kintsugi-stack-dsa-java

> Write once, run anywhere.

alt text

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

- kintsugi-stack-dsa-java
    - Table of Contents
    - Introduction
        - What is a Data Structure?
        - What is an Algorithm?
        - Why Learn Data Structures and Algorithms?
    - Data Structures
        - 1. Stack (LIFO - Last In First Out)
            - Stack Operations:
            - Real-life Example:
            - Stack Implementation in Java:
            - Stack Methods:
            - Memory Considerations:
            - Real-World Uses of Stacks:
        - 2. Queue (FIFO - First In First Out)
            - Queue Operations:
            - Real-life Example:
            - Queue Implementation in Java:
            - Queue Methods (from Collection Interface):
            - Real-World Uses of Queues:
        - 3. Priority Queue
            - Key Characteristics:
            - Priority Queue Implementation in Java:
            - Reverse Order (Descending):
            - String Priority Queue:
            - Real-World Applications:
        - 4. Linked List
            - Linked List vs. Array:
            - Singly Linked List:
            - Doubly Linked List:
            - Linked List Node Structure:

        - Linked List Implementation in Java:
        - Linked List Methods:
        - Advantages of Linked Lists:
        - Disadvantages of Linked Lists:
        - Real-World Uses:
    - 5. Dynamic Array (ArrayList)
        - Known As:
        - Static Array vs. Dynamic Array:
        - Dynamic Array Behavior:
        - ArrayList Implementation in Java:
        - Operations Time Complexity:
        - Advantages:
        - Disadvantages:
        - Real-World Uses:
    - 6. Custom Dynamic Array Implementation
        - LinkedList vs. ArrayList Performance Test Results:
- Algorithms
    - Big O Notation
        - Key Principles:
        - Common Big O Complexities (Best to Worst):
        - Performance Grades (with Large Datasets):
        - Visual Comparison:
        - Examples:
        - Space Complexity:
- Search Algorithms
    - Linear Search
        - Characteristics:
        - Implementation:
        - Advantages:
        - Disadvantages:
    - Binary Search
        - Characteristics:
        - How It Works:
        - Implementation:
        - Performance Comparison (1 Million Elements):
    - Interpolation Search
        - Characteristics:
        - Formula to Calculate Probe Position:
        - How It Works:
        - Implementation:
        - Advantages:
        - Disadvantages:
- Sorting Algorithms
    - Bubble Sort
        - Characteristics:
        - How It Works:

      - Implementation:
      - Advantages:
      - Disadvantages:
    - Selection Sort
      - Characteristics:
      - How It Works:
      - Implementation:
      - Advantages:
      - Disadvantages:
    - Quick Sort
      - Characteristics:
      - How It Works:
      - Implementation:
      - Advantages:
      - Disadvantages:
    - Merge Sort
      - Characteristics:
      - How It Works:
      - Implementation:
      - Advantages:
      - Disadvantages:
  - Advanced Data Structures
    - Graph
      - Graph Types:
      - Graph Representations:
      - Adjacency List Implementation:
      - Time Complexities:
    - Depth-First Search (DFS)
      - Characteristics:
      - How It Works:
      - Implementation Using Recursion:
      - Real-World Uses:
    - Breadth-First Search (BFS)
      - Characteristics:
      - How It Works:
      - Implementation:
      - Real-World Uses:
      - DFS vs BFS:
    - Binary Search Tree (BST)
      - BST Property:
      - Node Structure:
      - BST Operations:
      - BST Time Complexities:
      - BST Traversals:
      - Advantages of BST:
      - Disadvantages of BST:

        - Real-World Uses:
    - Conclusion
    - Quick Reference: Time Complexities
        - Data Structures Access Patterns:
        - Sorting Algorithms Comparison:

Introduction

What is a Data Structure?

A **data structure** is a named location that can be used to store and organize data. It's a way to manage and structure information efficiently.

Real-life Example:

- A family tree is a data structure that represents a hierarchy of family relationships
- Arrays are collections of elements stored at contiguous memory locations
- Trees organize data hierarchically

Key Concept: Different data structures store and organize data in different ways, each with their own advantages and disadvantages.

What is an Algorithm?

An **algorithm** is a collection of steps to solve a problem. It's a set of instructions to reach a solution.

Example of an Algorithm:

- Problem: I'm hungry and need a pizza
- Algorithm to bake a pizza:
    1. Heat the oven
    2. Knead the dough
    3. Add the toppings
    4. Bake until done

Example of a Searching Algorithm:

- **Linear Search:** Examine elements of an array one by one to find a value

Why Learn Data Structures and Algorithms?

1. Write code that is both **time and memory efficient**
2. **Commonly asked questions** in coding job interviews
3. Improve problem-solving skills in technical assessments

Data Structures

1. Stack (LIFO - Last In First Out)

A **stack** is a LIFO data structure that stores objects in a vertical tower-like structure, similar to a stack of books or CDs.

Stack Operations:

- **Push:** Add an object to the top of the stack
- **Pop:** Remove an object from the top of the stack
- **Peek:** View the object at the top without removing it
- **Empty:** Check if the stack is empty
- **Search:** Find an object in the stack

Real-life Example:

```
Pushing elements: Minecraft → Skyrim → Doom → Borderlands → Final Fantasy
VII
(Top)                                                      Final Fantasy
VII
                                                           Borderlands
                                                           Doom
                                                           Skyrim
                                                           Minecraft
(Bottom)
```

To access Minecraft at the bottom, you must pop/remove all elements from the top first.

