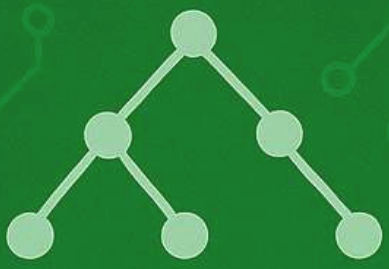
COMP202 COURSE NOTES



DATA STRUCTURES & ALGORITHMS

AYKHAN AHMADZADA

KOÇ UNIVERSITY · COMP 202

Data Structures & Algorithms — Overview

What This Course Covers

- Algorithms, complexity, recursion
- Linked lists, stacks, queues
- Trees, tries, skip lists
- Heaps & hash tables
- Graphs & graph algorithms
- Sorting algorithms

Prereqs: Discrete Math, Java; complete COMP 106 & COMP 132 beforehand.

Goals & Outcomes

- Choose appropriate data structures & algorithms
- Employ & implement fundamental structures
- Use standard implementations correctly
- Analyze time/space complexity

Grading Snapshot

- Assignments: **40%**
- Midterm: **30%**
- Final: **30%**

≥ 40/100 on each exam; catalog-like (no curve); objections within 3 days.

Textbook

Goodrich–Tamassia–Goldwasser, *Data Structures and Algorithms in Java* (Wiley).

POLICIES · EXPECTATIONS

Course Policies & How to Succeed

Assignments

- No late work without prior permission & valid excuse
- Handwritten OK if legible; target best asymptotic complexity
- Attach the mandatory declaration (blue pen)

Exam Rules

- Individual, closed book/notes
- **No devices** (phones, watches, headphones, calculators, etc.)
- Return all exam sheets; violations → report & disciplinary action

Academic Integrity

- Your work must be your own (no outsourcing or reuse of old solutions)
- KU Student Code of Conduct applies; violations may lead to F and discipline

How to Succeed

- Read several sources; implement a lot
- Do all assignments & extra exercises yourself
- Ask questions before topics change; work regularly
- Track announcements; don't cheat

Make-Up & Device Policy

- Approved midterm health report → final counts as midterm
- Approved final health report → apply for make-up
- No remedial exam (course offered both semesters)
- Device use in class is strictly forbidden

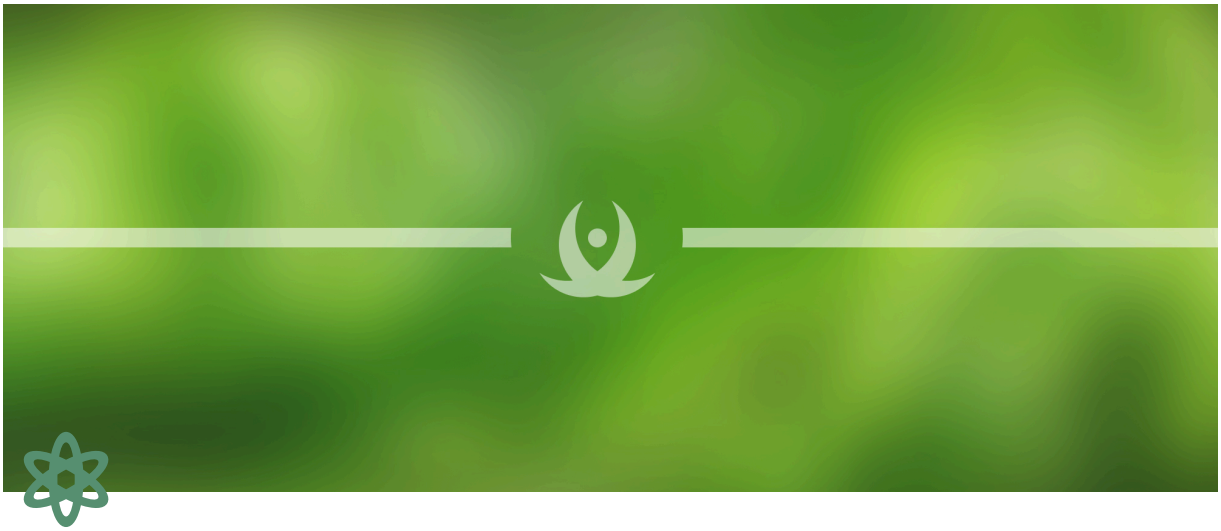
Memorial

```
/** In memory of those who turned Θ(n²) confusion into O(n log n) clarity. */
```

This work is a personal academic compilation created for educational purposes as part of the **COMP202 (DATA STRUCTURES & ALGORITHMS)** course at **Koç University**.

Compiled in **Istanbul, Turkey**.

COMP202



1. Analysis of Algorithms

Introduction

This note focuses on the **analysis of algorithms** (Lecture 1 Slide), a fundamental topic in data structures and algorithms. The primary goal is to understand how to estimate the efficiency of an algorithm as the size of its input grows. We will explore both **experimental** and **theoretical** techniques, introduce asymptotic notation (including **Big-Oh**, **Big-Omega**, and **Big-Theta**), and discuss why these notations are crucial for comparing and selecting algorithms.

Running Time and Experimental Analysis

Many algorithms transform an **input** into an **output** by performing a series of operations. The **running time** typically increases with **input size** n. To categorize an algorithm's performance, we can consider:

- **Best case**: The most favorable input scenario.
- **Average case**: The "typical" or expected scenario (often difficult to analyze rigorously).
- **Worst case**: The most unfavorable input scenario, guaranteeing an upper bound on running time.

> **RUNNING TIME:** The length of time an algorithm takes to process an input. Often measured by counting the number of basic operations rather than raw wall-clock time.

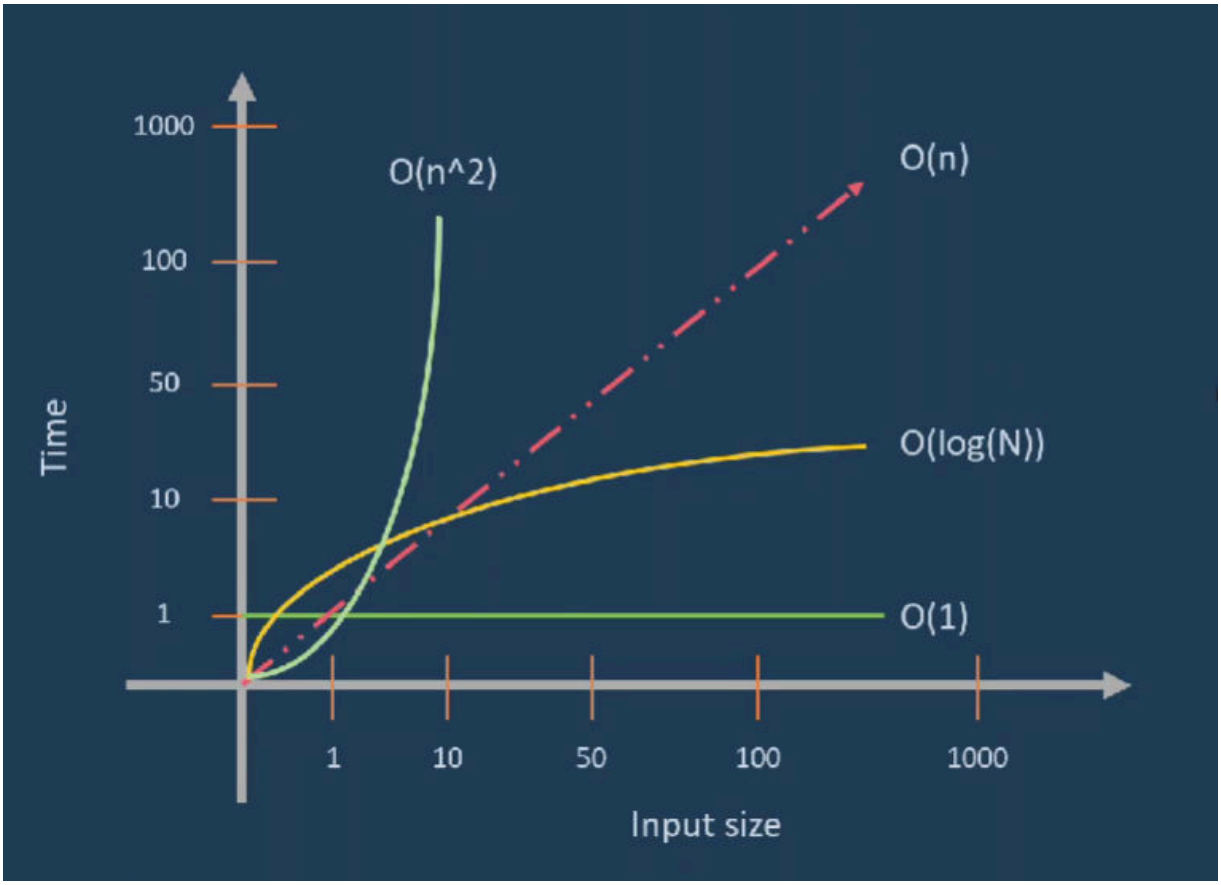
Experimental Approach

One naive way to measure performance is:

1. **Implement** the algorithm in a programming language.
2. **Run** the program on inputs of varying sizes.
3. **Record** the time taken (e.g., milliseconds).
4. **Plot** or compare the results.

Example (Pseudocode for an experimental run):

```
for n in [1, 10, 50, 100, 1000]:
    input_data = generate_data(n)
    start_time = now()
    result = myAlgorithm(input_data)
    end_time = now()
    print("Input Size:", n, "Running Time:", end_time - start_time)
```



Limitations of Experimental Approach

While experiments can offer real performance data, they have drawbacks:

1. **Implementation Overhead**: Writing a correct, optimized implementation can be difficult.
2. **Coverage**: Trials may not cover all possible input distributions.
3. **Environment Dependence**: The results rely on specific hardware and software conditions. Comparing two algorithms requires the same environment.
4. **Scalability**: Testing large n might be too time-consuming or resource-intensive.

Hence, experimental data alone may not provide a complete picture, prompting the need for a more **theoretical** analysis.

Theoretical Analysis

Rather than coding and measuring, **theoretical analysis** employs a higher-level description (often **pseudocode**) to estimate running time as a function of input size n. It:

- Considers **all possible inputs** (especially worst case).
- Abstracts away hardware and compiler differences by counting **primitive operations**.
- Enables comparisons of algorithms on an **equal** footing.

This approach provides a **machine-independent** measure of an algorithm's efficiency. The **theoretical analysis** of an algorithm is the study of its **efficiency** and **correctness** without implementing it in code. It involves evaluating the **running time** and **space usage** as functions of the input size n, typically using asymptotic notation like $O(n)$. Is it just counting **primitive operations**? Yes, **one method** of theoretical analysis is **counting primitive operations**, but it is not the only approach. **Primitive operation counting** is useful for deriving **exact complexity functions**.
There are also o**ther methods.** One method is **recurrence relations**, which are used for analyzing recursive algorithms like Merge Sort, where the time complexity is expressed as $T(n) = 2T(n/2) + O(n)$ and solved using techniques such as the **Master Theorem**. Another method is **worst, best, and average case analysis**, where the worst case $O(n)$ represents the maximum number of operations needed, the best case $\Omega(n)$ represents the minimum, and the average case $\Theta(n)$ gives the expected number of operations over random inputs. Lastly, **amortized analysis** is useful when some operations take significantly longer than others, such as in dynamic arrays and hash tables, where the cost of expensive operations is averaged over multiple operations.

Common Functions in Algorithm Analysis

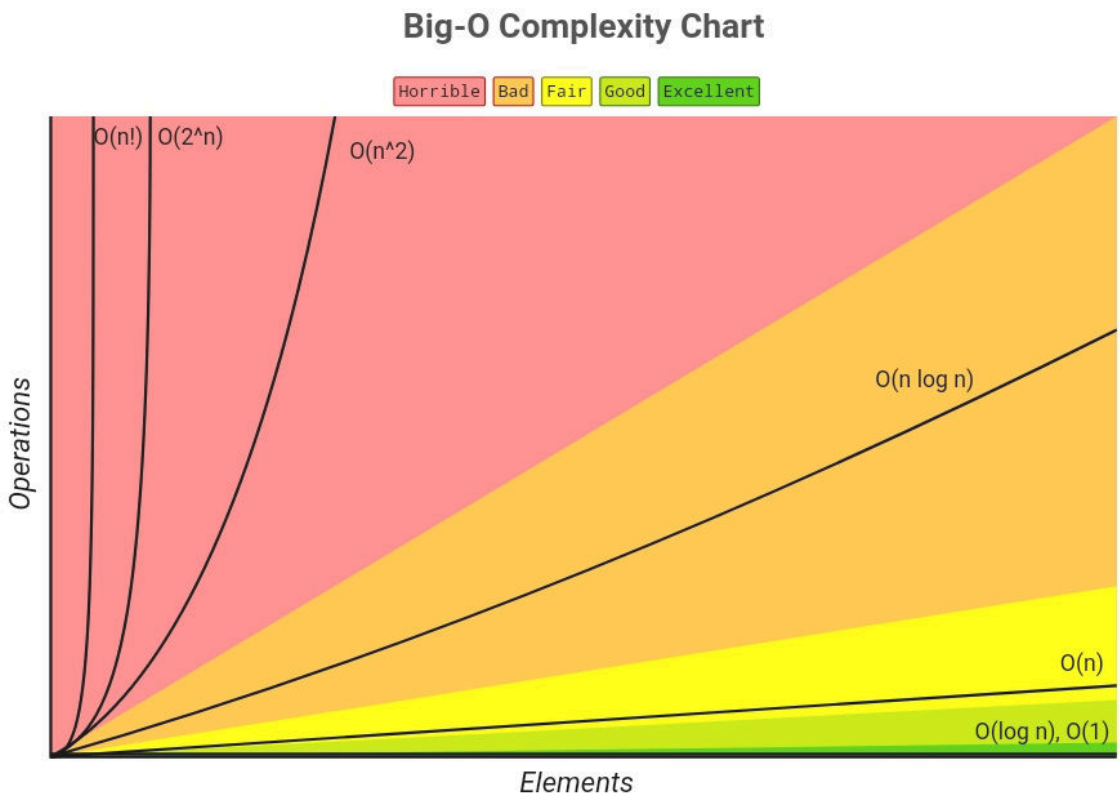
Certain mathematical functions appear repeatedly when describing algorithm running times:

1. **Constant** – $O(1)$
2. **Logarithmic** – $O(\log n)$
3. **Linear** – $O(n)$
4. **Linearithmic** – $O(n \log n)$
5. **Quadratic** – $O(n^2)$

6. **Cubic** – $O(n^3)$
7. **Exponential** – $O(2^n)$

> **GROWTH RATE:** Describes how quickly a function (or algorithm's running time) increases as n grows.

In practice, **logarithmic**, **linear**, and **n log n** complexities are usually considered more scalable than **quadratic**, **cubic**, or **exponential**.



Primitive Operations

> **PRIMITIVE OPERATION:** A low-level computation assumed to take constant time in the theoretical (RAM) model. Examples include:

- Evaluating an expression (e.g., `x + 1`)
- Assigning a value to a variable

- Indexing an array element
- Calling or returning from a function
- Comparing two numbers

In pseudocode, counting these operations helps approximate the total running time.

Counting Operations in an Algorithm

To analyze an algorithm, we **inspect** its pseudocode and **estimate** the total number of primitive operations as a function of n. For instance, consider a pseudocode snippet:

```
max_value = A[0]                // 2 operations (index + assign)
for i in 1 to n-1:              // loop runs (n-1) times
    if A[i] > max_value:        // 2 operations each time (index + c
ompare)
        max_value = A[i]        // 1 operation (assign)
return max_value                // 1 operation
```

- The loop body might execute up to **(n−1)** times (worst case).
- Summing these gives a total that typically looks like $c_1 \cdot n + c_2$. $(T(n) = 5n - 1)$

Such a result indicates a **linear** time complexity, $O(n)$.

Growth Rate of Running Time

Once we have a function $T(n)$ representing the count of primitive operations, we observe:

- Changing hardware or software usually **multiplies** $T(n)$ by a constant factor, but does not alter its fundamental growth pattern.
- For large n, the **leading term** of $T(n)$ dominates.

Example:

- $T(n) = 5n - 1$ is effectively **linear**, so we might denote $T(n)$ as $O(n)$.

Big-Oh Notation

> **BIG-OH (O):** Formally, $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 1$ such that:
>
> $$f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

In simpler terms, $f(n)$ grows at most as fast as $g(n)$ (up to constant multiples) for sufficiently large n.

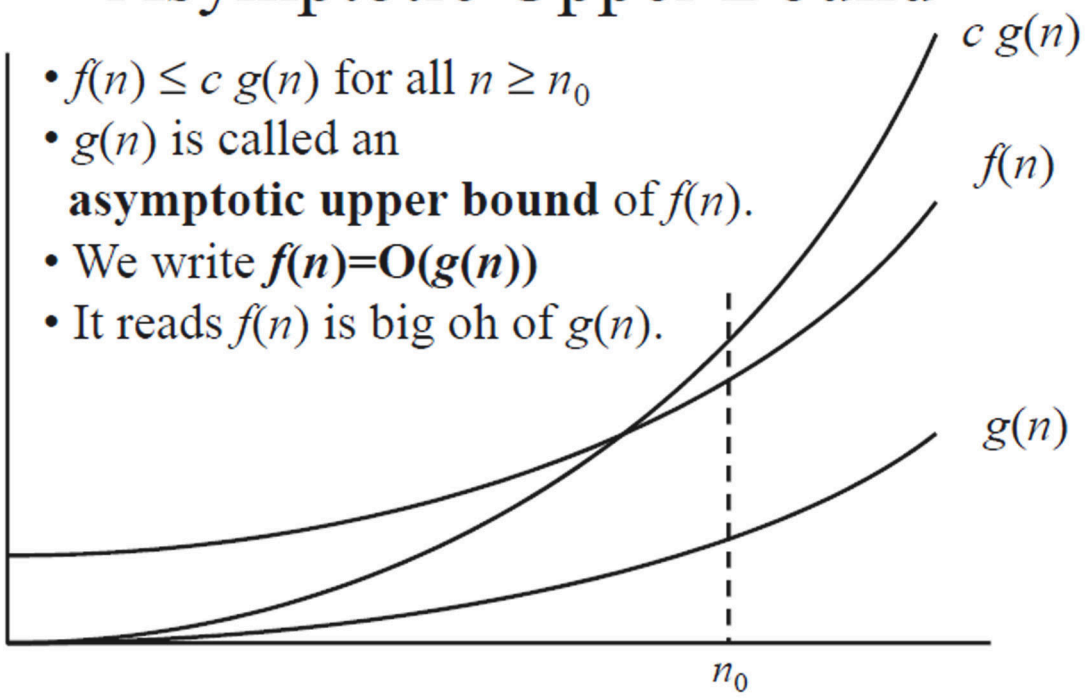
Examples of Big-Oh

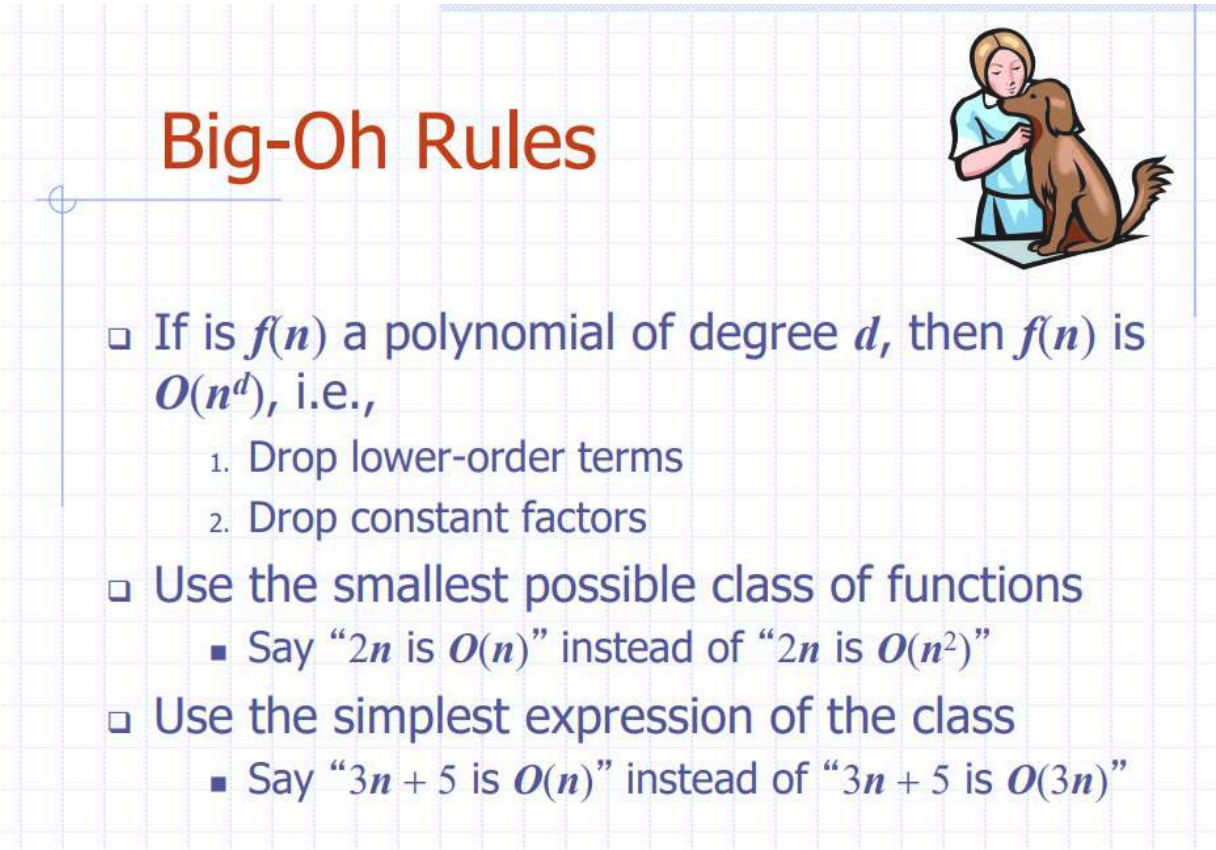
1. $2n + 10$ is $O(n)$. We can pick $c = 3$ and $n_0 = 10$.
2. $3n^3 + 20n^2 + 5$ is $O(n^3)$ by ignoring constant factors and lower-order terms.
3. $3 \log n + 5$ is $O(\log n)$.

   If an algorithm's time is $5n + 5$, we say it is $O(n)$. The big-Oh focuses on **dominant** terms and **ignores** constant coefficients.
   Big-O notation represents the **asymptotic upper bound** (worst-case complexity) of an algorithm.
   But **O(n) is not an exact count**. Instead, it describes the dominant term, ignoring constants and lower-order terms

Asymptotic Upper Bound

- $f(n) \leq c\, g(n)$ for all $n \geq n_0$
- $g(n)$ is called an **asymptotic upper bound** of $f(n)$.
- We write $\boldsymbol{f(n)=O(g(n))}$
- It reads $f(n)$ is big oh of $g(n)$.





Examples and Comparisons

Comparing Two Algorithms

If one algorithm is $O(n^2)$ (insertion sort) and another is $O(n \log n)$ (merge sort), we can see that for large n:

- n^2 eventually outgrows $n \log n$.
- For a million elements, an $O(n^2)$ sort can be dramatically slower.

Constant factors and smaller terms do not affect which function eventually dominates. The difference in growth rate can yield dramatic differences in execution times for large inputs.

Big-Omega and Big-Theta

Besides big-Oh, we have two other asymptotic notations:

> **BIG-OMEGA (Ω):** $f(n)$ is $\Omega(g(n))$ if $f(n)$ grows at least as fast as $g(n)$. Formally, there exist $c > 0$ and n_0 such that:
>
> $$f(n) \geq c \cdot g(n) \quad \text{for } n \geq n_0.$$

> **BIG-THETA (Θ):** $f(n)$ is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$. In other words, $f(n)$ grows on the same order as $g(n)$. There exist constants c' and c'' and an n_0 such that:
>
> $$c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n), \quad \text{for } n \geq n_0.$$

Intuitive Guide:

- $O(\cdot)$: up to
- $\Omega(\cdot)$: at least
- $\Theta(\cdot)$: same order

Asymptotic Algorithm Analysis

Asymptotic analysis uses big-Oh, big-Omega, and big-Theta to describe the growth behavior for large n.

1. **Focus** on the worst-case count of operations (or sometimes average case).
2. **Simplify** the resulting function to its dominant term, ignoring constants and lower-order parts.
3. **Express** the final result in big-Oh (for an upper bound) or big-Theta (for a tight bound).

Example:

- If `arrayMax` scanning an array does about $5n + 5$ operations, then we say it runs in $\Theta(n)$ (also $O(n)$ and $\Omega(n)$).

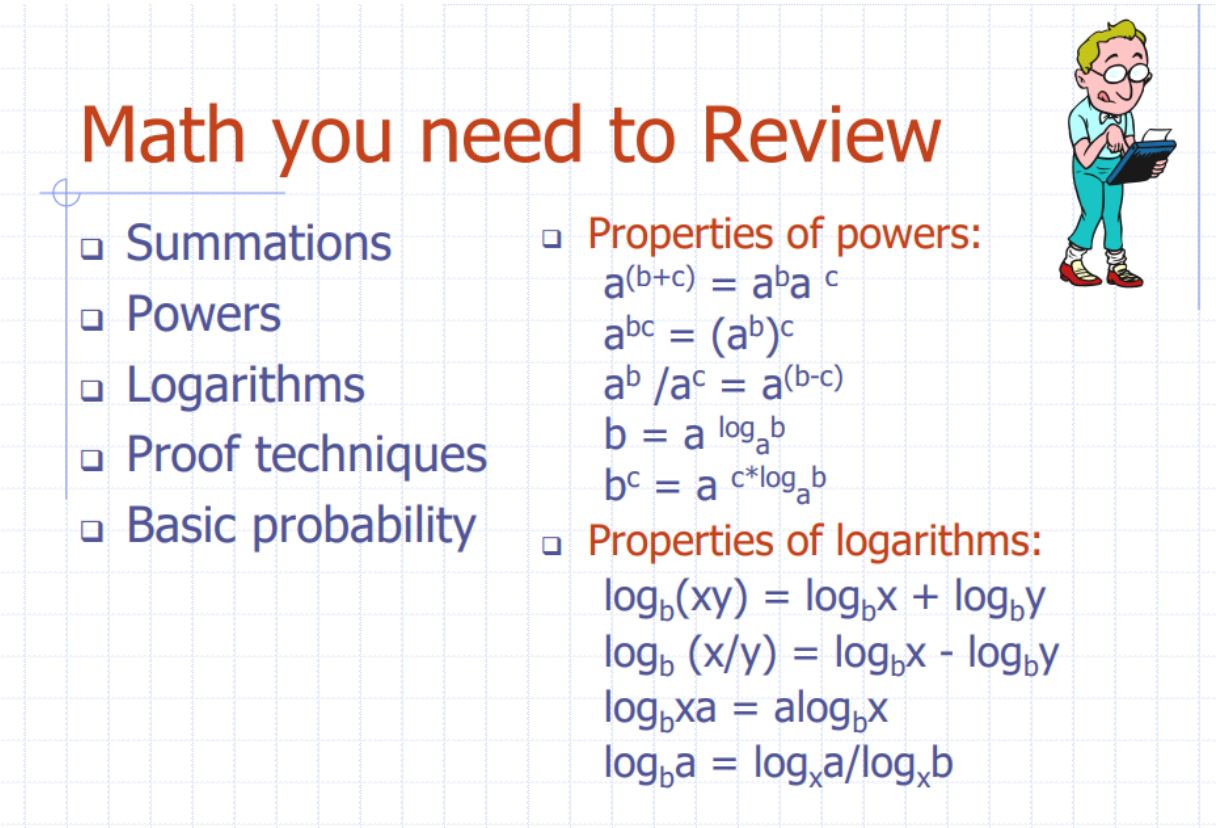
Review of Basic Math

Analyzing algorithms often involves properties of:

- **Powers** ($2^{a+b} = 2^a \cdot 2^b$)

- **Logarithms** ($\log(xy) = \log x + \log y$)
- **Summations** (arithmetic series, geometric series)
- **Exponential** and polynomial relations
- **Proof techniques** (induction, contradiction)
- **Elementary probability** (expected values, distributions)

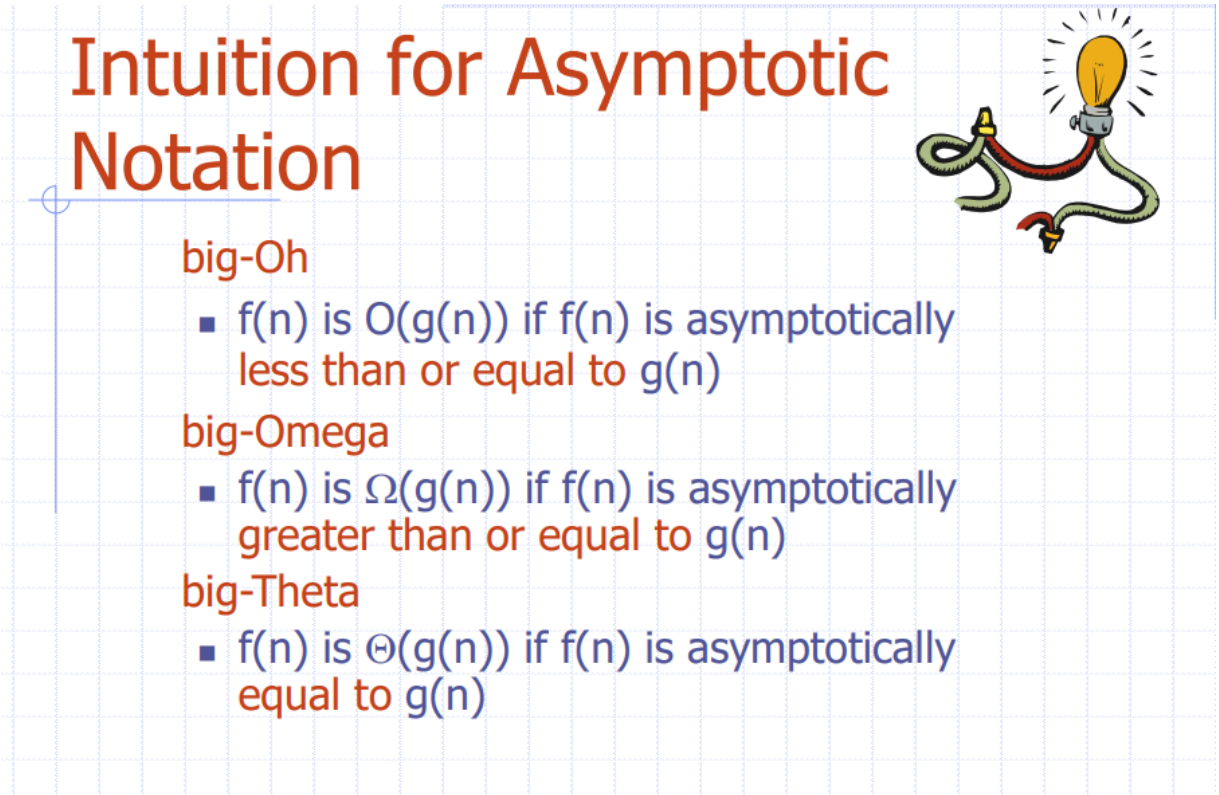
Having these fundamentals at hand helps in reasoning about algorithm complexities like $n \log n$, n^2, or 2^n.



Putting It All Together

1. **Identify** the algorithm's basic operations in pseudocode.
2. **Count** how many times these operations execute for worst-case (or average-case).
3. **Express** that count as a function $T(n)$.
4. **Simplify** $T(n)$ to its dominant terms and use big-Oh notation to summarize.

5. **(Optionally)** refine using big-Omega or big-Theta to indicate lower and exact bounds.

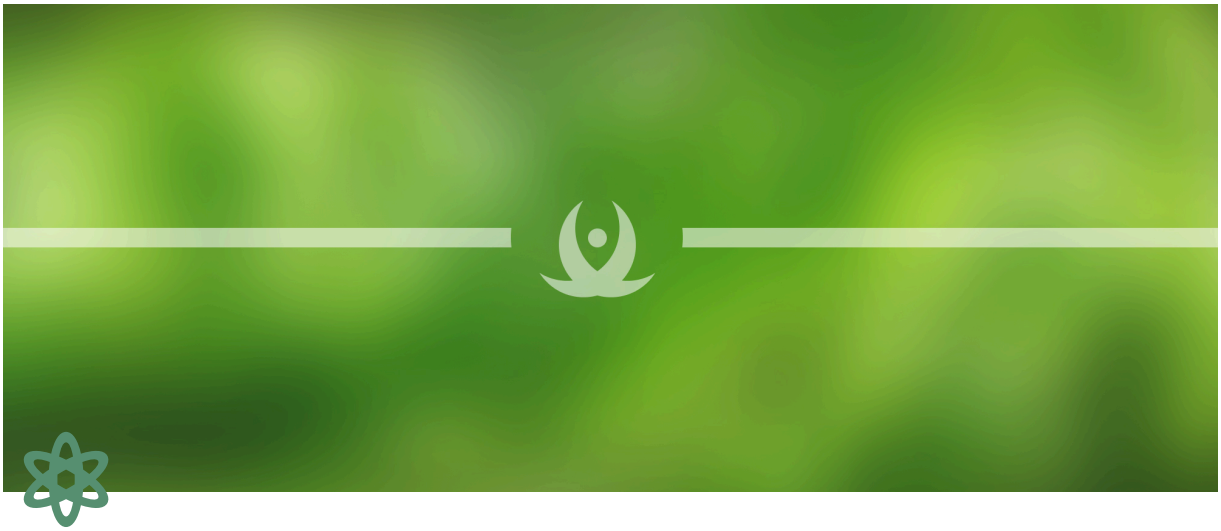


Example Scenario

- Sorting algorithms: Compare $\Theta(n^2)$ insertion sort with $\Theta(n \log n)$ merge sort. For large n, the $n \log n$ approach is asymptotically faster, even if insertion sort might have a smaller constant factor for small n.

Self Test

Self-Test: Lecture 1



2. Recursion

Introduction to Recursion

Recursion is a programming technique where a method calls itself to solve a problem. It is widely used in algorithm design to break down complex problems into simpler sub-problems.

Key Concepts in Recursion

> **RECURSION:** The process in which a method calls itself to solve smaller instances of the same problem.

Essential Components of a Recursive Method

- **Base Case(s):**

  > **BASE CASE:** The condition under which the recursion terminates. Every chain of recursive calls must eventually reach a base case where no further recursive calls are made.

- **Recursive Calls:**

  > **RECURSIVE CALL:** A call within a method to itself, which must make progress toward reaching the base case.

The Recursion Pattern

A classic example of recursion is the factorial function. The recursive definition of the factorial is:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n \geq 1 \end{cases}$$

Factorial Function Example in Java

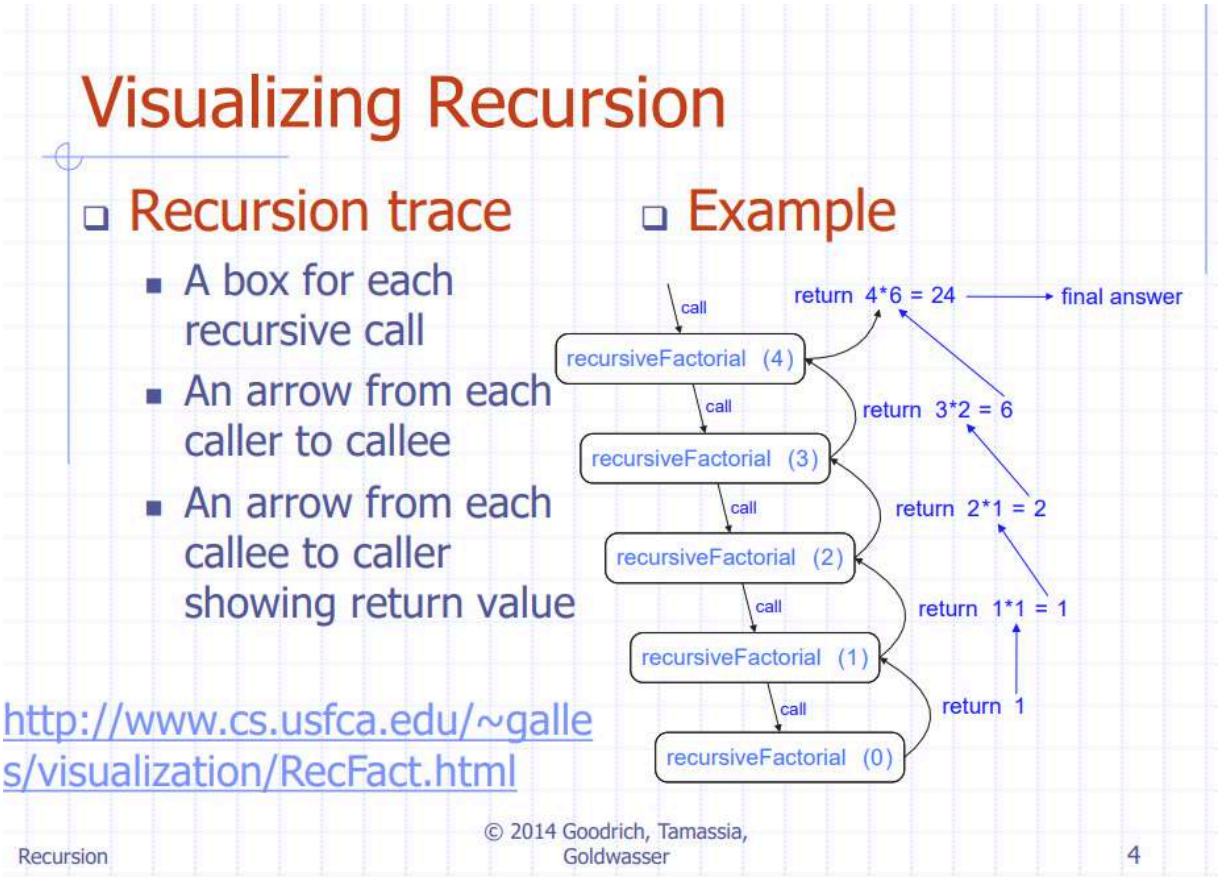
```
public int factorial(int n) {
    if (n == 0) {
        return 1; // Base case
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}
```

This example demonstrates how each recursive call reduces the problem until the base case is reached.

Visualizing Recursion

Understanding recursion can be aided by a visual recursion trace, where:

- Each recursive call is represented by a box.
- Arrows show the flow from the caller to the callee and the return values.



Recursion in Binary Search

Binary search is an efficient algorithm that uses recursion to search for an integer in an ordered list. It works by comparing the target value with the middle element of the array:

- If the target equals the middle element, the search is complete.
- If the target is less than the middle element, the search recurses on the first half.
- If the target is greater than the middle element, the search recurses on the second half.

Analysis of Binary Search

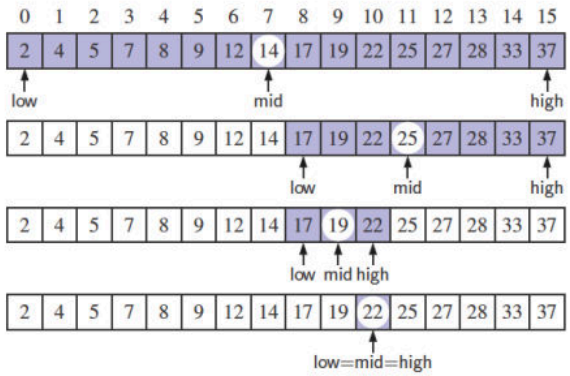
- **Time Complexity:**

  > $O(\log n)$: Each recursive call reduces the search region by half, resulting in at most $\log n$levels of recursion.

https://www.cs.usfca.edu/~galles/visualization/Search.html

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



 Recursion 10

Computing Powers Using Recursion

Naive Recursive Power Function

The power function $p(x, n) = x^n$ can be computed recursively by multiplying x by itself n times. This approach, however, makes n recursive calls, resulting in an $O(n)$ time complexity.

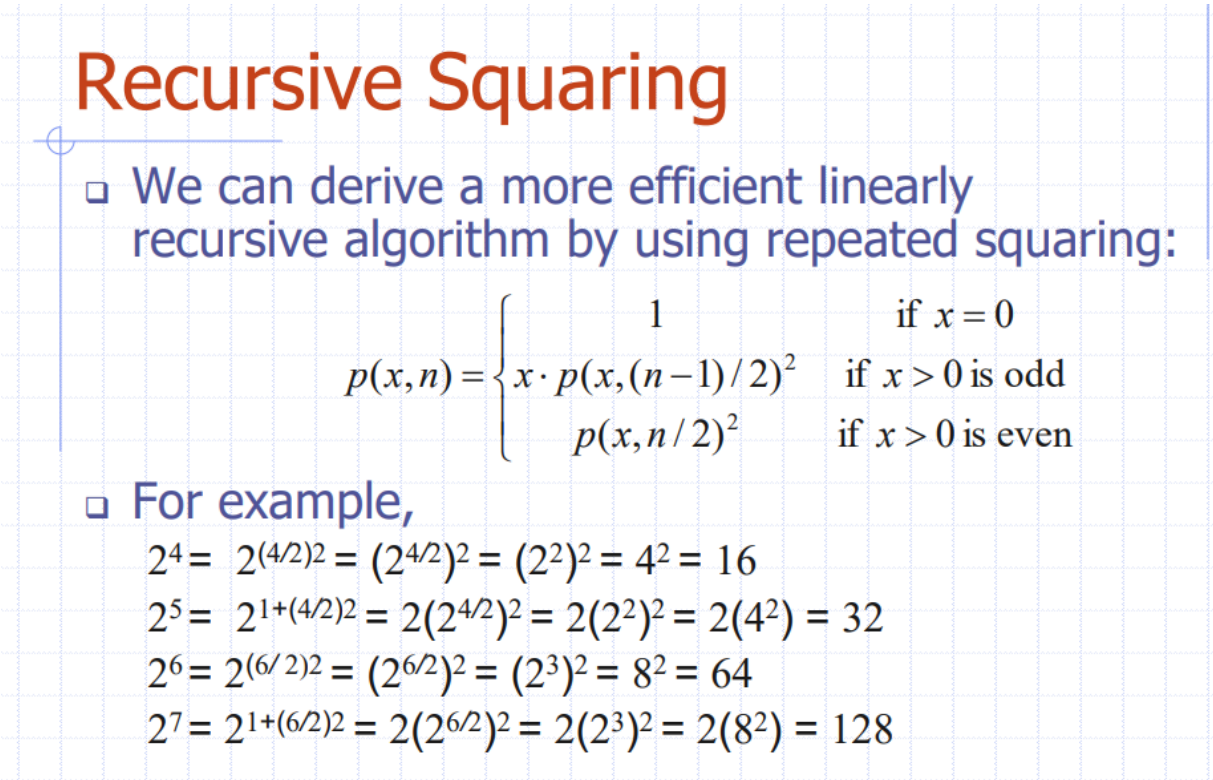
Recursive Squaring Method

A more efficient method is to use **recursive squaring**. The idea is to reduce the number of multiplications by halving the exponent at each recursive call.

Algorithm: Recursive Squaring

1. **Base Case:**
   If $n = 0$, return 1.

