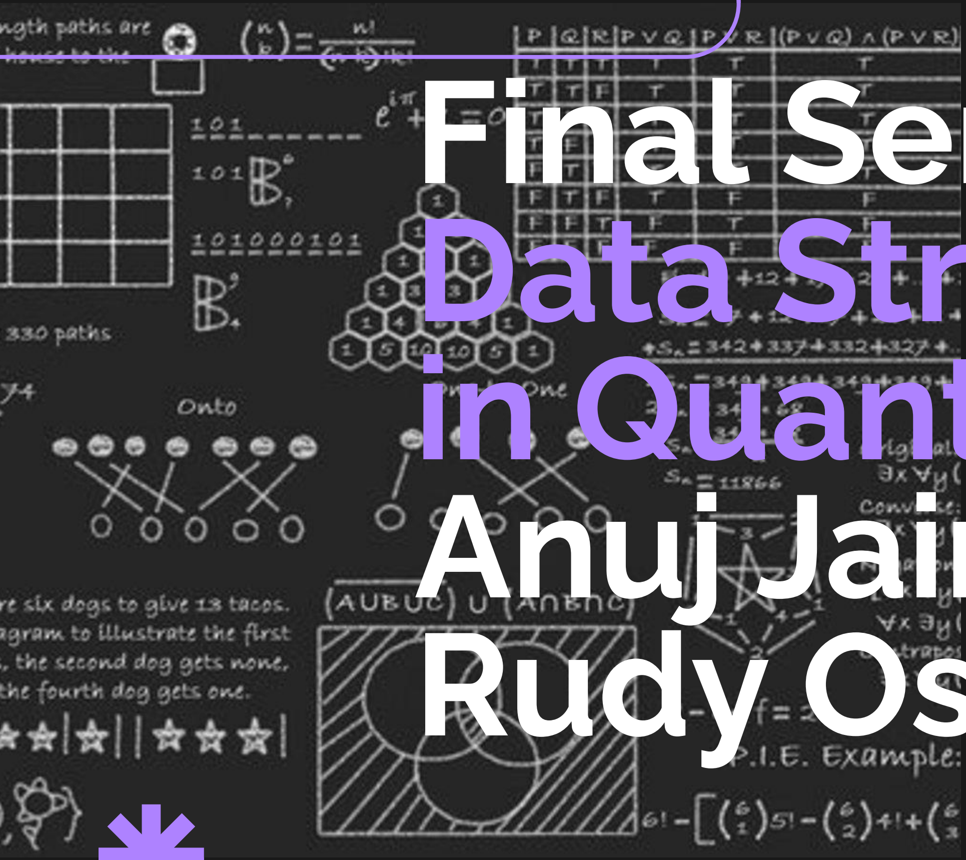
@TQT_UCSD



Final Seminar Data Structures in Quant Trading

Anuj Jain && Rudy Osuna

Optimizing Algorithms for Speed and Efficiency

Announcements: Farewell to Anuj!

As we bid farewell to our TQT President, Anuj Jain, let's celebrate his incredible leadership and dedication. Anuj has been instrumental in driving our success, and we wish him all the best in his next chapter!

Open Positions – Join the Board!

We're excited to welcome new talent to our team! Applications are now open for the following positions:

- **Marketing-Design Chair** (Discord, LinkedIn)
- **Math Content Developer**
- **ML Engineer**
- Additional **Technical Roles**

What we're looking for:

- **Low Attitude, High Caliber**
- **Consistent Time Commitment**
- **Deep Interest** demonstrated through past involvement

Check in Code:

Overview of Data Structures in Quantitative Trading

Efficient data structures are essential for optimizing financial algorithms, enabling faster data processing and decision-making, we will cover the following:

CSE 12:
- Deque (Sliding Window)
- Hash Map
- Priority Queue/Heap
- (Extra) Stack

CSE 100:
- Graphs - Data Analysis & Visualization
- (Extra) Binary Search Trees and AVL Trees

CSE 140 && CSE 30:
Advanced Topic:
- Bit Manipulation, Bitwise Operations, Masking
- Matrices (Sparse covariance matrices)

Disclaimer: this is not financial advice

Agenda

For each data structure, we'll follow this structured approach:

01
How It Works:

Refresh of the data structure and its properties.

02
Runtime Analysis:

Detailed runtimes for common operations like insertion, deletion, and access.

03
Quant-Trading Application:

Explanation of how this data structure is used in quant trading.

04
Code algo. implementation

A concise Python example demonstrating its use in trading algorithms.

Disclaimer: this is not financial advice

Double Ended Queue (Deque)

Properties:

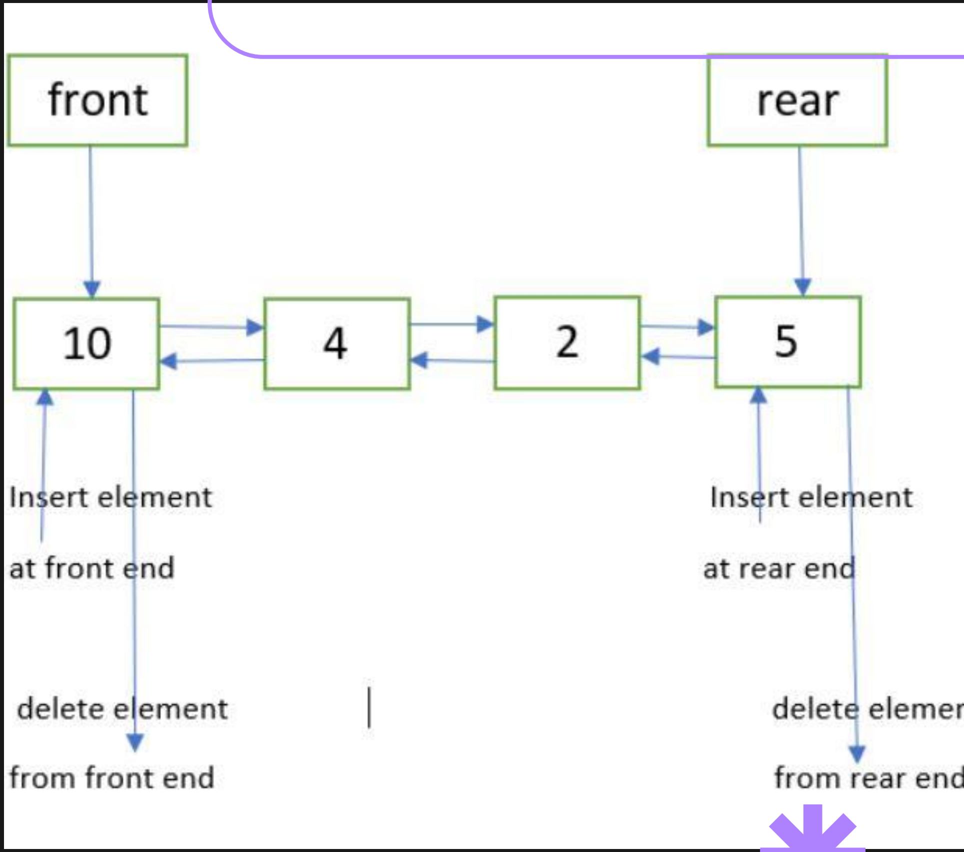
- Allows appending and popping elements from both the front and the back

