A Low-Latency FPGA-based Infrastructure for HFT Systems

Andrew Boutros, Brett Grady and Mustafa Abbas

ECE1373 Course Project

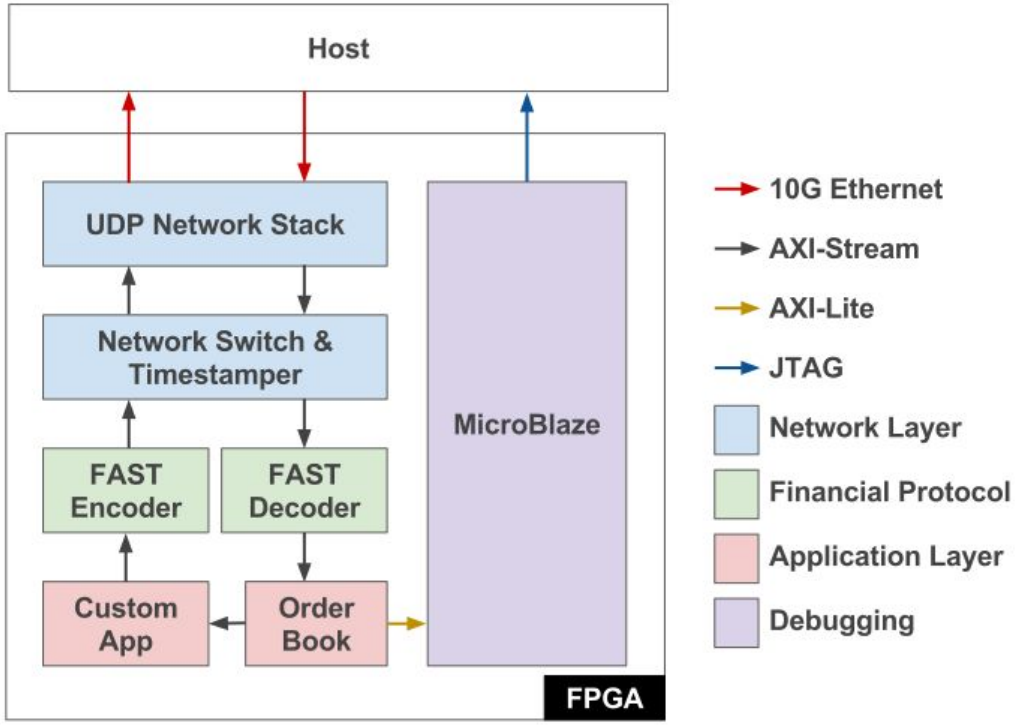
UNIVERSITY OF TORONTO

Recap

High Frequency Trading (HFT) is the rapid automated exchange of financial instruments using networked computers.

Our Application:

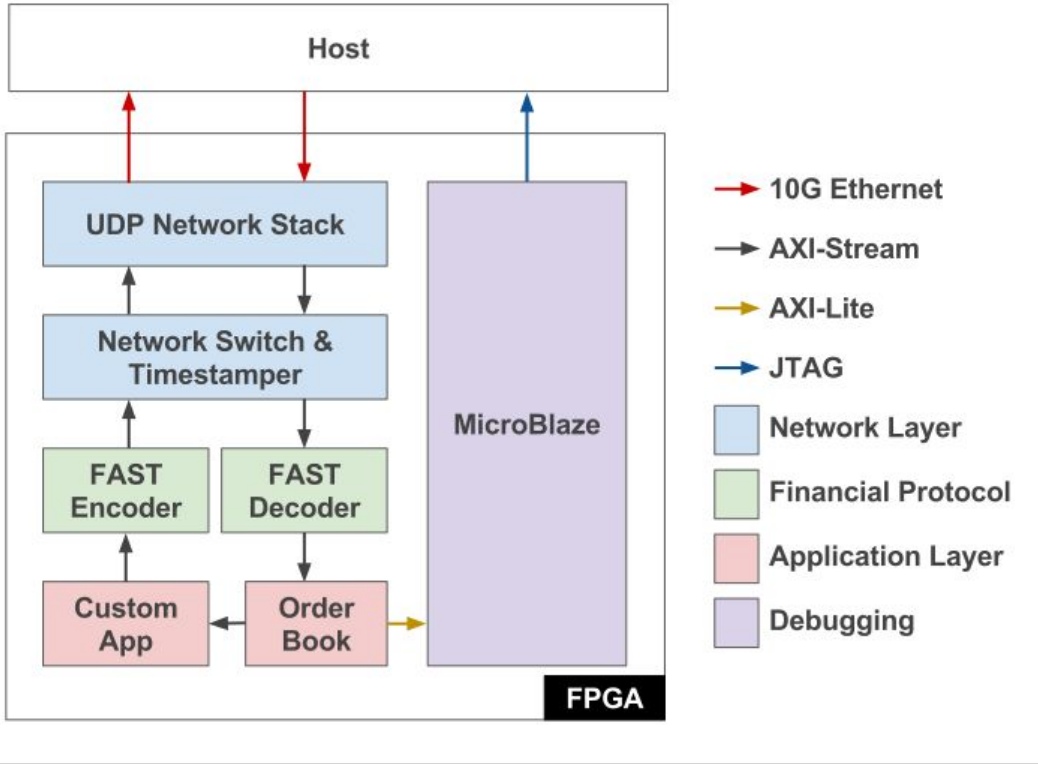
- Receives financial data in compressed form (**FAST**) over the network
- Decode incoming packet (market update)
- Update model and possibly make a trading decision (bid/ask)
- Encode outgoing data (market order)
- Send bid/ask over network with latency estimate.



Project Goals

Design an FPGA-based infrastructure for HFT that:

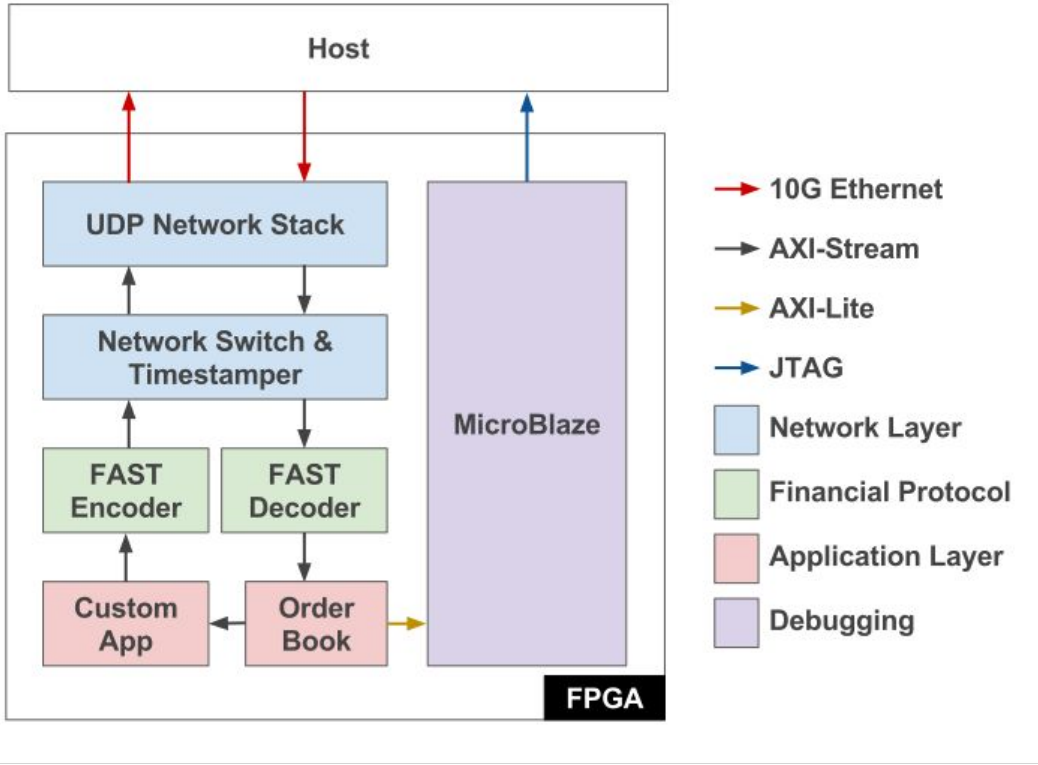
- Abstracts the details of networking & financial encoding/decoding.
- Handles order book keeping (pre-processing) to ease development of HFT algorithms on FPGAs.
- Is extensible and easy to interface with, but still has very low round-trip latency (<1 µs including the network stack).



System Overview

Our system has five modules:

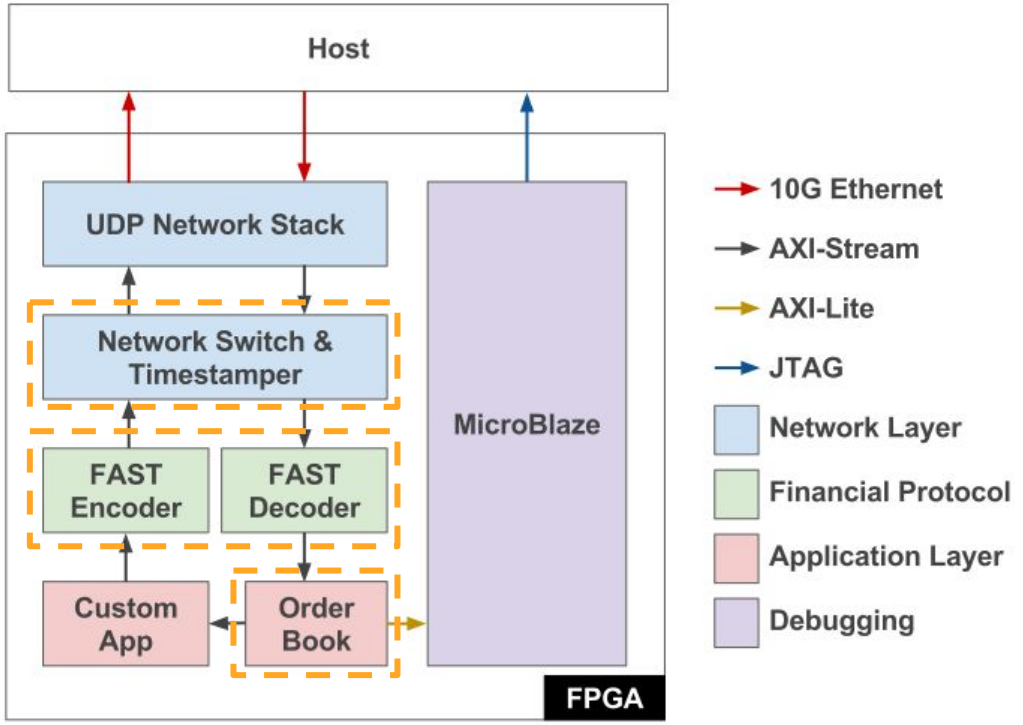
- **UDP Network Stack** - Xilinx's UDP network stack.
- **Network Switch/Timestamper** - Timestamps incoming and outgoing packets to track latency.
- **FAST Encoder/Decoder** - Performs FAST protocol conversion on incoming and outgoing packets
- **Order Book** - Sorts all current valid bid and ask orders by price, and passes these top values to the application layer.
- **Application Layer** - Simple demo "client" hardware that uses the order book data to execute trades.