Stack Implementation in Java:

```
Stack<String> stack = new Stack<>();

// Push elements
stack.push("Minecraft");
stack.push("Skyrim");
stack.push("Doom");
stack.push("Borderlands");
stack.push("Final Fantasy VII");

// Check if empty
System.out.println(stack.empty()); // false

// Print all elements
System.out.println(stack); // [Minecraft, Skyrim, Doom, Borderlands, Final
Fantasy VII]

// Pop element (removes and returns)
String game = stack.pop(); // Returns "Final Fantasy VII"
System.out.println(game); // Final Fantasy VII

// Peek at top element (view without removing)
System.out.println(stack.peek()); // Final Fantasy VII
```

```
// Search for element (position starts at 1, not 0)
System.out.println(stack.search("Borderlands")); // 2 (from top)
System.out.println(stack.search("FallOut")); // -1 (not found)
```

Stack Methods:

Method	Description	Returns
`push(E)`	Add element to top	Element
`pop()`	Remove and return top element	Element
`peek()`	View top element	Element
`empty()`	Check if stack is empty	Boolean
`search(Object)`	Find element position	int (1-indexed, -1 if not found)

Memory Considerations:

- Adding a billion elements can cause **OutOfMemoryError** due to Java heap space
- Each element consumes memory

Real-World Uses of Stacks:

1. **Undo/Redo Features** - Text editors use stacks to track changes
2. **Browser History** - Back and forward navigation buttons
3. **Backtracking Algorithms** - Maze navigation, file directory searches
4. **Function Calls** - Call stack in programming languages (frame management)

---

2. Queue (FIFO - First In First Out)

A **queue** is a FIFO data structure designed for holding elements prior to processing. It's like a line of people where the first person served is the first person who arrived.

Queue Operations:

- **Enqueue (Offer):** Add an element to the tail/end of the queue
- **Dequeue (Poll):** Remove an element from the head/front of the queue
- **Peek:** Examine the head element without removing it
- **Empty:** Check if the queue is empty
- **Size:** Get the number of elements
- **Contains:** Check if element exists in the queue

Real-life Example:

```
Queue of people at a ticket counter:
Karen (Head) → Chad → Steve → Harold (Tail)

Karen is helped first → removed
Chad (Head) → Steve → Harold (Tail)

Chad is helped → removed
Steve (Head) → Harold (Tail)

Steve is helped → removed
Harold (Head/Tail) - only one person left
```

Queue Implementation in Java:

Important: Queue is an interface, not a class. Use `LinkedList` to implement:

```
Queue<String> queue = new LinkedList<>();

// Enqueue (add to tail)
queue.offer("Karen");
queue.offer("Chad");
queue.offer("Steve");
queue.offer("Harold");

// Display queue
System.out.println(queue); // [Karen, Chad, Steve, Harold]

// Peek at head
System.out.println(queue.peek()); // Karen

// Dequeue (remove from head)
queue.poll(); // Karen removed
System.out.println(queue); // [Chad, Steve, Harold]

// Check if empty
System.out.println(queue.isEmpty()); // false

// Get size
System.out.println(queue.size()); // 3

// Check if contains
System.out.println(queue.contains("Harold")); // true
```

Queue Methods (from Collection Interface):

Method	Description	Returns
`offer(E)`	Add element to tail	boolean

Method	Description	Returns
`poll()`	Remove and return head	Element (null if empty)
`peek()`	View head element	Element (null if empty)
`isEmpty()`	Check if queue is empty	boolean
`size()`	Get number of elements	int
`contains(Object)`	Check if element exists	boolean

Real-World Uses of Queues:

1. **Keyboard Buffers** - Characters typed are processed in order
2. **Printer Queues** - Print jobs completed in order (page 1, page 2, etc.)
3. **Linked Lists** - Can be implemented using queues
4. **Priority Queues** - Elements served by priority
5. **Breadth-First Search (BFS)** - Graph traversal algorithm

---

3. Priority Queue

A **priority queue** is a queue where elements are served based on their priority rather than simple FIFO order.

Key Characteristics:

- Elements are arranged in order before being pulled
- Higher priority elements are served first
- Default: Elements sorted in ascending order (for numbers)
- Can use custom comparators for different orderings

Priority Queue Implementation in Java:

```java
// Default - ascending order (min-heap)
PriorityQueue<Double> pQueue = new PriorityQueue<>();
pQueue.offer(3.0);
pQueue.offer(2.5);
pQueue.offer(4.0);
pQueue.offer(1.5);
pQueue.offer(2.0);

// Polling returns elements in ascending order
while (!pQueue.isEmpty()) {
    System.out.println(pQueue.poll());
}
// Output: 1.5, 2.0, 2.5, 3.0, 4.0
```

Reverse Order (Descending):

```
PriorityQueue<Double> pQueue = new PriorityQueue<>
(Collections.reverseOrder());
pQueue.offer(3.0);
pQueue.offer(2.5);
pQueue.offer(4.0);

while (!pQueue.isEmpty()) {
    System.out.println(pQueue.poll());
}
// Output: 4.0, 3.0, 2.5
```

String Priority Queue:

```
// Ascending (alphabetical order)
PriorityQueue<String> pQueue = new PriorityQueue<>();
pQueue.offer("B");
pQueue.offer("C");
pQueue.offer("A");

// Descending (reverse alphabetical)
PriorityQueue<String> pQueue = new PriorityQueue<>
(Collections.reverseOrder());
pQueue.offer("B");
pQueue.offer("C");
pQueue.offer("A");
```

Real-World Applications:

- Task scheduling with priority levels
- Dijkstra's shortest path algorithm
- Huffman coding
- Load balancing in systems

4. Linked List

A **linked list** is a data structure consisting of a chain of nodes. Each node contains data and a reference (pointer) to the next node.

