CUDA and GPU programming: an opportunity for scientists

Nathan Clisby
MASCOS, University of Melbourne

May 27, 2009

Why learn GPU programming?

What problems are appropriate for the GPU?

Case study: backtracking

Where to next?

Hardware

- Moore's Law:
  - clock speeds no longer rapidly increasing.
  - massively multicore architectures circumvent this problem.
- Change of paradigm from single threaded applications to parallel and stream processing.
- Double precision on GPUs now supported.
- Major players NVIDIA, AMD, Intel, all serious about GPGPU (general purpose GPU programming).

Software

- CUDA: not a high level language, but an appropriate level for the hardware. Much easier to learn than graphics APIs such as OpenGL.
- For easy problems, CUDA is easy to apply (many problems are easy).
- OpenCL: similar to CUDA, code will run on variety of hardware.
- Details may change, but broad brush-strokes and techniques will remain same. Skills developed now won't go to waste.

- Improving hardware and software make GPU computing very attractive, large benefits for not too large time investment.
- Discontinuity in Moore's Law - one-off opportunity (until quantum computing!).
- Grab low hanging fruit while you can (implement embarrassingly parallel algorithms).
- As time goes by, more applications will be ported to GPU hardware.
- *Desktop supercomputer*: workstation with Tesla GPUs can be as powerful as a CPU cluster.
- Soon, CPU / GPU clusters will be common.
- If current trends continue, will become widespread paradigm in high performance computing.

- Easiest problems: embarrassingly parallel, e.g.
  - Computer graphics.
  - Simulation of large physical systems via molecular dynamics.
  - Large graphs / networks.
  - Numerical integration.
  - Many more.
- To some extent, most applications can probably benefit from GPU . . .

- Easiest problems: embarrassingly parallel, e.g.
  - Computer graphics.
  - Simulation of large physical systems via molecular dynamics.
  - Large graphs / networks.
  - Numerical integration.
  - Many more.
- To some extent, most applications can probably benefit from GPU . . .

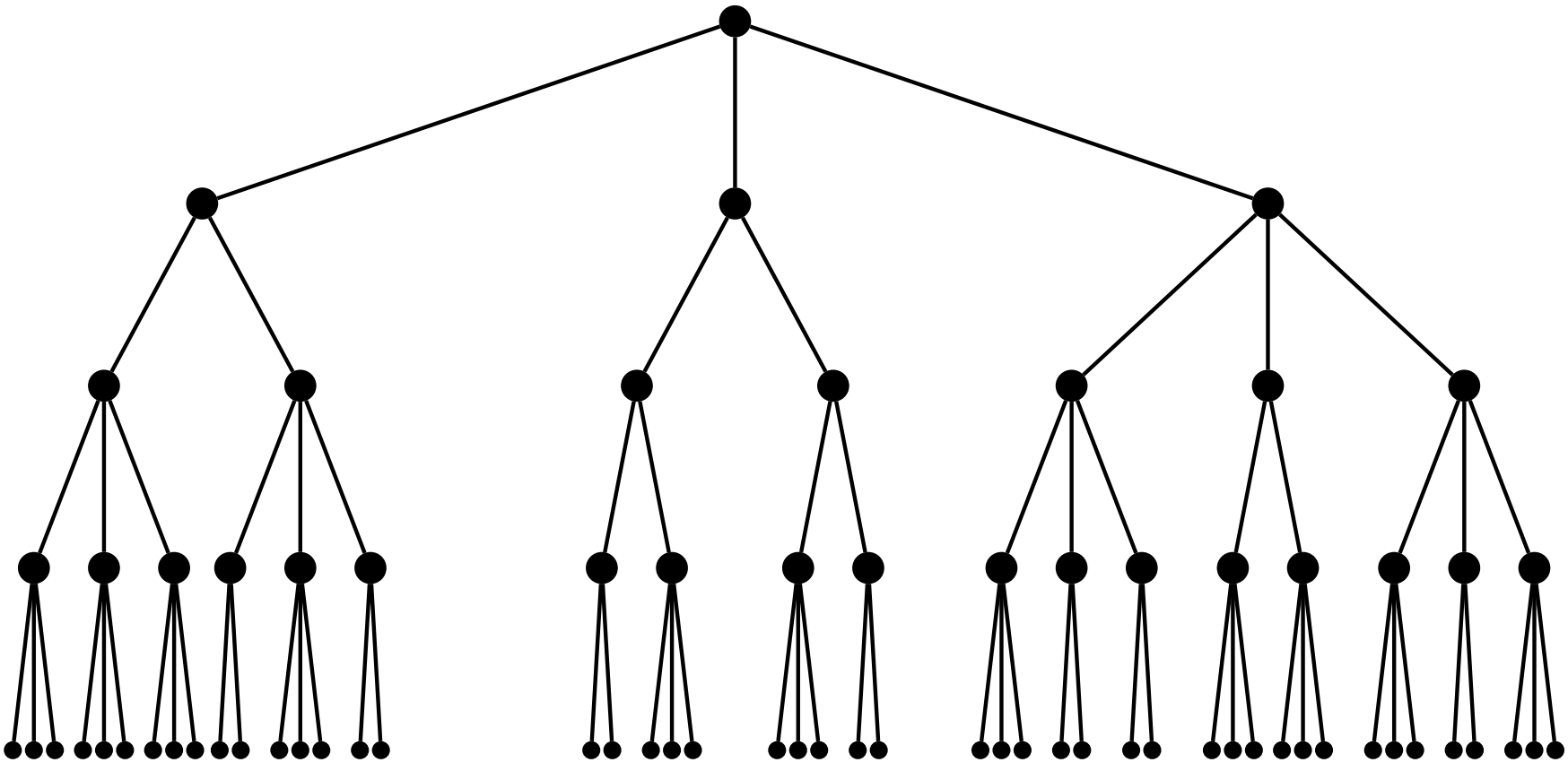
. . . with sufficient ingenuity.

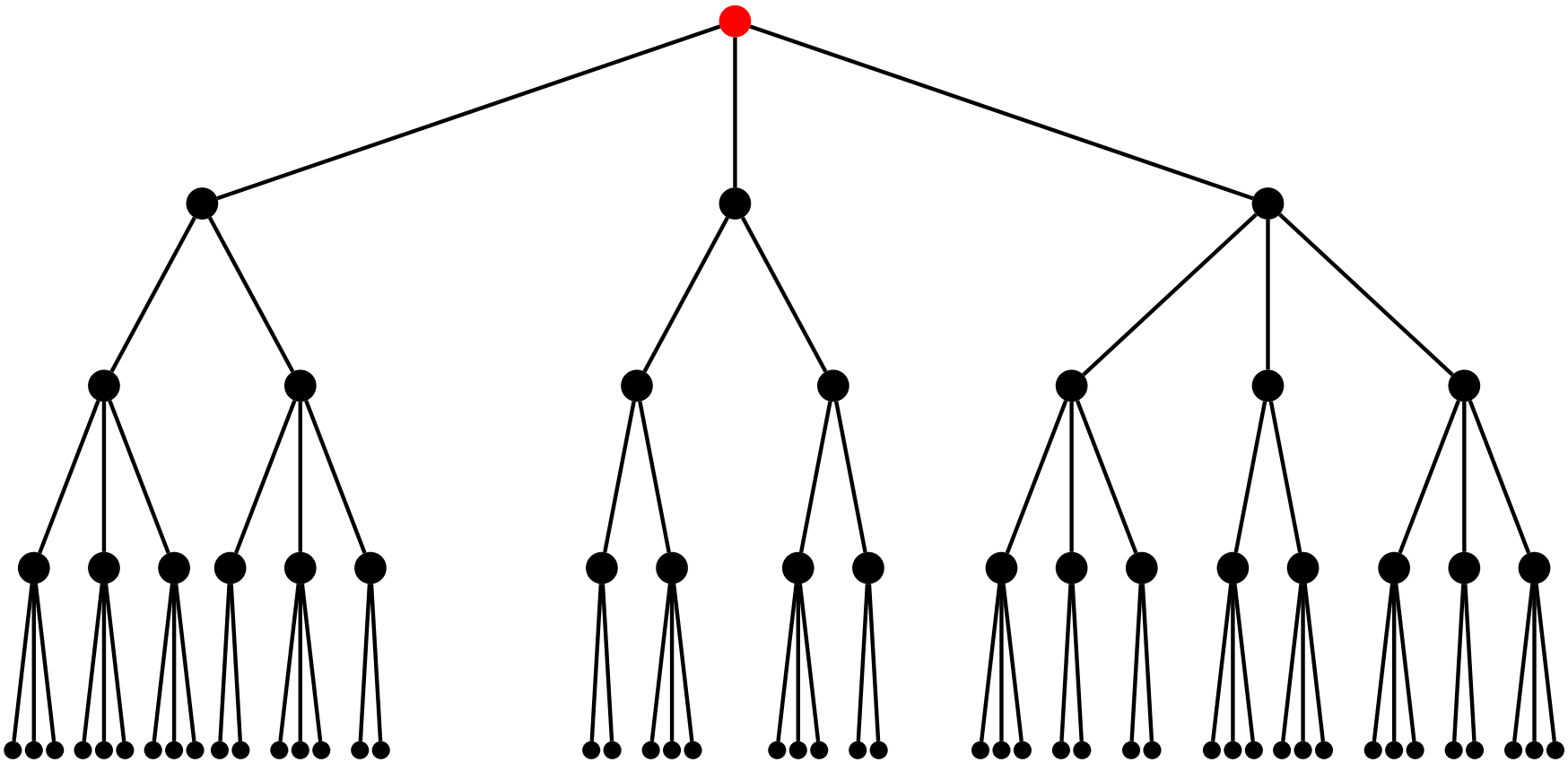
My research

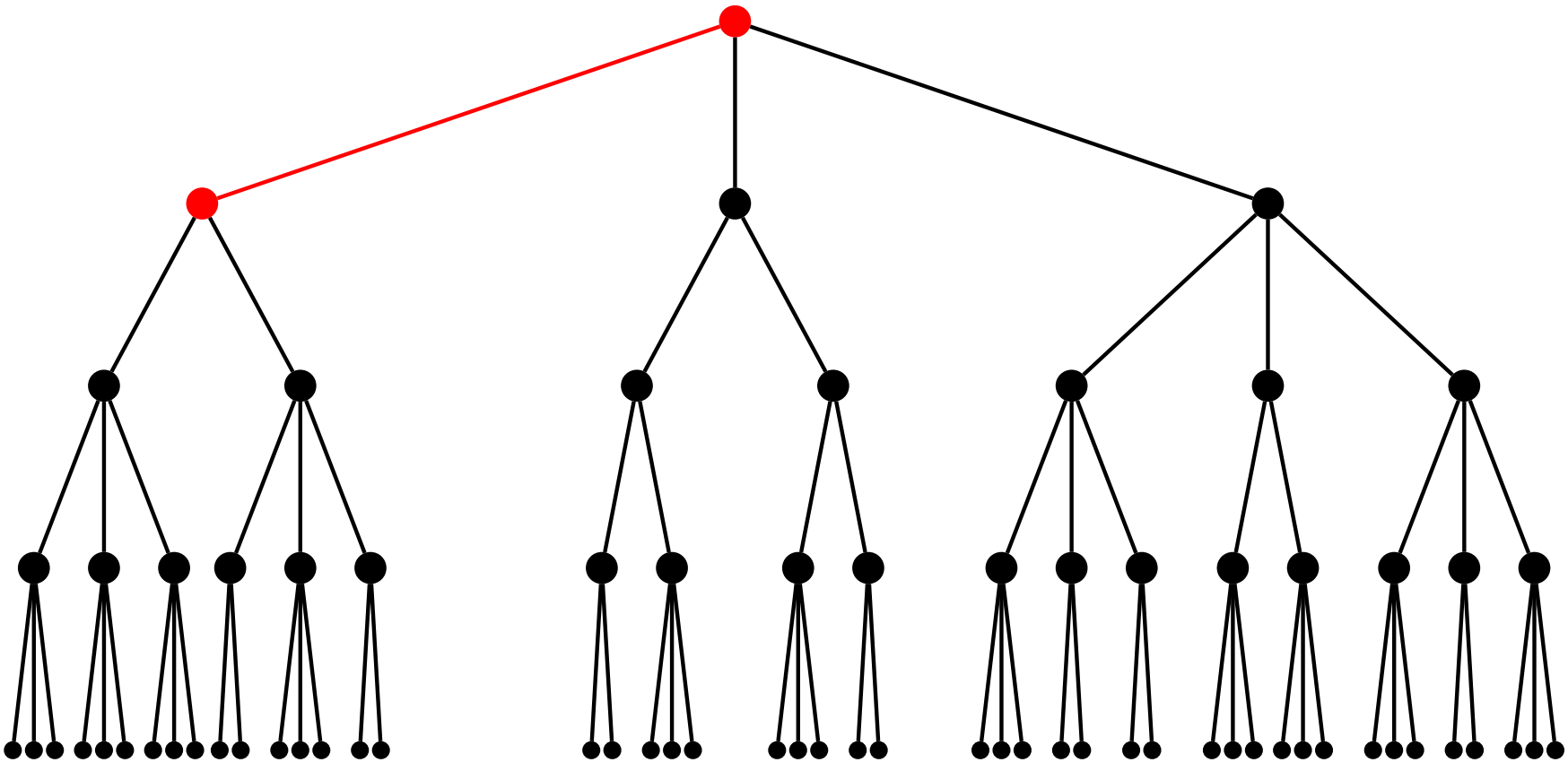
- Monte Carlo integration - embarrassingly parallel, but not what I'm interested at the moment.
- Monte Carlo simulation of self-avoiding walks - complicated algorithm, too hard.
- Self-avoiding walk enumeration via backtracking - ok?

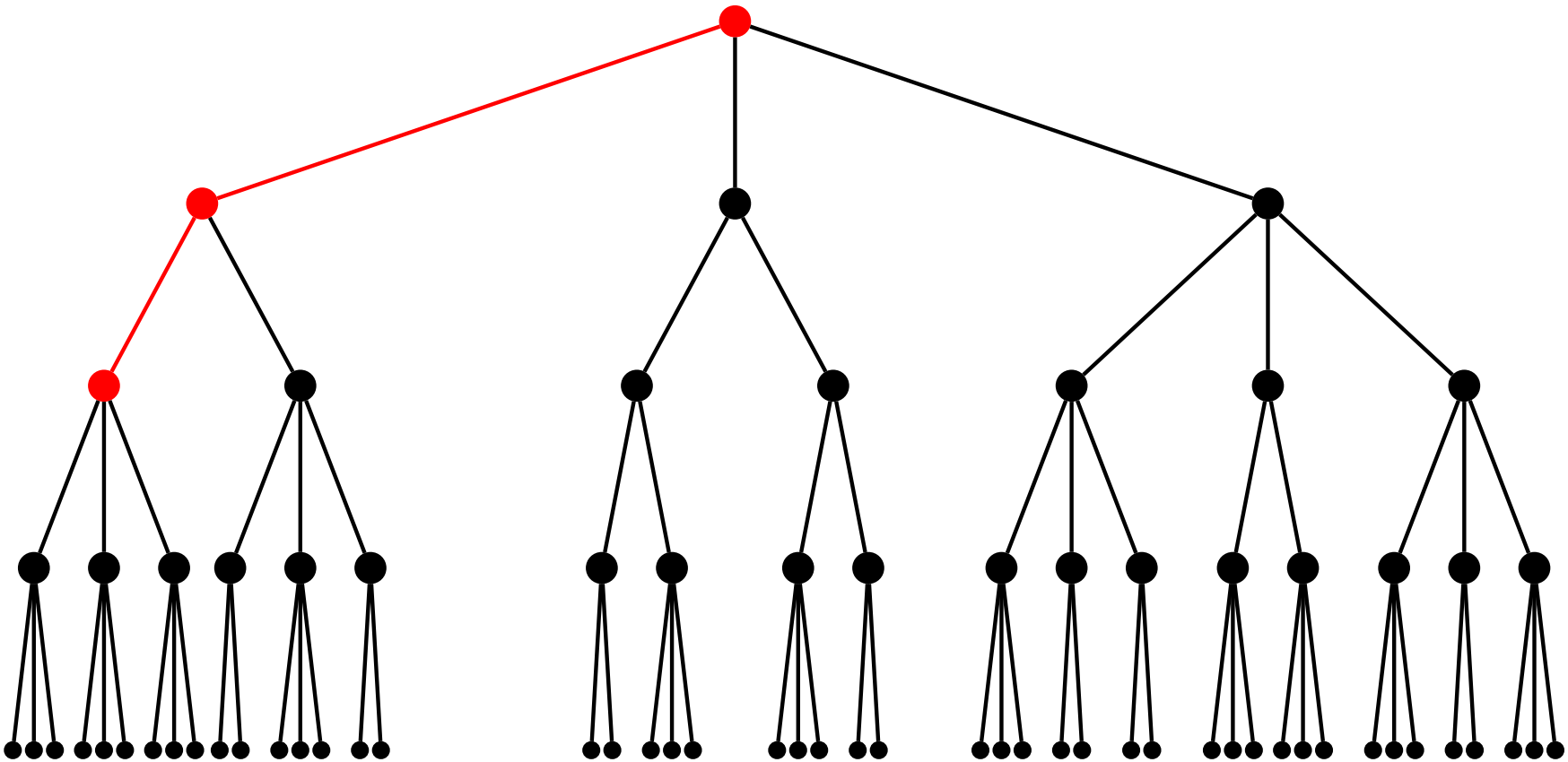
Backtracking

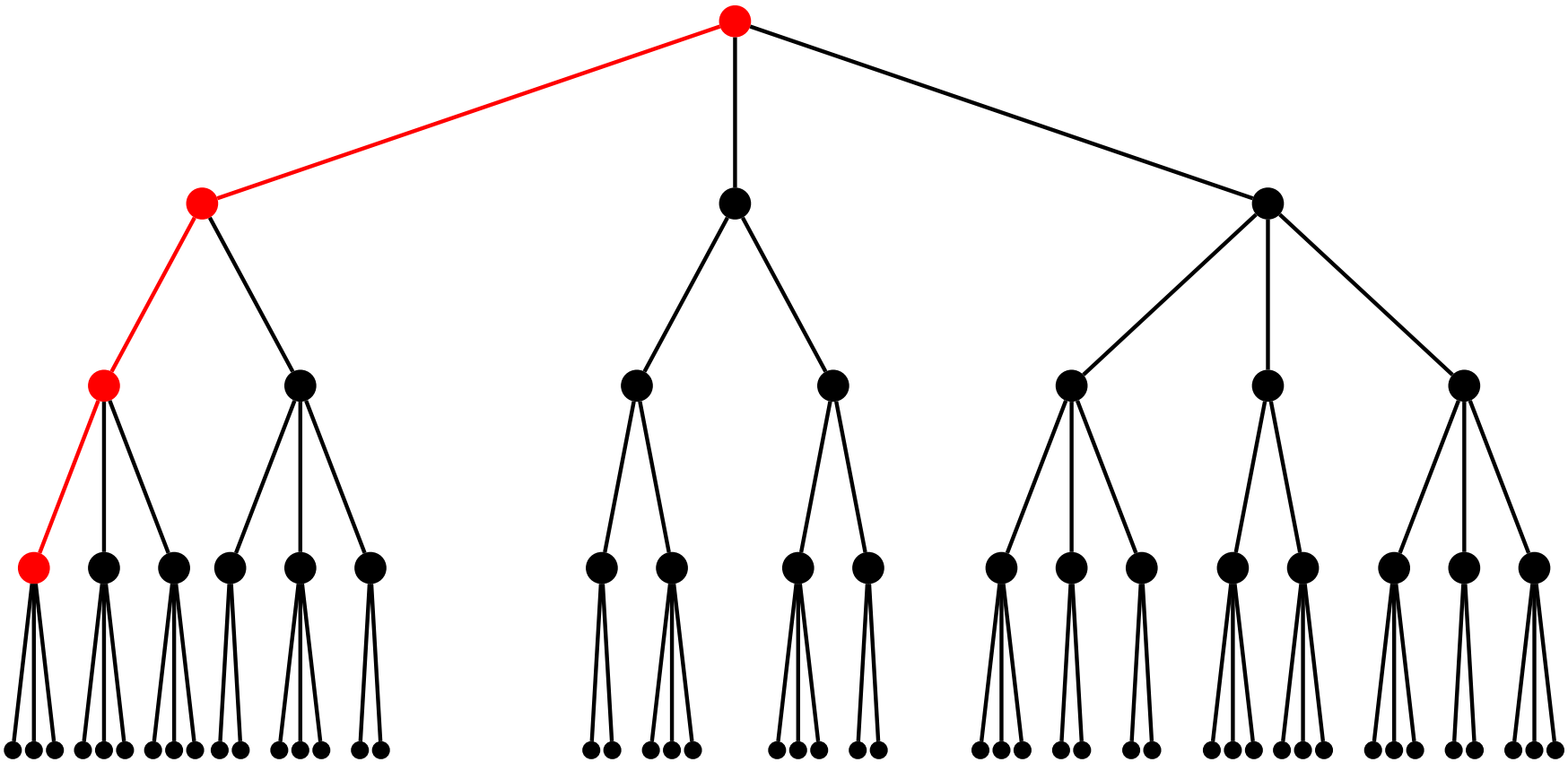
- Exhaustively generate 'objects' by incrementally building up on smaller objects.
- e.g. binary strings of 3 digits:
  $\{empty\} \to 0 \to 00 \to 000 \to 00 \to 001 \to 00 \to 0 \to 01 \to 010 \to 01 \to 011 \to \cdots$
- $\equiv$ depth first search.
- Each object has a unique parent, this defines the *backtracking tree*.