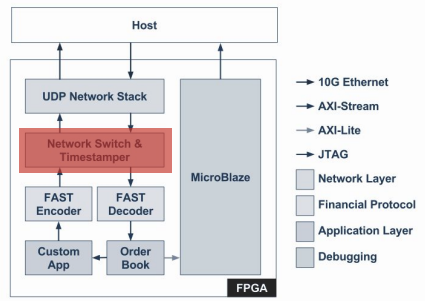
System Overview

Our system has five modules:

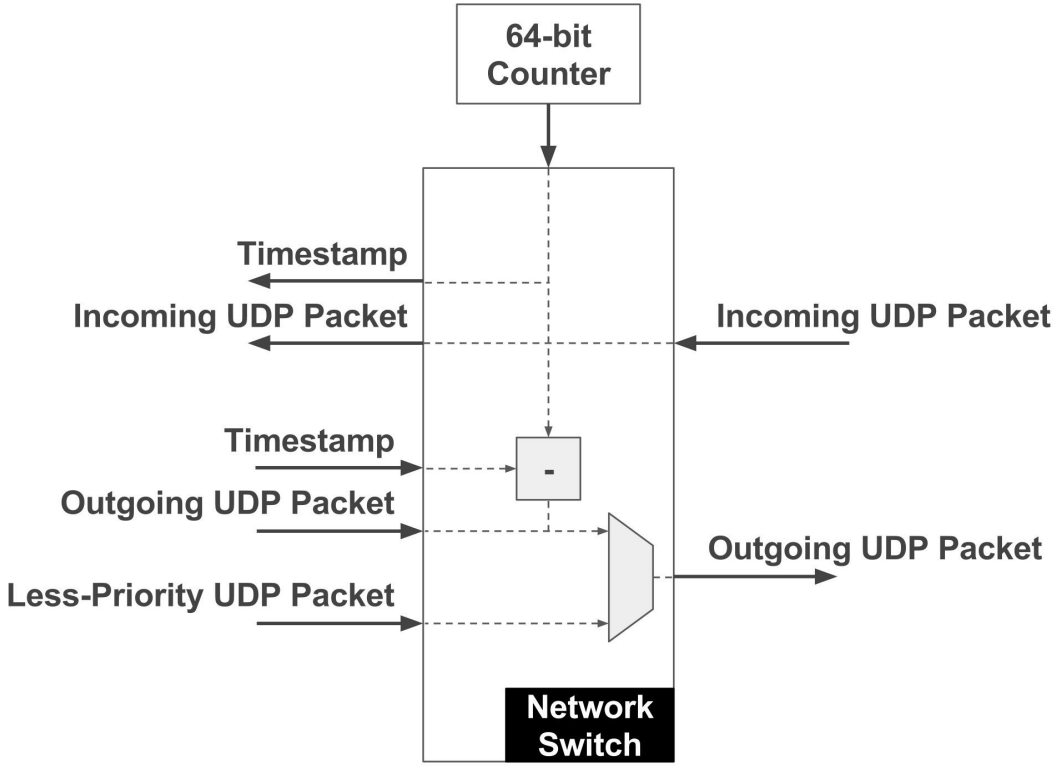
- **UDP Network Stack** - Xilinx's UDP network stack.
- **Network Switch/Timestamper** - Timestamps incoming and outgoing packets to track latency.
- **FAST Encoder/Decoder** - Performs FAST protocol conversion on incoming and outgoing packets
- **Order Book** - Sorts all current valid bid and ask orders by price, and passes these top values to the application layer.
- **Application Layer** - Simple demo "client" hardware that uses the order book data to execute trades.



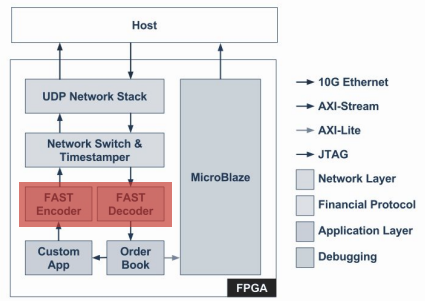
Network Switch & Timestamper



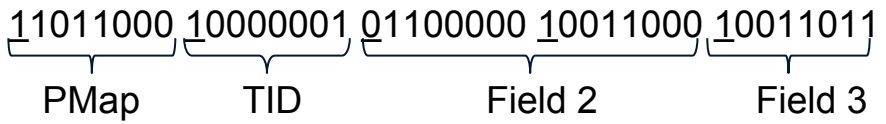
- Automatically tags incoming packets with a timestamp
- This timestamp is passed throughout the the downstream system unchanged.
- If an incoming message *triggers an order*, that message's timestamp gets transmitted back to the timestamper alongside the outgoing order.
- The timestamp then computes the latency of the packets and appends it to the packet.
- Also provides multiplexing on the transmit side for monitoring (this was not used in the final project iteration).



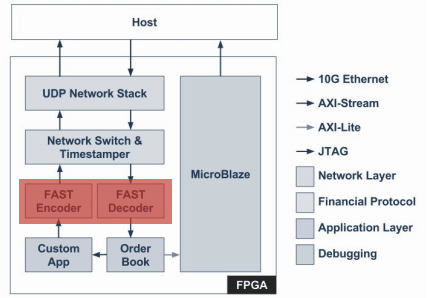
FAST Protocol - Recap



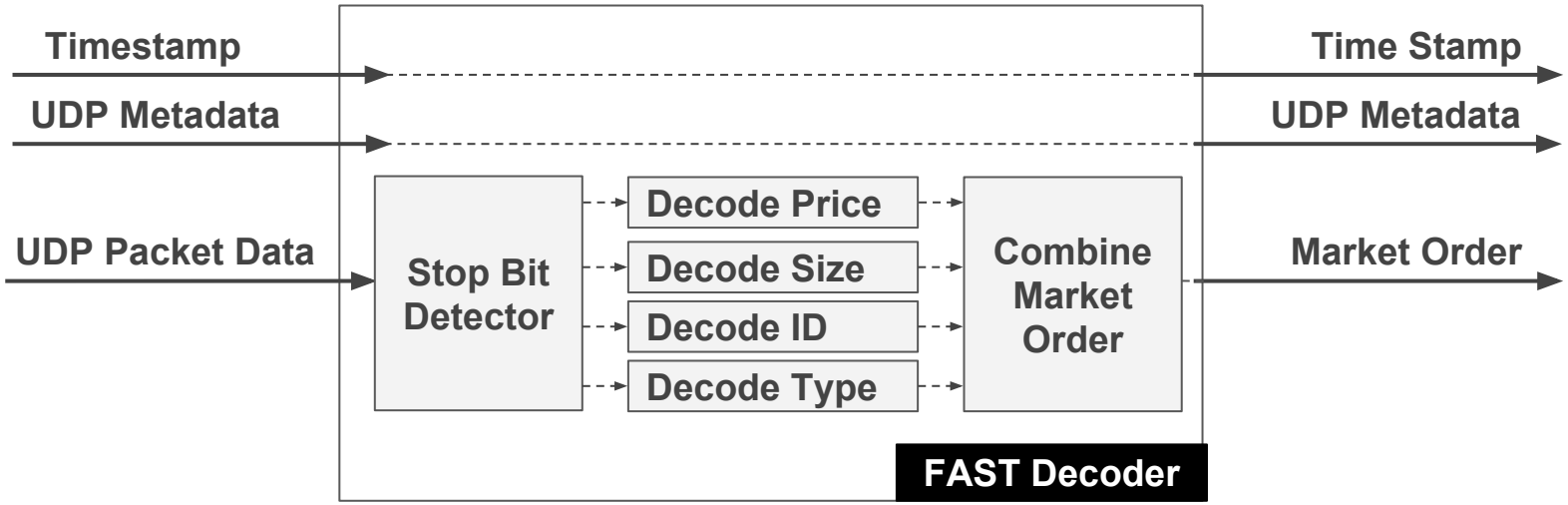
- Financial Information Exchange protocol (FIX) adapted for streaming
  - Variable length message encoded in bytes
  - Decoding depends on a template
  - Stop bits determine end of field
- Used to:
  - Decode UDP packets containing financial data coming in from the network Layer
  - Encode packets coming from the Application Layer



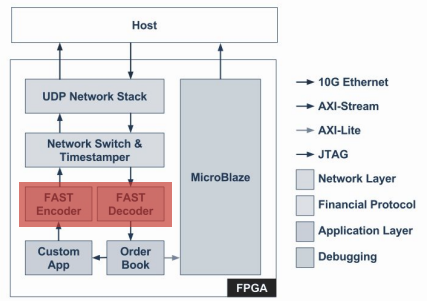
FAST Decoder



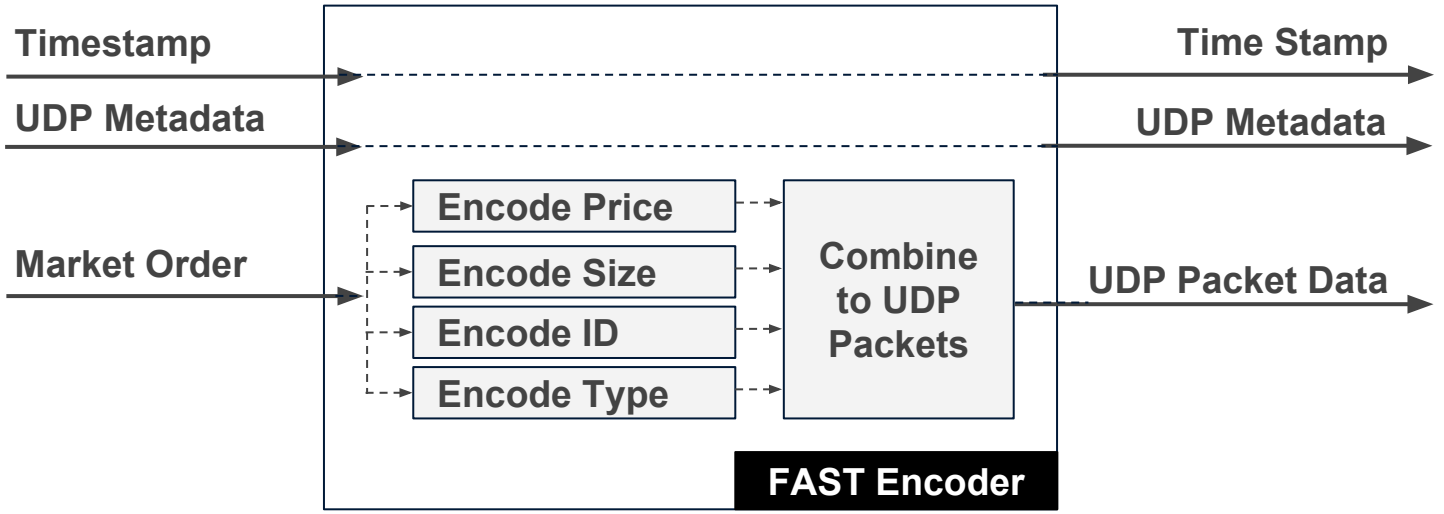
- An input UDP packet comes in 64 bit chunks
- Packets are buffered into bytes before processing
- Dataflow:
  1) **Stop bit detector** inspects the MSB of each byte
  2) Multiple decoders are run in parallel
  3) Decoded message is sent to the Order Book
- Latency of 9 cycles @ 5 ns estimated by Vivado HLS



