Data Structures: Non-Linear Data Structures

Minseok Jeon
DGIST

November 2, 2025

Contents

Introduction

What are Non-Linear Data Structures?

Definition

Non-linear data structures organize data in hierarchical or networked relationships, unlike linear structures where elements follow a sequential order.

Key Characteristics

- Elements not in sequence
- Hierarchical or networked
- Multiple paths between elements
- Model real-world relationships

Main Types

- **Trees**: Hierarchical
- **Graphs**: Networked

Knowledge Points

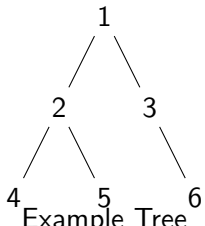
1. Trees: binary trees and BSTs
2. Tree traversals: inorder/preorder/postorder
3. Heaps: min-heap and max-heap
4. Balanced trees: AVL and Red-Black
5. Graph representations: list vs matrix
6. Graph traversals: BFS and DFS
7. Applications and modeling

Trees and Binary Search Trees

Tree Terminology

Basic Terms

- **Root**: Topmost node
- **Parent**: Node with children
- **Child**: Node connected below
- **Leaf**: Node with no children
- **Edge**: Connection between nodes
- **Height**: Longest path to leaf
- **Depth**: Distance from root



Example Tree

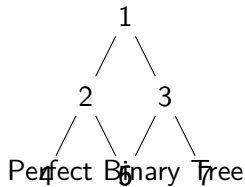
Binary Trees

Definition

A **binary tree** is a tree where each node has at most two children (left and right).

Types of Binary Trees:

- **Full**: Every node has 0 or 2 children
- **Complete**: All levels filled except possibly last
- **Perfect**: All internal nodes have 2 children, all leaves at same level
- **Degenerate**: Each node has only one child



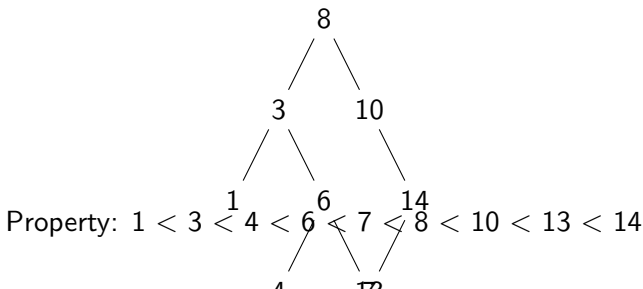
Perfect Binary Tree

Binary Search Tree (BST)

Definition

A **BST** is a binary tree with ordering property:

- Left subtree contains only nodes with values **less than** parent
- Right subtree contains only nodes with values **greater than** parent
- Both subtrees are also BSTs



Property: $1 < 3 < 4 < 6 < 7 < 8 < 10 < 13 < 14$

BST Operations

Search - O(h)

```
def search(root, target):
    if root is None or
       root.value == target:
        return root

    if target < root.value:
        return search(root.left,
                      target)
    else:
        return search(root.right,
                      target)
```

Insert - O(h)

```
def insert(root, value):
    if root is None:
        return TreeNode(value)

    if value < root.value:
        root.left = insert(
            root.left, value)
    elif value > root.value:
        root.right = insert(
            root.right, value)

    return root
```

Complexity

h = height of tree. Average: O(log n), Worst: O(n) when tree is skewed

BST vs Array vs Linked List

Operation	Sorted Array	Linked List	BST (balanced)
Search	O(log n)	O(n)	O(log n)
Insert	O(n)	O(1)*	O(log n)
Delete	O(n)	O(1)*	O(log n)
Min/Max	O(1)	O(n)	O(log n)
Sorted order	Built-in	No	Inorder traversal

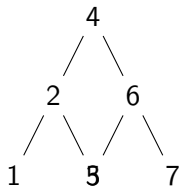
*assuming position is known

Applications

Databases (indexing), file systems, expression trees, symbol tables, priority queues

Tree Traversals

Depth-First Traversals (DFS)



