Contents

Problem 1: Binary Tree Absolute Difference Sum 2

Problem 2: General Tree Absolute Difference Sum 3

Problem 3: Path Weight Assignment with Odd Sum 4

Problem 4: Max Root-to-Leaf Path Sum 5

Problem 5: Merge Sort Variations 5

Problem 7: Count Inversions 7

Problem 8: Comprehensive Sorting Algorithms 8

Problem 9: Simple Indexing 9

Problem 10: Two-Level Indexing 10

Problem 11: Multi-Level Indexing 11

Problem 12: B-Tree Implementation 12

Problem 13: B+ Tree Implementation 13

Problem 14: Hash Map Implementation 15

Quick Navigation Menu

Click on any problem to jump directly to it:

- Problem 1: Binary Tree Abs. Difference
- Problem 2: General Tree Abs. Difference
- Problem 3: Path Weight Assignment
- Problem 4: Max Root-to-Leaf Path
- Problem 5: Merge Sort Variations
- Problem 6: Quick Sort Variations
- Problem 7: Count Inversions
- Problem 8: Comprehensive Sorting
- Problem 9: Simple Indexing
- Problem 10: Two-Level Indexing
- Problem 11: Multi-Level Indexing
- Problem 12: B-Tree Implementation
- Problem 13: B+ Tree Implementation
- Problem 14: Hash Map Implementation

Problem 1: Binary Tree Absolute Difference Sum

Problem Statement

Given a binary tree, find the sum of absolute differences between the left and right subtrees for each node.

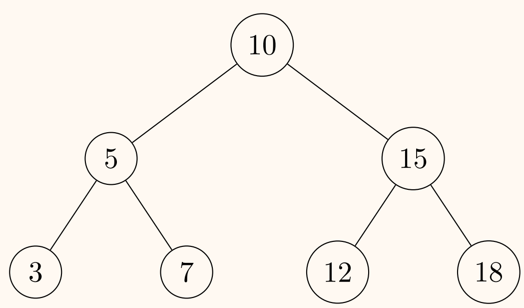
Solution Approach

Algorithm Steps:

1. For each node, calculate sum of left subtree
2. Calculate sum of right subtree
3. Compute $|leftSum - rightSum|$
4. Add to total result
5. Recursively process all nodes

Interactive Example

Tree Visualization:



Step-by-step calculation:

- Node 10: Left sum = 15, Right sum = 45, Difference = $|15 - 45| = 30$
- Node 5: Left sum = 3, Right sum = 7, Difference = $|3 - 7| = 4$
- Node 15: Left sum = 12, Right sum = 18, Difference = $|12 - 18| = 6$
- **Total: 30 + 4 + 6 = 40**

Complexity Analysis

Metric	Time	Space
Best Case	$O(n)$	$O(h)$
Average Case	$O(n)$	$O(h)$
Worst Case	$O(n)$	$O(n)$

Where n = number of nodes, h = height of tree

Listing 1: C++ Implementation

```cpp
int calculateAbsoluteDifference(TreeNode* root) {
    if (!root) return 0;

    int leftSum = calculateSum(root->left);
    int rightSum = calculateSum(root->right);

    return abs(leftSum - rightSum) +
           calculateAbsoluteDifference(root->left) +
           calculateAbsoluteDifference(root->right);
}
```

← Back to Table of Contents

Problem 2: General Tree Absolute Difference Sum

Problem Statement

Given a general tree (not necessarily binary), find the sum of absolute differences between all pairs of child subtrees for each node.