FAST Encoder



- An input market order received from the Custom App module
- Dataflow:
  1) Multiple encoders are run in parallel
  2) Encoded message is sent to the Network layer in 64-bit chunks
- Latency of 0 cycles @ 5 ns estimated by Vivado HLS

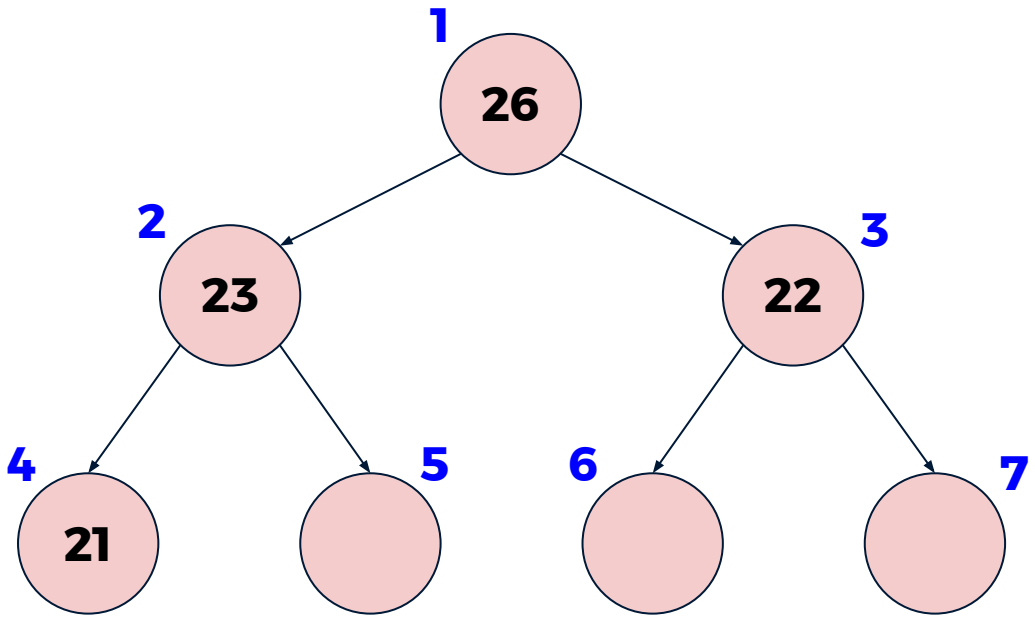


Order Book Keeping - Recap

- It is an essential pre-processing stage for almost all financial algorithms.
- Keeps track of **Bids** (offers to buy) and **Asks** (offers to sell) in order of their price.
- This translates into building a hardware **Priority Queue** with some special features:
  - Low-latency insert and delete.
  - Modify top entry.
  - Remove a specific entry.
  - Remove multiple entries.

Bid Order Book			
Order ID	**Time**	**Size**	**Bid Price**
101	12:02:36	4	27.4
104	12:03:18	2	27.4
102	12:03:07	6	27.2
Ask Order Book			
Order ID	**Time**	**Size**	**Price**
105	12:03:25	4	27.6
103	12:03:25	2	27.7

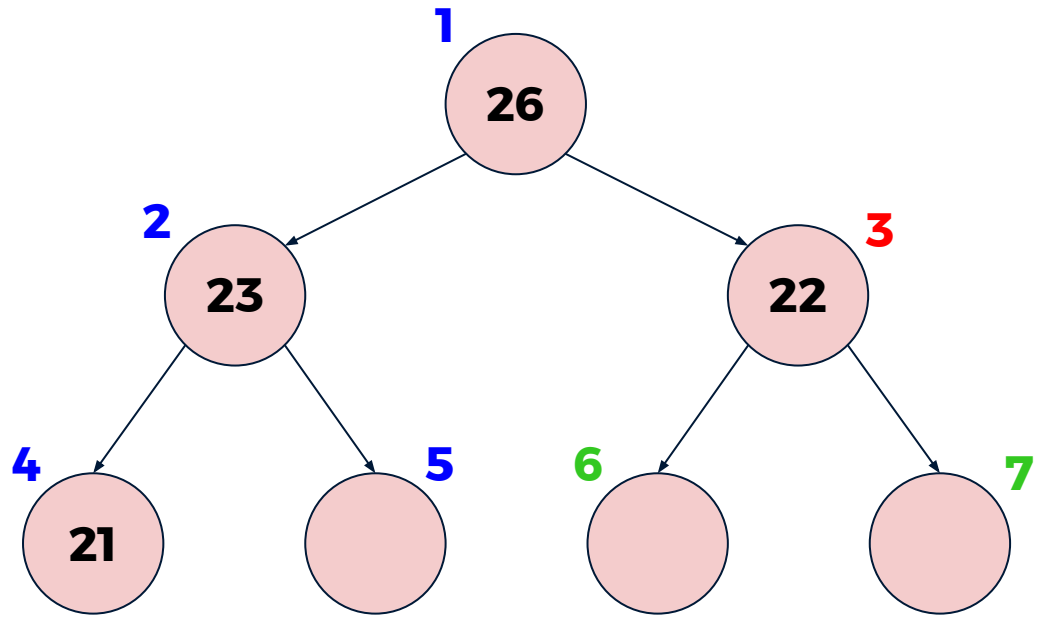
Order Book Keeping - Implementation



	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Implementation

- For any node at index *j*:
  - Left child has index *2j*.
  - Right child has index *2j+1*.



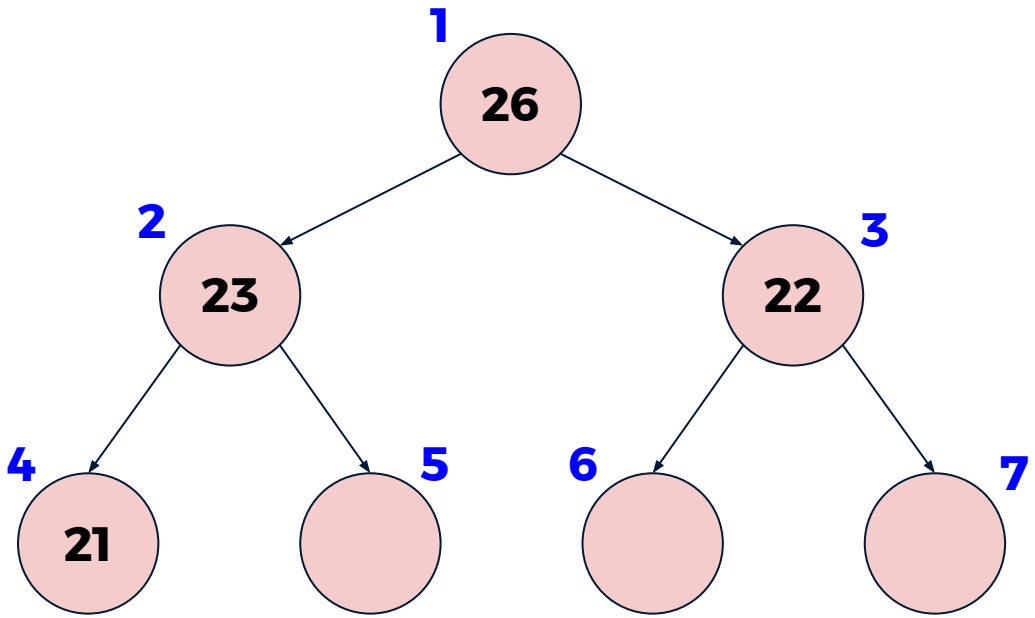
	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Implementation

- **For any node at index *j*:**
  - Left child has index *2j*
  - Right child has index *2j+1*.

- **For any node at index *j* and level *i*:**

  Path to this node is the binary representation of the least significant *i-bits* of $j-2^i$.



	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Implementation

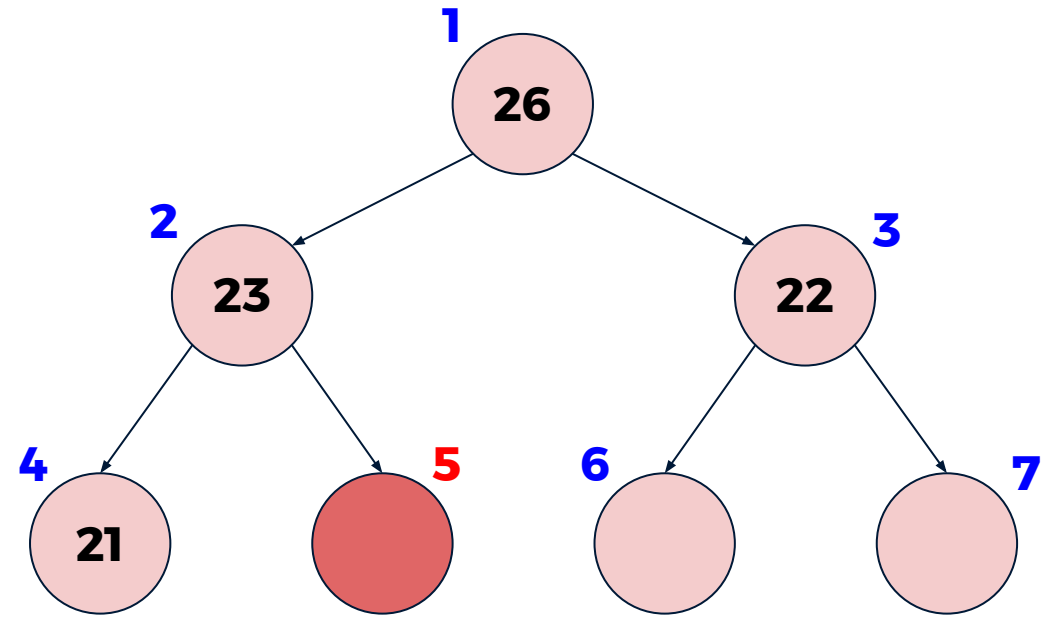
- **For any node at index *j*:**
  - Left child has index ***2j***
  - Right child has index ***2j+1***.

