LE/CSSD 2101 - Object-Oriented Programming

Lab 5 - Introduction to Data Structures and Algorithms

Prerequisites: Advanced OOP Knowledge

Submission: Source Code + Markdown Report + 20 Minutes Video Reflection

Learning Objectives:

- Implement doubly linked list data structure with proper pointer manipulation
- Design and implement binary search tree with insertion, deletion, and search operations
- Master various tree traversal algorithms (in-order, pre-order, post-order)
- Implement breadth-first search (BFS) and depth-first search (DFS) algorithms
- Apply recursive and iterative approaches for tree operations
- Analyze time and space complexity of different data structures and algorithms
- Implement generic data structures using Java generics
- Handle edge cases and error conditions in data structure operations
- Design clean APIs and interfaces for data structure implementations
- Evaluate software quality through detailed testing and documentation

Department of Electrical Engineering and Computer Science

York University - Fall 2025

Contents

1 Background and Pre-lab Readings

1.1 Big-O: $O(f(n))$

Big-O notation describes the **upper bound** of an algorithm's growth rate. Formally, we say $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,

$$T(n) \leq cf(n).$$

In simpler terms, it means the running time (or memory usage) of the algorithm will never grow faster than some constant multiple of $f(n)$ when n becomes sufficiently large. This is why Big-O is often used to express the *worst-case upper limit* of an algorithm's performance.

For example, if an algorithm takes at most $5n^2 + 3n + 2$ steps, then it is $O(n^2)$, since the quadratic term dominates growth for large n. Big-O does not assert that the runtime is exactly n^2, only that it won't exceed it asymptotically. This abstraction allows computer scientists to focus on scalability rather than machine-specific constants or lower-order terms.

1.2 Little-o: $o(f(n))$

Little-o notation is a **strict upper bound**, meaning that $T(n)$ grows slower than $f(n)$ in the limit, but never matches its growth rate. Formally, $T(n)$ is $o(f(n))$ if for all constants $c > 0$, there exists some n_0 such that for all $n \geq n_0$,

$$T(n) < cf(n).$$

In other words, the ratio $T(n)/f(n) \to 0$ as $n \to \infty$. Unlike Big-O, Little-o guarantees that $f(n)$ is an overestimate that $T(n)$ will never "catch up" to.

As an example, consider $T(n) = n$ and $f(n) = n^2$. Here, n is $o(n^2)$, because the linear function grows much slower than the quadratic one, and as $n \to \infty$, the ratio $n/n^2 = 1/n \to 0$. This makes Little-o notation useful when we want to emphasize that the growth of a function is not only bounded but strictly smaller compared to a given function.

1.3 Big-Omega: $\Omega(f(n))$

Big-Omega notation describes the **lower bound** of an algorithm's growth rate. Formally, $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,

$$T(n) \geq cf(n).$$

This ensures that the algorithm takes at least as much time as $f(n)$, asymptotically. In other words, Big-Omega tells us the *minimum amount of resources* required by the algorithm for sufficiently large inputs.

For example, if a sorting algorithm always requires at least $n \log n$ comparisons, then we say its complexity is $\Omega(n \log n)$. Even if in practice the algorithm may take more time due to other factors, it cannot do better than this lower bound asymptotically. Big-Omega is particularly useful when analyzing best-case complexity or when proving lower theoretical limits for problem-solving.

1.4 Big-Theta: $\Theta(f(n))$

Big-Theta notation describes a **tight bound**, meaning that $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$. Formally, $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$,

$$c_1 f(n) \leq T(n) \leq c_2 f(n).$$

In this case, $f(n)$ accurately characterizes the growth rate of $T(n)$ both above and below. Thus, Big-Theta provides the most precise asymptotic classification of an algorithm.

For example, if an algorithm's running time is $3n^2 + 2n + 5$, then it is $\Theta(n^2)$, because the quadratic term bounds it from both sides for large n. This means the runtime neither grows faster nor slower than a constant multiple of n^2. Big-Theta is typically the preferred notation when we know the algorithm's runtime exactly, up to constant factors.

1.5 Little-Omega: $\omega(f(n))$

Little-Omega is the counterpart to Little-o, but for lower bounds. It provides a **strict lower bound** on growth. Formally, $T(n)$ is $\omega(f(n))$ if for every constant $c > 0$, there exists an n_0 such that for all $n \geq n_0$,

$$T(n) > cf(n).$$

This means $T(n)$ grows strictly faster than $f(n)$ as $n \to \infty$. In other words, the ratio $T(n)/f(n) \to \infty$ as $n \to \infty$.

For example, $T(n) = n^2$ is $\omega(n)$, because $n^2/n = n \to \infty$. This notation is useful when you want to emphasize that an algorithm's growth eventually outpaces a comparison function, regardless of constant scaling.

1.6 Soft-O: $\tilde{O}(f(n))$

Soft-O notation, written $\tilde{O}(f(n))$, is used to hide *polylogarithmic factors*. Formally, $T(n)$ is $\tilde{O}(f(n))$ if there exists a polynomial in $\log n$ such that $T(n) \in O(f(n) \cdot \text{polylog}(n))$. This makes it convenient to focus on the main growth term while ignoring logarithmic factors.

For example, an algorithm running in $O(n \log^2 n)$ is often written as $\tilde{O}(n)$. This shorthand is common in advanced algorithm design and theoretical computer science, especially when log factors are not the central concern.

1.7 Average-Case and Amortized Analysis

Asymptotic notation is not limited to worst-case scenarios. We also analyze **average-case complexity**, where $T(n)$ represents the expected cost of running the algorithm under some input distribution. Additionally, **amortized analysis** studies the average cost of a sequence of operations, even if a single operation may be expensive. The amortized bound is often expressed in Big-O notation as well.

For example, in dynamic array resizing, a single insertion may cost $O(n)$ when reallocation occurs, but averaged over many insertions, the amortized cost is $\Theta(1)$. Amortized complexity is therefore a refinement of asymptotic notation that captures behavior over sequences rather than isolated calls.

1.8 Algorithm Analysis

Mathematical analysis of algorithms may look intimidating at first, but in practice it is simply a disciplined way of explaining your reasoning. What you are really doing is mapping program behavior to a mathematical function that describes how the running time grows with input size. If you approach it step by step, the process becomes structured, repeatable, and much less daunting. Think of analysis as telling a story: you begin with the code, translate its operations into counts, and then generalize those counts into a function that captures the algorithm's complexity.

1.8.1 Step 0: Provide the Code

We begin with a simple factorial function in Python. This will serve as the subject of our analysis.

```python
def factorial(n):
rc = 1                          # (1 operation)
for i in range(2, n + 1):       # (n-1) iterations
    rc = rc * i             # (2 operations per iteration)
return rc                               # (1 operation)
```

1.8.2 Step 1: Define Variables

We now declare the mathematical entities for our analysis:

- Let n represent the input size (the number we compute $n!$ for).
- Let $T(n)$ represent the number of primitive operations required by the code.

1.8.3 Step 2: Count the Operations

$$\begin{aligned} T(n) &= 1 && \text{(initial assignment)} \\ &+ (n-1) && \text{(loop iterations)} \\ &+ 2(n-1) && \text{(multiplications inside loop)} \\ &+ 1 && \text{(return statement)} \end{aligned}$$

1.8.4 Step 3: Form the Expression

Adding these together:

$$T(n) = 1 + (n-1) + 2(n-1) + 1.$$

1.8.5 Step 4: Simplify the Equation

Simplifying:

$$T(n) = 3n + 1.$$

1.8.6 Step 5: State the Final Result

The dominating term is $3n$, and by definition of Big-O notation, constant factors are ignored. Therefore:

$$T(n) \in O(n).$$

By following the five steps—code reference, variable definition, operation counting, equation formation, and simplification—we conclude that the factorial function runs in linear time, $O(n)$. This simple process illustrates how algorithm analysis transforms raw code into a precise mathematical description of complexity.

