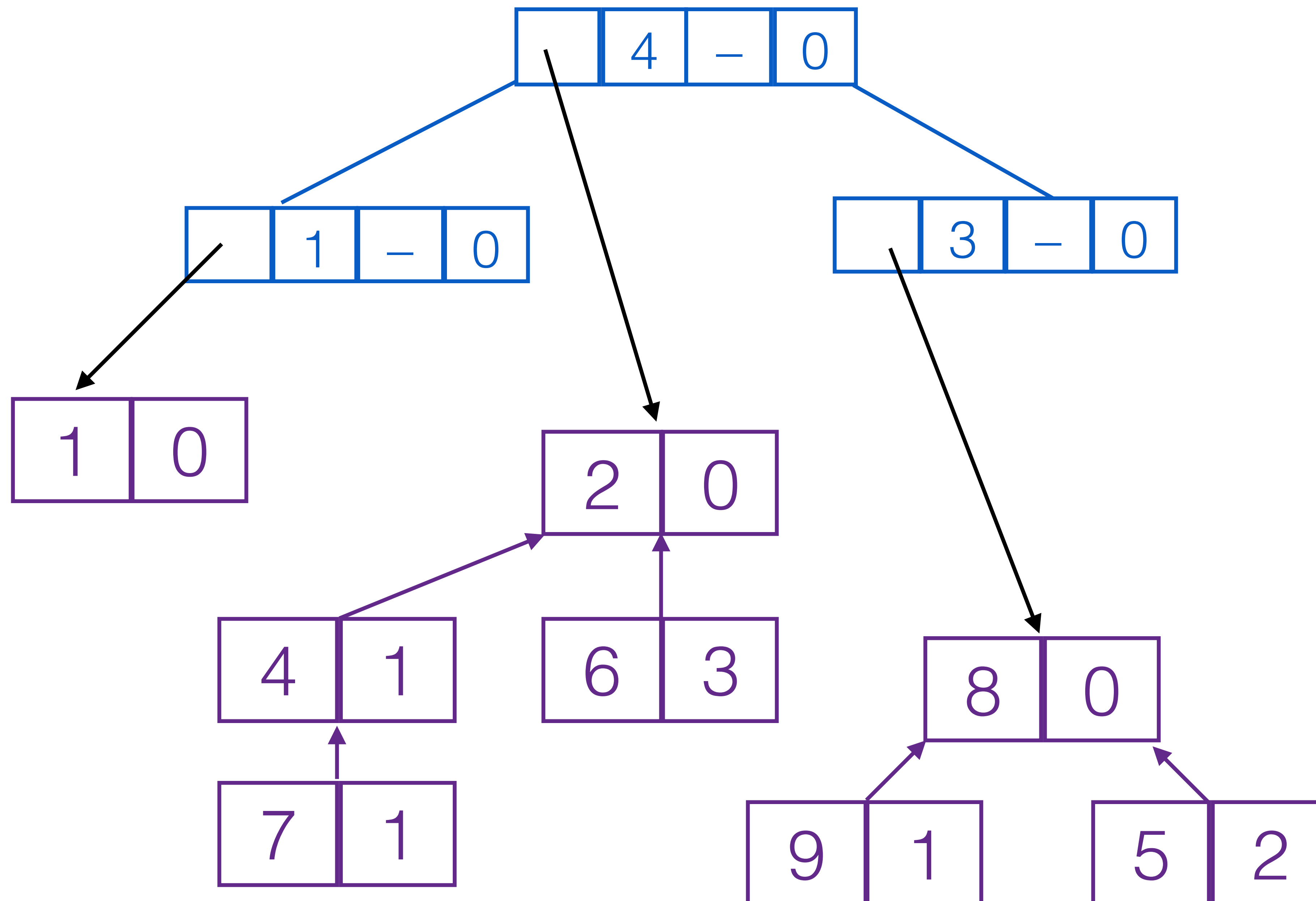
size of resulting list ← size1+ size2

An efficient wait-free implementation:

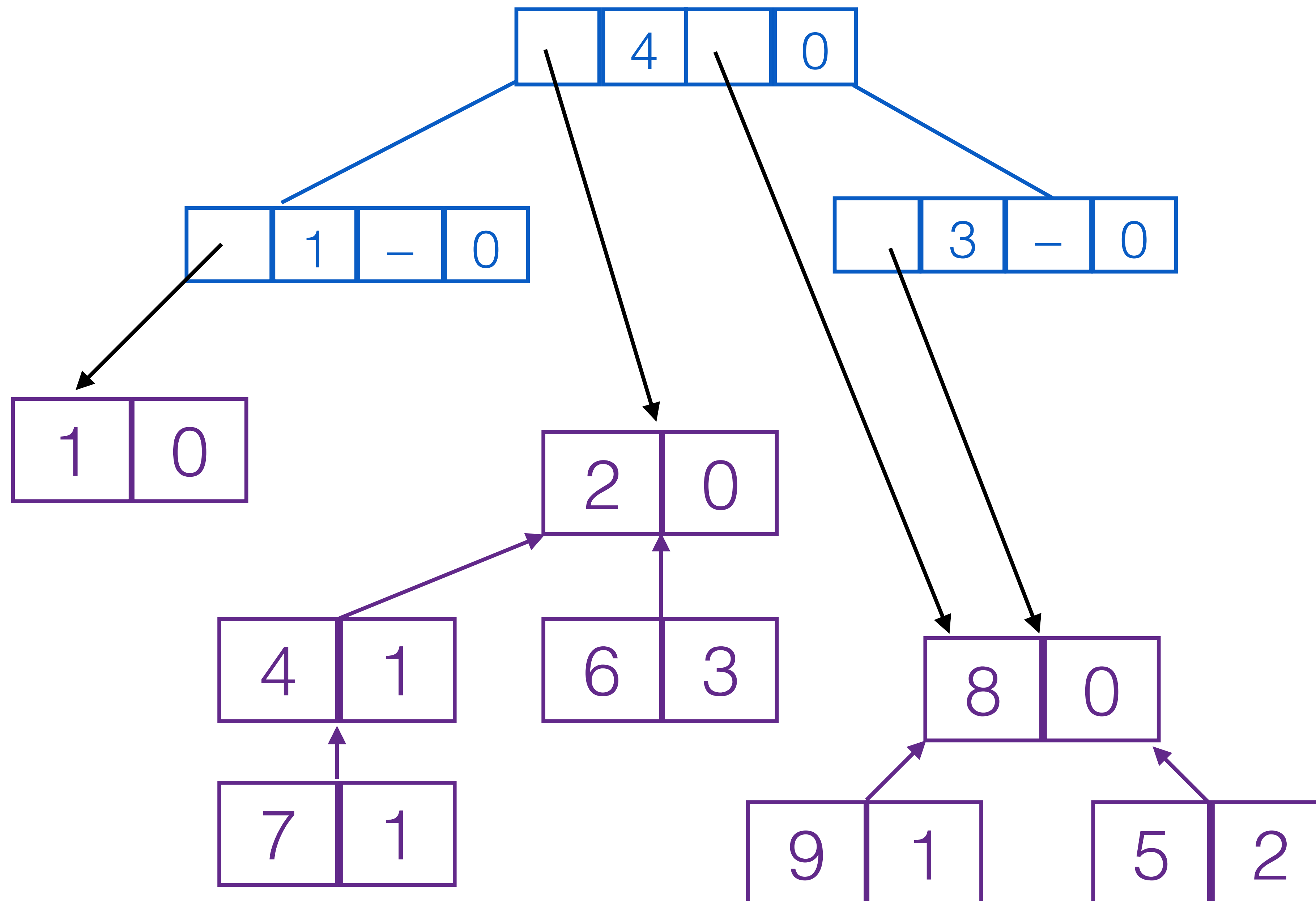


to perform FI, a process:
creates a list with 1 node at its leaf
moves it up the tree to the root,
concatenating when possible