- **For any node at index *j* and level *i*:**

  Path to this node is the binary representation of the least significant ***i-bits*** of $j-2^i$.

  For example: **Node 5 in level 2**

  $5 - 2^2 = 1 \rightarrow$ in binary 0001



	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Implementation

- **For any node at index *j*:**
  - Left child has index ***2j***
  - Right child has index ***2j+1***.

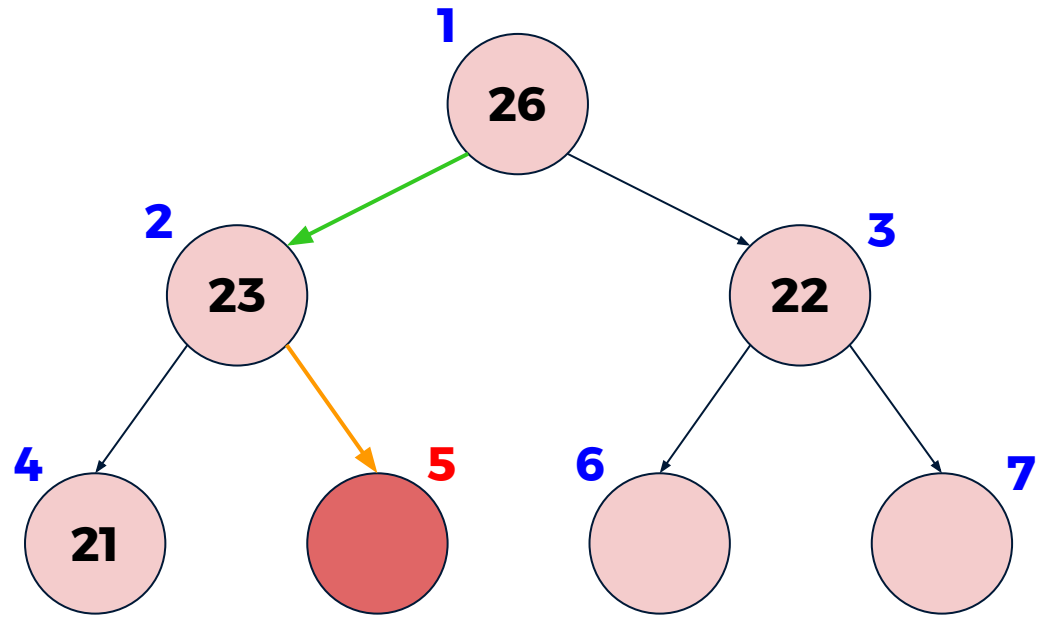
- **For any node at index *j* and level *i*:**

  Path to this node is the binary representation of the least significant ***i-bits*** of $j-2^i$.

  For example: **Node 5 in level 2**

  $5 - 2^2 = 1 \rightarrow$ in binary 0001

  left right

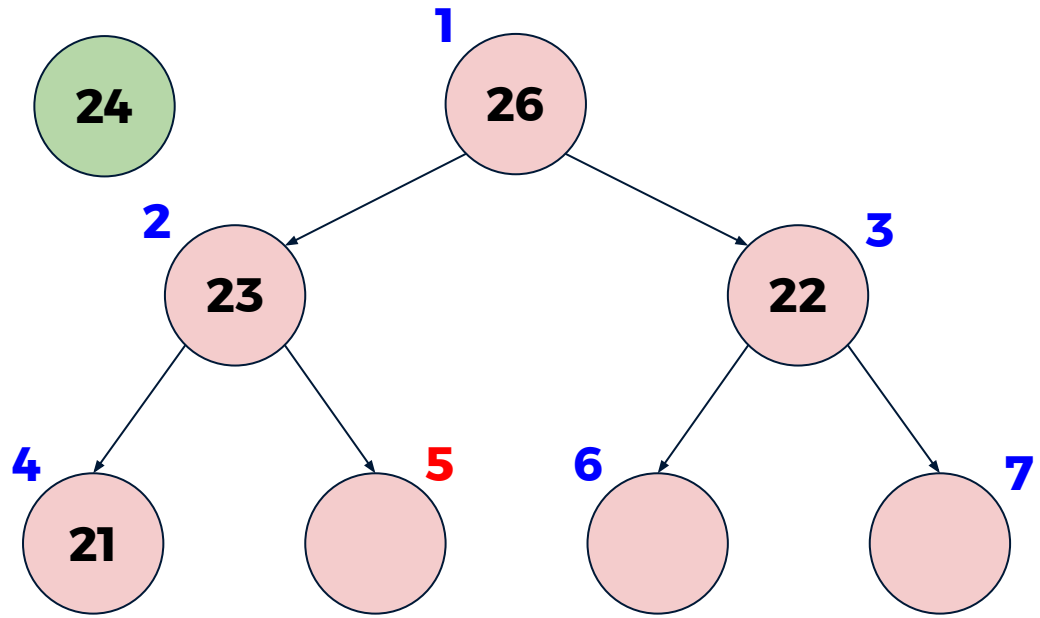


	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Get index of next empty node → *5*

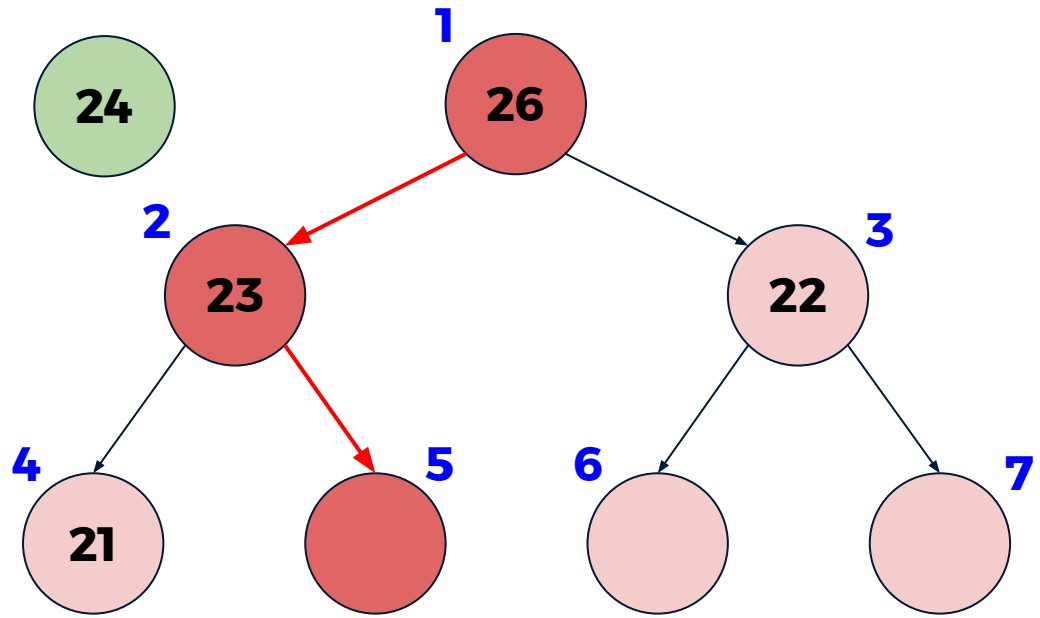


	1	2	3	4	5	6	7
heap	26	23	25	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Get index of next empty node → 5
2. Get the path → 5-4= 01 **(left - right)**

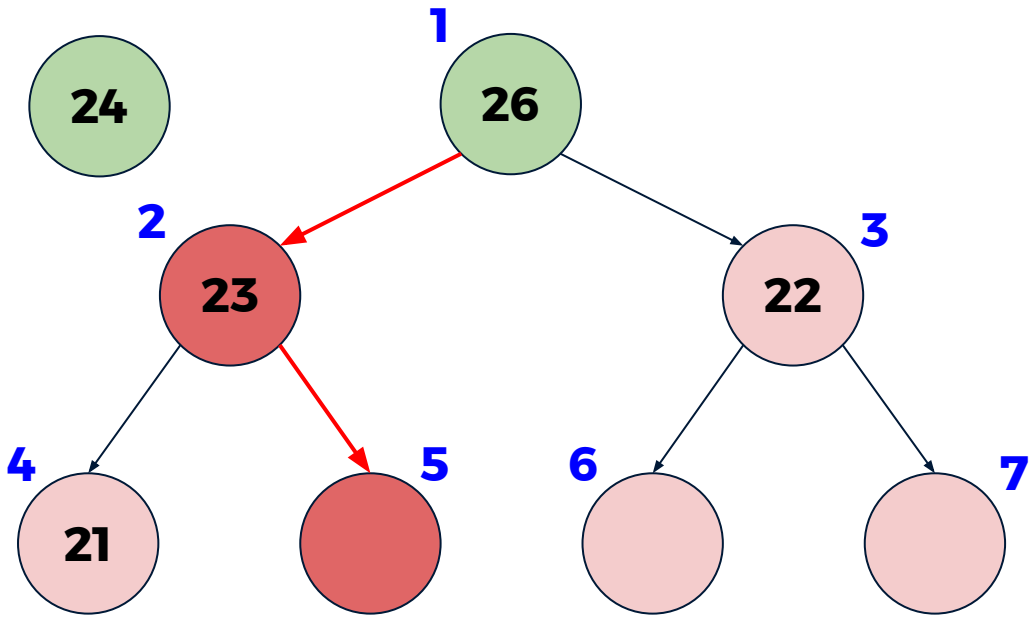


	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Get index of next empty node → 5
2. Get the path → 5-4= 01 (left - right)
3. Compare to index 1 → 26 > 24 (No swap)