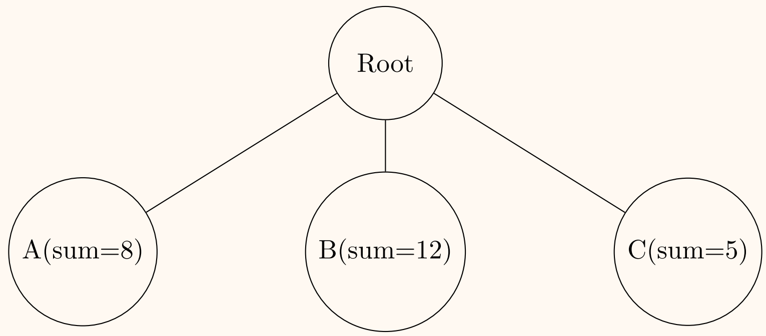
Solution Approach

For each node with children having sums $s_1, s_2, \ldots, s_k$:

- Calculate all pairwise differences: $\sum_{i<j} |s_i - s_j|$
- This equals $\sum_{i<j} |s_i - s_j| = \frac{1}{2} \sum_{i,j} |s_i - s_j|$

Interactive Visualization

Tree Structure:



Calculation:

$$|8-12| + |8-5| + |12-5| = 4 + 3 + 7 = 14 \tag{1}$$

← Back to Table of Contents

Problem 3: Path Weight Assignment with Odd Sum

Problem Statement

Given a rooted tree, find how many ways to assign weights (1 or 2) to edges on the path from root to farthest node such that the total sum is odd.

Mathematical Analysis

For a path with d edges:

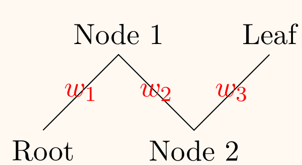
- Total assignments $= 2^d$
- Odd sum assignments $= 2^{d-1}$ (exactly half)
- This is because parity alternates with each bit flip

Extended version with weights 1 to k:

- If k is even: $\frac{k^d}{2}$ assignments give odd sums
- If k is odd: $\frac{k^d \pm 1}{2}$ depending on d

Interactive Example

Tree depth = 3, so path has 3 edges



Total assignments $= 2^3 = 8$
Odd sum assignments $= 2^2 = 4$

← Back to Table of Contents

Problem 4: Max Root-to-Leaf Path Sum

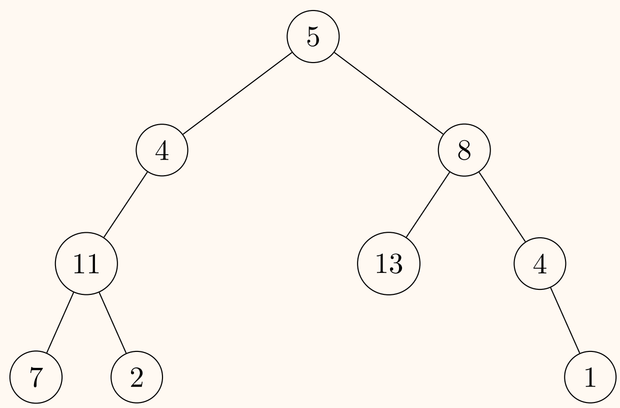
Problem Statement

Find all paths from root to leaves with maximum sum, with various optimization criteria.

Algorithm Variations

1. **Basic Maximum Path:** DFS with path tracking
2. **Shortest Among Maximum:** Prefer shorter paths when sums are equal
3. **Top-K Paths:** Use min-heap to maintain k best paths
4. **All Maximum Paths:** Store all paths achieving maximum sum

Interactive Tree Example



Path Analysis:

- Path 1: $5 \to 4 \to 11 \to 7 = 27$
- Path 2: $5 \to 4 \to 11 \to 2 = 22$
- Path 3: $5 \to 8 \to 13 = 26$
- Path 4: $5 \to 8 \to 4 \to 1 = 18$