2. **Recursive Case for Odd n:**
   - Compute $y = \text{Power}(x, \frac{n-1}{2})$
   - Return $x \times y \times y$
3. **Recursive Case for Even n:**
   - Compute $y = \text{Power}(x, \frac{n}{2})$
   - Return $y \times y$



Pseudocode

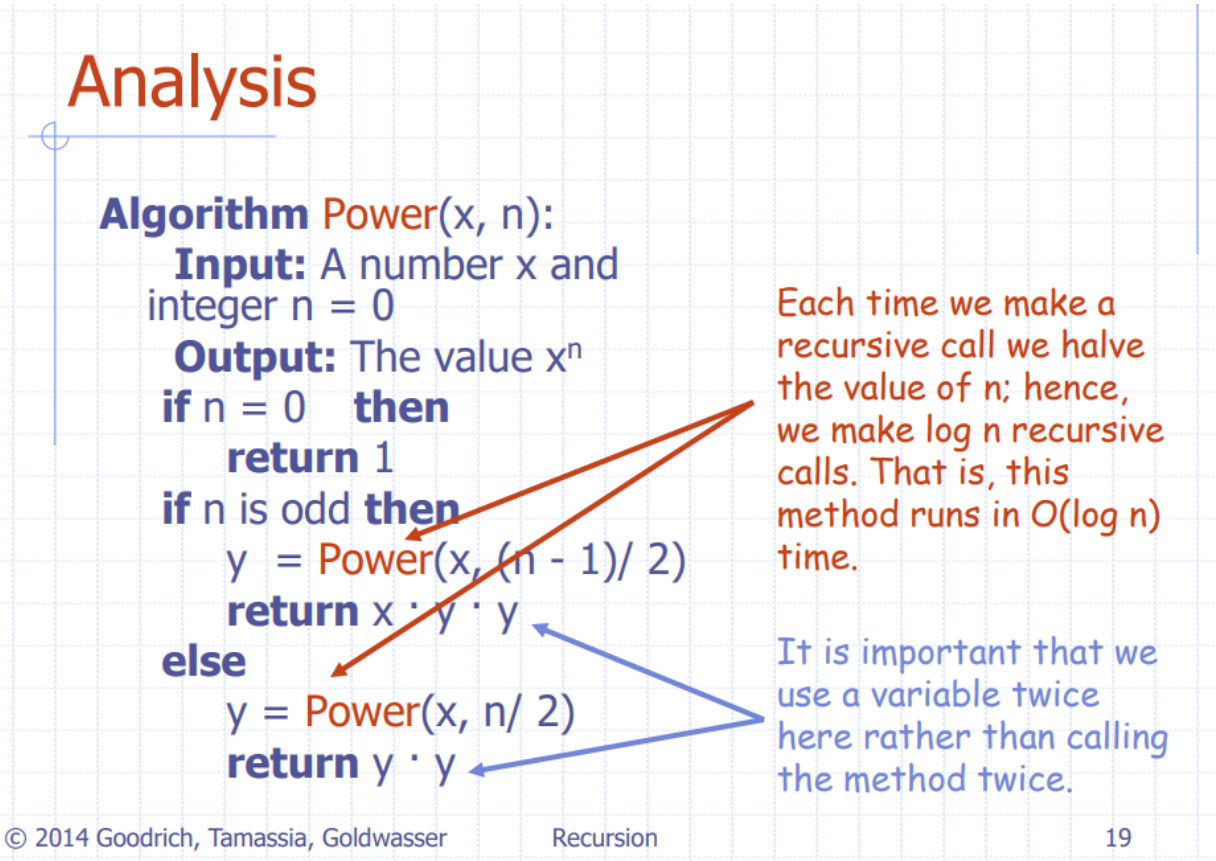
```
Algorithm Power(x, n):
    if n == 0 then
        return 1
    if n is odd then
        y = Power(x, (n - 1) / 2)
        return x * y * y
    else
```

```
        y = Power(x, n / 2)
        return y * y
```

> **RECURSIVE SQUARING:** A method to compute powers in $O(\log n)$ time by halving the exponent with each recursive call.

Analysis of Recursive Squaring

- **Efficiency:**
  By halving the exponent at each step, the algorithm only makes $\log n$ recursive calls, making it significantly faster than the naive approach.

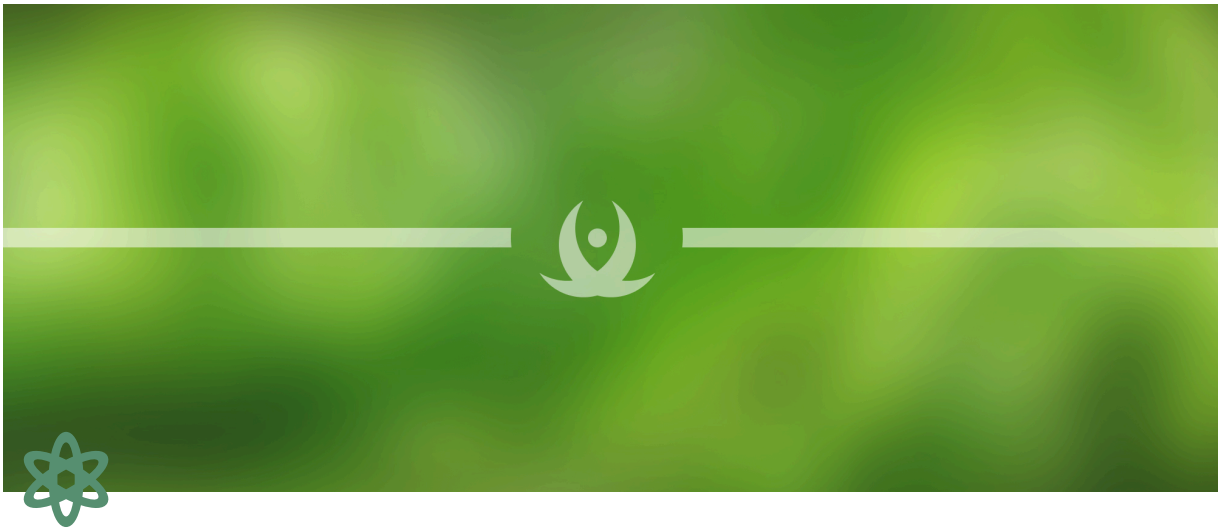


Final Summary & Takeaways

- **Recursion** simplifies complex problems by breaking them into smaller, manageable sub-problems.

- **Base cases** are critical to ensure termination of recursion.
- **Visual aids** like recursion traces can help in understanding the flow of recursive calls.
- **Binary search** demonstrates how recursion can efficiently solve search problems in $O(\log n)$ time.
- **Recursive squaring** offers a powerful method for computing powers with logarithmic time complexity.

By mastering these recursive techniques, one can design efficient algorithms for a variety of computational problems.



3. Recurrence Relations and Complexity Analysis

Objective & Scope

This note explains how to analyze the time complexity of recursive algorithms by solving recurrence relations. We focus on two recurrences:

- $T(n) = c + 2 \cdot T(n/2)$, which solves to $O(n)$
- $T(n) = c + T(n/2)$, which solves to $O(\log n)$

We also emphasize the importance of considering the input size when analyzing algorithm complexity.

Analyzing Recurrences

The key idea behind these analyses is to understand how the recurrence expands as the input size is reduced step by step, often until the base case is reached. In our derivations, the number of steps (or levels of recursion) is denoted by **k**, which is typically **log n** when the problem size is halved at each step.

Recurrence 1: $T(n) = c + 2 \cdot T(n/2)$

Derivation by Iteration

1. **Initial Formulation:**

$$T(n) = c + 2 \cdot T\left(\frac{n}{2}\right)$$

2. **First Level of Recursion:**

$$T\left(\frac{n}{2}\right) = c + 2 \cdot T\left(\frac{n}{4}\right)$$

Substitute back:

$$T(n) = c + 2\left[c + 2 \cdot T\left(\frac{n}{4}\right)\right] = c + 2c + 2^2 \cdot T\left(\frac{n}{4}\right)$$

3. **Second Level of Recursion:**

$$T\left(\frac{n}{4}\right) = c + 2 \cdot T\left(\frac{n}{8}\right)$$

Substitute:

$$T(n) = c + 2c + 2^2\left[c + 2 \cdot T\left(\frac{n}{8}\right)\right] = c + 2c + 2^2 c + 2^3 \cdot T\left(\frac{n}{8}\right)$$

4. **General Pattern:**
After **k** levels, where $n/2^k = 1$ (i.e., $k = \log_2 n$):

$$T(n) = c + 2c + 2^2 c + \cdots + 2^{k-1} c + 2^k \cdot T(1)$$

Let $T(1)$ be some constant c'. The summation is a geometric series:

$$c\left(1 + 2 + 2^2 + \cdots + 2^{k-1}\right) = c\left(2^k - 1\right)$$

Then:

$$T(n) = c\left(2^k - 1\right) + 2^k c' \quad \text{with } 2^k = n$$

Thus:

$$T(n) = c(n-1) + c'n = O(n)$$

> **KEY RESULT:** $T(n) = c + 2 \cdot T(n/2)$ resolves to an overall complexity of $O(n)$.

Recurrence 2: $T(n) = c + T(n/2)$

Derivation by Iteration

1. **Initial Formulation:**

$$T(n) = c + T\left(\frac{n}{2}\right)$$

2. **First Level of Recursion:**

$$T\left(\frac{n}{2}\right) = c + T\left(\frac{n}{4}\right)$$

Substitute back:

$$T(n) = c + \left[c + T\left(\frac{n}{4}\right)\right] = 2c + T\left(\frac{n}{4}\right)$$

3. **Second Level of Recursion:**

$$T\left(\frac{n}{4}\right) = c + T\left(\frac{n}{8}\right)$$

Substitute:

$$T(n) = 2c + \left[c + T\left(\frac{n}{8}\right)\right] = 3c + T\left(\frac{n}{8}\right)$$

4. **General Pattern:**
After **k** levels, where $n/2^k = 1$ (i.e., $k = \log_2 n$):

$$T(n) = kc + T(1)$$

With $T(1)$ being constant, we have:

$$T(n) = c\log_2 n + \text{constant} = O(\log n)$$

> **KEY RESULT:** $T(n) = c + T(n/2)$ resolves to an overall complexity of $O(\log n)$.

Considering Input Size in Complexity Analysis

When analyzing algorithm complexity, **the input size is critical** because it determines the number of operations performed by the algorithm.

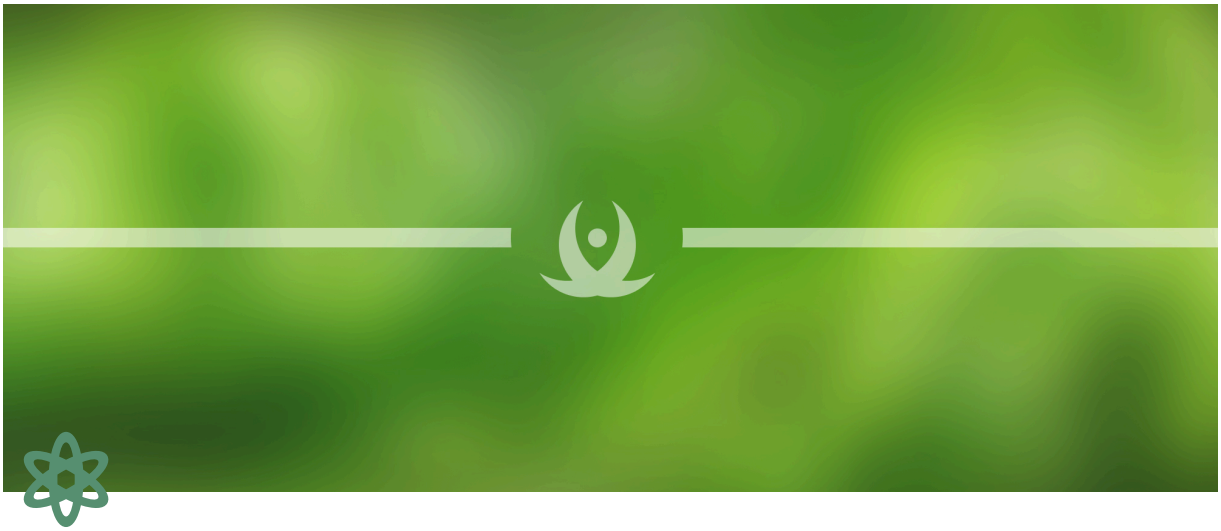
- For the recurrence $T(n) = c + 2 \cdot T(n/2)$, each level of recursion doubles the number of calls, leading to a total work proportional to n.
- For $T(n) = c + T(n/2)$, the problem size halves with each recursive call, resulting in **logarithmically** many levels, hence $O(\log n)$ complexity.

> **INPUT SIZE:** The variable n represents the input size, and how n is processed (e.g., divided in half at each step) directly impacts the total computational work required.

Final Summary & Takeaways

- **Recurrence Analysis:**
  - $T(n) = c + 2 \cdot T(n/2)$ sums up to a geometric series leading to $O(n)$ complexity.
  - $T(n) = c + T(n/2)$ expands linearly with the number of levels $k = \log n$, leading to $O(\log n)$ complexity.
- **Input Size Impact:**
  The input size determines the number of recursive calls or levels in an algorithm, which is fundamental to understanding its overall efficiency.
- **Visual Aids:**
  Recursion trees or diagrams are powerful tools for visualizing how recurrences expand and summing the work done at each level.

By grasping these recurrence relations and the role of input size in complexity analysis, you can better evaluate and design efficient algorithms.



4. Arrays and Singly Linked Lists

Objective & Scope

This note covers two fundamental data structures:

- **Arrays:** Their structure, usage, and how they store primitive elements or object references.
- **Singly Linked Lists:** Their structure, node composition, and basic operations such as insertion and removal.

Arrays

Arrays Overview

> **ARRAYS:** A contiguous block of memory used to store a fixed number of elements. Arrays can contain either primitive data types (such as characters) or references to objects.

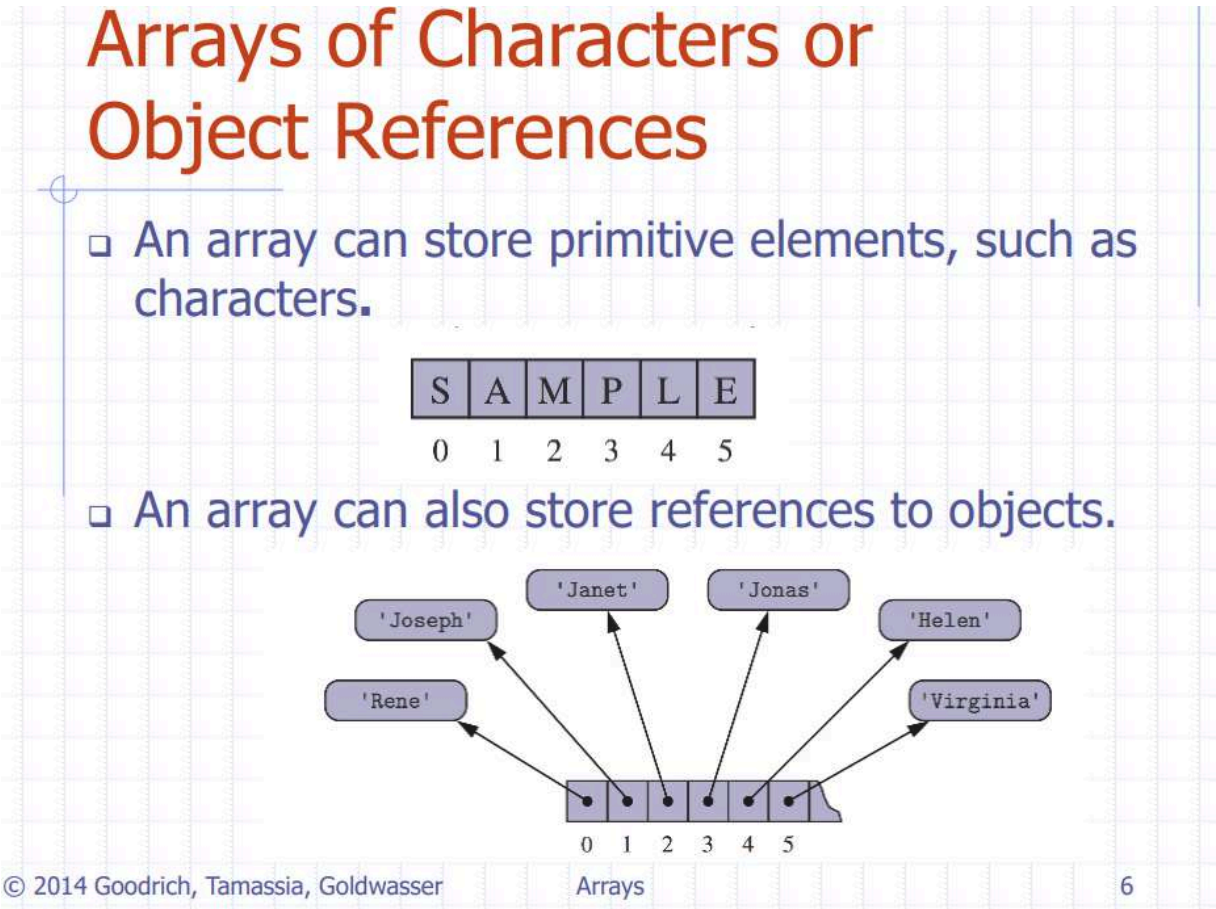
Key Characteristics

- **Storage:** Elements are stored in consecutive memory locations.
- **Access:** Fast random access using indices.

- **Fixed Size:** Once created, the size of an array cannot change without creating a new array.

Arrays of Characters or Object References

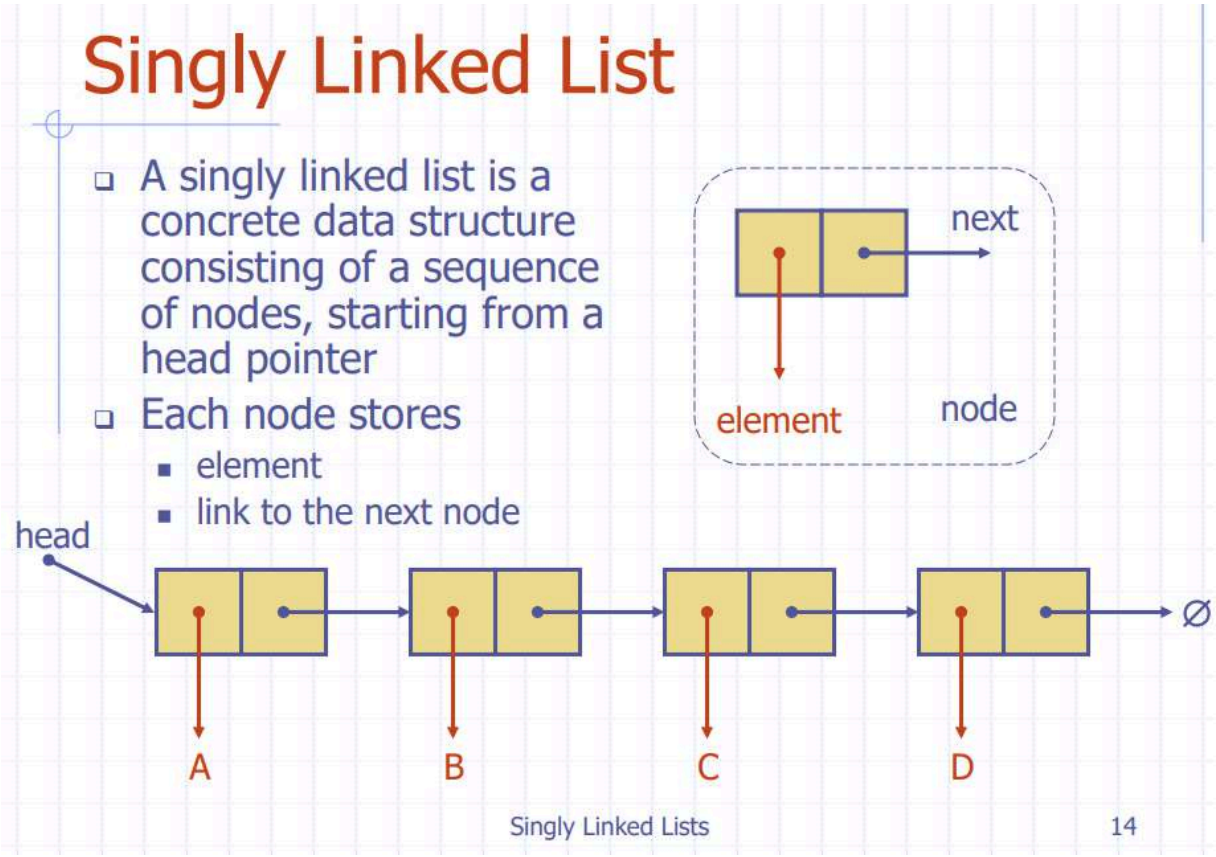
- Arrays can store primitive elements like characters.
- They can also store references to objects, allowing for the management of more complex data types.



Singly Linked Lists

Singly Linked List Overview

> **SINGLY LINKED LIST:** A dynamic data structure consisting of nodes where each node stores an element and a reference (link) to the next node in the sequence. The list is accessed starting from a head pointer.



Node Structure in a Singly Linked List

- **Element:** The data value stored in the node.
- **Link to Next Node:** A pointer that references the next node in the list.
- **Head Pointer:** Points to the first node in the list.

A Nested Node Class

- Typically, the node is implemented as a nested class within the linked list class.
- This encapsulation helps keep the implementation details hidden from the user.

Accessor Methods

> **ACCESSOR METHODS:** Functions that allow access to the data stored in nodes (e.g., retrieving the element from a node).

```
1  public class SinglyLinkedList<E> {
...    (nested Node class goes here)
14   // instance variables of the SinglyLinkedList
15   private Node<E> head = null;          // head node of the list (or null if empty)
16   private Node<E> tail = null;          // last node of the list (or null if empty)
17   private int size = 0;                 // number of nodes in the list
18   public SinglyLinkedList( ) { }        // constructs an initially empty list
19   // access methods
20   public int size( ) { return size; }
21   public boolean isEmpty( ) { return size == 0; }
22   public E first( ) {                   // returns (but does not remove) the first element
23     if (isEmpty( )) return null;
24     return head.getElement( );
25   }
26   public E last( ) {                    // returns (but does not remove) the last element
27     if (isEmpty( )) return null;
28     return tail.getElement( );
29   }
```

These methods are essential for traversing the list and for various operations such as searching or displaying list contents.

Insertion Operations in Singly Linked Lists

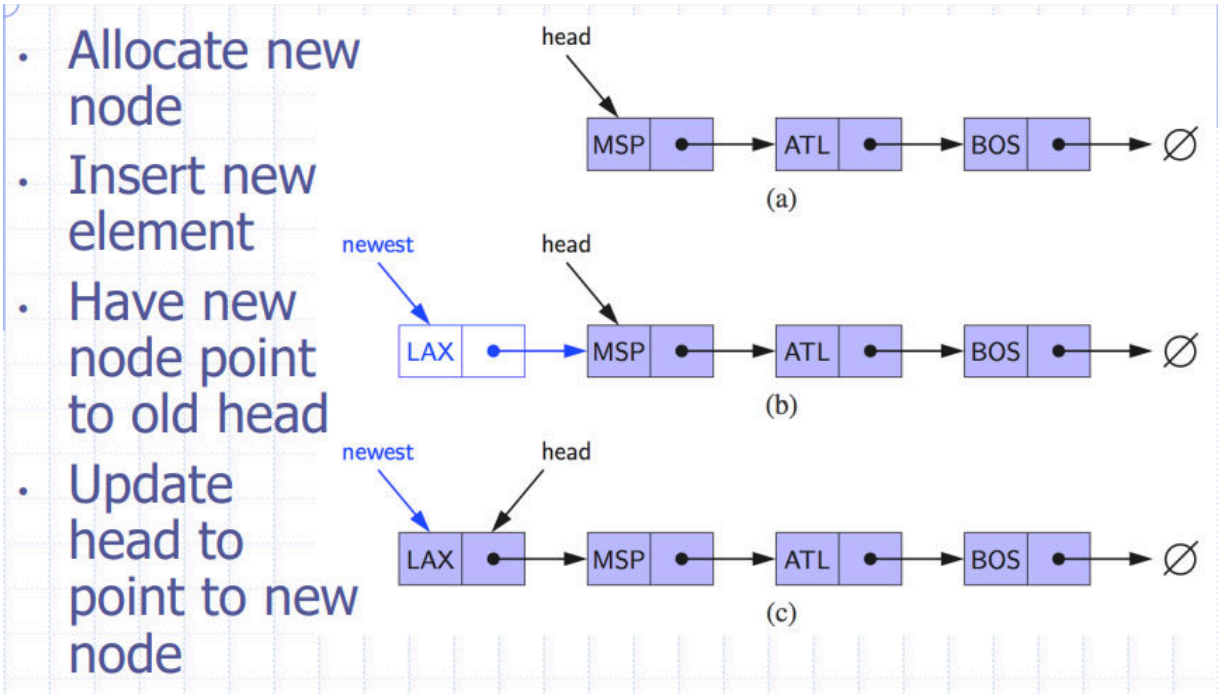
Inserting at the Head

- **Procedure:**
  - Allocate a new node.
  - Set the new node's element with the value to insert.
  - Make the new node point to the current head.
  - Update the head pointer to the new node.

> **INSERT AT HEAD:** Fast operation ($O(1)$ time complexity) because it involves updating a few pointers.

```
public void insertAtHead(E element) {
    Node<E> newNode = new Node<>(element, head);
    head = newNode;
```

```
    // If the list was empty, update the tail to the new node as we
ll.
    if (size == 0) {
        tail = newNode;
    }
    size++;
}
```

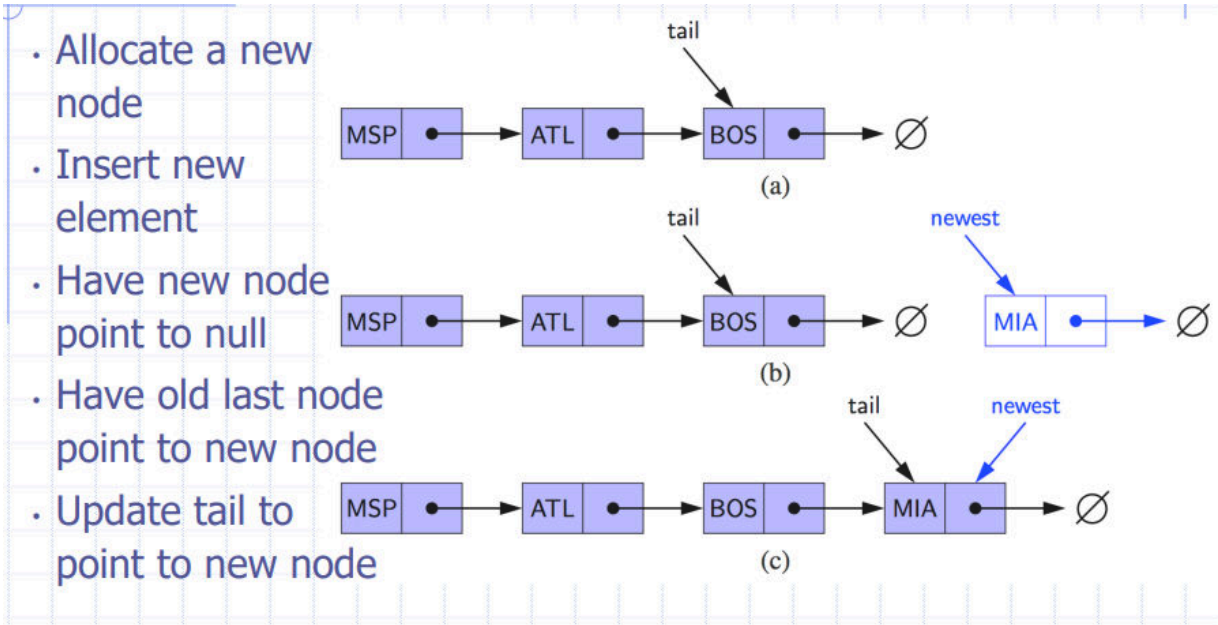


Inserting at the Tail

- **Procedure:**
  - Allocate a new node.
  - Set the new node's element with the value to insert.
  - The new node's next pointer is set to null.
  - The current tail's next pointer is updated to point to the new node.
  - Update the tail pointer to the new node.

> **INSERT AT TAIL:** May require traversing the list if no tail pointer is maintained, which can lead to $O(n)$ time complexity. But normally it is $O(n)$ time complexity.

```
public void insertAtEnd(E element) {
    Node<E> newNode = new Node<>(element, null);
    if (size == 0) {
        head = newNode;
    } else {
        tail.next = newNode;
    }
    tail = newNode;
    size++;
}
```



Removal Operations in Singly Linked Lists

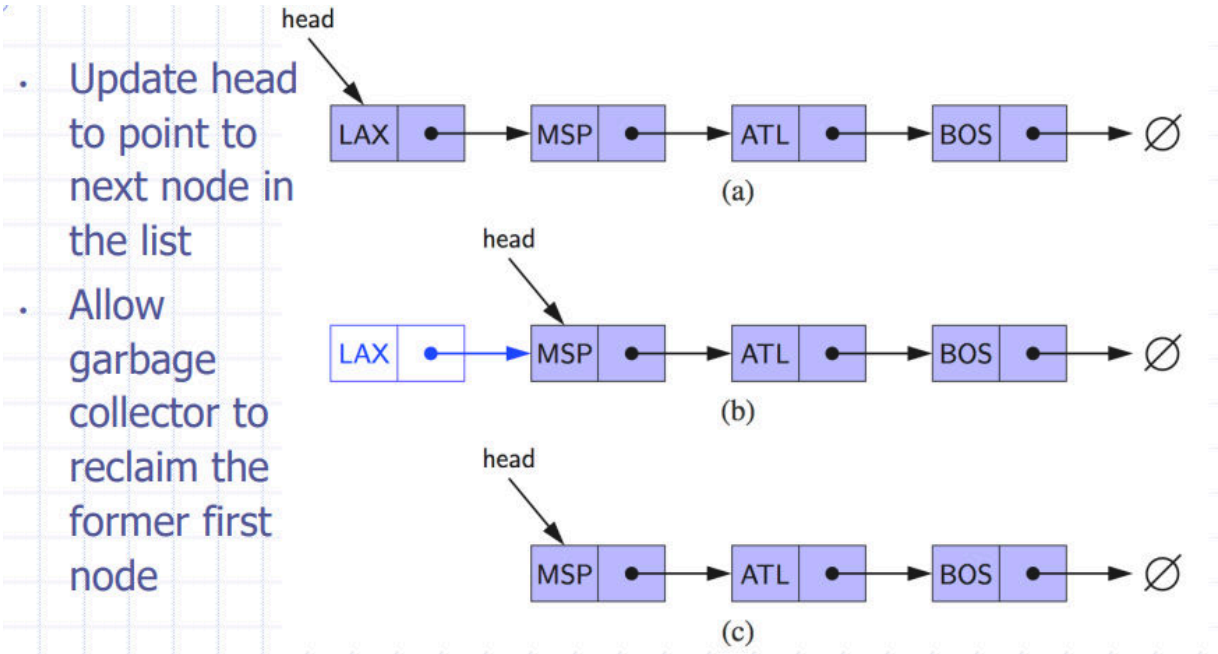
Removing at the Head

- **Procedure:**
    - Update the head pointer to point to the next node in the list.

- The removed node becomes eligible for garbage collection.

> **REMOVE AT HEAD:** Fast operation ($O(1)$ time complexity) as it simply involves reassigning the head pointer.

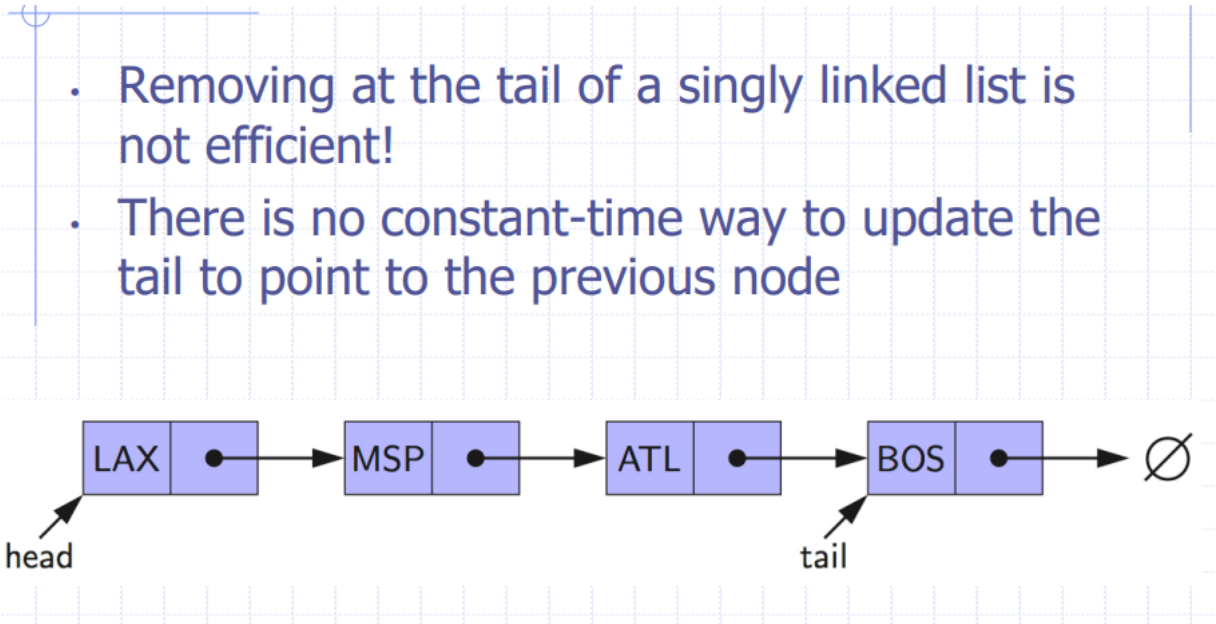
```
public E removeHead() {
    if (size == 0) {
        throw new NoSuchElementException("List is empty");
    }
    E removedElement = head.element;
    head = head.next;
    size--;
    // If the list becomes empty after removal, update tail to nul
l.
    if (size == 0) {
        tail = null;
    }
    return removedElement;
}
```



Removing at the Tail

- **Procedure:**
  - Removing the tail node is less efficient because it typically requires traversal from the head to find the node immediately preceding the tail.
  - Once found, update its next pointer to null and update the tail pointer accordingly.

> **REMOVE AT TAIL:** Inefficient in singly linked lists ($O(n)$ time complexity) due to the need for traversal.

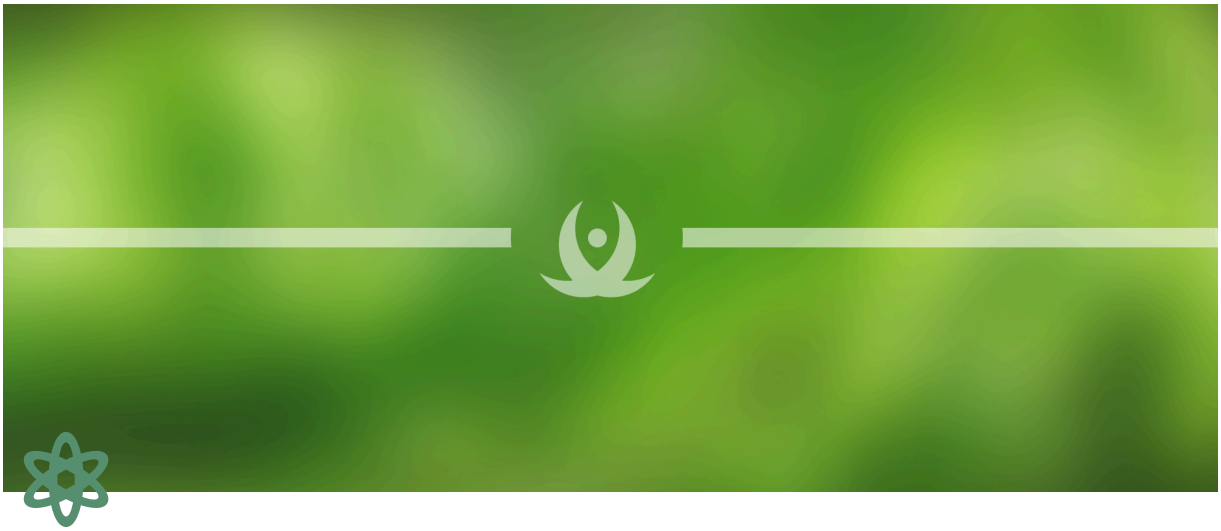


Final Summary & Takeaways

- **Arrays** offer fast random access and efficient storage but are fixed in size.
- **Singly Linked Lists** provide dynamic memory allocation, with efficient insertion and removal at the head, though tail operations can be less efficient without additional pointers.
- **Key Concepts:**
  - **Arrays:** Contiguous memory, fixed size, fast indexing.
  - **Singly Linked Lists:** Dynamic nodes, head pointer, insertion/removal strategies.

- Visual aids, such as diagrams, can significantly enhance understanding of pointer operations and structure.

By mastering these basic data structures, you build the groundwork necessary for understanding more complex data handling techniques in algorithms.



5. Doubly Linked Lists

Objective & Scope

This note focuses on **Doubly Linked Lists**, a dynamic data structure that supports bidirectional traversal. We cover:

- The structure and key components of doubly linked lists.
- Insertion and deletion operations.
- Implementation considerations, particularly in Java.

This note stops before the introduction of iterators.

Doubly Linked List Overview

> **DOUBLY LINKED LIST:** A data structure where each node maintains two pointers—one to the previous node and one to the next node—allowing traversal in both directions.

Key Characteristics

- **Bidirectional Traversal:** Nodes link both forward and backward.
- **Sentinel Nodes:** Often use special header and trailer nodes to simplify insertion and deletion at the boundaries.

- **Dynamic Nature:** Supports efficient insertions and deletions anywhere in the list without requiring a full traversal for previous-node access.

Structure of a Doubly Linked List

Node Structure

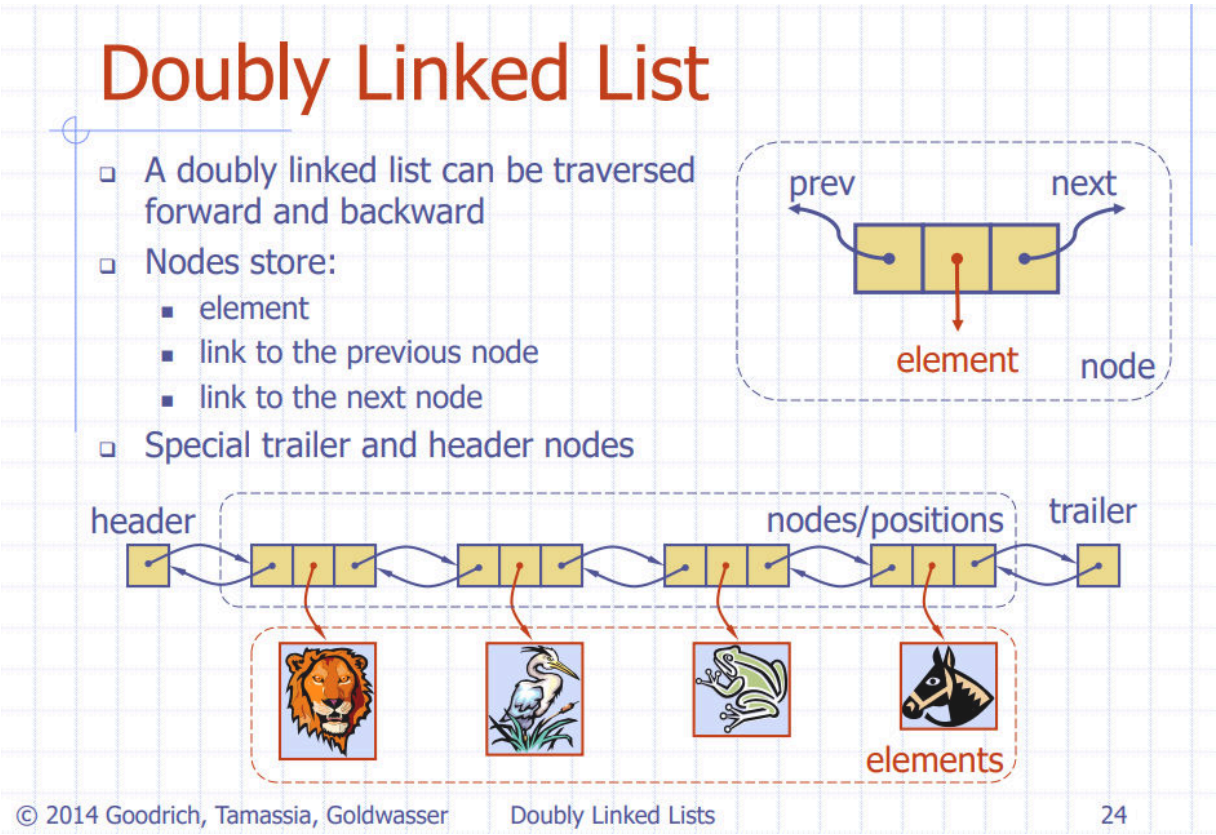
Each node in a doubly linked list contains:

- **Element:** The data stored within the node.
- **Previous Pointer:** A reference to the previous node in the list.
- **Next Pointer:** A reference to the next node in the list.

> **NODE:** The basic building block that holds the element along with pointers to both its predecessor and successor.

Special Nodes: Header and Trailer

- **Header Node:** A sentinel node at the beginning that does not hold user data but simplifies operations at the front.
- **Trailer Node:** A sentinel node at the end that similarly aids in managing edge conditions.



Insertion in Doubly Linked Lists

Insertion Operation

To insert a new node `q` between an existing node `p` and its successor:

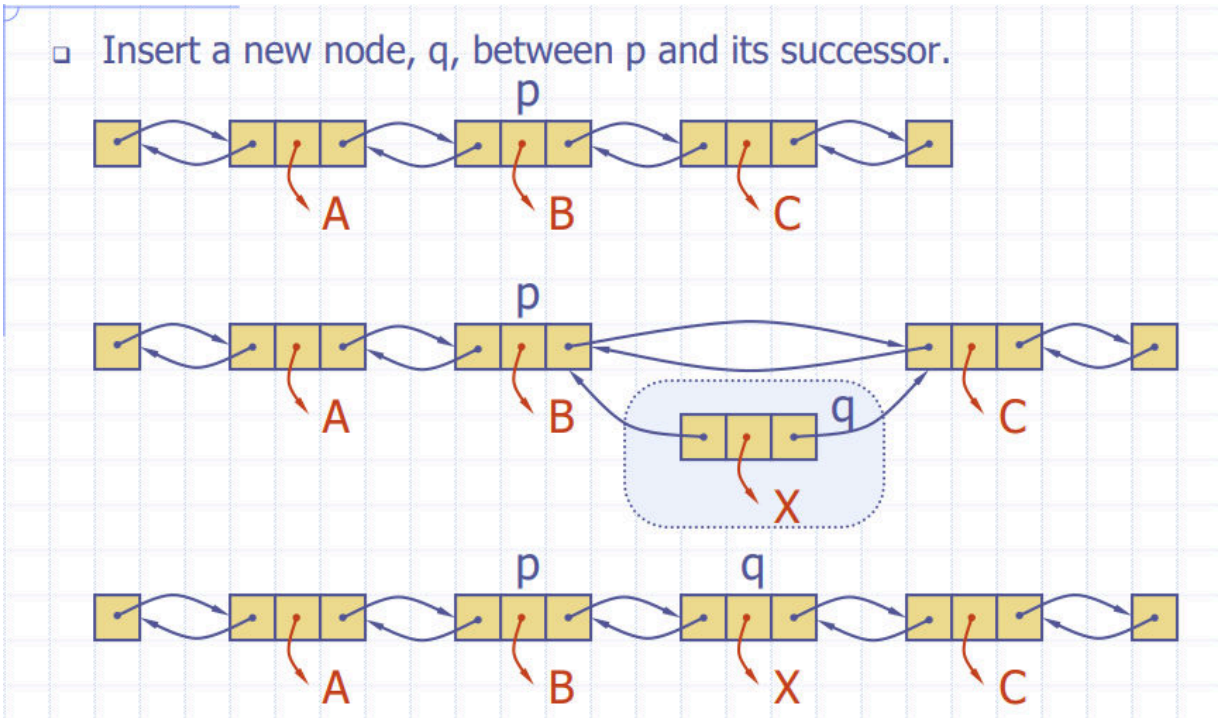
1. **Allocate a New Node:** Create node `q` with the desired element.
2. **Update Pointers:**
   - Set `q.prev` to point to node `p`.
   - Set `q.next` to point to `p`'s current next node.
   - Update `p`'s next node's previous pointer to point back to `q`.
   - Update `p.next` to point to `q`.

