The RT Patch

What needs to be done to get it into Mainline?

Steven Rostedt
rostedt@goodmis.org
srostedt@redhat.com

RT Patch, What is it?

- Goal: to make a deterministic operating system
- How?
  - Highest priority task responds immediately
  - Control of interrupts
  - Increase preemption areas
  - Prevent unbounded latency

Control of interrupts

- **Interrupt handlers are threads**
  - **Except for timer interrupts (something must control scheduling)**
- **Prioritize interrupt handlers (just like threads)**
- **Interrupts can be preempted**

Increase preemption areas

- Do not disable interrupts
- Do not disable preemption
  - Both prevent a task from being scheduled
- Most places disable preemption or interrupts with spin_lock()s
- PREEMPT_RT converts spin_locks into a sleepable mutex
  - They do not disable preemption nor interrupts.
  - Even spin_lock_irq() does not disable interrupts.

PREEMPT_RT what's in the patch?

- **Version I wrote this for: v4.0.5-rt4**
- **350 patches!**
- **415 files changed, 11713 insertions(+), 1927 deletions(-)**

```
 4.3% arch/x86/
 6.3% arch/
10.0% drivers/misc/
 4.0% drivers/
15.8% include/linux/
 9.1% kernel/locking/
 6.0% kernel/sched/
 5.3% kernel/time/
10.5% kernel/trace/
16.0% kernel/
 3.4% mm/
```

History of the RT patch

- **Started in 2004 by Ingo Molnar**
- **Maintained by Thomas Gleixner**
- **Responsible for:**

High resolution timers
Generic interrupts
mutex code
Priority inheritance futexes
lockdep
RT scheduler
ftrace / latency tracers
Conversions of semaphores to completions
SMP stability!

How does the RT patch make Linux RT?

- Add high resolution timers
- Add interrupts as threads
- Add priority inheritance
- Turn spin locks into sleeping locks

How does the RT patch make Linux RT?

- Add high resolution timers
- Add interrupts as threads
- Add priority inheritance
- Turn spin locks into sleeping locks

How does the RT patch make Linux RT?

- Add high resolution timers
- Add interrupts as threads
- Add priority inheritance
- **Turn spin locks into sleeping locks**

How does the RT patch make Linux RT?

- Add high resolution timers
- Add interrupts as threads
- Add priority inheritance
- **Turn spin locks into sleeping locks**
  - **Spin locks have lots of side effects**
  - **These side effects are depended on**

Spin lock side effects

- **Disables preemption**
- **Some disable interrupts**
- **Some disable softirqs**
- **Disables migration**

Spin lock side effects

- **Disables preemption**
- **Some disable interrupts**
- **Some disable softirqs**
- **Disables migration**
  - This is the easy one
  - rt_mutex (what spin locks are converted to) disable migration
  - Solves accessing per cpu variables
  - May prevent RT tasks from migrating if preempted by higher priority task

Spinning lock or sleeping lock?

- **When a lock is converted to an rt_mutex**
  - **It's all or nothing!**
  - **Every use case of the lock is the same (sleep or spin)**
  - **Sleeping locks can not be taken when preempt/irq is disabled**
    - **May use trylock()**
  - **Sleeping locks can not be taken in hard interrupts**
    - **May NOT use trylock()!!!**

Taking sleeping lock in interrupt

- **Sleeping locks have priority inheritance**
- **Priority inheritance detects deadlocks**
- **Sleeping locks can be interrupted by interrupt**

Taking sleeping lock in interrupt

```
    CPU0                        CPU1
                              lock(A)
lock(A) <block>
  <interrupt>
      trylock(B)

                              lock(B) <block>
```

Looks to be ABBA deadlock, but it is not.

Wait queues

- **Wait queues can be complex**
  - **wake_up()**
  - **wake_up_nr()**
  - **wake_up_all()**
  - **wake_up_interruptible()**
  - **wake_up_interruptible_nr()**
  - **wake_up_interruptible_all()**
  - **wake_up_interruptible_sync()**

Wait queues

- **Wait queues can be long**
  - **Holds locks for long periods of time**
  - **Must be rt_mutex such that we can preempt**
  - **There's critical sections that use wait queues**
- **What to do?**

Simple wait queues

- **Uses raw spin locks**
- **90% of cases don't need the complexity**
- **Wake up one type only (interruptible or non-interruptible)**
- **Wake all can still be an issue**
  - **Still need to avoid holding a raw lock for a long time**
- **Currently trying to get simple wait queues in**
  - **Need to solve the wake all issue**

The trylock deadlock

- **When you need to take a lock in reverse order**
- **Works fine when you can't preempt**

```
again:
   spin_lock(&a);
   if (!spin_trylock(&b)) {
      spin_unlock(&a);
      goto again;
   }
```

The trylock deadlock

- **But when you can preempt, it's an issue!**
- **If Task-Y is an RT task > prio than Task-X it will loop forever.**

```
<Task-X>
spin_lock(&b);
<preempt => schedule Task-Y>
again:
   spin_lock(&a);
   if (!spin_trylock(&b)) {
      spin_unlock(&a);
      goto again;
   }
```

The trylock deadlock

- **One solution (great when applicable)**

```
again:
   spin_lock(&a);
   if (!spin_trylock(&b)) {
      spin_unlock(&a);
      spin_lock(&b);
      spin_unlock(&b);
      goto again;
   }
```

The trylock deadlock

- **But not always applicable**

```
fs/dcache.c:
dput() {
    repeat:
        if (fast_dput(dentry)) {
            return;
        } /* has dentry->lock */
    [...]
    dentry = dentry_kill(dentry);
    if (dentry) goto repeat;

dentry_kill() {
    if (!spin_trylock(&inode->i_lock))
        goto failed;
    if (!spin_trylock(&parent->i_lock)) {
        spin_unlock(&inode->i_lock);
        goto failed;
```

The trylock deadlock

You've heard of “cpu_relax()”?

The trylock deadlock

cpu_chill()

```
failed:
   spin_unlock(&dentry->lock);
   cpu_chill();
   return dentry;
```

The trylock deadlock

- **cpu_chill()**
- **A hack!**
  - **!PREEMPT_RT(), it is equivalent to cpu_relax()**
  - **otherwise**

```
static inline void cpu_chill(void)
{
        msleep(1);
}
```

per cpu variables

- **In mainline, just need to disable preemption**
  - That is bad for PREEMPT_RT
- **Disabling migration is not enough**
  - get_cpu() / put_cpu()
  - get_cpu_light() / put_cpu_light() when it is enough
- **Need to add locking**
  - local_lock()
  - local_spin_lock()
  - get_locked_var()
  - put_locked_var()

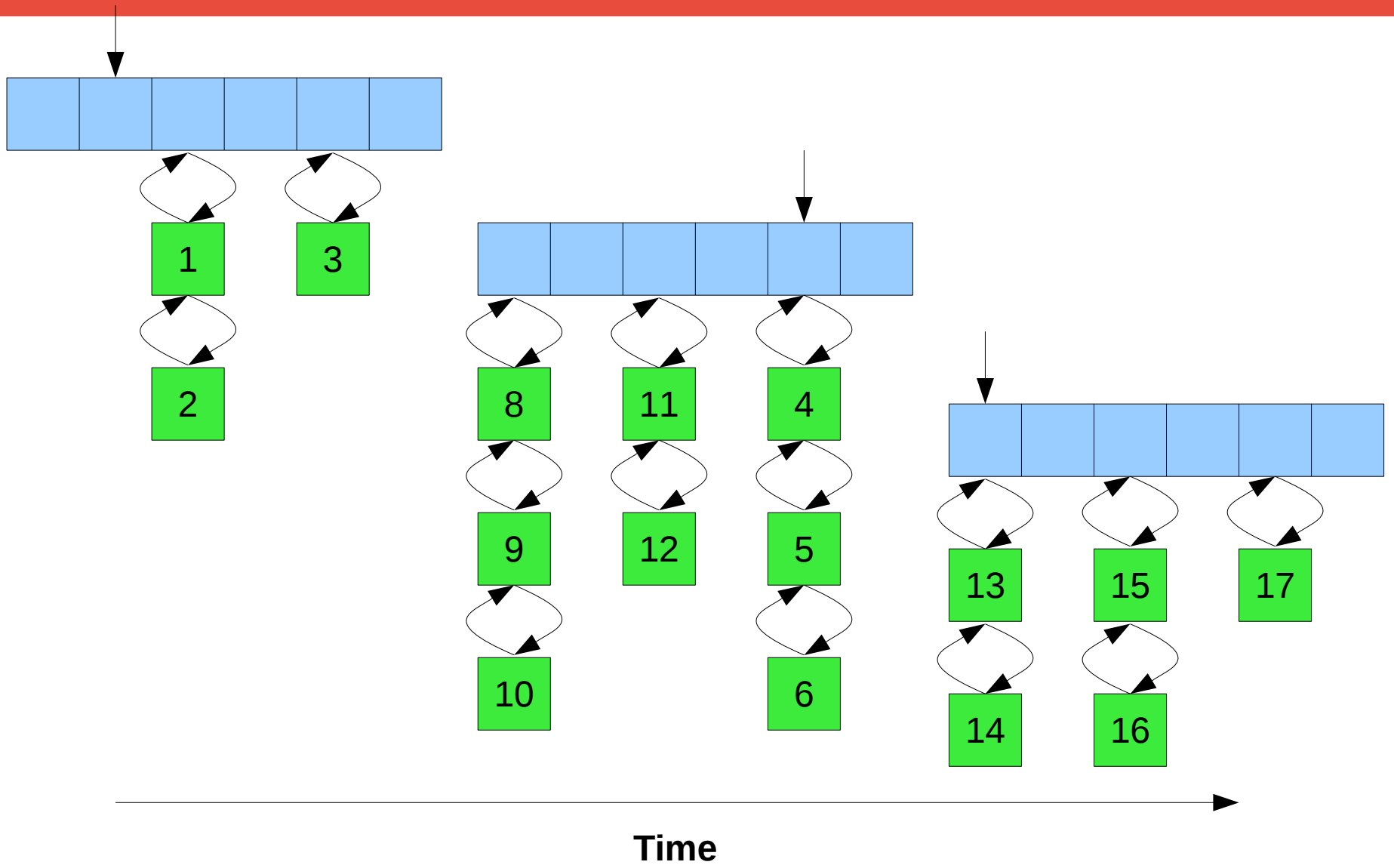
Local locks

- **preempt_disable -> local_lock(lvar)**
- **local_irq_disable -> local_lock_irq(lvar)**
- **local_irq_save -> local_lock_irqsave(lvar)**
- **get_cpu -> local_lock_cpu(lvar)**
- **get_cpu_var -> get_locked_var(lvar, var)**
- **spin_lock_irq() -> local_spin_lock_irq(lvar, lock)**
  - **Used when a per cpu variable is protected by the preemption or irq disabling of a spin lock**

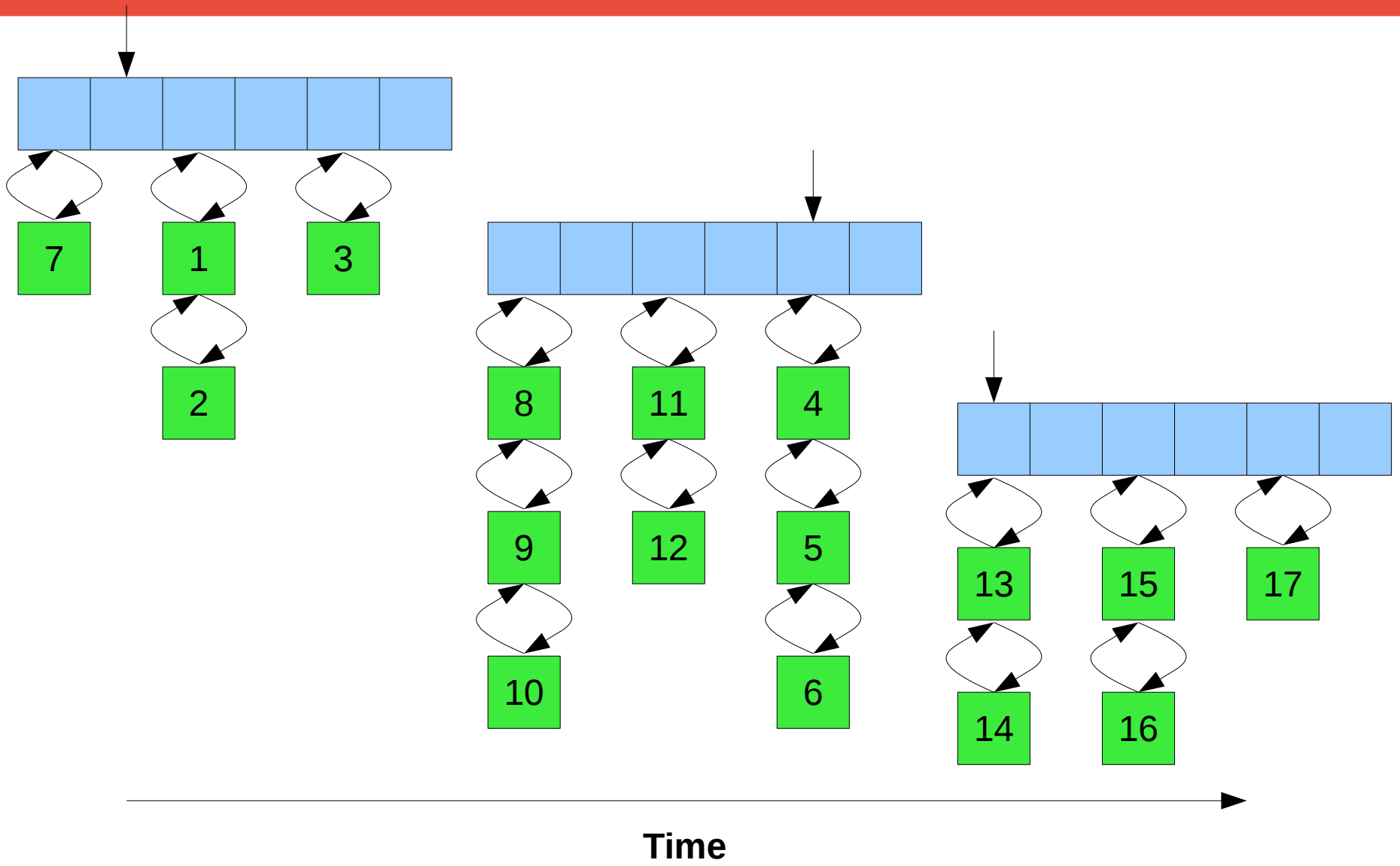
The Timer Wheel

- **Fast add and remove**
  - O(1)
- **Great for timeouts**
  - Network timers
    - Hardly trigger (most likely removed)
- **But has a “cascade” effect**
  - moves one list to the next for individual events
  - 5 levels, first 256 entries, 64 entries for the rest