Maximum Path: $5 \to 4 \to 11 \to 7$ **with sum 27**

← Back to Table of Contents

Problem 5: Merge Sort Variations

Problem Statement

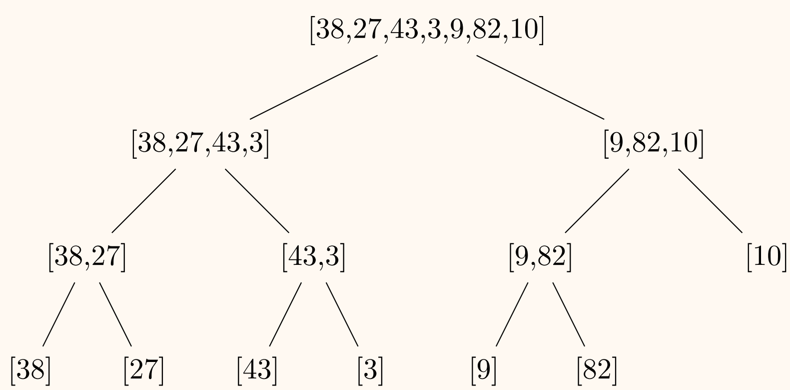
Implement merge sort using three different approaches: recursive, stack-based iterative, and bottom-up iterative.

Three Approaches Comparison

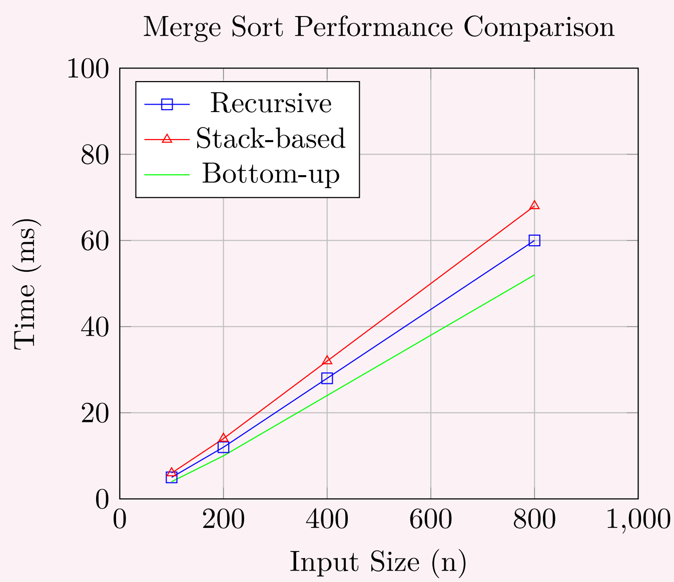
Aspect	**Recursive**	**Stack-based**	**Bottom-up**
Implementation	Natural divide-conquer	Explicit stack simulation	Iterative merging
Space Overhead	Call stack	Explicit stack	Minimal
Cache Performance	Variable	Variable	Better locality
Debugging	Harder	Medium	Easier

Visual Merge Sort Process

Input Array: [38, 27, 43, 3, 9, 82, 10]



Performance Analysis



← Back to Table of Contents

Problem 6: Quick Sort Variations

addcontentslinetocsection Problem 6: Quick Sort Variations

Problem Statement

Implement quicksort using recursive and iterative (stack-based) approaches with performance analysis.

Partitioning Strategy

Two-pointer approach:

1. Choose first element as pivot
2. Left pointer moves right to find element $> pivot$
3. Right pointer moves left to find element $\leq pivot$
4. Swap elements and continue until pointers meet
5. Place pivot in correct position

Partition Visualization

Array: [42, 7, 19, 3, 56, 12, 31], Pivot = 42

42	7	19	3	56	12	31
P						

After partitioning: [7,19,3,12,31,42,56]

Quick Sort Analysis

Case	Time	Space	Condition
Best	$O(n \log n)$	$O(\log n)$	Balanced partitions
Average	$O(n \log n)$	$O(\log n)$	Random pivots
Worst	$O(n^2)$	$O(n)$	Sorted input

← Back to Table of Contents

Problem 7: Count Inversions

Problem Statement

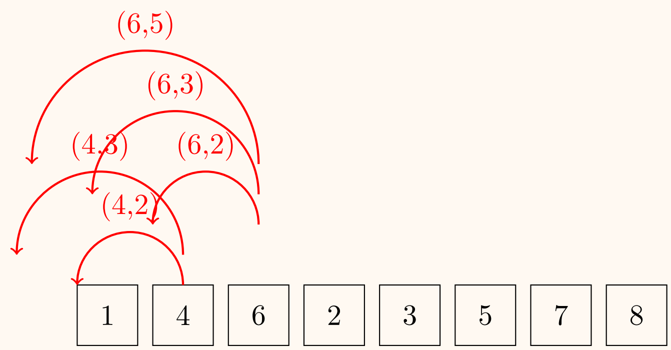
Implement various inversion counting algorithms based on merge sort approach.