> **INSERTION:** Accomplished by adjusting four pointers, ensuring the new node is seamlessly integrated into the list without the need for a full traversal.

Example

For a list with nodes A → B → C, inserting element X between B and C results in:

- A → B → X → C, with `X.prev = B`, `X.next = C`, `B.next = X`, and `C.prev = X`.



Deletion in Doubly Linked Lists

Deletion Operation

Removing a node `p` involves:

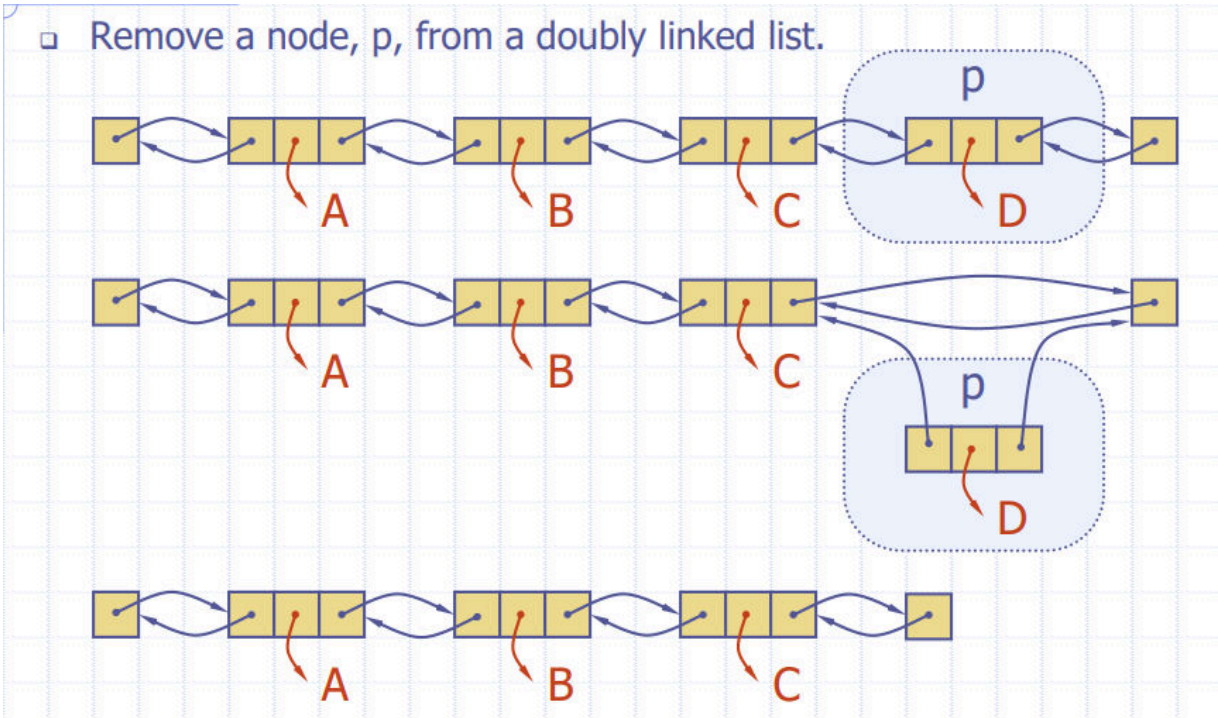
1. **Adjusting Pointers:**
   - Update the previous node's (`p.prev`) next pointer to point to `p`'s next node.
   - Update the next node's (`p.next`) previous pointer to point to `p`'s previous node.
2. **Removing the Node:**
   - Once the pointers are reassigned, node `p` is effectively removed and can be garbage collected.

> **DELETION:** Simplified by the fact that each node directly references both its neighbors, allowing immediate pointer adjustments without a complete list traversal.

Example

For a list A → B → C → D, removing node C leads to:

- A → B → D, where `B.next` is updated to D and `D.prev` is updated to B.



Implementation Considerations in Java

- **Nested Node Class:**
  Typically, the node is implemented as a private inner class within the doubly linked list class to encapsulate its structure.
- **Pointer Management:**
  Ensuring both `prev` and `next` pointers are correctly updated during insertion and deletion is crucial to maintain list integrity.
- **Boundary Operations:**
  The use of header and trailer nodes minimizes edge-case errors when performing operations at the beginning or end of the list.

```
public class DoublyLinkedList<E> {

    // Nested Node class representing each node in the doubly linke
d list
    private static class Node<E> {
        E element;      // The data stored in the node
        Node<E> prev;   // Pointer to the previous node
        Node<E> next;   // Pointer to the next node

        public Node(E element, Node<E> prev, Node<E> next) {
            this.element = element;
            this.prev = prev;
            this.next = next;
        }
    }

    private Node<E> head; // Points to the first node in the list
    private Node<E> tail; // Points to the last node in the list
    private int size;     // Number of elements in the list

    // Constructor: Initializes an empty doubly linked list
    public DoublyLinkedList() {
        head = null;
        tail = null;
        size = 0;
    }

    // Insert a new element at the head of the list
    public void insertAtHead(E element) {
        Node<E> newNode = new Node<>(element, null, head);
        if (head != null) {
            head.prev = newNode;
        } else {
            // List was empty, so newNode becomes both head and tai
l
```

```
            tail = newNode;
        }
        head = newNode;
        size++;
    }

    // Insert a new element at the tail of the list
    public void insertAtTail(E element) {
        Node<E> newNode = new Node<>(element, tail, null);
        if (tail != null) {
            tail.next = newNode;
        } else {
            // List was empty, so newNode becomes both head and tai
l
            head = newNode;
        }
        tail = newNode;
        size++;
    }

    // Insert a new element immediately after a given node 'p'
    public void insertAfter(Node<E> p, E element) {
        if (p == null) {
            throw new IllegalArgumentException("Given node cannot b
e null");
        }
        Node<E> newNode = new Node<>(element, p, p.next);
        if (p.next != null) {
            p.next.prev = newNode;
        } else {
            // p is the tail, so update tail to newNode
            tail = newNode;
        }
        p.next = newNode;
        size++;
    }
```

```
    // Remove a given node 'p' from the list and return its element
    public E remove(Node<E> p) {
        if (p == null) {
            throw new IllegalArgumentException("Given node cannot b
e null");
        }
        if (p.prev != null) {
            p.prev.next = p.next;
        } else {
            // p is the head
            head = p.next;
        }
        if (p.next != null) {
            p.next.prev = p.prev;
        } else {
            // p is the tail
            tail = p.prev;
        }
        size--;
        return p.element;
    }

    // Return the current size of the list
    public int getSize() {
        return size;
    }

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

    // Utility method to print the list elements in forward and bac
kward order
    public void printList() {
```

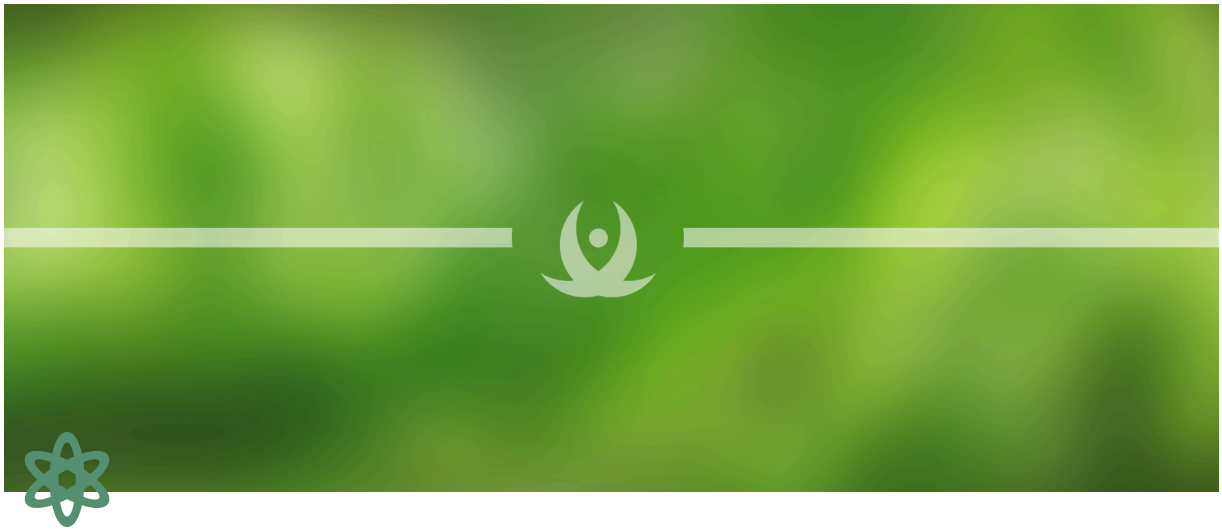
```
        Node<E> current = head;
        System.out.print("Forward: ");
        while (current != null) {
            System.out.print(current.element + " ");
            current = current.next;
        }
        System.out.println();
        current = tail;
        System.out.print("Backward: ");
        while (current != null) {
            System.out.print(current.element + " ");
            current = current.prev;
        }
        System.out.println();
    }

    // Getter for the head node (useful for operations like insertA
fter)
    public Node<E> getHead() {
        return head;
    }
}
```

Final Summary & Takeaways

- **Bidirectional Traversal:**
  Doubly linked lists enable efficient traversal in both directions due to nodes having pointers to both previous and next nodes.
- **Efficient Updates:**
  Insertion and deletion operations are streamlined with direct pointer manipulations, eliminating the need for full traversals.
- **Simplified Boundaries:**
  The use of header and trailer sentinel nodes simplifies operations at the boundaries of the list.

- **Java Implementation:**
  Implementing doubly linked lists in Java typically involves a nested node class and careful pointer management to maintain structural integrity.

Understanding these principles is essential for effectively implementing and manipulating doubly linked lists in various applications.



6. Lists and Iterators

Lists are abstract data types (ADTs) that represent ordered sequences of elements. Java provides a robust interface for lists and iterators.

- **java.util.List ADT:**

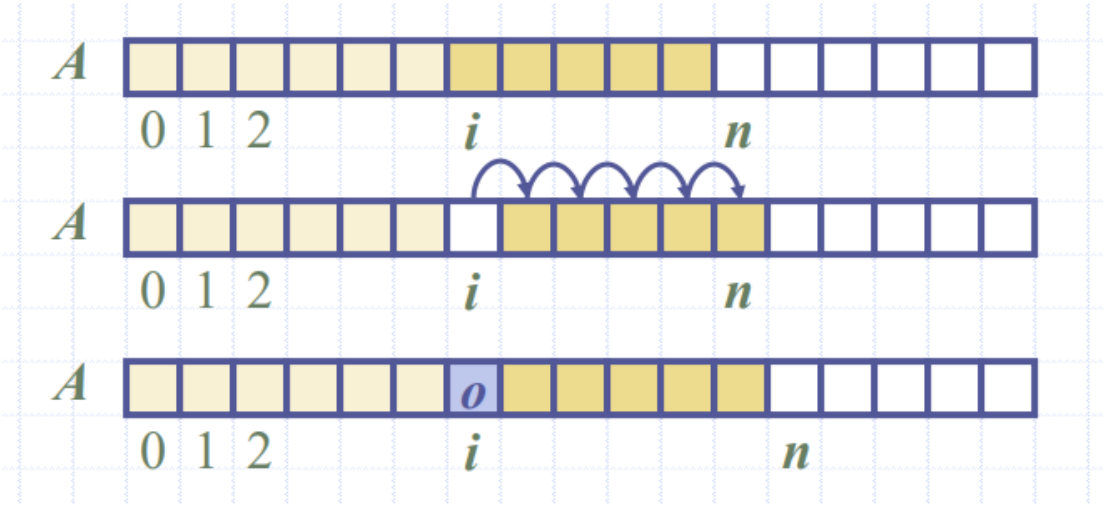
  The List interface in Java defines methods for managing ordered collections, such as:

  - Methods for retrieving an element (`get(i)`) or modifying an element (`set(i, e)`).
  - Methods for inserting an element at a specific index and removing an element from the list.

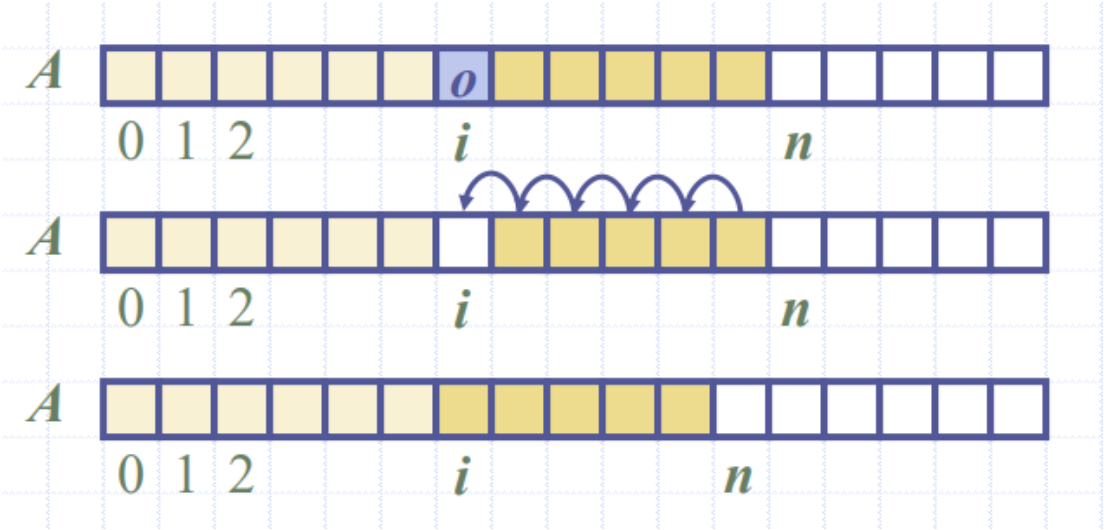
- **Array Lists:**

  An obvious choice for implementing a list ADT is to use an array:

  - Each element is stored in an array slot (e.g., `A[i]` holds the element at index `i`).
  - The `get(i)` and `set(i, e)` operations are straightforward, as they directly access the array element at index `i`.
  - **Insertion:**
    In an operation like `add(i, o)`, you must shift all elements from index `i` to the end of the array forward by one. In the worst case (inserting at index 0), this takes $O(n)$ time.



- **Removal:**
  Removing an element at index `i` requires shifting all elements from index `i+1` to the end backward by one to fill the gap, again taking $O(n)$ time in the worst case.



- **Performance Considerations:**

  In an array-based dynamic list:

  - Space used is $O(n)$.
  - Indexing is $O(1)$ time.
  - Add and remove operations can take $O(n)$ time due to shifting elements.

- When the underlying array is full, it can be replaced with a larger array to accommodate new elements.

- **Growable Array-Based Array List:**

  When implementing a dynamic array (or growable array-based list), a common challenge is what to do when the underlying array becomes full. Two strategies are commonly used: the Incremental Strategy and the Doubling Strategy. The performance of these strategies can be analyzed by examining the total time spent on all push operations, especially the cost of copying elements when resizing.

Incremental Strategy

In the incremental strategy, when the array is full, you increase its size by a constant amount, say `c`. For each push operation that requires resizing, you perform the following steps:

1. Allocate a new array with size equal to (current size + c).
2. Copy all the elements from the old array to the new array.
3. Insert the new element.

Assume you start with an initial array size of `s`. Every time the array fills up, you need to copy all its elements. If you perform a total of `n` push operations, then the number of times you need to resize is roughly proportional to `n / c`. For the k-th resize, the array size is approximately `s + k * c`, and copying these elements takes `O(s + k * c)` time.

The total time spent on copying over all resizes is approximately:

$$T_{copy} \approx \sum_{k=1}^{m} (s + k \cdot c)$$

where $m \approx \frac{n}{c}$. This sum is an arithmetic series whose total is proportional to:

$$m \cdot s + c \cdot \frac{m(m+1)}{2} = O\left(\frac{n}{c} \cdot s + c \cdot \left(\frac{n}{c}\right)^2\right)$$

If we assume that the initial size s and constant c are fixed, then the dominant term becomes:

$$O\left(\frac{n^2}{c}\right)$$

Thus, the incremental strategy results in an overall worst-case time of $O(n^2)$ for performing n push operations, meaning that the amortized cost per operation is $O(n)$ in the worst-case scenario when many resizes occur.

Doubling Strategy

In the doubling strategy, when the array becomes full, its size is doubled. This means that the sizes grow exponentially: $s, 2s, 4s, 8s, \ldots$ Each resize operation involves copying all elements from the old array to the new array.

If the array is doubled each time, the total cost of copying can be analyzed as follows:

- When the array first resizes from size s to $2s$, you copy s elements.
- The next resize copies $2s$ elements.
- Then $4s$ elements, and so on.

If n push operations are performed, the total number of elements ever copied is:

$$s + 2s + 4s + \cdots + (\text{largest power of 2 less than or equal to } n)$$

This is a geometric series that sums up to approximately:

$$2n - s$$

which is $O(n)$.

Because the total extra cost of all resizing operations is $O(n)$ and these resizes happen infrequently (only logarithmically many times), the amortized cost per push operation is:

$$\frac{O(n)}{n} = O(1)$$

Thus, the doubling strategy is much more efficient in practice, with an amortized time of $O(1)$per operation and an overall time of $O(n)$ for n operations.

- **Amortized Analysis:**

  The incremental strategy incurs a higher cost because the array is replaced many times, while the doubling strategy minimizes the number of times the array is reallocated. The amortized time for the doubling strategy is $O(1)$ per operation. We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e. $T(n)/n$

Positional Lists

A positional list ADT provides an abstraction for a sequence of elements with the additional ability to identify each element's location (or position) within the list.

- **Concept of Position:**
  A position is a marker or token that represents a location in the list. A position remains valid until it is explicitly deleted.
  - It supports a method like `P.getElement()` to retrieve the element at that position.
- **Implementation:**
  The most natural way to implement a positional list is with a doubly linked list, as it naturally supports insertions and deletions at arbitrary positions without needing to traverse the entire list.
- **Methods:**
  The positional list ADT includes accessor methods for retrieving elements and update methods for inserting, replacing, or removing elements.

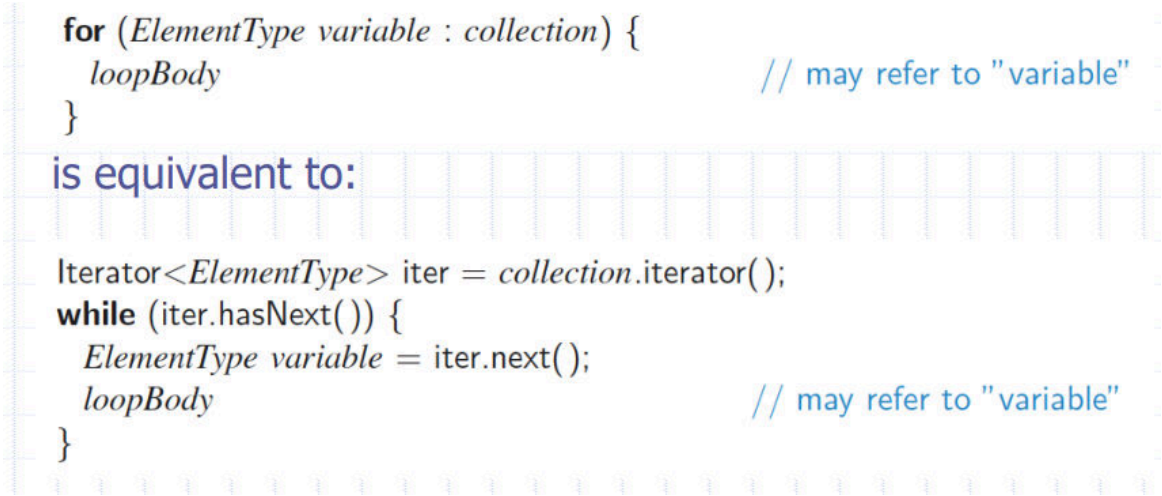
A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last( ))	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Iterators and the Iterable Interface

Iterators provide a standardized way to traverse through elements in a collection without exposing the underlying representation. An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

- **Iterator Concept:**
  An iterator abstracts the process of scanning through a sequence of elements one at a time.
- **The Iterable Interface:**
  In Java, the `Iterable` interface is parameterized and includes a single method:
  - `iterator()`: Returns an iterator over the collection's elements.
  - Each call to `iterator()` returns a new iterator, allowing multiple or simultaneous traversals.



- **For-each Loop:**
  The for-each loop syntax in Java leverages the Iterable interface to simplify iteration over collections. This loop internally calls the iterator to process each element.

Abstract Data Types (ADTs)

- **Definition:** An abstract data type (ADT) is a conceptual model for a data structure that specifies:
  - **Data Stored:** The type of data maintained.
  - **Operations:** The functions or methods provided (e.g., insertion, deletion).
  - **Error Conditions:** How error situations are handled (e.g., attempting an invalid operation).
- **Example:**

  An ADT for a simple stock trading system might include:
  - **Data:** Buy/sell orders.
  - **Operations:** `orderBuy(stock, shares, price)`, `orderSell(stock, shares, price)`, `cancel(order)`.
  - **Errors:** Handling cases such as non-existent stocks or orders.

The Stack ADT

- **Definition:** A stack is an ADT that stores a collection of objects, supporting operations based on the *Last-In, First-Out (LIFO)* principle.

  Think of a spring-loaded plate dispenser: the last plate placed on top is the first one removed.

- **Primary Operations:**
  - **push(object):** Inserts an element onto the top of the stack.
  - **pop():** Removes and returns the element most recently added.
- **Auxiliary Operations:**
  - **top():** Returns the top element without removing it.
  - **size():** Returns the number of elements in the stack.
  - **isEmpty():** Checks if the stack is empty.

Stack Interface in Java

The following Java interface outlines the Stack ADT. Note that it is different from Java's built-in `java.util.Stack`.

```
public interface Stack<E> {
    int size();
    boolean isEmpty();
    E top();
    void push(E element);
    E pop();
}
```

- **Key Point:**
  In this implementation, methods `top()` and `pop()` return `null` if the stack is empty instead of throwing exceptions.

Handling Error Conditions

- **Exceptions vs. Returning Null:**
  Instead of using exceptions for error conditions (such as performing `pop` or `top` on

an empty stack), the design here chooses to return `null`. This simplifies error handling in many scenarios, though it requires the user to check for `null` values.

Applications of Stacks

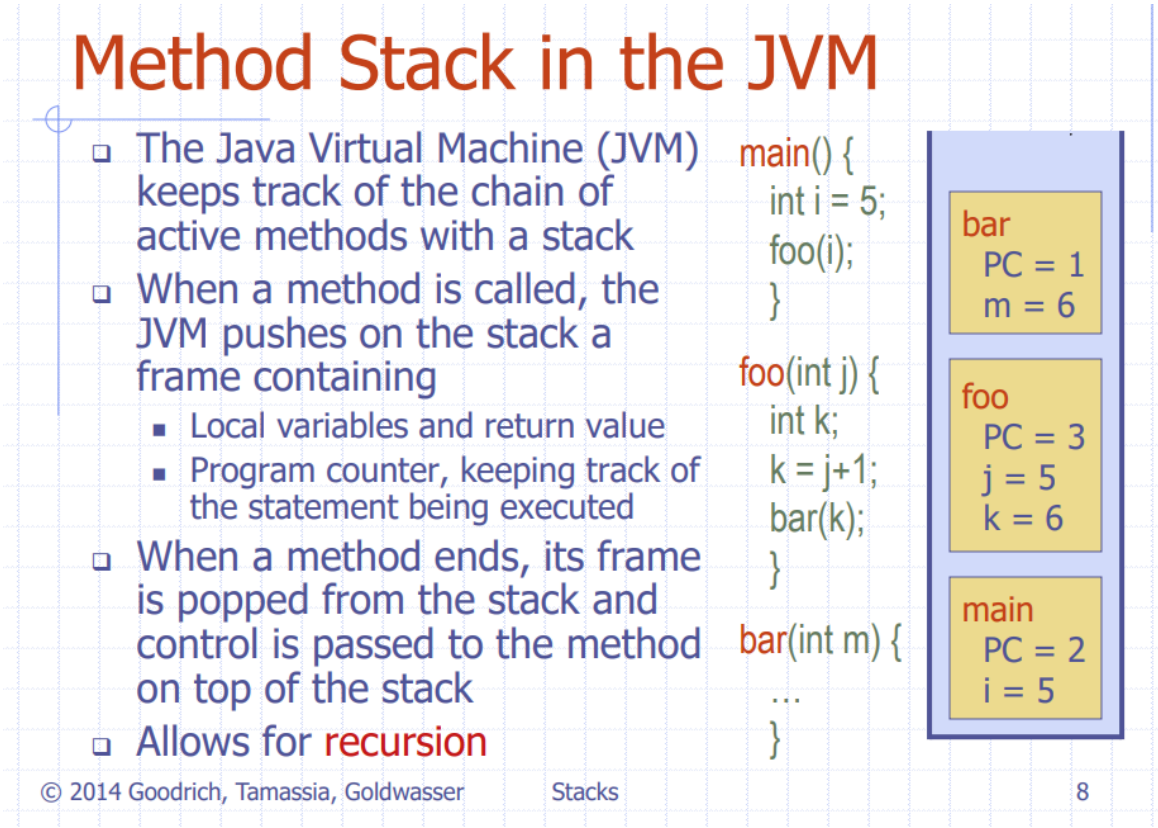
- **Direct Applications:**
  - **Web Browsers:** Storing page-visited history.
  - **Text Editors:** Implementing undo operations.
  - **JVM:** Maintaining the chain of active method calls (call stack).
- **Indirect Applications:**
  - Serving as auxiliary structures in various algorithms.
  - Acting as components within more complex data structures.

Method Stack in the JVM

- **Purpose:**

  The JVM uses a stack to keep track of active method calls.
- **Mechanism:**
  - **Method Invocation:** When a method is called, a *frame* containing local variables, the return address, and a program counter is pushed onto the stack.
  - **Method Return:** Upon completion, the frame is popped, and control is returned to the previous method.
- **Supports Recursion:**

  Each recursive call creates a new frame, allowing the method to call itself multiple times while preserving state.



Array-Based Stack Implementation

- **Concept:**

  One common way to implement a stack is by using an array. Elements are stored from left to right with a variable tracking the index of the top element.

- **Pseudocode Examples:**

  Size Operation:

```
algorithm size():
    return t + 1
```

  Pop Operation:

```
algorithm pop():
    if isEmpty() then
        return null
```

```
    else:
        t = t - 1
        return S[t + 1]
```

Push Operation:

```
algorithm push(o):
    if t == S.length - 1 then
        throw IllegalStateException  // Stack is full
    else:
        t = t + 1
        S[t] = o
```

- **Limitation:**

  The array-based implementation has a fixed maximum size, meaning it cannot expand dynamically without additional logic.

Performance and Limitations

- **Performance:**
  - **Space Complexity:** $O(n)$, where n is the number of elements.
  - **Time Complexity:** $O(1)$ for each operation (push, pop, top).
- **Limitations:**
  - A fixed-size array means the maximum size must be predetermined.
  - Pushing an element on a full stack will result in an exception (or error condition).

Parentheses Matching

- **Problem Statement:**

  Check whether every opening delimiter (i.e., `(`, `{`, `[`) in an expression has a corresponding matching closing delimiter (i.e., `)`, `}`, `]`).

- **Examples:**
  - **Correct:** `( )(( )){([( )])}`

- **Incorrect:** `)( ( )){([( )])}`, `({[ ])}`, `(`

- **Java Implementation:**

```
public static boolean isMatched(String expression) {
    final String opening = "({["; // opening delimiters
    final String closing = ")}]"; // respective closing delimiters
    Stack<Character> buffer = new LinkedStack<>();
    for (char c : expression.toCharArray()) {
        if (opening.indexOf(c) != -1) // left delimiter
            buffer.push(c);
        else if (closing.indexOf(c) != -1) { // right delimiter
            if (buffer.isEmpty()) // nothing to match with
                return false;
            if (closing.indexOf(c) != opening.indexOf(buffer.pop
()))
                return false; // mismatched delimiter
        }
    }
    return buffer.isEmpty(); // check if all delimiters matched
}
```

HTML Tag Matching

- **Objective:**

  Validate that every HTML opening tag has a corresponding closing tag.

- **Example HTML:**

```
<body>
  <center>
    <h1>The Little Boat</h1>
  </center>
  <p>The storm tossed the little boat...</p>
  <ol>
    <li>Will the salesman die?</li>
```

```
    <li>What color is the boat?</li>
    <li>And what about Naomi?</li>
  </ol>
</body>
```

- **Java Implementation:**

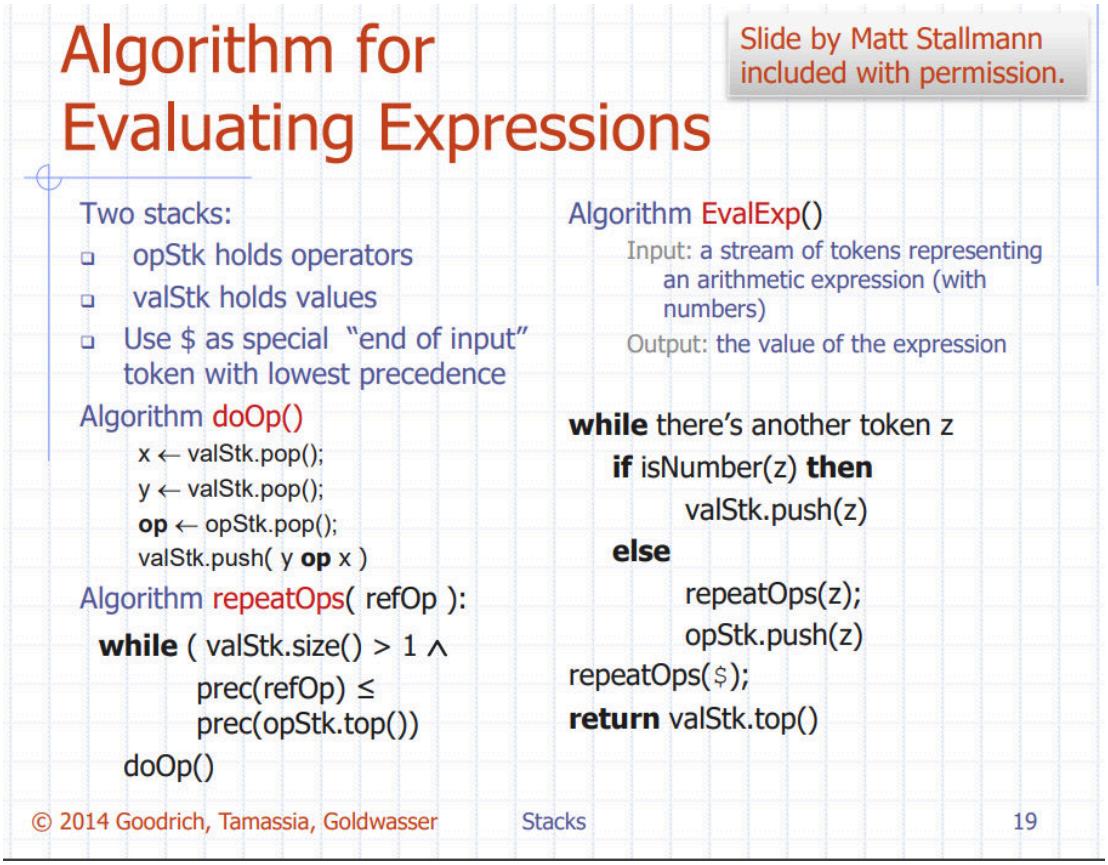
```
public static boolean isHTMLMatched(String html) {
    Stack<String> buffer = new LinkedStack<>();
    int j = html.indexOf('<'); // find first '<' character
    while (j != -1) {
        int k = html.indexOf('>', j + 1); // find next '>' characte
r
        if (k == -1)
            return false; // invalid tag
        String tag = html.substring(j + 1, k); // extract tag
        if (!tag.startsWith("/")) { // opening tag
            buffer.push(tag);
        } else { // closing tag
            if (buffer.isEmpty())
                return false; // no tag to match
            if (!tag.substring(1).equals(buffer.pop()))
                return false; // mismatched tag
        }
        j = html.indexOf('<', k + 1); // find next '<'
    }
    return buffer.isEmpty(); // check if all tags matched
}
```

Evaluating Arithmetic Expressions

- **Overview:**

  Use stacks to evaluate arithmetic expressions that respect operator precedence and associativity.

- **Key Concepts:**
  - **Operator Precedence:** Multiplication and division have higher precedence than addition and subtraction.
  - **Associativity:** Operators with the same precedence are evaluated from left to right.
- **Algorithm Outline Using Two Stacks:**
  - **Stacks Used:**
    - **opStk:** Holds operators.
    - **valStk:** Holds numerical values.
  - **Core Functions:**
    - **doOp():** Pops the top two values and one operator, performs the operation, and pushes the result.
    - **repeatOps(refOp):** While the operator on the top of the operator stack has higher or equal precedence to the current token (`refOp`), perform the operation.
    - **EvalExp():** Processes the stream of tokens, pushing numbers onto `valStk` and handling operators via `repeatOps`, finally returning the final value.

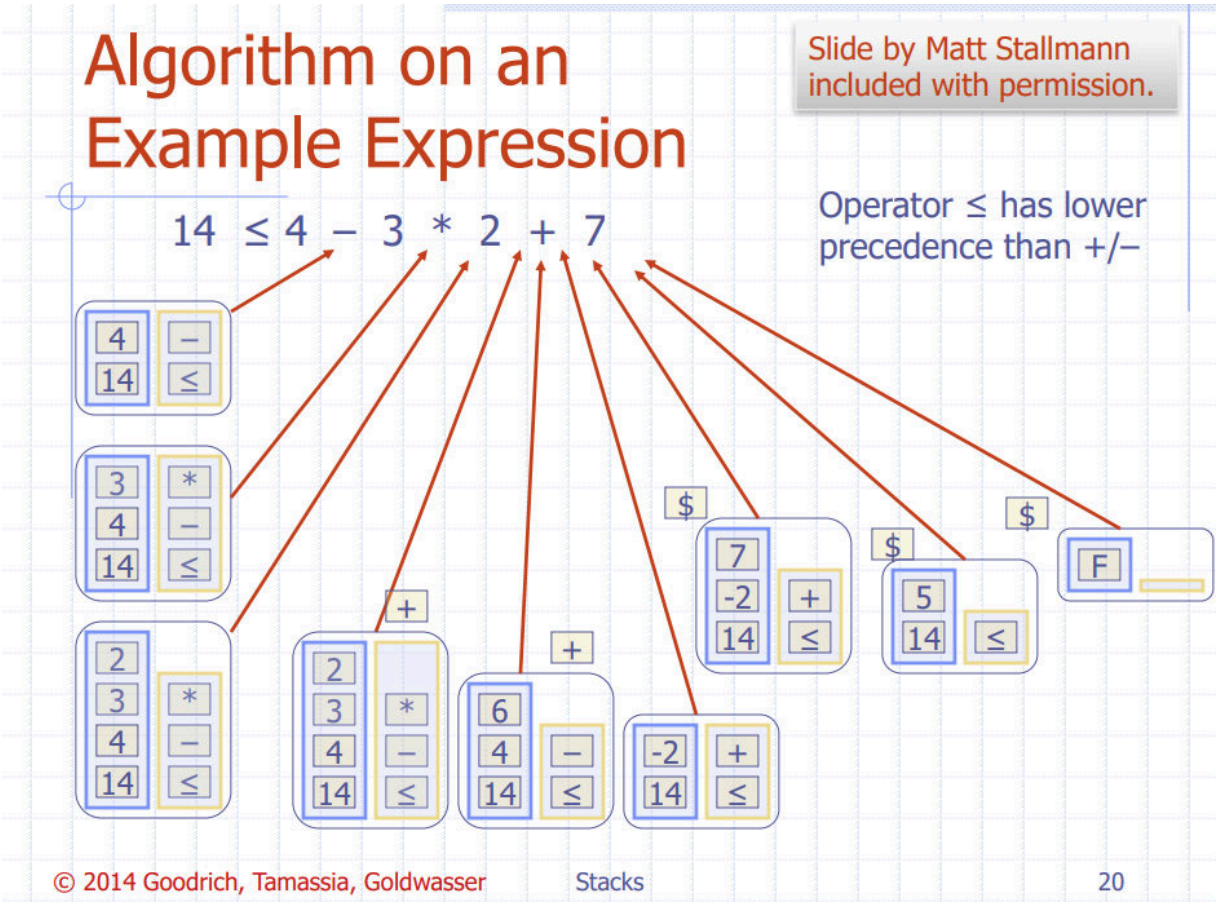


- **Example Expression:**

  For the expression:

  ```
  14 - 3 * 2 + 7
  ```

  The algorithm ensures that multiplication is performed before addition and subtraction, following the correct order of operations.

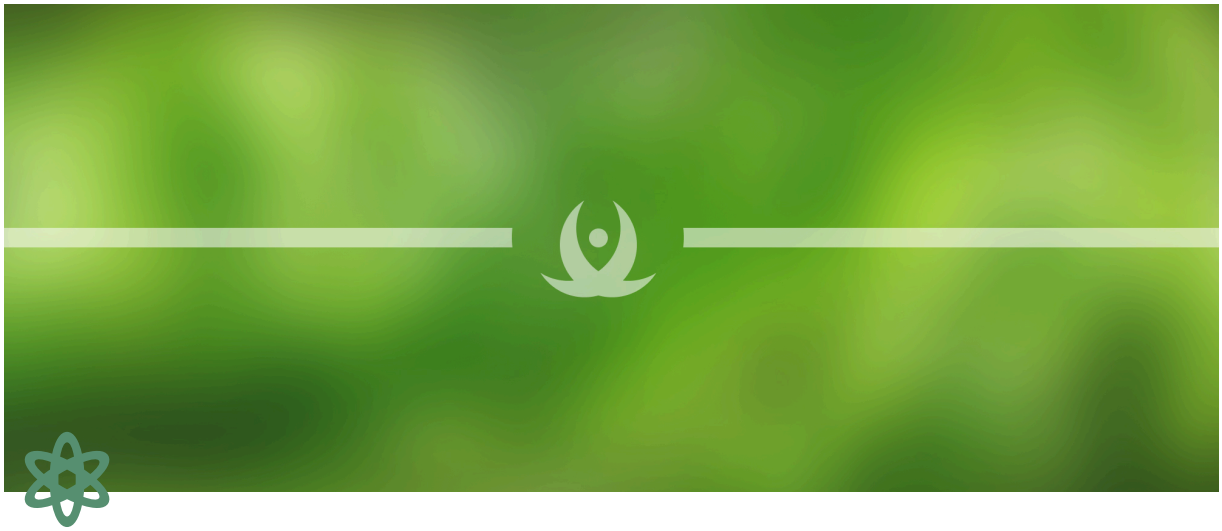


Summary of Key Concepts

- Singly linked lists are efficient for head insertions but inefficient for tail removals due to traversal requirements.
- Doubly linked lists support bidirectional traversal, enabling efficient insertions and deletions anywhere in the list.
- Array-based list implementations provide fast random access but may require shifting elements for insertions and removals.
- Dynamic array lists can be implemented using incremental or doubling strategies, with the doubling strategy being more efficient in terms of amortized cost.
- Positional lists add a layer of abstraction by providing explicit positions for elements, naturally implemented via doubly linked lists.
- Iterators and the Iterable interface enable clean and abstracted traversal of collections, supporting the for-each loop syntax in Java.

- **Stack ADT:** A stack is a LIFO data structure supporting push, pop, and auxiliary operations like top, size, and isEmpty.
- **Implementation Approaches:** Can be implemented using arrays (with fixed capacity) or linked structures. Array-based implementations are efficient but limited by size.
- **Practical Applications:** Stacks are used in method call management, expression evaluation, and various algorithms including delimiter matching.
- **Error Handling:** Instead of exceptions, returning `null` is one strategy for handling operations on empty stacks.

Enjoy your review and happy coding!



7. Queues and Their Applications

The Queue ADT

- **Definition:** A queue is an abstract data type that stores elements in a First-In, First-Out (FIFO) order.
- **Main Operations:**
  - **enqueue(object):** Inserts an element at the rear (end) of the queue.
  - **dequeue():** Removes and returns the element at the front of the queue.
- **Auxiliary Operations:**
  - **first():** Returns (without removing) the element at the front.
  - **size():** Returns the number of elements currently stored.
  - **isEmpty():** Checks whether the queue is empty.
- **Boundary Condition:**
  When performing `dequeue()` or `first()` on an empty queue, the operations return `null`.

Queue Operations and Examples

- **Example Sequence:**
  Consider the following sequence of operations on a queue:
  1. `enqueue(5)` → Queue becomes: (5)
  2. `enqueue(3)` → Queue becomes: (5, 3)
  3. `dequeue()` → Returns 5; Queue becomes: (3)
  4. `enqueue(7)` → Queue becomes: (3, 7)
  5. `dequeue()` → Returns 3; Queue becomes: (7)
  6. `first()` → Returns 7; Queue remains: (7)
  7. `dequeue()` → Returns 7; Queue becomes: ()
  8. `dequeue()` → Returns `null` (queue is empty)
  9. `isEmpty()` → Returns `true`
  10. Further operations (e.g., enqueues and size checks) follow similarly.