Linked List vs. Array:

Feature	Array	Linked List
Memory	Contiguous locations	Non-contiguous locations
Access Time	O(1) - Random access	O(n) - Sequential access
Insertion	O(n) - Requires shifting	O(1) - No shifting needed

Feature	Array	Linked List
Deletion	O(n) - Requires shifting	O(1) - No shifting needed
Memory Usage	Less memory	More memory (stores pointers)
Cache Performance	Better	Worse (jumping around memory)

Singly Linked List:

- Each node has: **data** and **pointer to next node**
- Traversal: Head → Tail (one direction only)

Doubly Linked List:

- Each node has: **data**, **pointer to next node**, and **pointer to previous node**
- Traversal: Head → Tail or Tail → Head (both directions)
- More memory usage but more flexible traversal

Linked List Node Structure:

```
class Node {
    Object data;
    Node next;  // Singly linked
    Node prev;  // Doubly linked
}
```

Linked List Implementation in Java:

```
LinkedList<String> linkedList = new LinkedList<>();

// Add elements
linkedList.push("A");
linkedList.push("B");
linkedList.push("C");
linkedList.push("D");
linkedList.push("F");

// Display
System.out.println(linkedList); // [F, D, C, B, A]

// Stack operations
linkedList.pop(); // Removes F
System.out.println(linkedList); // [D, C, B, A]

// Queue operations
linkedList.offer("A"); // Adds to tail
linkedList.poll(); // Removes from head
```

```
// Insert at specific position
linkedList.add(3, "E"); // Insert E at index 3

// Remove element
linkedList.remove("E");

// Peek operations
System.out.println(linkedList.peekFirst()); // First element
System.out.println(linkedList.peekLast()); // Last element

// Add at specific positions
linkedList.addFirst("0");
linkedList.addLast("G");

// Remove from specific positions
String first = linkedList.removeFirst();
String last = linkedList.removeLast();

// Find index
System.out.println(linkedList.indexOf("F")); // Index of F
```

Linked List Methods:

Method	Description
`peekFirst()` / `peekLast()`	View first/last element
`addFirst(E)` / `addLast(E)`	Add to head/tail
`removeFirst()` / `removeLast()`	Remove from head/tail
`indexOf(Object)`	Find position of element
`add(int, E)`	Insert at specific index
`remove(Object)`	Remove element

Advantages of Linked Lists:

1. **Dynamic data structure** - Allocates memory during runtime
2. **Easy insertion/deletion** - O(1) constant time (no shifting)
3. **No memory waste** - Uses only what's needed

Disadvantages of Linked Lists:

1. **Greater memory usage** - Must store additional pointers
2. **No random access** - Must traverse from head/tail
3. **Slow searching** - O(n) linear time
4. **Poor cache performance** - Nodes scattered in memory

Real-World Uses:

1. **GPS Navigation** - Each waypoint is a node; easy to insert/delete waypoints
2. **Music Playlists** - Songs don't need to be adjacent; easy to reorder
3. **Undo/Redo** - History of actions stored as linked nodes
4. **Browser History** - Navigation through pages

---

5. Dynamic Array (ArrayList)

A **dynamic array** is an array with resizable capacity. When it reaches capacity, it automatically expands.

Known As:

- **ArrayList** in Java
- **Vector** in C++
- **Array** in JavaScript
- **List** in Python

Static Array vs. Dynamic Array:

Static Array:

- Fixed capacity determined at compile time
- Cannot expand or shrink
- If full, cannot add more elements

Dynamic Array:

- Capacity increases automatically when needed
- Usually increases capacity by 1.5x to 2x
- Better memory management

Dynamic Array Behavior:

```
Initial state:
Capacity: 10, Size: 5
[A, B, C, D, E, null, null, null, null, null]

Add element (still room):
Capacity: 10, Size: 6
[A, B, C, D, E, F, null, null, null, null]

Add element at capacity:
Capacity: 20 (doubled), Size: 7
[A, B, C, D, E, F, G, null, null, null, null, null, null, null, null, null,
null, null, null, null]
```

ArrayList Implementation in Java:

```
ArrayList<String> arrayList = new ArrayList<>();

// Default capacity is 10
int size = arrayList.size(); // 0
int capacity = 10; // Default

// Add elements
arrayList.add("A");
arrayList.add("B");
arrayList.add("C");

// Insert at specific index
arrayList.add(0, "X"); // Insert X at index 0

// Remove element
arrayList.remove("A"); // Remove by value
arrayList.remove(0); // Remove by index

// Search
int index = arrayList.indexOf("B");

// Display
System.out.println(arrayList);
```

Operations Time Complexity:

Operation	Time Complexity
Random access (by index)	O(1)
Search	O(n)
Insert at end	O(1) - amortized
Insert at beginning	O(n) - requires shifting
Insert in middle	O(n) - requires shifting
Delete at end	O(1)
Delete at beginning	O(n) - requires shifting
Delete in middle	O(n) - requires shifting

Advantages:

1. **Random access** - O(1) constant time
2. **Good cache utilization** - Contiguous memory locations
3. **Easy insertion/deletion at end** - No shifting required
4. **Space-efficient** - Less memory overhead than linked lists

Disadvantages:

1. **Wastes memory** - Capacity exceeds actual size
2. **Shifting is expensive** - Inserting/deleting near beginning is slow
3. **Expansion is costly** - Must copy all elements to new array

Real-World Uses:

- Any collection where you need fast random access
- Dynamic arrays for temporary storage
- Implementing other data structures
- Caching