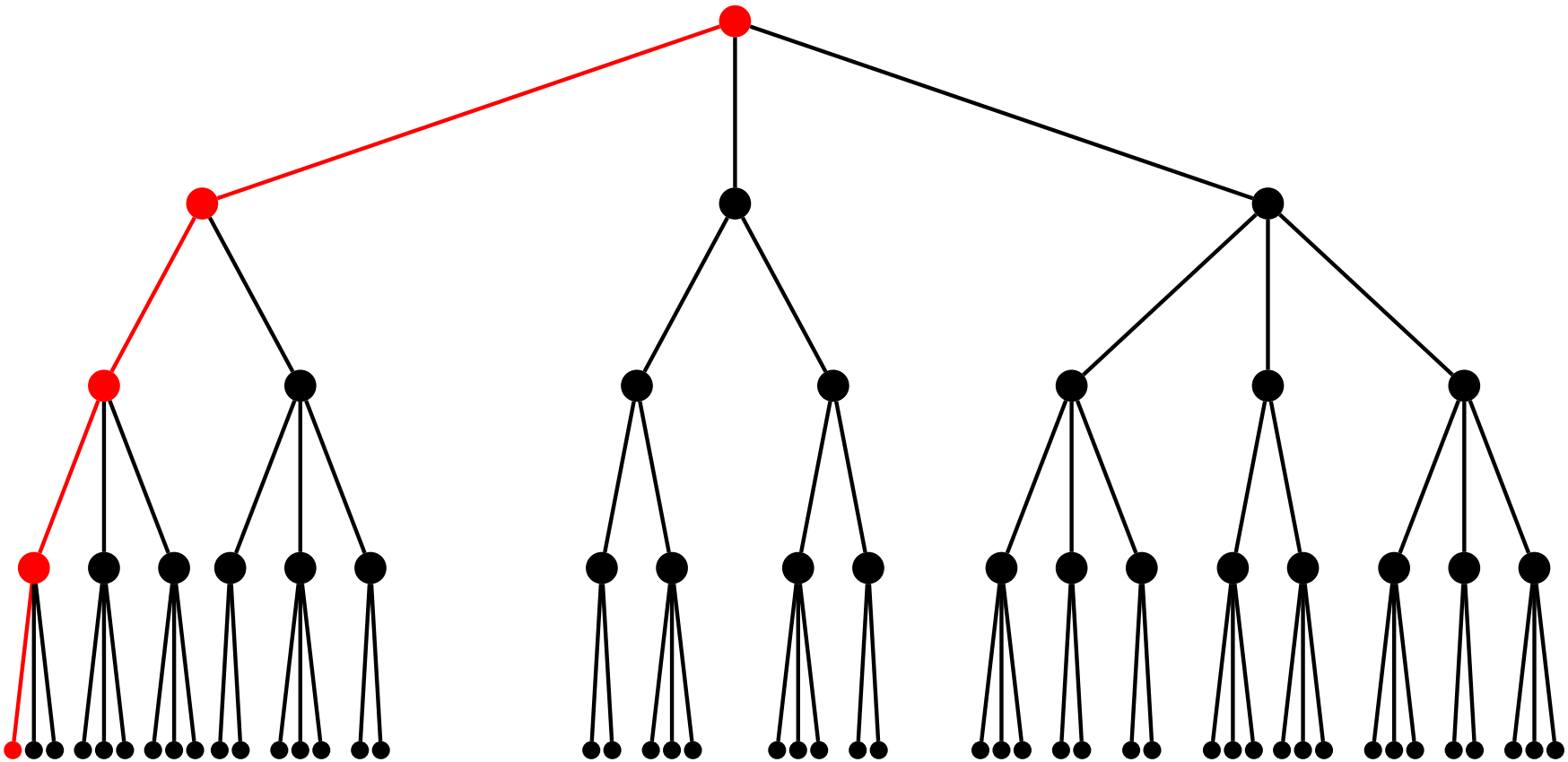


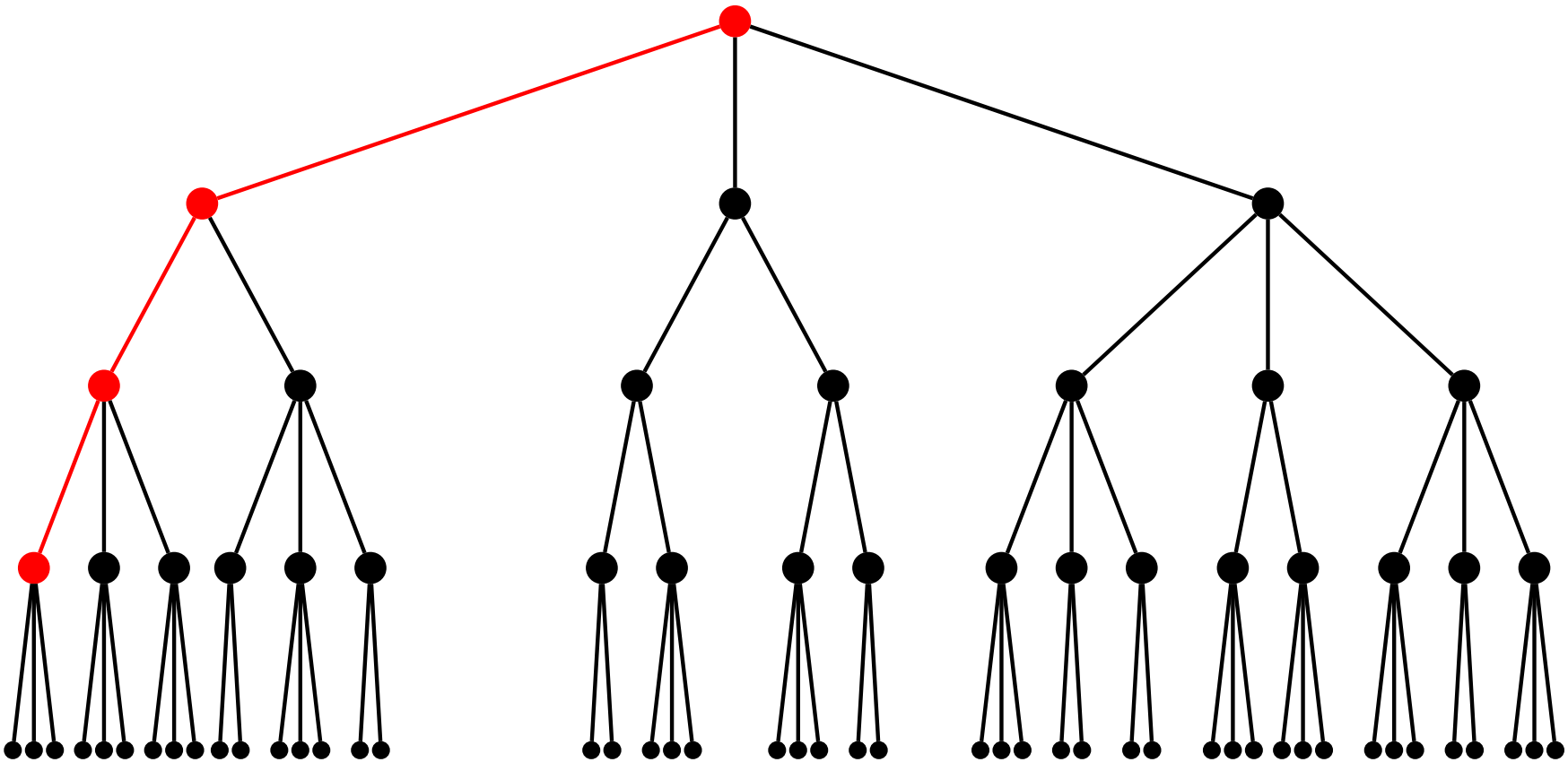


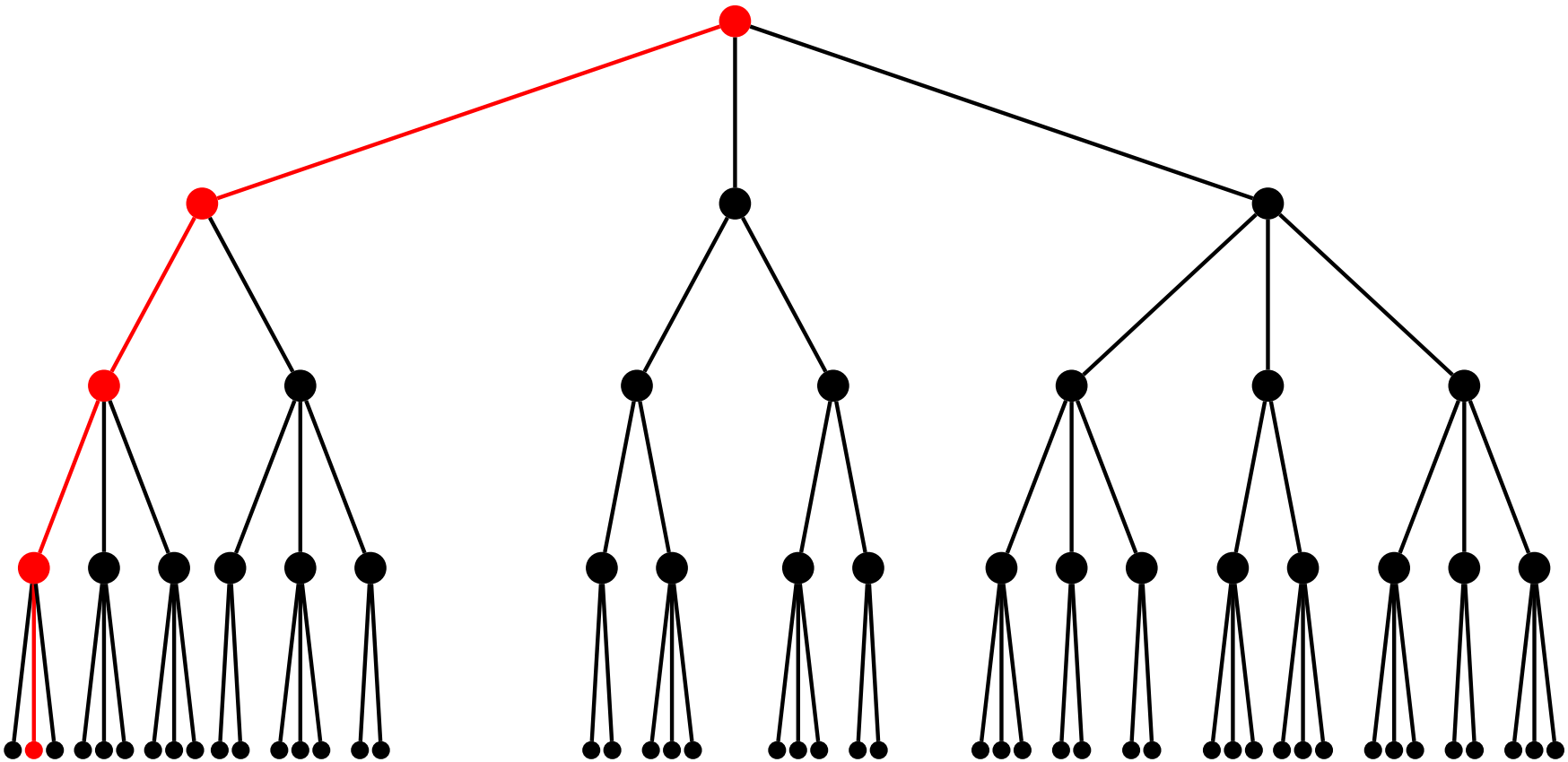


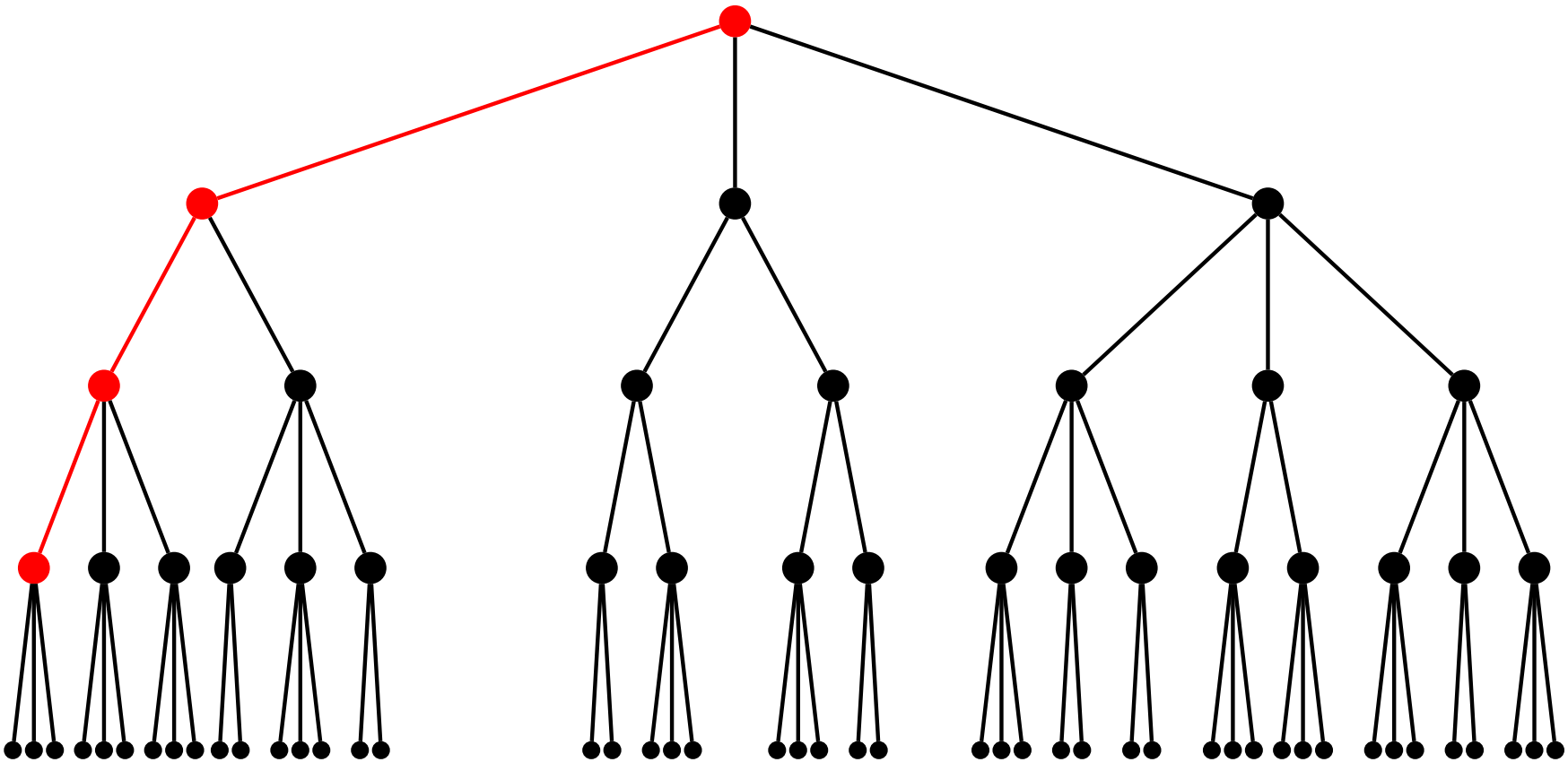


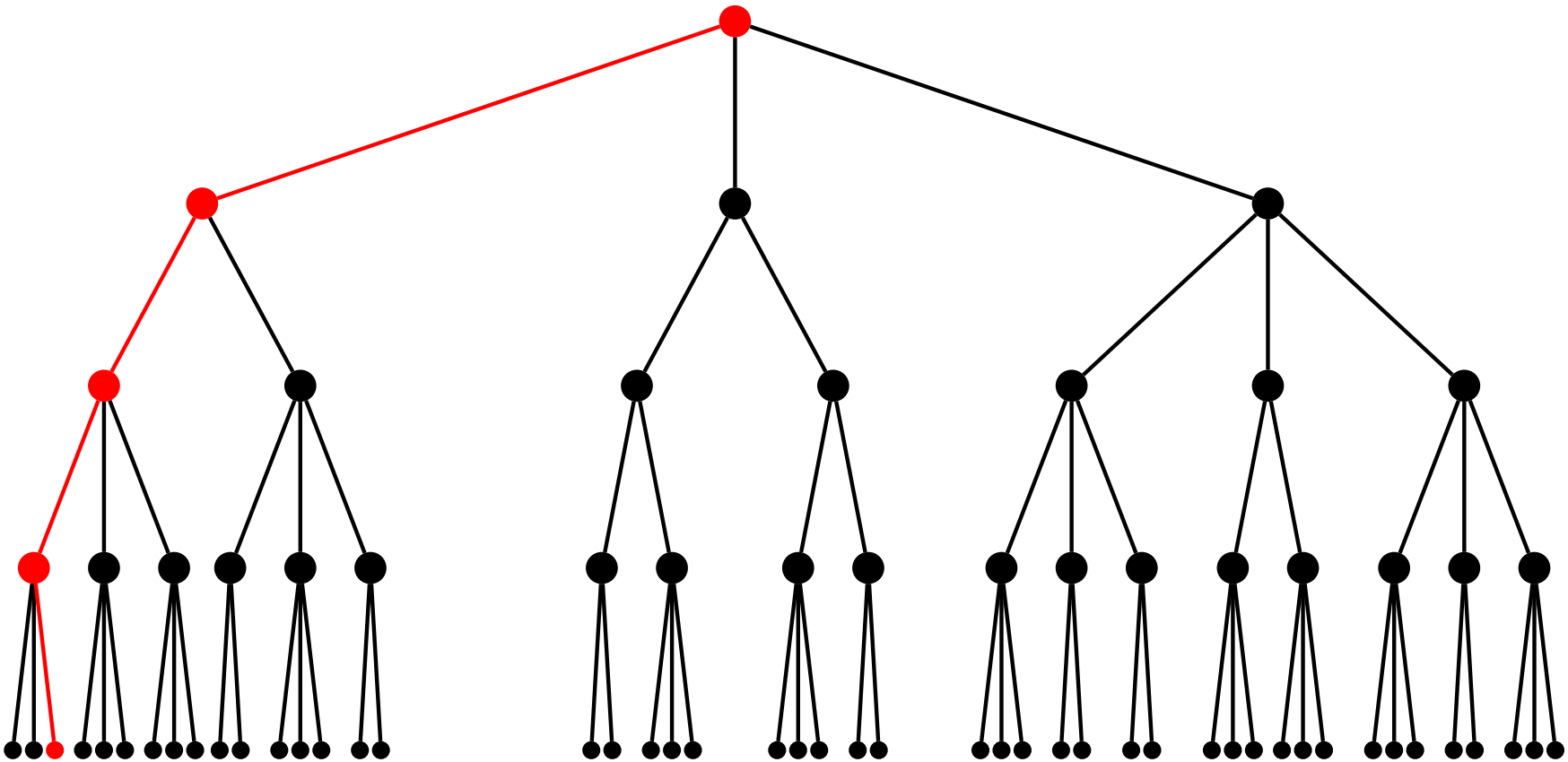


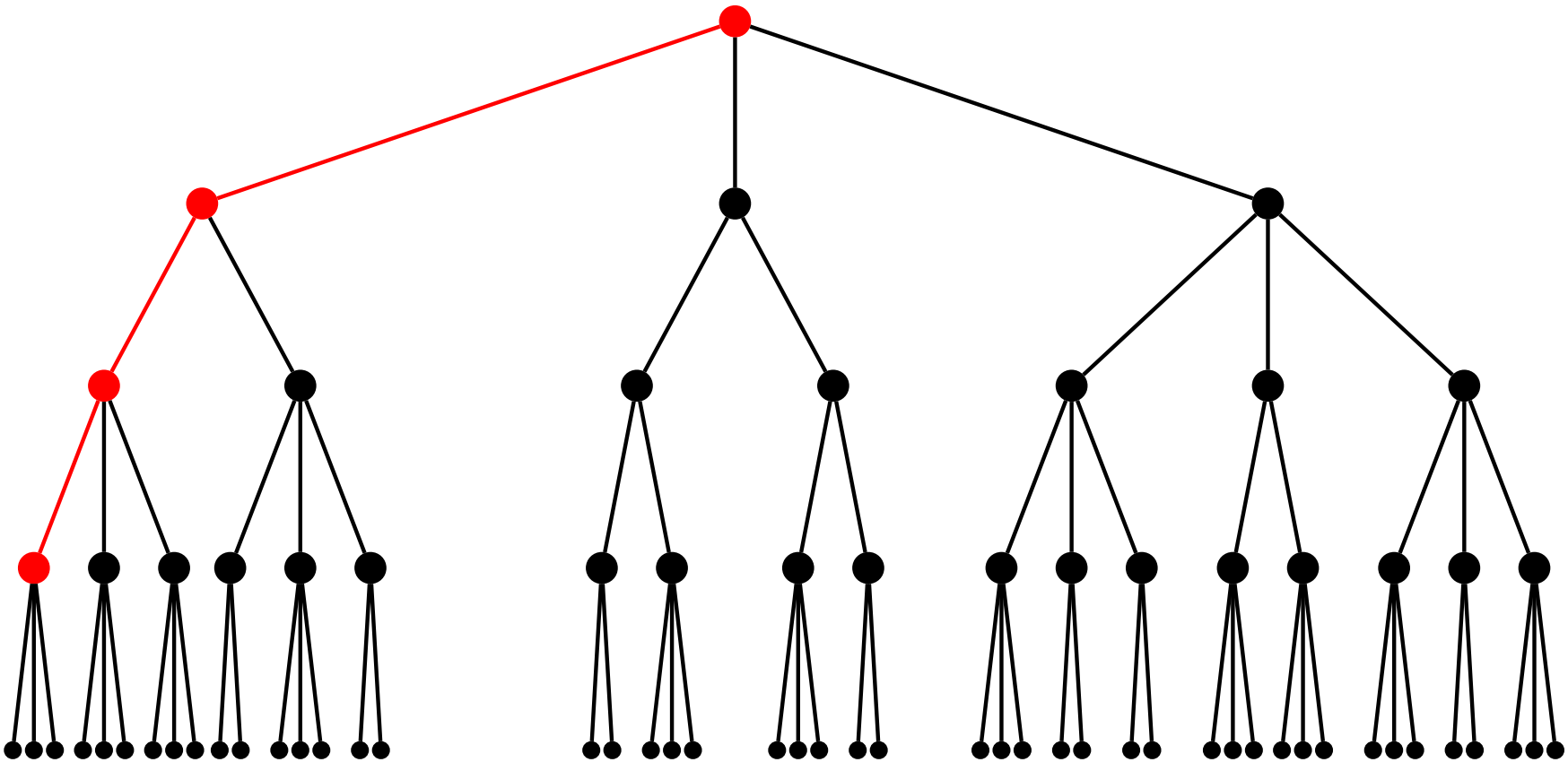


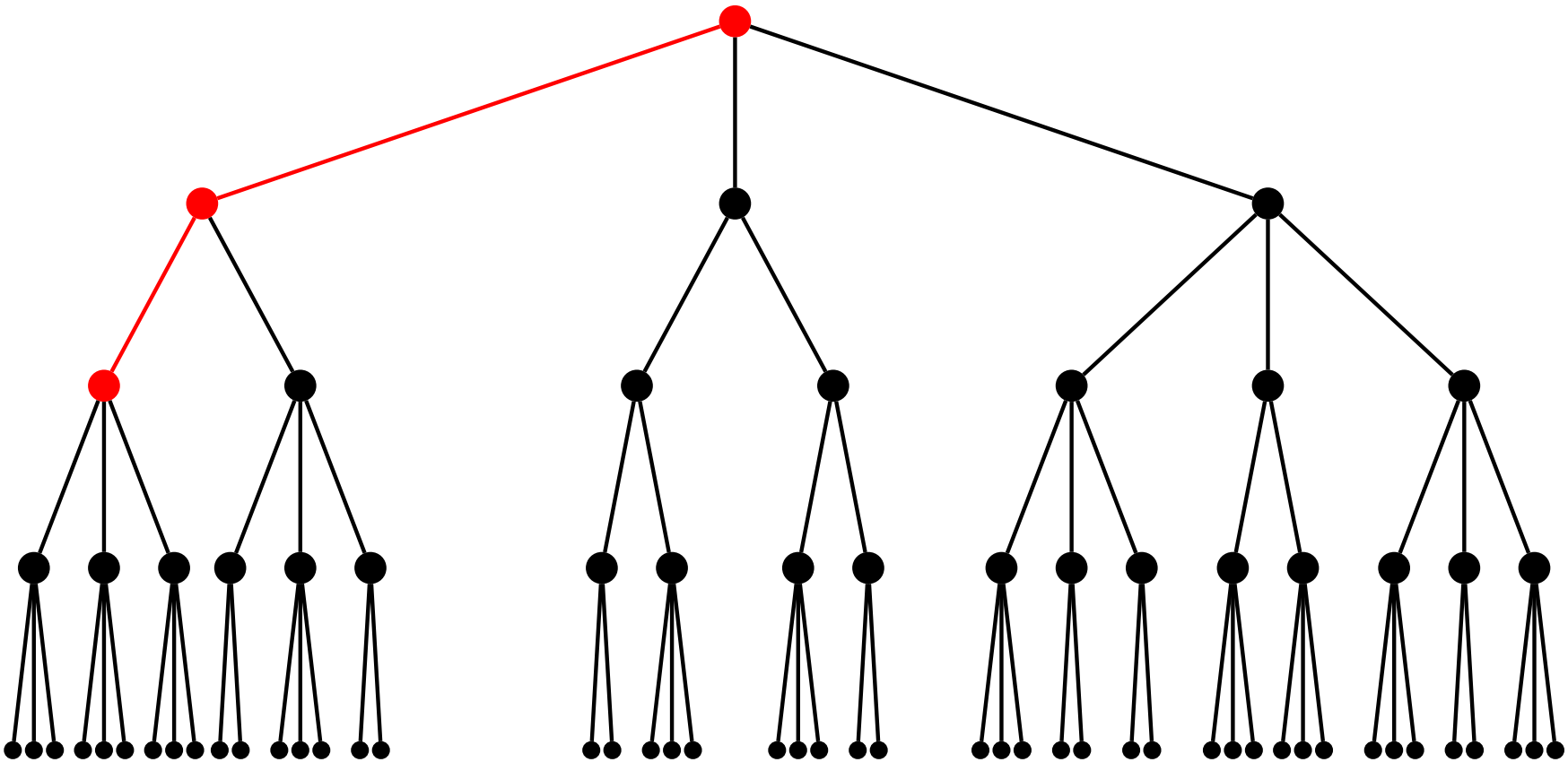


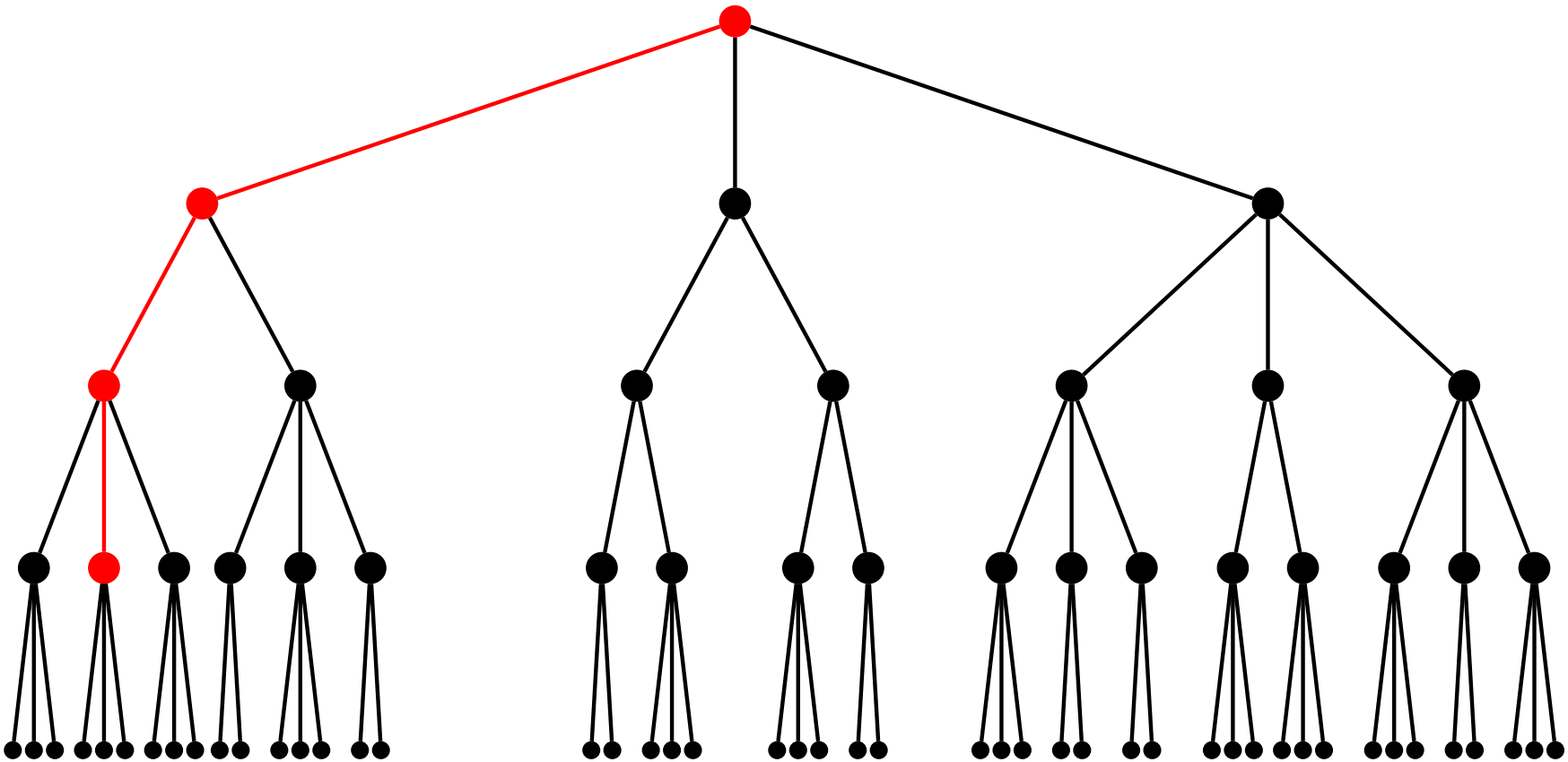


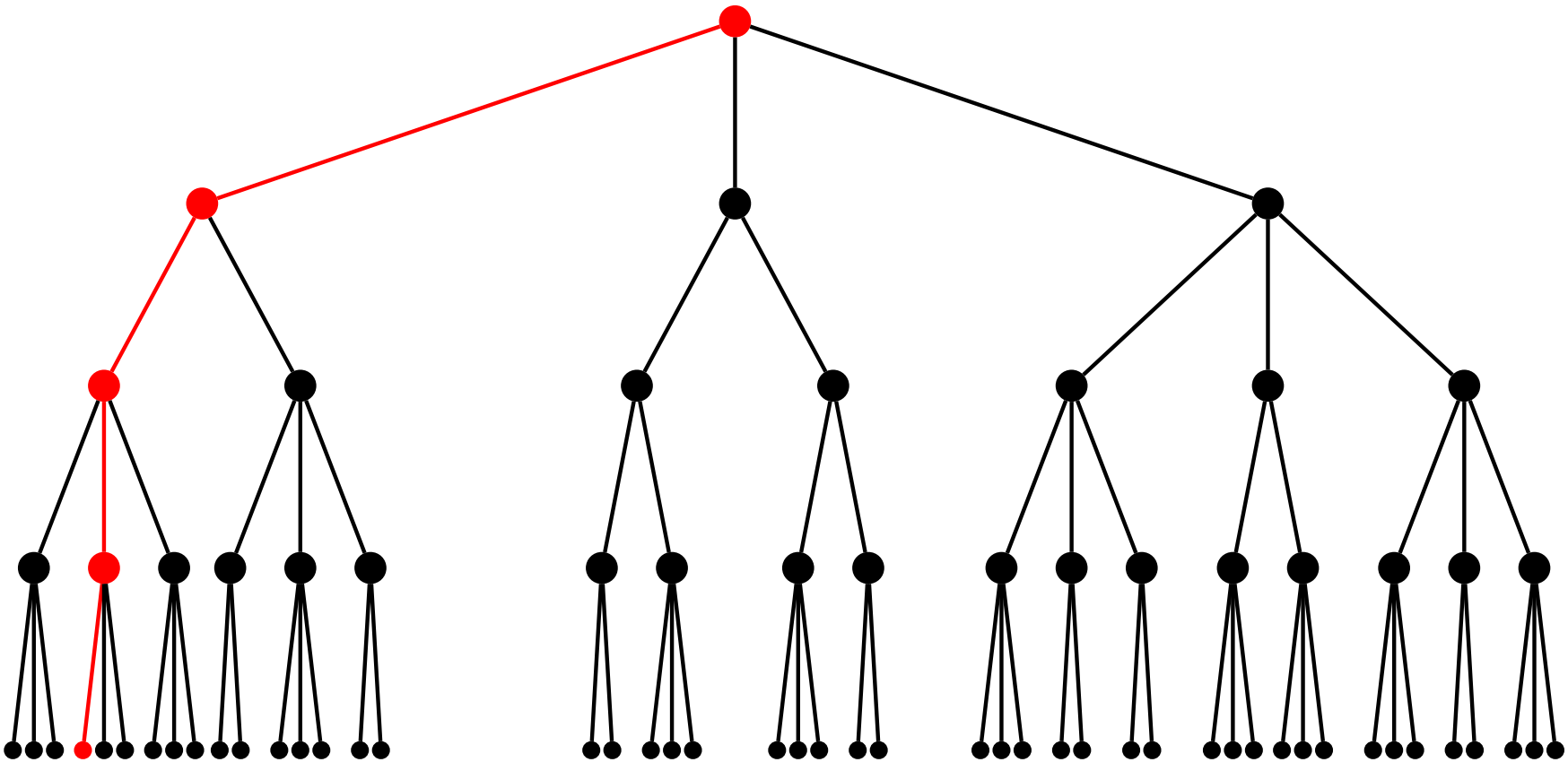


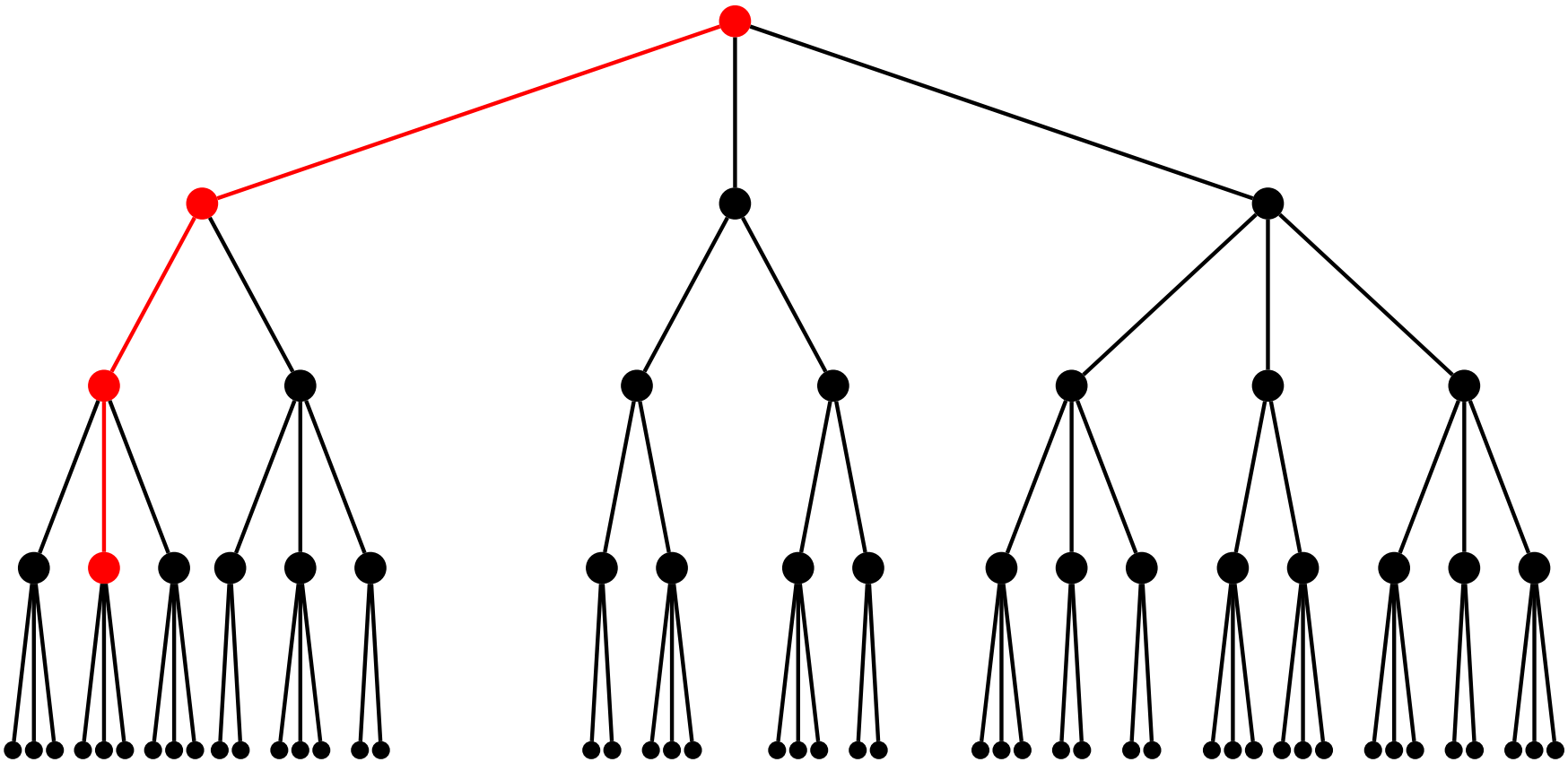


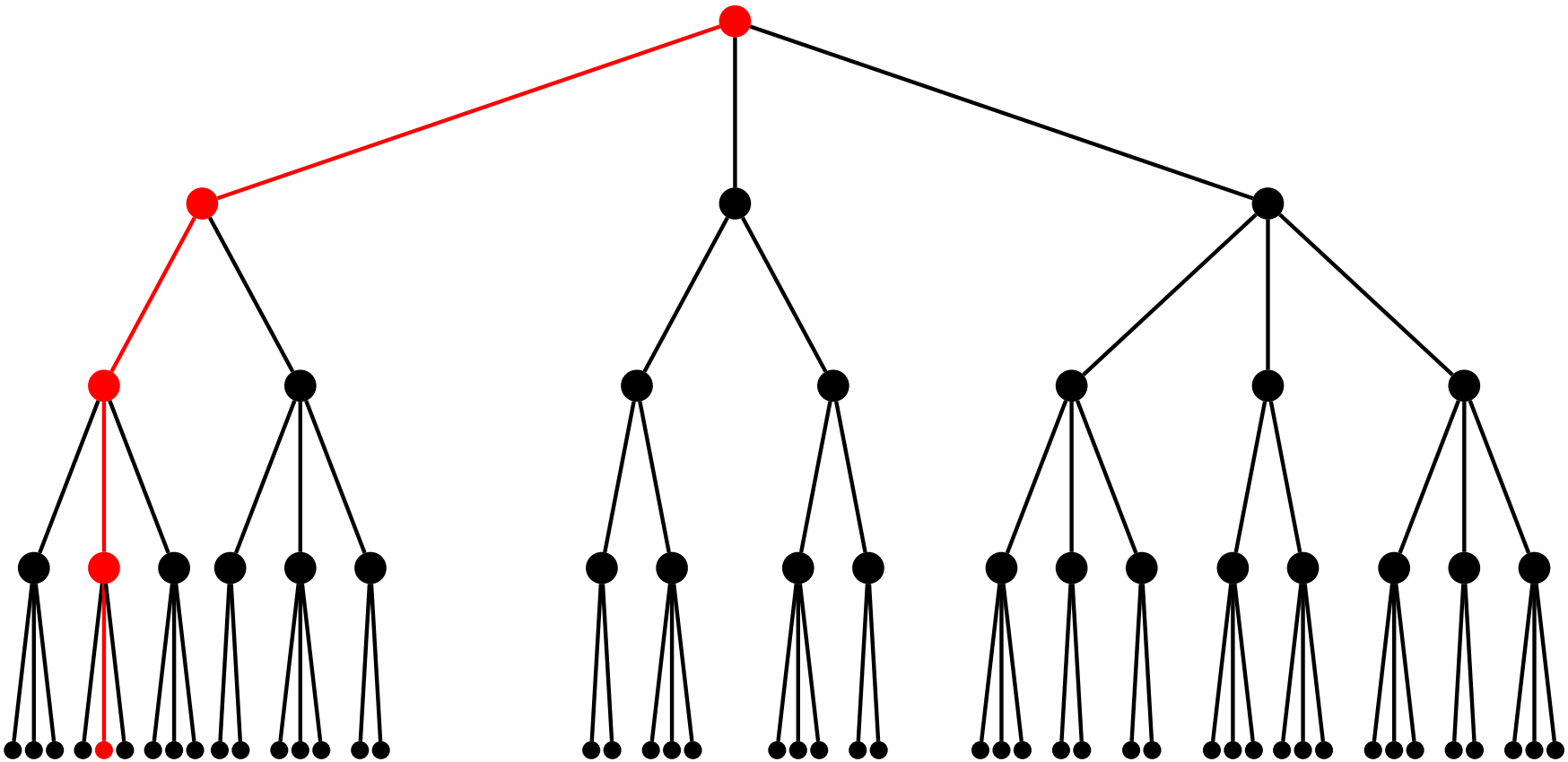


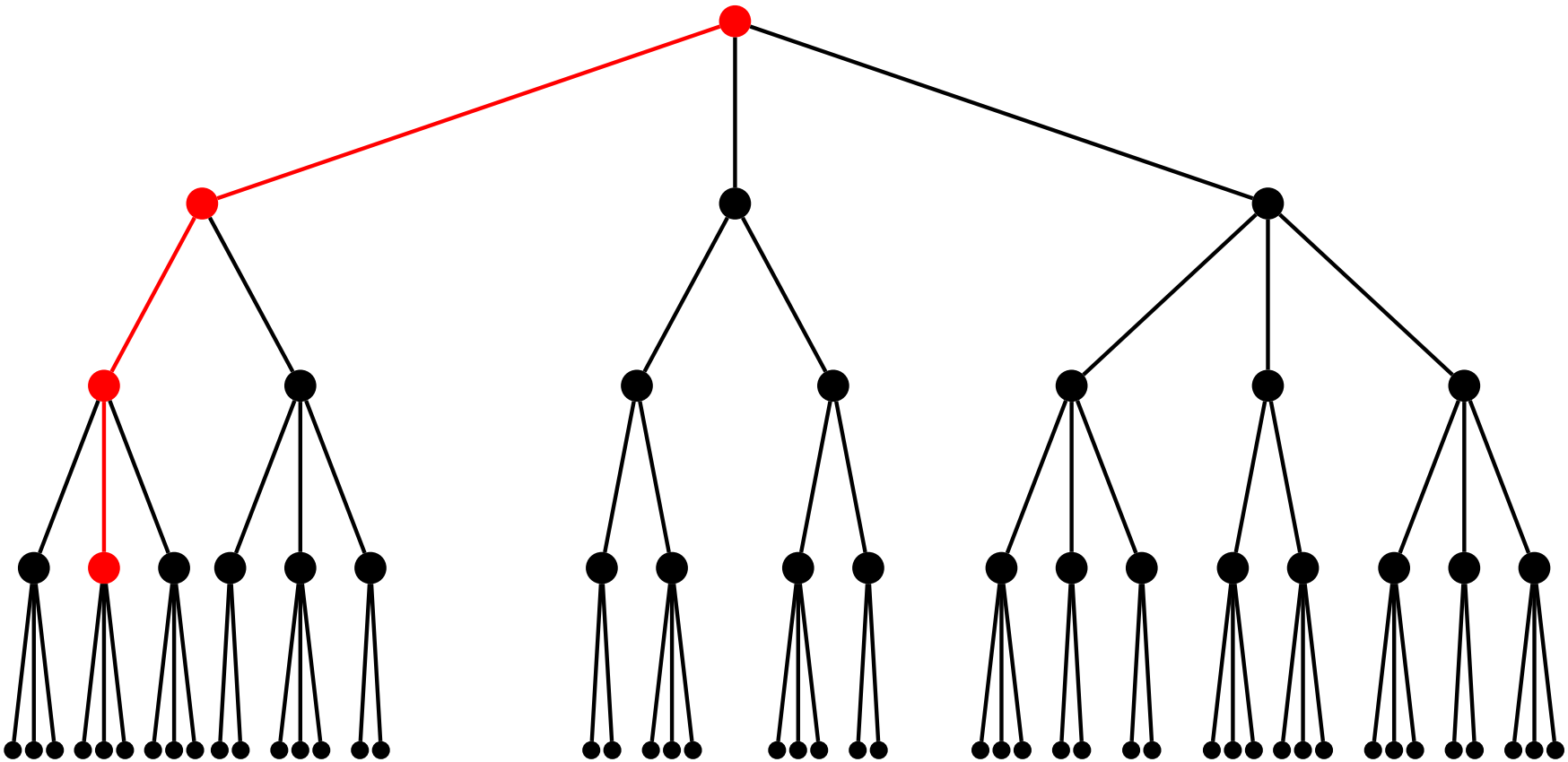