Array-Based Queue Implementation

- **Concept:**
  A common implementation of a queue is to use an array in a circular manner. This allows efficient use of storage when elements are enqueued and dequeued repeatedly.
- **Key Variables:**
  - **f:** Index of the front element.
  - **sz:** Number of stored elements.
  - **r:** Computed as `(f + sz) mod N`, representing the index of the first empty slot (i.e., the rear of the queue).
- **Operations with Modulo Arithmetic:**
  - **size():**

```
algorithm size():
    return sz
```

- **isEmpty():**

```
algorithm isEmpty():
    return (sz == 0)
```

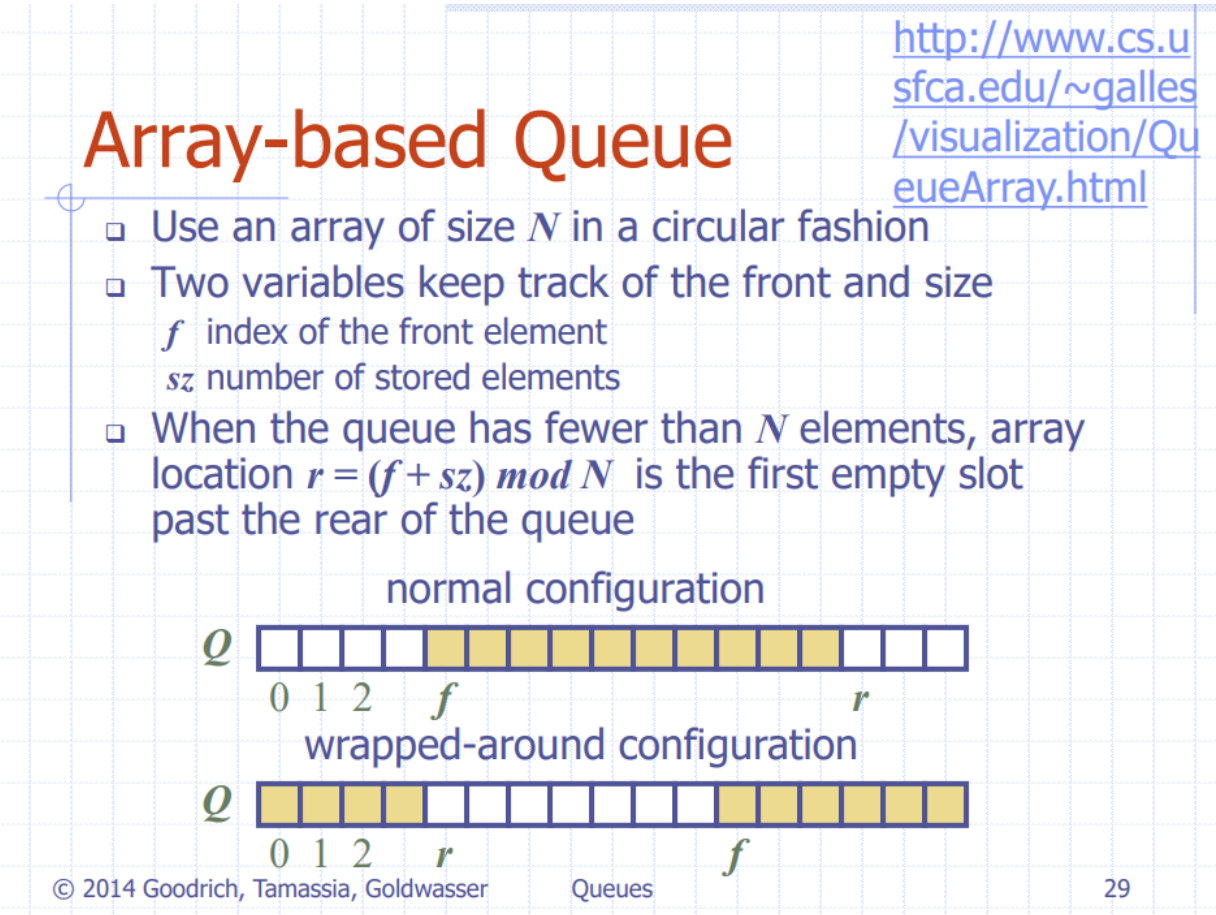
- **enqueue(o):**

```
algorithm enqueue(o):
    if size() == N - 1 then
        throw IllegalStateException  // The array is full
    else
        r = (f + sz) mod N
        Q[r] = o
        sz = sz + 1
```

- **dequeue():**

```
algorithm dequeue():
    if isEmpty() then
        return null
    else:
        o = Q[f]
        f = (f + 1) mod N
        sz = sz - 1
        return o
```

- **Note:**
  The enqueue operation throws an exception if the underlying array is full. This is an implementation-specific behavior.



Queue Interface in Java

The following Java interface defines the Queue ADT in a manner consistent with the described operations:

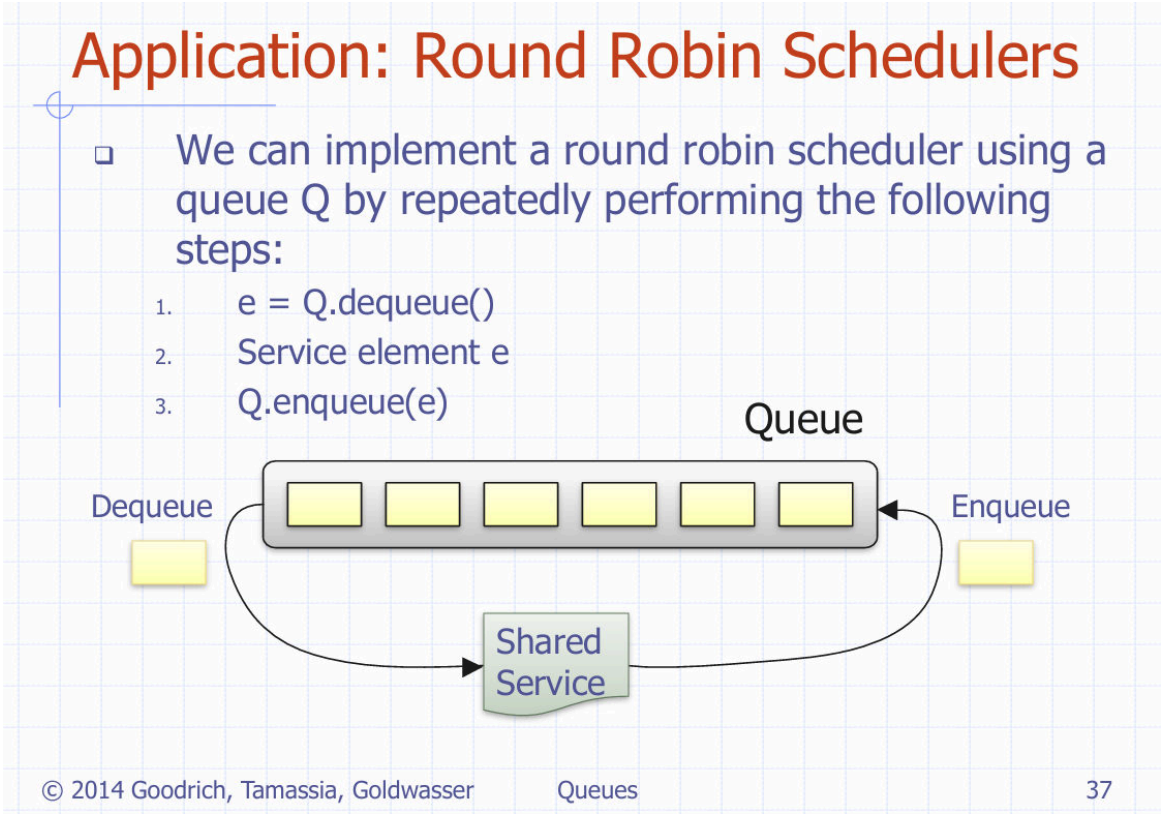
```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    E first();
    void enqueue(E e);
    E dequeue();
}
```

- **Assumptions:**
  - Methods `first()` and `dequeue()` return `null` when the queue is empty.

- The interface supports generic types for flexibility.

Applications of Queues

- **Direct Applications:**
  - **Waiting Lists and Bureaucracy:**
    Managing tasks or clients in the order of arrival.
  - **Access to Shared Resources:**
    Examples include printer queues.
  - **Multiprogramming:**
    Managing processes in an operating system.
- **Indirect Applications:**
  - **Auxiliary Data Structure:**
    Queues are used within algorithms for breadth-first search (BFS) and other operations.
  - **Component in Other Data Structures:**
    Serving as the backbone for complex structures or systems.
- **Application Example: Round Robin Scheduling**
  - **Method:**
    1. **Dequeue:** Remove the first element.
    2. **Service:** Process the dequeued element.
    3. **Enqueue:** Place the serviced element at the end of the queue.
  - **Purpose:**
    This scheduling technique is used to ensure fairness in process scheduling by cyclically rotating tasks.



Trees

A tree is an abstract data structure used to represent hierarchical relationships. It is composed of nodes connected by parent-child links. Trees are widely used in areas such as:

- Organization charts
- File systems
- Programming environments

Key Terminology

- **Root:** The unique node with no parent.
- **Internal Node:** A node with at least one child.
- **External Node (Leaf):** A node without any children.
- **Ancestors:** The sequence of nodes from a given node up to the root.

- **Depth:** The number of ancestors of a node.
- **Height:** The maximum depth among all nodes in the tree.
- **Descendant:** Any node that is below another node in the hierarchy.
- **Subtree:** A portion of a tree consisting of a node and all its descendants.

Tree ADT (Abstract Data Type)

The Tree ADT defines a set of operations to interact with tree structures:

- **Generic Methods:**
  - `size()` – returns the number of nodes
  - `isEmpty()` – checks if the tree is empty
  - `iterator()` – provides an iterator over nodes
  - `positions()` – returns an iterable collection of node positions
- **Accessor Methods:**
  - `root()` – returns the root node
  - `parent(p)` – returns the parent of node *p*
  - `children(p)` – returns the children of node *p*
  - `numChildren(p)` – returns the number of children of node *p*
- **Query Methods:**
  - `isInternal(p)` – checks if node *p* is internal
  - `isExternal(p)` – checks if node *p* is a leaf
  - `isRoot(p)` – checks if node *p* is the root

Tree Traversals

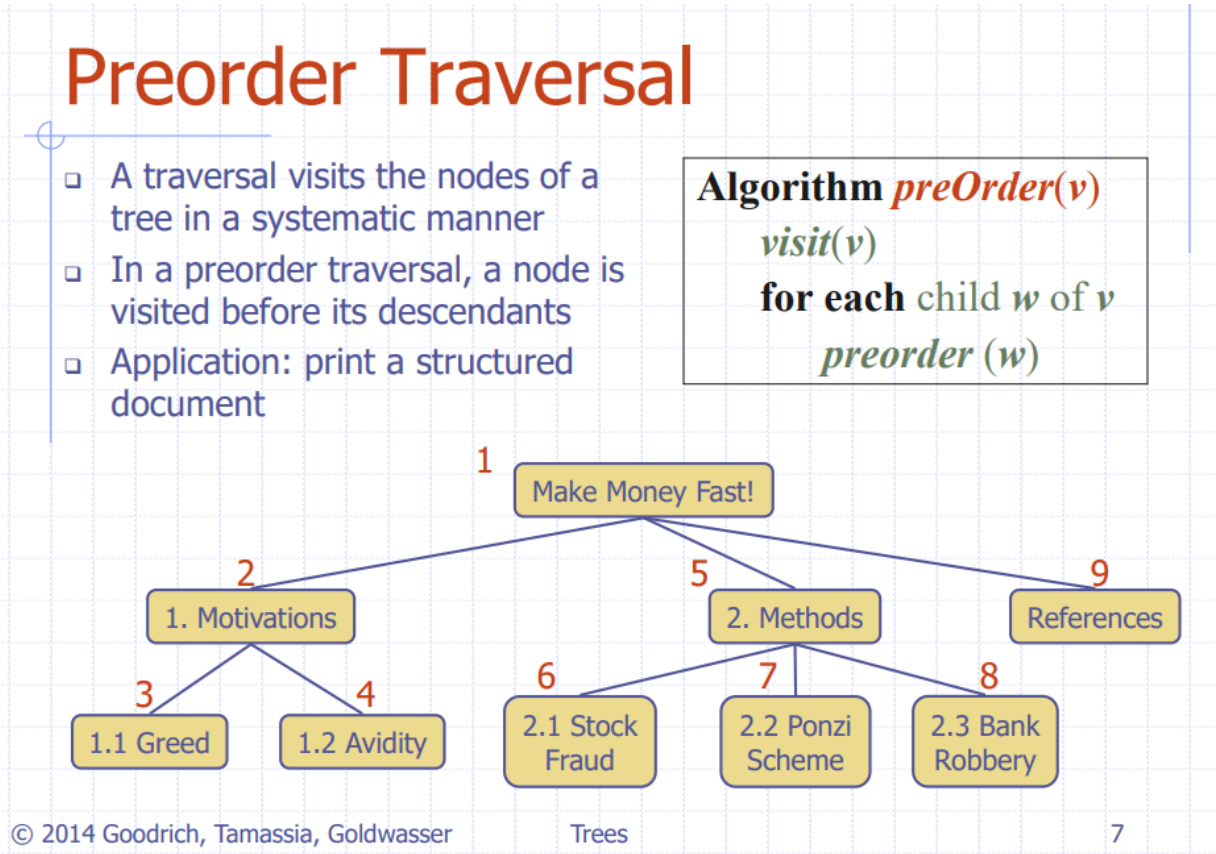
Preorder Traversal

Preorder traversal visits a node before its descendants:

1. Visit the node.

2. Recursively perform a preorder traversal on each child.

Use Case: Printing or serializing the tree structure.

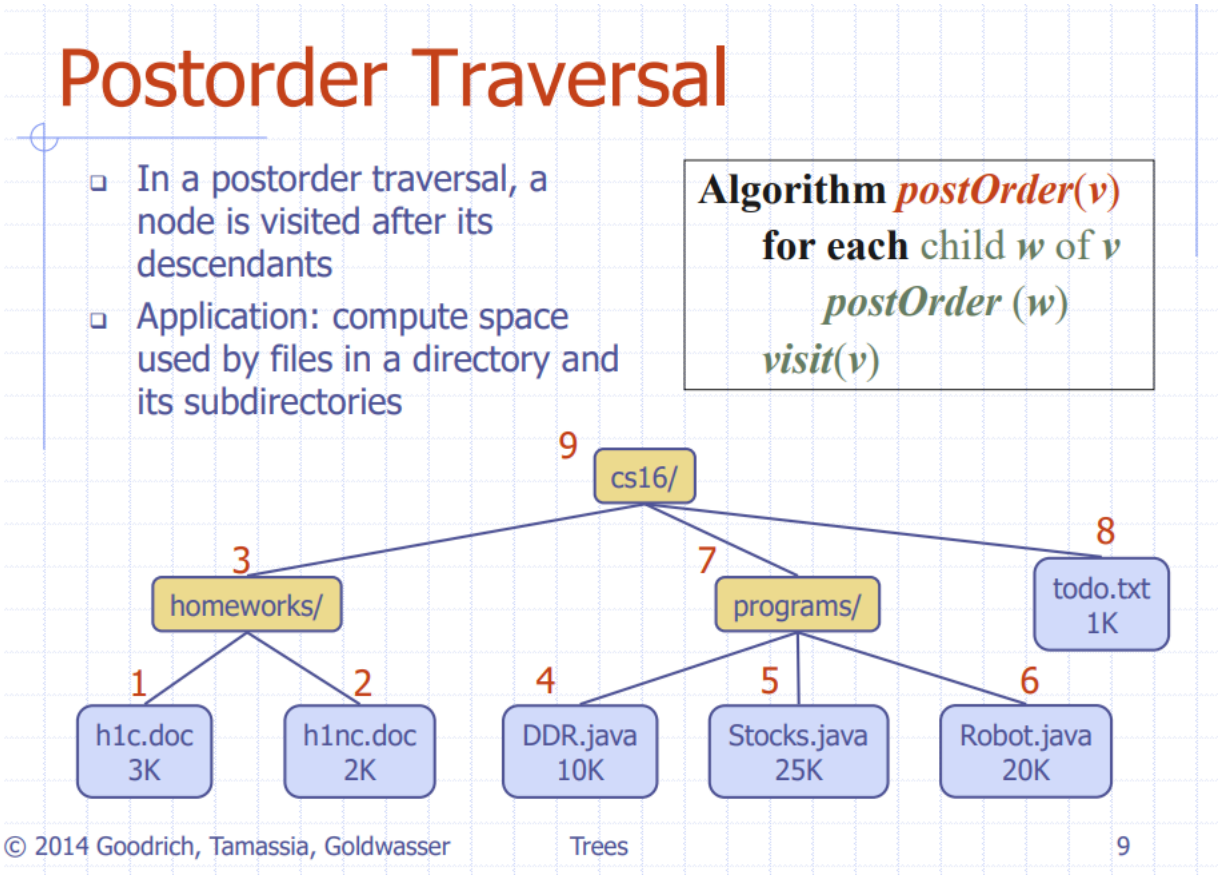


Postorder Traversal

Postorder traversal visits a node after its descendants:

1. Recursively perform a postorder traversal on each child.
2. Visit the node.

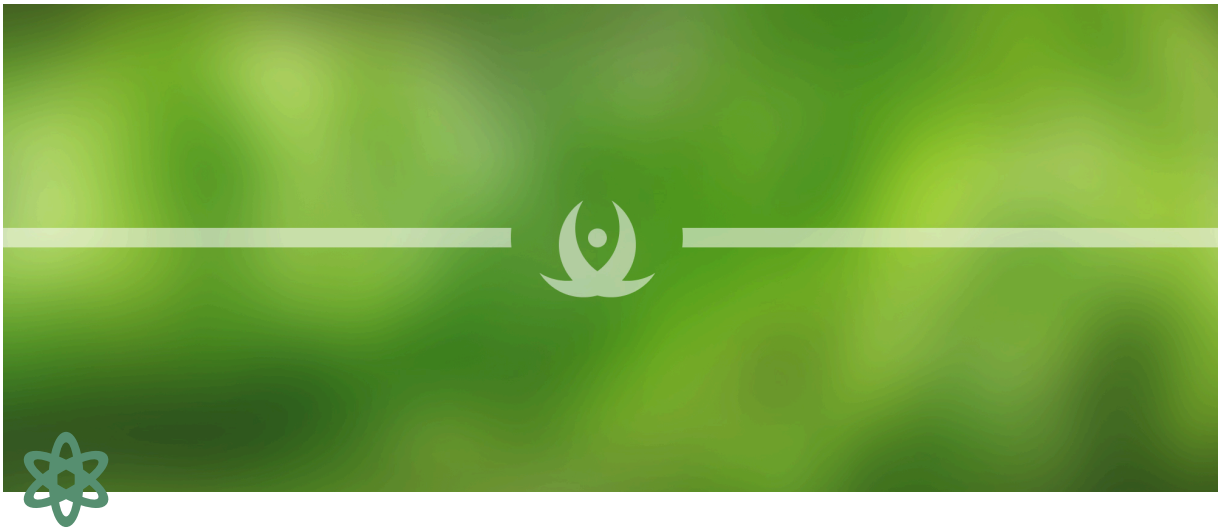
Use Case: Evaluating or aggregating data where child results are needed before processing the parent.



Summary and Takeaways

- **Queue ADT Overview:** Queues operate on a FIFO basis and support primary operations like `enqueue` and `dequeue`, along with useful auxiliary operations.
- **Implementation Insights:**
  - The circular array approach uses modulo arithmetic to efficiently manage the queue.
  - The array-based queue has limitations in size, which must be managed via exception handling or by dynamic resizing.
- **Real-World Relevance:** Queues are fundamental to many systems, from process scheduling in operating systems to managing resources in everyday applications.
- **Java Interface:** A clear interface helps encapsulate queue operations, ensuring consistency and flexibility for implementation.
- Trees represent hierarchical relationships with nodes and parent-child connections.

- Fundamental terms include root, internal/external nodes, depth, height, and subtrees.
- The Tree ADT provides a standard interface for tree operations.
- Preorder and postorder traversals are essential techniques, each suited for different applications.



8. Binary Trees and Binary Search Trees: A Structured Note

Objective & Scope

This note summarizes key concepts from the lecture on binary trees and binary search trees. It covers definitions, traversal methods, properties, and implementation strategies, supporting applications like arithmetic expression evaluation and decision-making processes.

Binary Trees

A binary tree is a tree structure where each internal node has at most two children, designated as the left and right child.

> **BINARY TREE:** A tree in which each internal node has at most two children (exactly two in a proper binary tree). A binary tree is either a single node or a root whose children (an ordered pair) are each the root of a binary tree.

- **Applications:** Arithmetic expressions, decision processes, and searching.

Inorder Traversal

In an inorder traversal, the process visits the left subtree, then the node, and finally the right subtree.

> **INORDER TRAVERSAL:** Visit left subtree → Node → Right subtree.

- **Pseudocode:**

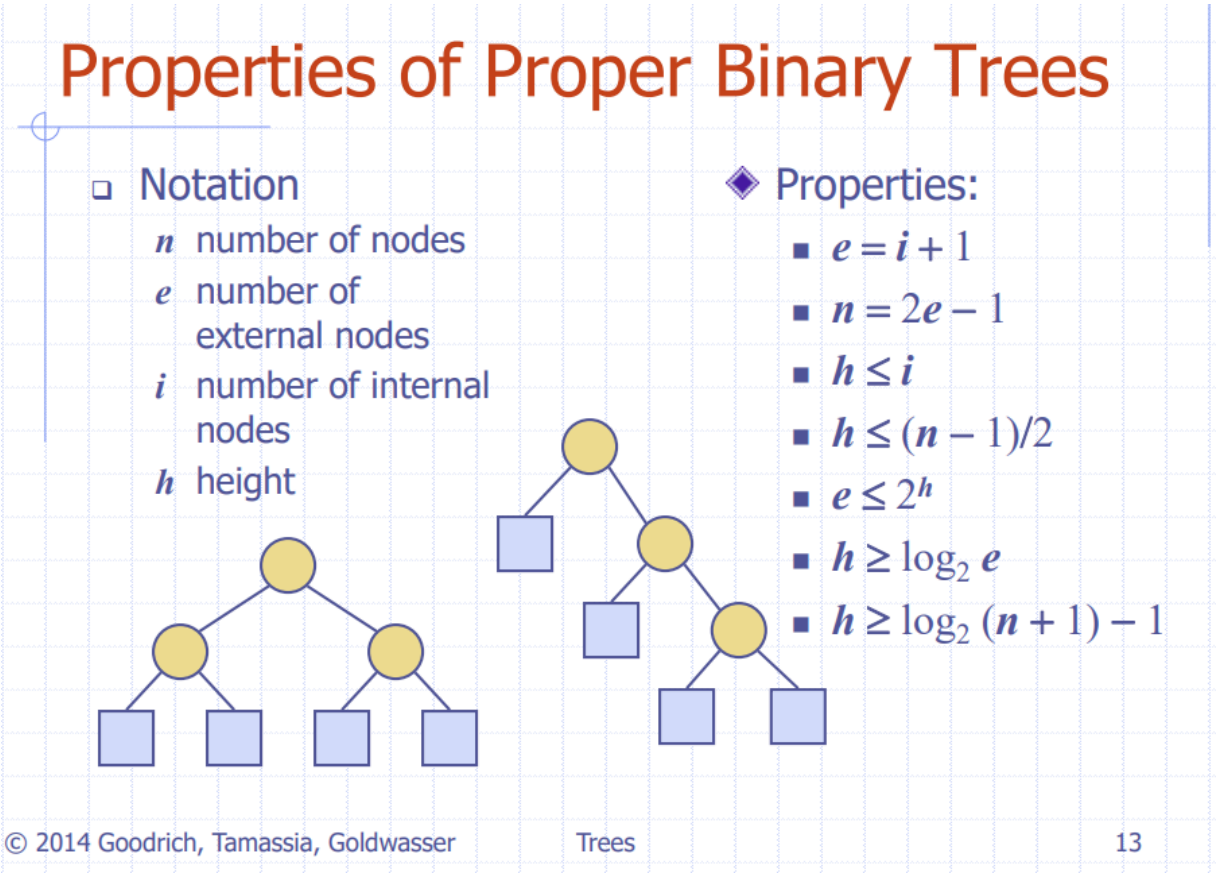
```
if left(v) ≠ null:
    inOrder(left(v))
visit(v)
if right(v) ≠ null:
    inOrder(right(v))
```

- **Use Case:** Often used to draw or represent binary trees in a sorted order.

Properties of Proper Binary Trees

Proper binary trees satisfy specific numerical relationships among their nodes.

> **Key Relationships:**
>
> - $e = i + 1$
> - $n = 2e - 1$
> - $h \leq i$, and $h \leq (n - 1)/2$
> - $e \leq 2^h$
> - $h \geq \log_2 e$ and $h \geq \log_2(n + 1) - 1$



BinaryTree ADT

The BinaryTree ADT extends the general Tree ADT with methods specific to binary trees.

> **Additional Methods:**
>
> - `left(p)`: Returns the left child of position *p*.
> - `right(p)`: Returns the right child of position *p*.
> - `sibling(p)`: Returns the sibling of position *p* (if it exists).

- It inherits methods such as `size()`, `isEmpty()`, `iterator()`, and `positions()`.

Decision Trees

Decision trees model decisions as a binary structure.

> **DECISION TREE:** A binary tree where internal nodes represent yes/no questions and external nodes represent decisions.

- **Example:** A dining decision tree guiding choices based on questions like "Want a fast meal?" leading to options such as fast food or sit-down restaurants.

Arithmetic Expression Trees

Arithmetic expression trees represent mathematical expressions where operators are internal nodes and operands are external nodes.

> **ARITHMETIC EXPRESSION TREE:** A binary tree with operators at internal nodes and operands at the leaves.

- **Printing Expressions:**
- Use a specialized inorder traversal that prints "(" before the left subtree and ")" after the right subtree.
- *Pseudocode Outline:*

```
printExpression(v):
    if left(v) ≠ null:
        print("(")
        printExpression(left(v))
    print(v.element)
    if right(v) ≠ null:
        printExpression(right(v))
        print(")")
```

- **Evaluating Expressions:**
- A postorder traversal computes the value by combining the results of the left and right subtrees using the operator at the node.
- *Pseudocode Outline:*

```
evalExpr(v):
    if isExternal(v):
        return v.element
    else:
        x = evalExpr(left(v))
```

```
        y = evalExpr(right(v))
        return x (operator) y
```

Linked Structures for Trees

Trees can be implemented using linked structures:

- **General Trees:** Nodes store an element, a reference to the parent, and a list of children.
- **Binary Trees:** Nodes store an element, a parent reference, and pointers to left and right children.

Array-Based Representation of Binary Trees

Binary trees may also be represented in an array where each node's position (or rank) is calculated as follows:

> **Rank Formula:**
>
> - rank(root) = 0
> - For a left child: rank = 2 * rank(parent) + 1
> - For a right child: rank = 2 * rank(parent) + 2

Binary Search Trees

Binary Search Trees (BSTs) are binary trees structured to facilitate efficient searching.

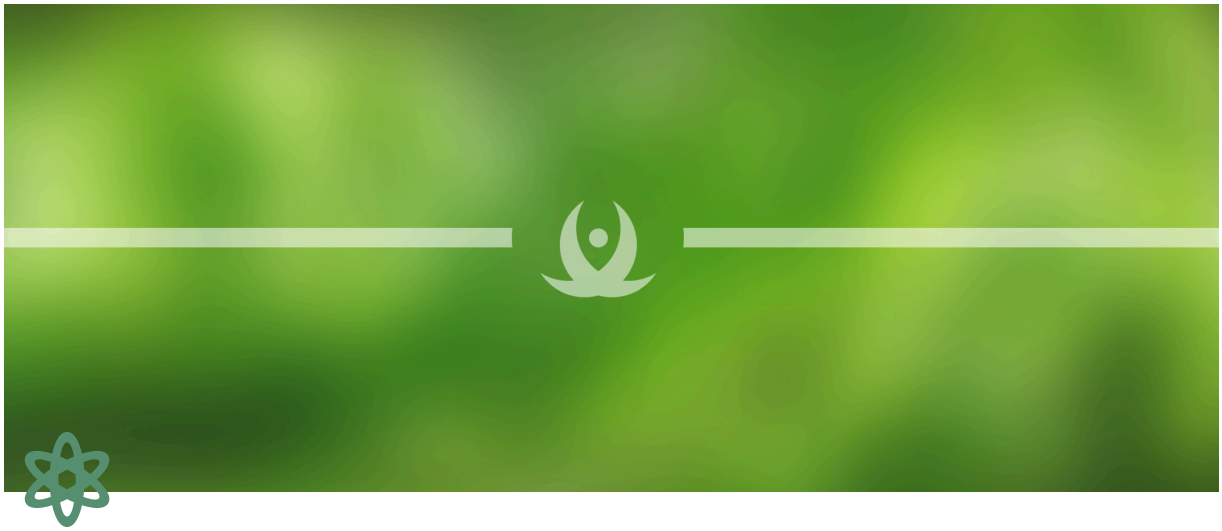
> **BINARY SEARCH TREE:** A binary tree in which, for every node, all elements in the left subtree are less than the node's element, and all elements in the right subtree are greater.

- **Applications:** Fast searching, sorting, and dynamic data management.

Final Summary & Takeaways

- **Binary Trees:** Provide a hierarchical structure with at most two children per node, used in diverse applications.

- **Traversals:** Inorder traversal is key for representing sorted order, while postorder is useful for evaluation tasks.
- **Properties:** Proper binary trees adhere to specific relationships between nodes, height, and structure.
- **Implementations:** Can be built using linked structures or array-based representations.
- **BSTs:** Leverage ordered structures to enable efficient search operations.



9. Binary Search Trees: Structure, Operations, and Complexity Analysis

Overview

This note explains the fundamentals of Binary Search Trees (BSTs), including their representations, defining properties, search operations, complexities, and other self-balancing tree variants (AVL, 2-4, Red-Black, and Splay Trees).

Linked Structure for Binary Trees

> **LINKED REPRESENTATION:** Each node contains its element (key), a pointer to its parent, and pointers to its left and right children. This design allows dynamic insertion, deletion, and reorganization of the tree by manipulating pointers.

Array-Based Representation of Binary Trees

> **ARRAY REPRESENTATION:** Nodes are stored in an array `A` using a rank-based scheme:

- The root is at index 0.
- For a node at index i, its left child is at $2i + 1$ and its right child at $2i + 2$.
  This approach is efficient for complete or nearly complete binary trees but can be less flexible if the tree frequently changes shape.

Binary Search Trees Overview

> **BINARY SEARCH TREE (BST):** A BST is a binary tree where each internal node v has all keys in the left subtree $\leq \text{key}(v)$, and all keys in the right subtree $\geq \text{key}(v)$. External nodes (leaves) do not store data, and an inorder traversal of a BST visits the keys in increasing (sorted) order.

BSTs are commonly used for fast searching, insertion, and deletion, forming the basis of many ordered data structures.

BST Search Operation

> **SEARCH PROCEDURE:** To locate a key k, begin at the root and compare k with the current node's key:
>
> - If k is less, go left.
> - If k is greater, go right.
> - If a leaf is reached without finding k, the key is absent.

Pseudocode:

```
TreeSearch(k, v):
     if v is external:
         return v

     if k < key(v):
       return TreeSearch(k, left(v))
     else if k == key(v):
        return v
```

```
    else:
       return TreeSearch(k, right(v))
```

Complexity and Performance

> **TIME COMPLEXITY:**
>
> - **Best case (balanced BST):** Height $h = O(\log n)$ leads to O($\log n$) operations for search, insertion, and deletion.
> - **Worst case (unbalanced BST):** Height $h = O(n)$ can degrade operations to O(n) time.

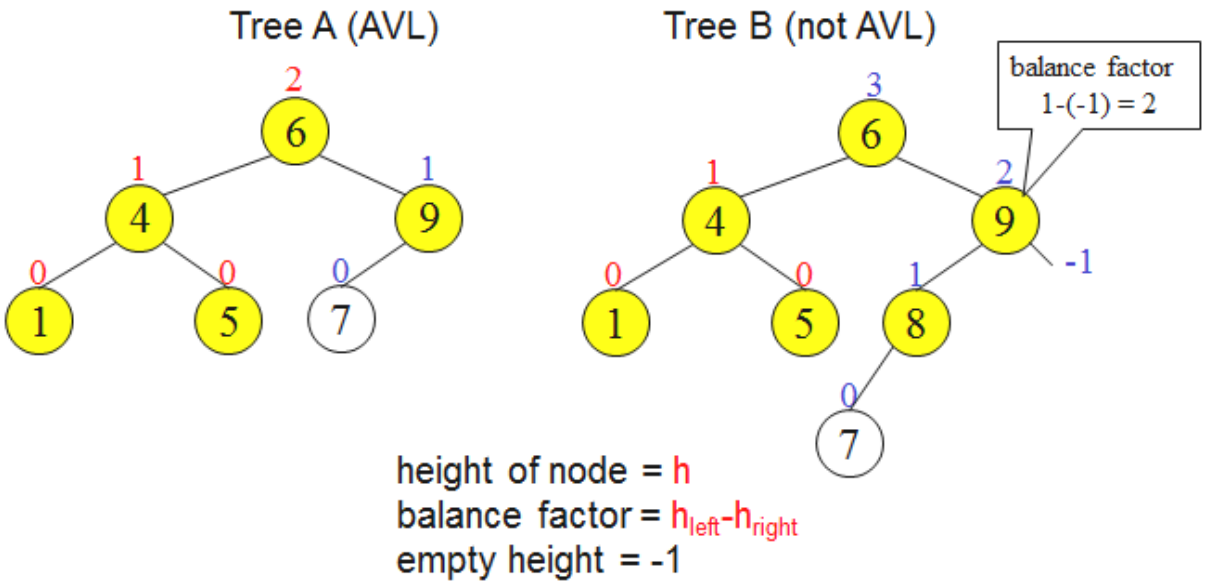
> **SPACE COMPLEXITY:**
>
> - Overall storage is O(n) for n nodes.
> - Recursive algorithms for search or insertion require up to O(h) auxiliary space.

Further Insights on BST Methods

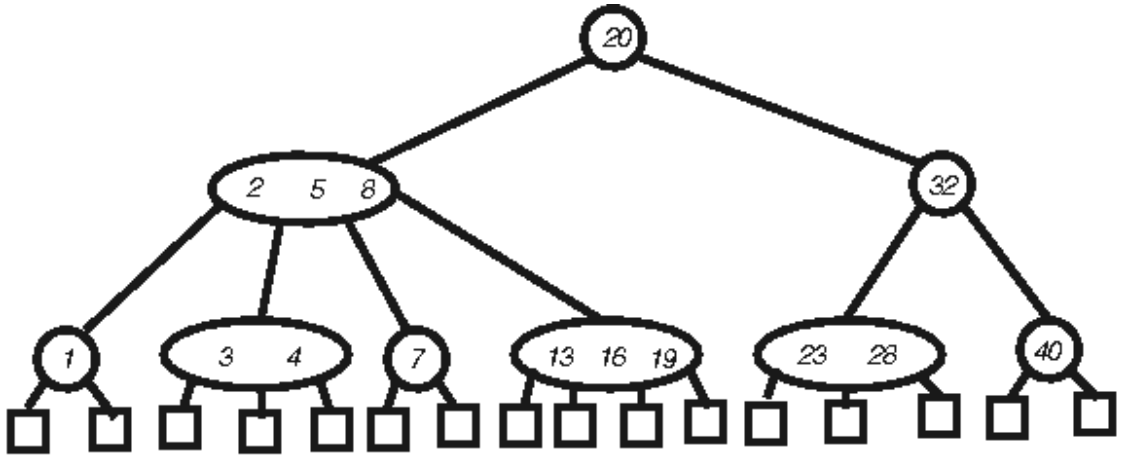
- **Insertion:**
  Follows the search path to a leaf where the key should reside, then replaces that leaf with an internal node containing the new key. Time is O(h).
- **Deletion:**
  If the node has zero or one child, it can be removed directly. If it has two children, typically replace its key with that of its inorder successor and remove the successor. Time is also O(h).
- **Inorder Traversal:**
  Yields sorted keys, confirming the BST property.
- **Balanced vs. Unbalanced Trees:**
  In practice, self-balancing BSTs are preferred to avoid O(n) height in worst-case scenarios.

Other Search Tree Variants

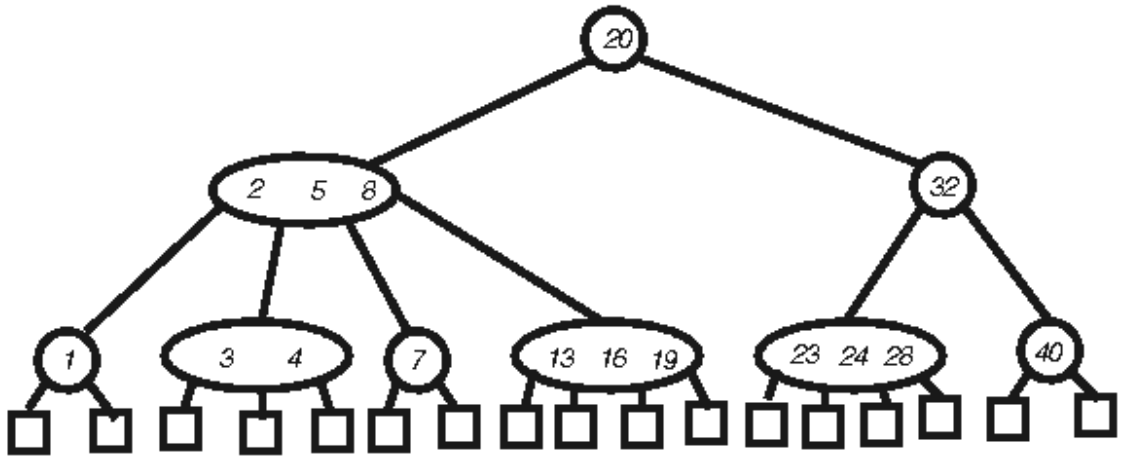
AVL Tree: Maintains strict balance by ensuring that the heights of the left and right subtrees of any node differ by at most 1. This guarantees $O(\log n)$ time for search, insertion, and deletion in the worst case.



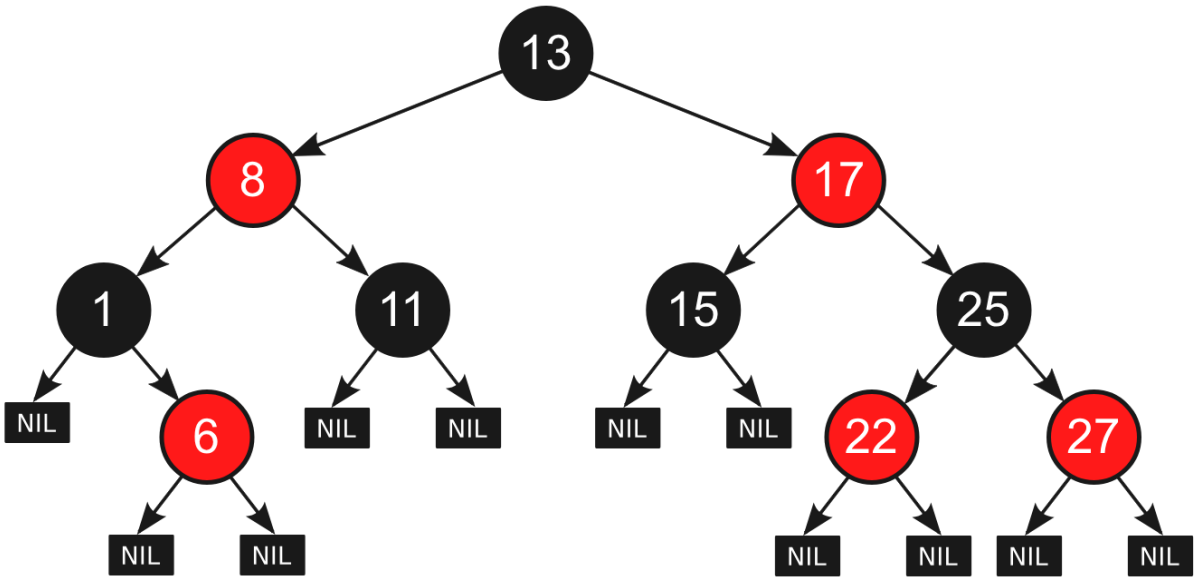
2-4 Tree: A type of B-tree where each internal node can have between 2 and 4 children. It enforces balance by keeping all leaves at the same level. Search, insertion, and deletion remain $O(\log n)$ in the worst case.



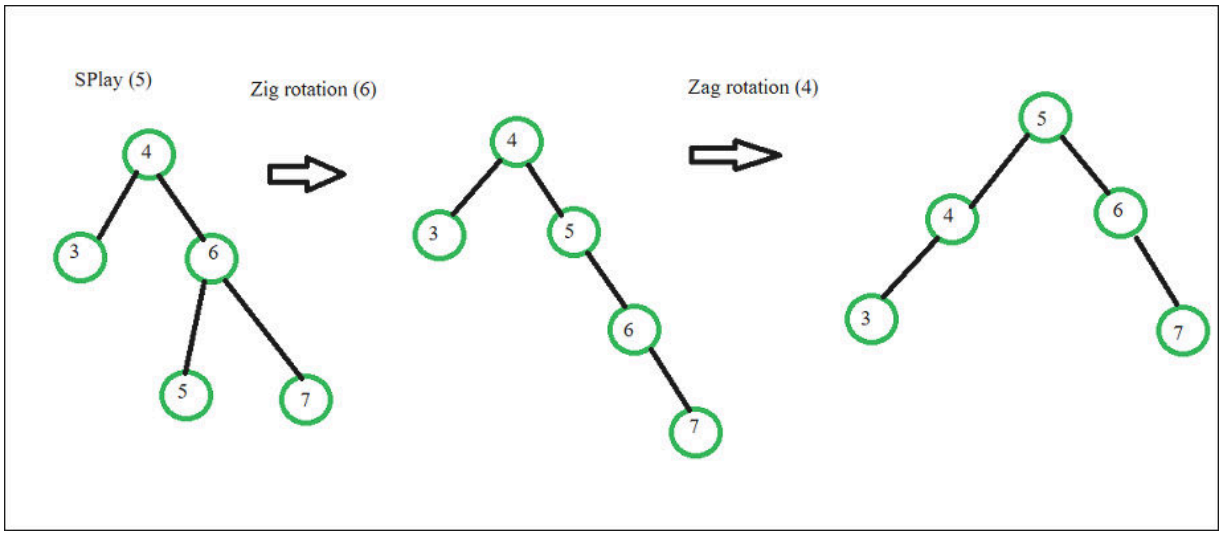
Insert 24



> **Red-Black Tree:** A balanced BST that uses color properties (red or black) on nodes to enforce balance criteria. Ensures worst-case $O(\log(n))$ time for search, insertion, and deletion. Often used in standard libraries (e.g., C++ `std::map`).



> **Splay Tree:** Uses a splaying operation (moving a node to the root via tree rotations) on every access. While any single operation can be O(n) in the worst case, the **amortized** time for search, insertion, and deletion is O($\log n$).



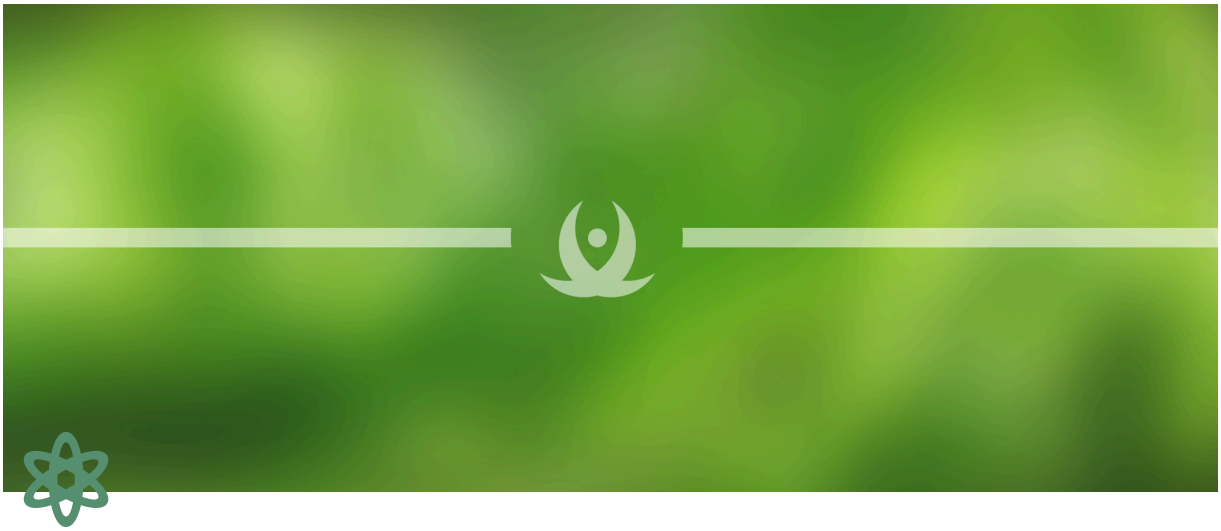
Final Summary & Takeaways

Key Points:

- A BST orders keys so that an inorder traversal lists them in sorted order.
- Search, insertion, and deletion in a BST take O(h) time, where h is the tree's height.