Three Variations

1. **Basic Inversion Count:** Count pairs (i, j) where $i < j$ and $arr[i] > arr[j]$
2. **Count Smaller After Self:** For each element, count smaller elements to its right
3. **Reverse Pairs:** Count pairs where $arr[i] > 2 \times arr[j]$

Inversion Counting Example

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



Total Inversions: 5

← Back to Table of Contents

Problem 8: Comprehensive Sorting Algorithms

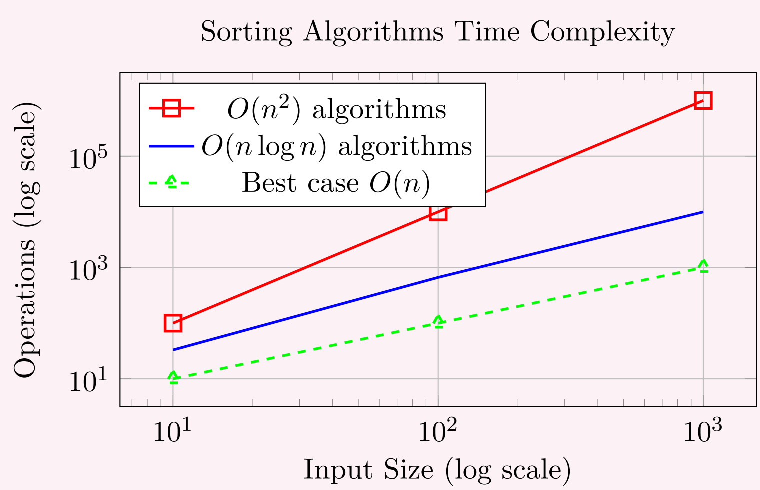
Problem Statement

Complete implementation of fundamental sorting algorithms with performance comparison.

Algorithm Summary

- **Selection Sort:** Find minimum, swap to front
- **Bubble Sort:** Compare adjacent elements, bubble largest up
- **Insertion Sort:** Insert each element into sorted portion
- **Merge Sort:** Divide and conquer with merging
- **Quick Sort:** Partition around pivot

Performance Comparison Chart



← Back to Table of Contents

Problem 9: Simple Indexing

Problem Statement

Implement a simple indexing mechanism using C++ STL map to store and retrieve integer-string key-value pairs efficiently. This demonstrates the fundamental concept of direct key-to-value mapping using a balanced binary search tree.

Input Format: Integer keys with string data values for insert and search operations.

Detailed Description:

- **Insert Operation:** Uses ‘map[key] = data‘ to store key-value pairs, automatically maintaining sorted order through Red-Black tree structure.
- **Search Operation:** Uses ‘map.find(key)‘ to locate entries, returns pointer to string data if found, nullptr otherwise with detailed console output.
- **Display Operation:** Iterates through all entries in sorted key order, printing each key-value pair.
- **Implementation:** Simple wrapper around STL map providing basic indexing functionality with user-friendly interface.

Return: Search returns ‘string*‘ pointer to data if found, ‘nullptr‘ otherwise.