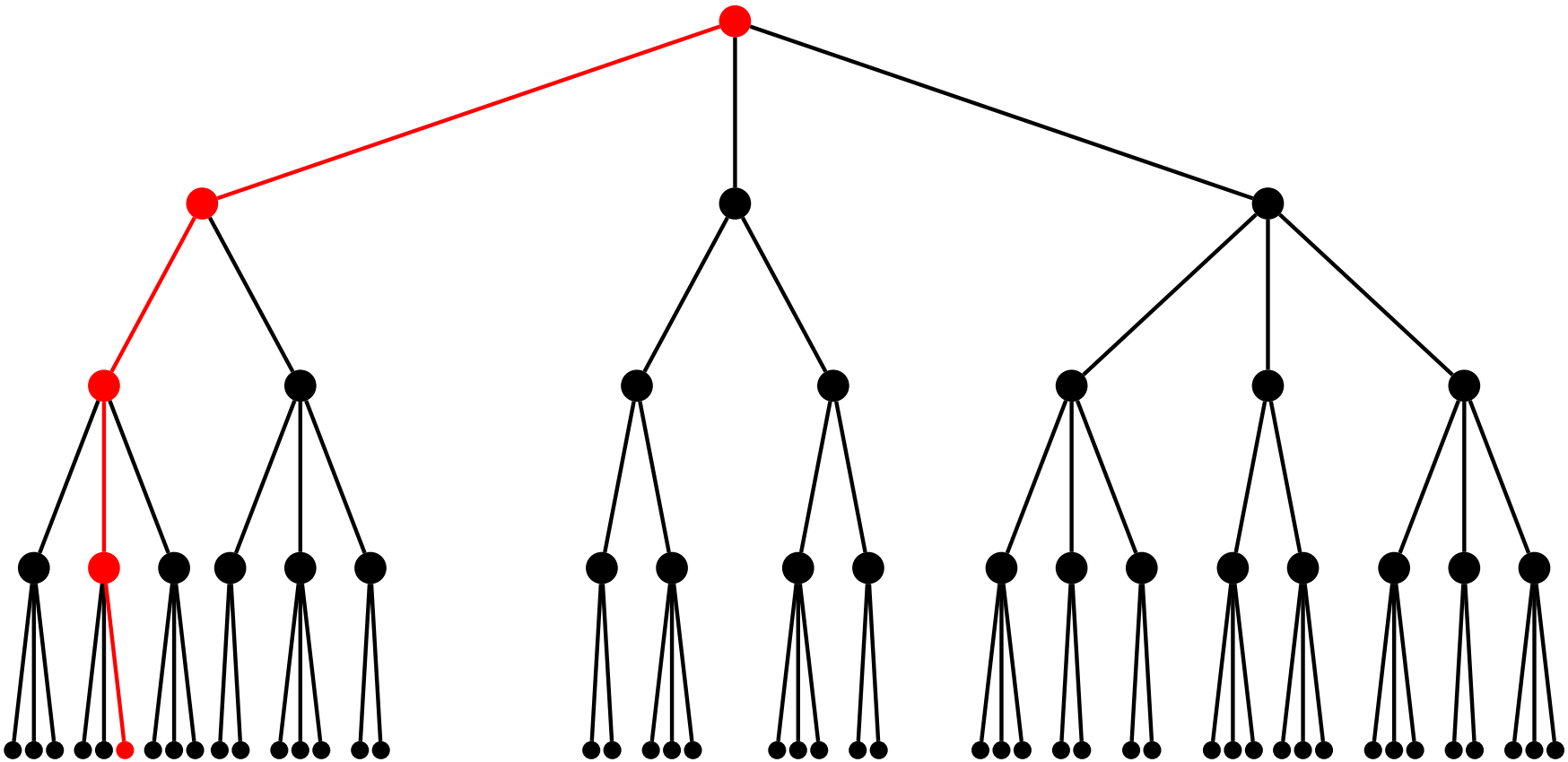


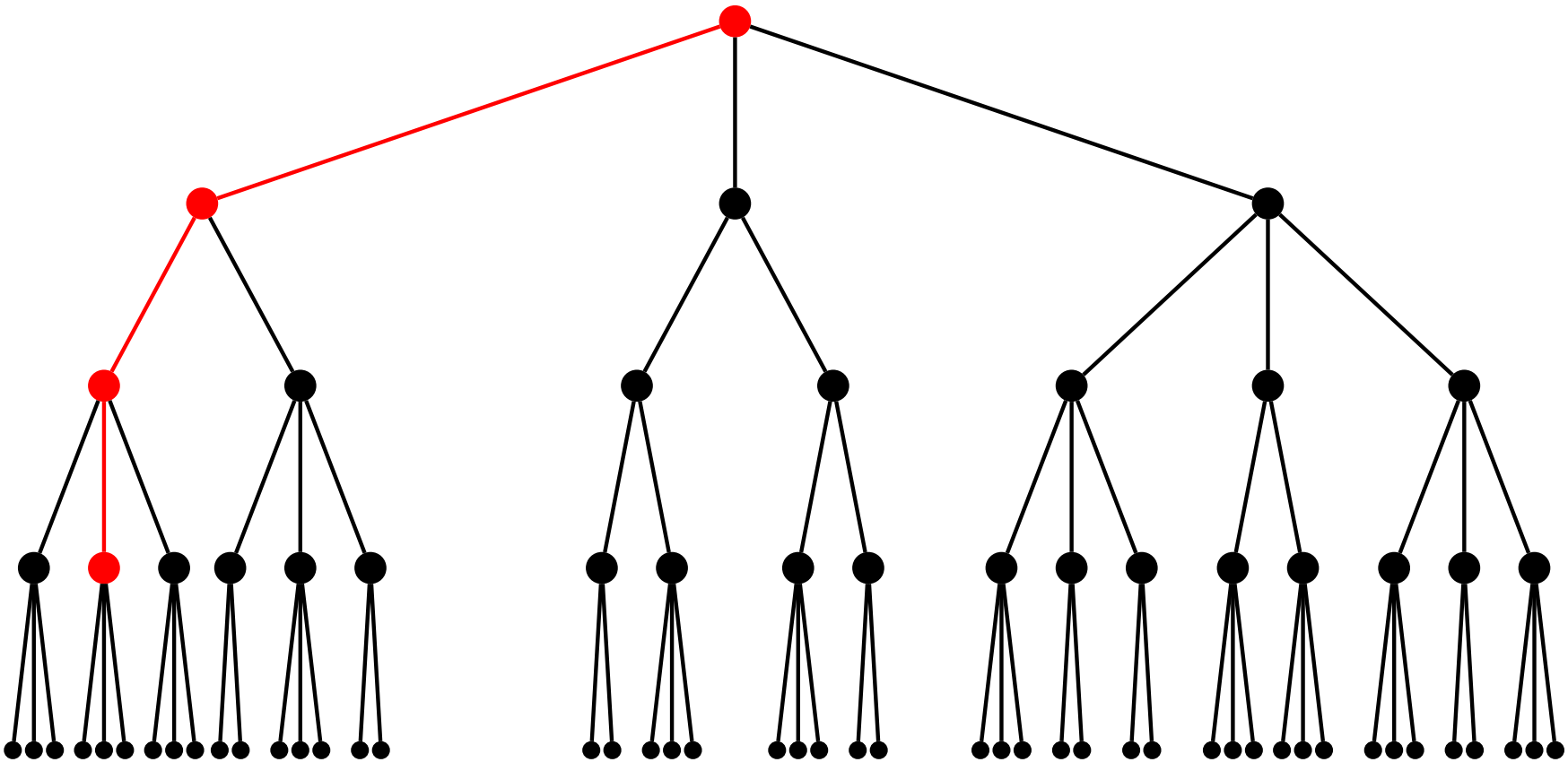


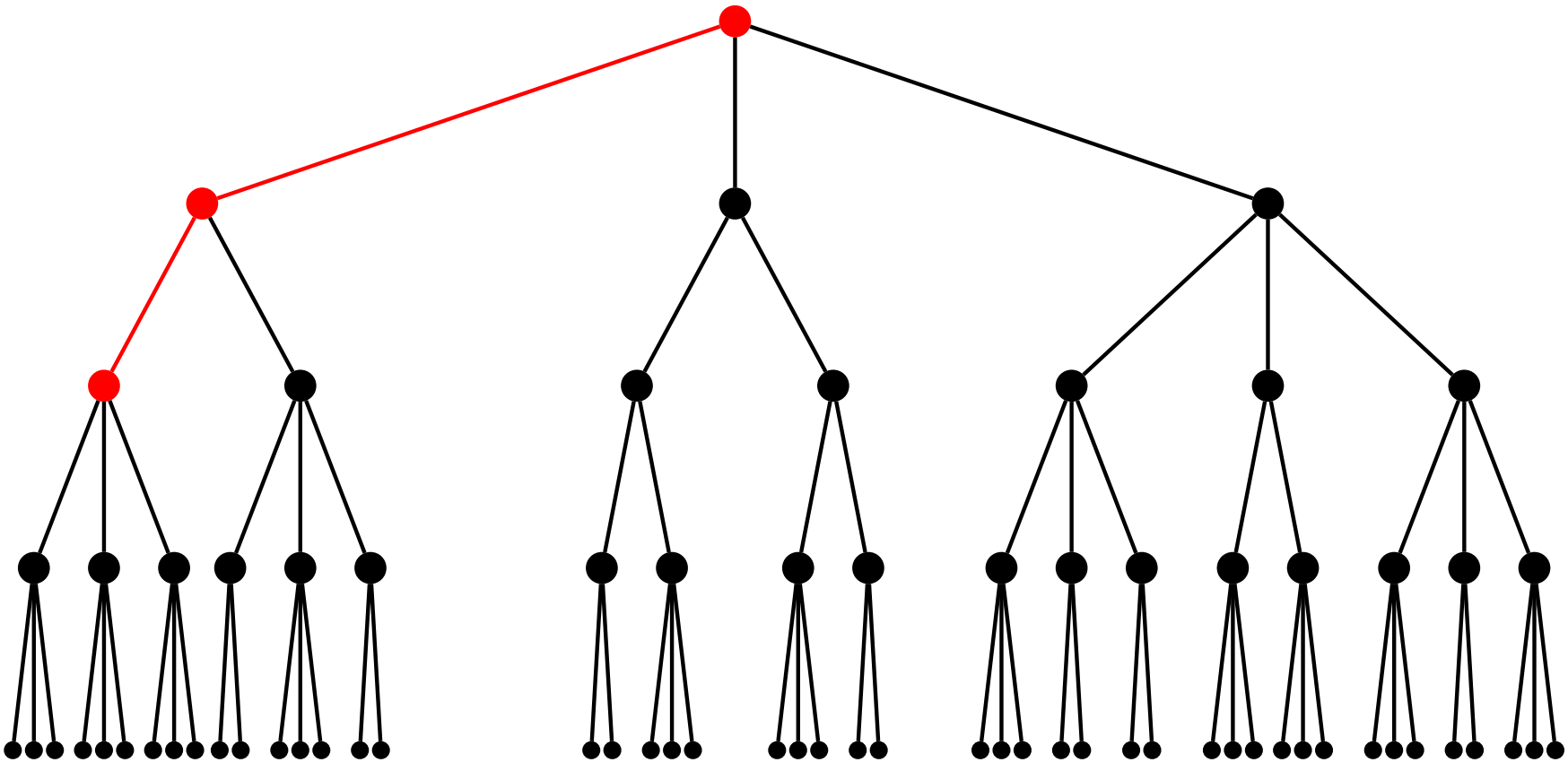


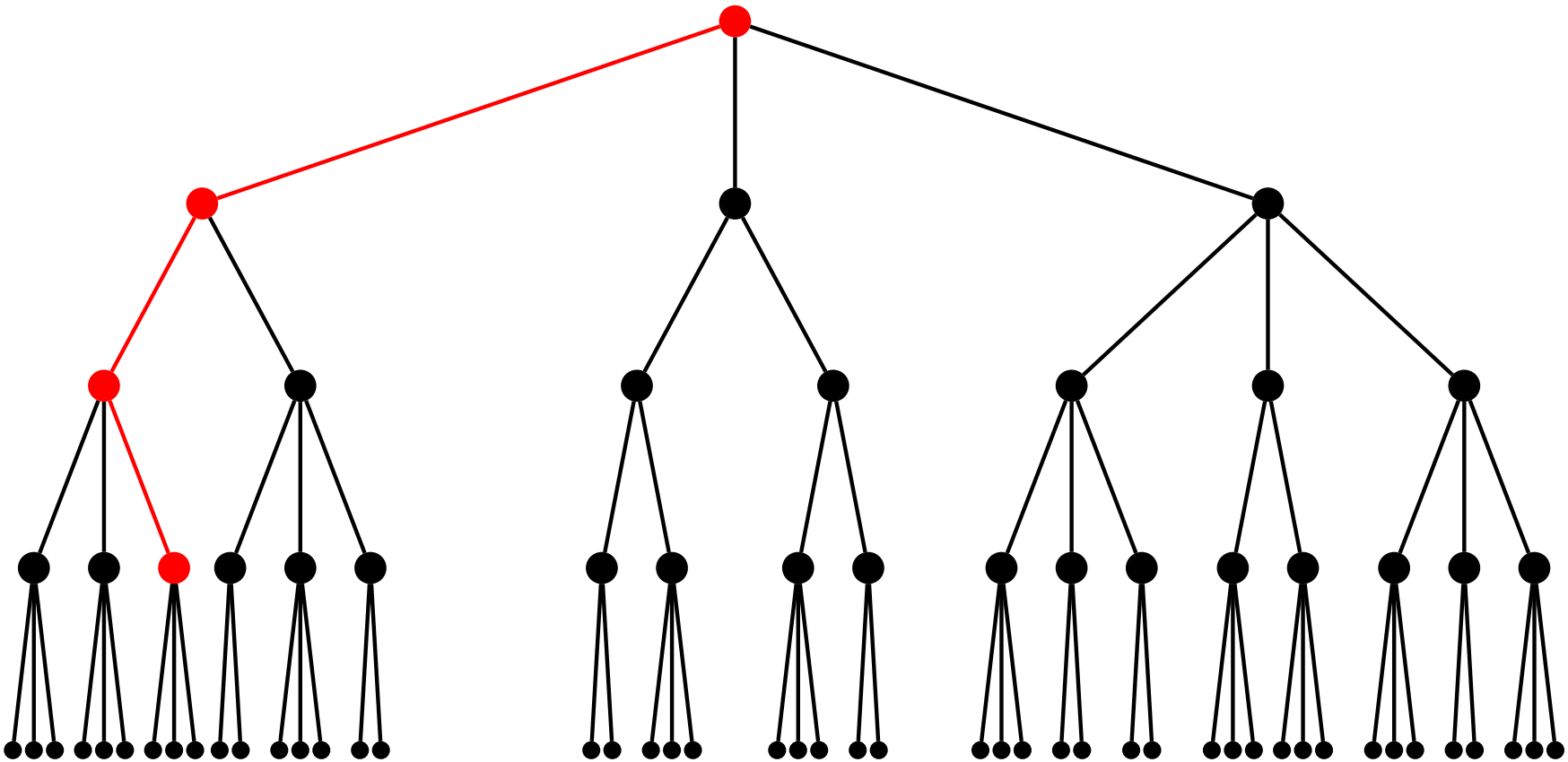


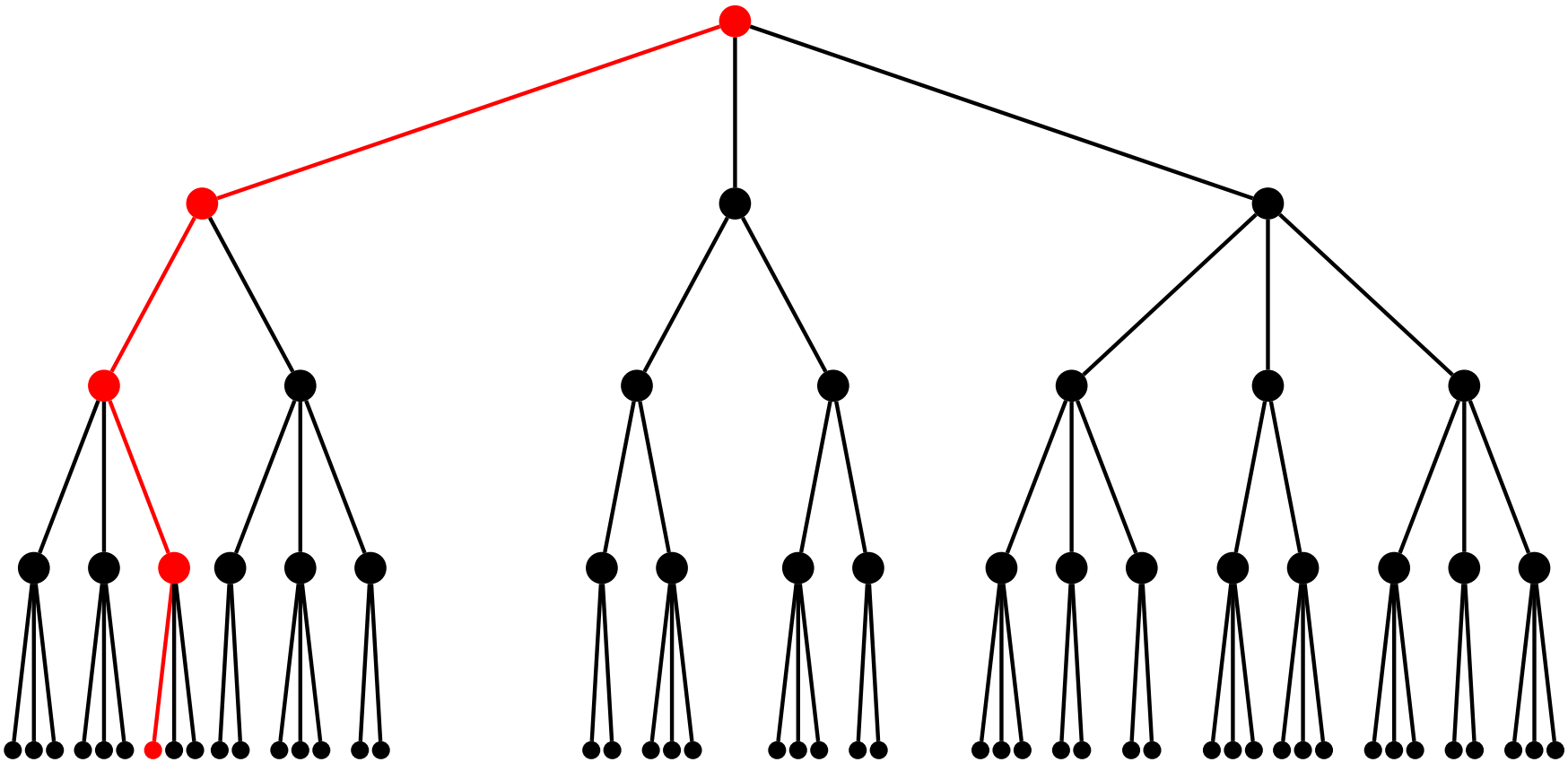


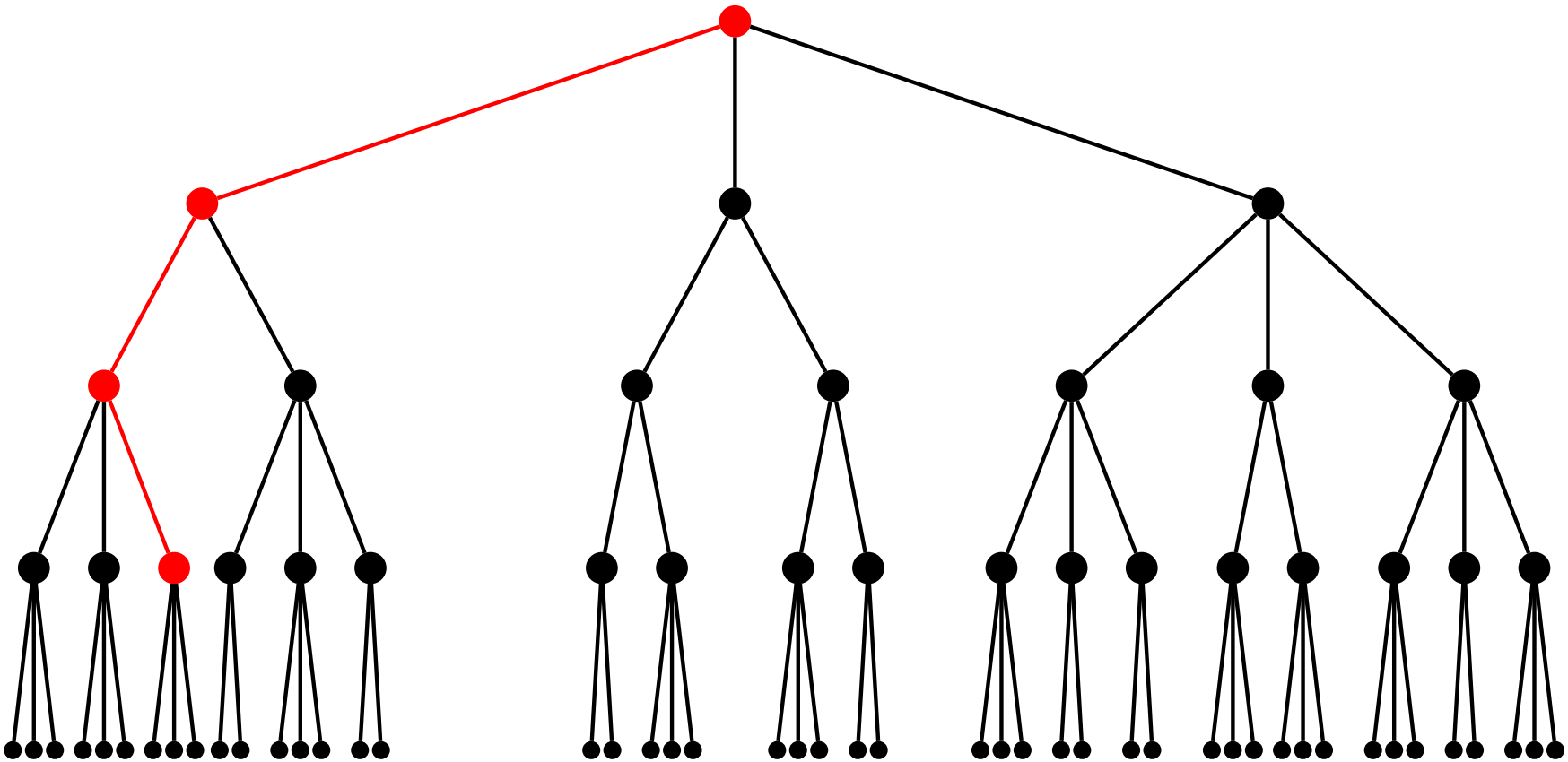


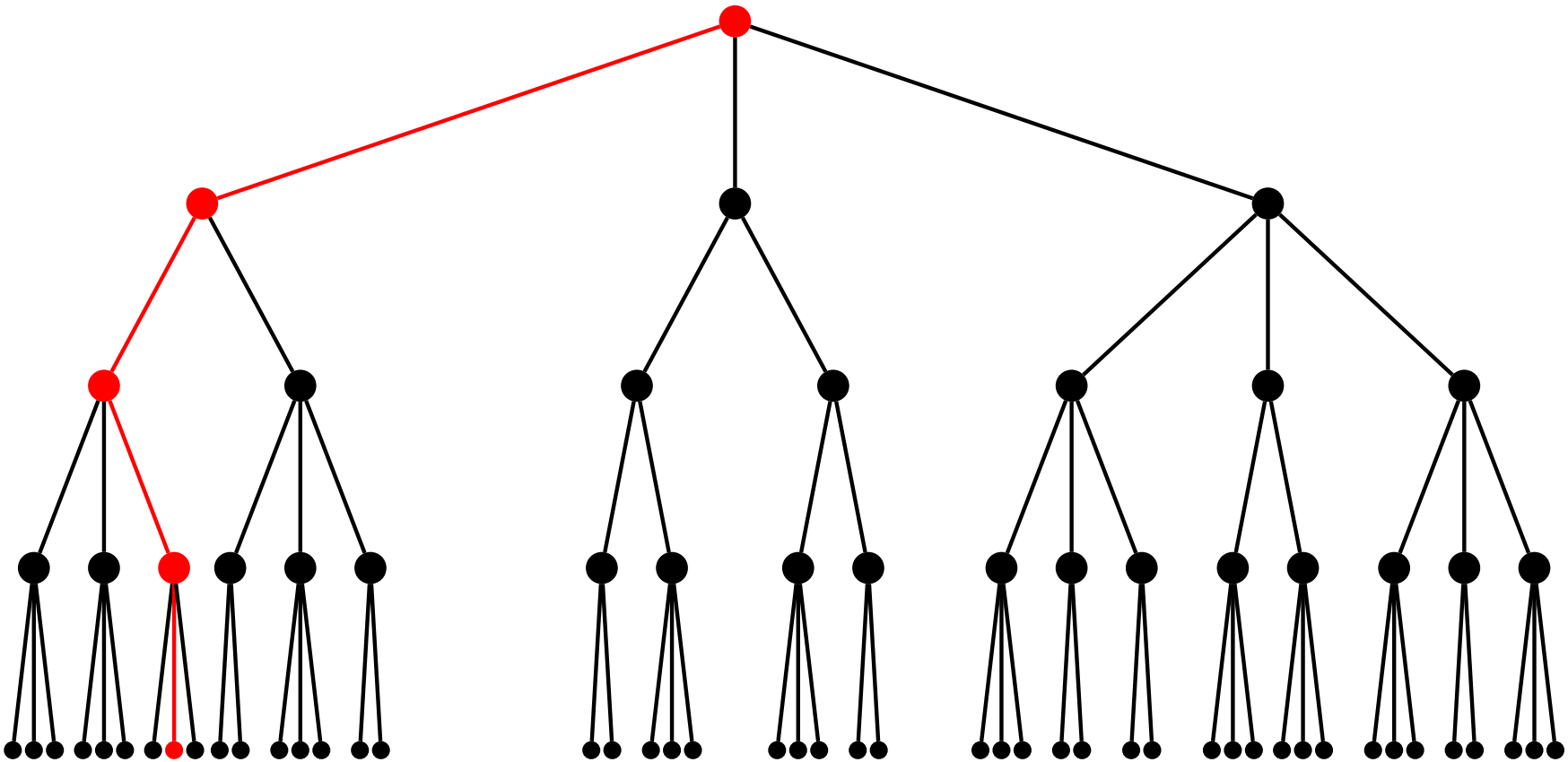


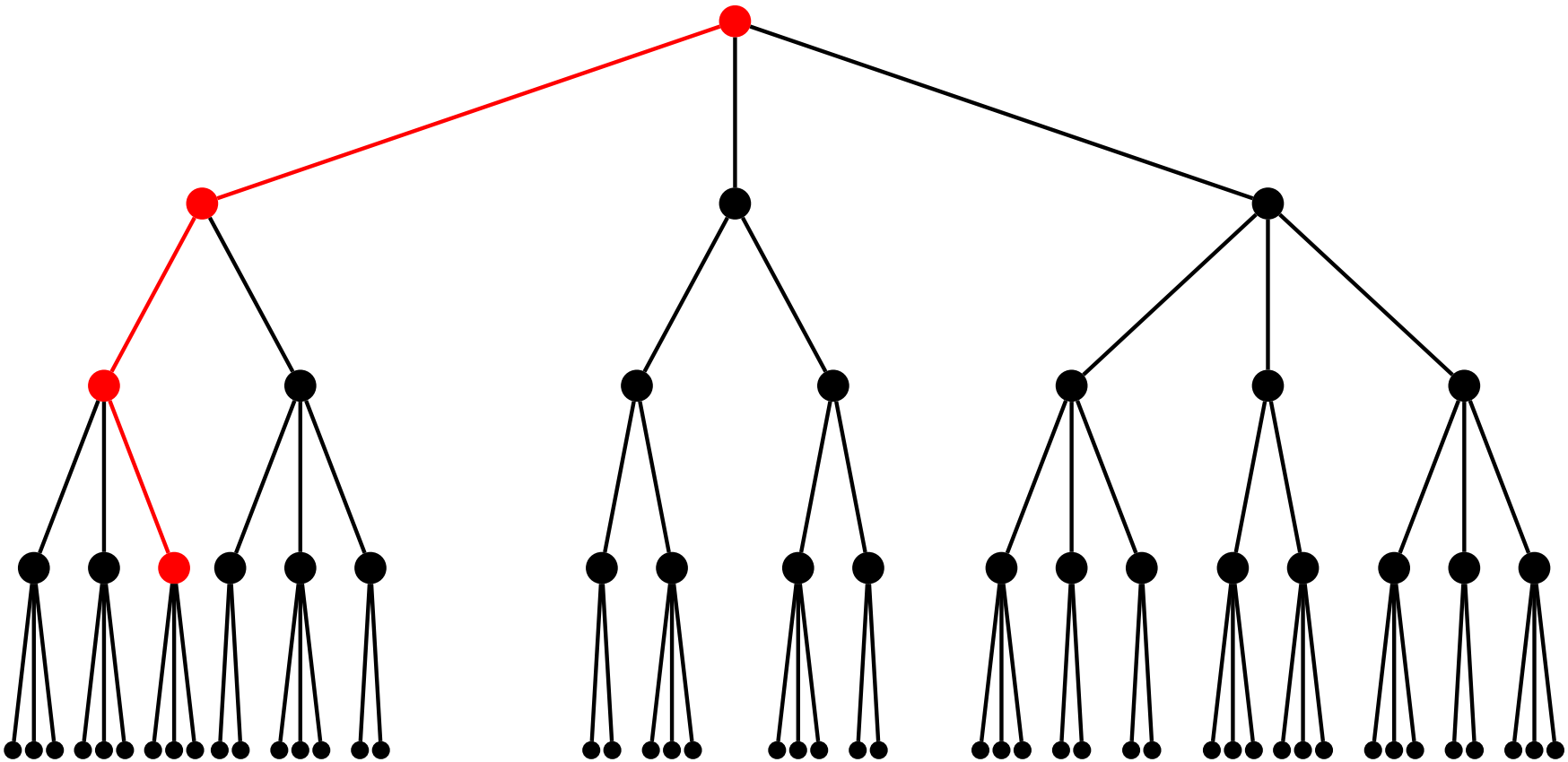


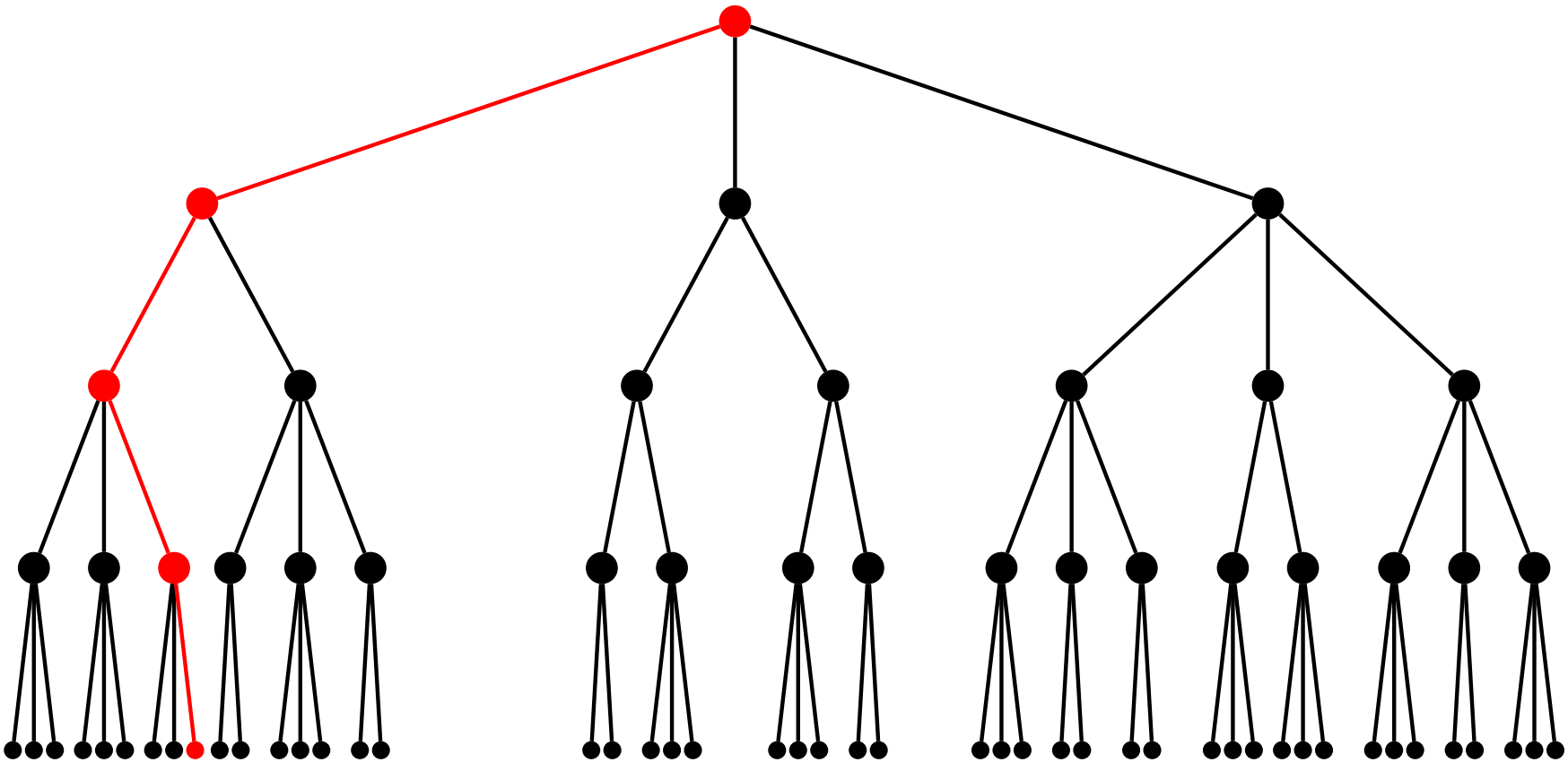


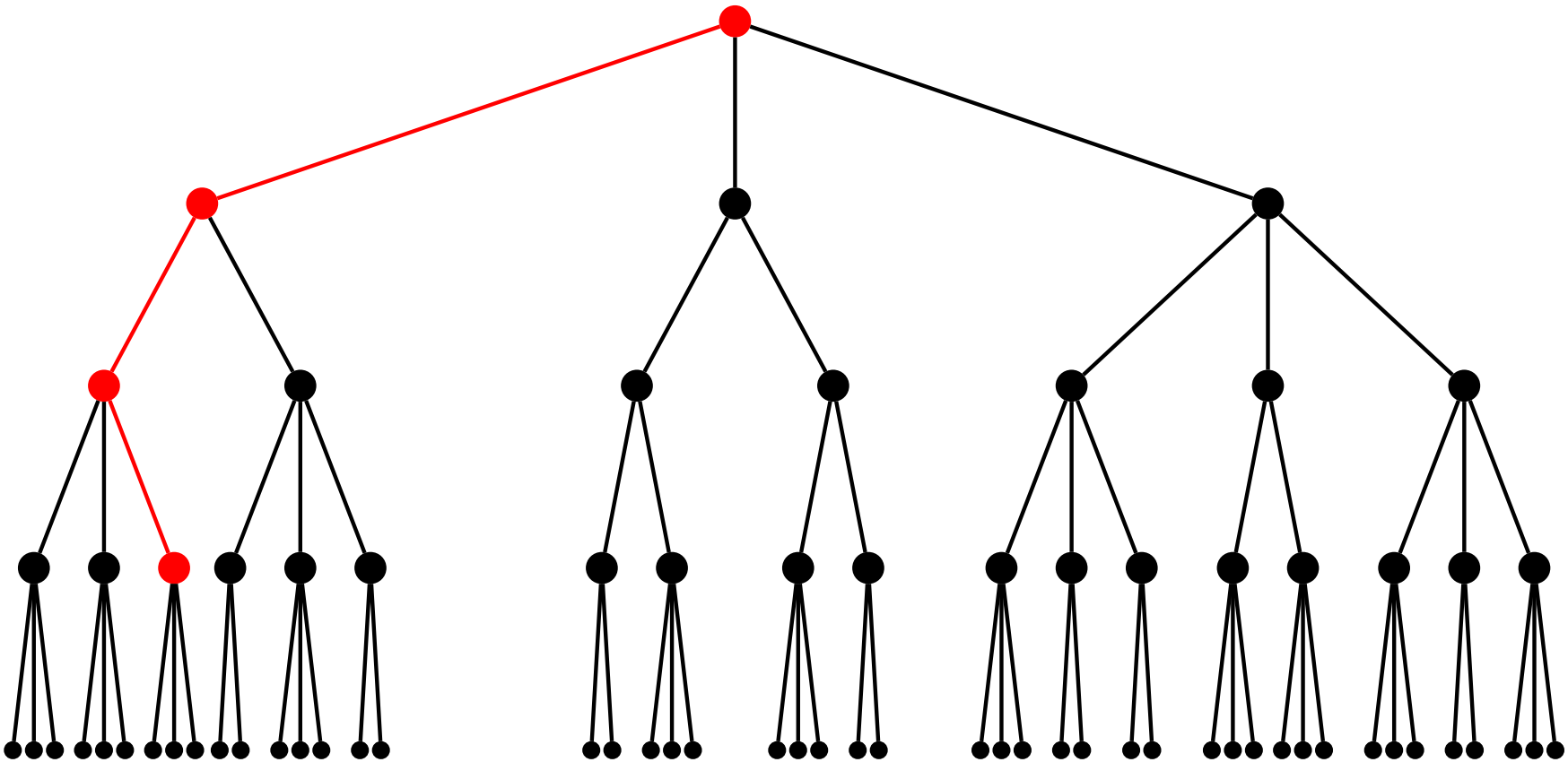


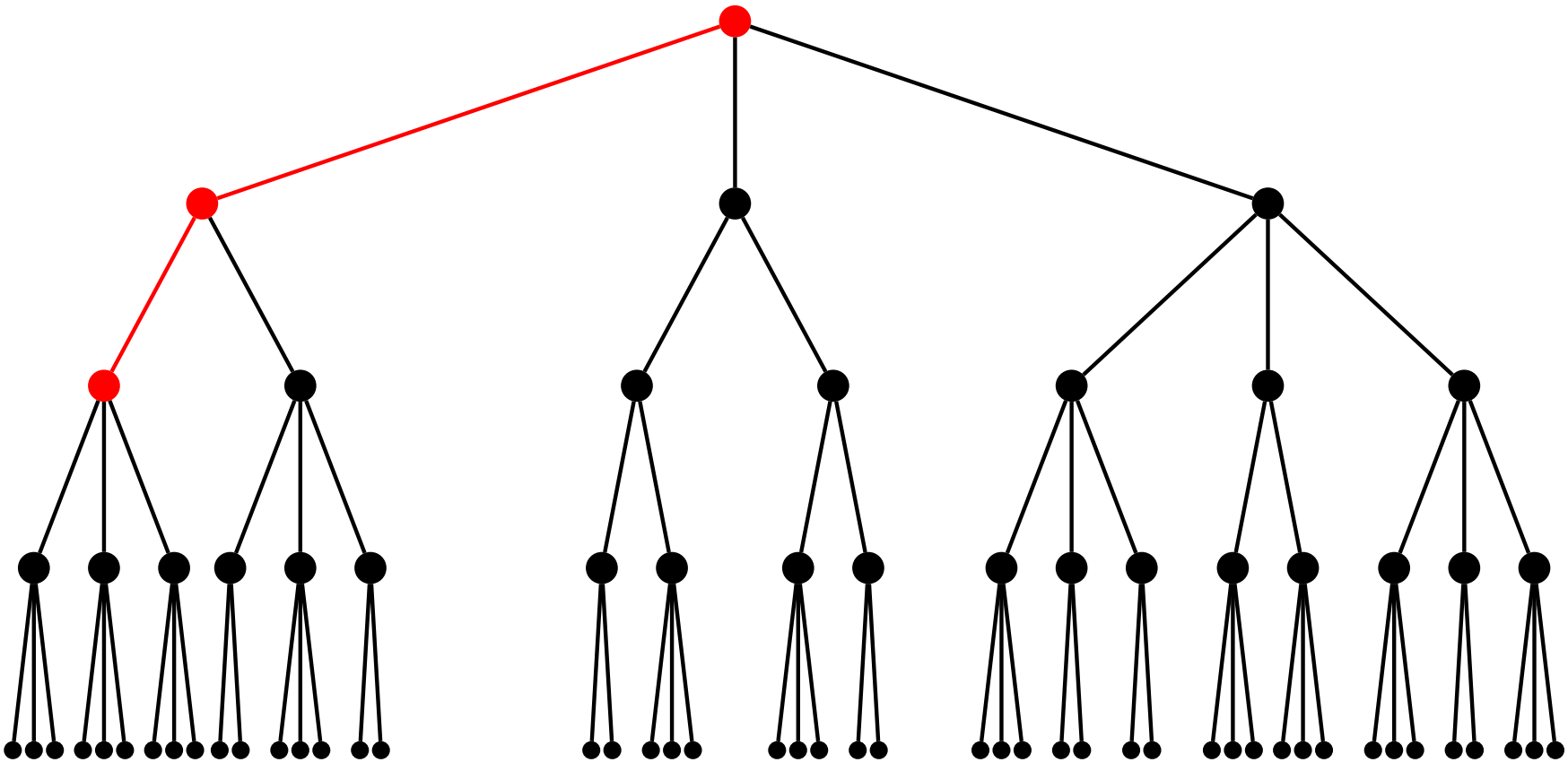


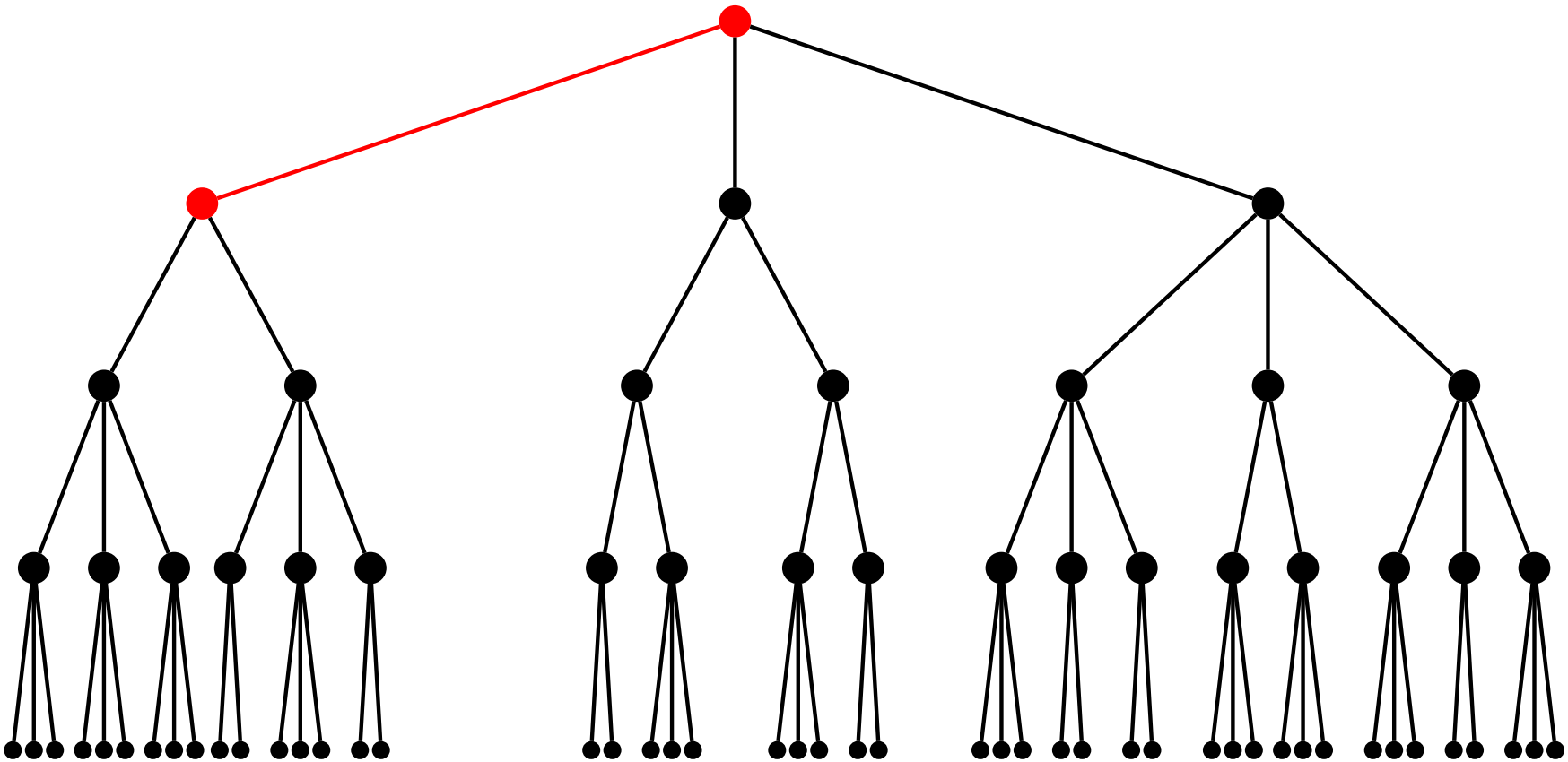