1.9 Arrays, Lists, and Array Lists: A Comparison

1.9.1 Arrays: The Basics

An **array** is a data structure that stores a fixed-size sequence of elements in contiguous memory. Each element can be accessed directly by index in constant time, because the memory address of an element is calculated by simple arithmetic:

$$\text{address}(A[i]) = \text{base}(A) + i \times \text{size\_of\_element}.$$

This property makes arrays extremely efficient for *random access* operations.

1.9.2 Strengths of Arrays

The strengths of arrays lie in their **simplicity and speed**. Indexing is $O(1)$, meaning accessing the k^{th} element is instantaneous. They are cache-friendly because of contiguous memory allocation, making traversal operations very fast in practice. They also have no extra memory overhead beyond the elements themselves.

1.9.3 Limitations of Arrays

However, arrays also have significant limitations. Their size must be fixed at creation; resizing requires allocating new memory and copying all elements, which is expensive. Insertions or deletions in the middle require shifting many elements, which can take $O(n)$ time. Therefore, arrays are not well-suited for dynamic data where size or structure changes frequently.

1.9.4 Lists: The Basics

By contrast, a **linked list** is a sequence of nodes where each node contains data and a pointer to the next node (and possibly the previous one in a doubly linked list). Unlike arrays, linked lists do not require contiguous memory. They can grow and shrink dynamically without costly reallocations, making them flexible.

1.9.5 Strengths of Lists

The main advantage of lists is that **insertion and deletion** can be done in constant time $O(1)$ if you already have a pointer to the node. This makes them attractive for scenarios where frequent structural modifications are needed, such as implementing queues, stacks, or adjacency lists in graphs. They also avoid wasted memory from over-allocation.

1.9.6 Limitations of Lists

The trade-off is that linked lists have **poor cache locality** and higher memory overhead (extra pointers/references per node). Accessing the i^{th} element requires traversing the list from the head, which takes $O(i)$ time. Therefore, random access is slow compared to arrays. Traversals are also more pointer-heavy, increasing cache misses.

1.9.7 Array vs. List: Summary of Trade-offs

In summary, arrays excel at **fast indexing and iteration**, while lists excel at **flexible insertion and deletion**. Arrays are memory-efficient but rigid in size, while lists are flexible but have more memory overhead. The choice between them depends on whether the priority is fast lookups or dynamic updates.

1.9.8 ArrayList (Dynamic Array)

An **array list** combines features of arrays and lists. It is implemented using an array internally but automatically resizes itself when full, usually by doubling capacity. This gives the user the convenience of a list's flexible size without giving up the fast random access of an array. In most programming languages, the default "list" type (like Python's `list` or Java's `ArrayList`) is actually an array list.

1.9.9 ArrayList Pros and Cons

The key advantage of array lists is that **appending is amortized** $O(1)$. Most insertions at the end are constant time, with occasional expensive resizes spread over many operations. Like arrays, random access is $O(1)$, and iteration is cache-friendly. However, insertions and deletions in the middle are still $O(n)$, because shifting elements is required. Memory reallocation during growth can also cause temporary performance spikes.

1.9.10 Comparing List, ArrayList, and Arrays

Compared to arrays, array lists add flexibility with minimal overhead, making them more practical for general-purpose programming. Compared to linked lists, array lists retain efficient indexing and cache performance while still supporting dynamic resizing. The main trade-off is that array lists are not ideal for frequent insertions in the middle, where linked lists still perform better. In practice, array lists often provide the best balance and are the default choice in modern languages.

1.9.11 Array List and Amortized Constant Time

An **array list** (also called a dynamic array, e.g., Python's `list` or Java's `ArrayList`) is a data structure that stores elements in a contiguous block of memory, like an array, but can grow as needed. When the array becomes full, the implementation allocates a new, larger block of memory (commonly twice the size), copies the existing elements, and then continues inserting. This allows the array list to support arbitrary growth while preserving fast element access by index.

At first glance, resizing looks expensive because copying n elements takes $O(n)$ time. However, these resizes occur only occasionally. Most insertions simply place a new element at the next available position, which is $O(1)$. To analyze the true performance we use **amortized analysis**: instead of measuring the worst-case of one operation, we average the cost over many operations.

Insertion #	**Capacity After Resize**	**Copy Cost**	**Total Cost**
1	1	0	1
2	2	1	2
3	4	2	3
4	4	0	1
5	8	4	5
6	8	0	1
7	8	0	1
8	8	0	1
9	16	8	9

Notice the pattern: most insertions cost only 1, but occasionally we pay more due to copying. If we insert n elements, the total cost of all resizes and insertions is at most $2n$. Therefore, the *average cost per insertion* is

$$\frac{\text{Total Cost over } n \text{ insertions}}{n} \leq O(1).$$

This is why we say that appending to an array list is **amortized** $O(1)$: occasional expensive operations are offset by many cheap ones, so the overall average cost remains constant.

1.10 Binary Search

The analysis of binary search differs from that of linear search because the loop in binary search does not scan elements sequentially from start to end. Instead, it begins in the middle of the array and repeatedly discards half of the remaining search space. Thus, not every element is examined during the search process, which changes how we approach its runtime analysis.

Important

A binary search can only be performed if:

- The list is sorted.
- Access to any element by index takes constant time ($O(1)$).

1.11 Code Reference

We consider the following Python implementation:

```python
def binary_search(my_list, key):
rc = -1
low = 0
high = len(my_list) - 1

while rc == -1 and low <= high:
    mid = (low + high) // 2
    if key < my_list[mid]:
        high = mid - 1
    elif key > my_list[mid]:
        low = mid + 1
    else:
        rc = mid

return rc
```

Here, `low` and `high` represent the first and last possible indices where the key may exist. At each iteration, the midpoint is checked, and half of the search space is discarded. The process repeats until the key is found or no elements remain.

1.12 Step 1: Define Variables

- Let n represent the size of the array.
- Let $T(n)$ represent the number of operations needed to find (or fail to find) the key.

1.13 Step 2: Count Constant Operations

Before the loop:

$$rc = -1 \quad (1)$$
$$low = 0 \quad (1)$$
$$high = \text{len}(\text{my\_list}) - 1 \quad (3)$$
$$\text{return } rc \quad (1)$$

Total outside the loop $= 1 + 1 + 3 + 1 = 6$.

1.14 Step 3: Count Loop Operations

Each loop iteration performs:

$$rc == -1 \ \& \ low \leq high \quad (3)$$
$$mid = (low + high)//2 \quad (3)$$

The conditional checks add:

$$\text{if } key < my\_list[mid] \quad (1), \qquad \text{elif } key > my\_list[mid] \quad (1), \qquad low = mid + 1 \quad (2).$$

Thus, in the worst case (when the search fails and we always execute both checks and update `low`):

$$\text{per iteration cost} = 10.$$

1.15 Step 4: Number of Iterations

At the start, n elements are possible. After one iteration, approximately $n/2$ remain; after two, $n/4$ remain, and so forth. After k iterations, about $n/2^k$ elements remain.