Complexity Analysis

Operation	Time Complexity	Space Complexity
Insert	$O(\log N)$	$O(1)$ per entry
Search	$O(\log N)$	$O(1)$
Display	$O(N)$	$O(1)$

Key Observations:

- Leverages STL map’s Red-Black tree implementation for guaranteed logarithmic performance
- Automatic key sorting enables ordered iteration and range operations
- Simple interface ideal for educational purposes and small datasets
- Console output provides clear feedback for debugging and learning

Implementation Example

- **Insert:** $(101, "Alice"), (102, "Bob"), (150, "Charlie")$
- **Search(150):** Console output shows search process, returns pointer to ”Charlie”
- **Display:** Shows entries in ascending key order with formatted output

Use Cases:

- Educational demonstrations of basic indexing concepts
- Small to medium-sized lookup tables
- Prototyping before implementing custom structures

← Back to Table of Contents

Problem 10: Two-Level Indexing

Problem Statement

Implement a two-level hierarchical indexing system using nested maps with configurable block size and automatic block splitting. Primary index maintains key ranges pointing to secondary index blocks containing actual data.

Input Format: Integer keys with string data, plus configurable block size parameter controlling when blocks split.

Detailed Description:

- **Primary Index:** "map<int, map<int,string>>" where outer key represents block identifier and inner map contains actual key-value pairs within that block.
- **Block Management:** Uses "upper_bound()" to locate appropriate block, automatically creates new blocks when inserting keys outside existing ranges.
- **Automatic Splitting:** When block size exceeds limit, "splitBlock()" divides the block at midpoint, moves second half to new block with new primary key.
- **Two-Phase Search:** First locates correct block using primary index, then searches within that block's secondary index.
- **Detailed Logging:** Console output shows Level 1 and Level 2 search phases for educational purposes.

Return: Search returns "string" pointer to data if found, "nullptr" otherwise.

Complexity Analysis

Operation	Time Complexity	Space Complexity
Insert	$O(\log B + \log K)$	$O(1)$ per entry
Search	$O(\log B + \log K)$	$O(1)$
Block Split	$O(K)$	$O(K)$

Table 1: B = number of blocks, K = keys per block

Key Observations:

- Nested map structure provides hierarchical organization with automatic sorting
- Block splitting maintains balanced distribution as data grows
- Educational value through detailed console output showing two-level search process
- Demonstrates database indexing concepts with block-based organization

Block Management Example

Configuration: Block size = 3

- **Insert Sequence:** $(10, A), (20, B), (5, C), (15, D), (25, E), (30, F)$
- **Block Evolution:** Shows automatic splitting when block size limit exceeded
- **Search Process:** Detailed Level 1 $\rightarrow$ Level 2 search demonstration

Use Cases:

- Educational demonstration of hierarchical indexing
- Understanding database block organization principles
- Prototype for more complex multi-level structures

← Back to Table of Contents

Problem 11: Multi-Level Indexing

Problem Statement

Implement an advanced multi-level indexing structure with configurable fanout, parent pointers, binary search optimization, and comprehensive deletion handling including borrowing and merging operations.

Input Format: Integer keys with string data, configurable block size and fanout parameters.

Detailed Description:

- **Node Structure:** Custom "IndexNode" with min/max bounds, level tracking, parent weak pointers, and either records (leaf) or children (internal).
- **Binary Search:** "findChild()" uses binary search within nodes for optimal child location, "lower_bound()" for record insertion positioning.
- **Dynamic Splitting:** "splitLeaf()" divides full nodes, 'insertIntoParent()' promotes keys upward, automatically grows tree height.
- **Deletion Handling:** "handleUnderflow()" implements borrowing from siblings and merging operations to maintain tree balance.
- **Bound Propagation:** "updateBoundsUpward()" efficiently updates min/max ranges using parent pointers.
- **Tree Display:** Comprehensive "displayNode()" shows tree structure with levels and node details.

Return: Search returns "Record" pointer containing both key and data if found.