Operations and Runtimes:

- Append/Pop from either end: O(1)
- Access by index: O(n)

Space Complexity:

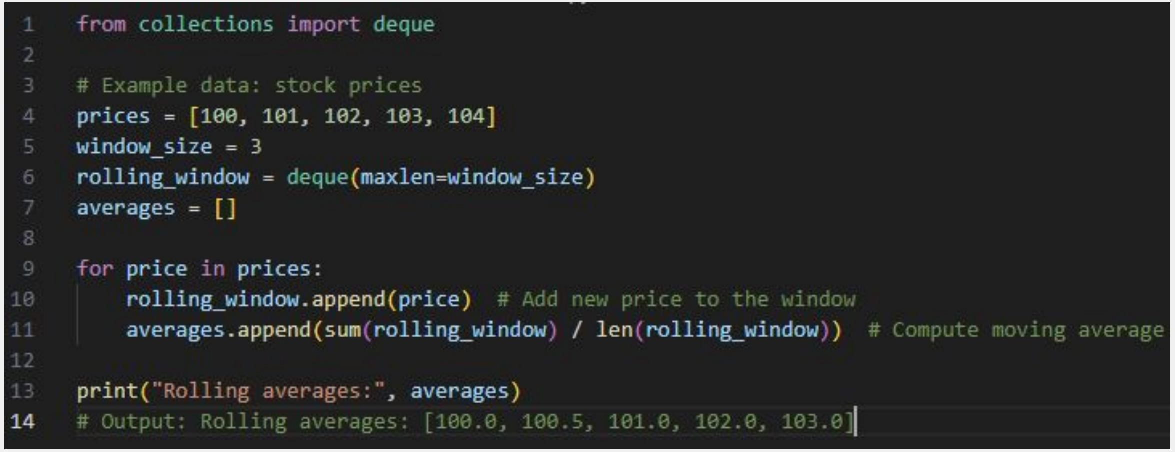
- O(n): Proportional to the number of elements stored.


front
rear
10
4
2
5
Insert element at front end
delete element from front end
Insert element at rear end
delete elemen
from rear end

Use Case:

- <u>**Custom Moving Average Indicator with a Rolling Window:**</u>
  - **A rolling window (or sliding window**) is a subset of data points (e.g., prices) that moves forward as new data arrives, removing the oldest data point to maintain a constant size.
- **In quantitative trading,** rolling windows are commonly used to compute indicators like moving averages, which smooth out price data over a fixed period to detect trends.
  - A **deque** is ideal for this application because it allows efficient addition of new prices and removal of the oldest prices at the ends of the window, keeping the computation time minimal.

Simple Python Implementation:

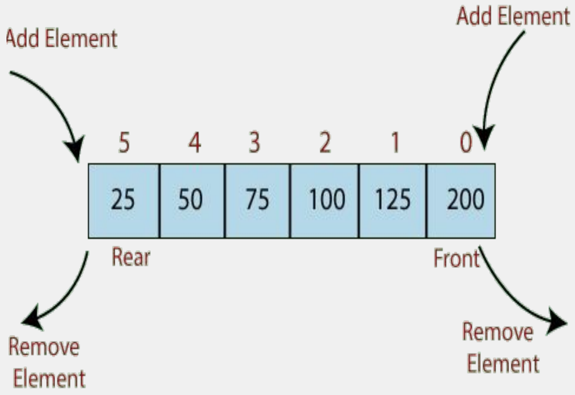


```python
from collections import deque

# Example data: stock prices
prices = [100, 101, 102, 103, 104]
window_size = 3
rolling_window = deque(maxlen=window_size)
averages = []

for price in prices:
    rolling_window.append(price)  # Add new price to the window
    averages.append(sum(rolling_window) / len(rolling_window))  # Compute moving average

print("Rolling averages:", averages)
# Output: Rolling averages: [100.0, 100.5, 101.0, 102.0, 103.0]
```



@TQT_UCSD

Hash Map

Properties:

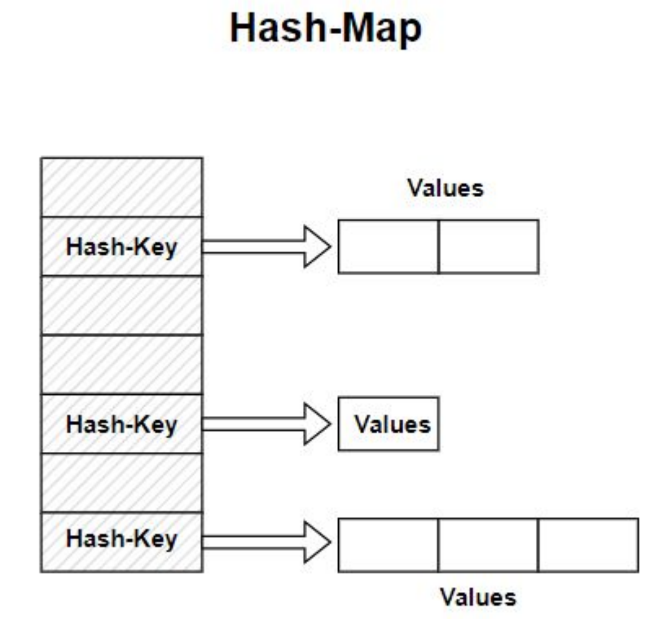
- Maps keys to values for fast lookup and storage