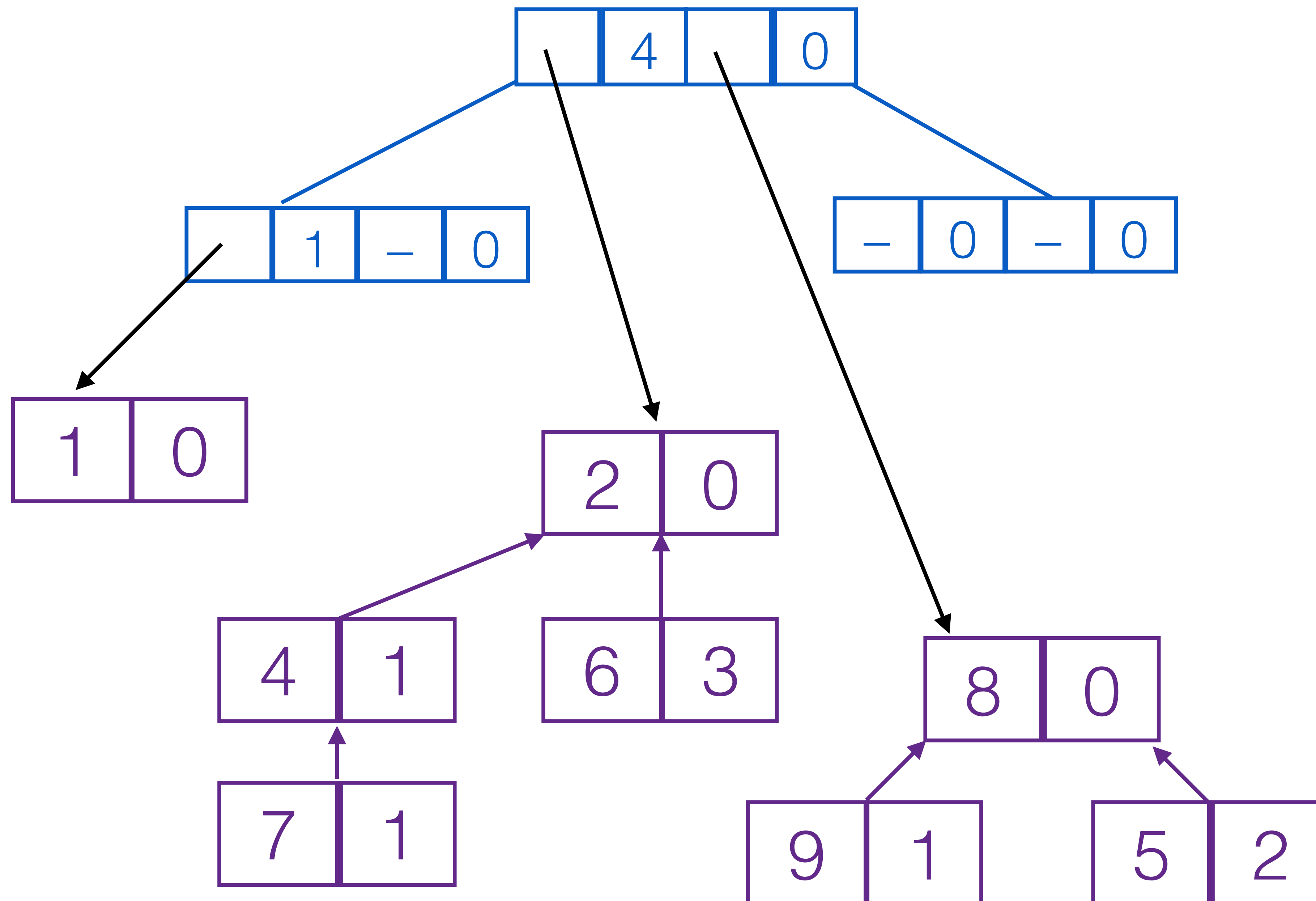
An efficient wait-free implementation:



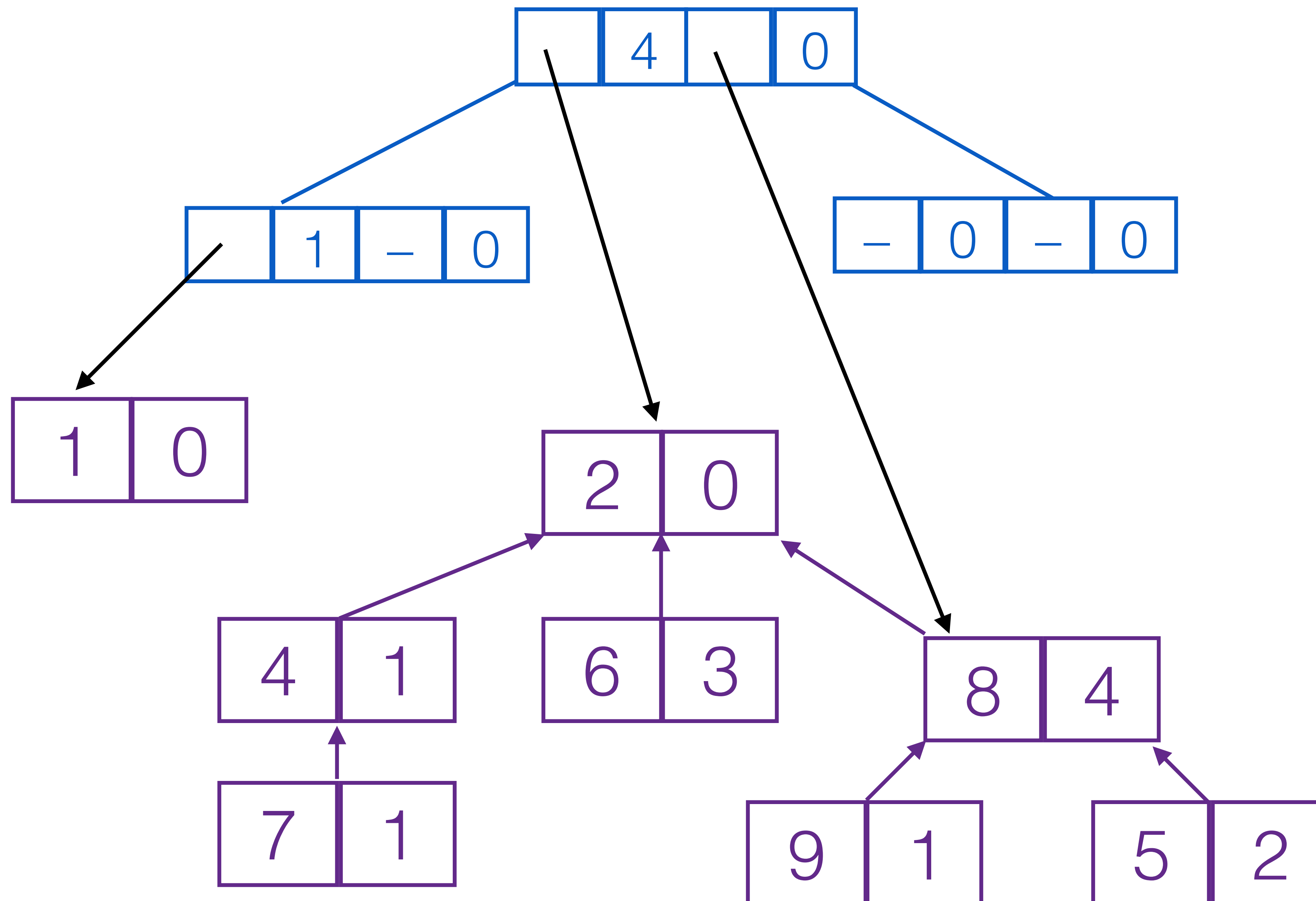
An efficient wait-free implementation:



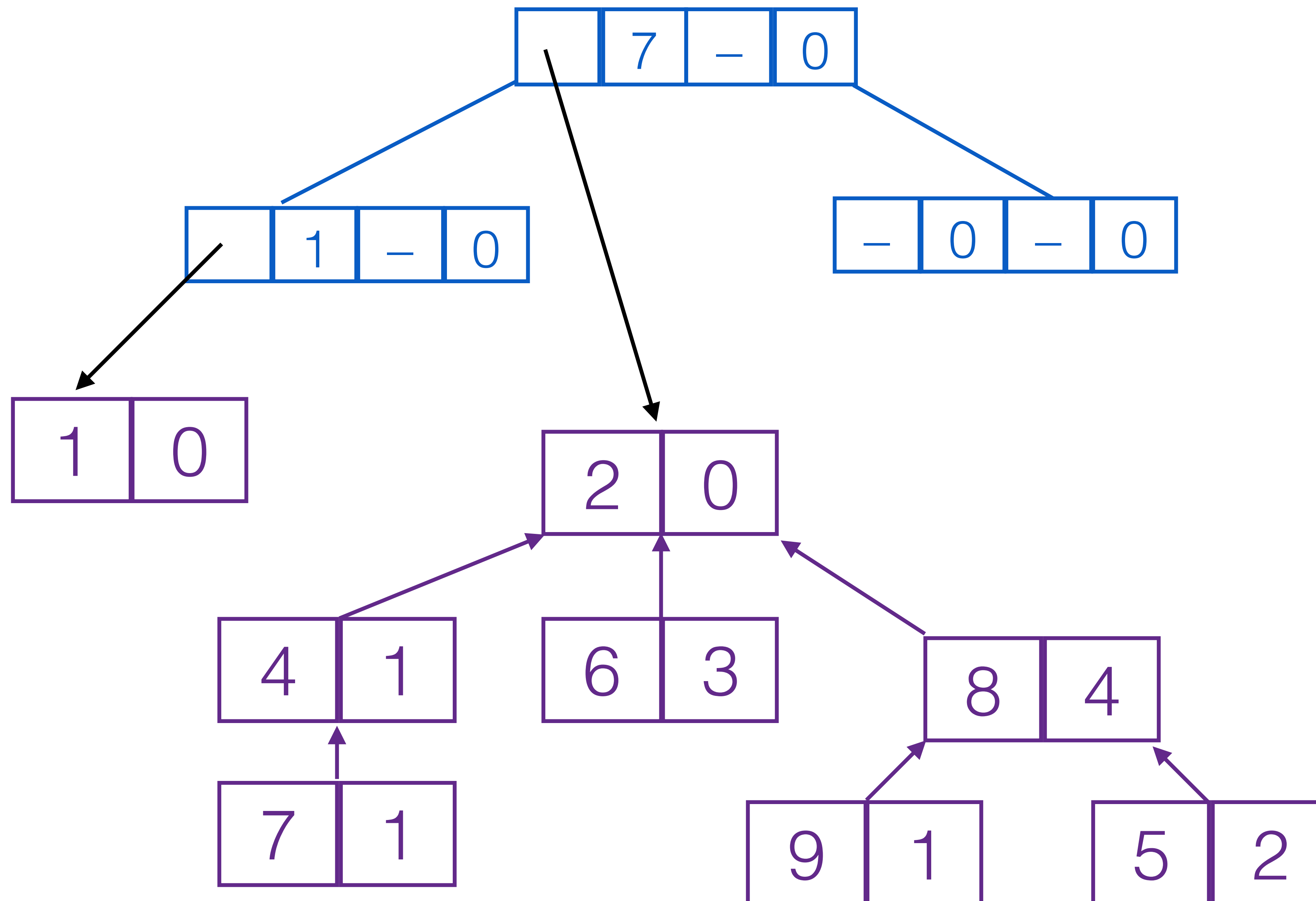
An efficient wait-free implementation:



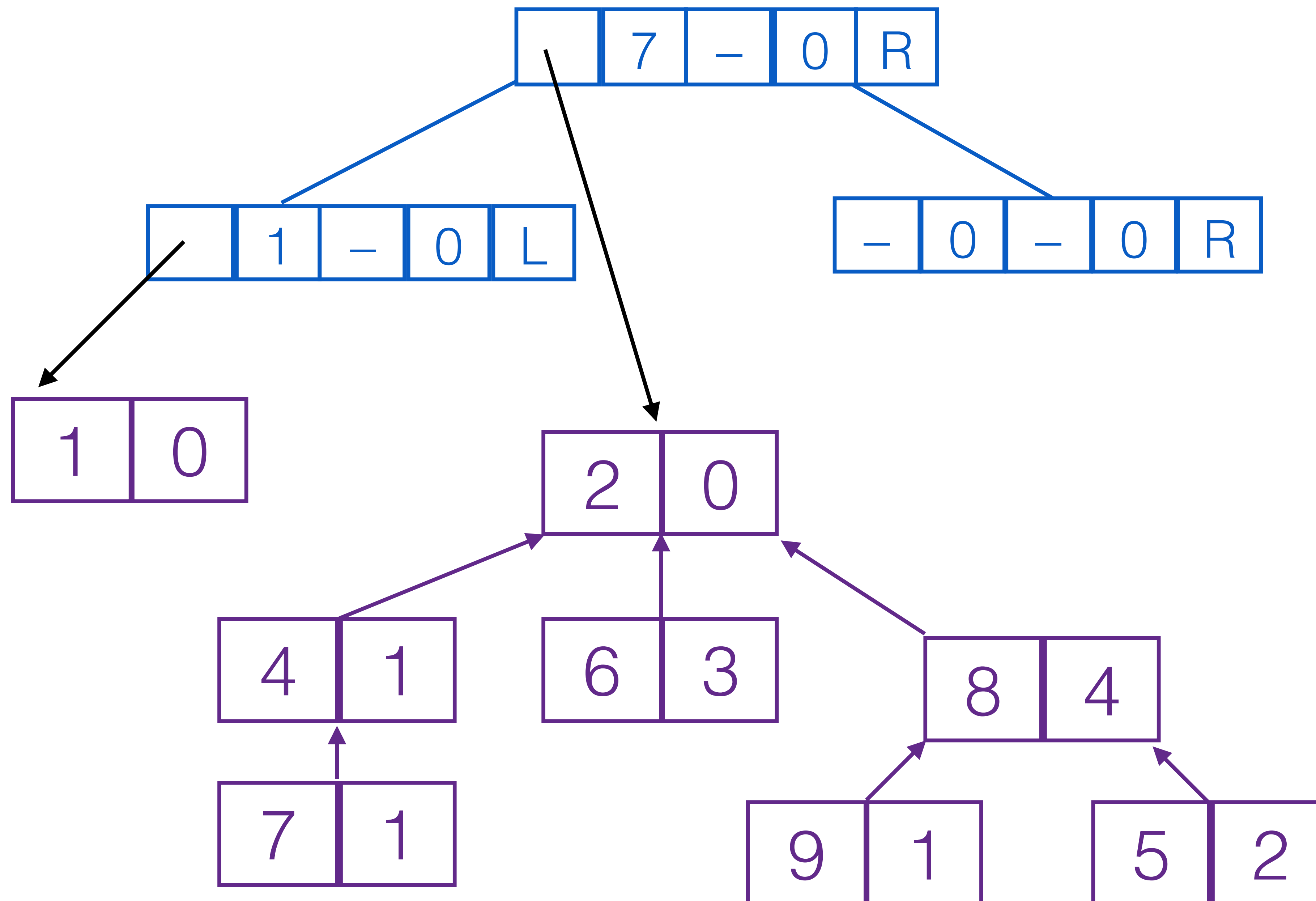
An efficient wait-free implementation:



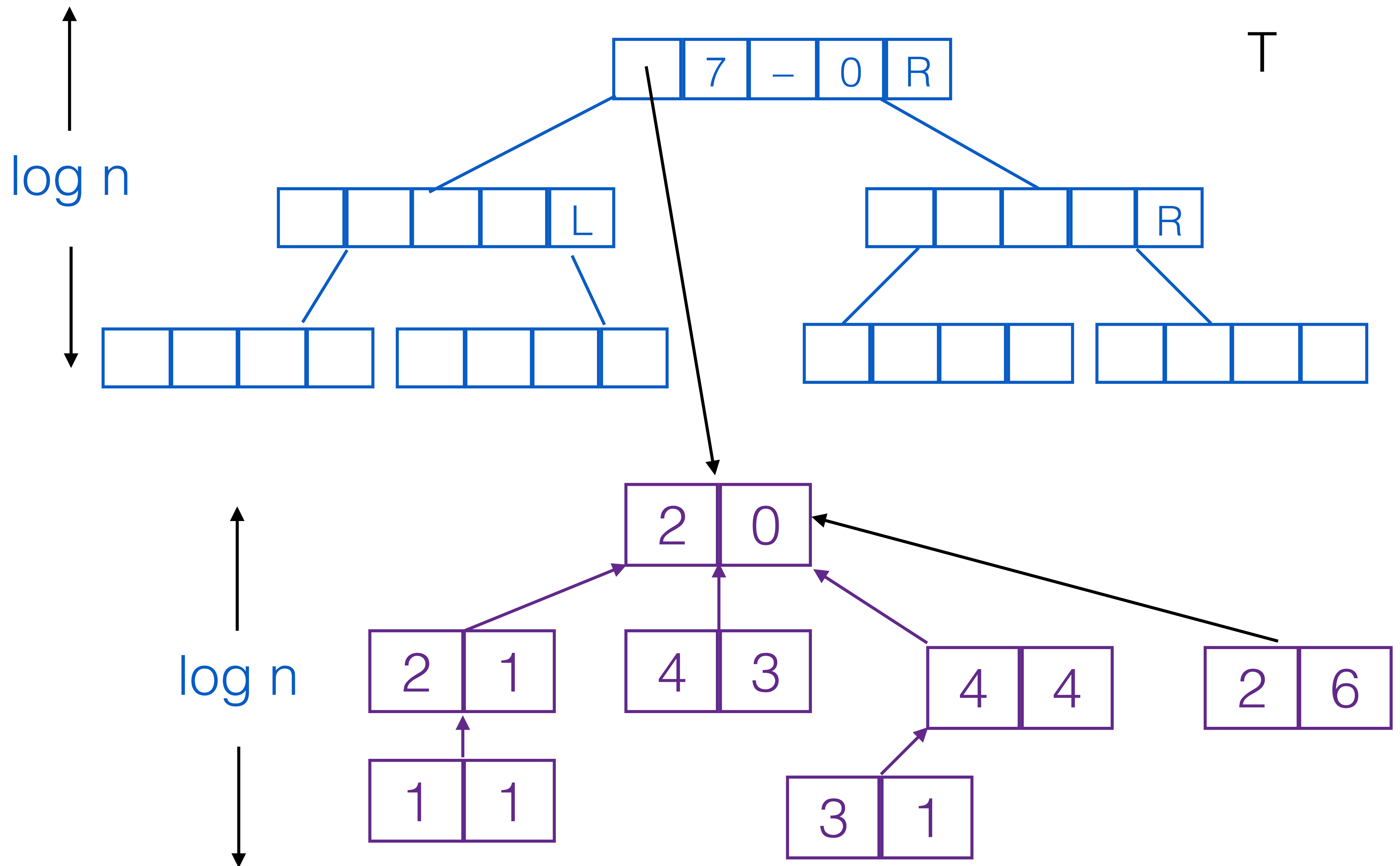
An efficient wait-free implementation:



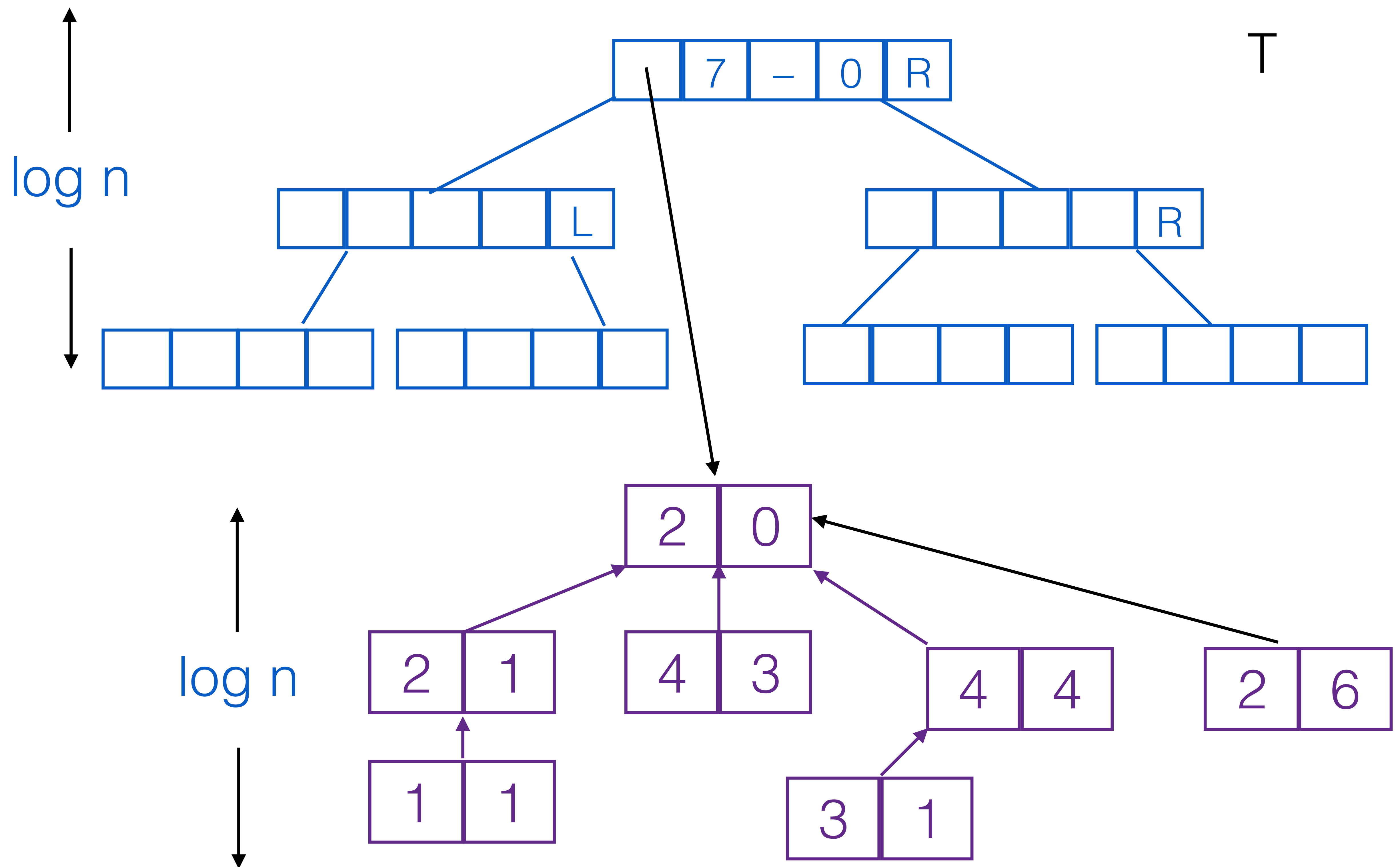
An efficient wait-free implementation:



Each FI takes O(log n) steps.



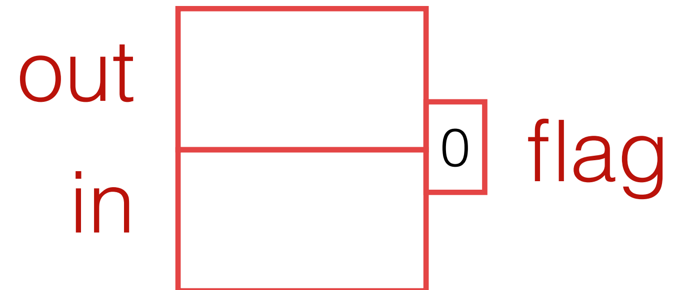
Each READ takes O(1) steps.



Information moving from a vertex shouldn't interfere with information moving to the vertex. Must ensure no information is lost or duplicated.

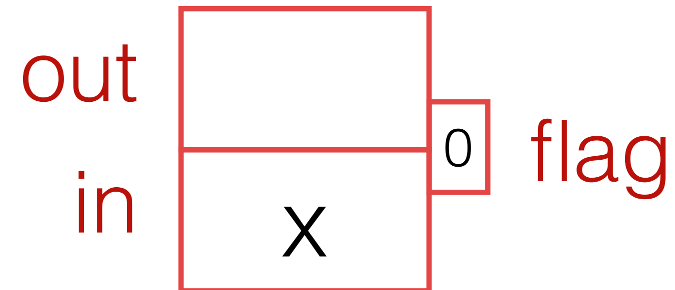
Information moving from a vertex shouldn't interfere with information moving to the vertex. Must ensure no information is lost or duplicated.

buffer object



Information moving from a vertex shouldn't interfere with information moving to the vertex. Must ensure no information is lost or duplicated.

