I/O Efficient Sorting in the Cache-Oblivious Model: Quicker than Quick

Dana Jansens
Carleton University

Overview

- Previous Work
- Introduce Cache-Oblivious Model
- Sorting Algorithm: *Funnelsort*
- Funnelsort in multi-core environment

Previous Work

- Cache-Oblivious Algorithms
  - Introduced in 1999 – Frigo, Leiserson, Prokop, Ramachandran
  - **Cache Oblivious Model**
  - Funnelsort algorithm
- Improvements on Funnelsort
  - Published in 2004 – Brodal, Fagerberg, Vinther
  - *Engineering a cache-oblivious sorting algorithm*
  - Empirical results with the algorithm

Previous Work

- Cache-Oblivious Algorithms
  - Introduced in 1999 – Frigo, Leiserson, Prokop, Ramachandran
  - Cache Oblivious Model
  - **Funnelsort algorithm**
- Improvements on Funnelsort
  - Published in 2004 – Brodal, Fagerberg, Vinther
  - *Engineering a cache-oblivious sorting algorithm*
  - Empirical results with the algorithm

Previous Work

- Cache-Oblivious Algorithms
  - Introduced in 1999 – Frigo, Leiserson, Prokop, Ramachandran
  - Cache Oblivious Model
  - Funnelsort algorithm
- Improvements on Funnelsort
  - Published in 2004 – Brodal, Fagerberg, Vinther
  - *Engineering a cache-oblivious sorting algorithm*
  - **Empirical results with the algorithm**

Cache Oblivious Model

- **Tool for analyzing algorithms**
  - should impart a *realistic* view of performance
- Count I/Os
  - performed between CPU and main memory
- Why is this I/O important ?
- Where did this approach come from ?

Cache Oblivious Model

- Tool for analyzing algorithms
  - should impart a *realistic* view of performance
- **Count I/Os**
  - performed between CPU and main memory
- Why is this I/O important ?
- Where did this approach come from ?

Cache Oblivious Model

- Tool for analyzing algorithms
  - should impart a *realistic* view of performance
- Count I/Os
  - performed between CPU and main memory
- **Why is this I/O important ?**
- Where did this approach come from ?

Cache Oblivious Model

- Tool for analyzing algorithms
  - should impart a *realistic* view of performance
- Count I/Os
  - performed between CPU and main memory
- Why is this I/O important ?
- **Where did this approach come from ?**

RAM Model (Random Access)

- **Standard model for measuring algorithms**
- Count CPU operations
- Generally a good measure of performance
- Makes a big assumption !

RAM Model (Random Access)

- Standard model for measuring algorithms
- **Count CPU operations**
- Generally a good measure of performance
- Makes a big assumption !

RAM Model (Random Access)

- Standard model for measuring algorithms
- Count CPU operations
- **Generally a good measure of performance**
- Makes a big assumption !

RAM Model (Random Access)

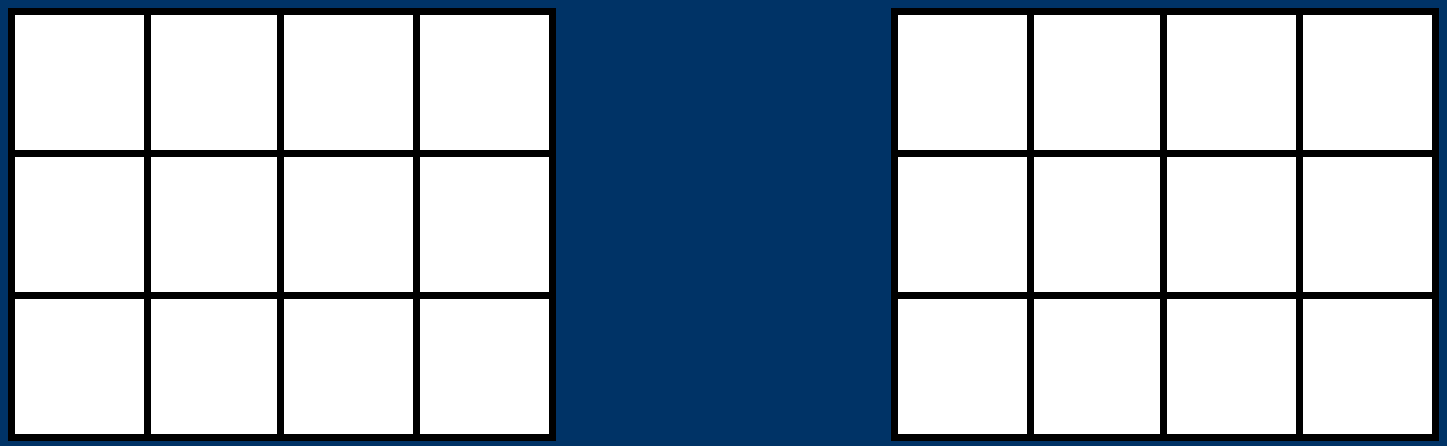
- Standard model for measuring algorithms
- Count CPU operations
- Generally a good measure of performance
- **Makes a big assumption !**

RAM Model (Random Access)

- RAM Model assumption:

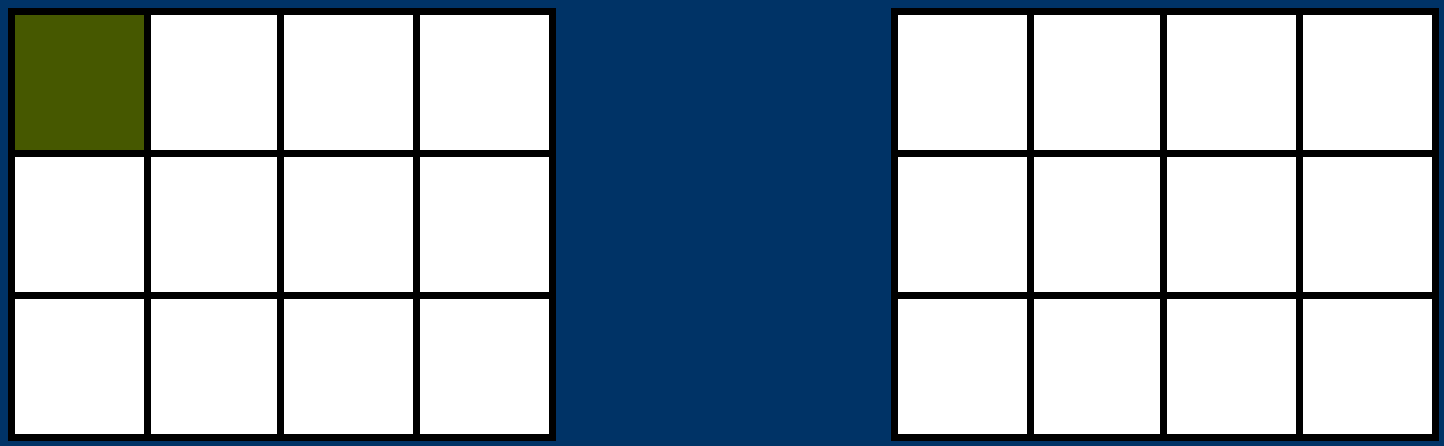
RAM Model (Random Access)

- RAM Model assumption:
  - Identical time to access to any memory location



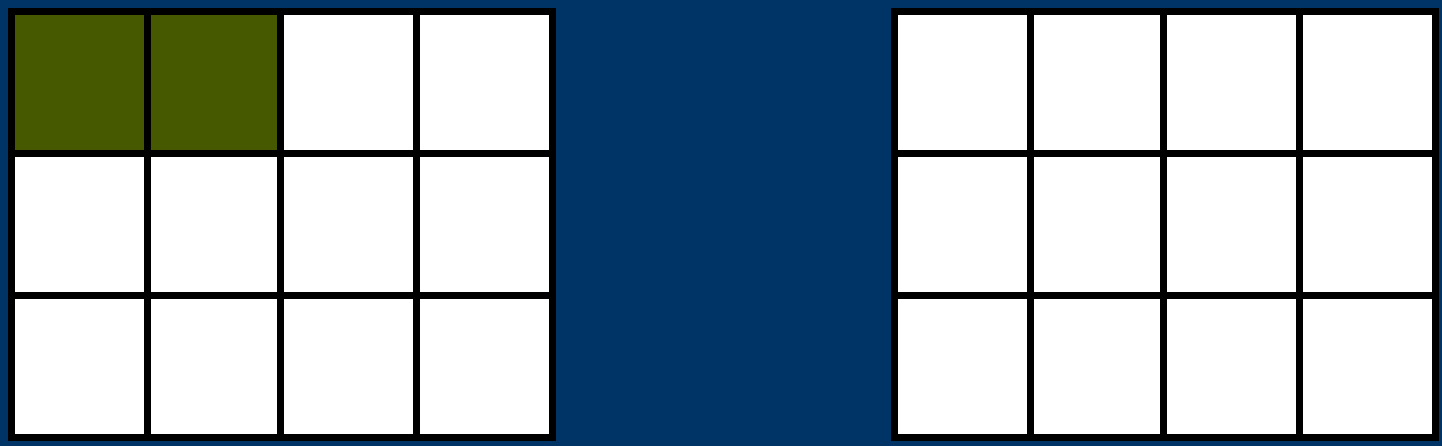
RAM Model (Random Access)

- RAM Model assumption:
  - Identical time to access to any memory location



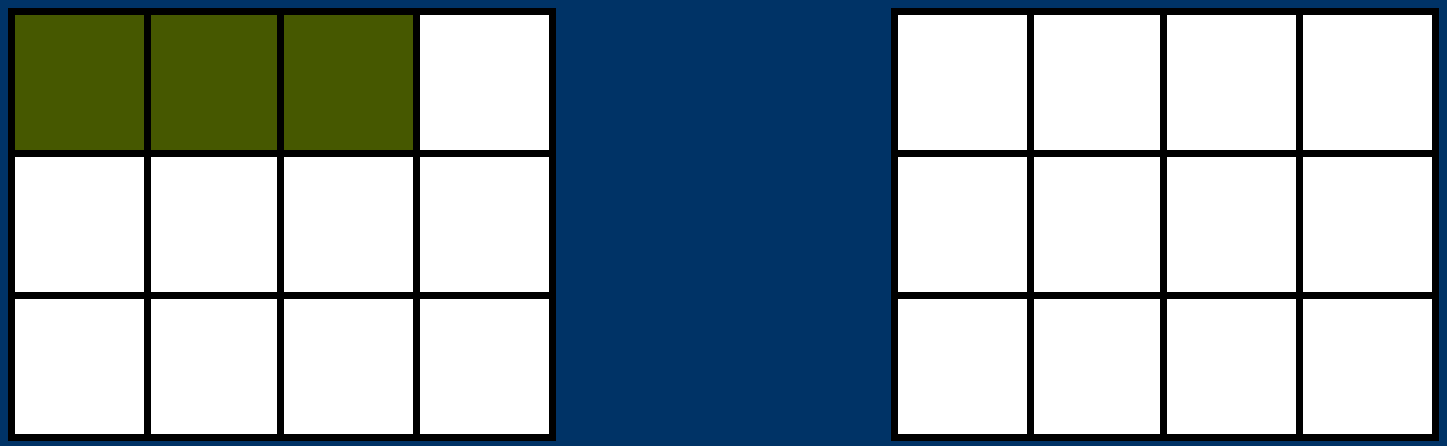
RAM Model (Random Access)

- RAM Model assumption:
  - Identical time to access to any memory location



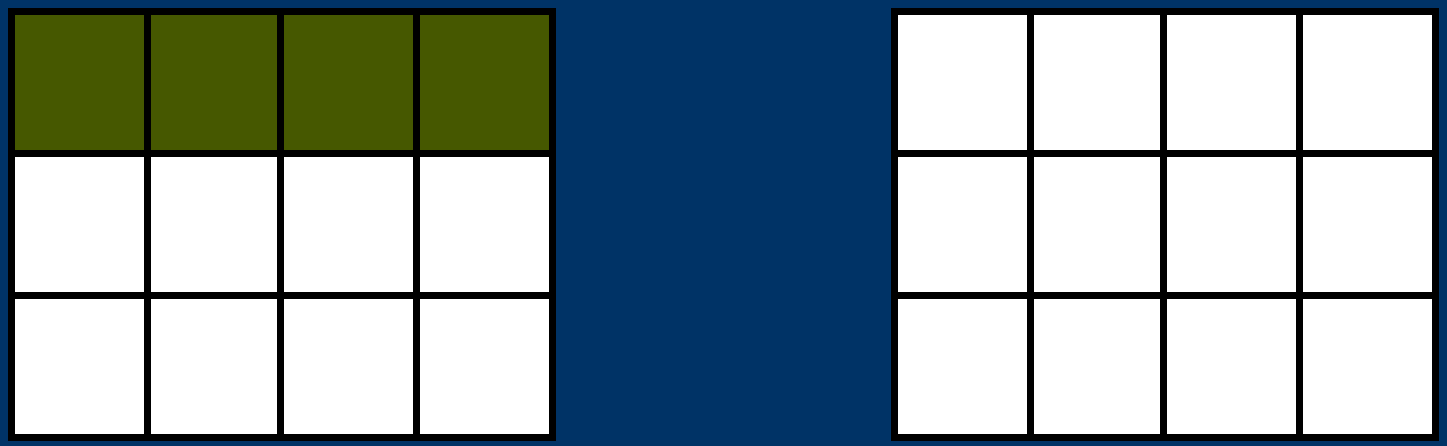
RAM Model (Random Access)

- RAM Model assumption:
  - Identical time to access to any memory location



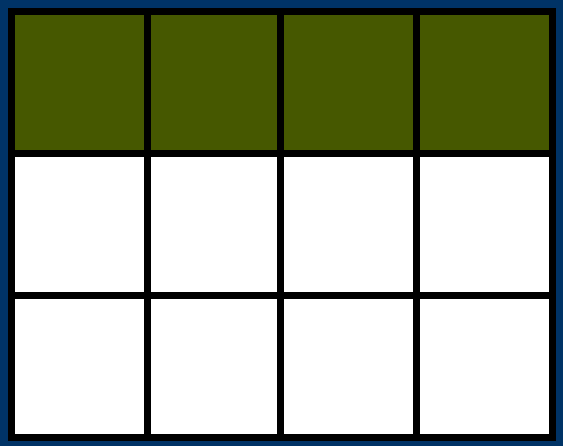
RAM Model (Random Access)

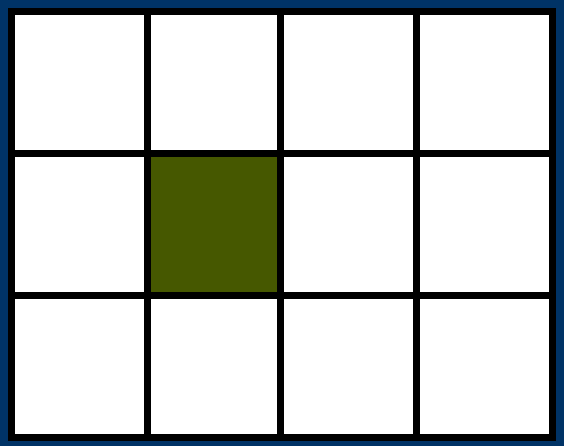
- RAM Model assumption:
  - Identical time to access to any memory location



RAM Model (Random Access)

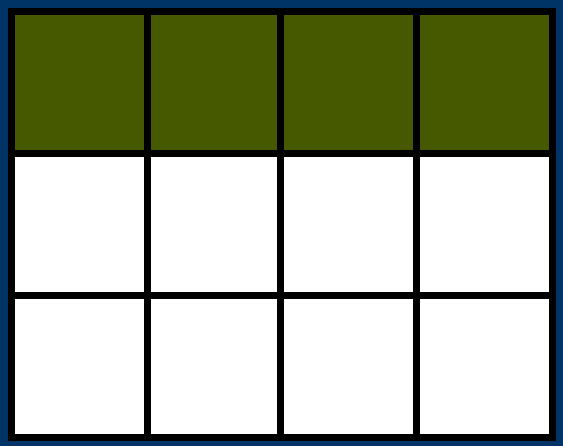
- RAM Model assumption:
  - Identical time to access to any memory location

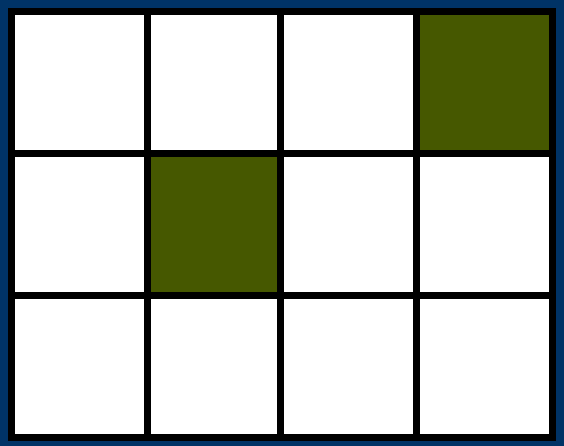




RAM Model (Random Access)

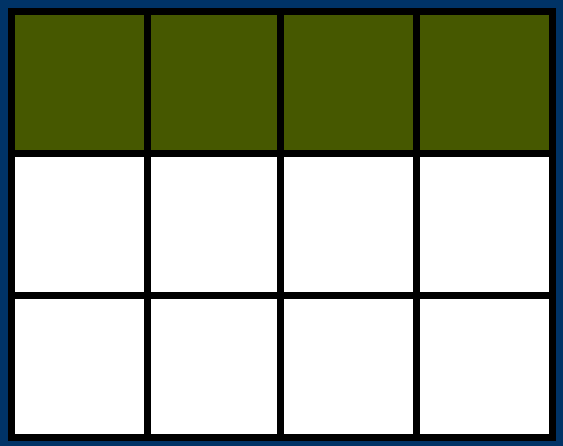
- RAM Model assumption:
  - Identical time to access to any memory location

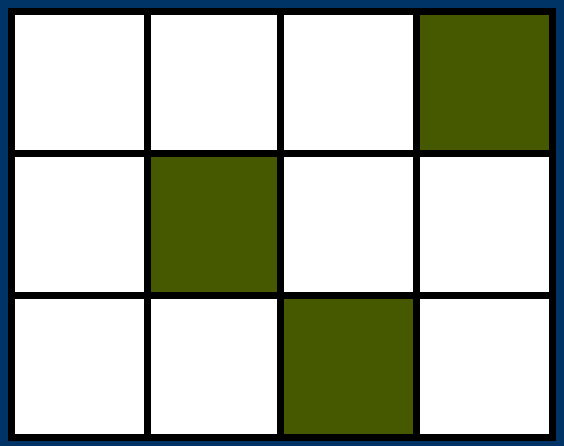




RAM Model (Random Access)

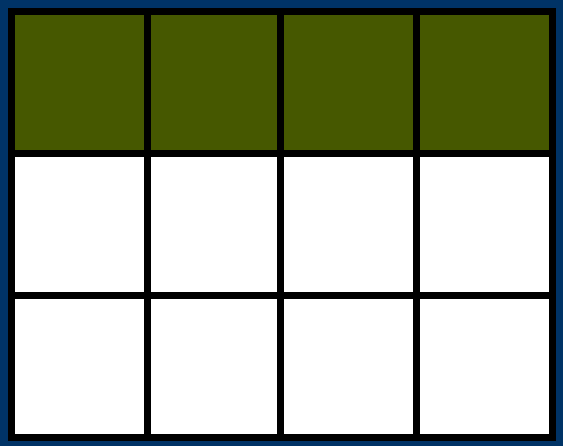
- RAM Model assumption:
  - Identical time to access to any memory location

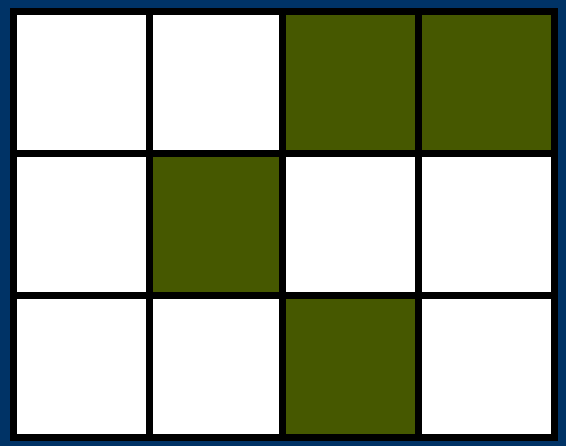




RAM Model (Random Access)

- RAM Model assumption:
  - Identical time to access to any memory location





RAM Model (Random Access)

- RAM Model assumption:
  - Identical time to access to any memory location
  - Memory access patterns don't affect performance

RAM Model (Random Access)

- However..

  Memory access patterns *do* affect performance.

RAM Model (Random Access)

- However..

  Memory access patterns *do* affect performance.

- Want to measure the efficiency of an algorithm's memory use.

EM Model (External Memory)

- **I/O very important when it is slow**
- External disks are much slower than CPU
  - Hard drives
- Want to make as few trips to disk as possible

EM Model (External Memory)

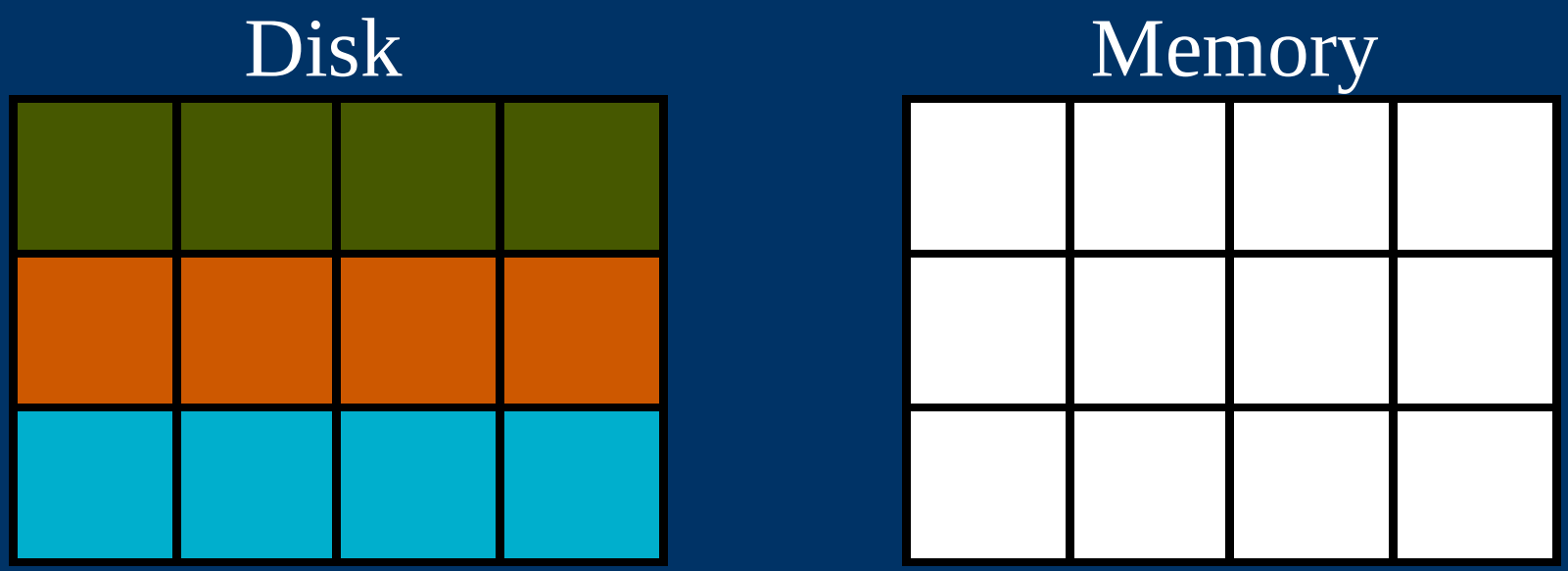
- I/O very important when it is slow
- **External disks are much slower than CPU**
  - Hard drives
- Want to make as few trips to disk as possible

EM Model (External Memory)

- I/O very important when it is slow
- External disks are much slower than CPU
  - Hard drives
- **Want to make as few trips to disk as possible**

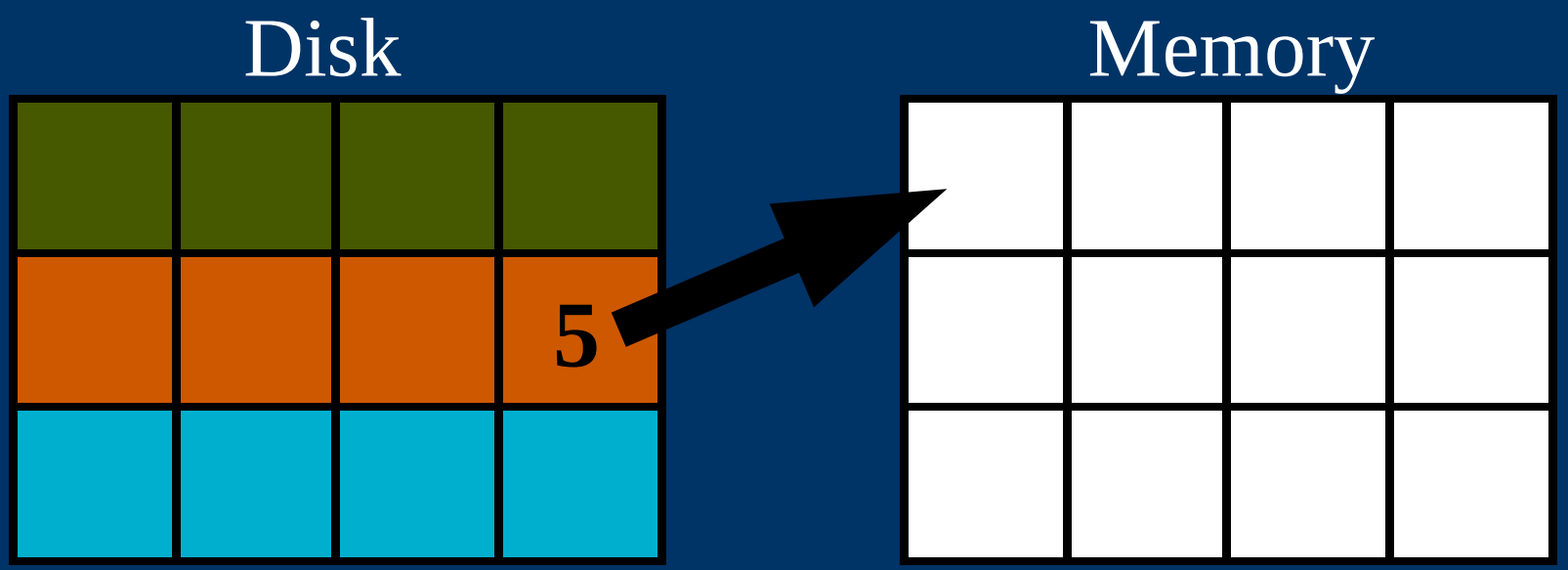
EM Model (External Memory)

- Disk transfers done in blocks
- B = block size



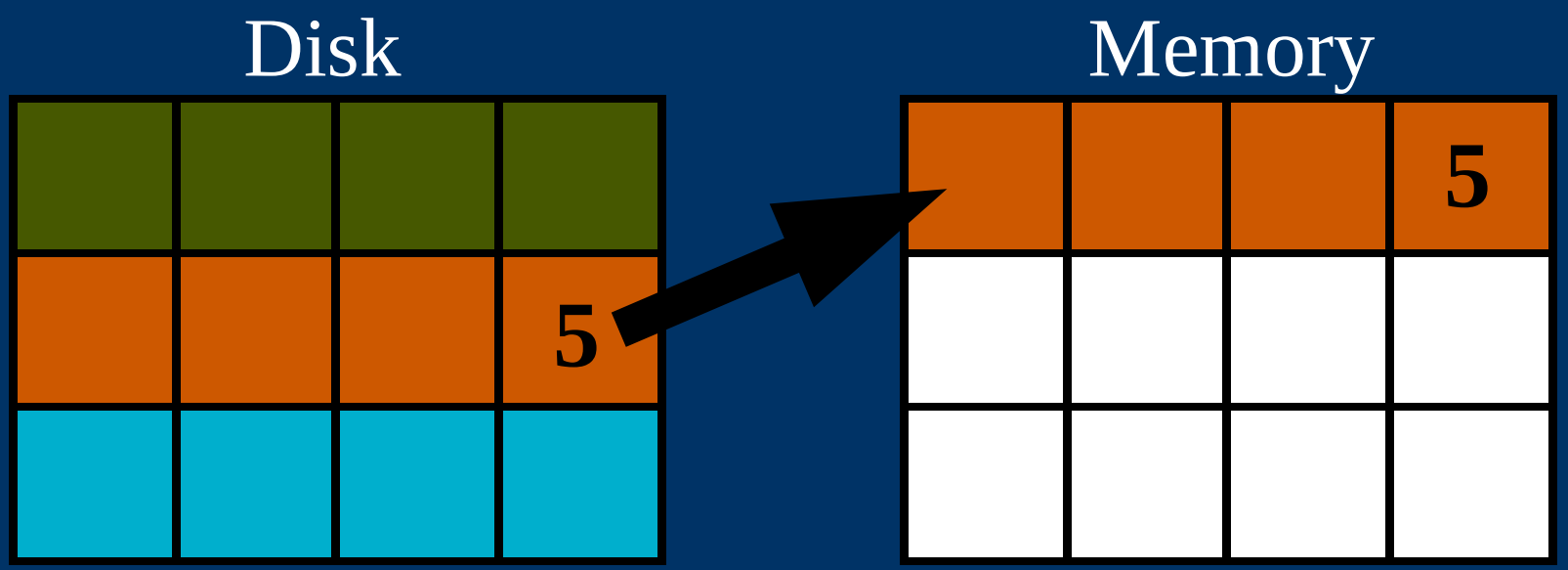
EM Model (External Memory)

- Disk transfers done in blocks
- B = block size



EM Model (External Memory)

- Disk transfers done in blocks
- B = block size



EM Model (External Memory)

- **I/O Efficiency**

  Analyze how many **blocks** are transferred between disk and memory.

- Big-O notation
  - Counts the number of I/O block transfers

EM Model (External Memory)

- I/O Efficiency

  Analyze how many **blocks** are transferred between disk and memory.

- **Big-O notation**
  - Counts the number of I/O block transfers

Cache-Oblivious Model

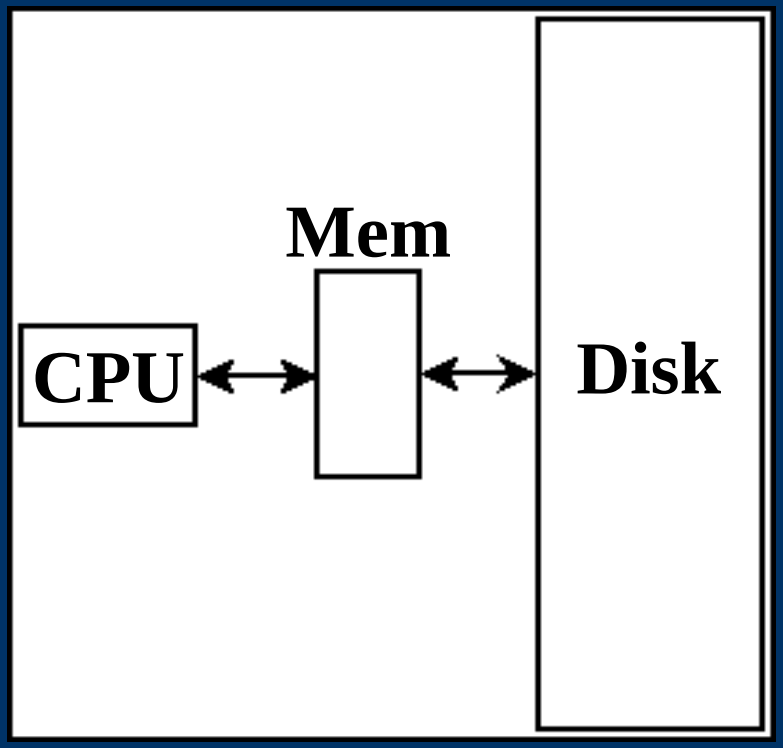
- **Adds ideas from the EM model to the RAM model**
- Measures I/O block transfers
  - between *main memory* and the processor *cache*

Cache-Oblivious Model

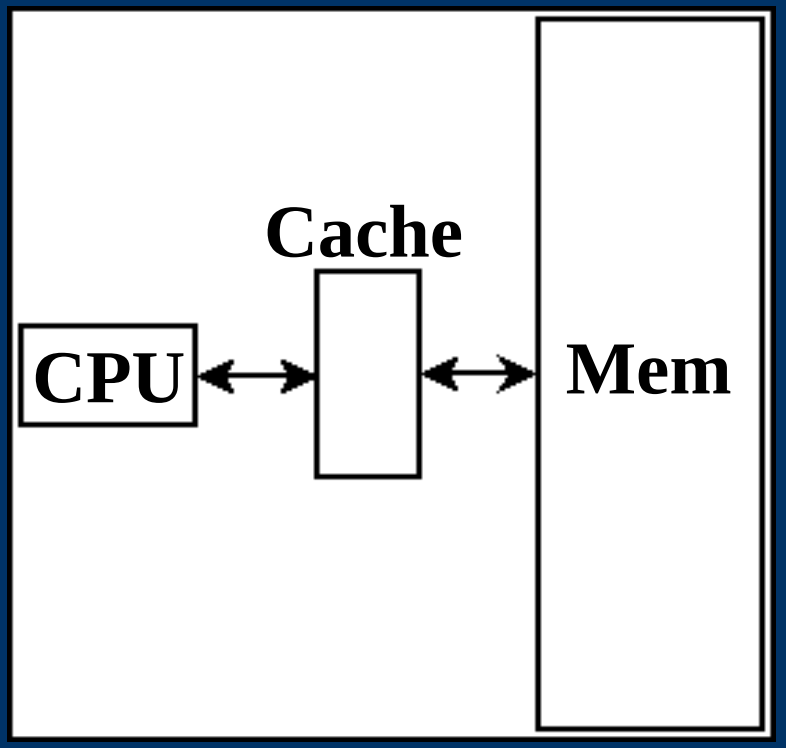
- Adds ideas from the EM model to the RAM model
- **Measures I/O block transfers**
  - between *main memory* and the processor *cache*

Cache-Oblivious Model

EM Model



Cache-Oblivious Model



Cache-Oblivious Model

- **But memory much faster than disk**
- Modern CPUs are getting faster relative to memory
- Memory feels “farther away” from the CPU
- Multiple cores compete for memory access

Cache-Oblivious Model

- But memory much faster than disk
- **Modern CPUs are getting faster relative to memory**
- Memory feels “farther away” from the CPU
- Multiple cores compete for memory access

Cache-Oblivious Model

- But memory much faster than disk
- Modern CPUs are getting faster relative to memory
- **Memory feels “farther away” from the CPU**
- Multiple cores compete for memory access

Cache-Oblivious Model

- But memory much faster than disk
- Modern CPUs are getting faster relative to memory
- Memory feels “farther away” from the CPU
- **Multiple cores compete for memory access**

Cache-Oblivious Model

- **Desire optimal work complexity**
  - Same as RAM model
- Desire optimal I/O complexity
  - Same as EM model
- Algorithm is unaware of the cache block size (B)
  - Should be optimal for *any* cache block size

Cache-Oblivious Model

- Desire optimal work complexity
  - Same as RAM model
- **Desire optimal I/O complexity**
  - Same as EM model
- Algorithm is unaware of the cache block size (B)
  - Should be optimal for *any* cache block size

Cache-Oblivious Model

- Desire optimal work complexity
  - Same as RAM model
- Desire optimal I/O complexity
  - Same as EM model
- **Algorithm is unaware of the cache block size (B)**
  - Should be optimal for *any* block size

Funnelsort

- **Sorting algorithm**
- Similar to merge sort
- Both work and I/O optimal
  - While being cache oblivious
  - $\Theta(n \log n)$ work
  - $\Theta((n \log n) / B)$ I/Os

Funnelsort

- Sorting algorithm
- **Similar to merge sort**
- Both work and I/O optimal
  - While being cache oblivious
  - $\Theta(n \log n)$ work
  - $\Theta((n \log n) / B)$ I/Os

Funnelsort

- Sorting algorithm
- Similar to merge sort
- **Both work and I/O optimal**
  - While being cache oblivious
  - $\Theta(n \log n)$ work
  - $\Theta((n \log n) / B)$ I/Os

Funnelsort

- **Simple algorithms run faster**
  - Less code, less CPU overhead
- Quicksort is very simple and very quick in practice
  - std::sort() in C++ STL
- Quicksort is not I/O optimal

Funnelsort

- Simple algorithms run faster
  - Less code, less CPU overhead
- **Quicksort is very simple and very quick in practice**
  - std::sort() in C++ STL
- Quicksort is not I/O optimal

Funnelsort

- Simple algorithms run faster
  - Less code, less CPU overhead
- Quicksort is very simple and very quick in practice
  - std::sort() in C++ STL
- **Quicksort is not I/O optimal**

Funnelsort

Can Funnelsort outperform Quicksort in practice?

Funnelsort

Can Funnelsort outperform Quicksort in practice?

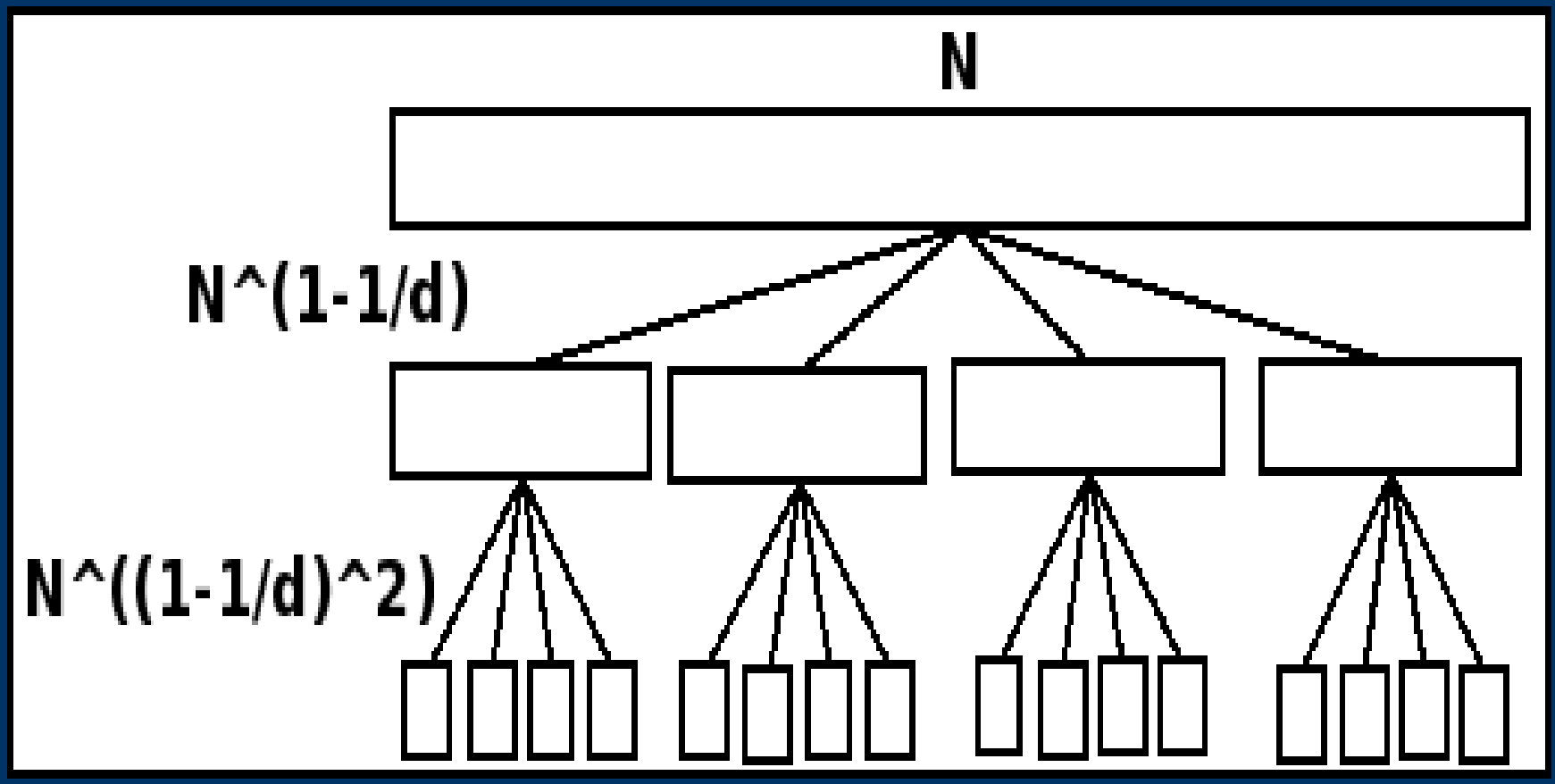
... Yes.

Funnelsort

- How does it work ?
  - Split the input into smaller groups
    - Split N elements into N^(1/d) groups of size N^(1-1/d)
  - Recursively sort each group
  - Merge the sorted groups together

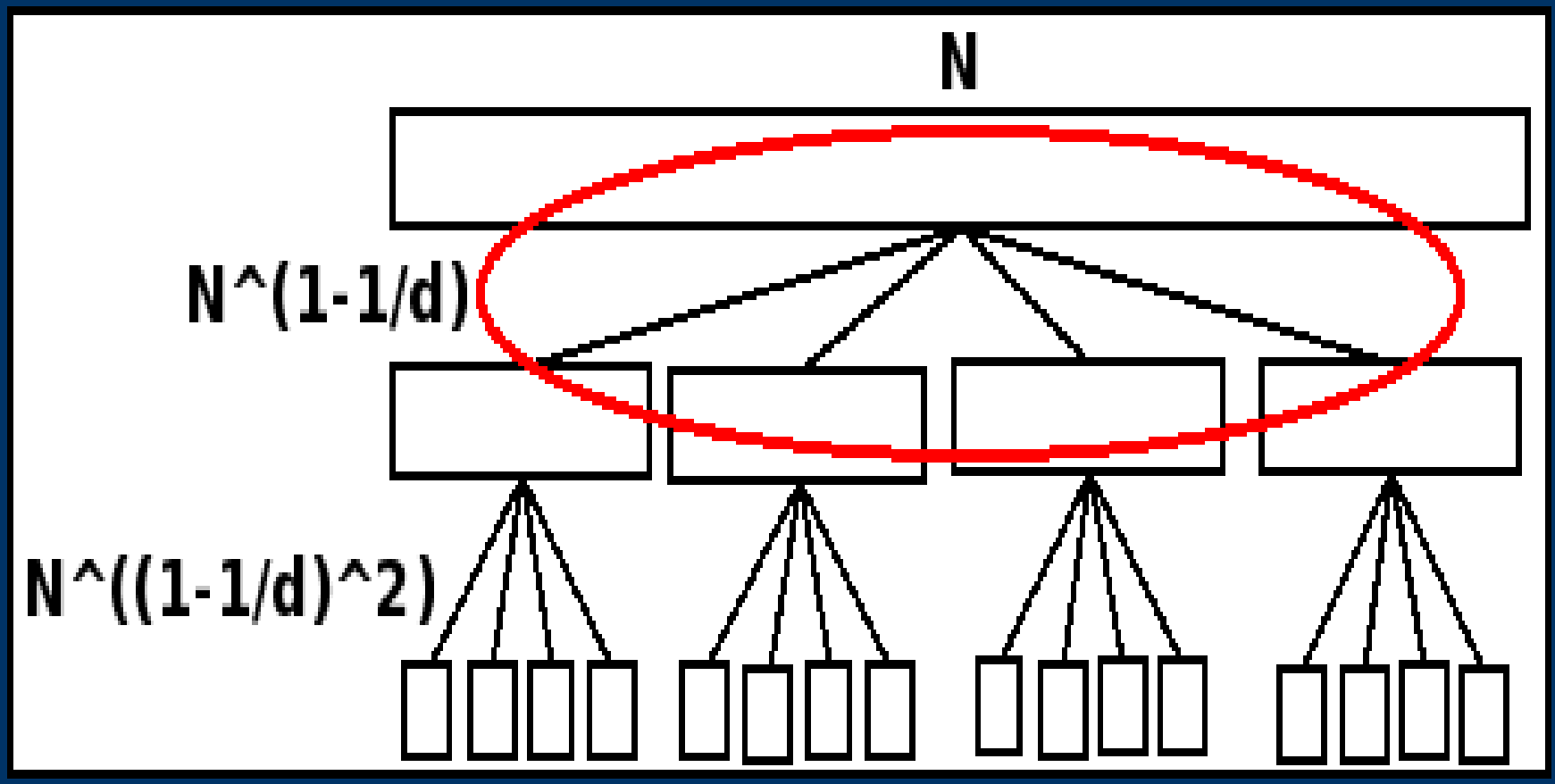
Funnelsort

- Looks a lot like standard merge sort



Funnelsort

- Do merging in an I/O optimal way



Funnelsort - Merging Process

- **Merge together *k* groups of sorted input**
- Use a tool called *k*-merger
- Tree structure
- Sorts from the leaves of the tree up to the root

Funnelsort - Merging Process

- Merge together *k* groups of sorted input
- **Use a tool called *k*-merger**
- Tree structure
- Sorts from the leaves of the tree up to the root

Funnelsort - Merging Process

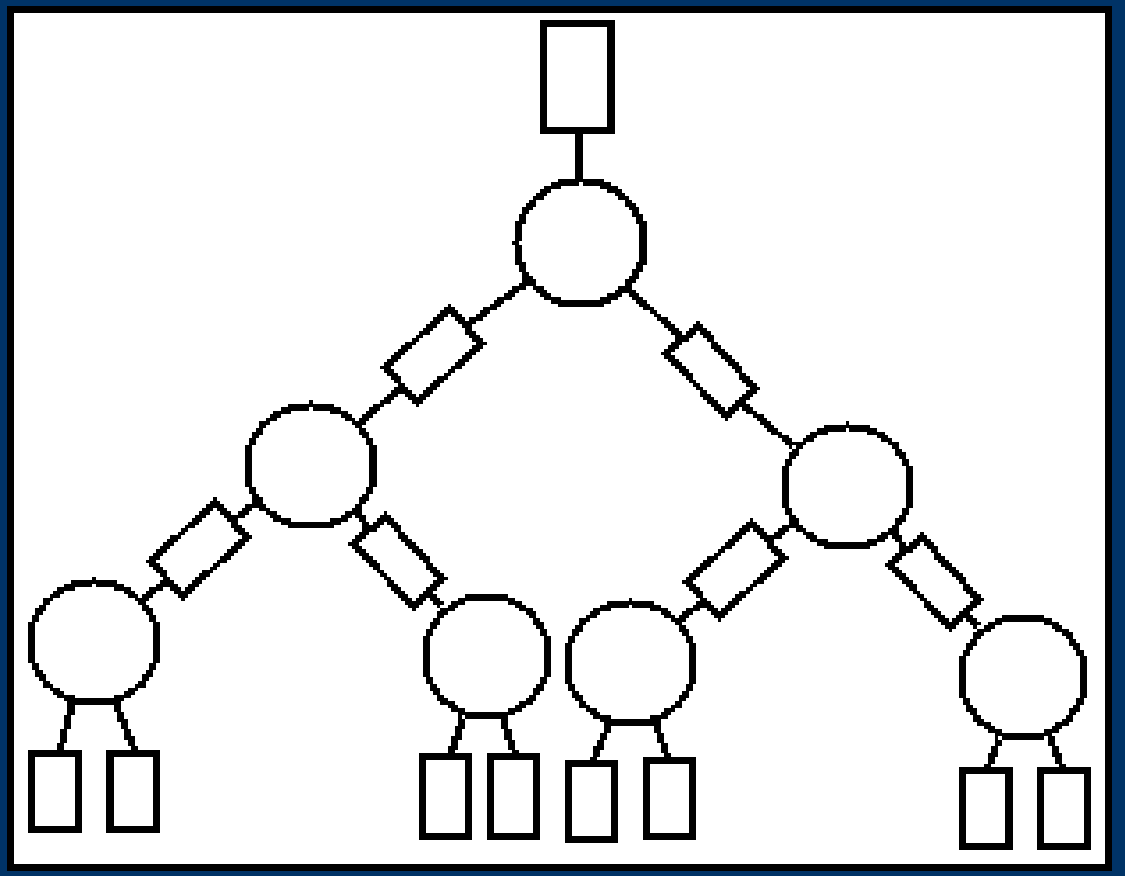
- Merge together *k* groups of sorted input
- Use a tool called *k*-merger
- **Tree structure**
- Sorts from the leaves of the tree up to the root

Funnelsort - Merging Process

- Merge together *k* groups of sorted input
- Use a tool called *k*-merger
- Tree structure
- **Sorts from the leaves of the tree up to the root**

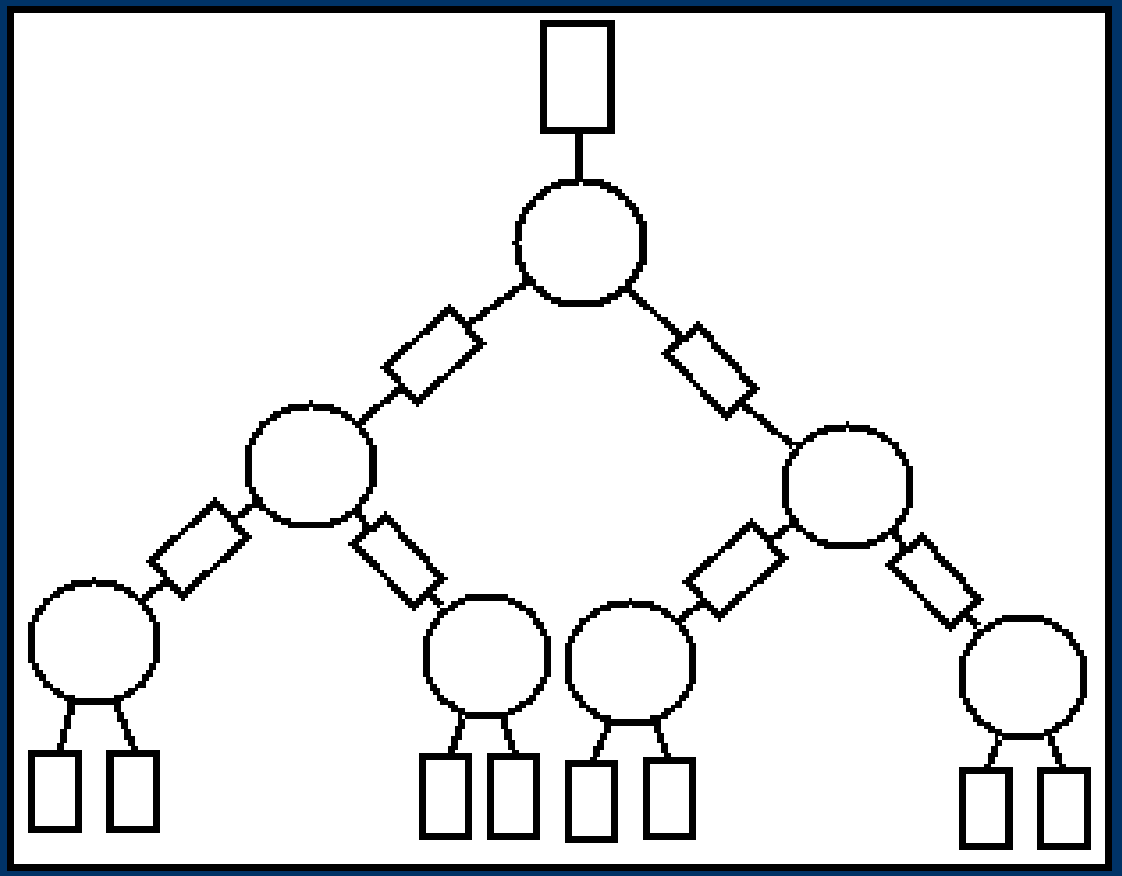
Funnelsort - k-merger

- k-merger (k=8)
- Boxes are buffers
- Each node merges into buffer above it



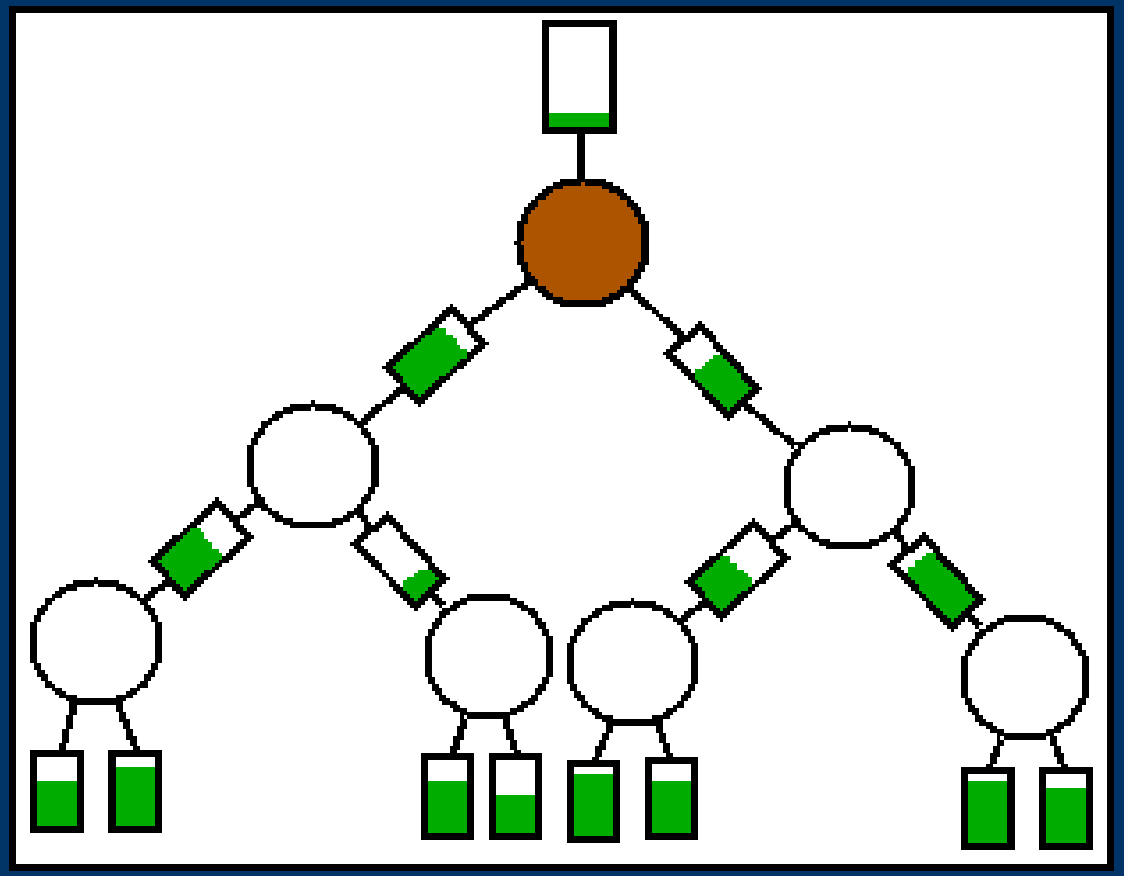
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



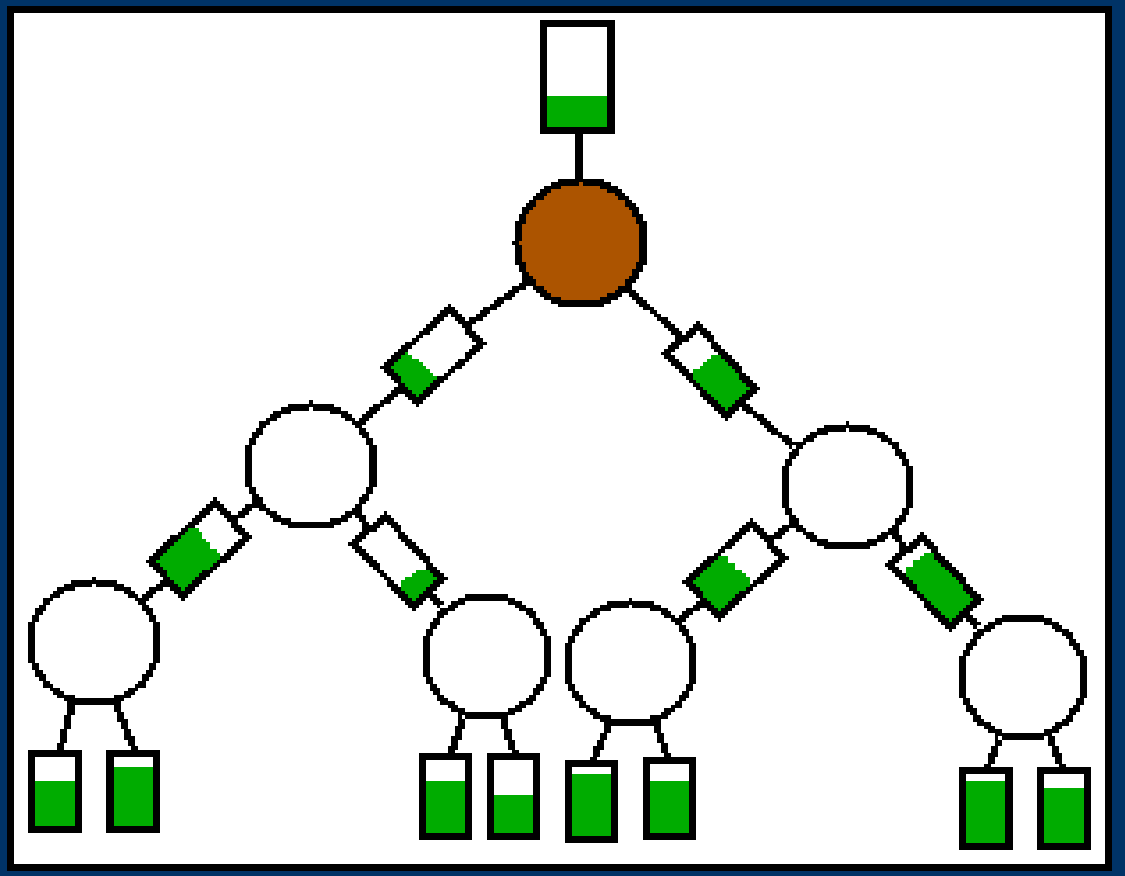
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



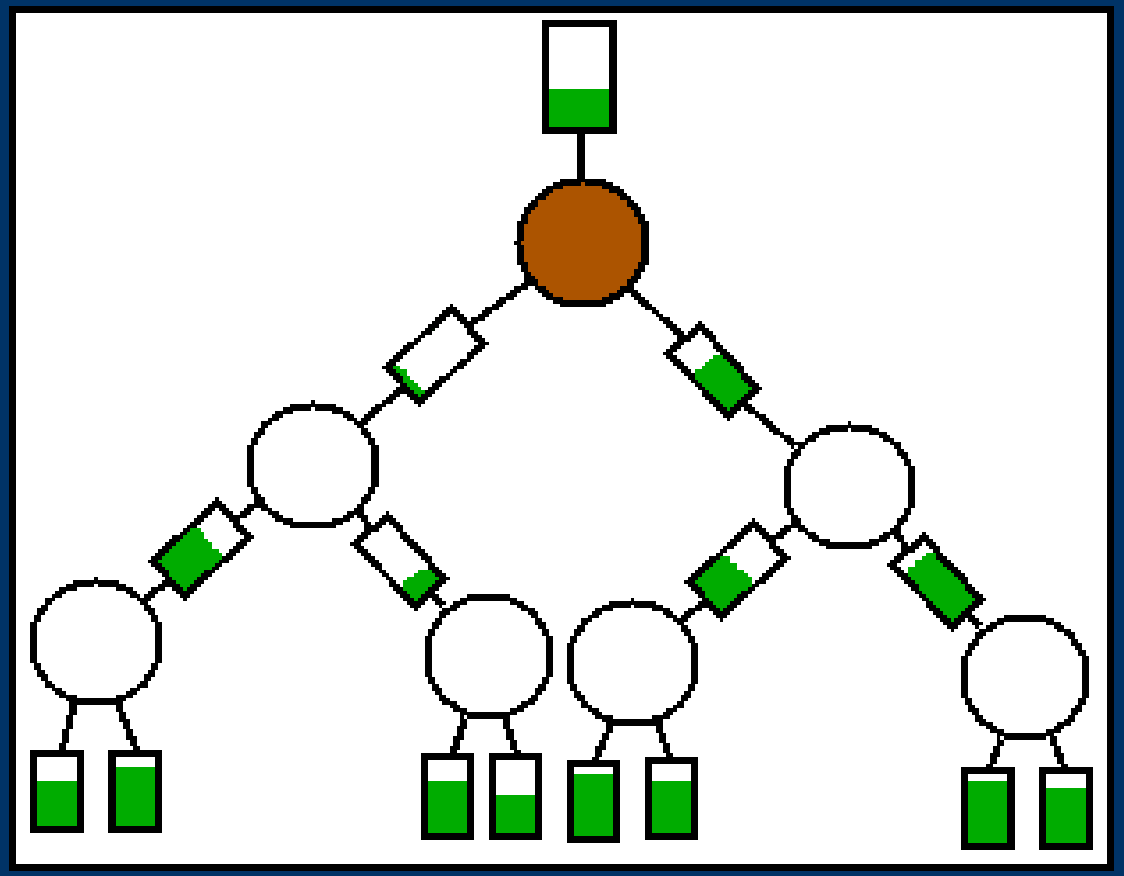
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



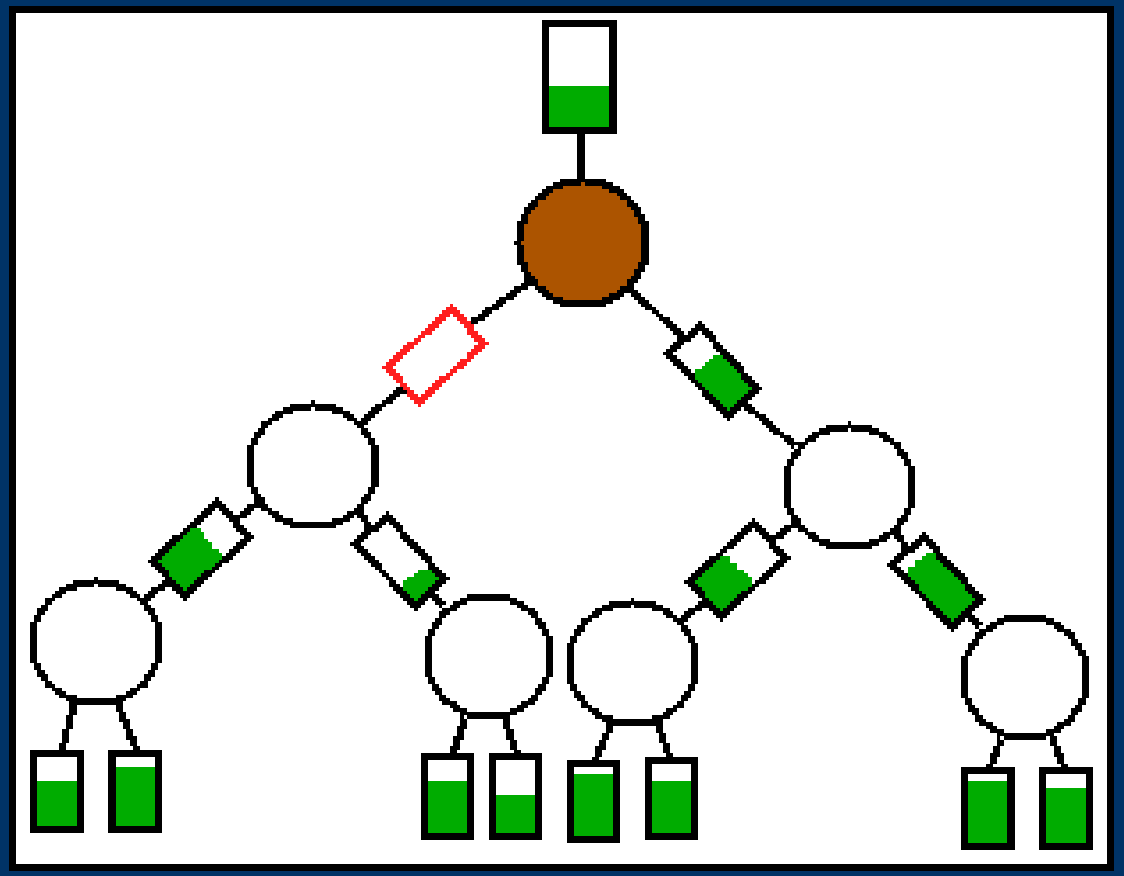
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



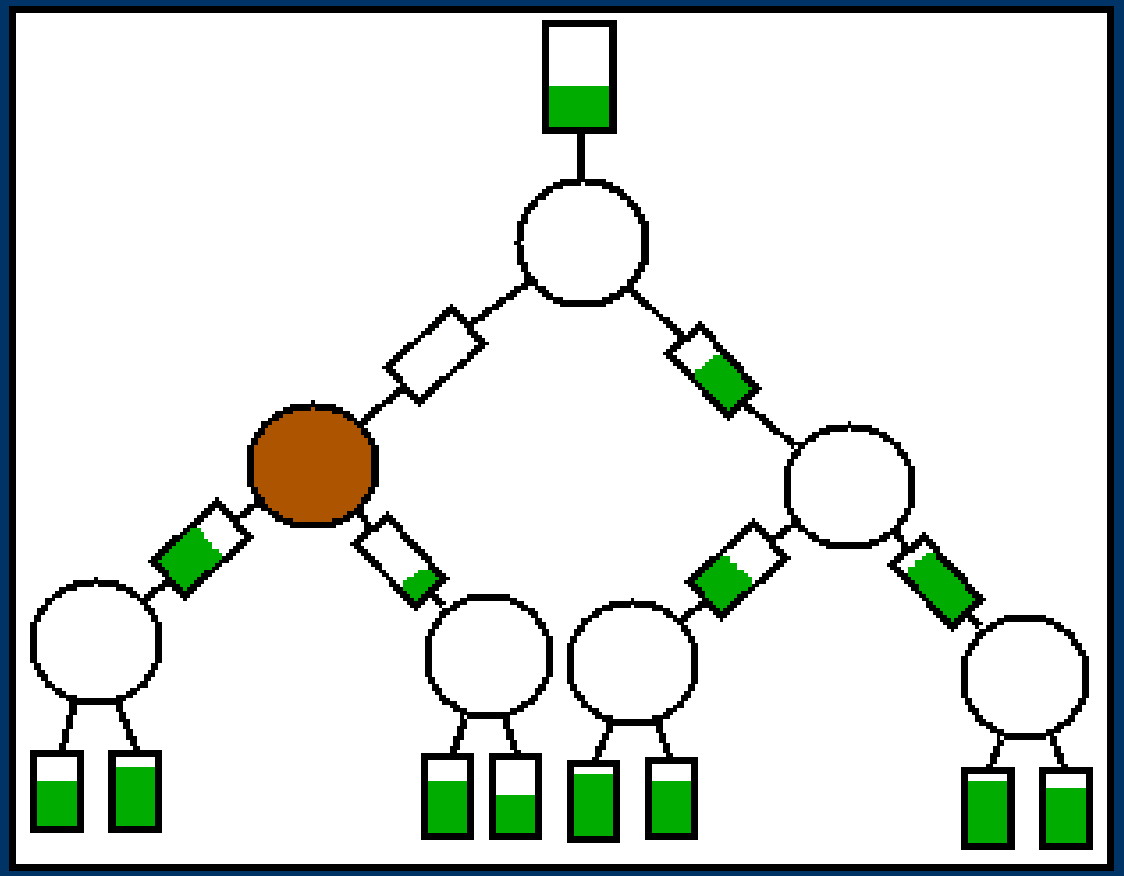
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



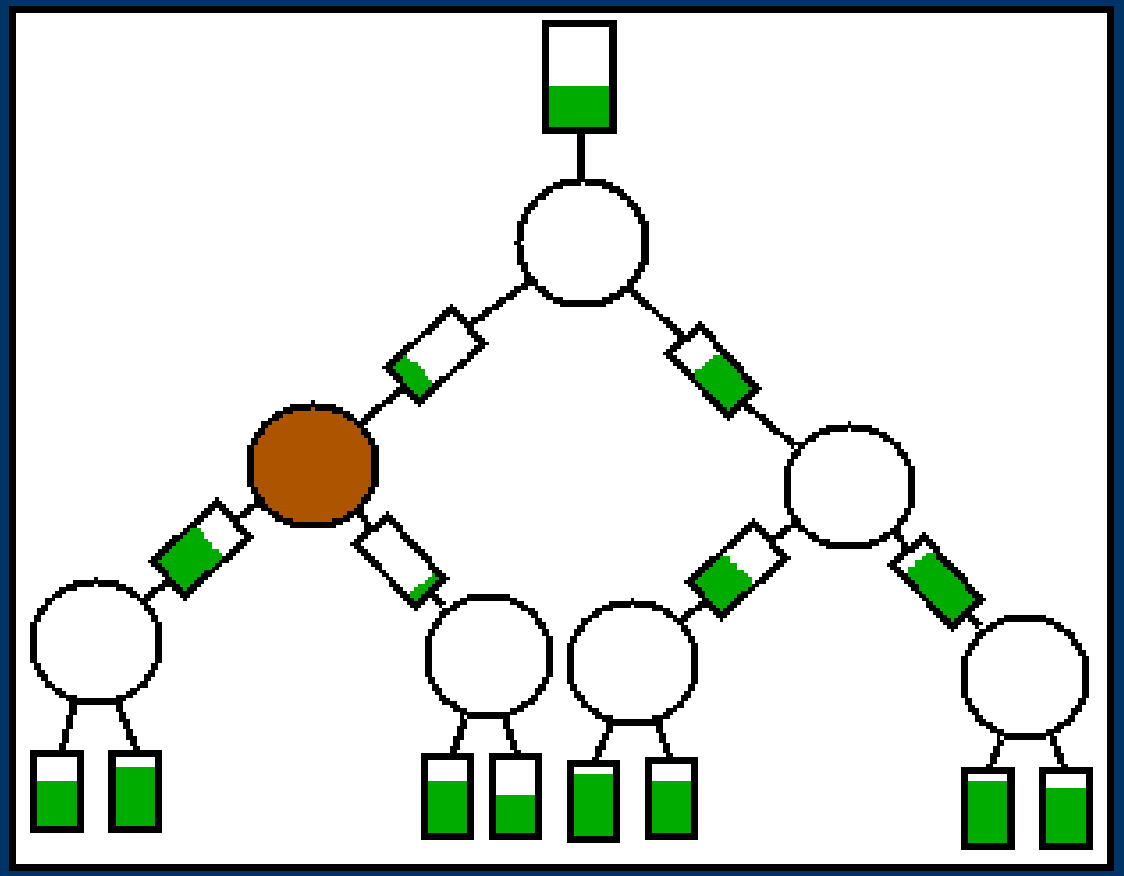
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



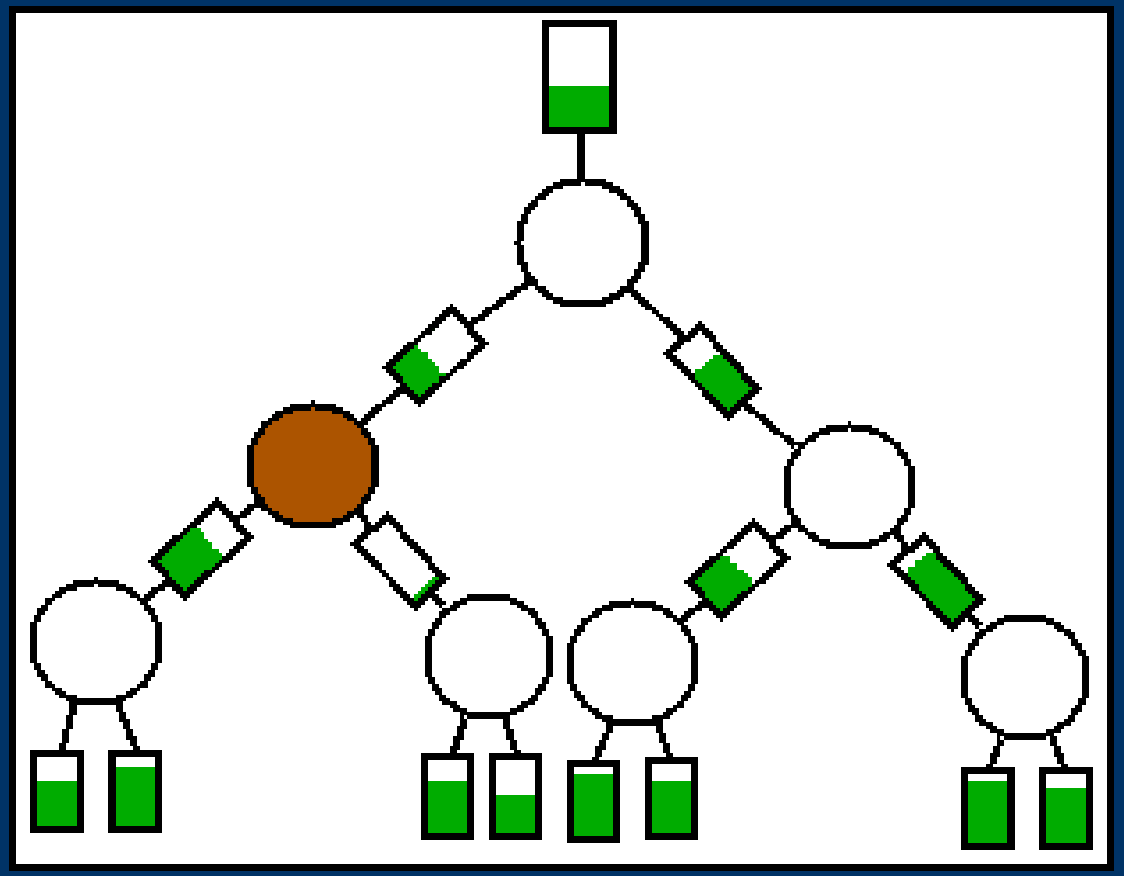
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



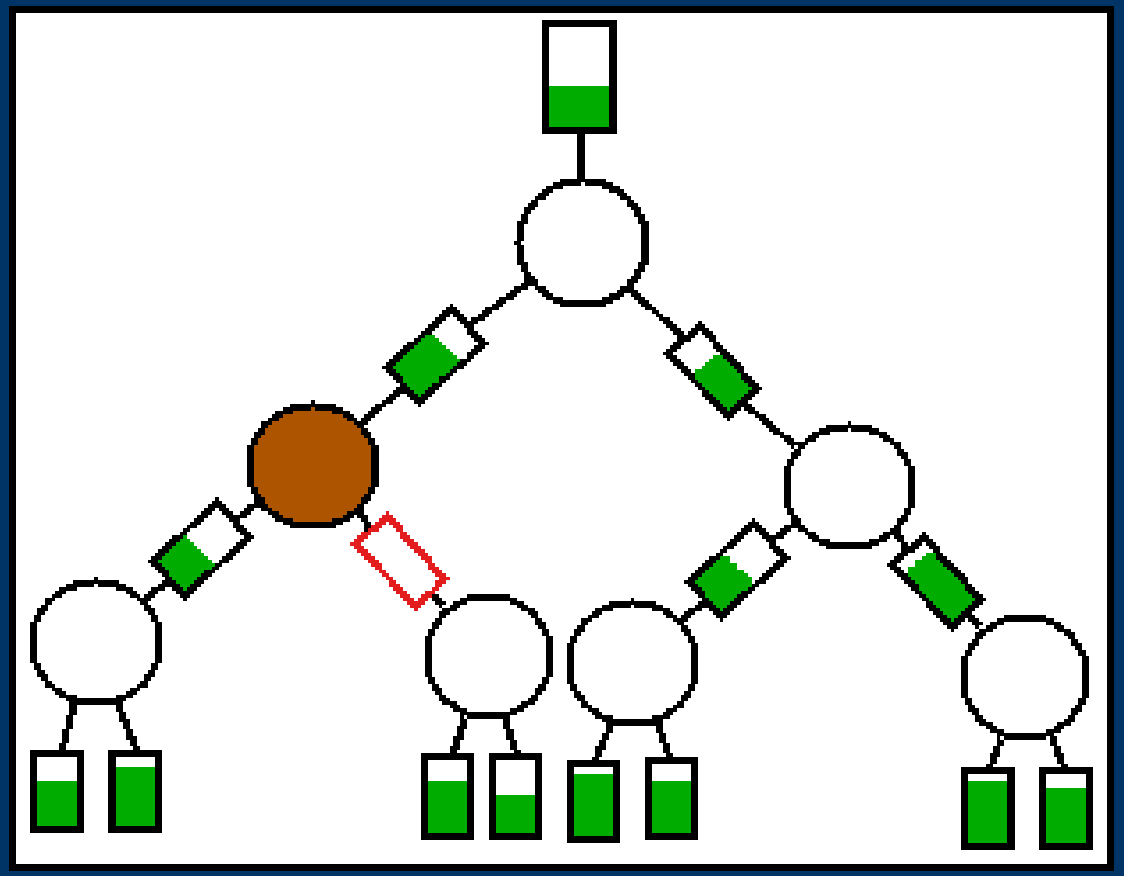
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



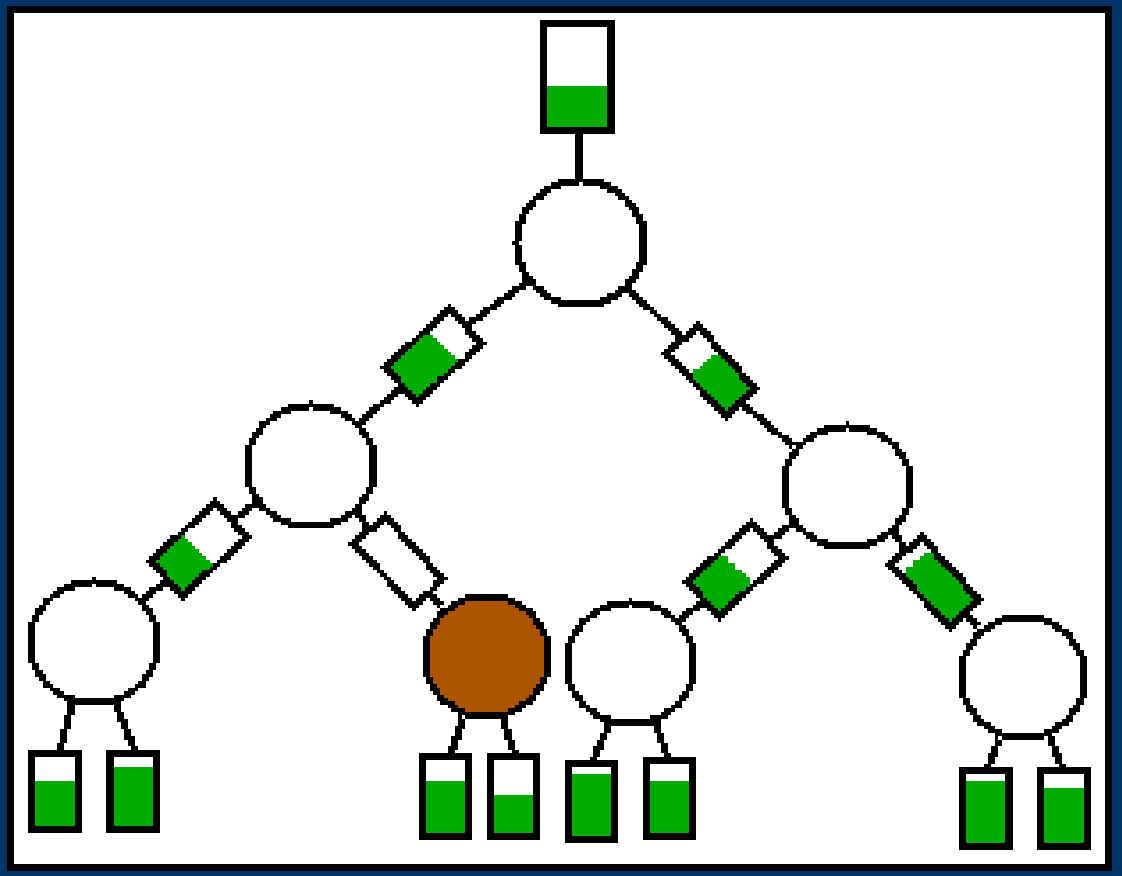
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



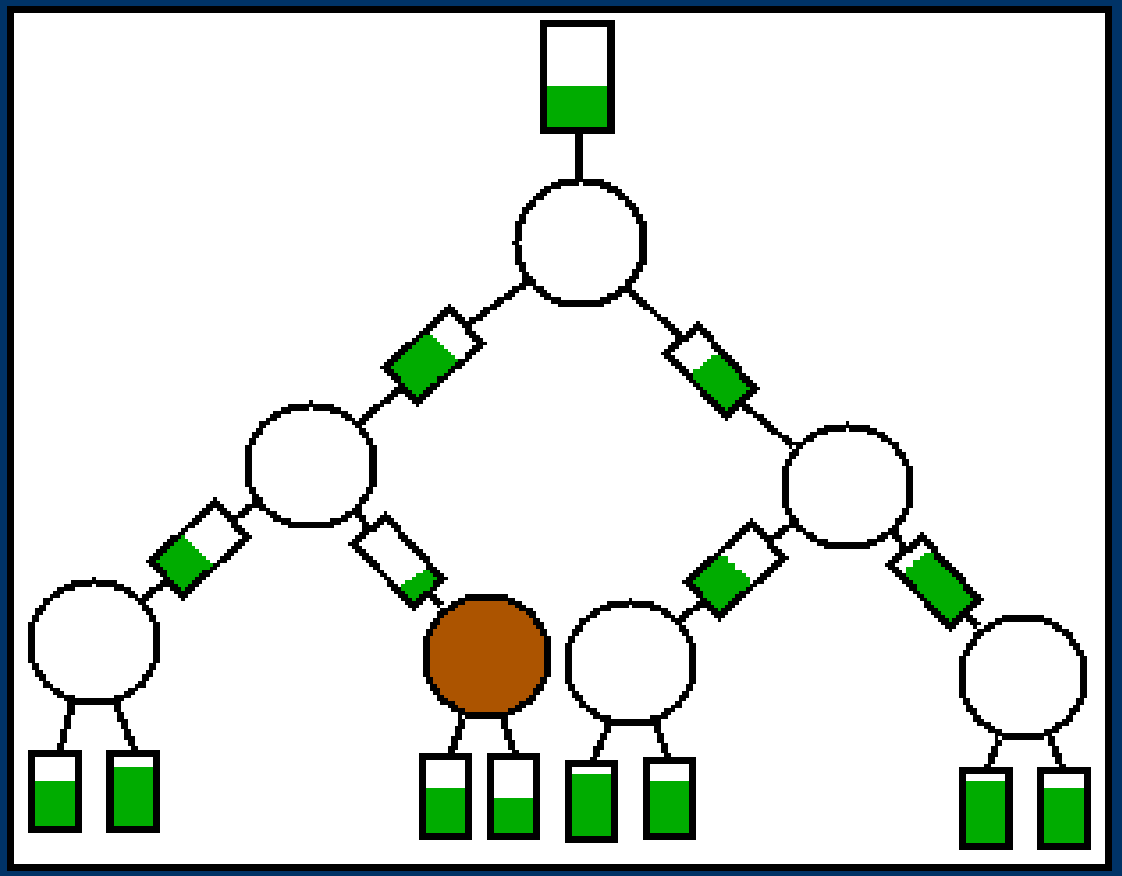
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



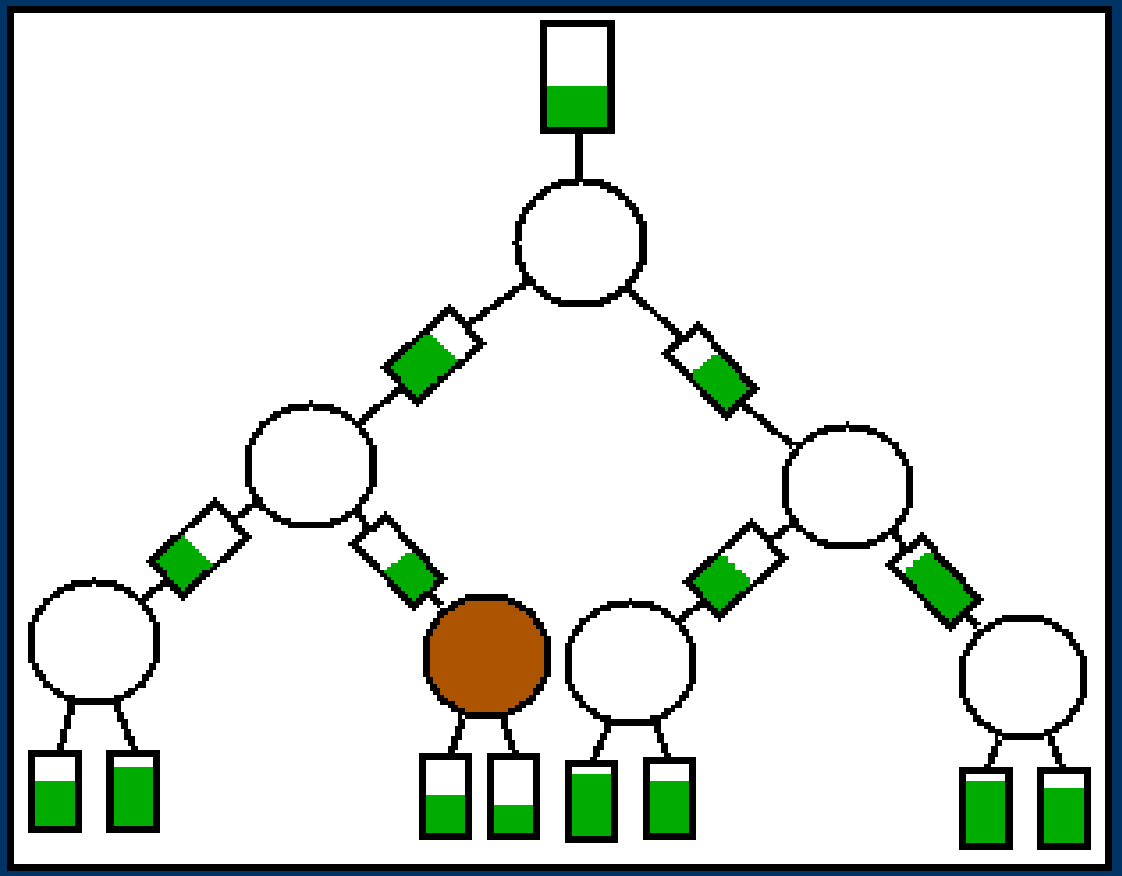
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



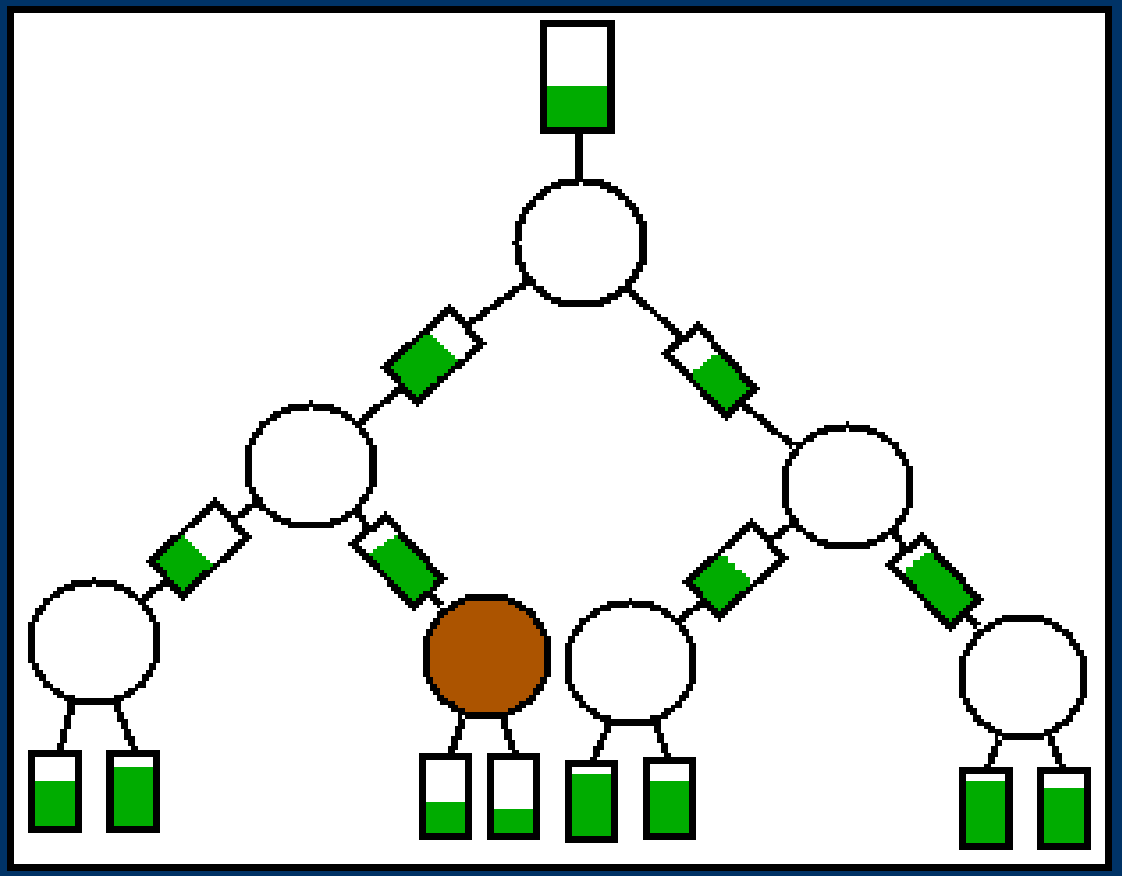
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



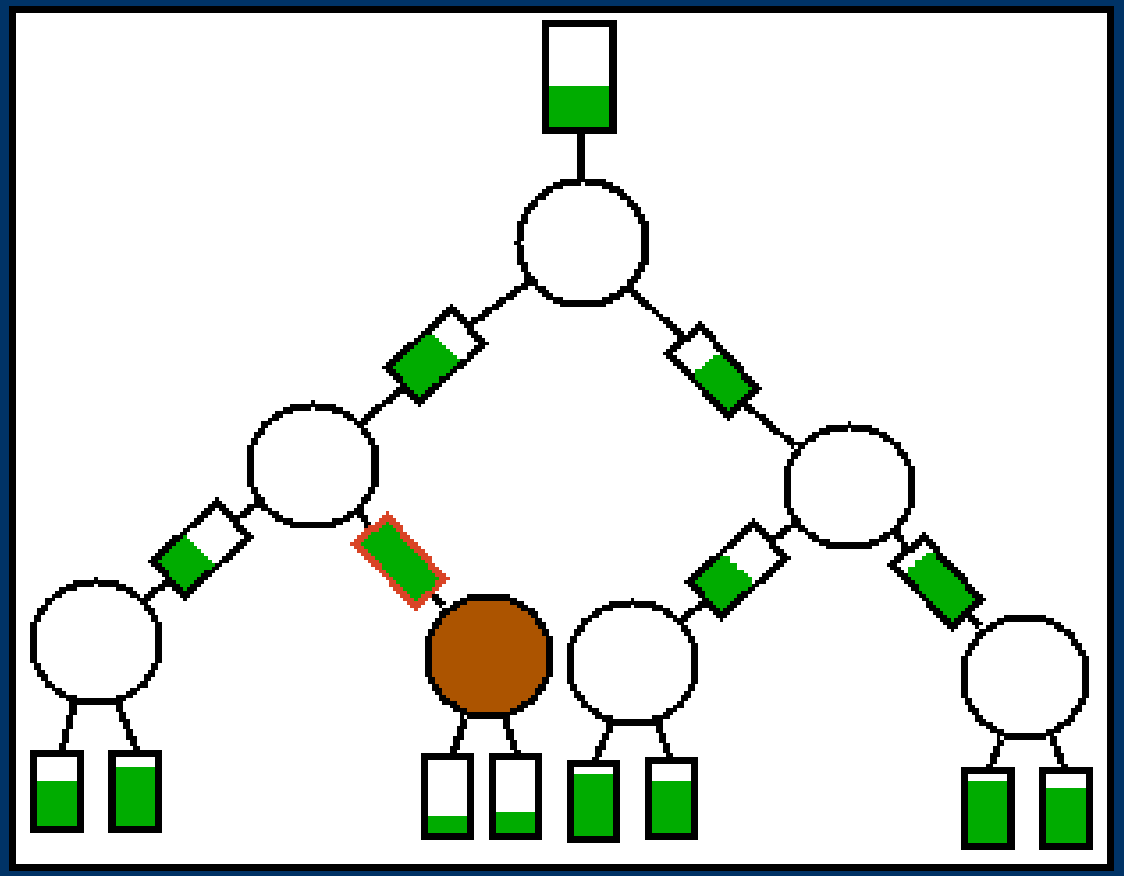
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



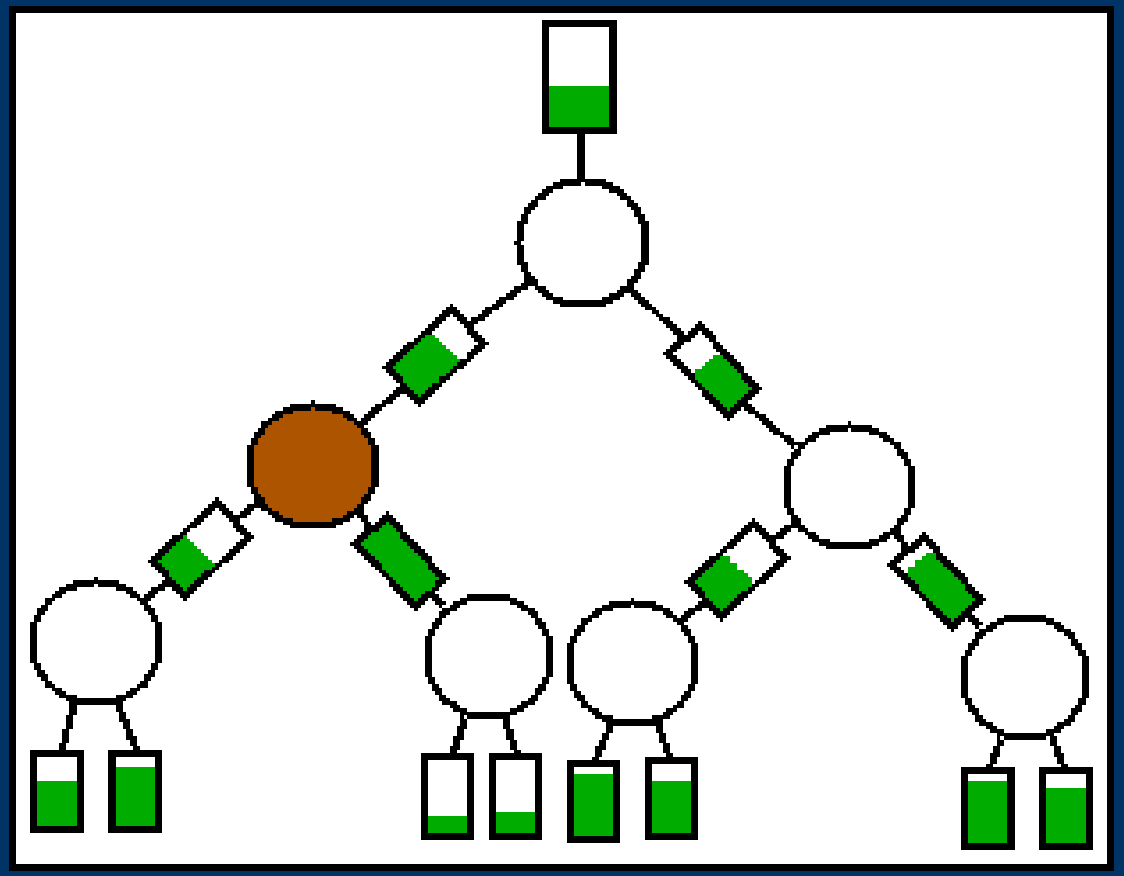
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



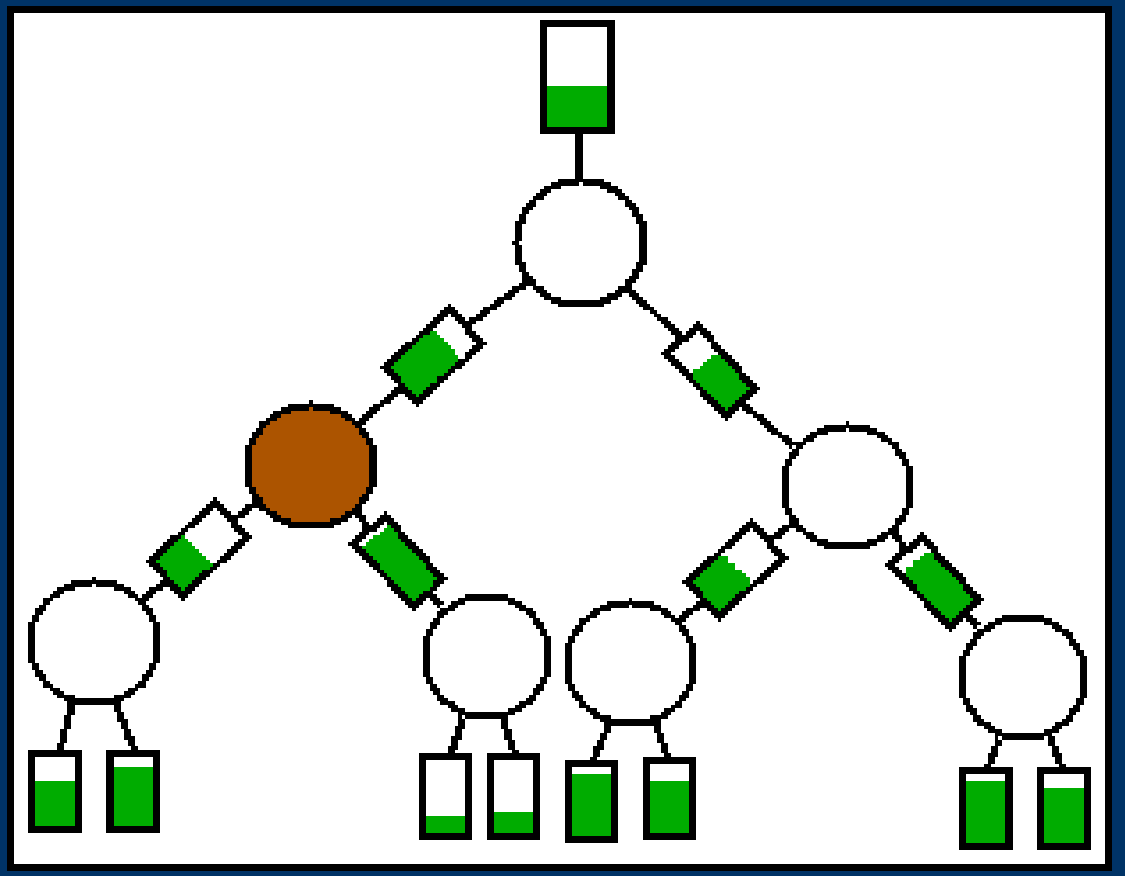
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



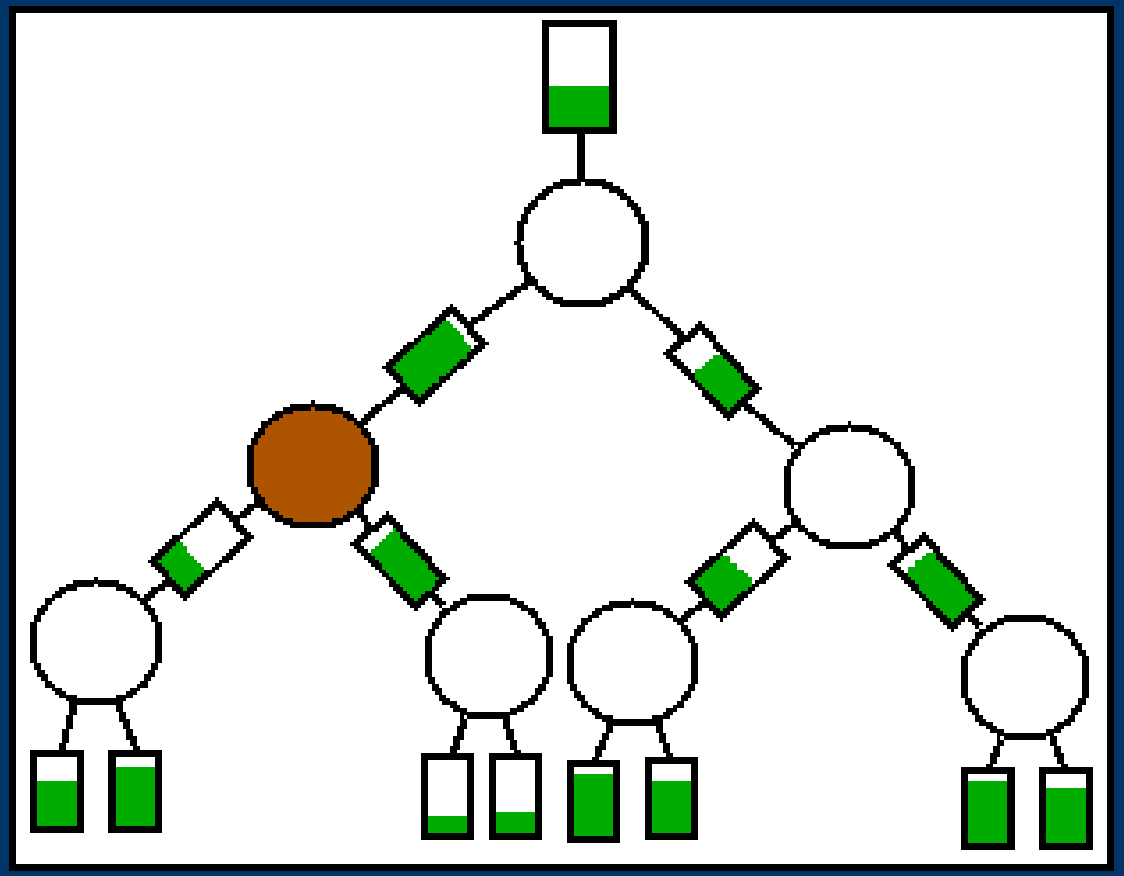
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



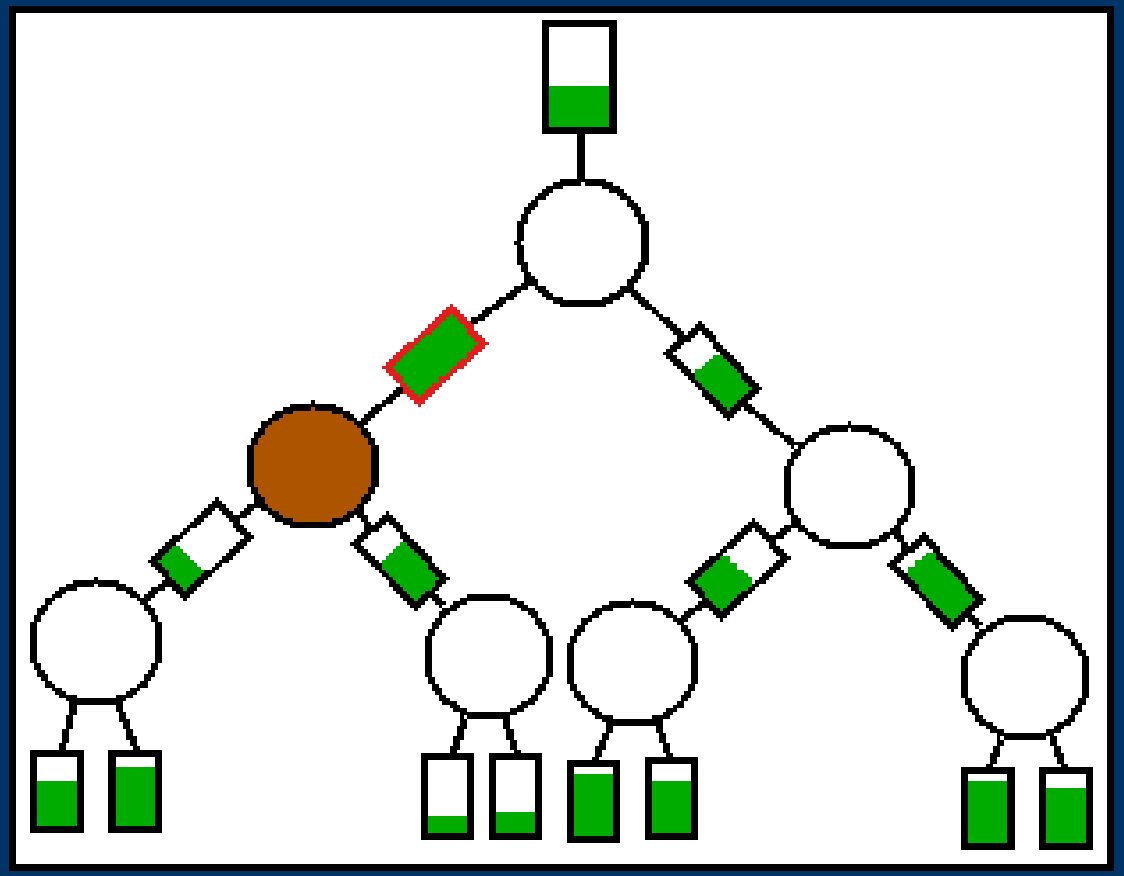
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



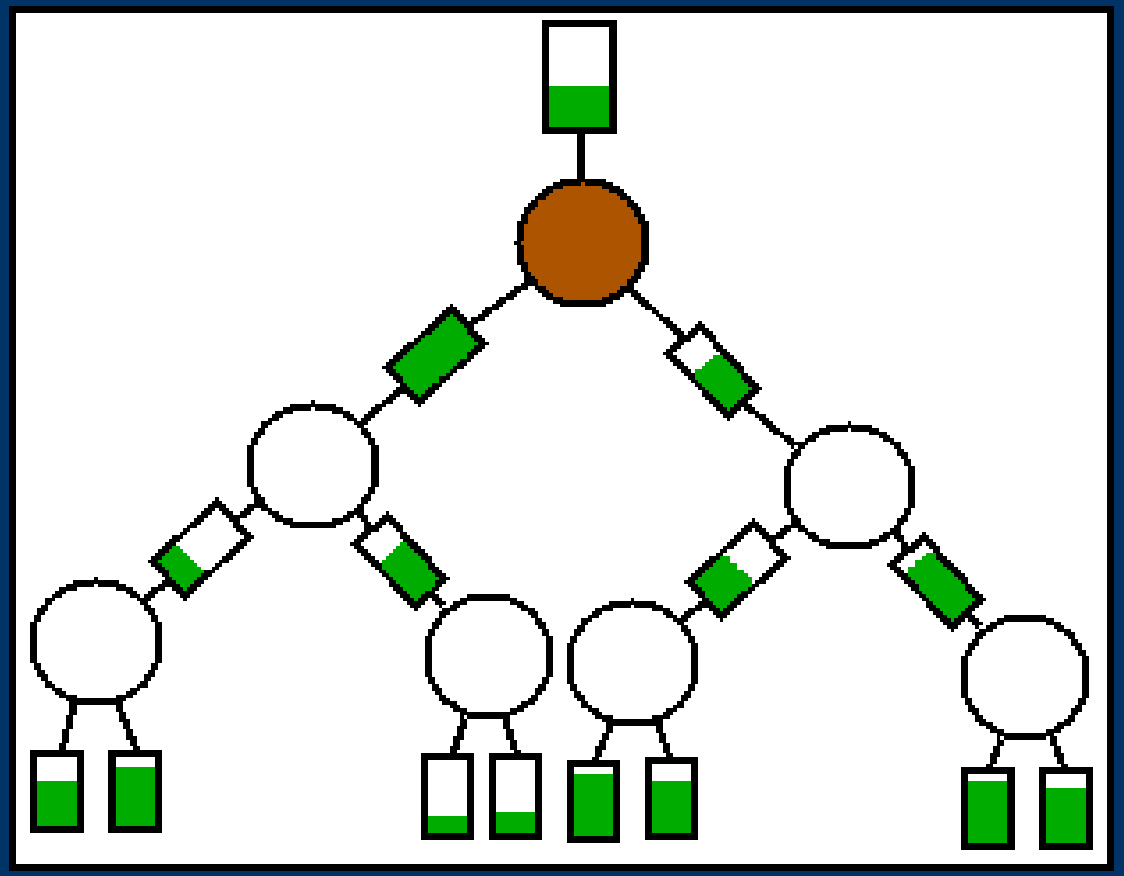
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



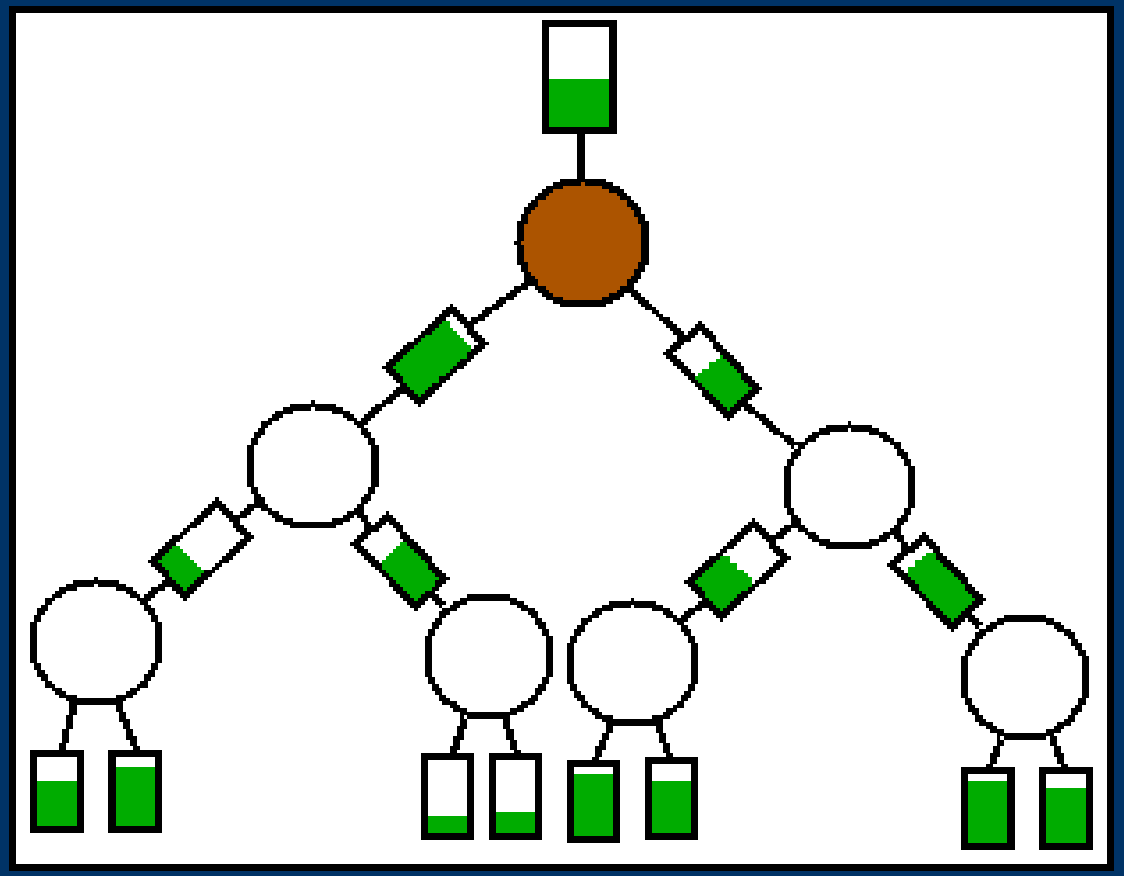
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



