Multifrontal Sparse QR Factorization on GPU

Nuri Yeralan & Tim Davis
University of Florida

February 26, 2013
SIAM CSE 2013

Outline

- ▶ **Introduction**
  - Multifrontal sparse QR
  - GPU considerations
- **GPU-based multifrontal sparse QR**
  - Memory traffic
  - Uberkernel
  - Front scheduling
  - Pipelined dense QR factorization

Multifrontal Sparse QR

- **Why QR?**
  - Wide Applicability
  - Numerically Stable
  - High ratio of flops to memory accesses
- **Why Multifrontal?**
  - CAQR (2008, Demmel et al)
  - No data dependence during child assembly
- **Why GPU?**
  - Teraflops of peak theoretical performance
  - Low amortized cost per flop (watts, dollars)

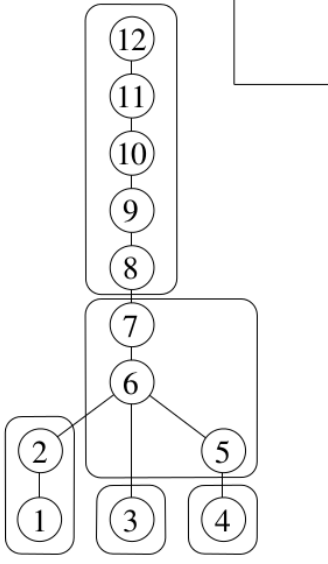
Multifrontal Sparse QR

the matrix A

	1	2	3	4	5	6	7	8	9	10	11	12
1	x	x				.		x			.	
2	x	.				x		.			x	
3	x	x				.		x			x	
4		x				x		.			x	
5		x				x		.			x	
6		x				.		x			.	
7			x			x	x	.				x
8			x			.	.	.				x
9			x			x	.	x				.
10			x			.	x	x				.
11			x			.	x	.				.
12				x	.		.		.		x	
13				x	x		x		x		.	
14				x	x		.		x		x	
15				x	x		.		.		x	
16					x	.	x	x	.		x	x
17					x	x	.	.	.		.	x
18					x	x	.	x	x		x	.
19					.	x	x	.	x		.	x
20					.	x	.	x	.		.	x
21					.	.	x	.	.		x	x
22					.	.	x	x	x		x	x
23								x	x	x	x	x

the QR factor R

	1	2	3	4	5	6	7	8	9	10	11	12
1	r	r				r		r			r	
2		r				r		r			r	
3			r			r	r	r				r
4				r	r		r		r		r	
5					r	r	r	r	r		r	r
6						r	r	r	r		r	r
7							r	r	r		r	r
8								r	r	r	r	r
9									r	r	r	r
10										r	r	r
11											r	r
12												r



the column elimination tree

Multifrontal Sparse QR

child front 1

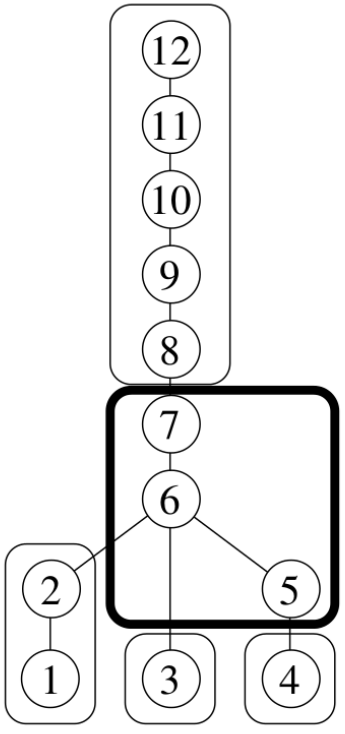
	6	8	11
3	c_1	c_1	c_1
4		c_1	c_1
5			c_1

child 3

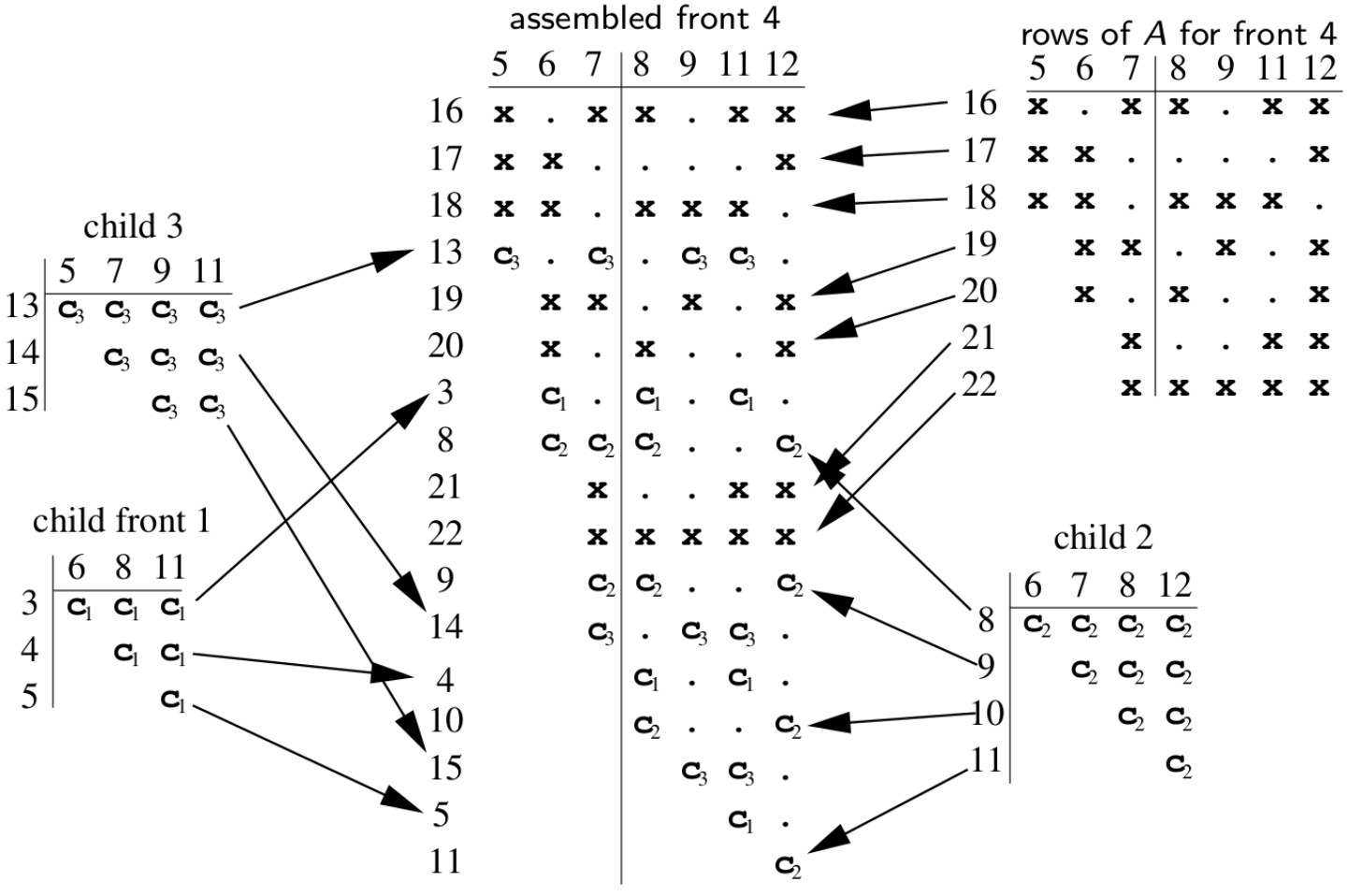
	5	7	9	11
13	c_3	c_3	c_3	c_3
14		c_3	c_3	c_3
15			c_3	c_3

child 2

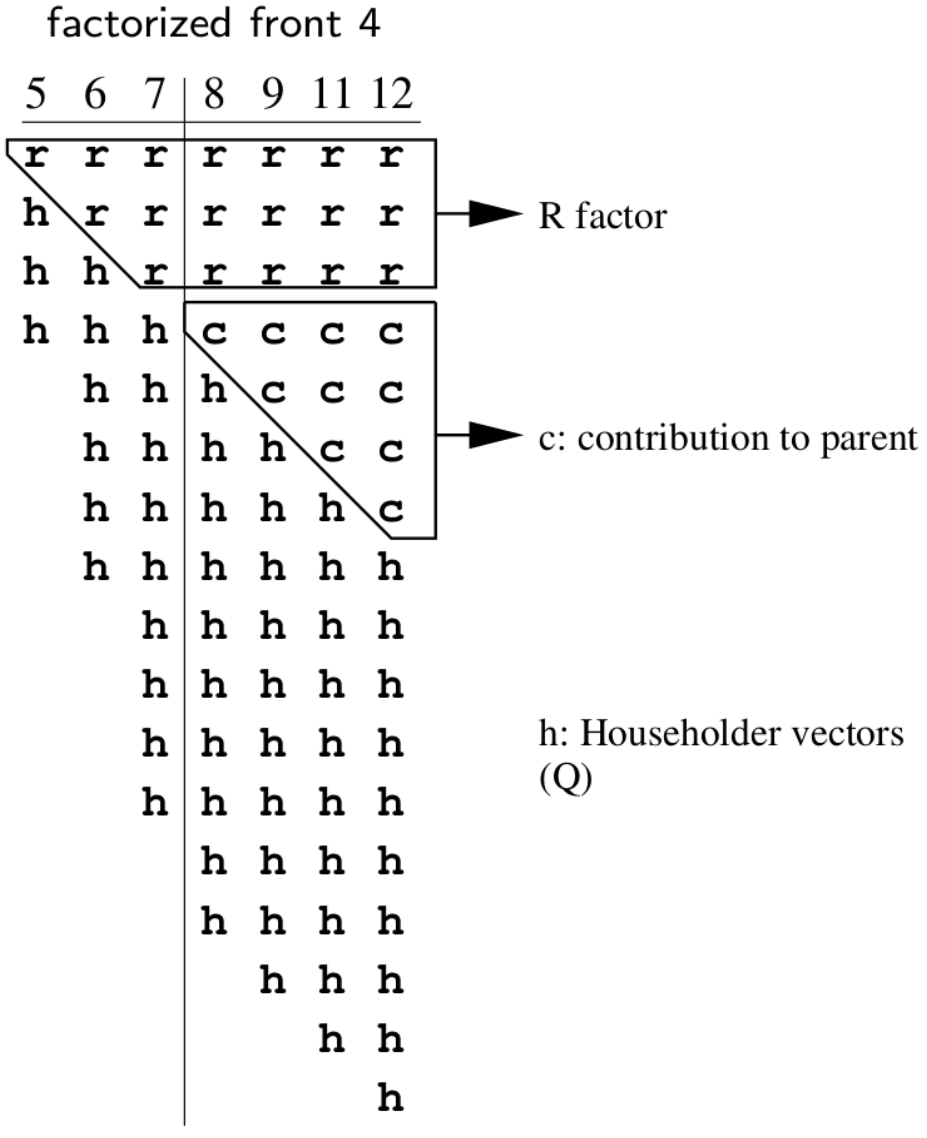
	6	7	8	12
8	c_2	c_2	c_2	c_2
9		c_2	c_2	c_2
10			c_2	c_2
11				c_2



Multifrontal Sparse QR



Multifrontal Sparse QR



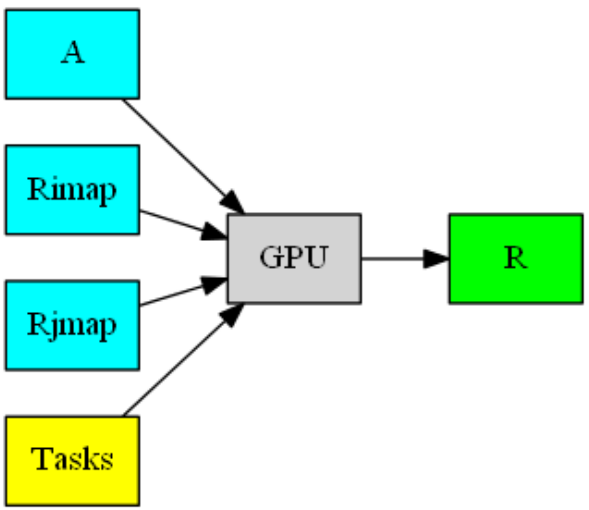
GPU Considerations

- **Memory limitations (availability, bandwidth)**
  - Maintain high ratio of computation to memory transfers
  - Use streams to mitigate transfer costs
  - Find opportunities to exploit locality & reuse memory
  - Use memory coalescing, "shared" memory, and register tricks
  - Avoid excessive cudaMalloc calls
- **Synchronization**
  - GPUs are great for data parallelism, bad for task parallelism
  - Ensuring memory consistency damages performance
  - No reliable inter-block synchronization primitives..yet

Outline

- **Introduction**
  - Multifrontal sparse QR
  - GPU considerations
- ▶ **GPU-based multifrontal sparse QR**
  - Memory traffic
  - Uberkernel
  - Front scheduling
  - Pipelined dense QR factorization

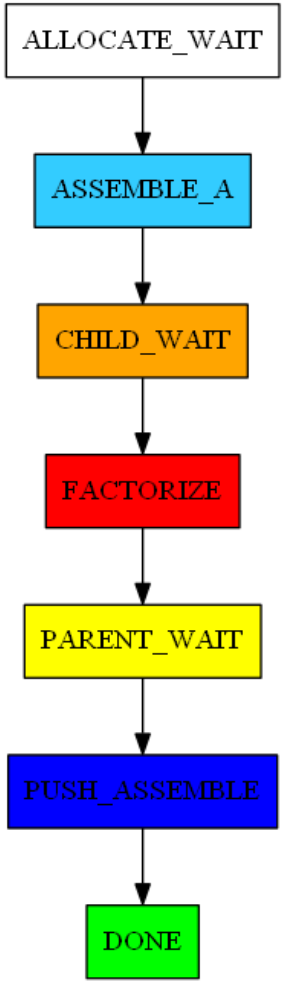
- **No superfluous memory traffic**
  - Data is transfered asynchronously, sometimes surgically
    - We can accomodate lazy JIT transfers if needed
  - Every front spends its entire life on the GPU
    - It's allocated once any of its children are finished factorizing
    - It assembles its contribution block directly into its parent
    - Its memory is recovered once its rows of R are off the GPU



GPU-based Multifrontal Sparse QR

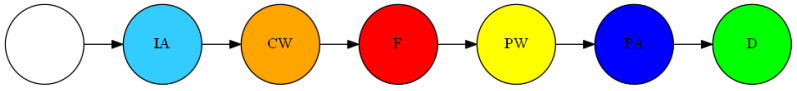
- **The uberkernel pattern allows us to process different types of tasks in the same kernel launch**
  - Initial Assembly (Input Problem to Front)
  - Householder Annihilate
  - Block Householder Apply
  - Pipelined Block Householder Apply & Annihilate
  - Push Assembly (Child to Parent)
- **Treat kernel launches as barrier syncs, and do as much work as possible per launch.**
  - Oversubscribe the GPU with work
  - Use events and streams to:
    - Launch the kernels asynchronously and build the next body of work while the GPU is busy
    - Ship the next set of tasks asynchronously

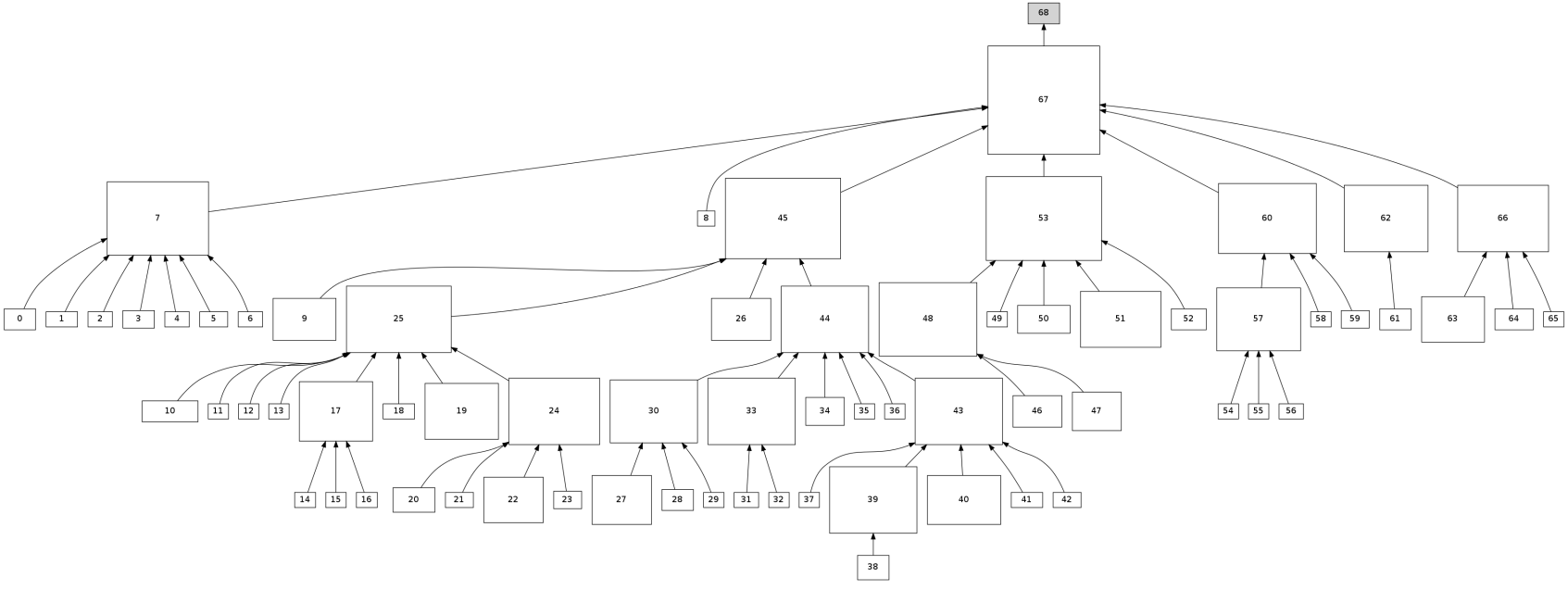
- **FSM-driven dynamic scheduling**



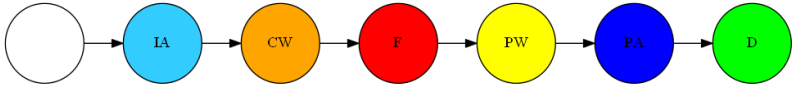
- **What does this look like?**
  - "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."
    - Brian Kernighan, "Unix for Beginners" (1979)