Operations and Runtimes:

- Insert/Lookup/Delete: O(1) (amortized)

Space Complexity:

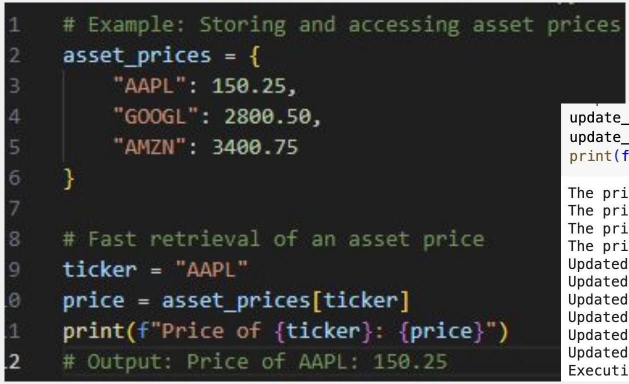
- O(n): Storage depends on the number of key-value pairs.
- Hash collisions can increase memory usage and degrade performance.



Use Case:

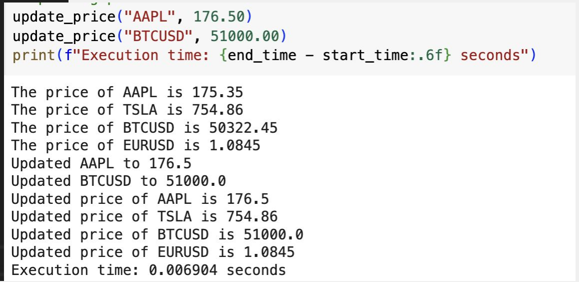
- <u>**Storing Asset Prices by Ticker for Rapid Access:**</u>
  - In quantitative trading, you often work with multiple assets, such as stocks, options, or forex pairs, and need to store their current or historical prices.
- A **hashmap** allows you to associate each asset (the key, e.g., the ticker symbol "AAPL") with its price (the value).
  - The hash map's **fast lookup speed** enables real-time decision-making, such as calculating **portfolio weights**, **comparing prices**, or **identifying arbitrage opportunities**.
- This efficient access becomes particularly valuable in **high-frequency (HFT)** or algorithmic trading, where milliseconds matter.

Simple Python Implementation:



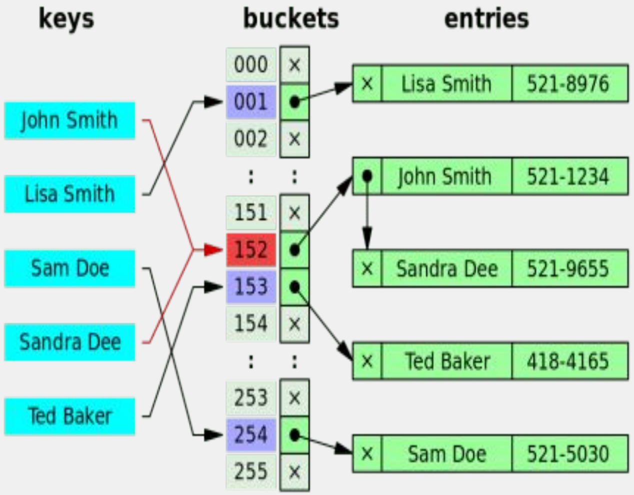
```python
# Example: Storing and accessing asset prices
asset_prices = {
    "AAPL": 150.25,
    "GOOGL": 2800.50,
    "AMZN": 3400.75
}

# Fast retrieval of an asset price
ticker = "AAPL"
price = asset_prices[ticker]
print(f"Price of {ticker}: {price}")
# Output: Price of AAPL: 150.25
```



```python
update_price("AAPL", 176.50)
update_price("BTCUSD", 51000.00)
print(f"Execution time: {end_time - start_time:.6f} seconds")
```

```
The price of AAPL is 175.35
The price of TSLA is 754.86
The price of BTCUSD is 50322.45
The price of EURUSD is 1.0845
Updated AAPL to 176.5
Updated BTCUSD to 51000.0
Updated price of AAPL is 176.5
Updated price of TSLA is 754.86
Updated price of BTCUSD is 51000.0
Updated price of EURUSD is 1.0845
Execution time: 0.006904 seconds
```



@TQT_UCSD

Priority Queue min/max Heap

Properties:

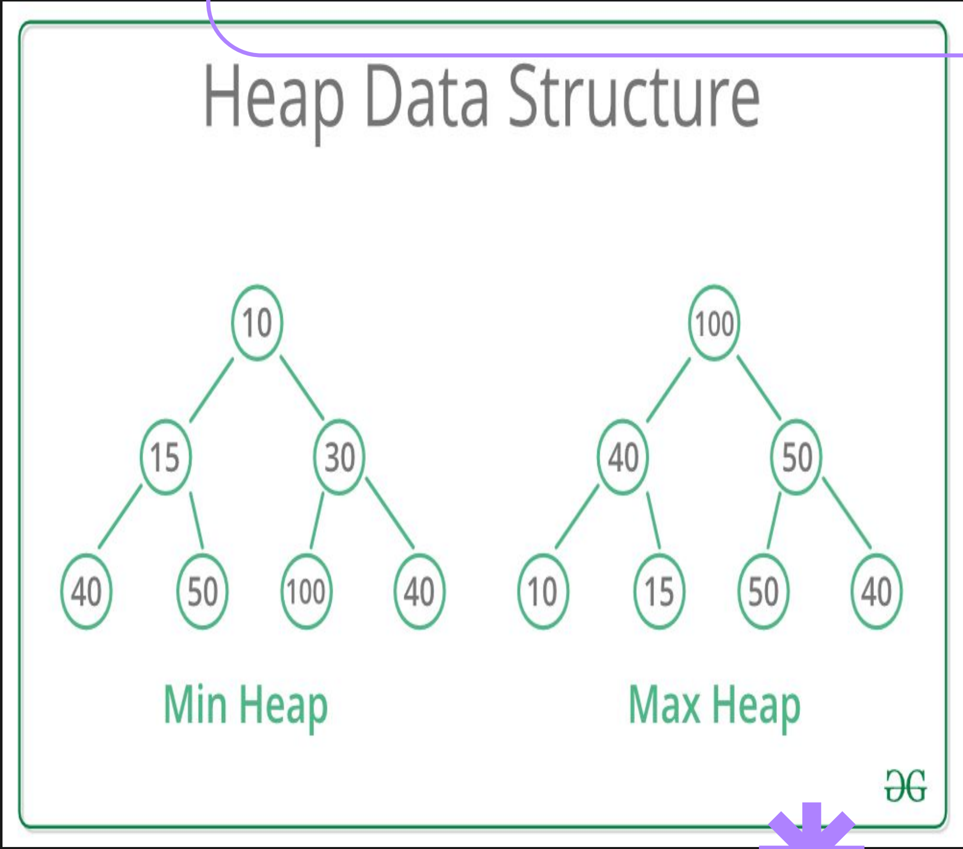
- Maintains the priority of elements in a heap structure.