- Without balancing, the height can degrade to O(n). Self-balancing BSTs (AVL, 2-4, Red-Black, and Splay Trees) maintain O($\log n$) complexity for these operations.
- Balanced BSTs are crucial for robust performance in applications requiring frequent insertions and lookups.

Use these principles to implement or analyze BST-based data structures effectively.



10. Tries and Skip Lists

Tries

Definition and Purpose

> **TRIE:** A trie is a tree-based data structure used to store a dynamic set of strings where each node represents a character. The paths from the root to the leaves represent the keys (strings) in the set.
>
> **Purpose:** Tries are particularly useful for efficient pattern matching and retrieval operations, especially when dealing with large texts or dictionaries.

Standard Tries

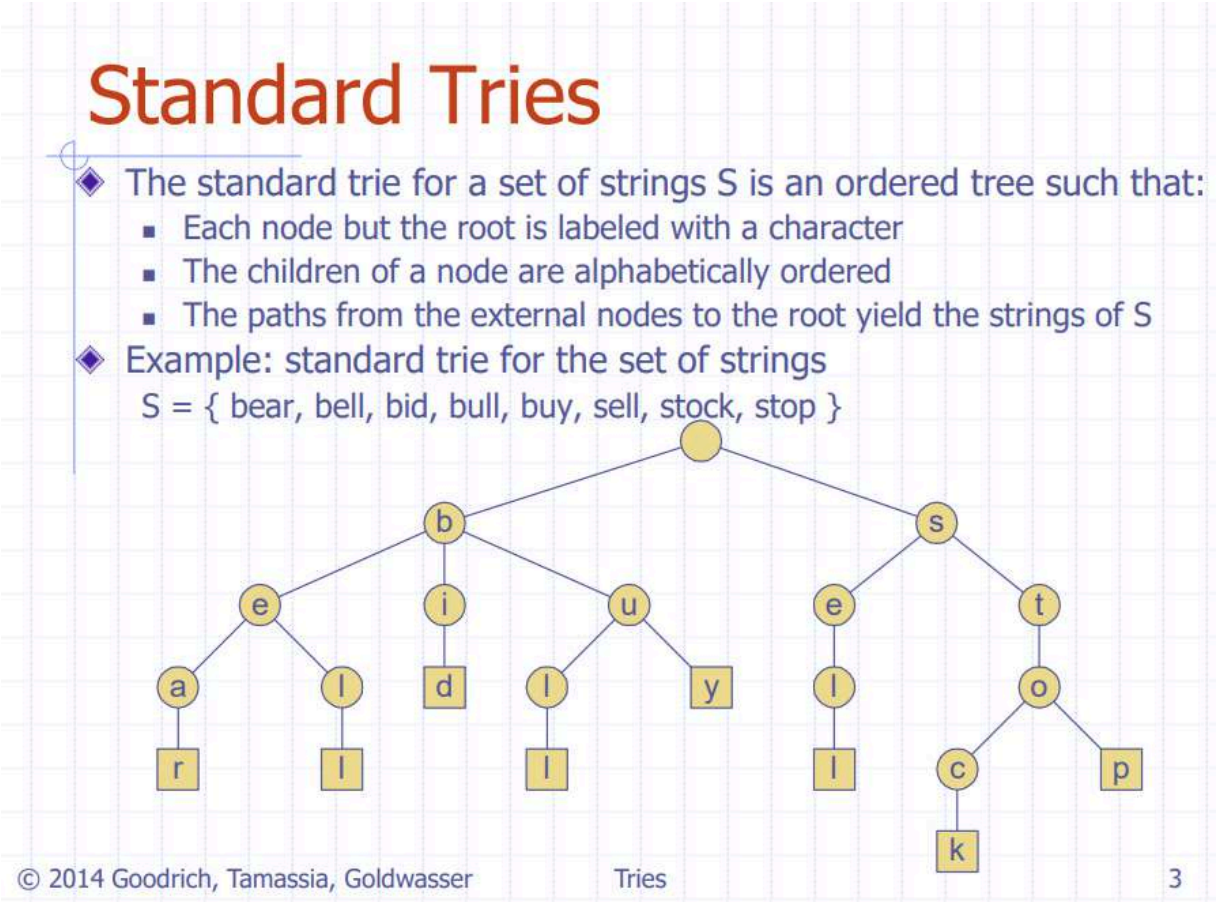
- **Structure:**
  - Each non-root node is labeled with a character.
  - The children of a node are maintained in alphabetical (or defined) order.
  - Each external node (leaf) corresponds to a complete string from the set.
- **Example:**
  For a set $S =$

$\{$"bear", "bell", "bid", "bull", "buy", "sell", "stock", "stop"$\}$, a standard trie is constructed so that each path from a leaf to the root spells out a word.

- **Operations:**
  - **Insertion:** Insert a word character by character, creating new nodes as needed.
  - **Search:** Traverse the trie following the characters of the query string.
- **Complexity:**
  - **Time Complexity:** $O(m)$ for search, insertion, and deletion, where m is the length of the word.
    - **Search, Insertion, and Deletion:**
      - **Worst Case:** $O(dm)$, where:
        - m is the length of the string.
        - d is the size of the alphabet.
      - **Optimized Case:** $O(m)$ when child pointers are accessed in constant time.
    - **Why `O(dm)` in Worst Case:**
      - Traversing the string takes $O(m)$.
      - Checking child pointers may take $O(d)$ at each level.
      - Total time: $O(dm)$.
    - **Optimized Case:**
      - Using an array or hash map for child pointers reduces lookup to $O(1)$.
      - Final time becomes $O(m)$ for most practical implementations.

    Final Complexity Recap:

    - **Worst Case:** $O(dm)$
    - **Optimized Case:** $O(m)$
  - **Space Complexity:** $O(n)$ where n is the total number of characters in all stored strings, though this can be improved with compression.



Compressed Tries

> **COMPRESSED TRIE:** A compressed trie reduces space by merging chains of single-child nodes into one edge labeled with the concatenated characters.

- **Advantage:** Reduces redundant nodes in cases where many words share common prefixes.
- **Example:** In a standard trie, the nodes for the letters "i" and "d" in the word "bid" might be merged into a single edge labeled "id" if they do not branch.
- **Additional Note:** Compressed tries often require more complex algorithms for insertion and deletion due to the variable-length edge labels.

Applications of Tries

- **Pattern Matching:**
  By preprocessing a set of strings into a trie, pattern matching queries (such as

autocomplete or spell checking) can be performed in time proportional to the query string's length.

- **Dictionary Storage:**
  Tries are used to store large dictionaries where quick lookup and prefix searches are essential.

Further Insights and Complexity for Tries

- **Space Usage:**
  A standard trie uses $O(n)$ space in terms of the total number of characters stored. In practice, the space can be high if many nodes are sparse; compressed tries help alleviate this.
- **Operation Complexity:**
  All basic operations (search, insert, delete) work in $O(m)$ time where m is the length of the input string, independent of the number of stored words.
- **Trade-offs:**
  Tries are excellent for fast prefix searches but may consume more memory compared to other data structures (like hash tables) unless compressed.

Skip Lists

Definition and Structure

> **SKIP LIST:** A skip list is a probabilistic data structure that allows fast search, insertion, and deletion operations. It consists of a hierarchy of linked lists, where the bottom level is an ordered list of all elements, and each higher level is a subset of the lower level.

- **Special Keys:**
  Each level includes special keys $+\infty$ and $-\infty$ to denote boundaries.
- **Layered Structure:**
  - Level S_0 contains all the keys.
  - Each higher level S_i is a subsequence of S_{i-1}.
  - The top level contains only the two special keys.

Operations in Skip Lists

Search

> **SEARCH IN A SKIP LIST:** Start at the top-left (smallest key at the highest level) and move forward until the next key exceeds the target. Then, drop down a level and continue.

- **Process:**
  - **Scan-forward:** Move right until the next node's key is greater than or equal to the target.
  - **Drop-down:** When no more nodes can be traversed at the current level, drop to the next lower level.
- **Complexity:**
  - **Expected Time:** $O(\log n)$
    The number of drop-down steps is bounded by the height h (which is $O(\log n)$ with high probability), and each level requires a constant expected number of scan-forward steps (fact: expected coin tosses for tails is 2).

Insertion

> **INSERTION:** To insert a key x:
>
> - Toss a coin repeatedly until a tail is observed. Let i be the number of heads.
> - If i exceeds the current height of the skip list, add new levels.
> - Find the insertion point at each level 0 to i and insert the key accordingly.

- **Randomized Nature:**
  The level at which an element is inserted is determined randomly, ensuring that the height of the skip list remains $O(\log n)$ with high probability.

Deletion

> **DELETION:** To remove a key x:
>
> - Search for x and find its occurrence in every level.

- Remove x from each level.
- If a level becomes empty (except for the special keys), it is removed.

- **Complexity:**
  Deletion operations also take $O(\log n)$ expected time.

Skip List Performance and Complexity

- **Space Complexity:**

  The expected space usage is $O(n)$ because each element appears in each level with probability $1/2^i$. Thus, the total expected number of nodes is proportional to n.

- **Height:**

  With high probability, the height h of a skip list is $O(\log n)$.

  - **Probabilistic Analysis:**
    - Fact 1: The probability of obtaining i consecutive heads is $1/2^i$.
    - Fact 2: The expected number of nodes at level i is $n/2^i$.
    - By choosing $i \approx 3 \log n$, the probability of having a node at that level is at most $1/n^2$.

- **Search, Insertion, Deletion:**

  All these operations have an expected time complexity of $O(\log n)$ due to the logarithmic height and constant expected scan-forward steps per level.

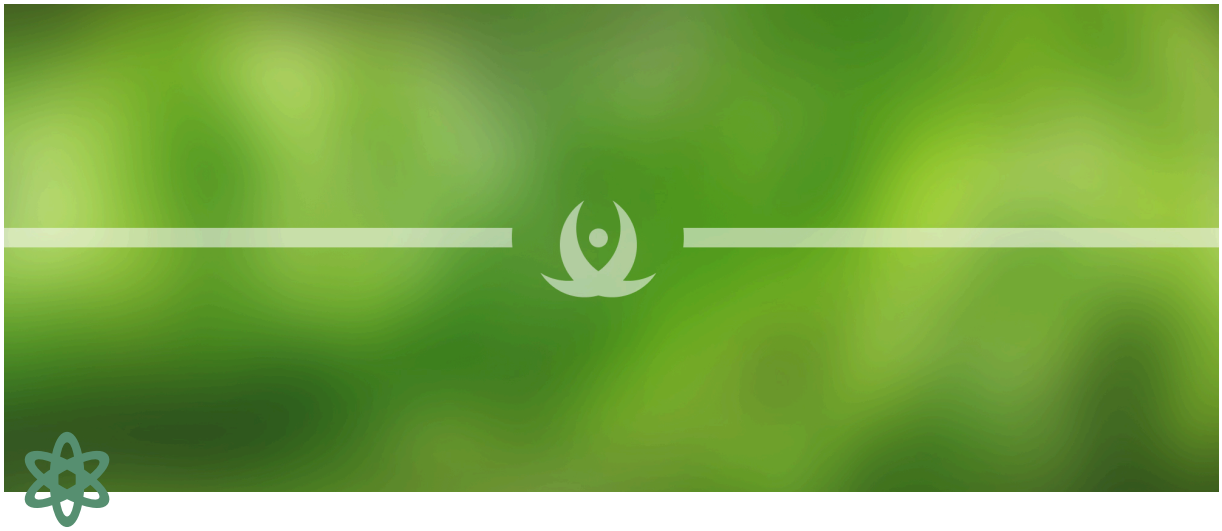
Additional Insights on Skip Lists

- **Randomized Algorithms:**
  Skip lists rely on coin tosses to determine the level of each inserted element. Although the worst-case time can be high, the probability of such cases is very low.
- **Comparison to BSTs:**
  Skip lists offer comparable expected performance to balanced BSTs but are often simpler to implement.
- **Use Cases:**
  Skip lists are used for implementing ordered maps and sets, and they are a practical

alternative to balanced trees in many applications.

Final Summary & Takeaways

> **Key Takeaways:**
>
> - **Tries** are specialized trees for storing and searching strings efficiently with operations proportional to the length of the string ($O(m)$). Compressed tries optimize space by collapsing chains of single-child nodes.
> - **Skip Lists** are randomized data structures that maintain multiple levels of linked lists. They support fast search, insertion, and deletion with expected time complexity $O(\log n)$ and use linear space.
> - **Complexity Analysis:**
>   - Tries: $O(m)$ per operation, with space usage $O(n)$ where n is the total number of characters.
>   - Skip Lists: Expected search, insertion, and deletion in $O(\log n)$; height is $O(\log n)$ with high probability.
> - Both data structures are powerful for applications in string matching, dictionary storage, and ordered maps, each with unique trade-offs in terms of implementation complexity and performance guarantees.



11. Review: Tries and Skip Lists

Overview

This review summarizes key concepts from the extensive note on two advanced data structures—tries and skip lists. Both structures are used to support efficient searching and dynamic data storage, with tries focusing on strings and skip lists offering a probabilistic alternative to balanced trees.

Tries

> **TRIE:** A tree-like data structure where each node represents a character. Paths from the root to leaves represent stored strings.

- **Standard Trie:**
  - Each non-root node holds a character.
  - Children are ordered (e.g., alphabetically).
  - Inorder traversal yields strings in sorted order.
- **Compressed Trie:**
  - Collapses chains of single-child nodes.
  - Reduces space by storing concatenated labels on edges.

- **Operations Complexity:**
  - **Search, Insert, Delete:** O(m), where m is the length of the string.
  - **Space Usage:** O(n) where n is the total number of characters stored.

Skip Lists

> **SKIP LIST:** A randomized layered linked-list structure that supports fast search, insertion, and deletion.

- **Structure:**
  - Multiple levels: the bottom level contains all keys; higher levels are increasingly sparse.
  - Special boundary keys ($-\infty$ and $+\infty$) are present in each level.
- **Search Process:**
  - Start at the top level and scan right.
  - Drop down a level when the next key exceeds the target.
  - Expected time complexity is O(log n).
- **Insertion and Deletion:**
  - Use coin tosses to decide the level for each new element.
  - Expected operations take O(log n) time.
- **Space Usage:**
  - Expected space is O(n) since each element appears with a decreasing probability at higher levels.

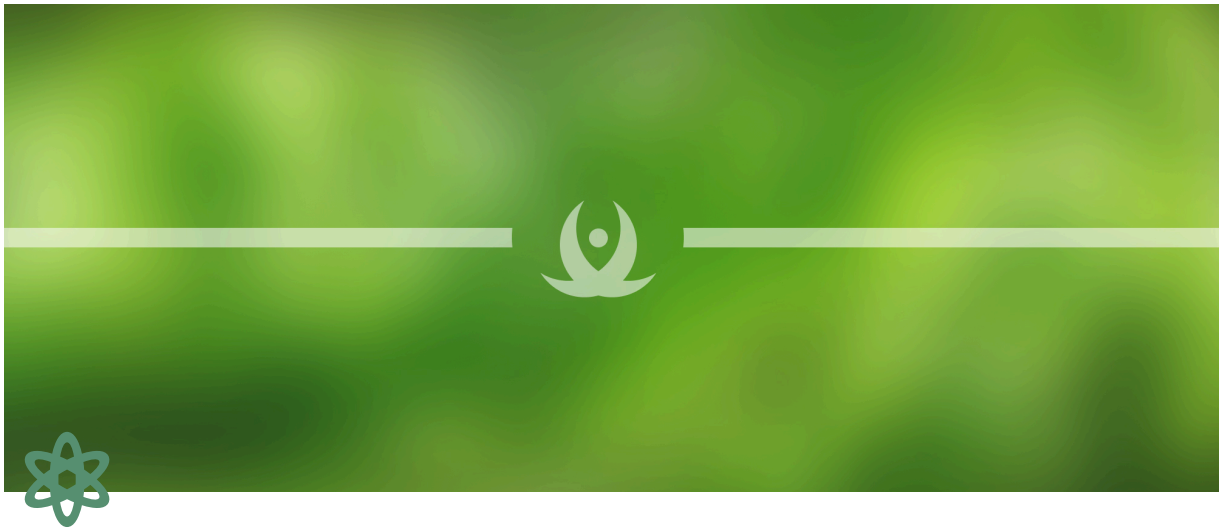
Complexity Analysis

- **Tries:**
  - Operations run in O(m) time (m = length of the key).
- **Skip Lists:**
  - Expected operation time is O(log n) for search, insertion, and deletion.

- Height is O(log n) with high probability due to the randomized level assignment.

Final Takeaways

- **Tries** are ideal for string storage and prefix matching, offering linear-time operations relative to the key length.
- **Skip Lists** provide a simple, randomized alternative to balanced trees with comparable expected performance.
- Both data structures are efficient for dynamic data operations and are chosen based on the specific application requirements.



12. Priority Queues and Heaps

Priority Queues

Overview of Priority Queue ADT

> **PRIORITY QUEUE ADT:** A priority queue stores a collection of entries, where each entry is a key–value pair. The key is used to determine the order among entries.
>
> **Main Methods:**
>
> - **insert(k, v):** Inserts an entry with key k and value v.
>
> *Worst-case complexity:* O(1) if using a sequence-based unsorted list; O(n) if using a sorted list (for locating insertion point).
>
> - **removeMin():** Removes and returns the entry with the smallest key, or returns null if empty.
>
> *Worst-case complexity:* O(n) for an unsorted list; O(1) for a sorted list.
>
> **Additional Methods:**
>
> - **min():** Returns (but does not remove) the entry with the smallest key, or returns null if empty.
>
> *Worst-case complexity:* O(n) for an unsorted list; O(1) for a sorted list.

- **size(), isEmpty():** Return the number of entries and whether the queue is empty, respectively (typically O(1)).

Total Order Relations

TOTAL ORDER RELATION: For keys in a priority queue, a total order relation ($\leq$) is defined such that for any two keys x and y:

- **Comparability:** Either $x \leq y$ or $y \leq x$.
- **Antisymmetry:** If $x \leq y$ and $y \leq x$, then $x = y$.
- **Transitivity:** If $x \leq y$ and $y \leq z$, then $x \leq z$.

Entry ADT

ENTRY ADT: An entry encapsulates a key–value pair and provides methods:

- `getKey()` – returns the key.
- `getValue()` – returns the value.

In Java, it is often specified as an interface.

```
public interface Entry<K,V>
{
        K getKey();
        V getValue();
}
```

Comparator ADT

COMPARATOR ADT: A comparator defines an external mechanism for comparing two objects according to a total order.

- **Primary Method:** `compare(x, y)` returns:
  - A negative number if $x < y$,
  - Zero if $x = y$,
  - A positive number if $x > y$.

Example: A lexicographic comparator for 2-D points compares x-coordinates first and then y-coordinates.

```
import java.util.Comparator;

/**
 * Represents a point in the 2D plane with integer coordinates.
 */
public class Point2D {
    private int x;
    private int y;

    /**
     * Constructs a Point2D with the specified (x, y) coordinate
s.
     */
    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Returns the x-coordinate of this point.
     */
    public int getX() {
        return x;
    }

    /**
     * Returns the y-coordinate of this point.
     */
    public int getY() {
        return y;
    }

    @Override
```

```
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}

/**
 * A comparator for Point2D objects that performs a lexicographic
ordering:
 *   1) Compare the x-coordinates.
 *   2) If x-coordinates are equal, compare the y-coordinates.
 */
class LexicographicComparator implements Comparator<Point2D> {
    @Override
    public int compare(Point2D a, Point2D b) {
        if (a.getX() < b.getX()) {
            return -1;
        } else if (a.getX() > b.getX()) {
            return 1;
        } else {
            // x-coordinates are equal, compare y
            return Integer.compare(a.getY(), b.getY());
        }
    }
}
```

Sequence-based Priority Queue Implementations

Unsorted List Implementation

> **UNSORTED LIST:** The data is stored in an unsorted sequence (e.g., a linked list).
>
> - **insert:** Simply add the new entry at the beginning or end of the list.
>
>   *Worst-case complexity:* O(1).
> - **removeMin / min:** Must traverse the entire list to locate the smallest key.
>
>   *Worst-case complexity:* O(n).

Sorted List Implementation

> **SORTED LIST:** The entries are kept in order based on their keys.
>
> - **insert:** Requires locating the correct position in the list, which may take traversing the list. *Worst-case complexity:* O(n).
> - **removeMin / min:** The smallest key is located at the beginning of the list. *Worst-case complexity:* O(1).

Heaps

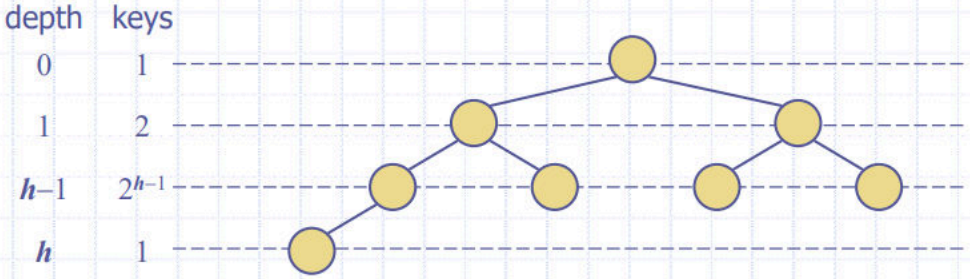
Heap Definition and Properties

> **HEAP:** A heap is a binary tree data structure that stores keys and satisfies two main properties:
>
> - **Heap-Order Property:** For every internal node (other than the root), the key at the node is greater than or equal to the key at its parent.
> - **Complete Binary Tree:** All levels are completely filled, except possibly the last one which is filled from left to right.
>
> The worst-case height of a heap storing n keys is O(log n).

Height of a Heap

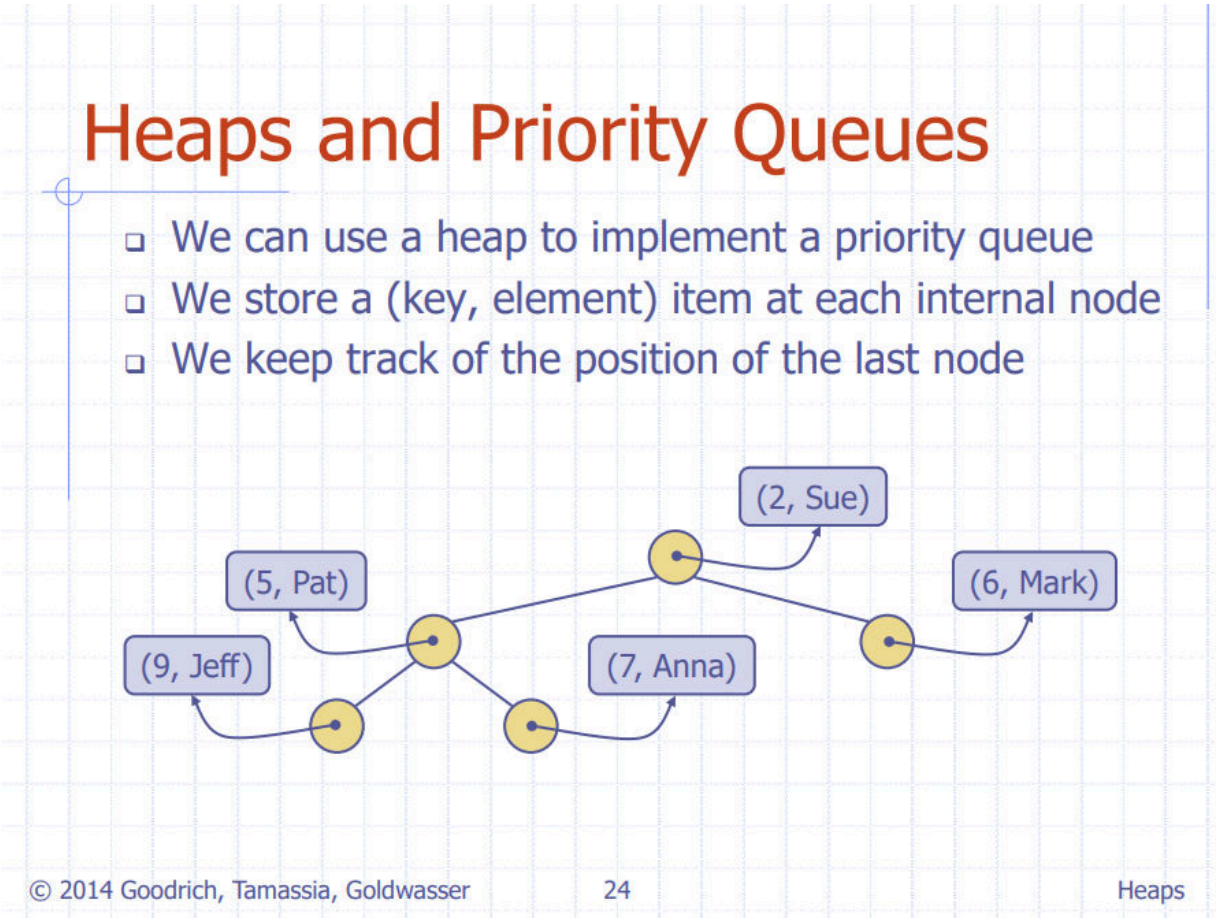
- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



Heap as a Priority Queue

> A heap can be used to implement a priority queue where each internal node holds an entry (key, element). The position of the last node is tracked to facilitate insertions and removals.



Insertion into a Heap

> **Insertion Algorithm:**
>
> - **Step 1:** Find the insertion node (new last node).
> - **Step 2:** Store the new key at this node.
> - **Step 3:** Restore the heap-order property via upheap.
>
> **Upheap Operation:**
>
> The new key is compared with its parent, and if it violates the heap-order property, it is swapped upward until the proper order is restored.
>
> *Worst-case complexity:* O(log n).

Removal from a Heap

> Removal (removeMin):

- **Step 1:** Remove the root key, which holds the minimum.
- **Step 2:** Replace the root with the key from the last node.
- **Step 3:** Restore the heap-order property via downheap.

Downheap Operation:

The key is swapped downward along a path until it is placed at a node with children that are both greater than or equal to it.

Worst-case complexity: O(log n).

Updating the Last Node

Updating the Last Node: To find the new last node (after an insertion or removal), the algorithm traverses a path of O(log n) nodes.

Updating the last node means that after you add or remove an element in a heap, you may need to reposition your pointer (or index) that identifies the last node in the complete binary tree. Since the tree's height is O(log n), finding the new last node involves walking down (or up) a path that is at most $\log_2(n)$ nodes long. In other words, even in the worst case, you'll only examine O(log n) nodes to update this pointer.

Worst-case complexity: O(log n).

Array-based Heap Implementation

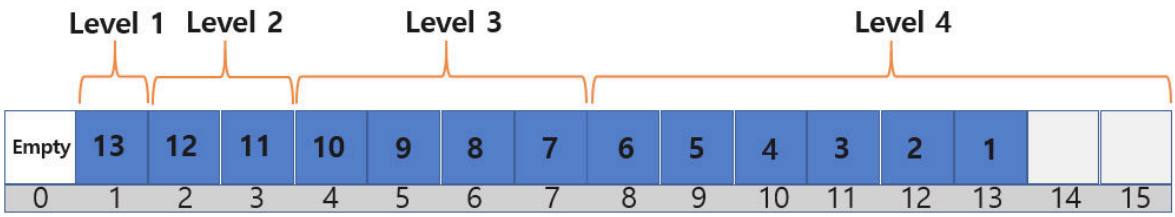
ARRAY-BASED REPRESENTATION: A heap can be efficiently represented as an array without explicit links:

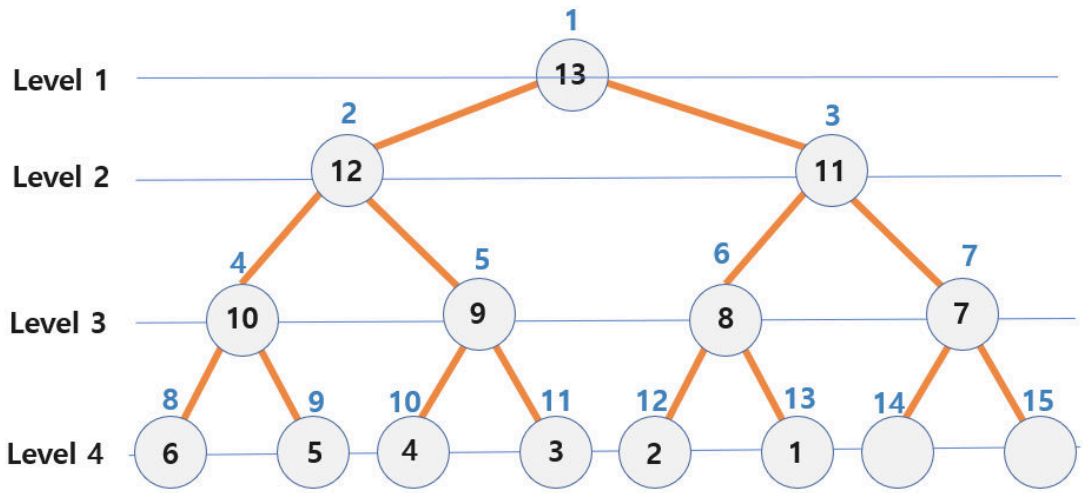
- For a node at index i:
  - Left child is at index 2i + 1.
  - Right child is at index 2i + 2.
- Insertion corresponds to adding at the next available position (at the end of the array), followed by **upheap**.
- Removal (removeMin) corresponds to replacing the root with the last element and then performing **downheap**.

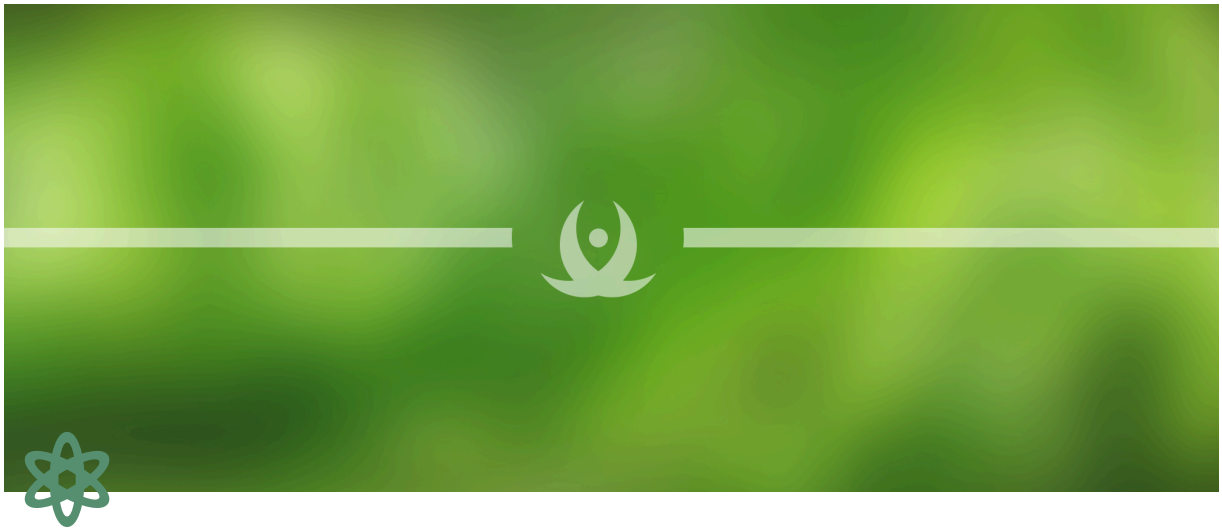
This representation leads directly to the in-place heapsort algorithm.

Worst-case complexity:

- **Insertion:** O(log n)
- **Removal:** O(log n)







13. Map ADT and Implementations (Hash Tables)

Objective & Scope

This note reviews the **Map** (associative array) Abstract Data Type, explores several elementary implementations (linked lists and arrays, both sorted and unsorted), and then introduces **hash tables**. We'll cover collision-handling strategies—linear and quadratic probing, open vs. closed hashing, and separate chaining—as well as insertion behavior and time complexities.

Map ADT

> **MAP (Associative Array):** A collection of key–value pairs supporting operations to insert a pair, find the value for a given key, and remove a key–value pair.

Primary operations:

- `put(key, value)` — insert or update
- `get(key)` — retrieve
- `remove(key)` — delete

- `size()`, `isEmpty()`

Elementary Implementations

We compare four simple implementations:

Implementation	`put`	`get`	`remove`	Notes
Unsorted Linked List	O(1) (at head)	O(n)	O(n)	Easy insert, slow searches
Sorted Linked List	O(n)	O(n)	O(n)	Can stop early in search, still linear time
Unsorted Array	O(1) (append)	O(n)	O(n)	Simple, but removal requires shift
Sorted Array	O(n) (shift)	O(log n) via binary search	O(n)	Fast search, expensive insert/remove

> **Sorted vs. Unsorted:** Sorting speeds up `get` (binary search) but makes `put` / `remove` pay the cost of shifting elements or relinking nodes.

Hash Tables

> **HASH TABLE:** A data structure that uses a hash function to map keys to bucket indices, aiming for **average-case** O(1) time for `put`, `get`, and `remove`.

Hash Function & Load Factor

- **Hash function** `h(key)` → integer in `[0, m-1]`, where `m` is table size.
- **Load factor** α = (number of entries) / (number of buckets).
  - Impacts performance: keep $\alpha < 1$ (usually $\alpha \leq 0.7$).

Collision Handling

When two keys map to the same bucket, we must resolve collisions.

1. Open Addressing (Closed Hashing)

All entries reside **in the table** itself; collisions trigger a probe sequence to find another slot.

Linear Probing

- Probe sequence:

$$h_i = (h_0 + i) \bmod m, \quad i = 0, 1, 2, \ldots$$

- **Primary clustering:** long runs of occupied slots slow performance when α grows.

Quadratic Probing

- Probe sequence:

$$h_i = \left(h_0 + c_1\, i + c_2\, i^2\right) \bmod m.$$

- Reduces primary clustering but can suffer **secondary clustering** (same initial hash).

Insertion (Open Addressing)

1. Compute `h = h(key)`.
2. For `i = 0,1,2,…`:
    - Compute probe index `h_i`.
    - If slot `h_i` is empty or marked deleted, insert there and stop.
3. If table is full or too many probes, resize (rehash) to a larger table.

2. Separate Chaining (Open Hashing)

Each bucket holds a pointer to a secondary data structure (e.g., a linked list) of all entries hashing to that bucket.

- `put` / `get` / `remove`:
    1. Compute `h = h(key)`.
    2. In the list at bucket `h`, search/update/delete the key.
- **Performance:**

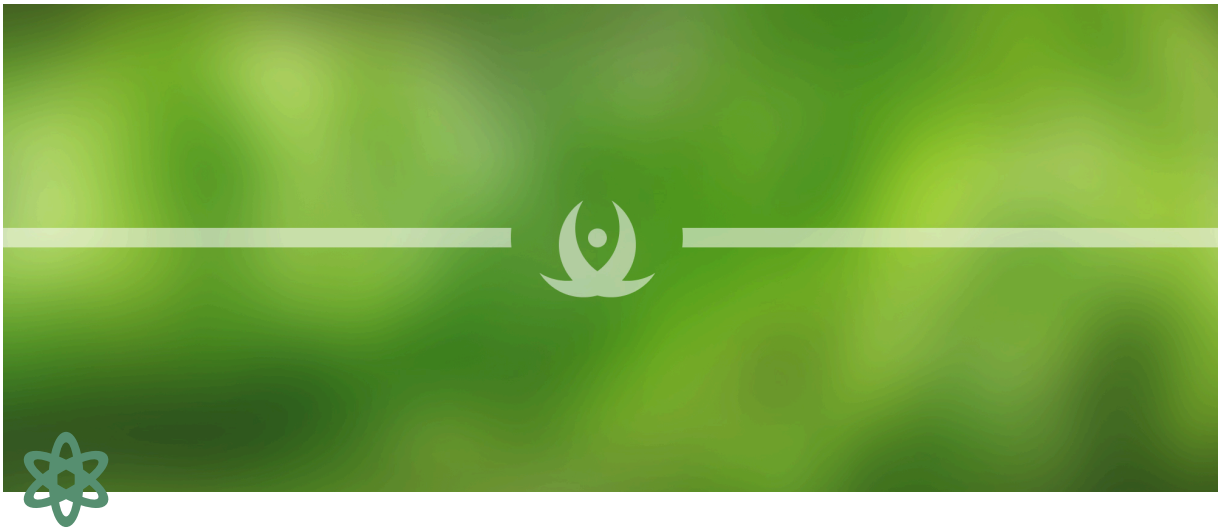
- Average cost O(1 + α), where α = load factor (entries per bucket).
- No clustering in table slots; chains grow instead.

Open vs. Closed Hashing

Feature	Open Addressing	Separate Chaining
Storage	Table only	Table + external chains
Load factor α limit	α < 1	α can exceed 1
Memory overhead	Low (table array)	Higher (extra pointers/nodes)
Clustering issues	Primary/secondary clustering	No clustering in table slots
Deletion	Requires tombstones or rehash	Simple list removal
Cache performance	Excellent (contiguous memory)	Lower (pointer chasing in chains)

Summary of Complexities

- **Unsorted linked list**: `put` O(1), `get` / `remove` O(n)
- **Sorted linked list**: `put` / `remove` O(n), `get` O(n)
- **Unsorted array**: `put` O(1), `get` / `remove` O(n)
- **Sorted array**: `put` / `remove` O(n), `get` O(log n)
- **Hash table (open addressing)**: `put` / `get` / `remove` O(1) expected, O(n) worst
- **Hash table (chaining)**: `put` / `get` / `remove` O(1 + α) expected



14. Map & HashMap Pseudocode and Rehashing

1. Map ADT Pseudocode (Unsorted List Implementation)

For a simple **Map** (aka **Dictionary**) implemented with an unsorted list of entries $(key, value)$:

```
// Assume map.entries is a list of (key, value) pairs

function Map_Put(map, key, value):
    // If key already exists, update its value
    for each entry in map.entries:
        if entry.key == key:
            entry.value = value
            return
    // Otherwise, append a new entry
    append map.entries, (key, value)

function Map_Get(map, key):
    for each entry in map.entries:
        if entry.key == key:
```

```
            return entry.value
    return null   // or raise KeyError

function Map_Remove(map, key):
    for i from 0 to length(map.entries) - 1:
        if map.entries[i].key == key:
            remove map.entries[i]
            return true
    return false  // key not found
```

- **Complexities:**
  - `put` – O(n) worst (search for existing key)
  - `get` – O(n)
  - `remove` – O(n)

2. HashMap Pseudocode (Separate Chaining)

A **HashMap** using **separate chaining** stores an array of buckets, each bucket is a list of entries.

```
// map.buckets is an array of lists; map.capacity = number of bucke
ts
// map.size = number of stored entries
// map.loadFactorThreshold e.g. 0.75

function HashMap_Put(map, key, value):
    if (map.size + 1) / map.capacity > map.loadFactorThreshold:
        HashMap_Rehash(map)

    index = Hash(key) mod map.capacity
    bucket = map.buckets[index]

    // Update existing key?
    for each entry in bucket:
        if entry.key == key:
```

```
            entry.value = value
            return

    // Otherwise insert new entry
    append bucket, (key, value)
    map.size += 1

function HashMap_Get(map, key):
    index = Hash(key) mod map.capacity
    for each entry in map.buckets[index]:
        if entry.key == key:
            return entry.value
    return null   // or raise KeyError

function HashMap_Remove(map, key):
    index = Hash(key) mod map.capacity
    bucket = map.buckets[index]
    for i from 0 to length(bucket)-1:
        if bucket[i].key == key:
            remove bucket[i]
            map.size -= 1
            return true
    return false  // key not found
```

Rehashing

```
function HashMap_Rehash(map):
    oldBuckets = map.buckets
    oldCapacity = map.capacity

    // Typically double capacity and choose next prime
    map.capacity = NextPrime(2 * oldCapacity)
    map.buckets = new array of lists of size map.capacity
    map.size = 0
```

```
    // Reinsert all entries
    for each bucket in oldBuckets:
        for each entry in bucket:
            HashMap_Put(map, entry.key, entry.value)
```

- **When to rehash?**

  Whenever the load factor $\alpha = \frac{\text{size}}{\text{capacity}}$ exceeds the chosen threshold (e.g. 0.75). Rehashing keeps average-case operations O(1).

3. HashMap Pseudocode (Open Addressing)

An alternative **HashMap** uses **open addressing** (e.g. linear or quadratic probing) with a single array of slots.

```
// map.table is an array of slots; each slot holds either: EMPTY, T
OMBSTONE, or (key, value)
// map.capacity, map.size, map.loadFactorThreshold

function OpenAddress_Put(map, key, value):
    if (map.size + 1) / map.capacity > map.loadFactorThreshold:
        OpenAddress_Rehash(map)

    base = Hash(key) mod map.capacity
    for i from 0 to map.capacity - 1:
        index = (base + i) mod map.capacity         // linear probi
ng
        // index = (base + c1*i + c2*i^2) mod capacity  // quadrati
c probing variant

        slot = map.table[index]
        if slot is EMPTY or slot is TOMBSTONE:
            map.table[index] = (key, value)
            map.size += 1
            return
        else if slot.key == key:
```

```
            slot.value = value
            return
    // If we exit loop, the table is full (shouldn't happen if reha
shing correctly)

function OpenAddress_Get(map, key):
    base = Hash(key) mod map.capacity
    for i from 0 to map.capacity - 1:
        index = (base + i) mod map.capacity
        slot = map.table[index]
        if slot is EMPTY:
            return null   // key not in table
        else if slot is TOMBSTONE:
            continue     // skip removed slot
        else if slot.key == key:
            return slot.value
    return null

function OpenAddress_Remove(map, key):
    base = Hash(key) mod map.capacity
    for i from 0 to map.capacity - 1:
        index = (base + i) mod map.capacity
        slot = map.table[index]
        if slot is EMPTY:
            return false // key not found
        else if slot is TOMBSTONE:
            continue
        else if slot.key == key:
            map.table[index] = TOMBSTONE
            map.size -= 1
            return true
    return false
```

Rehashing (Open Addressing)

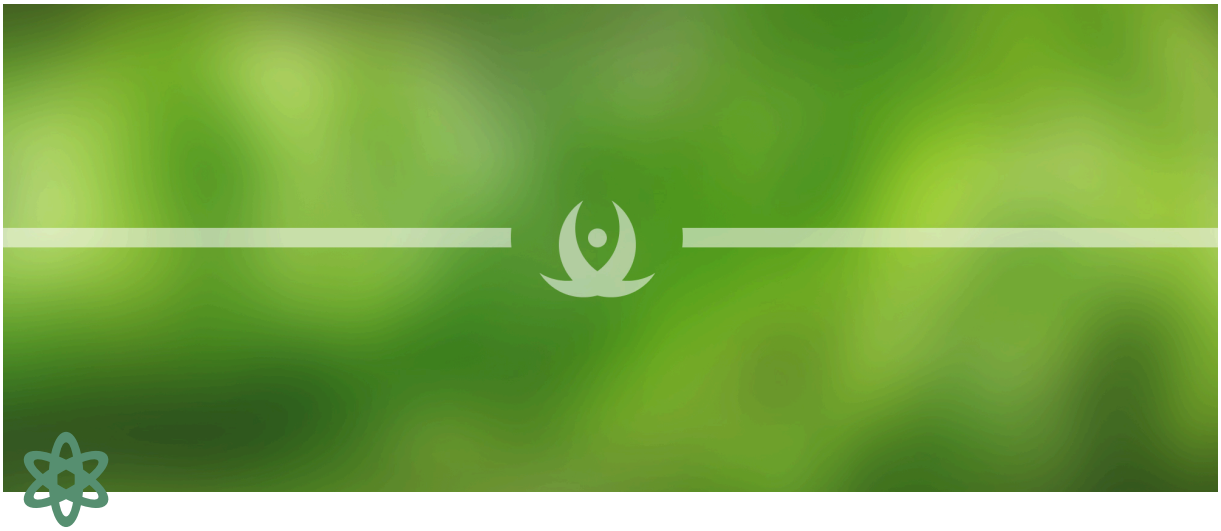
```
function OpenAddress_Rehash(map):
    oldTable = map.table
    oldCapacity = map.capacity

    map.capacity = NextPrime(2 * oldCapacity)
    map.table = new array of slots of size map.capacity
    map.size = 0

    for each slot in oldTable:
        if slot is (key, value):
            OpenAddress_Put(map, slot.key, slot.value)
```

- **Tombstones** allow iteration to continue past removed slots but are cleared during rehash.