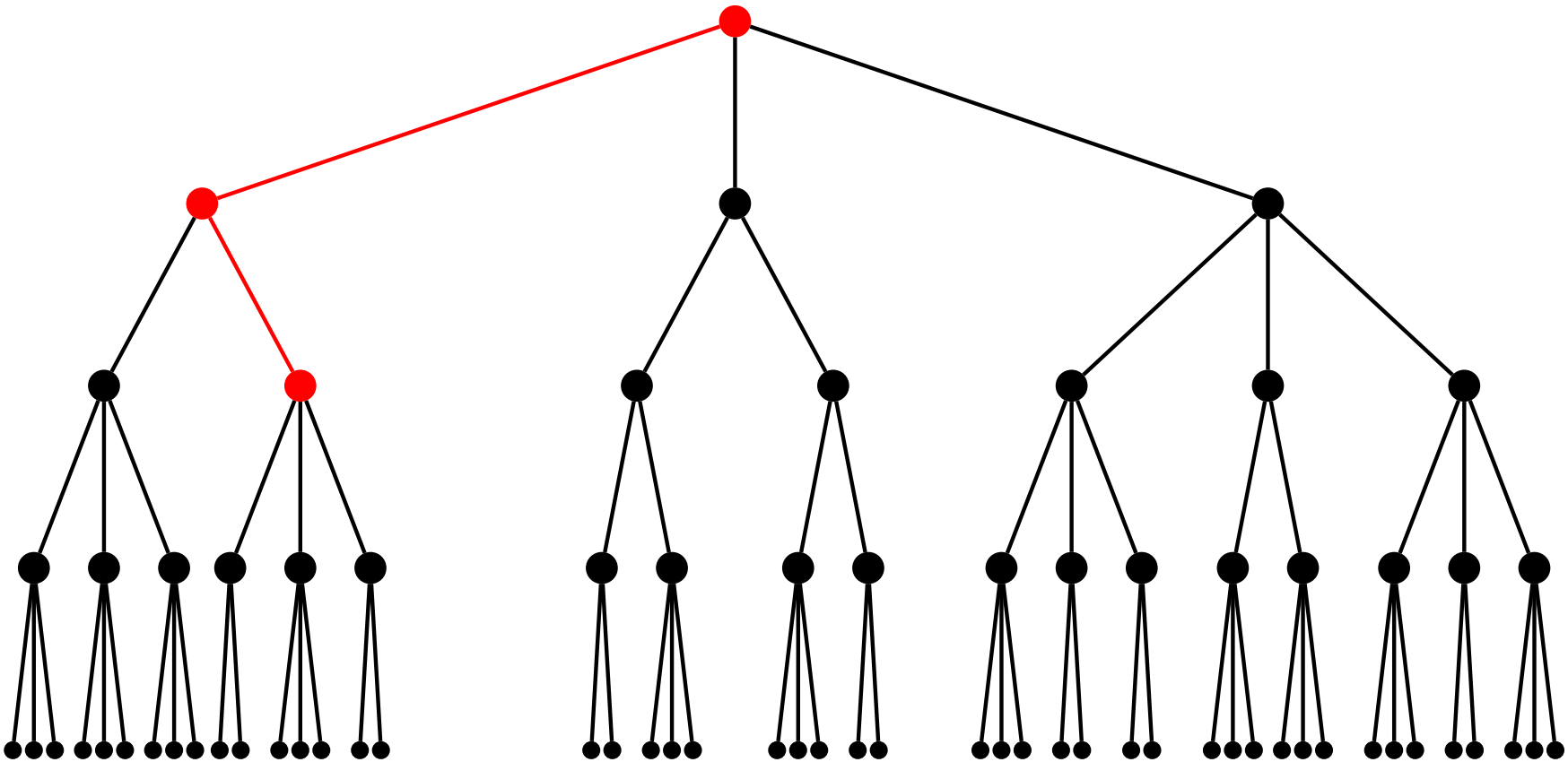


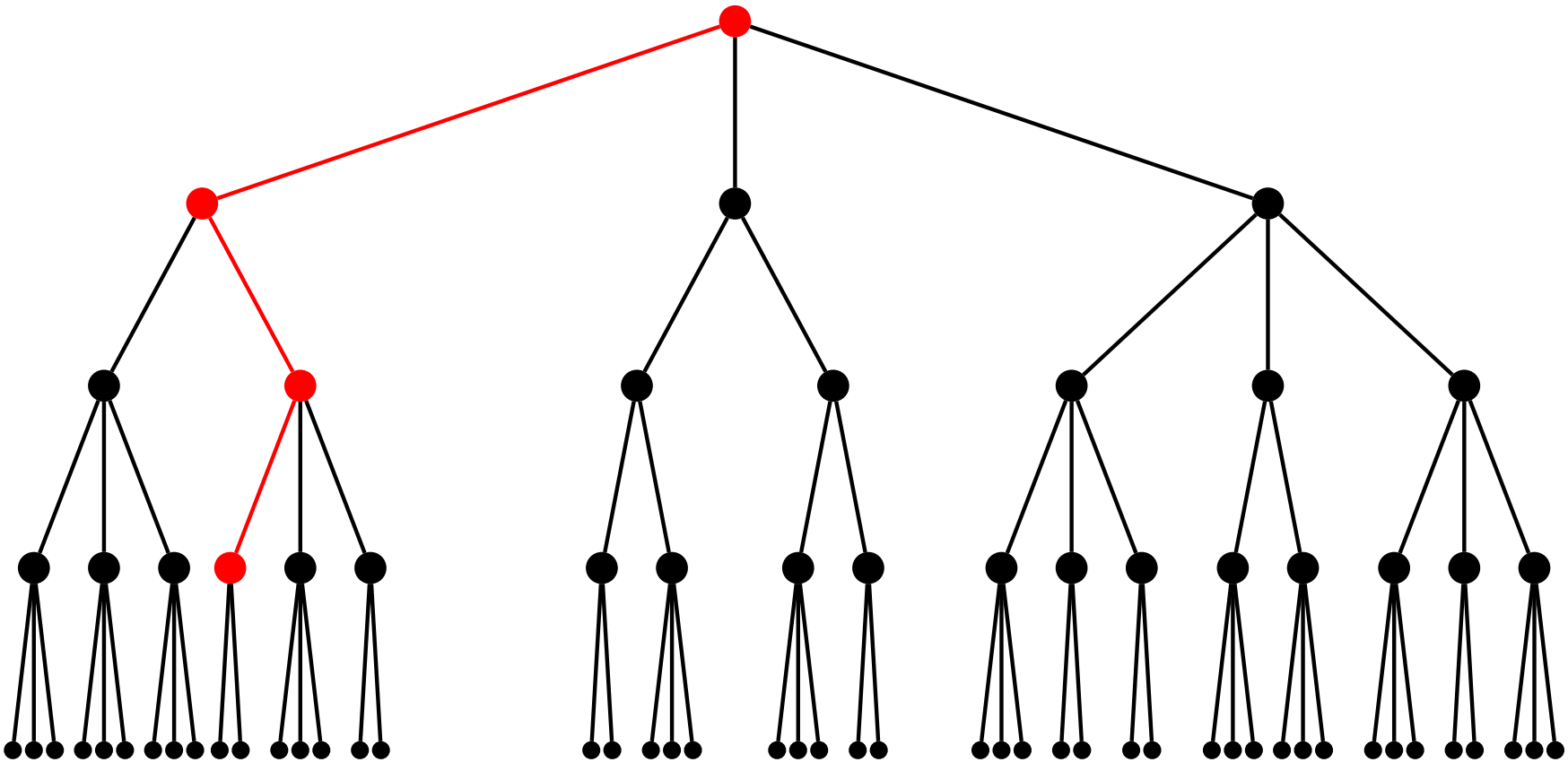


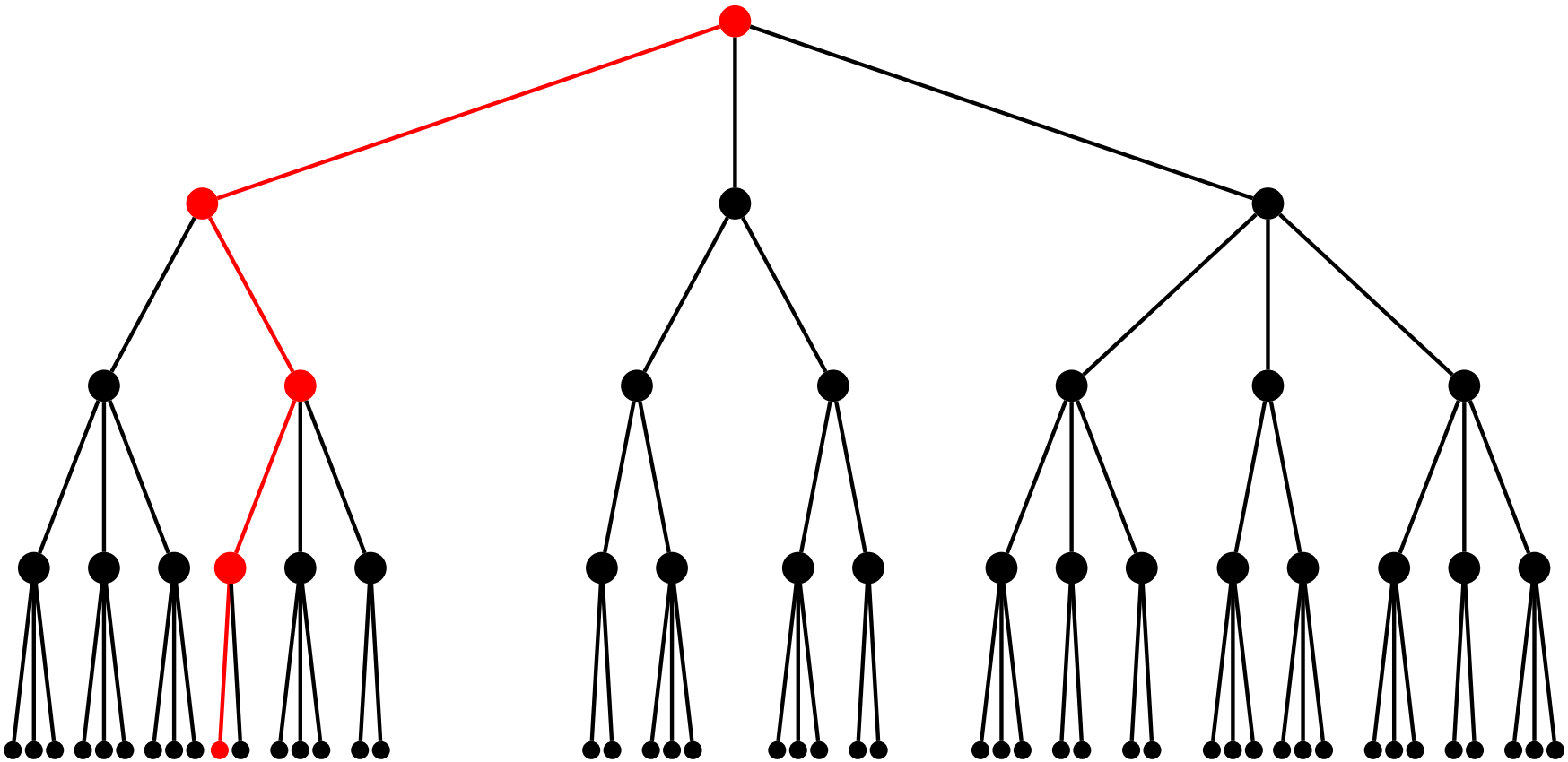


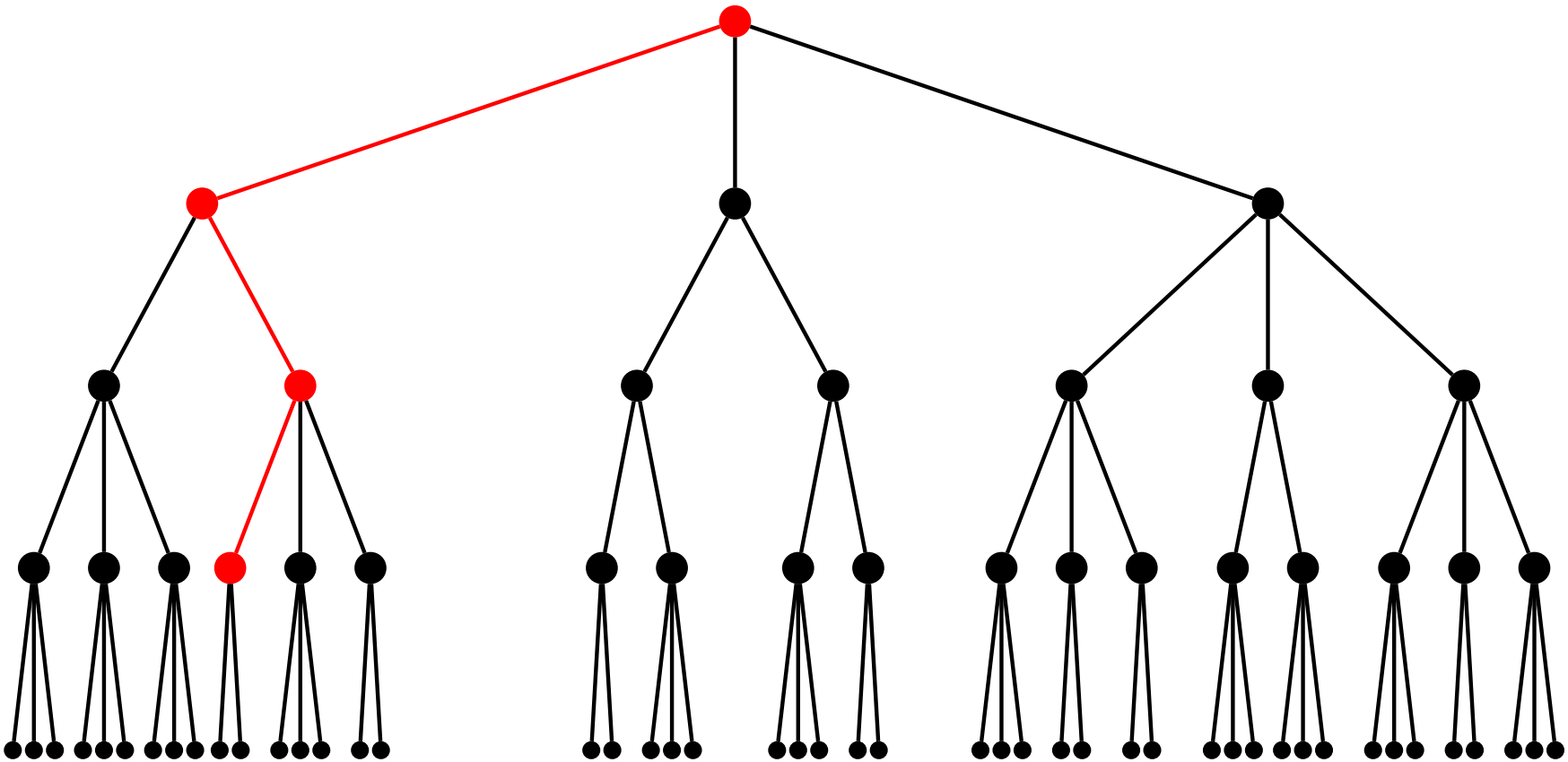


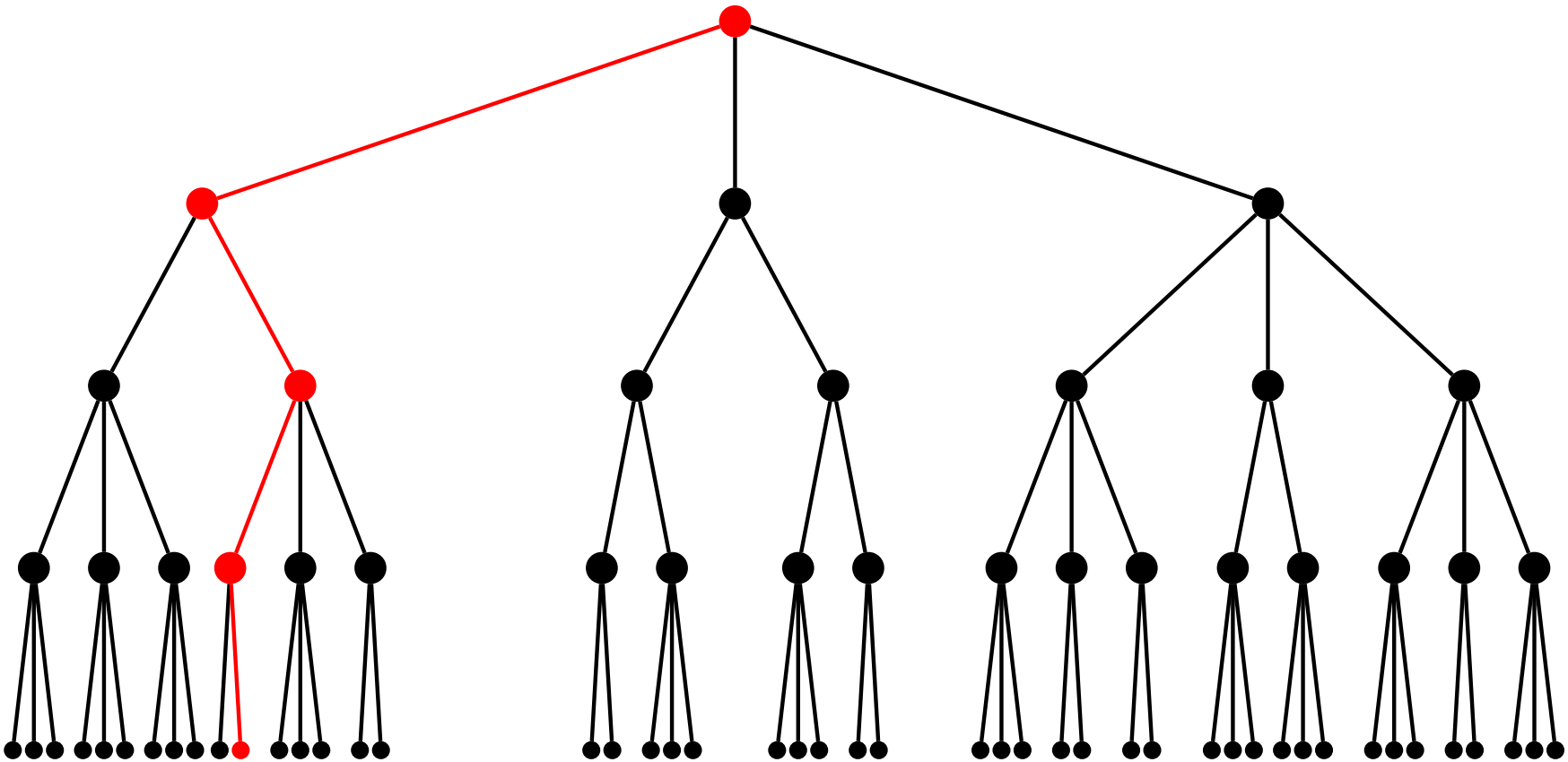


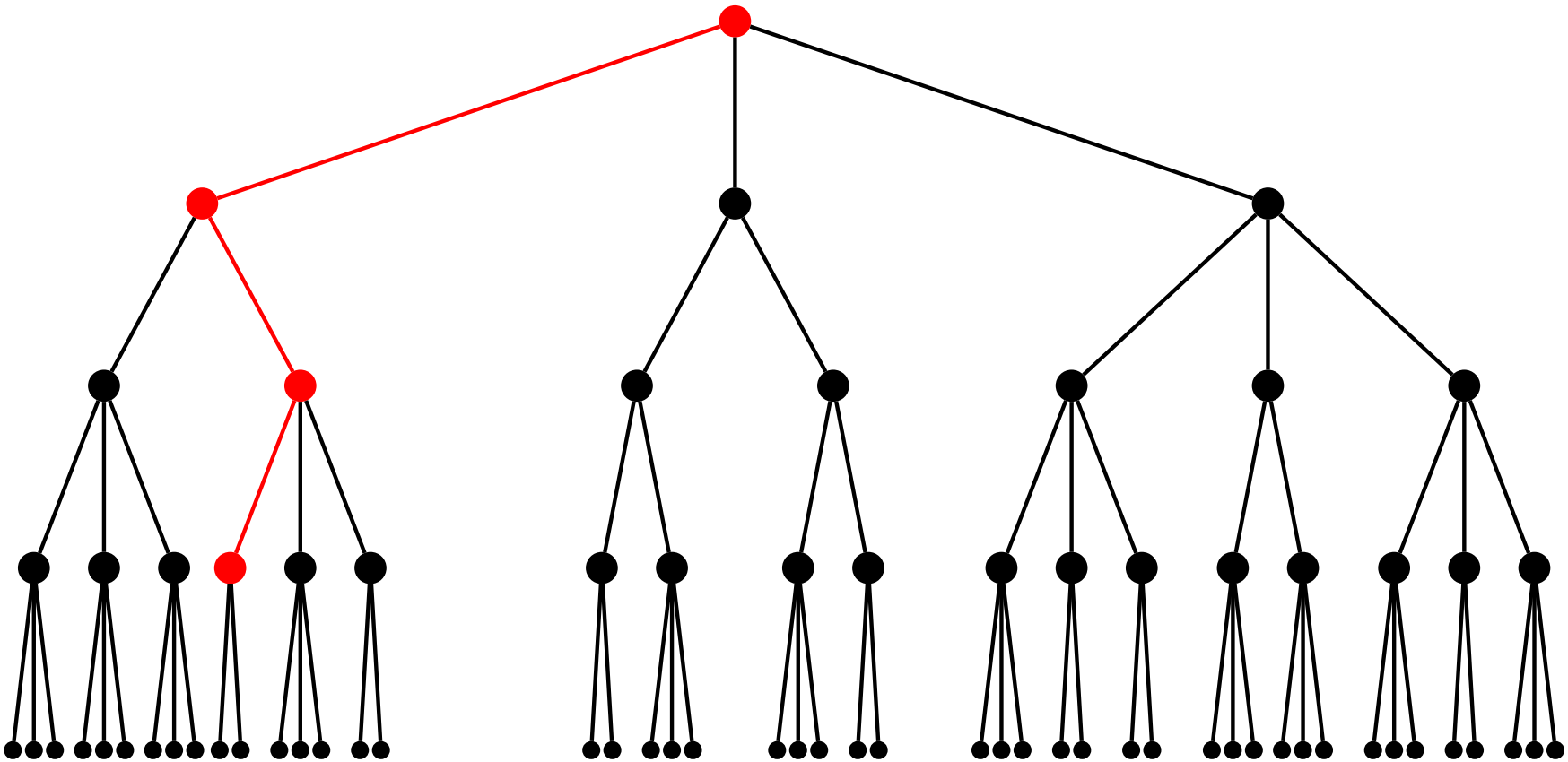


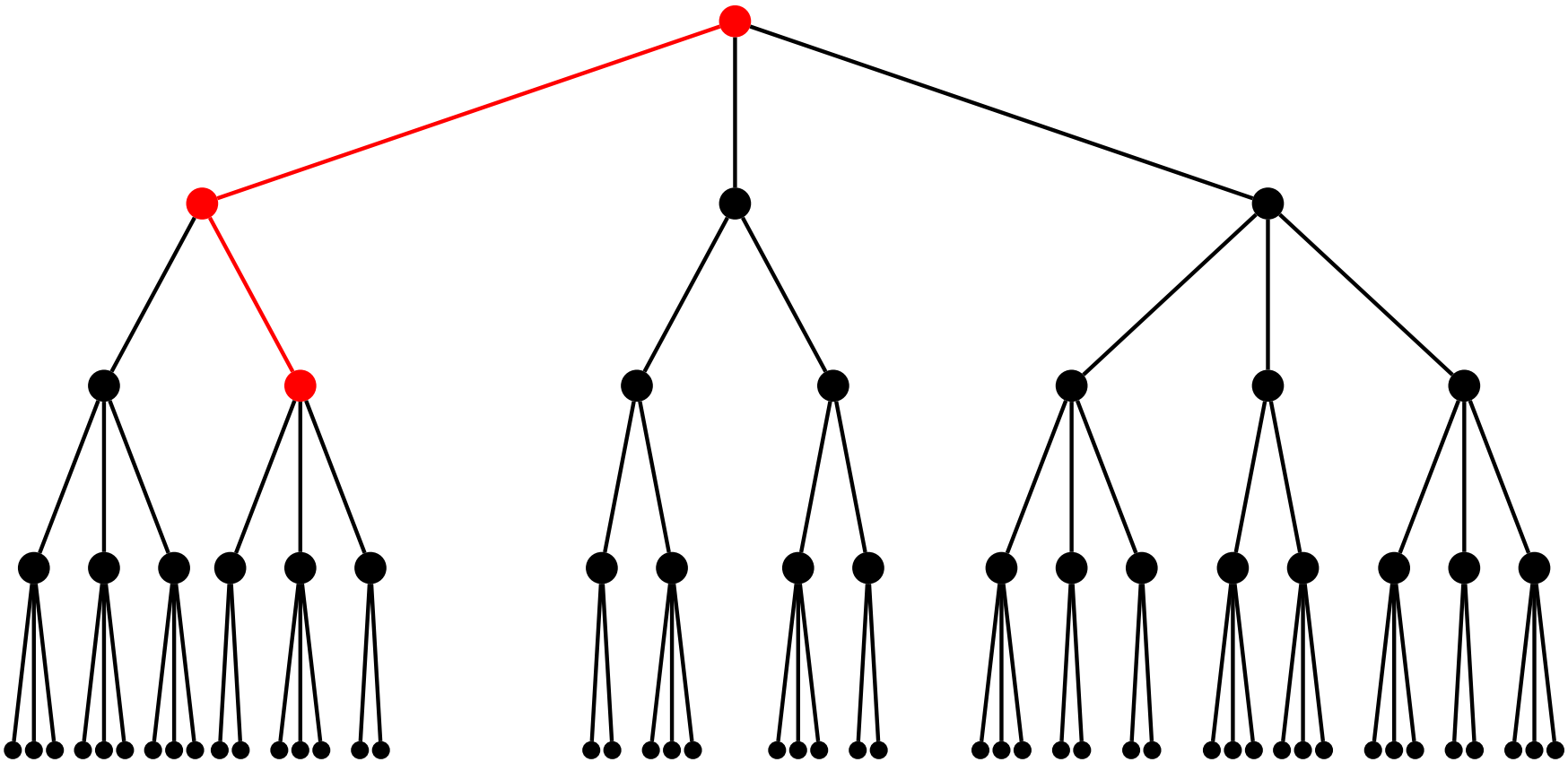


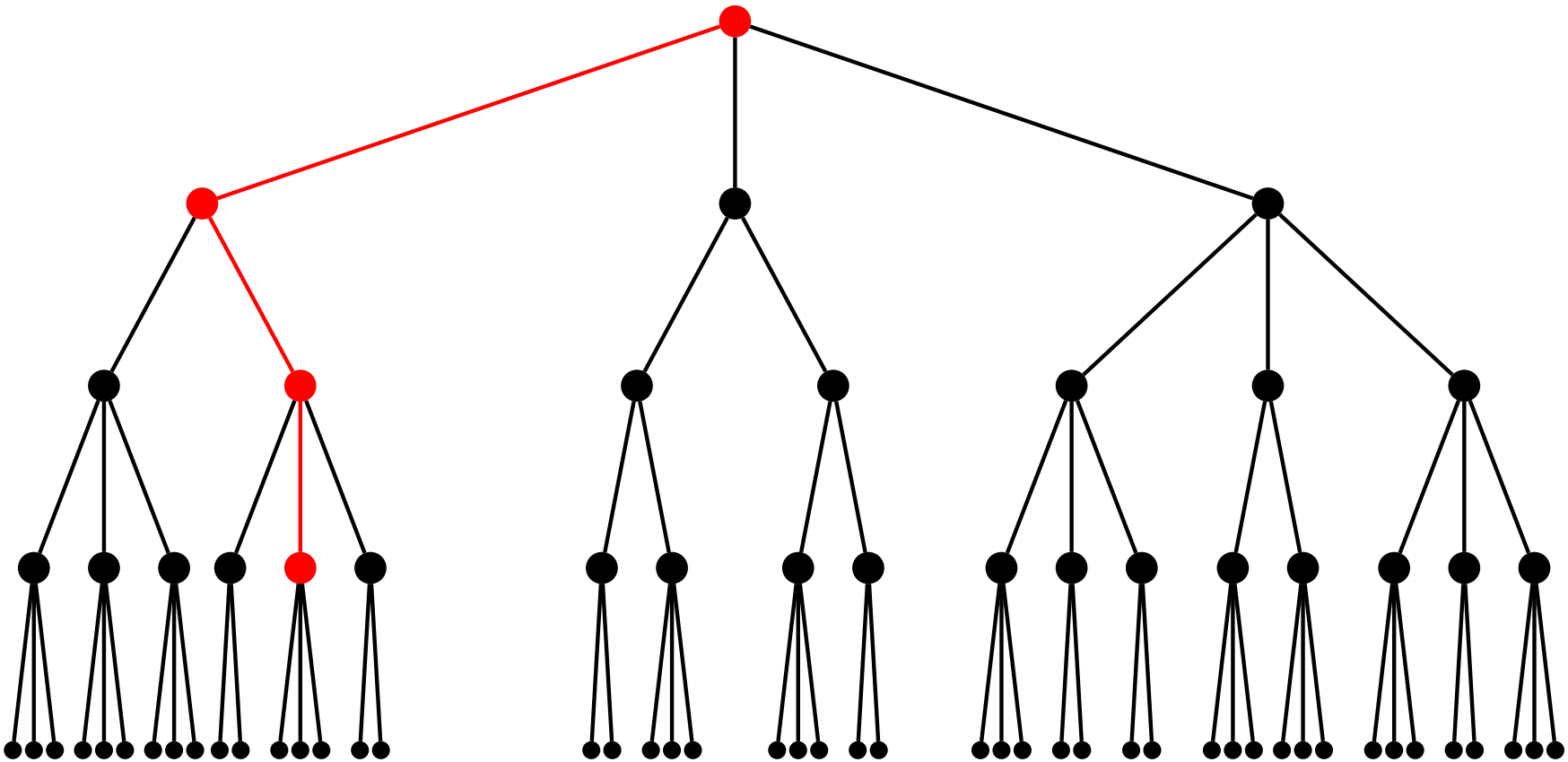






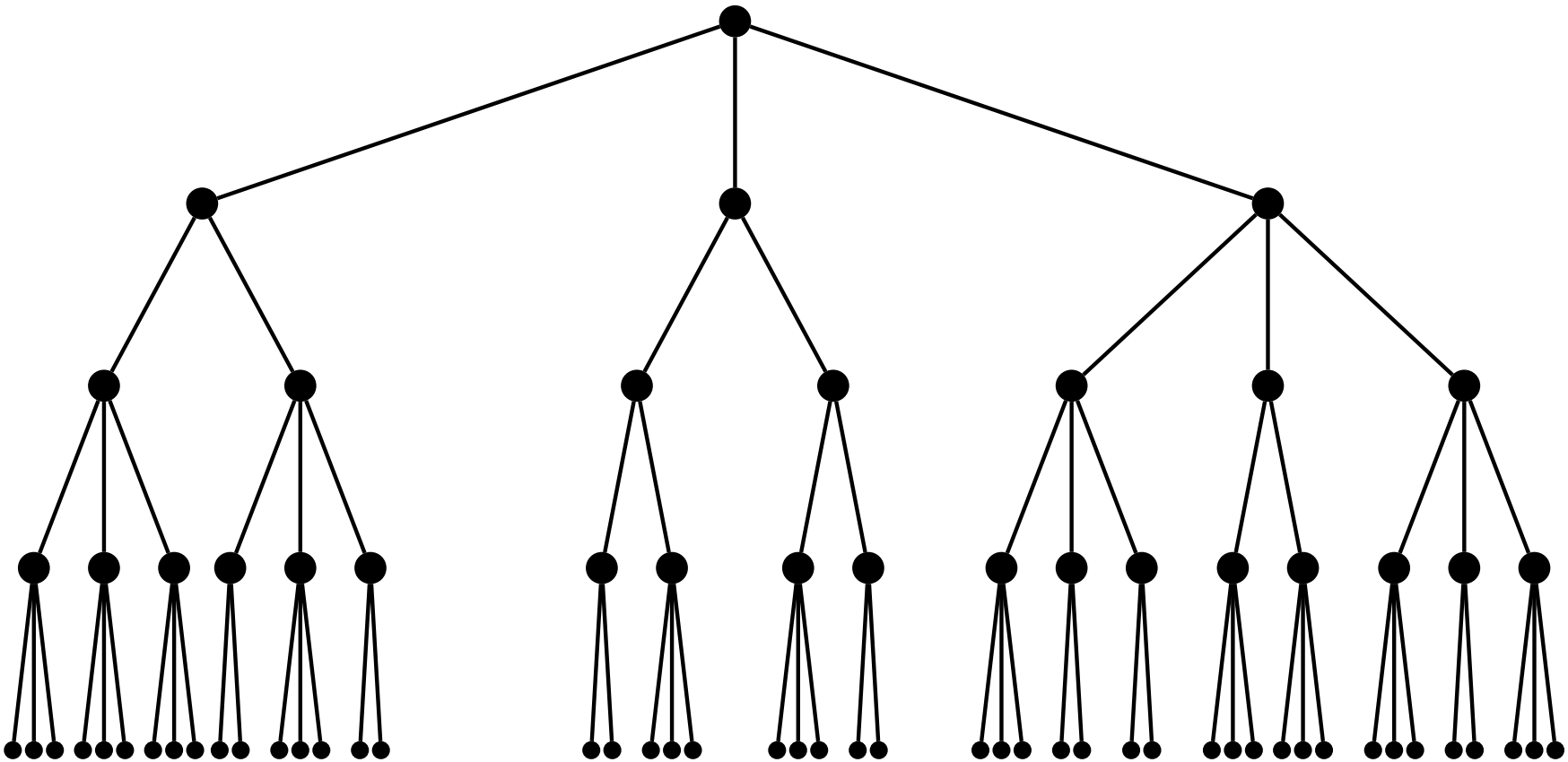


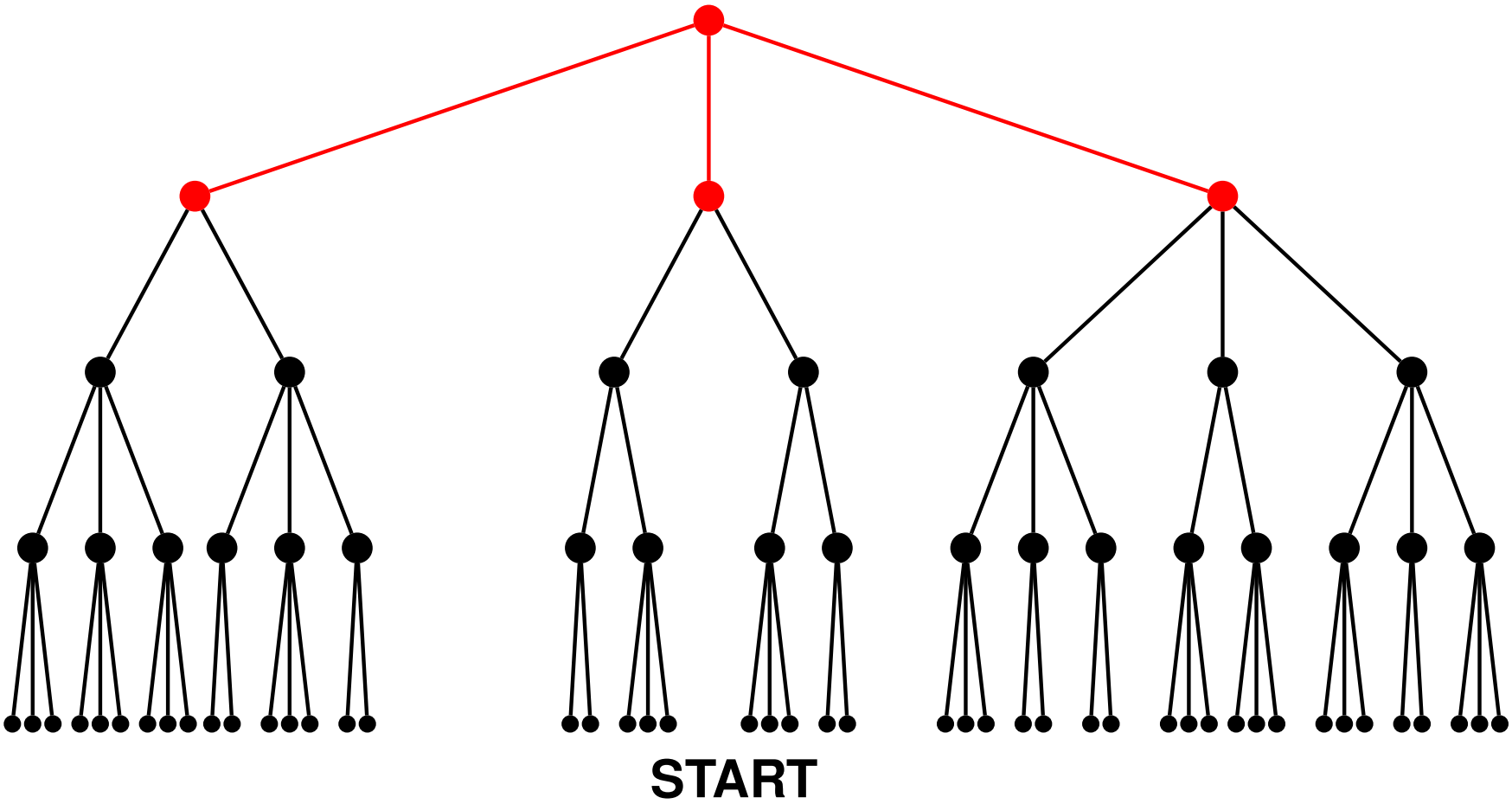


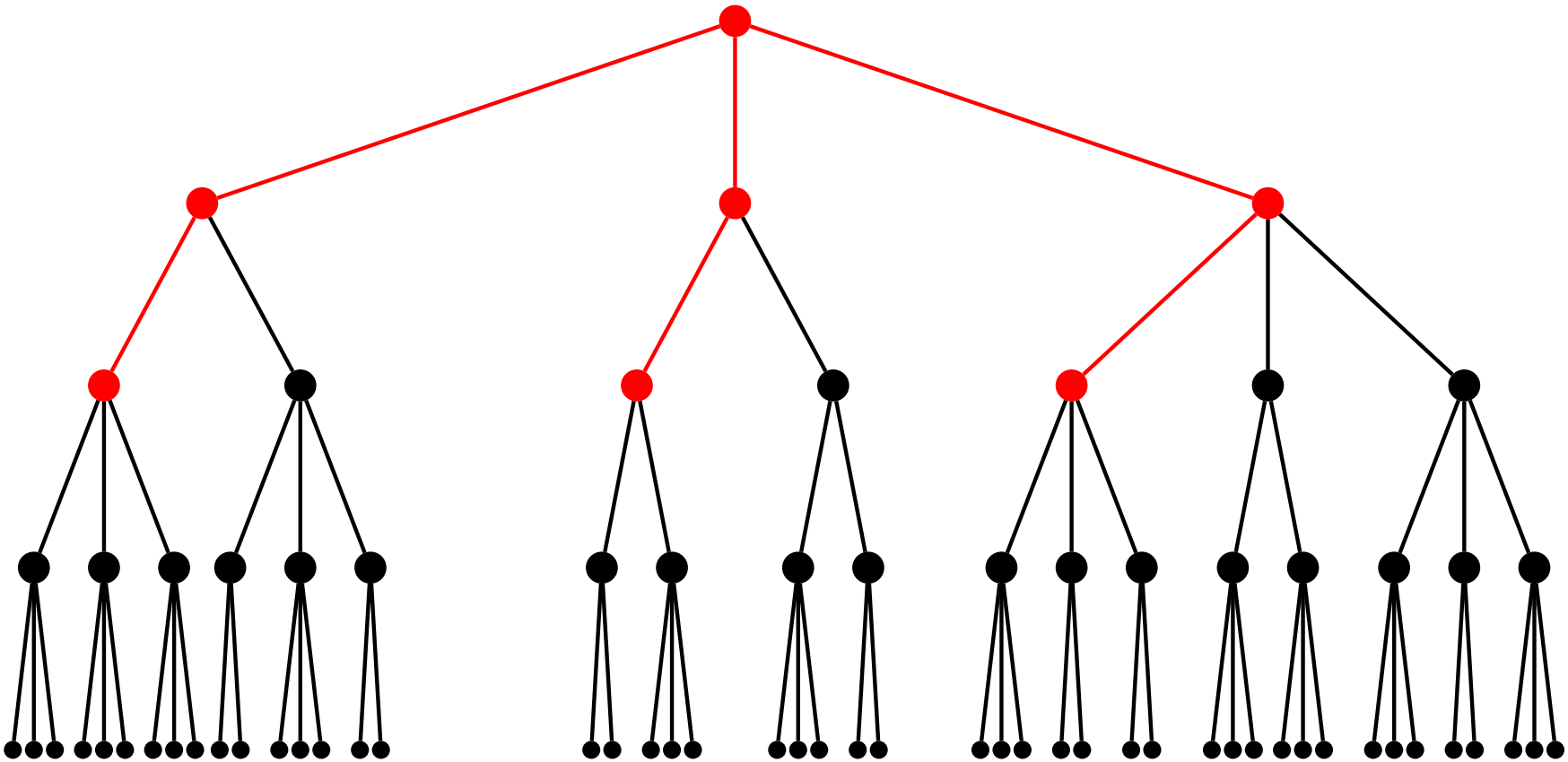


To adapt this problem to GPU some issues need to be resolved:

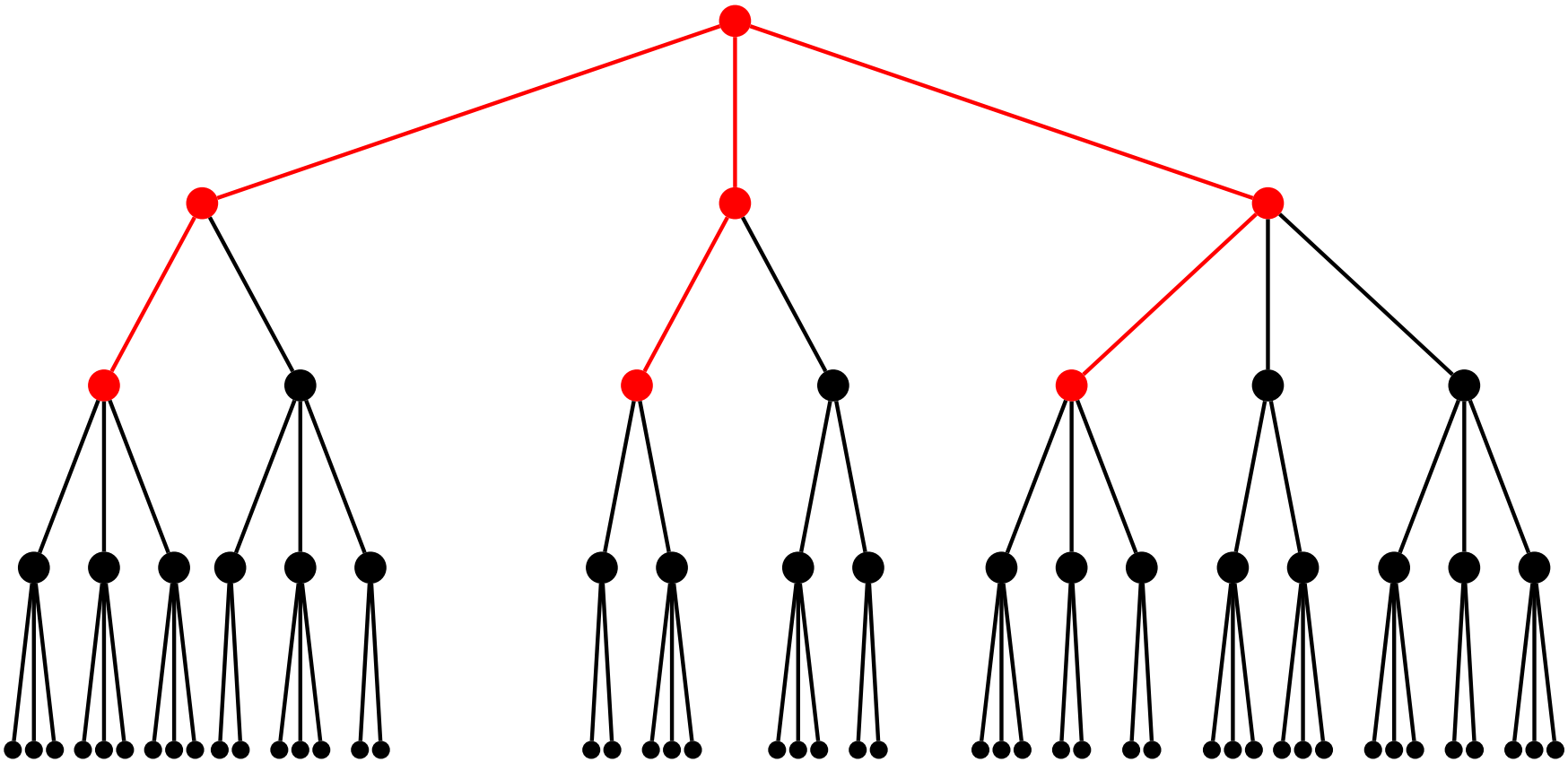
- Parallelize $\Rightarrow$ split backtracking tree at a fixed depth (easy).
- No recursion $\Rightarrow$ stack implementation.
- SIMT architecture (Single Instruction Multiple Thread) $\Rightarrow$ need to eliminate branching, as much as possible.
- SIMT $\Rightarrow$ ensure most threads doing useful work most of the time.
- Memory model $\Rightarrow$ for this application need to fit all variables in shared memory (effectively 512B) and registers.