Operations and Runtimes:

- Insert/Delete Min/Max: O(log n)
- Access Min/Max: O(1)

Space Complexity:

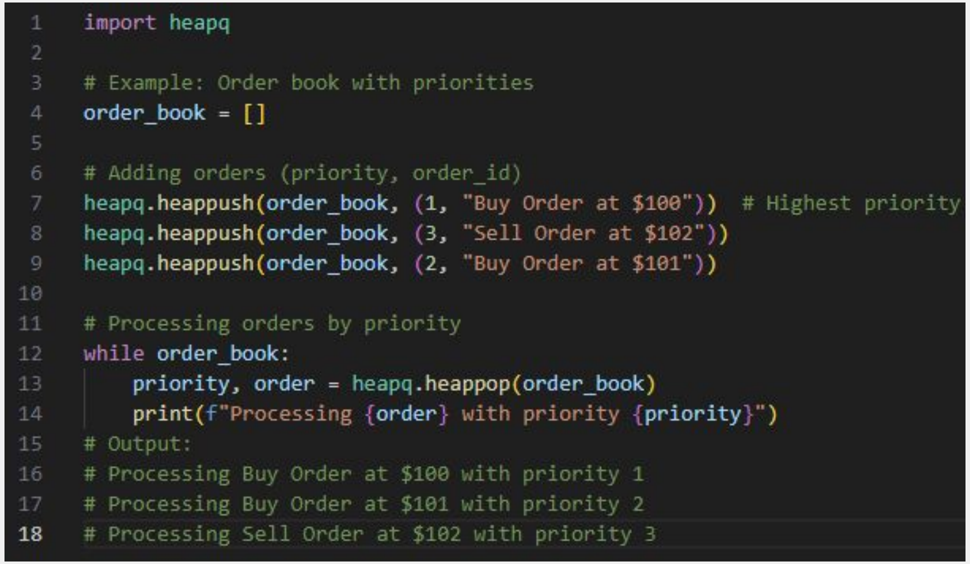
- O(n): Space is proportional to the number of elements.
- Con: Maintaining the heap structure can add overhead in memory.



Use Case:

- Managing Order Books in High-Frequency Trading:
  - **Order books** in trading list all buy and sell orders for a security, sorted by price and priority.
- A **priority queue** is a perfect fit for this task because it efficiently organizes orders so the most urgent (e.g., highest bid or lowest ask) can be processed first.
  - Traders use priority queues to manage the flow of **trade executions**, ensuring that the best orders are matched as quickly as possible.

Simple Python Implementation:

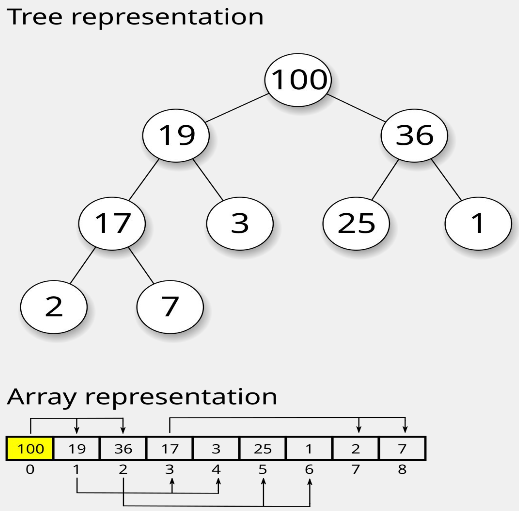


```python
import heapq

# Example: Order book with priorities
order_book = []

# Adding orders (priority, order_id)
heapq.heappush(order_book, (1, "Buy Order at $100"))  # Highest priority
heapq.heappush(order_book, (3, "Sell Order at $102"))
heapq.heappush(order_book, (2, "Buy Order at $101"))

# Processing orders by priority
while order_book:
    priority, order = heapq.heappop(order_book)
    print(f"Processing {order} with priority {priority}")
# Output:
# Processing Buy Order at $100 with priority 1
# Processing Buy Order at $101 with priority 2
# Processing Sell Order at $102 with priority 3
```



Graphs

Properties:

- **Directed:** Edges have a direction. (one way street)
- **Undirected:** Edges have no direction. (both way street)
- **Weighted:** Edges have weights representing the cost of traversal (the price to travel from one side to the other)

Representation:

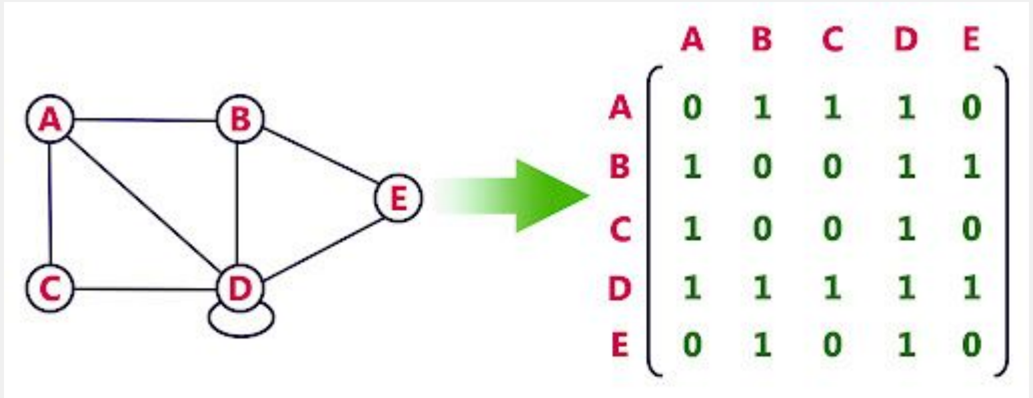
- Adjacency Matrix: 2D array where each cell (i, j) indicates if an edge exists between node i and j.
- Adjacency List: Each node has a list of adjacent nodes, more space-efficient for sparse graphs.

Space Complexity:

- Adjacency Matrix: $O(n^2)$
- Adjacency List: $O(n + e)$ (where n is the number of nodes, and e is the number of edges)

Use Case:

- Data Analysis & Visualization - Modeling Asset Correlations:
    - **Portfolio Optimization**: Use graphs to model relationships between different assets, helping traders identify which assets tend to move together, which is crucial for optimizing a diversified portfolio.
- **Market Sector Relationships**: Create a graph representing different market sectors and their correlations. This helps traders understand how sectors are interconnected and how shocks in one sector can affect others.
    - **Risk Management**: By modeling dependencies and correlations between assets, graphs help in identifying risk exposures and implementing strategies to hedge against them.
- **Network Analysis for Liquidity**: Analyze liquidity flows between markets or exchanges by using graph structures to model liquidity as edges between market participants.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Implementation:

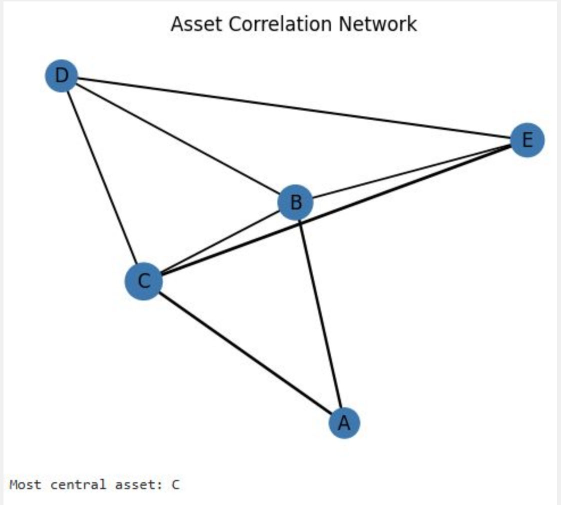
Knowing the **most central asset** in a financial network is important in quantitative trading because it provides:

Diversification:

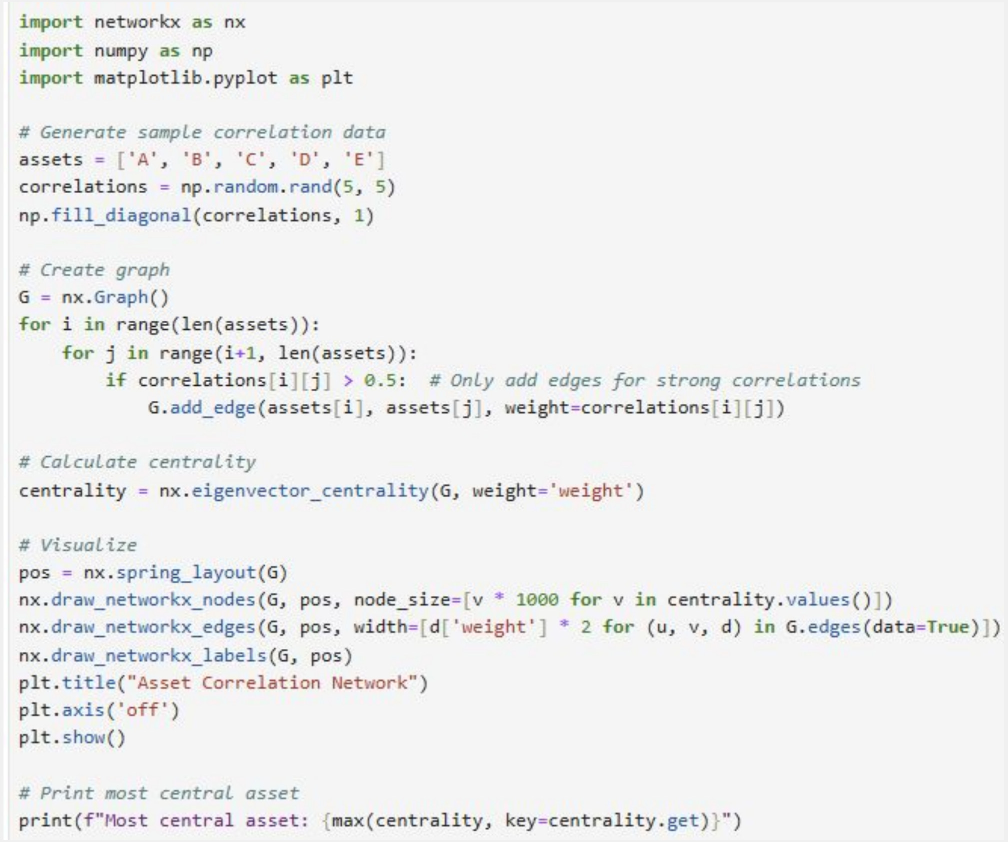
- By knowing which assets dominate the network, traders can choose to diversify away from these assets to reduce exposure to correlated movements.