The loop stops when only one element remains:

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \log_2 n.$$

Thus, the loop executes approximately $1 + \log_2 n$ times.

1.16 Step 5: Build the Expression

The total operation count is:

$$T(n) = 10(1 + \log_2 n) + 6.$$

Simplifying:

$$T(n) = 10 \log_2 n + 16.$$

1.17 Step 6: Final Result

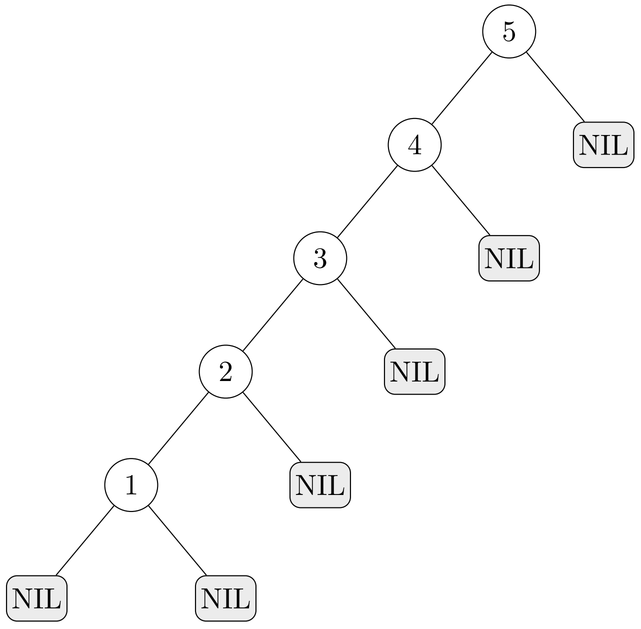
The dominating term is $10 \log_2 n$. Constants are ignored in asymptotic analysis, so:

$$T(n) \in O(\log n).$$

Binary search eliminates half the search space on each iteration, leading to logarithmic runtime. Compared to linear search ($O(n)$), this represents an exponential improvement in efficiency for large input sizes, provided the array is sorted and random access is constant time.

1.18 Naïve Binary Search Tree Degenerating into a List

Inserting numbers `5, 4, 3, 2, 1` into a naive binary search tree results in a completely unbalanced structure. Each new element is smaller than the previous one, so it is always inserted to the left. The tree degenerates into a linked list, with height n instead of $\log n$.



This plain BST (descending inserts) degenerates to a list of height n, hence $O(n)$ search.

2 Lab Overview

Lab 5 focuses on implementing fundamental data structures: **Doubly Linked Lists** and **Binary Search Trees**. This lab builds upon the object-oriented programming concepts from previous labs and introduces students to data structure design, algorithm implementation, and complexity analysis.

Why These Data Structures Matter:

- **Doubly Linked Lists:** Used in web browsers (back/forward navigation), music players (playlist management), and undo/redo functionality in text editors
- **Binary Search Trees:** Foundation for databases (indexing), file systems (directory structures), and search engines (ranking algorithms)

Real-World Applications:

- **Linked Lists:** Browser history, playlist management, undo operations
- **Binary Trees:** Database indexing, expression evaluation, decision trees in AI
- **Traversals:** File system navigation, XML parsing, compiler syntax analysis

2.1 Key Concepts Covered in Lab5

- **Data Structures:** Doubly linked lists, binary search trees
- **Algorithms:** Tree traversals, search algorithms, sorting

- **Complexity Analysis:** Time and space complexity evaluation
- **Generic Programming:** Type-safe data structure implementation
- **Testing:** Detailed unit testing for data structures

3 Part 1: Doubly Linked List Implementation (25 points)

3.1 Background on Doubly Linked Lists

A doubly linked list is a linear data structure where each node contains data and two pointers/references: one to the next node and one to the previous node. This bidirectional linking allows for efficient traversal in both directions and easier insertion/deletion operations.

Visual Representation:

`null` $\leftarrow$ `[A]` $\leftrightarrow$ `[B]` $\leftrightarrow$ `[C]` $\leftrightarrow$ `[D]` $\rightarrow$ `null`

Key Advantages:

- **Bidirectional Traversal:** Can move forward and backward through the list
- **Efficient Deletion:** Can delete a node in O(1) time if you have a reference to it
- **Better for Certain Operations:** Ideal for undo/redo functionality, browser history

Common Use Cases:

- **Web Browsers:** Back/forward button navigation
- **Music Players:** Previous/next song functionality
- **Text Editors:** Undo/redo operations
- **Caching Systems:** LRU (Least Recently Used) cache implementation

3.2 API Skeleton for Doubly Linked List

```java
/**
 * A generic Doubly Linked List interface for Lab 5.
 * Students must implement all methods.
 */
public interface DoublyLinkedList<T> {
    // Basic operations
    void addFirst(T data);          // Insert at the head
    void addLast(T data);           // Insert at the tail
    void insertAt(int index, T data); // Insert at a given index

    // Deletion
    T removeFirst();                // Remove from head
    T removeLast();                 // Remove from tail
    T removeAt(int index);          // Remove from a given index

    // Access
    T getFirst();                   // Get head element
    T getLast();                    // Get tail element
    T getAt(int index);             // Get element at index

    // Utility
    int size();                     // Number of elements
    boolean isEmpty();              // Check if empty
    void clear();                   // Remove all elements

    // Search
    boolean contains(T data);       // Check existence
    int indexOf(T data);            // First index of element
}
```

3.3 Node Class Skeleton

```java
/**
 * Node class for Doubly Linked List.
 */
class Node<T> {
    T data;
    Node<T> prev;
    Node<T> next;

    Node(T data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}
```

3.4 Implementation Requirements

Students must implement:

- Proper pointer/reference manipulation for all operations
- Edge case handling (empty list, single element, out-of-bounds)
- Generic type safety
- Proper error handling and validation
- Unit tests for all methods

Implementation Tips:

- **Always Update Both Pointers/References:** When inserting/deleting, update both `prev` and `next` pointers/references
- **Handle Edge Cases:** Empty list, single element, first/last element operations

- **Index Validation:** Check bounds before accessing elements by index

Common Pitfalls to Avoid:

- **Forgetting to Update Pointers/References:** Always update both `prev` and `next`
- **Null Pointer Exceptions:** Check for null before dereferencing
- **Circular References:** Ensure proper cleanup to avoid memory leaks
- **Index Out of Bounds:** Validate indices before access

4 Part 2: Binary Search Tree Implementation (30 points)

4.1 Understanding Recursion in Data Structures

Recursion is a fundamental concept in computer science where a function calls itself to solve smaller instances of the same problem. In the context of binary search trees, recursion provides an elegant and natural way to traverse and manipulate tree structures. Recursion typically consists of two main components:

- **Base Case:** The condition that stops the recursion (prevents infinite loops)
- **Recursive Case:** The part where the function calls itself with a smaller problem

Key Characteristics of Recursion Functions:

- **Self-Reference:** Function calls itself
- **Problem Reduction:** Each call works on a smaller version of the problem
- **Stack-Based:** Each recursive call creates a new stack frame
- **Natural for Trees:** Perfect fit for hierarchical data structures

4.1.1 Tail Recursion vs Regular Recursion

Regular Recursion:

- **Definition:** Function performs additional work after the recursive call
- **Stack Usage:** Each call creates a new stack frame
- **Memory:** Linear space complexity O(n) for n recursive calls
- **Example:** Factorial calculation, tree traversals

Tail Recursion:

- **Definition:** The recursive call is the last operation in the function

- **Optimization:** Can be optimized by compilers to use constant space
- **Memory:** Can be optimized to O(1) space complexity
- **Example:** Accumulator-based functions, iterative-style recursion

Code Examples:

Regular Recursion (Non-Tail):

```java
public class RegularRecursionExample {

    // Regular recursion - NOT tail recursive
    // Additional work after recursive call (multiplication)
    public static int factorial(int n) {
        if (n <= 1) {
            return 1;  // Base case
        }
        // Recursive call is NOT the last operation
        // Multiplication happens after the recursive call returns
        return n * factorial(n - 1);  // Additional work after recursion
    }

    // Another example of regular recursion
    public static int sum(int n) {
        if (n <= 0) {
            return 0;  // Base case
        }
        // Addition happens after recursive call returns
        return n + sum(n - 1);  // Additional work after recursion
    }

    // Tree traversal - regular recursion
    public static void printTree(TreeNode node) {
        if (node == null) {
            return;  // Base case
        }
        // Multiple recursive calls with work in between
        printTree(node.left);     // First recursive call
        System.out.println(node.data);  // Work between calls
        printTree(node.right);    // Second recursive call
    }
}
```

Tail Recursion (Optimizable):

```java
public class TailRecursionExample {

    // Tail recursive version - recursive call is the LAST operation
    public static int factorialTail(int n, int accumulator) {
        if (n <= 1) {
            return accumulator;  // Base case - return accumulated result
        }
        // Recursive call is the LAST operation (tail position)
        return factorialTail(n - 1, n * accumulator);
    }

    // Wrapper function for clean interface
    public static int factorial(int n) {
        return factorialTail(n, 1);
    }

    // Another tail recursive example
    public static int sumTail(int n, int accumulator) {
        if (n <= 0) {
            return accumulator;  // Base case - return accumulated result
        }
        // Recursive call is the LAST operation
        return sumTail(n - 1, n + accumulator);
    }

    // Wrapper for sum
    public static int sum(int n) {
        return sumTail(n, 0);
    }
}
```

Key Differences:

Regular Recursion Stack Trace:

```
// factorial(4) execution:
// factorial(4) calls factorial(3)
// factorial(3) calls factorial(2)
// factorial(2) calls factorial(1)
// factorial(1) returns 1
// factorial(2) returns 2 * 1 = 2      ← Work after recursion
// factorial(3) returns 3 * 2 = 6      ← Work after recursion
// factorial(4) returns 4 * 6 = 24    ← Work after recursion

// Stack frames: [factorial(4), factorial(3), factorial(2), factorial(1)]
// Memory: O(n) - each call needs its own stack frame
```

Tail Recursion Stack Trace:

```
// factorialTail(4, 1) execution:
// factorialTail(4, 1) calls factorialTail(3, 4)
// factorialTail(3, 4) calls factorialTail(2, 12)
// factorialTail(2, 12) calls factorialTail(1, 24)
// factorialTail(1, 24) returns 24

// Optimized by compiler to:
// accumulator = 1
// accumulator = 4 * 1 = 4
// accumulator = 3 * 4 = 12
// accumulator = 2 * 12 = 24
// return 24

// Memory: O(1) - only one stack frame needed (optimized)
```

4.1.2 Stack Overflow and Memory Issues

Common Problems with Deep Recursion:

- **Stack Overflow:** When recursion depth exceeds available stack space
- **Memory Overhead:** Each recursive call consumes stack memory
- **Performance Impact:** Function call overhead for each recursive invocation
- **Debugging Difficulty:** Deep call stacks can be hard to trace

Stack Frame Overhead:

- **Local Variables:** Each frame stores local variables
- **Return Address:** Memory to store where to return after completion
- **Parameters:** Function arguments passed to each call
- **Control Information:** Additional metadata for function execution

4.1.3 Benefits and Drawbacks of Recursion

Benefits:

- **Elegant Code:** Often more readable and intuitive
- **Natural for Trees:** Perfect fit for hierarchical structures
- **Mathematical Clarity:** Mirrors mathematical definitions
- **Problem Decomposition:** Breaks complex problems into simpler ones

Drawbacks:

- **Memory Usage:** Each call consumes stack space
- **Performance:** Function call overhead
- **Stack Overflow Risk:** Deep recursion can cause crashes
- **Debugging Complexity:** Hard to trace deep call stacks

4.1.4 Caching and Recursion

Recursive Caching Strategies:

- **Memoization:** Store results of expensive recursive calls
- **Dynamic Programming:** Build solutions from smaller cached results
- **Tabulation:** Pre-compute and store all possible results
- **Space-Time Tradeoff:** Use memory to reduce computation time

Benefits of Caching:

- **Performance:** Avoid redundant calculations
- **Efficiency:** Transform exponential algorithms to polynomial
- **Scalability:** Handle larger problem sizes
- **Optimization:** Reduce both time and space complexity

4.2 Classic Recursive Examples

4.2.1 Factorial Function

Regular Recursion (Inefficient):

```java
public class FactorialExample {

    // Regular recursion - O(n) time, O(n) space
    public static long factorial(int n) {
        // Base case: stop recursion
        if (n <= 1) {
            return 1;
        }
        // Recursive case: n! = n * (n-1)!
        return n * factorial(n - 1);
    }

    // Example usage and stack trace
    public static void demonstrateFactorial() {
        System.out.println("Computing 5! using regular recursion:");
        long result = factorial(5);
        System.out.println("5! = " + result);

        // Stack trace for factorial(5):
        // factorial(5) calls factorial(4)
        // factorial(4) calls factorial(3)
        // factorial(3) calls factorial(2)
        // factorial(2) calls factorial(1)
        // factorial(1) returns 1
        // factorial(2) returns 2 * 1 = 2
        // factorial(3) returns 3 * 2 = 6
        // factorial(4) returns 4 * 6 = 24
        // factorial(5) returns 5 * 24 = 120
    }
}
```

Tail Recursion (Optimizable):

```java
public class TailRecursiveFactorial {

    // Tail recursive version - can be optimized by compiler
    public static long factorialTail(int n, long accumulator) {
        // Base case: return accumulated result
        if (n <= 1) {
            return accumulator;
        }
        // Tail recursive call: accumulator carries the result
        return factorialTail(n - 1, n * accumulator);
    }

    // Wrapper function for cleaner interface
    public static long factorial(int n) {
        return factorialTail(n, 1);
    }

    // Example usage
    public static void demonstrateTailRecursion() {
        System.out.println("Computing 5! using tail recursion:");
        long result = factorial(5);
        System.out.println("5! = " + result);

        // Tail recursion trace for factorial(5):
        // factorialTail(5, 1) calls factorialTail(4, 5)
        // factorialTail(4, 5) calls factorialTail(3, 20)
        // factorialTail(3, 20) calls factorialTail(2, 60)
        // factorialTail(2, 60) calls factorialTail(1, 120)
        // factorialTail(1, 120) returns 120
    }
}
```

4.2.2 Tower of Hanoi Problem

The Tower of Hanoi is a classic mathematical puzzle that perfectly demonstrates the power and elegance of recursive thinking. This ancient problem, also known as the Tower of Brahma, was invented by the French mathematician Édouard Lucas in 1883 and has become a fundamental example in computer science education.

Problem Description: The puzzle consists of three pegs (A, B, C) and a number of disks of different sizes that can slide onto any peg. The puzzle starts with all disks stacked in ascending order of size on one peg (the source), with the smallest disk at the top.