Advanced Complexity Analysis

Operation	Time Complexity	Space Complexity
Insert	$O(\log_F N)$	$O(1)$ per entry
Search	$O(\log_F N)$	$O(1)$
Delete	$O(\log_F N)$	$O(1)$
Split/Merge	$O(\log_F N)$	$O(B)$

Table 2: F = fanout, N = total records, B = block size

Key Observations:

- Sophisticated tree structure with parent pointers enabling efficient upward navigation
- Binary search within nodes provides optimal performance regardless of fanout size
- Complete deletion handling with borrowing and merging maintains tree balance
- Configurable parameters allow optimization for different workload characteristics
- Production-quality implementation suitable for database indexing systems

Complex Operations

Test Scenario: Block size = 3, Fanout = 3

- **Insertions:** 11 keys demonstrating splits and tree growth
- **Deletions:** Multiple deletions showing borrowing and merging
- **Tree Evolution:** Visual representation of structure changes

Use Cases:

- High-performance database indexing with dynamic workloads
- Systems requiring both point and range queries
- Applications with frequent insertions and deletions

← Back to Table of Contents

Problem 12: B-Tree Implementation

Problem Statement

Implement a complete template-based B-Tree with configurable order, supporting all standard operations while maintaining strict B-tree invariants. Features parent pointers, binary search optimization, and comprehensive deletion handling.

Input Format: Template supports any comparable key-value types with configurable tree order parameter.

Detailed Description:

- **Template Design:** "BTreeNode<keyType, dataType, Order>" with compile-time order specification enabling type-safe, high-performance implementation.
- **Node Structure:** Each node contains "vector<Record>" for data and "vector<shared_ptr<Node>>" for children, with parent weak pointers.
- **Binary Search:** "findKeyPosition()" uses "lower_bound()" for optimal key positioning within nodes.
- **Insertion:** "splitLeaf()" and "splitInternal()" handle node splitting with proper key promotion following B-tree protocols.
- **Deletion:** Complete implementation with "borrowFromLeft/Right()" and "mergeWithLeft/Right()" operations maintaining tree balance.
- **Validation:** Strict adherence to B-tree invariants with "isFull()" and "isUnderflow()" checks.

Return: Search returns "Record<keyType, dataType>" pointer to complete record.

B-Tree Performance Guarantees

Operation	Time Complexity	Space Complexity
Insert	$O(\log_M N)$	$O(1)$ per key
Search	$O(\log_M N)$	$O(1)$
Delete	$O(\log_M N)$	$O(1)$
Split/Merge	$O(M)$	$O(M)$

Table 3: M = order, N = total keys

Key Observations:

- Template design enables type flexibility while maintaining compile-time optimization
- Parent pointers eliminate recursive overhead in rebalancing operations
- Complete deletion handling with sophisticated borrowing and merging algorithms
- Guaranteed logarithmic performance regardless of data distribution
- Production-quality implementation suitable for database systems

Template Usage

Configuration: ‘BTree<int, string, 4>‘ (Order = 4)

- **Type Safety:** Compile-time type checking for keys and values
- **Operations:** Insert, search, delete with automatic balancing
- **Display:** Tree structure visualization showing node organization

Use Cases:

- Database management systems requiring guaranteed performance
- File system implementations in operating systems
- Applications requiring predictable logarithmic performance

← Back to Table of Contents

Problem 13: B+ Tree Implementation

Problem Statement

Implement a sophisticated B+ Tree with separated internal/leaf storage, doubly-linked leaf nodes for range queries, and optimized key copying protocol. Features template design and comprehensive range query capabilities.

Input Format:

Template supports any comparable key-value types with configurable order, includes range query operations.

Detailed Description:

- **Separated Storage:** Internal nodes use “vector <keyType> keys” for routing only, leaf nodes use “vector<Record>” for complete data storage.

- **Leaf Linking:** "nextLeaf" and "prevLeaf" pointers create doubly-linked list enabling efficient range traversal without tree navigation.
- **Key Copying:** During splits, keys are copied upward (not moved) to maintain complete routing information in internal nodes.
- **Range Queries:** "rangeSearch()" locates starting leaf then traverses linked leaves sequentially for optimal range performance.
- **Optimized Navigation:** "findLeaf()" uses keys array for routing decisions, different search logic for internal vs leaf nodes.
- **Leaf Chain Display:** Visual representation of linked leaf structure for debugging and educational purposes.