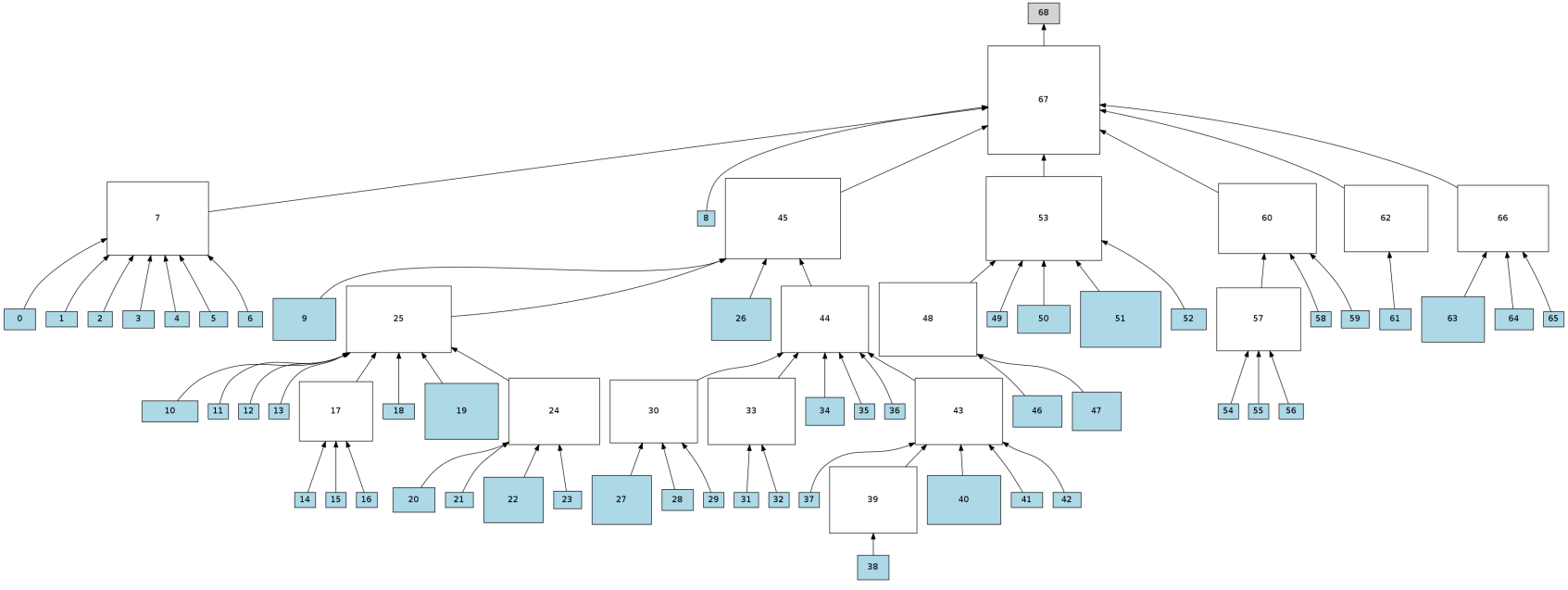
Front Scheduling



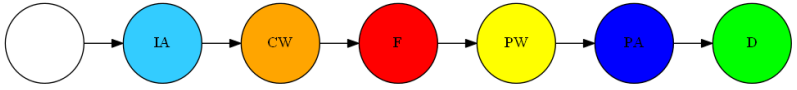


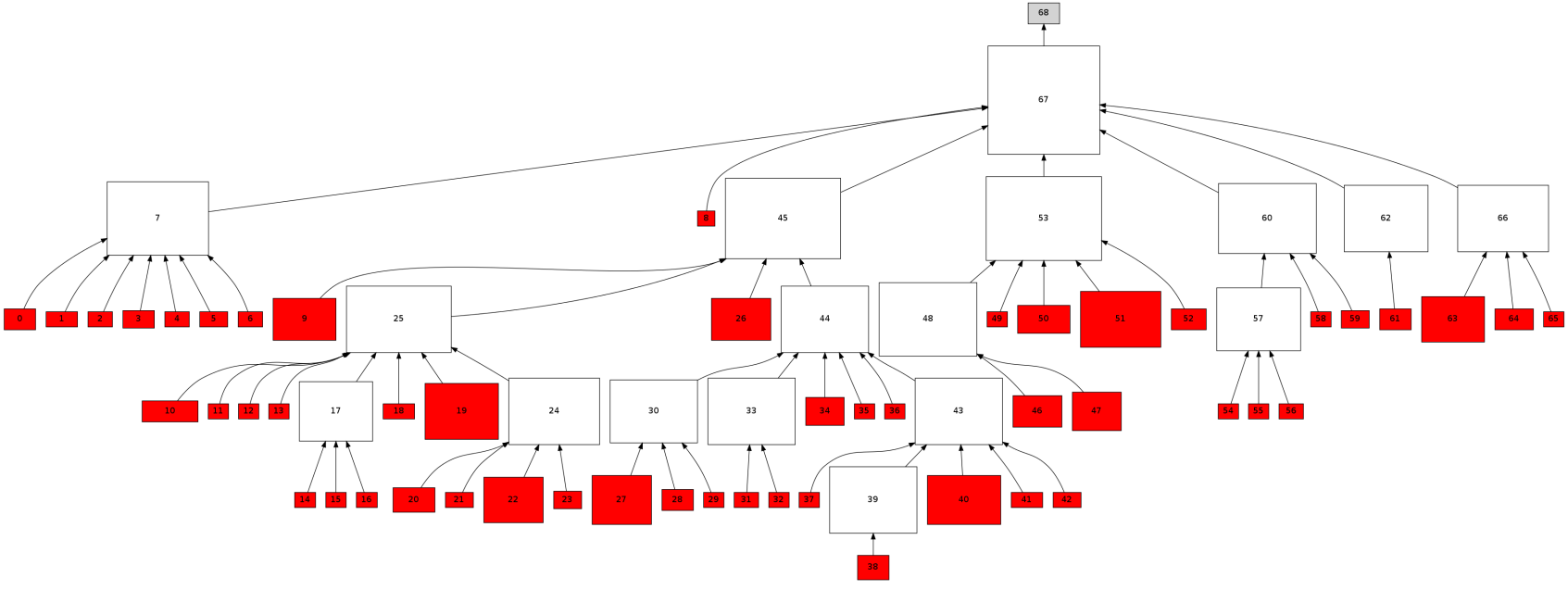
Front Scheduling



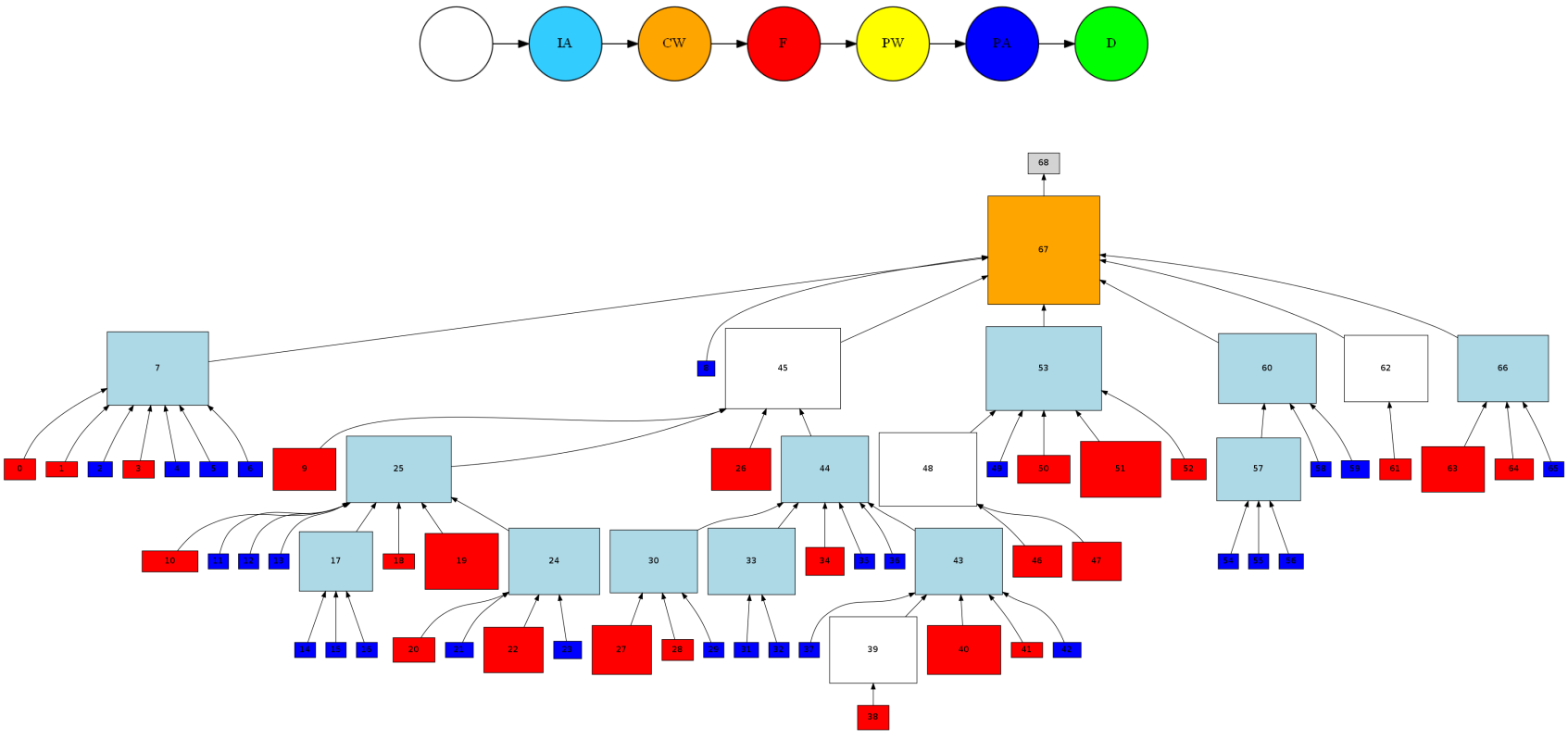


Front Scheduling

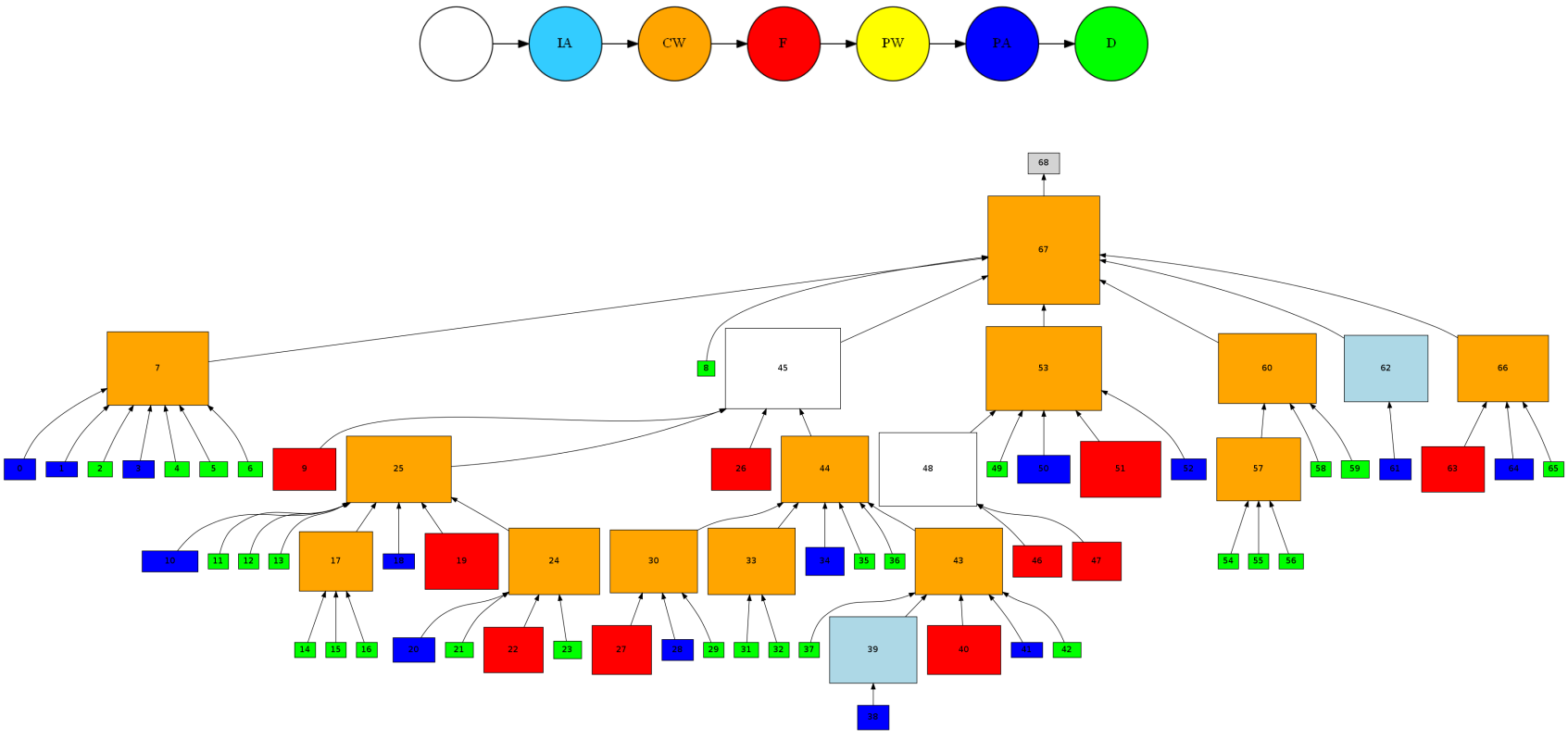




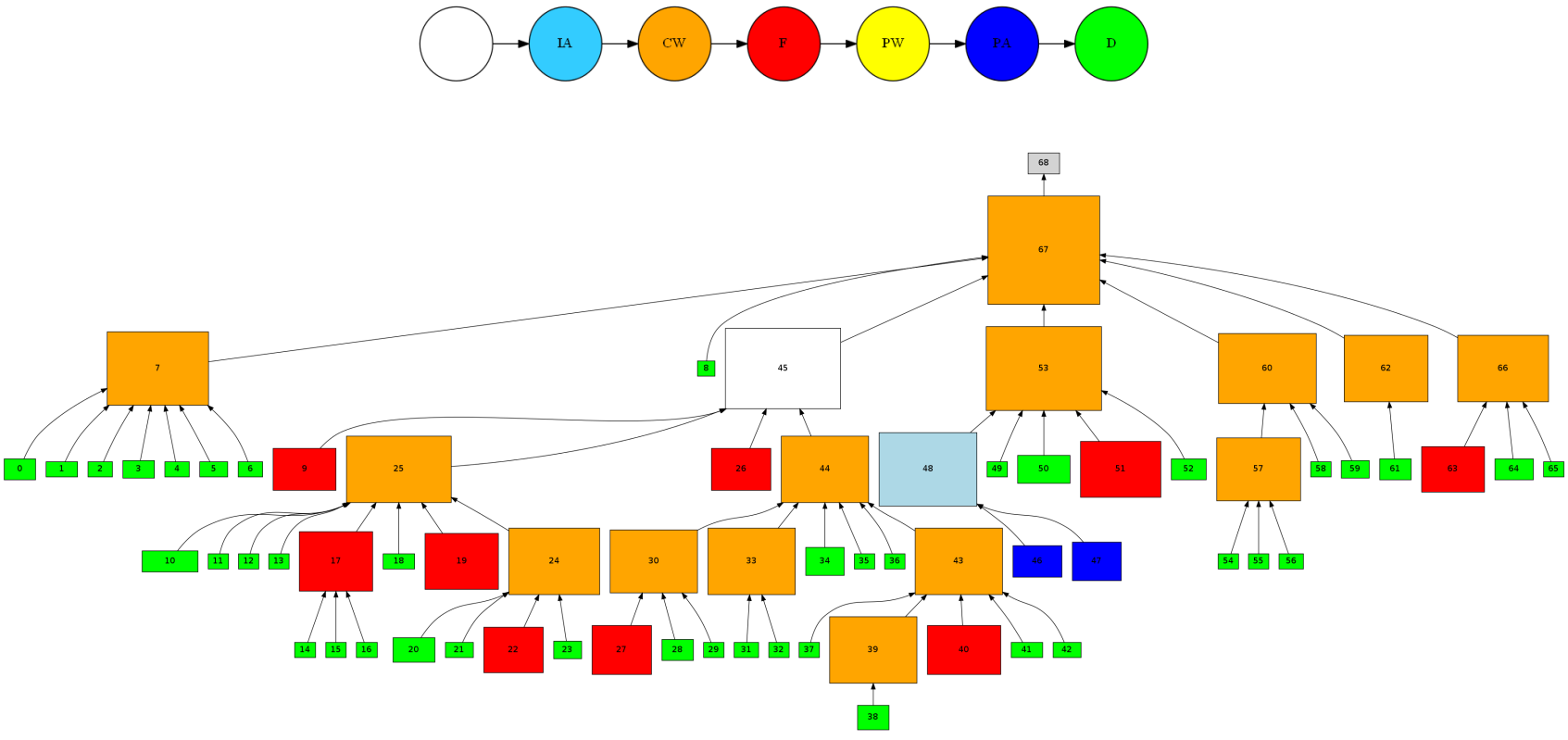
Front Scheduling



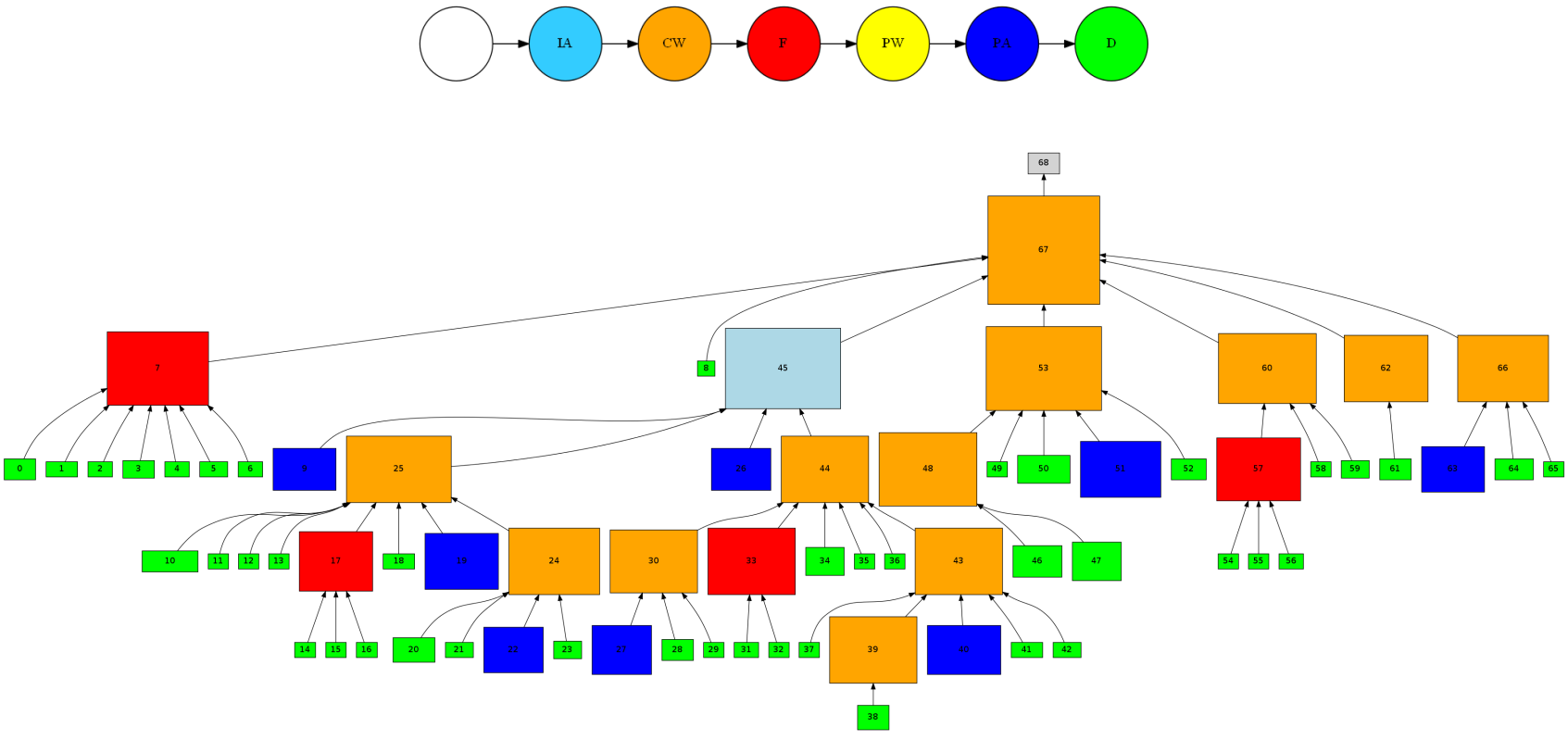
Front Scheduling



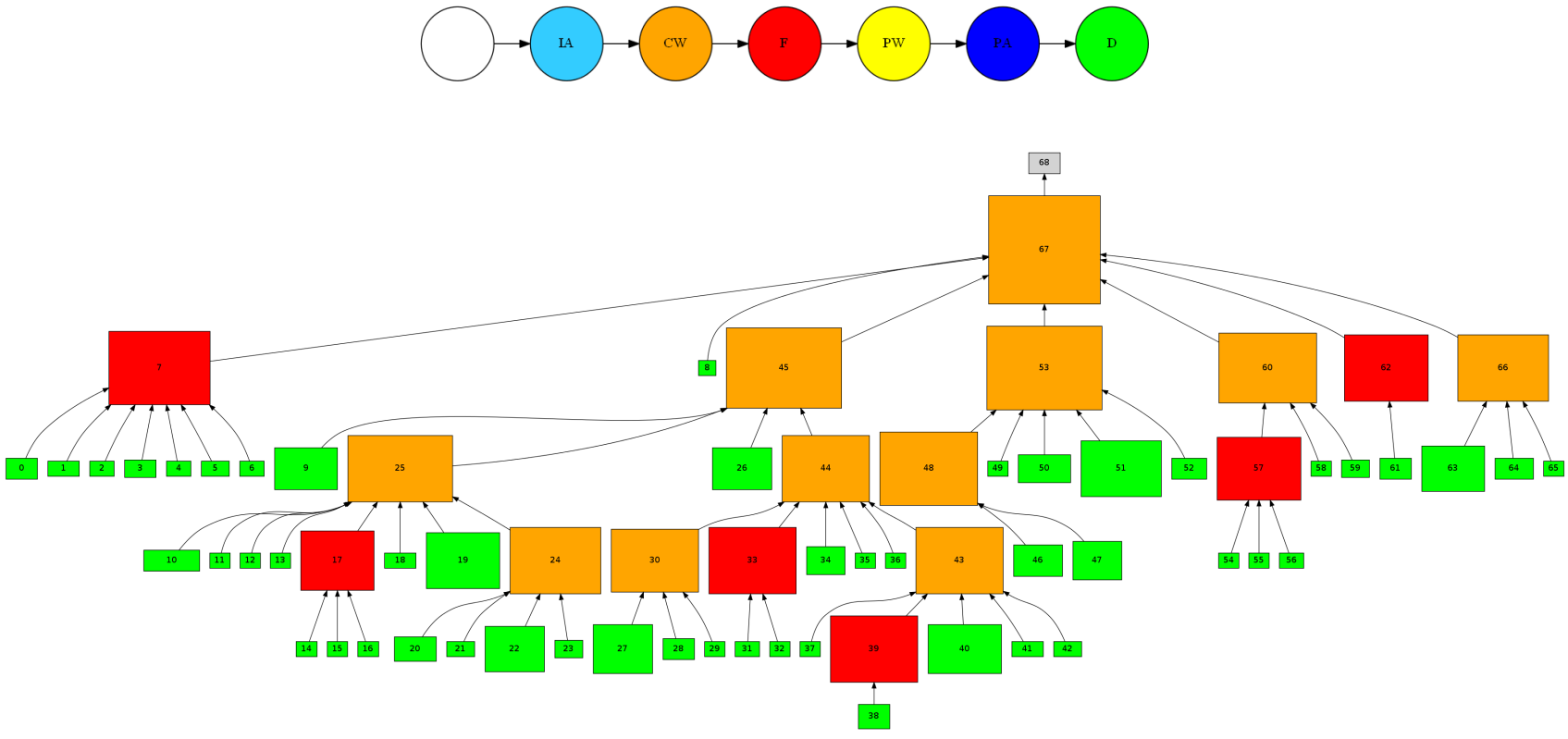
Front Scheduling



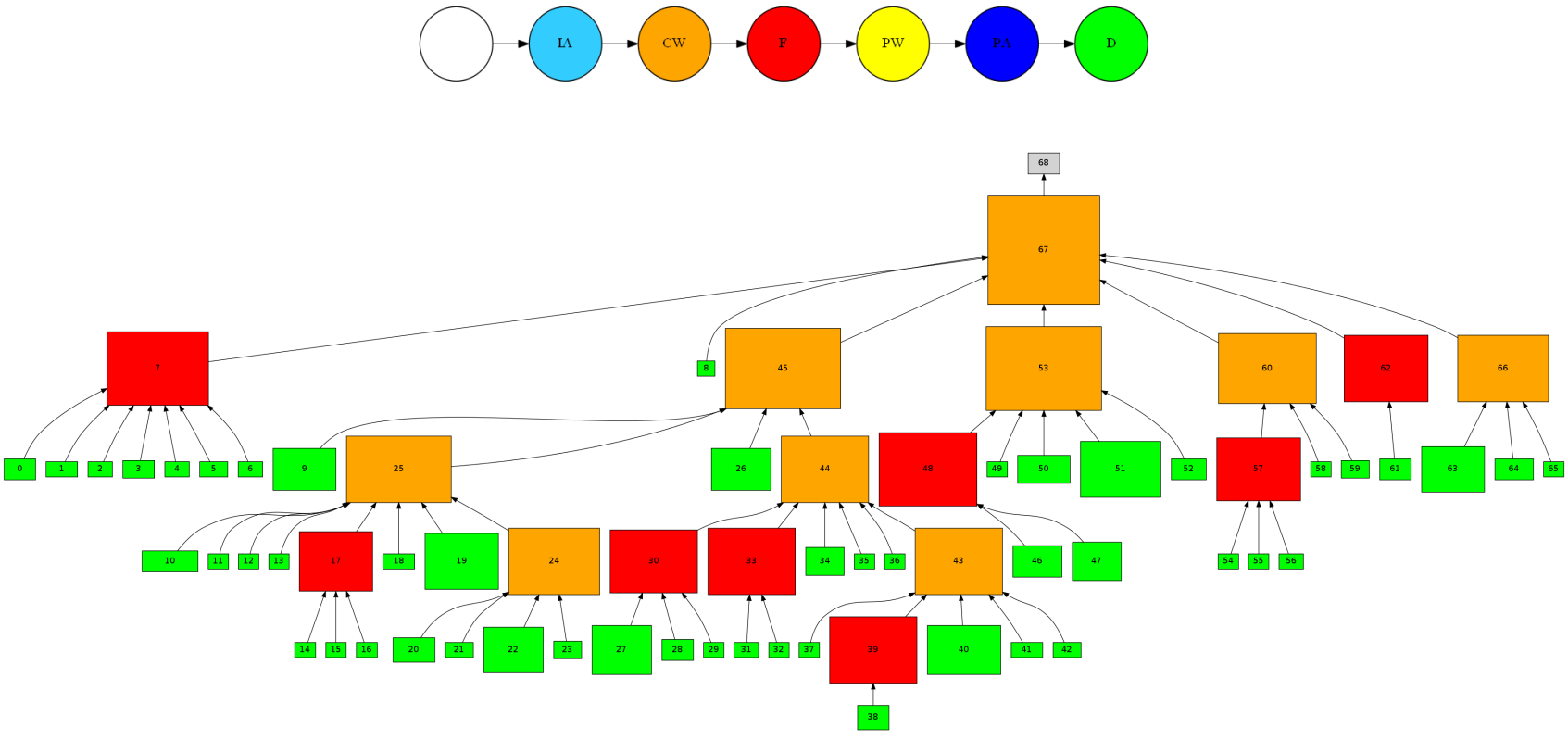
Front Scheduling



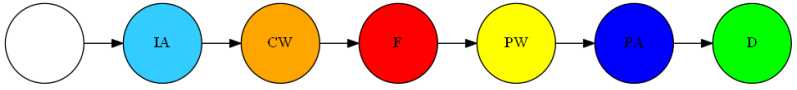
Front Scheduling

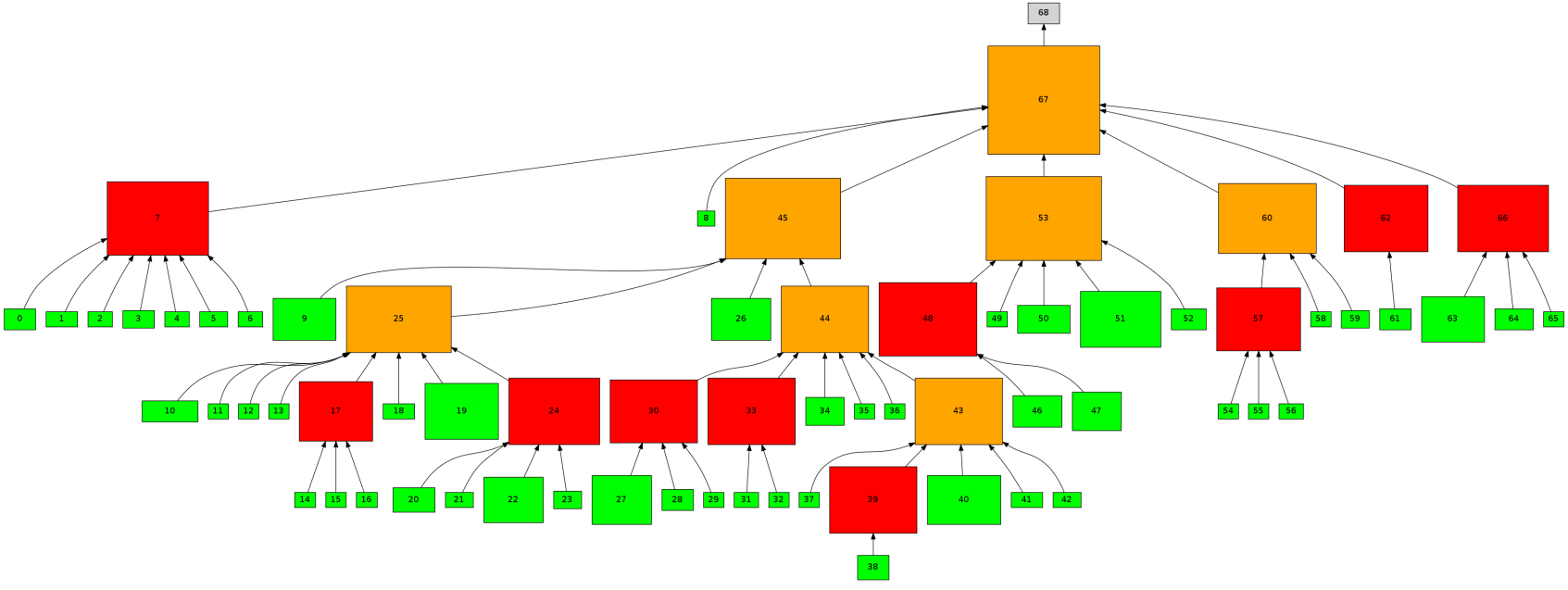


Front Scheduling

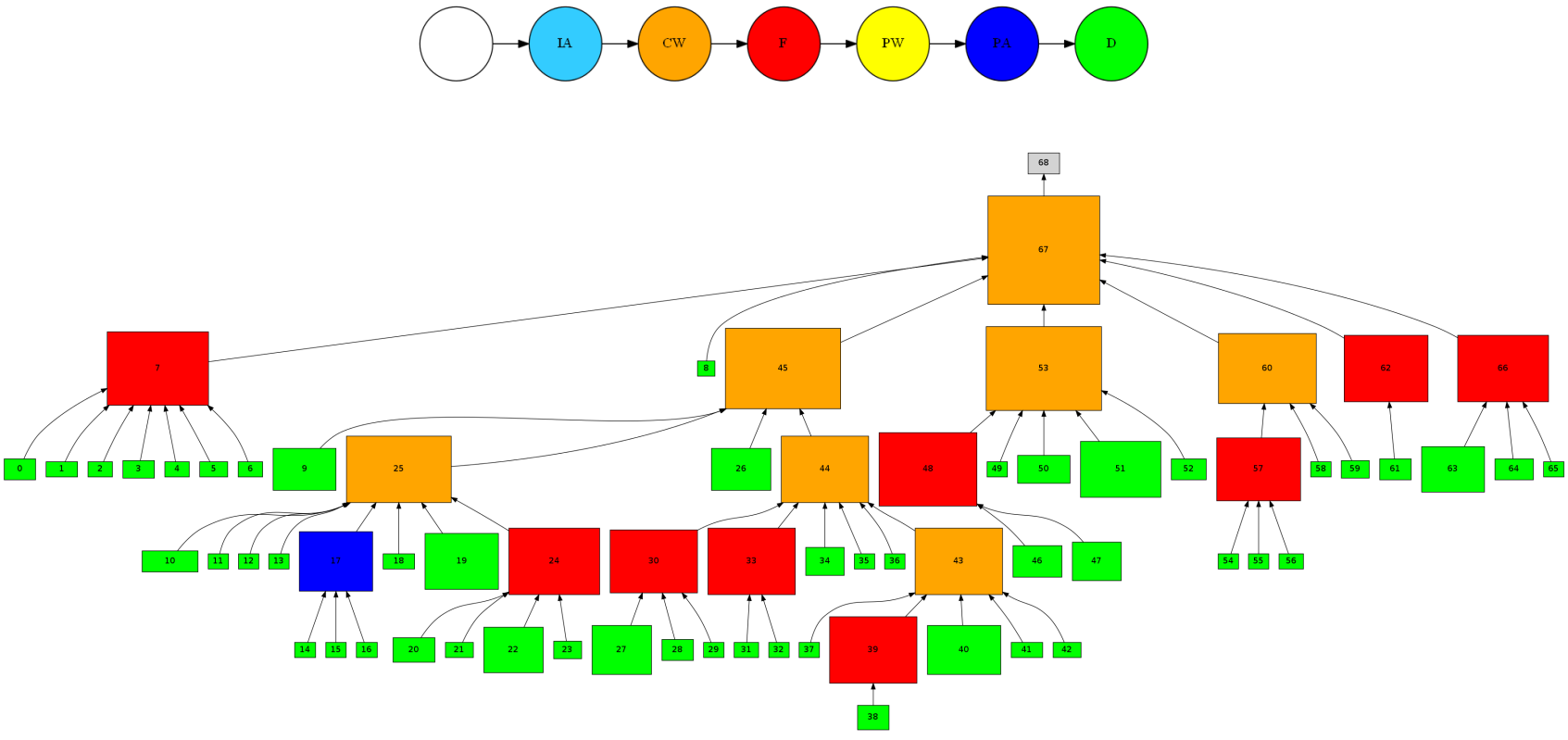


Front Scheduling

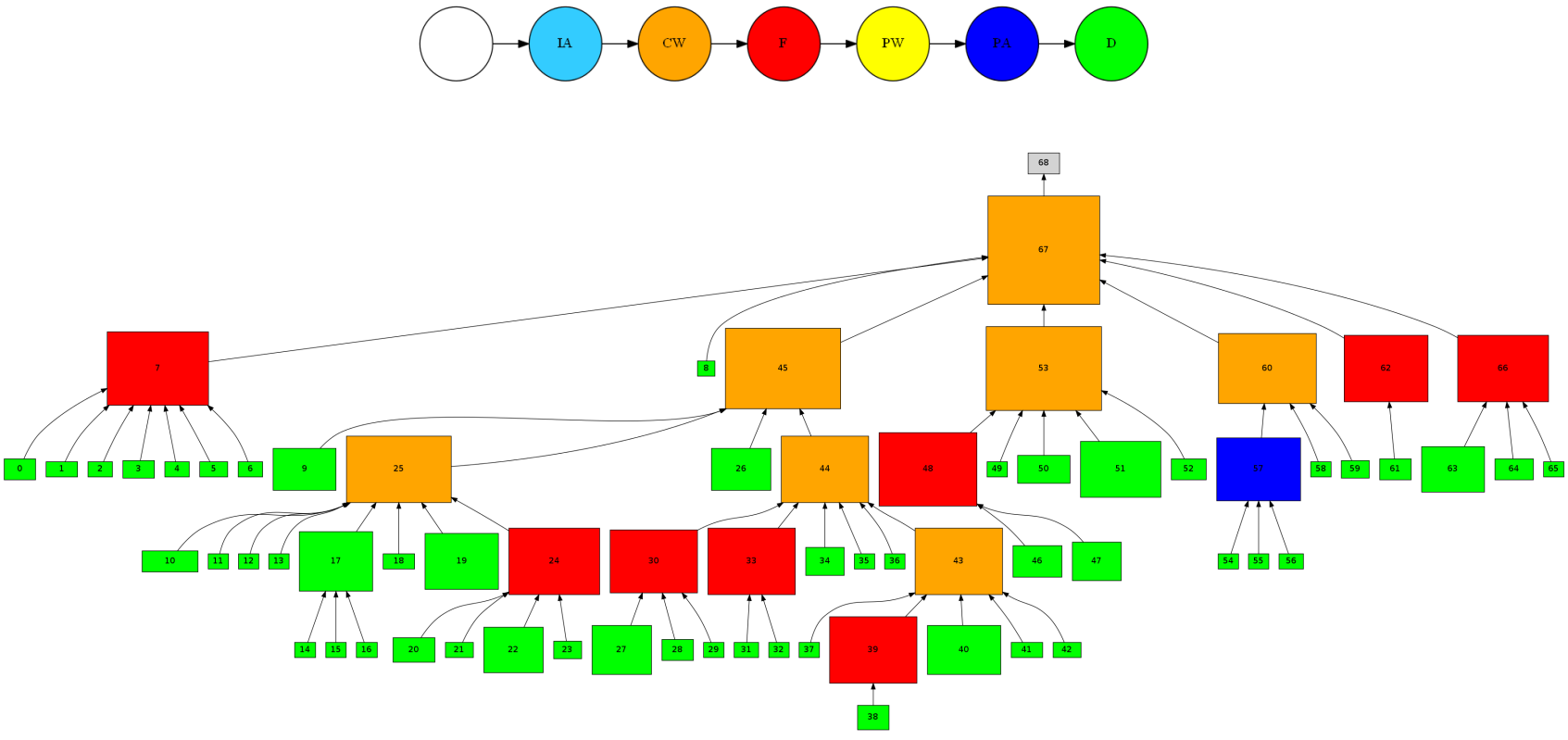




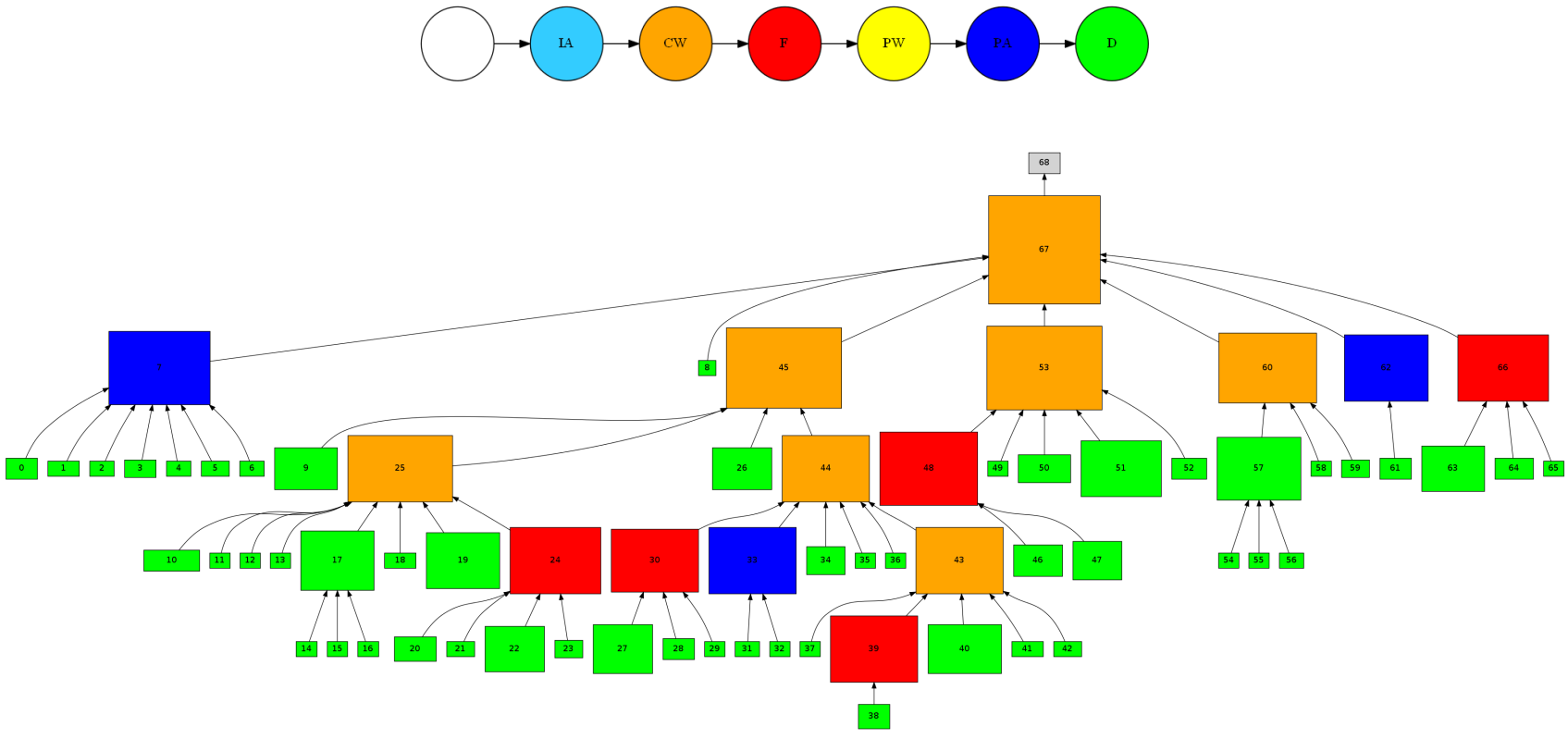
Front Scheduling



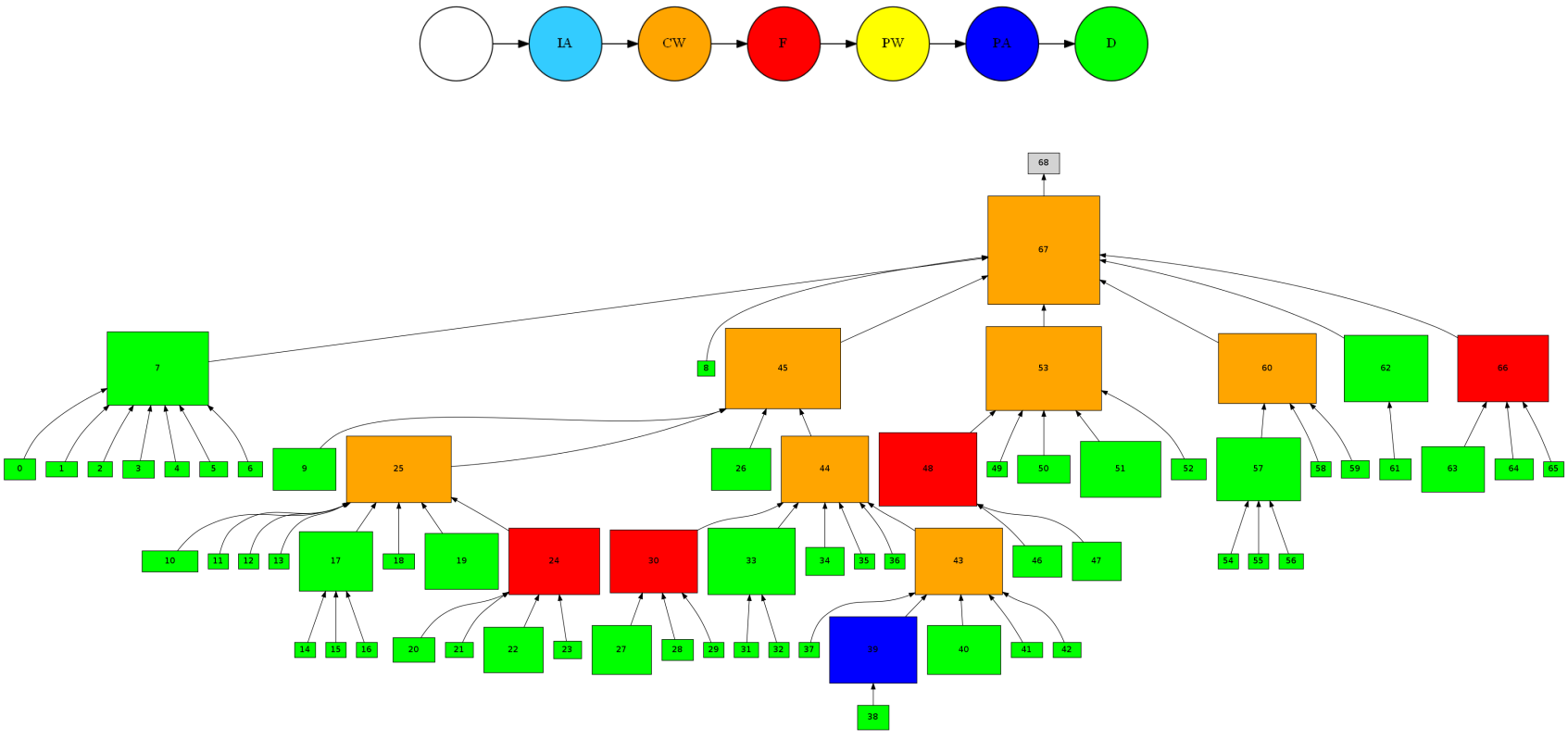
