Data Structures Review

Refreshing the Fundamentals

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

Why Are We Doing This?

What we'll cover today:

1. Arrays and Linked Lists
2. Stacks and Queues
3. Trees and Binary Search Trees
4. Heaps and Priority Queues
5. Hash Tables

This should be review - but let's make sure we're all on the same page!

Arrays - The Foundation

What is an array?

- Contiguous memory
- Fixed size
- Same type elements
- Random access

Strengths:

- Fast access
- Cache-friendly
- Simple

Key operations:

Operation	Time
Access	$O(1)$
Search	$O(n)$
Insert end	$O(1)$
Insert at i	$O(n)$
Delete at i	$O(n)$

Weaknesses:

- Fixed size
- Expensive insert/delete

Dynamic Arrays (Vector, ArrayList)

Arrays that grow

Problem: Traditional arrays have fixed size

Solution: Dynamic arrays (C++ `vector`, Java `ArrayList`)

How they work:

1. Start with small capacity (e.g., 10 elements)
2. When full, allocate larger array (typically $2\times$ size)
3. Copy old elements to new array
4. Delete old array

Key insight:

- Individual insertions can be expensive: $O(n)$ when resizing
- But *amortized* cost is $O(1)$ per insertion
- We'll prove this rigorously next lesson!

Linked Lists

Dynamic size, no shifting needed

Structure:

```
struct Node {
    int data;
    Node* next;
};
```

Types:

- Singly linked
- Doubly linked
- Circular

Key operations:

Operation	Time	Why?
Access i	$O(n)$	Traverse
Search	$O(n)$	Check all
Insert head	$O(1)$	Update ptrs
Insert at i	$O(n)$	Traverse
Delete at i	$O(n)$	Traverse

Arrays vs Linked Lists - The Tradeoffs

Feature	Array	Linked List
Access time	$O(1)$	$O(n)$
Insert at beginning	$O(n)$	$O(1)$
Insert at end	$O(1)^*$	$O(n)^{**}$
Memory overhead	None	Pointer per node
Cache performance	Excellent	Poor
Size flexibility	Fixed*	Dynamic

*If space available (dynamic arrays amortize this)

**Unless you maintain a tail pointer

When to use arrays:

- Need fast random access
- Size is known or changes rarely
- Memory is tight (no pointer overhead)

Stacks - LIFO

Last In, First Out

Stack ADT operations:

- `push(x)` - $O(1)$
- `pop()` - $O(1)$
- `top()` - $O(1)$
- `isEmpty()` - $O(1)$

Implementations:

- Array-based
- Linked-list-based

Applications:

- Function call stack
- Undo mechanisms
- Expression evaluation
- Depth-first search
- Browser back button

Queues - FIFO

First In, First Out

Queue ADT operations:

- `enqueue(x)` - $O(1)$
- `dequeue()` - $O(1)$
- `front()` - $O(1)$
- `isEmpty()` - $O(1)$

Implementations:

- Circular array
- Linked list (head + tail)

Applications:

- Breadth-first search
- Task scheduling
- Printer queues
- I/O buffering
- Network packets

Deques - Double-Ended Queues

Deque: Can insert/remove from both ends

Operations (all $O(1)$):

- `pushFront(x)`, `pushBack(x)`
- `popFront()`, `popBack()`

Key insight: Deques generalize both stacks and queues

- Stack = use one end only
- Queue = push at one end, pop at other

Implementation: Typically circular array or doubly-linked list

C++ STL: `std::deque`

Trees - Basic Terminology

You should know these terms

Tree: Hierarchical structure

Key terms:

- **Root:** Top node
- **Leaf:** No children
- **Parent/Child:** Connected
- **Siblings:** Same parent
- **Depth:** Distance from root
- **Height:** Path to leaf
- **Level:** Same depth

Binary Tree:
At most 2 children per node

Properties:

- n nodes $= n - 1$ edges
- Height: $\log n$ to n
- Balanced: $h = O(\log n)$
- Degenerate: $h = O(n)$

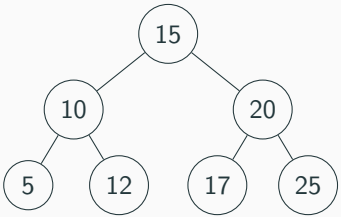
Binary Search Trees (BST)

Ordered trees for fast search

BST Property:

- Left subtree $<$ node
- Right subtree $>$ node

Example BST:



Operations (balanced):

- Search: $O(\log n)$
- Insert: $O(\log n)$
- Delete: $O(\log n)$

Problem:

Can degenerate to $O(n)$ if unbalanced

(We'll fix this with balanced trees next week!)