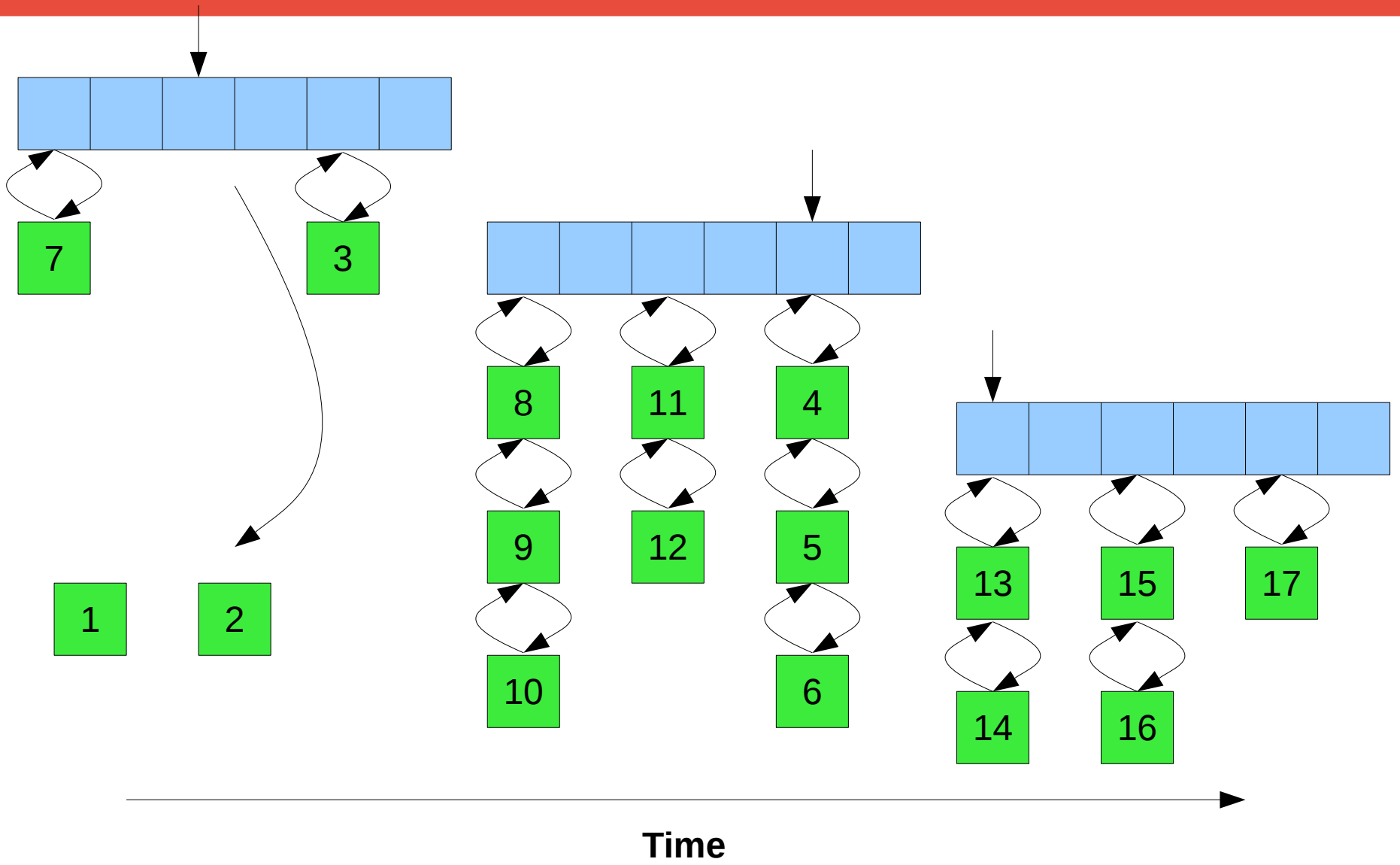
The Timer Wheel



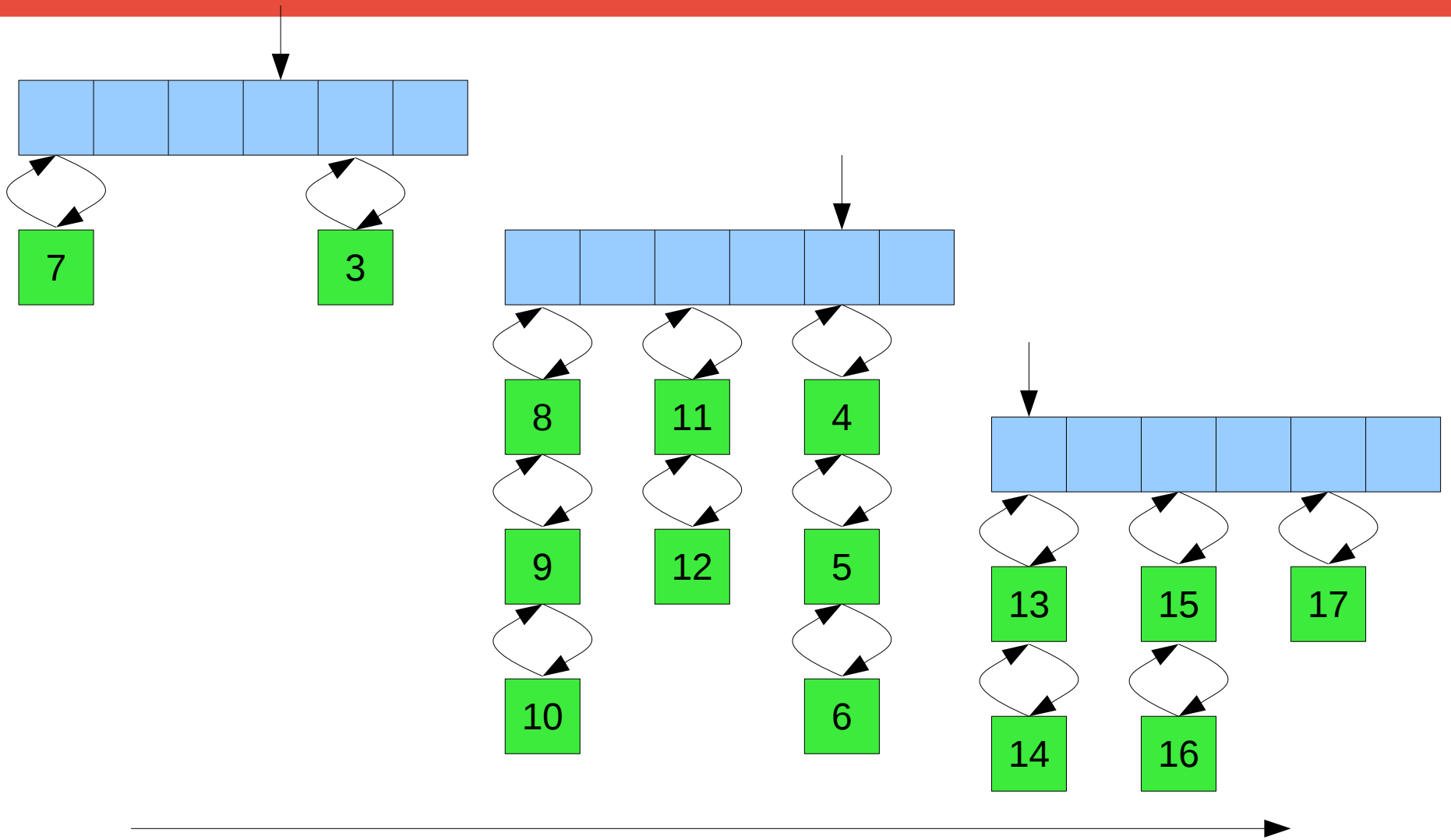
The Timer Wheel



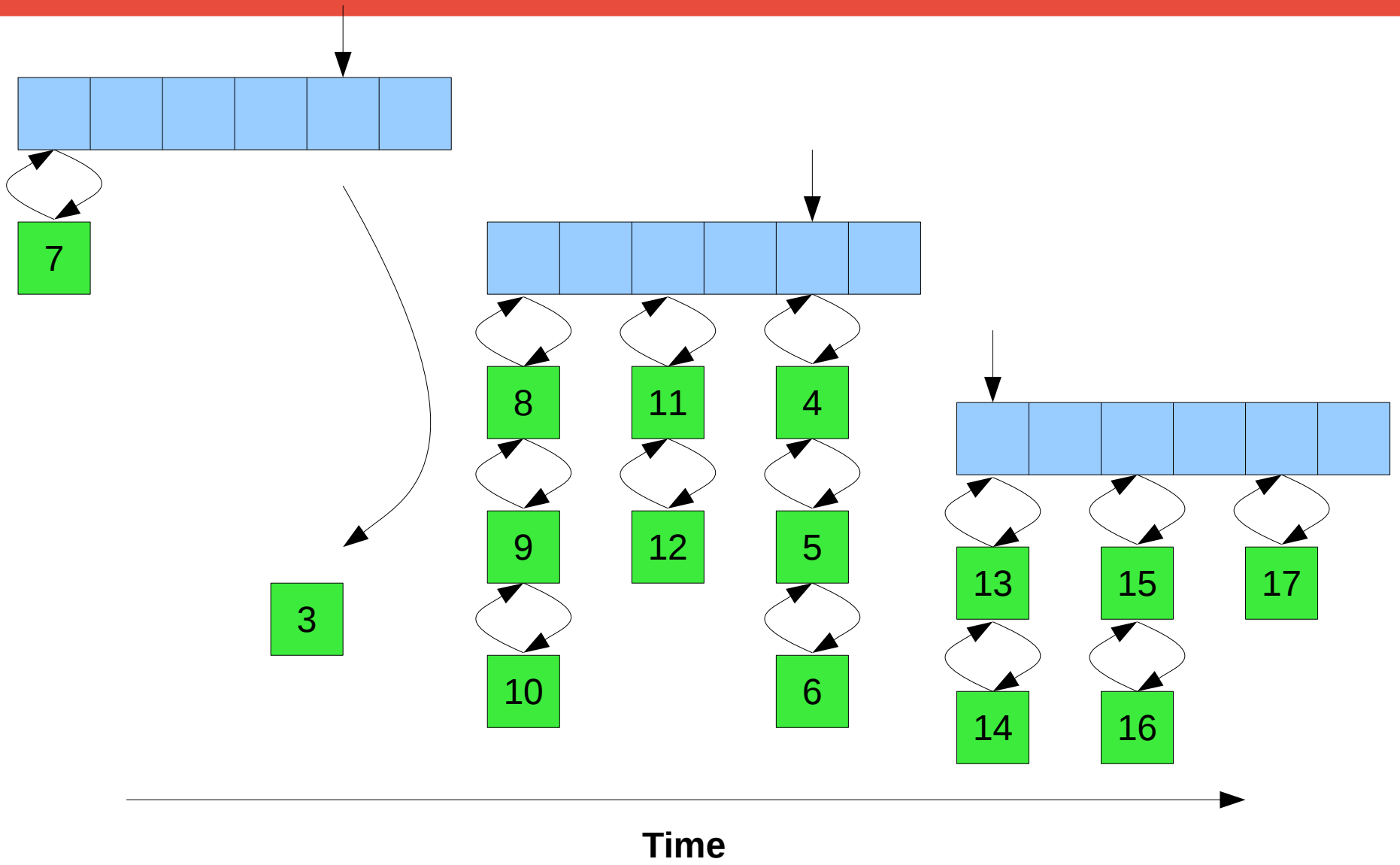
The Timer Wheel



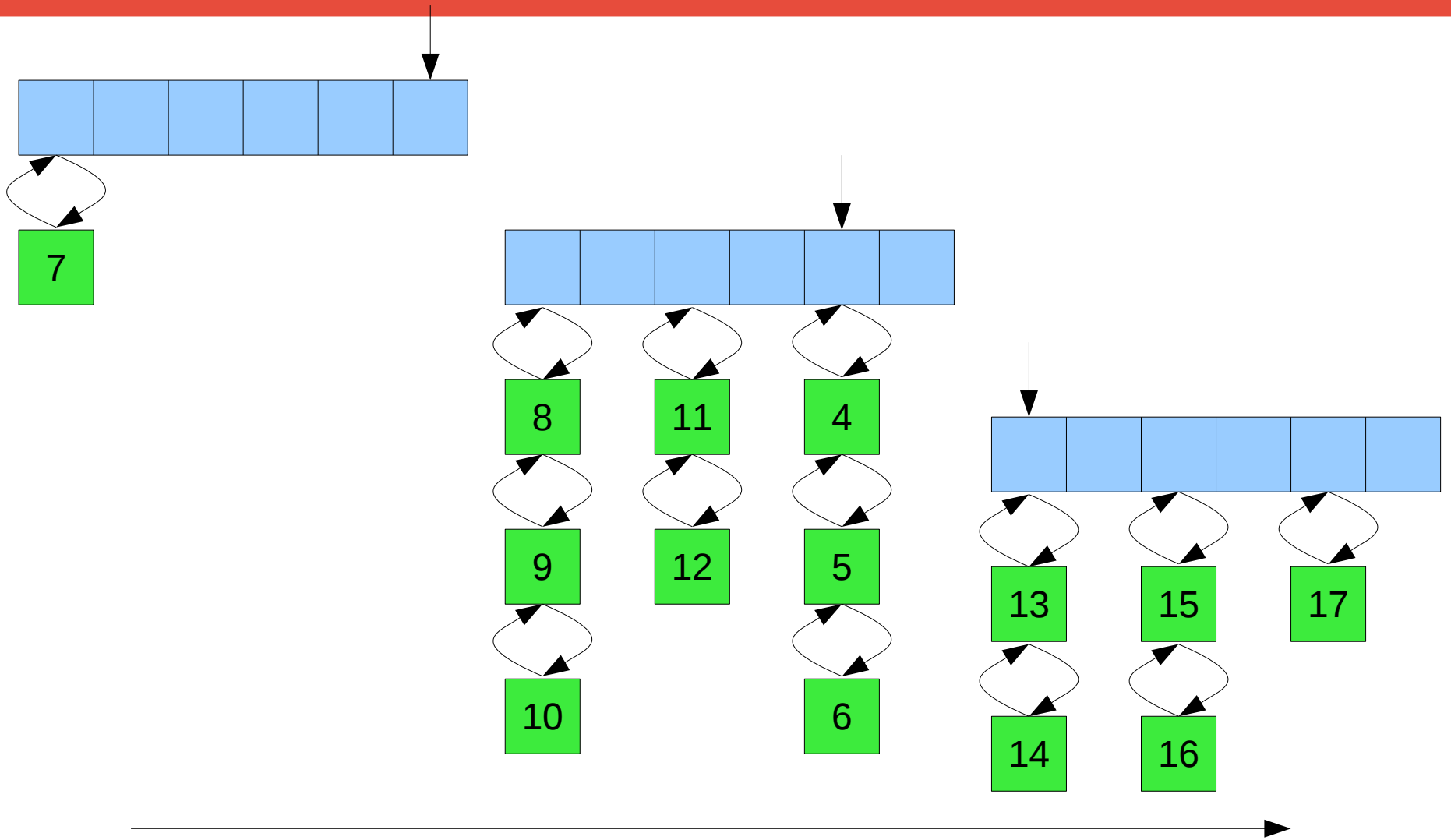
The Timer Wheel



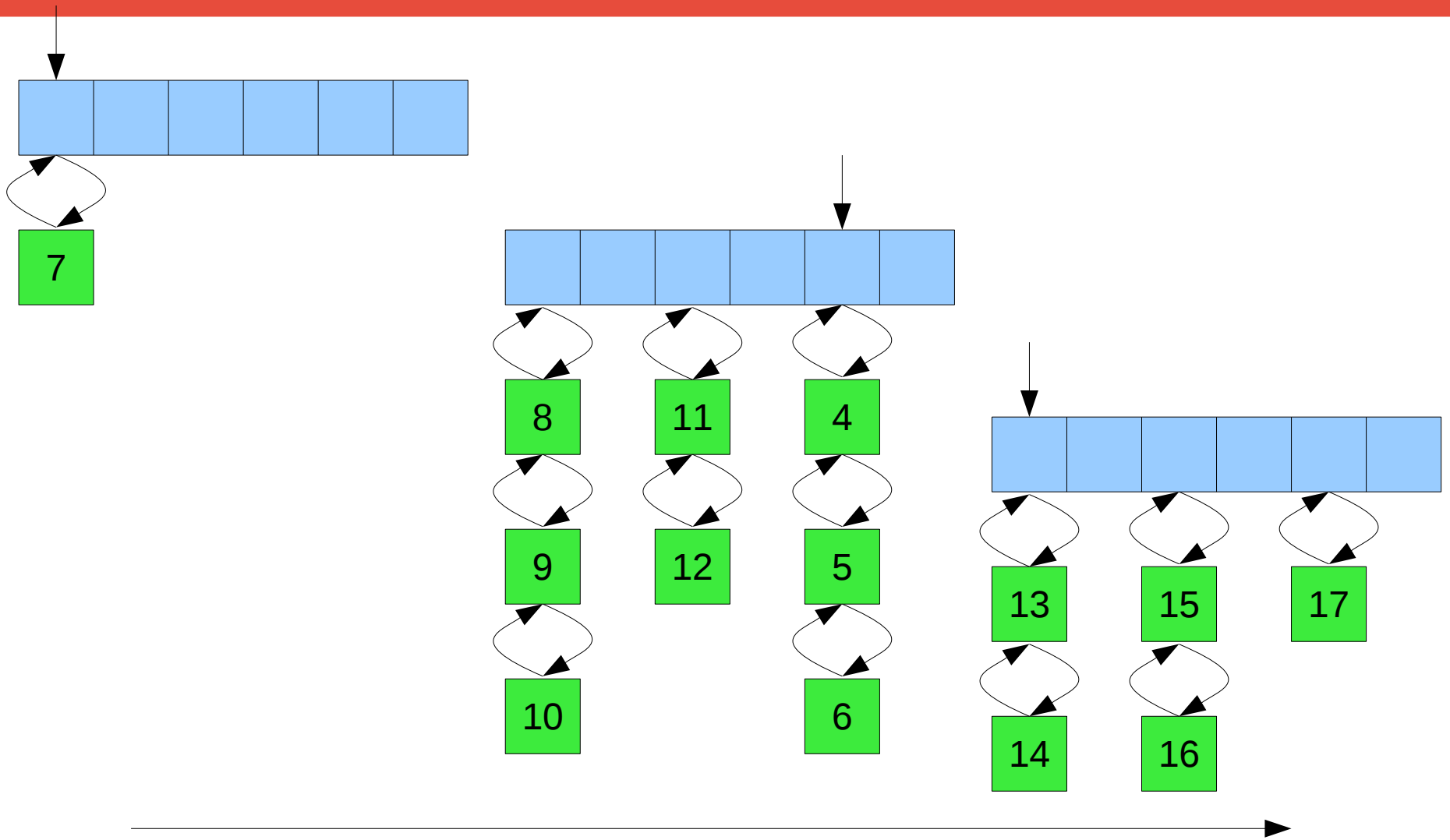
The Timer Wheel



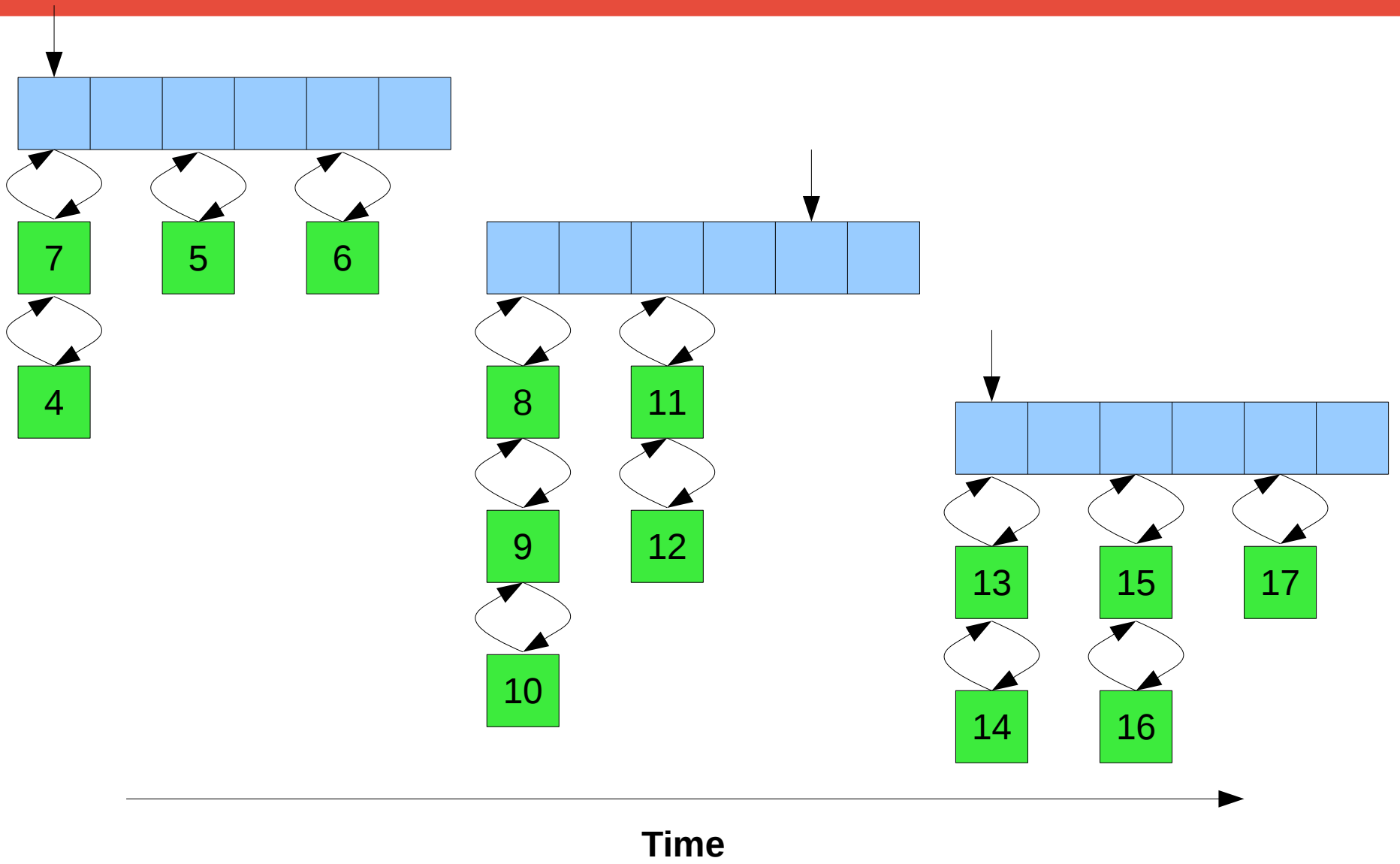
The Timer Wheel



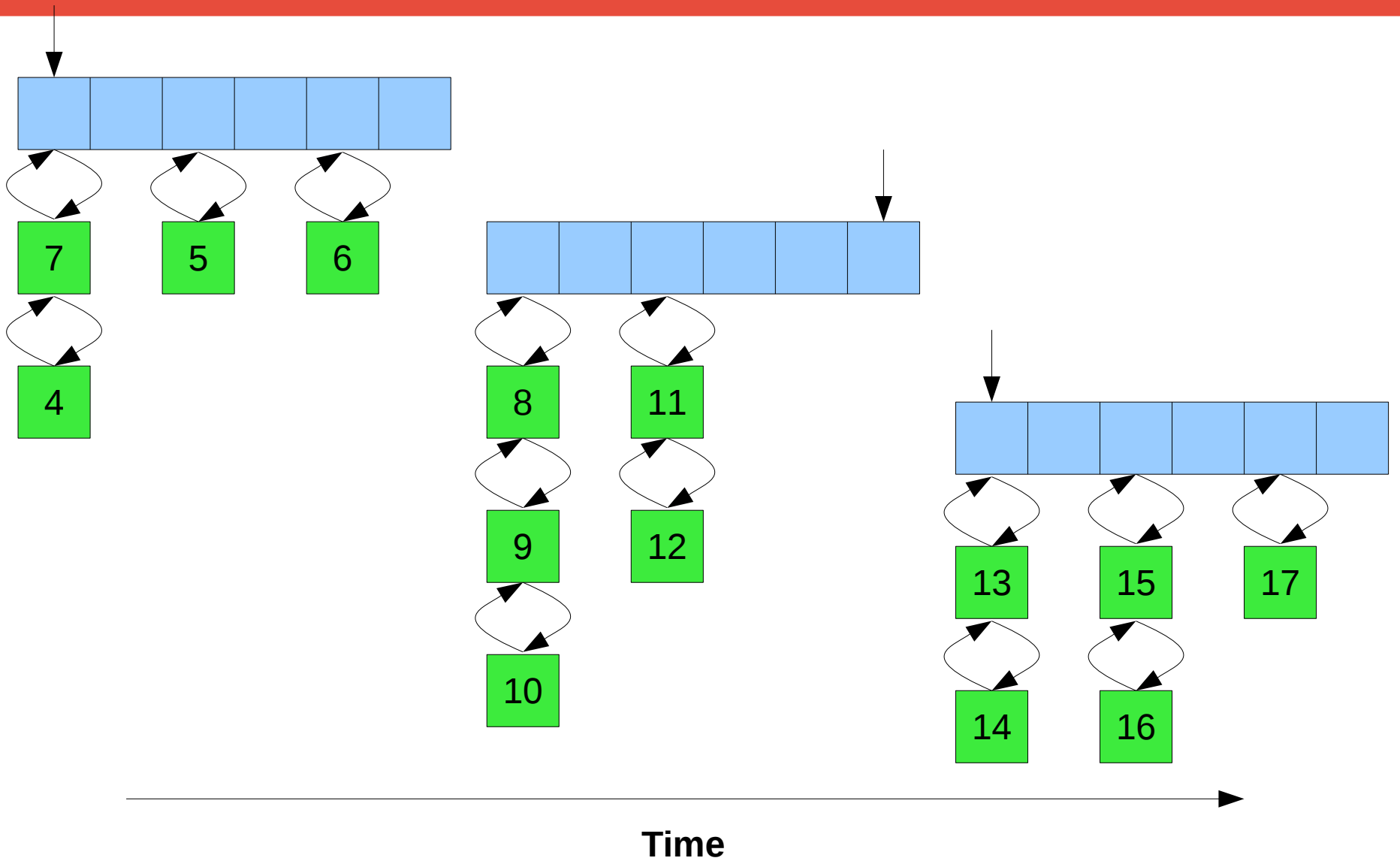
The Timer Wheel



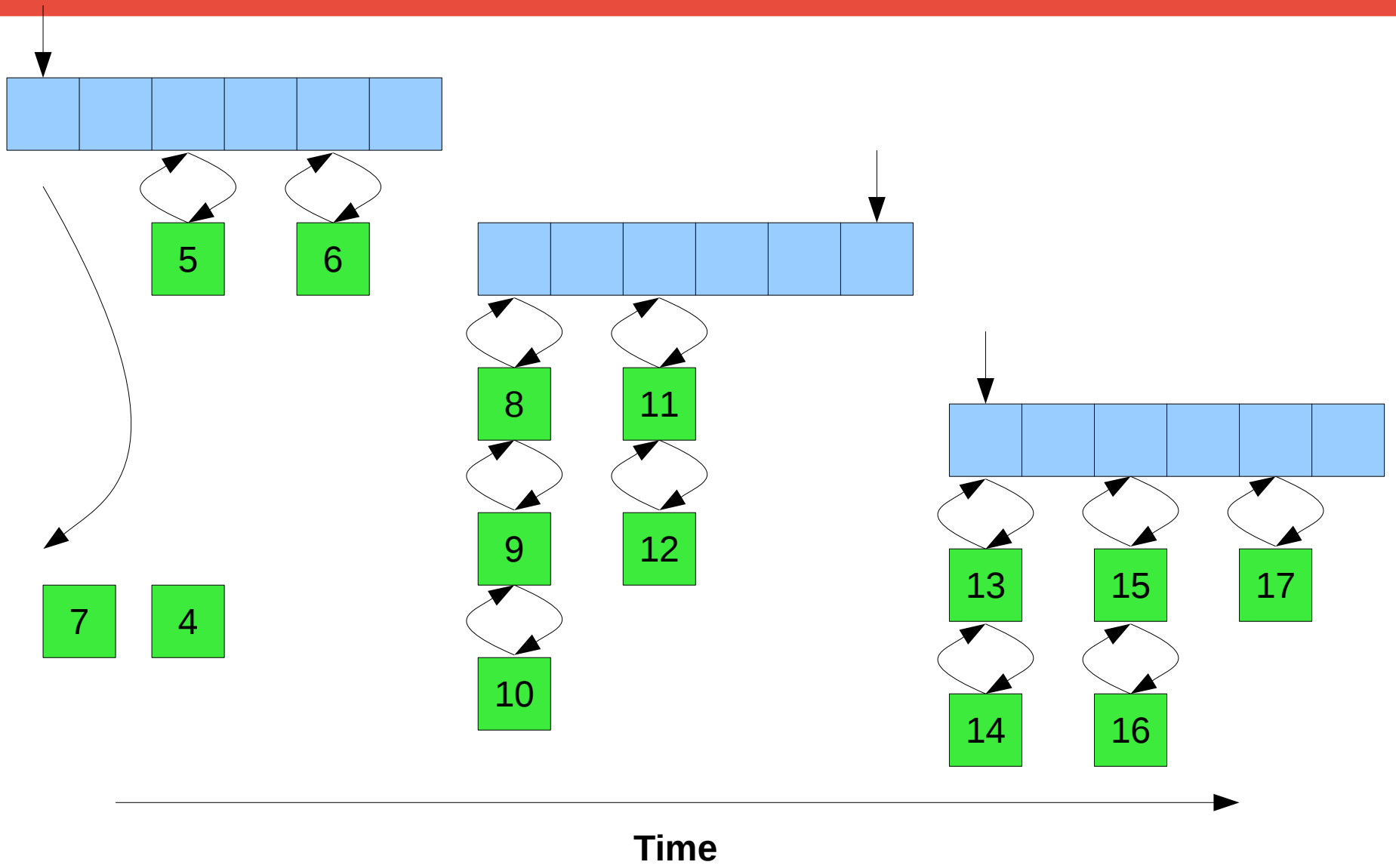
The Timer Wheel