4. Summary

- **Map ADT** can be implemented simply with lists or arrays (sorted/unsorted) but these cost O(n) in at least one operation.
- **HashMaps** achieve average-case O(1) by using a hash function and handling collisions via:
  - **Separate chaining** (external lists per bucket).
  - **Open addressing** (probe sequences within the array).
- **Rehashing** is triggered when load factor exceeds a threshold to maintain performance, by allocating a larger table and reinserting all entries.

This pseudocode provides a blueprint for implementing Map and HashMap data structures along with their key behaviors and rehashing strategies.



15. Midterm Preparation

1. Arrays

- **Access**:
  - **Time Complexity**: $O(1)$ — Direct access by index.
  - **Limitation**: Fixed size; resizing is costly.
- **Space Complexity**: $O(n)$, where n is the number of elements.
- **Use Cases**:
  - Fast lookups with known indices.
  - Storing data that does not change in size (static datasets).
- **Drawbacks**:
  - **Insertion/Deletion**: Costly in the middle; requires shifting elements.
  - Fixed size (unless dynamic resizing is implemented).
- **Insert at index i**:

```
function insert(arr, index, value):
    if index >= length(arr):
```

```
        print("Index out of bounds")
        return
    for j = length(arr) - 1 down to index:
        arr[j + 1] = arr[j]
    arr[index] = value
    return arr
```

- **Get at index i**:

```
function get(arr, index):
    if index < 0 or index >= length(arr):
        print("Index out of bounds")
        return null
    return arr[index]
```

- **Remove at index i**:

```
function remove(arr, index):
    if index < 0 or index >= length(arr):
        print("Index out of bounds")
        return null
    for j = index to length(arr) - 2:
        arr[j] = arr[j + 1]
    arr[length(arr) - 1] = null
    return arr
```

2. Linked Lists

- **Singly Linked List**:
  - **Insertions/Deletions**: $O(1)$ at the head or tail (if tail reference is maintained).
  - **Traversal**: $O(n)$ because of the need to visit each node.
  - **Space Complexity**: $O(n)$ for storing references in addition to data.
- **Doubly Linked List**:

    - **Insertions/Deletions**: $O(1)$ for any node (if node is known).
    - **Traversal**: Can go both forward and backward with a tail reference.
    - **Space Complexity**: $O(n)$ (more overhead due to extra pointers).
- **Use Cases**: Dynamic data structures, when frequent insertions/deletions are required.
- **Drawbacks**: Extra memory usage per node, slower access compared to arrays.
- **Insert at head**:

```
function insertHead(list, value):
    newNode = new Node(value)
    newNode.next = list.head
    list.head = newNode
```

- **Get at index i**:

```
function get(list, index):
    current = list.head
    count = 0
    while current is not null:
        if count == index:
            return current.value
        count += 1
        current = current.next
    return null  // If index is out of bounds
```

- **Remove at index i**:

```
function remove(list, index):
    if index == 0:
        list.head = list.head.next
        return
    current = list.head
    count = 0
```

```
    while current is not null:
        if count == index - 1:
            current.next = current.next.next
            return
        count += 1
        current = current.next
```

3. Stacks

- **LIFO (Last In, First Out)**: Elements are processed in reverse order of their insertion.
- **Operations**:
  - **Push/Pop**: Both take $O(1)$ time.
  - **Auxiliary Operations**: **Peek** is $O(1)$, checking if empty is $O(1)$.
- **Applications**:
  - Undo mechanisms, recursion, expression evaluation (postfix notation).
  - Evaluating expressions (using operator precedence).
- **Drawbacks**:
  - Stack overflow in array-based implementation if the stack size is not managed dynamically.
- **Push**:

```
function push(stack, element):
    stack.top = new Node(element, stack.top)
```

- **Pop**:

```
function pop(stack):
    if stack.isEmpty():
        return null
    value = stack.top.value
```

```
    stack.top = stack.top.next
    return value
```

- **Peek**:

```
function peek(stack):
    if stack.isEmpty():
        return null
    return stack.top.value
```

4. Queues

- **FIFO (First In, First Out)**: The first inserted element is the first to be removed.
- **Operations**:
  - **Enqueue**: $O(1)$ — Insertion at the rear.
  - **Dequeue**: $O(1)$ — Removal from the front.
  - **Peek**: $O(1)$ — View front element.
- **Applications**:
  - Task scheduling (e.g., print jobs, CPU scheduling).
  - Breadth-First Search (BFS) in graph traversal.
- **Drawbacks**:
  - Fixed size in array implementation (may cause overflow).
  - **Circular Queue**: Can solve space inefficiency in a static array-based queue.
- **Enqueue**:

```
function enqueue(queue, element):
    newNode = new Node(element)
    if queue.isEmpty():
        queue.front = queue.rear = newNode
        return
```

```
    queue.rear.next = newNode
    queue.rear = newNode
```

- **Dequeue**:

```
function dequeue(queue):
    if queue.isEmpty():
        return null
    value = queue.front.value
    queue.front = queue.front.next
    if queue.front is null:
        queue.rear = null  // If the queue is now empty
    return value
```

- **Peek**:

```
function peek(queue):
    if queue.isEmpty():
        return null
    return queue.front.value
```

5. Trees

- **Binary Tree**:
  - **Height**: The height is the longest path from the root to a leaf.
  - **Balanced vs. Unbalanced**:
    - **Balanced**: $O(\log n)$ operations.
    - **Unbalanced**: $O(n)$ operations (degenerates to a linked list).
- **Binary Search Tree (BST)**:
  - **Operations**: Insertion, deletion, and search all take $O(\log n)$ in a balanced BST.
  - **Balancing**: Use AVL or Red-Black trees to ensure balanced structure.

- **AVL Tree**: Self-balancing binary search tree, ensures $O(\log n)$ time for all operations.
- **Red-Black Tree**: A type of self-balancing BST with slightly relaxed balance rules for efficiency in insertion and deletion.
- **Applications**: Sorting, searching, storing hierarchical data.
- **Drawbacks**: Complex implementation, need to maintain balancing properties.
- **Insert in BST**:

```
function insertBST(root, value):
    if root is null:
        return new Node(value)
    if value < root.value:
        root.left = insertBST(root.left, value)
    else:
        root.right = insertBST(root.right, value)
    return root
```

- **Get in BST**:

```
function getBST(root, value):
    if root is null or root.value == value:
        return root
    if value < root.value:
        return getBST(root.left, value)
    return getBST(root.right, value)
```

- **Remove in BST**:

```
function removeBST(root, value):
    if root is null:
        return root
    if value < root.value:
        root.left = removeBST(root.left, value)
    else if value > root.value:
```

```
        root.right = removeBST(root.right, value)
    else:
        if root.left is null:
            return root.right
        if root.right is null:
            return root.left
        minNode = findMin(root.right)
        root.value = minNode.value
        root.right = removeBST(root.right, minNode.value)
    return root
```

- **Find Minimum**:

```
function findMin(root):
    while root.left is not null:
        root = root.left
    return root
```

6. Heaps

- **Max Heap**: Parent node is always greater than or equal to its children.
- **Min Heap**: Parent node is always smaller than or equal to its children.
- **Operations**:
  - **Insert**: $O(\log n)$ — Bubble up the new element.
  - **Remove Max/Min**: $O(\log n)$ — Swap root with last node and heapify.
  - **Peek**: $O(1)$ — Always access the root.
- **Applications**:
  - Priority queues, heap sort, graph algorithms (e.g., Dijkstra's algorithm).
- **Drawbacks**: Requires additional space, more complex than regular trees.
- **Insert**:

```
function insertHeap(heap, value):
    heap.add(value)  // Add to the end of the array
    index = heap.size - 1
    while index > 0 and heap[parent(index)] < heap[index]:
        swap(heap, index, parent(index))
        index = parent(index)
```

- **Remove Max (for Max Heap)**:

```
function removeMax(heap):
    if heap.isEmpty():
        return null
    max = heap[0]
    heap[0] = heap[heap.size - 1]
    heap.size -= 1
    heapifyDown(0)
    return max
```

- **Heapify Down**:

```
function heapifyDown(index):
    largest = index
    left = leftChild(index)
    right = rightChild(index)
    if left < heap.size and heap[left] > heap[largest]:
        largest = left
    if right < heap.size and heap[right] > heap[largest]:
        largest = right
    if largest != index:
        swap(heap, index, largest)
        heapifyDown(largest)
```

7. Hash Tables

- **Operations**:
  - **Insert/Search/Delete**: $O(1)$ on average (depends on hash function).
  - **Worst Case**: $O(n)$ if all keys hash to the same index (collision).
- **Collision Handling**:
  - **Separate Chaining**: Store colliding elements in a linked list or other structures.
  - **Open Addressing**: Linear probing, quadratic probing, or double hashing to find the next available slot.
- **Use Cases**:
  - Fast lookups, implementing associative arrays, caching.
- **Drawbacks**: Collisions reduce performance; requires a good hash function.
- **Insert**:

```
function insertHashTable(table, key, value):
    index = hash(key)
    if table[index] is empty:
        table[index] = new LinkedList()
    table[index].add((key, value))
```

- **Get**:

```
function getHashTable(table, key):
    index = hash(key)
    if table[index] is not empty:
        for each (k, v) in table[index]:
            if k == key:
                return v
    return null
```

- **Remove**:

```
function removeHashTable(table, key):
    index = hash(key)
    if table[index] is not empty:
        for each (k, v) in table[index]:
            if k == key:
                table[index].remove((k, v))
                return v
    return null
```

8. Tries

- **Definition**: A tree-like structure that stores strings by breaking them into characters.
- **Operations**:
  - **Search**: $O(m)$ — Where m is the length of the string.
  - **Insert**: $O(m)$
  - **Space Complexity**: $O(n)$, where n is the number of characters in all strings.
- **Compressed Trie**: Merges chains of nodes to reduce memory usage.
- **Applications**:
  - Dictionary implementation, autocomplete systems.
- **Drawbacks**: Can use a lot of memory when storing a large number of strings with common prefixes.
- **Insert**:

```
function insertTrie(root, word):
    currentNode = root
    for each character in word:
        if character not in currentNode.children:
            currentNode.children[character] = new TrieNode()
        currentNode = currentNode.children[character]
    currentNode.isEndOfWord = true
```

- **Get**:

```
function getTrie(root, word):
    currentNode = root
    for each character in word:
        if character not in currentNode.children:
            return null
        currentNode = currentNode.children[character]
    return currentNode if currentNode.isEndOfWord else null
```

- **Remove**:

```
function removeTrie(root, word, index=0):
    if index == length(word):
        root.isEndOfWord = false
        return isEmpty(root)
    char = word[index]
    if char in root.children:
        if removeTrie(root.children[char], word, index + 1):
            del root.children[char]
            return isEmpty(root)
    return false
```

- **Check if Empty**:

```
function isEmpty(node):
    return len(node.children) == 0
```

9. Skip Lists

- **Definition**: A probabilistic data structure for fast search, insertion, and deletion.
- **Operations**:
    - **Insert/Search/Delete**: $O(\log n)$ on average due to multiple levels of linked lists.

    - **Space Complexity**: $O(n)$.
- **Applications**:
    - As an alternative to balanced trees, in applications requiring fast search with simpler implementation.
- **Drawbacks**: Randomized structure may not always guarantee perfect balancing.
- **Insert**:

```
function insertSkipList(skipList, value):
    level = randomLevel()
    update = array of skipList.levels
    node = new Node(value, level)
    for i from 0 to level:
        update[i].next[i] = node
    skipList.size += 1
```

- **Search**:

```
function searchSkipList(skipList, value):
    current = skipList.head
    for i from highest level down to 0:
        while current.next[i] is not null and current.next[i].value
< value:
            current = current.next[i]
    current = current.next[0]
    return current if current.value == value else null
```

- **Delete**:

```
function deleteSkipList(skipList, value):
    update = array of skipList.levels
    current = skipList.head
    for i from highest level down to 0:
        while current.next[i] is not null and current.next[i].value
< value:
```

```
            current = current.next[i]
    current = current.next[0]
    if current is not null and current.value == value:
        for i from 0 to current.level:
            update[i].next[i] = current.next[i]
        skipList.size -= 1
```

10. Maps

- **Definition**: A collection of key-value pairs.
- **Operations**:
  - **get(k)**: Retrieve the value associated with the key.
  - **put(k, v)**: Insert a new key-value pair or update the value of an existing key.
  - **remove(k)**: Delete the entry with the key.
- **Hash Maps**:
  - **Operations**: $O(1)$ on average.
  - **Handling Collisions**: Separate chaining or open addressing.
- **Applications**: Storing and retrieving data with unique keys (e.g., address book, database indexing).
- **Drawbacks**: Collisions reduce efficiency; keys need to be hashable.
- **Put (Insert or Update)**:

```
function putMap(map, key, value):
    index = hash(key)
    if map[index] is empty:
        map[index] = new LinkedList()
    for each (k, v) in map[index]:
        if k == key:
            v = value  // Update value
```

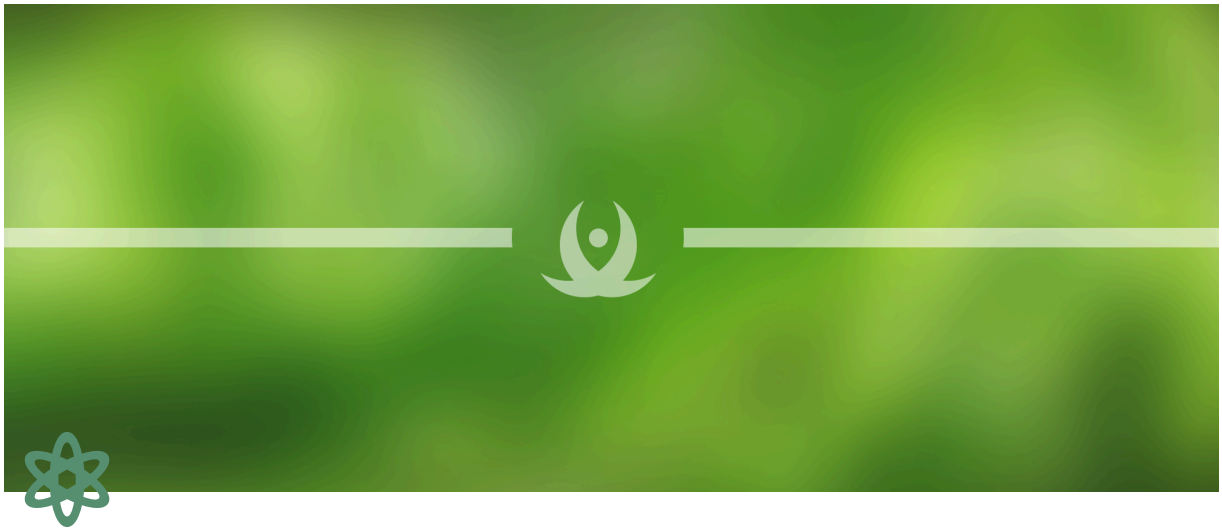
```
            return
    map[index].add((key, value))
```

- **Get**:

```
function getMap(map, key):
    index = hash(key)
    if map[index] is not empty:
        for each (k, v) in map[index]:
            if k == key:
                return v
    return null
```

- **Remove**:

```
function removeMap(map, key):
    index = hash(key)
    if map[index] is not empty:
        for each (k, v) in map[index]:
            if k == key:
                map[index].remove((k, v))
                return v
    return null
```



16. Graphs — Theory, ADT & Data Structure Implementations

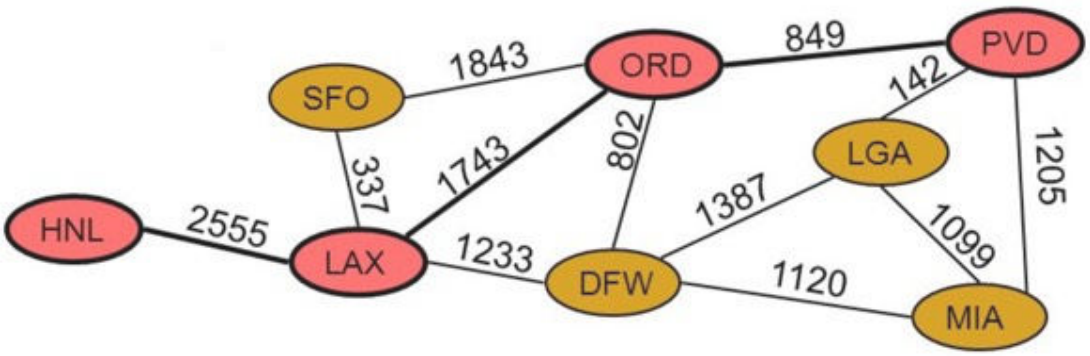
Graph Basics

Definition

> **GRAPH:** A pair (V, E) where
>
> - V is a set of **vertices** (nodes),
> - E is a collection of **edges** (pairs of vertices).
>
> Vertices and edges are positions that store user-provided elements.

Example:

A vertex stores a three-letter airport code; an edge stores the mileage between two airports.



Edge Types

> **DIRECTED EDGE:** An ordered pair (u, v) with origin u and destination v.
>
> **UNDIRECTED EDGE:** An unordered pair $\{u, v\}$.

> **DIRECTED GRAPH:** All edges are directed (e.g., one-way flight network).
>
> **UNDIRECTED GRAPH:** All edges are undirected (e.g., bidirectional flight network).

Applications

- Electronic circuits (PCBs, integrated circuits)
- Transportation networks (highways, flight routes)
- Computer networks (LAN, Internet, Web)
- Databases (Entity–Relationship diagrams)

Terminology

Endpoints & Incidence

> **ENDPOINTS:** For edge $e = (u, v)$, the vertices u and v are its endpoints.
>
> **INCIDENT EDGES:** Edges that have a given vertex as one of their endpoints.

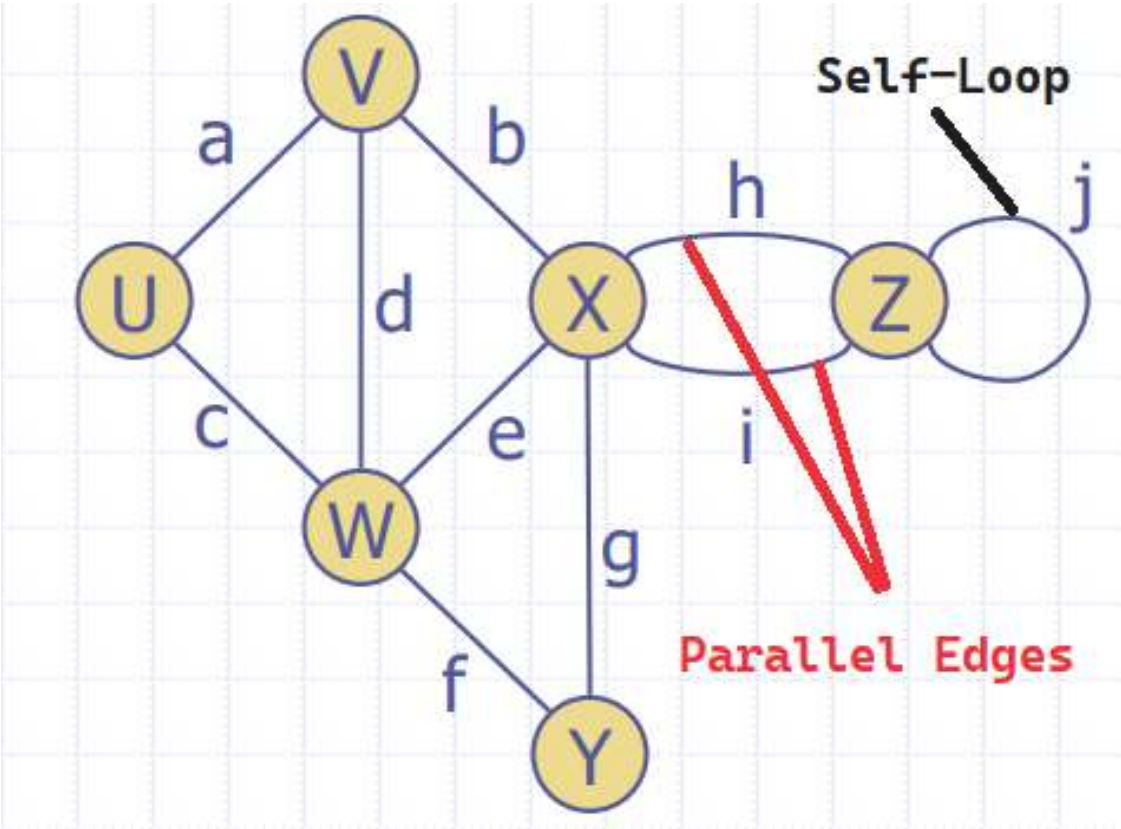
Adjacency & Degree

> **ADJACENT VERTICES:** Two vertices are adjacent if they are connected by an edge.

DEGREE ($\deg(v)$): Number of edges incident on vertex v.

PARALLEL EDGES: Multiple edges connecting the same pair of vertices.

SELF-LOOP: An edge connecting a vertex to itself.



Paths & Cycles

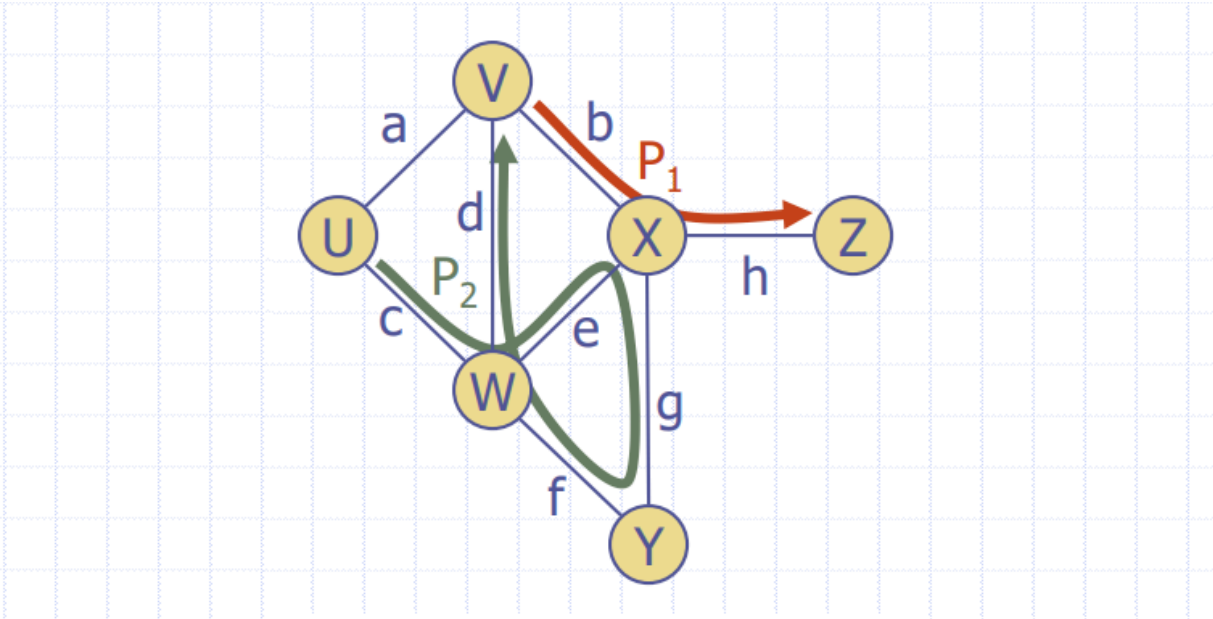
Paths

PATH: A sequence of alternating vertices and edges, beginning and ending with vertices.

SIMPLE PATH: A path with all distinct vertices and edges.

Examples:

- $P_1 = (V, b, X, h, Z)$ is simple.
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is not simple.

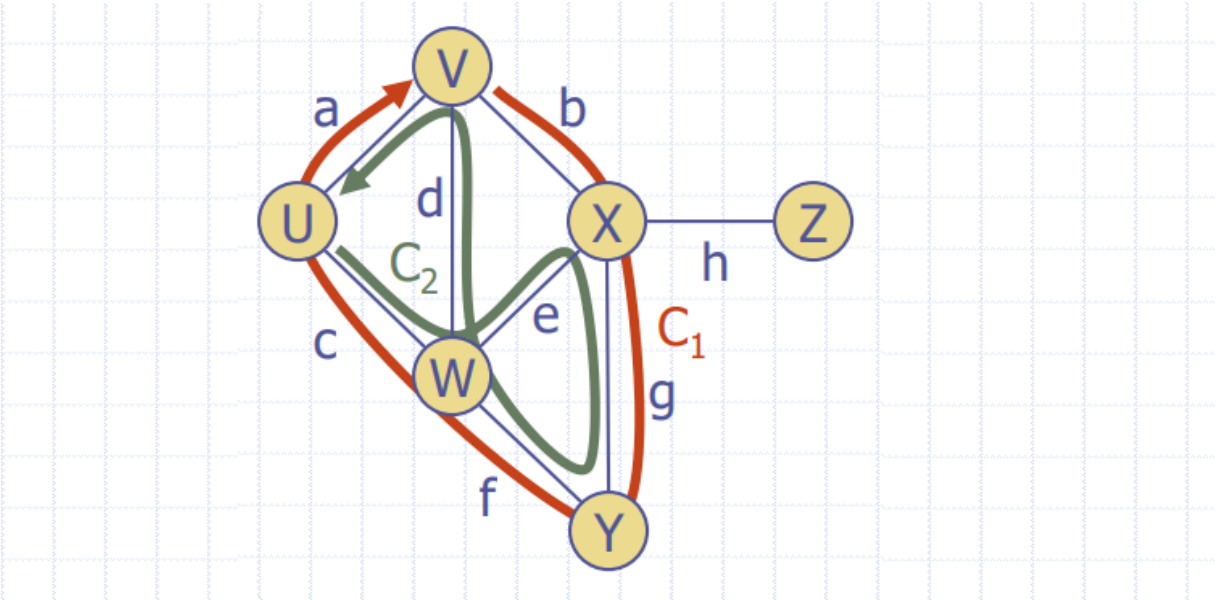


Cycles

> **CYCLE:** A circular sequence of alternating vertices and edges.
>
> **SIMPLE CYCLE:** A cycle with all distinct vertices and edges.

Examples:

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, \circlearrowright)$ is simple.
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, \circlearrowright)$ is not simple.



Graph Properties

> **NOTATION:**
>
> n = number of vertices,
>
> m = number of edges,
>
> $\deg(v)$ = degree of vertex v.

Property 1: Sum of Vertex Degrees

> **Property 1:** $\sum_{v \in V} \deg(v) = 2m.$
>
> **Proof:** Each undirected edge contributes 1 to the degree of each endpoint.

Property 2: Maximum Number of Edges in an Undirected Graph

> **Property 2:** In an undirected graph with no self-loops or parallel edges,
>
> $m \leq \frac{n(n-1)}{2}.$
>
> **Proof:** Each of the n vertices can connect to at most $(n-1)$ others; dividing by 2 avoids double-counting.

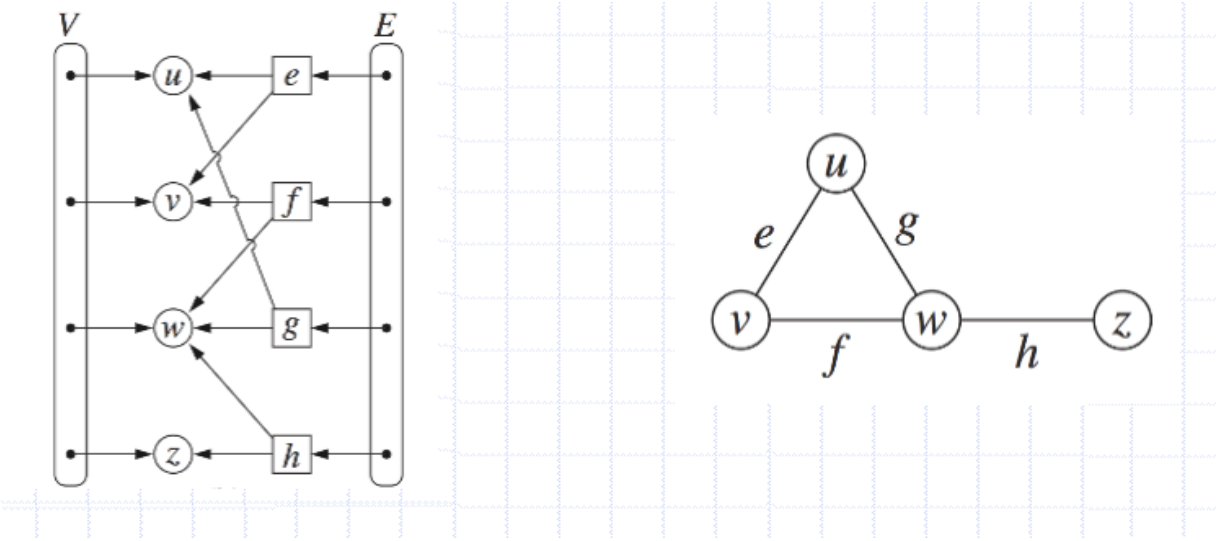
> **Directed Graph Bound:** Without loops,
> $m \leq n(n-1)$.

Graph ADT

> **VERTEX:** Object storing an element; supports element().
>
> **EDGE:** Object storing an element and references to origin and destination vertices; supports `element()`.
>
> **GRAPH:** Encapsulates vertex and edge collections and supports graph operations.

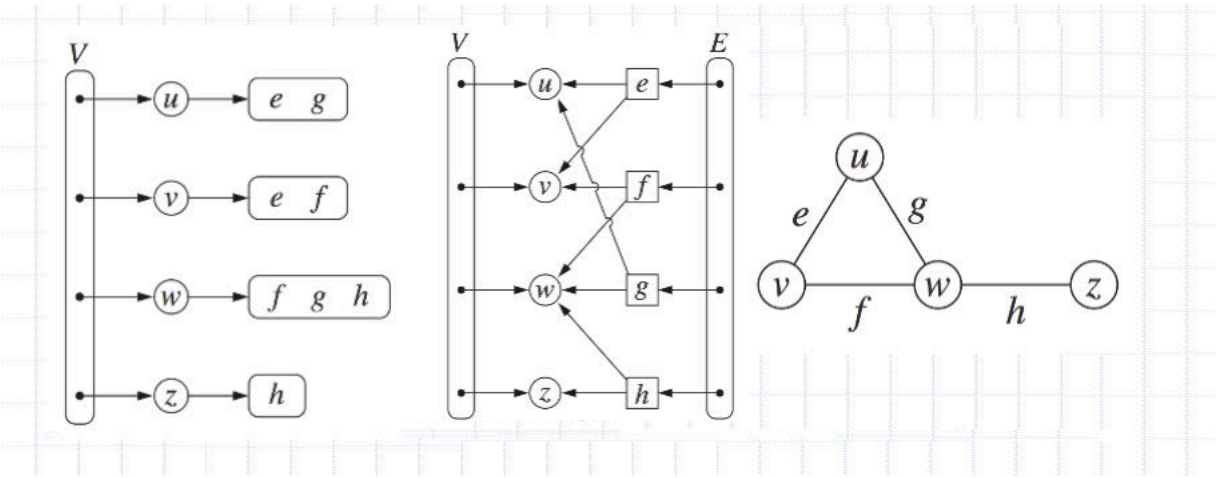
Edge List Structure (ELS)

> **VERTEX OBJECT:**
>
> - `element`
> - reference to position in vertex sequence
>
> **EDGE OBJECT:**
>
> - `element`
> - references to origin and destination vertices
> - reference to position in edge sequence
>
> **VERTEX SEQUENCE:** Sequence of all vertex objects
>
> **EDGE SEQUENCE:** Sequence of all edge objects



Adjacency List Structure (ALS)

> **INCIDENCE SEQUENCE:** For each vertex, a list of references to its incident edges.
>
> **AUGMENTED EDGE OBJECT:** Contains pointers to its positions in both endpoints' incidence lists.



Complexity Analysis

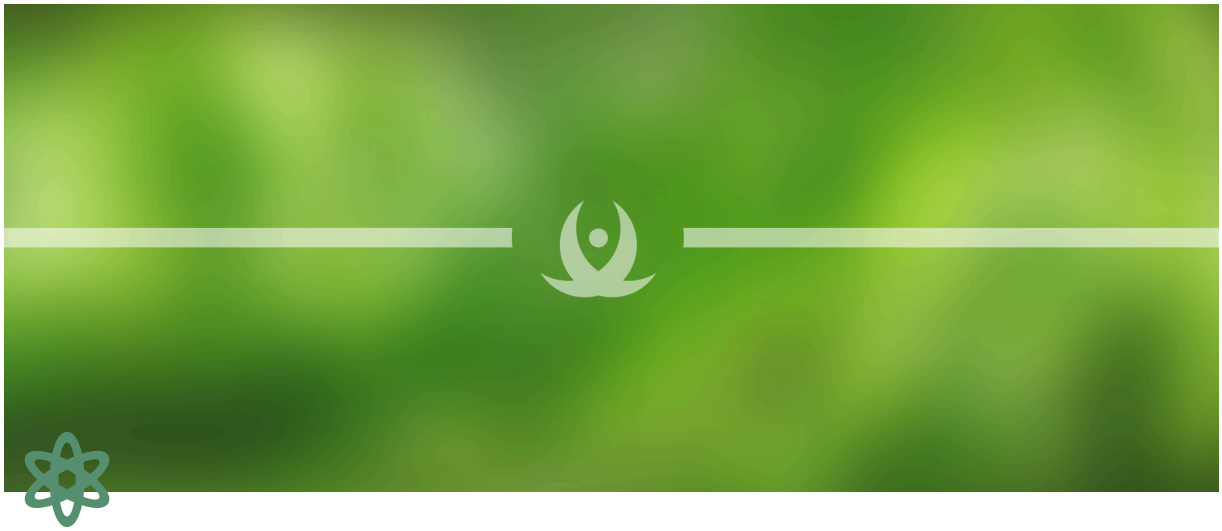
Operation	ELS Complexity	ALS Complexity
insertV	$O(1)$	$O(1)$
insertE	$O(1)$	$O(1)$

Operation	ELS Complexity	ALS Complexity
removeV	$O(m)$	$O(d(v))$ (or $O(m)$ worst-case)
removeE	$O(1)$	$O(1)$
opposite	$O(1)$	$O(1)$
incident	$O(m)$	$O(d(v))$
areAdjacent	$O(m)$	$O(\min(\deg(u), \deg(v)))$
insert (list)	$O(1)$	$O(1)$
remove (list)	$O(1)$	$O(1)$

- In ALS, `removeV` traverses v's incidence list and removes each edge in $O(1)$ per edge.

Final Summary & Takeaways

> SUMMARY: Covered graph definitions, types, terminology, fundamental properties, the Graph ADT, two storage structures (ELS & ALS), and their operation complexities.
>
> **KEY TAKEAWAYS:**
>
> - **Choice of structure** affects local vs. global operation cost.
> - **Property:** $\sum \deg(v) = 2m$ underpins many algorithms.
> - **Edge bounds** guide worst-case scenario analysis.
> - **ALS** excels at localized queries; **ELS** simplifies global edge management.



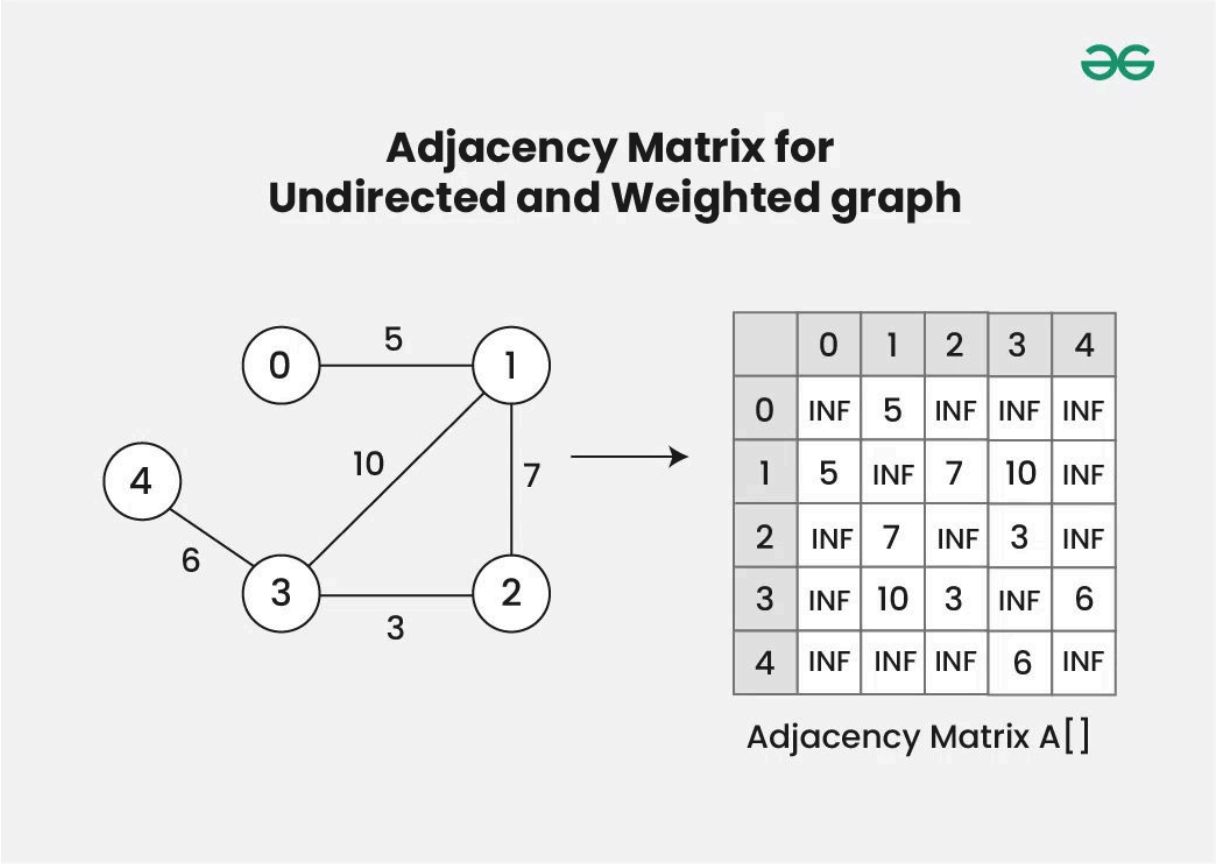
17. Graph Representations & Breadth-First Search

Adjacency Matrix Structure

Definition & Structure

> **ADJACENCY MATRIX:** A 2D array A of size n × n (for n vertices) where
>
> - `A[i][j]` holds a reference to the edge object between vertices *i* and *j*, or
> - `Null` (or `0` in the "old-fashioned" version) if no edge exists.

> **AUGMENTED VERTEX OBJECT:** Each vertex carries an integer key (its index) allowing direct access into the matrix.



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF

Performance Comparison

For a graph with n vertices and m edges (no parallel edges or self-loops):

Operation	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(v)	$O(m)$	$O(deg(v))$	$O(n)$
areAdjacent(v, w)	$O(m)$	$O(min(deg(v),deg(w)))$	$O(1)$
insertVertex(o)	$O(1)$	$O(1)$	$O(n^2)$
insertEdge(v, w, o)	$O(1)$	$O(1)$	$O(1)$
removeVertex(v)	$O(m)$	$O(deg(v))$	$O(n^2)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$

> **Remark: Adjacency-matrix** excels at constant-time **adjacency checks**, but costs $O(n^2)$ space.

Breadth-First Search (BFS)

Overview

> **BFS TRAVERSAL:** Visits all vertices and edges of a graph in layers, computes connected components, and builds a spanning forest in O(n + m) time.

- Determines connectivity of *G*
- Computes connected components and spanning forest
- Finds shortest path (fewest edges) between two vertices
- Detects a simple cycle if one exists

BFS Algorithm

```
BFS(G, s):
  L[0] ← [ s ]
  setLabel(s, VISITED)
  i ← 0

  while L[i] not empty:
    L[i+1] ← []
    for v in L[i]:
      for e in incidentEdges(v):
        if getLabel(e) = UNEXPLORED:
          w ← opposite(v, e)
          if getLabel(w) = UNEXPLORED:
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            L[i+1].append(w)
          else:
            setLabel(e, CROSS)
    i ← i +
```

> **Comment:** Traverses the graph from a single start vertex in “layers,” marking edges as discovery or cross and building successive frontier lists.

```
BFS(G):
  for each u in G.vertices:
    setLabel(u, UNEXPLORED)
  for each e in G.edges:
    setLabel(e, UNEXPLORED)
  for each v in G.vertices:
    if getLabel(v) = UNEXPLORED:
      BFS(G, v)
```

> **Comment:** Prepares every vertex and edge for exploration, then invokes the core routine on each unvisited component to cover the entire graph.

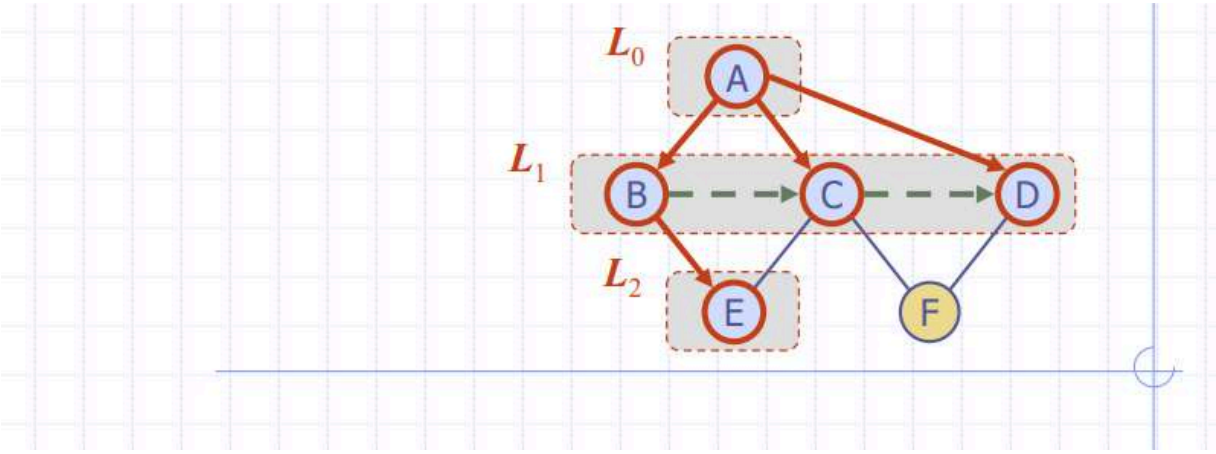
Example Walkthrough

Starting at vertex **s = A**:

1. **Layer L_0:** `{A}`
   - Only the start vertex is visited.
2. **Layer L_1:** `{B, C, D}`
   - All neighbors of **A** discovered via **DISCOVERY** edges (solid red).
3. **Layer L_2:** `{E, F}`
   - Neighbors of **B**, **C**, or **D** not yet visited, discovered in the next wave.