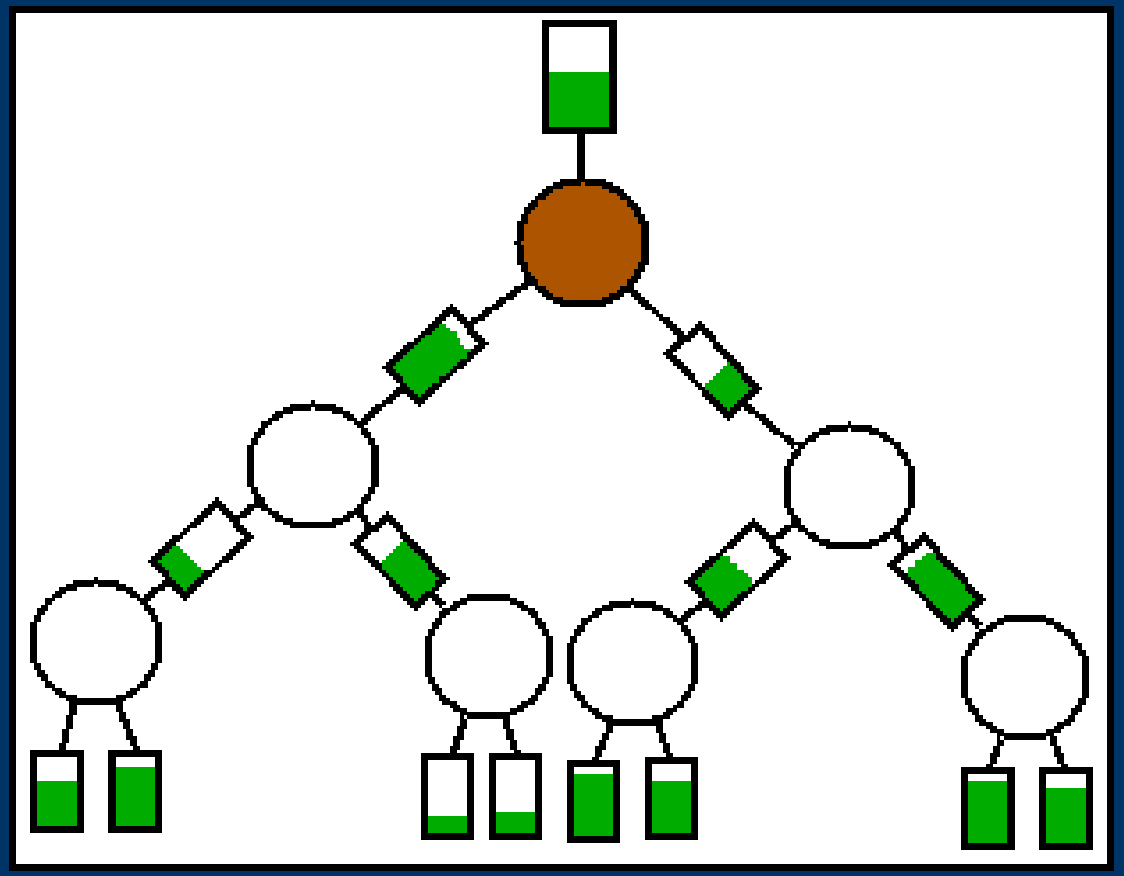
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



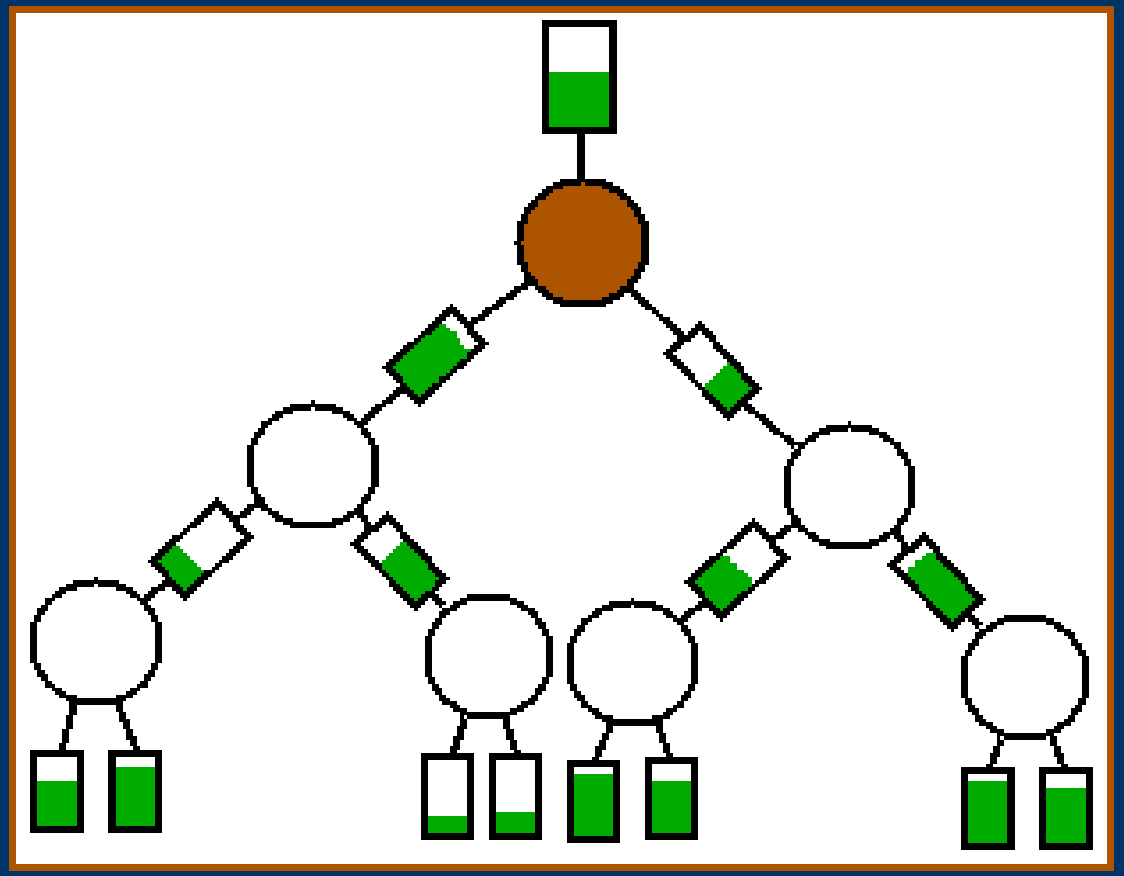
Funnelsort - k-merger

- Simple rules
- Start at the root
- Merge until:
  - input is empty
    or
  - output is full



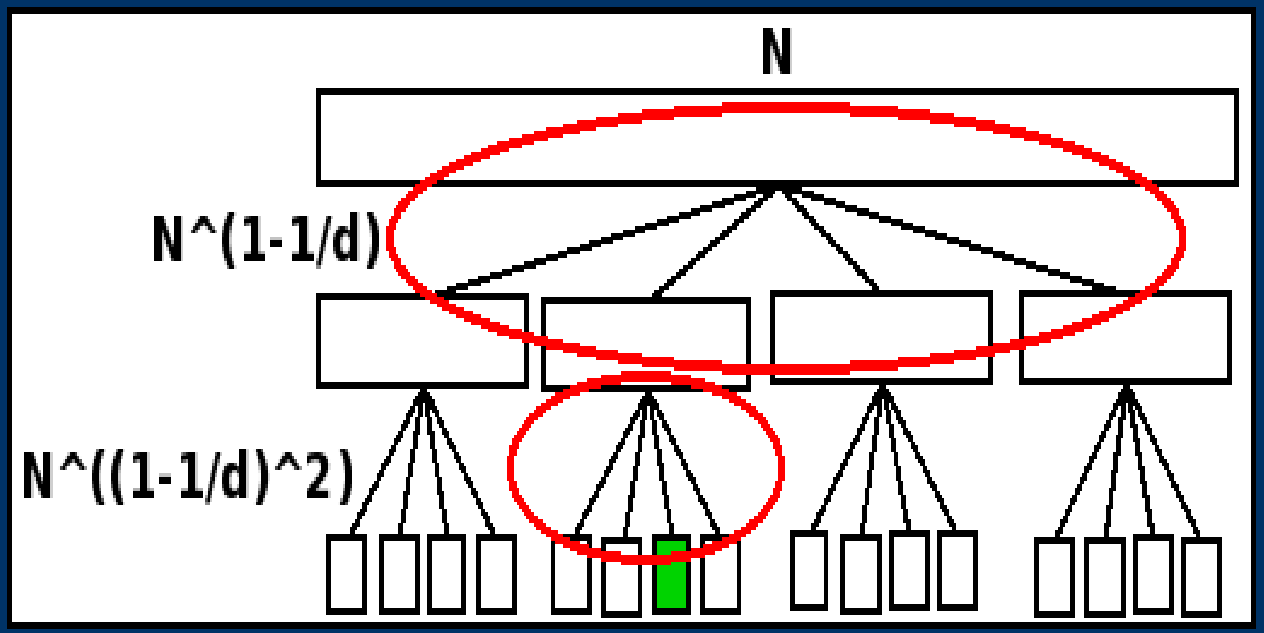
Funnelsort - k-merger

- And so on..
- Until input buffers are empty
- Output buffer is full



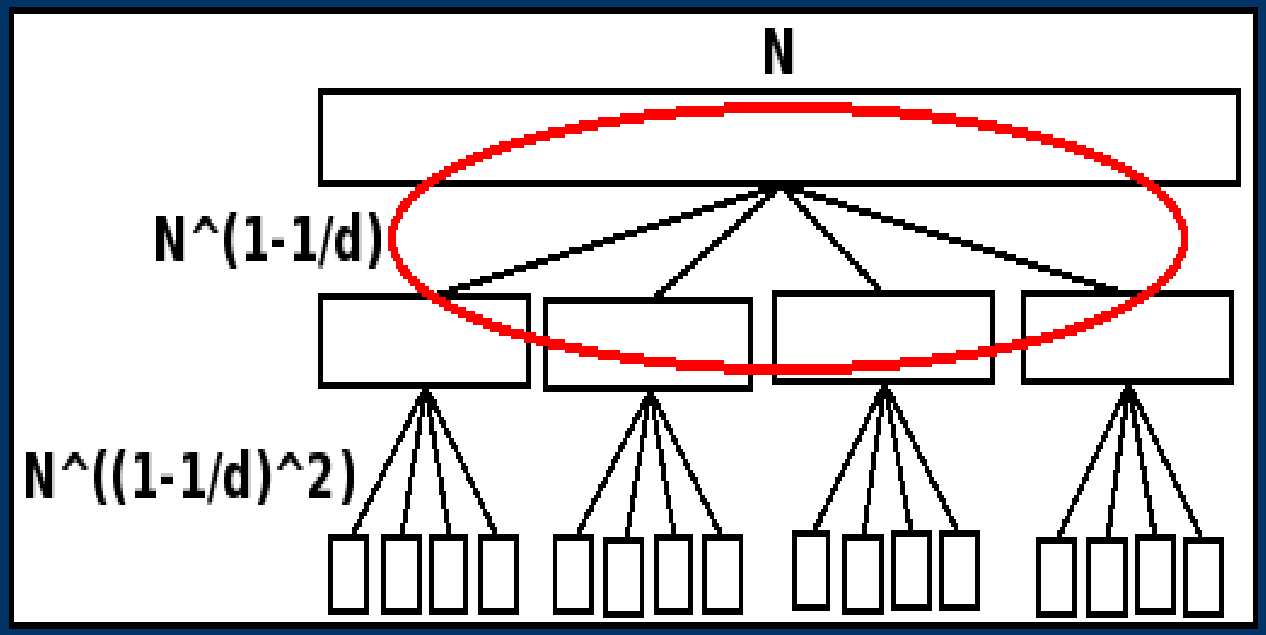
Funnelsort – I/O cost

- I/O cost to sort one element:
  - Sum of I/Os at each merge step



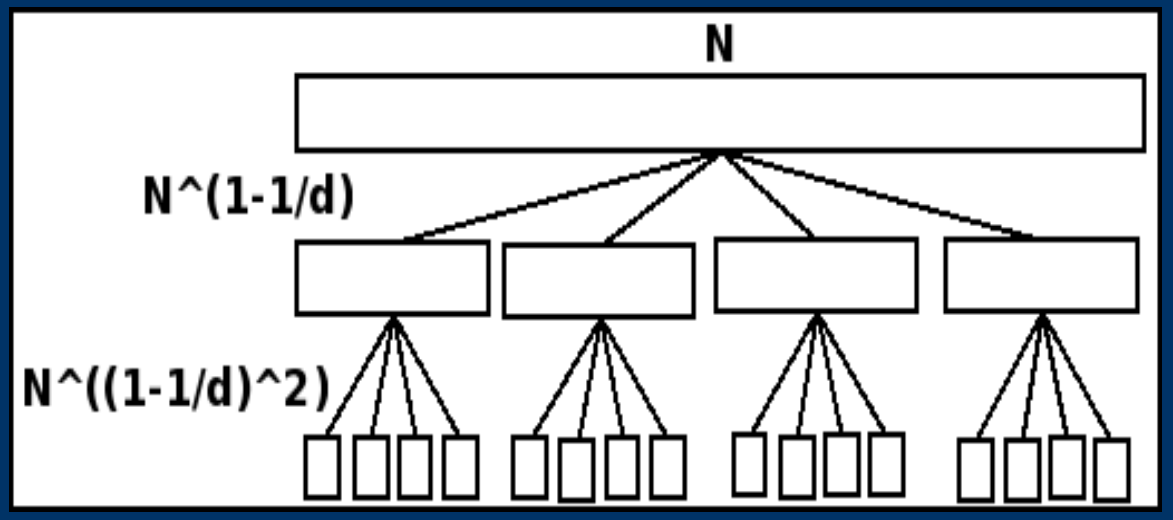
Funnelsort – I/O cost

- Using a k-merger (merging *m* elements)
  - Each element costs **O(log m / B)** I/O's
- *m* is the size of the inputs being merged



Funnelsort – I/O cost

- Total I/O cost to sort *one* element:
  - **O((log N) / B)** I/O's
- Total I/O cost to sort *N* elements:
  - **O((N log N) / B)** I/O's



Funnelsort – I/O cost

- I/O cost to sort one element:
  - Sum of I/Os at each merge step
- Total cost to sort 1 element is:

$$O((d \log n) / B)$$

Comparison

- Comparison with std:sort() from g++ 4.1.2 on Fedora 8
- Single quad-core processor
- 4KB CPU cache
- 8000MB memory (488 MB data set)

Comparison

- Sorting with a single thread
  - Quicksort is able to beat Funnelsort
  - By factor of 1.20
  - I/O isn't the largest factor
- Sorting with 4 threads
  - Funnelsort is faster than Quicksort
  - By factor of 1.06

Comparison

- With four cores competing:
  - Funnelsort improved relative to Quicksort by 22%
  - Funnelsort was able to outperform Quicksort

Comparison

- Fastest sorting methods
  - Using a bucket sort to merge results between processors

1)Funnelsort with 4 cores (5.10s)

2)Quicksort with 4 cores (5.59s)

3)Quicksort with 1 core (10.72s)

4)Funnelsort with 1 core (12.51s)

Conclusion

- When I/O access is limited, I/O efficiency becomes important
- Multi-core processors highlight the need for I/O efficient algorithm design
  - Even when data sets fit entirely inside main memory
- Utilizing multiple cores with I/O in mind provides the best solutions