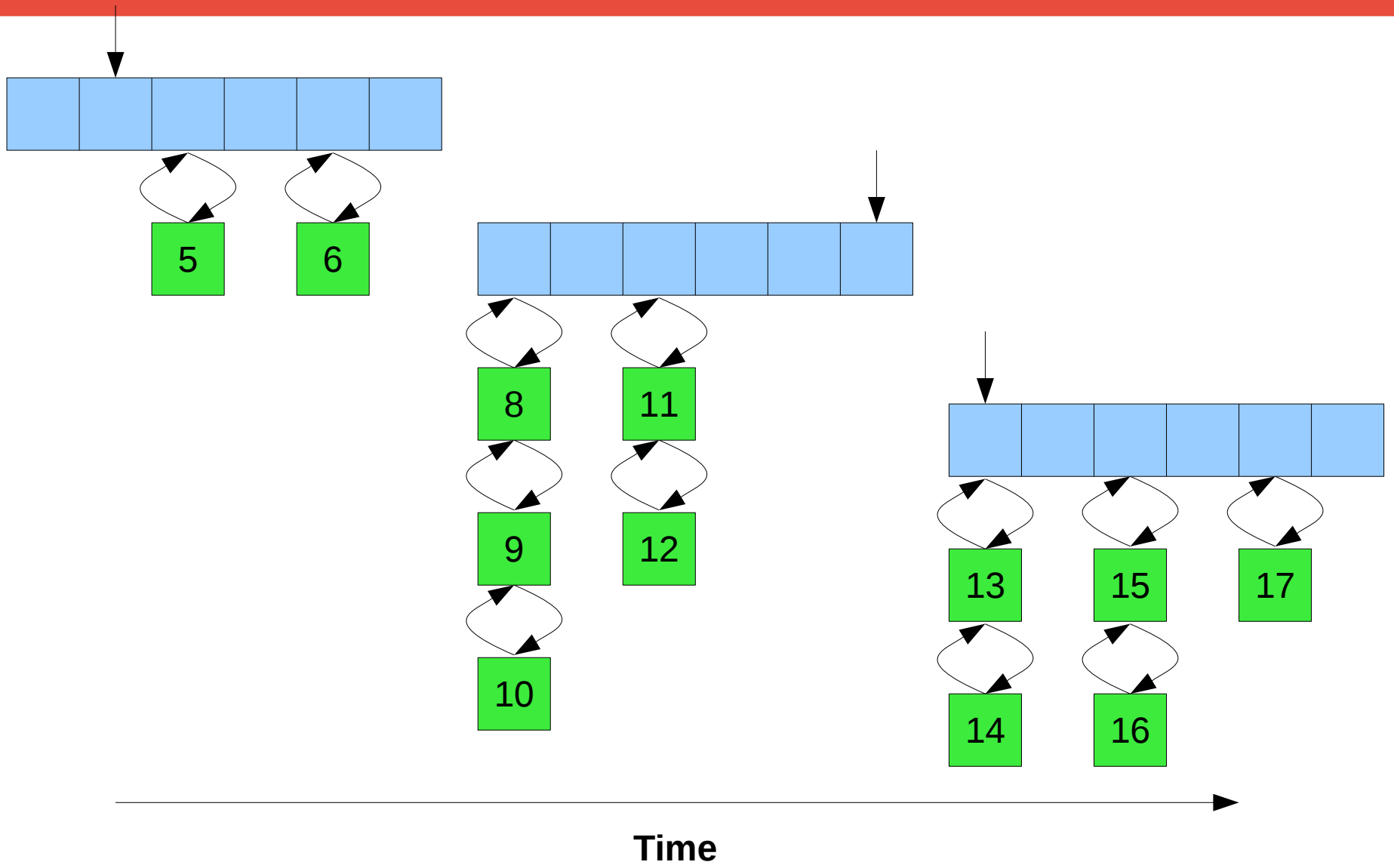
The Timer Wheel



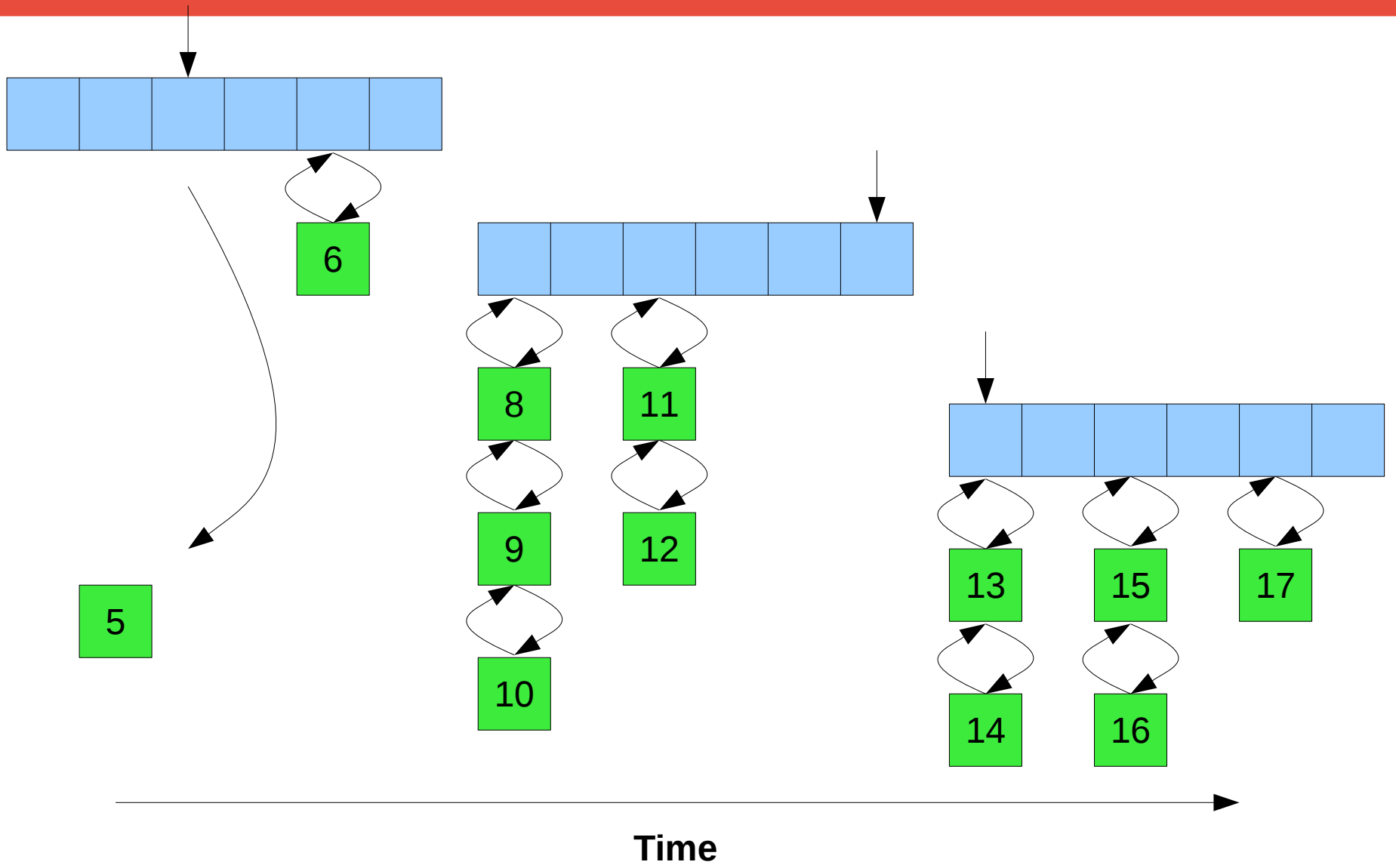
The Timer Wheel



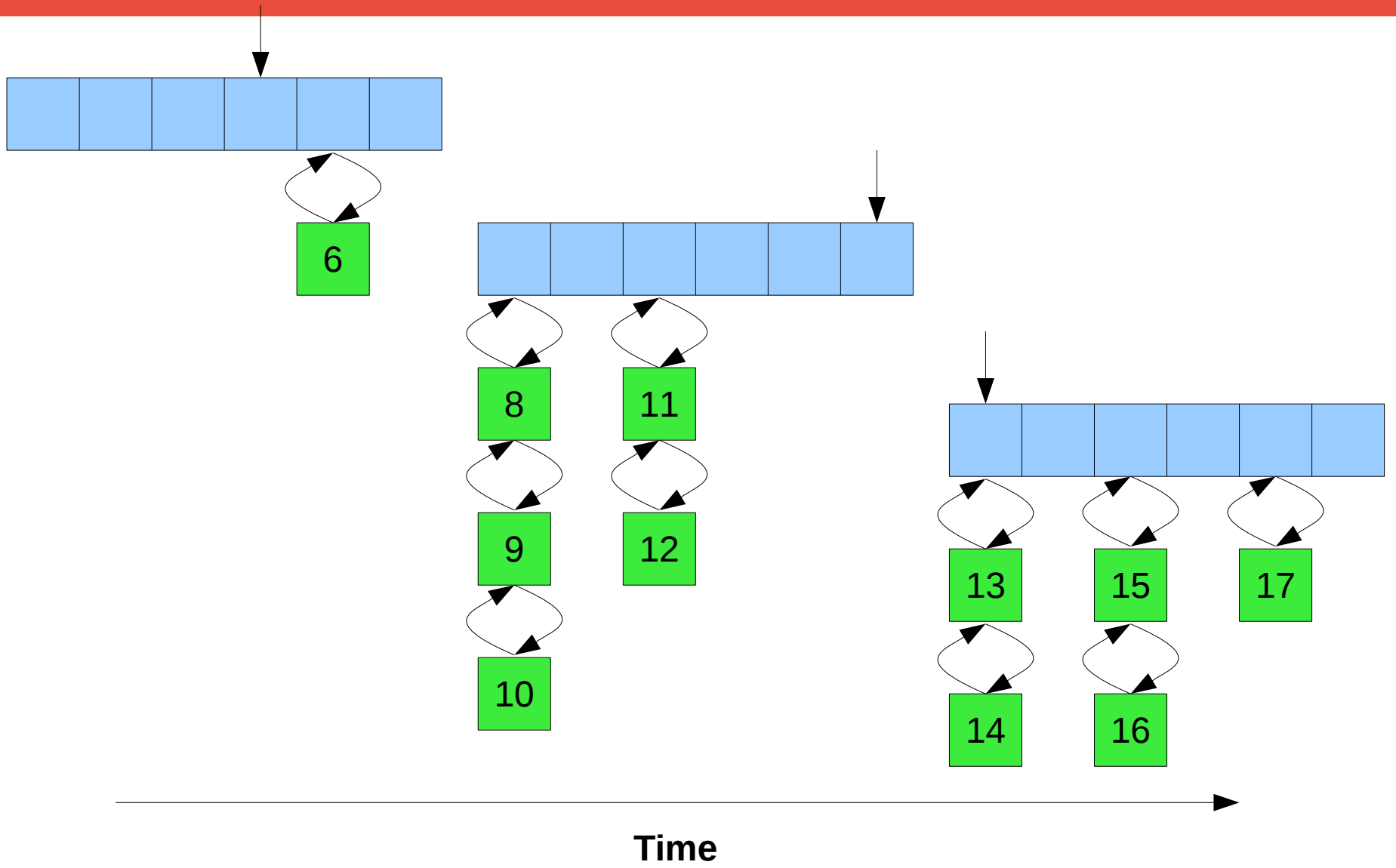
The Timer Wheel



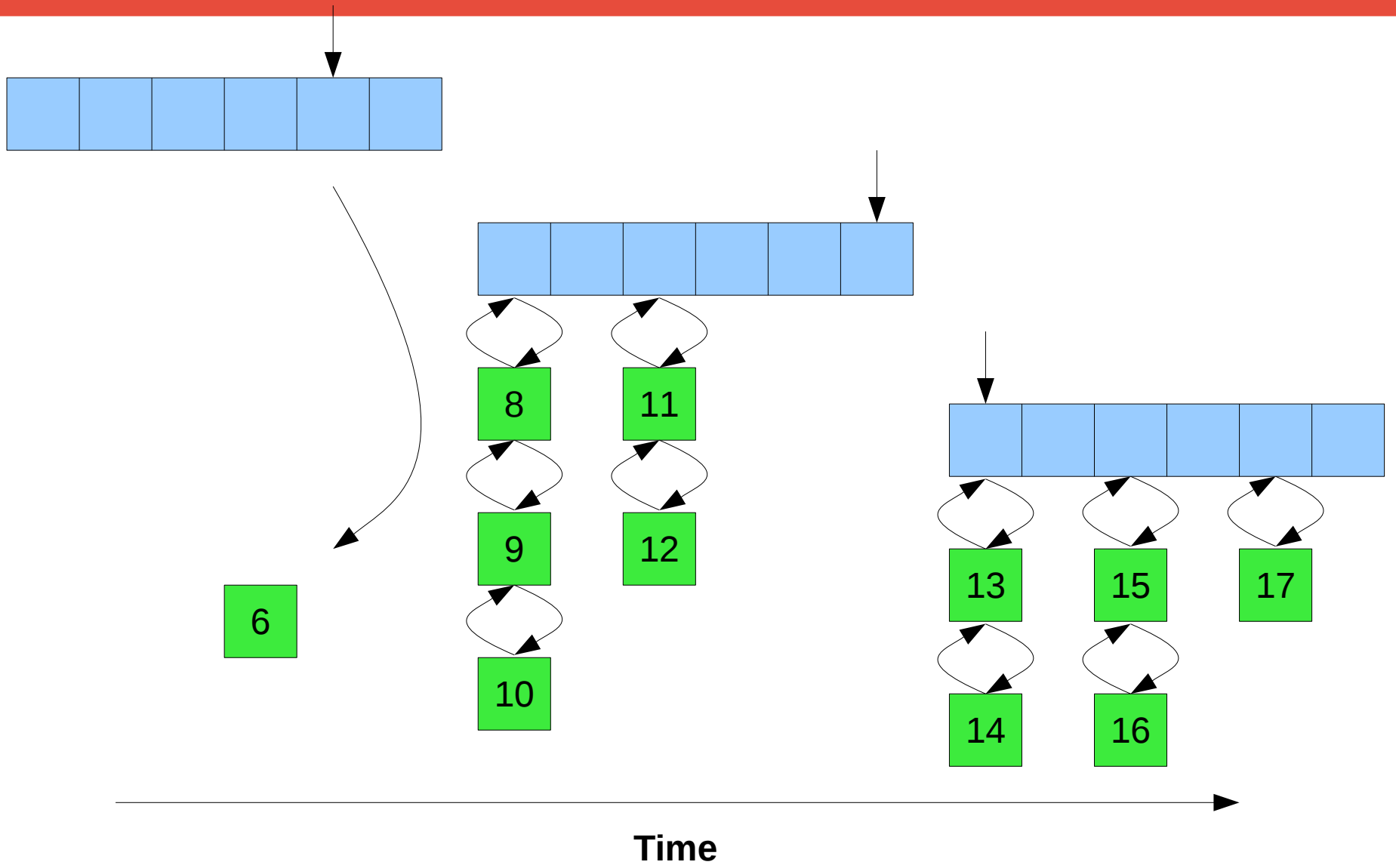
The Timer Wheel



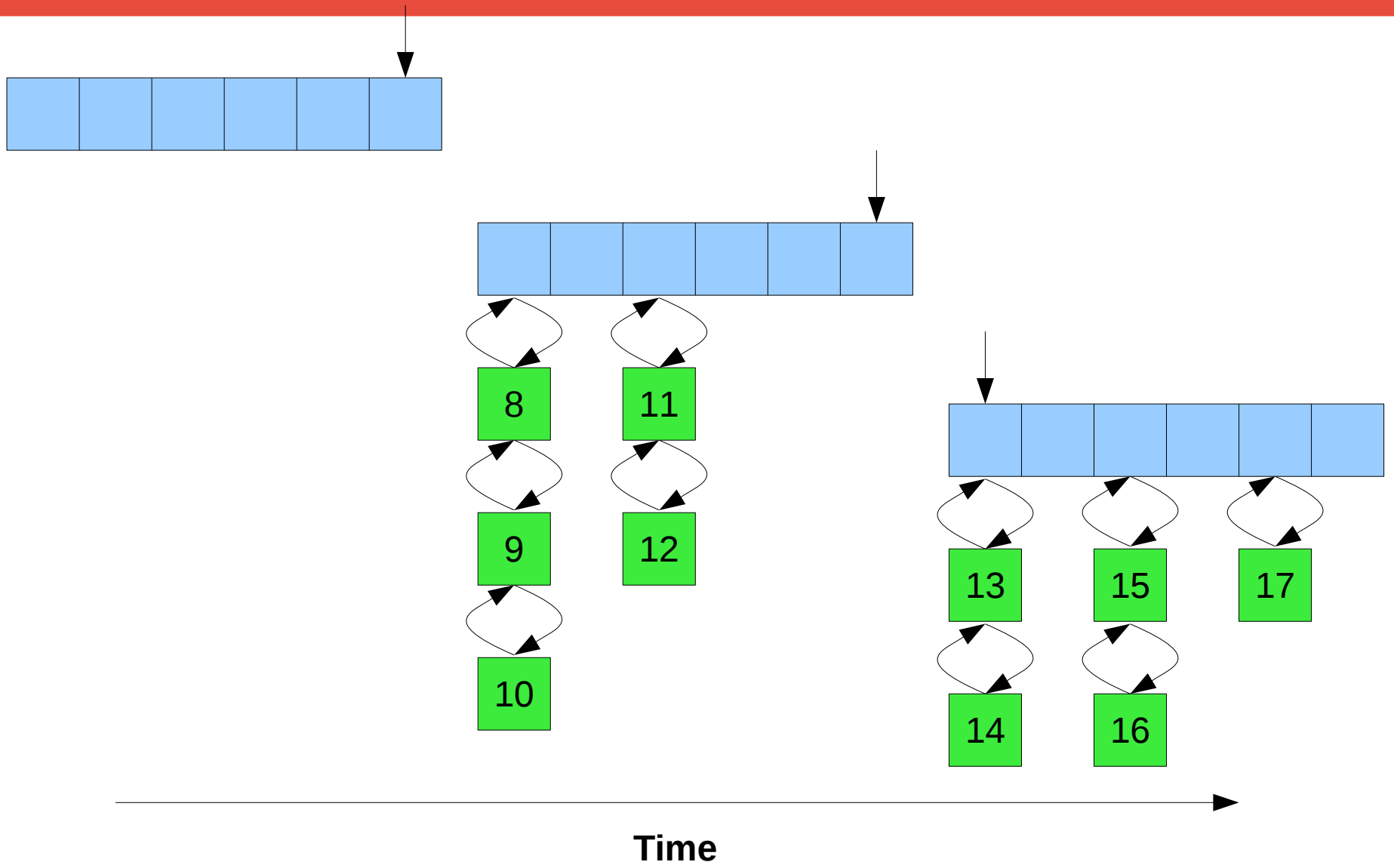
The Timer Wheel



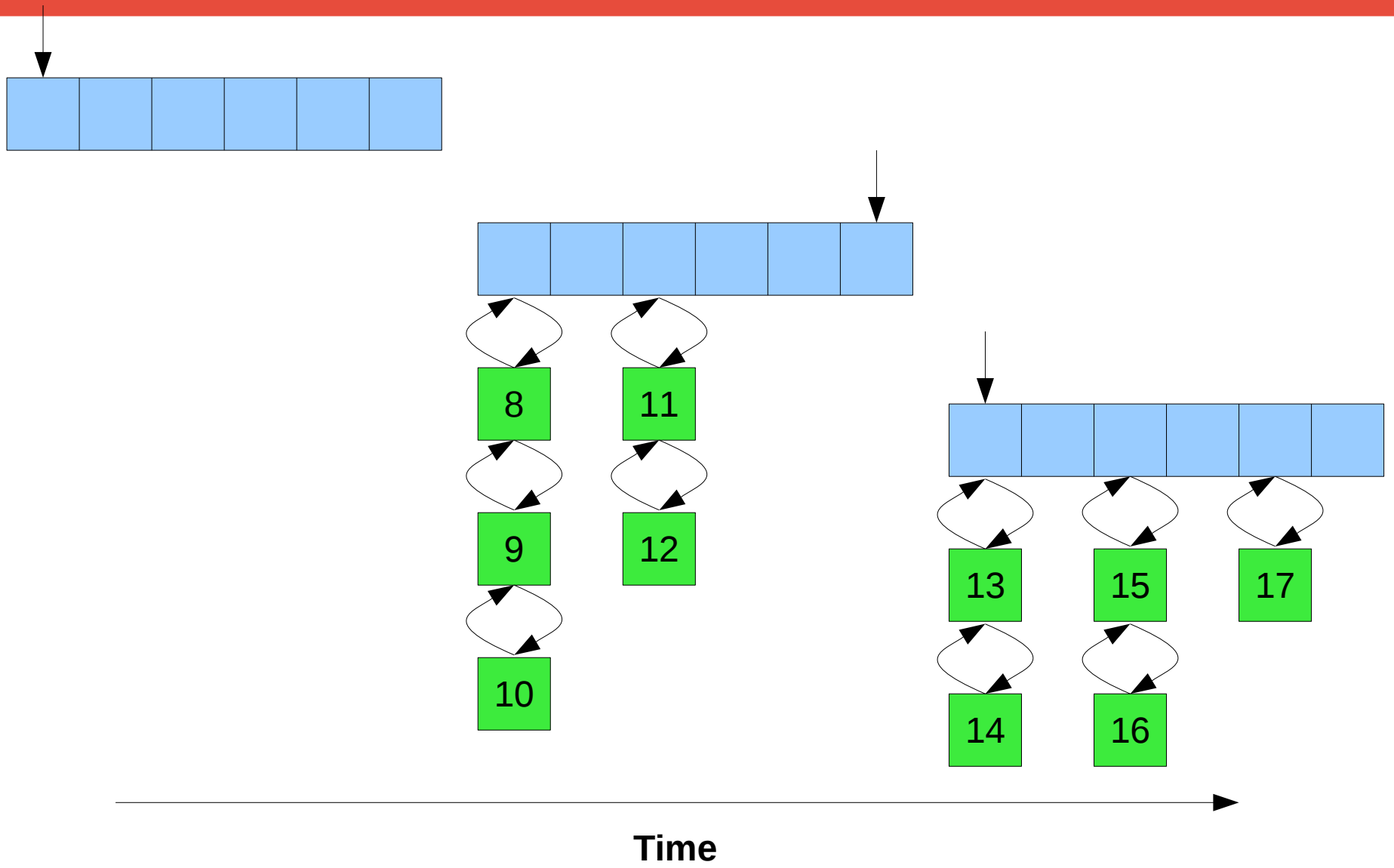
The Timer Wheel



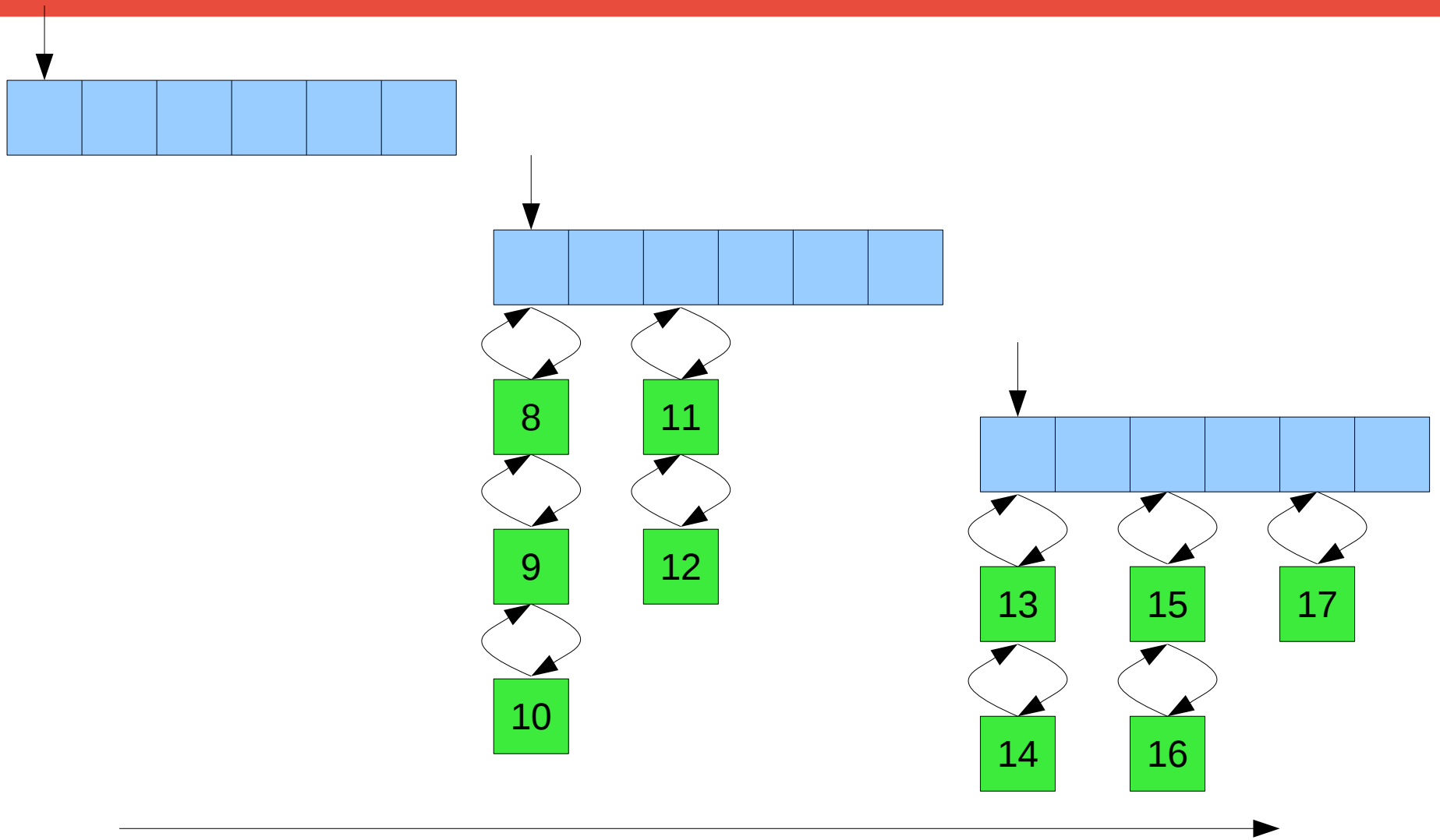
The Timer Wheel



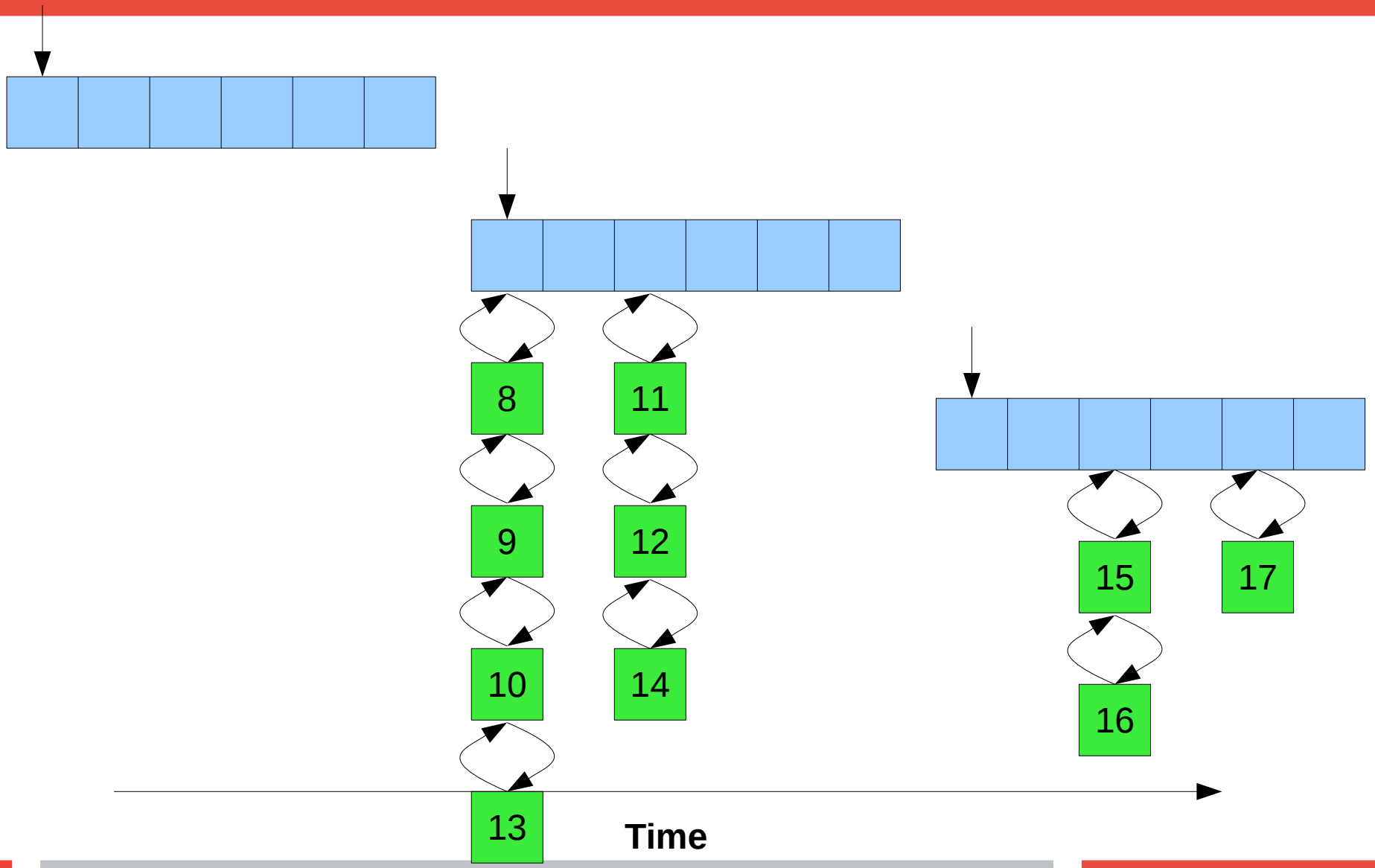
The Timer Wheel



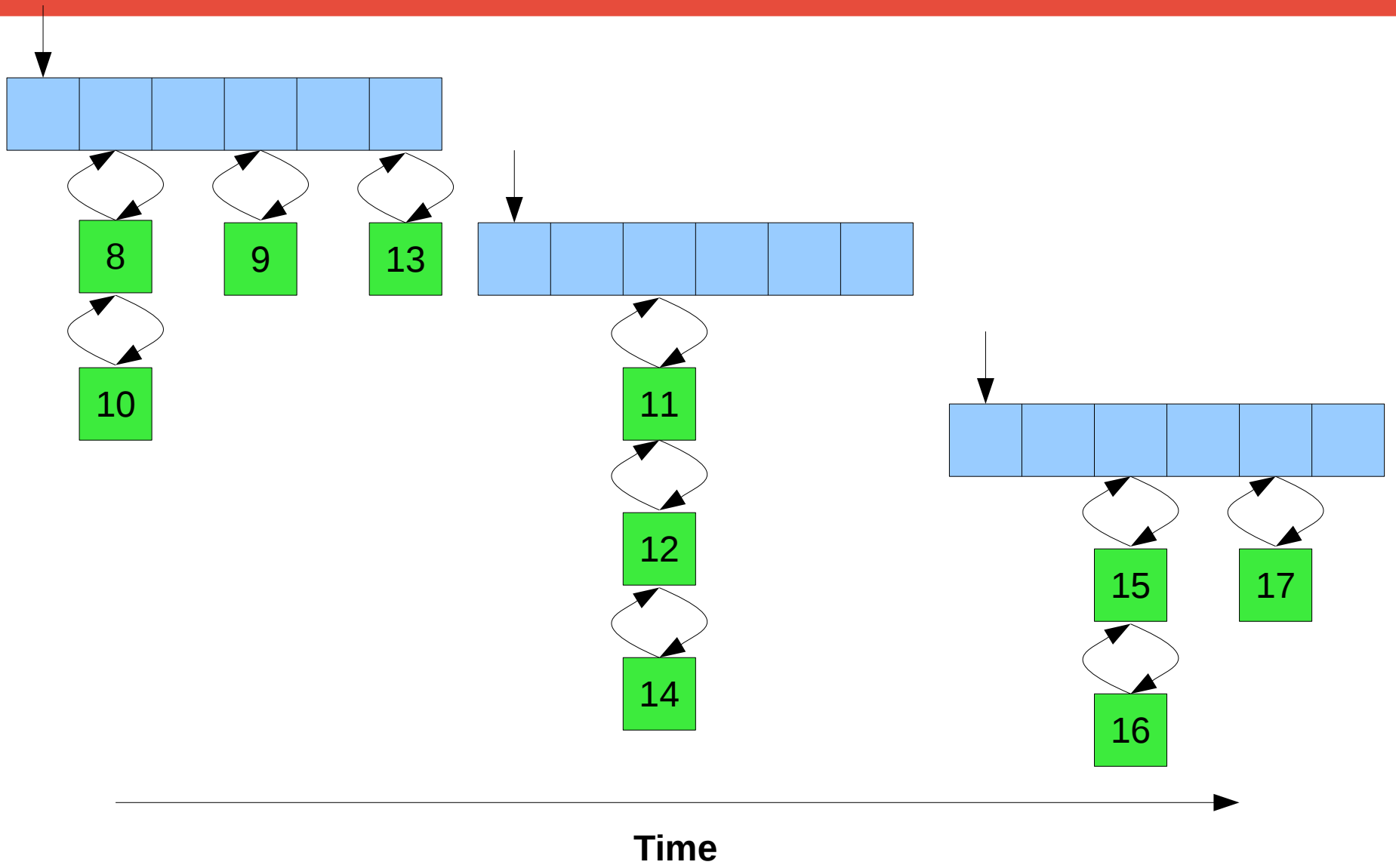
The Timer Wheel



The Timer Wheel



The Timer Wheel



The New Timer Wheel

Remove the cascade effect

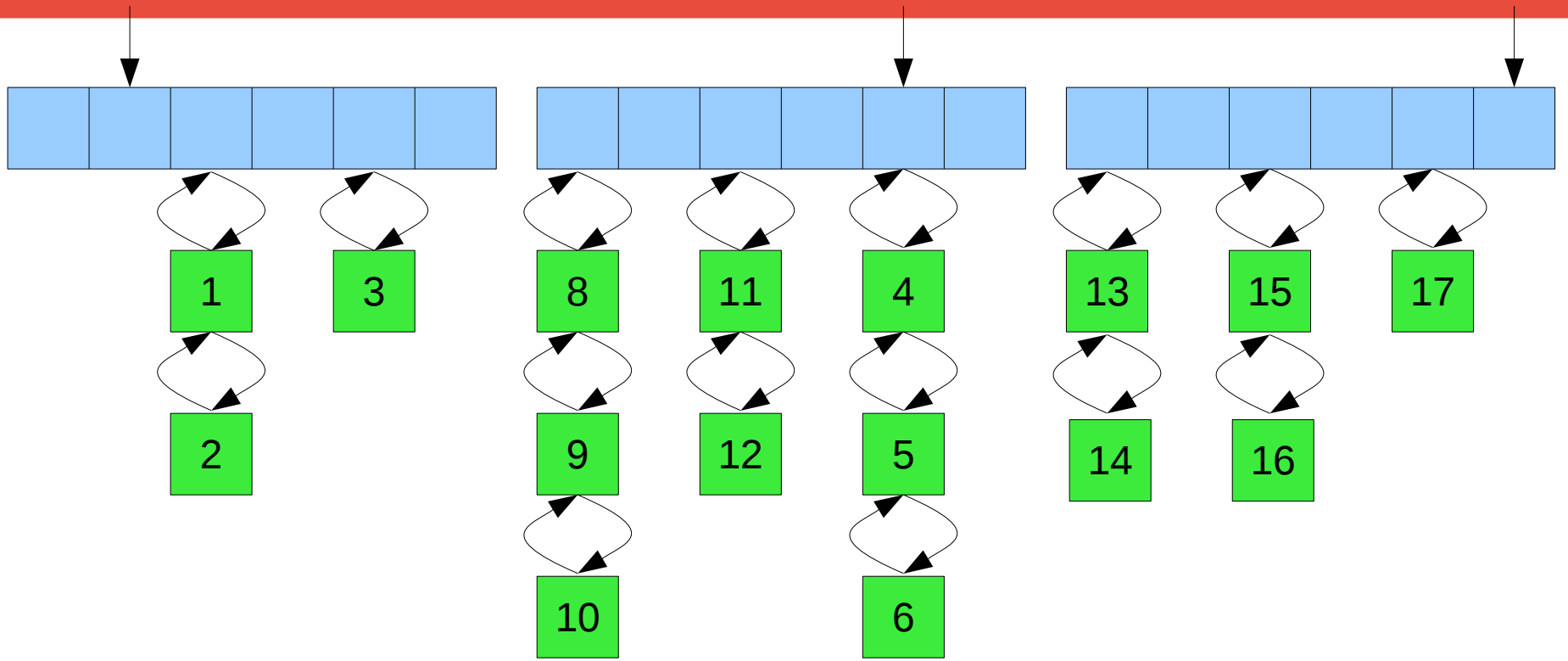
Farther out the timer, the less granularity

More levels (8 instead of 5)

Each level contains 32 entries

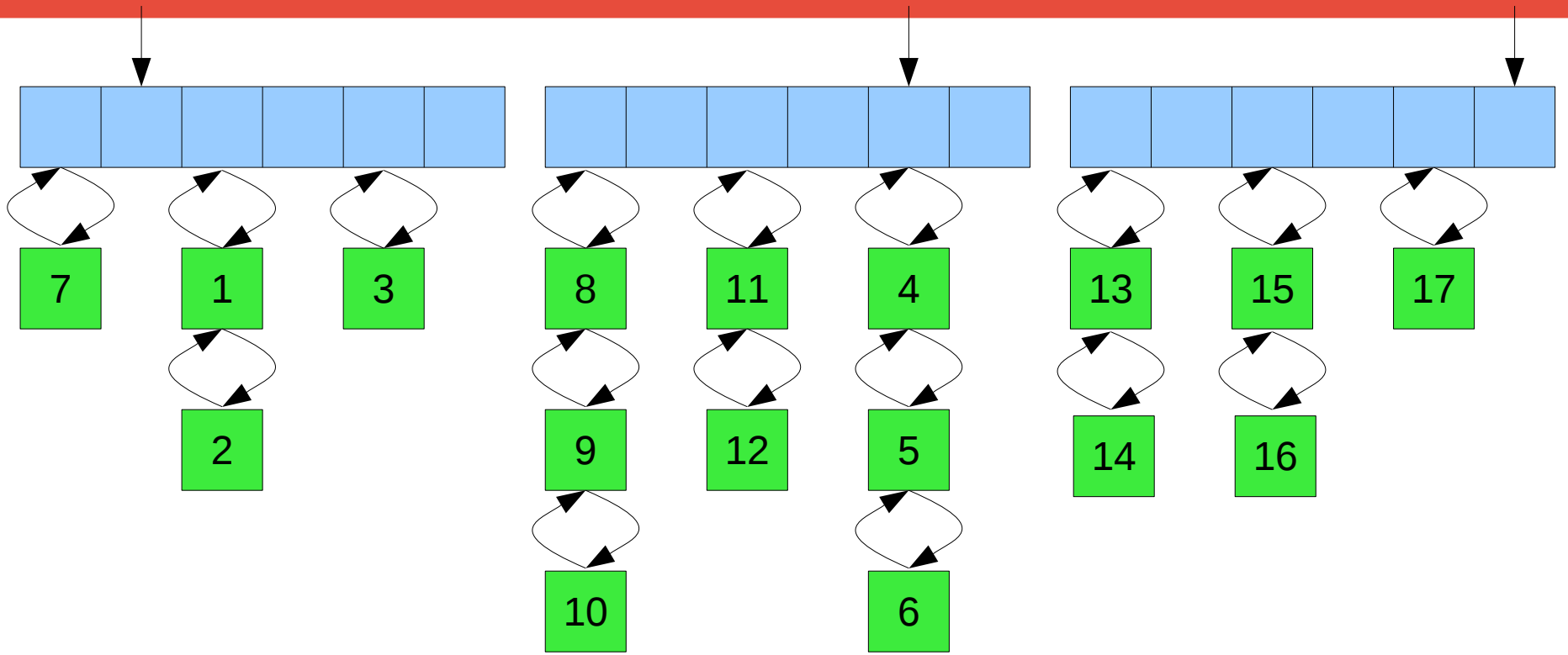
```
HZ 1000
Level Offset  Granularity              Range
 0     0            1 ms                0 ms -         31 ms
 1    32            8 ms               32 ms -        255 ms
 2    64           64 ms              256 ms -       2047 ms (256ms - ~2s)
 3    96          512 ms             2048 ms -      16383 ms (~2s - ~16s)
 4   128         4096 ms (~4s)      16384 ms -     131071 ms (~16s - ~2m)
 5   160        32768 ms (~32s)    131072 ms -    1048575 ms (~2m - ~17m
 6   192       262144 ms (~4m)    1048576 ms -    8388607 ms (~17m - ~2h)
 7   224      2097152 ms (~34m)   8388608 ms -   67108863 ms (~2h - ~18h)
```

The New Timer Wheel



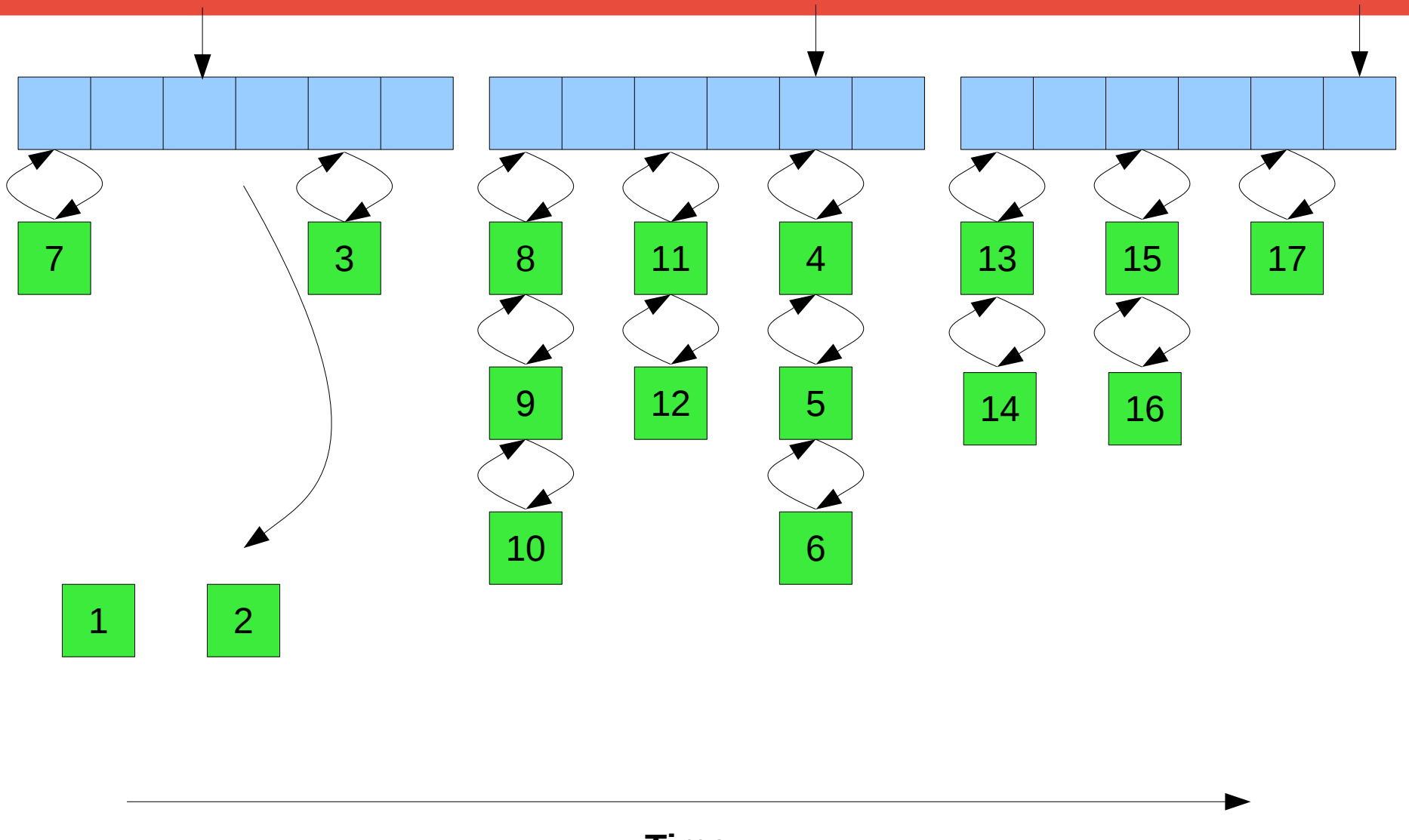
Time

The New Timer Wheel

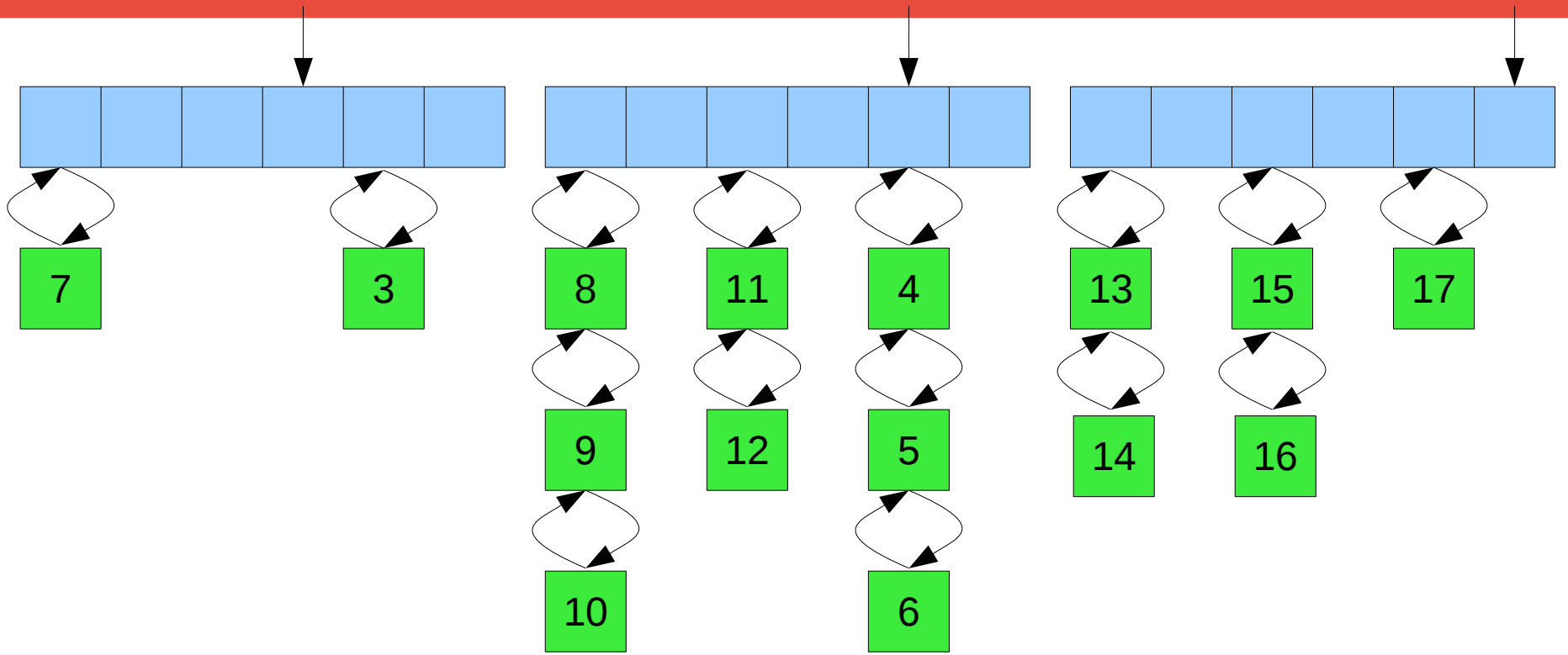


Time

The New Timer Wheel

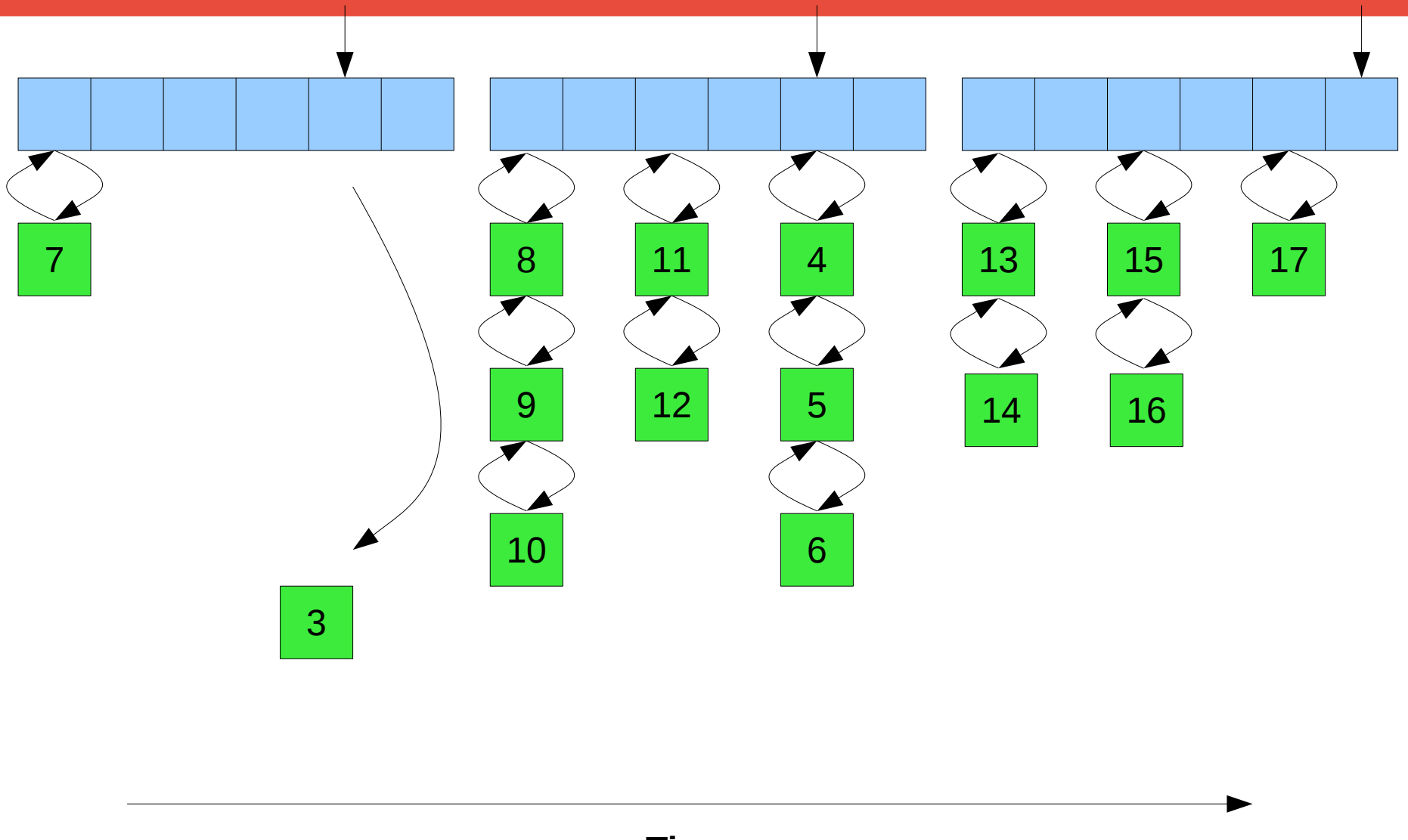


The New Timer Wheel

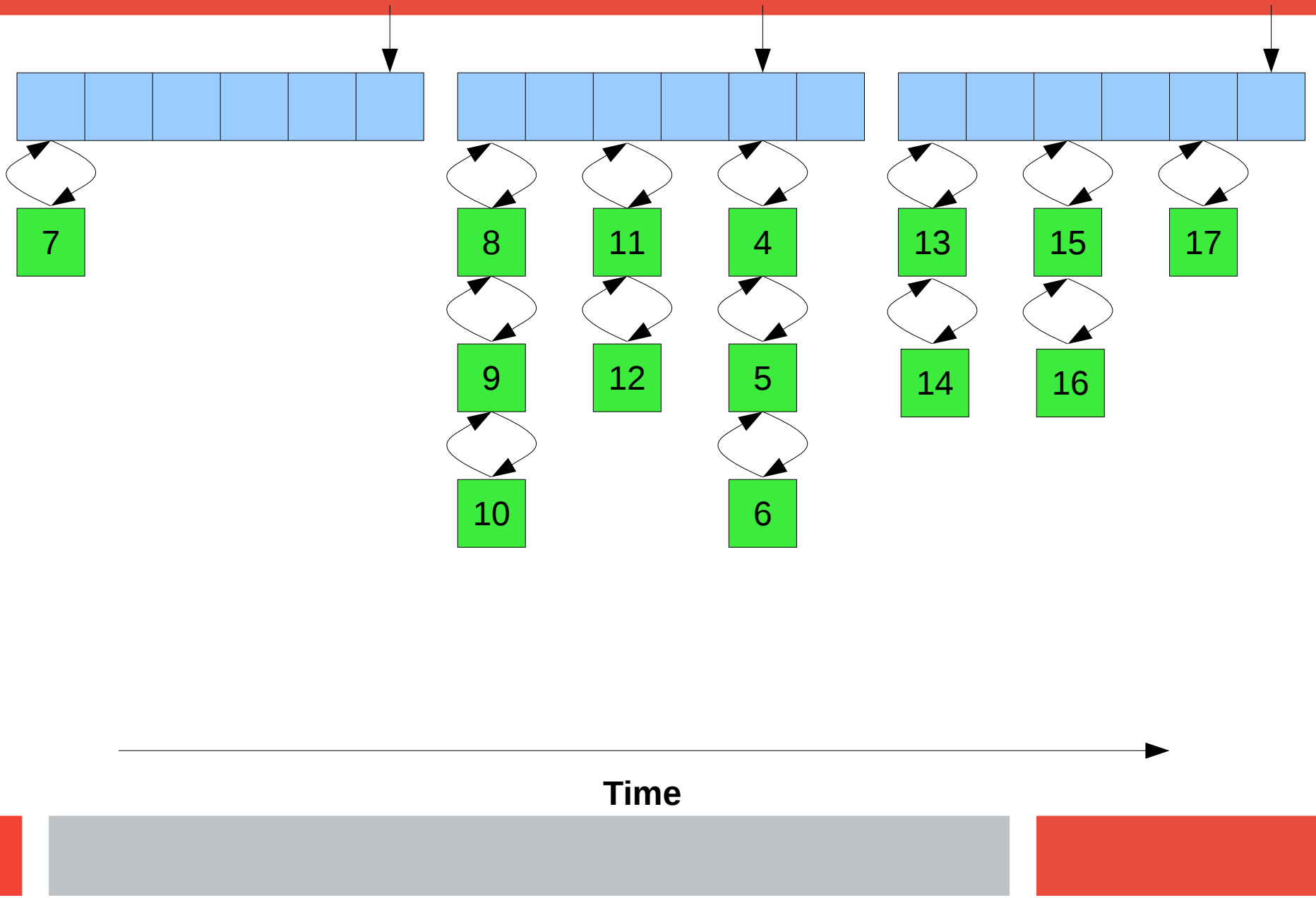


Time

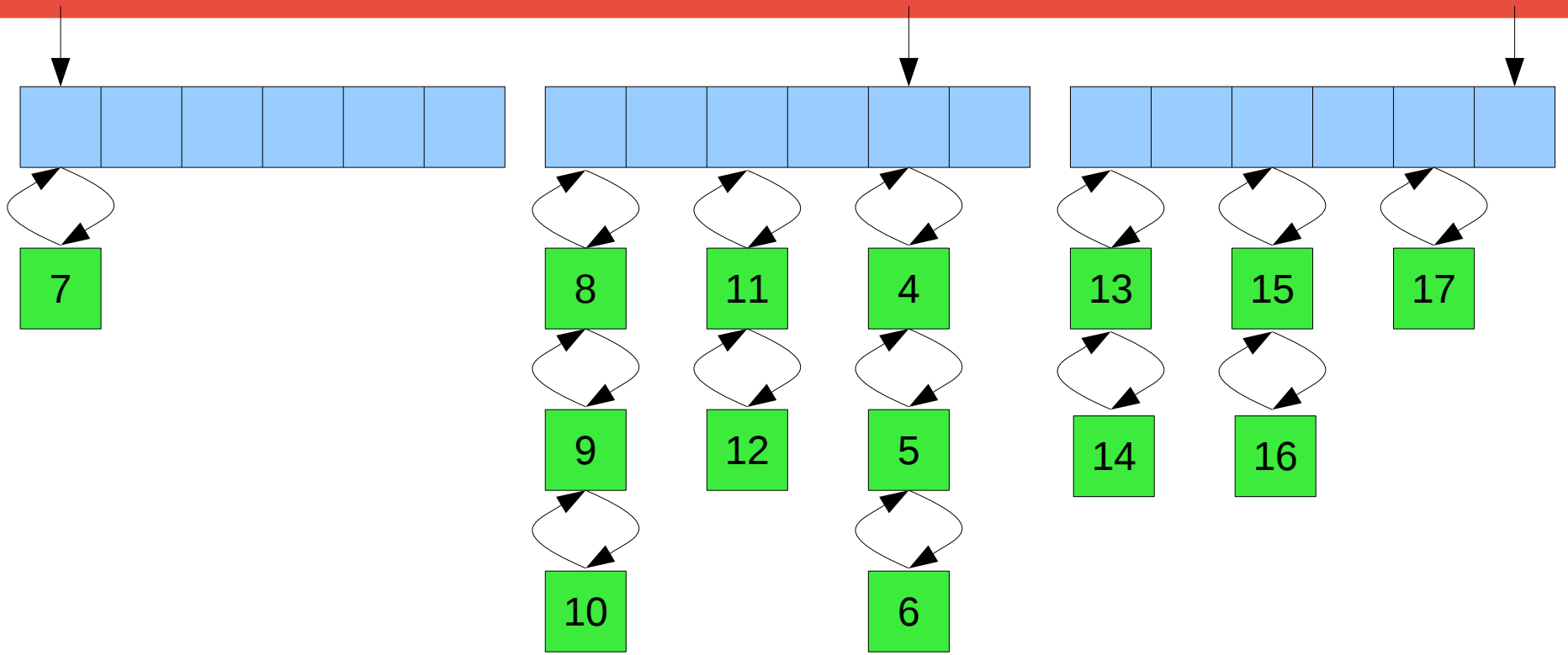
The New Timer Wheel



The New Timer Wheel

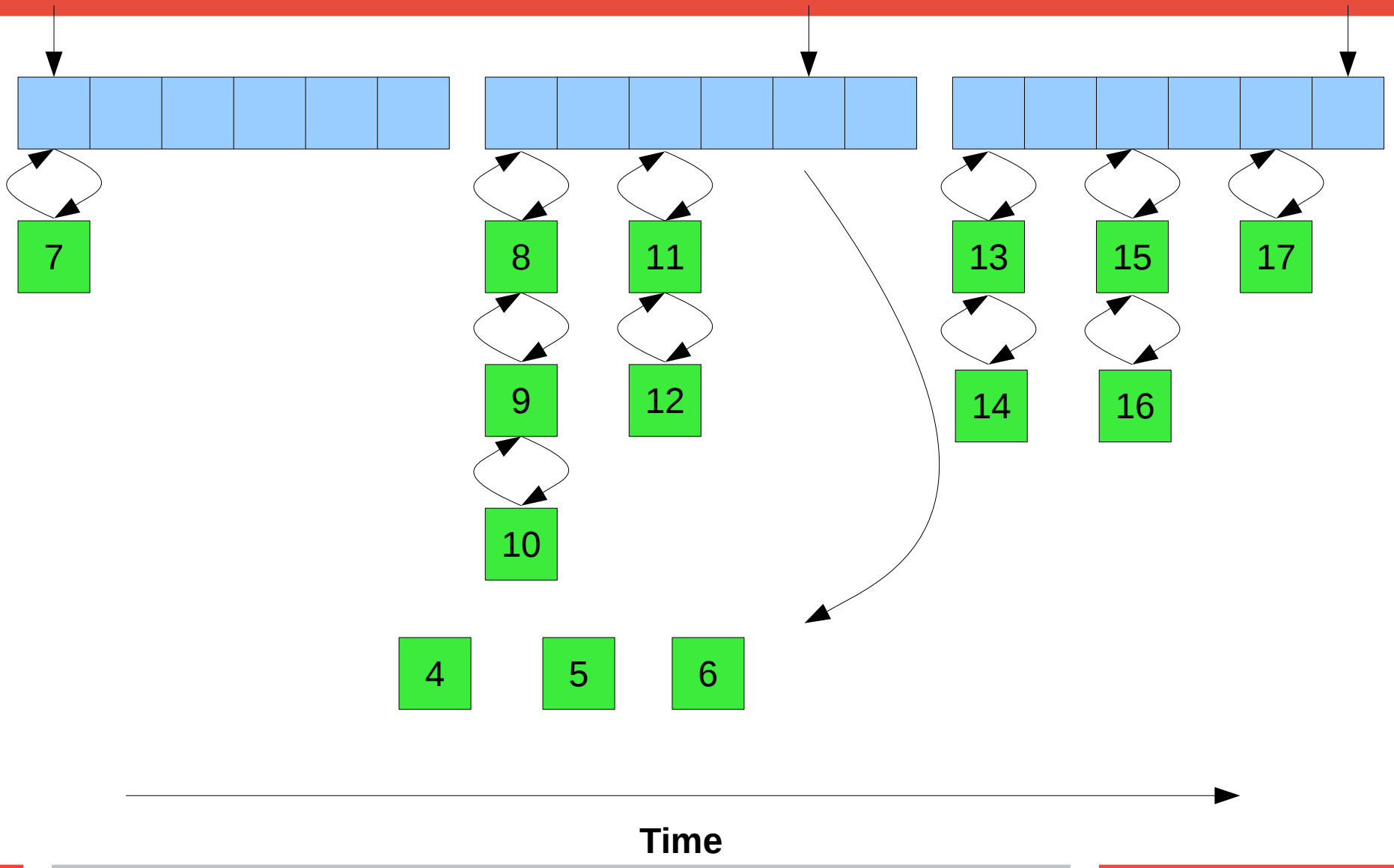


The New Timer Wheel

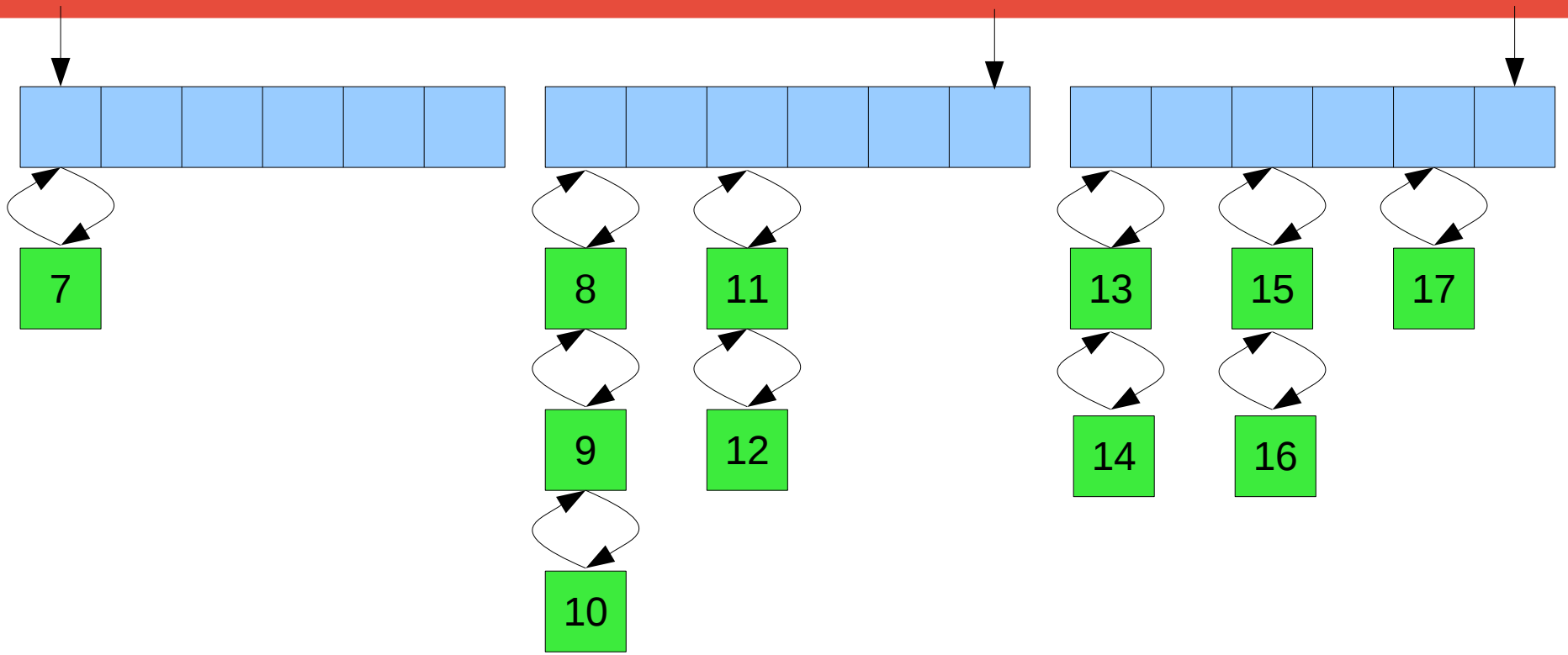


Time

The New Timer Wheel

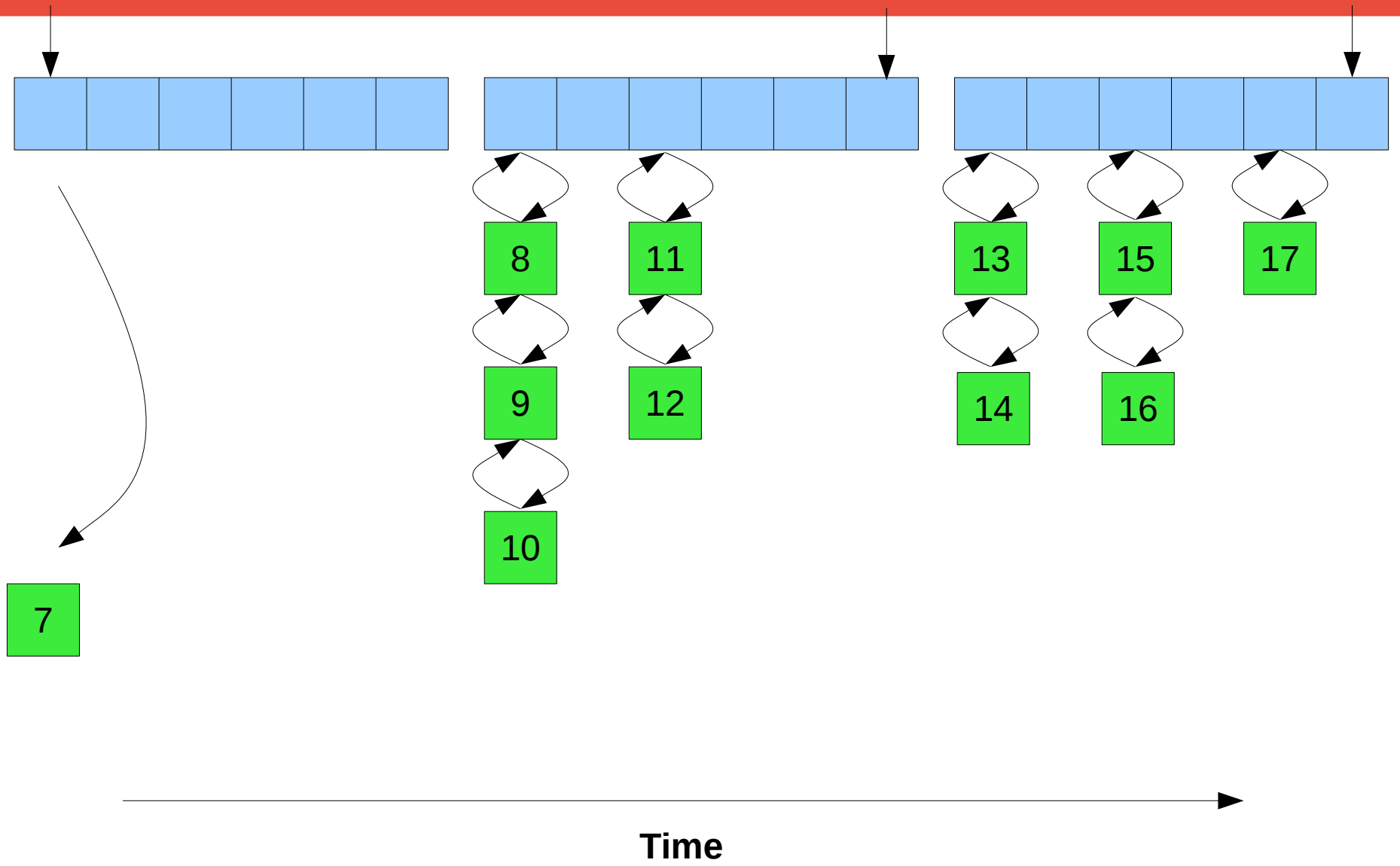


The New Timer Wheel

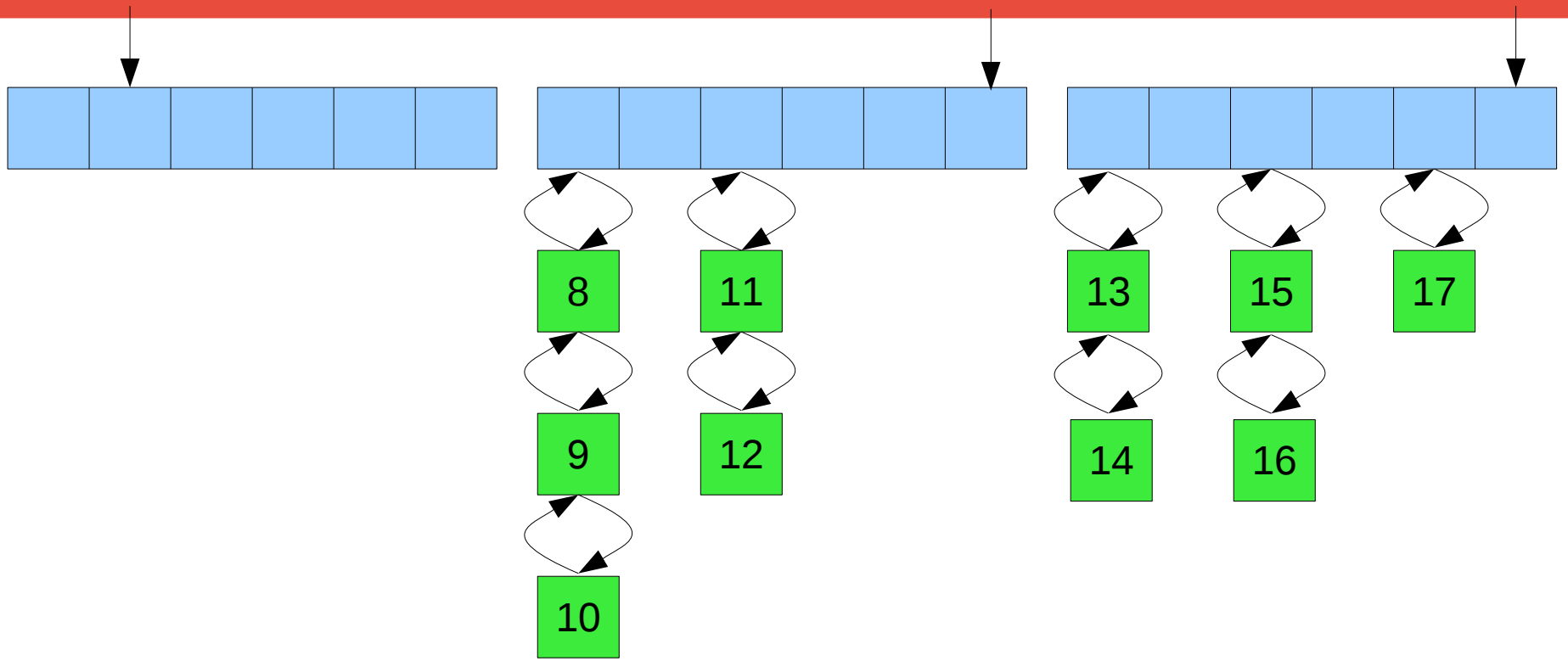


Time

The New Timer Wheel

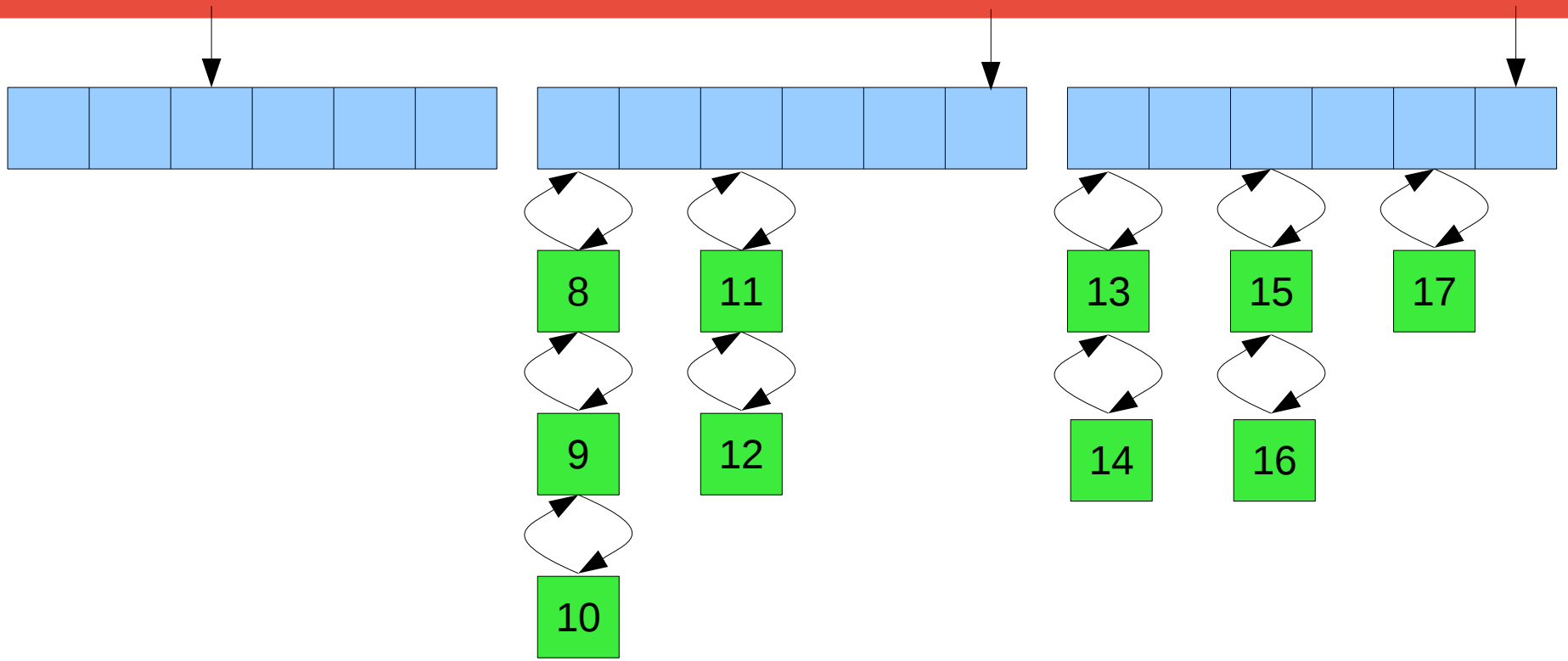


The New Timer Wheel



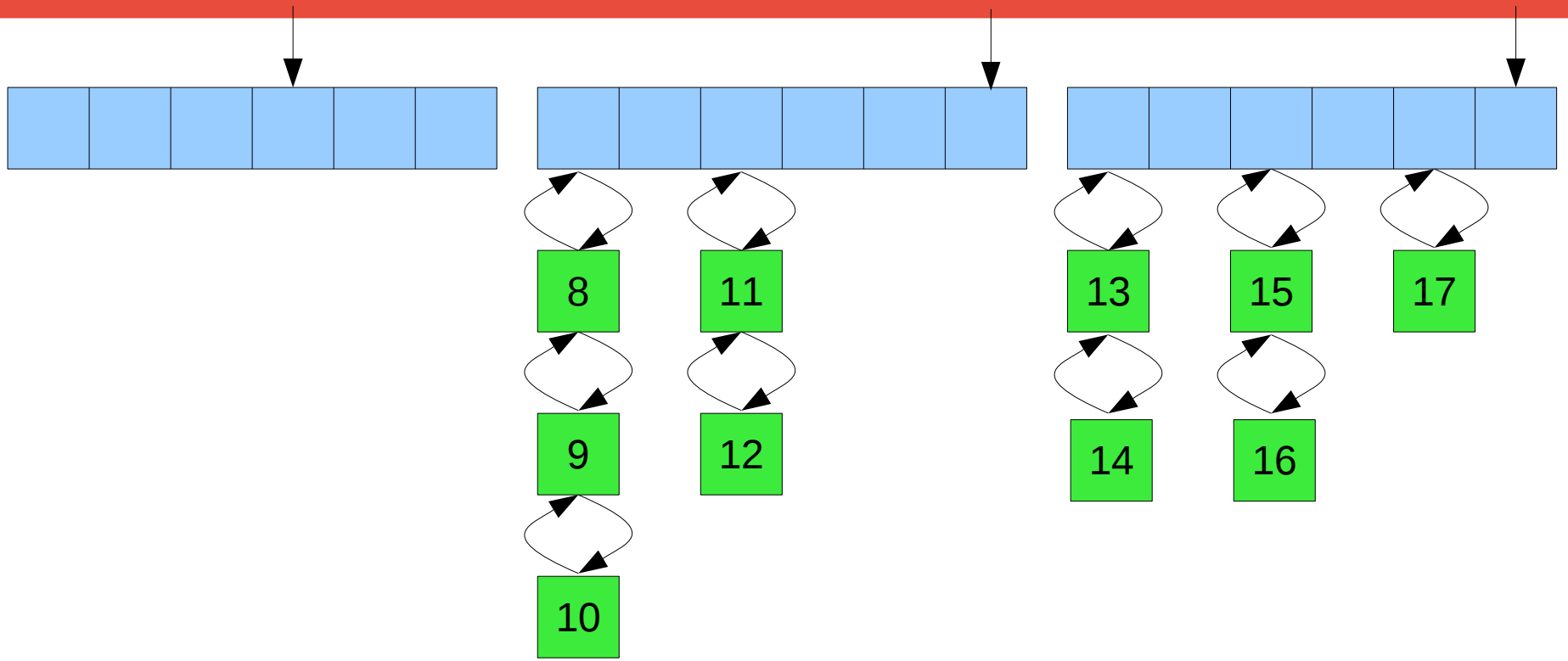
Time

The New Timer Wheel



