① benchmark : Measuring performance through experiments

```c
/** @file bst_vs_array_benchmark.c
 *  @brief Benchmark: O(n) linear scan vs O(log n) BST search with real timings.
 *
 *  @section Assumptions
 *  - Keys are ints in [0..N); BST is built from a shuffled sequence to avoid skew.
 *
 *  @section Complexity
 *  - Linear scan: O(n) per query (poor cache locality).
 *  - BST search:  expected O(log n) (random inserts ≈ balanced height).
 *  - BST build:   expected O(n log n).
 *
 *  @section Gotchas
 *  - Recursive insert can overflow the stack on skewed input.
 *  - Dead-code elimination can nuke timing loops → use a volatile sink.
 *  - clock() has coarse resolution → use clock_gettime(CLOCK_MONOTONIC).
 *
 *  @section ExperimentLog
 *  - 2025-10-02, M2 Pro, -O3
 *    N=1e6,  Q=2e4 → Array 2983.5 ms, BST(iter) 5.133 ms (~581x)
 *    N=5e6,  Q=2e4 → Array 15005.9 ms, BST(iter) 13.481 ms (~1113x)
 *    (Earlier recursive cap=200k mismatch resolved in current build.)
 *
 *  @section TODO
 *  - Add qsort+bsearch baseline (static array, O(log n)).
 *  - Control hit/miss ratio; sweep N and plot.
 */

#define _POSIX_C_SOURCE 199309L

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

· Volatile : A type qualifier instructing the compiler to not optimize a specific variable.
↓
Declares variable can change unpredictably from outside the program.

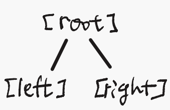
```c
// ========================================
// Global sink to defeat dead-code elimination
// ========================================
static volatile long long g_sink = 0; // volatile access = observable side-effect ⇒ No optimization
//                  64-bit integer   ↳ global data sink

// ========================================
// High-resolution timer (ms) 1/1000 sec
// ========================================
static double now_ms(void) {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (double)ts.tv_sec * 1000.0 + (double)ts.tv_nsec / 1e6;
}
```

Monotonic : 항상 한쪽 방향으로만 변화함.
→ CLOCK_MONOTONIC = 시계가 부팅시점 부터 계속 증가만 한다. cf. NTP (Network Time Protocol)

```c
// ========================================
// Safe allocation helpers
// ========================================
static void* xmalloc(size_t n) {
    void* p = malloc(n);
    if (!p) {
        fprintf(stderr, "malloc failed (requested %zu bytes)\n", n);
        exit(EXIT_FAILURE);
    }
    return p;
}
```

② Tree : a hierarchical data structure with nodes connected by edges.



BST (Binary Search Tree)
· Each node has at most 2 children (left & right)
· left child < parent < right child.

```c
// ========================================
// BST Node
// ========================================
typedef struct Node {
    int data;  // value stored in node
    struct Node* left;  // pointer to left child
    struct Node* right; // pointer to right child
} Node;

static Node* create_node(int value) {
    Node* p = (Node*)xmalloc(sizeof(Node)); // 1. allocate memory
    p->data = value;                        // 2. set value
    p->left = p->right = NULL;              // 3. No children yet
    return p;                               // 4. Return new node
}

// ========================================
// BST Operations — Recursive
// ========================================
static Node* insert_recursive(Node* root, int value) {
```

```c
    if (root == NULL) return create_node(value);  // Base case
    if (value < root->data) {
        root->left = insert_recursive(root->left, value); // Go left
    } else if (value > root->data) {
        root->right = insert_recursive(root->right, value); // Go right
    } // duplicates ignored
    return root;
}

static Node* search_recursive(Node* root, int value) {
    if (root == NULL || root->data == value) return root;  // Found or not found
    if (value < root->data) return search_recursive(root->left, value); // Search left
    return search_recursive(root->right, value); // Search right
}