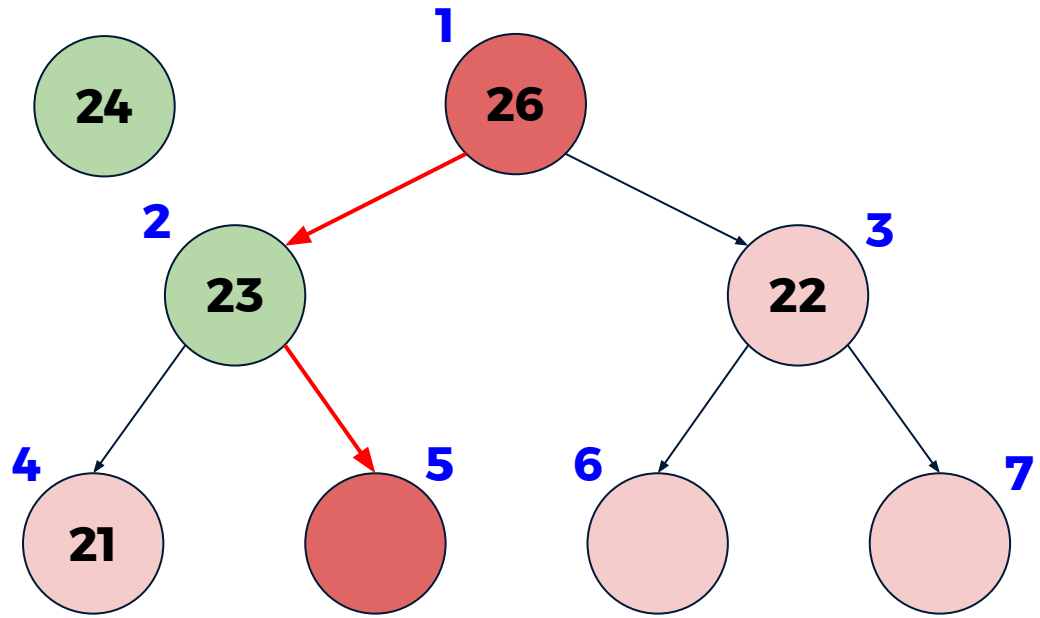


	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Get index of next empty node → 5
2. Get the path → 5-4= 01 (left - right)
3. Compare to index 1 → 26 > 24 (No swap)
4. Move to left node → index 2x1 = **2**

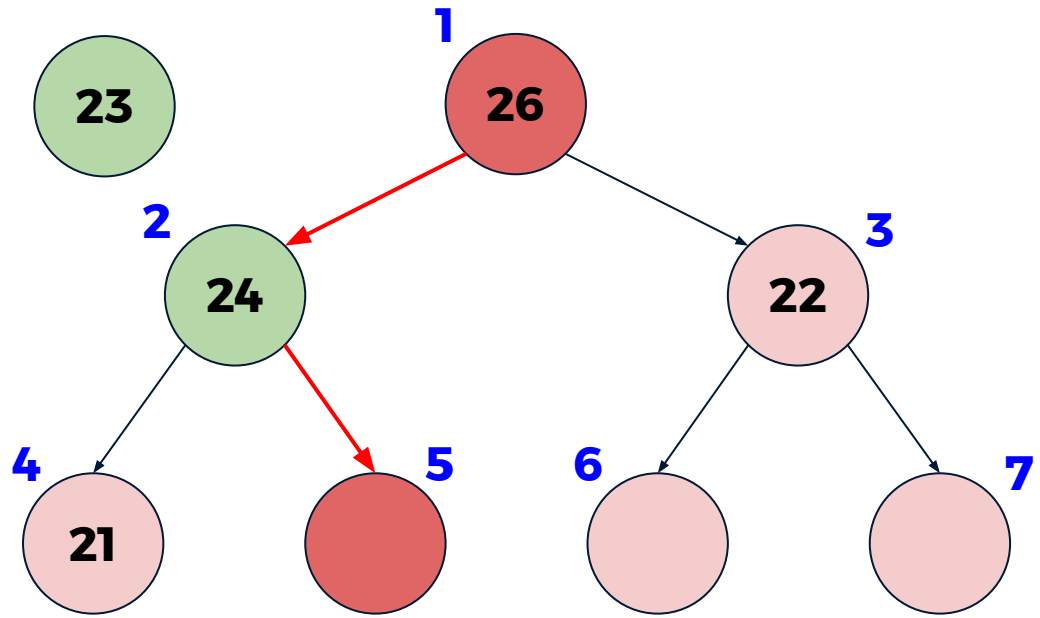


	1	2	3	4	5	6	7
heap	26	23	22	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Get index of next empty node → 5
2. Get the path → 5-4= 01 (left - right)
3. Compare to index 1 → 26 > 24 (No swap)
4. Move to left node → index 2x1 = 2
5. Compare to index 2 → 23 < 24 **(Swap)**

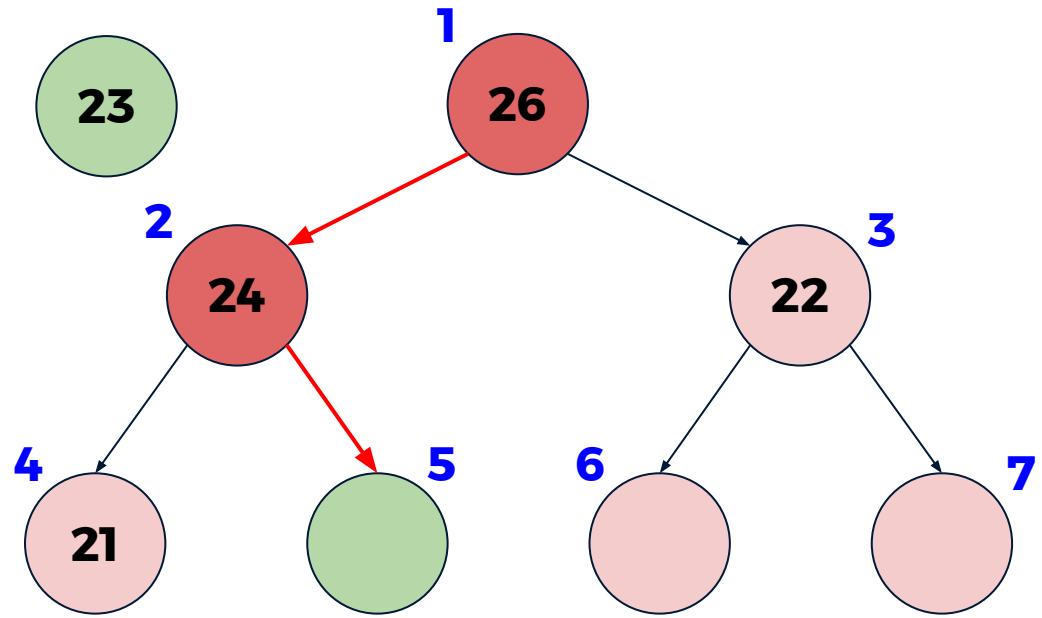


	1	2	3	4	5	6	7
heap	26	24	22	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Get index of next empty node → 5
2. Get the path → 5-4= 01 (left - right)
3. Compare to index 1 → 26 > 24 (No swap)
4. Move to left node → index 2x1 = 2
5. Compare to index 2 → 23 < 24 (Swap)
6. Move to right node → index (2x2)+1 = *5*

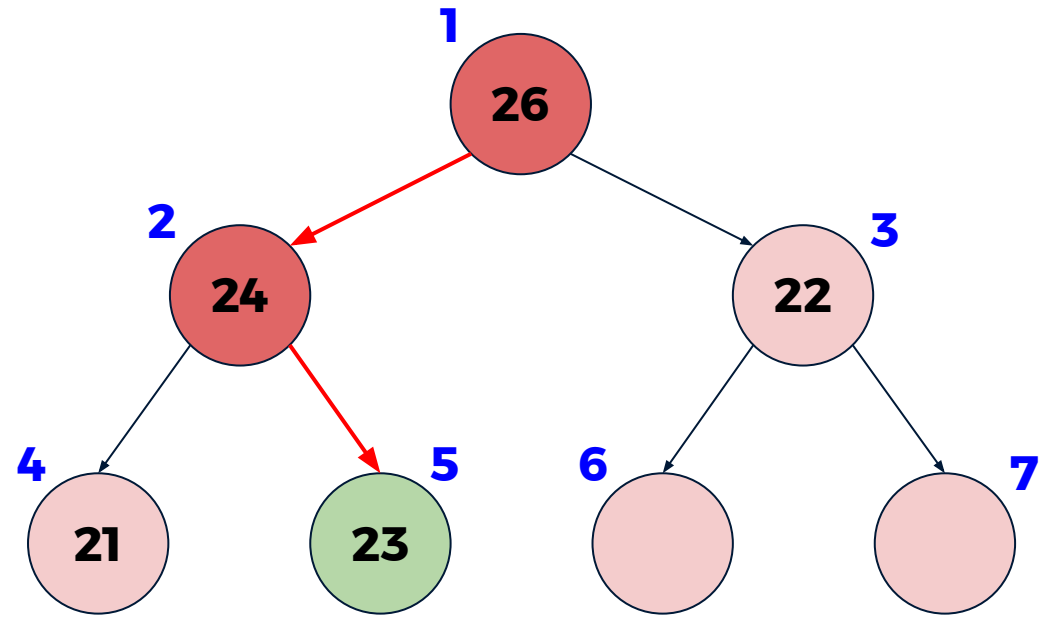


	1	2	3	4	5	6	7
heap	26	24	22	21			
holes							

Order Book Keeping - Insertion

Using those two simple yet very helpful observations we can build very efficient PQ:

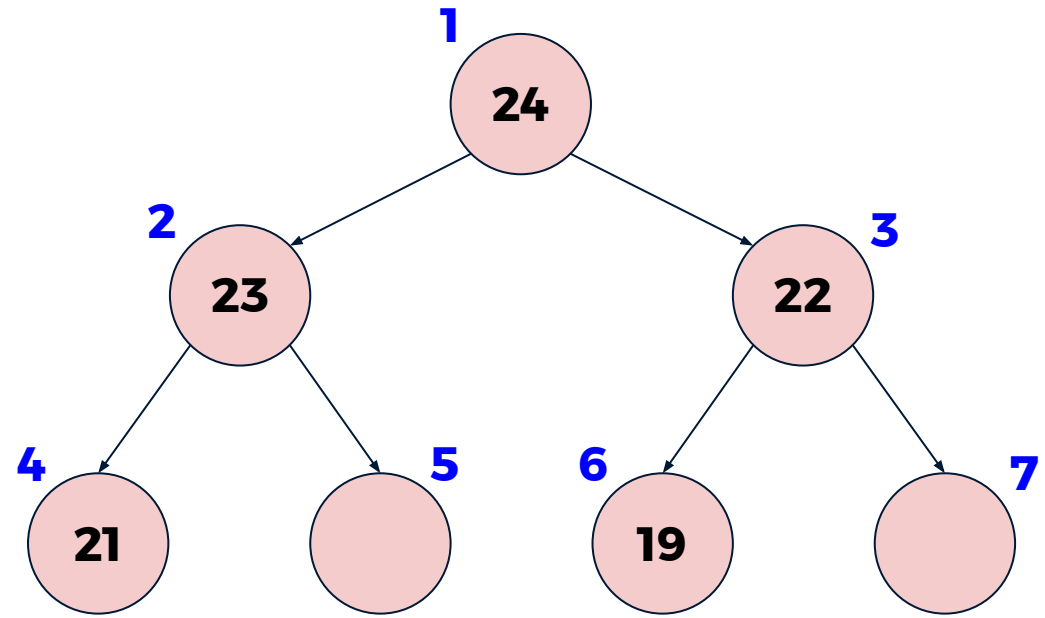
1. Get index of next empty node → 5
2. Get the path → 5-4= 01 (left - right)
3. Compare to index 1 → 26 > 24 (No swap)
4. Move to left node → index 2x1 = 2
5. Compare to index 2 → 23 < 24 (Swap)
6. Move to right node → index (2x2)+1 = 5
7. **Destination reached**