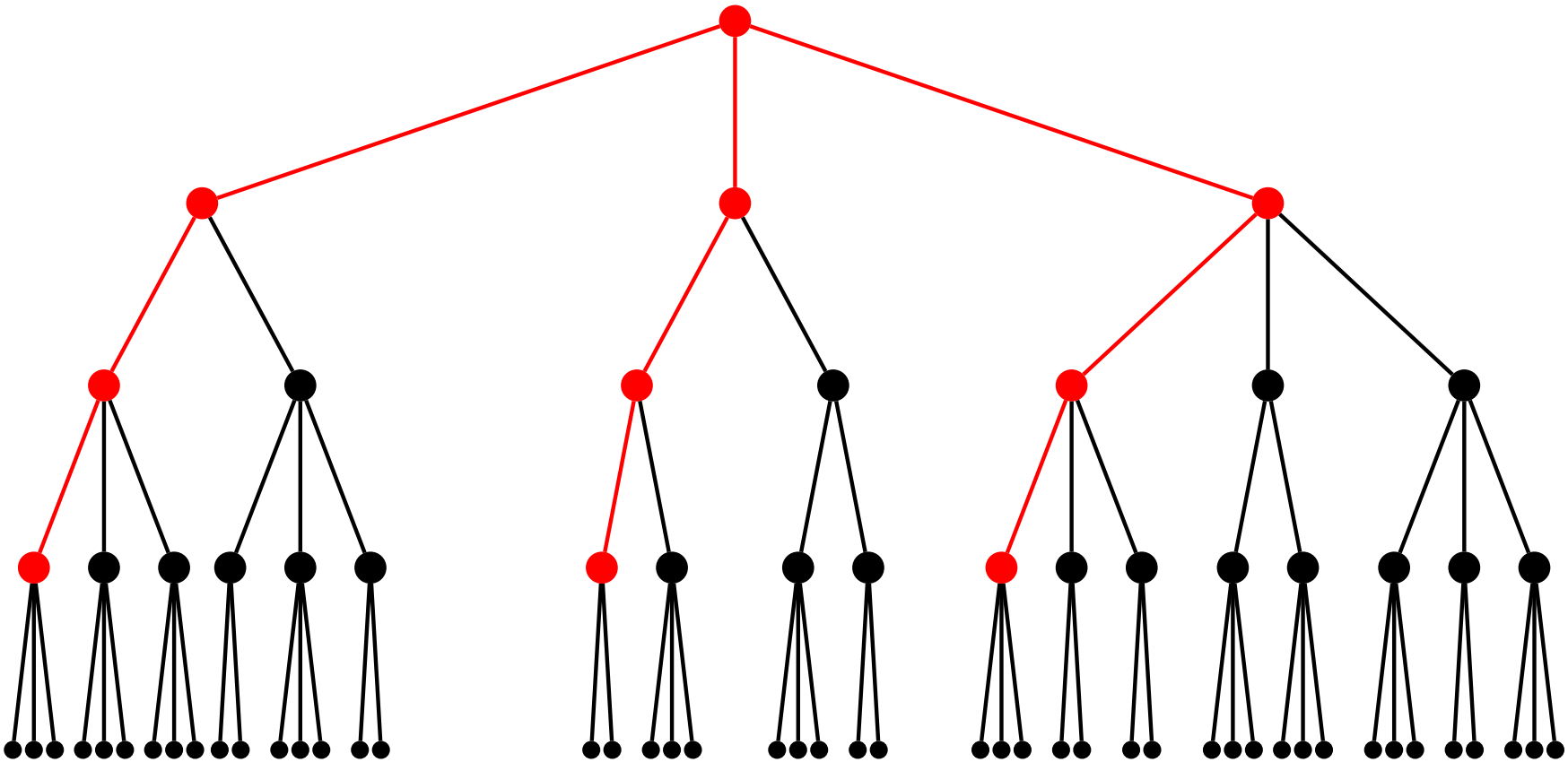



START

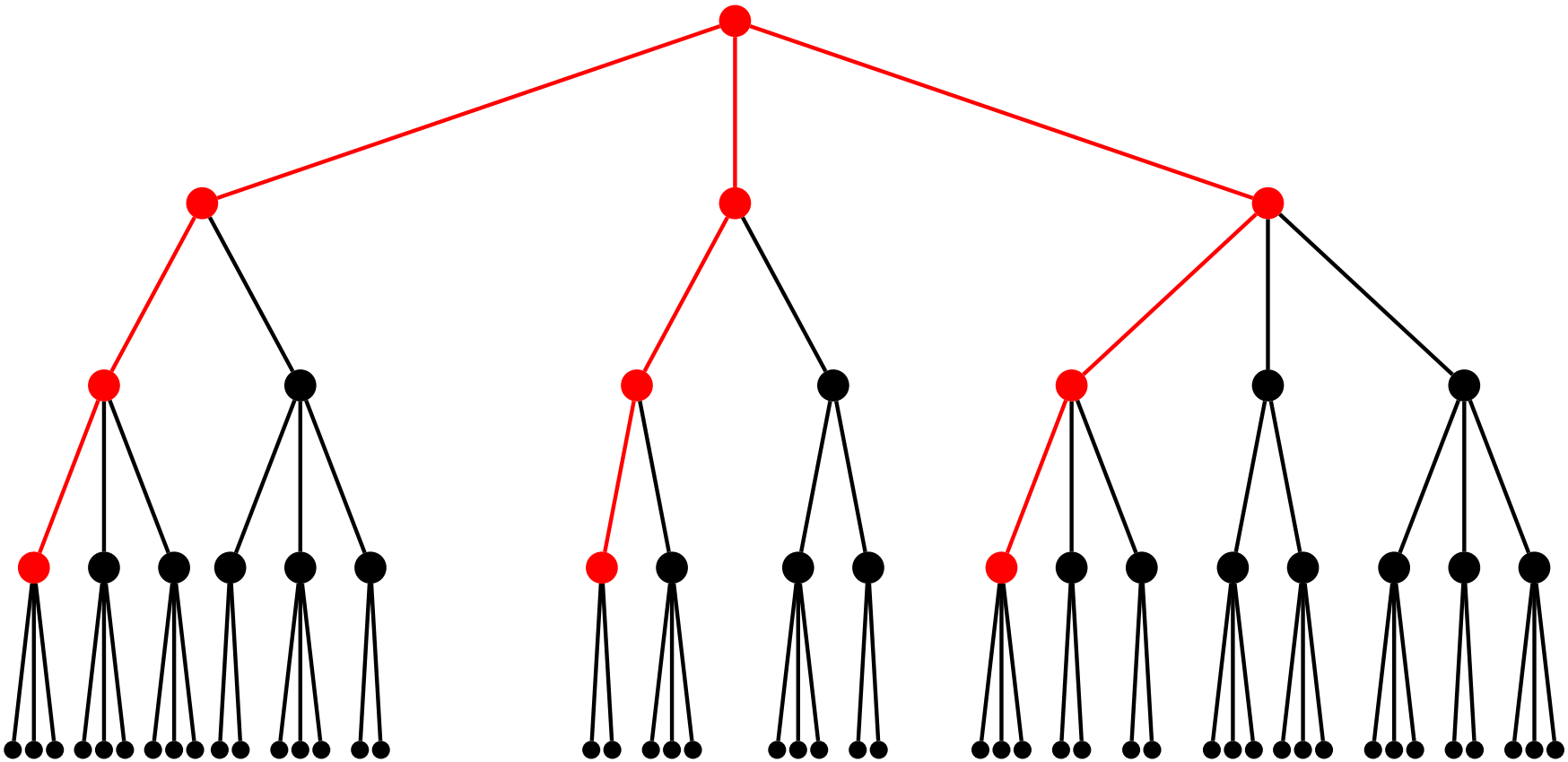


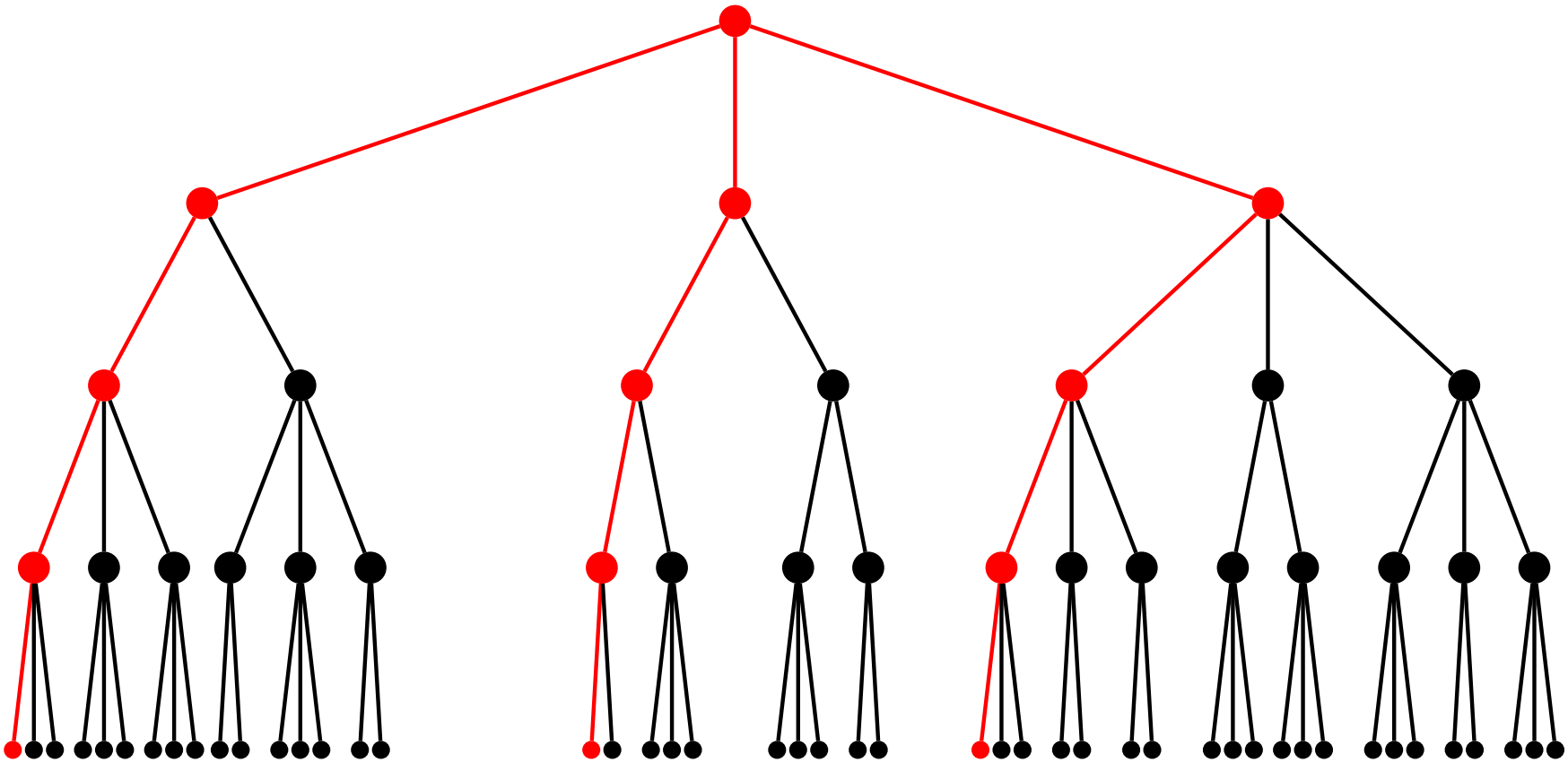
BUILD


REMOVE

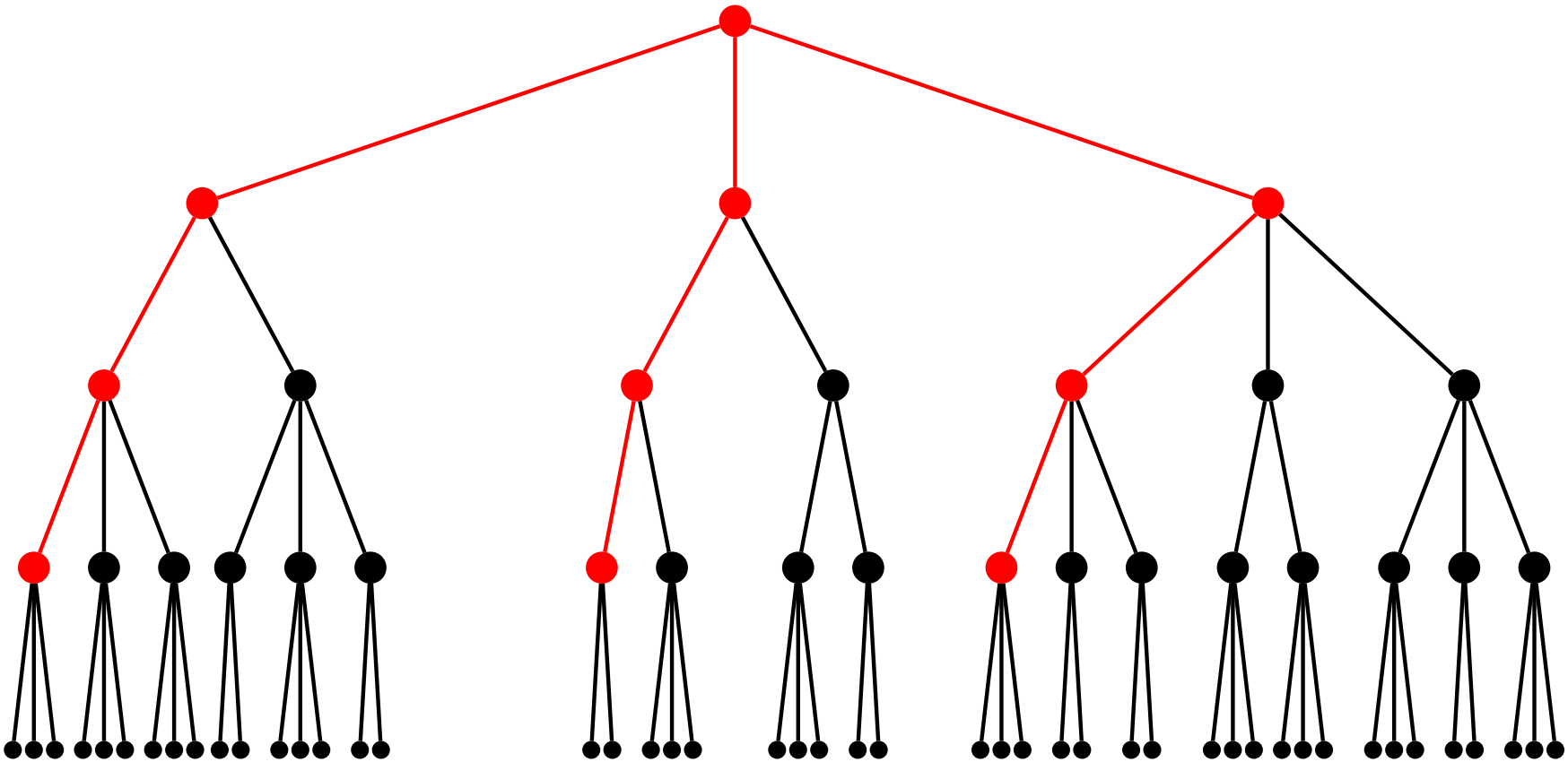


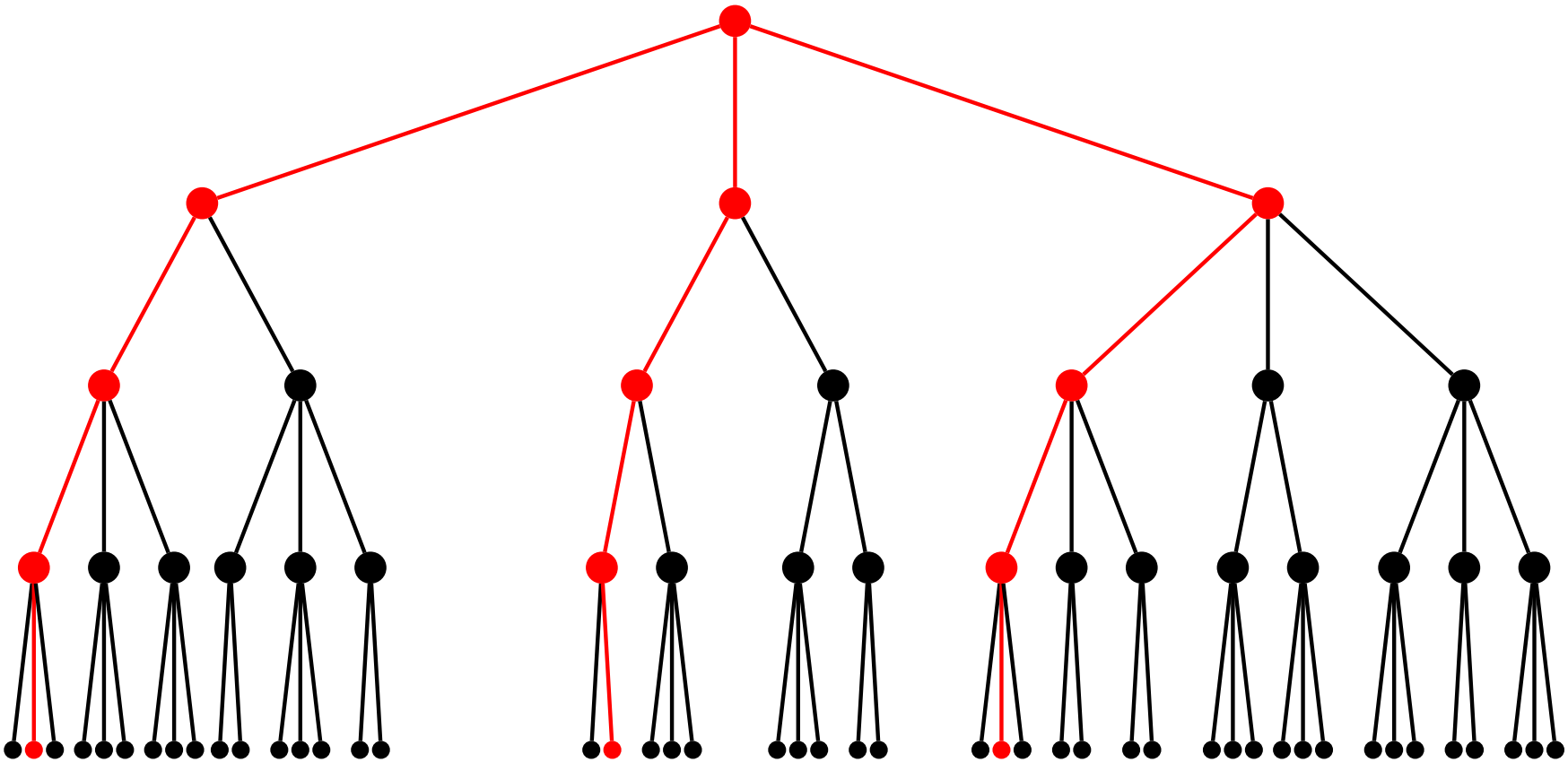
BUILD


REMOVE

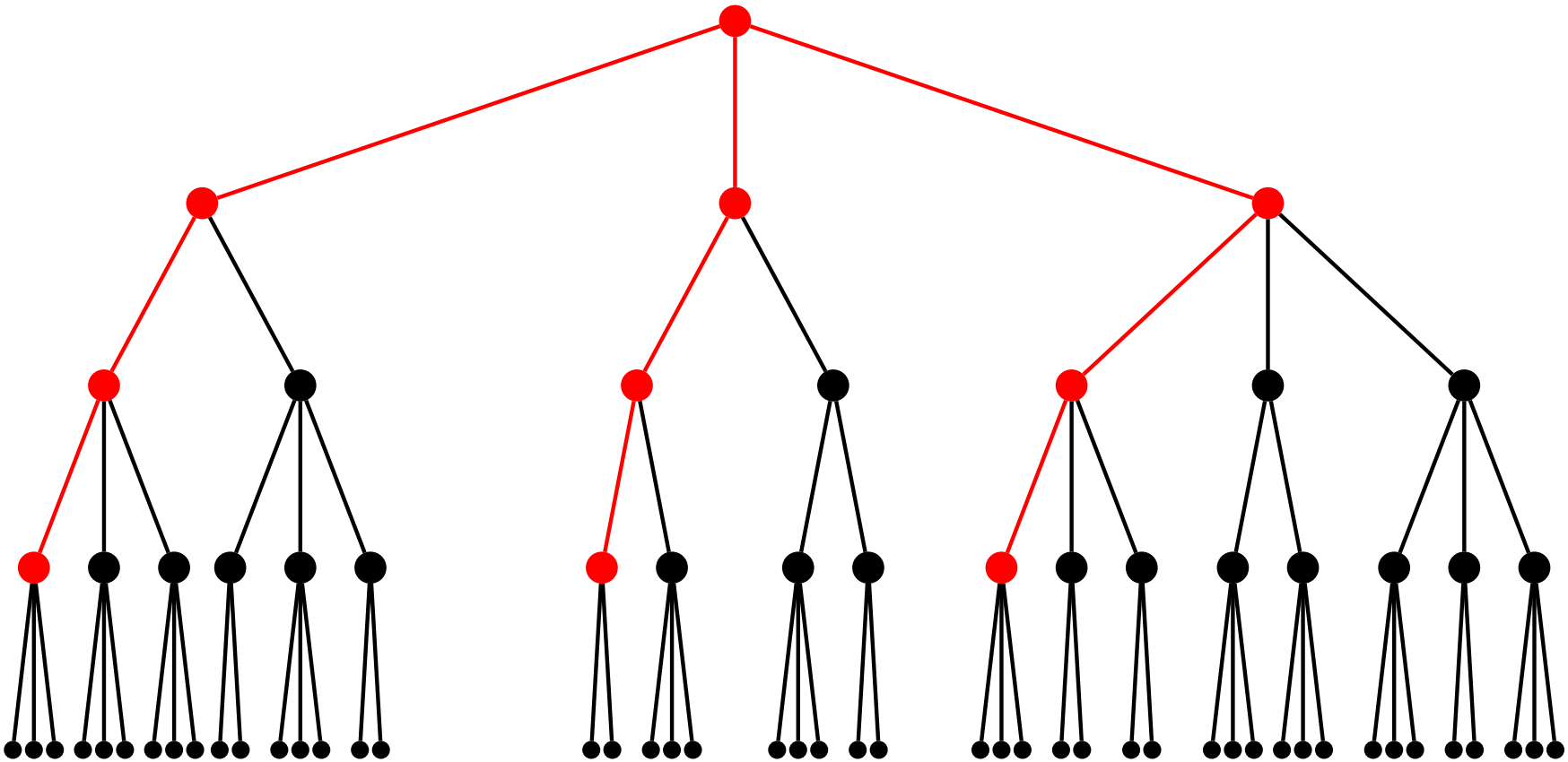


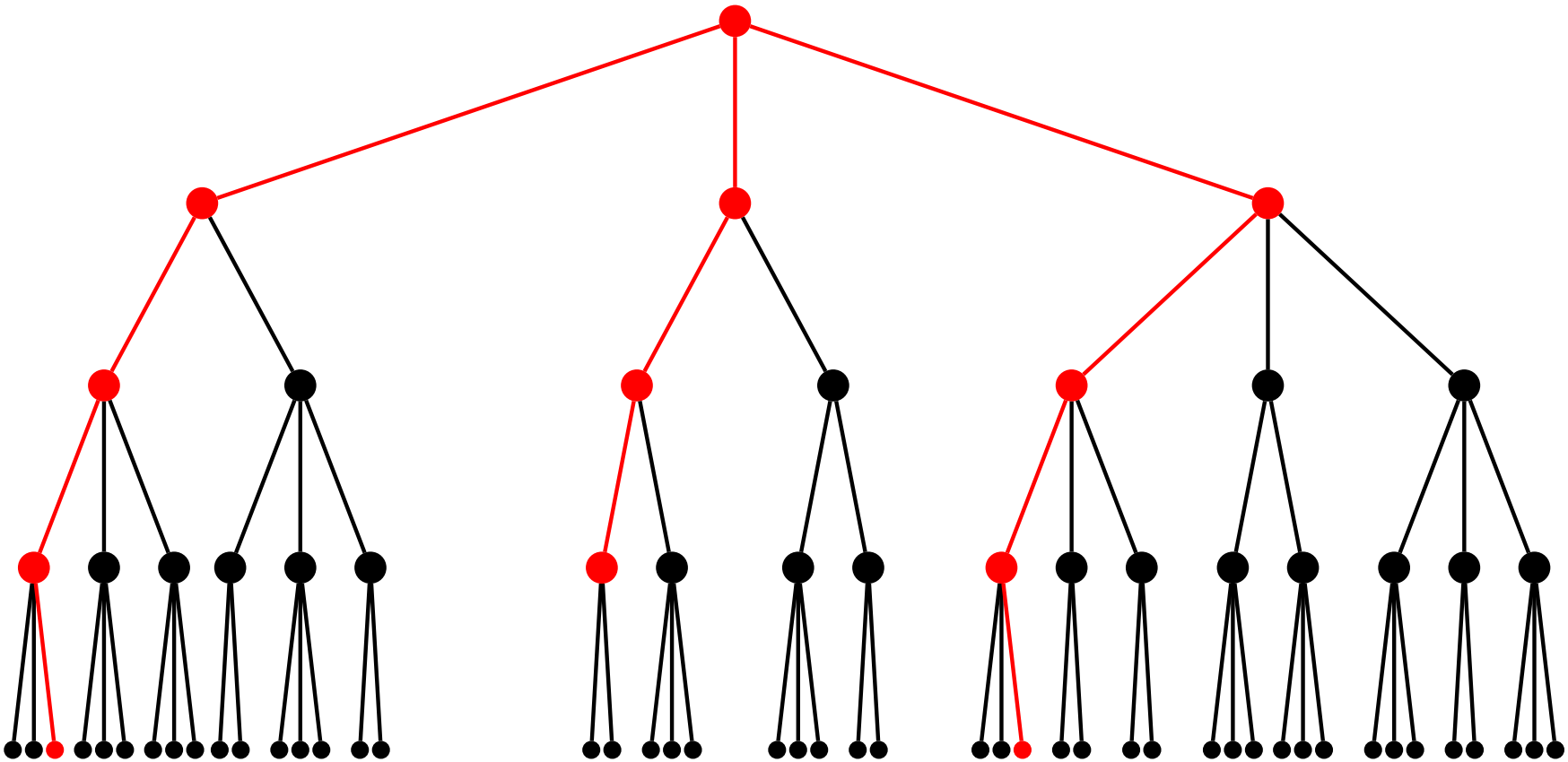
BUILD


REMOVE

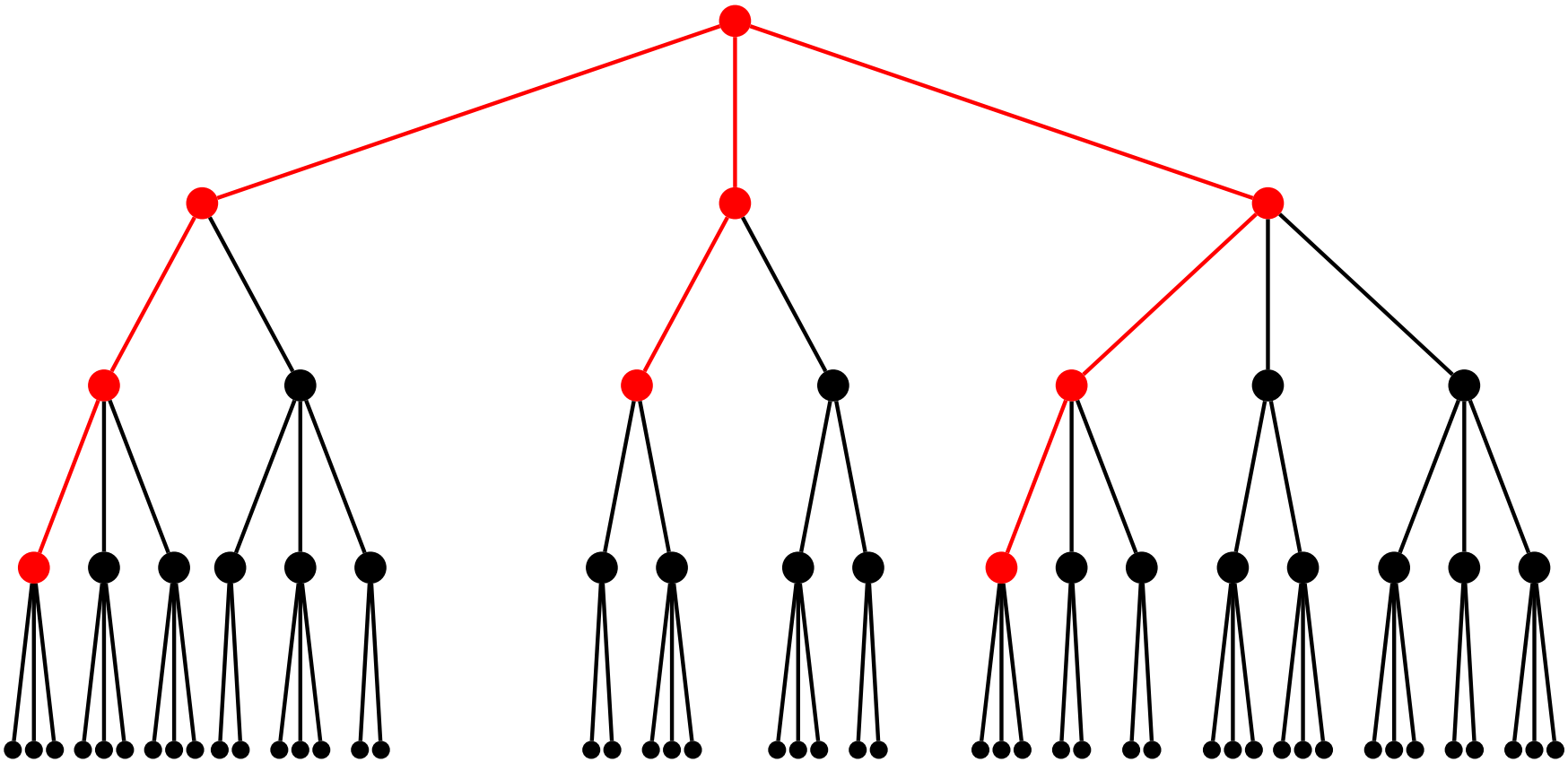


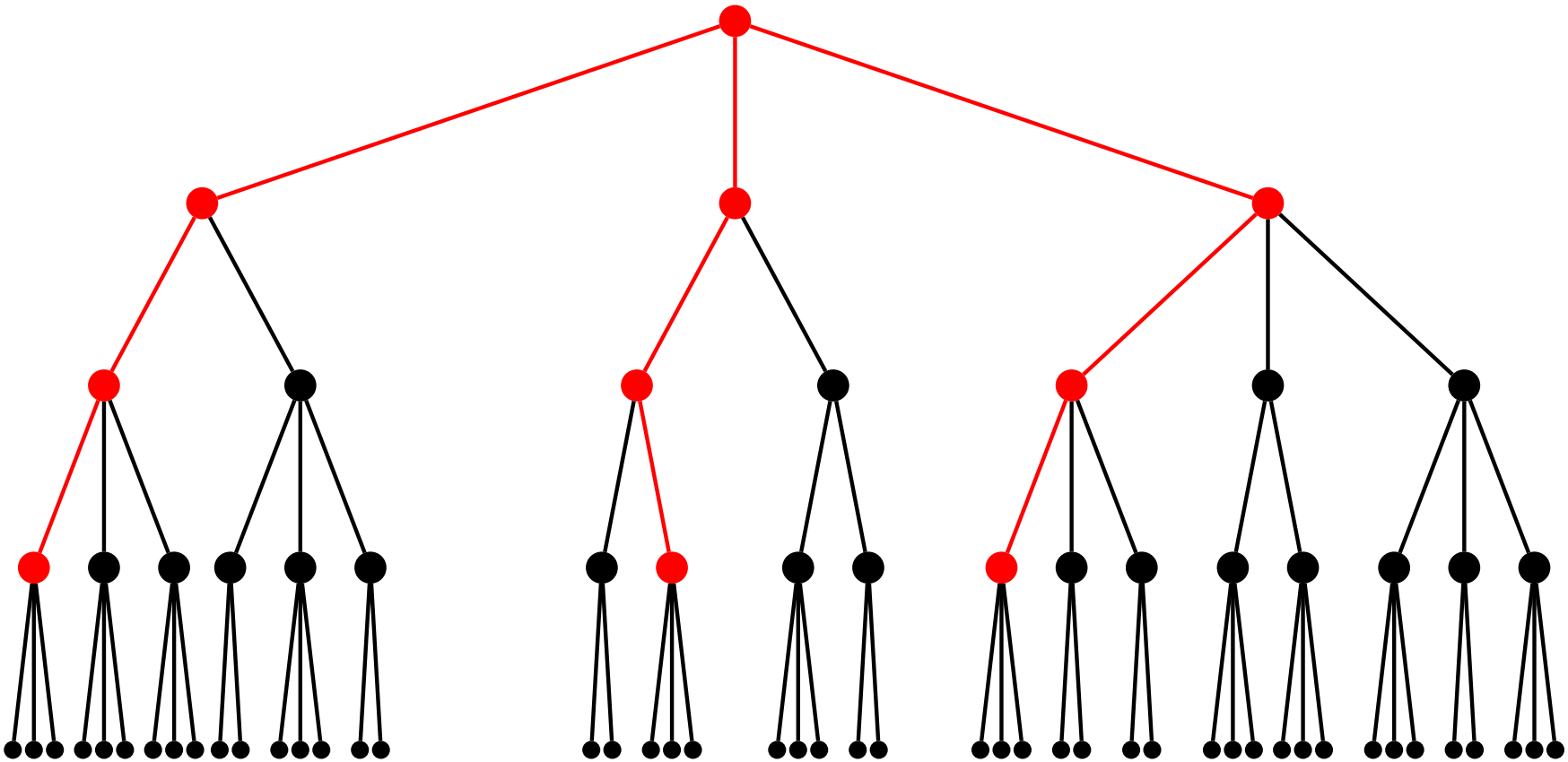
BUILD


REMOVE

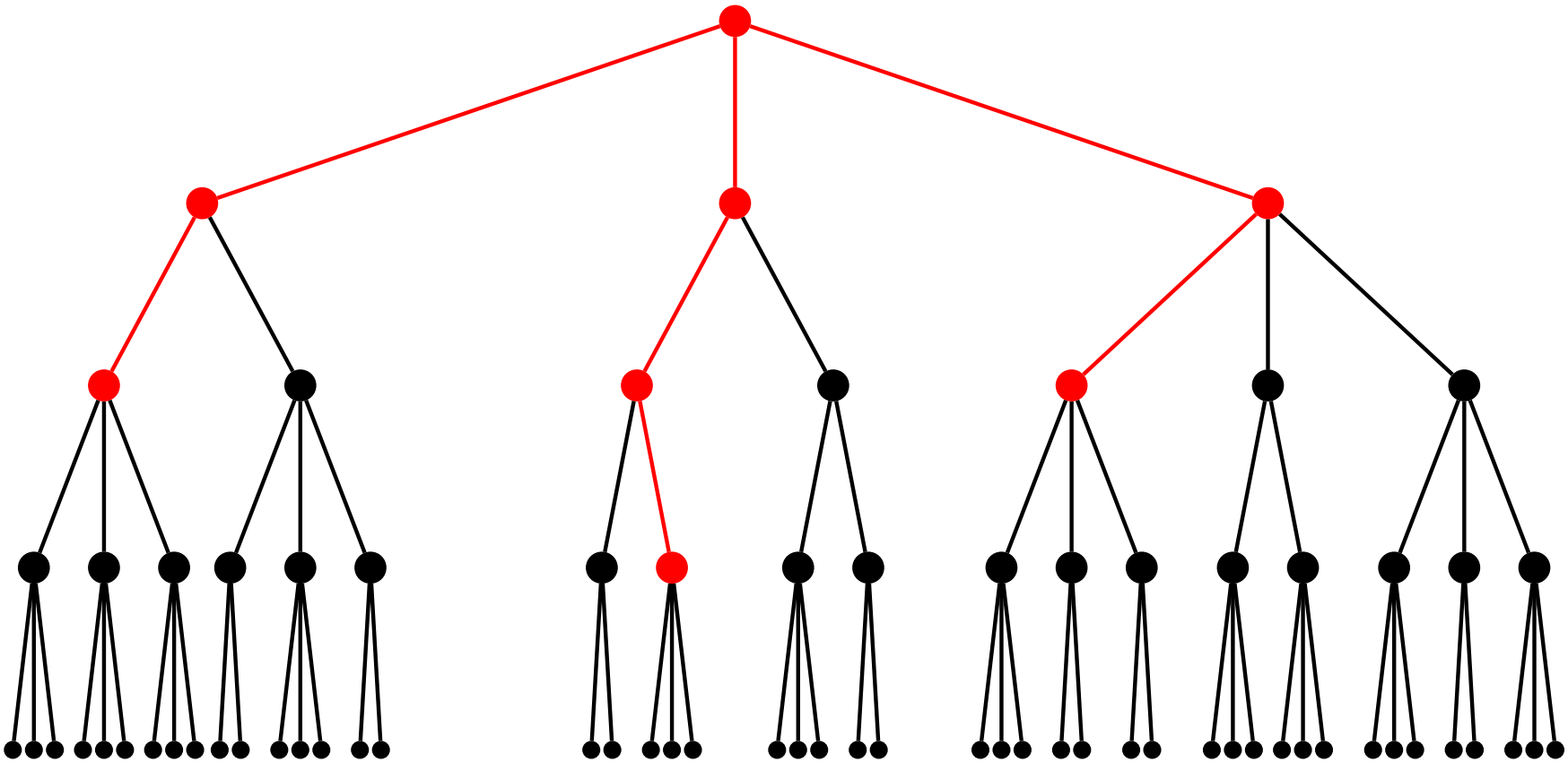


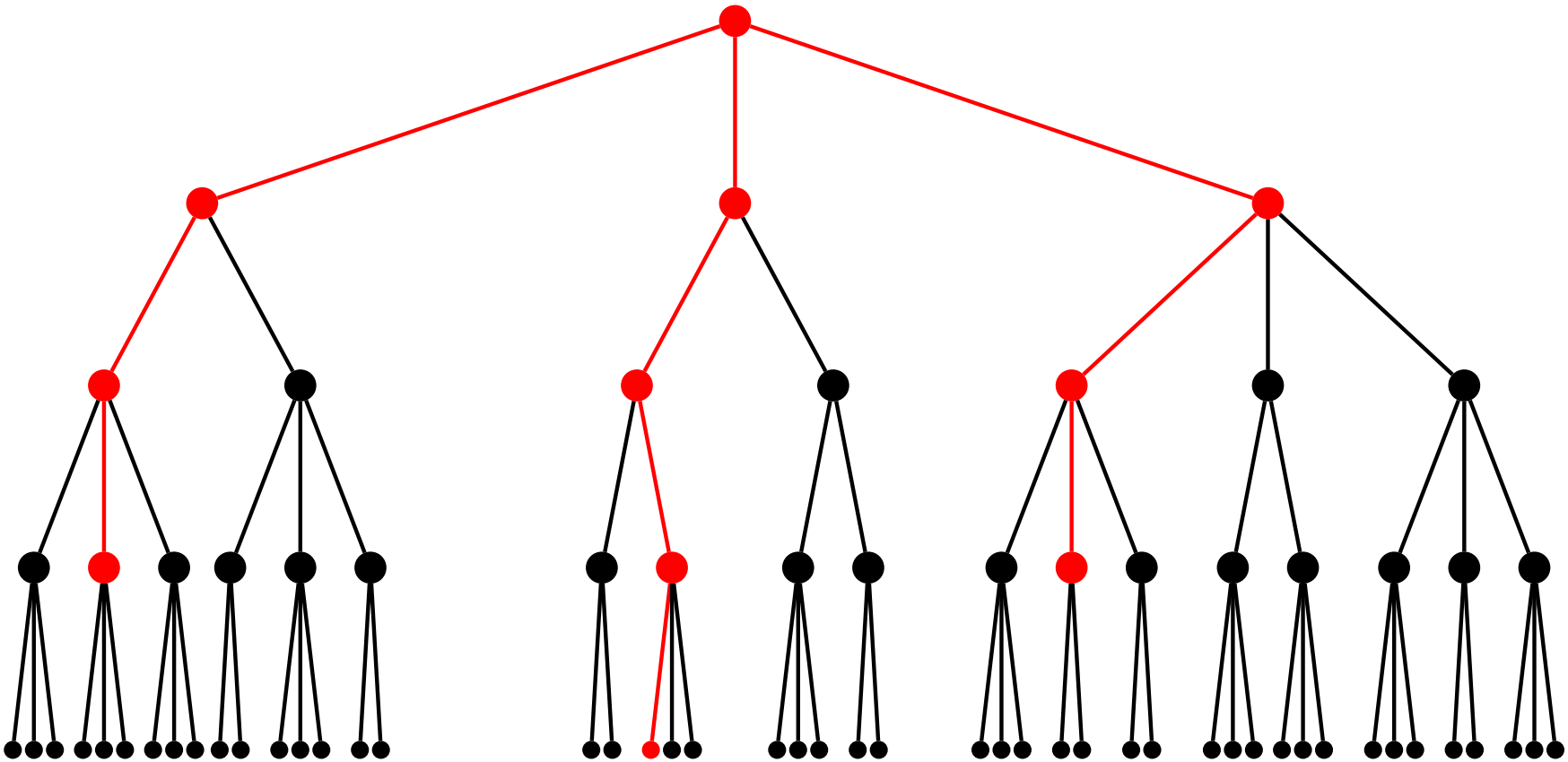
BUILD


REMOVE

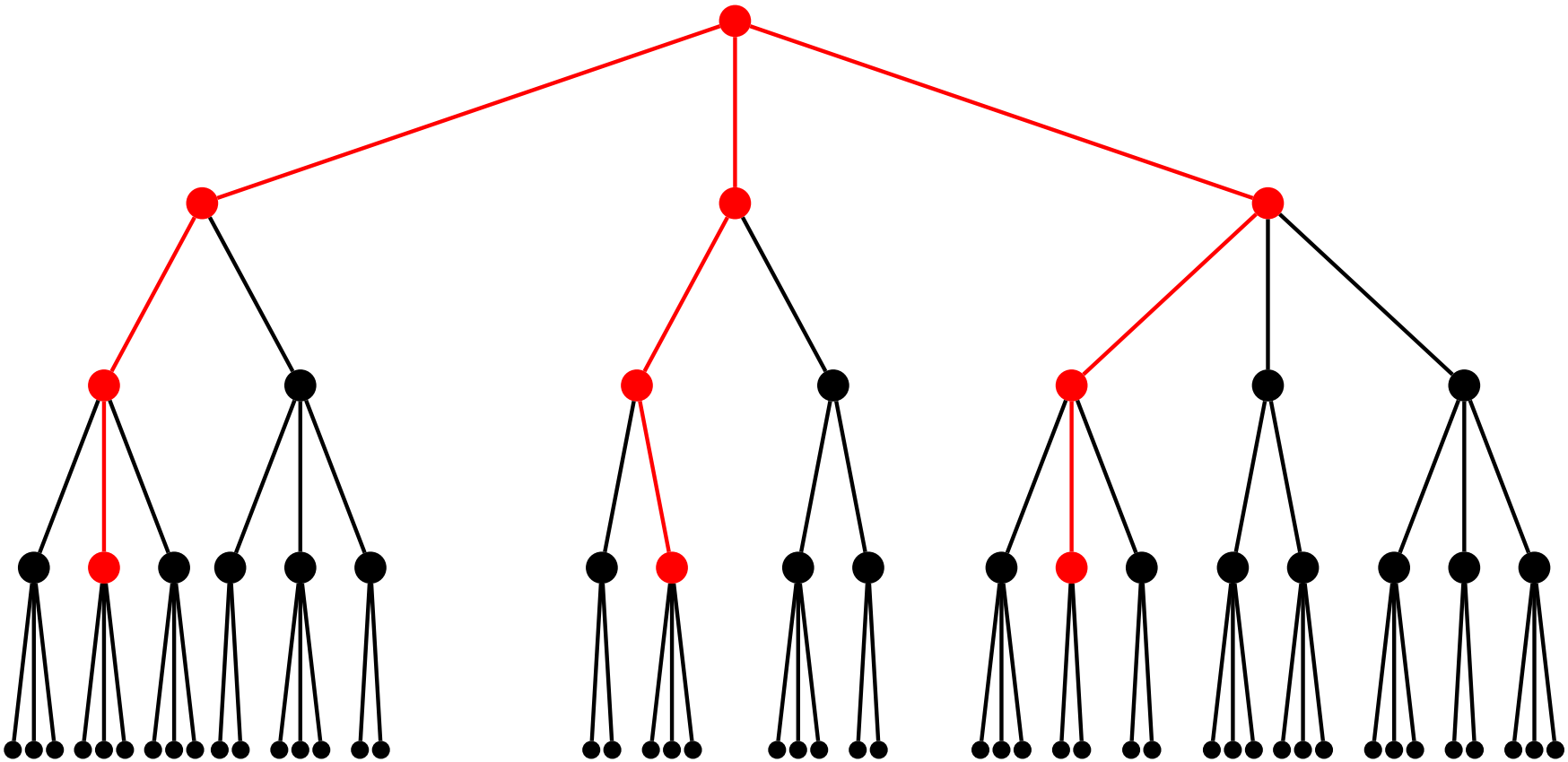


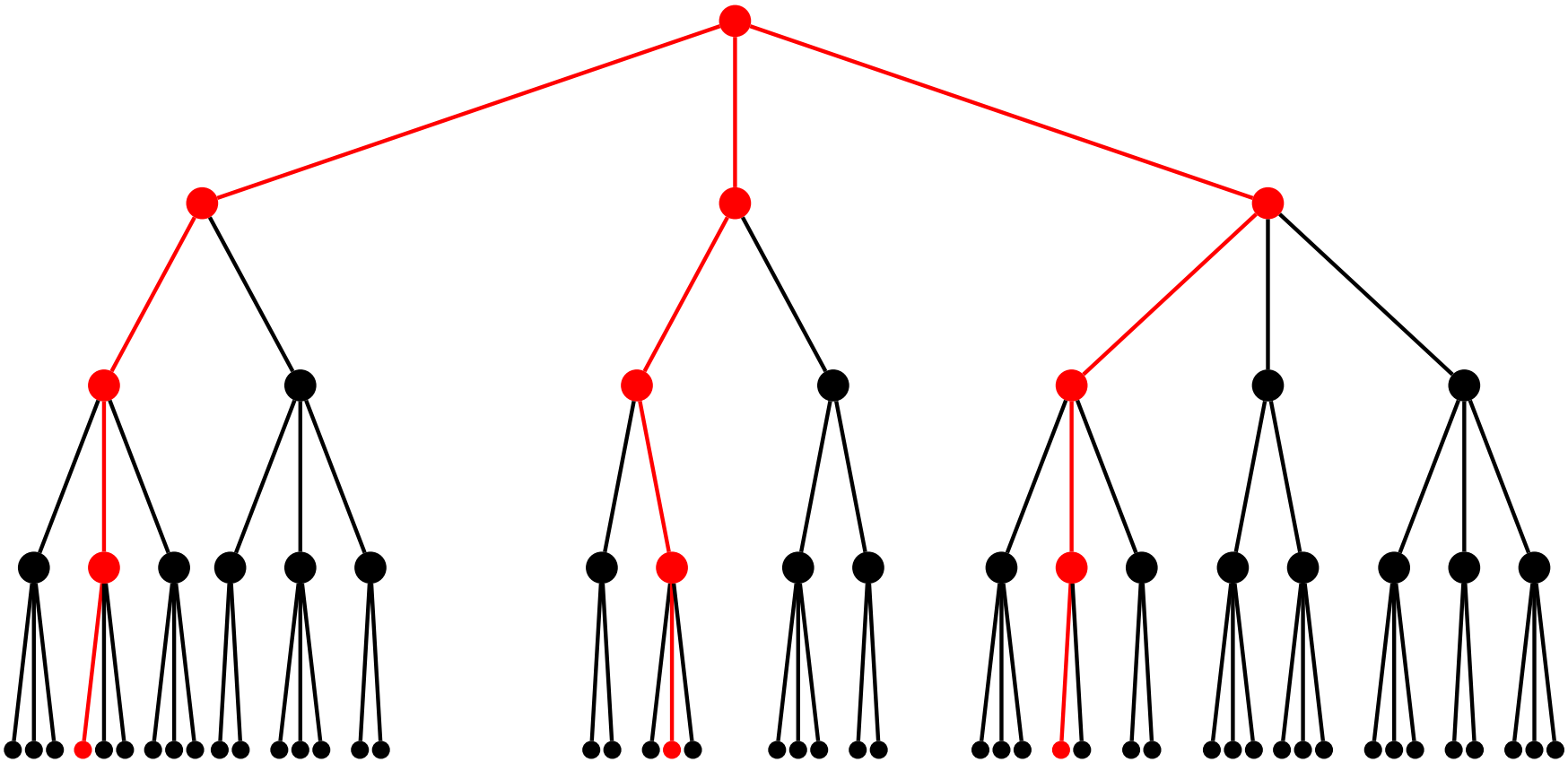
BUILD


REMOVE

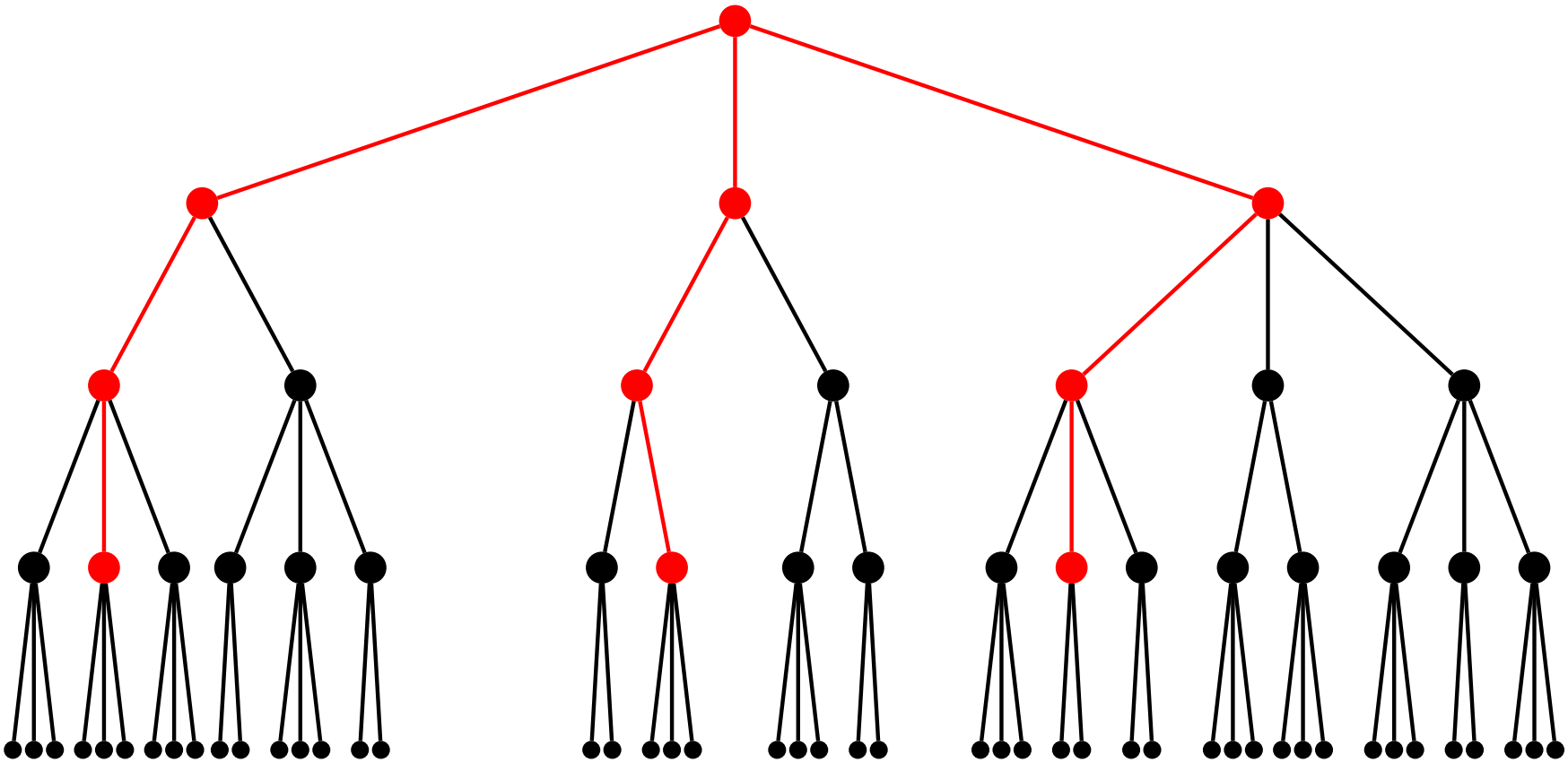


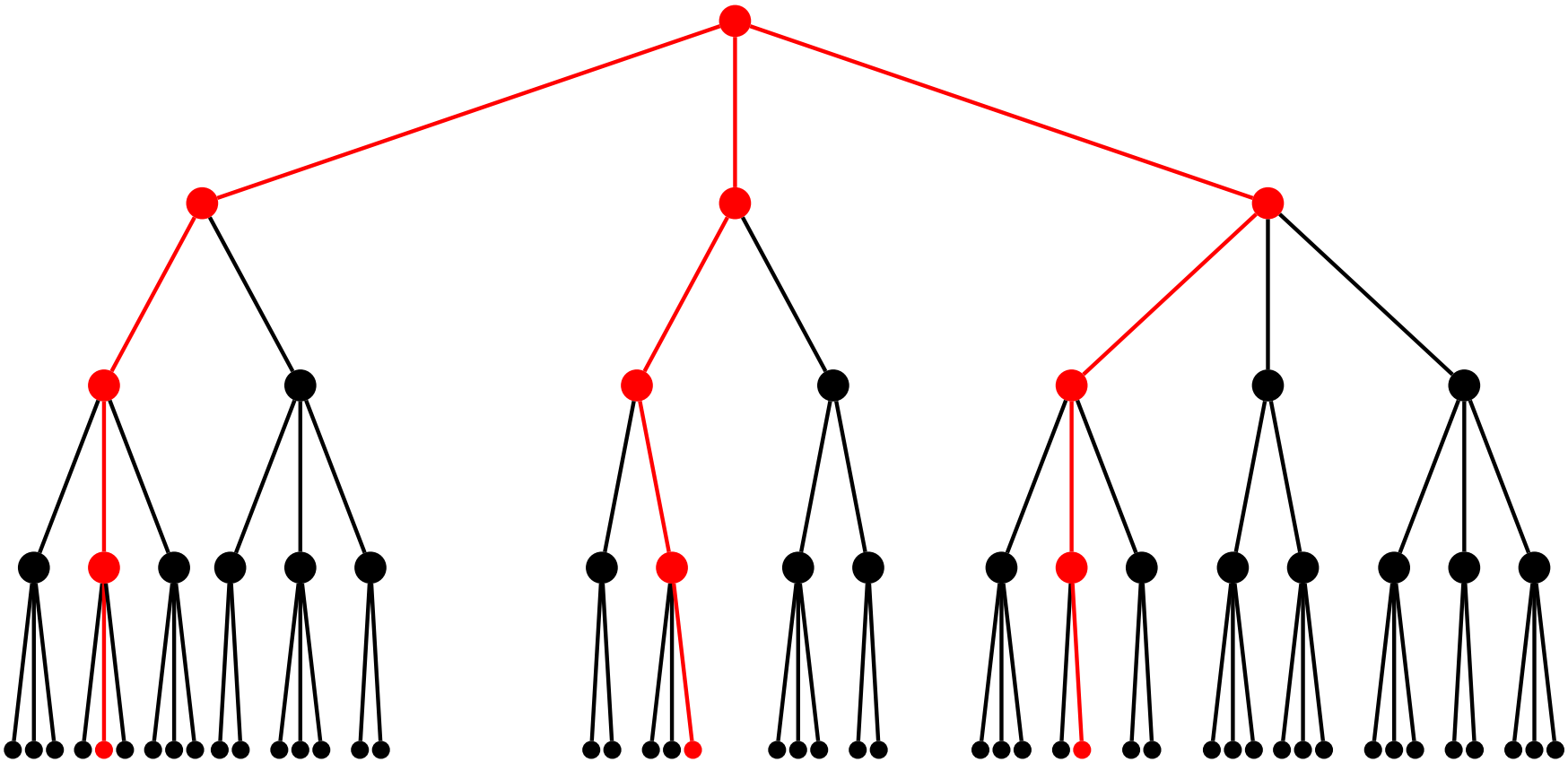
BUILD


REMOVE