/** @brief Insert a key into the BST iteratively.
 *  @param root Current root pointer (may be NULL).
 *  @param value Key to insert.
 *  @return Updated root pointer.
 *  @note Duplicates are ignored (set semantics).
 *  @warning Worst-case height O(n) if inserts are skewed (no shuffle).
 *  @remark Expected height ~O(log n) with randomized inserts.
 */
static Node* insert_iterative(Node* root, int value) {
    if (root == NULL) return create_node(value); // empty tree

    Node* cur = root;     // current position
    Node* parent = NULL;  // track parent for insertion
    // Step 1. Find the insertion point
    while (cur != NULL) {
        parent = cur;      // Remember current as parent
        if (value < cur->data) {
            cur = cur->left; // Go left
        } else if (value > cur->data) {
            cur = cur->right; // Go right
        } else {
            // duplicate: do nothing
            return root;
        }
    }
    // Step 2: Create new node and attach to parent
    Node* n = create_node(value);
    if (value < parent->data) parent->left = n;  // Attach as left child
    else parent->right = n;                      // Attach as right child

    return root;
}

/** @brief Search a BST iteratively.
 *  @param root Root pointer (may be NULL).
 *  @param value Key to find.
 *  @return Node pointer if found, otherwise NULL.
 */
static Node* search_iterative(Node* root, int value) {
    Node* cur = root;
    while (cur && cur->data != value) {  // loop until found or NULL
        cur = (value < cur->data) ? cur->left : cur->right; // Ternary operator
    }
    return cur; // Returns node if found, NULL if not found.
}

// ========================================
// Utilities: linear search, shuffle, height, free
// ========================================
static int linear_search(const int* arr, int size, int value) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == value) return i;   // Found: return index
    }
    return -1; // Not found
}

/** @brief Shuffle array in-place using Fisher–Yates.
 *  @param arr Array of ints.
 *  @param size Number of elements.
 */
static void shuffle(int* arr, int size) {
    for (int i = size - 1; i > 0; i--) {
        int j = rand() % (i + 1);                          // Random index: 0 to i
        int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;   // Swap arr[i] and arr[j]
    }
}
```

Use recursive when:
- Code clarity is priority
- Tree depth is small
- Learning / Understanding BST

Use iterative when:
- performance matters
- Tree might be very deep (avoid stack overflow)
- Production code

same as:
```
if (value < cur->data)
    cur = cur->left;
else
    cur = cur->right;
```

Array: [5, 2, 9, 1, 9]
Search for 9:
i=0: arr[0]=5 5==9? NO
i=1: arr[1]=2 5==2? NO
i=2: arr[2]=9 9==9? Yes! ⇒ Return 2

Time complexity: O(n)

```c
static int get_tree_height(const Node* root) {
    if (!root) return 0;                    // Base case: empty = height 0
    int lh = get_tree_height(root->left);   // Left subtree height
    int rh = get_tree_height(root->right);  // Right subtree height
    return 1 + (lh > rh ? lh : rh);         // 1 + max(left, right)
}

static void free_tree(Node* node) {
    if (!node) return;
    free_tree(node->left);
    free_tree(node->right);
    free(node);
}

// ========================================
// CLI args
// ========================================
typedef struct {
    int size;         // number of elements
    int queries;      // number of search queries
    int seed;         // RNG seed (0 => time-based)
    int demo_small_N; // small N for skew vs balanced demo
} BenchArgs;

static BenchArgs parse_args(int argc, char** argv) {
    BenchArgs a = { .size = 1000000, .queries = 20000, .seed = 0, .demo_small_N = 100 };
    for (int i = 1; i < argc; i++) {
        if (!strcmp(argv[i], "--size") && i + 1 < argc) {
            a.size = atoi(argv[++i]);
        } else if (!strcmp(argv[i], "--queries") && i + 1 < argc) {
            a.queries = atoi(argv[++i]);
        } else if (!strcmp(argv[i], "--seed") && i + 1 < argc) {
            a.seed = atoi(argv[++i]);
        } else if (!strcmp(argv[i], "--demoN") && i + 1 < argc) {
            a.demo_small_N = atoi(argv[++i]);
        } else if (!strcmp(argv[i], "--help") || !strcmp(argv[i], "-h")) {
            printf("Usage: %s [--size N] [--queries Q] [--seed S] [--demoN K]\n", argv[0]);
            exit(0);
        }
    }
    return a;
}