BST Operations - Quick Review

Search:

```
Node* search(Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    return search(root->right, key);
}
```

Insert:

```
Node* insert(Node* root, int key) {
    if (root == NULL) return new Node(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else root->right = insert(root->right, key);
    return root;
}
```

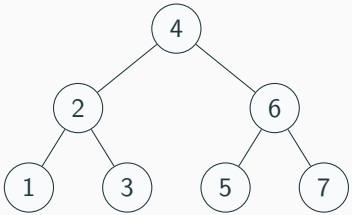
Delete cases:

1. Leaf: Remove
2. One child: Replace with child
3. Two children: Use successor

Tree Traversals: The Rules

We will trace all four orders on the same tree

The tree we will use:



Four traversal orders:

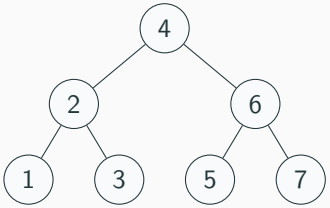
Inorder	Left $\rightarrow$ Root $\rightarrow$ Right
Preorder	Root $\rightarrow$ Left $\rightarrow$ Right
Postorder	Left $\rightarrow$ Right $\rightarrow$ Root
Level-order	Level by level (use a queue)

Memory trick:

- Pre/In/Post tells you *when the root is visited*
- Left always comes before Right

Tree Traversals: Traced on Our Example

Same tree, four different visit orders



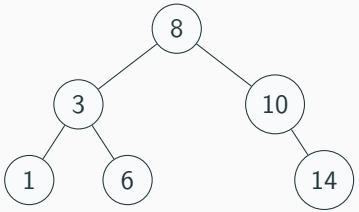
Order	Output	Key use
Inorder	1, 2, 3, **4**, 5, 6, 7	Sorted output from BST
Preorder	**4**, 2, 1, 3, 6, 5, 7	Copy / serialize tree
Postorder	1, 3, 2, 5, 7, 6, **4**	Delete tree safely
Level-order	**4**, 2, 6, 1, 3, 5, 7	BFS; level-by-level

Notice: Inorder on a BST always gives sorted order — this is a direct consequence of the BST property.

All traversals: $O(n)$ — every node visited exactly once.

Tree Traversals: Practice

Try these on the tree below before checking your answers.



1. What is the **inorder** traversal?
2. What is the **preorder** traversal?
3. What is the **postorder** traversal?
4. What is the **level-order** traversal?
5. Is this a valid BST? How can you tell from your inorder answer?
6. Node 10 has no left child. How does that affect each traversal?

Answers: (1) 1,3,6,8,10,14 (2) 8,3,1,6,10,14 (3) 1,6,3,14,10,8 (4) 8,3,10,1,6,14

Heaps - Almost Complete Binary Trees

Efficient priority queues

Heap Property:

- **Min-heap:** Parent $\leq$ children
- **Max-heap:** Parent $\geq$ children

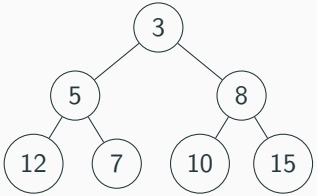
Structure: Complete binary tree

Array indices (0-indexed):

- Left child: $2i + 1$
- Right child: $2i + 2$
- Parent: $(i - 1)/2$

Example min-heap:

Array: `[3,5,8,12,7,10,15]`



Why heaps?

The best way to repeatedly find and remove the minimum (or maximum)

The core operation: always access the most important element next.

Why not just sort?