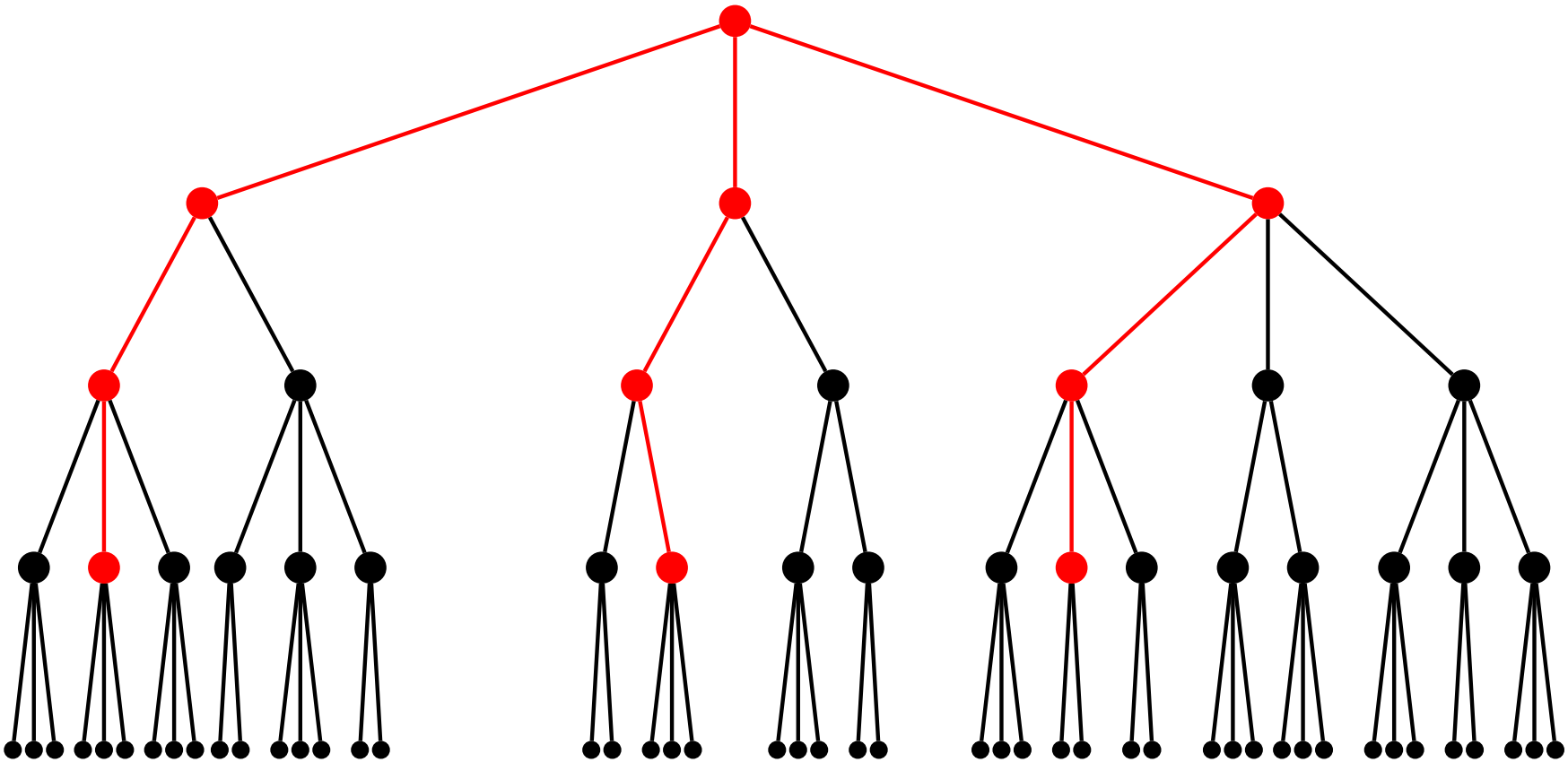


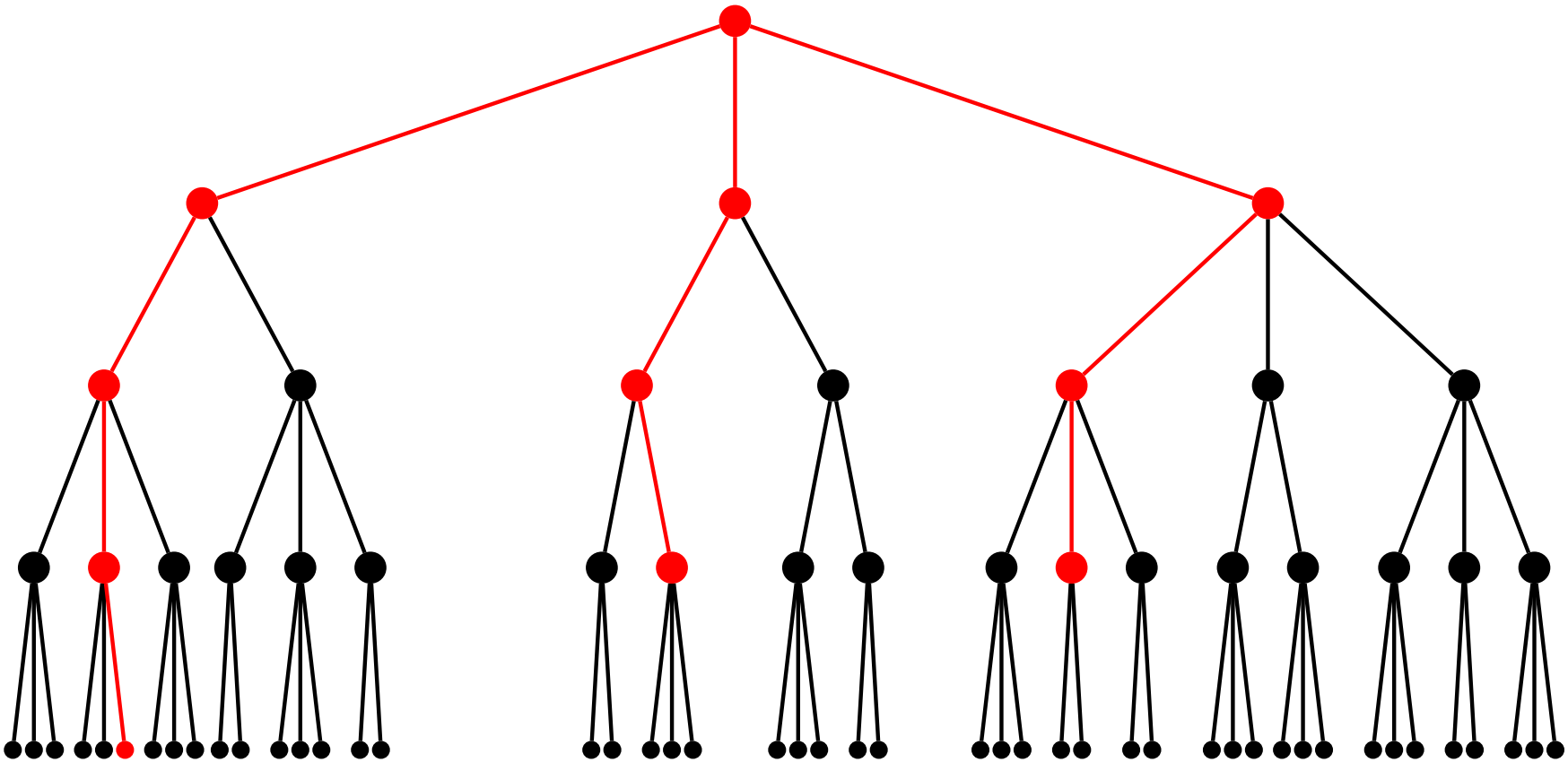
BUILD


REMOVE

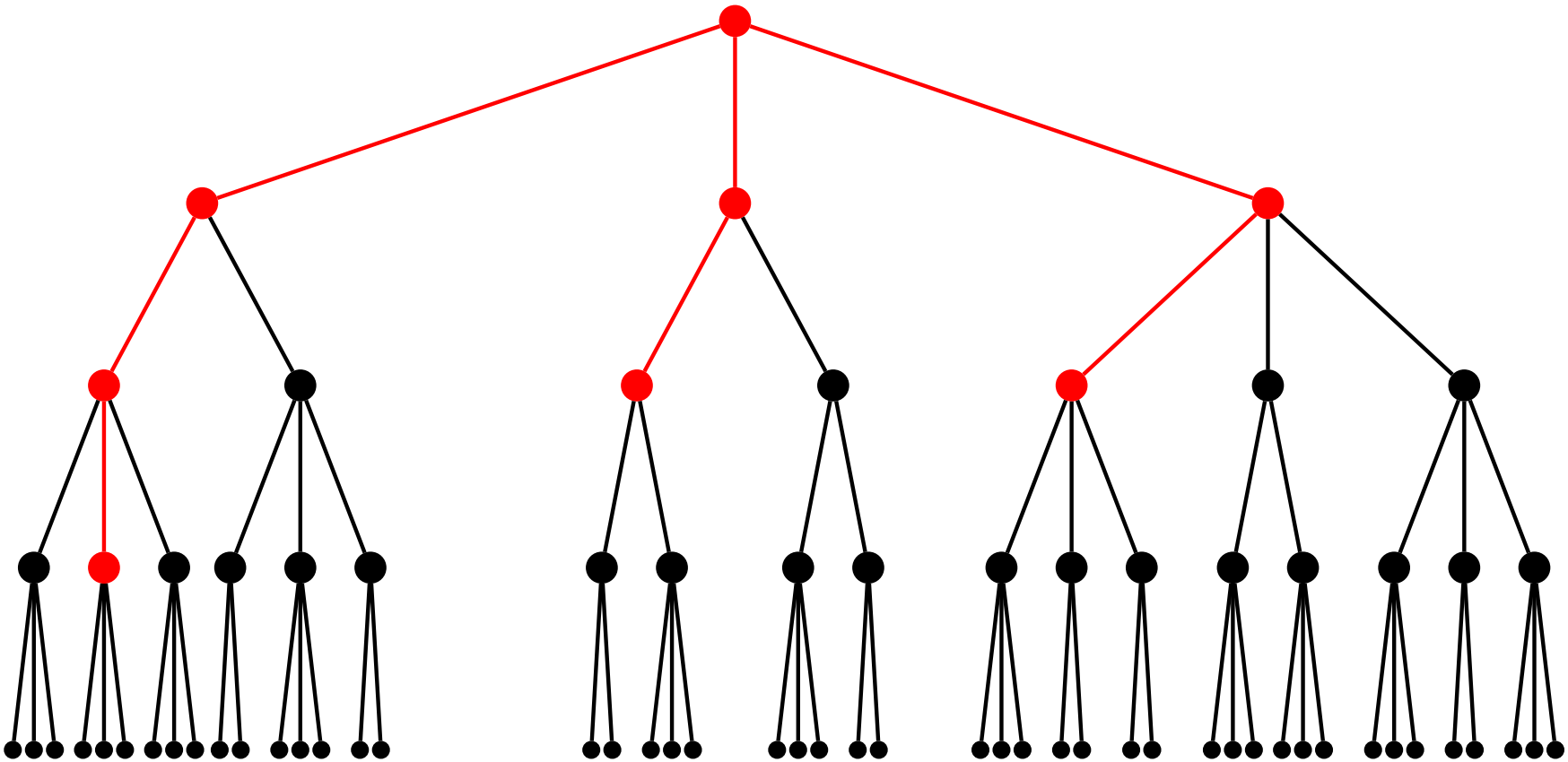


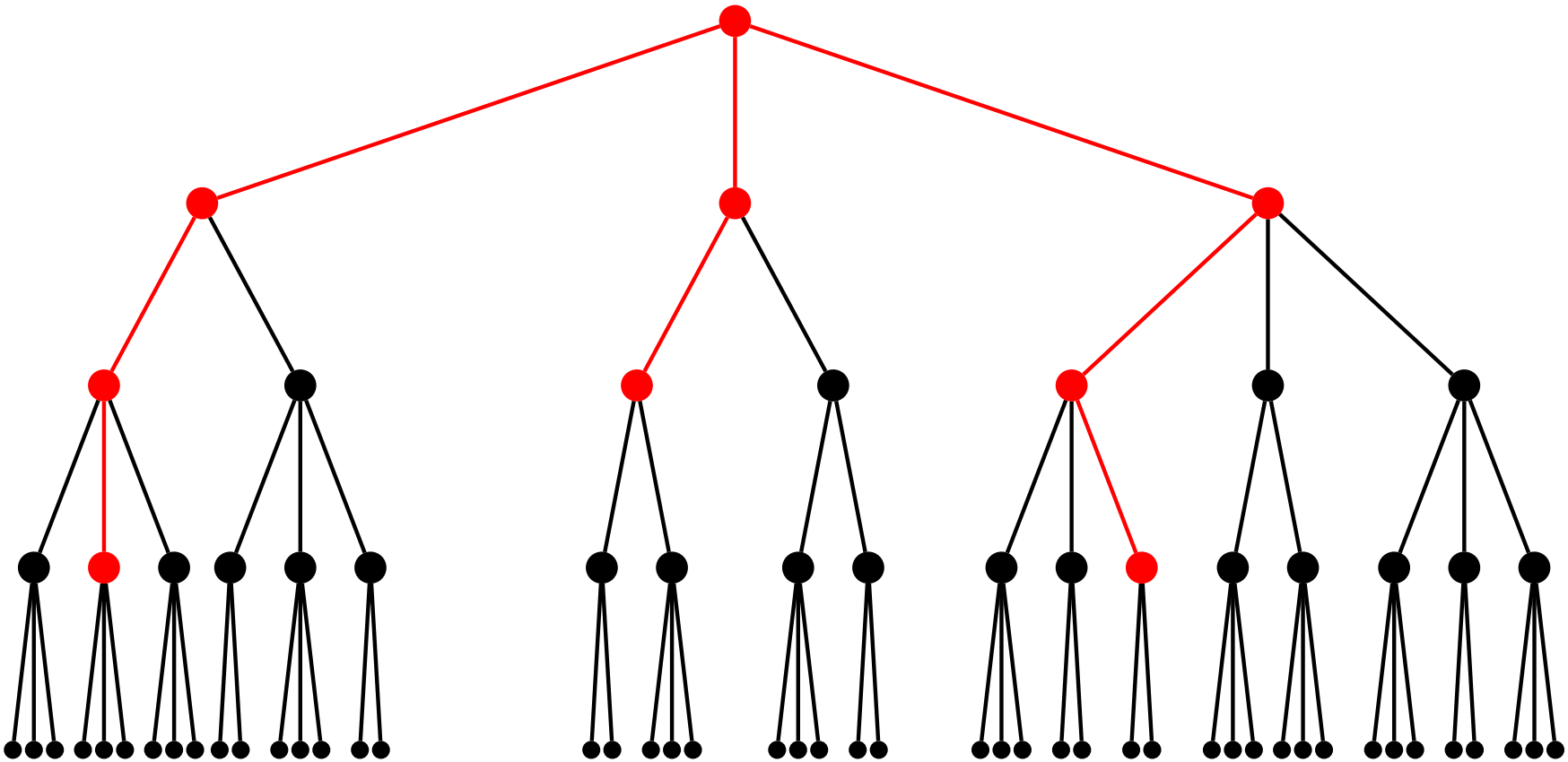
BUILD

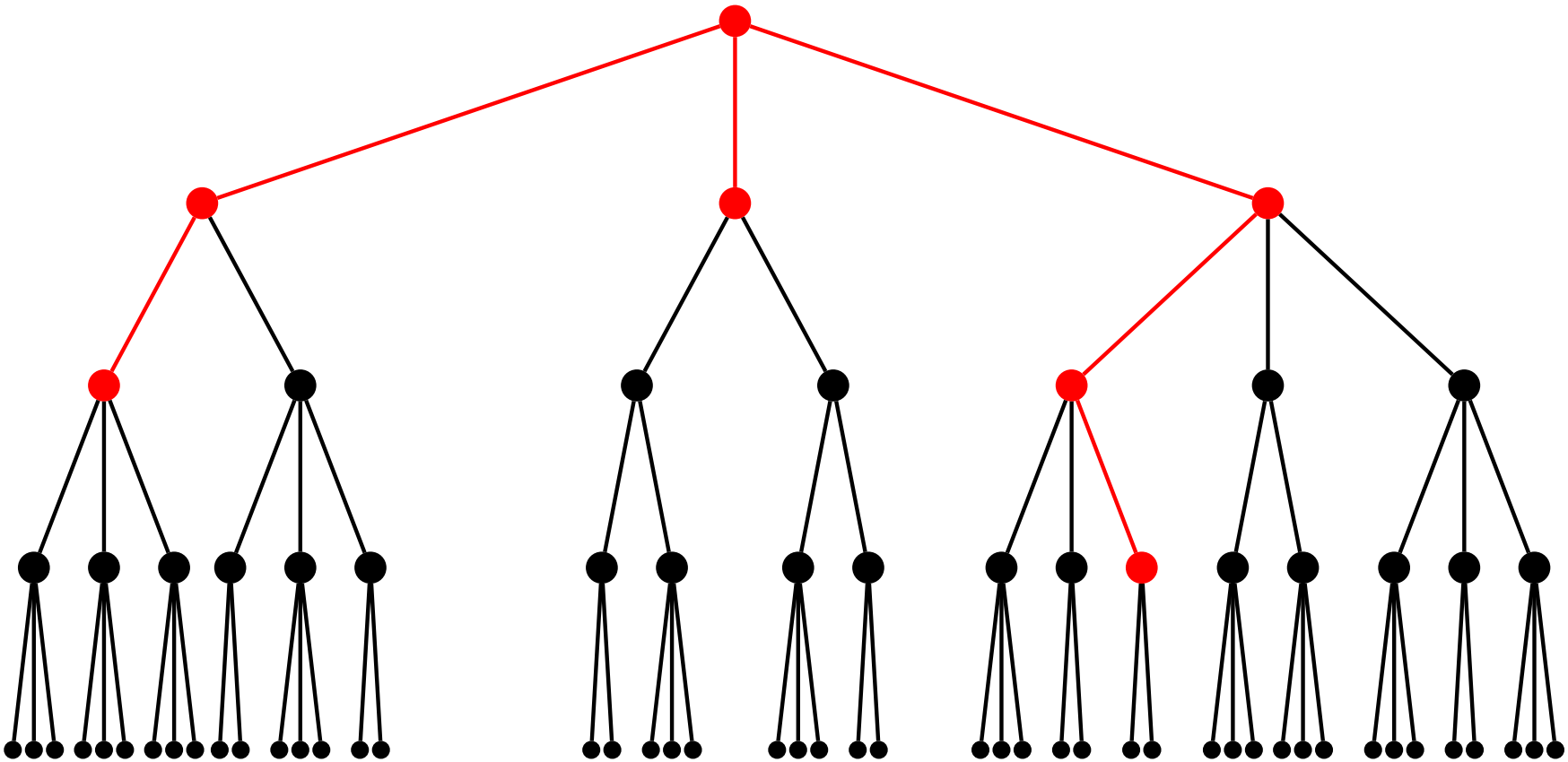

REMOVE

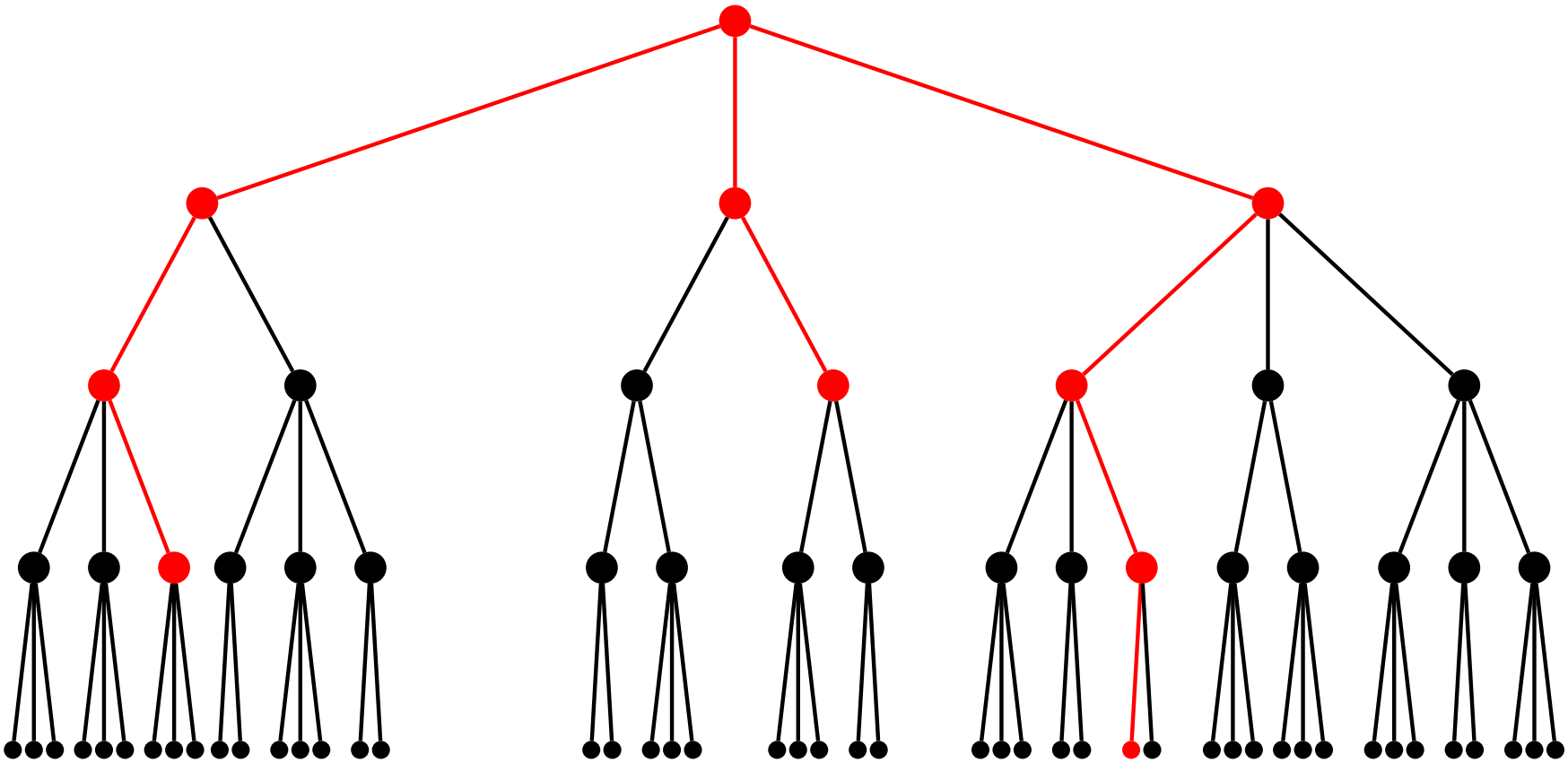


BUILD

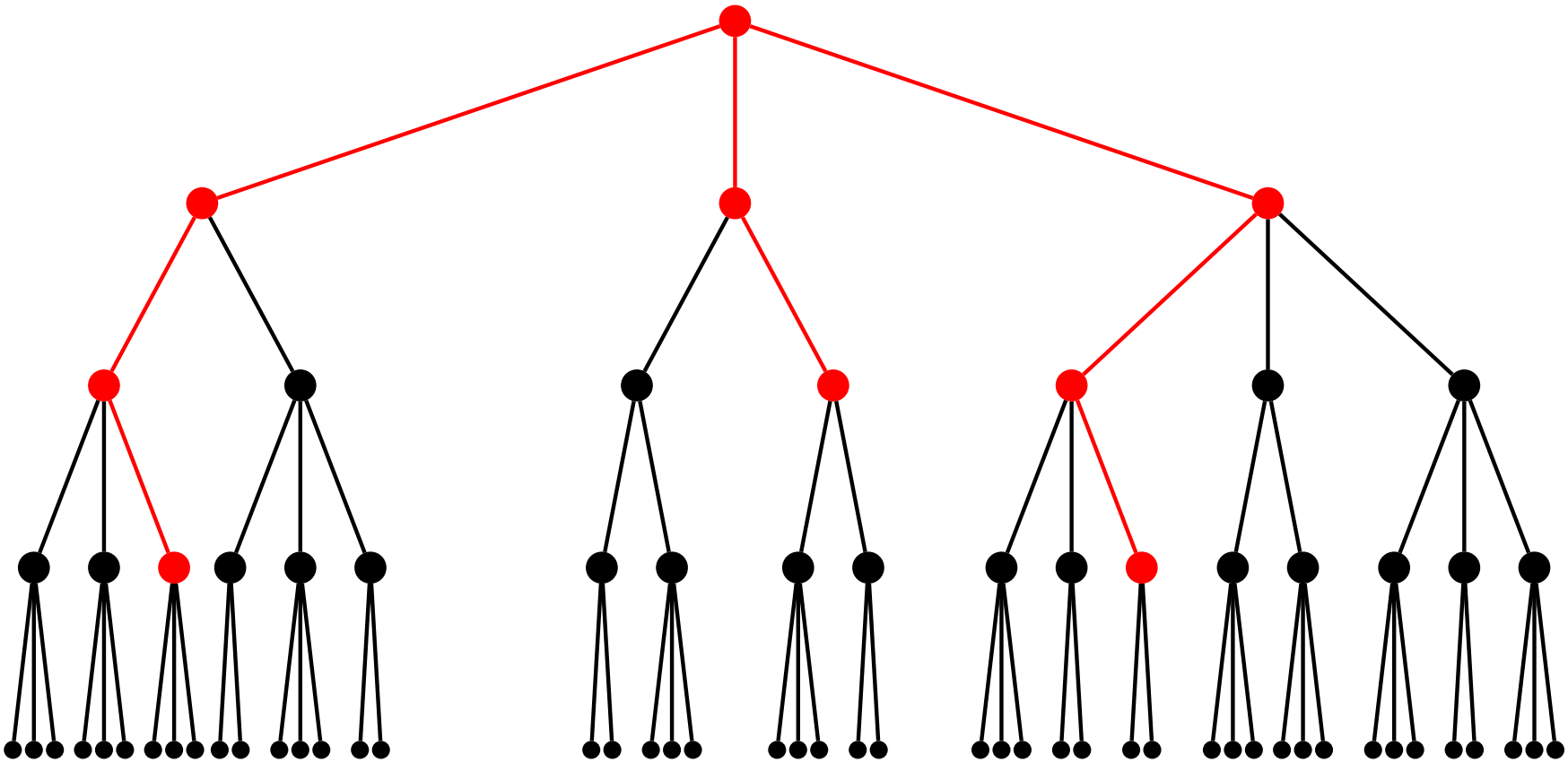

REMOVE

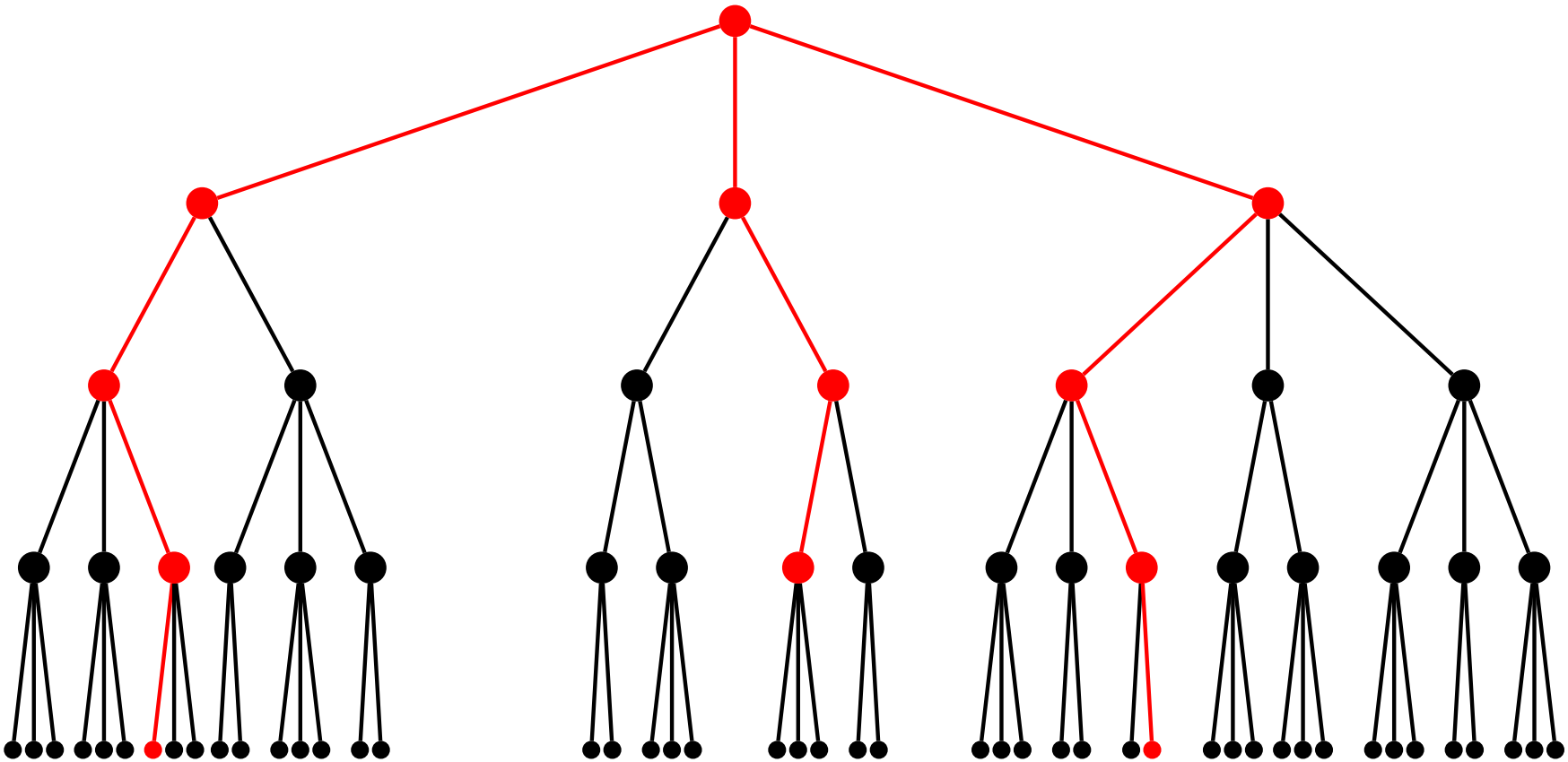

BUILD


REMOVE

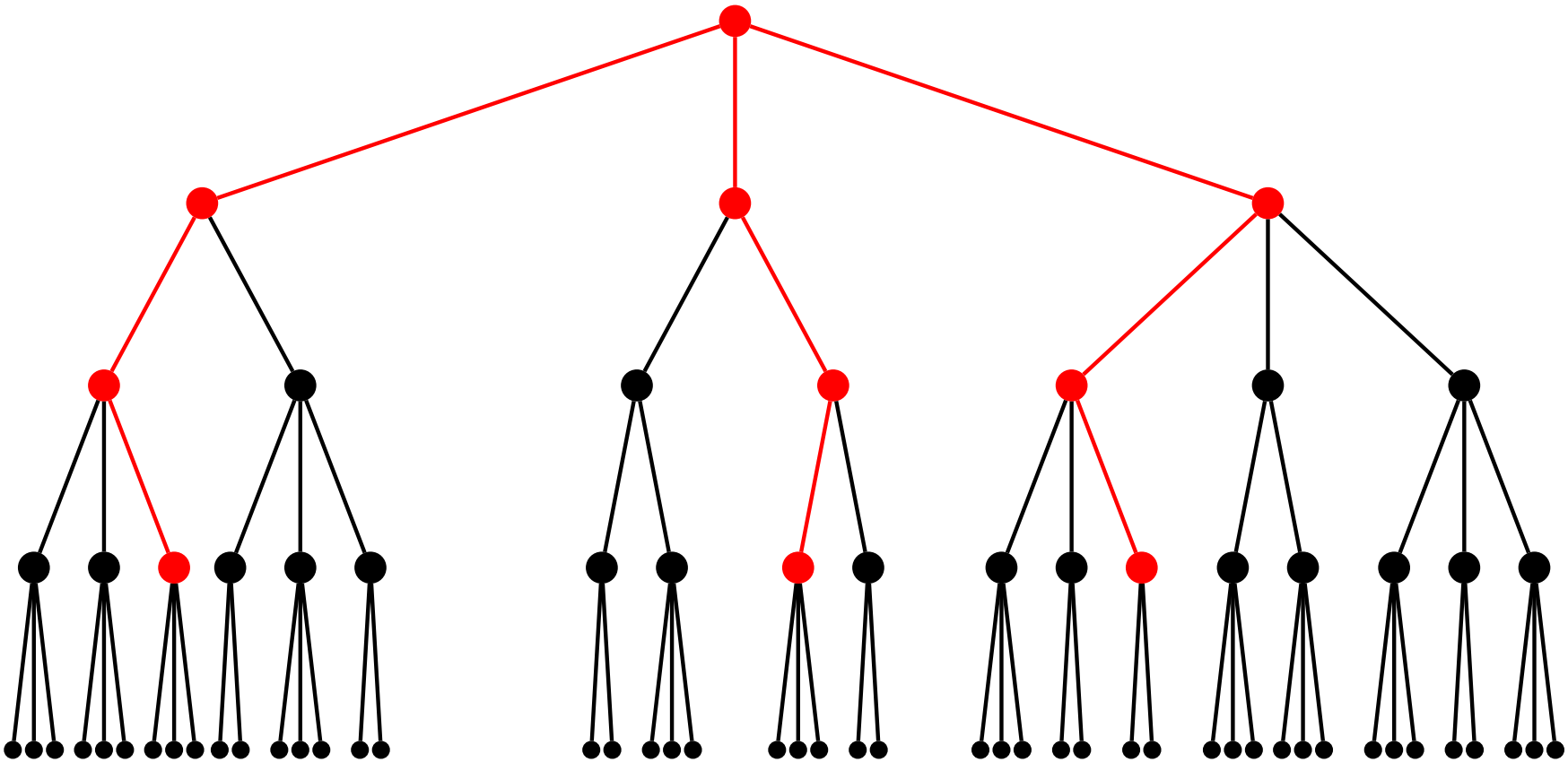


BUILD


REMOVE



BUILD


REMOVE

Backtracking - conclusion

- Thinking this way helped me develop a 25% faster C implementation.
- 8 GPU cores = 1 CPU core
- $\Rightarrow$ 240 GPU cores = 30 CPU cores (worth doing).

Backtracking - conclusion

- Thinking this way helped me develop a 25% faster C implementation.
- 8 GPU cores = 1 CPU core
- $\Rightarrow$ 240 GPU cores = 30 CPU cores (worth doing).
- Not the end of the story: additional algorithmic improvements over brute force backtracking possible.
- GPU can't compete without these improvements (better algorithms trump hardware).
- More work, and ingenuity, needed.

Where to next?

- Start programming, even if you can't think of a problem (yet).
- Different from CPU (parallelism, SIMT), will take time to become expert in thinking this way (I'm not).
- Will also give you a different way to consider problems.
- Help form a GPU community? Largest barrier to taking advantage of hardware and software improvements is lack of technical skill.
- See Melbourne University GPU Group http://groups.google.com.au/group/mugpug
- Dave Rawlinson speaking on June 19.