Front Scheduling



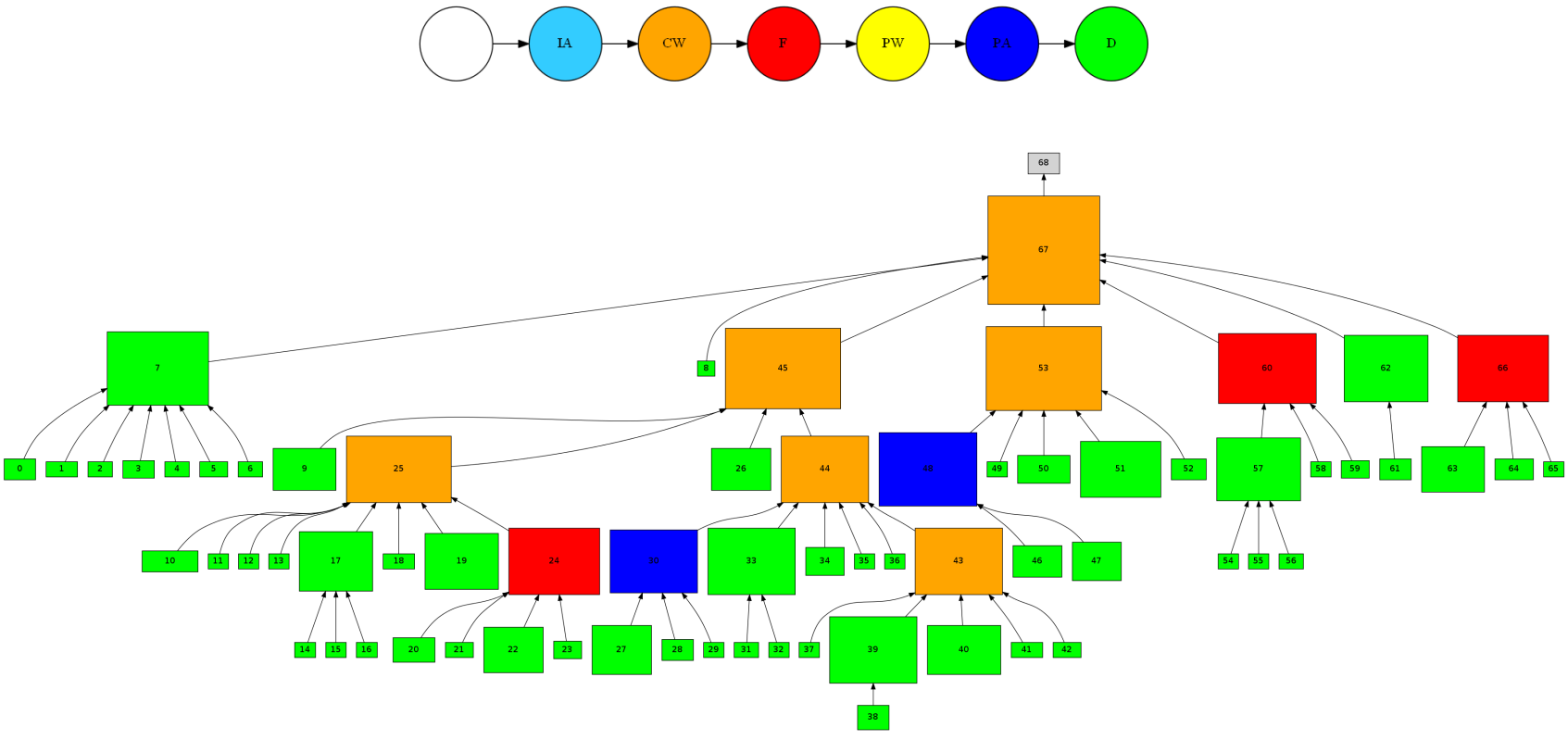
Front Scheduling



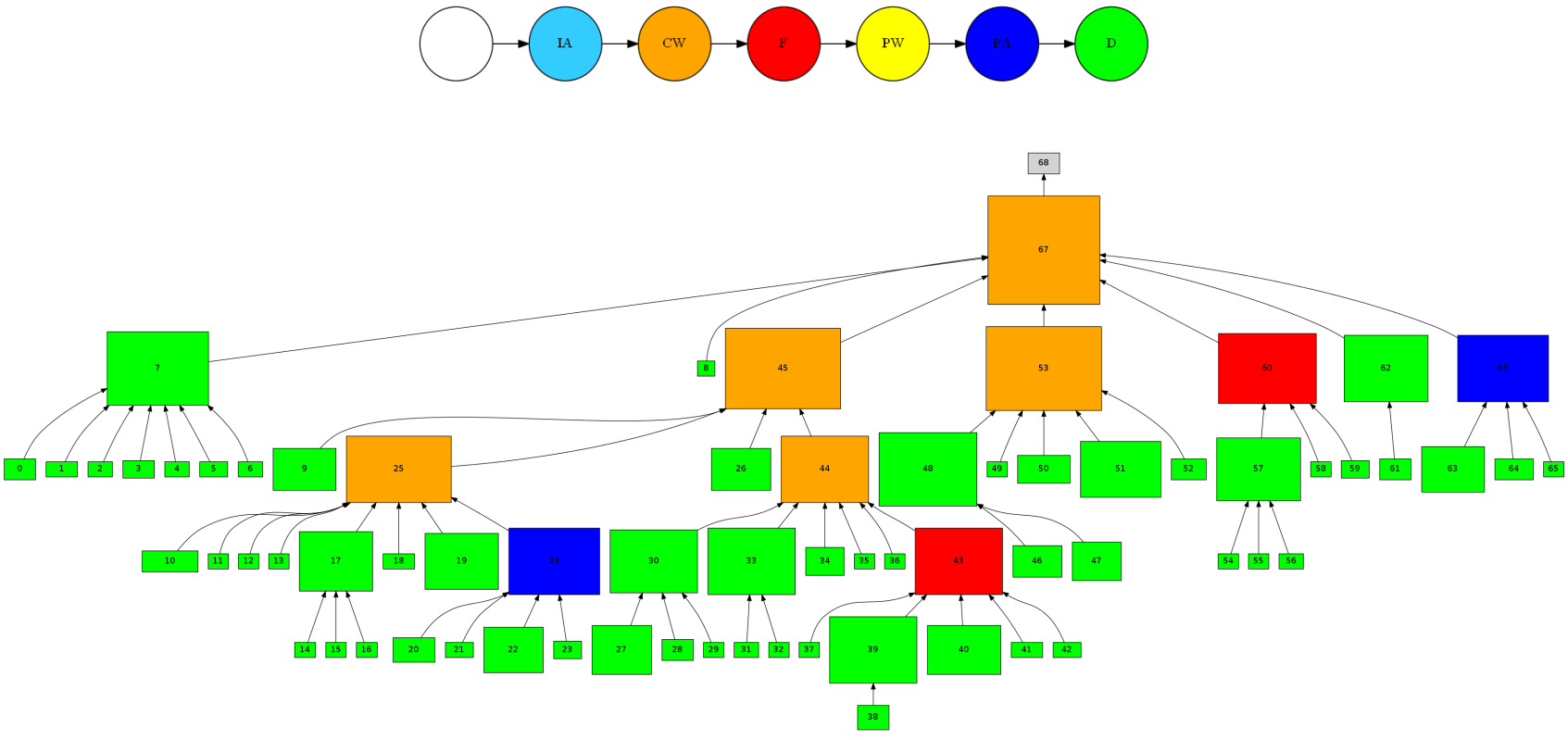
Front Scheduling



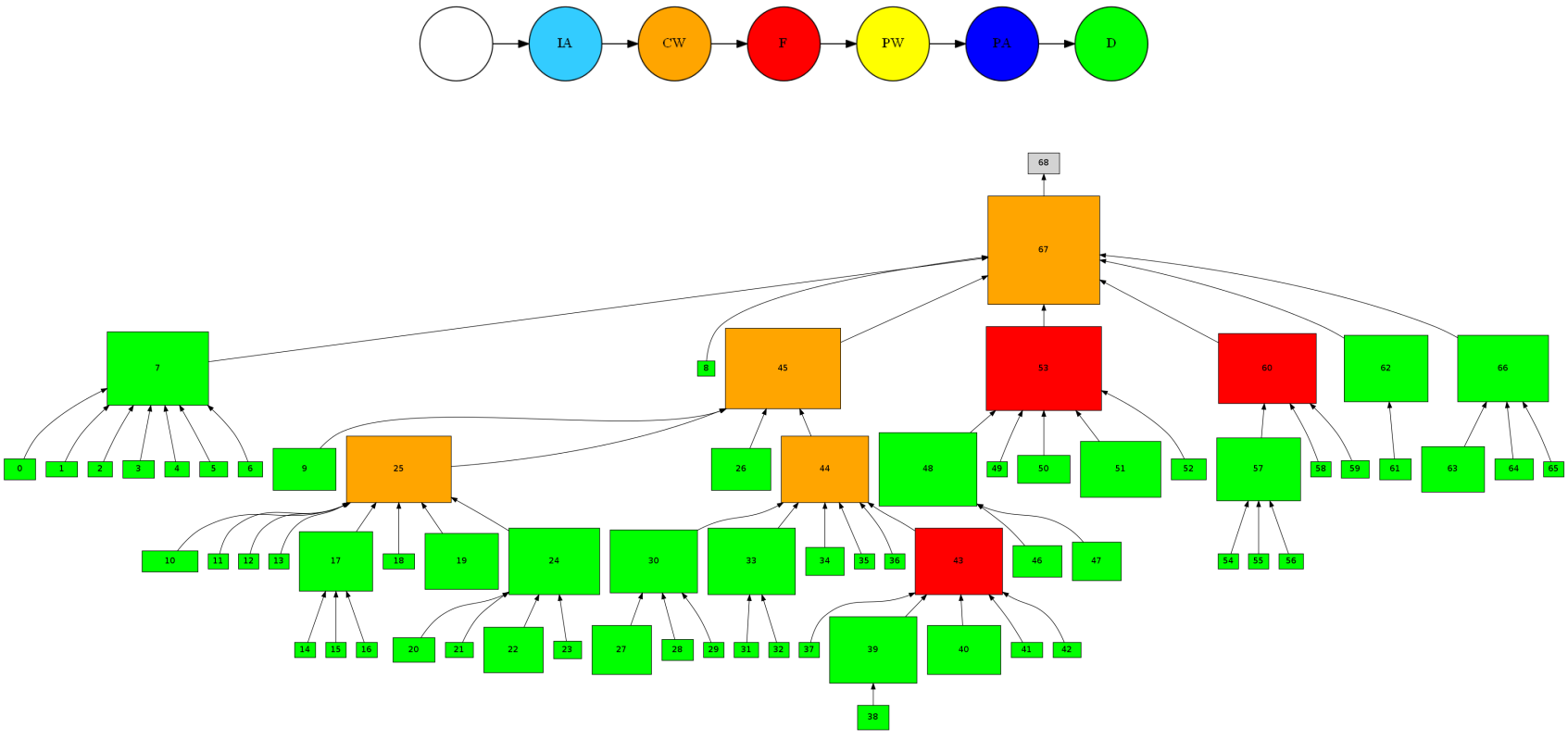
Front Scheduling



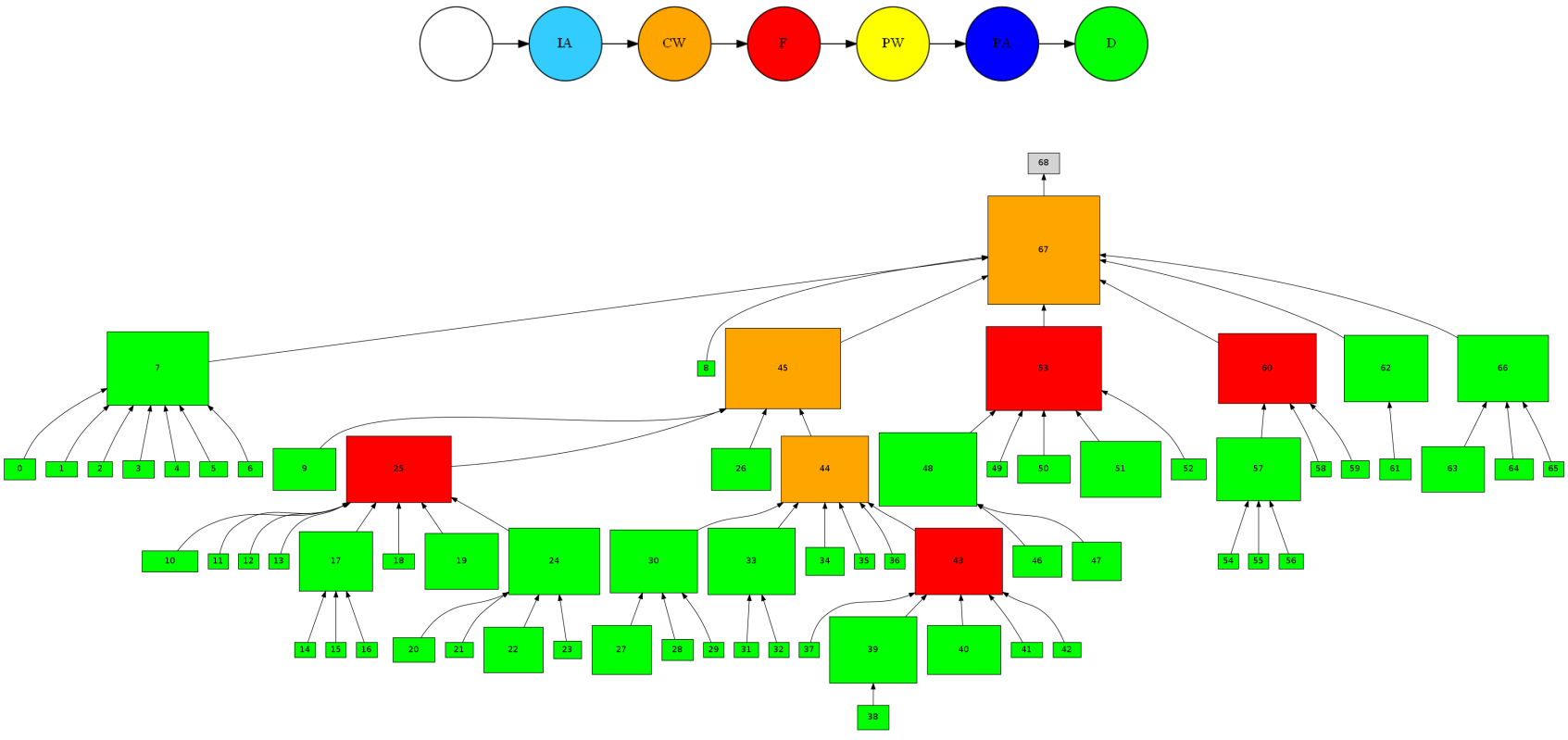
Front Scheduling



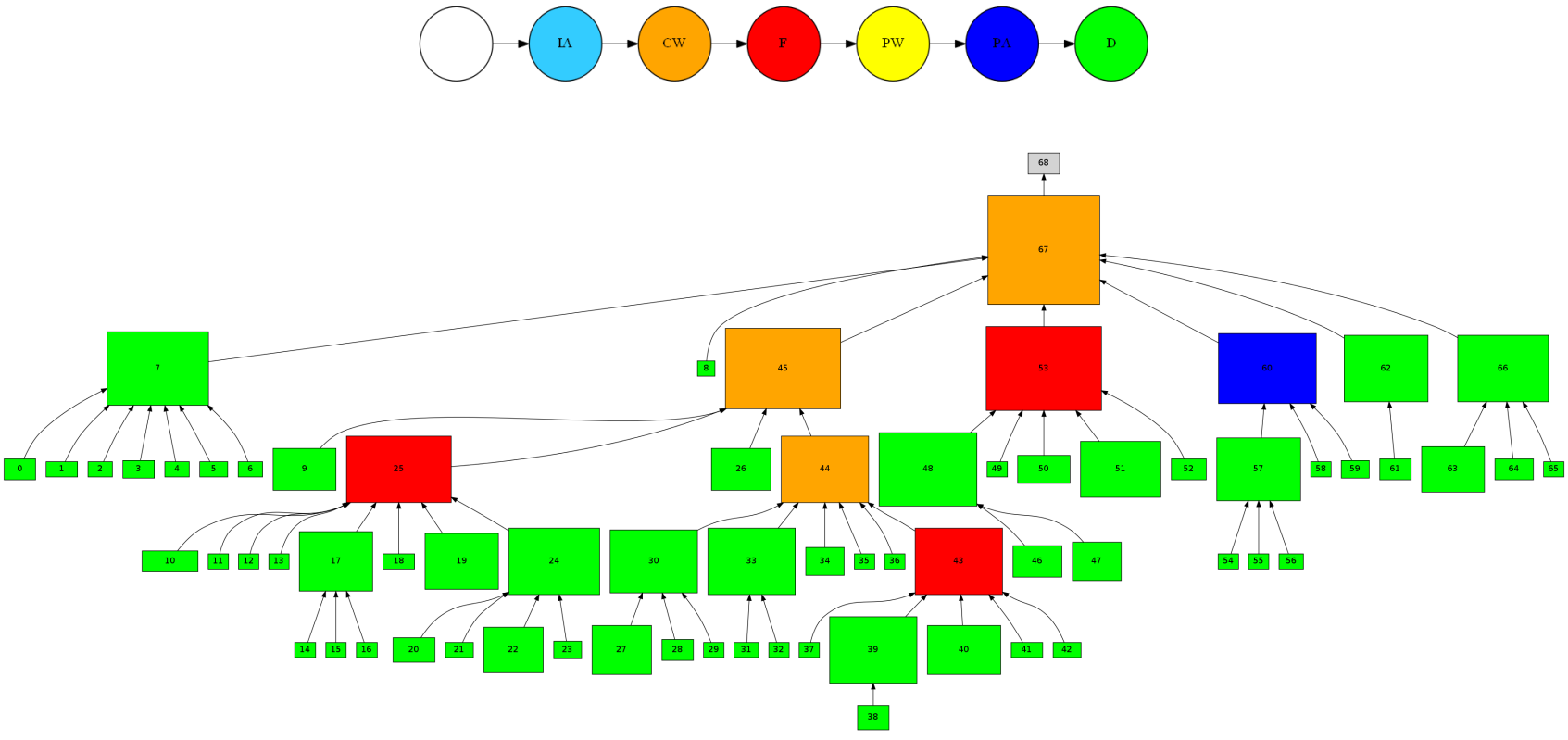
Front Scheduling



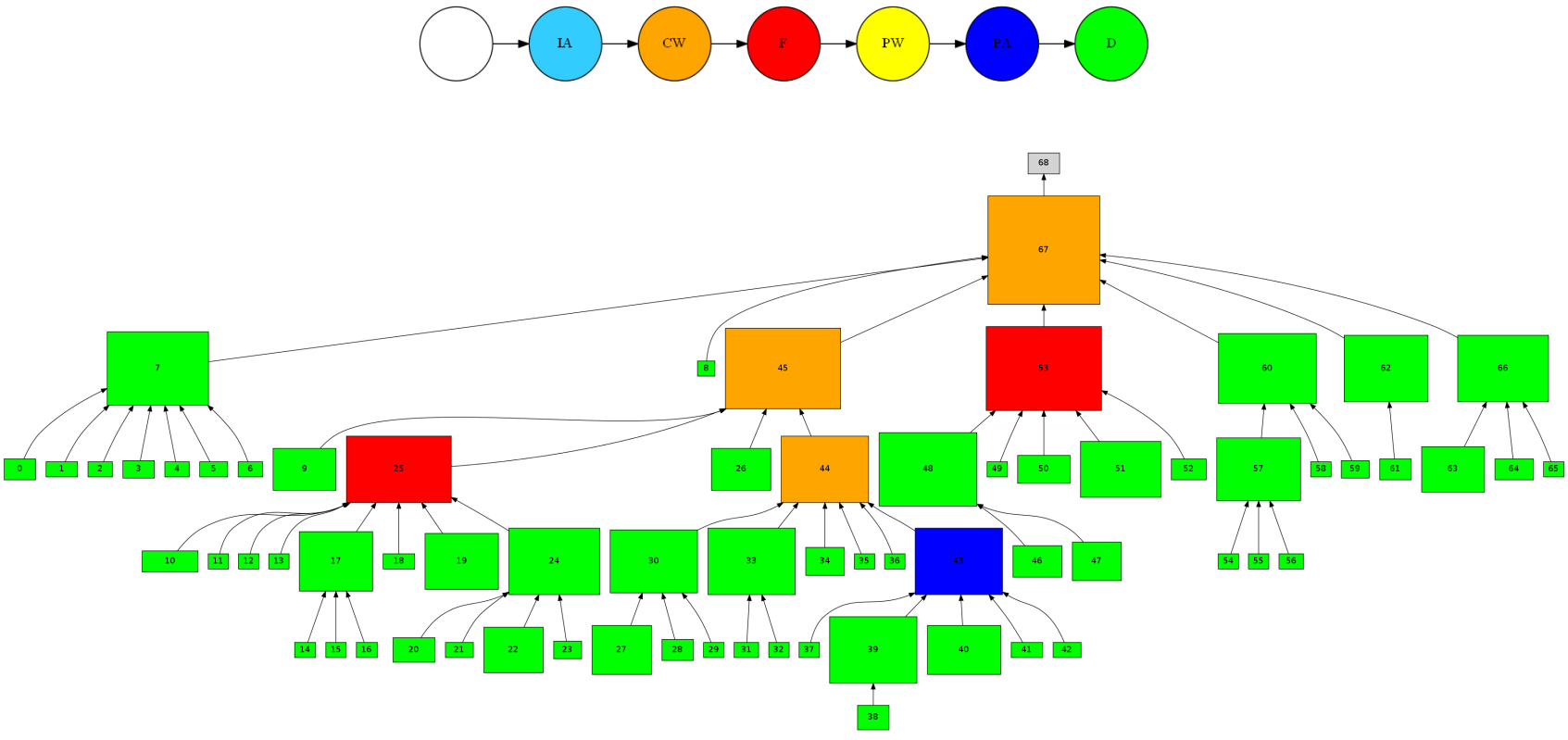
Front Scheduling



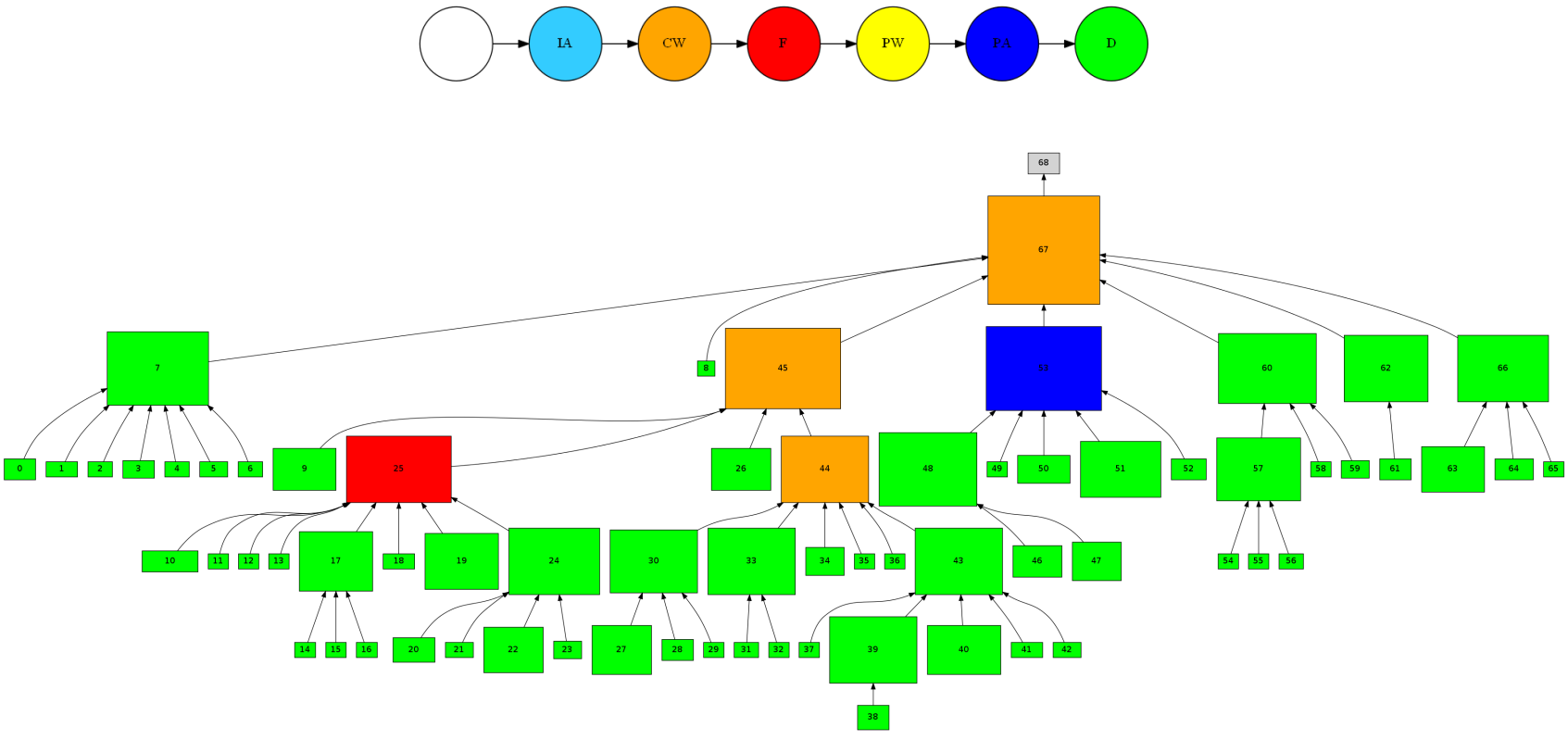
Front Scheduling



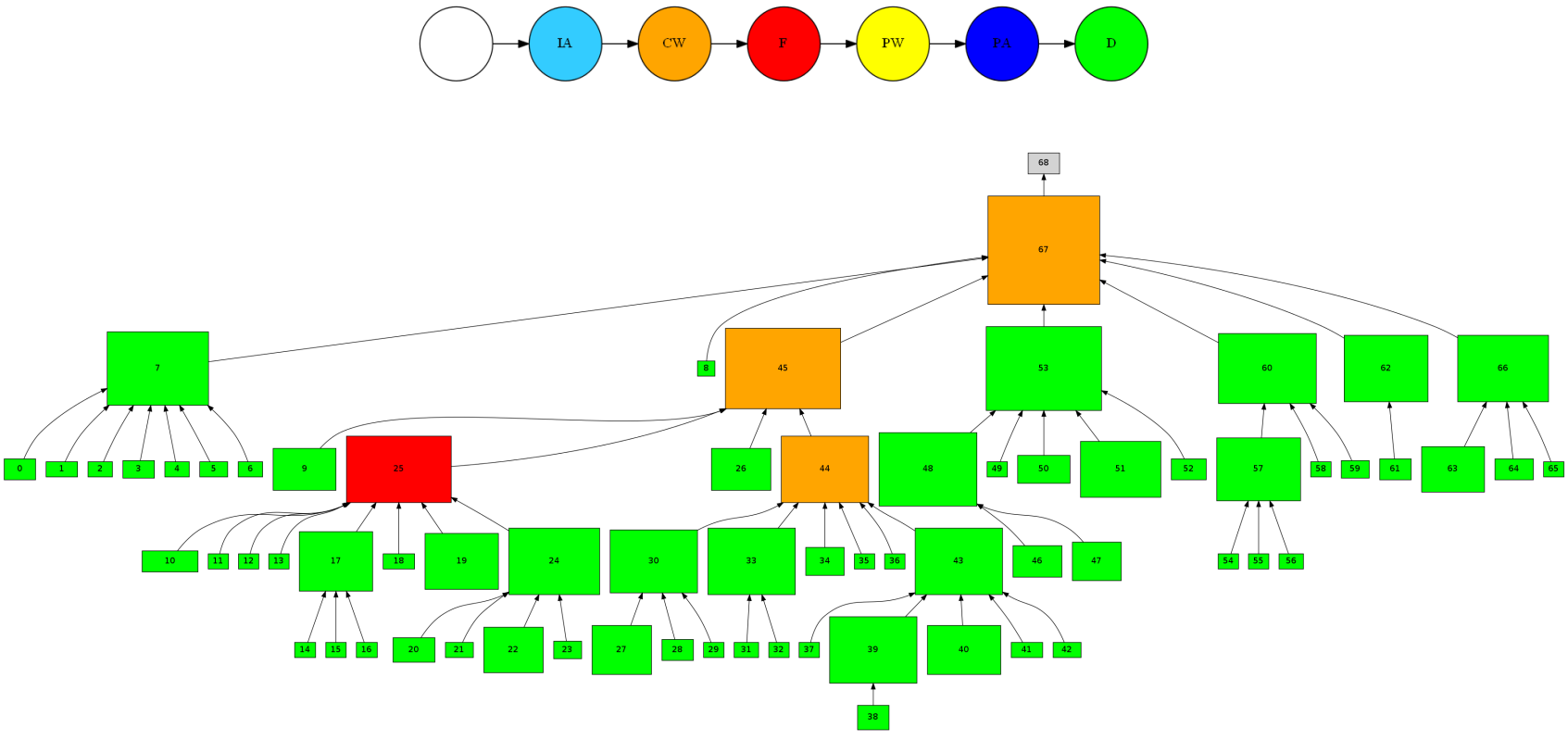
Front Scheduling



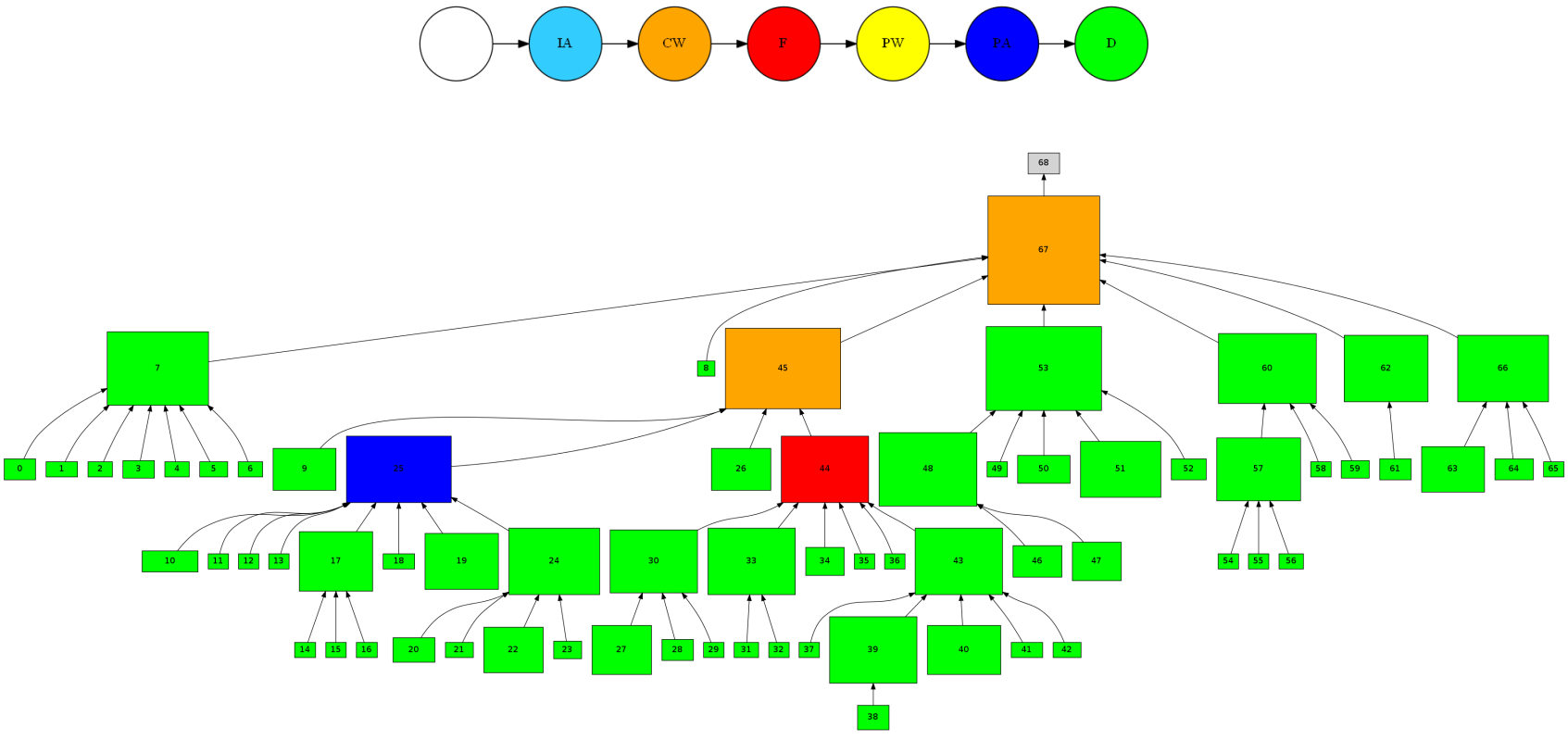
Front Scheduling



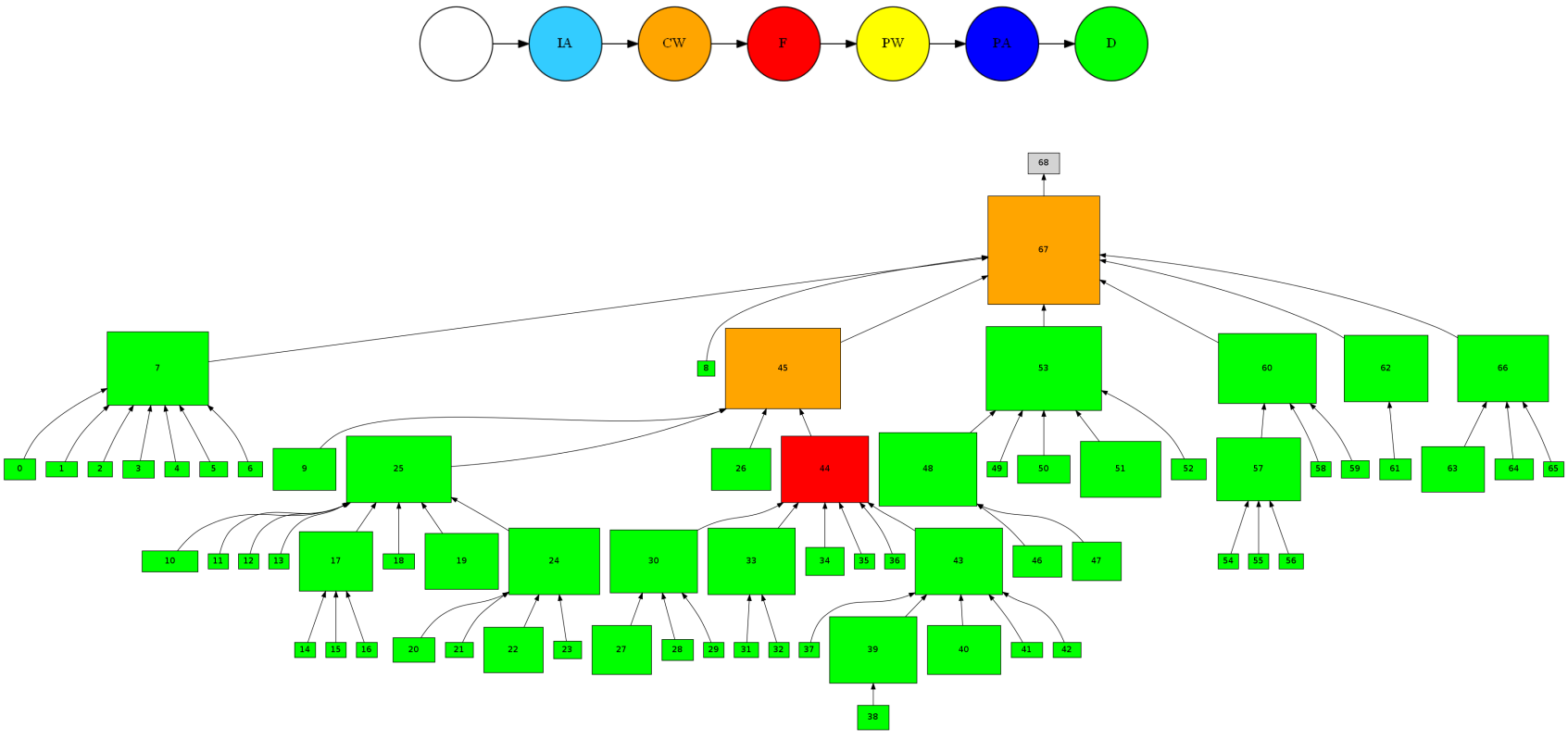
Front Scheduling



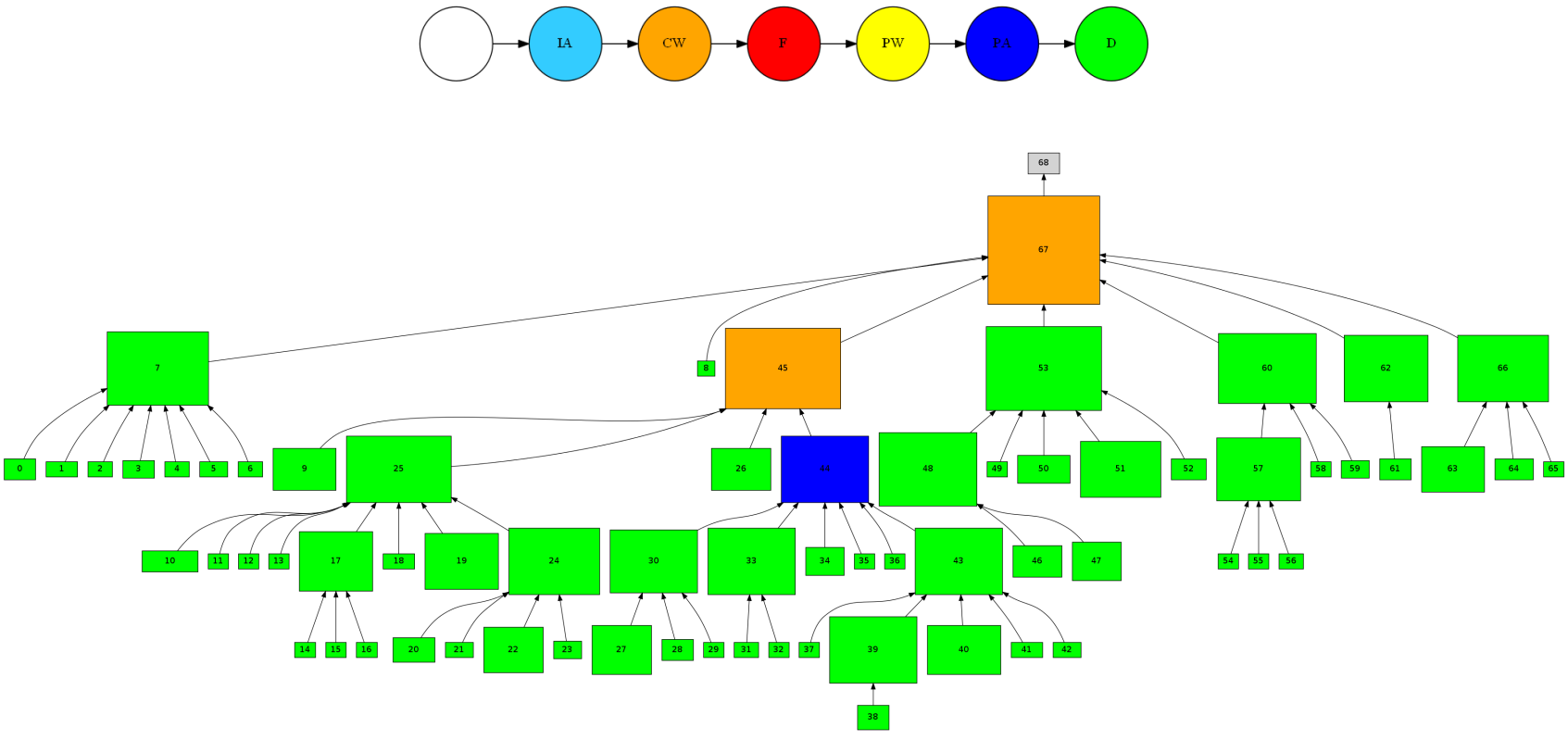
Front Scheduling



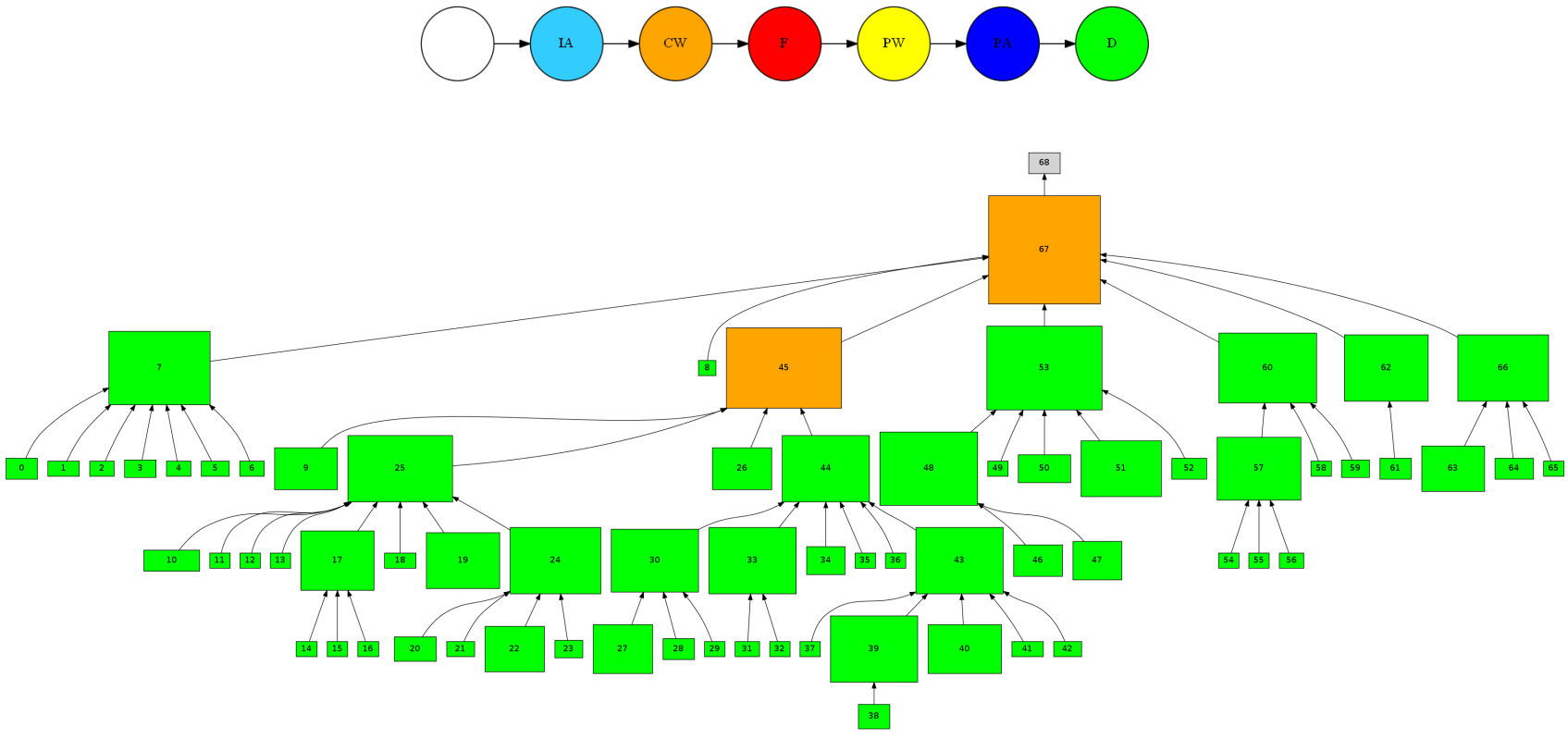
Front Scheduling



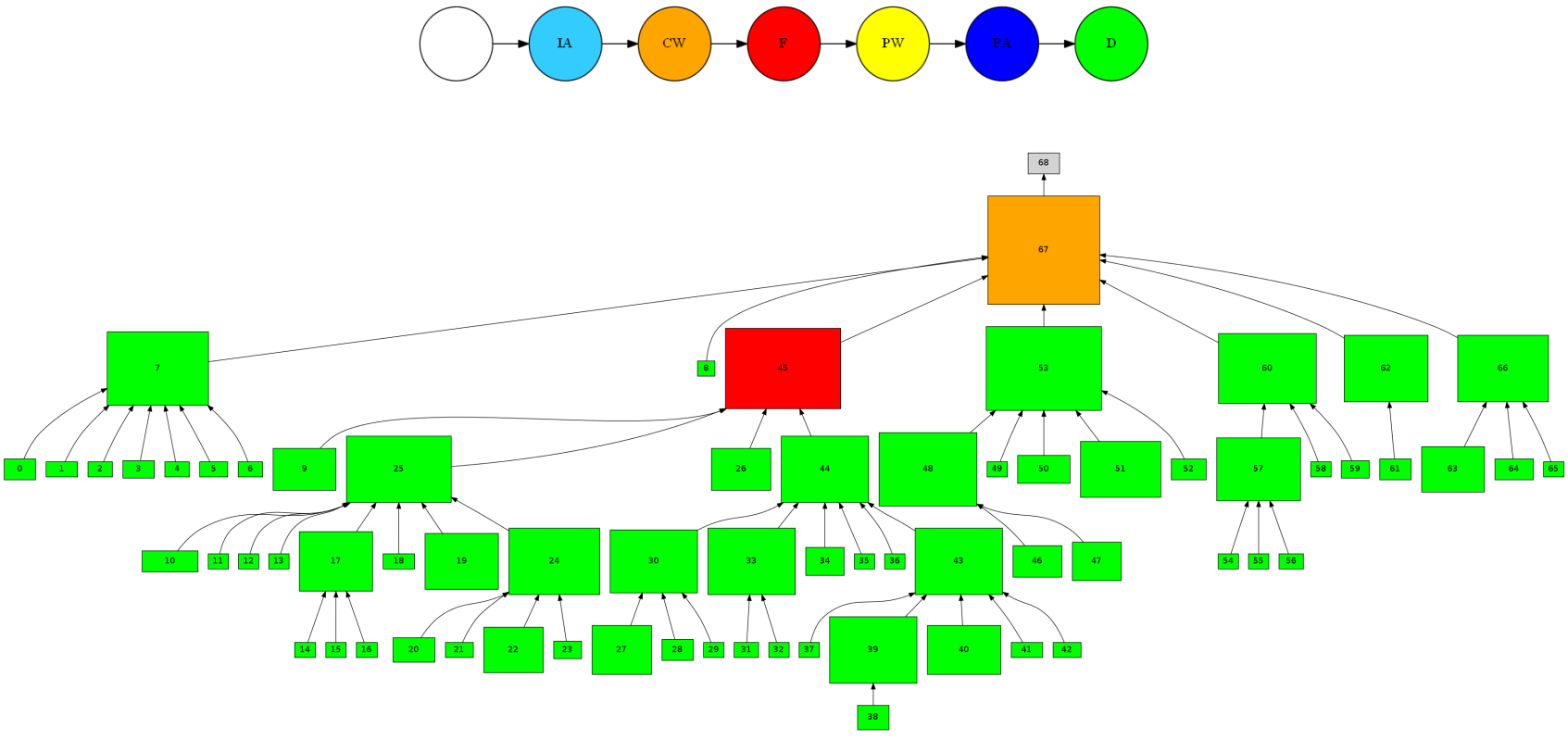
Front Scheduling