	1	2	3	4	5	6	7
heap	26	24	22	21	23		
holes							

Order Book Keeping - Specs

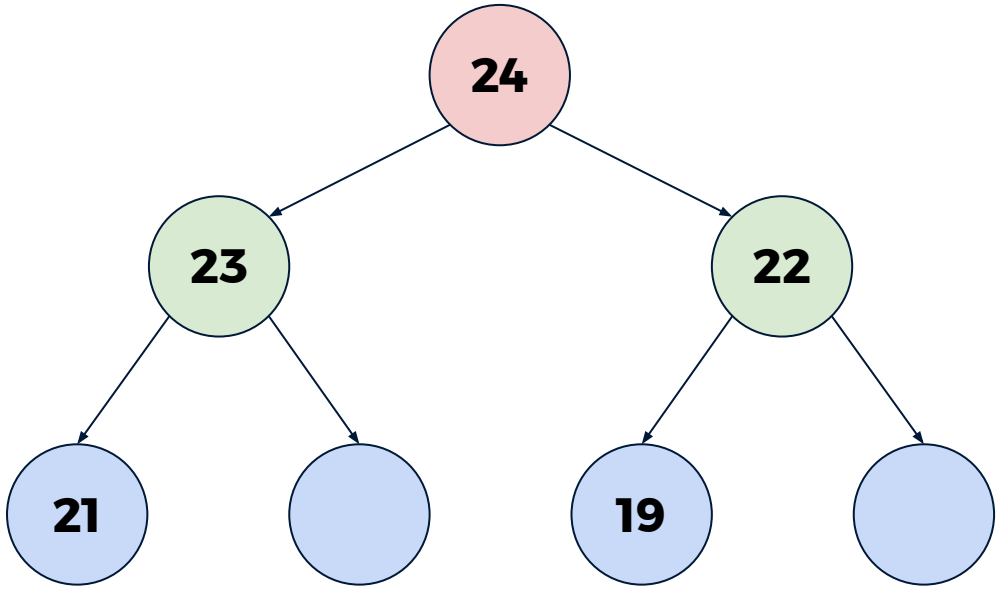
- Capacity of **4096 bids** and **4096 asks**.
- **Streaming output** after first comparison.
- **II=1** for all insertion and deletion loops.
- Ability to **modify top order** size if only a portion of the order is sold.
- Ability to **delete any arbitrary node** is supported by adding another heap for the “to-be-removed” orders.

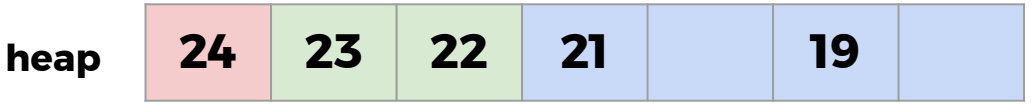


	1	2	3	4	5	6	7
heap	24	23	22	21		19	
holes	5						

HLS Shortcoming

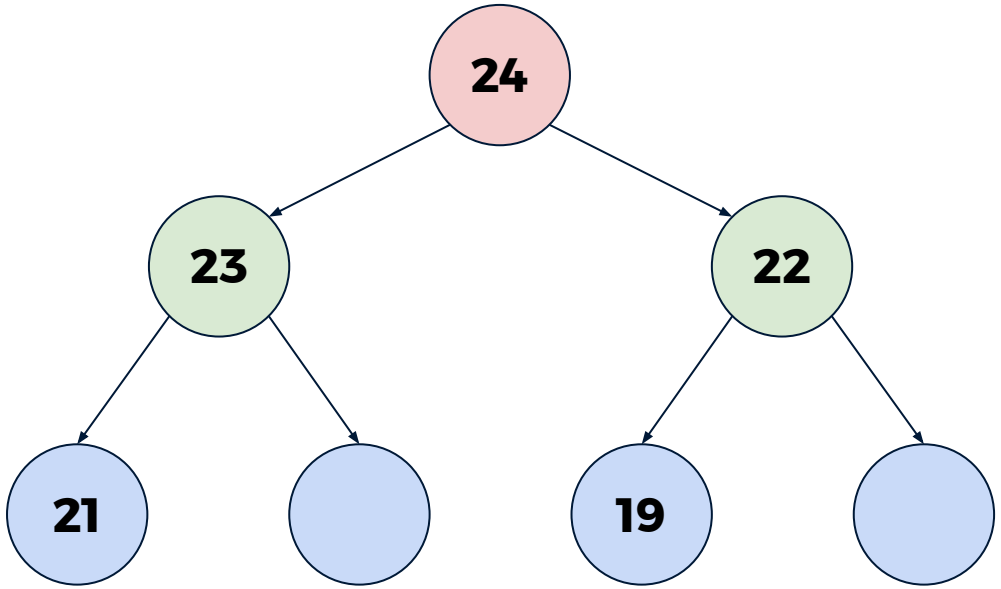
- When optimizing the order book for II=1, we needed to partition the heap array such that each level of the tree is in a separate partition.
- Array partitioning with varying partition sizes is **not** doable in HLS.





HLS Shortcoming

- When optimizing the order book for II=1, we needed to partition the heap array such that each level of the tree is in a separate partition.
- Array partitioning with varying partition sizes is **not** doable in HLS.
- We had to implement the heap array as a 2D array with dimensions equal to the number of levels times the size of the tree base.
- Wastes a lot of memory resources that can be avoided by more complex coding.



heap

24			
23	22		
21		19	

Testing and Verification

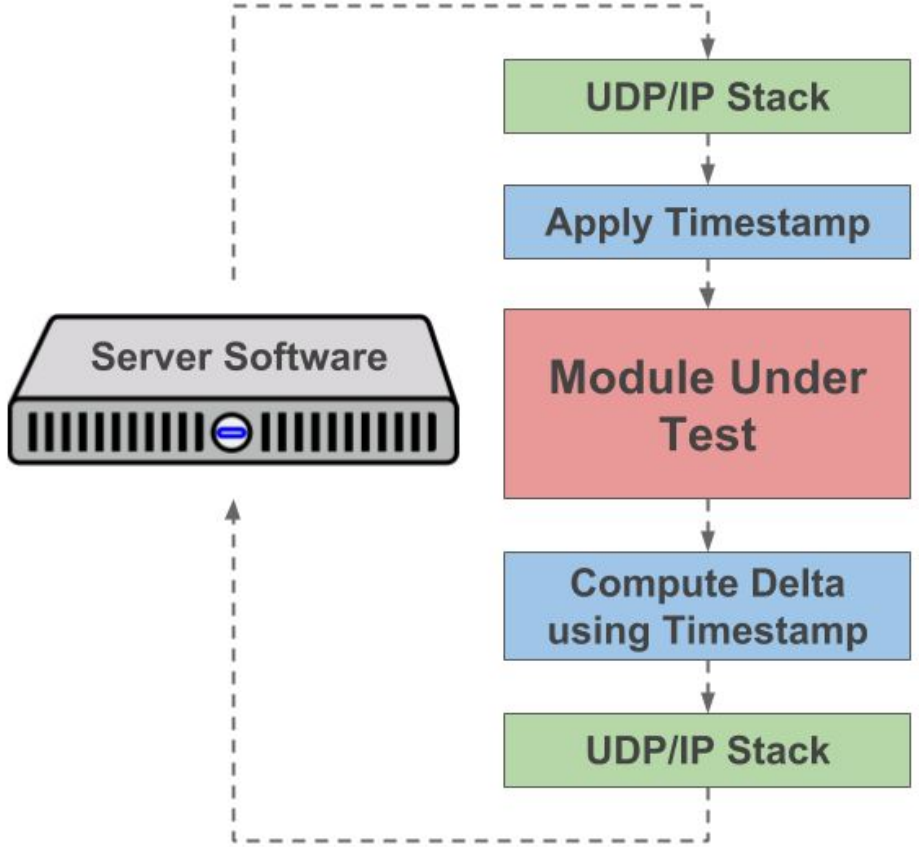
- We used an **incremental** approach to integrate the system.

  Iter 1. Network stack & timestamping

  Iter 2. FAST Encoder/Decoder w/ Network

  Iter 3. Entire system inc. Order Book

- This allowed more robust verification *in hardware* of components such as the FAST encoder.
- The incremental approach also allowed creating a latency breakdown of the individual blocks once synthesized.

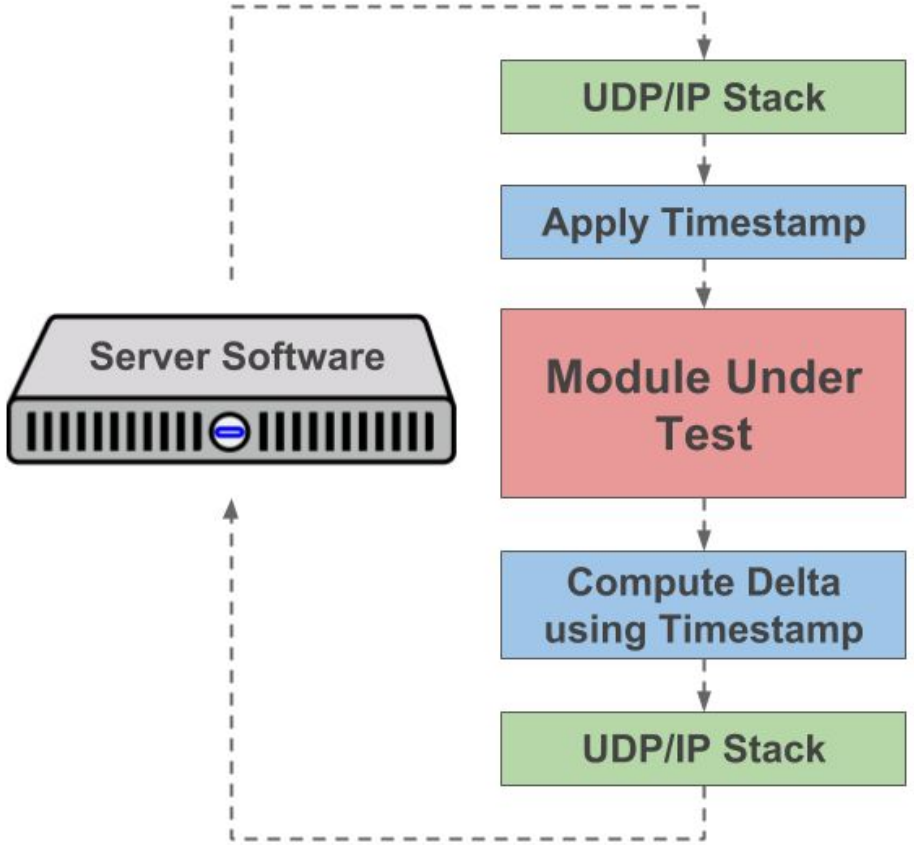


Testing and Verification

Two different monitoring mechanisms were used in the hardware:

1 | MicroBlaze Monitoring

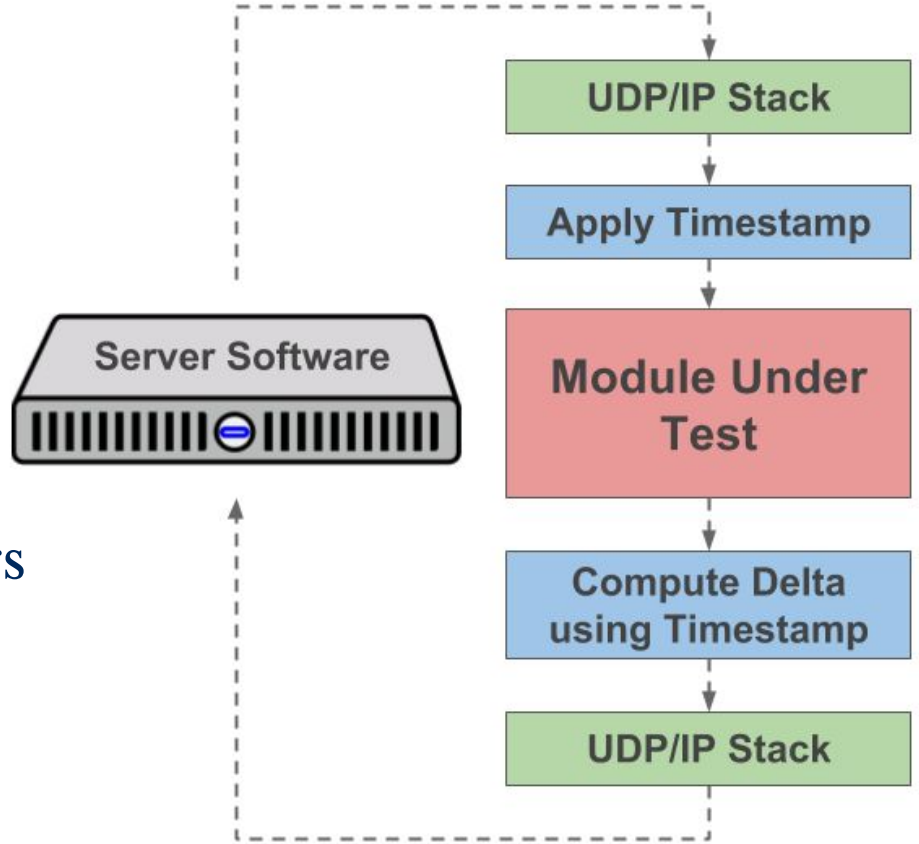
- Order Book exposes an **AXI-Lite** interface with top bid/ask, which is identical to the last streaming output.
- Top Bid/Ask were reported periodically (~1s) through JTAG debug interface.
- Allows observation of the Order Book state, which is normally "hidden" behind the trading algorithm.



Testing and Verification

2 | Network-Based Testing

- System is tested as a black-box; can only see the outgoing packets it transmits.
- Latency data is appended unto the outgoing order packets.
- **Server-side software:**
  - Generates test data, encodes it as FAST orders and sends them over Ethernet interface
  - Receives and decodes orders from the hardware
  - Computes and displays an equivalent Order Book state given the test data.



FPGA Platform

- Xilinx Kintex Ultrascale FPGA on the Alpha Data 8K5
  - 10 Gigabit Ethernet
  - Can reuse the same UDP/IP subsystem used for the Shell project
  - Readily available and already configured through the Savi server
- Vivado HLS for high level synthesis cores
- Vivado IP integrator for connecting together HLS cores

Timing Results

Module Under Test	Frequency	Round Trip Latency	**Round Trip Runtime**
Full System	156 MHz	42 cycles	**269.2 ns**
Full System + Ethernet	156 MHz	85 cycles	**525 ns**

- Final System ran at **156 MHz** limited by Ethernet port
- Measured Latency of system is **42 cycles (269.2 ns)**
- Additional **85 ns** for the network transmit and **170 ns** for the network receive
- Total runtime **525 ns** on average using 10,000 random test packets

Timing breakdown

Module	Latency	**Total Latency**	Runtime
Network Transmit	13 cycles	-	85 ns
Network Receive	27 cycles	-	170 ns
Network Switch	12 cycles	**12 cycles**	76.9 ns
FAST Encode/ Decode	18 cycles	**30 cycles**	115.4 ns
Order Book	12 cycles	**42 cycles**	76.9 ns

- Reported Network switch latency shows us that the streaming interface adds a significant amount of latency.
- Adding the Order book only added **12 cycles** of Latency: a minimal amount.

Area Results

Resource	Utilization	Available	**Utilization %**
LUT	49,638	663,360	**7.48**
LUTRAM	2,148	29,3760	**0.73**
FF	32,718	1,326,720	**2.47**
BRAM	474	2,160	**21.92**
DSP	3	5,520	**0.05**

- Total utilization is minimal compared to the FPGA size
- Largest utilization is due to BRAM
  - A trade off for having the order book run at little latency

Other HFT Work on the FPGA

System	FPGA Platform	Network Transmit	Network Receive	Decode/Encode	Rest of system	**Total Runtime**
Our solution	Kintex Ultrascale	85 ns	170 ns	192.3 ns	76.9 ns	**525 ns**
Leber et. al [1]	Virtex-4 FX100	-	-	-	-	**2.6 us**
Lockwood et. al [2]	NetFPGA-10G (Virtex-5)	400ns	400 ns	200 ns	Not included	**1 us**

- Leber et. al created an in parallel multiple FAST stream solution that sends decoded data to a software processor at a total latency of **2.6 us**
- Lockwood el. al used the FIX financial exchange protocol for encoding and decoding and report **200 ns**
  - We measured our system with only FAST encoder / decoder and achieved average runtimes of **192.3 ns**

HLS *ROCKS*

- Streaming interfaces used HLS library calls
  - Easy for integration, but gave little control over the latency of communication between HLS IP cores.

- Regarding the FAST Protocol each exchange has their own message template
  - In C, encoding/ decoding functions are simple to order in a way that matches the message template

- Experts in financial trading algorithms can easily tinker/build onto our design with basic understanding of hardware and no need for RTL expertise.

HLS *ROCKS*

- **Design Space Exploration and Optimization**
  - HLS directives allow easier and faster tuning for performance.
  - In HDL, all tuning is essentially manual code changes.
- **Testing**
  - Getting RTL to work correctly requires lots of low level debugging.
  - HLS design can be tested in C to verify the basic functionality before worrying about hardware concerns.
- **Empirical Observation**
  - Before this course was HLS, many projects finished after the summer
  - Our cohort finished all projects by early June; in our case Mid-May.
  - We believe this indicates that HLS is ~**2x** more productive

Database Organization

- **Git/ bitbucket**: cloud source code control
- **Directory structure**
  - Each member worked on a project in the hls folder
    - Network switch
    - FAST Protocol
    - Order book
  - Separate folder for IP Integration
    - Used for individual, partial and full integration
  - Scripts folder contains python files used to send information through the network to the FPGA

hft
base project folder

- **src**
  src code folder
  - **hls**
    vivado hls projects
  - **ip**
    built IP cores
- **build**
  vivado IP integrator projects
- **scripts**
  network scripts

References

[1] C. Leber, B. Geib and H. Litz, "High Frequency Trading Acceleration Using FPGAs," 2011 21st International Conference on Field Programmable Logic and Applications, Chania, 2011, pp. 317-322.

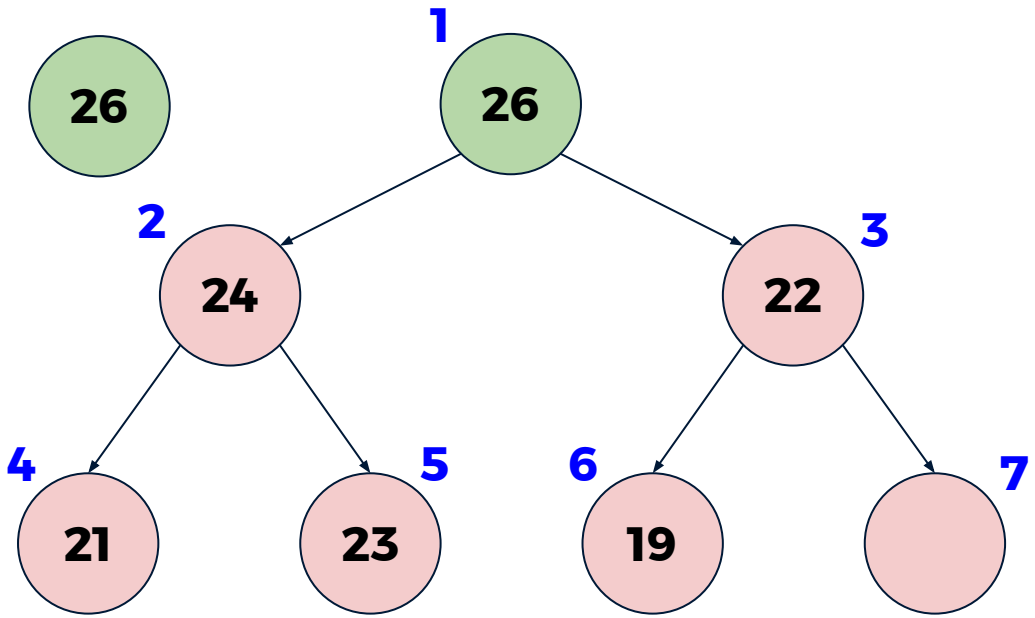
[2] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English and K. Vissers, "A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT)," 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, Santa Clara, CA, 2012, pp. 9-16.

Thank You!