Visual Representation:

```
Initial State (3 disks):          Goal State:
    A     B     C                    A     B     C
    |     |     |                    |     |     |
   [1]    |     |                    |     |    [1]
  [2]     |     |                    |     |   [2]
 [3]      |     |                    |     |  [3]
----+-----+-----+----              ----+-----+-----+----
```

Game Rules:

- **Single Disk Movement:** Only one disk can be moved at a time
- **Size Constraint:** A larger disk cannot be placed on top of a smaller disk
- **No Skipping:** You cannot skip over a peg - disks must be moved to adjacent pegs

- **Complete Transfer:** All disks must be moved from the source peg to the destination peg
- **Optimal Solution:** The goal is to find the minimum number of moves required

Why Tower of Hanoi is Perfect for Recursion:

- **Self-Similar Structure:** The problem for n disks contains the same problem for n-1 disks
- **Natural Decomposition:** Each recursive call works on a smaller version of the same problem
- **Clear Base Case:** Moving a single disk is trivial (the base case)
- **Mathematical Beauty:** The solution reveals the power of divide-and-conquer thinking

Mathematical Properties:

- **Minimum Moves:** For n disks, the minimum number of moves is $2^n - 1$
- **Exponential Growth:** The number of moves grows exponentially with the number of disks
- **Time Complexity:** $O(2^n)$ - each disk must be moved multiple times
- **Space Complexity:** O(n) - recursion depth equals the number of disks

Real-World Applications:

- **Algorithm Design:** Demonstrates divide-and-conquer strategy
- **Recursion Teaching:** Perfect example for understanding recursive thinking
- **Mathematical Induction:** Proves properties about recursive algorithms
- **Complexity Analysis:** Shows exponential vs polynomial algorithm differences

Recursive Solution Strategy:

The beauty of the Tower of Hanoi solution lies in its recursive decomposition. The key insight is that to move n disks from source to destination, we can break it down into three steps:

The Three-Step Algorithm:

1. **Step 1:** Move the top n-1 disks from source to auxiliary peg
2. **Step 2:** Move the largest disk directly from source to destination
3. **Step 3:** Move the n-1 disks from auxiliary to destination

Why This Works:

- **Step 1** reduces the problem to moving n-1 disks (smaller subproblem)
- **Step 2** handles the base case of moving one disk

- **Step 3** again reduces to moving n-1 disks (another smaller subproblem)
- Each step is a recursive call with a smaller problem size

Complete Recursive Implementation:

```java
public class TowerOfHanoi {

    private static int moveCount = 0;  // Track total moves

    /**
     * Recursive solution for Tower of Hanoi
     * @param n Number of disks to move
     * @param source Source peg (where disks start)
     * @param destination Destination peg (where disks should end up)
     * @param auxiliary Auxiliary peg (helper peg for intermediate moves)
     */
    public static void hanoi(int n, char source, char destination, char auxiliary) {
        // Base case: only one disk to move
        if (n == 1) {
            moveCount++;
            System.out.println("Move " + moveCount + ": Move disk 1 from " +
                             source + " to " + destination);
            return;
        }

        // Recursive case: solve for n-1 disks
        // Step 1: Move n-1 disks from source to auxiliary
        // Note: destination becomes auxiliary, auxiliary becomes destination
        hanoi(n - 1, source, auxiliary, destination);

        // Step 2: Move the largest disk from source to destination
        moveCount++;
        System.out.println("Move " + moveCount + ": Move disk " + n +
                         " from " + source + " to " + destination);

        // Step 3: Move n-1 disks from auxiliary to destination
        // Note: auxiliary becomes source, source becomes auxiliary
        hanoi(n - 1, auxiliary, destination, source);
    }

    /**
     * Calculate minimum number of moves using recursive formula
     * Formula: T(n) = 2 * T(n-1) + 1, with T(1) = 1
     * Solution: T(n) = 2^n - 1
     */
    public static long minimumMoves(int n) {
        if (n == 1) {
            return 1;  // Base case: one disk requires one move
        }
        // Recursive formula: 2 * (moves for n-1 disks) + 1 (for largest disk)
        return 2 * minimumMoves(n - 1) + 1;
    }

    /**
     * Non-recursive calculation using mathematical formula
     * More efficient for large n
     */
    public static long minimumMovesFormula(int n) {
        return (long) Math.pow(2, n) - 1;
    }

    /**
     * Demonstrate Tower of Hanoi with step-by-step solution
     */
    public static void demonstrateHanoi(int disks) {
        System.out.println("=== Tower of Hanoi with " + disks + " disks ===");
        System.out.println("Minimum moves required: " + minimumMoves(disks));
        System.out.println("Formula verification: " + minimumMovesFormula(disks));
        System.out.println("\nStep-by-step solution:");
        System.out.println("Initial state: All disks on peg A");
        System.out.println("Goal: Move all disks to peg C using peg B as auxiliary");
        System.out.println();

        moveCount = 0;  // Reset move counter
        hanoi(disks, 'A', 'C', 'B');

        System.out.println("\nTotal moves: " + moveCount);
        System.out.println("Final state: All disks on peg C");
    }

    /**
     * Analyze the recursive call tree for educational purposes
     */
    public static void analyzeRecursion(int n) {
        System.out.println("=== Recursion Analysis for " + n + " disks ===");
        System.out.println("Recursive calls: " + (Math.pow(2, n) - 1));
        System.out.println("Maximum recursion depth: " + n);
        System.out.println("Time complexity: O(2^n)");
        System.out.println("Space complexity: O(n)");
    }
}
```

Step-by-Step Execution Example (3 disks):

```
// Call: hanoi(3, 'A', 'C', 'B')
//
// Move 1: Move disk 1 from A to C
// Move 2: Move disk 2 from A to B
// Move 3: Move disk 1 from C to B
// Move 4: Move disk 3 from A to C
// Move 5: Move disk 1 from B to A
// Move 6: Move disk 2 from B to C
// Move 7: Move disk 1 from A to C
//
// Total moves: 7 = 2^3 - 1
```

Recursive Call Tree Visualization:

```
hanoi(3, A, C, B)
|-- hanoi(2, A, B, C)
|   |-- hanoi(1, A, C, B) -> Move disk 1: A->C
|   |-- Move disk 2: A->B
|   +-- hanoi(1, C, B, A) -> Move disk 1: C->B
|-- Move disk 3: A->C
```

```
+-- hanoi(2, B, C, A)
    |-- hanoi(1, B, A, C) -> Move disk 1: B->A
    |-- Move disk 2: B->C
    +-- hanoi(1, A, C, B) -> Move disk 1: A->C
```

4.2.3 Fibonacci with Caching

Naive Recursion (Inefficient):

```java
public class FibonacciNaive {

    // Naive recursive Fibonacci - O(2^n) time complexity
    public static long fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    // This approach recalculates the same values multiple times
    // For fibonacci(5), it calculates fibonacci(3) multiple times
}
```

Recursive Fibonacci with Memoization:

```java
import java.util.HashMap;
import java.util.Map;

public class FibonacciMemoized {

    private static Map<Integer, Long> cache = new HashMap<>();

    // Memoized recursive Fibonacci - O(n) time complexity
    public static long fibonacci(int n) {
        // Check if result is already cached
        if (cache.containsKey(n)) {
            return cache.get(n);
        }

        // Base cases
        if (n <= 1) {
            cache.put(n, (long) n);
            return n;
        }

        // Recursive calculation with caching
        long result = fibonacci(n - 1) + fibonacci(n - 2);
        cache.put(n, result);
        return result;
    }

    // Clear cache for fresh calculations
    public static void clearCache() {
        cache.clear();
    }

    // Example usage
    public static void demonstrateMemoization() {
        System.out.println("Fibonacci sequence with memoization:");
        for (int i = 0; i <= 10; i++) {
            System.out.println("fibonacci(" + i + ") = " + fibonacci(i));
        }

        // Cache shows which values were computed:
        System.out.println("Cache contents: " + cache);
    }
}
```