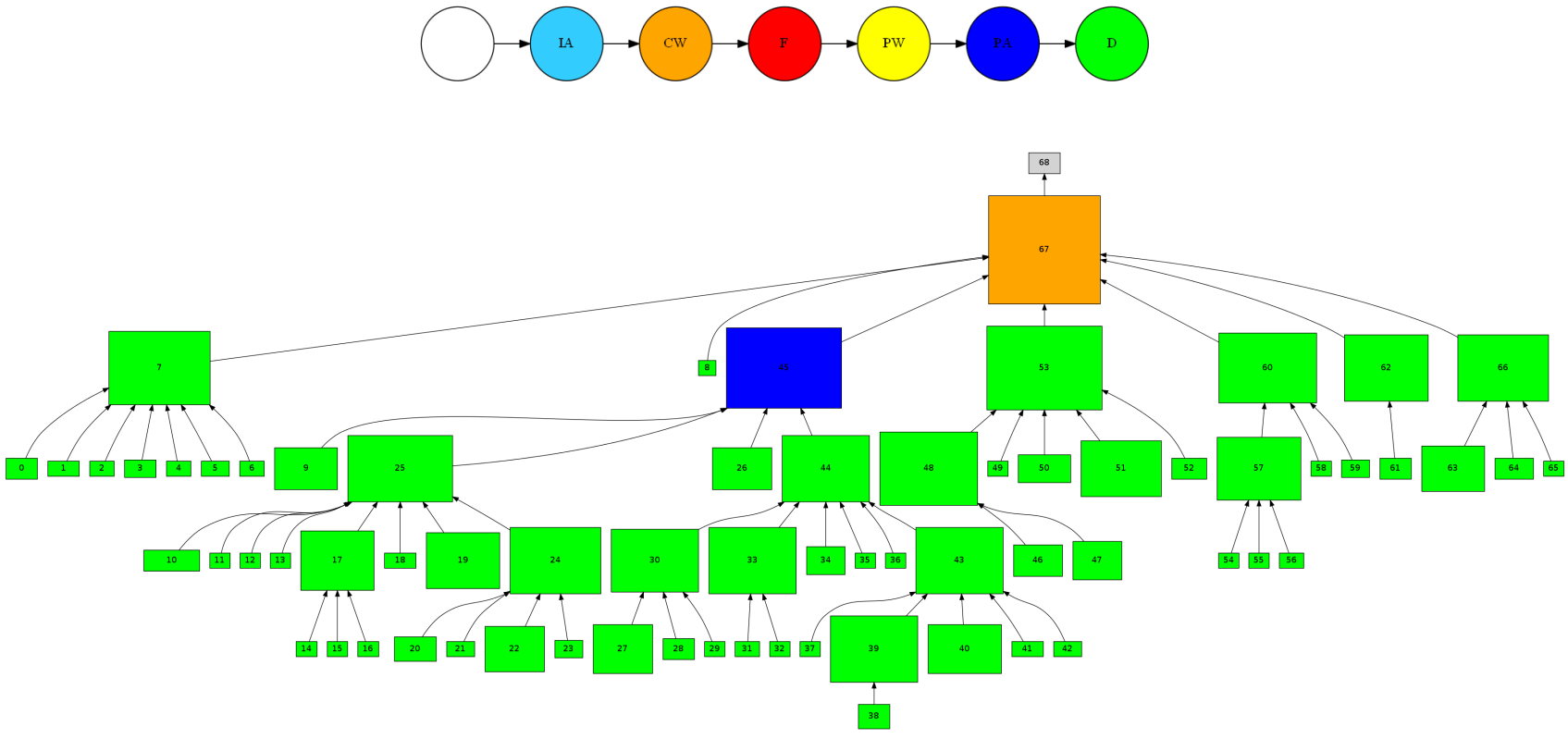
Front Scheduling



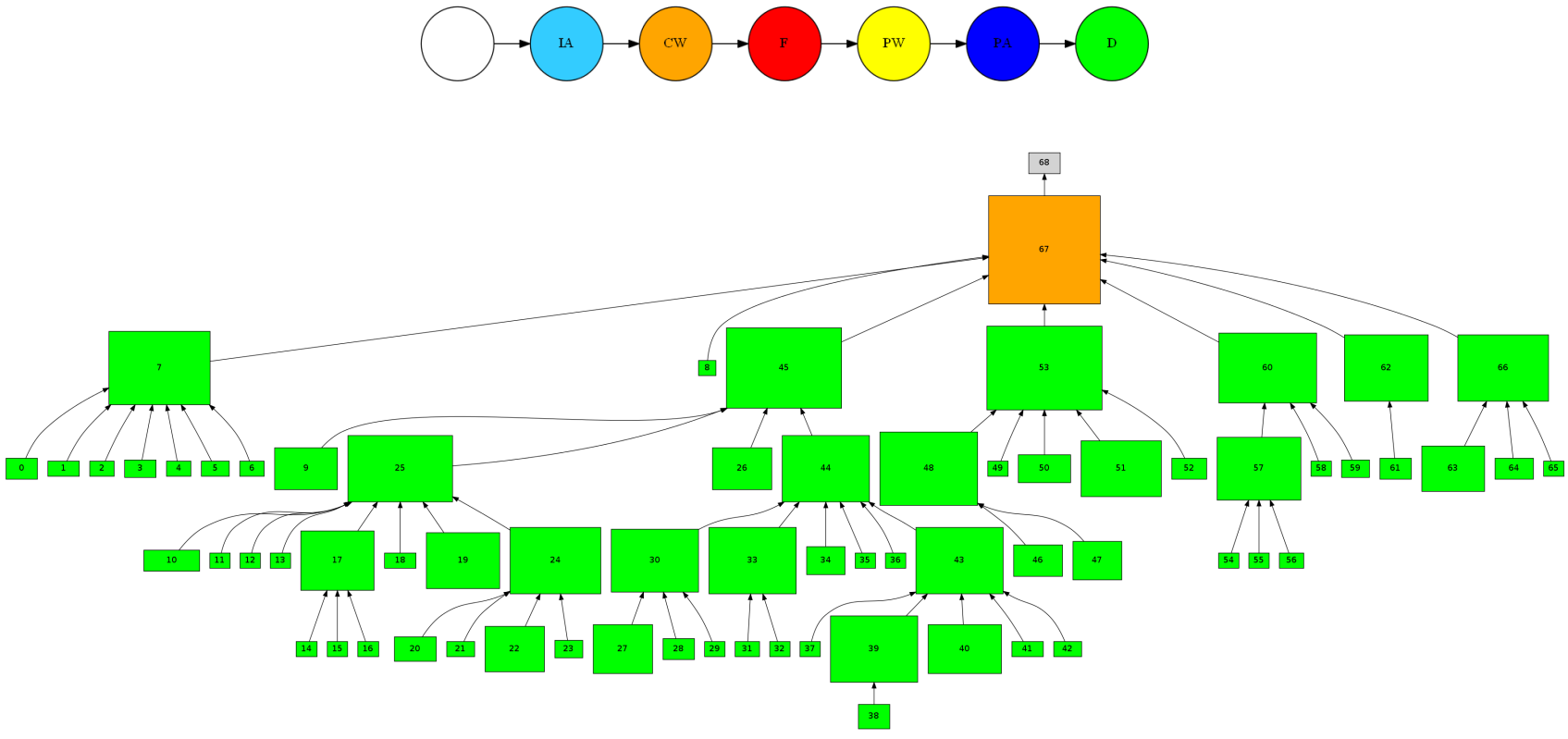
Front Scheduling



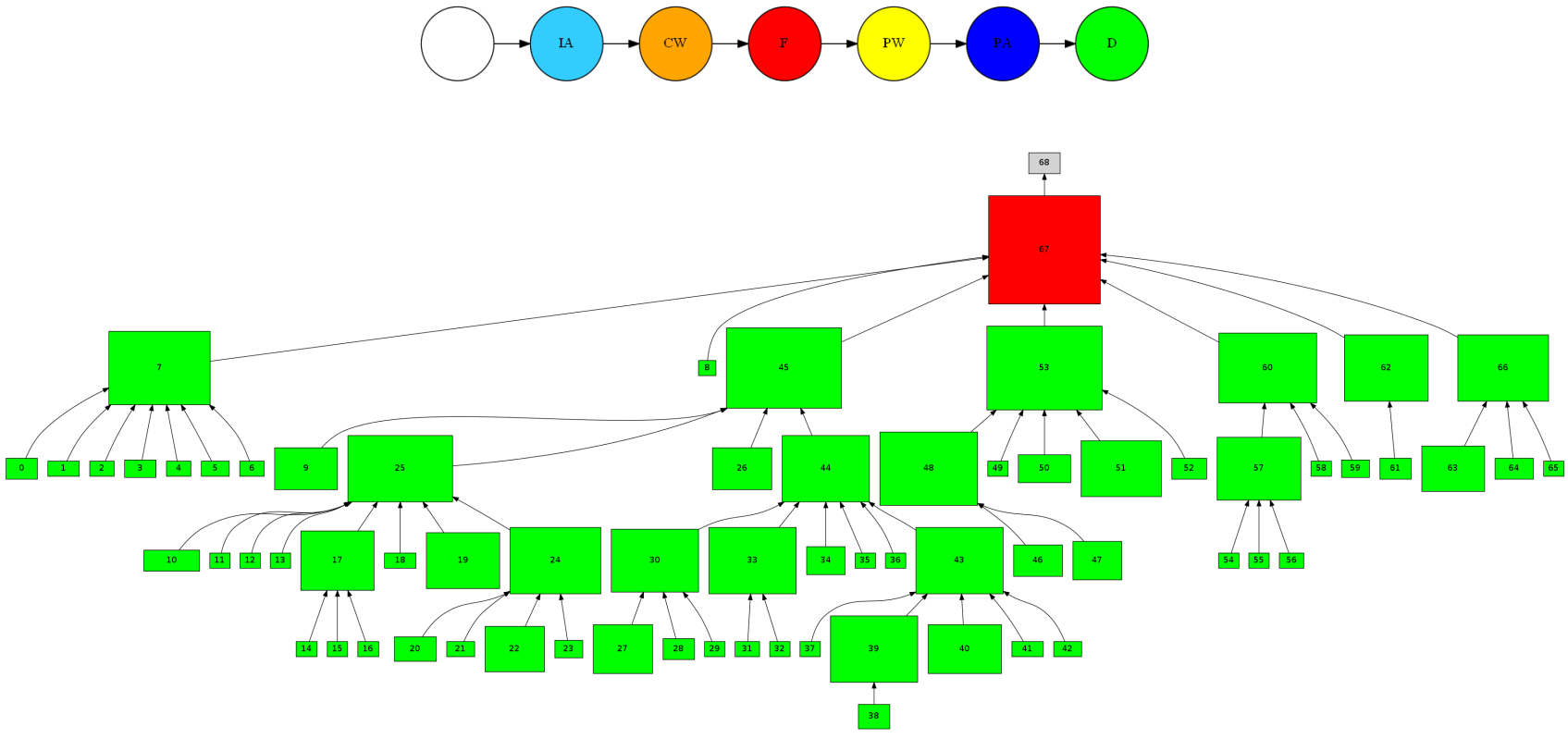
Front Scheduling



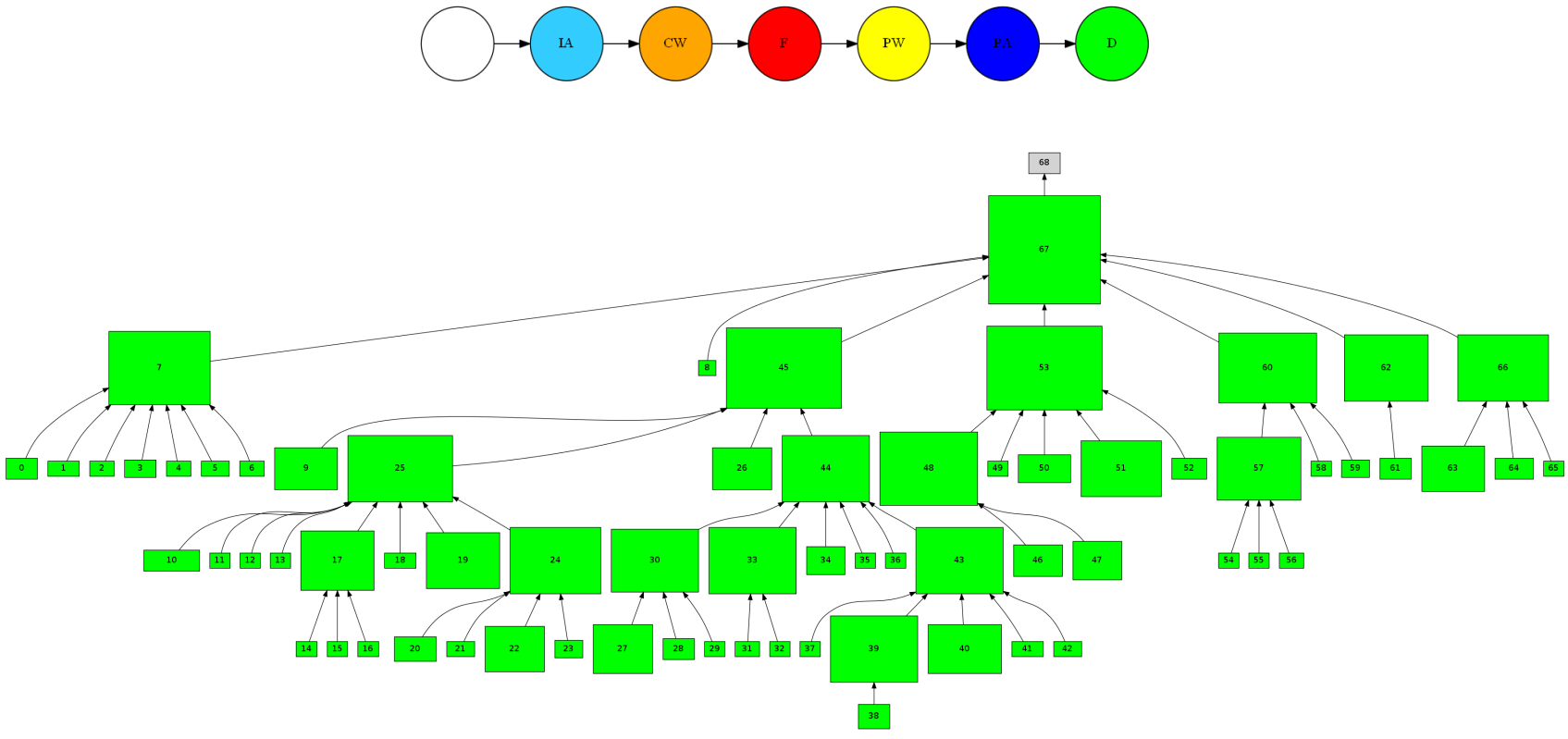
Front Scheduling



Front Scheduling

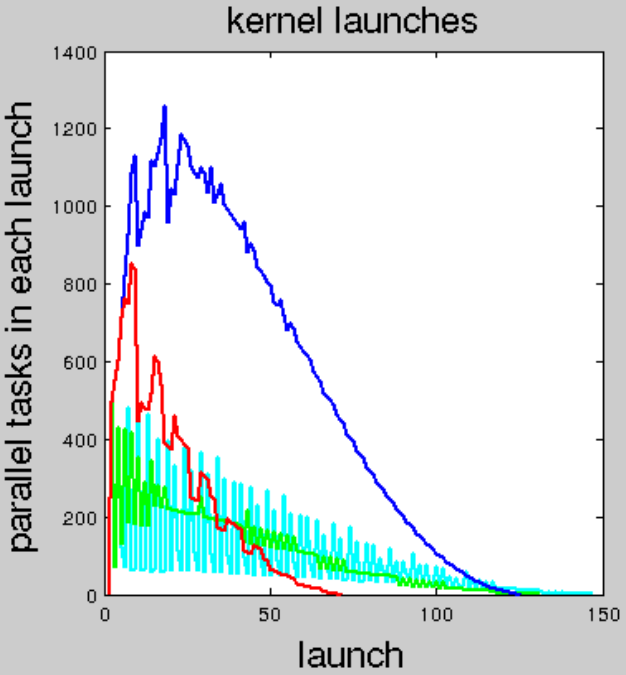


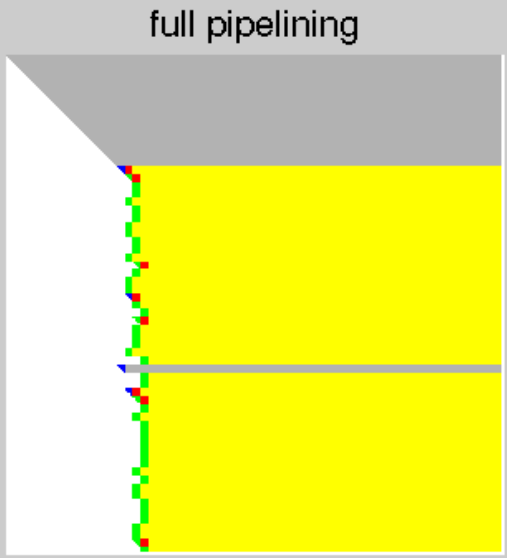
Front Scheduling

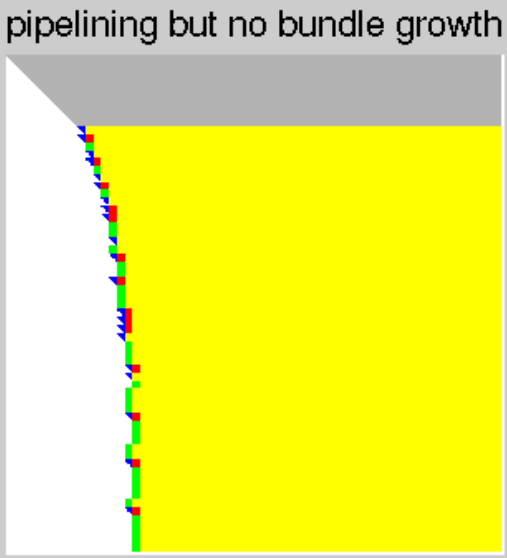


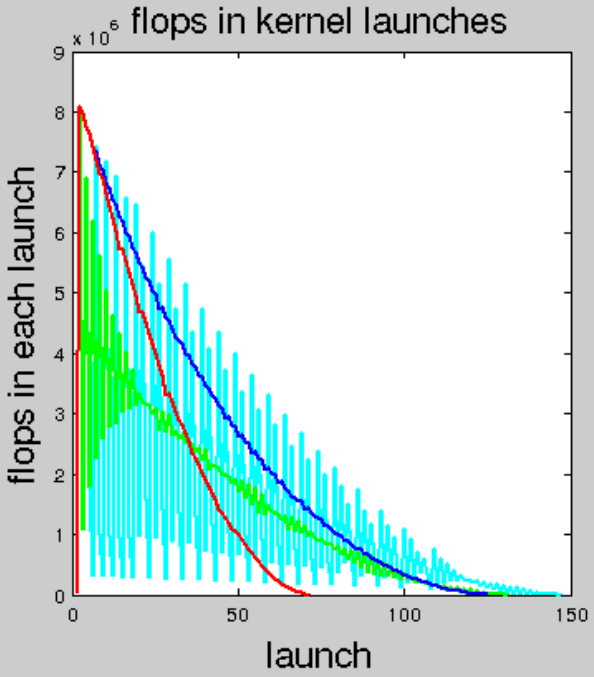
Pipelined Dense QR Factorization

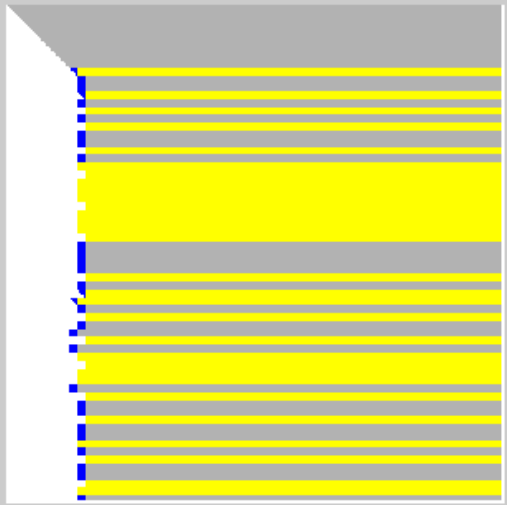
- **Results are coming soon..**
- **Our existing implementation is not efficient**
  - About half as fast as existing methods
  - Not exploiting enough parallelism!
  - We eliminate column blocks in order from left to right
  - GPU workload and throughput "thrashes," negating any gains from overlapping communication with computation