Three Main Types

1. **Inorder** (Left → Root → Right)
   - Order: 1, 2, 3, 4, 5, 6, 7
   - Use: Get sorted values from BST
2. **Preorder** (Root → Left → Right)
   - Order: 4, 2, 1, 3, 6, 5, 7
   - Use: Copy tree, serialize
3. **Postorder** (Left → Right → Root)
   - Order: 1, 3, 2, 5, 7, 6, 4
   - Use: Delete tree, calculate size

Inorder Traversal Implementation

Recursive Version

```python
def inorder(root):
    if root is None:
        return []

    result = []
    # Left
    result.extend(
        inorder(root.left))
    # Root
    result.append(root.value)
    # Right
    result.extend(
        inorder(root.right))

    return result
```

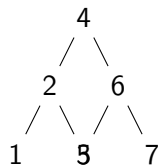
Iterative Version (Stack)

```python
def inorder_iterative(root):
    result = []
    stack = []
    current = root

    while current or stack:
        # Go to leftmost
        while current:
            stack.append(current)
            current = current.left

        # Process node
        current = stack.pop()
        result.append(
            current.value)

        # Visit right
        current = current.right

    return result
```

Breadth-First Traversal (BFS)

Level-Order Traversal

Visit nodes level by level, left to right



Order: 4, 2, 6, 1, 3, 5, 7

```python
from collections import deque

def level_order(root):
    if root is None:
        return []

    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        result.append(node.value)

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

    return result
```

Use Case

Shortest path, level processing, check if tree is complete

Traversal Comparison

Traversal	Order	Use Case	Data Structure
Inorder	Left-Root-Right	Sorted values (BST)	Stack
Preorder	Root-Left-Right	Copy, serialize	Stack
Postorder	Left-Right-Root	Delete, calculate	Stack
Level-order	Level by level	Shortest path	Queue

Complexity

- **Time**: O(n) - visit each node once
- **Space**: O(h) for recursion/stack, O(w) for queue (w = max width)

Heaps

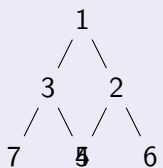
What is a Heap?

Definition

A **heap** is a specialized tree-based data structure that satisfies the **heap property**, typically implemented as a complete binary tree stored in an array.

Min-Heap

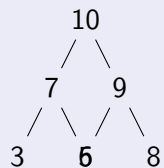
Parent $\leq$ Children



Array: [1, 3, 2, 7, 5, 4, 6]

Max-Heap

Parent $\geq$ Children



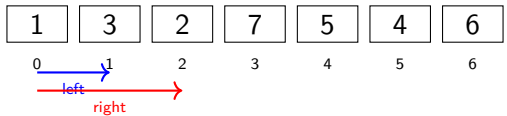
Array: [10, 7, 9, 3, 5, 6, 8]

Heap Array Representation

Index Relationships

For a node at index i:

- **Left child**: $2i + 1$
- **Right child**: $2i + 2$
- **Parent**: $\lfloor (i - 1)/2 \rfloor$



Efficiency

Complete binary tree structure ensures O(log n) height

Heap Operations

Insert - O(log n)

1. Add to end of array
2. Bubble up to restore heap property

```
def insert(self, value):
    self.heap.append(value)
    current = len(self.heap)-1

    while current > 0:
        parent = (current-1)//2
        if self.heap[current] <
           self.heap[parent]:
            self.swap(current,
                      parent)
            current = parent
        else:
            break
```

Extract Min - O(log n)

1. Store minimum (root)
2. Move last element to root
3. Bubble down to restore heap

```
def extract_min(self):
    if len(self.heap) == 0:
        raise IndexError()

    min_val = self.heap[0]
    self.heap[0] =
        self.heap.pop()
    self.heapify_down(0)

    return min_val
```

Peek Min/Max: O(1)

Simply return the root element without removing

Heap Applications

Priority Queue

- Task scheduling
- CPU scheduling
- Event-driven simulation

Heapsort

- Build heap: O(n)
- Extract all: O(n log n)
- In-place sorting

K Largest/Smallest

- Use min-heap of size k for k largest
- O(n log k) time complexity

Graph Algorithms

- Dijkstra's shortest path
- Prim's minimum spanning tree

Heap vs BST

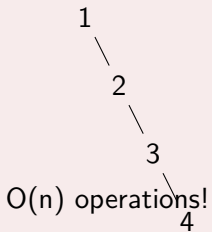
Heap: Quick min/max access, partial order. BST: Full order, any element search

Balanced Trees

Why Balanced Trees?