   Note:
   - **Solid red arrows** = **DISCOVERY** edges that bring a new vertex into the frontier.
   - **Dashed green arrows** = **CROSS** edges encountered between already-visited vertices.

...and the process continues until no unexplored edges remain.



Properties of BFS

> **Property 1:** BFS(G, s) visits all vertices and edges of the connected component G_s.

> **Property 2:** Discovery edges form a spanning tree T_s of G_s.

> **Property 3:** For any vertex v in layer L_i:
>
> - The path in T_s from *s* to *v* has exactly *i* edges.
> - No path in G_s from *s* to *v* has fewer than *i* edges.

Analysis

- Labelling operations (vertex/edge) are *O(1)* each.
- Each vertex is labeled twice (UNEXPLORED → VISITED).
- Each edge is labeled twice (UNEXPLORED → DISCOVERY/CROSS).
- Each vertex appears once in some layer list L[·].
- `incidentEdges(v)` called once per vertex.
- **Time complexity:** *O(n + m)* with an adjacency-list representation (since $\sum_v \deg(v) = 2m$).

Applications of BFS

Using the BFS template, we can solve in *O(n + m)* time:

- **Connected Components:** Identify all components by running BFS from unexplored vertices.
- **Spanning Forest:** Union of BFS trees from each component.
- **Cycle Detection:** If any edge is labeled CROSS in an undirected graph, a cycle exists.
- **Shortest Path:** Find a minimum-edge path between two vertices in an unweighted graph.

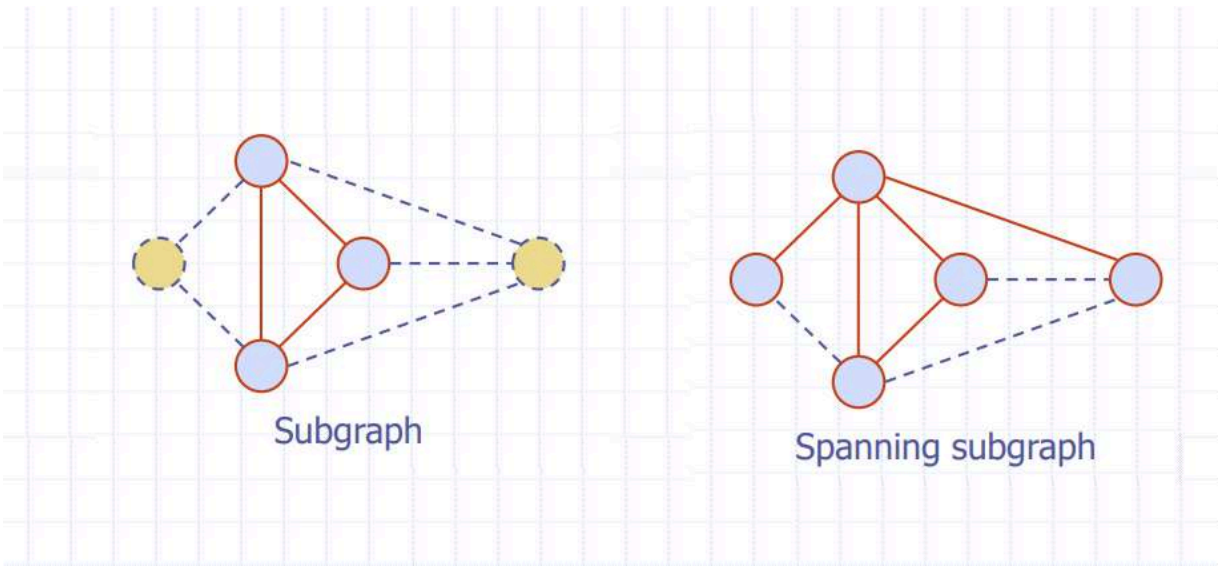
Final Summary & Takeaways

- **Representation Choice:**
  - Adjacency-matrix: constant-time adjacency, high space cost ($O(n^2)$).
  - Adjacency-list: space-efficient ($O(n + m)$), linear-time adjacency checks.
- **BFS Key Points:**
  - Traverses in layers, builds spanning forest.
  - Guarantees shortest path (edge-count) in unweighted graphs.
  - Runs in $O(n + m)$ time with adjacency lists.
- **Common Pitfalls:**
  - Using adjacency matrix for very sparse graphs leads to wasted space.
  - Forgetting to mark CROSS edges can mask cycle detection.
  - Omitting initialization of all labels before traversal can produce incorrect results.

18. Graph Traversal, DFS, Structural Properties

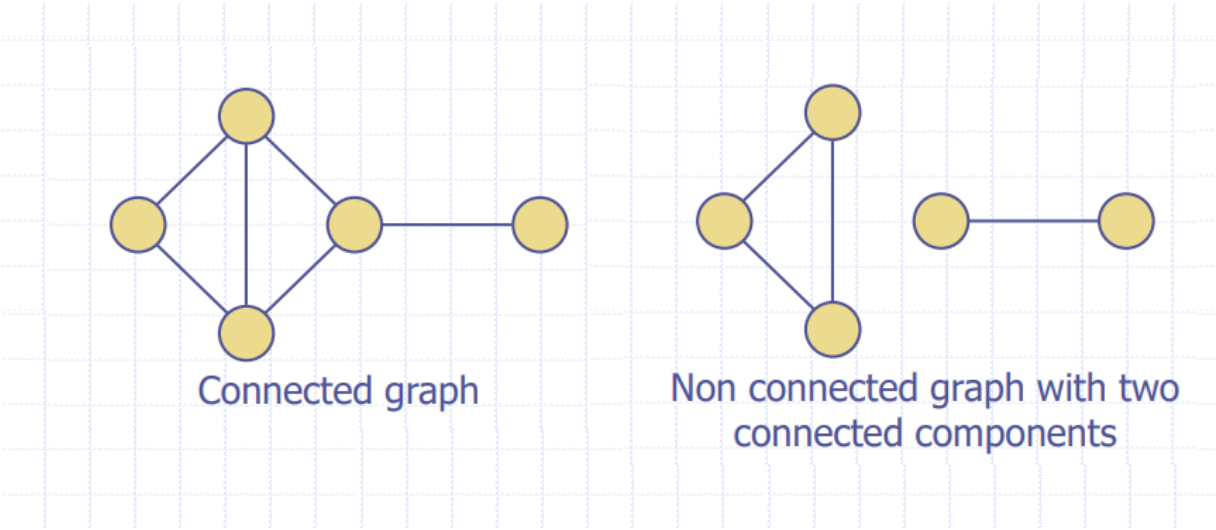
Subgraphs & Spanning Subgraphs

> **Subgraph:** A graph S whose vertices and edges are subsets of those of G.
>
> **Spanning Subgraph:** A subgraph containing all vertices of G (but possibly fewer edges).



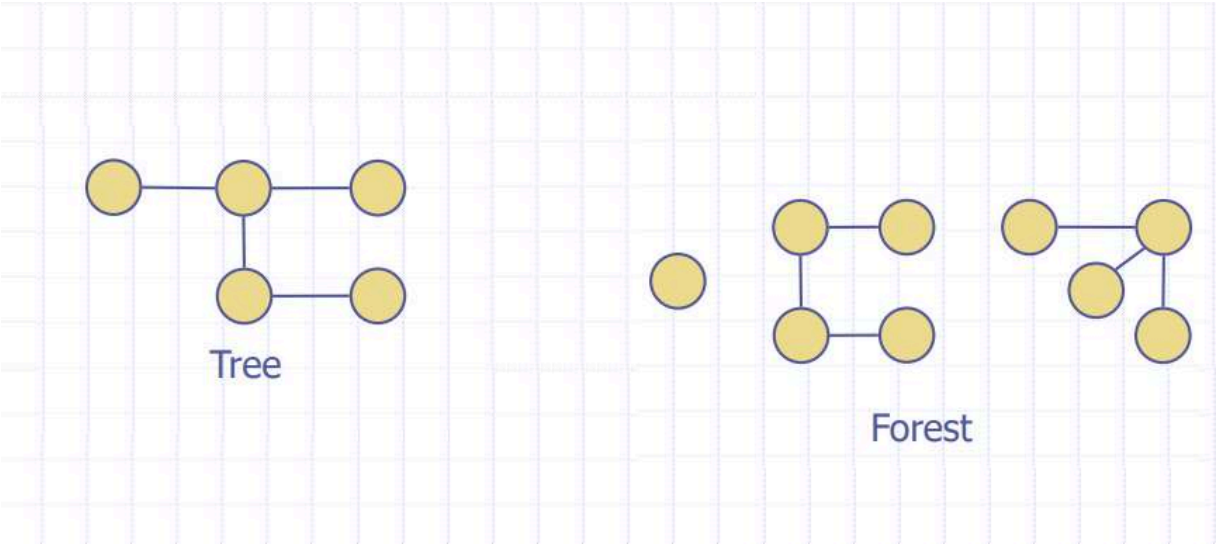
Connectivity & Components

> **Connected Graph:** Every pair of vertices is joined by a path.
>
> **Connected Component:** A maximal connected subgraph of G.



Trees & Forests

> **Tree:** A connected, cycle-free undirected graph.
>
> **Forest:** An undirected, cycle-free graph (a collection of trees).
>
> **Spanning Tree:** A spanning subgraph of a connected graph that is a tree.
>
> **Spanning Forest:** A spanning acyclic subgraph of any graph.



Depth-First Search (DFS)

Overview

> **DFS:** Traversal that explores as far as possible along each branch before backtracking.

- Discovers connected components and builds a spanning forest.
- Labels edges as **discovery** (part of DFS tree) or **back** (to an ancestor).
- Runs in $O(n + m)$ time for n vertices and m edges.

DFS Template

- Mark start vertex as VISITED.
- For each unexplored incident edge, follow to an unexplored neighbor, label edge DISCOVERY, recurse, then backtrack.
- Label any remaining unexplored edge as BACK.

DFS Specializations

Path Finding

Use a stack to record the path from start u to target z; halt when z is discovered.

> Example: Finding any simple path in a maze graph via DFS.

Cycle Finding

Track edges on recursion stack; upon encountering a BACK edge (v, w), extract the cycle from w to v.

> **Example:** Detecting a cycle in an undirected graph when a discovery edge leads to an already VISITED vertex.

DFS vs. BFS Edge Classification

- **Discovery Edge:** Tree edge in DFS/BFS.
- **Back Edge (DFS):** Connects to ancestor.
- **Forward/Cross Edges (Directed DFS):** Connect to proper descendant or to a node in the same/lower level (see Directed DFS).

Directed Graphs (Digraphs)

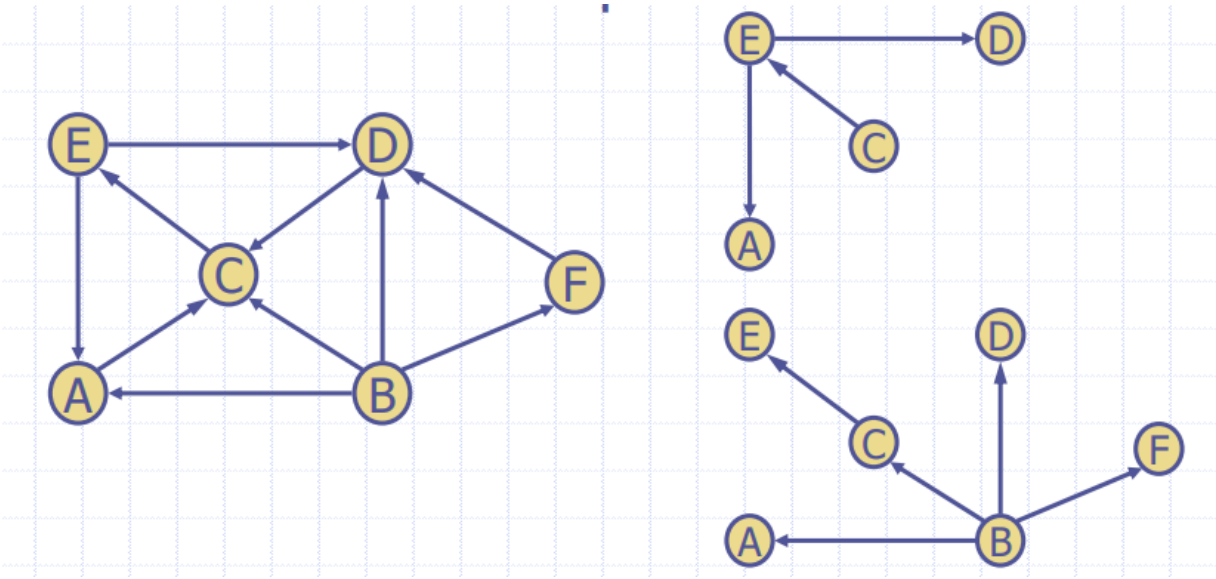
> Digraph: A graph with directed edges (a, b).

- Store separate adjacency lists for outgoing and incoming edges for efficient traversal.
- Used for modeling one-way streets, flight routes, task scheduling.

Directed DFS & Reachability

> **Directed DFS:** Traverse only along edge direction, labeling edges as discovery, back, forward, or cross.

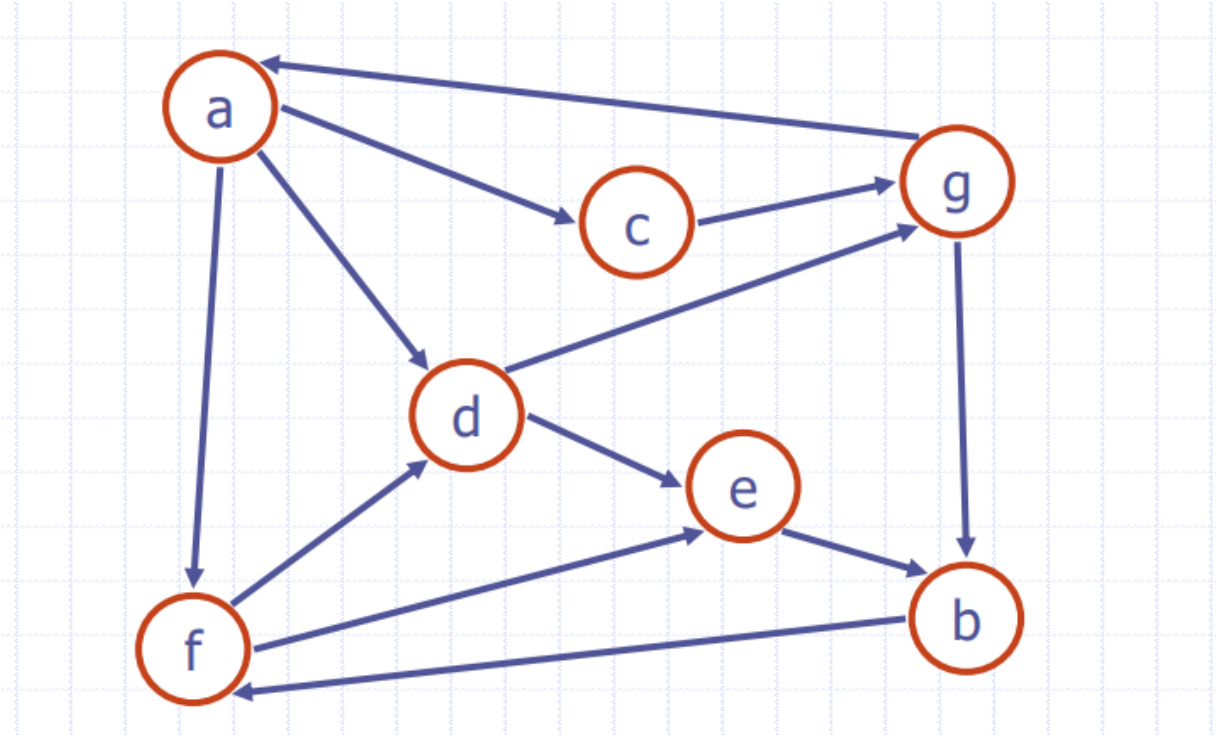
- Determines vertices reachable from a start vertex s.



Strong Connectivity & SCCs

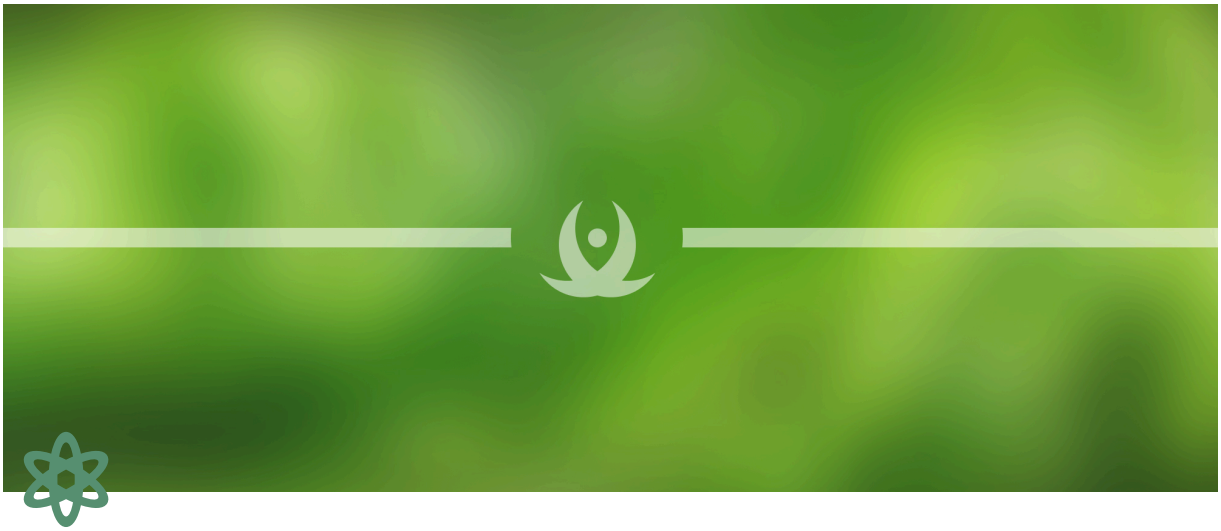
> **Strongly Connected:** Every vertex can reach every other via directed paths.
>
> **Strongly Connected Component (SCC):** A maximal strongly connected subgraph.

- **Algorithm:**
  - Run DFS from arbitrary v; if any vertex is unvisited, not strongly connected.
  - Reverse all edges and run DFS from v again; if all visited, graph is strongly connected.
- **SCC Decomposition:** More advanced DFS-based methods (e.g., Kosaraju's) find all SCCs in $O(n + m)$.



Final Summary & Takeaways

- Subgraphs, connectivity, trees & forests underpin graph structure.
- DFS builds spanning forests, labels edges, and runs in linear time.
- Specializations of DFS solve path-finding and cycle detection.
- Directed graphs require careful edge-direction handling; DFS reveals reachability.
- Strong connectivity checks and SCC algorithms rely on two DFS passes or variants.



19. Graph Algorithms: Reachability, Ordering & Shortest Paths

Transitive Closure

> **Transitive Closure:** Given a digraph $G = (V, E)$, its transitive closure $G^{=}(V, E^{)}$ has an edge (u, v) in E^* whenever there is a directed path from u to v in G.

Computing Transitive Closure

> **DFS-Based Method:** Run DFS from each vertex u; mark all reachable v and add (u, v) to E^*.
>
> **Time Complexity:** $O(|V| \cdot (|V| + |E|))$.

> **Floyd–Warshall Algorithm:** A dynamic-programming approach using a distance matrix to infer reachability in $O(|V|^3)$.

Example:

For G with edges $A \to B$, $B \to C$, the closure G^* adds $A \to C$ since $A \to B \to C$.

Directed Acyclic Graphs & Topological Ordering

> **DAG:** A digraph with no directed cycles.
>
> **Topological Ordering:** A labelling of vertices $v_1, \ldots, v_n$ such that for every edge (v_i, v_j), $i < j$.

Existence Theorem

> **Theorem:** A digraph admits a topological ordering if and only if it is a DAG.

Topological Sorting Algorithms

> **Remove-Sink Method:**
>
> - Repeatedly remove a vertex with no outgoing edges, assign it the highest remaining label, and delete it from the graph.
> - Runs in $O(|V| + |E|)$.

> **DFS-Based Method:**
>
> - Perform DFS on each unvisited vertex.
> - After exploring all descendants of v, prepend v to a list.
> - The final list is a topological ordering in $O(|V| + |E|)$.

Shortest Paths in Weighted Graphs

> **Weighted Graph:** Each edge (u, v) has a weight $w(u, v)$.
>
> **Shortest Path Problem:** Find a path from s to t minimizing the sum of edge weights.

Fundamental Properties

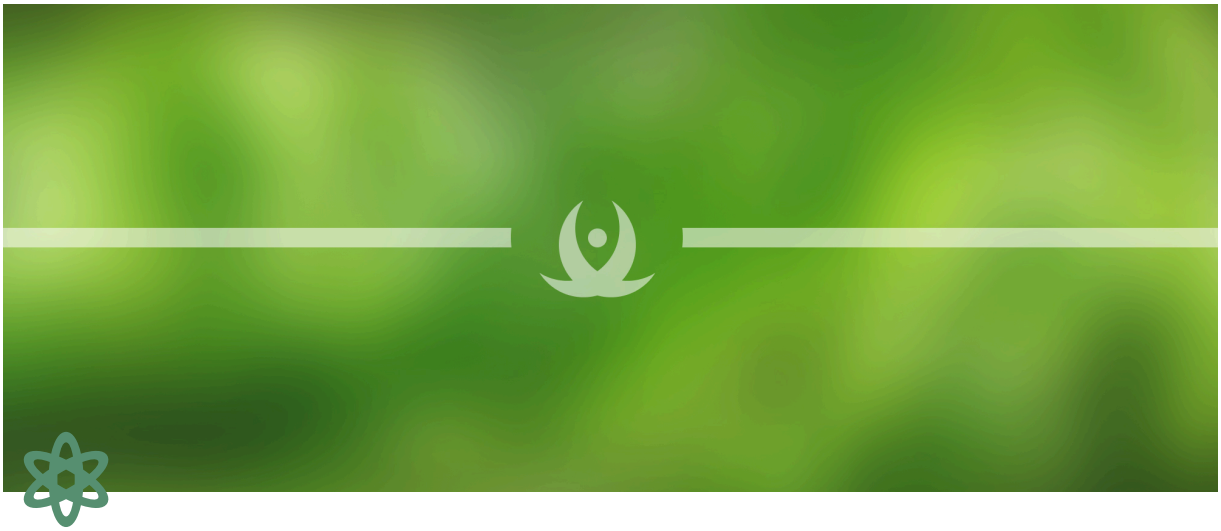
> **Subpath Optimality:** Every subpath of a shortest path is itself a shortest path.
>
> **Shortest-Path Tree:** For a fixed source s, the collection of shortest paths from s to all reachable vertices forms a tree.

Example:

In a flight-route graph with weighted edges as distances, the shortest path from Providence to Honolulu is found by accumulating minimal distances through intermediate hubs.

Final Summary & Takeaways

- **Transitive Closure** reveals all reachabilities; computed via repeated DFS or Floyd–Warshall.
- **DAGs** permit topological ordering; detect cycles by failure to order.
- **Topological Sort** can be implemented by removing sinks or via DFS postorder.
- **Shortest Paths** rely on subpath optimality; yield a tree of minimal-cost routes.
- **Common Mistake:** Applying topological sort on graphs with cycles or using unweighted methods for weighted-graph shortest paths.



20. Dijkstra's Algorithm

Graph Preliminaries

Definitions

> **GRAPH:** A pair $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges.
>
> **WEIGHTED GRAPH:** A graph where each edge $e \in E$ has an associated weight $w(e)$.
>
> **PATH:** A sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $(v_{i-1}, v_i) \in E$.
>
> **PATH WEIGHT:** The sum of the weights of edges along the path: $\sum_{i=1}^{k} w(v_{i-1}, v_i)$.
>
> **SPANNING TREE:** A subgraph of G that includes all vertices and is itself a tree.

Shortest Path Problem

Problem Statement

> **SHORTEST-PATH PROBLEM:** Given a weighted graph $G = (V, E)$ and source $s \in V$, compute the minimum path weight $d(v)$ from s to every $v \in V$.

Dijkstra's Algorithm

> **DIJKSTRA'S ALGORITHM:** Computes shortest-path distances $d(v)$ from source s in graphs with nonnegative edge weights.

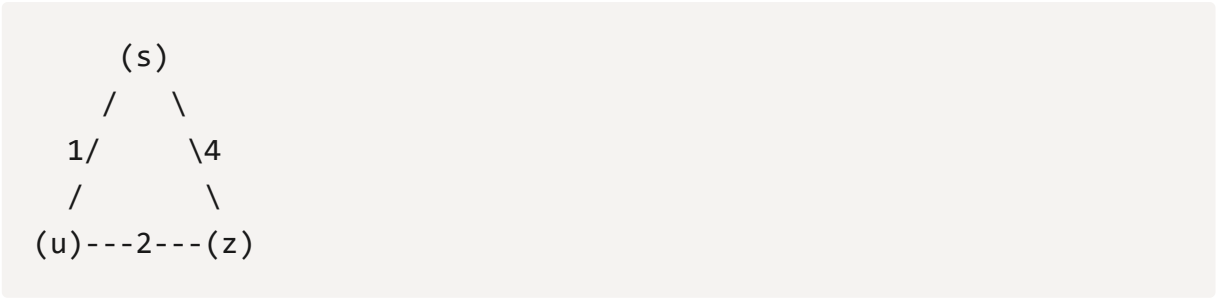
1. **Initialization:**
   - For all $v \in V$, set $d(v) = \infty$; set $d(s) = 0$.
   - Let cloud $S = \emptyset$.
2. **Main Loop:**
   - Extract $u \notin S$ with minimum $d(u)$.
   - Add u to S.
   - For each neighbor z of u, **relax** edge (u, z) :

$$d(z) \leftarrow \min\{d(z),\ d(u) + w(u, z)\}.$$

3. **Termination:** When $S = V$, distances $d(v)$ are final.

Example

Consider graph:



- **Step 1:** $d(s) = 0$; $d(u) = \infty$; $d(z) = \infty$.
- **Step 2:** Relax from s: $d(u) = 1$; $d(z) = 4$.
- **Step 3:** Extract u ($d = 1$); relax (u, z) : $d(z) = 3$.
- **Step 4:** Extract z ($d = 3$) → done.

Complexity Analysis

> **TIME:** $O((|V| + |E|) \log |V|)$ using a binary-heap priority queue.

Correctness & Constraints

> Requires nonnegative edge weights; fails on negative-weight edges.

Bellman–Ford Algorithm

> **BELLMAN–FORD:** Handles negative weights (directed graphs) in $O(|V| \cdot |E|)$ time and detects negative cycles.

```
for each v in V:
  if v = s then d(v)=0 else d(v)=∞
for i = 1 to |V|-1:
  for each edge (u,z) in E:
    if d(u)+w(u,z) < d(z):
      d(z) = d(u)+w(u,z)
```

Minimum Spanning Trees

Problem Statement

> **MINIMUM SPANNING TREE (MST):** A spanning tree of G with minimum total edge weight.

Cycle & Partition Properties

> **Cycle Property:** For any cycle in G, the maximum-weight edge on that cycle is not included in some MST.
>
> **Partition Property:** For any cut (U, V) of V, the minimum-weight edge crossing the cut is in some MST.

Final Summary & Takeaways

- **Shortest Paths:**
  - Dijkstra's: $O((V + E) \log V)$, nonnegative weights.
  - Bellman–Ford: $O(VE)$, supports negatives and detects cycles.

- **Greedy Paradigm:** Both problems solved by iteratively choosing the local optimum (minimum distance or minimum edge).
- **Pitfalls:** Dijkstra breaks with negative weights; Kruskal needs cycle detection; Bellman–Ford is slower but more versatile.

21. Minimum Spanning Trees

Basics of Spanning Trees

Definitions

> **SPANNING TREE:** A subgraph of a graph that is a tree containing all vertices.
>
> **MINIMUM SPANNING TREE (MST):** A spanning tree in a weighted graph with the smallest possible total edge weight.

Properties of MST

Cycle Property

> **CYCLE PROPERTY:** For any cycle in the graph, the maximum-weight edge on that cycle does not belong to any MST.
>
> **Sketch of Proof:** Removing the heaviest edge from the cycle reduces total weight while preserving connectivity.

Partition (Cut) Property

> **PARTITION PROPERTY:** For any partition of the graph's vertices into sets U and V, the minimum-weight edge crossing the cut (U, V) belongs to some MST.

> **Sketch of Proof:** If the MST does not include this lightest crossing edge, swapping it with a heavier edge across the cut yields a lighter spanning tree.

Greedy Algorithms for MST

Kruskal's Algorithm

> **KRUSKAL'S ALGORITHM:** Builds MST by sorting edges and adding the smallest ones that do not form a cycle.
>
> **Procedure:**
>
> 1. Create a forest where each vertex is its own tree.
> 2. Sort all edges by increasing weight.
> 3. For each edge (u, v) in sorted order:
>   - If u and v are in different trees, add (u, v) to the MST and union their trees.
> 4. Repeat until the forest merges into a single tree.

Prim's Algorithm

> **PRIM'S ALGORITHM:** Grows an MST from an initial vertex by repeatedly adding the cheapest edge crossing the current tree boundary.
>
> **Procedure:**
>
> 1. Start with a single vertex s in set S.
> 2. Initialize key(v) = ∞ for all v ≠ s; set parent(v) = null.
> 3. While S ≠ V:
>   - Select u ∉ S with minimum key(u).
>   - Add u to S.
>   - For each neighbor v of u not in S, if weight(u, v) < key(v), set key(v) = weight(u, v) and parent(v) = u.

Data Structures and Complexity

Union-Find Structure

> **UNION-FIND:** Maintains disjoint sets with operations makeSet, find, and union for Kruskal's algorithm.

- makeSet(u): Create set containing u.
- find(u): Find representative of u's set.
- union(A, B): Merge sets A and B.
  Use union by rank and path compression for near-constant time operations.

Complexity Analysis

> **Kruskal's Algorithm:** O(E log E) for sorting edges and union-find operations.
>
> **Prim's Algorithm:** O((V + E) log V) using a binary heap for the priority queue.

Example Execution

Kruskal's Example

Graph with vertices {A, B, C, D}, edges with weights.

Steps:

1. Sort edges: (A, B), (C, D), ...
2. Add edges, skipping those forming cycles.
3. Final MST: list of edges.

Prim's Example

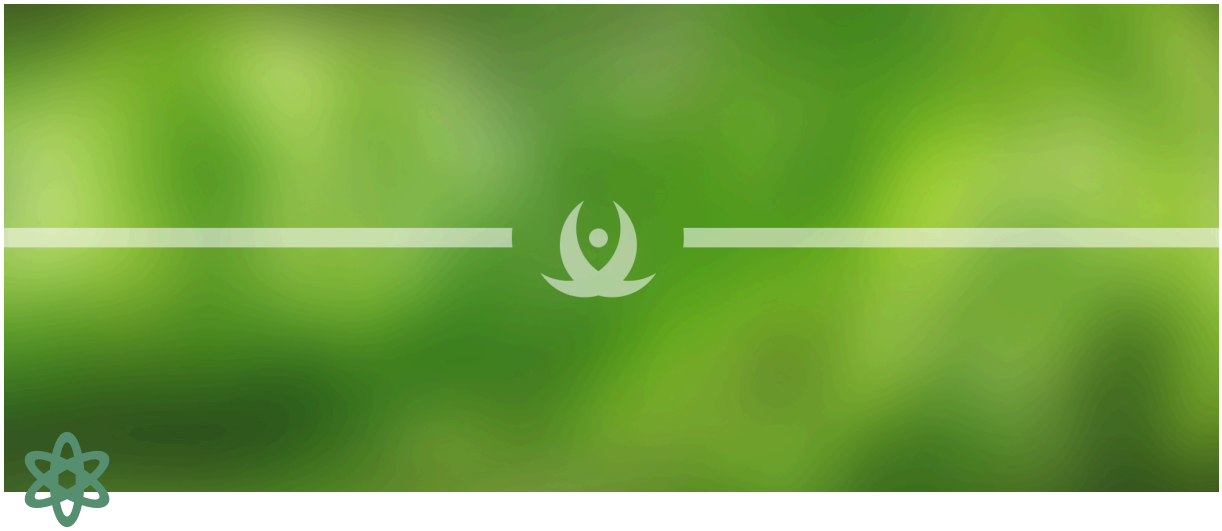
Same graph as above.

Steps:

1. Start at A, set keys.
2. Add the lightest outgoing edge, update keys.
3. Continue until all vertices included.

Final Summary & Takeaways

- MSTs connect all vertices with minimum total weight.
- Cycle and partition properties ensure greedy choice correctness.
- Kruskal's uses edge sorting and union-find; Prim's uses a priority queue.
- Both algorithms run in O(E log V) time for sparse graphs.



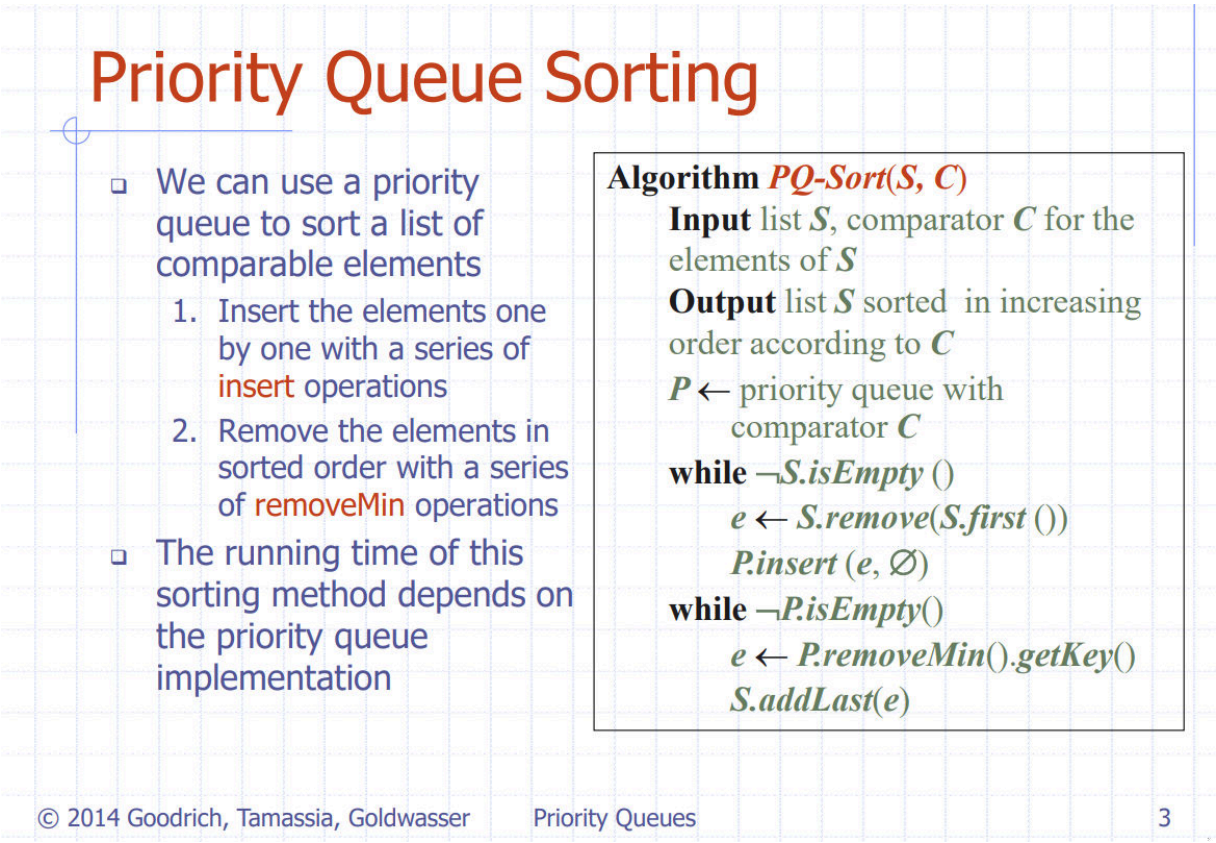
22. Priority Queues & Heap-Based Sorting

Priority Queue ADT

A priority queue stores a collection of entries `(key, value)` so that the entry with the smallest (or largest) key can be accessed quickly. Common operations:

- **insert(k, v)** — add a new entry
- **min()** — peek at the entry with smallest key
- **removeMin()** — remove and return the entry with smallest key
- **size(), isEmpty()** — query the number of entries

All operations must maintain the priority ordering.



Implementations & Complexities

- **Unsorted list**
  - insert in O(1) by appending
  - min() / removeMin() in O(n) by scanning
  - overall PQ-sort cost: $O(n^2)$
- **Sorted list**
  - insert in O(n) by shifting into position
  - min() / removeMin() in O(1) at head
  - overall PQ-sort cost: $O(n^2)$
- **Binary heap (array-based)**
  - insert: add at end and "up-heap" in O(log n)
  - min(): O(1) at root

- removeMin: swap root with last, “down-heap” in O(log n)
- size(), isEmpty(): O(1)

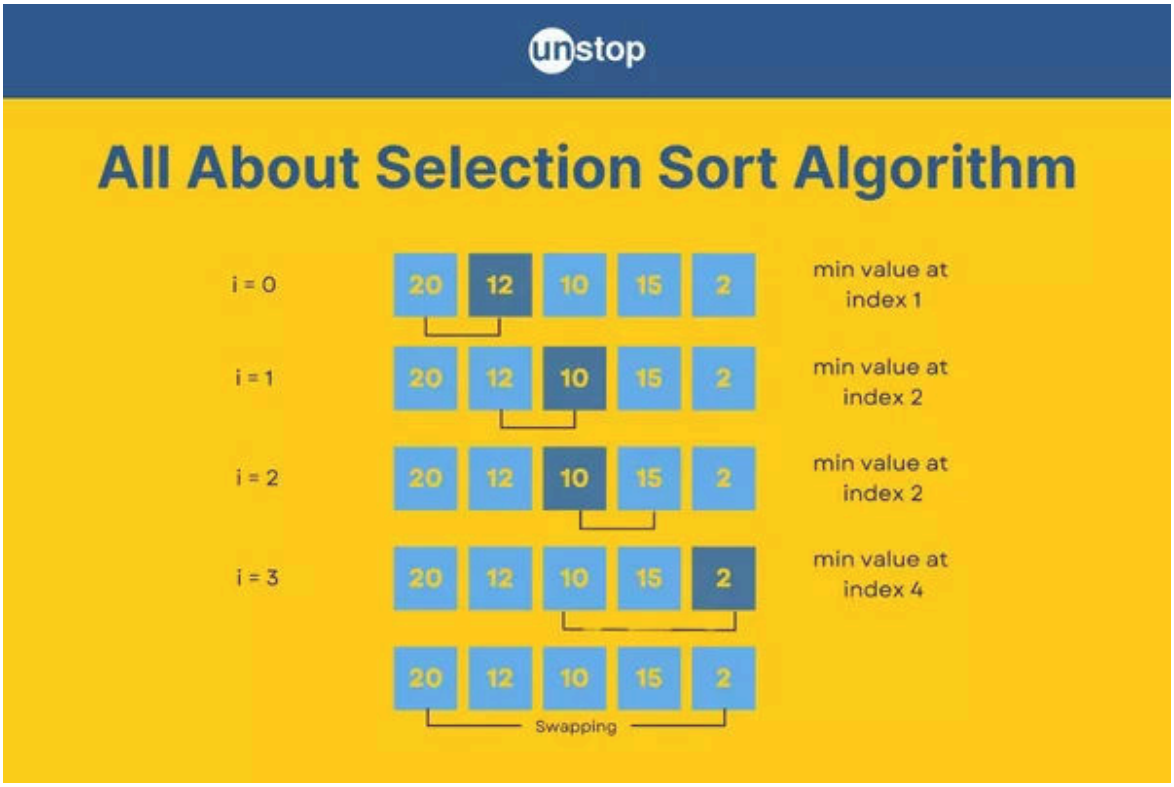
Priority-Queue Sorting

1. Insert all n items into the chosen PQ
2. Repeatedly removeMin to emit items in sorted order

- Using a binary heap yields O(n log n) total time

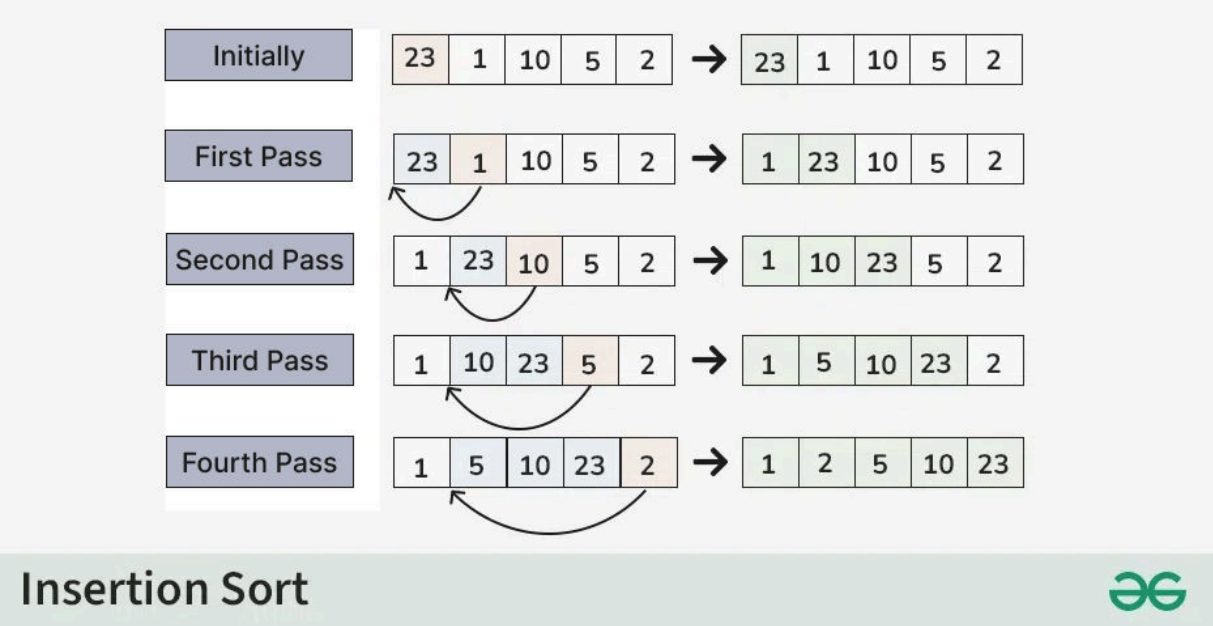
Selection Sort

- **Idea**: Repeatedly select the minimum from the unsorted suffix and swap it into place.
- **Algorithm**:
  1. For each index `i` from 0 to n−2:
     - Find index `minIdx` of smallest element in A[i...n−1].
     - Swap A[i] and A[minIdx].
- **Time Complexity**:
  - Best/Average/Worst: $O(n^2)$
- **Space Complexity**: O(1) (in-place)
- **Stable**: No (by default)



Insertion Sort

- **Idea**: Build a sorted portion at the front by inserting each new element into its correct place.
- **Algorithm**:
  1. For each index `i` from 1 to n−1:
     - Save `key = A[i]`.
     - Compare with elements to its left, shifting larger items right.
     - Insert `key` into its proper position.
- **Time Complexity**:
  - Best: O(n) (already sorted)
  - Average/Worst: $O(n^2)$
- **Space Complexity**: O(1) (in-place)
- **Stable**: Yes



Heap Data Structure

- Stored in an array of length n
- Parent at index *i* has children at 2·i+1 and 2·i+2
- Maintains the **heap property**: each node's key ≤ its children's keys

Bottom-Up Heap Construction

Instead of n successive inserts (O(n log n)), you can build a heap in O(n) by:

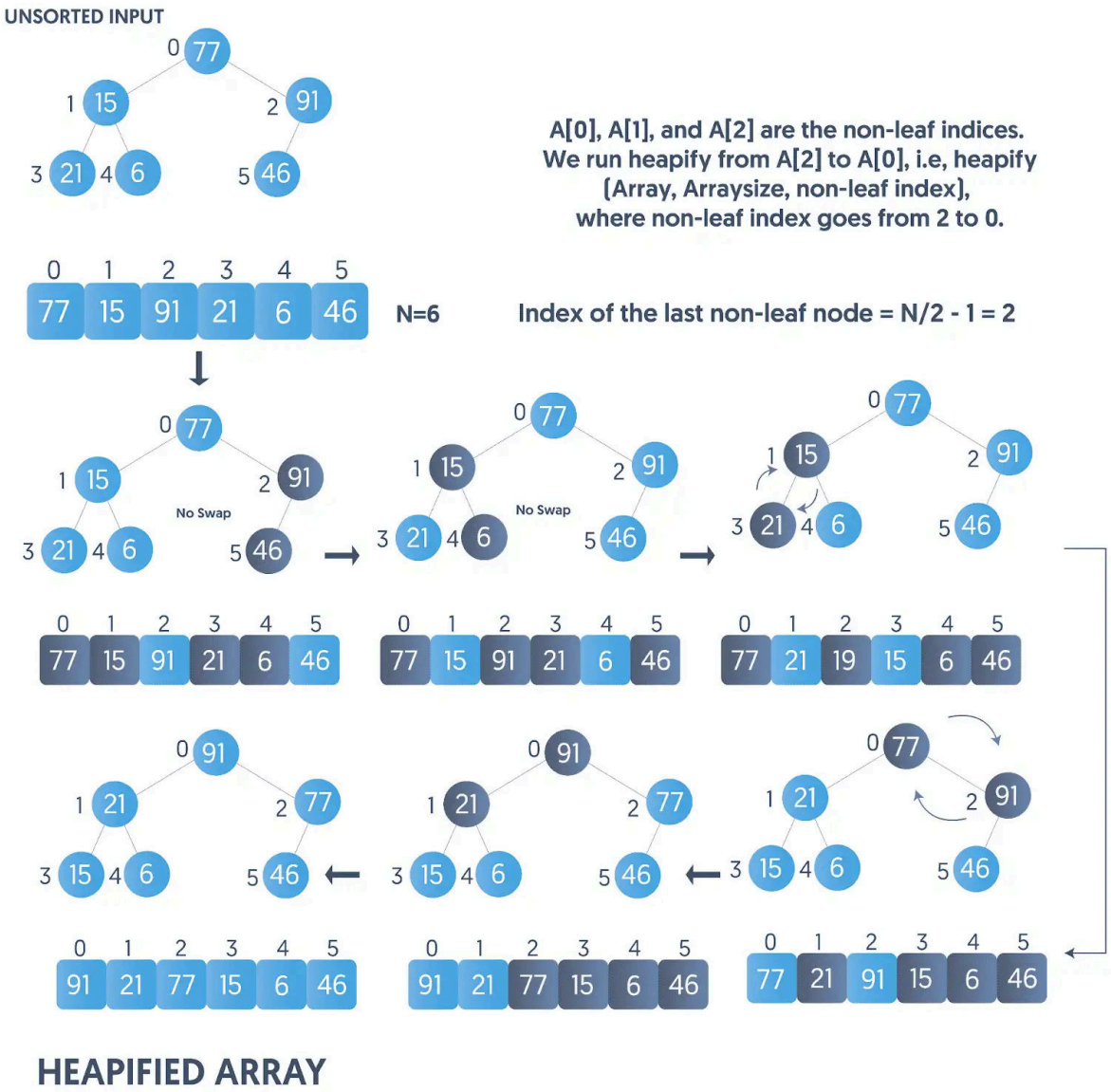
- Treat the array as a complete tree
- Perform down-heap ("heapify") on each non-leaf node from ⌊n/2⌋–1 down to 0

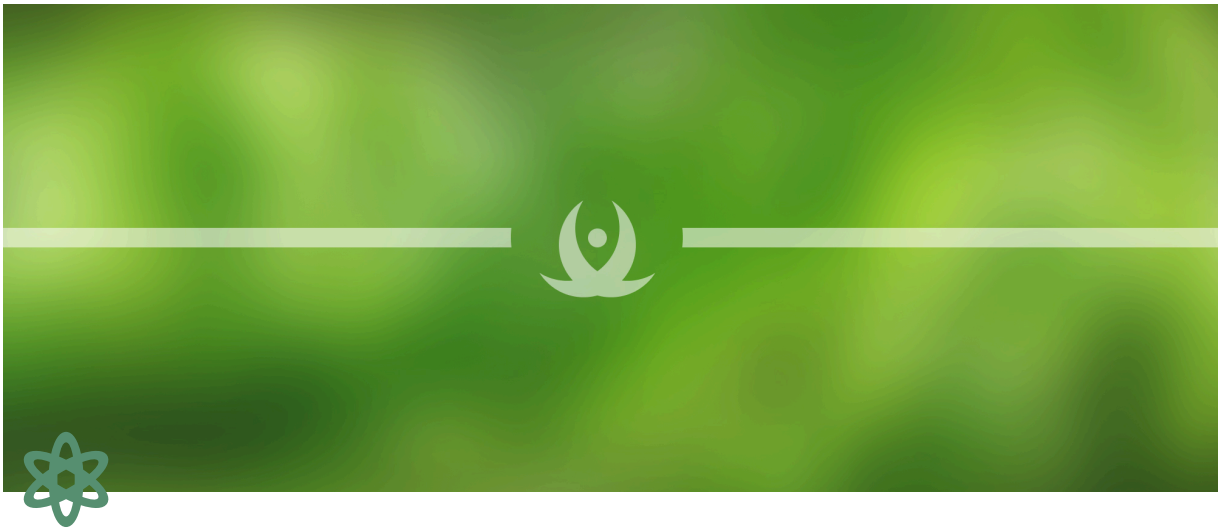
This linear-time build phase speeds up the first half of heap-sort without changing its overall O(n log n) cost.