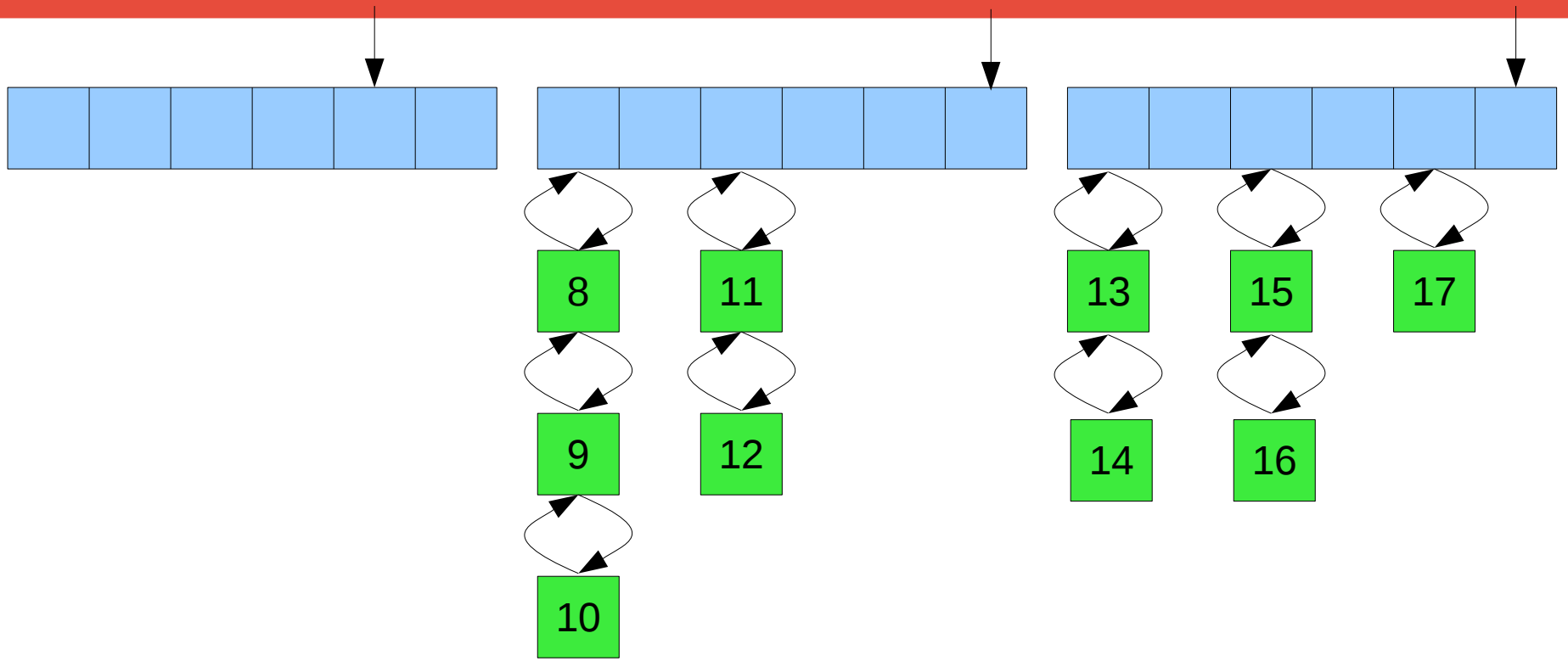
Time

The New Timer Wheel

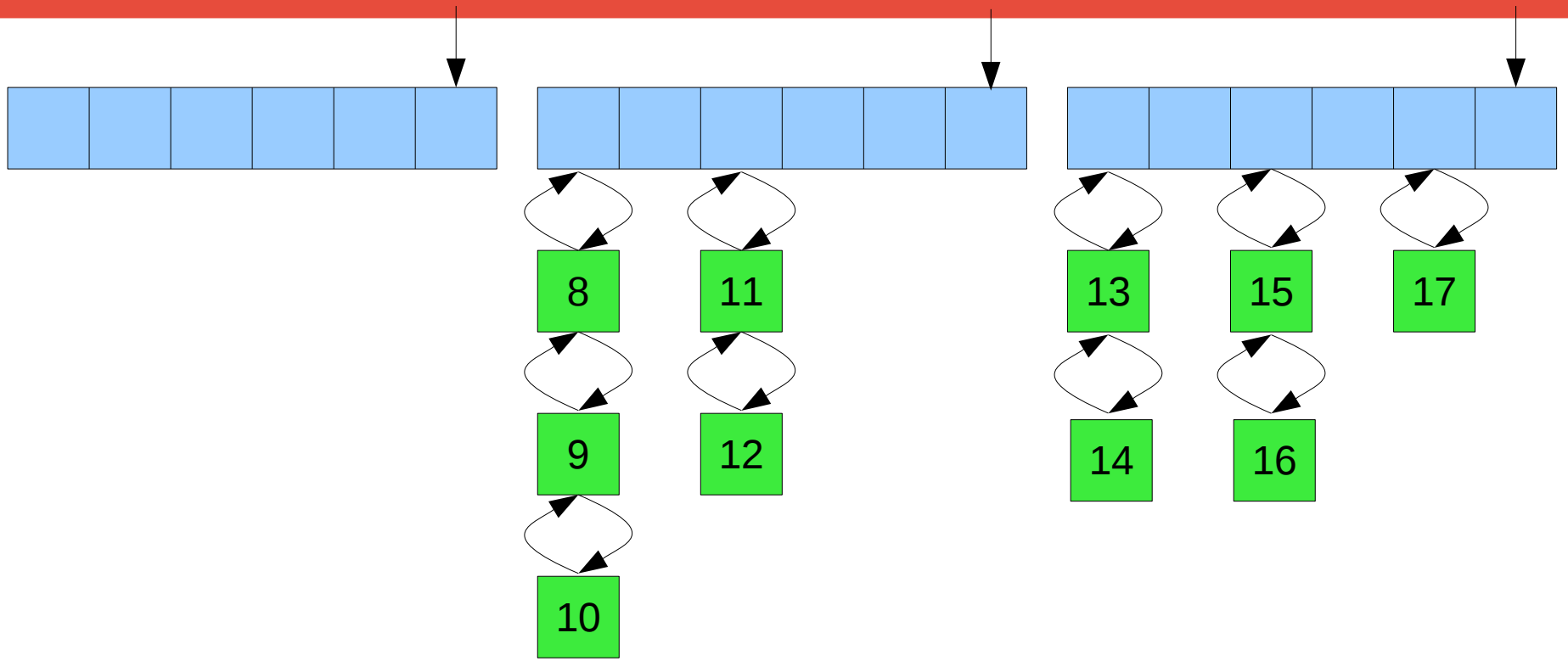


Time

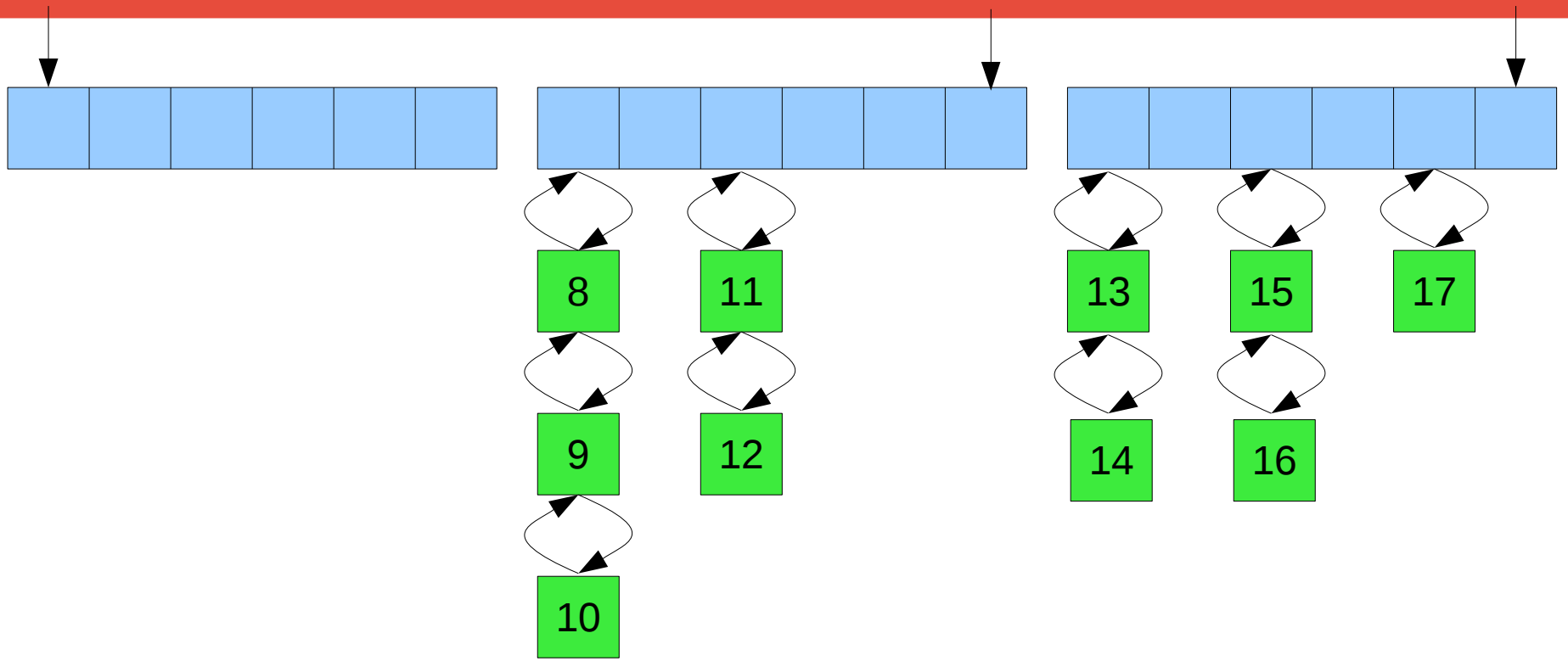
The New Timer Wheel



The New Timer Wheel

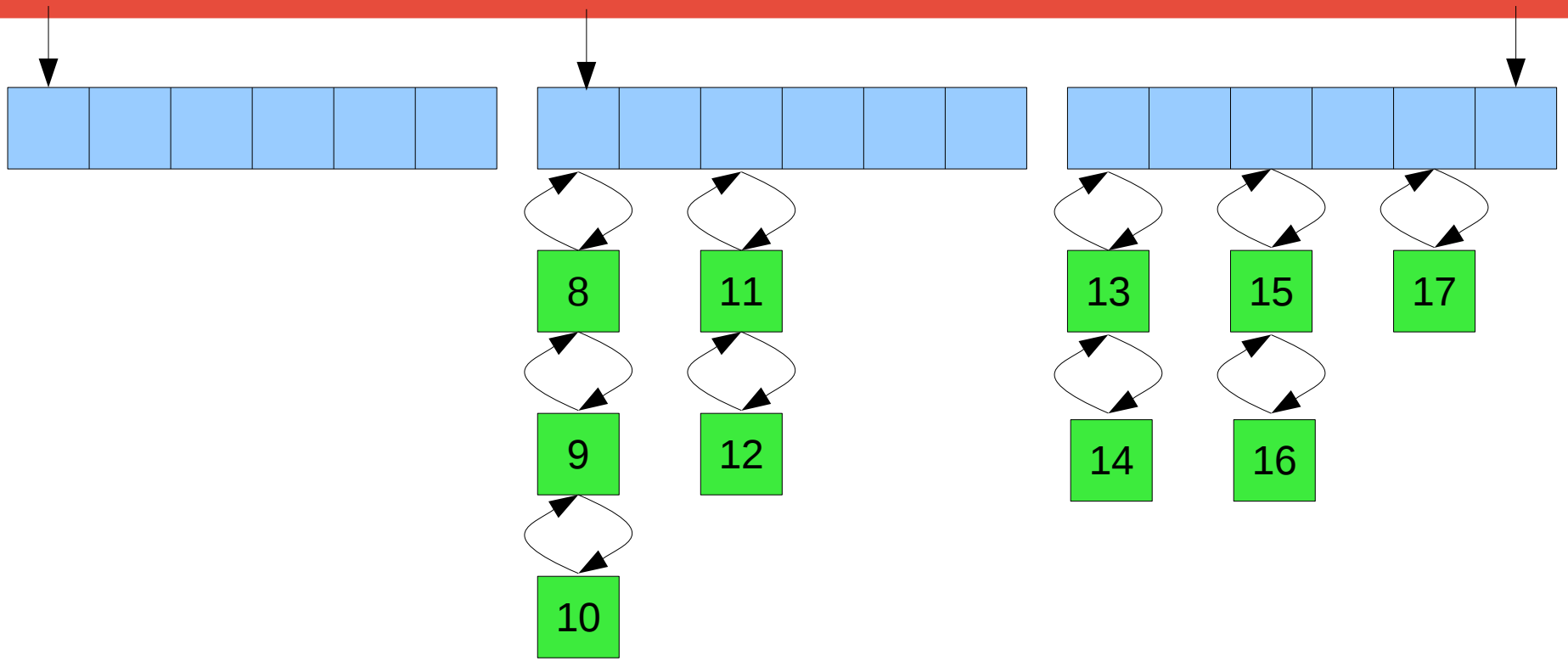


The New Timer Wheel



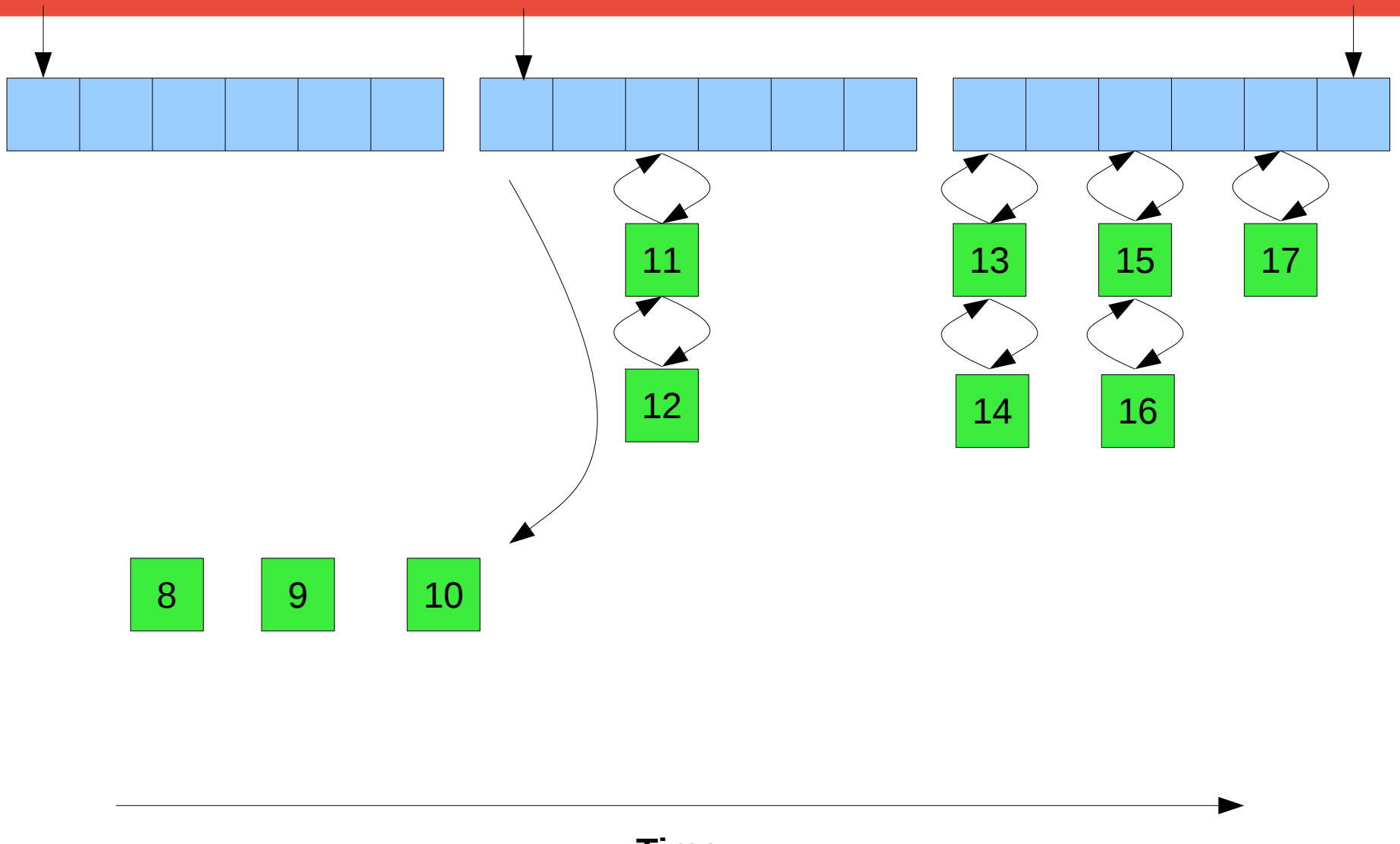
Time

The New Timer Wheel

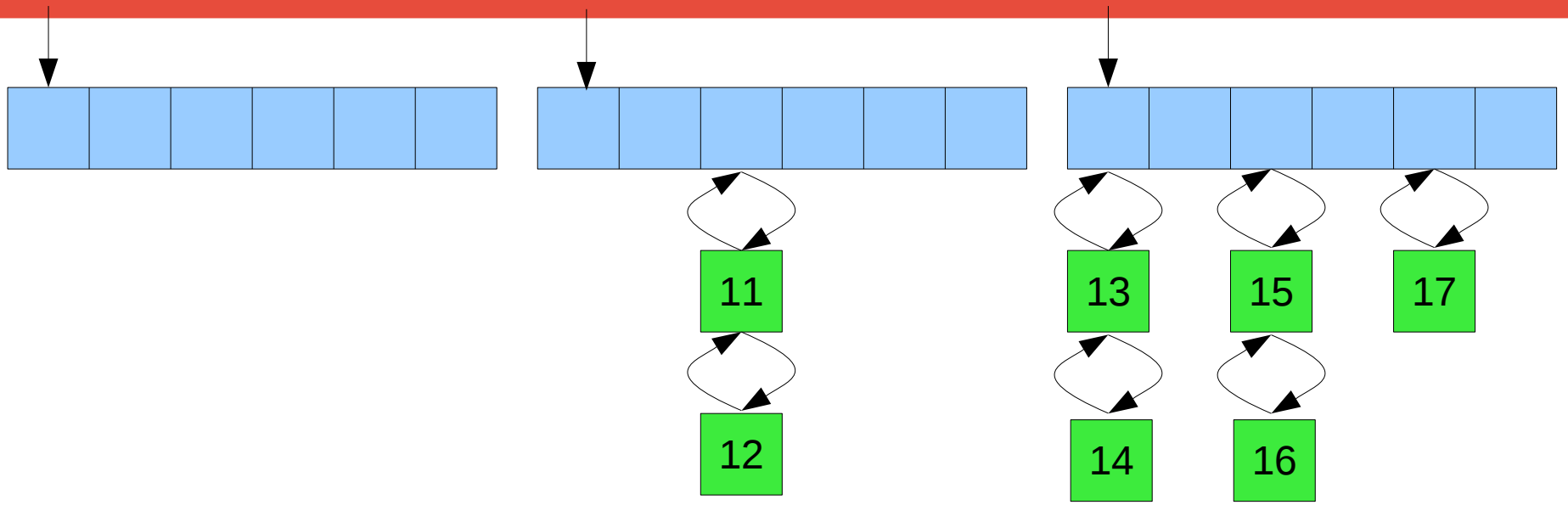


Time

The New Timer Wheel

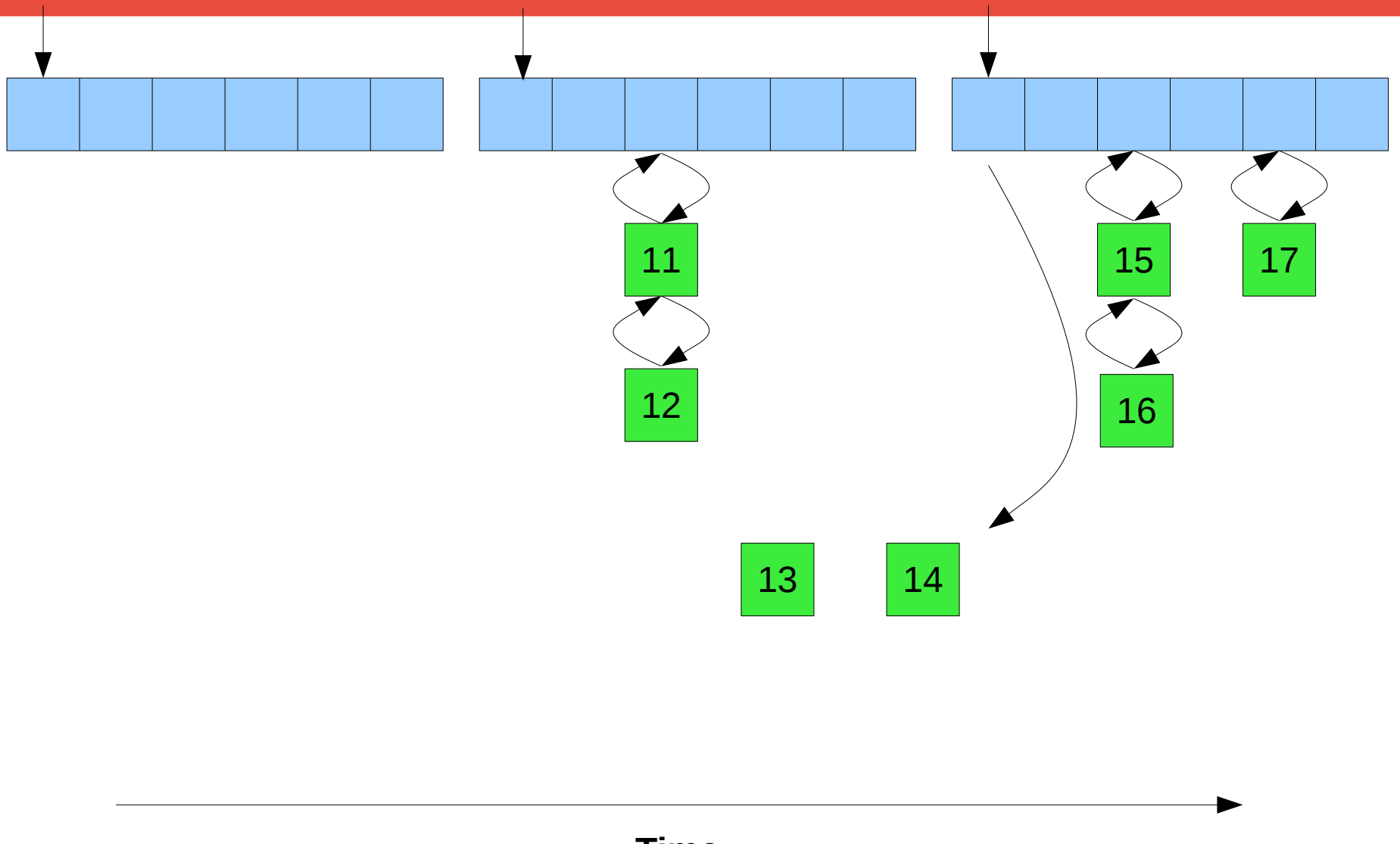


The New Timer Wheel



Time

The New Timer Wheel



Softirqs

- Runs after a hard interrupt
- In interrupt context, but with interrupts enabled
- Runs on the CPU they were enabled on
- local_bh_disable() stops them (and disables preemption)

Softirqs

- **In PREEMPT RT, all (most really) interrupts are threads**
- **Making softirqs theads has a problem**
  - **What prio should they be?**
  - **Low and high prio tasks may use them**
- **Running after interrupts can have issues as interrupts and softirqs have different CPU affinities**

Softirqs

- Run by whoever called for them at their prio
- If local_bh_disable() set, run raised softirqs in local_bh_enable()
- Run directly when softirq is raised if bh is not disabled.
- Works for tasks as well as interrupt threads
- Hard interrupts must not call softirqs

CPU Hot Plug

- **Now must handle tasks with migration disabled**
- **Notifiers!**
  - **Called at each stage of bringing CPU up or down**
  - **May have issues with sleeping spin locks**
- **Bringing a CPU online can be just as bad**
  - **There may be cases with notifiers called with interrupts disabled**

Questions?