  - See BatchQR in CP7 for more details

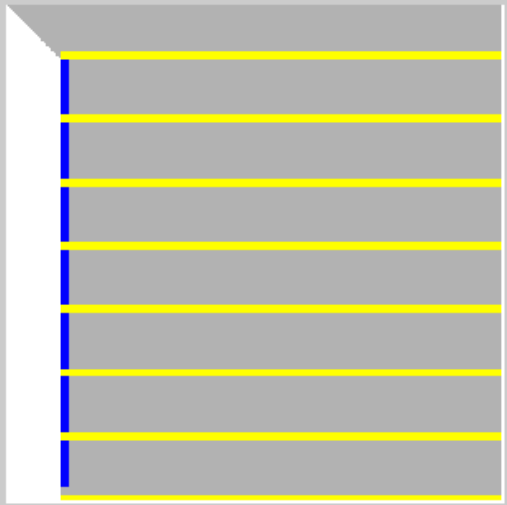

kernel launches
parallel tasks in each launch
launch
0
200
400
600
800
1000
1200
1400
0
50
100
150


full pipelining


pipelining but no bundle growth


flops in kernel launches
x 10^6
flops in each launch
launch
0
1
2
3
4
5
6
7
8
9
0
50
100
150


no pipelining


no pipelining, no lookahead

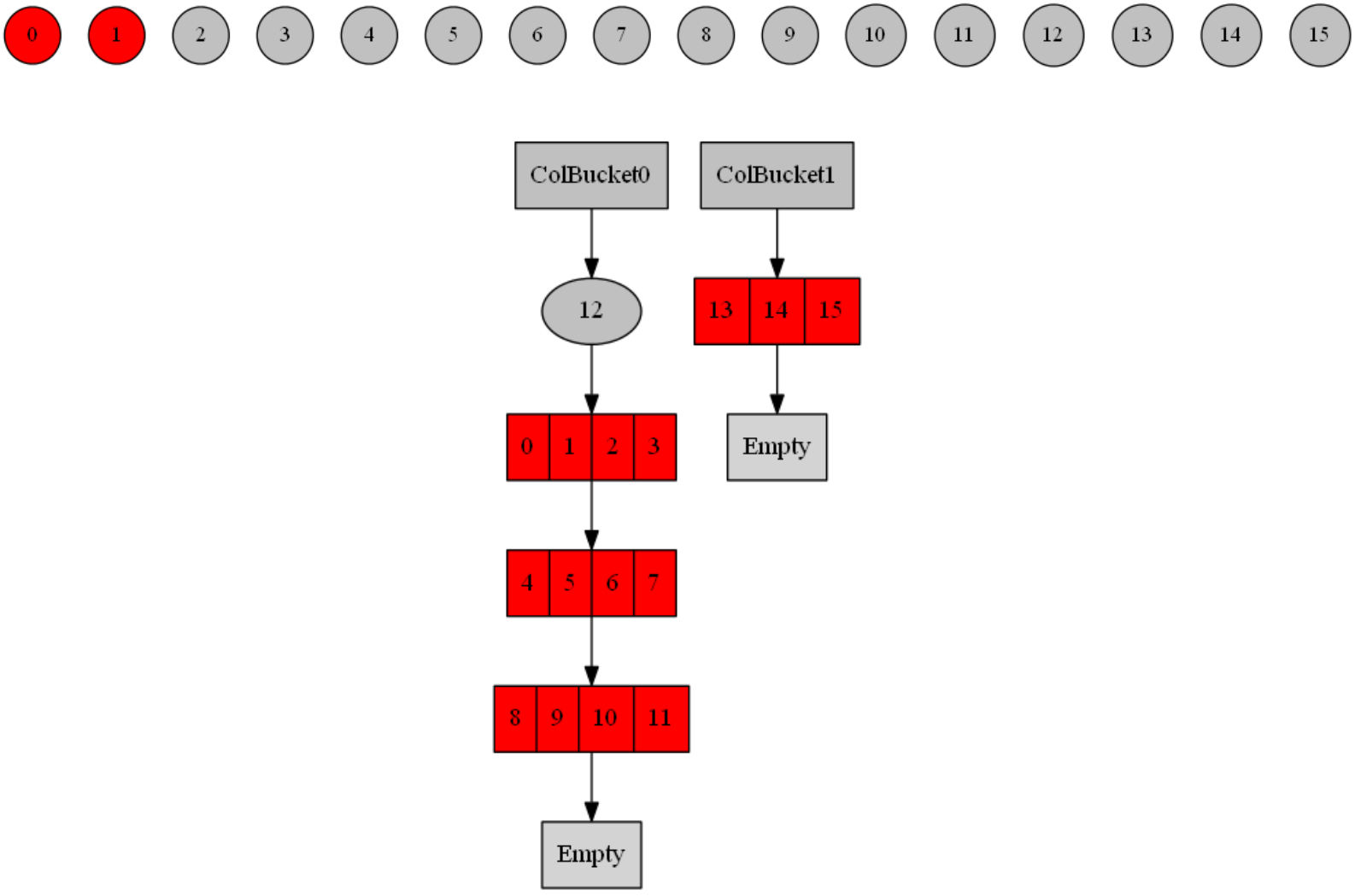
Pipelined Dense QR Factorization

- **Our current development thrust is promising**
  - Tile each front
  - Track each block row's leftmost tile
  - Initialize the tracking using the Staircase
  - Bundle tiles into packets of parallel tasks
  - Launch each GPU kernel with the most work we can do before we need to synchronize
  - Pipeline the block Householder applies with an annihilate