// ========================================
// Main
// ========================================
int main(int argc, char** argv) {
    BenchArgs cfg = parse_args(argc, argv);

    if (cfg.seed == 0) cfg.seed = (int)time(NULL);
    srand((unsigned)cfg.seed);

    printf("BST vs Array Performance Benchmark\n");
    printf("  size    : %d\n", cfg.size);
    printf("  queries : %d\n", cfg.queries);
    printf("  seed    : %d\n\n", cfg.seed);

    // ------------------------------------------------------------------------
    // Experiment 1: Skewed vs Balanced (small N demo)
    // ------------------------------------------------------------------------
    printf("=== Experiment 1: Skewed vs Balanced (N=%d) ===\n", cfg.demo_small_N);

    int smallN = cfg.demo_small_N;
    int* seq = (int*)xmalloc(sizeof(int) * smallN);
    int* shf = (int*)xmalloc(sizeof(int) * smallN);
    for (int i = 0; i < smallN; i++) {
        seq[i] = i;
        shf[i] = i;
    }
    shuffle(shf, smallN);

    Node* skew = NULL;
    for (int i = 0; i < smallN; i++) {
        skew = insert_iterative(skew, seq[i]); // sequential -> skewed
    }
    Node* bal = NULL;
    for (int i = 0; i < smallN; i++) {
        bal = insert_iterative(bal, shf[i]); // shuffled -> balanced-ish
    }

    int h_skew = get_tree_height(skew);
    int h_bal  = get_tree_height(bal);
```

- Height = longest path from root to leaf
  - Empty tree: height 0
  - Single node: height 1

1. Parse command-line arguments
2. Experiment 1: skewed vs balanced tree demo — skewed: sequential insert, worst case O(n) vs balanced: shuffled insert, best case O(log n)
3. Experiment 2: Large-scale performance test
   - Array O(n) linear search
   - BST O(log n) iterative search
   - BST recursive search
4. Cleanup and exit

```
    printf("Skewed (sequential insert)  height: %d  (worst-case ~N)\n", h_skew);
    printf("Balanced (shuffled insert)  height: %d  (~log N expected)\n\n", h_bal);

    free_tree(skew);
    free_tree(bal);
    free(seq);
    free(shf);

    // ------------------------------------------------------------------------
    // Experiment 2: Large-scale performance
    // ------------------------------------------------------------------------
    printf("=== Experiment 2: Large-scale Performance ===\n");
    const int N = cfg.size;
    const int Q = cfg.queries;

    // Data for array and BST(iter)
    int* data = (int*)xmalloc(sizeof(int) * N);
    for (int i = 0; i < N; i++) data[i] = i;

    printf("Shuffling %d integers...\n", N);
    shuffle(data, N);

    int* queries = (int*)xmalloc(sizeof(int) * Q);
    for (int i = 0; i < Q; i++) queries[i] = rand() % N;

    // 1) Array O(n) linear search
    {
        // warm-up
        for (int i = 0; i < Q; i++) (void)linear_search(data, N, queries[i]);

        long long hits = 0;
        double t0 = now_ms();
        for (int i = 0; i < Q; i++) {
            int idx = linear_search(data, N, queries[i]);
            hits += (idx >= 0);
        }
        double t1 = now_ms();
        g_sink += hits;
        printf("\n[Array O(n)] Searching %d queries...\n", Q);
        printf("Array hits: %lld\n", hits);
        printf("Array linear search time: %.3f ms\n", (t1 - t0));
    }

    // 2) BST O(log n) iterative build + search
    Node* bst = NULL;
    double t_build0 = now_ms();
    for (int i = 0; i < N; i++) {
        bst = insert_iterative(bst, data[i]);
    }
    double t_build1 = now_ms();
    double bst_build_ms = (t_build1 - t_build0);

    int bst_h = get_tree_height(bst);
    printf("\n[BST O(log n) - Iterative] Building tree with %d nodes...\n", N);
    printf("BST build time          : %.3f ms\n", bst_build_ms);
    printf("BST height              : %d\n", bst_h);