---

6. Custom Dynamic Array Implementation

Here's how to build a dynamic array from scratch:

```
public class DynamicArray {
    private int size;
    private int capacity;
    private Object[] array;

    // Constructor with default capacity
    public DynamicArray() {
        this.capacity = 10;
        this.array = new Object[capacity];
        this.size = 0;
    }

    // Constructor with custom capacity
    public DynamicArray(int capacity) {
        this.capacity = capacity;
        this.array = new Object[capacity];
        this.size = 0;
    }

    // Add element
    public void add(Object data) {
        if (size >= capacity) {
            grow();
        }
        array[size] = data;
        size++;
    }

    // Insert at specific index
    public void insert(int index, Object data) {
        if (size >= capacity) {
            grow();
        }
        // Shift elements to the right
        for (int i = size - 1; i >= index; i--) {
            array[i + 1] = array[i];
        }
        array[index] = data;
        size++;
    }

    // Delete element
    public void delete(Object data) {
        for (int i = 0; i < size; i++) {
            if (array[i].equals(data)) {
                // Shift elements to the left
                for (int j = 0; j < size - i - 1; j++) {
                    array[i + j] = array[i + j + 1];
                }
                array[size - 1] = null;
                size--;

                // Shrink if needed
                if (size <= capacity / 3) {
                    shrink();
                }
                break;
            }
        }
    }

    // Search for element
    public int search(Object data) {
        for (int i = 0; i < size; i++) {
            if (array[i].equals(data)) {
                return i;
            }
        }
        return -1; // Not found
    }

    // Grow capacity
    private void grow() {
        int newCapacity = capacity * 2;
        Object[] newArray = new Object[newCapacity];

        // Copy elements
        for (int i = 0; i < size; i++) {
            newArray[i] = array[i];
        }

        capacity = newCapacity;
        array = newArray;
    }

    // Shrink capacity
    private void shrink() {
        int newCapacity = capacity / 2;
        Object[] newArray = new Object[newCapacity];
```

```java
        // Copy elements
        for (int i = 0; i < size; i++) {
            newArray[i] = array[i];
        }

        capacity = newCapacity;
        array = newArray;
    }

    // Check if empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Display array
    @Override
    public String toString() {
        String string = "";
        for (int i = 0; i < size; i++) {
            string += array[i];
            if (i < size - 1) {
                string += ", ";
            }
        }
        return "[" + string + "]";
    }
}
```

LinkedList vs. ArrayList Performance Test Results:

Getting element at index 0 (first):

- LinkedList: ~11,800 ns
- ArrayList: ~6,700 ns
- Winner: ArrayList (faster)

Getting element at middle (500,000th out of 1 million):

- LinkedList: ~7.5 million ns
- ArrayList: ~6,900 ns
- Winner: ArrayList (much faster)

Getting last element (999,999th):

- LinkedList: ~63,000 ns (doubly linked can approach from end)
- ArrayList: ~17,000 ns
- Winner: ArrayList (faster)

Removing first element:

- LinkedList: ~17,000 ns

- ArrayList: ~2.2 million ns
- Winner: LinkedList (no shifting needed)

Removing middle element (500,000th):

- LinkedList: ~7 million ns
- ArrayList: ~1.6 million ns
- Winner: ArrayList (fewer elements to shift)

Conclusion: ArrayList is generally faster for most operations, but LinkedList is better when frequently inserting/deleting at the beginning.

Algorithms

Big O Notation

Definition: A notation that describes the performance of an algorithm as the amount of data increases. Answers the question: "How does code slow as data grows?"

Key Principles:

1. **Machine independent** - Focuses on number of steps, not machine speed
2. **Ignore smaller operations** - n + 1 is simplified to n
3. **Focus on worst case** - How bad can it get?

Common Big O Complexities (Best to Worst):

Complexity	Name	Example
O(1)	Constant	Accessing array element by index
O(log n)	Logarithmic	Binary search
O(n)	Linear	Loop through array
O(n log n)	Quasi-linear	Quick sort, merge sort
$O(n^2)$	Quadratic	Bubble sort, nested loops
$O(2^n)$	Exponential	Recursive fibonacci
O(n!)	Factorial	Permutations

Performance Grades (with Large Datasets):

- O(1) - A+
- O(log n) - B
- O(n) - C
- O(n log n) - D
- $O(n^2)$ - F
- $O(2^n)$ - F (expelled)

Visual Comparison:

```
Time
  |
  |                        O(n!)
  |                    O(2^n)
  |                O(n^2)
  |            O(n log n)
  |        O(n)
  |    O(log n)
  | O(1)
  |_____________________________ Data (n)
```

Examples:

O(1) - Constant Time:

```
int value = array[0]; // Always 1 step
```

O(n) - Linear Time:

```
for (int i = 0; i < n; i++) {
    sum += i; // n steps
}
```

O(n²) - Quadratic Time:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // n * n steps
    }
}
```

O(log n) - Logarithmic Time:

```
// Binary search cuts data in half each time
```

Space Complexity:

Space complexity describes memory usage as data grows.

Structure	Space Complexity
Stack	O(n)
Queue	O(n)
Linked List	O(n) + pointer overhead
Array	O(n)
BST	O(n)
Graph (Adjacency List)	O(V + E)
Graph (Adjacency Matrix)	$O(V^2)$

Search Algorithms

Linear Search

Definition: An algorithm that checks each element sequentially until the target is found.

Characteristics:

- Time Complexity: **O(n)**
- Works with **unsorted data**
- Fast for **small to medium datasets**
- Good for **data structures without random access** (like linked lists)

Implementation:

```java
public static int linearSearch(int[] array, int value) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == value) {
            return i; // Element found
        }
    }
    return -1; // Element not found
}

// Usage:
int[] array = {5, 2, 8, 1, 9, 3, 7};
int index = linearSearch(array, 1);

if (index != -1) {
    System.out.println("Element found at index: " + index);
} else {
    System.out.println("Element not found");
}
```

Advantages:

- Works with unsorted data
- Fast for small datasets
- Works with linked lists
- Simple to implement

Disadvantages:

- Slow for large datasets
- O(n) time complexity

Binary Search

Definition: An algorithm that eliminates half of the remaining elements with each comparison. Only works on **sorted data**.

Characteristics:

- Time Complexity: **O(log n)**
- **MUST be sorted**
- **Much faster** for large datasets
- Uses **divide and conquer** approach

How It Works:

```
Array: [A, B, C, D, E, F, G, H, I, J, K]
Search for: H

Step 1: Check middle (F)
        H > F, so search right half

Step 2: Check middle of right (I)
        H < I, so search left half

Step 3: Check middle of remaining (H)
        H == H, FOUND!
```

Implementation:

```java
public static int binarySearch(int[] array, int target) {
    int low = 0;
    int high = array.length - 1;

    while (low <= high) {
        int middle = low + (high - low) / 2;
        int value = array[middle];

        if (value == target) {
            return middle; // Found
        } else if (value < target) {
            low = middle + 1; // Search right
        } else {
            high = middle - 1; // Search left
        }
    }
    return -1; // Not found
}

// Usage:
int[] array = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
int index = binarySearch(array, 7);

if (index != -1) {
    System.out.println("Element found at index: " + index);
} else {
    System.out.println("Element not found");
}
```

Performance Comparison (1 Million Elements):

Linear Search for element 777,777:

- Steps: 777,777 (searching sequentially)

Binary Search for element 777,777:

- Steps: ~20 (cutting in half each time)

Binary search is 38,888x faster!

Interpolation Search

Definition: An improvement over binary search. Makes intelligent guesses about where an element might be based on the distribution of data.

Characteristics:

- Average Time Complexity: **O(log log n)** (better than binary search for uniform distribution)
- Worst Case: **O(n)**
- Best for **uniformly distributed data**
- Poor for **exponentially increasing data**

Formula to Calculate Probe Position:

```
probe = low + (value - array[low]) / (array[high] - array[low]) * (high - low)
```

How It Works:

```
If searching for 256 in [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

Guess 1: Calculate where 256 likely is → find it or narrow search
Guess 2: Recalculate for remaining range
... repeat until found
```

Implementation:

```java
public static int interpolationSearch(int[] array, int value) {
    int low = 0;
    int high = array.length - 1;

    while (value >= array[low] && value <= array[high] && low <= high) {
        int probe = low + (value - array[low]) / (array[high] - array[low]) * (high - low);

        if (array[probe] == value) {
            return probe; // Found
        } else if (array[probe] < value) {
            low = probe + 1; // Search right
        } else {
            high = probe - 1; // Search left
        }
    }
    return -1; // Not found
}

// Usage:
int[] array = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};
int index = interpolationSearch(array, 256);
```

Advantages:

- Better than binary search for uniform data
- O(log log n) average case

Disadvantages:

- Terrible for exponentially increasing data
- More complex than binary search
- Requires sorted data

Sorting Algorithms

Bubble Sort

Definition: A sorting algorithm that repeatedly compares adjacent elements and swaps them if they're in the wrong order.

Characteristics:

- Time Complexity: **$O(n^2)$** - Quadratic
- Space Complexity: **O(1)** - In-place
- **Simple but inefficient** for large datasets
- **Stable sort**

How It Works:

```
Pass 1: [5, 2, 8, 1] → [2, 5, 1, 8]
        Compare 5 & 2, swap → [2, 5, 8, 1]
        Compare 5 & 8, no swap
        Compare 8 & 1, swap → [2, 5, 1, 8]

Pass 2: [2, 5, 1, 8] → [2, 1, 5, 8]
        Compare 2 & 5, no swap
        Compare 5 & 1, swap → [2, 1, 5, 8]

Pass 3: [2, 1, 5, 8] → [1, 2, 5, 8]
        Compare 2 & 1, swap → [1, 2, 5, 8]

Result: [1, 2, 5, 8] (sorted)
```

Implementation:

```
public static void bubbleSort(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        for (int j = 0; j < array.length - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                // Swap
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

// For descending order, change > to <
public static void bubbleSortDescending(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        for (int j = 0; j < array.length - i - 1; j++) {
            if (array[j] < array[j + 1]) { // Changed operator
                // Swap
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

// Usage:
int[] array = {5, 2, 8, 1, 9};
bubbleSort(array);
System.out.println(Arrays.toString(array)); // [1, 2, 5, 8, 9]
```

Advantages:

- Very simple to understand
- No extra space needed (in-place)
- Stable sort

Disadvantages:

- **Very inefficient** - $O(n^2)$ complexity
- Not suitable for large datasets
- Many unnecessary comparisons

Selection Sort

Definition: A sorting algorithm that repeatedly finds the minimum element and places it at the beginning.

Characteristics:

- Time Complexity: **$O(n^2)$** - Quadratic
- Space Complexity: **O(1)** - In-place
- Simple but still inefficient
- **Not stable**

How It Works:

```
Array: [9, 1, 8, 2, 7, 3, 6, 4, 5]

Pass 1: Find minimum (1) → swap with position 0
        [1, 9, 8, 2, 7, 3, 6, 4, 5]

Pass 2: Find minimum in remaining (2) → swap with position 1
        [1, 2, 8, 9, 7, 3, 6, 4, 5]

Pass 3: Find minimum in remaining (3) → swap with position 2
        [1, 2, 3, 9, 7, 8, 6, 4, 5]
```

```
... Continue until fully sorted
```

Implementation:

```java
public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int minIndex = i;

        // Find minimum in remaining array
        for (int j = i + 1; j < array.length; j++) {
            if (array[j] < array[minIndex]) {
                minIndex = j;
            }
        }

        // Swap if needed
        if (minIndex != i) {
            int temp = array[i];
            array[i] = array[minIndex];
            array[minIndex] = temp;
        }
    }
}

// Usage:
int[] array = {9, 1, 8, 2, 7, 3, 6, 4, 5};
selectionSort(array);
System.out.println(Arrays.toString(array)); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Advantages:

- Simple to understand and implement
- No extra space needed (in-place)
- Minimal number of swaps

Disadvantages:

- **O(n^2) complexity** - inefficient for large datasets
- Not stable
- Many comparisons needed

Quick Sort

Definition: A divide-and-conquer sorting algorithm that partitions the array and recursively sorts sub-arrays.