Order Book Keeping - Deletion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. **Return the top node.**

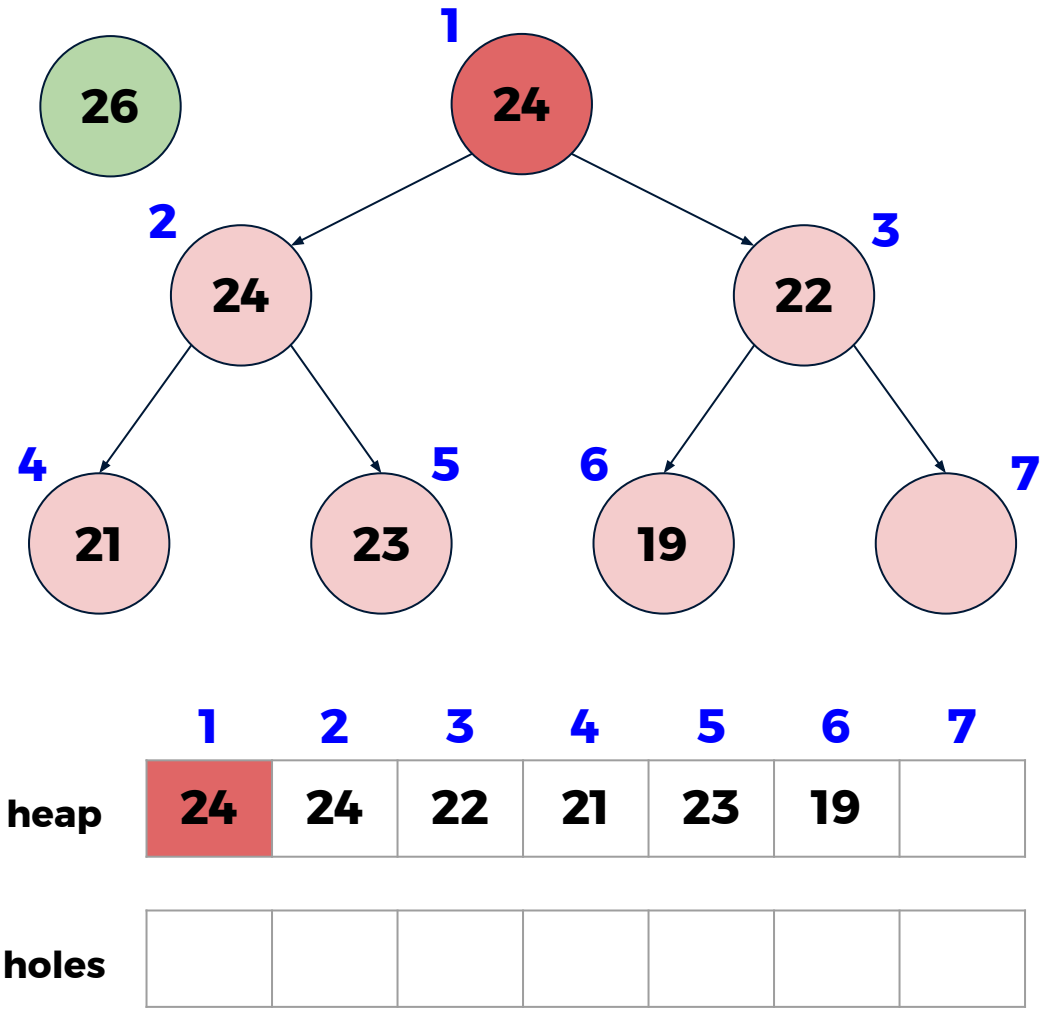


	1	2	3	4	5	6	7
heap	26	24	22	21	23	19	
holes							

Order Book Keeping - Deletion

Using those two simple yet very helpful observations we can build very efficient PQ:

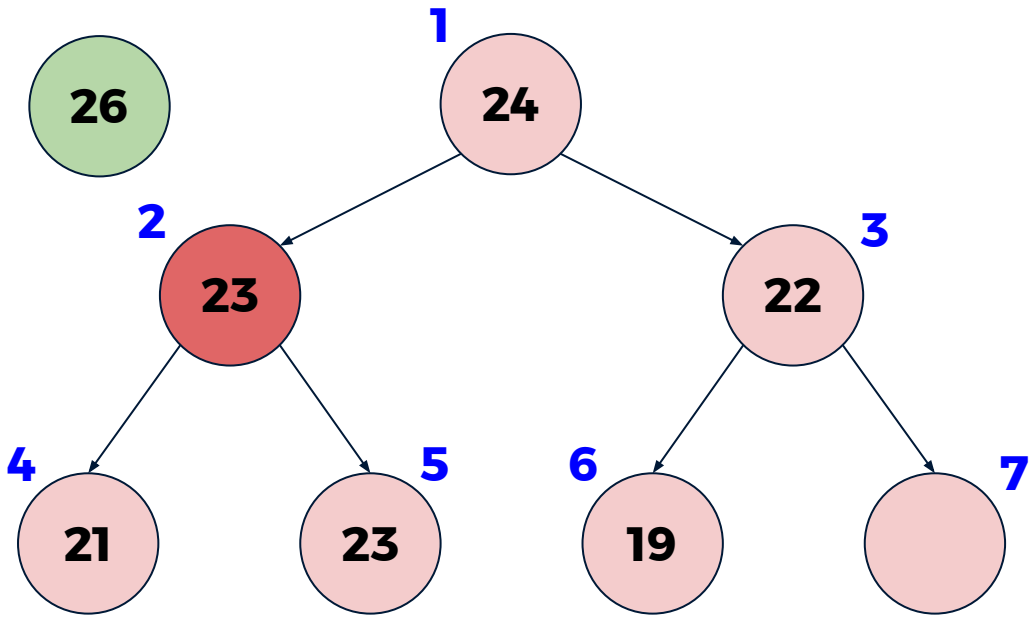
1. Return the top node.
2. **Pick its larger child to replace it.**



Order Book Keeping - Deletion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Return the top node.
2. Pick its larger child to replace it.
3. **Go to the picked child & repeat.**

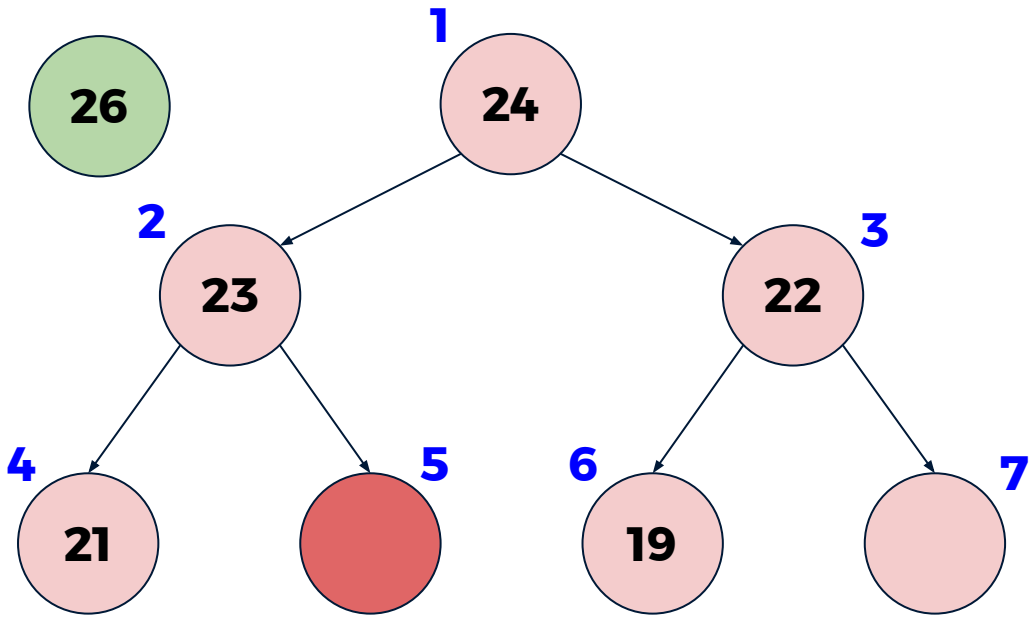


	1	2	3	4	5	6	7
heap	24	23	22	21	23	19	
holes							

Order Book Keeping - Deletion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Return the top node.
2. Pick its larger child to replace it.
3. Go to the picked child & repeat.
4. **Reaching a leaf node → Add to holes.**



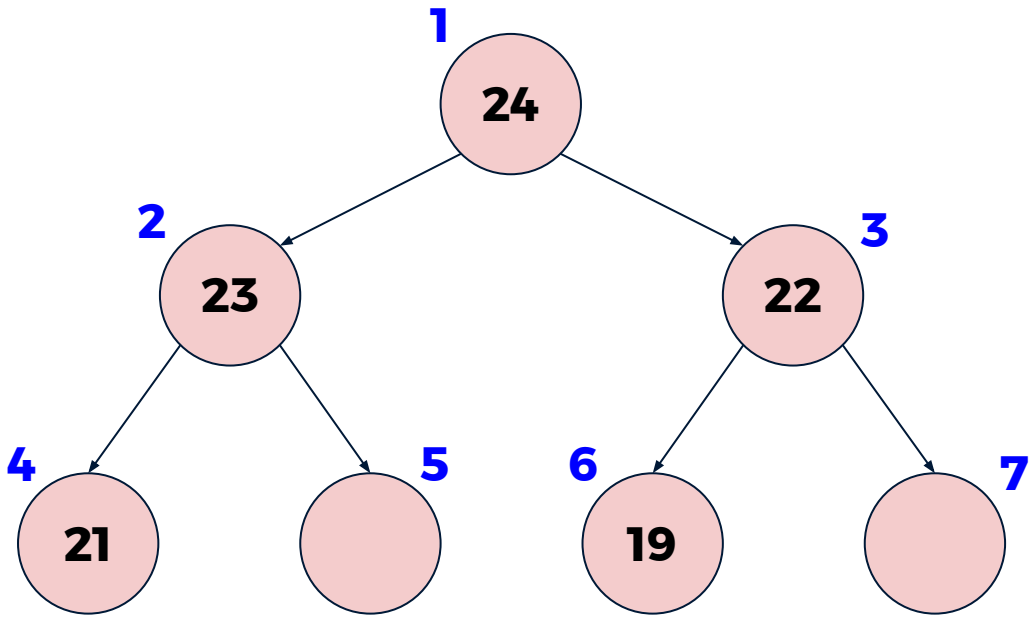
	1	2	3	4	5	6	7
heap	24	23	22	21		19	
holes	5						

Order Book Keeping - Deletion

Using those two simple yet very helpful observations we can build very efficient PQ:

1. Return the top node.
2. Pick its larger child to replace it.
3. Go to the picked child & repeat.
4. Reaching a leaf node → Add to holes.

If an insertion occurs, fill the holes first before the next empty node to maintain the heap structure.



	1	2	3	4	5	6	7
heap	24	23	22	21		19	
holes	5						