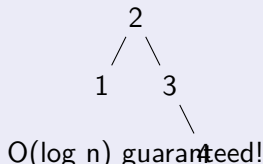
Problem: Unbalanced BST

Regular BST can degenerate to linked list



Solution: Self-Balancing

Maintain balanced structure automatically



Two Main Approaches

AVL Trees: Strictly balanced. **Red-Black Trees**: Loosely balanced

AVL Trees

Definition

AVL tree: Self-balancing BST where height difference between left and right subtrees is at most 1 for every node.

Balance Factor

BF = height(left subtree) - height(right subtree)
Must be in {-1, 0, 1}. If $|BF| > 1$, rebalancing needed.

Rotations

- Right Rotation (LL case)
- Left Rotation (RR case)
- Left-Right Rotation (LR case)
- Right-Left Rotation (RL case)

Properties

- Height $\leq 1.44 \log n$
- Up to 2 rotations per insert
- O(log n) rotations for delete

Red-Black Trees

Definition

Red-Black tree: Self-balancing BST where each node has a color (red or black) with specific properties.

Properties

1. Every node is either red or black
2. Root is always black
3. All leaves (NIL) are black
4. Red nodes cannot have red children
5. Every path from root to leaf has same number of black nodes

Key Differences from AVL

- Looser balance (height $\leq 2 \log n$)
- Faster insert/delete (fewer rotations)

AVL vs Red-Black Comparison

Feature	AVL Tree	Red-Black Tree
Balance	Strictly balanced	Loosely balanced
Height	$\leq 1.44 \log n$	$\leq 2 \log n$
Rotations (insert)	Up to 2	Up to 2
Rotations (delete)	O(log n)	Up to 3
Search	Faster	Slightly slower
Insert/Delete	Slower	Faster
Use case	Search-heavy	Insert/delete-heavy
Memory	Less	More (color bit)

Real-world Usage

AVL: Database indexing. **Red-Black**: Java TreeMap, C++ std::map, Linux kernel

Graphs

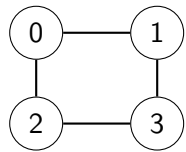
Graph Fundamentals

Definition

A **graph** G = (V, E) consists of vertices (V) and edges (E) connecting them.

Graph Types

- **Undirected**: Edges have no direction (friendships)
- **Directed**: Edges have direction (follows)
- **Weighted**: Edges have values (distances)
- **Unweighted**: No edge values



Undirected Graph

Graph Representations

Adjacency Matrix

2D array: matrix[i][j] = 1 if edge exists

	0	1	2	3
0	0	1	1	0
1	1	0	0	1
2	1	0	0	1
3	0	1	1	0

Pros: O(1) edge check
Cons: O(V^2) space

Adjacency List

Array of lists: list[i] = neighbors

0 $\rightarrow$ [1, 2]
1 $\rightarrow$ [0, 3]
2 $\rightarrow$ [0, 3]
3 $\rightarrow$ [1, 2]

Pros: O(V + E) space
Cons: O(V) edge check

When to Use

Matrix: Dense graphs, fast edge lookup. **List**: Sparse graphs (most real-world)

Graph Representation Comparison

Operation	Adjacency Matrix	Adjacency List
Space	$O(V^2)$	O(V + E)
Add edge	O(1)	O(1)
Remove edge	O(1)	O(V)
Check if edge exists	O(1)	O(V)
Find all neighbors	O(V)	O(degree)
Iterate all edges	$O(V^2)$	O(V + E)

Space Example

Graph with 1000 vertices, 5000 edges:

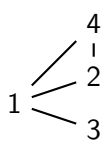
- Matrix: $1000 \times 1000 = 1{,}000{,}000$ entries
- List: $1000 + 2 \times 5000 = 11{,}000$ entries
- List is $\sim$90x more space-efficient!

Graph Traversals

Breadth-First Search (BFS)

Algorithm

1. Start at source vertex
2. Visit all unvisited neighbors
3. Then visit neighbors of neighbors
4. Uses **Queue** (FIFO)



Order: 1→2→3→4

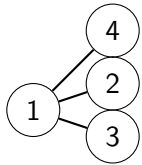
Applications

Shortest path (unweighted), level-order processing, web crawling, social networks

Depth-First Search (DFS)

Algorithm

1. Start at source vertex
2. Visit unvisited neighbor
3. Recursively visit that neighbor's neighbors
4. Backtrack when no unvisited neighbors
5. Uses **Stack** (LIFO)



BFS vs DFS

Feature	BFS	DFS
Data Structure	Queue	Stack/Recursion
Memory	O(V) - wider	O(h) - deeper
Shortest Path	Yes (unweighted)	No
Completeness	Yes	Yes
Time	O(V + E)	O(V + E)

Use BFS for:

- Shortest path (unweighted)
- Level-order processing
- Minimum hops/distance

Use DFS for:

- Cycle detection
- Topological sorting
- Finding all paths

Applications

Tree Applications

File Systems

- Directories and files form tree
- Root directory at top
- Recursive size calculation

Database Indexes

- B-Trees and B+ Trees
- Fast range queries
- O(log n) search/insert/delete

HTML DOM

- Each HTML tag is a node
- Parent-child relationships
- Tree traversal for rendering

Decision Trees (ML)

- Classification and regression
- Each node is a decision
- Leaves are outcomes

More Applications

Abstract syntax trees (compilers), Huffman coding (compression), expression evaluation

Graph Applications

Social Networks

- Vertices: Users
- Edges: Friendships/Follows
- BFS: Degrees of separation
- DFS: Connection exploration

Web Page Ranking

- Vertices: Web pages
- Edges: Hyperlinks
- PageRank algorithm
- Google's foundation

Maps & Navigation

- Vertices: Locations
- Edges: Roads (weighted)
- Dijkstra's: Shortest path
- Applications: GPS, routing

Course Prerequisites

- Vertices: Courses
- Edges: Prerequisites
- Topological sort: Valid order
- Cycle detection: Invalid prereqs

More Applications

When to Use Trees vs Graphs

Use Trees When:

- Hierarchical relationships
- One path between any two nodes
- Clear parent-child structure
- No cycles allowed

Examples:

- File systems
- DOM
- Organizational charts
- Expression trees

Use Graphs When:

- Many-to-many relationships
- Multiple paths between nodes
- Network-like structure
- Cycles may exist

Examples:

- Social networks
- Maps and roads
- Web links
- Dependencies

Summary

Key Takeaways

Trees

- Binary trees, BSTs: O(log n) operations when balanced
- Traversals: Inorder (sorted), Preorder (copy), Postorder (delete), Level-order (BFS)
- Heaps: Priority queue, O(1) peek, O(log n) insert/extract
- Balanced trees (AVL, Red-Black): Guaranteed O(log n)

Graphs

- Adjacency matrix (dense) vs list (sparse)
- BFS: Shortest path, level-order, queue-based
- DFS: Cycle detection, topological sort, stack-based
- Both: O(V + E) time complexity

Applications Everywhere

File systems, databases, social networks, maps, compilers, machine learning, web ranking

Complexity Summary

Data Structure	Search	Insert	Delete
BST (balanced)	O(log n)	O(log n)	O(log n)
BST (worst)	O(n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)
Red-Black Tree	O(log n)	O(log n)	O(log n)
Heap (min/max)	O(n)	O(log n)	O(log n)
Heap (peek)	O(1)	-	-

Graph Traversals

BFS and DFS both have $O(V + E)$ time complexity

Thank You!

Questions?