Characteristics:

- Average Time Complexity: **O(n log n)**
- Worst Case: **O(n^2)** (if pivot is always smallest/largest)
- Space Complexity: **O(log n)** (recursion stack)
- **Very efficient** in practice
- **Not stable**

How It Works:

1. Choose a **pivot** element
2. **Partition** array into elements less than pivot and greater than pivot
3. **Recursively** sort left and right partitions

Implementation:

```
public static void quickSort(int[] array) {
    if (array.length == 0) return;
    quickSort(array, 0, array.length - 1);
}

private static void quickSort(int[] array, int low, int high) {
    if (low < high) {
        int pi = partition(array, low, high);

        quickSort(array, low, pi - 1);   // Sort left partition
        quickSort(array, pi + 1, high);  // Sort right partition
    }
}

private static int partition(int[] array, int low, int high) {
    int pivot = array[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (array[j] < pivot) {
            i++;
            // Swap
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    // Swap pivot to correct position
    int temp = array[i + 1];
    array[i + 1] = array[high];
    array[high] = temp;

    return i + 1;
}

// Usage:
```

```java
    int[] array = {5, 2, 8, 1, 9, 3, 7};
    quickSort(array);
    System.out.println(Arrays.toString(array));
```

Advantages:

- **Very efficient** - O(n log n) average
- In-place sorting (minimal extra space)
- Fast in practice
- Used in many standard libraries

Disadvantages:

- Worst case $O(n^2)$ complexity
- Not stable
- Recursive (uses stack space)

Merge Sort

Definition: A divide-and-conquer sorting algorithm that divides array in half and recursively merges sorted sub-arrays.

Characteristics:

- Time Complexity: **O(n log n)** - Always
- Space Complexity: **O(n)** - Requires extra space
- **Guaranteed** good performance
- **Stable sort**

How It Works:

```
[5, 2, 8, 1, 9, 3, 7, 6]

Divide:
[5, 2, 8, 1] | [9, 3, 7, 6]
[5, 2] [8, 1] | [9, 3] [7, 6]
[5] [2] [8] [1] | [9] [3] [7] [6]

Merge:
[2, 5] [1, 8] | [3, 9] [6, 7]
[1, 2, 5, 8] | [3, 6, 7, 9]
[1, 2, 3, 5, 6, 7, 8, 9]
```

Implementation:

```java
public static void mergeSort(int[] array) {
    if (array.length < 2) return;
    mergeSort(array, 0, array.length - 1);
}

private static void mergeSort(int[] array, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(array, left, mid);      // Sort left half
        mergeSort(array, mid + 1, right); // Sort right half
        merge(array, left, mid, right);   // Merge
    }
}

private static void merge(int[] array, int left, int mid, int right) {
    int leftSize = mid - left + 1;
    int rightSize = right - mid;

    int[] leftArray = new int[leftSize];
    int[] rightArray = new int[rightSize];

    // Copy data to temp arrays
    for (int i = 0; i < leftSize; i++) {
        leftArray[i] = array[left + i];
    }
    for (int i = 0; i < rightSize; i++) {
        rightArray[i] = array[mid + 1 + i];
    }

    // Merge the arrays
    int i = 0, j = 0, k = left;

    while (i < leftSize && j < rightSize) {
        if (leftArray[i] <= rightArray[j]) {
            array[k] = leftArray[i];
            i++;
        } else {
            array[k] = rightArray[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements
    while (i < leftSize) {
        array[k] = leftArray[i];
        i++;
        k++;
    }
    while (j < rightSize) {
        array[k] = rightArray[j];
        j++;
        k++;
    }
}

// Usage:
int[] array = {5, 2, 8, 1, 9, 3, 7};
mergeSort(array);
System.out.println(Arrays.toString(array));
```

Advantages:

- **Guaranteed O(n log n)** - Always consistent
- **Stable** - Preserves order of equal elements
- Predictable performance

Disadvantages:

- **Requires O(n) extra space** - Not in-place
- Overhead of creating temporary arrays
- Slower than Quick Sort for random data

Advanced Data Structures

Graph

A **graph** is a collection of nodes (vertices) connected by edges (links).

Graph Types:

1. **Directed Graph** - Edges have direction (one-way)
2. **Undirected Graph** - Edges have no direction (two-way)
3. **Weighted Graph** - Edges have weights/values
4. **Unweighted Graph** - All edges equal

Graph Representations:

Adjacency Matrix:

- 2D array where element [i][j] indicates connection
- Fast lookup: O(1)
- Space: O(V^2) where V = number of vertices
- Best for: Dense graphs

```
// 1 = connected, 0 = not connected
int[][] matrix = {
    {0, 1, 1, 0},  // A connects to B, C
    {1, 0, 1, 1},  // B connects to A, C, D
    {1, 1, 0, 1},  // C connects to A, B, D
    {0, 1, 1, 0}   // D connects to B, C
};
```

Adjacency List:

- Array of linked lists or other collections
- Space efficient: O(V + E)
- Lookup: O(degree of vertex)
- Best for: Sparse graphs

```
ArrayList<LinkedList<Node>> adjacencyList = new ArrayList<>();

// Each index is a vertex
// LinkedList contains adjacent neighbors
adjacencyList.get(0).add(new Node(1)); // A connects to B
adjacencyList.get(0).add(new Node(2)); // A connects to C
```