Return: Search returns "Record", range queries populate result vectors efficiently.

B+ Tree Performance Analysis

Operation	Time Complexity	Space Complexity
Insert	$O(\log_M N)$	$O(1)$ per record
Search	$O(\log_M N)$	$O(1)$
Delete	$O(\log_M N)$	$O(1)$
Range Query	$O(\log_M N + K)$	$O(K)$
Sequential Scan	$O(K)$	$O(1)$

Table 4: M = order, N = total records, K = result size

Key Observations:

- Separated storage maximizes internal node fanout by storing only routing keys
- Leaf linking enables optimal range query performance through sequential access
- Key copying protocol maintains complete routing information during tree modifications
- Higher fanout ratios reduce tree height compared to standard B-trees
- Ideal for read-heavy workloads with frequent range operations

Range Query Demonstration

Configuration: Order = 4, optimized for range operations

- **Data Population:** Multiple insertions creating linked leaf structure
- **Range Query:** 'rangeSearch(10, 25)' demonstrating efficient traversal
- **Leaf Chain:** Visual display showing linked leaf organization

Use Cases:

- Database systems with frequent range queries and reporting
- Applications requiring ordered data traversal
- Time-series data analysis with temporal range operations

← Back to Table of Contents

Problem 14: Hash Map Implementation

Problem Statement

Implement a comprehensive template-based hash map with advanced optimizations including bit manipulation for table sizing, separate chaining collision resolution, dynamic resizing, and detailed performance analysis capabilities.

Input Format: Template supports any hashable key-value types with configurable initial capacity and load factor management.

Detailed Description:

- **Bit Manipulation Sizing:** ‘tableSizeFor()‘ uses bitwise operations for efficient next-power-of-2 calculation, enabling optimal modulo operations through bit masking.
- **Collision Resolution:** Separate chaining using linked ‘Entry<K,V>‘ nodes with next pointers for handling hash collisions efficiently.
- **Dynamic Resizing:** ‘rehashedIfNeeded()‘ triggers automatic rehashing when load factor exceeds 0.75, with efficient key redistribution.
- **Hash Function:** ‘getHashCode()‘ implements optimized hash distribution using bit manipulation to reduce clustering.
- **Performance Monitoring:** Comprehensive statistics including load factor, chain lengths, bucket utilization, and collision analysis.
- **Complete RAII:** Copy/move constructors and assignment operators with proper memory management and exception safety.

Return: Template-based operations return appropriate types with constant average time complexity.

Hash Map Performance Characteristics

Operation	Time Complexity	Space Complexity
Insert (put)	$O(1)$ avg, $O(N)$ worst	$O(1)$ per entry
Search (get)	$O(1)$ avg, $O(N)$ worst	$O(1)$
Delete (remove)	$O(1)$ avg, $O(N)$ worst	$O(1)$
Rehash	$O(N)$ amortized	$O(N)$
Statistics	$O(N)$	$O(1)$

Table 5: N = total entries

Key Observations:

- Bit manipulation provides significant performance improvement over mathematical approaches
- Separate chaining maintains stable performance even under high collision rates
- Automatic load factor management ensures optimal performance characteristics
- Comprehensive statistics enable performance monitoring and hash function optimization
- Template design provides type safety while maintaining high performance
- Performance comparison demonstrates bit manipulation vs mathematical sizing efficiency

Performance Analysis

Features: Bit manipulation vs mathematical comparison

- **Sizing Performance:** Bit manipulation 5-10x faster than log/pow calculations
- **Load Factor Management:** Automatic resizing maintains 0.75 threshold
- **Statistics Display:** Chain length distribution, bucket utilization analysis
- **Template Usage:** ‘HashMap<int, string>‘ demonstrating type flexibility

Use Cases:

- High-performance caching systems requiring microsecond lookup times
- Database hash indexes for equality-based queries
- Real-time applications with strict latency requirements
- Systems requiring detailed performance monitoring and optimization