4.2.4 Tree Traversal with Recursion

Recursive Tree Traversals:

```java
public class RecursiveTreeTraversals {

    // In-order traversal (Left-Root-Right)
    public static void inOrderTraversal(TreeNode node) {
        if (node != null) {
            inOrderTraversal(node.left);    // Visit left subtree
            System.out.print(node.data + " "); // Process current node
            inOrderTraversal(node.right);   // Visit right subtree
        }
    }

    // Pre-order traversal (Root-Left-Right)
    public static void preOrderTraversal(TreeNode node) {
        if (node != null) {
            System.out.print(node.data + " "); // Process current node first
            preOrderTraversal(node.left);     // Visit left subtree
            preOrderTraversal(node.right);    // Visit right subtree
        }
    }

    // Post-order traversal (Left-Right-Root)
    public static void postOrderTraversal(TreeNode node) {
        if (node != null) {
            postOrderTraversal(node.left);    // Visit left subtree
            postOrderTraversal(node.right);   // Visit right subtree
            System.out.print(node.data + " "); // Process current node last
        }
    }

    // Calculate tree height recursively
    public static int treeHeight(TreeNode node) {
        if (node == null) {
            return -1; // Height of empty tree is -1
        }

        // Height = 1 + max(height of left subtree, height of right subtree)
        int leftHeight = treeHeight(node.left);
        int rightHeight = treeHeight(node.right);

        return 1 + Math.max(leftHeight, rightHeight);
    }

    // Count nodes recursively
    public static int countNodes(TreeNode node) {
        if (node == null) {
            return 0;
        }

        // Total nodes = 1 (current) + nodes in left subtree + nodes in right subtree
        return 1 + countNodes(node.left) + countNodes(node.right);
    }
}
```

4.3 When to Use Recursion vs Iteration

Use Recursion When:

- **Tree/Graph Traversal:** Natural fit for hierarchical structures
- **Divide and Conquer:** Problems that can be split into smaller subproblems
- **Mathematical Definitions:** When the problem has a recursive mathematical definition
- **Backtracking:** Exploring all possible solutions

Use Iteration When:

- **Performance Critical:** When every millisecond counts
- **Deep Recursion:** When recursion depth might cause stack overflow
- **Memory Constraints:** When stack space is limited
- **Simple Loops:** When the iterative solution is straightforward

4.4 Background on Binary Search Trees

A Binary Search Tree (BST) is a hierarchical data structure where each node has at most two children, and for any node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater than the node's value.

BST Property (Invariant):

- **Left Subtree:** All values < current node's value
- **Right Subtree:** All values > current node's value
- **This property must hold for every node in the tree**

Visual Example:

```
    8
   / \
  3   10
 / \    \
1   6    14
   / \   /
  4   7 13
```

Key Advantages:

- **Fast Search:** O(log n) average case, O(n) worst case. Worst case happens when the tree becomes a linked list (all elements inserted in order), when items are inserted in sorted order!
- **Ordered Traversal:** In-order traversal gives sorted sequence
- **Dynamic Size:** Can grow and shrink as needed
- **Efficient Insertion/Deletion:** O(log n) average case

Real-World Applications:

- **Databases:** Indexing for fast lookups
- **File Systems:** Directory structures
- **Expression Trees:** Mathematical expression evaluation
- **Decision Trees:** AI and machine learning algorithms

4.5 API Skeleton for Binary Search Tree

```java
/**
 * A generic Binary Search Tree interface for Lab 5.
 * Students must implement all methods.
 */
public interface BinarySearchTree<T extends Comparable<T>> {
    // Core operations
    void insert(T data);             // Insert a new value
    boolean contains(T data);        // Search for a value
    void delete(T data);             // Delete a value

    // Traversals
    void inOrderTraversal();         // Left, Root, Right
    void preOrderTraversal();        // Root, Left, Right
    void postOrderTraversal();       // Left, Right, Root
    void levelOrderTraversal();      // BFS

    // Properties
    int height();                    // Height of tree
    int size();                      // Number of nodes
    boolean isEmpty();               // Check if empty
}
```

4.6 TreeNode Class Skeleton

```java
/**
 * Node class for Binary Search Tree.
 */
class TreeNode<T> {
    T data;
    TreeNode<T> left;
    TreeNode<T> right;

    TreeNode(T data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

4.7 Implementation Starter Class

```java
/**
 * Implementation of Binary Search Tree (students complete methods).
 */
public class MyBinarySearchTree<T extends Comparable<T>>
    implements BinarySearchTree<T> {

    private TreeNode<T> root;
    private int size;

    public MyBinarySearchTree() {
        root = null;
        size = 0;
    }

    @Override
    public void insert(T data) {
        // TODO: Implement recursive or iterative insert
    }

    @Override
    public boolean contains(T data) {
        // TODO: Implement search logic
        return false;
    }

    @Override
    public void delete(T data) {
        // TODO: Implement delete logic
    }

    @Override
    public void inOrderTraversal() {
        // TODO: Implement recursive in-order
    }

    @Override
    public void preOrderTraversal() {
        // TODO: Implement recursive pre-order
    }

    @Override
    public void postOrderTraversal() {
        // TODO: Implement recursive post-order
    }

    @Override
    public void levelOrderTraversal() {
        // TODO: Implement BFS using Queue
    }

    @Override
    public int height() {
        // TODO: Return tree height
        return -1;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }
}
```

5 Part 3: Tree Traversal Algorithms (20 points)

5.1 Understanding Tree Traversals

Tree traversals are systematic ways to visit all nodes in a tree exactly once. The choice of traversal depends on the specific problem requirements and the order in which nodes need to be processed.

Why Traversals Matter:

- **Data Processing:** Extract information from tree structures
- **Tree Operations:** Copy, serialize, or analyze tree contents
- **Problem Solving:** Many algorithms require systematic tree exploration

Traversal Categories:

- **Depth-First (DFS):** Explore one branch completely before moving to another
- **Breadth-First (BFS):** Explore all nodes at one level before moving to the next

Example Tree for Demonstrations:

```
    8
```

```
   / \
  3   10
 / \    \
1   6    14
   / \   /
  4   7 13
```

5.2 Depth-First Search (DFS) Traversals

DFS traversals explore as far as possible along each branch before backtracking. There are three main DFS traversal orders:

5.2.1 In-Order Traversal (Left-Root-Right)

Visits nodes in the order: Left → Root → Right

Algorithm:

1. Traverse the left subtree
2. Visit the root node
3. Traverse the right subtree

Characteristics:

- Time Complexity: O(n)
- Space Complexity: O(h) where h is the height of the tree
- Use Case: Produces sorted output for BST
- Recursive Implementation: Natural fit for recursive structure

Example Implementation:

```java
public void inOrderTraversal(TreeNode<T> node) {
    if (node != null) {
        inOrderTraversal(node.left);    // Visit left subtree
        System.out.print(node.data + " "); // Process root
        inOrderTraversal(node.right);   // Visit right subtree
    }
}
```

Step-by-Step Example: For the tree: `8 / 3 10 / 1 6 14 / 4 7 13`