- Sorting costs $O(n \log n)$ up front
- But elements keep arriving — you'd re-sort every time
- A heap maintains order *dynamically*: $O(\log n)$ per insert or remove

Where this matters:

- **Dijkstra's algorithm**: always expand the closest unvisited node
- **Prim's algorithm**: always add cheapest edge
- **OS schedulers**: always run highest-priority process

The counterintuitive result:

Building a heap from n elements takes $O(n)$, not $O(n \log n)$.

Operation	Time
Insert	$O(\log n)$
Extract min/max	$O(\log n)$
Peek min/max	$O(1)$
Build from array	$O(n)$

Takeaway: Efficient algorithm design requires choosing the right data structure.

Bubble up

Restoring the heap after insertion

When: after inserting a new element.

Steps:

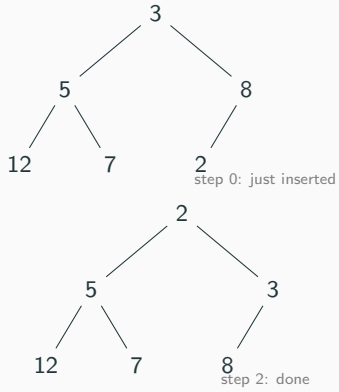
1. Append new element at end of array
2. Compare with its parent
3. If smaller than parent $\rightarrow$ swap
4. Repeat until heap property holds or root is reached

Example: insert **2** into this min-heap.

Array before: `[3, 5, 8, 12, 7]`
Array after append: `[3, 5, 8, 12, 7, 2]`

Step	Action	Array
1	2 vs parent 8: swap	`[3,5,2,12,7,8]`
2	2 vs parent 3: swap	`[2,5,3,12,7,8]`
3	2 is root: done	

Visualized (after each swap):



Cost: at most $h = O(\log n)$ swaps.

Bubble down

Restoring the heap after extract-min

When: after removing the root (extract-min).

Steps:

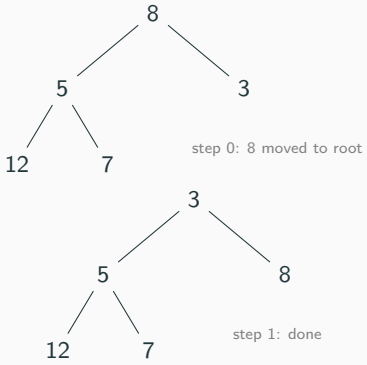
1. Remove root; move last element to root
2. Compare with both children
3. If larger than either child → swap with the *smaller* child
4. Repeat until heap property holds or a leaf is reached

Why the smaller child? If you swap with larger child, that child becomes the parent but is still larger than the other child violating heap property immediately.

Example: extract min from `[2, 5, 3, 12, 7, 8]`.

Step	Array
Move last to root	`[8, 5, 3, 12, 7]`
8 vs children 5, 3: swap 3	`[3, 5, 8, 12, 7]`
8 is a leaf: done	

Visualized:



Cost: at most $h = O(\log n)$ swaps.

Putting it together

Every heap operation reduces to bubble up or bubble down

Operation	Steps	Cost
Insert	Append, bubble *up*	$O(\log n)$
Extract-min	Move last to root, bubble *down*	$O(\log n)$
Peek	Return root	$O(1)$
Heapify	Bubble *down* each non-leaf	$O(n)$

The pattern:

- Adding at the bottom $\rightarrow$ bubble *up*
- Placing something at the top $\rightarrow$ bubble *down*
- Heapify works bottom-up: start at the last non-leaf and bubble down each node

Why heapify is $O(n)$, not $O(n \log n)$:

Most nodes are near the bottom and travel very little.

Level	Nodes	Max swaps
Leaves	$n/2$	0
Level above	$n/4$	1
Level above	$n/8$	2
$\vdots$	$\vdots$	$\vdots$
Root	1	$\log n$

$$\text{Total work} = \sum_{k=0}^{\log n} \frac{n}{2^{k+1}} \cdot k \leq 2n = O(n)$$

Heap Operations

1. **Insert** — $O(\log n)$**:**
   - Add at end of array
   - Bubble *up*: swap with parent while heap property violated
   - At most $h = O(\log n)$ swaps
2. **Extract-Min** — $O(\log n)$**:**
   - Remove root (the min)
   - Move last element to root
   - Bubble *down*: swap with smaller child while violated
   - At most $h = O(\log n)$ swaps
3. **Heapify (build from array)** $O(n)$**:**
   - Naïve: insert n elements one by one $\Rightarrow O(n \log n)$
   - Better: start at last non-leaf, bubble *down* each node
   - **Why** $O(n)$**?** Most nodes are near the bottom and barely move. Only $n/2$ nodes need any work at all, and the average bubble-down distance is $O(1)$, not $O(\log n)$.
   - Exact proof via geometric series — we'll revisit with amortized analysis.
4. **Peek** — $O(1)$**:** Return root.

Heapify: building a heap from scratch

The smart way is $O(n)$, not $O(n \log n)$

The problem: given an arbitrary unsorted array, rearrange it in-place so that every parent is $\leq$ its children (a valid min-heap).

Naive approach — insert one by one:

- Insert each element and bubble up
- Each insertion costs $O(\log n)$
- Total: $n \times O(\log n) = O(n \log n)$

Smarter approach — bottom-up:

- Start at the last non-leaf node
- Work backwards to the root
- Bubble *down* at each node
- Total: $O(n)$ (surprisingly!)

Why is bottom-up $O(n)$?

Leaf nodes — roughly half the array — need *zero* work. Nodes one level up need at most 1 swap. Only the root can require a full $\log n$ bubble-down.

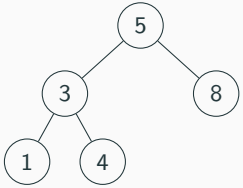
Level	Nodes	Max swaps
Leaves	$n/2$	0
Level above	$n/4$	1
Level above	$n/8$	2
$\vdots$	$\vdots$	$\vdots$
Root	1	$\log n$

The total work is a geometric series that sums to $\leq 2n = O(n)$. The work is concentrated at the *top*, not spread evenly.

Heapify: worked example

Building a min-heap from `[5, 3, 8, 1, 4]` bottom-up

Step 0: draw the array as a tree.
Last non-leaf $= \lfloor 5/2 \rfloor - 1 =$ index 1 (value 3).
Work right-to-left from index 1 to index 0.



Not a valid min-heap: root 5 $>$ child 3.

Step 1: bubble down index 1 (value 3).
Children of index 1: index 3 (value 1), index 4 (value 4).
$3 > 1$, so swap $3 \leftrightarrow 1$.

`[5, 1, 8, 3, 4]`

Step 2: bubble down index 0 (value 5).
Children of index 0: index 1 (value 1), index 2 (value 8).
$5 > 1$, so swap $5 \leftrightarrow 1$.

`[1, 5, 8, 3, 4]`
Continue bubble-down on 5 at index 1:
Children: index 3 (value 3), index 4 (value 4).
$5 > 3$, so swap $5 \leftrightarrow 3$.

`[1, 3, 8, 5, 4]` ✓ valid min-heap.

Key: leaves (indices 2, 3, 4) were never touched.

Heap Operations: Practice

Given this array:

```
[10, 3, 7, 1, 8, 2, 6]
```

Questions:

1. Draw the array as a complete binary tree. Is it a valid min-heap?
2. Apply heapify (build a min-heap). Show the state of the array after each bubble-down step.
3. Why is heapify $O(n)$ and not $O(n \log n)$? Which nodes do the most work, and which do almost none?

Recall — array index rules (0-indexed):

- Left child of i: $2i + 1$
- Right child of i: $2i + 2$
- Parent of i: $\lfloor (i - 1)/2 \rfloor$
- Last non-leaf: $\lfloor n/2 \rfloor - 1$

Hint for Q3: There are $n/2$ leaves — they need zero bubble-down steps. The nodes one level up need at most 1 step. Only the root can take $\log n$ steps. Adding these up gives $O(n)$ total work, not $O(n \log n)$.

Heap Operations: Practice Answers

Starting array: `[10, 3, 7, 1, 8, 2, 6]`

Q1: Not a valid min-heap. Node 10 (index 0) has children 3 and 7, violating parent $\leq$ children.