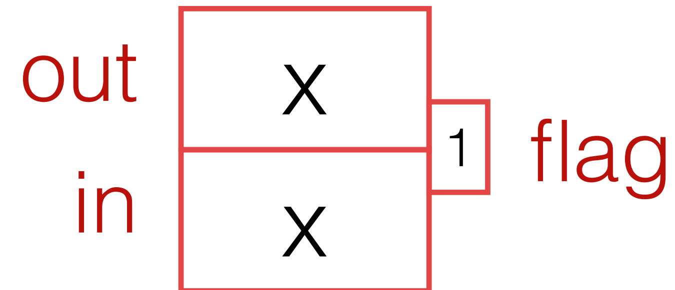
buffer object



- a value can be put into the in-buffer

Information moving from a vertex shouldn't interfere with information moving to the vertex. Must ensure no information is lost or duplicated.

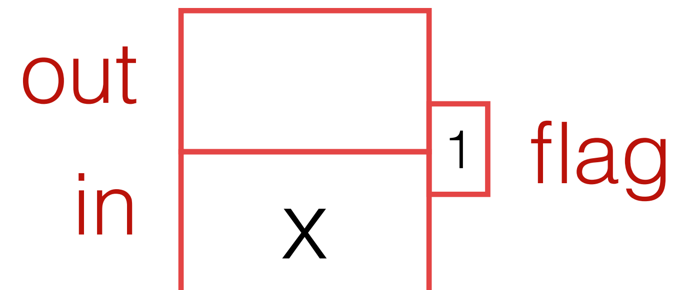
buffer object



- a value can be put into the in-buffer
- a value can be copied from the in-buffer to the out-buffer

Information moving from a vertex shouldn't interfere with information moving to the vertex. Must ensure no information is lost or duplicated.

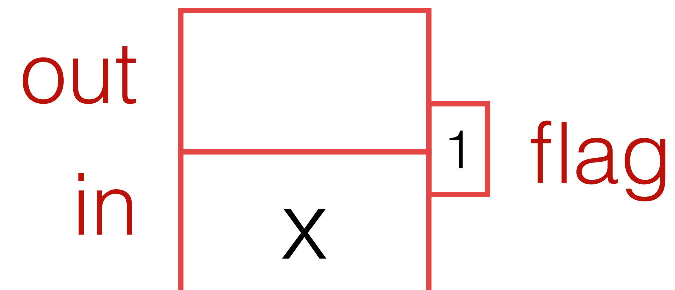
buffer object



- a value can be put into the in-buffer
- a value can be copied from the in-buffer to the out-buffer
- the out-buffer can be emptied

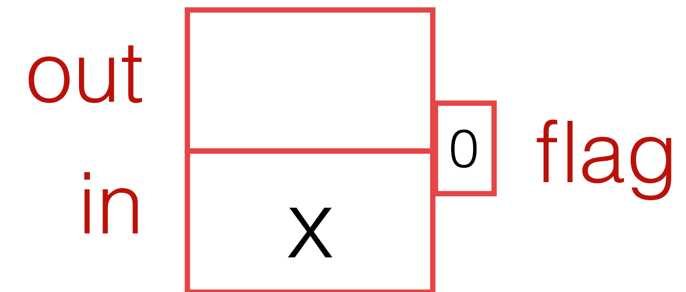
Information moving from a vertex shouldn't interfere with information moving to the vertex. Must ensure no information is lost or duplicated.

buffer object



- a value can be put into the in-buffer
- a value can be copied from the in-buffer to the out-buffer
- the out-buffer can be emptied
- flag indicates whether the value in the in-buffer has been successfully copied to the out-buffer

buffer object v



LLIn(v): returns the values of in and flag

SCIn(v,x): if, since the process last performed LLIn(v), no successful SCIn or Copy has been performed on v, stores the value x into in and resets flag to F

LLOut(v): returns the values of out

RCOut(v): if, since the process last performed LLOut(v), no successful RCOut or Copy has been performed on v, resets the value of out to $\perp$

Copy(v): copies in to out and sets flag to T, provided out = $\perp$, in $\neq$ $\perp$, and flag = F

THEOREM There is a linearizable wait-free implementation of a buffer object from 3 LL/SC objects so that each operation has $O(1)$ step complexity.

THEOREM There is a linearizable wait-free implementation of a buffer object from 3 LL/SC objects so that each operation has O(1) step complexity.

- one for out
- one for (in, flag)
- one which indicates that a Copy operation is in progress and other processes should help it complete

Theorem A wait-free, linearizable, unbounded Fetch&Increment object shared by n processes can be implemented using O(log m)-bit LL/SC objects, so that
each FI operation takes O(log n) steps and
each READ operation takes O(1) steps.

Theorem A wait-free, linearizable, unbounded Fetch&Add object shared by n processes can be implemented using O(log m)-bit LL/SC objects, so that each FA operation takes O(log n) steps and
each READ operation takes O(1) steps.

at each node, store the argument of the FA operation it represents and the sum of these values in the elements that precede it in the sequence at its parent, together with
the offset (the number of such elements)