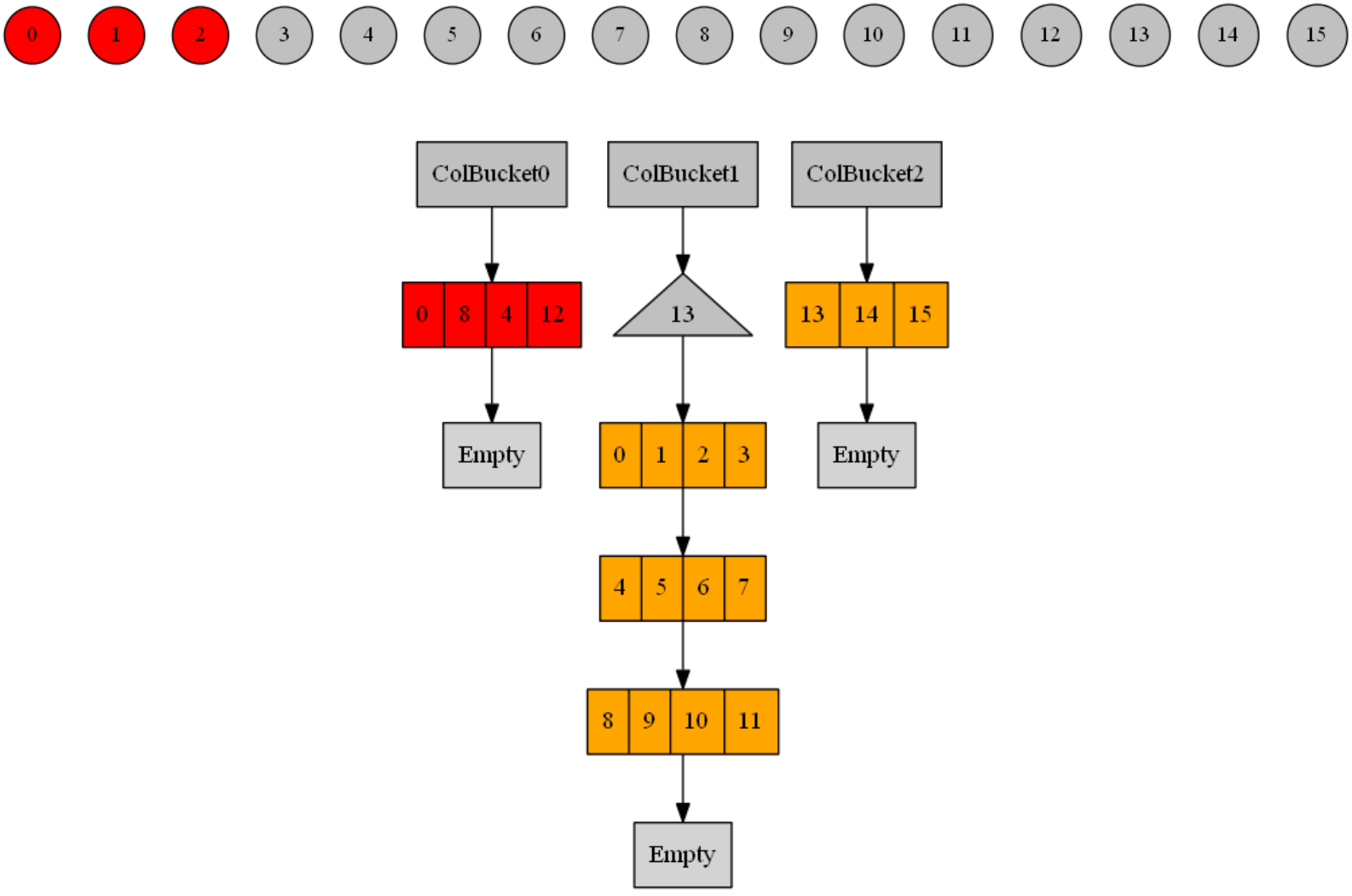
Pipelined Dense QR Factorization

- **What does this look like?**

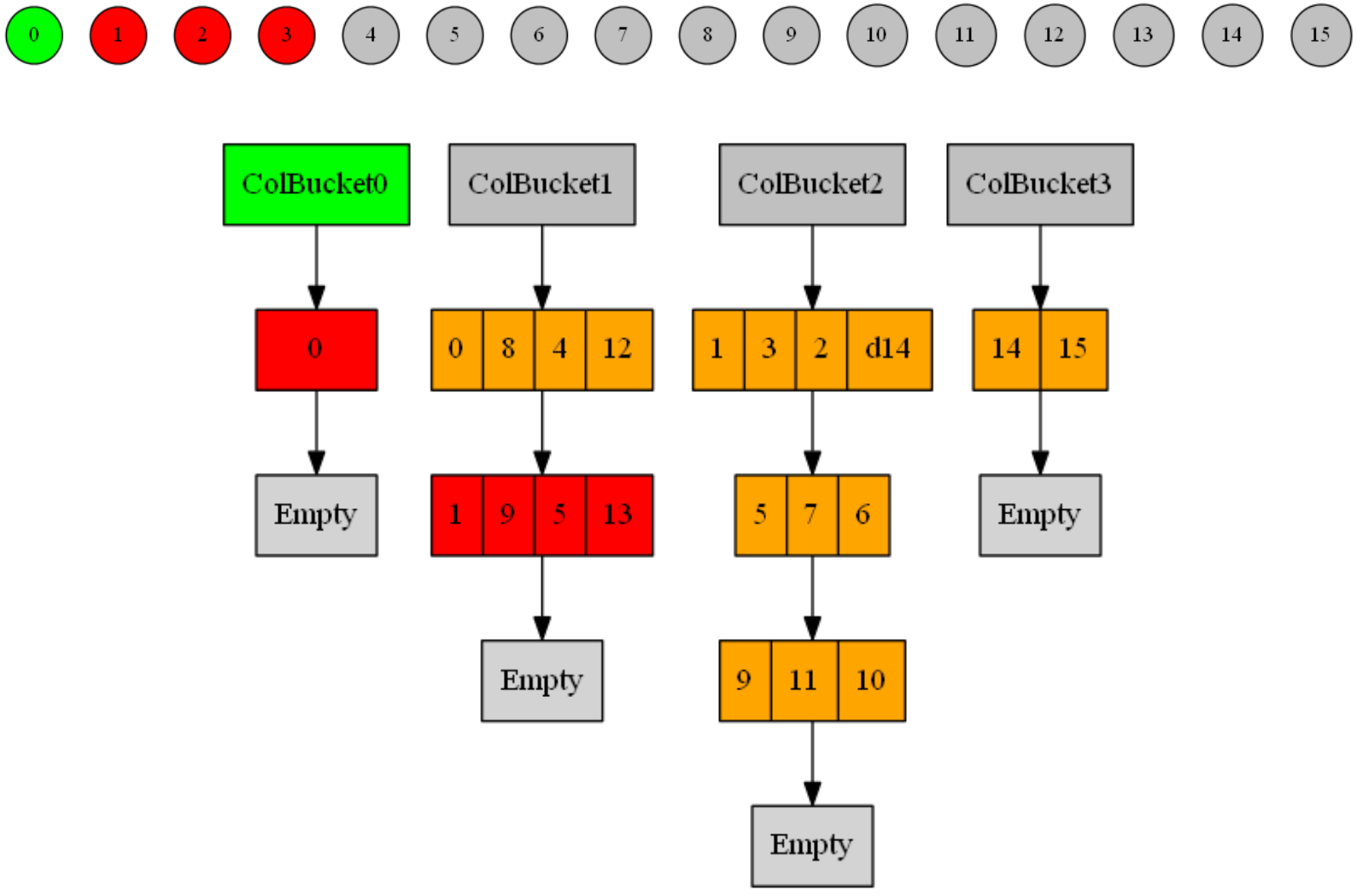
Pipelined Factorization



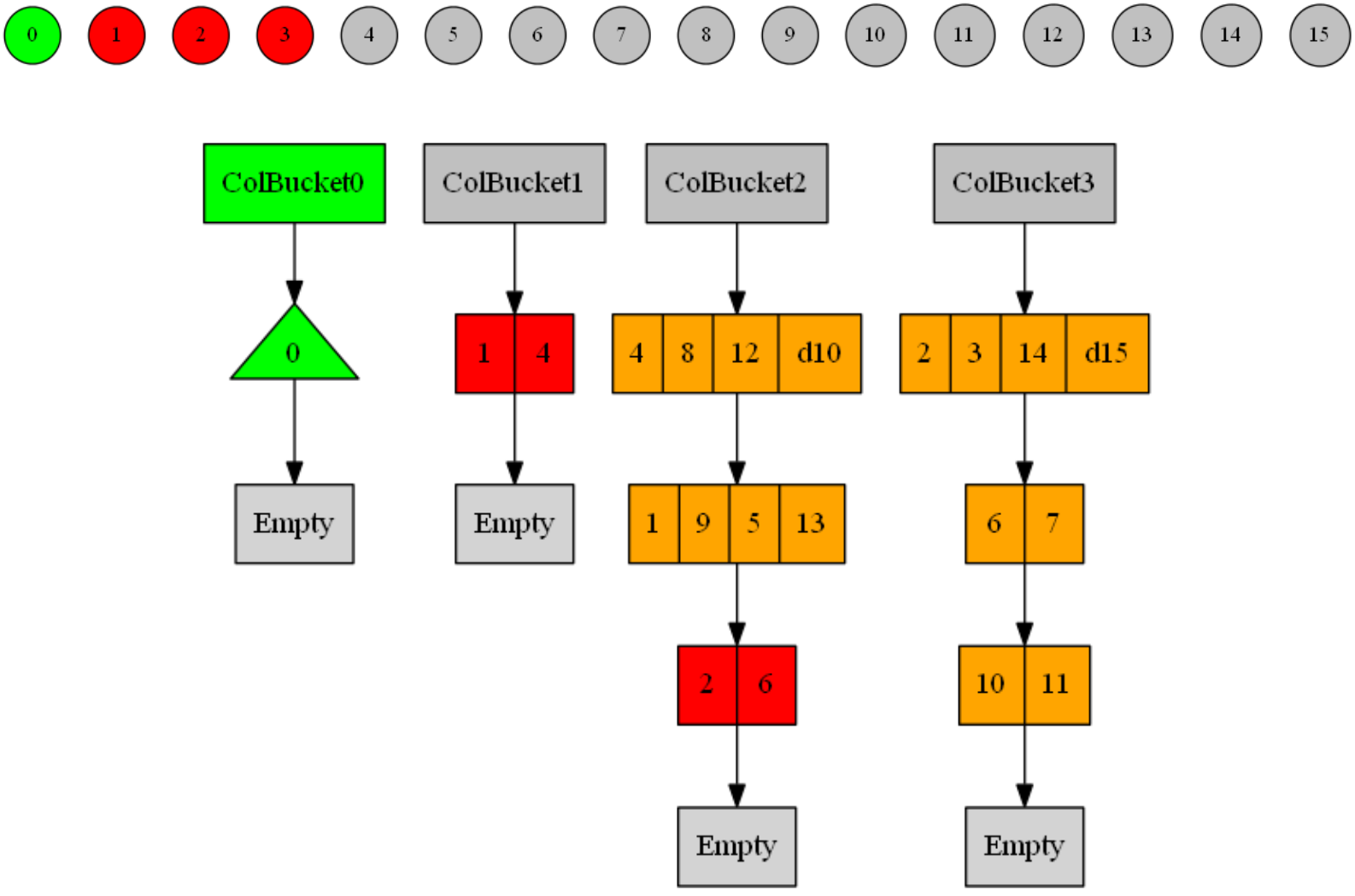
Pipelined Factorization



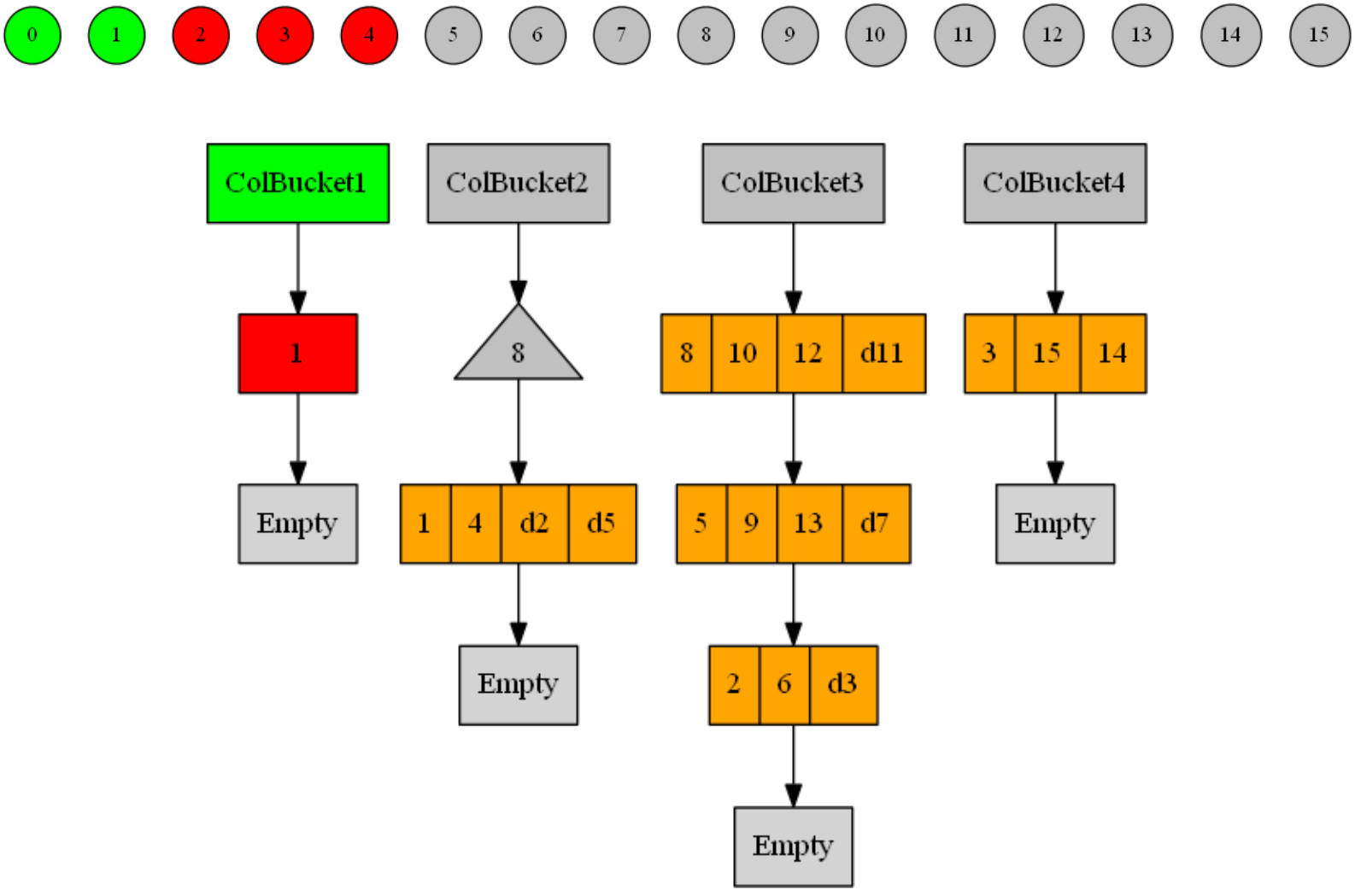
Pipelined Factorization



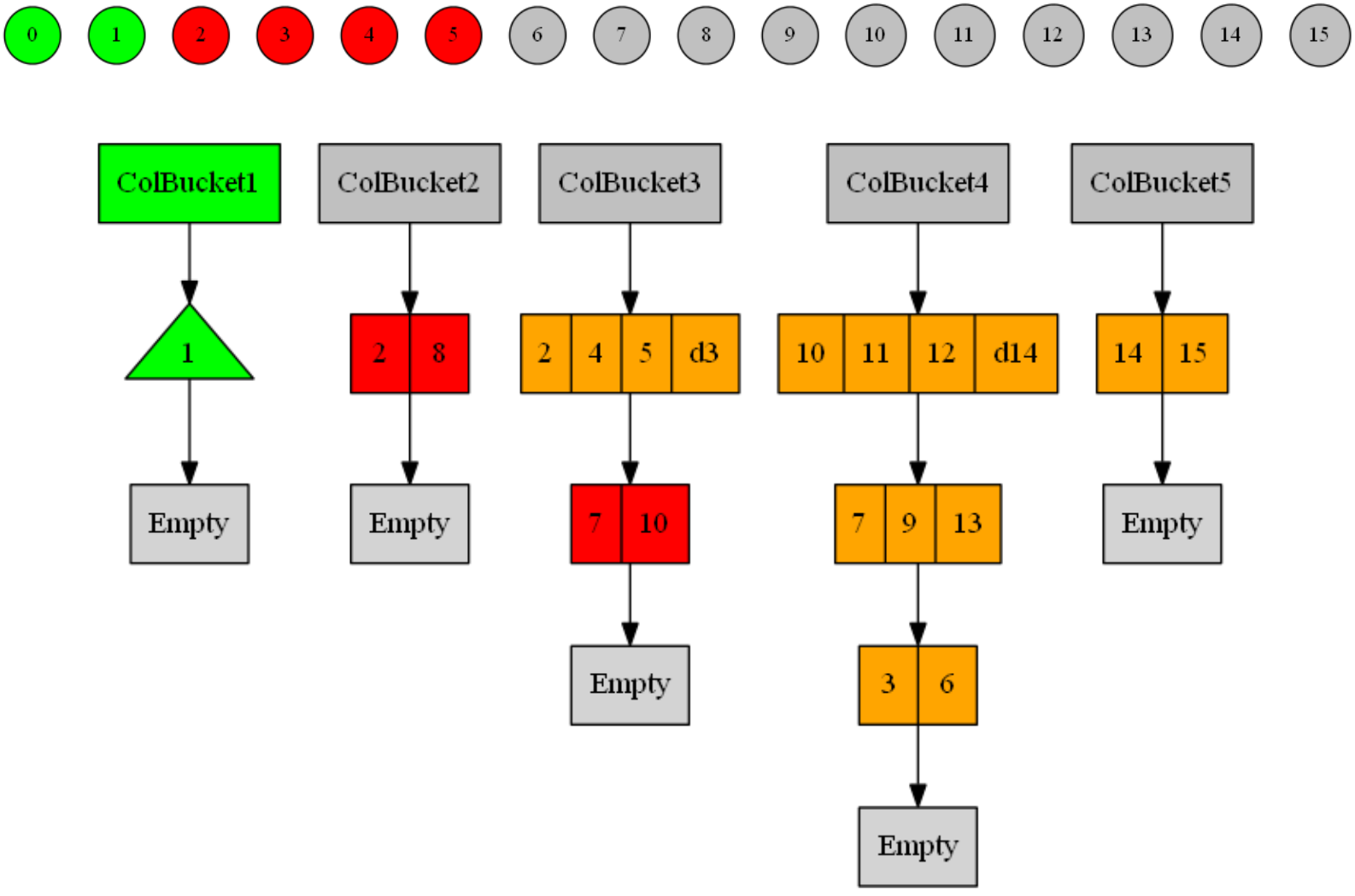
Pipelined Factorization



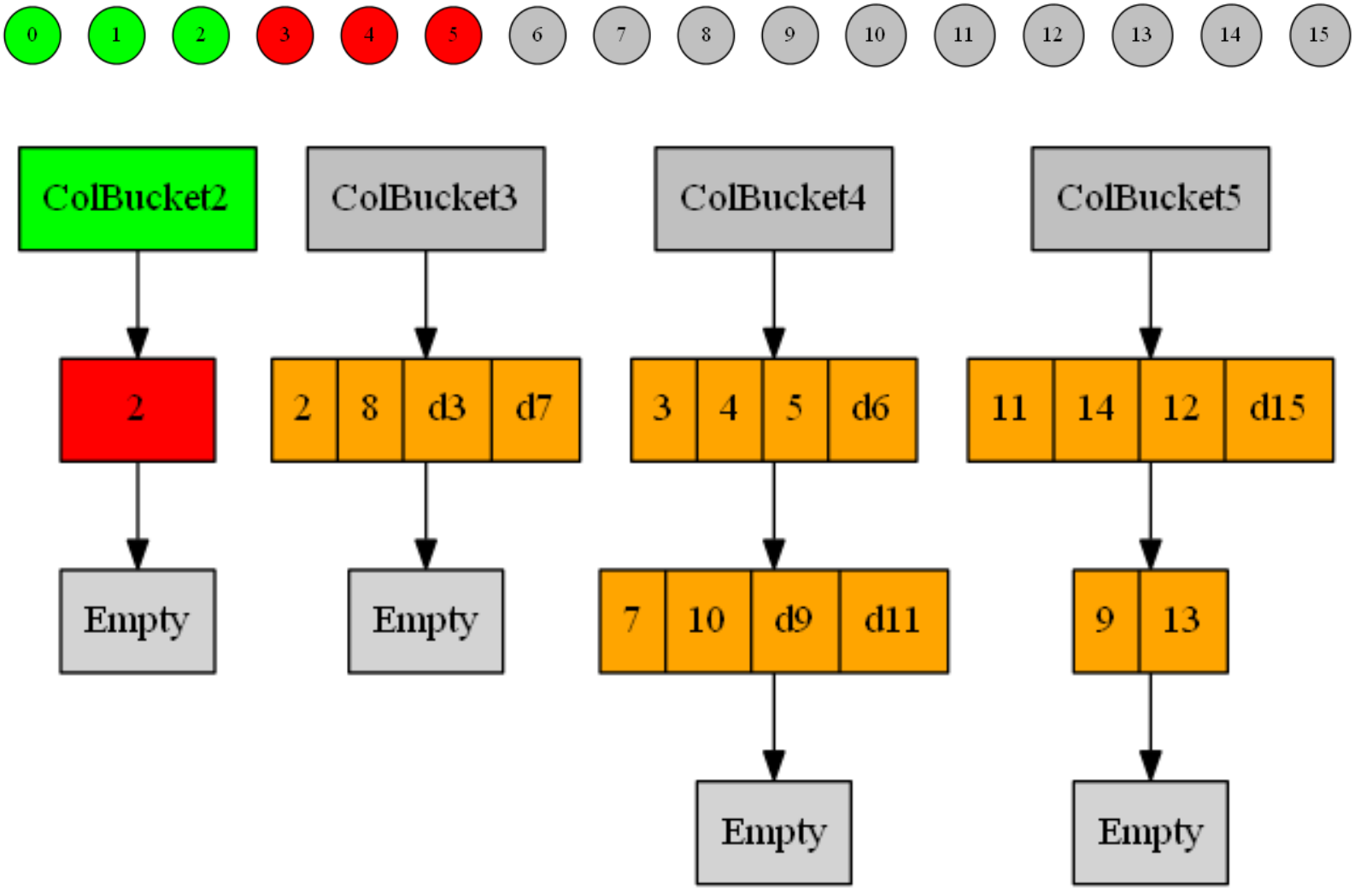
Pipelined Factorization



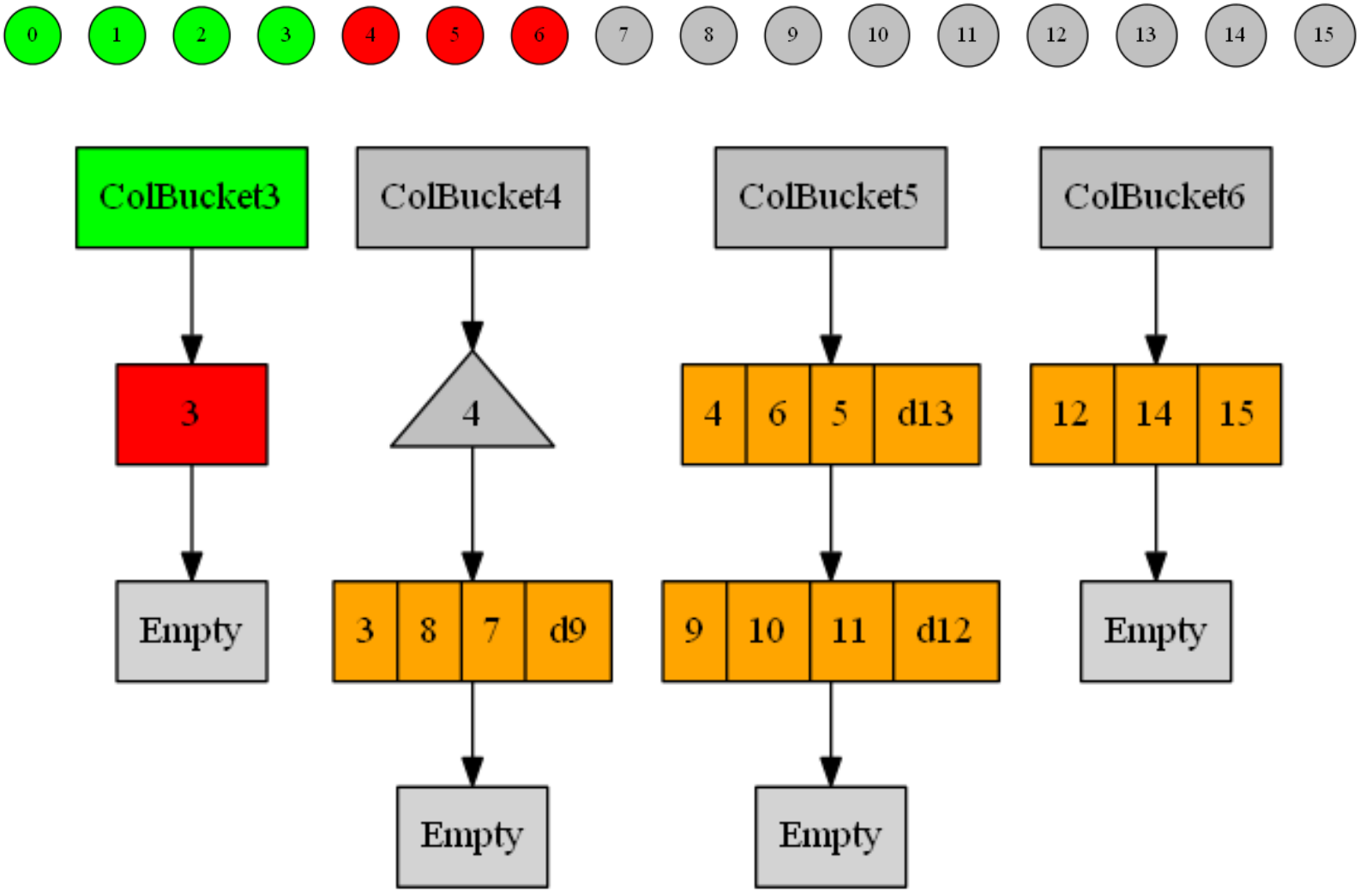
Pipelined Factorization



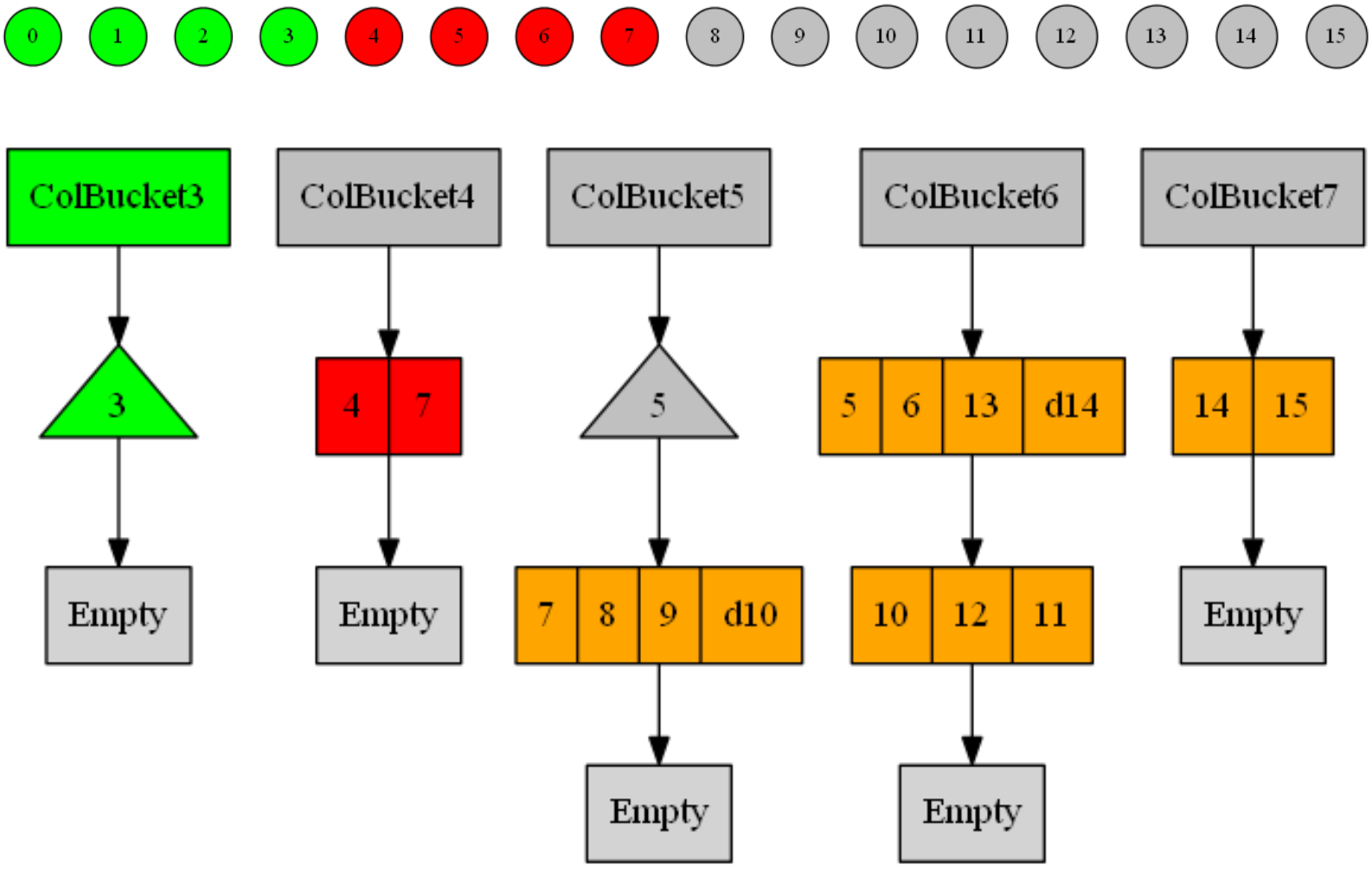
Pipelined Factorization



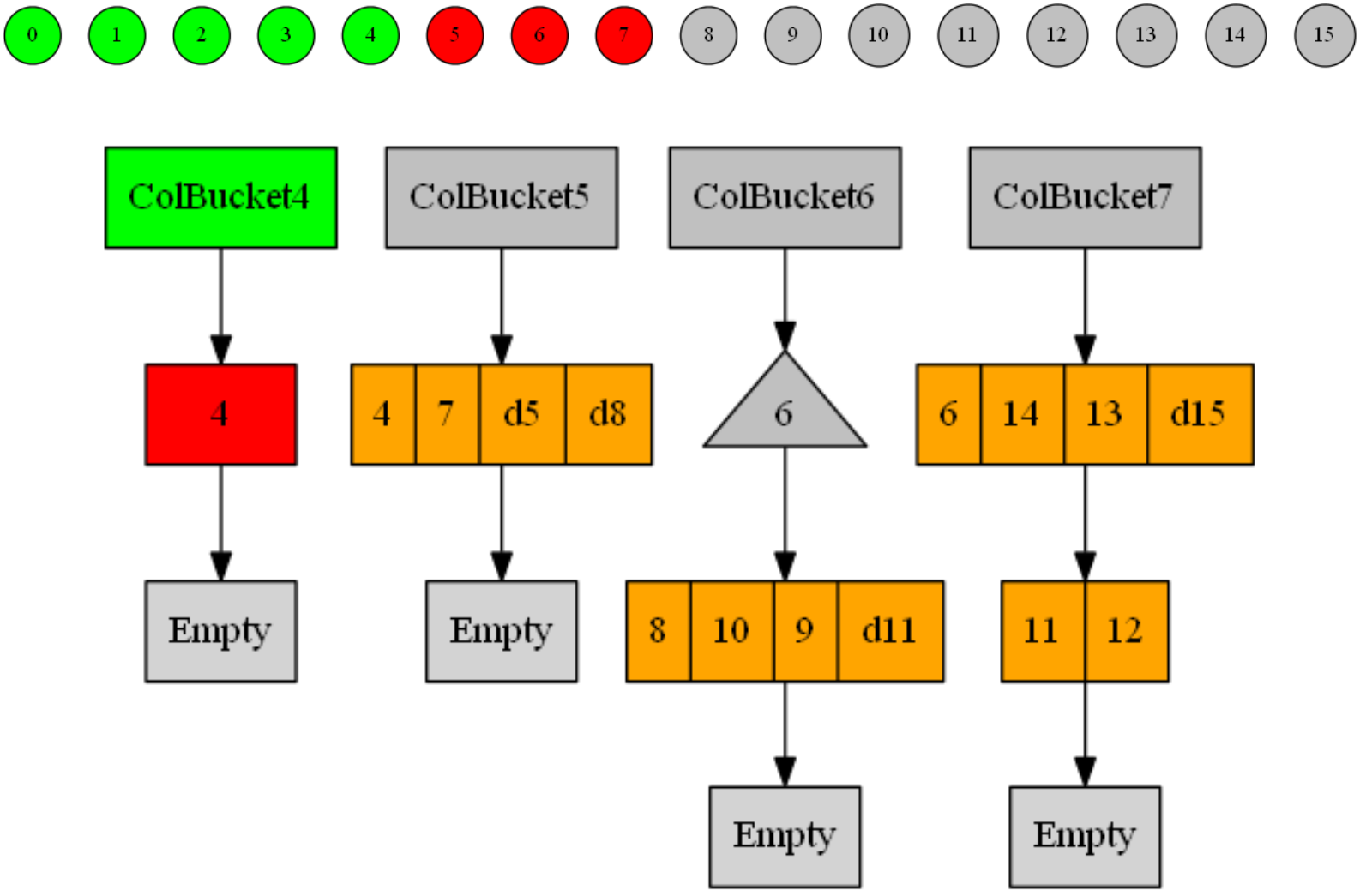