Liquidity Flows:

- Central assets often have the highest trading volume or liquidity, making them key players in understanding liquidity patterns.



```
Most central asset: C
```



```python
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt

# Generate sample correlation data
assets = ['A', 'B', 'C', 'D', 'E']
correlations = np.random.rand(5, 5)
np.fill_diagonal(correlations, 1)

# Create graph
G = nx.Graph()
for i in range(len(assets)):
    for j in range(i+1, len(assets)):
        if correlations[i][j] > 0.5:  # Only add edges for strong correlations
            G.add_edge(assets[i], assets[j], weight=correlations[i][j])

# Calculate centrality
centrality = nx.eigenvector_centrality(G, weight='weight')

# Visualize
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, node_size=[v * 1000 for v in centrality.values()])
nx.draw_networkx_edges(G, pos, width=[d['weight'] * 2 for (u, v, d) in G.edges(data=True)])
nx.draw_networkx_labels(G, pos)
plt.title("Asset Correlation Network")
plt.axis('off')
plt.show()

# Print most central asset
print(f"Most central asset: {max(centrality, key=centrality.get)}")
```

Advanced Segment:

Runtime Optimization

Bit Manipulation

Bitmask Creation:

- A bitmask is an integer used for bitwise operations, where each bit represents a boolean flag.
- Quant relevance: Bitmasks can efficiently encode multiple trading signals or asset characteristics in a single integer.

Sparse Matrices:

- Definition: Matrices with mostly zero elements.
- In this example: A 1000x1000 matrix with 95% probability of zero elements.
- Quant relevance: Common in financial modeling, e.g., representing sparse covariance matrices or large, mostly empty datasets/no correlation between assets.

Bitwise Operations:

- Definition: Operations that manipulate individual bits of binary numbers (bitwise OR (|=) and AND (&) operations)
- Quant relevance: Enables fast computations on binary data, useful for option pricing, risk management, and signal processing.

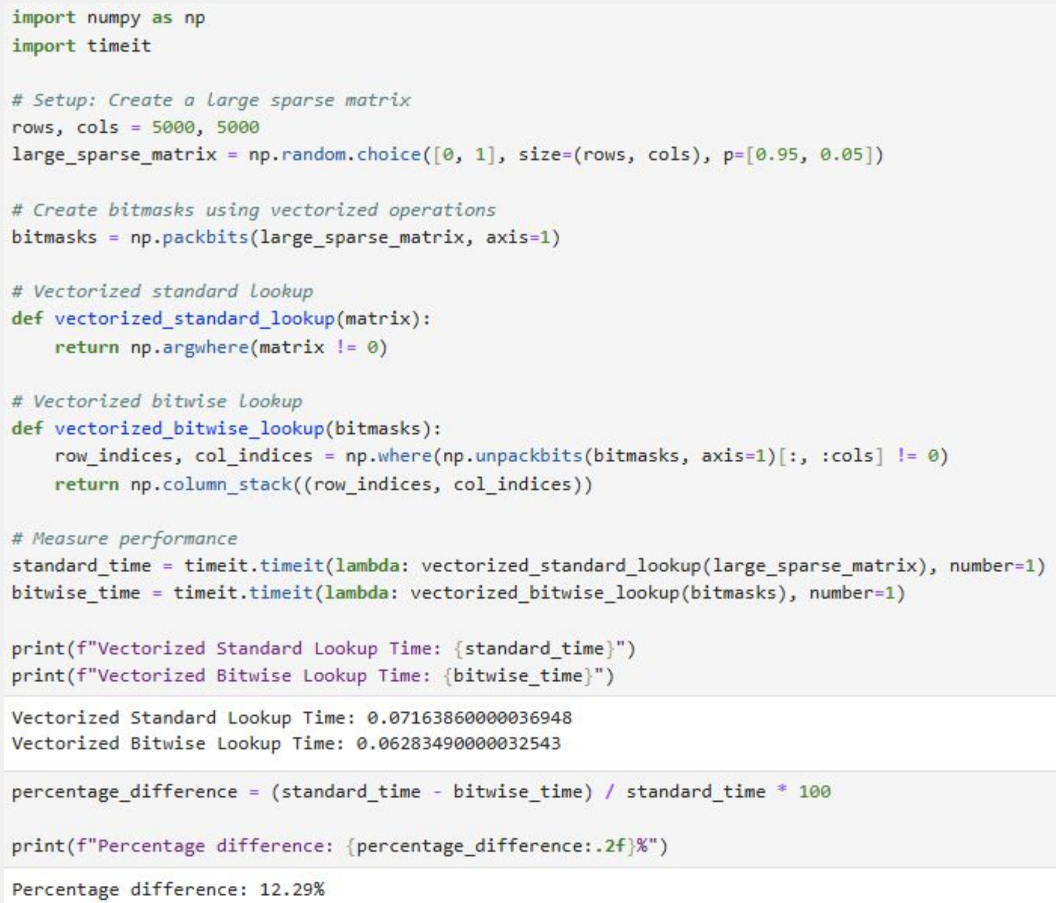
Use Case - Python Implementation:

Bitmask Creation:

- Each row of the matrix is converted into a bitmask.
- Non-zero elements are represented by setting the corresponding bit in the bitmask (95% probability of zero elements)
- This compact representation significantly reduces memory usage for sparse data.

Bitwise Operations:

- The bitwise_lookup function uses the bitwise AND operation (&) to check for non-zero elements.
- This approach is generally faster than iterating through each element in the matrix.



```python
import numpy as np
import timeit

# Setup: Create a large sparse matrix
rows, cols = 5000, 5000
large_sparse_matrix = np.random.choice([0, 1], size=(rows, cols), p=[0.95, 0.05])

# Create bitmasks using vectorized operations
bitmasks = np.packbits(large_sparse_matrix, axis=1)

# Vectorized standard lookup
def vectorized_standard_lookup(matrix):
    return np.argwhere(matrix != 0)

# Vectorized bitwise lookup
def vectorized_bitwise_lookup(bitmasks):
    row_indices, col_indices = np.where(np.unpackbits(bitmasks, axis=1)[:, :cols] != 0)
    return np.column_stack((row_indices, col_indices))

# Measure performance
standard_time = timeit.timeit(lambda: vectorized_standard_lookup(large_sparse_matrix), number=1)
bitwise_time = timeit.timeit(lambda: vectorized_bitwise_lookup(bitmasks), number=1)

print(f"Vectorized Standard Lookup Time: {standard_time}")
print(f"Vectorized Bitwise Lookup Time: {bitwise_time}")
```

```
Vectorized Standard Lookup Time: 0.07163860000036948
Vectorized Bitwise Lookup Time: 0.06283490000032543
```

```python
percentage_difference = (standard_time - bitwise_time) / standard_time * 100

print(f"Percentage difference: {percentage_difference:.2f}%")
```

```
Percentage difference: 12.29%
```

Bit Manipulation - Results

Relevance to Quant Trading:

- Efficient Data Handling: In high-frequency trading, where speed is crucial, such optimizations can provide a competitive edge.
- Correlation Analysis: The **sparse matrix** could represent **correlations between assets**. Faster lookups enable quicker identification of correlated assets.
- Portfolio Optimization: Rapid processing of large datasets allows for more frequent portfolio rebalancing and optimization.
- Risk Management: Quick analysis of sparse matrices can aid in faster risk assessment across multiple assets.