Heap-Sort Overview

- **Build** a heap in O(n) (using bottom-up)
- **RemoveMin** (or swap root to end and shrink) n times in O(log n) each
- Total running time: O(n log n)

- In-place: uses the input array and O(1) extra space





23. Heap-Based Sorting & Merge-Sort Overview

Heap-Sort

A heap-sort uses a binary heap as a priority queue to sort n elements in place.

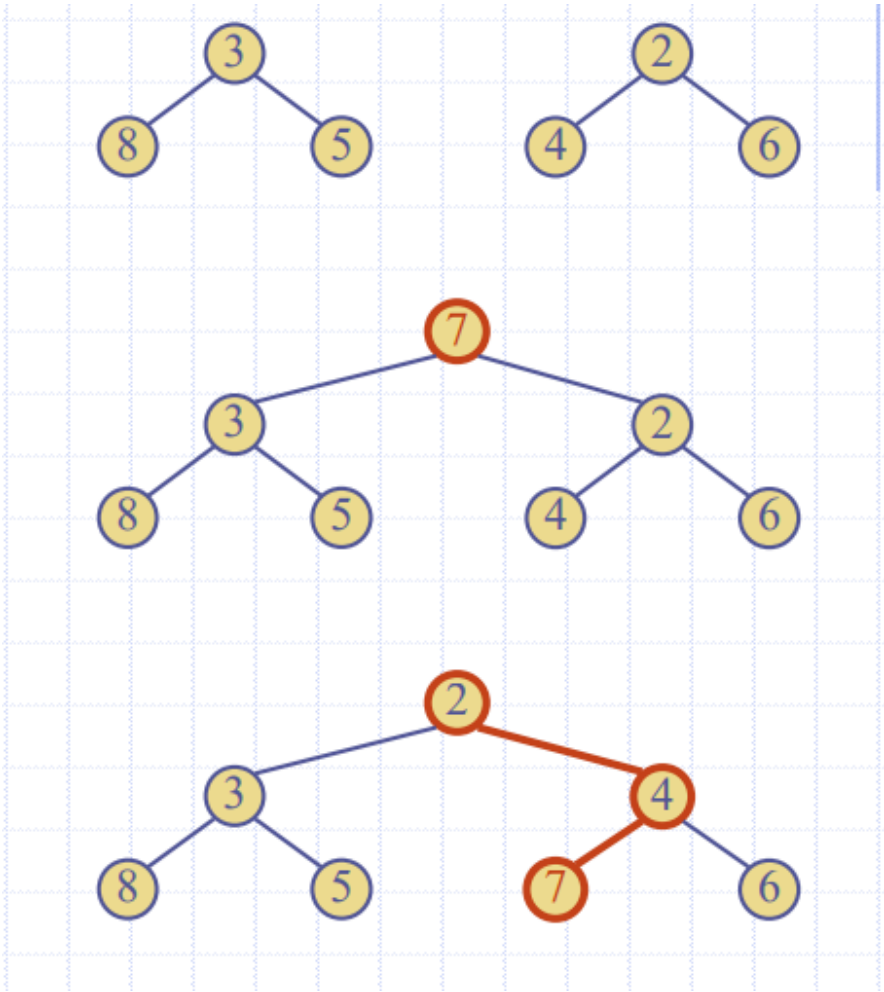
- **Space:** O(n)
- **Operations on heap of size n:**
  - insert, removeMin → O(log n)
  - min, size, isEmpty → O(1)
- **Overall sort time:**
  1. Build heap (bottom-up) → O(n)
  2. Repeatedly removeMin n times → n·O(log n) = O(n log n)

Heap-sort outperforms quadratic sorts (insertion, selection) for large n.

Merging Two Heaps

To merge two heaps H_1 and H_2 with a new key k:

- Create a new root node storing k.
- Hang H_1 and H_2 as its left and right subtrees.
- Perform a down-heap from the root to restore the heap-order property in O(log n) time ($n = |H_1|+|H_2|+1$).

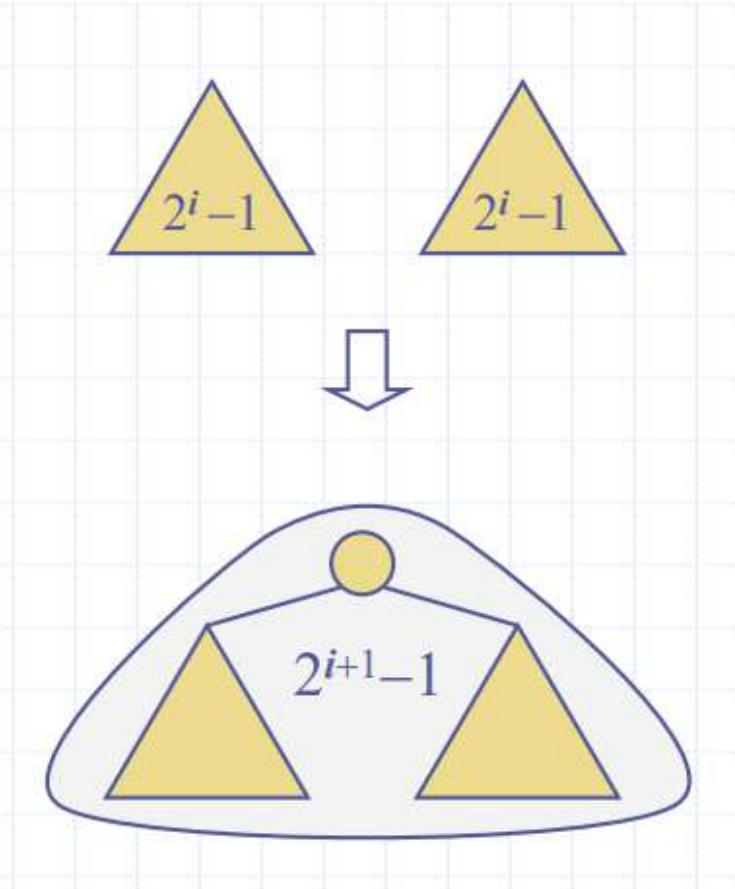


Bottom-Up Heap Construction

Instead of n successive inserts (O(n log n)), you can build a heap in linear time:

- View the input array as a complete binary tree.
- For each non-leaf node from $\lfloor n/2 \rfloor - 1$ down to 0, perform a down-heap ("heapify").
- **Total cost:** O(n)

- Speeds up heap-sort's build phase without changing overall O(n log n) runtime.



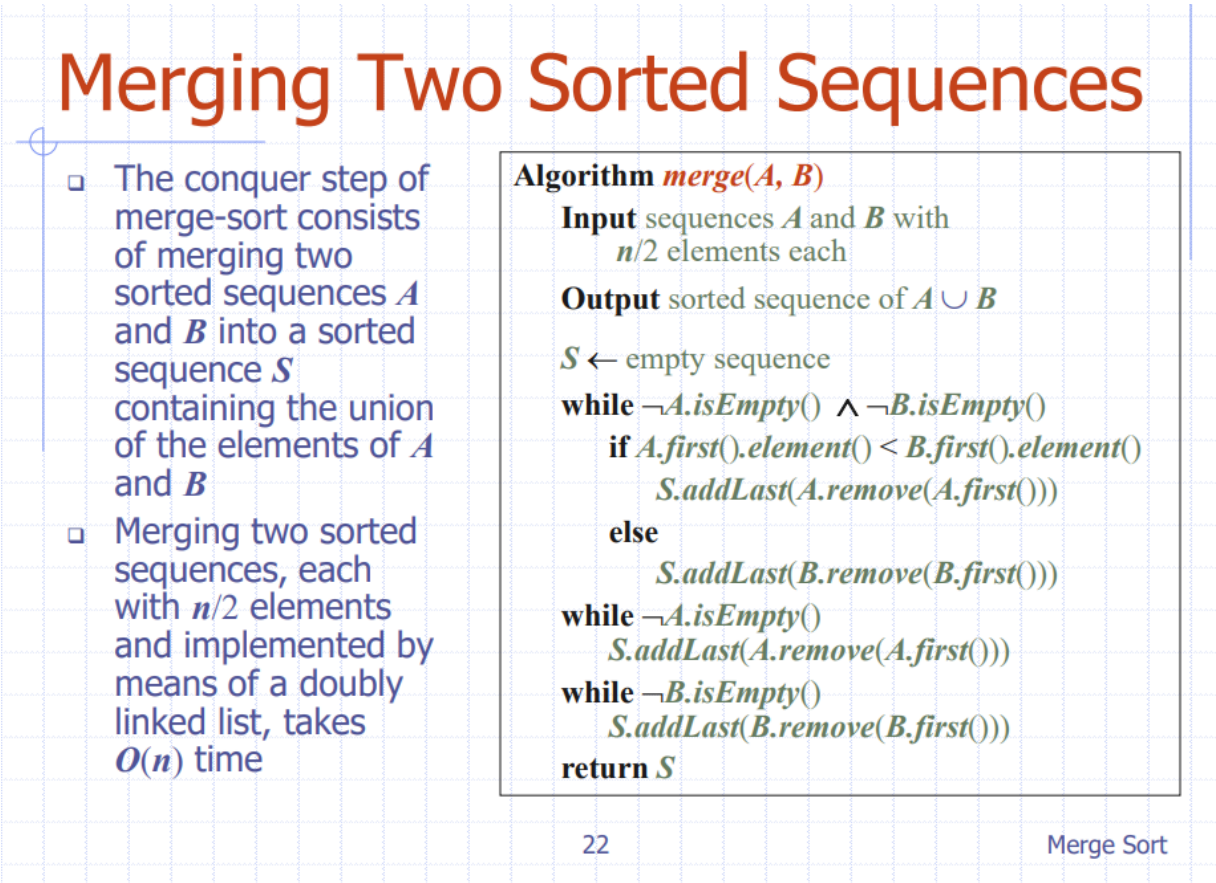
Merge-Sort

A classic divide-and-conquer sorter for n elements:

1. **Divide:** split the sequence into two halves of ~n/2 each.
2. **Recur:** sort each half recursively.
3. **Conquer:** merge the two sorted halves into one.

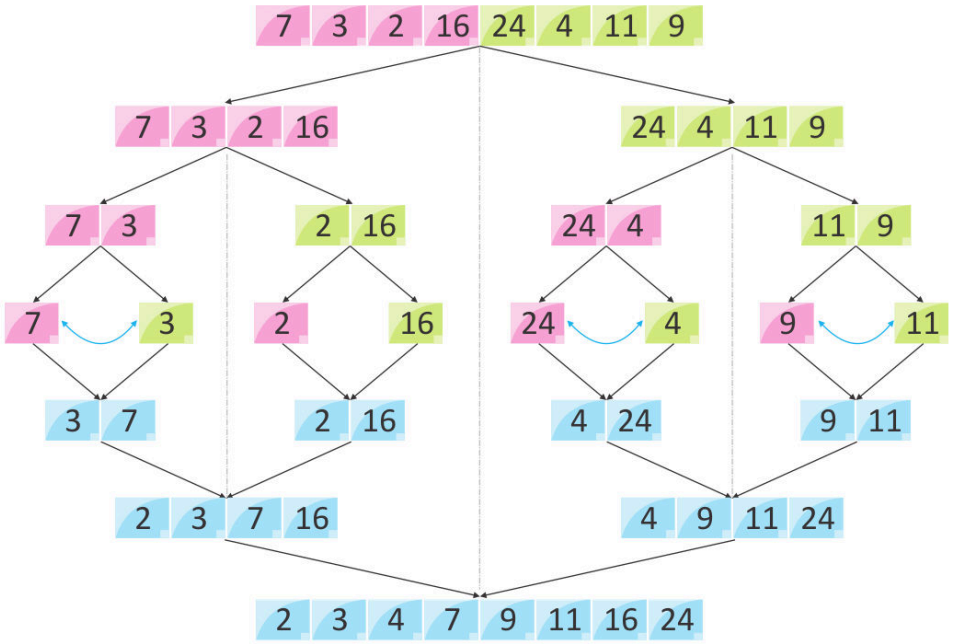
- Like heap-sort it has O(n log n) running time.
- **Merge step:** O(n) time and O(n) extra space.
- **Recurrence:** $T(n) = 2\,T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$.
- **Characteristics:**
    - Stable sort (if implemented carefully).

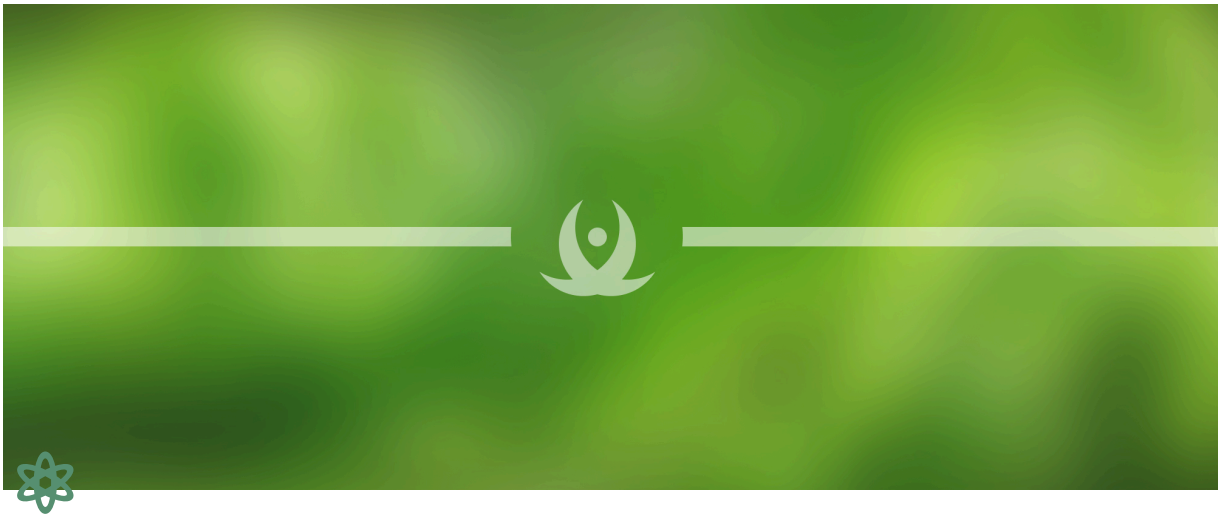
- Accesses data sequentially (good for external/disk-based sorting).
- Requires O(n) auxiliary space for merging.



Merge Procedure

- Maintain two pointers at the heads of the sorted sublists A and B.
- Repeatedly compare and remove the smaller element, appending it to the output.
- After one list is exhausted, append the remainder of the other.
- **Time:** O(n) for total of |A|+|B| = n elements.


7 3 2 16 24 4 11 9
7 3 2 16
24 4 11 9
7 3
2 16
24 4
11 9
7
3
2
16
24
4
9
11
3 7
2 16
4 24
9 11
2 3 7 16
4 9 11 24
2 3 4 7 9 11 16 24



24. Sorting Algorithms

Selection Sort

Description

> **SELECTION SORT:** Repeatedly selects the smallest (or largest) element from the unsorted portion of the array and swaps it into its correct position at the front (or end).

Pseudocode

```
Algorithm selectionSort(A, n)
    Input: Array A[0..n-1]
    Output: A sorted in nondecreasing order

    for i from 0 to n-2 do
        minIndex ← i
        for j from i+1 to n-1 do
            if A[j] < A[minIndex] then
                minIndex ← j
        // swap A[i] and A[minIndex]
        temp ← A[i]
        A[i] ← A[minIndex]
        A[minIndex] ← temp
```

Time Complexity

- **Best-case:** $O(n^2)$
- **Average-case:** $O(n^2)$
- **Worst-case:** $O(n^2)$

Space Complexity

- **Auxiliary Space:** O(1) (in-place)

Advantages

- Simple to understand and implement.
- Performs well for very small arrays (n < 1000).
- Number of swaps is at most n (one per outer iteration).

Disadvantages

- Quadratic time makes it inefficient for large n.
- Always performs $O(n^2)$ comparisons, even if the array is already sorted.

Insertion Sort

Description

> **INSERTION SORT:** Builds a sorted prefix one element at a time by taking an element from the unsorted portion and inserting it into the correct position within the sorted prefix.

Pseudocode

```
Algorithm insertionSort(A, n)
    Input: Array A[0..n-1]
    Output: A sorted in nondecreasing order

    for i from 1 to n-1 do
        key ← A[i]
        j ← i - 1
        // Shift elements of A[0..i-1] that are greater than key
        while j ≥ 0 and A[j] > key do
            A[j+1] ← A[j]
            j ← j - 1
        A[j+1] ← key
```

Time Complexity

- **Best-case:** O(n) (when array is already sorted; only one comparison per element)
- **Average-case:** $O(n^2)$
- **Worst-case:** $O(n^2)$

Space Complexity

- **Auxiliary Space:** O(1) (in-place)

Advantages

- Efficient for small arrays or nearly sorted data.
- Adaptive: runs in linear time if the input is nearly sorted.
- Stable sort (does not change the relative order of equal keys).
- Online algorithm: can sort as elements arrive.

Disadvantages

- Quadratic time for large or reverse-sorted inputs.
- Less efficient than more advanced algorithms (Merge, Quick, Heap) on large n.

Heap Sort

Description

> **HEAP SORT:** Builds a max-heap (binary heap) from the input array, then repeatedly extracts the maximum element and places it at the end of the array, shrinking the heap until empty.

Pseudocode

```
Algorithm heapSort(A, n)
    Input: Array A[0..n-1]
    Output: A sorted in nondecreasing order

    // Build max-heap in-place
    for i from ⌊n/2⌋ - 1 down to 0 do
        heapify(A, n, i)

    // Repeatedly extract the maximum and restore heap
    for i from n - 1 down to 1 do
        // Move current root (max) to end
        swap A[0], A[i]
        // Reduce heap size by one and heapify root
        heapify(A, i, 0)

    Procedure heapify(A, heapSize, i)
        largest ← i
        left ← 2 * i + 1
        right ← 2 * i + 2
        if left < heapSize and A[left] > A[largest] then
            largest ← left
        if right < heapSize and A[right] > A[largest] then
            largest ← right
        if largest ≠ i then
            swap A[i], A[largest]
            heapify(A, heapSize, largest)
```

Time Complexity

- **Best-case:** O(n log n)
- **Average-case:** O(n log n)
- **Worst-case:** O(n log n)

Space Complexity

- **Auxiliary Space:** O(1) (in-place)

Advantages

- In-place sort (requires no extra array).
- Guarantees O(n log n) time regardless of input order.
- Not sensitive to input distribution—consistent performance.

Disadvantages

- Not stable (may reorder equal elements).
- Access pattern is less sequential than merge sort; potentially poorer cache performance.
- Constant factors in `heapify` may be larger than quick sort's partition.

Merge Sort

Description

> **MERGE SORT:** Divide-and-conquer algorithm that recursively splits the array into halves, sorts each half, and merges the two sorted halves into a single sorted array.

Pseudocode

```
Algorithm mergeSort(A, left, right)
    Input: Array A[left..right]
    Output: A[left..right] sorted

    if left < right then
        mid ← ⌊(left + right) / 2⌋
        mergeSort(A, left, mid)
          mergeSort(A, mid+1, right)
        merge(A, left, mid, right)

    Procedure merge(A, left, mid, right)
        n1 ← mid - left + 1
        n2 ← right - mid
        create arrays L[0..n1] and R[0..n2]
        for i from 0 to n1-1 do
            L[i] ← A[left + i]
        for j from 0 to n2-1 do
            R[j] ← A[mid + 1 + j]
        L[n1] ← ∞   // sentinel
        R[n2] ← ∞   // sentinel
        i ← 0, j ← 0
        for k from left to right do
            if L[i] ≤ R[j] then
                A[k] ← L[i]
                i ← i + 1
            else
                A[k] ← R[j]
                j ← j + 1
```

Time Complexity

- **Best-case:** O(n log n)
- **Average-case:** O(n log n)
- **Worst-case:** O(n log n)

Space Complexity

- **Auxiliary Space:** O(n) (requires temporary arrays for merging)

Advantages

- Stable sort (maintains order of equal elements).
- Guarantees O(n log n) time for all inputs.
- Excellent for sorting linked lists (no extra space needed for merging).
- Access pattern is sequential—good for external sorting and cache performance.

Disadvantages

- Uses O(n) extra space, which can be expensive for large arrays.
- Recursive calls incur overhead; constant factors larger than some in-place sorts.

Quick Sort

Description

> **QUICK SORT:** Divide-and-conquer algorithm that picks a pivot element, partitions the array into three groups (elements less than, equal to, and greater than the pivot), recursively sorts the "less" and "greater" subarrays, and concatenates results. Often implemented in-place using two indices scanning from ends.

Pseudocode (In-Place Randomized Quick Sort)

```
Algorithm quickSort(A, low, high)
    Input: Array A[low..high]
    Output: A[low..high] sorted

    if low < high then
        // Randomly choose pivot index
        pivotIndex ← RANDOM(low, high)
        pivot ← A[pivotIndex]
        // Partition A around pivot: returns two boundaries h and k
        (h, k) ← partitionInPlace(A, low, high, pivot)
        quickSort(A, low, h - 1)
        quickSort(A, k + 1, high)

    Procedure partitionInPlace(A, low, high, pivot)
        i ← low
        j ← high
        while i ≤ j do
            while i ≤ j and A[i] < pivot do
                i ← i + 1
            while i ≤ j and A[j] > pivot do
                j ← j - 1
            if i ≤ j then
                swap A[i], A[j]
                i ← i + 1
                j ← j - 1
        // Now, A[low..j] ≤ pivot, A[i..high] ≥ pivot
        return (i, j)
```

Time Complexity

- **Best-case:** O(n log n) (balanced partitions every time)
- **Average-case:** O(n log n)

- **Worst-case:** $O(n^2)$ (if pivot is always the smallest or largest element)

Space Complexity

- **Auxiliary Space:** O(log n) on average (stack space for recursion); O(n) in worst case if recursion is unbalanced.

Advantages

- In-place sort (no extra array needed for partition).
- Typically faster in practice than other O(n log n) algorithms due to low overhead and good cache utilization.
- Average-case time is optimal for comparison sorts.

Disadvantages

- Unstable (equal keys may change order).
- Worst-case time $O(n^2)$ if pivot selection is poor (e.g., already sorted data and pivot = first element).
- Requires careful pivot selection or randomization (or "median-of-three" heuristic) to avoid worst-case scenarios.

Comparative Differences

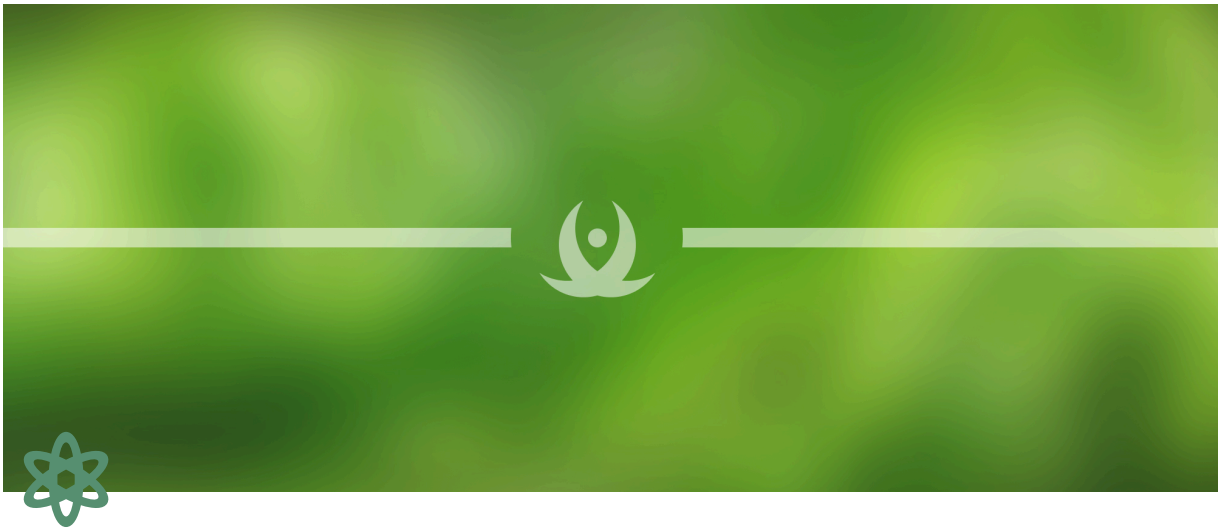
Algorithm	Best Case	Avg. Case	Worst Case	Space	Stable	In-Place	Notes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	No	Yes	Simple; few swaps; inefficie for large n.
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Yes	Yes	Adaptive; gooc for nearly sorte data.
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No	Yes	In-place; consistent performance; not stable.
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	No	Guaranteed n log n; sequenti access; extra memory.
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)*	No	Yes	In-place; faste on average; avoid worst-ca by randomization

Final Summary & Takeaways

- **Quadratic vs. n log n:**
  - *Selection* and *Insertion* sorts run in $O(n^2)$ and are only suitable for very small or nearly sorted arrays.
  - *Merge*, *Heap*, and *Quick* sorts run in O(n log n) on average or guaranteed, making them appropriate for large inputs.
- **Stability:**
  - *Insertion* and *Merge* are stable; useful when the relative order of equal elements must be preserved (e.g., sorting records by multiple keys).
  - *Selection*, *Heap*, and *Quick* are not stable without modifications.
- **Space Usage:**
  - *Selection*, *Insertion*, and *Heap* are in-place (O(1) extra space).

- *Quick* is in-place but may use O(log n) stack overhead.
- *Merge* requires O(n) additional memory for merging.

- **Practical Considerations:**
  - For large datasets, *Quick Sort* (with random pivot) often outperforms *Heap Sort* and *Merge Sort* due to cache friendliness.
  - *Merge Sort* excels on linked lists or when stability is required, and for external sorting (data on disk).
  - *Heap Sort* is a good choice when O(1) extra space and guaranteed O(n log n) time are needed.
- **Pivot and Partitioning:**
  - Proper pivot selection (random or median-of-three) is essential in *Quick Sort* to avoid worst-case $O(n^2)$.
  - Three-way partitioning (elements <, =, > pivot) reduces overhead when many duplicates exist.
- **Adaptive vs. Non-Adaptive:**
  - *Insertion Sort* is adaptive: linear on nearly sorted arrays.
  - Others (*Selection, Merge, Heap, Quick*) are non-adaptive: performance does not improve significantly on partially sorted data.



25. Sorting Lower Bounds and the Selection Problem

1. Sorting Lower Bound

1.1. Comparison-Based Sorting Model

> **COMPARISON-BASED SORTING:** Any sorting algorithm that determines order by comparing pairs of elements. Common examples: bubble sort, selection sort, insertion sort, merge sort, heap sort, quick sort.

- Each comparison can be viewed as a binary decision ("Is $x_i < x_j$?").
- The sorting process corresponds to traversing a **decision tree**, where each internal node is a comparison and each leaf represents a possible permutation of the input.

1.2. Decision Tree and Height

- **Decision Tree Height (h):** The worst-case number of comparisons needed is at least the height of this binary decision tree.
- For an input of size n, there are n! possible orderings (permutations), so the tree must have at least n! leaves.

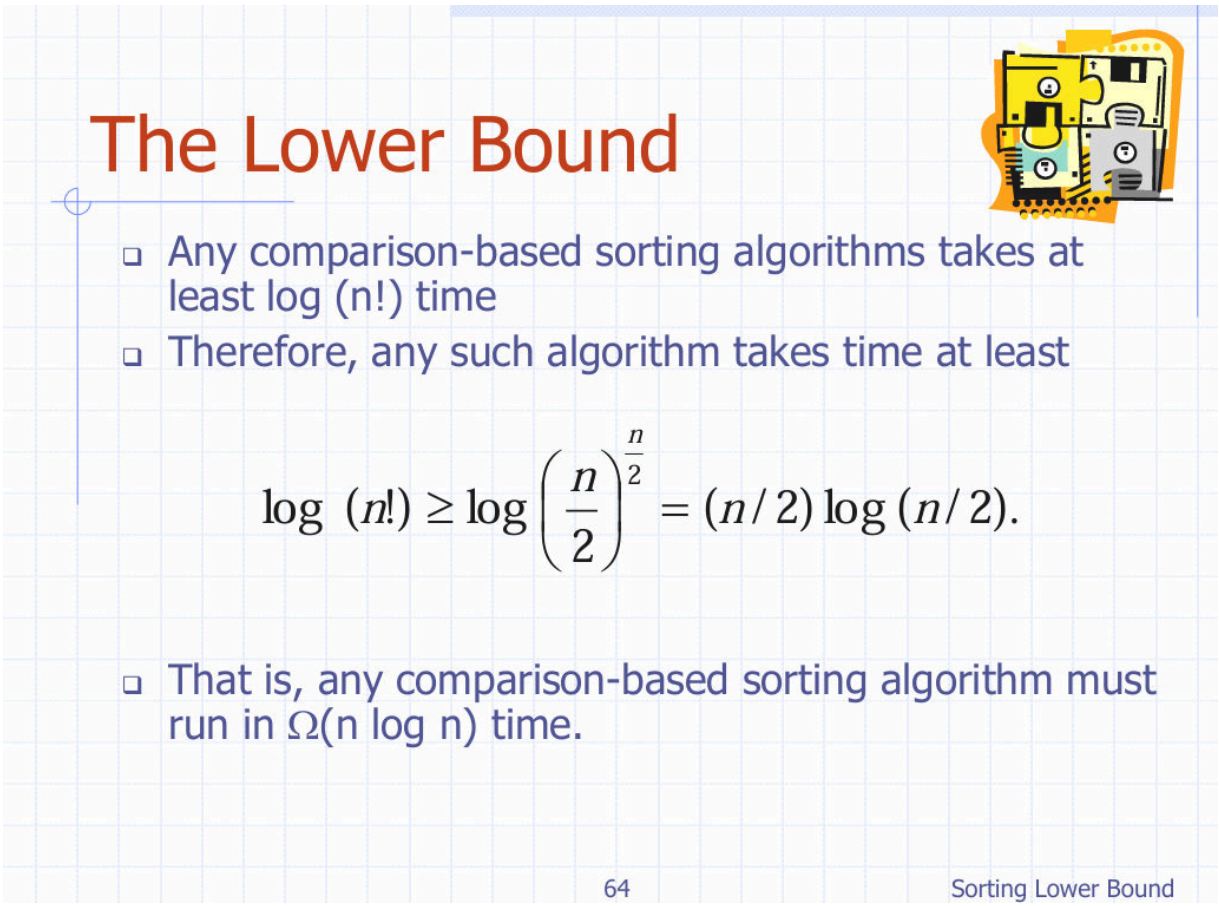
- A binary tree with L leaves has height at least $\log_2 L$. Therefore:

$$h \geq \log_2(n!) = \sum_{i=1}^{n} \log_2 i = n \log_2 n - n \log_2 e + O(\log n).$$

- **Lower Bound Theorem:** Any comparison-based sorting algorithm must perform at least Ω(n log n) comparisons in the worst case.

1.3. Implications

- No comparison-based sorting algorithm can have a worst-case time better than Ω(n log n).
- Algorithms like merge sort, heap sort, and (randomized) quick sort achieve this bound in average or worst case, matching the lower bound up to constant factors.



2. The Selection Problem

2.1. Problem Definition

> **k-TH SMALLEST (SELECTION) PROBLEM:** Given a set S of n elements drawn from a total order and an integer k ($1 \le k \le n$), find the element whose rank is k (the k-th smallest element) without fully sorting the set.

- **Naive Approach:** Sort the n elements (O(n log n)), then pick the k-th position.
- **Goal:** Achieve O(n) expected time (randomized) or O(n) worst-case time (deterministic) without full sorting.

2.2. Quick-Select (Randomized Selection)

2.2.1. Idea & Overview

- **Quick-Select:** A randomized "prune-and-search" algorithm similar to quick sort's partition step.
- **Steps:**
  1. Choose a random pivot x from S.
  2. Partition S into three subsets:
     - L = { elements < x }
     - E = { elements = x }
     - G = { elements > x }
  3. Let $\ell = |L|$, $e = |E|$.
     - If $k \le \ell$, recurse on L to find the k-th smallest.
     - Else if $k > \ell + e$, recurse on G to find the $(k - \ell - e)$-th smallest.
     - Otherwise ($\ell < k \le \ell + e$), x is the answer.

> **RUNTIME:** O(n) expected time, because each partition step takes O(n), and the expected size of the recursive subproblem is at most ¾ n when the pivot falls between the 25th and 75th percentiles (with probability ≥ ½).

2.2.2. Pseudocode

```
Algorithm quickSelect(S[1..n], k)
Input: Array S[1..n] of n elements, integer k (1 ≤ k ≤ n)
Output: The k-th smallest element of S

if n = 1 then
    return S[1]        // only one element

// Randomly choose pivot index p ∈ {1, …, n}
pivotIndex ← RANDOM(1, n)
pivot ← S[pivotIndex]

// Partition into L, E, G
L ← empty list
E ← empty list
G ← empty list
for i from 1 to n do
    if S[i] < pivot then
        append S[i] to L
    else if S[i] > pivot then
        append S[i] to G
    else
        append S[i] to E

ℓ ← |L|          // size of L
e ← |E|          // size of E

if k ≤ ℓ then
    return quickSelect(L, k)
else if k ≤ ℓ + e then
    return pivot    // pivot is the k-th smallest
else
    return quickSelect(G, k – ℓ – e)
```

2.2.3. Complexity Analysis

- **Partition Step:** O(n) time to build L, E, G.
- **Expected Subproblem Size:** With probability ≥ ½, pivot is in the middle 50% of the sorted order, so both |L| and |G| ≤ ¾ n.
- **Worst Case:** $O(n^2)$ if pivot is always the minimum or maximum (rare with randomization). Choosing pivot randomly or via "median-of-three" reduces worst-case likelihood.

2.2.4. Advantages & Disadvantages

- **Advantages:**
  - Expected linear time, simple to implement.
  - In-place version can be written that uses only O(1) additional memory (by swapping elements around the pivot).
- **Disadvantages:**
  - Worst case $O(n^2)$ if pivot choices are poor.

3. Deterministic Linear-Time Selection (Median of Medians)

3.1. Overview

> **DETERMINISTIC SELECT (Median-of-Medians):** Guarantees O(n) worst-case time by choosing a "good" pivot deterministically.

- **Main Idea:**
  1. Divide S into ⌈n/5⌉ groups of 5 elements each (the last group may have fewer).
  2. For each group, find its median by sorting the 5-element group in O(1) time (constant work).
  3. Gather all group medians into an array M (size ≈ n/5).
  4. Recursively compute the median of M; call this pivot "median-of-medians."
  5. Partition S around this pivot into L, E, G as in Quick-Select.

6. Recur on the appropriate subset (L or G) depending on k.

- By choosing the median-of-medians, we ensure that at least 3n/10 elements are "good" (≥ pivot or ≤ pivot), guaranteeing that the larger recursive subproblem has size ≤ 7n/10.

3.2. Pseudocode

```
Algorithm deterministicSelect(S[1..n], k)
Input: Array S[1..n] of n elements, integer k (1 ≤ k ≤ n)
Output: The k-th smallest element of S

if n ≤ 5 then
    sort S in O(1) time
    return S[k]

# 1. Partition S into groups of 5 and find medians
M ← empty list
for i from 1 to n step 5 do
    group ← S[i .. min(i+4, n)]
    sort group        // O(1) since group size ≤ 5
    median ← group[⌊|group|/2⌋ + 1]
    append median to M

# 2. Find pivot by selecting median of M recursively
m ← length(M)
pivot ← deterministicSelect(M, ⌈m/2⌉)

# 3. Partition S around pivot
L ← empty list
E ← empty list
G ← empty list
for i from 1 to n do
    if S[i] < pivot then
        append S[i] to L
    else if S[i] > pivot then
```

```
        append S[i] to G
    else
        append S[i] to E

ℓ ← |L|
e ← |E|

if k ≤ ℓ then
    return deterministicSelect(L, k)
else if k ≤ ℓ + e then
    return pivot
else
    return deterministicSelect(G, k - ℓ - e)
```

3.3. Complexity Analysis

- **Grouping & Median Computation:**
  - O(n) time to form ⌈n/5⌉ groups.
  - Sorting each 5-element group in O(1) time per group → O(n) total.
  - Recursively selecting the median of ⌈n/5⌉ medians takes T(n/5).
- **Partitioning Around Pivot:** O(n).
- **Solution:** This recurrence solves to T(n) = O(n).

3.4. Advantages & Disadvantages

- **Advantages:**
  - Guaranteed worst-case linear time O(n), no randomization needed.
  - Useful in real-time systems or when deterministic guarantees are critical.
- **Disadvantages:**
  - Higher constant factors compared to randomized Quick-Select.
  - More complicated to implement (grouping, median-finding, recursive calls).

4. Comparative Summary

Algorithm	Expected Time	Worst-Case Time	Space Overhead	Notes
Comparison-Based Sorting LB	—	$\Omega(n \log n)$	—	Provably no comparison sort can beat n log n time.
Quick-Select (Randomized)	O(n)	$O(n^2)$	O(1) or O(log n)	Simple; average-case O(n); worst-case rare.
Deterministic Select	O(n)	O(n)	O(n)	Worst-case O(n); larger constants; no randomization.

5. Final Takeaways

- **Sorting Lower Bound ($\Omega(n \log n)$):** Any comparison-based sort must use at least $\sim n \log_2 n$ comparisons in the worst case.
- **Selection without Sorting:** The k-th smallest element can be found in expected linear time via Quick-Select or in deterministic linear time via the median-of-medians algorithm.
- **Trade-Offs:**
  - **Quick-Select** has lower constants in practice but only expected O(n).
  - **Deterministic Select** guarantees O(n) worst-case but with higher overhead.
- **Applications:** Selection algorithms are used in order statistics, finding medians, percentile computations, and quickly partitioning data.