Q2: Heapify trace. Last non-leaf = index 2 (value 7). Work right-to-left:

- $i = 2$: children are 2, 6. Swap 7$\leftrightarrow$2. `[10,3,2,1,8,7,6]`
- $i = 1$: children are 3, 8. No swap needed.
- $i = 0$: children are 3, 2. Swap 10$\leftrightarrow$2. `[2,3,10,1,8,7,6]`
  Bubble down 10: children 7,6. Swap 10$\leftrightarrow$6. `[2,3,6,1,8,7,10]`

Final heap: `[2,3,6,1,8,7,10]` ✓

Q3: Half the nodes are leaves (0 steps). The next level needs $\leq$1 step. Only the root can take $\log n$ steps. The total sums to $O(n)$.

Q4: Insert 0. Append: `[2,3,6,1,8,7,10,0]` (index 7).
Bubble up: parent of 7 is index 3 (value 1). Swap.
`[2,3,6,0,8,7,10,1]`
Parent of 3 is index 1 (value 3). Swap.
`[2,0,6,3,8,7,10,1]`
Parent of 1 is index 0 (value 2). Swap.
`[0,2,6,3,8,7,10,1]` ✓

Q5: Extract min. Remove root (0). Move last (1) to root.
`[1,2,6,3,8,7,10]`
Bubble down 1: children 2,6. No swap needed.
Final: `[1,2,6,3,8,7,10]` ✓

Priority Queues

Heaps are the standard implementation

Priority Queue ADT:

- Elements have priorities
- Highest priority dequeues first
- Not FIFO!

Operations:

- `insert(x, priority)`
- `extractMin()`
- `peek()`

Implementation comparison:

Type	Insert	Extract
Unsorted	$O(1)$	$O(n)$
Sorted	$O(n)$	$O(1)$
Heap	$O(\log n)$	$O(\log n)$

Heap wins! Balanced $O(\log n)$ for both.

Heap Sort - Sorting with Heaps

Algorithm:

1. Build a max-heap from the array - $O(n)$
2. Repeatedly extract max (swap root with last element, heapify) - $O(n \log n)$

Total complexity: $O(n \log n)$

Properties:

- ✓ Always $O(n \log n)$ (no worst case like quick sort)
- ✓ In-place (no extra memory like merge sort)
- ✗ Not stable (relative order not preserved)
- ✗ Poor cache performance (jumps around in memory)

When to use: When you need guaranteed $O(n \log n)$ and can't afford extra space

Hash Tables

Fast lookups with hash functions

Goal: $O(1)$ search, insert, delete (on average)

Key idea: A hash table is an array where a function decides the index based on the key (no searching, just arithmetic).

How it works:

1. **Hash function:** Map key to array index
   Example: `hash(key) = key % tableSize`
2. **Store value:** At computed index
3. **Retrieve value:** Hash key, look at that index

Problem: **Collisions!** What if two keys hash to same index?

Example: With `tableSize = 10`

- Key 15 $\rightarrow$ index 5
- Key 25 $\rightarrow$ index 5 (collision!)

Need collision resolution strategy...

Collision Resolution Strategies

1. Chaining:

- Each slot = linked list
- Add collisions to list
- Search: $O(1 + \alpha)$
- $\alpha = n/m$ (load factor)
- Simple, robust

2. Open Addressing:

- All in table (no lists)
- Probe for empty slot:
  - Linear: $h(k), h(k) + 1, ...$
  - Quadratic: $h(k) + i^2$
  - Double hash: 2nd function
- Cache-friendly
- More complex

Key: Keep load factor $\alpha = n/m < 0.7$ for good performance!

Hash Functions

Good hashing is crucial

Good hash properties:

1. Deterministic
2. Fast - $O(1)$
3. Uniform distribution
4. Minimize collisions

Common methods:

Division: `h(k) = k % m`

- Simple, fast
- Use prime m

Multiplication:

- Use constant $A \approx 0.618$
- Less sensitive to m

Strings: Polynomial

- `s[0]*31^2 + s[1]*31 + s[2]`

Hash Table Performance

Complexity:

Op	Avg	Worst
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

When does worst case happen?

- A bad hash function sends *every* key to the same slot
- An adversary who knows your hash function can craft such inputs
- **Not** typical with a good hash on real data

Load factor $\alpha = n/m$:

- n = number of elements stored
- m = table size
- Keep $\alpha < 0.7$ for good average performance
- High $\alpha \rightarrow$ more collisions $\rightarrow$ longer chains

Resizing:

- When α too high: allocate $2\times$ table, rehash everything
- Costs $O(n)$ but happens rarely — amortized $O(1)$ per insert
- Same idea as dynamic arrays!

Why hash tables?

$O(1)$ **average lookup — the most practically useful complexity in computing**

The core idea: turn a lookup into arithmetic.

Instead of searching through n elements, compute an index directly from the key. With a good hash function and a low load factor, this is essentially instantaneous.

Where this matters:

- **Database indexing**: looking up a record by ID
- **Caching / memoization**: have we computed this before?
- **Detecting duplicates**: scan an array once, not n^2 times
- **Compiler symbol tables**: is this variable declared?

Every high-level language's built-in dictionary or map is a hash table. You use one every time you write `dict[key]` or `map.get(key)`.

Average vs. worst case:

Operation	Average	Worst
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Worst case requires *bad hash function* or adversary crafting inputs to cause collisions (not typical).

What hash tables cannot do:

- Find the min or max: use a heap
- Iterate in sorted order: use a BST
- Range queries (all keys b/t a and b): use BST

Need fast lookup, no ordering? Hash table. Need ordering? BST.

Quick Reference - When to Use Each Structure

Structure	Use When...
Array	Need fast random access, size is fixed or rarely changes
Dynamic Array	Need fast random access, size changes gradually
Linked List	Frequent insertions/deletions at beginning, no random access needed
Stack	Need LIFO behavior (undo, recursion, DFS)
Queue	Need FIFO behavior (scheduling, BFS)
BST	Need sorted data with fast search/insert/delete
Heap	Need to repeatedly find/remove min/max element
Hash Table	Need very fast lookups, don't need ordering

Remember: Choose based on the operations you'll do most frequently!

Complexity Cheat Sheet

Structure	Access	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Dynamic Array	$O(1)$	$O(n)$	$O(1)*$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)**$	$O(1)**$
Stack	-	-	$O(1)$	$O(1)$
Queue	-	-	$O(1)$	$O(1)$
BST (balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
BST (unbalanced)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Heap	-	$O(n)$	$O(\log n)$	$O(\log n)$
Hash Table (avg)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Hash Table (worst)	$O(n)$	$O(n)$	$O(n)$	$O(n)$

*Amortized for push_back

**If you have pointer to location

Key takeaways:

- Arrays: great access, poor insert/delete
- Linked lists: opposite of arrays
- BST: $O(\log n)$ if balanced (next week: how to balance!)
- Hash tables: $O(1)$ average, but no ordering

Quick Check - Are We Ready?

Answer these to yourself:

1. What's the difference between a stack and a queue?
2. Why is array access $O(1)$ but linked list access $O(n)$?
3. What's the BST property?
4. What makes a heap a heap?
5. Why are hash tables $O(1)$ on average but $O(n)$ worst case?
6. When would you use a heap vs a BST?
7. What happens when a dynamic array runs out of space?

If you can answer these, you're ready for advanced topics!

If not: Review your notes from the first course, come to office hours.

What's Coming Next

Next lesson: Amortized Analysis

- Why dynamic array push_back is $O(1)$ amortized (we'll prove it!)
- Three methods: aggregate, accounting, potential
- Union-Find with path compression

Then: Advanced Data Structures

- Balanced trees (AVL) - solving the BST degeneration problem
- Advanced hashing - universal hashing, perfect hashing

Then: Algorithm Design Paradigms

- Greedy algorithms
- Dynamic programming
- Graph algorithms

This course builds on the foundation we reviewed today!

Resources for Review

If you need more review:

- **Office hours:** Come ask questions!
- **Textbook:** Review chapters on basic data structures
- **Online resources:**
  - VisuAlgo: `visualgo.net` (visualize data structures)
  - GeeksforGeeks: Data structures tutorials
  - YouTube: mycodeschool, Abdul Bari
- **Practice:**
  - Implement each structure from scratch
  - Solve problems on LeetCode, HackerRank

Don't fall behind! This course moves fast and builds on these concepts.