Memory Efficiency:

- The bitmask representation is particularly useful in quant applications dealing with large datasets where memory constraints are a concern.

Scalability:

- This technique becomes more advantageous as the matrix size increases, which is common in quant finance when dealing with numerous assets or time series data.

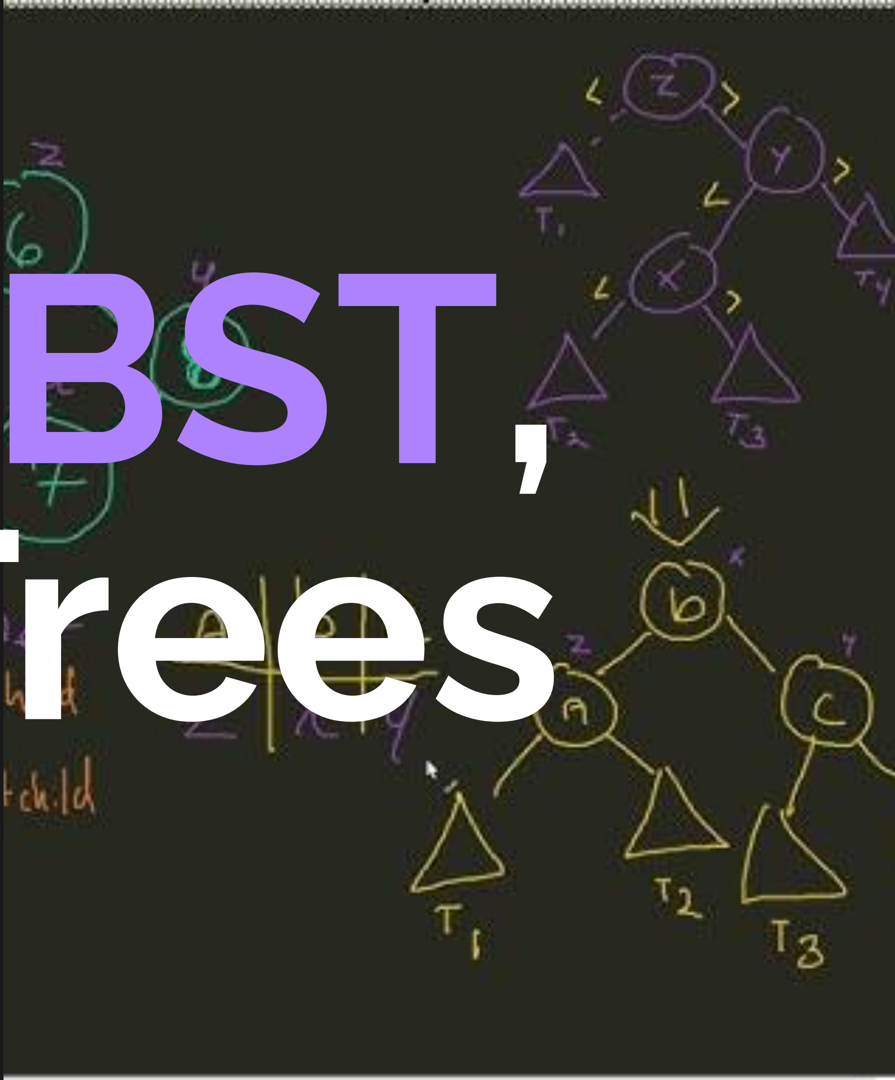
Thank You!

LinkedIn:

- Rudy Osuna - Will be posting Quant Algorithm content & walkthroughs
- Anuj Jain - Follow Anuj as he takes the next step in his journey in Berkeley for Financial Engineering!

(Extra):

Stacks, BST, AVL Trees



@TQT_UCSD

Stack

Properties:

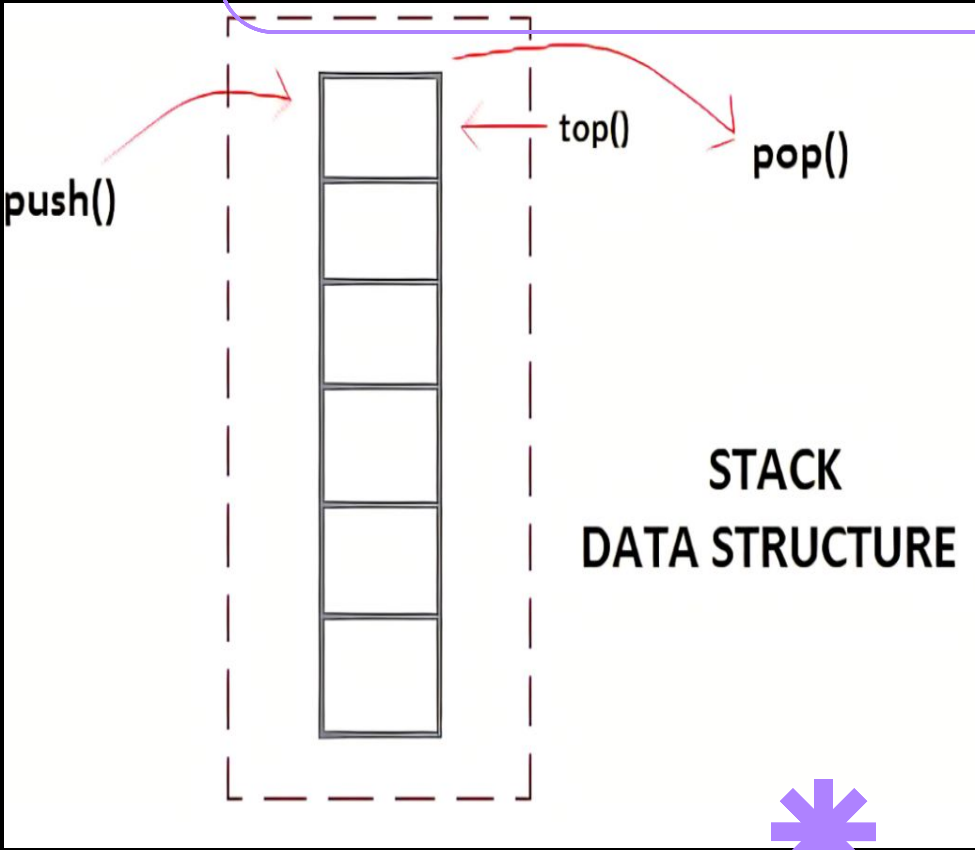
- LIFO (Last In, First Out)
- the last element added is the first to be removed.