Adjacency List Implementation:

```
class Node {
    int data;

    public Node(int data) {
        this.data = data;
    }
}

class Graph {
    ArrayList<LinkedList<Node>> adjacencyList;

    public Graph(int vertices) {
        adjacencyList = new ArrayList<>();
        for (int i = 0; i < vertices; i++) {
            adjacencyList.add(new LinkedList<>());
        }
    }

    public void addEdge(int source, int destination) {
        LinkedList<Node> currentList = adjacencyList.get(source);
        Node destinationNode = new Node(destination);
        currentList.add(destinationNode);
    }

    public boolean checkEdge(int source, int destination) {
        LinkedList<Node> currentList = adjacencyList.get(source);
        Node destinationNode = new Node(destination);

        for (Node node : currentList) {
            if (node.data == destinationNode.data) {
                return true;
            }
        }
        return false;
    }

    public void print() {
        for (LinkedList<Node> currentList : adjacencyList) {
            for (Node node : currentList) {
                System.out.print(node.data + " → ");
            }
            System.out.println();
        }
    }
}

// Usage:
Graph graph = new Graph(5);
graph.addEdge(0, 1); // A to B
graph.addEdge(0, 2); // A to C
graph.addEdge(1, 3); // B to D
graph.addEdge(2, 3); // C to D
graph.print();
```

Time Complexities:

Operation	Adjacency Matrix	Adjacency List
Check edge	O(1)	O(degree)
Add edge	O(1)	O(1)
Remove edge	O(1)	O(degree)
Storage	$O(V^2)$	O(V + E)

Depth-First Search (DFS)

Definition: A graph traversal algorithm that explores as far as possible along each branch before backtracking.

Characteristics:

- Time Complexity: **O(V + E)** (vertices + edges)
- Uses **Stack** or **Recursion**
- Explores one branch completely before moving to next
- Good for: Backtracking, topological sort, detecting cycles

How It Works:

```
Graph: A→B, A→D, B→C, B→E, ...

Start at A (mark visited)
├─ Go to B (first unvisited neighbor, mark visited)
│  ├─ Go to C (mark visited, dead end)
│  │  └─ Backtrack to B
│  └─ Go to E (mark visited, dead end)
│     └─ Backtrack to B
│        └─ Backtrack to A
└─ Go to D (mark visited)
   └─ Backtrack to A (done)

Visit order: A, B, C, E, D
```

Implementation Using Recursion:

```java
class Graph {
    ArrayList<LinkedList<Node>> adjacencyList;

    // ... constructor and other methods ...

    public void depthFirstSearch(int index) {
        boolean[] visited = new boolean[adjacencyList.size()];
        dfsHelper(index, visited);
    }

    private void dfsHelper(int source, boolean[] visited) {
        if (visited[source]) {
            return; // Already visited
        }

        visited[source] = true;
        System.out.print(source + " "); // Process node

        // Visit all adjacent neighbors
        for (Node node : adjacencyList.get(source)) {
            if (!visited[node.data]) {
                dfsHelper(node.data, visited);
            }
        }
    }
}

// Usage:
Graph graph = new Graph(5);
graph.addEdge(0, 1); // 0 → 1
graph.addEdge(0, 2); // 0 → 2
graph.addEdge(1, 3); // 1 → 3
graph.addEdge(2, 3); // 2 → 3
graph.addEdge(3, 4); // 3 → 4
graph.depthFirstSearch(0); // Output: 0 1 3 4 2 (or similar)
```

Real-World Uses:

1. **Maze solving** - Backtracking
2. **Topological sorting** - Dependency resolution
3. **Detecting cycles** - In graphs or dependencies
4. **Path finding** - Finding any path between two nodes
5. **Puzzle solving** - Trying all possibilities

Breadth-First Search (BFS)

Definition: A graph traversal algorithm that explores all neighbors at the current depth before moving to deeper neighbors.

Characteristics:

- Time Complexity: **O(V + E)**
- Uses **Queue**
- Explores level by level
- Finds **shortest path** in unweighted graphs
- Good for: Finding shortest path, level-order traversal

How It Works:

```
Graph: A connects to B,C; B connects to D,E; C connects to F; ...

Level 0: [A]
Level 1: [B, C] (neighbors of A)
Level 2: [D, E, F] (neighbors of B and C)
Level 3: [G, H] (neighbors of D, E, F)

Visit order: A, B, C, D, E, F, G, H
```

Implementation:

```
class Graph {
    ArrayList<LinkedList<Node>> adjacencyList;

    // ... constructor and other methods ...

    public void breadthFirstSearch(int index) {
        Queue<Integer> queue = new LinkedList<>();
        boolean[] visited = new boolean[adjacencyList.size()];
        queue.offer(index);
        visited[index] = true;

        while (!queue.isEmpty()) {
            int source = queue.poll();
            System.out.print(source + " "); // Process node

            // Add all unvisited neighbors to queue
            for (Node node : adjacencyList.get(source)) {
                if (!visited[node.data]) {
                    queue.offer(node.data);
                    visited[node.data] = true;
                }
            }
        }
    }
}

// Usage:
Graph graph = new Graph(6);
graph.addEdge(0, 1); // 0 → 1
graph.addEdge(0, 2); // 0 → 2
graph.addEdge(1, 3); // 1 → 3
graph.addEdge(2, 3); // 2 → 3
graph.addEdge(3, 4); // 3 → 4
graph.addEdge(4, 5); // 4 → 5

graph.breadthFirstSearch(0); // Output: 0 1 2 3 4 5
```

Real-World Uses:

1. **Shortest path** - Navigation systems, GPS
2. **Social networking** - Finding degrees of separation
3. **Web crawler** - Crawling web pages level by level
4. **Peer-to-peer networks** - Finding nearby peers
5. **Game AI** - Finding closest enemies or items

DFS vs BFS:

Aspect	DFS	BFS
Data Structure	Stack/Recursion	Queue
Use Case	Backtracking, cycle detection	Shortest path, level-order
Memory	O(h) where h = height	O(w) where w = width
Best For	Deep graphs	Wide graphs

Binary Search Tree (BST)

A **Binary Search Tree** is a binary tree where each node has at most two children, and values follow the BST property.