Pipelined Factorization



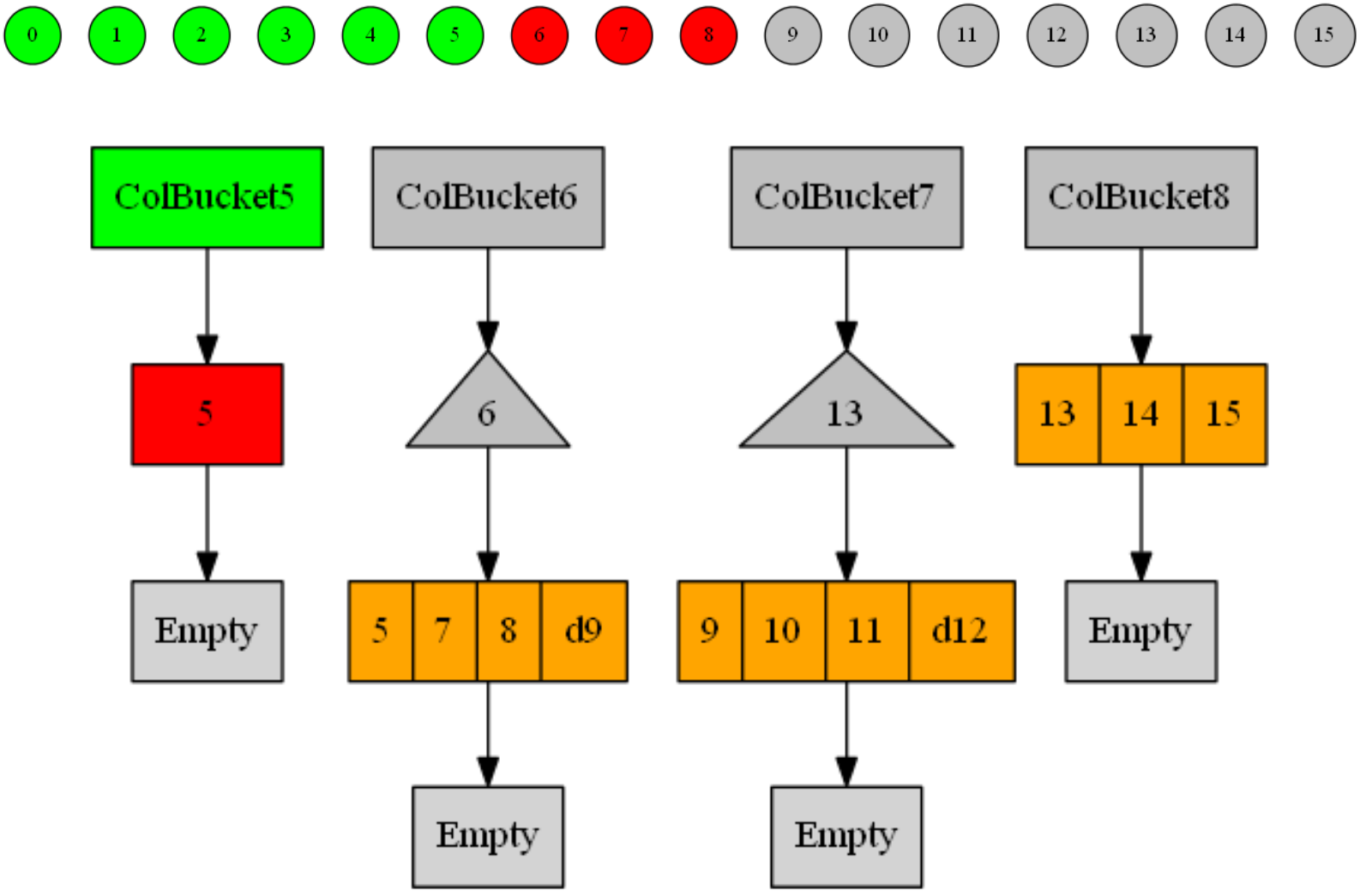
Pipelined Factorization



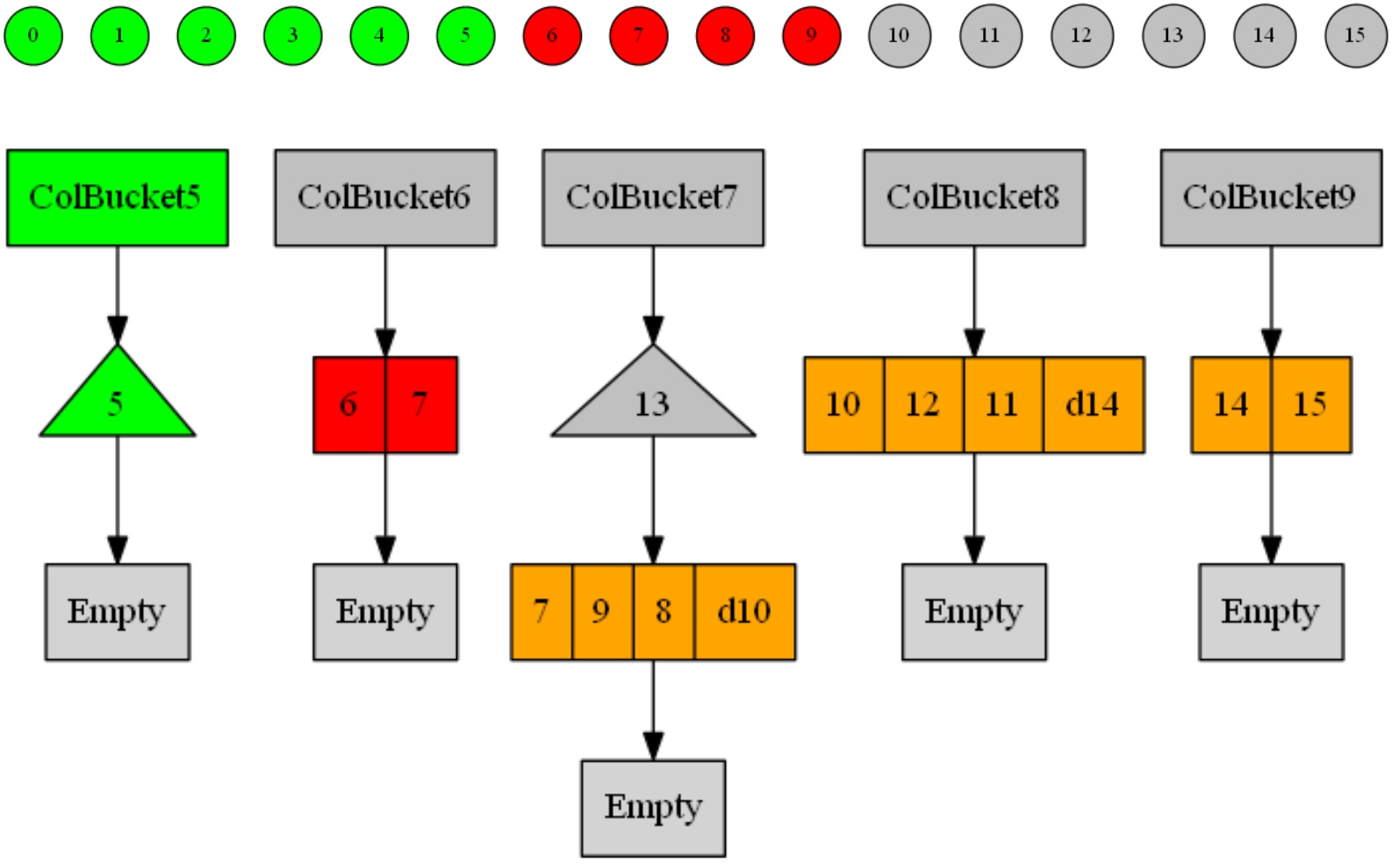
Pipelined Factorization



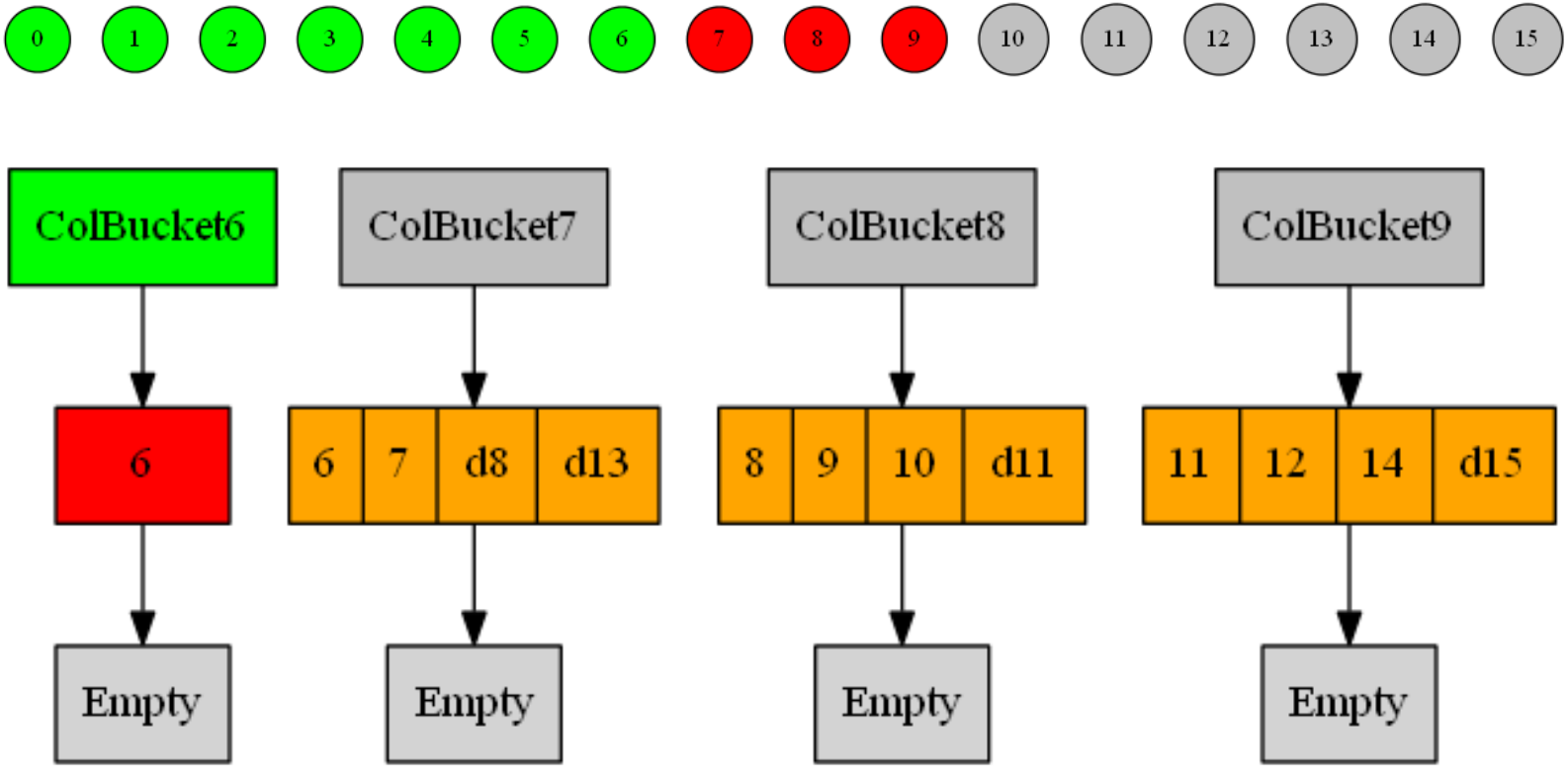
Pipelined Factorization



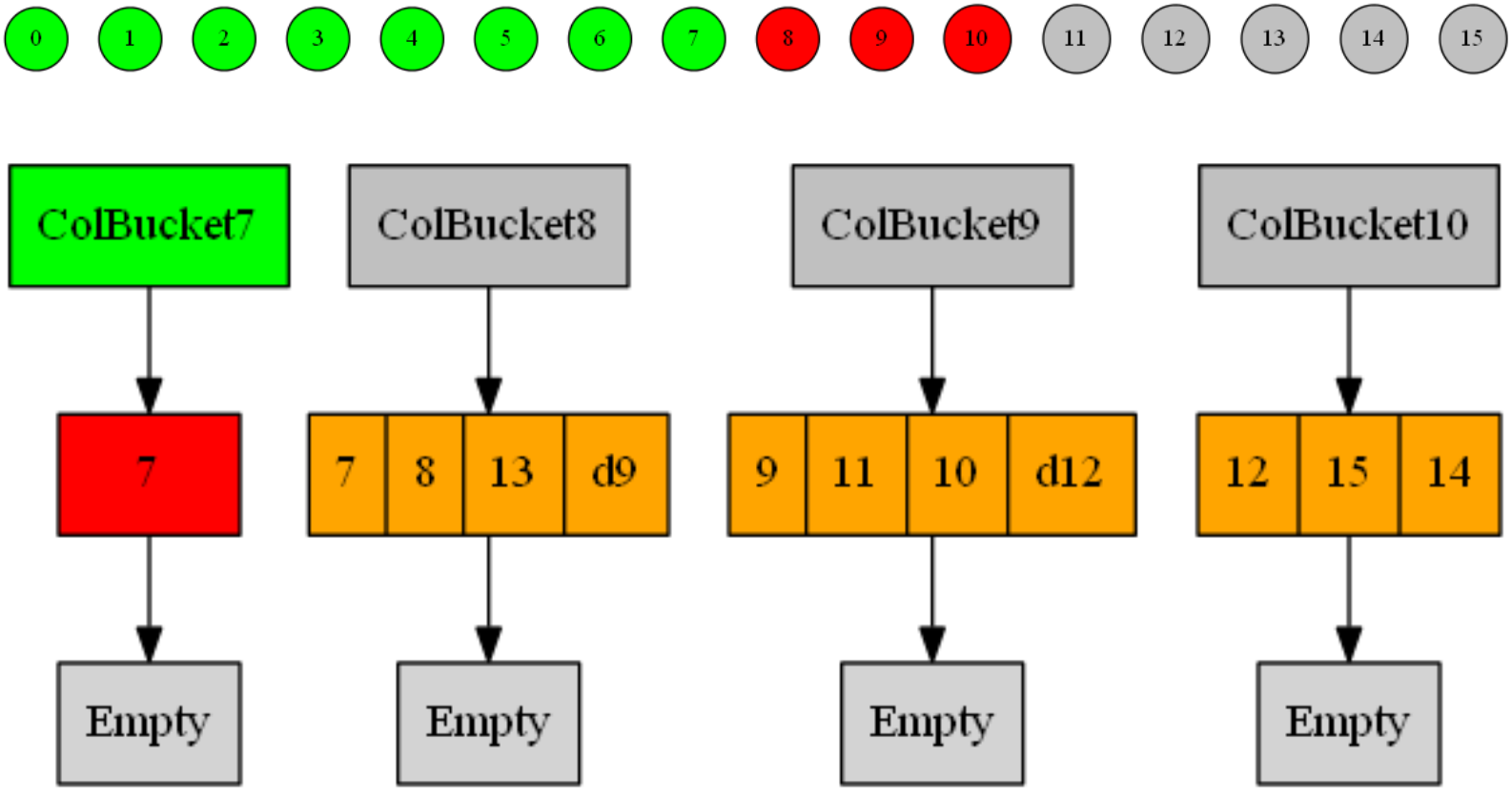
Pipelined Factorization



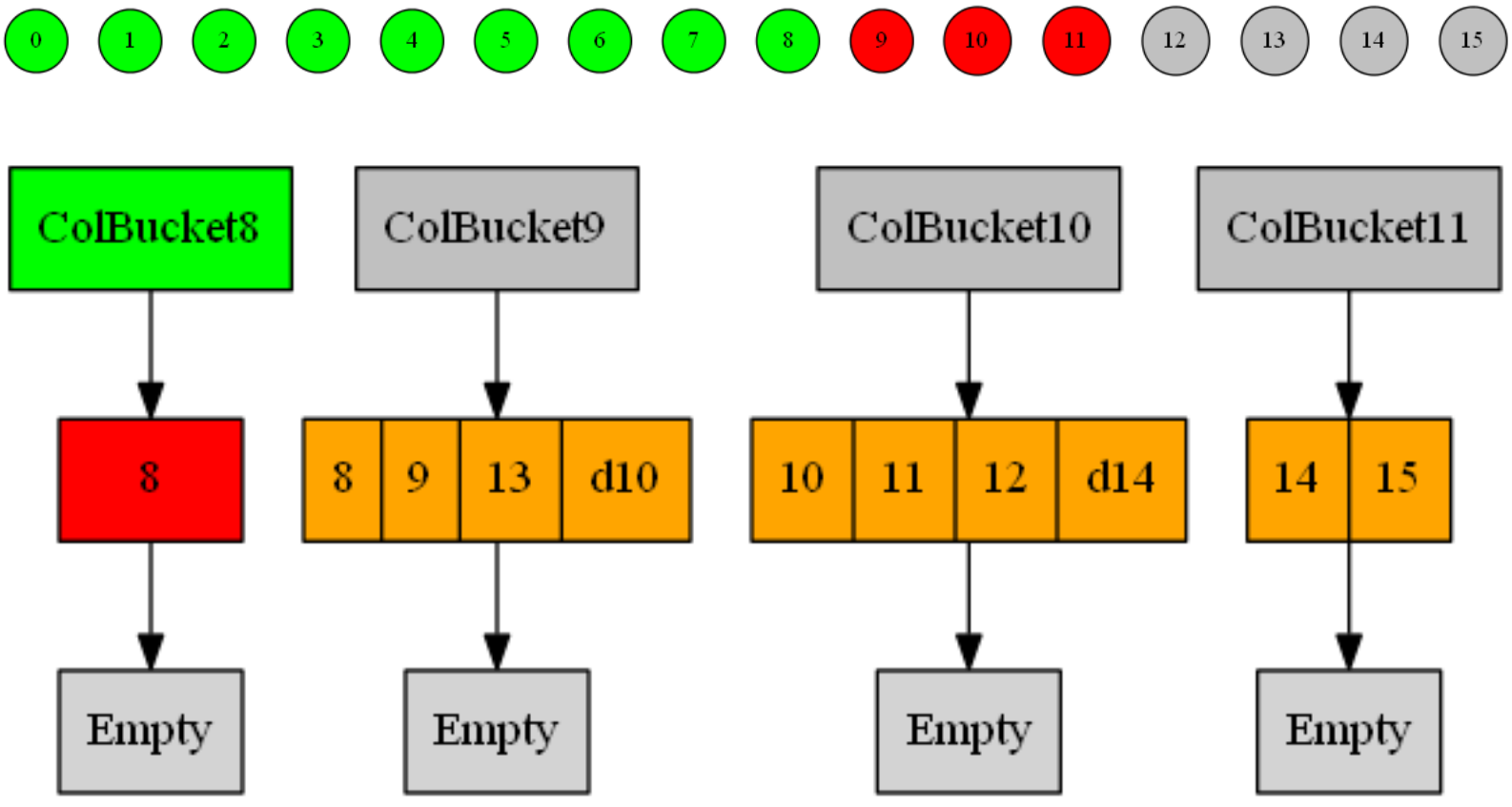
Pipelined Factorization



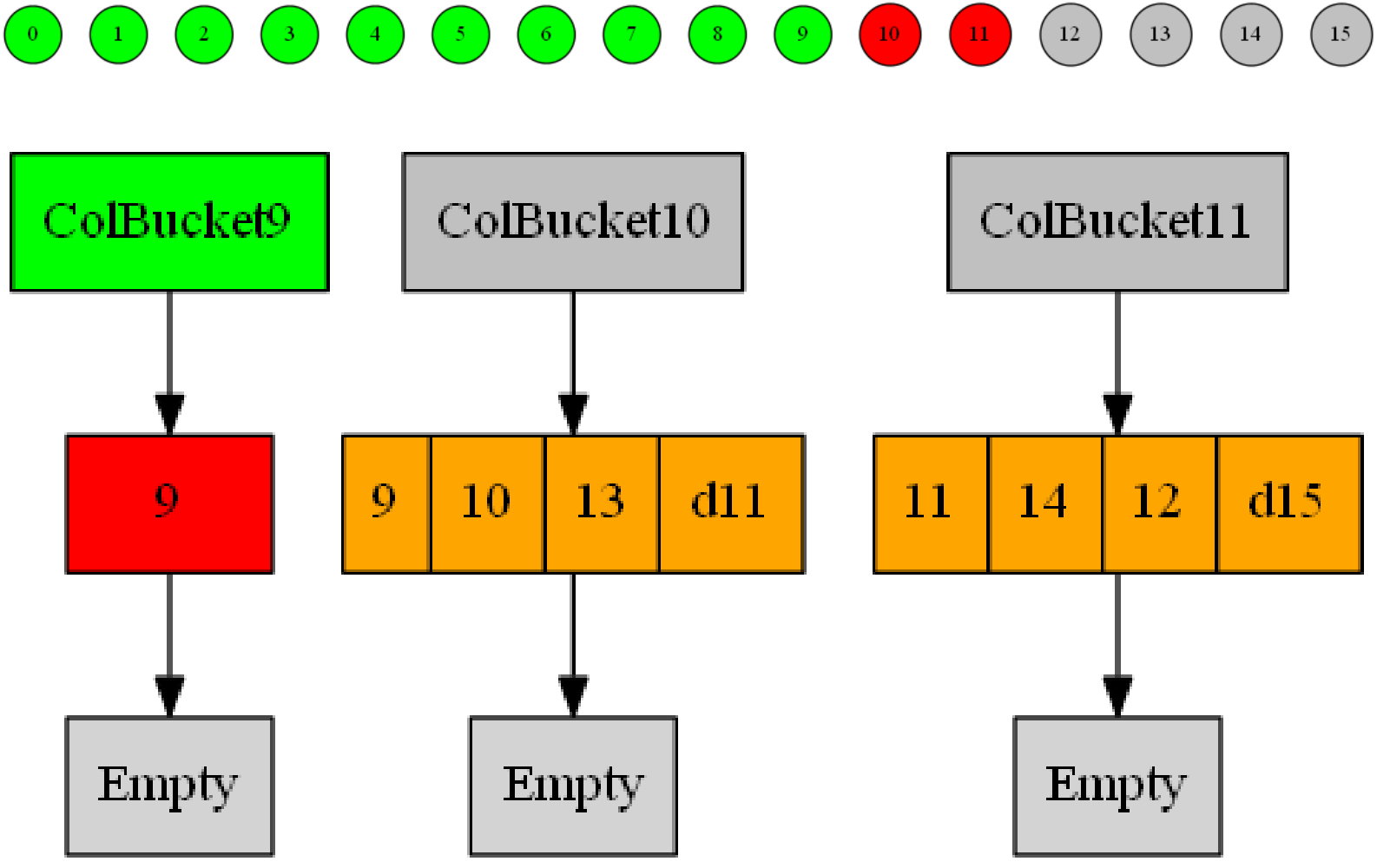
Pipelined Factorization



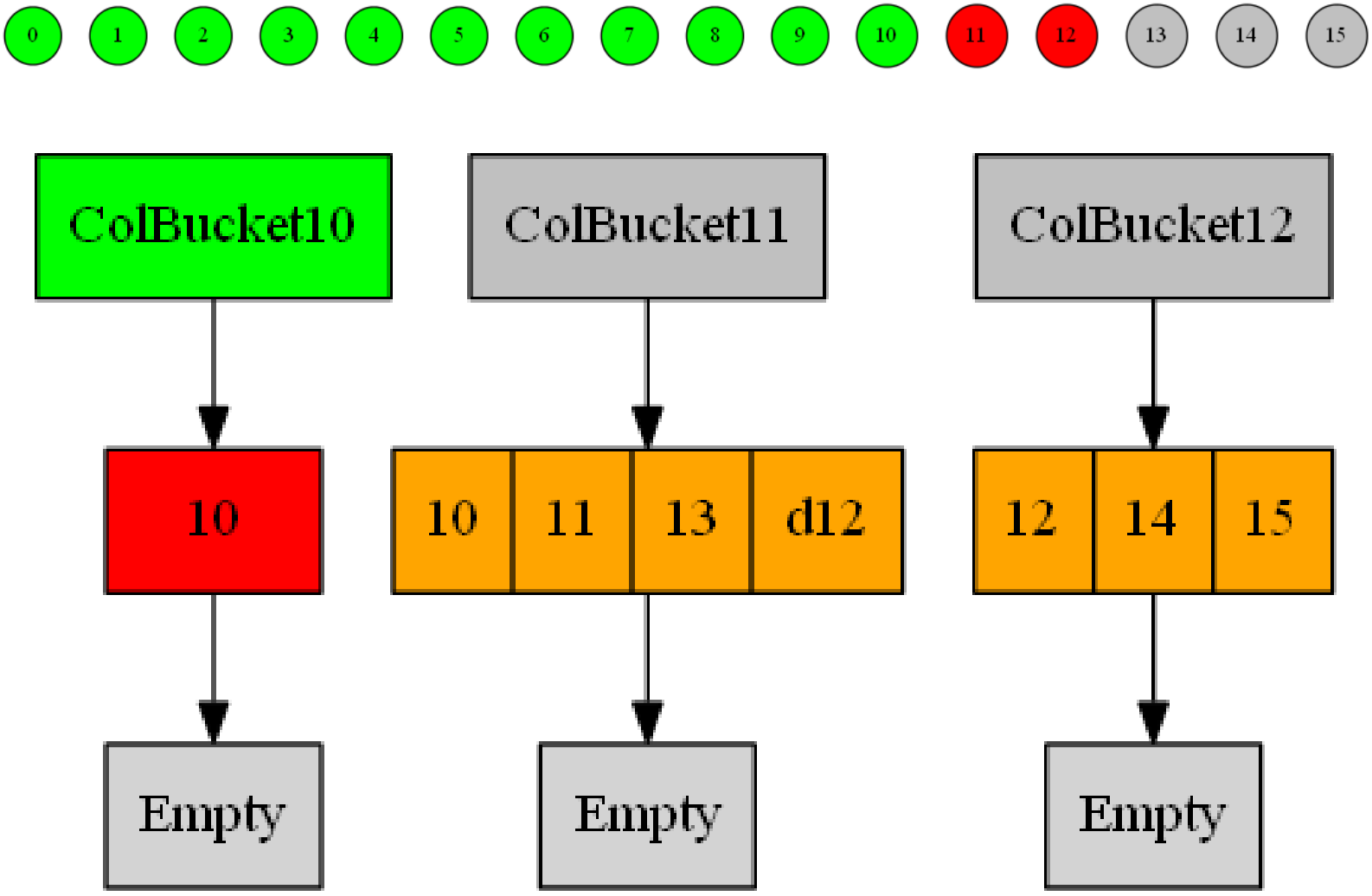
Pipelined Factorization



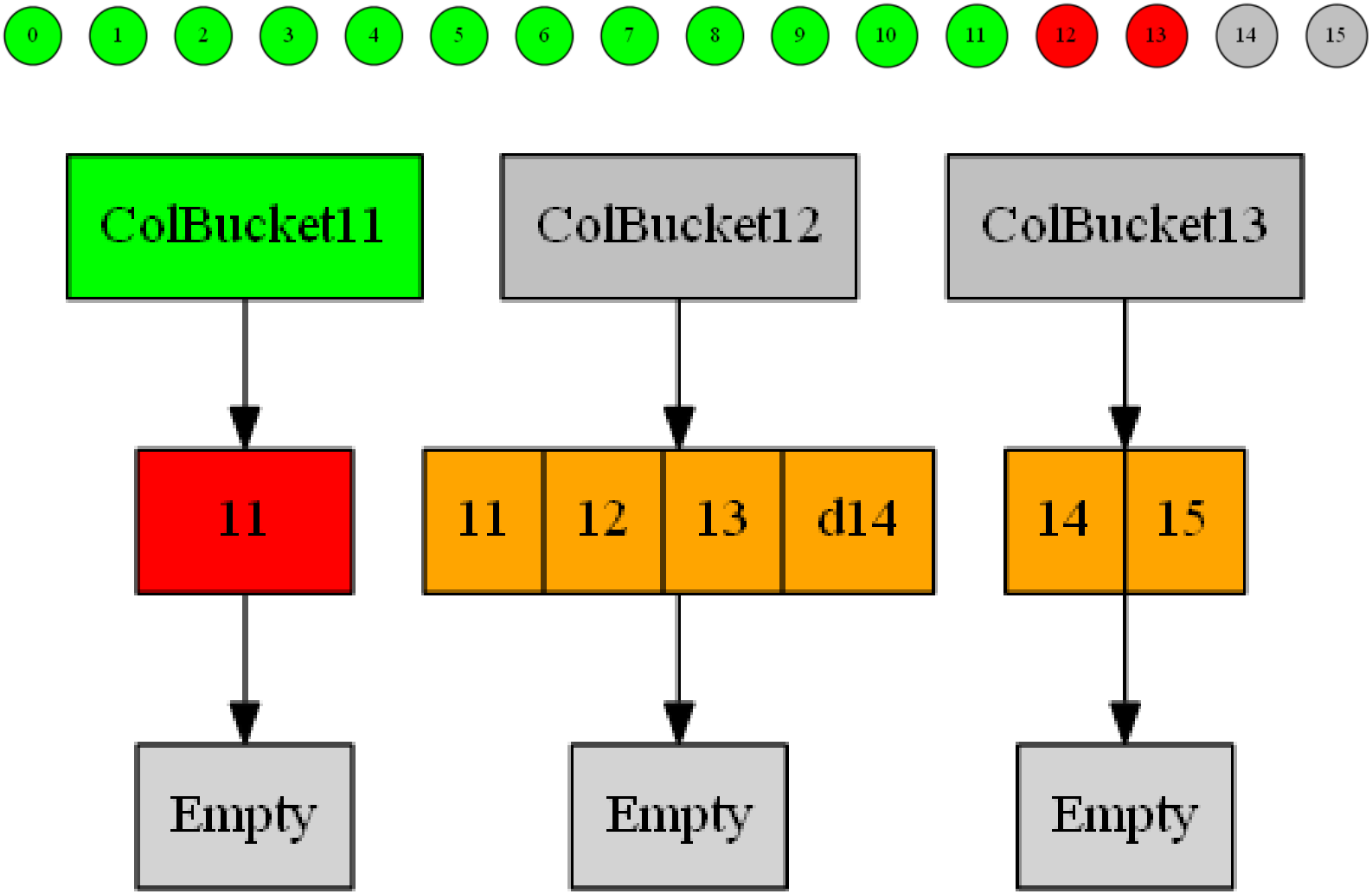
Pipelined Factorization



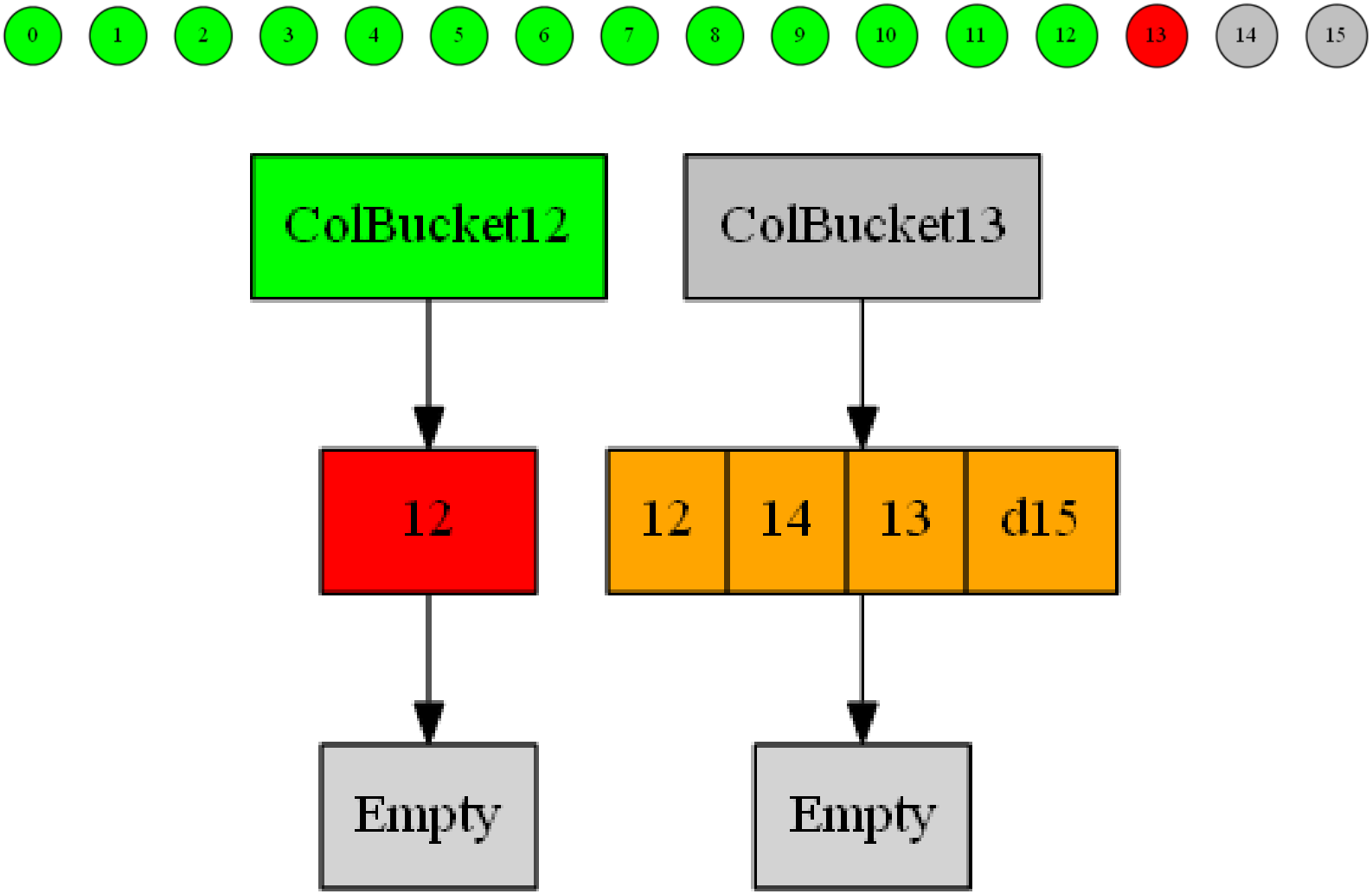