    {
        // warm-up
        for (int i = 0; i < Q; i++) (void)search_iterative(bst, queries[i]);

        long long hits = 0;
        double t0 = now_ms();
        for (int i = 0; i < Q; i++) {
            hits += (search_iterative(bst, queries[i]) != NULL);
        }
        double t1 = now_ms();
        g_sink += hits;
        printf("[BST O(log n) - Iterative] Searching %d queries...\n", Q);
        printf("BST(iter) hits: %lld\n", hits);
        printf("BST(iter) search time   : %.3f ms\n", (t1 - t0));
    }

    // 3) Recursive build + search (FIXED): build from 0..REC_N-1
    const int REC_N = (N < 200000 ? N : 200000);
    int* rec_data = (int*)xmalloc(sizeof(int) * REC_N);
    for (int i = 0; i < REC_N; i++) rec_data[i] = i;    // 0..REC_N-1
    shuffle(rec_data, REC_N);                           // avoid skew for fair depth

    Node* bst_rec = NULL;
    double t_rec_b0 = now_ms();
    for (int i = 0; i < REC_N; i++) {
```

```c
        bst_rec = insert_recursive(bst_rec, rec_data[i]);
    }
    double t_rec_b1 = now_ms();
    int bst_rec_h = get_tree_height(bst_rec);
    printf("\n[BST (Recursive) — build on 0..%d shuffled] Building...\n", REC_N - 1);
    printf("BST(rec) build time     : %.3f ms\n", (t_rec_b1 - t_rec_b0));
    printf("BST(rec) height         : %d\n", bst_rec_h);

    {
        // Make queries in 0..REC_N-1 so they match the recursive tree's keyset
        int* queries_rec = (int*)xmalloc(sizeof(int) * Q);
        for (int i = 0; i < Q; i++) queries_rec[i] = rand() % REC_N;

        // warm-up
        for (int i = 0; i < Q; i++) (void)search_recursive(bst_rec, queries_rec[i]);

        long long hits = 0;
        double t0 = now_ms();
        for (int i = 0; i < Q; i++) {
            hits += (search_recursive(bst_rec, queries_rec[i]) != NULL);
        }
        double t1 = now_ms();
        g_sink += hits;
        printf("[BST (Recursive)] Searching %d queries (0..%d)...\n", Q, REC_N - 1);
        printf("BST(rec) hits: %lld\n", hits);
        printf("BST(rec) search time    : %.3f ms\n", (t1 - t0));

        free(queries_rec);
    }

    // Cleanup
    free_tree(bst);
    free_tree(bst_rec);
    free(data);
    free(rec_data);
    free(queries);

    // Use g_sink so compiler cannot drop earlier loops
    if (g_sink == 42) printf("sink=%lld\n", g_sink);

    return 0;
}
```

```
➜  03-experiments git:(main) ✗ ./bst_bench --size 100000000 --queries 30000
BST vs Array Performance Benchmark
  size    : 100000000
  queries : 30000
  seed    : 1759539532

═══ Experiment 1: Skewed vs Balanced (N=100) ═══
Skewed (sequential insert)  height: 100  (worst-case ~N)
Balanced (shuffled insert)  height: 12  (~log N expected)

═══ Experiment 2: Large-scale Performance ═══
Shuffling 100000000 integers...

[Array O(n)] Searching 30000 queries...
Array hits: 30000
Array linear search time: 457107.719 ms

[BST O(log n) - Iterative] Building tree with 100000000 nodes...
BST build time          : 128883.499 ms
BST height              : 73
[BST O(log n) - Iterative] Searching 30000 queries...
BST(iter) hits: 30000
BST(iter) search time   : 43.134 ms

[BST (Recursive) — build on 0..199999 shuffled] Building...
BST(rec) build time     : 32.456 ms
BST(rec) height         : 42
[BST (Recursive)] Searching 30000 queries (0..199999)...
BST(rec) hits: 30000
BST(rec) search time    : 3.156 ms
```