BST Property:

- **Left child < Parent < Right child**
- All left descendants < parent
- All right descendants > parent

Node Structure:

```
class Node {
    int data;
    Node left;
    Node right;

    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

BST Operations:

1. Insert:

```
public void insert(int data) {
    root = insertHelper(root, new Node(data));
}

private Node insertHelper(Node root, Node node) {
    if (root == null) {
        return node; // First node becomes root
    }

    if (node.data < root.data) {
        root.left = insertHelper(root.left, node); // Go left
    } else if (node.data > root.data) {
        root.right = insertHelper(root.right, node); // Go right
    }

    return root;
}
```

2. Search:

```
public boolean search(int data) {
    return searchHelper(root, data);
}

private boolean searchHelper(Node root, int data) {
    if (root == null) {
        return false; // Not found
    }

    if (root.data == data) {
        return true; // Found
    } else if (data < root.data) {
        return searchHelper(root.left, data); // Search left
    } else {
        return searchHelper(root.right, data); // Search right
    }
}
```

3. Display (In-Order Traversal):

```
public void display() {
    displayHelper(root);
}

private void displayHelper(Node root) {
    if (root == null) {
        return;
    }

    displayHelper(root.left);       // Left
    System.out.print(root.data + " "); // Root
    displayHelper(root.right);      // Right
    // Output: Sorted ascending order
}
```

4. Remove:

```
public void remove(int data) {
    if (search(data)) {
        root = removeHelper(root, data);
    } else {
        System.out.println(data + " could not be found");
    }
}

private Node removeHelper(Node root, int data) {
    if (root == null) {
        return root;
    }
```

```
        if (data < root.data) {
            root.left = removeHelper(root.left, data); // Go left
        } else if (data > root.data) {
            root.right = removeHelper(root.right, data); // Go right
        } else {
            // Node found

            // Case 1: Leaf node (no children)
            if (root.left == null && root.right == null) {
                return null; // Remove node
            }

            // Case 2: Node has right child
            if (root.right != null) {
                root.data = successor(root);
                root.right = removeHelper(root.right, root.data);
            }
            // Case 3: Node has only left child
            else {
                root.data = predecessor(root);
                root.left = removeHelper(root.left, root.data);
            }
        }

        return root;
    }

    private int successor(Node root) {
        root = root.right;
        while (root.left != null) {
            root = root.left;
        }
        return root.data; // Smallest in right subtree
    }

    private int predecessor(Node root) {
        root = root.left;
        while (root.right != null) {
            root = root.right;
        }
        return root.data; // Largest in left subtree
    }
```

BST Time Complexities:

Operation	Average	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)

Operation	Average	Worst Case
Delete	O(log n)	O(n)
Traversal	O(n)	O(n)

Note: Worst case occurs when tree becomes skewed (like a linked list)

BST Traversals:

In-Order (Left, Root, Right):

- Output: Sorted ascending order
- Used for: Sorted retrieval

Pre-Order (Root, Left, Right):

- Used for: Copying tree, creating expression trees

Post-Order (Left, Right, Root):

- Used for: Deleting tree, postfix expressions

Advantages of BST:

1. **Sorted data** - In-order traversal gives sorted output
2. **Fast search** - O(log n) average case
3. **Dynamic** - Can add/remove elements
4. **Efficient range queries** - Find all values between x and y

Disadvantages of BST:

1. **Skewed tree** - Can degrade to O(n) if unbalanced
2. **No random access** - Must traverse
3. **Extra space** - For pointers

Real-World Uses:

1. **File systems** - Directory structures
2. **Database indexing** - B-trees (variant of BST)
3. **Expression evaluation** - Parsing mathematical expressions
4. **Range searching** - Interval queries

Conclusion

This comprehensive guide covers fundamental data structures and algorithms:

Data Structures Covered:

- Stack (LIFO)
- Queue (FIFO)

- Priority Queue
- Linked List (singly and doubly)
- Dynamic Array/ArrayList
- Graph (Adjacency List and Matrix)
- Binary Search Tree

Algorithms Covered:

- Linear Search - O(n)
- Binary Search - O(log n)
- Interpolation Search - O(log log n)
- Bubble Sort - O(n²)
- Selection Sort - O(n²)
- Quick Sort - O(n log n)
- Merge Sort - O(n log n)
- Depth-First Search - O(V + E)
- Breadth-First Search - O(V + E)

Key Takeaways:

1. Choose the right data structure for your use case
2. Understand time and space complexity
3. Consider whether data needs to be sorted
4. Know when to use each algorithm
5. Practice implementing from scratch to understand deeply

Remember: The best data structure/algorithm depends on your specific problem requirements!

Quick Reference: Time Complexities

Data Structures Access Patterns:

Operation	Array	Linked List	BST	Hash Table
Access	O(1)	O(n)	O(log n)	O(1) avg
Search	O(n)	O(n)	O(log n)	O(1) avg
Insert	O(n)	O(1)	O(log n)	O(1) avg
Delete	O(n)	O(1)	O(log n)	O(1) avg

Sorting Algorithms Comparison:

Algorithm	Best	Average	Worst	Space	Stable
Bubble	O(n)	O(n²)	O(n²)	O(1)	Yes
Selection	O(n²)	O(n²)	O(n²)	O(1)	No
Quick	O(n log n)	O(n log n)	O(n²)	O(log n)	No

Algorithm	Best	Average	Worst	Space	Stable
Merge	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes

End-of-File

The KintsugiStack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with 💚 Kintsugi-Programmer