Pipelined Factorization



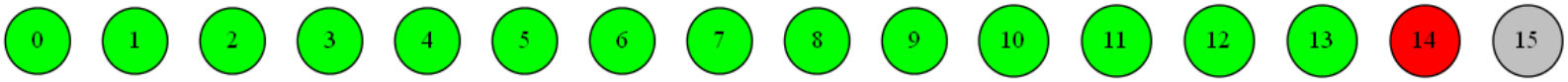
Pipelined Factorization

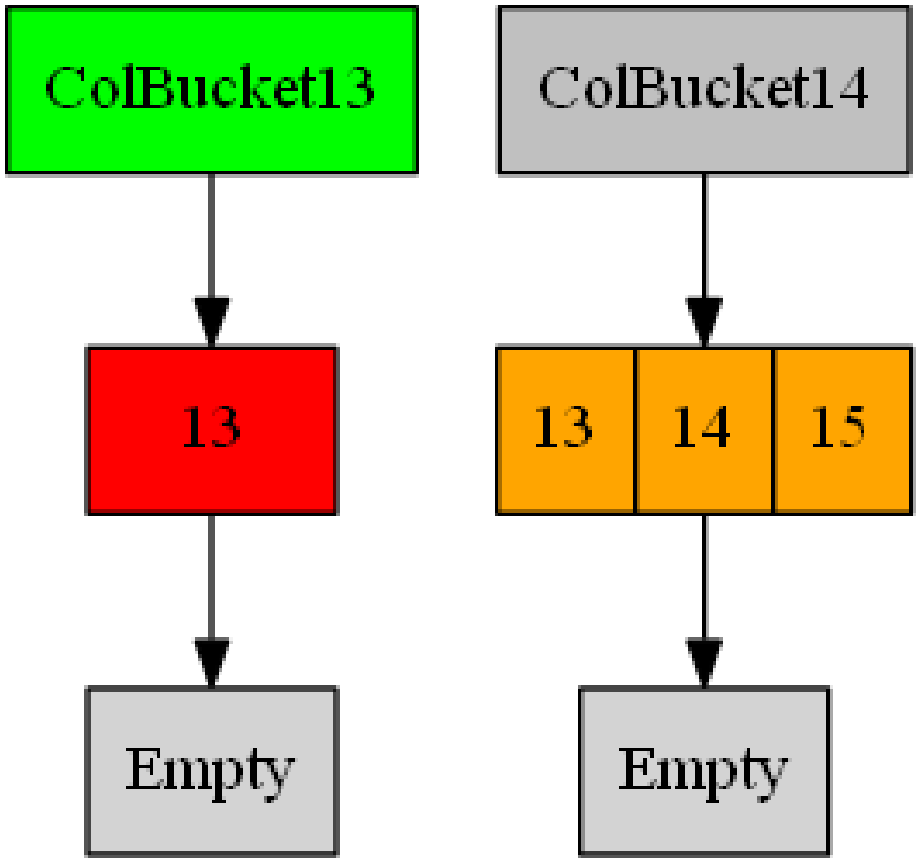


Pipelined Factorization

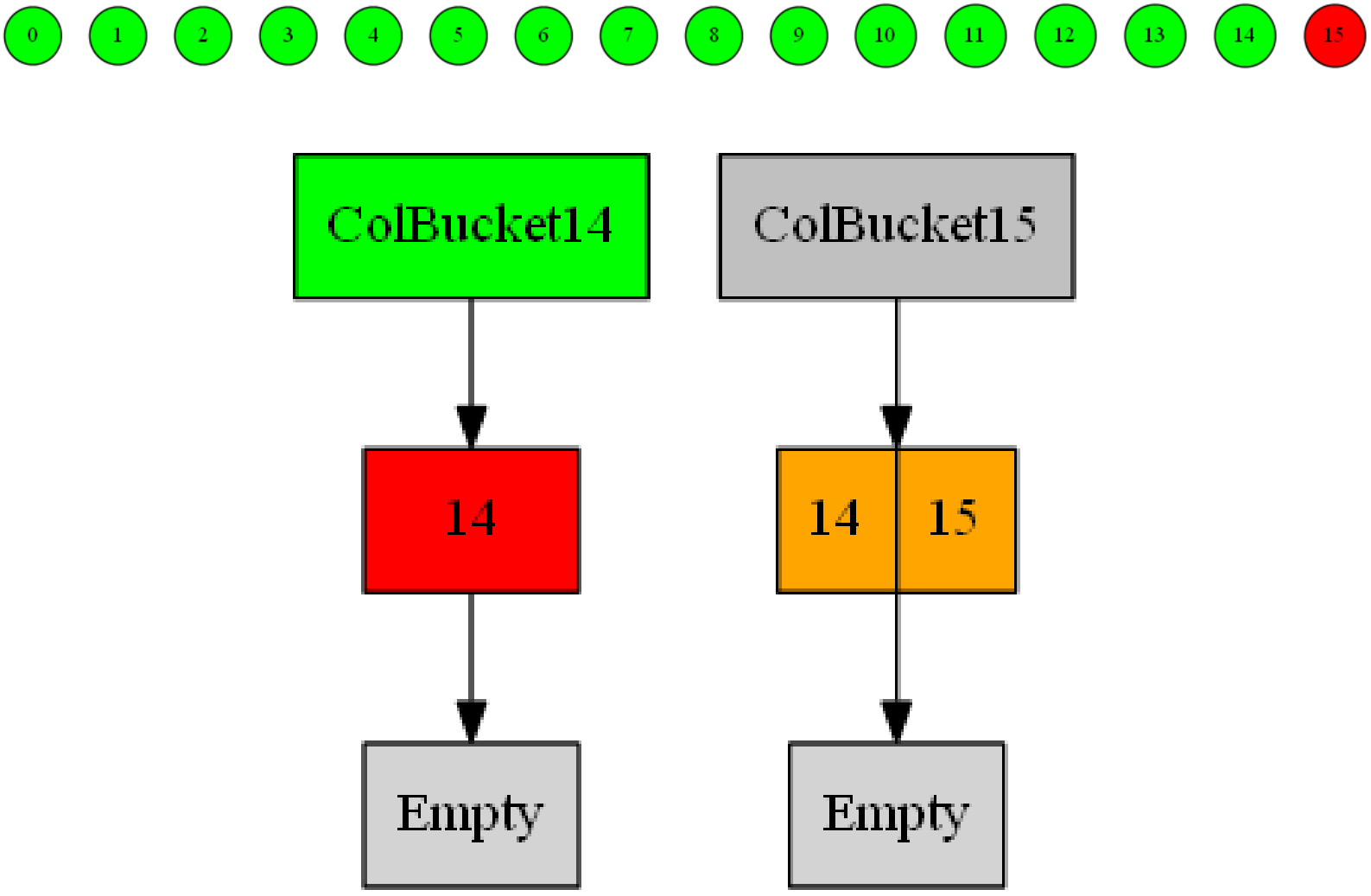


Pipelined Factorization

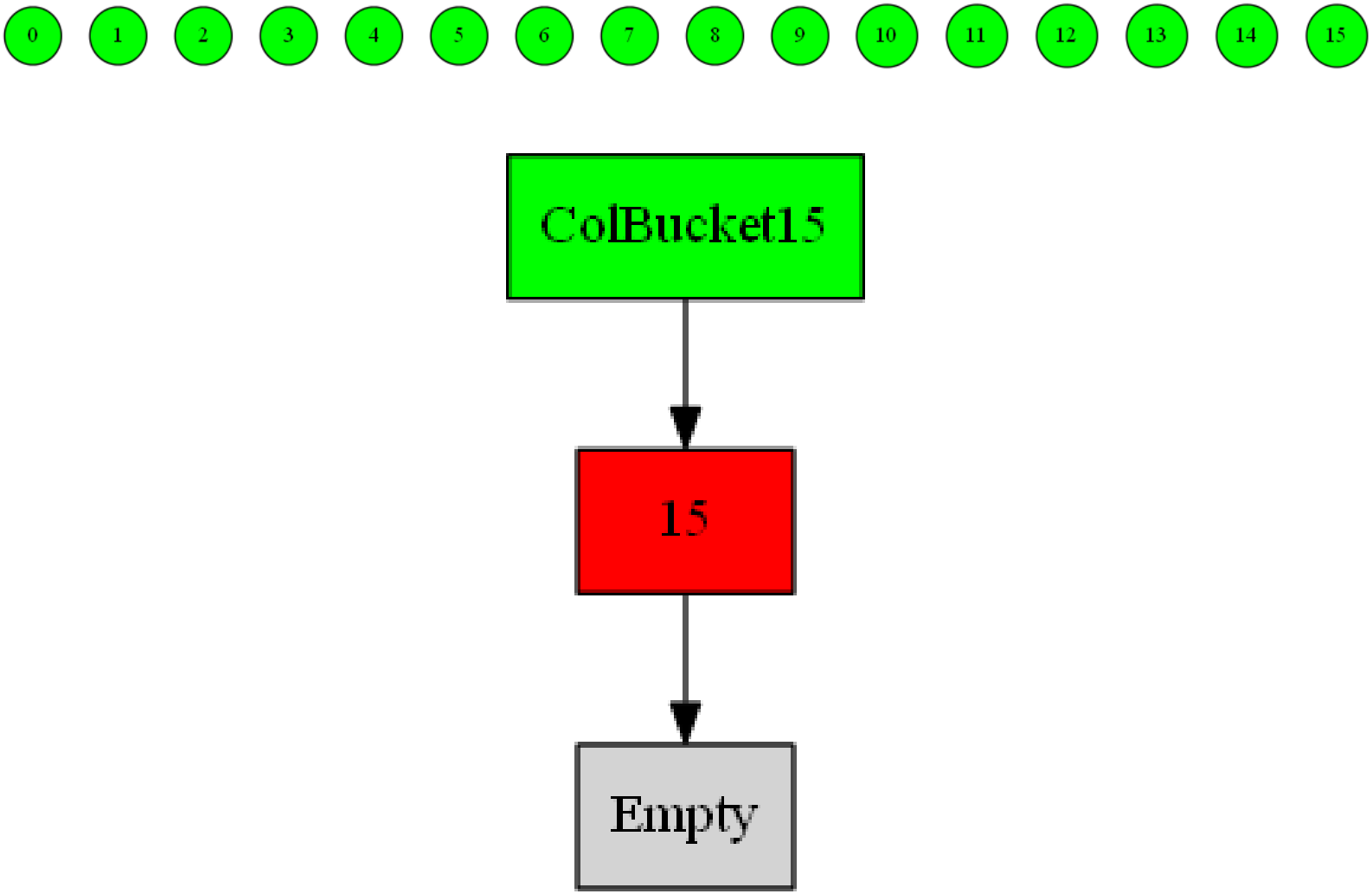




Pipelined Factorization



Pipelined Factorization



Future Work

- Add rank detection as a postprocessing step
- Consider bringing back Q in addition to R
- Decompose column elimination tree into subtrees to combat the GPU memory wall
- Subtree parallelism in a multi-GPU configuration
- Handle fronts that don't fit in GPU memory (MAGMA-style)
- Questions?