Operations and Runtimes:

- Push: O(1) – Adding an element to the top of the stack.
- Pop: O(1) – Removing the top element from the stack.
- Access top: O(1) – Viewing the element at the top of the stack.

Space Complexity:

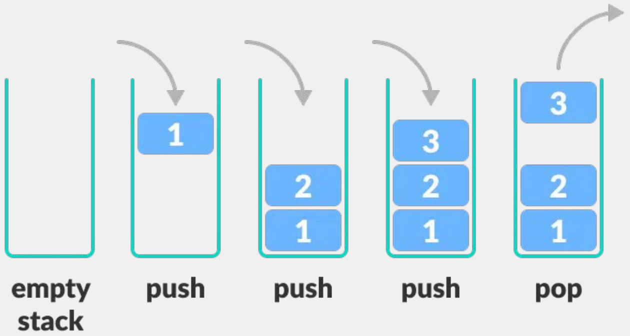
- O(n) – depending on the number of elements in the stack.



Use Case:

Used for Backtracking - Similar to breakpoint debugging: A stack allows you to quickly and efficiently revisit past market states in reverse chronological order. This is particularly important when you want to:

- **Analyze historical performance:** Understand why specific trades were made.
- **Debug strategies:** Test the logic of buy/sell triggers.
- **Optimize strategies:** Identify flaws in decision-making by reevaluating past states.
- **Trading Signal Verification:** Backtracking with a stack lets you store and revisit market conditions (e.g., price, indicators) that led to signals. If a signal is triggered (e.g., an overbought/oversold condition), you can use the stack to check prior states to confirm if the trigger was valid or if it aligns with strategy rules.



Scenario:

  - Imagine a strategy that triggers a buy when the price crosses below its moving average (MA) and an RSI indicator drops below 30. Later, if the price crosses back above the MA and RSI exceeds 70, it triggers a sell.

- **Using a stack, we can:**
  - Store each market state during backtesting (timestamp, price, MA, RSI).
  - Analyze historical states when a buy/sell signal is triggered.
  - Compare past market states to confirm whether the strategy is working correctly.

Binary Search Tree (BST) AVL Trees

Properties:

- **BST**: Each node has at most two children, and for each node, all values in the left subtree are smaller, and in the right subtree are larger.
- **AVL Tree**: A self-balancing binary search tree, where the height difference between left and right subtrees is at most 1.

Operations and Runtimes:

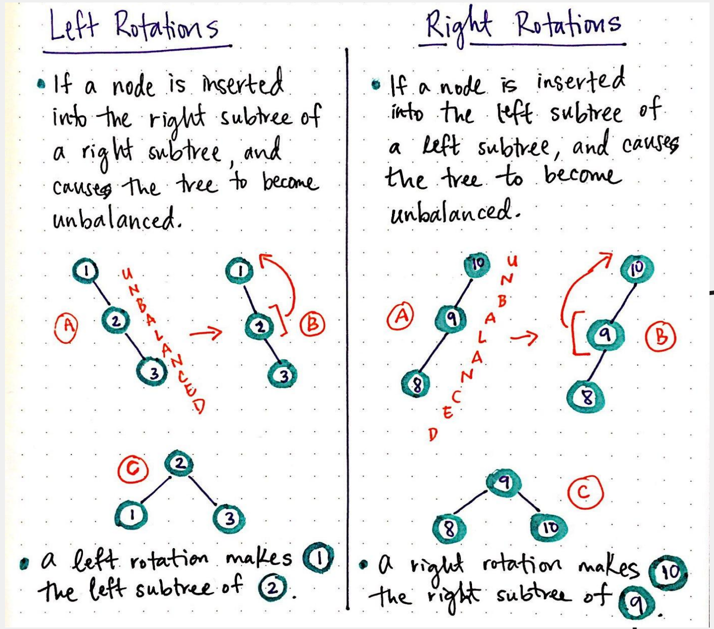
- Search/Insert/Delete: O(log n) for both BST and AVL Trees (AVL ensures balanced structure).

Space Complexity:

- O(n) – Space complexity is proportional to the number of nodes in the tree.add overhead in memory.

Use Case:

- Storing Historical Trade Data for Range Queries:
  - **Efficient Price-Based Queries:** Use a binary search tree (BST) or AVL tree to store historical trades by price. The tree structure enables efficient searching and retrieval of trades within a specified price range.
- **Time Series Data Analysis**: Implement trees to store trade timestamps, allowing quick access to trades within specific time ranges.
  - **Efficient Order Matching:** In high-frequency trading, BSTs and AVL trees can help efficiently match orders based on time or price criteria.
- **Dynamic Updates:** Trees offer efficient insertion and deletion, making them ideal for real-time trading environments where data is constantly changing.



Left Rotations

- If a node is inserted into the right subtree of a right subtree, and causes the tree to become unbalanced.

- a left rotation makes 1 the left subtree of 2.

Right Rotations

- If a node is inserted into the left subtree of a left subtree, and causes the tree to become unbalanced.

- a right rotation makes 10 the right subtree of 9.