```
        8
       / \
      3   10
```

```
 / \    \
1   6    14
   / \   /
  4   7 13
```

1. Start at root (8), go left to 3
2. From 3, go left to 1 (no children) → print 1
3. Back to 3 → print 3
4. Go right to 6, then left to 4 → print 4
5. Back to 6 → print 6
6. Go right to 7 → print 7
7. Back to 8 → print 8
8. Go right to 10, then right to 14, then left to 13 → print 13
9. Back to 14 → print 14

Result: 1 3 4 6 7 8 13 14 (sorted order!)

5.2.2 Pre-Order Traversal (Root-Left-Right)

Visits nodes in the order: Root → Left → Right

Algorithm:

1. Visit the root node
2. Traverse the left subtree
3. Traverse the right subtree

Characteristics:

- Time Complexity: O(n)
- Space Complexity: O(h)
- Use Case: Tree structure reconstruction, expression trees
- Advantage: Root is processed first, useful for copying trees

Example Implementation:

```java
public void preOrderTraversal(TreeNode<T> node) {
    if (node != null) {
        System.out.print(node.data + " "); // Process root first
        preOrderTraversal(node.left);     // Visit left subtree
        preOrderTraversal(node.right);    // Visit right subtree
    }
}
```

Step-by-Step Example: For the tree: `8 / 3 10 / 1 6 14 / 4 7 13`

```
        8
       / \
      3   10
     / \    \
    1   6    14
       / \   /
      4   7 13
```

1. Start at root 8 → print 8
2. Go left to 3 → print 3
3. Go left to 1 → print 1
4. Back to 3, go right to 6 → print 6
5. Go left to 4 → print 4
6. Back to 6, go right to 7 → print 7
7. Back to 8, go right to 10 → print 10
8. Go right to 14 → print 14
9. Go left to 13 → print 13

Result: 8 3 1 6 4 7 10 14 13 (root-first order)

Use Cases:

- **Tree Copying:** Reconstruct tree structure
- **Expression Trees:** Prefix notation (e.g., + * 3 4 5)
- **File System:** Directory listing with subdirectories

5.2.3 Post-Order Traversal (Left-Right-Root)

Visits nodes in the order: Left → Right → Root

Algorithm:

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root node

Characteristics:

- Time Complexity: O(n)
- Space Complexity: O(h)
- Use Case: Tree deletion, expression evaluation, directory size calculation
- Advantage: Children are processed before parent

Example Implementation:

```java
public void postOrderTraversal(TreeNode<T> node) {
    if (node != null) {
        postOrderTraversal(node.left);    // Visit left subtree
        postOrderTraversal(node.right);   // Visit right subtree
        System.out.print(node.data + " "); // Process root last
    }
}
```

Step-by-Step Example: For the tree: `8 / 3 10 / 1 6 14 / 4 7 13`

```
      8
     / \
    3   10
   / \    \
  1   6    14
     / \   /
    4   7 13
```

1. Start at root 8, go left to 3
2. From 3, go left to 1 (no children) → print 1
3. Back to 3, go right to 6
4. From 6, go left to 4 (no children) → print 4
5. Back to 6, go right to 7 (no children) → print 7

6. Back to 6 → print 6
7. Back to 3 → print 3
8. Back to 8, go right to 10
9. From 10, go right to 14
10. From 14, go left to 13 (no children) → print 13
11. Back to 14 → print 14
12. Back to 10 → print 10
13. Back to 8 → print 8

Result: 1 4 7 6 3 13 14 10 8 (children-first order)

Use Cases:

- **Tree Deletion:** Delete children before parent
- **Expression Trees:** Postfix notation (e.g., 3 4 * 5 +)
- **Directory Size:** Calculate subdirectory sizes before parent
- **Memory Cleanup:** Free child resources before parent

5.3 Breadth-First Search (BFS) - Level Order Traversal

BFS explores nodes level by level, using a queue to maintain the order of nodes to be visited.

Algorithm:

1. Start with root node in queue
2. While queue is not empty:
   (a) Dequeue a node and process it
   (b) Enqueue its left child (if exists)
   (c) Enqueue its right child (if exists)

Characteristics:

- Time Complexity: O(n)
- Space Complexity: O(w) where w is the maximum width of the tree
- Use Case: Level-by-level processing, finding shortest path
- Advantage: Processes nodes by depth levels

Example Implementation:

```java
public void levelOrderTraversal() {
    if (root == null) return;

    Queue<TreeNode<T>> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        TreeNode<T> current = queue.poll();
        System.out.print(current.data + " ");

        if (current.left != null) queue.offer(current.left);
        if (current.right != null) queue.offer(current.right);
    }
}
```

Step-by-Step Example: For the tree: `8 / 3 10 / 1 6 14 / 4 7 13`

```
      8
     / \
    3   10
   / \    \
  1   6    14
     / \   /
    4   7 13
```

1. Start: Queue = [8]
2. Process 8 → print 8, add children 3, 10 → Queue = [3, 10]
3. Process 3 → print 3, add children 1, 6 → Queue = [10, 1, 6]
4. Process 10 → print 10, add child 14 → Queue = [1, 6, 14]
5. Process 1 → print 1, no children → Queue = [6, 14]
6. Process 6 → print 6, add children 4, 7 → Queue = [14, 4, 7]
7. Process 14 → print 14, add child 13 → Queue = [4, 7, 13]
8. Process 4 → print 4, no children → Queue = [7, 13]
9. Process 7 → print 7, no children → Queue = [13]
10. Process 13 → print 13, no children → Queue = []

Result: 8 3 10 1 6 14 4 7 13 (level-by-level order)

Use Cases:

- **Shortest Path:** Find minimum path in unweighted graphs
- **Level-by-Level Processing:** Print tree by levels

- **Network Broadcasting:** Send messages to all nodes at same distance
- **Game Trees:** Evaluate all moves at current level before going deeper

5.4 Traversal Comparison and Use Cases

Traversal	**Order**	**Best Use Case**	**Space Complexity**
In-Order	Left → Root → Right	Sorted output (BST)	O(h)
Pre-Order	Root → Left → Right	Tree copying, structure	O(h)
Post-Order	Left → Right → Root	Tree deletion, evaluation	O(h)
Level-Order	Level by level	Shortest path, levels	O(w)

Table 1: Comparison of Tree Traversal Methods

5.5 Iterative vs Recursive Implementations

5.5.1 Recursive Approach

- **Advantages:** Simple, intuitive, natural for tree structure
- **Disadvantages:** Stack overflow risk for deep trees, function call overhead
- **Space:** O(h) due to call stack

5.5.2 Iterative Approach

- **Advantages:** No stack overflow, better control over memory
- **Disadvantages:** More complex implementation, requires explicit stack/queue
- **Space:** O(h) for DFS, O(w) for BFS

5.6 Practical Applications

- **In-Order:** Database queries, sorted data processing
- **Pre-Order:** File system traversal, expression tree evaluation
- **Post-Order:** Memory cleanup, dependency resolution
- **Level-Order:** Network routing, game tree analysis

6 Part 4: Complexity Analysis (15 points)

6.1 Time Complexity Analysis

- **Doubly Linked List:**

  - Insertion: O(1) at head/tail, O(n) at arbitrary position
  - Deletion: O(1) at head/tail, O(n) at arbitrary position
  - Search: O(n)

- **Binary Search Tree:**
  - Insertion: O(log n) average, O(n) worst case
  - Deletion: O(log n) average, O(n) worst case
  - Search: O(log n) average, O(n) worst case

Complexity Examples:

- **Linked List Search:** Finding an element in a list of 1000 items may require checking all 1000 elements
- **BST Search:** Finding an element in a balanced BST of 1000 items requires at most 10 comparisons ($\log_2(1000) \approx 10$)
- **Worst Case BST:** If BST becomes a linked list (all elements inserted in order), search becomes O(n)

When to Use Each Structure:

- **Use Linked List when:** Frequent insertions/deletions at known positions, simple sequential access
- **Use BST when:** Frequent searching, need sorted data, range queries
- **Consider alternatives:** Hash tables for O(1) search, balanced trees (AVL, Red-Black) for guaranteed O(log n)

6.2 Space Complexity Analysis

- **Doubly Linked List:** O(n) for n elements
- **Binary Search Tree:** O(n) for n nodes
- **Traversal Algorithms:** O(h) for recursive, O(w) for BFS

7 Part 5: Testing and Quality Assurance (10 points)

7.1 Unit Testing Requirements

- Test all public methods with various inputs
- Test edge cases (empty structures, single elements)
- Test error conditions and exception handling

- Achieve minimum 90% code coverage

Testing Strategy:

- **Basic Operations:** Test insert, delete, search with normal data
- **Edge Cases:** Empty list/tree, single element, duplicate values
- **Boundary Testing:** First/last elements, out-of-bounds access
- **Error Conditions:** Invalid inputs, null values, negative indices

Sample Test Cases for Doubly Linked List:

- Insert at beginning, middle, end
- Delete from empty list, single element, multiple elements
- Search for existing and non-existing elements
- Test iterator functionality

Sample Test Cases for Binary Search Tree:

- Insert elements in various orders
- Delete leaf nodes, nodes with one child, nodes with two children
- Test all traversal methods
- Verify BST property is maintained after operations

7.2 Performance Testing

- Benchmark insertion/deletion operations
- Measure traversal performance
- Compare recursive vs iterative implementations

Performance Metrics to Measure:

- **Time Complexity:** Measure actual execution time for different input sizes
- **Memory Usage:** Track memory consumption during operations
- **Scalability:** Test performance with increasing data sizes (100, 1000, 10000 elements)
- **Comparison:** Compare your implementation with Java's built-in collections

Benchmarking Tools:

- **JUnit 5:** For unit testing and basic performance measurement
- **JMH (Java Microbenchmark Harness):** For precise performance testing
- **VisualVM:** For memory profiling and performance analysis
- **Simple Timing:** Use `System.nanoTime()` for basic measurements

Performance Test Example:

```java
@Test
public void testInsertionPerformance() {
    long startTime = System.nanoTime();
    for (int i = 0; i < 10000; i++) {
        list.add(i);
    }
    long endTime = System.nanoTime();
    long duration = endTime - startTime;
    System.out.println("Insertion time: " + duration + " ns");
}
```

8 Deliverables

8.1 Source Code

- Complete implementation of DoublyLinkedList interface
- Complete implementation of BinarySearchTree interface
- Detailed unit tests
- Performance benchmarks

8.2 Documentation

- API documentation with Javadoc
- Complexity analysis report
- Implementation notes and design decisions

8.3 Video Reflection

- Explain design choices and trade-offs
- Discuss complexity analysis results
- Demonstrate working implementations
- Reflect on learning outcomes

9 Sample Questions for Lab Test Preparation

9.1 Data Structure Questions

1. **What are the advantages and disadvantages of doubly linked lists compared to singly linked lists?**

   Answer: Doubly linked lists allow bidirectional traversal and easier deletion of arbitrary nodes, but require more memory (extra pointer per node) and more complex pointer manipulation.

2. **Explain the time complexity of different BST operations and when BST degenerates to O(n) complexity.**

   Answer: BST operations are O(log n) when balanced, but O(n) when the tree becomes a linear chain (worst case). This happens when elements are inserted in sorted order.

3. **Compare the space complexity of recursive vs iterative tree traversal implementations.**

   Answer: Recursive implementations use O(h) stack space, while iterative implementations use O(w) queue space for BFS or O(h) stack space for DFS.

9.2 Algorithm Questions

4. **When would you use pre-order vs post-order traversal?**

   Answer: Pre-order is used for tree structure reconstruction, while post-order is used for tree deletion and expression evaluation where children must be processed before parents.

5. **How does BFS differ from DFS in terms of memory usage and use cases?**

   Answer: BFS uses more memory (queue size = tree width) but finds shortest paths, while DFS uses less memory (stack size = tree height) but may not find optimal solutions.

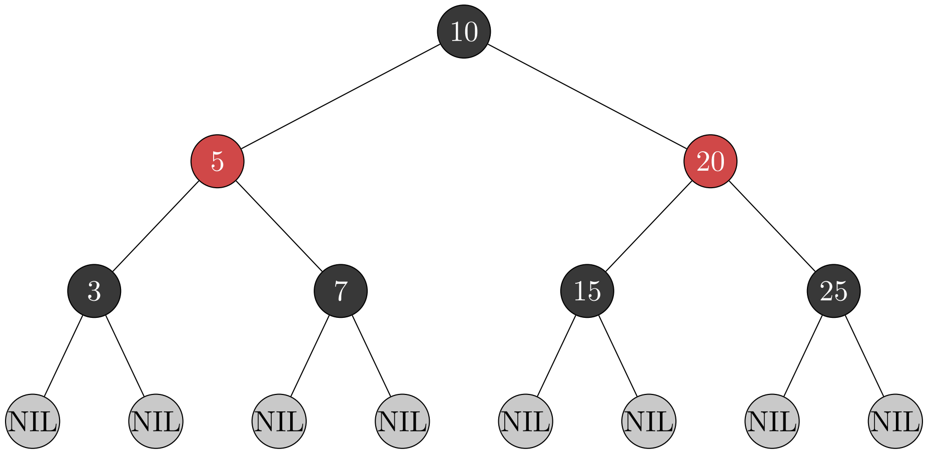
10 Food for Thought: Red–Black Tree

Balanced trees are a type of binary search tree that ensures that the height of the tree is logarithmic in the number of nodes. This is important because it ensures that the time complexity of the tree is $O(\log n)$ for search, insert, and delete operations. As we discussed breifly in our BST section, a BST degenerates to a linked list if the elements are inserted in sorted order and the tree becomes unbalanced.

A **Red–Black Tree** is a type of self-balancing binary search tree. It ensures that insertion, deletion, and lookup operations run in $O(\log n)$ time by maintaining a set of structural invariants. Each node is either red or black, and every path from the root to a NIL (sentinel leaf) has the same number of black nodes. This property guarantees that no path is more than twice as long as any other, keeping the tree balanced.

In the example below, the root node 10 is black. Its two children (5 and 20) are red, and each of those red nodes only has black children. Notice that all leaf NIL nodes are explicitly drawn — these are sentinel nodes used to simplify insert and delete logic. If we follow any path from the root to a NIL, we see the same number of black nodes, which confirms the black-height property. This balance condition is what keeps search efficient.

Red–Black Trees are widely used in practice because they combine good theoretical guarantees with simple implementation details. For instance, the C++ `std::map` and `std::set` containers are implemented using Red–Black Trees. They ensure predictable logarithmic time for critical operations even in the worst case, unlike a naïve binary search tree which can degrade to $O(n)$ if the data is inserted in sorted order.


10
5
20
3
7
15
25
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL